22nd International Workshop on Algorithms in Bioinformatics

WABI 2022, September 5–7, 2022, Potsdam, Germany

Edited by

Christina Boucher

Sven Rahmann

Editors

Christina Boucher
Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA
cboucher@cise.ufl.edu

Sven Rahmann
Department of Computer Science and Center for Bioinformatics, Saarland University, Saarbrücken, Germany
https://www.rahmannlab.de/people/rahmann
sven.rahmann@uni-saarland.de

ACM Classification 2012
Applied computing → Bioinformatics; Applied computing → Computational Biology; Theory of computation → Design and analysis of algorithms; Mathematics of computing → Discrete mathematics; Mathematics of computing → Information theory

ISBN 978-3-95977-243-3

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-243-3.

Publication date
September, 2022

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.WABI.2022.0

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

22nd International Workshop on Algorithms in Bioinformatics (WABI 2022).
Editors: Christina Boucher and Sven Rahmann
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This proceedings volume contains papers presented at the 22nd Workshop on Algorithms in Bioinformatics (WABI 2022), which was held in Potsdam, Germany, September 5–7, 2022, and followed by a workshop on Computational Pangenomics, September 7–9 (without published proceedings).

The Workshop on Algorithms in Bioinformatics is an annual conference established in 2001 to cover all aspects of algorithmic work in bioinformatics, computational biology, and systems biology. The conference is intended as a forum for presentation of new insights about discrete algorithms and machine-learning methods that address important problems in biology (particularly problems based on molecular data and phenomena), that are founded on sound models, that are computationally efficient, and that have been implemented and tested in simulations and on real datasets. The focus of the meeting is on recent research results, including significant work-in-progress, as well as identifying and exploring directions of future research.

Over the 22 instances of WABI, computational biology has grown significantly in importance, and now computational analysis methods – some furthered significantly over the years at WABI – have been crucial for the global response to the CoViD-19 pandemics and for the development of vaccines. After two years of workshops strongly affected by the pandemic, the community was happy to meet again in person.

WABI 2022 was organized within the ALGO federation of conferences that in 2022 included WABI, ESA (European Symposium on Algorithms), ALGOCLOUD (International Symposium on Algorithmic Aspects of Cloud Computing), ALGOSENSORS (International Symposium on Algorithms and Experiments for Wireless Sensor Networks), ATMOS (International Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems), IPEC (International Symposium on Parameterized and Exact Computation), and WAOA (Workshop on Approximation and Online Algorithms).

In 2022, a total of 44 manuscripts were submitted to WABI from which 24 were selected for presentation at the conference and are included in this proceedings volume as full papers. While the number of submissions is still lower than in some previous years, it has increased in comparison to the past (pandemic) years, and many rejected submissions were of high quality and solely had to be rejected due to lack of further time slots at the conference. Extended versions of selected papers have been invited for publication in a thematic series in the journal Algorithms for Molecular Biology (AMB), published by BioMed Central. The 24 papers selected for the conference underwent a thorough peer review, involving at least three (and most frequently four) independent reviewers per submitted paper, followed by discussions among the WABI Program Committee members. The selected papers cover a wide range of topics including phylogenetic trees and networks, biological network analysis, sequence alignment and assembly, genomic-level evolution, sequence and genome analysis, RNA and protein structure, topological data analysis, and more. They are ordered randomly within this volume.

We thank all the authors of submitted papers for making this conference possible. A special thanks goes to all the members of the WABI 2022 Program Committee and their subreviewers for their participation in a very active review process with numerous exchanges that culminated in constructive review reports for the authors. We are also grateful to the WABI Steering Committee for their availability, help and advice. We thank all the conference participants, session chairs, and speakers who contributed to a great scientific program.

22nd International Workshop on Algorithms in Bioinformatics (WABI 2022).
Editors: Christina Boucher and Sven Rahmann

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In particular, we are indebted to the keynote speaker of the conference, Leena Salmela (University of Helsinki), for her presentation "Efficient solutions to biological problems using de Bruijn graphs". WABI 2022 is grateful for the support of the University of Potsdam and of the Hasso Plattner Institute, Potsdam. We thank the ALGO 2022 Organizing Committee for setting up the event in these times of uncertainty.

Previous proceedings of WABI appeared in LNCS/LNBI volumes 2149 (WABI 2001, Aarhus), 2452 (WABI 2002, Rome), 2812 (WABI 2003, Budapest), 3240 (WABI 2004, Bergen), 3692 (WABI 2005, Mallorca), 4175 (WABI 2006, Zurich), 4645 (WABI 2007, Philadelphia), 5251 (WABI 2008, Karlsruhe), 5724 (WABI 2009, Philadelphia), 6293 (WABI 2010, Liverpool), 6833 (WABI 2011, Saarbrücken), 7534 (WABI 2012, Ljubljana), 8126 (WABI 2013, Sophia Antipolis), 8701 (WABI 2014, Wroclaw), 9289 (WABI 2015, Atlanta), and 9838 (WABI 2016, Aarhus). As of 2017, they appeared in LIPIcs volumes 88 (WABI 2017, Boston), 113 (WABI 2018, Helsinki), 143 (WABI 2019, Boston), 172 (WABI 2020, virtually in Pisa) and 201 (WABI 2021, virtually in Chicago).

Christina Boucher & Sven Rahmann

WABI 2022 Committees

Steering Committee

Bernard Moret
EPFL, Switzerland

Vincent Moulton
University of East Anglia, UK

Jens Stoye
Bielefeld University, Germany

Tandy Warnow
University of Illinois at Urbana-Champaign

PC Chairs

Christina Boucher
University of Florida, USA

Sven Rahmann
Saarland University, Germany

Program Committee

Tatsuya Akutsu
Kyoto University

Marco Antoniotti
Università degli Studi di Milano-Bicocca

Jasmijn Baaijens
TU Delft

Anne Bergeron
Universite du Quebec à Montréal

Paola Bonizzoni
Università di Milano-Bicocca

Lenore Cowen
Tufts University

Gianluca Della Vedova
Univ. degli Studi Milano-Bicocca

Daniel Doerr
Heinrich Heine University Düsseldorf

Mohammed El-Kebir
University of Illinois at Urbana-Champaign

Nadia El-Mabrouk
University of Montreal

Travis Gagie
Diego Portales University

Anna Gambin
Warsaw University

Shilpa Garg
University of Copenhagen

Bjarni Halldorsson
deCODE genetics and Reykjavik University

Katharina Huber
University of East Anglila

Carl Kingsford
Carnegie Mellon University

Gregory Kucherov
CNRS and LIGM

Manuel Lafond
Université de Sherbrooke

Elodie Laine
Sorbonne Université

Yu Lin
The Australian National University

Stefano Lonardi
UC Riverside

Camille Marchet
CNRS

Guillaume Marçais
Carnegie Mellon University

Erin Molloy
University of Maryland at College Park

Vincent Moulton
University of East Anglia

Francesca Nadalin
EMBL

William Stafford Noble
University of Washington

Aida Ouangraoua
Université de Sherbrooke

22nd International Workshop on Algorithms in Bioinformatics (WABI 2022).
Editors: Christina Boucher and Sven Rahmann
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Ion Petre
University of Turku

Nadia Pisanti
University of Pisa

Solon Pissis
CWI Amsterdam

Alberto Policriti
University of Udine

Mihai Pop
University of Maryland

Teresa Przytycka
National Center of Biotechnology Information, NLM, NIH

Knut Reinert
FU Berlin

Eric Rivals
CNRS and LIRMM Montpellier

Sebastien Roch
UW Madison

Giovanna Rosone
Università di Pisa

Alexander Schoenhuth
Bielefeld University

Mingfu Shao
Carnegie Mellon University

Krister Swenson
CNRS and Université de Montpellier

Ewa Szczurek
University of Warsaw

Sharma V. Thankachan
University of Central Florida

Hélène Touzet
CNRS and CRIStAL, Lille

Tomas Vinar
Comenius University in Bratislava

Prudence Wong
University of Liverpool

Simone Zaccaria
UCL Cancer Institute

Louxin Zhang
National University of Singapore

Michael Ziv-Ukelson
Ben Gurion University of the Negev

Subreviewers

Paniz Abedin

Giulia Bernardini

Konstantinn Bonnet

Broňa Brejová

Laurent Bulteau

Sakshar Chakravarty

Ibrahim Chegrane

Ke Chen

Wei Chen

Seth Commichaux

Dan DeBlasio

Luca Denti

Abigail Djossou

Norbert Dojer

Hannes P. Eggertsson

Estéban Gabory

Askar Gafurov

Ali Ghaffaari

Daniel Gibney

Krzysztof Gogolewski

Khodor Hannoush

Quang Minh Hoang

Guillaume Holley

Dmitry Kosolobov

Benjamin Langmead

Felipe A. Louza

Anna Macioszek

Amirsadra Mohseni

Harihara Subrahmaniam Muralidharan

Njagi Mwaniki

Wend Yam Donald Davy Ouédraogo

Fabio Pardi

Luca Parmigiani

Murray Patterson

Yuri Pirola

Nicola Prezza

Simone Procaccia

Yutong Qiu

Raffaella Rizzi

Johannes Schlüter

Tizian Schulz

Remy Schwab

Saleh Sereshki

Yihang Shen

Michał Startek

Aakash Sur

Laura Tung

Gianvito Urgese

Sebastian Wild

Roland Wittler

Ran Zhang

Hongyu Zheng

ALGO Organizing Committee

Tobias Friedrich (chair)

Simon Krogmann

Timo Kötzing

Gregor Lagodzinski

Pascal Lenzner

(all at Hasso Plattner Institute,
University of Potsdam)

Efficient Solutions to Biological Problems Using de Bruijn Graphs

Leena Salmela
University of Helsinki, Finland

Abstract

The de Bruijn graph has become a standard method in the analysis of sequencing reads in computational biology due to its ability to represent the information contained in large read sets in small space. A de Bruijn graph represents a set of sequencing reads by its k-mers, i.e. the set of substrings of length k that occur in the reads. In the classical definition, the k-mers are the edges of the graph and the nodes are the $k-1$ bases long prefixes and suffixes of the k-mers. Usually only k-mers occurring several times in the read set are kept to filter out noise in the data. De Bruijn graphs have been used to solve many problems in computational biology including genome assembly [4, 9, 1, 8], sequencing error correction [10, 7, 11, 5], reference free variant calling [13], indexing read sets [6], and so on. Next I will discuss two of these problems in more depth.

The de Bruijn graph first emerged in computation biology in the context of genome assembly [4, 9] where the task is to reconstruct a genome based on sequencing reads. As the de Bruijn graph can represent large read sets compactly, it became the standard approach to assemble short reads [1, 8]. In the theoretical framework of de Bruijn graph based genome assembly, a genome is thought to be the Eulerian path in the de Bruijn graph built on the sequencing reads. In practise, the Eulerian path is not unique and thus not useful in the biological context. Therefore, practical implementations report subpaths that are guaranteed to be part of any Eulerian path and thus part of the actual genome. Such models include unitigs, which are nonbranching paths of the de Bruijn graph, and more involved definitions such as omnitigs [12].

In genome assembly the choice of k is a crucial matter. A small k can result in a tangled graph, whereas a too large k will fragment the graph. Furthermore, a different value of k may be optimal for different parts of the genome. Variable order de Bruijn graphs [3, 2], which represent de Bruijn graphs of all orders k in a single data structure, have been proposed as a solution but no rigorous definition corresponding to unitigs has been presented. We give the first definition of assembled sequences, i.e. contigs, on such graphs and an algorithm for enumerating them.

Another problem that can be solved with de Bruijn graphs is the correction of sequencing errors [10, 7, 11, 5]. Because each position of a genome is sequenced several times, it is possible to correct sequencing errors in reads if we can identify data originating from the same genomic region. A de Bruijn graph can be used to represent compactly the reliable information and the individual reads can be corrected by aligning them to the graph.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Applied computing → Sequencing and genotyping technologies

Keywords and phrases de Bruijn graph, variable order de Bruijn graph, genome assembly, sequencing error correction, k-mers

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.1

Category Invited Talk

Funding Supported by Academy of Finland [grants 308030 and 323233].

References

1 Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A. Gurevich, Mikhail Dvorkin, Alexander S. Kulikov, Valery M. Lesin, Sergey I. Nikolenko, Son Pham, Andrey D. Prjibelski, Alexey V. Pyshkin, Alexander V. Sirotkin, Nikolay Vyahhi, Glenn Tesler, Max A. Alekseyev, and Pavel A. Pevzner. SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012.

2 Djamal Belazzougui, Travis Gagie, Veli Mäkinen, Marco Previtali, and Simon J. Puglisi. Bidirectional variable-order de Bruijn graphs. In *Proceedings of LATIN 2016*, pages 164–178. Springer, 2016.

3 Christina Boucher, Alex Bowe, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane. Variable-order de Bruijn graphs. In *Proceedgins of Data Compression Conference 2015*, pages 383–392, 2015.

4 Ramana M. Idury and Michael S. Waterman. A new algorithm for DNA sequence assembly. *Journal of Computational Biology*, 2(2):291–306, 1995.

5 Antoine Limasset, Jean-François Flot, and Pierre Peterlongo. Toward perfect reads: self-correction of short reads via mapping on de Bruijn graphs. *Bioinformatics*, 36(5):1374–1381, 2019.

6 Camille Marchet, Christina Boucher, Simon J. Puglisi, Paul Medvedev, Mikaël Salson, and Rayan Chikhi. Data structures based on k-mers for querying large collections of sequencing data sets. *Genome Research*, 31:1–12, 2021.

7 Giles Miclotte, Mahdi Heydari, Piet Demeester, Stephane Rombauts, Yves Van de Peer, Pieter Audenaert, and Jan Fostier. Jabba: hybrid error correction for long sequencing reads. *Algorithms for Molecular Biology*, 11(10), 2016.

8 Yu Peng, Henry C. M. Leung, S. M. Yiu, and Francis Y. L. Chin. IDBA – A practical iterative de Bruijn graph de novo assembler. In *Proceedings of RECOMB 2010*, volume 6044 of *LNBI*, pages 426–440. Springer, 2010.

9 Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Acadademy of Sciences*, 98(17):9748–9753, 2001.

10 Leena Salmela and Eric Rivals. LoRDEC: accurate and efficient long read error correction. *Bioinformatics*, 30(24):3506–3514, 2014.

11 Leena Salmela, Riku Walve, Eric Rivals, and Esko Ukkonen. Accurate selfcorrection of errors in long reads using de Bruijn graphs. *Bioinformatics*, 33(6):799–806, 2017. (Also in RECOMB-seq 2016).

12 Alexandru I. Tomescu and Paul Medvedev. Safe and complete contig assembly through omnitigs. *Journal of Computational Biology*, 24(6):590–602, 2017.

13 Raluca Uricaru, Guillaume Rizk, Vincent Lacroix, Elsa Quillery, Olivier Plantard, Rayan Chikhi, Claire Lemaitre, and Pierre Peterlongo. Reference-free detection of isolated SNPs. *Nucleic Acids Research*, 43(2):e11, 2015.

Eulertigs: Minimum Plain Text Representation of k-mer Sets Without Repetitions in Linear Time

Sebastian Schmidt[1]
University of Helsinki, Finland

Jarno N. Alanko
University of Helsinki, Finland

Abstract

A fundamental operation in computational genomics is to reduce the input sequences to their constituent k-mers. For maximum performance of downstream applications it is important to store the k-mers in small space, while keeping the representation easy and efficient to use (i.e. without k-mer repetitions and in plain text). Recently, heuristics were presented to compute a near-minimum such representation. We present an algorithm to compute a minimum representation in optimal (linear) time and use it to evaluate the existing heuristics. For that, we present a formalisation of arc-centric bidirected de Bruijn graphs and carefully prove that it accurately models the k-mer spectrum of the input. Our algorithm first constructs the de Bruijn graph in linear time in the length of the input strings (for a fixed-size alphabet). Then it uses a Eulerian-cycle-based algorithm to compute the minimum representation, in time linear in the size of the output.

2012 ACM Subject Classification Applied computing → Computational biology; Theory of computation → Data compression; Theory of computation → Graph algorithms analysis; Theory of computation → Data structures design and analysis

Keywords and phrases Spectrum preserving string sets, Eulerian cycle, Suffix tree, Bidirected arc-centric de Bruijn graph, k-mer based methods

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.2

Related Version *Full Version*: https://www.biorxiv.org/content/10.1101/2022.05.17.492399

Supplementary Material *Software (Source Code)*: https://github.com/algbio/matchtigs, archived at swh:1:dir:e33705e470606ba7ba1d670041010c056a781fb8
Software (Experiment Pipeline): https://doi.org/10.5281/zenodo.6538261

Funding *Sebastian Schmidt*: Funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851093, SAFEBIO).
Jarno N. Alanko: Funded by NIH NIAID grant No. R01HG011392 and Academy of Finland grant 339070.

1 Introduction

Motivation

A k-mer is a DNA string of length k that is considered equal to itself and its reverse complement. A common pattern in bioinformatics is to reduce a set of input strings to their constituent k-mers. Such representations are at the core of many bioinformatics pipelines – see e.g. Schmidt et al. [23] or Brinda et al. [6] for an overview of applications. The wide-spread use of k-mer sets has prompted the question of what is the smallest *plain text representation* for a set of k-mers. Here, a plain text representation means a set of strings that have the same set of k-mers as the input strings, i.e. the *spectrum* is preserved. Such representations are also called *spectrum preserving string sets* (SPSS) [22], or simplitigs [6]. This has the following advantages over encoded representations:

[1] corresponding author

- When storing k-mer sets to disk, plain text may remove the need of decompression before usage, as some tools that usually take unitigs as input can take any other plain text representation without modification (e.g. Bifrost [13]).
- Within an application, an encoded representation would require decoding whenever a k-mer is accessed, which may slow down the application a lot compared to when each k-mer is in RAM in plain text.

Further, in applications, it might be useful if the representation contains each k-mer exactly once. This is because some applications, like e.g. SSHash [21], are able to take any set of k-mers as input, but cannot easily deal with duplicate k-mers in the input.

Related work

There are two heuristic approaches to the construction of a small SPSS without repetitions, namely *prophasm* [6] and *UST* [22]. While neither of these computes a minimum representation, Rahman et al. [22] also present a lower bound to the minimum size of any representation without repetition, and they show that they are within 3% of this lower bound in practice. They also present a counter-example showing that their lower bound is not tight. Small SPSSs without repetitions are used e.g. in SSHash [21] and are also computed by state-of-the-art de Bruijn graph compactors like Cuttlefish 2 [15].

When k-mer repetitions are allowed in an SPSS, there is a known polynomially computable minimum representation, namely *matchtigs* [23]. While matchtigs are expensive to compute, the authors also present a more efficient greedy heuristic that is able to compute a near-minimum representation on a modern server with no significant penalty in runtime (when compared to computing just unitigs), but a significant increase in RAM usage.

In [6, 23] the authors also showed that decreasing the size of an SPSS results in significantly better performance in downstream applications, i.e. when further compressing the representation with general purpose compressors, or when performing k-mer-based queries.

The authors of both [6] and [22] consider whether computing a minimum representation without repetitions may be NP-hard, as it is equivalent to computing a minimum path cover in a de Bruijn graph, which is NP-hard in general graphs by reduction from Hamiltonian cycle. However, computing a Hamiltonian cycle in a de Bruijn graph is actually polynomial [14]. The authors of [14] argue that de Bruijn graphs are a subclass of *adjoint* graphs, in which solving the Hamiltonian cycle problem is equivalent to solving the Eulerian cycle problem in the *original* of the adjoint graph, which can be computed in linear time[2]. However, the argument is only made for normal directed (and not bidirected) graphs, and thus is not applicable to our setup, where a k-mer is also considered equal to its reverse complement.

Our contributions

Our first technical contribution is to carefully define the notion of a bidirected de Bruijn graph such that the spectrum of the input is accurately modelled in the allowed walks of the graph. Our definition also takes into account k-mers that are their own reverse complement. This technicality is often neglected in the literature, and sidestepped by requiring that the

[2] The original of an adjoint graph can be computed by splitting each node v into two nodes v' and v'' such that v' keeps the incoming arcs, and v'' the outgoing arcs as in [5, Figure 4]. Then, the graph is a collection of complete bipartite graphs [5]. These graphs can be contracted into single nodes, and then we add an arc between the contracted representations of each v' and v''. This can be computed in linear time and is the original graph, since all nodes have become arcs again, and the arcs have the correct predecessors and successors.

value of k is odd, in which case this special case does not occur. We give a suffix-tree-based deterministic linear-time algorithm to construct such a graph, filling a theory gap in the literature, as existing approaches [8, 15, 13, 1] depend on the value of k and/or are probabilistic due to the of use hashing, minimizers or Bloom filters, or do not use the reverse-complement-aware definition of k-mers [7].

Given the bidirected de Bruijn graph, we present an algorithm that computes a minimum plain text representation of k-mer sets without repetitions, which runs in output sensitive linear time. Steps 1 to 3 run in linear time in the number of nodes and arcs in the graph. In short, it works as follows:

1. Add breaking arcs into this graph to make it Eulerian.
2. Compute a Eulerian cycle in the resulting graph.
3. Break that cycle at the breaking arcs.
4. Output the strings spelled by the resulting walks.

The algorithm is essentially an adaption of the matchtigs algorithm [23], removing the possibility of joining walks by repeating k-mers. We give detailed descriptions for all these steps and prove their correctess in our bidirected de Bruijn graph model. Together with our linear-time de Bruijn graph construction algorithm, we obtain the main result of our paper:

▶ **Theorem 1.** *Let k be a positive integer and let I be a set of strings of length at least k over some alphabet Σ. Then we can compute a set of strings I' of length at least k with minimum cumulative length and $\mathrm{CS}_k(I) = \mathrm{CS}_k(I')$ in $O(||I|| \log |\Sigma|)$ time.*

where $\mathrm{CS}_k(I) = \mathrm{CS}_k(I')$ means that I' is an SPSS of I, and $||I||$ is the cumulative length of I (see Section 2 for accurate definitions). This gives a positive answer to the open question if a minimum SPSS without repetitions can be computed in polynomial time. Additionally, we give an easily computable tight lower bound on the size of a minimum SPSS without repetitions.

For our experiments, we have implemented steps 1 to 4 in Rust, taking the de Bruijn graph as given. The implementation is available on github: `https://github.com/algbio/matchtigs`. Our experimental evaluation shows that our algorithm does not result in significant practical improvements, but for the first time allows to benchmark the quality the heuristics prophasm and UST against an optimal solution. It turns out that both produce close-to-optimal results, but with a different distribution of computational resources.

Our work also shows that using arc-centric de Bruijn graphs can aid the intuition for certain problems, as in this case, the node-centric variant hides the relationship between Eulerian cycles and minimum SPSS without repetition.

Organisation of the paper

In Section 2 we give preliminary definitions of well-known concepts. In Section 3 we define de Bruijn graphs and prove the soundness of the definitions. In Section 4 we show how to construct de Bruijn graphs by our definitions in linear time. In Section 5 we show how to construct a minimum SPSS without repetitions in linear time if the de Bruijn graph is given. In Section 6 we compare our algorithm and Eulertigs against strings computed with prophasm and UST on practical data sets.

2 Preliminaries

In this section we give the prerequisite knowledge required for this paper.

2.1 Bidirected graphs

In this section we define our notion of the bidirected graphs and the incidence model.

A multiset is defined as a set M, and an implicit function $\#_M : M \to \mathbb{Z}^+$ mapping elements to their multiplicities. The cardinality is defined as $|M| := \sum_{s \in M} \#_M(s)$.

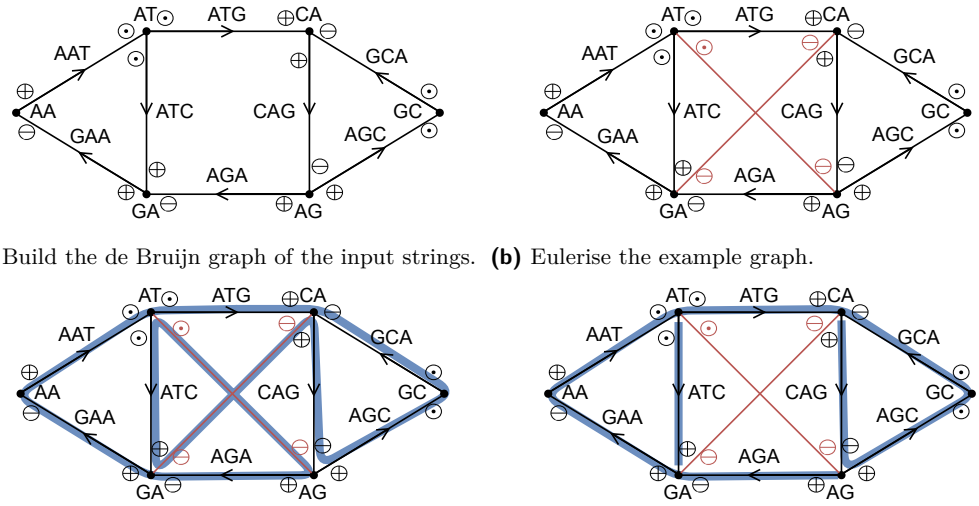
An *alphabet* Σ is an ordered set, and an Σ-*word* is a string of characters of that set. String concatenation is written as ab for two strings a and b. The set Σ^k is the set of all Σ-words of length k and the set Σ^* is the set of all Σ-words, including the empty word ϵ. Given a positive integer k, the k-*suffix* $\mathrm{suf}_k(w)$ (k-*prefix* $\mathrm{pre}_k(w)$) of a word w is the substring of its last (first) k characters. A k-*mer* is a word of length k. A *complement function* over Σ is a function $\mathrm{comp} : \Sigma \to \Sigma$ mapping characters to characters that is self-inverse (i.e. $\mathrm{comp}(\mathrm{comp}(x)) = x$). A *reverse complement* function for alphabet Σ is a function $\mathrm{rc} : \Sigma^* \to \Sigma^*$ defined as $\mathrm{rc}((w_1, \ldots, w_\ell)) := (\mathrm{comp}(w_\ell), \ldots, \mathrm{comp}(w_1))$, for some arbitrary complement function comp. On sets, rc is defined to compute the reverse complement of each element in the set. Note that rc is self-inverse. A *canonical* k-mer is a k-mer that is lexicographically smaller than or equal to its reverse complement.

Given an integer k and an alphabet Σ, the k-*spectrum* of a set of strings $I \subseteq \bigcup_{k' \geq k} \Sigma^{k'}$ is a set of strings $\mathrm{S}_k(I) := \{w \in \Sigma^k \mid \exists i \in I : w \text{ is substring of } i \text{ or } \mathrm{rc}(i)\}$. The *canonical* k-*spectrum* of I is $\mathrm{CS}_k(I) := \{w \in \mathrm{S}_k(I) \mid w \text{ is canonical}\}$. For simplicity, the spectrum and canonical spectrum are defined for a single string w as if it were a set $\{w\}$. A *spectrum preserving string set* of a set of strings I is a set of strings I' such that $\mathrm{CS}_k(I) = \mathrm{CS}_k(I')$. The cumulative length of I is $||I|| := \sum_{w \in I} |w|$.

Our definition of a bidirected graph is mostly standard like in e.g. [17], however we allow self-complemental nodes that occur in bidirected de Bruijn graphs. A *bidirected graph* is a tuple $G = (V, E, c)$ with a set of normal and *self-complemental* nodes $v \in V$, a set of arcs $e \in E$, and a function $c : V \to \{1, 0\}$ marking self-complemental nodes with 1, and normal nodes with 0. An *incidence* is a pair vd, where $d \in \{\oplus, \ominus, \odot\}$ is called its *sign* (e.g. $v\oplus$). The negation of a sign is defined as $\neg\oplus := \ominus$, $\neg\ominus := \oplus$ and $\neg\odot := \odot$. For self-complemental nodes $v \in V$, only incidences $v\odot$ are allowed, and for normal nodes only incidences $v\oplus$ and $v\ominus$ are allowed. An *arc* $(v_1 d_1, v_1' d_1', \eta) \in E$ is a tuple of incidences and a unique identifier η, where η can be of any type. The *reversal* of an arc is denoted by $(v_1 d_1, v_1' d_1', \eta)^{-1} := (v_1' d_1', v_1 d_1, \eta)$. If not required, we may drop the identifier (i.e. just write $(v_1\ominus, v_1'\odot) \in E$). If a node $v \in V$ is present with a $\oplus$ ($\ominus$) sign in an arc, then the arc is *outgoing* (*incoming*) from (to) v.

Note that, other than in standard directed graphs, in bidirected graphs arcs can be outgoing or incoming on both ends, and the order of the incidences in the arc does not affect if it is outgoing or incoming to a node. Further, our notation differs from that of standard bidirected graphs in that arcs have a direction. This is required because we will work with arc-centric de Bruijn graphs (see Section 3), which have labels on the arcs and not the nodes. Using the sign of the incidence pairs, it is possible to decide if a node is traversed forwards or backwards, but not if the arc is traversed forwards or backwards. But to decide which label (forwards or reverse complement) to use when computing the string spelled by an arc, the direction is relevant. See Figure 1 (a) for an example of a bigraph, which has labels that make it a de Bruijn graph as well.

A *walk* in a bigraph is a sequence of arcs $W := ((v_1 d_1, v_1' d_1', \eta_1), (v_2 d_2, v_2' d_2', \eta_2), \ldots, (v_\ell d_\ell, v_\ell' d_\ell', \eta_\ell))$ where for every i it holds that $(v_i d_i, v_i' d_i', \eta_i) \in E$ or $(v_i' d_i', v_i d_i, \eta_i) \in E$ (we can arbitrarily walk over arcs forwards and reverse), and for every $i < \ell$ it holds that $v_i' = v_{i+1}$ and $d_i' = \neg d_{i+1}$. The *length* of a walk is $\ell = |W|$. If $v_1 = v_\ell'$ and $d_1 = \neg d_\ell'$, then W is a *cycle*. A bigraph is *connected*, if for each pair of nodes $v_1, v_2 \in V$ there is a walk from v_1 to v_2.



(a) Build the de Bruijn graph of the input strings. **(b)** Eulerise the example graph.

(c) Compute a Eulerian cycle. **(d)** Break the cycle at the arcs inserted in step (b).

Figure 1 Overview of our algorithm executed on the input strings $\{GAATG, ATCTGCT\}$ with $k = 3$. After step (d), the resulting spelled SPSS is $\{ATC, AGAATGCTG\}$.

For a node $v \in V$, the *multiset of incidences* is defined as $\mathrm{I}(v) := \{vd \mid d \in \{\oplus, \ominus, \odot\}\}$, with multiplicities $\#_{\mathrm{I}(v)}(vd) := \sum_{e \in E} \#_e(vd)$ (treating the arcs as multisets such that self-loops count as two separate incidences). For a node $v \in V$ that is not self-complemental, the *outdegree* is defined as $\delta^+(v) := \#_{\mathrm{I}(v)}(v\oplus)$, and the *indegree* is defined as $\delta^-(v) := \#_{\mathrm{I}(v)}(v\ominus)$. For a self-complemental node $v \in V$, the *degree* is defined as $\delta(v) := \#_{\mathrm{I}(v)}(v\odot)$.

We define the *imbalance* of a node $v \in V$ that is not self-complemental as the difference of its outdegree and indegree $\mathrm{imbalance}(v) := \delta^+(v) - \delta^-(v)$. For a self-complemental node $v \in V$ the imbalance is defined as $\mathrm{imbalance}(v) := 1$ if $\delta(v)$ is odd, and $\mathrm{imbalance}(v) := 0$ otherwise. A node $v \in V$ is called *unbalanced*, if $\mathrm{imbalance}(v) \neq 0$, and *balanced* otherwise.

A *labelled graph* is a bidirected graph $G = (V, E, c)$ where the identifiers of arcs are strings over some alphabet Σ (e.g. $(v_1\oplus, v_2\ominus, ACCTG) \in E$).

2.2 Suffix arrays and suffix trees

Section 4 requires knowledge of suffix arrays and suffix trees. We assume the reader is familiar with these data structures, and briefly give the relevant definitions and properties below. We point the reader to Gusfield [12] and Mäkinen [18] for an in-depth treatment of the topics.

A suffix array SA_T for a string T is an array of length $|T|$ such that $SA_T[i]$ is the starting position of the lexicographically i-th suffix of T. The suffix array interval of a string x is the maximal interval $[i..j]$ such that all the suffixes pointed by $SA_T[i], \ldots, SA_T[j]$ have x as a prefix, or the empty interval if x is not a substring of T.

A suffix tree of a string T is a compacted version of the trie of all suffixes of T, such that non-branching paths are merged into single arcs, with arcs pointing away from the root. The compactification concatenates the labels of the arcs on the compacted path. The nodes that were compacted away and are now in the middle of an arc are called implicit nodes, and the rest of the nodes are explicit. A *locus* (plural *loci*) is a node that is either explicit or implicit. A locus v is represented by a pair (u, d), where u is the explicit suffix tree node at the end of the arc containing v (u is equal to v if v is explicit), and d is the depth of locus v in the trie of loci. The suffix array interval of a node is the interval of leaves in the subtree of the node. The suffix array interval of an implicit locus (u, d) is the same as the suffix array interval of u.

The suffix tree can be constructed in linear time in $|T|$ using e.g. Ukkonen's algorithm [24]. The tree comes with a function `child` that takes an explicit node and a character, and returns the child at the end of the arc from that node whose label starts with the given character (if such node exists). This can be implemented in $O(\log |\Sigma|)$ time by binary searching over child pointers sorted by labels. The `child` function can also be easily implemented for implicit loci. Ukkonen's algorithm also produces *suffix links* for the explicit nodes, which map from the suffix tree node of a string cx to the suffix tree node of string x. It is possible to emulate suffix links on the implicit loci using constant-time weighted level-ancestor queries [4] by mapping $(u, d) \mapsto (f_{d-1}(SL(u)), d-1)$, where $SL(u)$ is the destination of a suffix link from u, and $f_{d-1}(SL(u))$ is the furthest suffix tree ancestor from $SL(u)$ at depth at least $d-1$ in the trie of loci. The inverse pointers of suffix links are called *Weiner links*, and they can also be simulated on the implicit loci by mapping $(u, d) \mapsto (WL(u, c), d+1)$, where $WL(u, c)$ is the destination of a Weiner link from u with character c.

3 De Bruijn graphs

Algorithm 1 DeBruijnGraph.

Input: An integer k and a set of strings I where each string has length at least k.
Output: A de Bruijn graph $G = (V, E, c)$ of order k.

$V \leftarrow \mathrm{CS}_{k-1}(I)$ `/* the nodes are the canonical (k − 1)-mers */`
foreach $v \in V$ **do**
  if $\mathrm{rc}(v) = v$ **then** $c(v) \leftarrow 1$ **else** $c(v) \leftarrow 0$
foreach $\eta \in \mathrm{CS}_k(I)$ **do**
  $w \leftarrow \mathrm{pre}_{k-1}(\eta)$ `/* compute v */`
  $v \leftarrow$ canonical w
  $w' \leftarrow \mathrm{suf}_{k-1}(\eta)$ `/* compute v′ */`
  $v' \leftarrow$ canonical w'
  if $c(v) = 1$ **then** $d \leftarrow \odot$ `/* compute the direction of v */`
  else if $v = w$ **then** $d \leftarrow \oplus$
  else $d \leftarrow \ominus$
  if $c(v') = 1$ **then** $d' \leftarrow \odot$ `/* compute the direction of v′ */`
  else if $v' \neq w'$ **then** $d' \leftarrow \oplus$ `/* note that ≠ differs from = above */`
  else $d' \leftarrow \ominus$
  $e \leftarrow (vd, v'd', \eta)$ `/* insert the arc into the graph */`
  $E \leftarrow E \cup \{e\}$

The *de Bruijn graph* of order k of a set of input strings I is defined as a labelled graph constructed by Algorithm 1. See Figure 1 (a) for an example. A de Bruijn graph computed by this algorithm has the following property (see Appendix B for some of the proofs of this section).

▶ **Lemma 2** (Sound labels). *Let k be a positive integer and let I be a set of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. For all pairs of arcs $e_1 := (v_1 d_1, v_1' d_1', \eta_1), e_2 := (v_2 d_2, v_2' d_2', \eta_2) \in E$ it holds that:*
(a) *$(v_1' = v_2$ and $d_1' = \neg d_2)$ if and only if $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\eta_2)$,*
(b) *$(v_1' = v_2'$ and $d_1' = \neg d_2')$ if and only if $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_2))$,*
(c) *$(v_1 = v_2$ and $d_1 = \neg d_2)$ if and only if $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_1)) = \mathrm{pre}_{k-1}(\eta_2)$, and*
(d) *$(v_1 = v_2'$ and $d_1 = \neg d_2')$ if and only if $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_1)) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_2))$.*

Algorithm 2 SPELL.

Input: A de Bruijn graph $G = (V, E, c)$ of order k and a walk $W = (e_1 := (v_1 d_1, v'_1 d'_1, \eta_1), \ldots, e_\ell := (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$.
Output: The string s spelled by W, i.e. spell(W).

```
if W is empty then
    s ← ϵ
else                                   /* compute the sequence of kmers from W */
    foreach i ∈ (1, ..., ℓ) do         /* iterate the sequence in order */
        if e_i ∈ E then κ_i ← η_i
        else κ_i ← rc(η_i)             /* e_i^{-1} ∈ E */
    s ← k − 1 prefix of κ_1
    foreach i ∈ (1, ..., ℓ) do         /* iterate the sequence in order */
        append the last character from κ_i to s
```

For a walk $W := (e_1 = (v_1 d_1, v'_1 d'_1, \eta_1), \ldots, e_\ell = (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ in a de Bruijn graph, its *sequence of k-mers* is $K := (\kappa_1, \ldots, \kappa_\ell)$, where for each i we define κ_i as η_i if $e_i \in E$, and as $\text{rc}(\eta_i)$ if $e_1^{-1} \in E$. The string spell(W) is the string *spelled* by W, which is defined as its *collapsed* sequence of kmers, i.e. its sequence of k-mers gets concatenated while overlapping consecutive k-mers by $k - 1$. This is computed by Algorithm 2. We prove the following lemmas to show that our definition of the spell$(\cdot)$ function is sound for our purposes, i.e. correctly spells the string belonging to a walk in a de Bruijn graph.

▶ **Lemma 3** (Sound sequence of k-mers). *Let k be a positive integer and let I be a set of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. Let $W := (e_1 = (v_1 d_1, v'_1 d'_1, \eta_1), \ldots, e_\ell = (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ be a walk in G, and $K := (\kappa_1, \ldots, \kappa_\ell)$ its sequence of k-mers. Then for each consecutive pair of kmers κ_i, κ_{i+1} it holds that $\text{suf}_{k-1}(\kappa_i) = \text{pre}_{k-1}(\kappa_{i+1})$.*
We define the *sequence of k-mers* $K = (\kappa_1, \ldots, \kappa_\ell)$ of a string $w = (a_1, \ldots, a_{\ell+k-1})$ by $\kappa_i := (a_i, \ldots, a_{i+k-1})$ for each i.

▶ **Lemma 4** (Sound spell). *Let k be a positive integer and let I be a set of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. Let W be a walk in G, K_W its sequence of k-mers and K'_W the sequence of k-mers of* spell(W). *Then $K_W = K'_W$.*

▶ **Lemma 5** (Complete representation). *Let k be a positive integer and let I be a set of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. Let w be a string with $\text{CS}_k(w) \subseteq \text{CS}_k(I)$. Then there exists a walk W in G with* spell$(W) = w$.

Proof. Let $K_w = (\kappa_1, \ldots, \kappa_\ell)$ be the sequence of k-mers of w. We construct $W = (e_1 = (v_1 d_1, v'_1 d'_1, \eta_1), \ldots, e_\ell = (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ as follows: for each i, let η_i be the canonical of κ_i and $f_i \in E$ be the arc whose identifier is η_i. We set $e_i = f_i$ if κ_i is canonical, and $e_i = f_i^{-1}$ otherwise.

For W to fulfil the definition of a walk we need that $v'_i = v_{i+1}$ and $d'_i = \neg d'_{i+1}$ for all i. Using Lemma 2, we get:

- If $e_i, e_{i+1} \in E$, then $\text{suf}_{k-1}(\eta_i) = \text{suf}_{k-1}(\kappa_i) = \text{pre}_{k-1}(\kappa_{i+1}) = \text{pre}_{k-1}(\eta_{i+1})$. Therefore, by Lemma 2 a, it holds that $v'_i = v_{i+1}$ and $d'_i = \neg d'_{i+1}$.

- If $e_i, e_{i+1}^{-1} \in E$, then $\text{suf}_{k-1}(\eta_i) = \text{suf}_{k-1}(\kappa_i) = \text{pre}_{k-1}(\kappa_{i+1}) = \text{pre}_{k-1}(\text{rc}(\eta_{i+1}))$. Therefore, by Lemma 2 b, it holds that $v'_i = v_{i+1}$ and $d'_i = \neg d'_{i+1}$.
- If $e_i^{-1}, e_{i+1} \in E$, then $\text{suf}_{k-1}(\text{rc}(\eta_i)) = \text{suf}_{k-1}(\kappa_i) = \text{pre}_{k-1}(\kappa_{i+1}) = \text{pre}_{k-1}(\eta_{i+1})$. Therefore, by Lemma 2 c, it holds that $v'_i = v_{i+1}$ and $d'_i = \neg d'_{i+1}$.
- If $e_i^{-1}, e_{i+1}^{-1} \in E$, then $\text{suf}_{k-1}(\text{rc}(\eta_i)) = \text{suf}_{k-1}(\kappa_i) = \text{pre}_{k-1}(\kappa_{i+1}) = \text{pre}_{k-1}(\text{rc}(\eta_{i+1}))$. Therefore, by Lemma 2 d, it holds that $v'_i = v_{i+1}$ and $d'_i = \neg d'_{i+1}$.

To complete the proof we need to show that $\text{spell}(W) = w$. By definition, the sequence of k-mers K_W of W is equivalent to K_w. And since W is a walk, by Lemma 4 we get that the sequence of k-mers of $\text{spell}(W)$ is equivalent to K_W, and therefore $\text{spell}(W) = w$. ◀

A *walk cover* $\mathcal{W}$ of a bigraph G is a set of walks such that for each arc $e \in E$ it holds that e is part of some walk $W \in \mathcal{W}$, or e^{-1} is part of some walk $W \in \mathcal{W}$.

▶ **Theorem 6** (Dualism between SPSS and walk cover)**.** *Let k be a positive integer and let I and I' be sets of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. Then it holds that $\text{CS}_k(I) = \text{CS}_k(I')$, if and only if there is a walk cover $\mathcal{W}$ in G that spells the strings in I'.*

Proof. If $\text{CS}_k(I') \subseteq \text{CS}_k(I)$, then for each string $w' \in I'$ it holds that $\text{CS}_k(w') \subseteq \text{CS}_k(I)$. Therefore, by Lemma 5, there exists a walk w in G with $\text{spell}(w) = w'$. Then, the set of all such walks $\mathcal{W}$ spells I'. Further, because $\text{CS}_k(I) \subseteq \text{CS}_k(I')$, the identifier η of each arc $e \in E$ is in $\text{CS}_k(I')$, and therefore in the sequence of kmers $K_{w'}$ of some string $w' \in I'$ (possibly as a reverse complement). By Lemma 4 it holds that $K_{w'} = K_w$, where K_w is the sequence of k-mers of walk w. By the definition of the sequence of k-mers of a walk, this implies that w visits e (possible in reverse direction). Since this holds for each $e \in E$, it holds that $\mathcal{W}$ is a walk cover of G.

Assume that there is a walk cover $\mathcal{W}$ in G that spells the strings in I', and let $w \in \mathcal{W}$ be a walk, K_w its sequence of k-mers, $w' := \text{spell}(w)$ and $K_{w'}$ the sequence of k-mers of w'. Then, by Lemma 4, $K_w = K_{w'}$, which, by the definition of the sequence of k-mers of a walk implies that $\text{CS}_k(I) \subseteq \text{CS}_k(I')$. And since $\mathcal{W}$ is a walk cover of G, we get $\text{CS}_k(I) = \text{CS}_k(I')$. ◀

▶ **Corollary 7.** *By setting $I = I'$ in Theorem 6 we can confirm that our definition of a de Bruijn graph is sound in that there is a set of walks that spells the strings used for its construction.*

A *compacted* de Bruijn graph is constructed from a de Bruijn graph by contracting all nodes $v \in V$ that are either self-complemental and have exactly two arcs that have exactly one incidence to v each, or that are not self-complemental and have exactly one incoming and one outgoing arc. For simplicity, we use uncompacted de Bruijn graphs in our theoretical sections, however all results equally apply to compacted de Bruijn graphs.

4 Linear-time construction of compacted bidirected de Bruijn graphs

In this section, we fill a gap in the literature by describing on a high level an algorithm to construct the bidirectional de Bruijn graph of a set of input strings in time linear in the total length of the input strings, independent of the value of k.

4.1 Algorithm

Let $I = \{w_1, \dots w_m\}$ be the set of input strings. Consider the following concatenation:

$$T = \$w_1\$w_2\$ \dots \$w_m\$ \operatorname{rc}(w_1)\$ \operatorname{rc}(w_2)\$ \dots \$ \operatorname{rc}(w_m)\$,$$

where \$ is a special character outside of the alphabet Σ of the input strings. We require an index on T that can answer the following queries: `extendRight`, `extendLeft`, `contractRight` and `contractLeft` in constant time. The extension operations take as input a character $c \in \Sigma$ and the interval of a string x in the suffix array of T, and return the suffix array intervals of xc in the case of `extendRight` and cx in the case of `extendLeft`. The contraction operations are the inverse operations of these, mapping the suffix array intervals of xc to x in the case of `contractRight` and cx to x in the case of `contractLeft`. For efficiency, we also require operations `enumerateRight` and `enumerateLeft`, which take a string x and give all characters such that `extendRight` and `extendLeft` respectively return a non-empty interval, in time that is linear in the number of such characters. Implementations for all the six subroutines are given in Section 4.2.

Using these operations, we can simulate the regular non-bidirected de Bruijn graph of T. Each k-mer of the input strings for a fixed k corresponds to a disjoint interval in the suffix array of T. The nodes are represented by their suffix array intervals. The outgoing arcs from a $(k-1)$-mer x are those characters c where `extendRight`(x, c) returns a non-empty interval. We can enumerate all the characters c with this property in constant time using `enumerateRight`(x). The incoming arcs can be enumerated symmetrically with the `enumerateLeft`(x). Finally, we can find the destination or origin of an arc labelled with x by running a `contractLeft` or `contractRight` operation respectively on x.

To construct the bidirected de Bruijn graph, we merge together nodes that are the reverse complement of each other. To find which nodes are complemental, we scan the input strings I while maintaining the suffix array interval of the current k-mer using `extendRight` and `contractLeft` operations, while at the same time maintaining the suffix array interval of the reverse complement using `extendLeft` and `contractRight` operations. Whenever we merge two nodes, we combine the incoming and outgoing arcs, assigning the incidences of the arcs according to the incidence rules in our definition. We are able to tell in constant time which k-mer of a pair of complemental k-mers is canonical by comparing the suffix array intervals of the k-mers: the k-mer whose suffix array interval has a smaller starting point is the canonical k-mer. If the starting points are the same, the k-mer is self-complemental.

Using the `enumerateRight` and `enumerateLeft` functions, we can check if a node would be contracted in a compacted de Bruijn graph. By extending k-mers over such nodes, we can in linear time also output only the arcs and nodes of a compacted de Bruijn graph. For storing the labels, we use one pointer into the input strings to store a single k-mer, as well as a flag that is set whenever the label is not canonical. If a label has multiple k-mers, then we store the remaining k-mers as explicit strings, however without their overlap with the "pointer-k-mer". This way, we can store each label in $O(\ell)$ space, where ℓ is the number of k-mers in the label. We additionally store the first and last character of each label, as an easy way to make the SPELL function run in output sensitive linear time.

4.2 Implementation of the subroutines

All required the subroutines `extendRight`, `extendLeft`, `contractRight`, `contractLeft`, `enumerateRight` and `enumerateLeft` can be implemented with the suffix tree of T by simulating the trie of the suffix tree loci as described in Section 2.2. The suffix array intervals

of explicit nodes can be stored with the nodes, so that we can operate on loci (u, d) and retrieve the suffix array intervals on demand. The operation `extendRight` follows an arc from a locus to a child, and the operation `contractRight` is implemented by going to the parent of the current locus. The operation `contractLeft` follows a suffix link from the current locus, and `extendLeft` follows a Weiner link. The operations `enumerateRight` and `enumerateLeft` are implemented by storing the children and the Weiner links from explicit suffix tree nodes as neighbor lists. The total number of these links is linear in $|T|$ [18]. With this implementation, the slowest operations are `extendRight` and `extendLeft`, taking $O(\log|\Sigma|)$ time to binary search the neighbor lists. We therefore obtain the following result:

▶ **Theorem 8.** *The compacted arc-centric bidirected de Bruijn graph of order k of a set of input strings I from the alphabet Σ can be constructed in time $O(||I||\log|\Sigma|)$.*

We note that the same operations can also be implemented on top of the bidirectional BWT index of Belazzougui and Cunial [2], using the data structures of Belazzougui et al. [3] for the enumeration operations. This gives an index that supports all the required subroutines in *constant time*. The drawback of the bidirectional BWT index is that only randomized construction algorithms are known, but the expected time is still linear in $|T|$. We leave as an open problem the construction of the compacted arc-centric bidirected de Bruijn graph in deterministic linear time independent of the alphabet size.

5 Linear-time minimum SPSS without repetitions

Let I be a set of strings. To compute an SPSS without repetitions we first build a compacted de Bruijn graph G from I. Because of Theorem 6, finding an SPSS is equivalent to finding a walk cover in G. Further, with Lemma 4, we get that an SPSS without repetitions is equivalent to a walk cover that visits each arc exactly once (either once forwards, or once reverse, but not both forwards and reverse). We call such a walk cover a *unique walk cover*.

For minimality, observe that the cumulative length of an SPSS S relates to its equivalent set of walks $\mathcal{W}$ as follows:

$$||S|| = \sum_{W\in\mathcal{W}} (k - 1 + |W|) \tag{1}$$

This is because in Algorithm 2, in Line 7, $k-1$ characters are appended to the result, and then in the loop in Line 8, one additional character per arc in W is appended. We cannot alter the sum $\sum_{W\in\mathcal{W}}|W|$, since we need to cover all arcs in G. However we can alter the number of strings, and decreasing or increasing this number by one will decrease or increase the cumulative length of S by $k-1$. Therefore, finding a minimum SPSS of I without repetitions equals finding a unique walk cover of G that has a minimum number of walks.

Note that computing a minimum SPSS in a bigraph that is not connected is equivalent to separately computing an SPSS in each maximal connected subgraph. Therefore we restrict to connected bigraphs from here on.

5.1 A lower bound for an SPSS without repetitions

Using the imbalance of the nodes of a bigraph, we can derive a lower bound for the number of walks in a walk cover.

▶ **Lemma 9.** *Let $v \in V$ be an unbalanced node in a bigraph $G = (V, E, c)$. Then in a unique walk cover $\mathcal{W}$ of G, either at least $|\operatorname{imbalance}(v)|$ walks start in v, or at least $|\operatorname{imbalance}(v)|$ walks end in v.*

Proof. If v is self-complemental, then its imbalance is 1, so by definition v has an odd number of incident arcs. Each walk that does not start or end in v needs to enter and leave v via two distinct arcs whenever it visits v. But since the number of incident arcs is odd, there is at least one arc that cannot be covered this way, implying that a walk needs to start or end in this arc.

If v is not self-complemental and has a positive imbalance, then it has $\text{imbalance}(v)$ more outgoing arcs then incoming arcs. Since walks need to leave v with the opposite sign than they entered v, at least $\text{imbalance}(v)$ arcs cannot be covered by walks that do not start or end in v. If v has negative imbalance, the situation is symmetric. ◀

▶ **Definition 10** (Imbalance of a bigraph). *The imbalance* $\text{imbalance}(G)$ *of a bigraph* $G = (V, E, c)$ *is the sum of the absolute imbalance of all nodes* $\sum_{v \in V} |\text{imbalance}(v)|$.

▶ **Theorem 11** (Lower bound). *Let G be a bigraph. A walk cover $\mathcal{W}$ of G has a minimum string count of* $\text{imbalance}(G)/2$.

Proof. Let $v \in V$ be an unbalanced node. Then, by Lemma 9 at least $|\text{imbalance}(v)|$ walks start in v or at least $|\text{imbalance}(v)|$ walks end in v. Since each walk has exactly one start node and one end node, $\mathcal{W}$ has a minimum string count of $\text{imbalance}(G)/2$. ◀

5.2 Eulerising a bigraph

Algorithm 3 EULERISE.

Input: Bigraph $G = (V, E, c)$.
Output: Eulerised bigraph $G' = (V, E', c)$.

```
1  G' ← G                                              /* G and G' share V and c */
2  L ← empty list                          /* collect missing incidences to balance G' */
3  foreach v ∈ V do
4      i ← imbalance(v)
5      if c(v) = 1 then
6          if i ≠ 0 then append v⊙ to L
7      else
8          if i > 0 then append i copies of v⊖ to L
9          if i < 0 then append i copies of v⊕ to L
10 while |L| > 0 do                                  /* insert missing incidences as arcs */
11     vd ← remove the first incidence from L
12     v'd' ← remove the first incidence from L
13     insert 1 arc (vd, v'd', |L|) into E'                  /* use distinct identifiers */
```

A directed graph is called *Eulerian*, if all nodes have indegree equal to outdegree, i.e. are balanced [10]. If the graph is strongly connected[3], then this is equivalent to the graph admitting a *Eulerian cycle*, i.e. a cycle that visits each arc exactly once. The same notion can be used with bidirected graphs, using our definition of imbalance.

▶ **Definition 12** (Eulerian bigraph). *A bigraph is* Eulerian*, if all nodes have imbalance zero.*

A connected bigraph can be transformed into a Eulerian bigraph by adding arcs using Algorithm 3. See Figure 1 (b) for an example.

[3] Strongly connected means that there is a directed path from each node v_1 to each node v_2.

▶ **Lemma 13.** *The imbalance of a bigraph is even.*

Proof. Adding or removing an arc changes the imbalance of two nodes by 1, or of one node by two. In both cases, the imbalance of the graph can only change by -2, 0, or 2. Since the imbalance of a graph without arcs is 0, this implies that there can be no graph with odd imbalance. ◀

▶ **Lemma 14.** *Given a connected bigraph $G = (V, E, c)$, Algorithm 3 outputs a Eulerian bigraph $G' = (V, E', c)$.*

Proof. Algorithm 3 is well-defined, since by Lemma 13, it holds that L has even length in each iteration of the loop in Line 10, so the removal operation in Line 12 always has something to remove.

The output of Algorithm 3 is a valid bigraph, since for self-complemental nodes $v \in V$, only incidences $v\odot$ are added to G', and for not self-complemental nodes $v \in V$, only incidences $v\oplus$ and $v\ominus$ are added to G'.

Further, the output is a Eulerian bigraph, because for all $v \in V$, it holds that imbalance(v) is 0, by the following argument:

- If $c(v) = 1$ and v has imbalance zero in G, then its imbalance stays the same in G'. If it has imbalance 1, then one incident arc is inserted, making its degree even and its imbalance therefore zero.
- If $c(v) = 0$ and v has positive imbalance i in G, then i incoming arcs are added to v (counting incoming self-loops twice), and no outgoing arcs are added. Therefore, it has imbalance zero in G'. By symmetry, if v has negative imbalance in G, it has imbalance zero in G'. ◀

▶ **Lemma 15.** *Given a bigraph $G = (V, E, c)$, Algorithm 3 terminates after $O(|V| + |E|)$ steps.*

Proof. For the list data structure we choose a doubly linked list, and for the graph an adjacency list (and array with an entry for each node containing a doubly linked list for the arcs).

The loop in Line 3 runs $|V|$ times and each iteration runs in $O(|\,\text{imbalance}(v)|)$ for a node v, because a doubly linked list supports appending in constant time. The sum of absolute imbalances of all nodes cannot exceed $2|E|$, because each arc adds at most 1 to the absolute imbalance of at most two nodes, or adds at most 2 to the absolute imbalance of at most one node. Therefore, the length of list L after completing the loop is at most $2|E|$, and the loop runs in $O(|V| + |E|)$ time.

The loop in Line 10 runs at most $|L| \leq 2|E|$ times and performs only constant-time operations, since L is a doubly linked list and we can insert arcs into an adjacency list in constant time. Therefore, this loop also runs in $O(|V| + |E|)$ time. ◀

With Lemmas 14 and 15 we get the following.

▶ **Theorem 16.** *Algorithm 3 is correct and runs in $O(|V| + |E|)$ time.*

5.3 Computing a Eulerian cycle in a bigraph

After Eulerising the bigraph, we can compute a Eulerian cycle using Algorithm 4. We do this similarly to Hierholzer's classic algorithm for Eulerian cycles [10]. First we find an arbitrary cycle. Then, as long as there are unused arcs left, we search along the current cycle for unused arcs, and find additional cycles through such unused arcs. We integrate each of those additional cycles into the main cycle. See Figure 1 (c) for an example of a Eulerian cycle.

Algorithm 4 EulerianCycle.

Input: Connected Eulerian bigraph $G = (V, E, c)$.
Output: Eulerian cycle W.

```
1  while |E| > 0 do
2      if |W| = 0 then
3          (vd, v'd', η) ← remove some arc from E
4          W' ← ((vd, v'd', η))                                  /* doubly linked list */
5      else                    /* search a used arc that connects to an unused arc */
6          (vd, v'd', η) ← dereference first_unfinished
7          while E has no arc with incidence v'¬d' do
8              advance first_unfinished to the next arc in W
9              (vd, v'd', η) ← dereference first_unfinished
       // extend W' without repeating arcs until it closes a cycle
10     while E contains an arc e = (v_e d_e, v'_e d'_e, η_e) with incidence v'¬d' do
11         remove e from E
12         if v_e d_e = v'¬d' then (vd, v'd', η) ← (v_e d_e, v'_e d'_e, η_e)
13         else (vd, v'd', η) ← (v'_e d'_e, v_e d_e, η_e)          /* v'_e d'_e = v'¬d' */
14         append (vd, v'd', η) to W'
15     if |W| = 0 then
16         W ← W'
17         first_unfinished ← pointer to the first arc in W
18     else
19         insert W' after first_unfinished in W
20         W' ← ()                                          /* empty doubly linked list */
```

▶ **Lemma 17.** *Given a connected Eulerian bigraph $G = (V, E, c)$, Algorithm 4 terminates and outputs a Eulerian cycle W.*

Proof. For $W = (e_1 = (v_1 d_1, v'_1 d'_1, \eta_1), \ldots, e_\ell = (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ to be a Eulerian cycle, it must be a cycle that contains each arc exactly once.

The sequence W' constructed by the loop in Line 10 is a walk by construction, and since G is Eulerian it is a cycle after the loop terminates. After finding the initial cycle in the first iteration of the outer loop, each additional cycle is started from a node on the initial cycle, and is a cycle again. Therefore it can be inserted into the original cycle without breaking its cycle property.

Since each arc is deleted when being added to W', there is no duplicate arc in W. And if the algorithm terminates, then $|E| = 0$ (Line 1), so W contains all arcs.

For termination, consider that if W is not complete after the first iteration of the outer loop, then the loop in Line 7 searches for an unused arc using the $first\_unfinished$ pointer. Since the prefix of W up to including $first\_unfinished$ is never modified (Line 19), and $first\_unfinished$ is only advanced when its pointee cannot reach any arc anymore, it holds that no arc in W can reach an arc in E when $first\_unfinished$ gets advanced over the end of W. Since G was initially Eulerian and only Eulerian cycles have been removed from G, this implies that all nodes visited by W are still balanced and therefore have no incident arcs anymore. And since G was originally connected, W has visited all nodes, i.e. $|E| = 0$. Therefore, $first\_unfinished$ cannot be advanced over the end of W, because the outer loop terminates before that.

To complete the proof of termination, consider that in each iteration of the outer loop, at least one arc gets removed from E. In the first iteration, this happens at least in Line 3, and in all following iterations, this happens in Line 11. ◀

▶ **Lemma 18.** *Given a connected Eulerian bigraph $G = (V, E, c)$, Algorithm 4 terminates after $O(|V| + |E|)$ steps.*

Proof. We use a doubly linked list for W and W', and an adjacency list for G. Then all lines can be executed in constant time.

The loop in Line 10 removes one arc from E each iteration, so it runs at most $|E|$ times in total (over all iterations of the outer loop). The loop in Line 7 advances $first\_unfinished$ each iteration. Since the algorithm is correct by Lemma 17, $|W| \leq |E|$ and $first\_unfinished$ never runs over the end of $first\_unfinished$, so the loop runs at most $|E|$ times in total (over all iterations of the outer loop).

The condition for the loop in Line 10 is true at least once in each iteration of the outer loop, since the preceding branch sets up $(vd, v'd', \eta)$ such that it has a successor (in the first iteration because of Eulerianess). So in each iteration of the outer loop, at least one arc gets removed, so the outer loop runs at most $|E|$ times in total.

As a result, all loops individually run at most $|E|$ times, therefore Algorithm 4 terminates after $O(|V| + |E|)$ steps. ◀

With Lemmas 17 and 18 we get the following.

▶ **Theorem 19.** *Algorithm 4 is correct and runs in $O(|V| + |E|)$ time.*

5.4 Computing a minimum SPSS without repetitions

We convert the Eulerian cycle into a walk cover of the original bigraph by breaking it at all arcs inserted by Algorithm 3, and removing those arcs (see Figure 1 (d) for an example). This results in a walk cover with either one walk, if Algorithm 3 inserted zero or one arcs, or imbalance$(G)/2$ arcs, if Algorithm 3 inserted more arcs. By Theorem 11, this is a minimum number of walks, and therefore the SPSS spelled by these walks is minimum as well. Constructing the de Bruijn graph takes $O(||I|| \log \Sigma)$ time, and it has $O(||I||)$ k-mers, so it holds that $|V| \in O(||I||)$ and $|E| \in O(||I||)$. Further, spelling the walk cover takes time linear to the cumulative length of the spelled strings. Since we compute a minimum representation, it holds that the output is not larger than the total length of the input strings. Therefore we get:

▶ **Theorem 1.** *Let k be a positive integer and let I be a set of strings of length at least k over some alphabet Σ. Then we can compute a set of strings I' of length at least k with minimum cumulative length and $\mathrm{CS}_k(I) = \mathrm{CS}_k(I')$ in $O(||I|| \log |\Sigma|)$ time.*

6 Experiments

We ran our experiments on a server running Linux with two 64-core AMD EPYC 7H12 processors with 2 logical cores per physical core, 1.96TiB RAM and an SSD. Our data sets are the same as in [23], and we also adapted their metrics *cumulative length* (CL), which is the total count of characters in all strings, and *string count* (SC), which is the number of strings. Our implementation does not use the formalisation of bidirected graphs introduced in this work, but instead uses the formalisation from [23]. For constructing de Bruijn graphs, we do not implement our purely theoretical linear time algorithm, since practical de Bruijn

Table 1 Experiments on references and read sets of single genomes with $k = 51$ and a min abundance of 10 for human and 1 for the others. The CL and SC ratios are compared to the CL-optimal Eulertigs. For time and memory, we report the total time and maximum memory required to compute the tigs from the respective data set. BCALM2 directly computes unitigs, while UST- and Eulertigs require a run of BCALM2 first before they can be computed themselves. Prophasm can only be run for $k \leq 32$, which does not make sense for large genomes. The number in parentheses behind time and memory indicates the slowdown/increase over computing just unitigs with BCALM2. BCALM2 was run with 28 threads, while all other tools support only one thread. The lengths of the genomes are 100Mbp for C. elegans, 482Mbp for B. mori and 3.21Gbp for H. sapiens and the read data sets have a coverage of 64x for C. elegans, 58x for B. mori and 300x for H. sapiens.

genome	algorithm	CL ratio	SC ratio	time [s]		memory [GiB]	
C. elegans (reads)	unitigs	1.789	2.831	1888		5.97	
	UST	1.035	1.080	2738	(1.45)	15.2	(2.54)
	Eulertigs	1	1	3735	(1.98)	25.0	(4.19)
B. mori (reads)	unitigs	1.912	3.136	7737		9.36	
	UST	1.050	1.118	10937	(1.41)	52.4	(5.60)
	Eulertigs	1	1	13793	(1.78)	79.4	(8.48)
H. sapiens (reads)	unitigs	1.418	2.143	56966		13.0	
	UST	1.016	1.044	57736	(1.01)	16.4	(1.26)
	Eulertigs	1	1	58861	(1.03)	29.2	(2.25)
C. elegans	unitigs	1.060	3.154	54.7		1.22	
	UST	1.002	1.089	58.0	(1.06)	1.22	(1.00)
	Eulertigs	1	1	65.9	(1.21)	1.22	(1.00)
B. mori	unitigs	1.262	3.310	224		3.32	
	UST	1.018	1.156	258	(1.16)	3.32	(1.00)
	Eulertigs	1	1	315	(1.41)	3.32	(1.00)
H. sapiens	unitigs	1.195	3.532	3166		10.0	
	UST	1.015	1.192	3369	(1.06)	10.0	(1.00)
	Eulertigs	1	1	3717	(1.17)	10.0	(1.00)

graph construction is a well-researched field [8, 13, 15, 9, 20, 19], and we want to focus more on computing the compressed representation from unitigs. UST only supports unitigs constructed by BCALM2 [8], since it needs certain additional data. BCALM2 is not a linear time algorithm, but works efficient in practice. Therefore, we use BCALM2 to construct a node-centric de Bruijn graph, and then convert it to an arc-centric variant using a hash table.

Our experimental pipeline is constructed with [16] and using the bioconda software repository [11]. We ran all multithreaded tools with up to 28 threads and never used more than 128 cores of our machine at once to prevent hyperthreading from affecting our timing. The code to reproduce our experiments is available at `https://doi.org/10.5281/zenodo.6538261`.

The performance figures are all very similar, with two exceptions. Prophasm does not support parallel computation at the moment, therefore its runtime is much higher. Compared to that, all other algorithms use parallel computation to compute unitigs, but computing

Table 2 Experiments on (references of) pangenomes with $k = 31$ and a min abundance of 1. The CL and SC ratios are compared to the CL-optimal Eulertigs. For time and memory, we report the total time and maximum memory required to compute the tigs from the respective data set. BCALM2 directly computes unitigs, while UST- and Eulertigs require a run of BCALM2 first before they can be computed themselves. Prophasm is run directly on the source data. The number in parentheses behind time and memory indicates the slowdown/increase over computing just unitigs with BCALM2. BCALM2 was run with 28 threads, while all other tools support only one thread. The N. gonorrhoeae pangenome contains 8.36 million unique kmers, the S. pneumoniae pangenome contains 19.3 million unique kmers and the E. coli pangenome contains 341 million unique kmers.

pangenome	tigs	CL ratio	SC ratio	time [s]		memory [MiB]	
1102x N. gonorrhoeae	unitigs	1.615	3.052	24.2		4328	
	UST	1.022	1.074	26.1	(1.08)	4328	(1.00)
	prophasm	1.00004	1.00013	774	(31.9)	208	(0.05)
	Eulertigs	1	1	26.9	(1.11)	4328	(1.00)
616x S. pneumoniae	unitigs	1.679	3.055	21.4		3135	
	UST	1.027	1.081	25.9	(1.21)	3135	(1.00)
	prophasm	1.00004	1.00012	436	(20.3)	434	(0.14)
	Eulertigs	1	1	28.5	(1.33)	3135	(1.00)
3682x E. coli	unitigs	1.705	3.092	334		7146	
	UST	1.031	1.092	416	(1.24)	7146	(1.00)
	prophasm	1.00008	1.00023	7456	(22.3)	7221	(1.01)
	Eulertigs	1	1	471	(1.41)	7146	(1.00)

the final tigs from unitigs seems to be negligible compared to computing the de Bruijn topology. Moreover, running UST or Eulertigs on read data sets of larger genomes consumes significantly more memory than computing just unitigs. This is likely because BCALM2 uses external memory to compute unitigs, while the other tools simply load the whole set of unitigs into memory.

It is notable that the Eulertigs algorithm is always slower than UST. This may be because of the Eulertig algorithm being more complex, but also because our loading and storing routines might not be as efficient. While UST uses node-centric de Bruijn graphs and can therefore directly make use of the topology output by BCALM2 (which is a fasta file with arcs stored as custom annotations), we need to convert the graph into arc-centric format. This is supported by e.g. the B. mori short read data set, on which the computation of Eulertigs uses only 11% of the runtime for the algorithm itself, while 89% are from loading the graph (including the conversion to arc-centric) and storing the result.

In terms of CL, we see that the SPSS computed with UST mostly remains within the expected 3% of the lower bound, but they are up to 5% above the lower bound on more compressible data sets. The SPSS computed by prophasm is very close to the optimum in all cases, and we assume that this difference in quality is because prophasm extends paths both forwards and backwards, while the UST heuristic merely extends them forwards.

Looking at SC, we see that Eulertigs are always the lowest, which is due to the string count directly being connected to the cumulative length by Equation (1). This also explains the correlation between CL and SC, which can be observed in all cases.

7 Conclusions

We have presented a linear and hence optimal algorithm for computing a minimum SPSS without repetitions for a fixed alphabet size. This closes the open question about its complexity raised in [6, 22]. Using our optimal algorithm, we were able to accurately evaluate the existing heuristics and show that they are very close to the optimum in practice. Further, we have published our algorithm as a command-line tool on github, allowing it to easily be used in other projects.

Further, we have presented how bidirected de Bruijn graphs can be formalised without excluding any corner cases. We have also shown how such a graph can be constructed in linear time for a fixed-size alphabet. The construction of the compacted arc-centric bidirected de Bruijn graph in linear time independent of the alphabet size stays an open problem.

References

1 Anton Bankevich, Andrey V Bzikadze, Mikhail Kolmogorov, Dmitry Antipov, and Pavel A Pevzner. Multiplex de bruijn graphs enable genome assembly from long, high-fidelity reads. *Nature biotechnology*, pages 1–7, 2022.

2 Djamal Belazzougui and Fabio Cunial. Fully-functional bidirectional burrows-wheeler indexes and infinite-order de bruijn graphs. In *30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

3 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Versatile succinct representations of the bidirectional burrows-wheeler transform. In *European Symposium on Algorithms*, pages 133–144. Springer, 2013.

4 Djamal Belazzougui, Dmitry Kosolobov, Simon J Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In *32nd Annual Symposium on Combinatorial Pattern Matching*, 2021.

5 Jacek Blazewicz, Alain Hertz, Daniel Kobler, and Dominique de Werra. On some properties of dna graphs. *Discrete Applied Mathematics*, 98(1-2):1–19, 1999.

6 Karel Břinda, Michael Baym, and Gregory Kucherov. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *Genome Biology*, 22(1):1–24, 2021.

7 Bastien Cazaux, Thierry Lecroq, and Eric Rivals. From indexing data structures to de Bruijn graphs. In *Symposium on combinatorial pattern matching*, pages 89–99. Springer, 2014.

8 Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 2016.

9 Victoria G Crawford, Alan Kuhnle, Christina Boucher, Rayan Chikhi, and Travis Gagie. Practical dynamic de bruijn graphs. *Bioinformatics*, 34(24):4189–4195, 2018.

10 Herbert Fleischner. *Eulerian graphs and related topics*. Elsevier, 1990.

11 Björn Grüning, Ryan Dale, Andreas Sjödin, Brad A Chapman, Jillian Rowe, Christopher H Tomkins-Tinch, Renan Valieris, and Johannes Köster. Bioconda: sustainable and comprehensive software distribution for the life sciences. *Nature Methods*, 15(7):475–476, 2018.

12 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.

13 Guillaume Holley and Páll Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome Biology*, 21(1):1–20, 2020.

14 Marta Kasprzak. Classification of de Bruijn-based labeled digraphs. *Discrete Applied Mathematics*, 234:86–92, 2018. Special Issue on the Ninth International Colloquium on Graphs and Optimization (GO IX), 2014. doi:10.1016/j.dam.2016.10.014.

15 Jamshed Khan, Marek Kokot, Sebastian Deorowicz, and Rob Patro. Scalable, ultra-fast, and low-memory construction of compacted de bruijn graphs with cuttlefish 2. *bioRxiv*, 2021.

16 Johannes Köster and Sven Rahmann. Snakemake – A scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2012.

17 Vamsi Kundeti, Sanguthevar Rajasekaran, and Heiu Dinh. An efficient algorithm for chinese postman walk on bi-directed de bruijn graphs. In Weili Wu and Ovidiu Daescu, editors, *Combinatorial Optimization and Applications*, pages 184–196, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

18 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I Tomescu. *Genome-scale algorithm design*. Cambridge University Press, 2015.

19 Martin D Muggli, Bahar Alipanahi, and Christina Boucher. Building large updatable colored de bruijn graphs via merging. *Bioinformatics*, 35(14):i51–i60, 2019.

20 Martin D Muggli, Alexander Bowe, Noelle R Noyes, Paul S Morley, Keith E Belk, Robert Raymond, Travis Gagie, Simon J Puglisi, and Christina Boucher. Succinct colored de bruijn graphs. *Bioinformatics*, 33(20):3181–3187, 2017.

21 Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *bioRxiv*, 2022. doi:10.1101/2022.01.15.476199.

22 Amatur Rahman and Paul Medevedev. Representation of k-mer sets using spectrum-preserving string sets. *Journal of Computational Biology*, 28(4):381–394, 2021.

23 Sebastian Schmidt, Shahbaz Khan, Jarno Alanko, and Alexandru I Tomescu. Matchtigs: minimum plain text representation of kmer sets. *bioRxiv*, 2021.

24 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

A Authors contributions

JNA and SS discovered the problem, SS solved the problem when the de Bruijn graph is given and wrote most of the manuscript, JNA designed the linear-time de Bruijn graph construction algorithm and wrote Section 4. SS implemented the algorithm and conducted and evaluated the experiments. All authors reviewed and approved the final version of the manuscript.

B Omitted proofs

▶ **Lemma 2** (Sound labels). *Let k be a positive integer and let I be a set of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. For all pairs of arcs $e_1 := (v_1 d_1, v_1' d_1', \eta_1), e_2 := (v_2 d_2, v_2' d_2', \eta_2) \in E$ it holds that:*

(a) *$(v_1' = v_2$ and $d_1' = \neg d_2)$ if and only if $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\eta_2)$,*

(b) *$(v_1' = v_2'$ and $d_1' = \neg d_2')$ if and only if $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_2))$,*

(c) *$(v_1 = v_2$ and $d_1 = \neg d_2)$ if and only if $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_1)) = \mathrm{pre}_{k-1}(\eta_2)$, and*

(d) *$(v_1 = v_2'$ and $d_1 = \neg d_2')$ if and only if $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_1)) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_2))$.*

Proof. Observe that the values of w and w' computed in Lines 5 and 7 of Algorithm 1 are equal to $\mathrm{pre}_{k-1}(\eta_1)$ and $\mathrm{suf}_{k-1}(\eta_1)$ for e_1 and equal to $\mathrm{pre}_{k-1}(\eta_2)$ and $\mathrm{suf}_{k-1}(\eta_2)$ for e_2. Further, observe that the values of v and v' computed in Lines 6 and 8 are equal to v_1 and v_1' for e_1 and equal to v_2 and v_2' for e_2. This makes v_1, v_1', v_2 and v_2' the canonicals of $\mathrm{pre}_{k-1}(\eta_1)$, $\mathrm{suf}_{k-1}(\eta_1)$, $\mathrm{pre}_{k-1}(\eta_2)$ and $\mathrm{suf}_{k-1}(\eta_2)$. Finally, observe that the sign values d and d' computed in Lines 9–14 are equal to d_1 and d_1' for e_1 and equal to d_2 and d_2' for e_2.

(a) If $v_1' = v_2$ and $d_1' = \neg d_2$, then $w_1' = w_2$ for all possible values of d_1', and therefore $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\eta_2)$.
If $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\eta_2)$, then $w_1' = w_2$, and therefore $v_1' = v_2$ because v_1' and v_2 are the canonicals of w_1' and w_2. Additionally, $d_1' = \neg d_2$ for all possible values of d_1'.

(b) If $v_1' = v_2'$ and $d_1' = \neg d_2'$, then $w_1' = \mathrm{rc}(w_2')$ for all possible values of d_1', and therefore $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{rc}(\mathrm{suf}_{k-1}(\eta_2)) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_2))$.
If $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_2))$, then $w_1' = \mathrm{rc}(w_2')$, and therefore $v_1' = v_2'$ because v_1' and v_2' are the canonicals of w_1' and w_2'. Additionally, $d_1' = \neg d_2'$ for all possible values of d_1'.

(c) If $v_1 = v_2$ and $d_1 = \neg d_2$, then $\mathrm{rc}(w_1) = w_2$ for all possible values of d_1, and therefore $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_1)) = \mathrm{rc}(\mathrm{pre}_{k-1}(\eta_1)) = \mathrm{pre}_{k-1}(\eta_2)$.
If $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_1)) = \mathrm{pre}_{k-1}(\eta_2)$, then $w_1 = \mathrm{rc}(w_2)$, and therefore $v_1 = v_2$ because v_1 and v_2 are the canonicals of w_1 and w_2. Additionally, $d_1 = \neg d_2$ for all possible values of d_1.
(d) This case is equivalent to the first case when swapping e_1 and e_2, because $\mathrm{suf}_{k-1}(\eta_1) = \mathrm{pre}_{k-1}(\eta_2) \iff \mathrm{suf}_{k-1}(\mathrm{rc}(\eta_2)) = \mathrm{pre}_{k-1}(\mathrm{rc}(\eta_1))$. ◀

▶ **Lemma 3** (Sound sequence of k-mers). *Let k be a positive integer and let I be a set of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. Let $W := (e_1 = (v_1 d_1, v'_1 d'_1, \eta_1), \ldots, e_\ell = (v_\ell d_\ell, v'_\ell d'_\ell, \eta_\ell))$ be a walk in G, and $K := (\kappa_1, \ldots, \kappa_\ell)$ its sequence of k-mers. Then for each consecutive pair of kmers κ_i, κ_{i+1} it holds that* $\mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1})$.

Proof. Let $i \in \{1, \ldots, \ell - 1\}$. By the definition of *walk* it holds that $v'_i = v_{i+1}$ and $d'_i = \neg d_{i+1}$. We can apply Lemma 2 case by case.
(a) If $e_i, e_{i+1} \in E$, then by Lemma 2 a, it holds that $\mathrm{suf}_{k-1}(\eta_i)$ equals $\mathrm{pre}_{k-1}(\eta_{i+1})$. By definition, $\kappa_i = \eta_i$ and $\kappa_{i+1} = \eta_{i+1}$, so $\mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1})$.
(b) If $e_i, e_{i+1}^{-1} \in E$, then by Lemma 2 b applied to e_i, e_{i+1}^{-1}, it holds that $\mathrm{suf}_{k-1}(\eta_i)$ equals $\mathrm{pre}_{k-1}(\mathrm{rc}(\eta_{i+1}))$. By definition, $\kappa_i = \eta_i$ and $\kappa_{i+1} = \mathrm{rc}(\eta_{i+1})$, so $\mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1})$
(c) If $e_i^{-1}, e_{i+1} \in E$, then by Lemma 2 c applied to e_i^{-1}, e_{i+1}, it holds that $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_i))$ equals $\mathrm{pre}_{k-1}(\eta_{i+1})$. By definition, $\kappa_i = \mathrm{rc}(\eta_i)$ and $\kappa_{i+1} = \eta_{i+1}$, so $\mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1})$.
(d) If $e_i^{-1}, e_{i+1}^{-1} \in E$, then by Lemma 2 d applied to e_i^{-1}, e_{i+1}^{-1}, it holds that $\mathrm{suf}_{k-1}(\mathrm{rc}(\eta_i))$ equals $\mathrm{pre}_{k-1}(\mathrm{rc}(\eta_{i+1}))$. By definition, $\kappa_i = \mathrm{rc}(\eta_i)$ and $\kappa_{i+1} = \mathrm{rc}(\eta_{i+1})$, so $\mathrm{suf}_{k-1}(\kappa_i) = \mathrm{pre}_{k-1}(\kappa_{i+1})$. ◀

▶ **Lemma 4** (Sound spell). *Let k be a positive integer and let I be a set of strings of length at least k. Let $G = (V, E, c)$ be the de Bruijn graph of order k constructed from I. Let W be a walk in G, K_W its sequence of k-mers and K'_W the sequence of k-mers of* $\mathrm{spell}(W)$. *Then $K_W = K'_W$.*

Proof. Let $(\kappa_1, \ldots, \kappa_\ell) := K_W$. We use induction over the length of W. For an empty W, K is empty, $\mathrm{spell}(W)$ is empty, and therefore K' is empty as well. For $|W| = 1$, Algorithm 2 outputs $\mathrm{spell}(W) = \kappa_1$ and it holds that $K'_W = (\kappa_1) = K_W$.

For $|W| \geq 2$ we consider that $K_X = K'_X$ holds for a prefix X of W with $|X| = |W| - 1$. When $i = |W|$ at the beginning of the loop in Line 8, then $s = \mathrm{spell}(X)$. By Lemma 3 it holds that the last $k-1$ characters of s are equal to the first $k-1$ characters of κ_ℓ. Therefore, by appending the last character from κ_ℓ to s, κ_ℓ is appended to K'_X forming K'_W. Therefore, last k-mer of K'_W equals the last k-mer of K_W, and the first $\ell - 1$ k-mers of K'_W equal those of K_W by induction. ◀

C Pseudocode for linear-time construction of compacted de Bruijn graphs

The pseudocode for computing a compacted de Bruijn graph in linear time is given by Algorithm 6 which uses Algorithm 5 as a subroutine. The data structure D used by the algorithms is that described in Section 4. Note that if we compute the arc labels as plain strings as in Algorithm 1, we need up to $O(k)$ bits to store a single-k-mer arc. And since arcs are not substrings of input strings (but potentially combinations of input strings), we would

need a string set of up to $O(k||I||)$ characters to store all arc labels without referring to the input strings. This contradicts the algorithm being linear in $||I||$. However, we can store the labels as tuples (p, η, q, r), where $p\eta q$ is the label where p and q are explicit strings while η is a pointer to a k-mer in the input. If r is true, then the label must be reverse complemented to match that defined by Algorithm 1. With this fix, the size of each label is linear in the number of k-mers it represents, and in total the de Bruijn graph represents $O(||I||)$ k-mers.

The comparison on Line 16 of Algorithm 6 can be done in linear time in $|\eta_1| + |\eta_2|$ by finding the suffix array intervals of $\eta_1\eta\eta_2$ and $\mathrm{rc}(\eta_1\eta\eta_2)$ with `extendLeft` and `extendRight` from η and $\mathrm{rc}(\eta)$ respectively, and comparing the starts of the intervals. This way, the total time taken by all those comparisons is proportional to the sum of $|\eta_1| + |\eta_2|$ over all unitigs, which is linear in $||I||$ because each character of η_1 and η_2 can be mapped to a distinct edge in the non-compacted de Bruijn graph of $||I||$. Therefore, the algorithm can be implemented to run in $O(||I||)$ time.

Our pseudocode does not compute the first and last character of each arc-label, but this can be easily computed in constant time using w_i, η_1 and η_2 in Algorithm 6.

Algorithm 5 FindUnitigEnd.

Input: A data structure D, a pair of suffix-intervals $[a_f, b_f], [a_r, b_r]$, an array S_E mapping from suffix-space to boolean, an array S_V mapping from suffix space to nodes, a set of nodes V. Each node in V contains a parameter c.

Output: A node v at the end of the unitig and a sign d, as well as the updated S_E, S_V, V and the label η of the traversed path.

```
1  [a_f, b_f] ← contractLeft(D, [a_f, b_f])
2  [a_r, b_r] ← contractRight(D, [a_r, b_r])
3  η ← ϵ
   // extend over (k − 1)-mers that have indegree and outdegree of 1
4  while |enumerateRight(D, [a_f, b_f]) ∪ rc(enumerateLeft(D, [a_r, b_r]))| =
     |enumerateLeft(D, [a_f, b_f]) ∪ rc(enumerateRight(D, [a_r, b_r]))| = 1 do
5      {σ} ← enumerateRight(D, [a_f, b_f]) ∪ rc(enumerateLeft(D, [a_r, b_r]))
6      η ← ησ
7      [a_f, b_f] ← extendRight(D, [a_f, b_f], σ)
8      [a_r, b_r] ← extendLeft(D, [a_r, b_r], rc(σ))
9      foreach h ∈ [a_f, b_f] ∪ [a_r, b_r] do S_E[h] ← true
10     [a_f, b_f] ← contractLeft(D, [a_f, b_f])
11     [a_r, b_r] ← contractRight(D, [a_r, b_r])
12 if S_V[a_f] = ⊥ then
13     insert node v into V
14     foreach h ∈ [a_f, b_f] ∪ [a_r, b_r] do S_V[h] ← v
15 else v ← S_V[a_f]
16 if a_f = a_r then c(v) ← 1 ; d ← ⊙          /* v self-complemental */
17 else if a_f < a_r then c(v) ← 0 ; d ← ⊖          /* v canonical */
18 else c(v) ← 0 ; d ← ⊕          /* v not canonical */
19 return (v, d, S_E, S_V, V, η)
```

Algorithm 6 LinearCompactedDbg.

Input: An integer k and a set of strings $I = (w_1, \ldots, w_\ell)$ where each string has length at least k.

Output: A de Bruijn graph $G = (V, E, c)$ of order k.

```
1  V ← ∅; E ← ∅                                   // c is stored as parameter of each node
   // $ is a special character outside of the alphabet
2  T ← $w_1$w_2$...$w_ℓ$ rc(w_1)$ rc(w_2)$...$ rc(w_ℓ)$
3  S_V ← array of length |T| filled with ⊥ mapping from suffix space to nodes in V
4  S_E ← array of length |T| filled with false marking used k-mers
5  build data structure D over T                  // See text in Section 4
6  foreach w_i ∈ I do
7      [a_f, b_f] ← find(D, pre_k(w_i))           // Suffix array interval of pre_k(w_i)
8      [a_r, b_r] ← find(D, pre_k(rc(w_i)))       // Suffix array interval of pre_k(rc(w_i))
9      foreach j ∈ (k+1, ..., |w_i|) do
10         if S_E[a_f] = false then               // create arc from unused k-mer
11             foreach h ∈ [a_f, b_f] ∪ [a_r, b_r] do S_E[h] ← true
12             η ← pointer to pre_k(w_i)
               // find unitig start by finding the end on the rev. comp.
13             (v_1, d_1, S_E, S_V, V, η_1) ← FindUnitigEnd(D, [a_r, b_r], [a_f, b_f], S_E, S_V, V)
               // find unitig end
14             (v_2, d_2, S_E, S_V, V, η_2) ← FindUnitigEnd(D, [a_f, b_f], [a_r, b_r], S_E, S_V, V)
               // Reverse because finding the start was done in reverse
15             η_1 ← rc(η_1)
16             if rc(η_1 η η_2) < η_1 η η_2 then   // arc labels are always canonical
17                 swap v_1 and v_2
18                 swap d_1 and d_2
19                 d_1 ← ¬d_1
20                 d_2 ← ¬d_2
21                 r ← true
22             else
23                 r ← false
24             insert e = (v_1 d_1, v_2 d_2, (η_1, η, η_2, r)) into E
25         [a_f, b_f] ← extendRight(D, [a_f, b_f], w_i[j])
26         [a_r, b_r] ← extendLeft(D, [a_r, b_r], rc(w_i)[j])
27         [a_f, b_f] ← contractLeft(D, [a_f, b_f])
28         [a_r, b_r] ← contractRight(D, [a_r, b_r])
```

Predicting Horizontal Gene Transfers with Perfect Transfer Networks

Alitzel López Sánchez
Computer Science Department, Université de Sherbrooke, Canada

Manuel Lafond
Computer Science Department, Université de Sherbrooke, Canada

Abstract

Horizontal gene transfer inference approaches are usually based on gene sequences: parametric methods search for patterns that deviate from a particular genomic signature, while phylogenetic methods use sequences to reconstruct the gene and species trees. However, it is well-known that sequences have difficulty identifying ancient transfers since mutations have enough time to erase all evidence of such events. In this work, we ask whether character-based methods can predict gene transfers. Their advantage over sequences is that homologous genes can have low DNA similarity, but still have retained enough important common motifs that allow them to have common character traits, for instance the same functional or expression profile. A phylogeny that has two separate clades that acquired the same character independently might indicate the presence of a transfer even in the absence of sequence similarity.

We introduce perfect transfer networks, which are phylogenetic networks that can explain the character diversity of a set of taxa. This problem has been studied extensively in the form of ancestral recombination networks, but these only model hybridation events and do not differentiate between direct parents and lateral donors. We focus on tree-based networks, in which edges representing vertical descent are clearly distinguished from those that represent horizontal transmission. Our model is a direct generalization of perfect phylogeny models to such networks. Our goal is to initiate a study on the structural and algorithmic properties of perfect transfer networks. We then show that in polynomial time, one can decide whether a given network is a valid explanation for a set of taxa, and show how, for a given tree, one can add transfer edges to it so that it explains a set of taxa.

2012 ACM Subject Classification Applied computing → Molecular evolution

Keywords and phrases Horizontal gene transfer, tree-based networks, perfect phylogenies, character-based, gene-expression, indirect phylogenetic methods

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.3

Funding *Alitzel López Sánchez*: acknowledges financial support from the programme de bourses d'excellence en recherche from the University of Sherbrooke.
Manuel Lafond: acknowledges financial support from the Natural Sciences and Engineering Research Council (NSERC) and the Fonds de Recherche du Québec Nature et technologies (FRQNT).

Acknowledgements The authors would like to thank the reviewers for their helpful comments and for pointing out paper [37].

1 Introduction

Evolution has historically been seen as a tree-like process in which genetic material is inherited through vertical descent. However, it is now established that co-existing species from most kingdoms of life, if not all, have exchanged genetic material laterally through hybridation or horizontal gene transfer (HGT). The latter is well-known to occur routinely between procaryotes [31, 49], but is believed to have affected eucaryotes as well [30, 25]. HGT is also known to occur between viruses and their hosts [27], between mitochondria and the nucleus [2], and between tumor cells [51].

Since HGTs play a significant role in shaping evolution, several bioinformatics approaches have been developed to identify them. Most of these can be classified as either parametric

or phylogenetic. Parametric methods are based on the sequence of one genome of interest and attempt to find DNA regions that exhibit a signature that is different from the rest of the genome (see [42]). Phylogeny-based methods consist of taking a set of taxa (e.g. genes and/or species), reconstructing their phylogenetic tree, and inferring the unseen transfer locations on the tree. A common way of achieving this is through *reconciliation*, which aims to explain the discrepancies between a gene tree and a species tree by finding where the gene duplication, loss, or transfer events occurred [5, 13, 23, 11]. Finding a most parsimonious reconciliation under this model is NP-hard [50, 32, 28], mainly because the inferred transfer locations need to be *time-consistent*, meaning that they must occur between species that may have co-existed. In addition, recent fundamental approaches propose to identify pairwise gene relationships to infer transfers. For instance, irregularities in the pairwise gene distances can pinpoint to possible transfers [47], or predictions of orthologs, paralogs, and xenologs can help reconstructing a gene tree and a species *network* that explain these relationships [19, 24, 33, 29].

The above approaches are all based on gene sequences in one way or another, either to reconstruct the phylogenies or to infer pairwise relationships. However, it is well-known that sequences have their limits for predicting HGT, especially in the case of ancient transfers [8]. In this work, we ask whether *character-based* approaches can instead be used to predict HGT on a phylogeny. A character is a generic term to denote a trait that a taxa may possess or not, which can be morphological or molecular. A common example of character-based data is gene expression, where a trait corresponds to whether a species expresses a gene or not in a condition of interest [10, 43, 40]. A major advantage of using gene expression profiles, and possibly other character traits, over sequence data comes when highly divergent sequences are involved. In [1], the authors used expression to recover phylogenetic signals better than using only sequence similarity measures. This could be because the necessary information to coordinate the folding or function of proteins is encoded in a small number of conserved fragments, in which case the two homologous proteins can share a small percentage of sequence similarity. This can be leveraged to detect HGTs that are hard to find using sequences, since one could hypothesize that two clades that started expressing the same gene independently could have acquired this behavior by transfer.

The task in this setting is, given a set of characters $\mathcal{C}$ and a set of taxa $\mathcal{S}$ that each possess a subset of $\mathcal{C}$, to explain the diversity of $\mathcal{S}$ in a phylogeny. Ideally, $\mathcal{S}$ can be explained by a tree in which taxa that possess a common character form a clade, in which case the tree is called a *perfect phylogeny* [6, 16, 4, 26]. When no such perfect phylogeny exists, transfers may be required to explain the data. We point out that recently, character-based methods have resurfaced in tumor phylogenetics, where they are used to represent whether a tumor clone has acquired a somatic mutation or not [12, 41, 34, 46].

Before gene expression and other character-based data can be used to predict HGT, appropriate models and algorithmic frameworks need to be devised. To our knowledge, character-based approaches have mostly been used to detect *hybridation* events, where two or more species recombinate to produce an hybrid offspring. In the most popular models, a set of taxa is explained by an *ancestral recombination graph* (ARG), which is an acyclic directed graph in which nodes with multiple parents represent hybrids, and nodes with a single parent represent vertical descent [52, 22, 21]. The task of finding recombination events is different from that of finding HGTs. Recombinations create offsprings whose genetic content is merged from the parents without vertical descent being involved directly. As a result, there is no donor/recipient relationship. In the case of transfers, it is important to distinguish which

traits were acquired vertically from the parent, and which traits were given by a donor.

Another model called perfect phylogenetic networks (PPN) was also introduced in [37, 36] to study the evolution of languages, but can also be used for biological characters. The model asks whether each character is compatible with some tree displayed by a given network. PPNs can be used to infer transfers, but since each character can choose a different parental edge in each reticulation, there is no obvious distinction between vertical descent and horizontal transmission. Also note that PPNs assume that all transfer edges are bidirectional.

In fact, the class of *tree-based networks* has been introduced recently for this purpose [18, 39]. This class of phylogenetic networks captures the idea of having a species tree on which a set of transfer highways were "attached". The *base tree* indicates where vertical descent occurred and the attached transfer edges clearly show where genetic material could have been exchanged.

Our contributions. We introduce *perfect transfer networks* (PTN), which are tree-based networks that can explain how each character was acquired/transferred in a given set of taxa. Our model is a direct generalization of perfect phylogenies to networks, as we use the same set of evolutionary rules. That is, we require that in the network, a character acquired by an ancestral species is never lost by its vertical descendants as in the Camin-Sokal parsimony model [9], and that each character has a unique origin. Additionally, a character can only be transferred horizontally on the edges that are explicitly labeled as transfers. It is worth mentioning that in [3], the authors study an HGT inference framework in which characters that admit a perfect phylogeny are ignored, whereas characters that do not are treated as evidence of transfers. Our work can be seen as an effort to formalize this idea.

We then study the structural and algorithmic aspects of PTNs. We first show that PTNs have two equivalent definitions that are both generalizations of perfect phylogenies. We then distinguish PTNs from recombination networks formally by showing that some taxa sets are explained by different networks depending on the model. As for the algorithmic aspects, we study the question of whether a given tree-based network can explain the characters of a set of taxa and provide a simple, polynomial-time algorithm for the problem. We finally study the tree completion problem where, given a tree, we are asked to add transfers to it so that it explains the input taxa. We show that any tree can explain any set of taxa, even if the characters at the ancestral nodes of the tree are constrained by the input. We then conclude with a discussion on open problems, including the problem of adding a minimum number of transfers to a tree to make it explain a set of taxa.

2 Preliminaries

In this section, we describe the standard phylogenetic notions used in the paper, and then define our perfect transfer network model.

2.1 Phylogenetic networks and tree-based networks

For an integer n, we use the notation $[n] = \{1, \ldots, n\}$. All graphs in this work are directed and loopless. A directed graph G is *connected* if the underlying undirected graph of G is connected. A *binary phylogenetic network*, or simply a *network* for short, is a directed acyclic graph $G = (V, E)$ such that either $|V| = 1$, or such that G satisfies the following conditions:

- there is a set of vertices with in-degree 1 and out-degree 0, called *leaves*.
- there is a unique vertex with in-degree 0, called the *root*.

- every other vertex has either in-degree 1 and out-degree 2 (*tree nodes*), in-degree 2 and out-degree 1 (*reticulation nodes*), or in-degree 1 and out-degree 1 (*subdivision nodes*).

We say that $(u, v) \in E$ is a *tree edge* if v is a tree node or a leaf. Note that the usual definition of a network forbids subdivision nodes. We allow them only because it simplifies some of the definitions and proofs.

For a network G, we write $\rho(G)$ for the root of G and $L(G)$ for the leaves. If $|V(G)| = 1$, then we define $\rho(G)$ as the single vertex of G and consider that $L(G) = \{\rho(G)\}$. If σ is a bijection from $L(G)$ to a set $\mathcal{S}$, we call σ an *$\mathcal{S}$-map for G*, or just an *$\mathcal{S}$-map* if G is understood. We say that $u \in V(G)$ *reaches* a node $v \in V(G)$ if there exists a directed path from u to v in G. We denote by $R_u(G)$ the set of nodes that u reaches in G, and we note that $u \in R_u(G)$. For a subset W of $V(G)$, we denote by $G[W]$ the subgraph of G induced by W. We will also denote by $G - W$ the graph obtained by the removal of W from $V(G)$ and all of its incident edges. In other words, $G - W = G[V(G) \setminus W]$.

A *tree* T is a network whose underlying undirected graph has no cycles. We say that $W \subseteq V(T)$ *forms a subtree of* T if $T[W]$ is a tree. We say that a vertex $v \in V(T)$ is an *ancestor* of $u \in V(T)$ if v is on the path from $\rho(T)$ to u. In this case, we will call u a *descendant* of v. Note that v is an ancestor and descendant of itself. For $v \in V(T)$, we will use $T(v)$ to refer to the subtree of T rooted at v (that is, $T(v)$ contains v and all of its descendants).

A network $G = (V, E)$ is a **tree-based network** [38] if G has no subdivision nodes, and there is a partition $\{E_S, E_T\}$ of E such that the subgraph $\mathcal{T}_G = (V, E_S)$ is a tree with the same set of leaves as G, which is called the *support tree* of G. The edges in E_S are called *support edges* and the edges in E_T are called *transfer edges*. Note that $\mathcal{T}_G$ contains subdivision nodes, unless E_T is empty. The tree obtained from $\mathcal{T}_G$ by suppressing its subdivision nodes is called the *base tree* of G (suppressing a subdivision node u with parent p and child v consists of removing u and adding an edge from p to v). Roughly speaking, a tree-based network G can be obtained by starting with a tree and inserting transfer edges into it.

As mentioned in the introduction, networks should be *biologically-feasible* in terms of time. We define a *time-consistent map* over a tree-based network G with support edges E_S and transfer edges E_T as a function $\tau : V \to \mathbb{R}$ such that:

- for every $(u, v) \in E_S$, $\tau(u) > \tau(v)$.
- for every $(u, v) \in E_T$, $\tau(u) = \tau(v)$.

We say that G is a *time-consistent tree-based network* if there exists a time consistent map for G [17]. Note that the existence of a time-consistent map on a network implies that it is tree-based [35] (but the converse does not necessarily hold). In the following sections, we will assume that all the tree-based networks are time-consistent without explicit mention.

2.2 Perfect transfer networks

We now propose to extend the *perfect phylogeny* model to tree-based networks. Let $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ be a set of taxa and $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$ a set of characters. We view a taxa S_i as the set of characters that it possesses, so that for each $i \in [n]$, S_i is a subset of $\mathcal{C}$. Our goal is to explain the character diversity of $\mathcal{S}$ using its evolutionary history. Given a tree-based network G with $\mathcal{S}$-map σ, we want to know where each character appeared in G under the conditions that each character has a single origin, that it cannot be lost once acquired, and that it can be transferred. Throughout the phylogenetic literature, requiring a single origin is called the *homoplasy-free* assumption (or sometimes the "no parallel evolution" or "no convergent evolution"), which states that characters cannot arise independently in

unrelated lineages [45, 48]. HGT is not considered to be a cause of homoplasy, but of course homoplasy can occur even in the presence of HGT. Nonetheless, this assumption has historically been used as a first step towards more complex models (see e.g. [44]).

To formalize this, given a tree-based network G, a *$\mathcal{C}$-labeling* of G is a function $l : V(G) \to 2^{\mathcal{C}}$ that maps each node of G to the subset of characters that it possesses (here, $2^{\mathcal{C}}$ represents the powerset of $\mathcal{C}$). For a character $c \in \mathcal{C}$, we will denote by $V_c(l) = \{v \in V(G) : c \in l(v)\}$ the set of nodes that possess character c, and we denote by $\overline{V}_c(l) = V(G) \setminus V_c(l)$ the nodes that do not have it. If l is clear from the context, then we may simply write V_c and $\overline{V}_c$.

Our evolutionary requirements are encapsulated in the following definition.

▶ **Definition 1** (Perfect transfer networks). *Let $\mathcal{S}$ be a set of taxa on characters $\mathcal{C}$, let $G = (V, E_S \cup E_T)$ be a tree-based network, and let σ be an $\mathcal{S}$-map for G. We say that a $\mathcal{C}$-labeling l of G* explains *$\mathcal{S}$ if the following conditions hold:*

- *for each $v \in L(G)$, $l(v) = \sigma(v)$;*
- *for each support edge $(u, v) \in E_S$, $c \in l(u)$ implies that $c \in l(v)$ (never lost once acquired);*
- *for each $c \in \mathcal{C}$, there exists a unique node $v \in V_c(l)$ that reaches every node of $V_c(l)$ in $G[V_c(l)]$ (single origin).*

Furthermore, we call the pair (G, σ) a perfect transfer network *(PTN) for $\mathcal{S}$ if there exists a $\mathcal{C}$-labeling of G that explains $\mathcal{S}$.*

See Figure 1.2 for an example of a PTN. Later on in Theorem 2, we will show that Definition 1 is similar to a connectedness requirement known as *convexity* on each character, see [21]. Notice that if G is a tree, then every edge is a support edge and Definition 1 coincides with the definition of a perfect phylogeny from [21]. If $G = (V, E_S \cup E_T)$ is a tree-based network, the definition does not explicitly state what can or cannot be done with transfer edges. The way to see this is that the definition *does not forbid* ancestral taxa from using transfer edges. That is, if $(u, v) \in E_T$, then u can transmit any subset of its characters to v horizontally. We are interested in the following algorithmic problems:

- *The PTN-recognition problem:* given a tree-based network G with $\mathcal{S}$-map σ, is (G, σ) a PTN for $\mathcal{S}$? That is, does there exist a $\mathcal{C}$-labeling of G that explains $\mathcal{S}$? See Figure 1.4 for an example of a network that is not a PTN.
- *The tree-completion problem:* given a tree T with $\mathcal{S}$-map σ, does there exist a PTN (G, σ) for $\mathcal{S}$ such that T is the base tree of G? See Figure 1.2.
- *The tree-minimization problem:* given a tree T with $\mathcal{S}$-map σ, find a PTN (G, σ) for $\mathcal{S}$ such that T is the base tree of G, and such that the number of transfer edges in G is minimized. See Figure 1.3.

We show that the $\mathcal{C}$-labeling problem can be solved in time $O(|\mathcal{C}||V(G)|^2)$. For the tree-completion problem, we provide a more general result: any tree with any given pre-labeling can explain any set of taxa. To be more specific, for any given tree T and any $\mathcal{C}$-labeling of T that satisfies the *never lost once acquired* condition, one can always explain $\mathcal{S}$ by adding transfers in a time-consistent manner while preserving the given labeling. This motivates the need for the minimization problem, which leads to several open problems.

3 Properties of the perfect transfer model

Before delving into the algorithms, we study our model a bit more in-depth. First, we provide an alternate definition of perfect transfer networks in terms of character connectedness. This

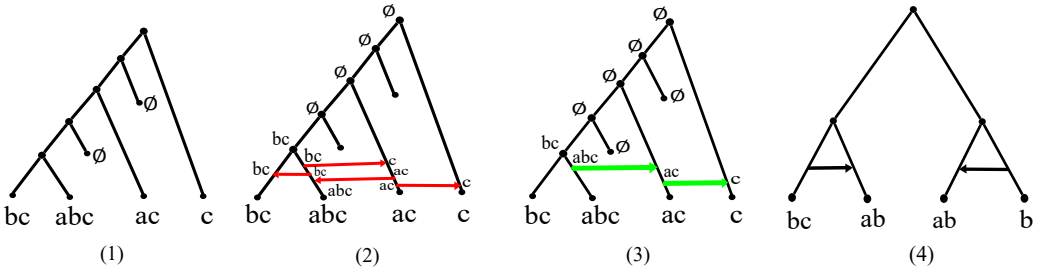


Figure 1 (1) A species tree T with a set of taxa $\mathcal{S}$ on characters $\mathcal{C} = \{a, b, c\}$. (2) A PTN with T as base tree that explains $\mathcal{S}$. Red arrows represent transfer edges. (3) Another PTN with T as base tree that also explains $\mathcal{S}$. Note that this PTN requires two less transfers. (4) A tree-based network G with $\mathcal{S}-$map σ for which no labeling can explain $\mathcal{S}$.

definition is sometimes easier to deal with in our proofs, and is akin to perfect phylogenies that also admit a similar equivalent definition. Second, we ensure that our model does not reinvent the wheel by explicitly stating its differences with the recombination model.

▶ **Theorem 2.** *Let $G = (V, E_S \cup E_T)$ be a tree-based network with $\mathcal{S}$-map σ. Then a $\mathcal{C}$-labeling l of G explains $\mathcal{S}$ if and only if the following conditions hold:*

- *for each $v \in L(G)$, $l(v) = \sigma(v)$;*
- *for each $c \in \mathcal{C}$, $G[V_c(l)]$ is connected and contains a unique node of in-degree 0;*
- *for each $c \in \mathcal{C}$, either $V = V_c(l)$, or $\mathcal{T}_G[\overline{V}_c(l)]$ is connected and contains $\rho(G)$.*

Proof. ($\Rightarrow$) Suppose that l is a $\mathcal{C}$-labeling of G that explains $\mathcal{S}$. We argue that the three conditions stated in the theorem are true. By definition, $l(v) = \sigma(v)$ for each $v \in L(G)$ holds. For the other conditions, let $c \in \mathcal{C}$. Let v be the unique node of $V_c(l)$ that reaches every node in $G[V_c(l)]$, which is guaranteed to exist by Definition 1. Because $G[V_c(l)]$ is acyclic, no node other than v can reach v in $G[V_c(l)]$. This implies that v has in-degree 0 in $G[V_c(l)]$. Moreover, $G[V_c(l)]$ is connected since v reaches all of its nodes. If $\overline{V}_c(l)$ is empty, then the third condition also holds, so assume this is not the case. Let us now focus on $\mathcal{T}_G[\overline{V}_c(l)]$. Observe that because characters are never lost once acquired, l satisfies the property that for every $(u, w) \in E_S$, $w \in \overline{V}_c(l)$ implies that $u \in \overline{V}_c(l)$. This in turn implies that for any $u \in \overline{V}_c(l)$, every ancestor of u in $\mathcal{T}_G$ is in $\overline{V}_c(l)$, including the root $\rho(G)$. Since we assume that $\overline{V}_c(l)$ is non-emtpy, it follows that $\rho(G) \in \overline{V}_c(l)$. It also follows that $\mathcal{T}_G[\overline{V}_c(l)]$ is connected because all of its nodes have a path to $\rho(G)$.

($\Leftarrow$) Suppose that l satisfies all the conditions of the theorem. We show that all properties of Definition 1 hold. For each $v \in L(G)$, we know that $l(v) = \sigma(v)$. For the other conditions, let $c \in \mathcal{C}$. First suppose for contradiction that there is a support edge $(u, v) \in E_S$ such that $u \in V_c(l)$ but $v \in \overline{V}_c(l)$. Thus $\overline{V}_c(l)$ is not empty, in which case $G[\overline{V}_c(l)]$ is connected and contains $\rho(G)$. The path in the support tree $\mathcal{T}_G$ from $\rho(G)$ to v goes through u. But $\rho(G) \in \overline{V}_c(l)$, which is a contradiction since $\mathcal{T}_G[\overline{V}_c(l)]$ is connected, but here u disconnects $\rho(G)$ from v in $\mathcal{T}_G$. Thus the condition of never losing acquired characters holds. Now let v be the unique node of $V_c(l)$ of in-degree 0 in $G[V_c(l)]$. Assume that there is some $u \in V_c(l)$ that v does not reach in $G[V_c(l)]$. Let P_u be the set of nodes of $V_c(l)$ that reach u in $G[V_c(l)]$. Because $G[V_c(l)]$ is acyclic, P_u must contain a node w of in-degree 0, contradicting that v is the unique node of in-degree 0. Thus v reaches every node in $G[V_c(l)]$ and, because it has in-degree 0, it is the unique such node. Thus the single origin condition is satisfied. ◀

Perfect transfer networks versus recombination networks

Before we move on, it is important to put our model in perspective with recombination networks which, to our knowledge, is the closest to our work. In this model, the indices of the characters $\mathcal{C} = \{c_1, \ldots, c_m\}$ determine an ordering of the characters. For a string B, $B[j]$ denotes its j-th character and $B[i..j]$ its substrings containing positions from i to j. Each taxa $S_i \subseteq \mathcal{C}$ can be represented as an m-bit string $\beta(S_i)$ in which $\beta(S_i)[j] = 1$ if and only if S_i possesses character c_j. Given an m-bit string B and an odd integer d, we say that B is a *d-crossover* of two other m-bit strings X and Y if there are indices $i_1, \ldots, i_d$ such that $B = X[1..i_1]Y[i_1 + 1..i_2]X[i_2 + 1..i_3] \ldots Y[i_d + 1..n]$. If d is even, the definition of a d-crossover is the same except that the last substring is $X[i_d + 1..n]$. For a network G and $\mathcal{S}$-map σ, a *binary $\mathcal{C}$-labeling* of G is a function f in which $f(v)$ is an m-bit binary string for each $v \in V(G)$, such that $f(\rho(G))$ only contains 0s. A binary $\mathcal{C}$-labeling f of G *explains* $\mathcal{S}$ *with d-crossovers* if $f(v) = \beta(\sigma(v))$ for each $v \in L(G)$, and the following holds:

- for each reticulation node v with parents u and w, $f(v)$ is a d-crossover of $f(u)$ and $f(w)$;
- for each tree edge (u, v), $f(v)$ is obtained from $f(u)$ by flipping some 0 positions to 1;
- for each $i \in [m]$, there is at most one tree edge (u, v) with $f(u)[i] = 0$ but $f(v)[i] = 1$.

We say that (G, σ) is an *ancestral recombination graph* (ARG) for $\mathcal{S}$ with d-crossovers if there exists a binary $\mathcal{C}$-labeling of G that explains $\mathcal{S}$ with d-crossovers. We denote by $ARG_d(\mathcal{S})$ the set of all ARGs for $\mathcal{S}$ with d-crossovers. We also denote $ARG_\infty(\mathcal{S}) = \bigcup_{d=1}^{\infty} ARG_d(\mathcal{S})$.

In the literature, single crossovers are modeled with $d = 1$ and have been studied extensively. The $d = 2$ case is often referred to as double crossovers and can also model gene conversion. It was stated in [21] that $d > 6$ is rarely considered in practice.

The most obvious difference between ARGs and PTNs is that ARGs were introduced to model hybridation events, whereas we created PTNs to model horizontal gene transfer. More specifically, ARGs do not differentiate between the two parents of a reticulation node, whereas in our model, one parent only transmits genetic content via vertical descent whereas the other does so via HGT. In fact, ARGs do not need to be tree-based networks. Although this is a fundamental difference, it is also interesting to ask whether, among the class of tree-based networks, the data explanation depends on the model.

To formalize this, we write $PTN(\mathcal{S})$ for the set of perfect transfer networks for $\mathcal{S}$. The next result shows that ARGs with d-crossovers are incomparable with PTNs unless we allow an arbitrary number of crossovers. We emphasize that even though infinite crossovers can emulate transfers, they still cannot distinguish vertical from horizontal inheritance.

▶ **Proposition 3.** *The following relationships between PTNs and ARGs hold:*

- *for any fixed $d \geq 1$, there exists a set of taxa $\mathcal{S}$ on $d + 2$ characters such that $PTN(\mathcal{S}) \setminus ARG_d(\mathcal{S})$ is non-empty;*
- *there exists a set of taxa $\mathcal{S}$ on two characters such that $ARG_1(\mathcal{S}) \setminus PTN(\mathcal{S})$ is non-empty;*
- *for any set of taxa $\mathcal{S}$, $PTN(\mathcal{S}) \subseteq ARG_\infty(\mathcal{S})$.*

The proof can be found in the Appendix. For (1) we show in Figure 2 an example of a network G such that for any set of characters $\mathcal{C}$, (G, σ) is in $PTN(\mathcal{S})$ but not in $ARG_d(\mathcal{S})$. Equivalently, for (2) in Figure 2 we present a network G that can be explained by using single crossovers but for which there is no $\mathcal{C}-$labeling that explains $\mathcal{S}$.

Perfect transfer networks versus perfect phylogenetic networks

The model that is the closest to ours is Perfect Phylogenetic Networks (PPN), as stated in [36, Definition 12.1.2]. The idea is that a network contains several evolutionary trees, and

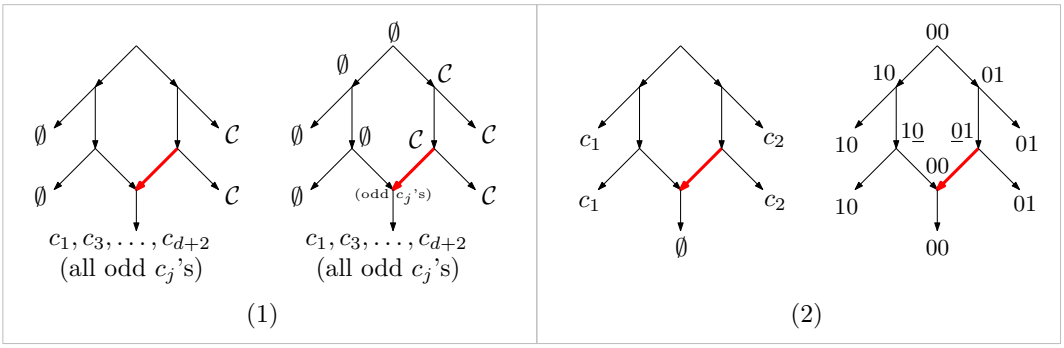


Figure 2 (1) Left: a network G with 5 taxa at the leaves. The fat red edge is a transfer edge. Two taxa have no character, two have all characters of $\mathcal{C}$, and one has only the odd numbered taxa (we assume odd d in the figure). Right: a $\mathcal{C}$-labeling that explains G (the reticulation receives the odd characters from the transfer edge). This network has no explanation with d-crossovers. (2) Left: a network G with 5 taxa on character set $\mathcal{C} = \{c_1, c_2\}$. Right: a binary $\mathcal{C}$-labeling with single crossovers that explains G. This network is not a PTN.

a character could evolve in any one of those trees. More specifically, given a network G and a tree T, we say that T is *displayed* by G if T can be obtained by successively removing reticulation edges and suppressing subdivision nodes. A network G is a PPN for a set of characters $\mathcal{C}$ if, for each $c \in \mathcal{C}$, G displays a tree T that can be labeled so that the nodes that contain c form a connected subtree of T, as well as the nodes that do not contain c.

Figure 3 shows that PPNs can be different from PTNs. The main reason is that PPNs consider all displayed trees as equal, whereas PTNs have a clearly defined base tree.

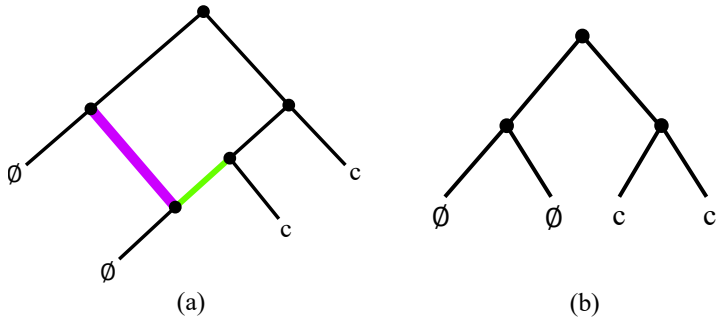


Figure 3 (a) A PPN network G with one character c where the two colored edges enter a reticulation node. (b) A tree displayed by G that admits a labeling such that every node that contains c, or not, forms a connected component. Note that this tree can be obtained by the removal of the green edge and suppression of the resulting subdivision node. As for PTNs, assume that the fat purple edge is a transfer edge. Then a solution for PTNs would not consider the removal of the green edge, in which case no $\mathcal{C}$-labeling is possible.

Also note that it is NP-hard to decide whether a given network G is a PPN for a given set of characters $\mathcal{C}$ [36]. Later on, we will show that the analogous recognition problem for PTNs can be solved in polynomial time. However, our result holds for binary character states and the hardness proof for PPNs relies on the character states being non-binary, and it is unclear whether the hardness is preserved for binary character states.

4 Algorithmic Problems

We first study the problem of deciding whether a given network with known transfers can explain a set of taxa $\mathcal{S}$. We then study the tree completion problem, where we must add the transfers to explain the taxa.

4.1 The PTN recognition problem

The first problem in this section is analogous to the perfect phylogeny problem, where we must find a labeling of all the inner nodes so that the network correctly represents the evolution of a given set of species. This is not as trivial as in the tree case, since there may be multiple options for the originator of a character $c \in \mathcal{C}$.

The PTN recognition problem.
Input. A set of taxa $\mathcal{S}$ on characters $\mathcal{C}$ and a tree-based network $G = (V, E_S \cup E_T)$ with $\mathcal{S}$-map σ.
Output. A $\mathcal{C}$-labeling of G that explains $\mathcal{S}$, if one exists.

Importantly, the above problem formulation lets us assume that we have knowledge of support and transfer edges. We first need some intermediate results.

▶ **Lemma 4.** *Let $G = (V, E_S \cup E_T)$ be a tree-based network with $\mathcal{S}$-map σ, and let l be a $\mathcal{C}$-labeling of G that explains $\mathcal{S}$. Let $u \in \overline{V}_c(l)$. Then for each ancestor v of u in $\mathcal{T}_G$, we have $v \in \overline{V_c}(l)$ as well.*

Proof. Suppose that $u \in \overline{V}_c(l)$ has an ancestor v in $\mathcal{T}_G$ such that but $v \in V_c(l)$. Consider the unique path of $\mathcal{T}_G$ from v to u, namely the sequence $(v, v_1, v_2, \ldots, v_k, u)$. Due to the *never lost once acquired* requirement of Definition 1 $\{v_1, v_2, \ldots, v_k\} \subseteq V_c(l)$. In this way the edge (v_k, u) will represent the loss of character c, a contradiction. ◀

In other words, any node that has at least one descendant in $\overline{V_c}(l)$ should be in $\overline{V_c}(l)$ in every possible $\mathcal{C}-$labeling l that explains $\mathcal{S}$. Since we must have $l(v) = \sigma(v)$ for each leaf v, we can already deduce that if v is a leaf such that $c \notin \sigma(v)$, then no ancestor of v in the support tree can possess c. This is a property similar to character state changes in the Camin-Sokal parsimony method [15]. We thus define the following subset of nodes that are forced to not have c:

$$F_c(G, \sigma) = \{v \in V(G) : \exists w \in L(\mathcal{T}_G(v)) \text{ such that } c \notin \sigma(w)\ \}$$

If G and σ are clear from the context, we will write F_c instead of $F_c(G, \sigma)$. Recall that for a network G, $R_v(G)$ denotes the set of nodes reachable from v. The following characterization of PTNs will let us recognize them easily.

▶ **Lemma 5.** *Let G be a tree-based network with $\mathcal{S}$-map σ. Then (G, σ) is a perfect transfer network for $\mathcal{S}$ if and only if for every character $c \in \mathcal{C}$, $G - F_c$ contains a node v such that $R_v(G - F_c)$ contains every leaf in $L(G - F_c)$.*

Proof. ($\Rightarrow$) Let l be a $\mathcal{C}-$labeling of G that explains $\mathcal{S}$, and let $c \in \mathcal{C}$. By Lemma 4, we must have $F_c \subseteq \overline{V}_c(l)$. The resulting graph $G - F_c$ thus contains all the nodes in $V_c(l)$. Due to the single origin requirement of our definition, $G - F_c$ contains a node v that is able to reach all the leaves in $V_c(l)$, which includes all the leaves of $G - F_c$.

($\Leftarrow$) To show the converse, we build a $\mathcal{C}-$labeling l in the following way: for every $c \in \mathcal{C}$, let v be a node such that its reachable set in $G - F_c$ contains every leaf in $L(G - F_c)$, and let us call v the *origin* of c. Then for $u \in V(G)$, we put $c \in l(u)$ if and only if $u \in R_v(G - F_c)$. That is, the origin of c is v and it is transmitted to every node that v reaches in $G - F_c$. We claim that l satisfies all the conditions of Theorem 2.

Let $u \in L(G)$ and let us first argue that $l(u) = \sigma(u)$. Let $c \in \mathcal{C}$ and let v be the chosen origin of c in l. If $c \notin \sigma(u)$, then $u \in F_c$ and v does not reach u in $G - F_c$ (because u is simply not in $G - F_c$), and by our construction, $c \notin l(u)$. If $c \in \sigma(u)$, then $u \notin F_c$ and v reaches u in $G - F_c$, in which case we put $c \in l(u)$. It follows that $l(u) = \sigma(u)$. We now argue on the connectedness requirements of the $G[V_c(l)]$ subgraphs. Again, let $c \in \mathcal{C}$ and let v be the chosen origin of c in l. Since $V_c(l)$ consists of nodes reachable from v, it is evident that v will be the only node in $G[V_c(l)]$ whose in-degree is 0. It is also evident that $(G - F_c)[V_c(l)]$ is connected, since $V_c(l)$ only contains nodes reachable from v in $G - F_c$. This implies that $G[V_c(l)]$ is also connected, since we cannot disconnect the $V_c(l)$ subgraph by putting back the nodes of F_c into $G - F_c$.

It remains to argue the third condition of Theorem 2. First assume that the origin v of c is equal to $\rho(G)$. Then every leaf must possess c, as otherwise if some leaf u would satisfy $c \notin \sigma(c)$, then $u \in F_c$ and the root would also be in F_c, by its definition. It follows that F_c is empty. Then our construction puts $V_c(l) = V(G)$ since $\rho(G)$ reaches every node in $G - F_c = G$. In this case, the third condition of the theorem holds. So we may assume that $v \neq \rho(G)$. In this case, v does not reach $\rho(G)$ in $G - F_c$ since no node reaches the root (except the root itself). Thus we may assume that $\overline{V}_c(l) \neq \emptyset$ and contains the root. Suppose for a contradiction that $\mathcal{T}_G[\overline{V_c}(l)]$ is disconnected, i.e. there exist some node $u \in \overline{V}_c(l)$ that $\rho(G)$ cannot reach in $G[\overline{V}_c(l)]$. Then in $\mathcal{T}_G$, u has an ancestor $w \in V_c(l)$, which implies that the origin, v, can reach w in $G - F_c$. This implies that $w \notin F_c$, which in turn implies that no node on the from from w to u in $\mathcal{T}_G$ can be in F_c. This means that the path from w to u exists in $G - F_c$. Thus in $G - F_c$, v reaches u through w, and so we would have put $c \in l(u)$, a contradiction. Hence, $\mathcal{T}_G[\overline{V_c}(l)]$ is connected and contains $\rho(G)$. ◀

The proof of the previous lemma implies the following verification algorithm:

▶ **Theorem 6.** *Algorithm 1 correctly solves the* PTN recognition *problem in time* $O(|\mathcal{C}||V(G)|^2)$.

Proof. We first argue that the algorithm is correct. By Lemma 5, it suffices to check, for every $c \in \mathcal{C}$, that some node v reaches every leaf in $G - F_c$. Moreover, when this is the case, the proof of Lemma 5 shows that we can add c to every node in $R_v(G - F_c)$ to obtain a labeling that explains $\mathcal{S}$. If the algorithm finds such a v, this is exactly the labeling that it applies. So we must argue that when such a v exists, the algorithm will find it.

For that, we claim that the verification in line 12 of Algorithm 1 is enough to find the node required by Lemma 5. That is, we do not need to check every node v of $\widehat{G_c} = G - F_c$, but only those in the set X, i.e. the nodes of in-degree 0. Suppose that there is $y \in V(\widehat{G_c})$ that reaches every leaf in $L(\widehat{G_c})$. If $y \in X$, we are done. Otherwise, because $\widehat{G_c}$ is acyclic, there must be $v \in X$ that reaches y (it can be found by following by iteratively following in-neighbors starting from y until such a node is reached). It follows that $R_y(\widehat{G_c}) \subseteq R_v(\widehat{G_c})$ and that v also reaches every leaf. Thus, it suffices to check every source in $\widehat{G_c}$.

Let us now argue the complexity. For a character $c \in \mathcal{C}$, computing F_c can be done in a post-order traversal of $\mathcal{T}_G$ in time $O(|V(G)|)$. Checking whether the resulting network is connected or not can be done in linear time too. Finding the nodes of in-degree 0 takes linear-time and, for each $O(|V(G)|)$ of these nodes, computing $R_v(\widehat{G_c})$ takes time $O(|V(G)|)$. Thus each character requires time $O(|V(G)|^2)$, and thus our algorithm solves the problem in $O(|\mathcal{C}||V(G)|^2)$. ◀

Algorithm 1 Check if a given tree-based network G explains $\mathcal{S}$.

function *findLabeling*$(G, \sigma, \mathcal{S})$
  Set $l(v) = \sigma(v)$ for every leaf $v \in L(G)$.
  Set $l(v) = \emptyset$ for all $v \in V(G) \setminus L(G)$.
  while l *not NULL and* $\mathcal{C}$ *is not empty* **do**
    Pick a character $c \in \mathcal{C}$.
    Compute $F_c(G, \sigma)$.
    Obtain the pruned network $\widehat{G_c} = G - F_c(G, \sigma)$
    if $\widehat{G_c}$ *is not connected* **then**
      Set $l =$ NULL.
    else
      Let X be the set of all nodes in $\widehat{G_c}$ with in-degree 0.
      if *there is* $v \in X$ *such that* $L(\widehat{G_c}) \subseteq R_v(\widehat{G_c})$ **then**
        Set $l(u) = l(u) \cup \{c\}$ for all u in $R_v(\widehat{G_c})$.
      else
        Set $l =$ NULL.
    Remove c from $\mathcal{C}$
  return l.

4.2 The tree-completion problem

We now turn to the problem of predicting transfer locations on a given species tree. As mentioned in [38], species trees depict vertical inheritance and thus serve as basis for our networks, and HGT events can be seen as additional evolutionary events that happen on the way. This motivates the feasibility question: given a set of taxa $\mathcal{S}$ and a species tree T on $\mathcal{S}$, can we add transfer arcs to T so that the resulting network explains $\mathcal{S}$? Moreover, can we do this so that the resulting network is time-consistent? One possibility is that a *universal tree-based network*, which contains all phylogenetic trees on a given set of n leaves [7], could explain any set of characters. However, a base tree needs to be specified in our model, and it is not clear that such a choice always exists in a universal network.

Nevertheless, it turns out that there is no feasibility problem. Indeed, it is relatively easy to show that, for a set of taxa $\mathcal{S}$, any tree T on $\mathcal{S}$ can be complemented with transfers to become a PTN. One way to achieve this is to add a transfer edge between the parent edge of every pair of leaves, from which point it can be argued that the network is a PTN. In fact, we can show a stronger statement: if T is "pre-labeled" in any way that acquired characters are never lost, then we can add transfers to T to explain $\mathcal{S}$ while preserving the given pre-labeling.

Formally, for a tree T, a $\mathcal{C}$-labeling λ of T is called a *no-loss $\mathcal{C}$-labeling* if, for any edge $(u, v) \in E(T)$ and any $c \in \mathcal{C}$, it holds that $c \in l(u)$ implies $c \in l(v)$. Recall that if G is a tree-based network, then the base tree T of G is obtained by removing transfer arcs and suppressing subdivision nodes. Hence, $V(T) \subseteq V(G)$, and we use $V(G) \cap V(T)$ to explicitly refer to the nodes of G that are also in the base tree. We have the following problem.

The Pre-labeled Tree Completion problem.

Input. A set of taxa $\mathcal{S}$ on characters $\mathcal{C}$, a tree T with $\mathcal{S}$-map σ, and a no-loss $\mathcal{C}-$labeling λ of T such that $\lambda(v) = \sigma(v)$ for each $v \in L(T)$.

Output. A time-consistent tree-based network G such that T is the base tree of G, and a $\mathcal{C}$-labeling l of G that explains $\mathcal{S}$ and such that $l(v) = \lambda(v)$ for every $v \in V(G) \cap V(T)$.

We will now prove that this is always possible, even with the time-consistency constraint. One interest of allowing a pre-labeling is that one can start with any hypothesis on where the characters appeared on a tree, and transfers can explain that hypothesis. Perhaps the most natural pre-labeling is the Fitch-like labeling where, for each character c, we add c in every maximal subtree whose leaves have the character. To be more specific, if v is a leaf, $\lambda(v) = \sigma(v)$ and otherwise, let u, w be the children of v, then $\lambda(v) = \lambda(u) \cap \lambda(w)$. This corresponds to the reasonable hypothesis that characters always appear at maximal clades that have it, and we provide and algorithm that can explain this. Again, this is one example of a possible pre-labeling, and our algorithm can explain any other that has the no-loss property, even if characters are again after the lowest common ancestor of the leaves that have this character.

We define a transfer operation between two nodes u and v on a tree-based network G. We write $G▲(u, v)$ to denote the tree-based network obtained by subdividing the respective incoming edges of u and v in $\mathcal{T}_G$, thereby creating new parent u' for u and v' for v, and adding the transfer edge (u', v').

It is important to point out that in a no-loss labeling, there can be multiple ancestral species that acquire a specific character that for the first time. In our algorithm, this property will also be maintained in the constructed network, and we will use the following notion:

▶ **Definition 7.** *Let G be a tree-based network with a no-loss $\mathcal{C}-$labeling λ. Let (u, v) be an edge of $\mathcal{T}_G$. We will say that v is a **first-appearance node for** c if it holds that $u \in \overline{V_c}(l)$ and every descendant of v in $\mathcal{T}_G$ belongs to $V_c(l)$.*

Note that if λ is a no-loss labeling, a first-appearance node for c cannot have a first-appearance node for c as a descendant. We can now describe our algorithm. The first step is to make G a copy of the given tree T, and then we add transfer edges to G. Note that in our problem, the given tree has no time map, and deciding where to put the transfers on T in a time-consistent manner becomes surprisingly complicated. For this reason, before adding any transfer, we start by constructing a time consistent map τ for G. It is easy to do this in such a way that no two vertices have the same time.

From that point, we look at each character c and their set of first-appearance nodes $a_1, \ldots, a_k$, ordered in decreasing order of age, and we greedily connect them using transfers. The τ that we constructed dictates the order of connections, in the sense that each a_i is assumed to transfer c to the younger a_{i+1} node. This is achieved by finding a descendant of a_i that could have co-existed between a_{i+1} and its parent. The τ map is also used to assign a time to the nodes created by transfers.

▶ **Lemma 8.** *The network returned by Algorithm 2 is time-consistent under the time map τ.*

▶ **Lemma 9.** *Let G and l be the network and $\mathcal{C}$-labeling returned by Algorithm 2, respectively, on input $T, \mathcal{S}$, and λ. Then T is the base tree of G. Furthermore, l explains $\mathcal{S}$ and satisfies $l(v) = \lambda(v)$ for every $v \in V(G) \cap V(T)$.*

The proofs of lemma 8 and lemma 9 can be found in the Appendix.

▶ **Theorem 10.** *Algorithm 2 solves the* Pre-labeled Tree Completion *problem correctly in time $O(|\mathcal{C}|^2|\mathcal{S}| + |\mathcal{C}||\mathcal{S}|^2)$.*

Proof. By Lemma 8, the network G output by the algorithm has T as its base tree and is time-consistent. Then by Lemma 9, the output labeling l preserves the pre-labeling λ and explains $\mathcal{S}$. Thus, the output is correct.

Algorithm 2 Place an edge between all the first appearance trees.

function *TransferAdditionGreedy*(T, S, λ)
  Let $G = (V, E_S \cup E_T)$ be the tree-based network such that $V(G) = V(T)$, $E_s = E(T)$ and $E_T = \emptyset$.
  Let l be the $\mathcal{C}$-labeling in which $l(v) = \lambda(v)$ for all $v \in V(G)$.
  Let τ be a time-consistent map for G in which every node has a distinct time.
  for $c \in \mathcal{C}$ **do**
    Compute A_c, the set of first appearance nodes of character c in $\mathcal{T}_G$.
    Let $X_c = (a_1, a_2, \ldots, a_k)$ be the ordering of A_c such that $\tau(a_i) \geq \tau(a_{i+1})$ for all $i \in [k-1]$.
    for $i \in [k-1]$ **do**
      Let a'_i be the parent of a_i in $\mathcal{T}_G$
      Let a'_{i+1} be the parent of a_{i+1} in $\mathcal{T}_G$.
      if *a_i has no descendant w in $\mathcal{T}_G$ such that* $(w, a'_{i+1}) \in E_T$ **then**
        Look for $(w', w) \in E(\mathcal{T}_G(a'_i))$ such that $\tau(w') > \tau(a_{i+1})$ and $\tau(w) \leq \tau(a_{i+1})$
        Apply the transformation $G\blacktriangle(w, a_{i+1})$.
        Let $\hat{w}$ and $\hat{a}_{i+1}$ be the new parents of w and a_{i+1}, respectively, in $\mathcal{T}_G$
        Set $l(\hat{w}) = (l(w) \cap l(w')) \cup \{c\}$ and $(l(\hat{a}_{i+1}) = l(a'_{i+1}) \cap l(a_{i+1})) \cup \{c\}$.
        Set $\tau(\hat{w}) = \tau(\hat{a}_{i+1}) = \dfrac{\min(\tau(w'), \tau(a'_{i+1})) + \tau(a_{i+1})}{2}$
  return(G, l)

As for the running time, the complexity is dominated by the main *while* loop over $c \in \mathcal{C}$. Note that first appearance nodes partition the leaves of $G[V_c(l)]$, and so each A_c has at most $|\mathcal{S}|$ elements and, for each c we add at most $|\mathcal{S}|$ transfers. It follows that the final network G output by the algorithm has at most $O(|\mathcal{C}||\mathcal{S}| + |V(T)|)$ nodes. Let us denote $n = |V(G)| \leq |\mathcal{C}||\mathcal{S}| + |V(T)|$. For a given $c \in \mathcal{C}$, computing the first appearance nodes can be done in time $O(n)$ and sorting them takes time $O(n \log n)$. Then for each of the $O(n)$ nodes in X_c, we must find a descendant w in time $O(n)$. The other operations take constant time. Thus, each iteration of the main loop takes time $O(n \log n + n^2) = O(n^2)$. This is repeated $O(|\mathcal{C}|)$ times, and thus the complexity is $O(|\mathcal{C}|n^2) = O(|\mathcal{C}| \cdot (|\mathcal{C}||\mathcal{S}| + |V(T)|))$. The claimed complexity follows from $|V(T)| \in O(|\mathcal{S}|)$ since T is a tree on leafset $\mathcal{S}$. ◀

4.3 On minimizing the number of transfers

We have shown that any tree whose leaves are mapped to $\mathcal{S}$ can explain $\mathcal{S}$. This means that if only $\mathcal{S}$ is known, the question of whether there exists a network that explains $\mathcal{S}$ is uninteresting: it suffices to take any tree on leaf set $\mathcal{S}$ and run Algorithm 2.

This motivates two optimization formulations:

1. The Minimum PTN problem: given a set of taxa $\mathcal{S}$ on character set $\mathcal{C}$, find a PTN for $\mathcal{S}$ with a minimum number of transfers;
2. The Minimum Perfect Transfer Completion problem: given a tree T with $\mathcal{S}$-map σ, add a minimum number of transfers to T so that it becomes a PTN for $\mathcal{S}$.

We leave the complexity status of these as open problems. However, it is interesting to put Algorithm 2 in perspective with the Minimum Perfect Transfer Completion problem. The most natural pre-labeling λ to use is the one where, for each $c \in \mathcal{C}$, we put $c \in \lambda(v)$ for each

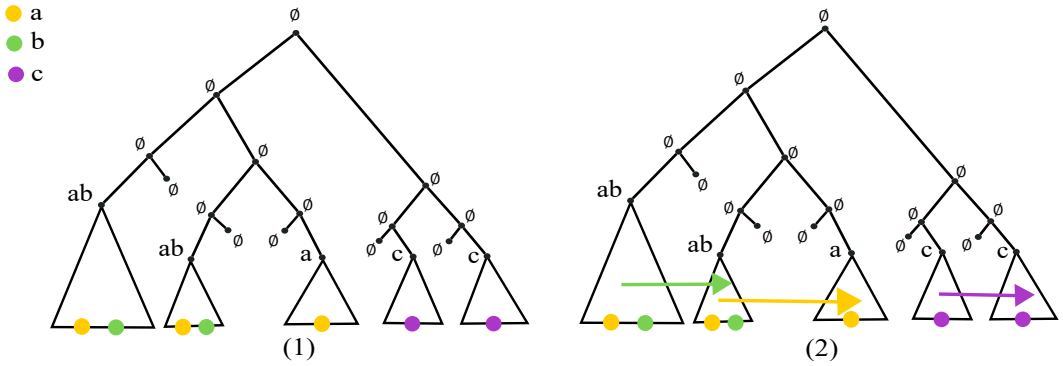


Figure 4 An example of a solution that will be given by our greedy algorithm (1) The given instance T with $\mathcal{C} = \{a, b, c\}$. Triangles represent subtrees and colors represent the characters that can be found on them (2) One possible solution output by Algorithm 2. The green arrow joins the first two subtrees that contain $\{a, b\}$, the yellow arrows joins the second subtree with the subtree that contains only $\{a\}$ and the final arrow connects two subtrees that contain only $\{c\}$.

node whose descending leaves *all* have c. Let us call this the Fitch-labeling. Note that for a minimization problem, an α-approximation is an algorithm whose solution is always at most α times the optimal.

▶ **Proposition 11.** *Suppose that Algorithm 2 is given $T, \mathcal{S}$, and the Fitch-labeling λ. Then Algorithm 2 does not always output an optimal solution to the* Minimum Perfect Transfer Completion *problem. However, it is a $|\mathcal{C}|$-approximation.*

Proof. To see that Algorithm 2 can be suboptimal, consider Figure 5 and the explanation in the caption.

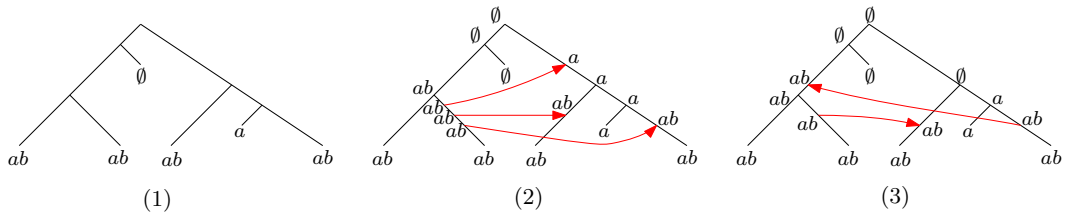


Figure 5 The greedy Algorithm 2 can add more transfers than the minimum. (1) A network G whose leaves are on two characters a and b. (2) One possible solution output by Algorithm 2. The left clade is chosen to originate a and a transfer is greedily added above all the a's on the right side. Then, the left clade is also chosen to originate b, but for that two more separate transfers are added. (3) A solution that is somewhat less natural, but with one less transfer.

Let us argue that it is a $\mathcal{C}$-approximation and let OPT denote the number of transfers added by an optimal solution. Let c be such that in the input T, the number of first-appearance nodes in A_c for c under λ is maximum. Notice that all nodes of T that do not descend from a first-appearance node cannot contain c in any solution by Lemma 4 (since they have a descendant not possessing c). Therefore, any solution for T must add at least $|A_c| - 1$ transfers to T to be able to connect the first-appearance subtrees. Thus $OPT \geq |A_c| - 1$. Algorithm 2 adds exactly $|A_c| - 1$ such transfers when it considers c in its *while* loop (noting that the number of first appearance nodes for c never increases in the algorithm). Since $|A_c|$ is maximum, it follows that Algorithm 2 adds a total of at most $|\mathcal{C}|(|A_c| - 1) \leq |\mathcal{C}|OPT$ transfers. ◀

The cases in which Algorithm 2 is suboptimal appear to have a common cause. The algorithm tends to add transfers as high as possible in the tree, whereas the optmal solution would add more transfers lower in the tree, but with the advantage of being reusable by more characters. For instance in Figure 5, a is greedily transferred by itself and two more separate transfers are required for b, whereas the optimal solution reuses the same transfers for both a and b. It appears difficult to determine when to add more transfers lower and when not to, which leads us to the following.

▶ **Conjecture 12.** *The* Minimum Perfect Transfer Completion *problem is NP-hard.*

We must reckon that a $|\mathcal{C}|$-approximation is far from interesting. We conclude by suggesting a candidate approximation algorithm that combines both algorithms presented here. Suppose that, given T and Fitch-labeling λ, we obtain G and l from Algorithm 2. A simple heuristic post-processing step can then be applied to detect unneeded transfers. That is, for each transfer edge (u, v) of G, consider the network $G - (u, v)$ obtained by removing (u, v) and the resulting subdivision nodes. Then, run Algorithm 1 to check if $G - (u, v)$ is a PTN for $\mathcal{S}$. If so, we know that (u, v) can safely be removed and we repeat. We try every such transfer edge until all of them are necessary. With this modified algorithm, we were unable to generate instances with more than twice as many transfers as the optimal solution. This suggests that it might not be far from optimal.

▶ **Conjecture 13.** *Algorithm 2, combined with the above post-processing step, achieves a constant factor approximation.*

5 Conclusion

In this contribution we have introduced *perfect transfer networks* (PTNs) as a model that aims to combine the structural properties of tree-based networks with the classical notion of perfect characters. In the absence of HGT events, PTNs coincide with the notion of perfect phylogenies. To better understand these, we studied their properties, stated the main differences between them and recombination networks which is to our knowledge the closest model related to ours. Additionally, we explored several algorithmic challenges that result from this model with potential applications for HGT inference methods using character-based information that does not rely on sequence similarities. To our knowledge, this is the first theoretical work that attempts to extend the notion of perfect phylogenies to tree-based networks for the inference of HGT events.

Although widely used throughout the literature, perfect characters impose strong restrictions for our model, since each character is allowed to change of state at most once. A potential extension of our model would be to include different models for character state changes as in *Dollo parsimony* [14]. This model allows losing an acquired character but not gaining it twice. This assumption has been shown to be more suitable for complex characters such as restriction sites and introns [15]. Another possible extension of our model would be to include expression levels of the different characters instead of discrete changes which is a problem similar to that of multi-state perfect phylogenies [20].

Several other questions seem to be appealing for future work. Most importantly, since finding experimental datasets that use gene expression seems like a challenging task, the generation of *in-silico* datasets to test our algorithms seems to be a pertinent solution. Nevertheless, to our knowledge there is no simulation framework that combines evolutionary histories with gene expression data. Therefore, a future direction for this project could also be the design of a simulation environment that is able to generate this type of data.

References

1 Patrick A Alexander, Yanan He, Yihong Chen, John Orban, and Philip N Bryan. The design and characterization of two proteins with 88% sequence identity but different structure and function. *Proceedings of the National Academy of Sciences*, 104(29):11963–11968, 2007.

2 Yoann Anselmetti, Nadia El-Mabrouk, Manuel Lafond, and Aïda Ouangraoua. Gene tree and species tree reconciliation with endosymbiotic gene transfer. *Bioinformatics*, 37(Supplement_1):i120–i132, 2021.

3 Eliran Avni and Sagi Snir. A new phylogenomic approach for quantifying horizontal gene transfer trends in prokaryotes. *Scientific reports*, 10(1):1–14, 2020.

4 Vineet Bafna, Dan Gusfield, Giuseppe Lancia, and Shibu Yooseph. Haplotyping as perfect phylogeny: A direct approach. *Journal of Computational Biology*, 10(3-4):323–340, 2003.

5 Mukul S Bansal, Eric J Alm, and Manolis Kellis. Efficient algorithms for the reconciliation problem with gene duplication, horizontal transfer and loss. *Bioinformatics*, 28(12):i283–i291, 2012.

6 Hans L Bodlaender, Mike R Fellows, and Tandy J Warnow. Two strikes against perfect phylogeny. In *International Colloquium on Automata, Languages, and Programming*, pages 273–283. Springer, 1992.

7 Magnus Bordewich and Charles Semple. A universal tree-based network with the minimum number of reticulations. *Discrete Applied Mathematics*, 250:357–362, 2018.

8 Luis Boto. Horizontal gene transfer in evolution: facts and challenges. *Proceedings of the Royal Society B: Biological Sciences*, 277(1683):819–827, 2010.

9 Joseph H. Camin and Robert R. Sokal. A method for deducing branching sequences in phylogeny. *Evolution*, 19(3):311, September 1965. doi:10.2307/2406441.

10 Gjalt De Jong. Phenotypic plasticity as a product of selection in a variable environment. *The American Naturalist*, 145(4):493–512, 1995.

11 Mattéo Delabre, Nadia El-Mabrouk, Katharina T Huber, Manuel Lafond, Vincent Moulton, Emmanuel Noutahi, and Miguel Sautie Castellanos. Evolution through segmental duplications and losses: a super-reconciliation approach. *Algorithms for Molecular Biology*, 15(1):1–15, 2020.

12 Gianluca Della Vedova, Murray Patterson, Raffaella Rizzi, and Mauricio Soto. Character-based phylogeny construction and its application to tumor evolution. In *Conference on Computability in Europe*, pages 3–13. Springer, 2017.

13 Jean-Philippe Doyon, Celine Scornavacca, K Yu Gorbunov, Gergely J Szöllősi, Vincent Ranwez, and Vincent Berry. An efficient algorithm for gene/species trees parsimonious reconciliation with losses, duplications and transfers. In *RECOMB international workshop on comparative genomics*, pages 93–108. Springer, 2010.

14 James S. Farris. Phylogenetic Analysis Under Dollo's Law. *Systematic Biology*, 26(1):77–88, March 1977. doi:10.1093/sysbio/26.1.77.

15 Joseph Felsenstein. *Inferring phylogenies*. Sunderland, Mass. : Sinauer Associates, 2004.

16 David Fernández-Baca. The perfect phylogeny problem. In *Steiner Trees in Industry*, pages 203–234. Springer, 2001.

17 Andrew Francis, Charles Semple, and Mike Steel. New characterisations of tree-based networks and proximity measures. *Advances in Applied Mathematics*, 93:93–107, 2018. doi:10.1016/j.aam.2017.08.003.

18 Andrew R Francis and Mike Steel. Which phylogenetic networks are merely trees with additional arcs? *Systematic Biology*, 64(5):768–777, 2015.

19 Manuela Geiß, John Anders, Peter F Stadler, Nicolas Wieseke, and Marc Hellmuth. Reconstructing gene trees from fitch's xenology relation. *Journal of Mathematical Biology*, 77(5):1459–1491, 2018.

20 Dan Gusfield. The multi-state perfect phylogeny problem with missing and removable data: Solutions via integer-programming and chordal graph theory. *Journal of Computational Biology*, 17(3):383–399, March 2010. doi:10.1089/cmb.2009.0200.

21 Dan Gusfield. *ReCombinatorics: the algorithmics of ancestral recombination graphs and explicit phylogenetic networks*. MIT press, 2014.

22 Dan Gusfield, Satish Eddhu, and Charles Langley. Optimal, efficient reconstruction of phylogenetic networks with constrained recombination. *Journal of Bioinformatics and Computational Biology*, 2(01):173–213, 2004.

23 Marc Hellmuth, Katharina T Huber, and Vincent Moulton. Reconciling event-labeled gene trees with mul-trees and species networks. *Journal of Mathematical Biology*, 79(5):1885–1925, 2019.

24 Marc Hellmuth, Carsten R Seemann, and Peter F Stadler. Generalized fitch graphs II: Sets of binary relations that are explained by edge-labeled trees. *Discrete Applied Mathematics*, 283:495–511, 2020.

25 Julie C Dunning Hotopp. Horizontal gene transfer between bacteria and animals. *Trends in genetics*, 27(4):157–163, 2011.

26 Leo Van Iersel, Mark Jones, and Steven Kelk. A third strike against perfect phylogeny. *Systematic Biology*, 68(5):814–827, 2019.

27 Nicholas AT Irwin, Alexandros A Pittis, Thomas A Richards, and Patrick J Keeling. Systematic evaluation of horizontal gene transfer between eukaryotes and viruses. *Nature microbiology*, 7(2):327–336, 2022.

28 Edwin Jacox, Mathias Weller, Eric Tannier, and Celine Scornavacca. Resolution and reconciliation of non-binary gene trees with transfers, duplications and losses. *Bioinformatics*, 33(7):980–987, 2017.

29 Mark Jones, Manuel Lafond, and Celine Scornavacca. Consistency of orthology and paralogy constraints in the presence of gene transfers. *Peer Community in Mathematical and Computational Biology*, 2012.

30 Patrick J Keeling and Jeffrey D Palmer. Horizontal gene transfer in eukaryotic evolution. *Nature Reviews Genetics*, 9(8):605–618, 2008.

31 Eugene V Koonin, Kira S Makarova, and L Aravind. Horizontal gene transfer in prokaryotes: quantification and classification. *Annual Reviews in Microbiology*, 55(1):709–742, 2001.

32 Misagh Kordi and Mukul S Bansal. On the complexity of duplication-transfer-loss reconciliation with non-binary gene trees. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 14(3):587–599, 2015.

33 Manuel Lafond and Marc Hellmuth. Reconstruction of time-consistent species trees. *Algorithms for Molecular Biology*, 15(1):1–27, 2020.

34 Salem Malikic, Farid Rashidi Mehrabadi, Simone Ciccolella, Md Khaledur Rahman, Camir Ricketts, Ehsan Haghshenas, Daniel Seidman, Faraz Hach, Iman Hajirasouliha, and S Cenk Sahinalp. Phiscs: a combinatorial approach for subperfect tumor phylogeny reconstruction via integrative use of single-cell and bulk sequencing data. *Genome research*, 29(11):1860–1877, 2019.

35 Yukihiro Murakami. *On Phylogenetic Encodings and Orchard Networks*. PhD thesis, TU Delft, 2021.

36 Luay Nakhleh. *Phylogenetic networks*. PhD thesis, The University of Texas at Austin, 2004.

37 Luay Nakhleh, Don Ringe, and Tandy Warnow. Perfect phylogenetic networks: A new methodology for reconstructing the evolutionary history of natural languages. *Language*, 81(2):382–420, 2005. URL: `http://www.jstor.org/stable/4489897`.

38 Joan Carles Pons, Charles Semple, and Mike Steel. Tree-based networks: characterisations, metrics, and support trees. *Journal of Mathematical Biology*, 78(4):899–918, October 2018. `doi:10.1007/s00285-018-1296-9`.

39 Joan Carles Pons, Charles Semple, and Mike Steel. Tree-based networks: characterisations, metrics, and support trees. *Journal of Mathematical Biology*, 78(4):899–918, 2019.

40 Beatriz Pontes, Raúl Giráldez, and Jesús S Aguilar-Ruiz. Configurable pattern-based evolutionary biclustering of gene expression data. *Algorithms for Molecular Biology*, 8(1):1–22, 2013.

41 Dikshant Pradhan and Mohammed El-Kebir. On the non-uniqueness of solutions to the perfect phylogeny mixture problem. In *RECOMB International Conference on Comparative Genomics*, pages 277–293. Springer, 2018.

42 Matt Ravenhall, Nives Škunca, Florent Lassalle, and Christophe Dessimoz. Inferring horizontal gene transfer. *PLoS Computational Biology*, 11(5):e1004095, 2015.

43 Arun Rawat, Georg J Seifert, and Youping Deng. Novel implementation of conditional co-regulation by graph theory to derive co-expressed genes from microarray data. In *BMC Bioinformatics*, volume 9, pages 1–9. Springer, 2008.

44 Don Ringe, Tandy Warnow, and Ann Taylor. Indo-european and computational cladistics. *Transactions of the Philological Society*, 100(1):59–129, 2002.

45 Michael J Sanderson and Larry Hufford. *Homoplasy: the recurrence of similarity in evolution*. Elsevier, 1996.

46 Palash Sashittal, Simone Zaccaria, and Mohammed El-Kebir. Parsimonious clone tree reconciliation in cancer. In *Leibniz International Proceedings in Informatics, LIPIcs*, volume 201, page 9. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021.

47 David Schaller, Manuel Lafond, Peter F Stadler, Nicolas Wieseke, and Marc Hellmuth. Indirect identification of horizontal gene transfer. *Journal of Mathematical Biology*, 83(1):1–73, 2021.

48 Charles Semple and Mike Steel. Tree reconstruction from multi-state characters. *Advances in Applied Mathematics*, 28(2):169–184, 2002.

49 Christopher M Thomas and Kaare M Nielsen. Mechanisms of, and barriers to, horizontal gene transfer between bacteria. *Nature Reviews Microbiology*, 3(9):711–721, 2005.

50 Ali Tofigh, Michael Hallett, and Jens Lagergren. Simultaneous identification of duplications and lateral gene transfers. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(2):517–535, 2010.

51 Catalina Trejo-Becerril, Enrique Pérez-Cárdenas, Lucía Taja-Chayeb, Philippe Anker, Roberto Herrera-Goepfert, Luis A Medina-Velázquez, Alfredo Hidalgo-Miranda, Delia Pérez-Montiel, Alma Chávez-Blanco, Judith Cruz-Velázquez, et al. Cancer progression mediated by horizontal gene transfer in an in vivo model. *PloS One*, 7(12):e52754, 2012.

52 Lusheng Wang, Kaizhong Zhang, and Louxin Zhang. Perfect phylogenetic networks with recombination. *Journal of Computational Biology*, 8(1):69–78, 2001.

A Appendix

Proof of Proposition 3

Proof. For (1), consider the network G in Figure 2.1. For any $\mathcal{C}$, the labeled network shows that (G, σ) is in $PTN(\mathcal{S})$. Put $\mathcal{C} = \{c_1, \ldots, c_{d+2}\}$. We argue that (G, σ) is not in $ARG_d(\mathcal{S})$. Note that in any binary $\mathcal{C}$-labeling of G that explains $\mathcal{S}$, the parent of each $\emptyset$ leaf taxa must be the string $00\ldots0$, as otherwise they would need to transmit a 1 to these leaves. Likewise, the parent of the upper $\mathcal{C}$ leaf taxon must be $11\ldots1$. If not, there would be a $0-1$ flip on the edge leading to the upper $\mathcal{C}$. But, we would also need another $0-1$ flip somewhere on the path to the lower $\mathcal{C}$ taxon, which is not allowed in ARGs. So, the parents of the single reticulation have labels $00\ldots0$ and $11\ldots1$. Moreover, the root is labeled $00\ldots0$, and thus all the possible $0-1$ flips occur between the root and its second child. We cannot have any new $0-1$ flip, and so the reticulation must be labeled $0101\ldots01$ to be able to transmit the required characters to the odd-characters leaf (or the last character is 0 if d is even). Whether d is odd or even, there are $d+2$ characters and we must alternate each of them between the parents. This requires $d+1$ crossovers, and so $G \notin ARG_d(\mathcal{S})$.

For (2), consider the network G in Figure 2.2. There are only two characters c_1, c_2. The right side of the figure shows that G can be explained under single crossovers. However, we can argue that (G, σ) is not a perfect transfer network. This is because if there is a

$\mathcal{C}$-labeling l that explains the taxa at the leaves, the parent of each c_1 leaf must contain c_1. But by the *never lose once acquired* condition, c_1 should then be transmitted vertically to the leaf that has no character, a contradiction. Thus $(G, \sigma) \notin PTN(\mathcal{S})$.

For (3), let $(G, \sigma) \in PTN(\mathcal{S})$. Let l be a $\mathcal{C}$-labeling of G that explains $\mathcal{S}$. To show that $(G, \sigma) \in ARG_\infty(\mathcal{S})$, we need to modify l slightly. Specifically, we ensure that characters never have their first appearance on a reticulation node. Let v be a reticulation node of G with parents u, w such that (u, v) is a support edge and (w, v) is a transfer edge. Suppose that there is $c_j \in \mathcal{C}$ such that $c_j \notin l(u), c_j \notin l(w)$, but $c_j \in l(v)$. It follows from Theorem 2 that v is the unique node of in-degree 0 in $G[V_{c_j}(l)]$. Consider the labeling l' obtained by taking l, but applying $l'(v) = l(v) \setminus \{c_j\}$. Let v' be the unique child of v. One can easily see that $G[V_{c_j}(l')]$ is connected and that v' is its unique node of in-degree 0. Moreover, $\mathcal{T}_G[\overline{V}_{c_j}(l')]$ is the same as $\mathcal{T}_G[\overline{V}_{c_j}(l)]$ but with v added. It is still connected since v is a child of $u \in \overline{V}_{c_j}(l)$, and it still contains the root. Hence by Theorem 2, l' also explains $\mathcal{S}$.

By applying the above argument to every character, we obtain a labeling l that explains $\mathcal{S}$ such that for any character c_j, the first appearance of c_j under l does not occur at a reticulation node. Assume that l has this property.

Consider the binary $\mathcal{C}$-labeling f of G in which, for each $v \in V(G)$, we put $f(v)[j] = 1$ if and only if $c_j \in l(v)$. We claim that f explains $\mathcal{S}$ with m-crossovers, where $m = |\mathcal{C}|$. First consider a reticulation node v with parents u and w, where (u, v) is a support edge and (w, v) is a transfer edge. We must argue that $f(v)$ is an m-crossover of $f(u)$ and $f(w)$. Because under l, no character first appears on a reticulation, we have that for each $c_j \in l(v)$, either $c_j \in l(u)$ or $c_j \in l(w)$. Moreover, for each $c_j \notin l(v)$, we must have $c_j \notin l(u)$ (by the *never lose once acquired* condition). In terms of f, this means that for each $j \in [m]$, if $f(v)[i] = 1$ then one of u or w also has a 1 in position j, and if $f(v) = 0$, then $f(u)[j] = 0$. It follows that $f(v)$ can be obtained with at most m crossovers from its parents.

Second, it is not hard to see that for each character c_j, there is at most one tree edge (u, v) that flips c_j from 0 to 1 under f. Indeed, if there were two such edges, then under l, there would be two trees nodes that possess c_j but not their (unique) parent. Thus $G[V_{c_j}(l)]$ would have two nodes of in-degree 0, contradicting Theorem 2. Third, the condition that characters are not lost after being acquired implies that, for every tree edge (u, v), $f(v)$ can be obtained from $f(u)$ by flipping 0s to 1s, but not vice-versa (this follows from the fact that tree edges are support edges). Thus, f explains $\mathcal{S}$ and we conclude that $(G, \sigma) \in ARG_\infty(\mathcal{S})$. ◀

Proof of Lemma 8

Proof. We argue that at any moment during the execution of the algorithm, τ is a time-consistent map of G. We prove this by induction on the number of iterations undertaken. Notice that as a base case, the statement is initially true before entering the main *for* loop. Now assume that we have inserted $j - 1$ transfer edges and that τ is time-consistent for G after these insertions. Consider the j-th transfer edge inserted into G. This transfer edge is $(\hat{w}, \hat{a}_{i+1})$, which are created between w and its parent w' in the support tree T', and between a_{i+1} and its parent a'_{i+1} in T', respectively.

We first claim that (w', w) as used in the algorithm exists in the subtree $T(a_i)$, where a_i is the node that precedes a_{i+1} in X_c. We know that $\tau(a_i) \geq \tau(a_{i+1})$. Suppose that $\tau(a_i) = \tau(a_{i+1})$. Let a'_i be the parent of a_i, then by induction hypotesis, since the network is time consistent we have that $\tau(a'_i) > \tau(a_i) = \tau(a_{i+1})$ and so $w' = a'_i$ and $w = a_i$. In this case, we do have $\tau(w') > \tau(a_{i+1})$ and $\tau(w) \leq \tau(a_{i+1})$. Now, suppose that $\tau(a_i) > \tau(a_{i+1})$. Note that we can always order the vertices of $T(a_i)$ with respect to τ in such a way that

$\tau(a_i) \geq v$ for all $v \in V(T(a_i))$. By induction hypothesis, every time we pick a descendant y of a_i, $\tau(y) < \tau(a_i)$, so w can be found by iteratively following the descendants of a_i and choosing the first one whose time is at most $\tau(a_{i+1})$.

Note that by adding the transfer edge $(\hat{w}, \hat{a}_{i+1})$, the times τ of every edge have remained unchanged and still satisfy the time-consistency definition, with the exception of the edges linking $w', \hat{w}$, and w, those linking $a'_{i+1}, \hat{a}_{i+1}$, and a_{i+1}, as well as the new transfer edge. Since $\tau(\hat{w}) = \tau(\hat{a}_{i+1})$ is made explicit on line 16, to conclude the proof it suffices to show that $\tau(w') > \tau(\hat{w}) > \tau(w)$ and that $\tau(a'_{i+1}) > \tau(\hat{a}_{i+1}) > \tau(a_{i+1})$. By induction we know that $\tau(w') > \tau(w)$ and that $\tau(a'_{i+1}) > \tau(a_{i+1})$ (this holds before and after the transfer insertion because they were not changed). Using these inequalities we have that:

$$\tau(\hat{w}) = \frac{\min(\tau(w'), \tau(a'_{i+1})) + \tau(a_{i+1})}{2} < \frac{\tau(w') + \tau(w')}{2} = \tau(w')$$

since $\min(\tau(w'), \tau(a'_{i+1})) \leq \tau(w')$ and $\tau(a_{i+1}) < \tau(w')$ both hold. Note that it is also true that

$$\tau(\hat{w}) > \frac{\tau(a_{i+1}) + \tau(a_{i+1})}{2} = \tau(a_{i+1}) \geq \tau(w)$$

since $\min(\tau(w'), \tau(a'_{i+1})) > \tau(a_{i+1})$ (because $\tau(w') > \tau(a_{i+1})$ and $\tau(a'_{i+1}) > \tau(a_{i+1})$ both hold). Using the same arguments, we have

$$\tau(\hat{a}_{i+1}) = \frac{\min(\tau(w'), \tau(a'_{i+1})) + \tau(a_{i+1})}{2} < \frac{\tau(a'_{i+1}) + \tau(a'_{i+1})}{2} = \tau(a'_{i+1})$$

and

$$\tau(\hat{a}_{i+1}) > \frac{\tau(a_{i+1}) + \tau(a_{i+1})}{2} > \tau(a_{i+1})$$ ◀

Proof of Lemma 9

Proof. Let $G = (V, E_S \cup E_T)$ be the network returned by the algorithm and let l be the returned labeling. To see that $l(v) = \lambda(v)$ for every node v in the base tree, it suffices to note that the algorithm never changes the labeling of a node initially present in T: it only assigns sets of characters to nodes created by transfer insertion operations. It is also easy to see that T is the base tree of G, since we start with a copy of T and only attach transfer edges to it.

We will argue that the returned $\mathcal{C}-$labeling l explains $\mathcal{S}$ using the conditions required by Definition 1. First, by requirement on the input λ, we know that for every $v \in L(G)$, $l(v) = \lambda(v) = \sigma(v)$.

Let us next show that for every character $c \in \mathcal{C}$ there exists a unique node $v \in V_c(l)$ that reaches every node in $G[V_c(l)]$. It is not hard to see that after handling a particular $c \in \mathcal{C}$ in the algorithm, this property will be satisfied for $V_c(l)$. However, it is not obvious that the subsequent iterations on other characters will not "break" this property for c. We thus prove the following statement. Assume that the algorithm handles the characters of $\mathcal{C}$ in order $c_1, \ldots, c_m$ in the main *while* loop. Then we claim that in the network G obtained after finishing the i-th iteration and handling $c_1, c_2, \ldots, c_i$, for every $j \leq i$, there exists a unique node $v \in V_{c_j}(l)$ that reaches every node in $G[V_{c_j}(l)]$. This shows the desired property since it will hold for $j = m$, i.e. for every character. As a base case, consider $i = 0$. Then it is true that for $j \leq i$, the desired node v exists (because there is no c_j to satisfy). Now, assume that $i > 0$ and that before entering the i-th iteration, the statement holds for every c_j with $j \leq i - 1$.

After we are done handling c_i on the $i-$th iteration we know that there exists a vertex $a_1 \in X_{c_i}$ such that $\tau(a_1) \geq \tau(a_h)$ for all $a_h \in X_{c_i}$. In particular, when equality holds for some a_h, it must be that $\tau(a_1) = \tau(a_2)$. In this case, when the algorithm iterates on a_1, we get $(w', w) = (a'_1, a_1)$ and the corresponding transformation yields $G\blacktriangle(a_1, a_2)$. In this case, we claim that the vertex $\hat{a}_1$ created by the subdivision of the edge $(a'_1, a_1) \in E_s$ will remain as the unique source for $V_{c_i}(l)$. To see this first note that a_1 has no incoming edge from $V_c(l)$ at the start of the iteration, because if that was not the case then $c \in l(a'_1)$ and so a_1 would not have been a first-appearance node in the first place. This in turn implies that $\hat{a}_1$ has no incoming edge after the i-th iteration, since all other transfer heads are added above $a_2, \ldots, a_k$. Additionally, $\hat{a}_1$ will become the first-appearance node for its corresponding subtree. After applying $G\blacktriangle(a_1, a_2)$, $\hat{a}_1$ now reaches the new parent $\hat{a}_2$ of a_2 and all its descendants in $\mathcal{T}_G$. Subsequently, we will now choose a descendant w of a_2 from which we will add a new transfer to the parent node $\hat{a}_3$ of a_3. In this way, $\hat{a}_1$ will also reach a_3 and all of its descendants. We will continue adding transfer operations in this way so that finally $\hat{a}_1$ will be able to reach the last vertex of X_{c_j}, a_k and all of its descendants. Note that any node of $V_{c_i}(l)$ is reachable by an element of A_{c_i}, thus after the $i-$th iteration $\hat{a}_1$ reaches every node in the $G[V_{c_i}(l)]$ subgraph.

On the other hand, when we have the strict inequality, i.e. when $\tau(a_1) > \tau(a_2)$, then the first chosen w is distinct from a_1, and using the same arguments, we see that a_1 is the unique origin for $V_{c_i}(l)$.

We must also argue that the i-th iteration does not "break" a c_j with $j < i$. Consider such a c_j. By induction, before the i-th iteration, $G[V_{c_j}(l)]$ had a unique origin v. Suppose that during the $i-$th iteration of the main loop we created some transformation $G\blacktriangle(w, a_h)$ after which some node, say z, that possesses c_j cannot be reached by v in the transformed graph. Let w', a'_h be the parents of w and a_h, respectively, before the addition of the transfer. Also let $\hat{w}$ and $\hat{a}_h$ be the nodes which were created by this transformation.

First assume that $z = \hat{w}$. Then $c_j \notin l(w')$, as otherwise if $c_j \in l(w')$, by induction, v would reach w' and thus also reach $\hat{w} = z$. But because $c_j \notin l(w')$, $c_j \notin l(w') \cap l(w)$ and so the algorithm would not have put c_j in $\hat{w} = z$, a contradiction. By the same argument, $z \neq \hat{a}_h$. Thus, z was present in G before the insertion of the transfer.

Next, assume that $c_j \notin l(\hat{w})$ and $c_j \notin l(\hat{a}_h)$. If v cannot reach z anymore, every path from v to z in $G[V_{c_j}(l)]$ must have been going through (w', w) or (a'_h, a_h) before the transfer insertion. But then, $c_j \in l(w') \cap l(w)$ and $c_j \in l(a'_h) \cap l(a_h)$, and the algorithm would have put $c_j \in l(\hat{w})$ and $c_j \in l(\hat{a}_h)$, a contradiction.

Then either $c_j \in l(\hat{w})$, $c_j \in l(\hat{a}_h)$ or $c_j \in l(\hat{w}) \cap l(\hat{a}_g)$. Assume $c_j \in l(\hat{w})$. By line 15, we know that this would only be possible if $c_j \in l(w') \cap l(w)$. So any path from v to z in $V_{c_j}(l)$ that used the (w', w) edge can now use the edges $(w', \hat{w}), (\hat{w}, w)$ to reach z. If $c_j \in l(\hat{a}_h)$ as well, the same idea applies, and we get that any path from v to z is still usable, albeit with either $\hat{w}$ or $\hat{a}_h$ as an additional vertex. So it must be that $c_j \notin l(\hat{a}_h)$, and that all paths used (a'_h, a_h). As before, this means that $c_j \in l(a'_h) \cap l(a_h)$ and that we should have $c_j \in l(\hat{a}_h)$, a contradiction. This covers the case $c_j \in l(\hat{w})$. The case $c_j \in l(\hat{a}_h)$ can be handled in the same manner.

We then show that for each support edge $(u, v) \in E_S, c \in l(u)$ implies that $c \in l(v)$. We argue that this property holds before and after any transfer edge is inserted. Notice that initially, when G is just a copy of T and l a copy of λ, $c \in l(u)$ implies $c \in l(v)$ because λ is a no-loss labeling. Now suppose inductively that the property holds before we insert some transfer $(\hat{w}, \hat{a}_{i+1})$ by line 13. It suffices to argue that the property holds on the support edges $(w', \hat{w}), (\hat{w}, w), (a'_{i+1}, \hat{a}_{i+1})$, and $(\hat{a}_{i+1}, a_{i+1})$, as defined in the algorithm, because no

other support edge is modified. Let $c' \in l(w')$ (we distinguish c' from c, the latter being the c the algorithm is currently iterating on). Then by assumption that the property held before the transfer addition, we must have $c' \in l(w)$ and, because $c' \in l(w) \cap l(w')$, c' will be added to $l(\hat{w})$, as desired. The same argument holds for $c' \in l(a'_{i+1})$ and the fact that $c' \in l(\hat{a}_{i+1})$. Now let $c' \in l(\hat{w})$. We want to argue that $c' \in l(w)$. If $c' \in l(w')$, then again by assumption we have $c' \in l(w)$ as well and our property holds. So suppose that $c' \notin l(w')$. The algorithm puts $l(\hat{w}) = (l(w) \cap l(w')) \cup \{c\}$ where c is the character of the current iteration, which means that only $c = c'$ is possible. Notice that the algorithm chooses the edge (w, w') in the subtree $\mathcal{T}_G(a'_i)$, where a'_i has child a_i that is a first appearance node for c. By assumption, every descendant of a_i in the support tree possesses c, so only $w' = a'_i$ is possible. Thus $w = a_i$ and $c \in l(a_i) = l(w)$, as desired. Finally, let $c' \in l(\hat{a}_{i+1})$. If $c' \in a'_{i+1}$, by assumption $c' \in l(a_{i+1})$ and we are done. Otherwise, as the previous case we must have $c' = c$ and, since a_{i+1} is a first appearance for c, we have $c \in l(a_{i+1})$ as desired. ◀

Haplotype Threading Using the Positional Burrows-Wheeler Transform

Ahsan Sanaullah
Department of Computer Science, University of Central Florida, Orlando, FL, USA

Degui Zhi[1]
School of Biomedical Informatics, University of Texas Health Science Center, Houston, TX, USA

Shaoije Zhang[1]
Department of Computer Science, University of Central Florida, Orlando, FL, USA

Abstract

In the classic model of population genetics, one haplotype (query) is considered as a mosaic copy of segments from a number of haplotypes in a panel, or threading the haplotype through the panel. The Li and Stephens model parameterized this problem using a hidden Markov model (HMM). However, HMM algorithms are linear to the sample size, and can be very expensive for biobank-scale panels. Here, we formulate the haplotype threading problem as the Minimal Positional Substring Cover problem, where a query is represented by a mosaic of a minimal number of substring matches from the panel. We show that this problem can be solved by a sequential set of greedy set maximal matches. Moreover, the solution space can be bounded by the left-most and the right-most solutions by the greedy approach. Based on these results, we formulate and solve several variations of this problem. Although our results are yet to be generalized to the cases with mismatches, they offer a theoretical framework for designing methods for genotype imputation and haplotype phasing.

2012 ACM Subject Classification Applied computing → Computational biology; Applied computing → Genetics

Keywords and phrases Substring Cover, PBWT, Haplotype Threading, Haplotype Matching

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.4

Funding This work was supported by the National Institutes of Health grants R01 HG010086 and R56 HG011509.

1 Introduction

In modeling a panel of haplotype sequences arising from population genetics processes, a useful model is to view a haplotype as a mosaic copy of subsequences of other haplotypes in a panel. In other words, a haplotype (the query) is threaded through different haplotypes (as templates) in the panel. The classic Li and Stephens model parameterized this process as a hidden Markov model which can take into account the uncertainties regarding mismatches (emission probabilities) and template switching (transition probabilities) [3]. This model is quite sufficient for moderate sample sizes (hundreds to thousands) as it scales linearly with sample size. As a result, this model has served as a foundation for haplotype phasing and genotype imputation for the past two decades. However, in the biobank era when the panel size is large, the standard Li and Stephens model may not be efficient enough.

PBWT as a foundational data structure enables efficient substring matching in a population panel with aligned haplotypes [2]. For a panel with M haplotypes of length N, existing algorithms include space-efficient scanning algorithms for all-vs-all within-panel long and set maximal matches in $O(MN + c)$ time, where c is the number of matches outputted [2].

[1] To whom correspondence should be addressed.

Algorithms for time-efficient one-vs-all out-of-panel query long and set maximal match search in $O(N + c)$ time are also available [6, 10]. These algorithms allow efficient match query on biobank scale databases.

Indeed, the PBWT has been leveraged for speeding up the Li and Stephens model. Lunter proposed a representation of the PBWT using the BWT [5]. On this representation, they perform a search for a maximum likelihood Viterbi path through the Li and Stephens Hidden Markov Model. While their algorithm can compute the optimal score in $O(N)$ time, outputting of the haplotype threading requires $O(N \log M)$ time. Rubinacci et al. use the PBWT to select closely related individuals efficiently. These closely related individuals are then imputed using IMPUTE5, an imputation based on the Li and Stephens model [9]. Loh et al. use the PBWT similarly for EAGLE2, their phasing algorithm. They use the PBWT to obtain representative data of a haplotype and then thread it using a haplotype copying model similar to the Li and Stephens model [4]. Lastly, Delanueau et al. use the PBWT to speed up phasing through the Li and Stephens model among other methodologies in their phasing method SHAPEIT4 [1].

However, these algorithms [1, 4, 5, 9] are designed within the Li and Stephens HMM framework and the PBWT is used as a subroutine. These algorithms mostly focus on the Viterbi path that gives the maximum likelihood solution. We argue that in the biobank-scale panel, a query may have a large number of high-quality matches, and thus outputting the single best Viterbi solution may not be informative to reveal the overall high probability possible paths. In this work, we formulate the haplotype threading problem as a combinatorial optimization problem: given a set of haplotypes X and a query z, represent z as segments of haplotypes in X and optimize a certain objective scoring function.

There are a number of possible scoring functions for threading, however there are common themes between them. Usually, one wants to represent z using a small amount of haplotypes in X or a small amount of distinct segments. In this paper, we minimize the number of segments we use to represent z. We formulate the Minimum Positional Substring Cover problem (MPSC), given a query z and a set of strings X, find a smallest set of positional substrings contained in z and a string in X that cover all characters of z. While this formulation simplifies the original haplotype threading by ignoring potential mismatches in the Li and Stephens model flavor, it enables efficient enumeration of all possible solutions, leveraging the structure of PBWT. Augmenting our algorithms with mismatch-tolerating methods such as random projection [7] or PBWT-smoothing [11], our formulation can capture the bulk of the high-probability threading paths, and thus provide flexibility for designing variations of downstream tasks such as genotype imputation and haplotype phasing.

We provide a solution in time linear to the length of the query string given a PBWT of X. Occasionally, the segment being present in only one string in X is not enough evidence to use it to represent z. Therefore, we formulate the h-Minimal Positional Substring Cover (h-MPSC) problem. Given a query string z and a set of strings X, find a smallest set of segments present in z and h strings in X. We provide a solution in $O(h|\mathcal{C}| + N)$ time, where $|\mathcal{C}|$ is the number of segments outputted and N is the length of z.

In Section 3, we formulate the Minimal Positional Substring Cover problem, discuss multi-allelic PBWTs, discuss properties of MPSCs, and provide a solution given a PBWT of X. In Section 4, we discuss three variations of the MPSC problem: the leftmost, rightmost, and set maximal match only minimal positional substring covers. Then we show that our original solution is leftmost and provide linear time solutions for the rightmost and set maximal match only MPSC problems. In Section 5, we discuss the h-MPSC problem and

provide a solution in $O(h|\mathcal{C}| + N)$ time. Then, in Section 6 we discuss boundary cases in these algorithms. Lastly, we cover possible future work in haplotype threading related to the Minimal Positional Substring Cover problem in Section 7.

2 Background

We index the characters of a string z with N characters from 0 to $N-1$. The first character is $z[0]$ and the last character is $z[N-1]$. The string $z[i,j)$ is the substring of z that starts at character i and ends at character $j-1$. $z[i,j]$ is the substring of z that starts at character i and ends at character j. $z(i,j]$ is the substring of z that starts at character $i+1$ and ends at character j, $z(i-1,j] = z[i,j] = z[i,j+1)$.

A **positional substring** of a string z is a 3-tuple, (i,j,z), where i and j are nonnegative integers, $i \leq j+1 \leq |z|$, and z is the "source" of the substring. The substring corresponding to the positional substring (i,j,z) is $z[i,j]$. If $i = j+1$, then (i,j,z) corresponds to the empty string, ε. The number of characters in z is $j-i+1$. A non-empty positional substring (i,j,z) is contained (or present) in a string s if $0 \leq i \leq j < |s|$ and $s[i,j] = z[i,j]$. A positional substring (i,j,z) corresponding to an empty string is contained in a string s iff $0 \leq j \leq |s|$. Two positional substrings, (i,j,s) and (k,l,t) are equal iff $i = k, j = l$, and $s[i,j] = t[k,l]$.

A **positional substring cover** $\mathcal{C}$ of a string z by a set of strings X is a set of positional substrings such that every character of z is contained in a positional substring and every positional substring in the set is present in z and a string in X. I.E., $\forall k \in \{0,\ldots,|z|-1\}, \exists (i,j,s) \in \mathcal{C}$ s.t. $i \leq k \leq j$, and $\forall (i,j,s) \in \mathcal{C}, \exists x \in X$ s.t. $x[i,j] = z[i,j] = s[i,j]$. The "source" s of a positional substring (i,j,s) in a positional substring cover can be any string s s.t. $s[i,j] = x[i,j] = z[i,j]$ for some $x \in X$. The size of a positional substring cover is the number of positional substrings it contains.

π is a projection on tuples. $\pi_1(i,j,z) = i, \pi_2(i,j,z) = j$, and $\pi_3(i,j,z) = z$.

2.1 Positional Burrows-Wheeler Transform

The Positional Burrows-Wheeler Transform (PBWT) is an index on X, a set of M binary strings of length N. It was published by Richard Durbin in 2014 [2]. It allows efficient search for positional matches in binary strings. Furthermore, it allows efficient compression of binary strings that have local correlation. The main idea of the PBWT is it stores four two-dimensional integer arrays of size around $M \times N$. The prefix array, a contains $N+1$ sortings of the M binary strings. Sorting i in a contains the strings $x \in X$ sorted by their reversed prefixes of length i. If two strings have the same reversed prefix of length i, their relative order in $a[i]$ and $a[0]$ are the same. The divergence array d contains $N+1$ integer arrays of length M. The j-th integer in the i-th array, $d[i][j]$, contains the starting position of the longest match between $a[i][j]$ and $a[i][j-1]$ that ends at i. The u array has N arrays of length M. The j-th position in the i-th array contains the position the string $a[i][j]$ would have in $a[i+1]$ if it had a 0 at index i. The v array has N arrays of length M. The j-th position in the i-th array contains the position the string $a[i][j]$ would have in $a[i+1]$ if it had a 1 at index i.

We now define the PBWT with notation commonly used in this paper in order to give the reader a deeper understanding of the PBWT and this paper's notation. We use $\mathrm{R}(z)$ to denote the reverse of the string z. Call $y_{i,j}$ the string corresponding to the ID $a[i][j]$. Each binary string in X is given a unique integer ID in $\{0,\ldots,M-1\}$. The $a[i]$ array contains the IDs of the strings $x \in X$ sorted by $R(x[0,i))$. $d[i][j] = \min_{k \in \{l \in \{0,\ldots,i\} : y_{i,j}[l,i) = y_{i,j-1}[l,i)\}} k$. $u[i][0] = 0$.

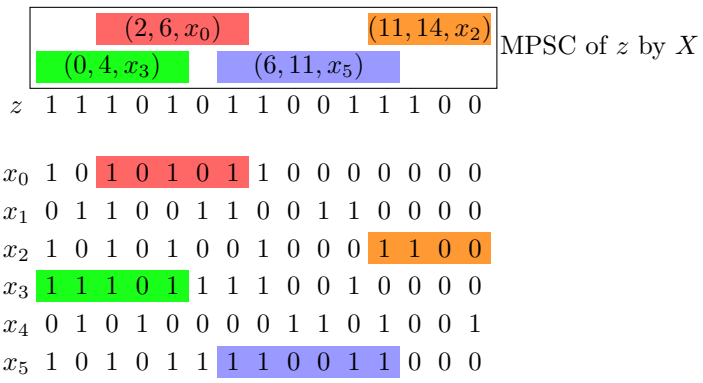


Figure 1 A minimal positional substring cover of z by X.

If $y_{i,j}[i] = 0$, $u[i][j+1] = u[i][j]+1$. Otherwise, $u[i][j+1] = u[i][j]$. $v[i][0] = u[i][M-1]+1$ if $y_{i,M-1}[i] = 0$, otherwise $v[i][0] = u[i][M-1]$. If $y_{i,j}[i] = 1$, $v[i][j+1] = v[i][j]+1$. Otherwise, $v[i][j+1] = v[i][j]$. The definitions in this paragraph and the one above are equivalent.

Richard Durbin introduced useful definitions of matches between strings. A match between two strings, s and t ($(i, j, s) = (i, j, t)$), is **locally maximal** if it can't be extended in any direction and still match. ($s[i-1] \neq t[i-1]$ or $i = 0$) and ($s[j+1] \neq t[j+1]$ or $j = N-1$). When comparing a string s to a set of strings X, a match from s to $x \in X, ((i, j, s) = (i, j, x))$, is **set maximal from s to X** if it is locally maximal and there does not exist a match from s to a string in X that is larger and contains this match. $\forall t \in X \forall k \in \{0, \ldots, i\} \forall l \in \{j, \ldots, M\}, (k = i$ and $l = j)$ or $t[k, l] \neq s[k, l]$. Lastly, a match from s to $x \in X, (i, j, s) = (i, j, x)$, is a **longest match from s to X ending at index j** if it is a longest match between s and all strings in X that ends at index j. $\forall t \in X \forall k \in \{0, \ldots, i\}, k = i$ or $t[k, j] \neq s[k, j]$.

3 Minimal Positional Substring Cover

The Minimal Positional Substring Cover (MPSC) problem is, given a set X of M strings and a string z, find a positional substring cover of z by X of the smallest size out of all positional substring covers of z by X. Call this cover a minimal positional substring cover of z by X. Refer to Figure 1 for a depiction of an MPSC.

3.1 Properties

▷ Claim 1. A minimal positional substring cover of z by X exists if and only if for every $i \in \{0, \ldots, |z|-1\}$, there exists a string in X that has the same character as z at index i.

Proof. If there exists an $i \in \{0, \ldots, |z|-1\}$ s.t. $\forall x \in X, z[i] \neq x[i]$, there exists no positional substring cover of z by X since a positional substring that covers index i and is contained in z and a string in X doesn't exist. There doesn't exist a positional substring cover of z by X, so a minimal positional substring cover of z by X doesn't exist.

If $\forall i \in \{0, \ldots, |z|-1\}, \exists x \in X$ s.t. $z[i] = x[i]$, then there exists a positional substring cover $\mathcal{C}$ of z by X. $\mathcal{C} = \{ (i, i, z) : i \in \{0, \ldots, |z|-1\} \}$. A positional substring cover of z by X exists, so a minimal positional substring cover of z by X exists. ◁

▷ Claim 2. For a minimal positional substring cover $\mathcal{C}$ of z by X, every index $k \in \{0, \ldots, |z|-1\}$ is contained in at most two of its positional substrings. $\forall k \in \{0, \ldots, |z|-1\}, \exists (i_0, j_0, s_0), (i_1, j_1, s_1) \in \mathcal{C}$ s.t. $\forall (i_2, j_2, s_2) \in \mathcal{C}, i_2 \leq k \leq j_2 \iff ((i_2 = i_1 \wedge j_2 = j_1 \wedge s_2 =$

$s_1) \vee (i_2 = i_0 \wedge j_2 = j_0 \wedge s_2 = s_0))$. Note that every index is contained in at least one positional substring by the definition of positional substring cover, therefore, every index is contained in at least one and at most two positional substrings in $\mathcal{C}$.

Proof. Suppose there exists an index k that is contained in more than two positional substrings in a minimal positional substring cover $\mathcal{C}$ of z by X. Take $\mathcal{D}$, the set of positional substrings in $\mathcal{C}$ that contain i, $\mathcal{D} = \{p \in \mathcal{C} : \pi_1(p) \leq i \leq \pi_2(p)\}$. Take the positional substrings $p, q \in \mathcal{D}$ that start the earliest and end the latest respectively. $\forall r \in \mathcal{D}, \pi_1(p) \leq \pi_1(r)$ and $\pi_2(q) \geq \pi_2(r)$. p covers at least $[\pi_1(p), i]$ and q covers at least $[i, \pi_2(q)]$. Therefore, all the indices contained in positional substrings in $\mathcal{D}$ are covered by p and q. Therefore, removing all positional substrings in $\mathcal{D}$ except p and q from $\mathcal{C}$ would yield a smaller set of positional substrings that covers z by X (since $|\mathcal{D}| > 2$). This contradicts the fact that $\mathcal{C}$ is a minimal positional substring cover of z by X. ◁

▷ Claim 3. Given a minimal positional substring cover $\mathcal{C}$ of z by X, the starting points of all positional substrings in $\mathcal{C}$ are unique and their ending points are unique. $\forall p, q \in \mathcal{C}, p \neq q \implies (\pi_1(p) \neq \pi_1(q) \wedge \pi_2(p) \neq \pi_2(q))$.

Proof. Suppose that there are two positional substrings in $\mathcal{C}$ that start at the same position, i. Then, the one with the smaller ending position can be removed from $\mathcal{C}$. This new set is a smaller positional substring cover of z by X because all of the sites are still covered by positional substrings contained in z and a string in X. This contradicts the assumption that $\mathcal{C}$ is a minimal positional substring cover of z by X. Similar reasoning can be applied to positional substrings with the same ending position. ◁

▷ Claim 4. For a minimal positional substring cover $\mathcal{C}$ of z by X. For any positional substring $p \in \mathcal{C}$, if p has the i-th smallest starting point out of all positional substrings in $\mathcal{C}$, it also has the i-th smallest ending point out of all positional substrings in $\mathcal{C}$. $\forall p, q \in \mathcal{C}, \pi_1(p) < \pi_1(q) \iff \pi_2(p) < \pi_2(q)$.

Proof. Suppose there are two positional substrings, $p, q \in \mathcal{C}$ s.t. the order of their starting points and ending points are different. Without loss of generality, say p starts earlier. Then, we have $\pi_1(p) < \pi_1(q)$ and $\pi_2(p) > \pi_2(q)$. In this case, q is completely covered by p. Removing q from $\mathcal{C}$ results in a positional substring cover of z by X that is smaller than $\mathcal{C}$. This contradicts the assumption that $\mathcal{C}$ is a minimal positonal substring cover. ◁

We use Claim 4 to define i-th positional substring in a minimal positional substring cover of z by X. The i-th positional substring in a minimal positional substring cover $\mathcal{C}$ is the positional substring with the i-th smallest starting position in the cover, 0-indexed. This is equivalent to ordering by ending position (by Claim 4). We use $\mathcal{C}[i]$ to denote the i-th positional substring of $\mathcal{C}$.

3.2 Main Idea

We present a solution to the Minimal Positional Substring Cover problem here. This solution assumes that a PBWT of X is provided. Note that a PBWT specifies a binary alphabet, whereas the Minimal Positional Substring Cover problem doesn't assume a binary alphabet. There are two ways to deal with this. Firstly, it is possible to convert a string s in an arbitrary alphabet Σ into a binary string where every character is represented by $\lceil \log |\Sigma| \rceil$ binary characters. Minor modifications on the PBWT algorithms would provide algorithms that take a multiplicative factor of $O(\log |\Sigma|)$ extra time and space. Another solution is as

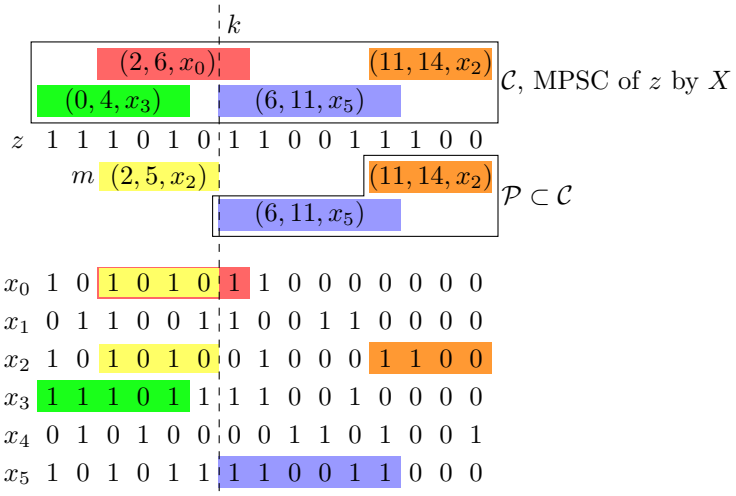


Figure 2 Depiction of Lemma 1. $\mathcal{C}$ is a MPSC of z by X. $\mathcal{P}$ is a subset of $\mathcal{C}$ that covers only indices $\{k, \ldots, N-1\}$. The longest match m ending at index $k-1$ and $\mathcal{P}$ form a subset of a MPSC $\mathcal{D}$ of z by X. $\mathcal{P} \cup \{m\} \subseteq \mathcal{D}$.

follows. Instead of two arrays, u, and v, that keep track of position of the sequence with 0 and 1 respectively, keep one three dimensional array w, that keeps track of position of the sequence with each character. Then, $w[i][j][c] = u[i][j]$ in the original formulation if $c = 0$ and $v[i][j]$ if $c = 1$. In general, $w[i][j][c]$ is the position in $a[i+1][j]$ of sequence $a[i][j]$ if it had a c at index i. Minor modifications of query algorithms on the PBWT would run on this data structure with no extra time, however a multiplicative factor of $O(|\Sigma|)$ extra space is taken. We will use the second solution for an arbitrary alphabet PBWT for the rest of this paper. The construction and all vs. all match algorithms for this second solution have been explored in a paper by Naseri et al. [8].

We start with $\mathcal{P} = \emptyset$. Add to $\mathcal{P}$ the positional substring corresponding to the longest match from z to X ending at $N-1$. Every minimal positional substring cover of z by X must cover index $N-1$. Replacing the positional substring in any such cover with the longest match from z to X ending at $N-1$ will yield a minimal positional substring cover of z by X because the sets are the same size and the longest match ending at $N-1$ covers all sites any match ending at $N-1$ covers by definition. Now, $\mathcal{P}$ is a subset of a minimal positional substring cover $\mathcal{C}$ of a string z by X. It covers indices $\{k, \ldots, N-1\}$. For some integer $0 < k \leq N-1$. And none of the indices $\{0, \ldots, k-1\}$. We will show in Lemma 1 that given a subset of a minimal positional substring cover that covers a contiguous section of indices, the longest match ending at the last index it doesn't cover is also in a minimal positional substring cover. Using Lemma 1, we add the positional substring corresponding to the longest match ending at $k-1$ to $\mathcal{P}$. We repeat this with our new $\mathcal{P}$ and new k until $\mathcal{P}$ is a minimal positional substring cover of z by X. Refer to Figure 2 for a depiction of Lemma 1.

▶ **Lemma 1** (Minimal Positional Substring Cover Modularity). *Given a subset $\mathcal{P}$ of a minimal positional substring cover $\mathcal{C}$ of a string z by X that covers all the indices in $\{k, \ldots, N-1\}$ and none of the indices in $\{0, \ldots, k-1\}$ for $0 < k \leq N-1$. Take $m = (i, k-1, s)$, the longest match ending at $k-1$ from z to X. $\mathcal{P} \cup \{m\}$ is a subset of a minimal positional substring cover of z by X.*

Proof. If $\mathcal{P}$ is a subset of a minimal positional substring cover $\mathcal{C}$ of a string z by X, then, the $(|\mathcal{C}| - |\mathcal{P}| - 1)$-th positional substring in $\mathcal{C}$ must cover index $k-1$. If it didn't, either the index $k-1$ is not covered by $\mathcal{C}$, or another substring in $\mathcal{C}$ covers index $k-1$. In the first case, our definition of $\mathcal{C}$ is contradicted.

In the second case, if an n-th positional substring covers $k-1$, if $n > |\mathcal{C}| - |\mathcal{P}| - 1$, our defintion of $\mathcal{P}$ is contradicted. If $n < |\mathcal{C}| - |\mathcal{P}| - 1$, our definition of n-th positional substring is contradicted (n-th positional substring ends after $(|\mathcal{C}| - |\mathcal{P}| - 1)$-th positional substring), or $\mathcal{C}$ is not a minimal positional substring cover ($(|\mathcal{C}| - |\mathcal{P}| - 1)$-th positional substring starts after $k-1$, in which case it can be removed while maintaining the coverage of all sites).

Now, for any $\mathcal{C}$, replacing the $(|\mathcal{C}| - |\mathcal{P}| - 1)$-th positional substring with the longest match ending at $k-1$ of z and X will yield a minimal positional substring cover of z by X. This is because the new set is the same size as $\mathcal{C}$ and all the sites $\mathcal{C}$ covered are also covered by the new set. Any indices greater than $k-1$ are covered by $\mathcal{P}$ and there are no indices less than $k-1$ that the original $(|\mathcal{C}| - |\mathcal{P}| - 1)$-th positional substring covered that the longest match ending at $k-1$ doesn't cover by the definition of longest match ending at $k-1$. ◀

Obtaining the longest match ending at index i is easy using the PBWT. At any index i, for some sequence $a[i+1][j]$, the longest match ending at i between it and $X - \{y_{i+1,j}\}$ starts at the smaller of $d[i+1][j]$ and $d[i+1][j+1]$ (the starting position of the longest match ending at i between $\{a[i+1][j], a[i+1][j-1]\}$ and $\{a[i+1][j], a[i+1][j+1]\}$ respectively. Precisely, the longest match ending at i from z to X is $(\min(d[i+1][j], d[i+1][j+1]), i, z)$. Unfortunately, the query sequence is not in the PBWT, so using this method directly is not possible. In 2021, Sanaullah et al. introduced a method of "virtually inserting" a query haplotype into a PBWT in time linear with respect to the length of the haplotype [10]. This calculates the positions in the prefix array and the divergence values a query string would have if it was present in a PBWT.

3.3 Algorithm

A virtual insertion entails the calculation of the positions the string would take in the prefix arrays a, and the calculation of the new divergence values of the string and the string below it in every prefix array index. The original algorithm was described for binary strings. Here we describe the algorithm on an arbitrary alphabet PBWT. We begin by calculating the locations of the query z in the prefix arrays.

We will keep track of the index $t[k], k \in \{0, \ldots, N\}$. $t[k]$ is the index in the k-th prefix array ($a[k]$) which the query sequence z would be placed above if it were in the PBWT. Choose $t[0]$ arbitrarily, we choose $t[0] = 0$. The position z would be above in $a[k+1]$ is the position $t[k]$ would be at in $a[k+1]$ if it had $z[k]$ at position k. In other words, $t[k+1] = w[k][t[k]][z[k]]$. We can use the w array to calculate each $t[k]$ in constant time. Overall, $O(N)$ where $N = |z|$. Next, we calculate the divergence values.

The key observation for the efficient calculation of divergence values is that the divergence value of the query sequence z at index k will be less than or equal to its divergence value at index $k+1$. The same holds for the divergence value of the sequence below z. See Section 2.3 of [10] for a proof of this claim. Therefore, we calculate two integer arrays, d_z and d_{belowz} of size $N+1$. We calculate $d_z[N]$ and $d_{belowz}[N]$ by starting at $d = N$ and decrementing until $z[d-1] \neq y_{N,t[N]-1}[d-1]$ for d_z (and until $z[d-1] \neq y_{N,t[N]}[d-1]$ for d_{belowz}). Then, we set $k = N-1 \to 0$ and calculate $d_z[k]$ and $d_{belowz}[k]$ by starting at $d_z[k+1]$ and $d_{belowz}[k+1]$ respectively and decrementing until the next characters are not equal, see condition in previous sentence. Overall, this takes $O(N)$ time since every index has a constant cost and over all the sites, the divergence calculation is a counter from N to 0.

As we noted before, obtaining the longest match ending at index k is easy given a PBWT of binary strings. This still holds in our arbitrary alphabet PBWT. The longest match ending at k of $a[k+1][j]$ to $X - \{y_{k+1,j}\}$ will be adjacent to it in $a[k+1]$, either above or below it. For our query z, the longest match ending at k starts at the smaller of $d_z[k+1]$ and $d_{belowz}[k+1]$. Therefore, using Lemma 1 after virtual insertion, we can output a minimal positional substring cover of z by X. Add the longest match ending at $N-1$ $(j, N-1, z)$ to the cover, save $k = j$. Repeat the following until $k = 0$: Add the longest match ending at $k-1$, $(j, k-1, z)$ to the cover, set $k = j$. At the end of this process, we have a minimal positional substring cover of z by X. Refer to Algorithm 1 for the pseudocode of this algorithm. Note that there are some boundary cases in the algorithm not included in the pseudocode, these are discussed in Section 6.

Algorithm 1 Minimal Positional Substring Cover.

```
// Virtual Insertion
t[0] = 0;
for k = 0 → N − 1 do
    t[k + 1] = w[k][t[k]][z[k]];
CURd_z = CURd_belowz = N;
for k = N → 0 do
    CURd_z = min(CURd_z, k);
    CURd_belowz = min(CURd_belowz, k);
    while CURd_z > 0 and z[CURd_z − 1] = y_{k,t[k]−1}[CURd_z − 1] do
        CURd_z − −;
    d_z[k] = CURd_z;
    while CURd_belowz > 0 and z[CURd_belowz − 1] = y_{k,t[k]}[CURd_belowz − 1] do
        CURd_belowz − −;
    d_belowz[k] = CURd_belowz;
// Minimal Positional Substring Cover Output
C = ∅;
k = N;
while k > 0 do
    oldk = k;
    k = min(d_z[k], d_belowz[k]);
    if k = oldk then
        output "No positional substring cover of z by X exists.";
        exit;
    C = C ∪ {(k, oldk − 1, z)};
output C;
```

3.4 Time Complexity

The time complexity of this algorithm is $O(N)$ where N is the length of the query string. The calculation of locations in the prefix array is clearly $O(N)$ since the calculation of each index is constant time and there are N indices. The calculation of divergence values is not constant time per index, however each index incurs a constant cost and a variable cost. Over all N indices, the sum of the variable costs is $O(N)$ since the variable costs are counters from N to 0. Therefore, the virtual insertion part of the algorithm takes $O(N)$ time. The minimal

positional substring cover output section of the algorithm takes constant time per positional substring in the ouput, therefore it takes $O(|\mathcal{C}|)$ time. The size of a minimal substring cover of z by X is at most N because there may not be any empty positional substrings in a minimal positional substring cover. Therefore, given a PBWT of a set of M strings, and a query string z of length N, Algorithm 1 outputs a minimal positional substring cover of z by X in $O(N)$ time.

4 Problem Variants

4.1 Leftmost Minimal Positional Substring Cover

A leftmost minimal positional substring cover $\mathcal{C}$ of z by X is a minimal positional substring cover of z by X with the following property: for any i-th substring in $\mathcal{C}$, it starts at least as early as the i-th substring of every other minimal positional substring cover of z by X. We use i-th positional substring of an MPSC as defined in Section 3.1. With this notation, a minimal positional substring cover $\mathcal{C}$ of z by X is leftmost if $\forall i \in \{0, \ldots, |\mathcal{C}|-1\} \forall \mathcal{D} \in \{\mathcal{P} : \mathcal{P}$ is a minimal positional substring cover of z by $X\}, \pi_1(C[i]) \leq \pi_1(D[i])$.

▷ Claim 5. If the i-th substring in a leftmost minimal positional cover $\mathcal{C}$ of z by X begins at index j, every $(i-1)$-th substring in a minimal positional substring cover of z by X contains index $j-1$. $\forall i \in \{1, \ldots, |\mathcal{C}|-1\} \forall \mathcal{D} \in \{\mathcal{P} : \mathcal{P}$ is a minimal positional substring cover of z by $X\}, \pi_1(\mathcal{D}[i-1]) \leq \pi_1(\mathcal{C}[i]) - 1 \leq \pi_2(\mathcal{D}[i-1])$.

Proof. Suppose there existed an $i \in \{1, \ldots, |\mathcal{C}|-1\}$ and $\mathcal{D} \in \{\mathcal{P} : \mathcal{P}$ is a minimal positional substring cover of z by $X\}$ such that $\pi_1(\mathcal{C}[i]) - 1 > \pi_2(\mathcal{D}[i-1])$. Then, since $\mathcal{D}$ is a cover of z by X, it contains a positional substring that contains index $\pi_1(\mathcal{C}[i]) - 1$, call it $\mathcal{D}[j]$. If $j < i-1$, the definition of i-th positional substring is contradicted since $\pi_2(\mathcal{D}[j]) > \pi_2(\mathcal{D}[i-1]$ and $j < i-1$. If $j > i-1$, $j = i$ by definition of i-th positional substring, and $\mathcal{C}$ is not leftmost since $\pi_1(\mathcal{C}[i]) > \pi_1(\mathcal{D}[i])$. Therefore, no such i and $\mathcal{D}$ exist.

Suppose there existed an $i \in \{1, \ldots, |\mathcal{C}|-1\}$ and $\mathcal{D} \in \{\mathcal{P} : \mathcal{P}$ is a minimal positional substring cover of z by $X\}$ such that $\pi_1(\mathcal{C}[i]) - 1 < \pi_1(\mathcal{D}[i-1])$. Then the set $\{\mathcal{D}[j] : 0 \leq j < i-1\}$ covers the indices $[0, \pi_1(\mathcal{D}[i-1]) - 1]$ with $i-1$ positional substrings and the set $\{\mathcal{C}[j] : i \leq j < |\mathcal{C}|\}$ covers the indices $[\pi_1(\mathcal{C}[i]), N-1]$ with $|\mathcal{C}| - i$ positional substring. Their union is a positional substring cover of z by X with $|\mathcal{C}| - 1$ positional substrings. This contradicts the assumption that $\mathcal{C}$ is a minimal positional substring cover of z by X, therefore, no such i and $\mathcal{D}$ exist. ◁

Here, we will show that the minimal positional substring cover $\mathcal{C}$ of z by X outputted by Algorithm 1 is leftmost. The $(|\mathcal{C}|-1)$-th substring in any MPSC $\mathcal{D}$ of z by X must cover index $N-1$. This is because it is the substring with the greatest ending position in $\mathcal{D}$ (in order for $\mathcal{D}$ to be minimal) and $\mathcal{D}$ must cover index $N-1$. The leftmost starting point of any positional substring that ends at $N-1$ contained in z and a string in X is the starting point of the longest match from z to X ending at index $N-1$. This is exactly the $(|\mathcal{C}|-1)$-th positional substring in $\mathcal{C}$.

Therefore, the set containing only the $\mathcal{C}[|\mathcal{C}|-1]$ is a subset of a leftmost minimal positional substring cover of z by X. By Claim 5, the $(|\mathcal{C}|-2)$-th substring of a leftmost minimal positional substring cover must contain index $\pi_1(\mathcal{C}[|\mathcal{C}|-1]) - 1$. The leftmost starting point of any positional substring that ends at $\pi_1(\mathcal{C}[|\mathcal{C}|-1]) - 1$ contained in z and a string in X is the longest match between z and X ending at that index. This is exactly the $(|\mathcal{C}|-2)$-th

positional substring in $\mathcal{C}$. Therefore, the set $\{\mathcal{C}[|\mathcal{C}|-2], \mathcal{C}[|\mathcal{C}|-1]\}$ is a subset of a leftmost minimal positional substring cover of z by X. This logic can be repeated for every positional substring in $\mathcal{C}$ to show that it is a leftmost minimal positional substring cover of z by X.

4.2 Rightmost Minimal Positional Substring Cover

A rightmost minimal positional substring cover $\mathcal{C}$ of z by X is a minimal positional substring cover of z by X with the following property: for any i-th substrings, it ends at least as late as the i-th substring of every other minimal positional substring cover of z by X. $\forall i \in \{0, \ldots, |\mathcal{C}|-1\} \forall \mathcal{D} \in \{\mathcal{P} : \mathcal{P}$ is a minimal positional substring cover of z by $X\}, \pi_2(\mathcal{C}[i]) \geq \pi_2(\mathcal{D}[i])$.

The logic for obtaining a rightmost minimal positional substring cover is similar to that for a leftmost. One method is to find the leftmost cover using Algorithm 1 of the query $\mathrm{R}(z)$ and $X_{\mathrm{R}} = \{\mathrm{R}(x) : x \in X\}$, then reverse all positional substrings in the outputted cover. This would require building a PBWT on X_{R}, which would take $O(|\Sigma|MN)$ time. Another method is to build it with the longest matches starting at i. The first positional substring the cover $\mathcal{C}$ would be the longest match starting at index 0. The second would be the longest match starting at $\pi_2(\mathcal{C}[0]) + 1$, the third at $\pi_2(\mathcal{C}[1]) + 1$, and so on. The proof for this cover being rightmost is very similar to the leftmost proof.

To obtain the longest match from z to X starting at index k efficiently, create an analogous array to the divergence array for longest match starting at k. Take array δ of length N where $\delta[i] = \min(d_z[i+1], d_{belowz}[i+1])$ for $i \in \{0, \ldots, N-1\}$ (use the definitions of d_z and d_{belowz} from Algorithm 1). Create the analogous array b of length N where $b[j] = \max_{i \in \{k \in \{0,\ldots,N-1\} : \delta[k] \leq j\}} i$ for $j \in \{0, \ldots, N-1\}$. This can be done in $O(N)$ time. Now $b[j]$ corresponds to the end of the longest match starting at j from z to X. Therefore, we can build a rightmost minimal positional substring cover of z by X in $O(N)$ time given a PBWT of X.

4.3 Minimal Positional Substring Cover Using Set Maximal Matches

In haplotype threading, it is usually better to have longer matches when possible. This is because matches that are longer are usually between individuals that are more closely related. In this case, it may be useful to have a MPSC of z by X that is composed of only set maximal matches. Here we provide a brief description of how to output a MPSC of z by X containing only set maximal matches in $O(N)$ time given a PBWT of X.

We begin by running Algorithm 1 on z and the PBWT of X. Call its outputted cover $\mathcal{D}$. All the positional substrings (i, j, s) in $\mathcal{D}$ are longest matches ending at j from z to X, therefore the longest match (m) starting at i from z to X is a set maximal match from z to X. This is because for another match to encompass this match, it needs to contain the indices $\{i, \ldots, j\}$ and start earlier or end later. No match starts earlier because (i, j, s) is the longest match ending at j. No match ends later because m is the longest match starting at i. Therefore, in order to obtain an MPSC containing only set maximal matches, we just have to replace every positional substring $(i, j, s) \in \mathcal{D}$ with the longest match starting at i from z to X. We can do this efficiently using the b array from Section 4.2. $b[i]$ contains the ending point of the largest match starting at i. We can compute the b array in $O(N)$ time. We can replace every match in $\mathcal{D}$ in $O(1)$ time and $|\mathcal{D}| \leq N$. Therefore we can obtain a MPSC composed of only set maximal matches in $O(N)$ time.

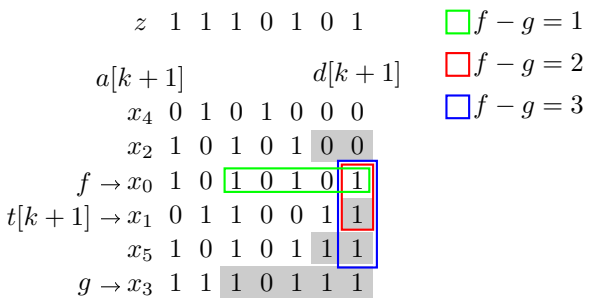


Figure 3 Finding the longest match ending at k present in h strings in X for $k = 6$ and $h = 3$. We only depict indices $\{0, \ldots, k\}$ for all strings. Highlighted in gray are divergence vales, $d[k+1][j]$. f and g values depicted are the final f and g values. The window is depicted for $f - g \in \{1, 2, 3\}$.

5 h-Minimal Positional Substring Cover

The h-Minimal Positional Substring Cover problem, is, given a query string z, and a set of strings X, find the smallest cover out of all positional substrings covers $\mathcal{C}$ of z by X where every positional substring in $\mathcal{C}$ is contained in at least h strings in X. The solution to this problem is similar to the solution to the Minimal Positional Substring Cover problem, and it is biologically useful because the large group of similar individuals for every region suggests that they are closely related. Note that Claims 2–4 hold for h-MPSCs. Furthermore, with this definition, a 1-MPSC is a MPSC of z by X and a MPSC is a 1-MPSC of z by X.

The h-Minimal Positional Substring Cover problem is similar to the Minimal Positional Substring Cover problem. The main difference is that in the new problem, we only consider positional substrings that have h matches in X. Any h-MPSC will contain a positional substring that contains the last index and is contained in h strings in X by definition. We can replace this positional substring with the largest match ending at index $N - 1$ contained in h strings in X. This will result in a set that is still a h-MPSC because it covers all the same sites as the previous set and is the same size. Therefore, we start building a h-MPSC with the longest match ending at index $N - 1$ with h matches. Finding this match is easy using the PBWT.

5.1 Longest Match ending at k present in h strings in X

Once a query string z is virtually inserted into a PBWT, it is easy to find the largest match ending at index k between z and X that matches at least h strings in X. The idea is to keep track of a window in column $k+1$ of the prefix and divergence arrays. We will keep track of the boundaries of the window, $0 \leq f < g \leq N$. f is the index of the first haplotype in the window and g is the index of the first haplotype after f not in the window. We will also keep track of e, the starting position of the match. We start with $e = \min(d_z[k+1], d_{belowz}[k+1])$, $f = t[k+1] - 1$ if $d_z[k+1] < d_{belowz}[k+1]$, otherwise $f = t[k+1]$. Lastly $g = f + 1$. The number of strings in the window at any point is $g - f$. Now, until $g - f = k$, we expand the boundaries of the window to include the next longest match to z. If $d[k+1][f] < d[k+1][g]$, then we decrement f and update e accordingly, $e = \max(d[k+1][f], e)$, $f = f - 1$. Otherwise, we update g and e accordingly, $e = \max(d[k+1][g], e)$, $g = g + 1$ Overall, the search of this match takes $O(h)$ time. Figure 3 depicts this process.

5.2 Algorithm

Then, using similar logic to Lemma 1, we repeatedly add the longest match with h matches ending at the site just before the beginning of the last match to obtain the h-MPSC. See Lemma 2.

Algorithm 2 h-Minimal Positional Substring Cover.

Perform the same steps to virtually insert as in Algorithm 1;
// h-MPSC Output
$\mathcal{C} = \emptyset$;
$k = N$;
while $k > 0$ **do**
  // longest match ending at $k-1$ with h matches in X search
  $e = \min(d_z[k], d_{belowz}[k])$;
  if $d_z[k] < d_{belowz}[k]$ **then**
    $f = t[k] - 1$;
  else
    $f = t[k]$;
  $g = f + 1$;
  while $g - f < k$ **do**
    if $d[k][f] < d[k][g]$ **then**
      $e = \max(d[k][f], e)$;
      $f - -$;
    else
      $e = \max(d[k][g], e)$;
      $g + +$;
  if $e = k$ **then**
    output "No h-positional substring cover of z by X exists.";
    exit;
  $\mathcal{C} = \mathcal{C} \cup \{(e, k-1, z)\}$;
  $k = e$;
output $\mathcal{C}$;

▶ **Lemma 2** (h-MPSC Modularity). *Given a subset $\mathcal{P}$ of a h-minimal positional substring cover $\mathcal{C}$ of a string z by X that covers all the indices in $\{k, \ldots, N-1\}$ and none of the indices in $\{0, \ldots, k-1\}$ for $0 < k \leq N-1$. Take $m = (i, j, s)$, the longest match ending at $k-1$ from z to X that is contained in h strings in X. $\mathcal{P} \cup \{m\}$ is a subset of a h-MPSC of z by X.*

Proof. If $\mathcal{P}$ is a subset of a h-minimal positional substring cover $\mathcal{C}$ of a string z by X, then the $(|\mathcal{C}| - |\mathcal{P}| - 1)$-th positional substring in $\mathcal{C}$ must cover index $k-1$. If it didn't, either the index $k-1$ is not covered by $\mathcal{C}$, or another substring in $\mathcal{C}$ covers index $k-1$. In the first case, our definition of $\mathcal{C}$ is contradicted.

In the second case, if an n-th positional substring of C covers $k-1$, if $n > |\mathcal{C}| - |\mathcal{P}| - 1$, our definition of $\mathcal{P}$ is contradicted. If $n < |\mathcal{C}| - |\mathcal{P}| - 1$,our definition of n-th positional substring is contradicted (n-th positional substring ends after $(|\mathcal{C}| - |\mathcal{P}| - 1)$-th positional substring), or $\mathcal{C}$ is not a minimal positional substring cover ($(|\mathcal{C}| - |\mathcal{P}| - 1)$-th positional substring starts after $k-1$, in which case it can be removed while maintaining the coverage of all sites).

Now, for any $\mathcal{C}$, replacing the $(|\mathcal{C}| - |\mathcal{P}| - 1)$-th positional substring with longest match ending at $k-1$ between z and h strings in X will yield a h-MPSC of z by X. This is because the new set is the same size as $\mathcal{C}$ and all the sites $\mathcal{C}$ covered are also covered by the new set. Any indices greater than $k-1$ are covered by $\mathcal{P}$ and there are no indices less than $k-1$ that the original $(|\mathcal{C}| - |\mathcal{P}| - 1)$-th positional substring covered that the longest match ending at $k-1$ between z and h strings in X doesn't cover by definition. ◀

5.3 Time Complexity

Given a string z, and a PBWT of a set of strings X, we output a h-MPSC of z by X. Finding each positional substring takes $O(h)$ time. Therefore, this algorithm takes $O(h|\mathcal{C}| + N)$ where $\mathcal{C}$ is the outputted cover. The N part of the time complexity comes from the virtual insertion and the $h|\mathcal{C}|$ component from the cover search and output. See Algorithm 2 for the pseudocode of this algorithm.

6 Boundary cases in implementation

In the implementation of the algorithms discussed in this paper, there are boundary cases that need to be accounted for which aren't dealt with in the pseudocode. These boundary cases are left out of the pseudocode because their handling would make the pseudocode unnecessarily long and difficult to understand. We briefly discuss these boundary cases here and how to handle them. For virtual insertion $t[k]$ may equal M, while there is no $a[k][M]$, this represents the fact that z would sort below every string in X in $a[k]$. While there is no value $w[k][M][c]$, it is simple to calculate in constant time what this value should be. $w[k][M][c]$ is the same as $w[k][0][c+1]$, where $c+1$ is the lexicographically smallest character that is larger than c. This can also be calculated in constant time as $w[k][M][c] = w[k][M-1][c]+1$ if $y_{k,M-1}[k] = c$, or $w[k][M][c] = w[k][M-1][c]$ otherwise. It should also be checked if z is longer than the strings in X, if so, no cover exists. Lastly, for the h-MPSC problem, there are two boundary cases. Firstly, if $h > |X|$, no cover exists. Secondly, while incrementally widening the window to contain h strings, care should be taken to avoid trying to obtain the divergence value $d[k][-1]$ or $d[k][M]$. In other words, if $f = 0$ or $g = M$, stop trying to increment f or decrement g respectively.

7 Discussion and Conclusion

In this paper, we have defined and proposed the Minimal Positional Substring Cover problem as a solution to the haplotype threading problem. We proved useful properties of Minimal Positional Substring Covers and provided a solution to the MPSC given a PBWT of X that takes time linear to the length of the query string. We also discussed variations of the Minimal Positional Substring Cover problem: leftmost, rightmost, and set maximal MPSCs. We provided solutions to these problems with the same time complexity as the original solution. Lastly, we discussed the h-MPSC problem, where every positional substring in the cover is contained in h strings in X. We provided a solution to this problem that takes $O(h|\mathcal{C}| + N)$ time given a PBWT of X.

We have laid the ground work for the application of the Minimal Positional Substring Cover problem to haplotype threading. Immediate future work includes the modifications of the MPSC problem that would make a solution more viable for a haplotype threading solution. One such modification is an L-MPSC problem. An L-MPSC is the smallest positional substring cover of z by X that only contains positional substrings of length L or

more. This is biologically useful because small matches can be considered uninformative due to the high likelihood of one occurring by random chance. Therefore, we may want to consider haplotype threading where every match is at least as long as some threshold length L. Another possibly useful variation is the o-MPSC problem. This is similar to the h-MPSC problem except instead of every substring being contained in h strings in X, every index needs to be covered by o overlapping positional substrings in C. The h-MPSC corresponds to finding a group of individuals that are closely related to each other and z for a region while the o-MPSC problem finds a group of individuals that are closely related to z for every index (they may not be closely related to each other). Lastly, a length maximal MPSC solution may be useful. A length maximal MPSC is an MPSC with a largest sum of lengths of positional substrings out of all MPSCs of z by X. A solution to this problem maximizes the overlapping regions on an MPSC, these overlaps are useful because they suggest relatedness between adjacent segments in the MPSC.

One benefit of formulating haplotype threading as a combinatorial problem like MPSC is that it enables the study of its solution space. We presented the leftmost and the rightmost solutions. All other solutions are bounded by these solutions. It may also be useful to design algorithms to derive another solution from an existing solution. E.g., given a solution $\mathcal{C}$, replacing a template $\mathcal{C}[i]$ by another one covering the same gap between $\mathcal{C}[i-1]$ and $\mathcal{C}[i+1]$ would be another solution. Finally, it may be possible to systematically enumerate all solutions. We leave these as future work. We believe efficient solutions to these biologically useful problems exist.

References

1 Olivier Delaneau, Jean-François Zagury, Matthew R Robinson, Jonathan L Marchini, and Emmanouil T Dermitzakis. Accurate, scalable and integrative haplotype estimation. *Nature communications*, 10(1):1–10, 2019.

2 Richard Durbin. Efficient haplotype matching and storage using the positional burrows–wheeler transform (pbwt). *Bioinformatics*, 30(9):1266–1272, 2014.

3 Na Li and Matthew Stephens. Modeling linkage disequilibrium and identifying recombination hotspots using single-nucleotide polymorphism data. *Genetics*, 165(4):2213–2233, 2003.

4 Po-Ru Loh, Petr Danecek, Pier Francesco Palamara, Christian Fuchsberger, Yakir A Reshef, Hilary K Finucane, Sebastian Schoenherr, Lukas Forer, Shane McCarthy, Goncalo R Abecasis, et al. Reference-based phasing using the haplotype reference consortium panel. *Nature genetics*, 48(11):1443–1448, 2016.

5 Gerton Lunter. Haplotype matching in large cohorts using the li and stephens model. *Bioinformatics*, 35(5):798–806, 2019.

6 Ardalan Naseri, Erwin Holzhauser, Degui Zhi, and Shaojie Zhang. Efficient haplotype matching between a query and a panel for genealogical search. *Bioinformatics*, 35(14):i233–i241, 2019.

7 Ardalan Naseri, Xiaoming Liu, Kecong Tang, Shaojie Zhang, and Degui Zhi. Rapid: ultra-fast, powerful, and accurate detection of segments identical by descent (IBD) in biobank-scale cohorts. *Genome biology*, 20(1):1–15, 2019.

8 Ardalan Naseri, Degui Zhi, and Shaojie Zhang. Multi-allelic positional Burrows-Wheeler transform. *BMC bioinformatics*, 20(11):1–8, 2019.

9 Simone Rubinacci, Olivier Delaneau, and Jonathan Marchini. Genotype imputation using the positional burrows wheeler transform. *PLoS genetics*, 16(11):e1009049, 2020.

10 Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. d-PBWT: dynamic positional Burrows-Wheeler transform. *Bioinformatics*, 37(16):2390–2397, 2021.

11 William Yue, Ardalan Naseri, Victor Wang, Pramesh Shakya, Shaojie Zhang, and Degui Zhi. P-smoother: Efficient PBWT smoothing of large haplotype panels. *Bioinformatics Advances*, 2022. doi:10.1093/bioadv/vbac045.

Non-Binary Tree Reconciliation with Endosymbiotic Gene Transfer

Mathieu Gascon
Département d'informatique et de recherche opérationnelle (DIRO), Université de Montréal, Canada

Nadia El-Mabrouk[1]
DIRO, Université de Montréal, Canada

Abstract

Gene transfer between the mitochondrial and nuclear genome of the same species, called endosymbiotic gene transfer (EGT), is a mechanism which has largely shaped gene contents in eukaryotes since a unique ancestral endosymbiotic event know to be at the origin of all mitochondria. The gene tree-species tree reconciliation model has been recently extended to account for EGTs: given a binary gene tree and a binary species tree, the EndoRex software outputs an optimal DLE-Reconciliation, that is an embedding of the gene tree into the species tree inducing a most parsimonious history of Duplications, Losses and EGT events. Here, we provide the first algorithmic study for DLE-Reconciliation in the case of a multifurcated (non-binary) gene tree. We present a general two-steps method: first, ignoring the mitochondrial-nuclear (or 0-1) labeling of leaves, output a binary resolution minimizing the DL-Reconciliation and, for each resolution, assign a known number of 0s and 1s to the leaves in a way minimizing EGT events. While Step 1 corresponds to the well studied non-binary DL-Reconciliation problem, the complexity of the formal label assignment problem related to Step 2 is unknown. Here, we show it is NP-complete even for a single polytomy (non-binary node). We then provide a heuristic which is exact for the unitary cost of operations, and a polynomial-time algorithm for solving a polytomy in the special case where genes are specific to a single genome (mitochondrial or nuclear) in all but one species.

2012 ACM Subject Classification Applied computing → Molecular evolution

Keywords and phrases Reconciliation, Duplication, Endosymbiotic gene transfer, Multifurcated gene tree, Polytomy

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.5

1 Introduction

Since an initial endosymbiotic event integrating an $\alpha-$proteobacterial genome into an eukaryotic cell, which is known to be at the origin of all extent mitochondria, eukaryote evolution has been marked by episodes of gene transfers, mainly from the mitochondria to the nucleus, resulting in a significant reduction of the mitochondrial genome. Understanding how both nuclear and mitochondrial genomes have been shaped by gene loss, duplication and transfer is important to shed light on a number of open questions regarding the origin, evolution, and characteristics of gene coding capacity of eukaryotes.

From a computational point of view, EndoRex [1] is the first algorithm developed for integrating endosymbiotic events (special cases of gene transfers, but only between the mitochondrial and nuclear genome of the same species) in a gene tree - species tree reconciliation model. Given a gene family with gene copies labeled by 0 or 1 depending on whether they are encoded in the mitochondrial or nuclear genome of a given species, a gene tree for the gene family and a species tree for the considered species, EndoRex infers a

[1] Corresponding author

most parsimonious scenario of duplications, losses and endosymbiotic gene transfers (EGT) explaining the gene tree given the species tree. It is an exact polynomial-time algorithm, which can be used to output all minimum cost solutions, for arbitrary costs of operations.

However, as it has been shown for other evolutionary events [6], the result of a reconciliation model strongly depends on the considered trees. For example, due to potential errors in the trees, some of the plant datasets analysed in [1] produced unrealistic evolutionary histories with unexpected high number of gene duplications and losses. A solution would be to ignore weakly supported parts of the tree, leading to a non-binary tree with multifurcated nodes, also called "polytomies", and simultaneously infer a binary refinement and optimal reconciliation of the multifurcated tree, more precisely, infer an optimal evolutionary scenario leading to a binary refinement of the tree. This strategy has been applied, for example, to infer the evolution of the gene families responsible for alkaloid accumulation in plants [11].

Reconciling a non-binary gene tree by minimizing a DL-Reconciliation cost (history with minimum Duplication/Loss cost) has been considered by many authors [2, 4, 10, 9, 12]. As far as we know, the most efficient algorithm is PolytomySolver [9] which handles unit costs in linear time, improves the best complexity of previous algorithms for the general DL cost model by a linear factor and enables to account for various evolutionary rates across the branches of a species tree, attributing to each taxa its specific duplication and loss cost.

In this paper, we explore the multifurcated gene tree reconciliation problem with a reconciliation model accounting for duplications, losses, but also EGT events. Our method is in two steps: ignoring the 0-1 labeling of the leaves, first output all resolutions minimizing the DL-Reconciliation cost and then, for each resolution (i.e. binary tree), assign a known number of 0s and 1s to the leaves in a way minimizing EGT events. As step one can be done efficiently, we then focus on the second step which consists in assigning an optimal 0-1 labeling for the nodes of a binary tree. We show in Section 3 that this problem is NP-complete, even when the multifurcated tree is restricted to a single polytomy. We then, in Section 4, present a general algorithm solving each polytomy separately, which is shown optimal for a unitary cost of operations.

Except for species conserving the traces of an ancestral eukaryotic origin, few genes are expected to reflect an intermediate endosymbiotic integration of the mitochondrial gene content to the nucleus, with gene copies in both the nuclear and mitochondrial genome. This is the case of the eukaryotes with complete mitochondrial genomes explored in [7] (statistics summarized in [1]): among the 2,486 species, only 52 species have mitochondrial-encoded genes also present in the nuclear genome. This motivates Section 5 where we develop a polynomial-time algorithm for the genome labeling problem in the special case where, in each polytomy, genes are specific to a single genome (mitochondrial or nuclear) in all but one species. We first begin, in the next section, by formally defining our problems.

2 Preliminaries, evolutionary model and definitions

All trees are considered rooted. Given a tree T, we denote by $r(T)$ its root, by $V(T)$ its set of nodes and by $L(T) \subseteq V(T)$ its leafset. A node x is a *descendant* of y if x is on the path from y to a leaf of T and an *ancestor* of y if x is on the path from $r(T)$ to y; x is a *strict descendant* (respect. *strict ancestor*) of x' if it is a descendant (respec. ancestor) of x' different from x'. Moreover, x is the *parent* of $y \neq r(T)$ if it directly precedes y on this path. In this latter case, y is a *child* of x. We denote by $E(T)$ the set of edges of T, where an edge is represented by its two terminal nodes (x, y), with x being the parent of y. More generally, if x is an ancestor of y, (x, y) denotes the path between x and y. The subtree of T rooted at

x (i.e. containing all the nodes descendant from x in T) is denoted $T[x]$. The *lowest common ancestor* (LCA) in T of a subset L' of $L(T)$, denoted $lca_T(L')$, is the ancestor common to all the nodes in L' which is the most distant from the root.

An internal node (a node which is not a leaf) is said to be *unary* if it has a single child, *binary* if it has two children, and a *polytomy* if it has more than two children. We will denote by x_l and x_r the two children of a binary node. The node x_l (respec. x_r) is called *the sibling* of x_r (respec. x_l).

A tree R is *an extension* of a tree T if it is obtained from T by *grafting* unary or binary nodes in T, where grafting a unary node x on an edge (u, v) consists in creating a new node x, removing the edge (u, v) and creating two edges (u, x) and (x, v), and in the case of grafting a binary node, also creating a new leaf y and an edge (x, y). In the latter case, we say that y is a *grafted leaf*.

A *species tree* for a set Σ of species is a tree S with a bijection between $L(S)$ and Σ. In this paper, we assume that the species tree S for a given set of species Σ is known, rooted and binary. A *gene family* is a set Γ of genes where each gene $x \in \Gamma$ belongs to a given species $s(x)$ of Σ. A tree G is a *gene tree* for a gene family Γ if its leafset is in bijection with Γ. We write $\langle G, s\rangle$ when each leaf of G is meant to be fully identified by its *species labeling*, i.e. the species $s(x)$ it belongs to (Figure 1. (3) and (4)). For a subset $G \subseteq \Gamma$ of genes, we write $s(G) = \{s(g) : g \in G\}$ as the set of species containing the genes of G. Then the LCA-mapping of G with S is the function assigning to each node x of G the LCA of $s(V(G[x]))$ in S.

In this paper, we will consider an additional *genome labeling* b for a gene x: $b(x) = 0$ if x belongs to the mitochondrial genome of $s(x)$, and $b(x) = 1$ if x belongs to the nuclear genome of $s(x)$. We write $\langle G, s, b\rangle$ when we want to specify that each leaf of G is fully identified by these two labels (Figure 1. (2) and (5)). To summarize, G, $\langle G, s\rangle$ and $\langle G, s, b\rangle$ are three notations for a gene tree, the two last specifying the way the leaves of G are identified.

A *binary tree* is a tree with all internal nodes being binary. If internal nodes have one or two children, then the tree is said *partially binary*. A *multifurcated tree* is a tree containing at least one polytomy. For example, in Figure 1, the tree (2) is a multifurcated tree with two polytomies.

A *binary refinement* of a multifurcated tree is a binary tree defined as follows.

▶ **Definition 1** (binary refinement). *Let $\langle G, s, b\rangle$ be a multifurcated tree. A binary tree $\langle G^B, s^B, b^B\rangle$ is said to be a* binary refinement *of $\langle G, s, b\rangle$ if $V(G) \subseteq V(G^B)$ and for every $x \in V(G)$, $L(\langle G, s, b\rangle[x]) = L(\langle G^B, s^B, b^B\rangle[x])$. We denote by $\mathcal{B}(\langle G, s, b\rangle)$ the set of binary refinements of $\langle G, s, b\rangle$.*

As for a multifurcated tree $\langle G, s\rangle$, a binary refinement $\langle G^B, s^B\rangle$ and the set of binary refinements $\mathcal{B}(\langle G, s\rangle)$ are defined in the same way, just ignoring the b labeling.

For example, in Figure 1, the tree in (3) and (4) is a binary refinement of $\langle G, s\rangle$ (i.e. the tree in (2) ignoring the 0-1 labeling of leaves), and the tree in (5) is a binary refinement of $\langle G, s, b\rangle$.

We need a final notation. Let $X \subseteq L(\langle G, s, b\rangle)$. The *count matrix* $Count(X)$ for X is a $|\Sigma| \times 2$ matrix defined as follows:

$$\begin{cases} Count(X)[\sigma, 0] = & \text{number of genes } g \in X \ / \ s(g) = \sigma \text{ and } b(g) = 0 \\ Count(X)[\sigma, 1] = & \text{number of genes } g \in X \ / \ s(g) = \sigma \text{ and } b(g) = 1 \end{cases}$$

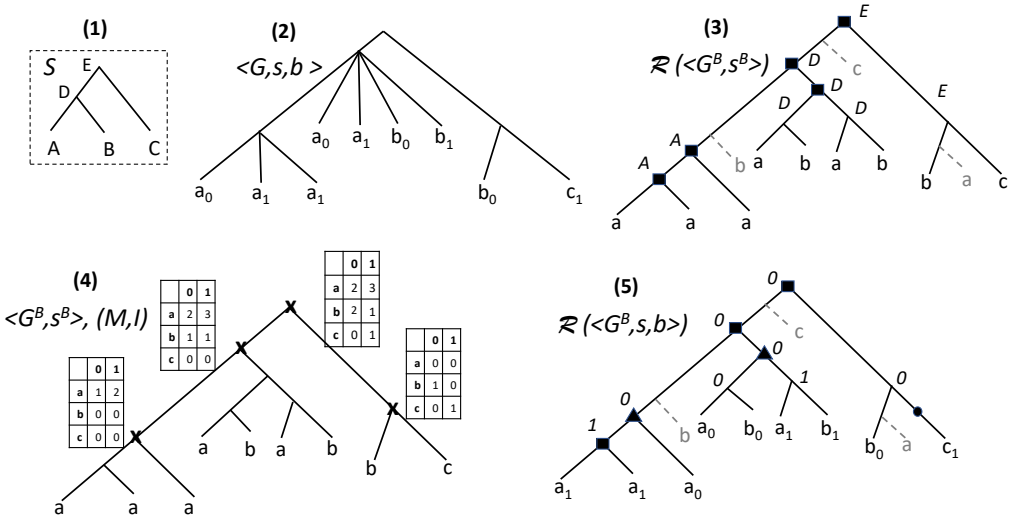


Figure 1 (1) A species tree S on $\Sigma = \{A, B, C\}$; (2) A multifurcated gene tree G where leaves are identified by a species mapping s (a lowercase letter corresponds to the genome identified by the same uppercase letter) and a genome mapping b (the 0-1 index of each leaf); (3) A DL-Reconciliation of the binary refinement $\langle G^B, s^B\rangle$ of $\langle G, s\rangle$. The internal node labeling corresponds to the LCA-mapping with S, squares correspond to duplications and dotted lines to losses (8 events in total). This DL-Reconciliation is optimal for the unitary cost of operations; (4) The tree $\langle G^B, s^B\rangle$ accompanied with an (M, I) b-Constraint, where I is the set of nodes indicated by crosses; (5) A DLE-Reconciliation of $\langle G^B, s^B, b^B\rangle$, where b^B is the genome labeling indicated on leaves, which is consistent with (M, I). The triangle indicates an EGT event and a unary node indicates an EGT-Loss event. For a unitary cost of operations, this reconciliation of cost 9 is optimal for the DLE-BinL problem.

2.1 DLE Reconciliation

Inside the species' genomes, genes undergo *Speciation* (Spe) when the species to which they belong do, but also *Duplication* (Dup) i.e. the creation of a new gene copy, *Loss* of a gene copy, and transfer when a gene is transmitted from a source to a target genome. In this paper, we consider a special case of transfers, called endosymbiotic gene transfers or EGT, only allowing the transmission of genes from the mitochondrial genome to the nuclear genome of the same species, or vice-versa. If the transmission of a gene from a genome A to a genome B is accompanied by the loss of the gene in A, we refer to the event as an *EGT-Loss* ($EGTL$).

Assume that we are given a binary species tree S and a binary gene tree $\langle G, s, b\rangle$. Given an extension R of G, *an extension of* s is a function $\tilde{s}$ from $V(R)$ to $V(S)$ such that, for each leaf x of G, $\tilde{s}(x) = s(x)$. Moreover, an *extension of* b is a function $\tilde{b}$ from $V(R)$ to $\{0, 1\}$ such that, for each leaf x of T, $\tilde{b}(x) = b(x)$. We are now ready to recall the definition of a DLE Reconciliation as introduced in [1].

▶ **Definition 2** (DLE-Reconciliation). *Let $\langle G, s, b\rangle$ be a rooted binary gene tree for a gene family Γ and S be a rooted binary species tree for the species Σ the genes belong to. A* DLE-Reconciliation *of $\langle G, s, b\rangle$ with S (or simply DLE-Reconciliation if no ambiguity) is a quadruplet $\langle R, \tilde{s}, \tilde{b}, e\rangle$ where R is a partially binary extension of G, $\tilde{s}$ is an extension of s, $\tilde{b}$ is an extension of b, and e is an event labeling of the internal nodes of R, such that:*

1. *Each unary node x with a single child y is such that $e(x) = EGTL$, $\tilde{s}(x) = \tilde{s}(y)$ and $\tilde{b}(x) \neq \tilde{b}(y)$; x is an EGT-Loss event with source genome $\sigma_{\tilde{b}(x)}$ and target genome $\sigma_{\tilde{b}(y)}$, where $\sigma = \tilde{s}(x)$ (or equivalently $\tilde{s}(y)$).*
2. *For each binary node x of R with two children x_l and x_r, one of the following cases holds:*
   a. *$\tilde{s}(x_l)$ and $\tilde{s}(x_r)$ are the two children of $\tilde{s}(x)$ in S and $\tilde{b}(x_l) = \tilde{b}(x_r) = \tilde{b}(x)$, in which case $e(x) = Spe$;*
   b. *$\tilde{s}(x_l) = \tilde{s}(x_r) = \tilde{s}(x) = \sigma$ and $\tilde{b}(x_l) = \tilde{b}(x_r) = \tilde{b}(x)$ in which case $e(x) = Dup$ representing a duplication in $\sigma_{\tilde{b}(x)}$;*
   c. *$\tilde{s}(x_l) = \tilde{s}(x_r) = \tilde{s}(x) = \sigma$ and $\tilde{b}(x_l) \neq \tilde{b}(x_r)$ in which case $e(x) = EGT$; let y be the element of $\{x_l, x_r\}$ such that $\tilde{b}(x) \neq \tilde{b}(y)$, then $\tilde{e}(x)$ is a transfer with source genome $\sigma_{\tilde{b}(x)}$ and target genome $\sigma_{\tilde{b}(y)}$.*

Grafted leaves in the extension R correspond to gene losses.

As R is as an extension of G, each node in G has a corresponding node in R. In particular, the $\tilde{s}$, $\tilde{b}$ and e labeling on R induce an $\tilde{s}$, $\tilde{b}$ and e labeling on the nodes of G. The difference between G and R are additional binary nodes with a child being a grafted leaf (a loss), and unary nodes corresponding to EGT-Losses.

A *DL-Reconciliation* of $\langle G, s\rangle$ is defined as in Definition 2, ignoring the binary assignment of genes, i.e. it is a tuple $\langle R, \tilde{s}, e\rangle$ where R is an extension of G.

Optimal reconciliation

Let c be a function attributing a cost to each event in $DLE = \{Spe, Dup, Loss, EGT, EGTL\}$. As it is usually the case, we will assume a 0 cost for speciations and positive costs for all the other events. Moreover, we assume that $c(Dup) \leq c(EGT) + c(EGTL)$ as otherwise duplications would never be inferred in a most parsimonious reconciliation. Similarly, we assume $c(EGT) \leq c(Dup) + c(EGTL)$ to allow for EGTs and $c(EGTL) \leq c(EGT) + c(Loss)$ to allow for EGT-Losses.

Given a DLE-Reconciliation $\mathcal{R} = \langle R, \tilde{s}, \tilde{b}, e\rangle$ (respec. DL-Reconciliation $\langle R, \tilde{s}, e\rangle$), the cost $C(\mathcal{R})$ of $\mathcal{R}$ is the sum of costs of the events labeling the internal nodes of R plus the sum of costs of the losses, i.e. $C(\mathcal{R}) = \sum_{x \in V(R) \setminus L(R)} c(e(x)) + |L(R)_{Loss}| * c(Loss)$ where $|L(R)_{Loss}|$ is the number of losses in $\mathcal{R}$. In this paper, we seek for a most parsimonious reconciliation, i.e. a reconciliation of minimum cost, also called *optimal reconciliation*. We denote by $DLE(G, S)$ (respec. $DL(G, S)$) the cost of an optimal DLE-Reconciliation (respec. DL-Reconciliation).

From now on, we denote by δ, λ, ρ and τ, respectively, the cost of a duplication, a loss, an EGT-loss and an EGT event. The cost function is said to be *unitary* when $\delta = \lambda = \rho = \tau$.

The following lemma makes the link between an optimal DLE-Reconciliation and the optimal DL-Reconciliation. Notice that such optimal DL-Reconciliation is unique [5].

▶ **Lemma 3.** *An optimal DLE-Reconciliation $\mathcal{R}_{DLE} = \langle R_{DLE}, \tilde{s}_{DLE}, \tilde{b}_{DLE}, e_{DLE}\rangle$ of $\langle G, s, b\rangle$ can be obtained from the optimal DL-Reconciliation $\mathcal{R}_{DL} = \langle R_{DL}, \tilde{s}_{DL}, e_{DL}\rangle$ where $\mathcal{R}_{DLE}$ is obtained from $\mathcal{R}_{DL}$ by possibly adding unary nodes (corresponding to EGT-loss), $\tilde{s}_{DLE}$ is an extension of $\tilde{s}_{DL}$ and e_{DLE} is obtained from e_{DL} by labeling unary nodes as EGT-Losses and possibly converting duplications into EGTs.*

Proof. This result follows from Lemma 2 in [1] proven for a unitary cost of operations, but it is easy to see that the proof of lemma 2 can be generalized to a non unitary cost in the case where $\rho \leq \tau + \lambda$, $\tau \leq \delta + \rho$ and $\delta \leq \tau + \rho$ which, as stated above, are the necessary

conditions to ensure that a duplication, EGT and EGT-Loss may be found in an optimal reconciliation. Note that in [1], *EGTcopy* holds for an *EGT* event and *EGTcut* holds for an *EGTL* event. ◀

From now on, we restrict the discussion to DLE-Reconciliations obtained from a DL-Reconciliation, as stated in Lemma 3. Moreover, given a DLE-Reconciliation $\mathcal{R}_{DLE}$, removing an even number of consecutive EGT-Loss nodes can only lead to a more parsimonious DLE-Reconciliation. Therefore, we assume that a reconciliation does not involve such nodes. This assumption is used in the following definition of a compressed reconciliation, aiming at providing a concise representation of a reconciliation, avoiding to represent losses.

▶ **Definition 4** (Compressed reconciliation)**.** *A* compressed DLE-Reconciliation *of* $\langle G, s, b\rangle$ *is a tuple* $\langle G, \tilde{s}, \tilde{b}, e_V, e_E\rangle$ *obtained from a DLE-Reconciliation* $\langle R, \tilde{s}, \tilde{b}, e\rangle$ *of* $\langle G, s, b\rangle$*, where* e_V *is simply* e *restricted to the nodes of* G *and* e_E *is a P/A (Presence/Absence) labeling of the edges of* G *corresponding to the unary (EGT-Loss) nodes of* R*, i.e. obtained as follows: Let* G' *be the tree obtained from* R *by removing grafted leaves and their parental nodes (i.e. ignoring losses). For each edge* (x, y) *of* G*, let* x', y' *be the corresponding nodes in* G' *(*G' *differs from* G *only by unary nodes). Then:*

$$e_E(x,y) = \begin{cases} P & \text{if the path } (x',y') \text{ in } G' \text{ contains an unary node} \\ A & \text{if the path } (x',y') \text{ in } G' \text{ contains no unary node} \end{cases}$$

A *compressed DL-Reconciliation* of $\langle G, s\rangle$ is defined similarly, ignoring the binary assignment of genes. For example, in Figure 1, (3) is a DL-Reconciliation of the gene tree in (4) with the species tree S in (1). The compressed DL-Reconciliation is simply that tree $\mathcal{R}(\langle G^B, s^B\rangle)$ where we ignore losses, i.e. dotted lines. Moreover, (5) is a DLE-Reconciliation, and the compressed DLE-Reconciliation is $\mathcal{R}\langle G^B, s^B, b^B\rangle$ where we ignore losses and replace the unary node (EGT-Loss) on the branch leading to c_1 by a label on that branch.

If $\langle G, \tilde{s}, \tilde{b}, e_V, e_E\rangle$ is a compressed DLE-Reconciliation of an optimal DLE-Reconciliation, then it follows from Lemma 3 that $\tilde{s}$ is the LCA-mapping of G with S. Therefore, from now, we only consider compressed DLE-Reconciliations with $\tilde{s}$ being the LCA-mapping.

For a compressed DLE-Reconciliation $\mathcal{R}^c = \langle G, \tilde{s}, \tilde{b}, e_V, e_E\rangle$ of $\langle G, s, b\rangle$, denote by $|e_{V_{EGT}}|$ the number of EGT nodes, by $|e_E|$ the number of edges labeled P, i.e. the number of EGT-Loss events, and define the cost of $\mathcal{R}^c$ as $C(\mathcal{R}^c) = DL(G, S) + |e_{V_{EGT}}| * (\tau - \delta) + |e_E| * \rho$.

▶ **Lemma 5.** *From a* compressed DLE-Reconciliation $\mathcal{R}^c = \langle G, \tilde{s}, \tilde{b}, e_V, e_E\rangle$ *for* $\langle G, s, b\rangle$*, we can obtain a DLE-Reconciliation* $\mathcal{R}$ *of* $\langle G, s, b\rangle$ *of cost* $C(\mathcal{R}) = C(\mathcal{R}^c)$*.*

Proof. Let $\mathcal{R}^c = \langle G, \tilde{s}, \tilde{b}, e_V, e_E\rangle$ be a compressed DLE-Reconciliation for $\langle G, s, b\rangle$.

Let $\mathcal{R}_{DL} = \langle R_{DL}, \tilde{s}, e_{DL}\rangle$ be the optimal DL-Reconciliation of G with S. We construct a DLE-Reconciliation $\mathcal{R} = \langle R_{DLE}, \tilde{s}_{DLE}, \tilde{b}_{DLE}, e_{DLE}\rangle$ from $\mathcal{R}_{DL}$ and $\mathcal{R}^c$ as follows:

- R_{DLE} is obtained from R_{DL} by grafting a unary node (EGT-Loss) on the edge $(parent(x), x)$ (in R_{DL}) for each node $x \in V(R_{DL}) \cap V(G)$ such that $e_E(parent(x), x) = P$ (in G).
- $\tilde{s}_{DLE}(x)$ is the LCA-mapping of R_{DLE} with S.
- $e_{DLE}(x) = e_{DL}(x)$ for each node $x \in V(R_{DL}) \cap V(R_{DLE})$ and $e_{DLE}(x) = EGTL$ for each unary node of R_{DLE}. For each node $x \in V(G) \cap V(R_{DLE})$, if $e_V(x) = EGT$ then we set $e_{DLE}(x) = EGT$.
- $\tilde{b}_{DLE}(x) = \tilde{b}(x)$ for each node $x \in V(R_{DLE}) \cap V(G)$. For each node $x \in V(R_{DLE}) \setminus V(G)$, let y be the lowest ancestor of x such that $y \in V(R_{DLE}) \cap V(G)$. If y is not an EGT node, then set $\tilde{b}_{DLE}(x) = \tilde{b}(y)$ if there is no EGT-loss event in the path (y, x) (in R_{DLE}), and set $\tilde{b}_{DLE}(x) = 1 - \tilde{b}(y)$ otherwise. Else if y is an EGT node, set $\tilde{b}_{DLE}(x) = \tilde{b}(y)$ if the EGT node y does not transfer in the direction of x and $\tilde{b}_{DLE}(x) = 1 - \tilde{b}(y)$ otherwise.

As $\mathcal{R}$ is constructed from $\mathcal{R}_{DL}$, it is easy to see that the species labeling of the nodes of R_{DLE} is correct. By construction, the genome labeling of the nodes of R_{DLE} is also correct, as the genome labeling $\tilde{b}$ is assumed correct (thus the genome labeling of the nodes $x \in V(R_{DLE}) \cap V(G)$ is correct) and the genome labeling of the nodes $x \in V(R_{DLE}) \setminus V(G)$ is set according to the definition.

Notice that there are $|e_E|$ EGT-loss events and $|e_{V_{EGT}}|$ EGT events in $\mathcal{R}$. Also, the number of loss events in $\mathcal{R}$ is the same as the number of loss events in $\mathcal{R}_{DL}$. Let $|e_{DL_{Dup}}|$ be the number of duplication nodes in the DL-Reconciliation. As an EGT event in $\mathcal{R}$ may only occur on a node that is a duplication in R_{DL}, there are $|e_{DL_{Dup}}| - |e_{V_{EGT}}|$ duplication events in $\mathcal{R}$. Therefore, the cost of $\mathcal{R}$ is: $C(\mathcal{R}) = DL(G,S) + |e_{V_{EGT}}| * (\tau - \delta) + |e_E| * \rho$ ◀

▶ **Corollary 6.** *From an optimal* compressed DLE-Reconciliation $\mathcal{R}^c = \langle G, \tilde{s}, \tilde{b}, e_V, e_E \rangle$*, an optimal DLE-Reconciliation* $\mathcal{R}$ *of* $\langle G, s, b \rangle$ *can be obtained in linear time.*

Proof. For a compressed DLE-Reconciliation $\mathcal{R}^c = \langle G, \tilde{s}, \tilde{b}, e_V, e_E \rangle$, a DLE-Reconciliation leading to $\mathcal{R}^c$, of the same cost as $\mathcal{R}^c$, can be found in linear-time by the constructive proof of Lemma 5. In particular, a DLE-Reconciliation $\mathcal{R}$ can be obtained from an optimal compressed DLE-Reconciliation $\mathcal{R}^c$, and this DLE-Reconciliation $\mathcal{R}$ is necessarily optimal. In fact, from Lemma 3, there is an optimal DLE-Reconciliation $\mathcal{R}_{DLE}$ obtained from the optimal DL-Reconciliation. Then, by construction of $\mathcal{R}_{DLE}$, $C(\mathcal{R}_{DLE}) = DL(G,S) + |e_{V_{EGT}}| * (\tau - \delta) + |e_E| * \rho$, which is also the cost of its compressed DLE-Reconciliation $\mathcal{R}^c_{DLE}$. But as $\mathcal{R}^c$ is optimal, $C(\mathcal{R}^c) \leq C(\mathcal{R}^c_{DLE})$, and thus $C(\mathcal{R}) \leq C(\mathcal{R}_{DLE})$, but as $\mathcal{R}_{DLE}$ is by definition an optimal DLE-Reconciliation, we have $C(\mathcal{R}) = C(\mathcal{R}_{DLE})$ and thus $\mathcal{R}$ is also optimal. ◀

The problem of finding an optimal DLE-Reconciliation is thus equivalent to that of finding an optimal compressed DLE-Reconciliation. From now on, we only consider compressed reconciliations and, for brevity, simply call them reconciliations.

2.2 Problem statements

The general problem of simultaneously refining and reconciling a multifurcated gene tree under the DLE evolutionary model is formulated as follows.

DLE Non-binary Reconciliation problem.

Input: A binary species tree S, a non-binary gene tree $\langle G, s, b \rangle$ and a cost function c on DLE.

Output: An optimal DLE-Reconciliation $\langle G', \tilde{s}', \tilde{b}', e_V, e_E \rangle$ of $\langle G, s, b \rangle$ where $\langle G', s', b' \rangle \in \mathcal{B}(\langle G, s, b \rangle)$.

The **DL Non-binary Reconciliation problem** is simply the restriction of the previous problem to DL-Reconciliation, namely given a non-binary gene tree $\langle G, s \rangle$, the problem is to find a minimum cost DL-Reconciliation $\langle R, \tilde{s}, e \rangle$ of a binary refinement of G. Notice that in this case, R is a binary rather than partially binary tree, as unary nodes only correspond to EGT-Loss events which are not considered in a DL-Reconciliation.

In this paper, we explore a resolution of the DLE Non-binary Reconciliation problem operating in two steps:

Resolution method.

Step 1: Find a binary refinement $\langle G^B, s^B \rangle$ of $\langle G, s \rangle$ leading to an optimal DL-Reconciliation $\langle G^B, \tilde{s}^B, e \rangle$ by solving the DL Non-binary Reconciliation problem.

Step 2: Given $\langle G^B, s^B \rangle$ obtained above, find a genome labeling b^B such that $\langle G^B, s^B, b^B \rangle$ is a binary refinement of $\langle G, s, b \rangle$, leading to an optimal DLE-Reconciliation $\langle G^B, \tilde{s}^{B'}, \tilde{b}^B, e'_V, e_E \rangle$.

Although not guaranteed to be optimal, this method is a natural greedy heuristic for the DLE Non-binary Reconciliation problem. In fact, as stated in Lemma 3, an optimal DLE binary reconciliation (result of Step 2) can be obtained from a DL binary reconciliation (result of Step 1) by simply converting some duplication nodes into EGT nodes and adding EGT-Loss labels on branches. Moreover, Step 1 which consists in solving the DL non-binary Reconciliation problem, can be done efficiently [9]. Having a binary refinement $\langle G^B, s^B\rangle$ of $\langle G, s\rangle$ leading to an optimal DL-Reconciliation, the problem then reduces (Step 2) to finding a genome labeling for G^B allowing for an optimal DLE-Reconciliation which, in the case of a unitary cost, is equivalent to finding a minimum number of added EGT-Loss events.

Notice however that, in contrast to the species labeling s^B, the genome labeling b^B of the leaves of G^B is unknown after Step 1. The problem is therefore not reduced to generalizing b^B to the internal nodes (extending b^B to $\tilde{b}^B$), but consists in finding an appropriate labeling b^B of the leaves as well. Although unknown, this genome labeling of $V(G^B)$ is constrained by the genome labeling of $V(G)$, as formulated in the next lemma which is directly deduced from the definition of a binary refinement (Definition 1).

▶ **Lemma 7.** *Let $\langle G, s, b\rangle$ be a multifurcated tree and $\langle G^B, s^B\rangle$ be a binary refinement of $\langle G, s\rangle$. Then $\langle G^B, s^B, b^B\rangle$ is a binary refinement of $\langle G, s, b\rangle$ if and only if, for any node x of G^B with a corresponding node (also denoted x) in G, $Count(L(\langle G, s, b\rangle[x])) = Count(L(\langle G^B, s^B, b^B\rangle[x]))$.*

Therefore, in addition to $\langle G^B, s^B\rangle$ corresponding to a binary refinement of $\langle G, s\rangle$, the input of Step 2 also includes a set of constraints induced by the genome labeling of $V(G)$. These constraints can be represented as a set of $|\Sigma| \times 2$ matrices $M(x)$ for each $x \in I$, where I is the subset of $V(G^B) \setminus L(G^B)$ with corresponding nodes in $V(G)$. (M, I) is called the b-Constraint for G^B (Figure 1. (4)).

▶ **Definition 8.** *Given a binary tree $\langle G^B, s^B\rangle$ and a b-Constraint labeling (M, I) for G^B, a labeling b^B is said to be* consistent with (M, I) *if, for any $x \in I$, $Count(L(\langle G^B, s^B, b^B\rangle[x]) = M(x)$.*

The main problem (Step 2) can thus be defined as follows (for simplicity, we avoid the "^{B}" notation). See an example in Figure 1 where (4) is the input of the DLE-BinL problem and its output is (5).

DLE Binary Labeling (or DLE-BinL) Problem.

Input: A binary tree $\langle G, s\rangle$, a b-Constraint (M, I) and a species tree S;

Output: An optimal DLE-Reconciliation $\langle G, \tilde{s}, \tilde{b}, e_V, e_E\rangle$ of $\langle G, s, b\rangle$ where b is a genome labeling consistent with (M, I).

We call DLE-BinLR the DLE-BinL problem where I is restricted to the root of G (which corresponds to considering a single polytomy in the initial multifurcated tree).

3 Complexity of the DLE-BinL and DLE-BinLR Problems

We show that the decision versions of both DLE-BinL and DLE-BinLR are NP-complete. In this section, the considered cost is unitary. The DLE-BinL problem in its decision version is defined bellow and DVDLE-BinLR is simply DVDLE-BinL when $I = \{r(G)\}$.

Decision version DLE Binary Labeling (or DVDLE-BinL) Problem.
Input: A binary tree $\langle G, s\rangle$, a b-Constraint (M, I), a species tree S and an integer $Cost$;
Question: Is there a DLE-Reconciliation $\langle G, \tilde{s}, \tilde{b}, e_V, e_E\rangle$ of $\langle G, s, b\rangle$ where b is a genome labeling consistent with (M, I) for which $C(\langle G, \tilde{s}, \tilde{b}, e_V, e_E\rangle) \leq Cost$?

We first show, by reducing from the Monotone not-all-equal 3-satisfiability problem (Monotone NAE3SAT Problem), that the DVDLE-BinLR problem is NP-complete. First observe that the DVDLE-BinLR problem is in NP. In fact, given a DLE-Reconciliation $\langle G, \tilde{s}, \tilde{b}, e_V, e_E\rangle$ of $\langle G, s, b\rangle$, we can calculate the cost of the DLE-Reconciliation (to verify if it is less than or equal to $Cost$) and verify if the genome labeling b is consistent with (M, I) in polynomial time by traversing the tree G.

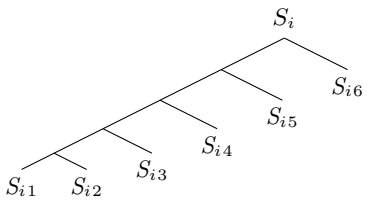
The Monotone NAE3SAT problem is the following (monotone meaning that there are no negation of variables in the clauses).

Monotone NAE3SAT.
Instance: A set of clauses $\mathcal{C} = (C_1 \wedge C_2 \wedge \cdots \wedge C_k)$ on a finite set $Ł = \{\ell_1, \ell_2, \ldots, \ell_m\}$ of variables where each C_i, $1 \leq i \leq k$, is a clause of the form $(x \vee y \vee z)$ with $\{x, y, z\} \subseteq Ł$;
Question: Is there a truth assignment satisfying $\mathcal{C}$ such that the values in each clause are not all equal to each other?

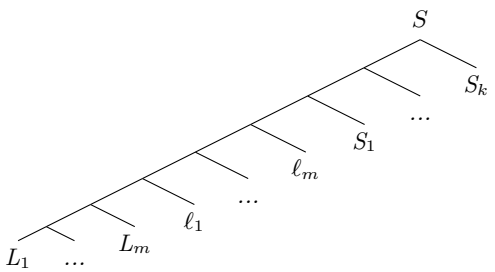
Given an instance $\mathcal{I} = (\mathcal{C}, Ł)$ of the Monotone NAE3SAT problem, we compute, in polynomial time, a corresponding instance $\mathcal{I}' = (\langle G, s\rangle, (M, I), S, Cost)$ of DVDLE-BinLR. First, the set of species Σ is computed as follows:

- For $1 \leq j \leq m$, Σ contains a species ℓ_j and for each clause $C_i \in \mathcal{C}$, $1 \leq i \leq k$ such that ℓ_j is in C_i, Σ contains a species ℓ_{j_i}.
- For each clause $C_i \in \mathcal{C}$, $1 \leq i \leq k$, Σ contains the species S_{i1}, S_{i2}, S_{i3}, S_{i4}, S_{i5} and S_{i6}.

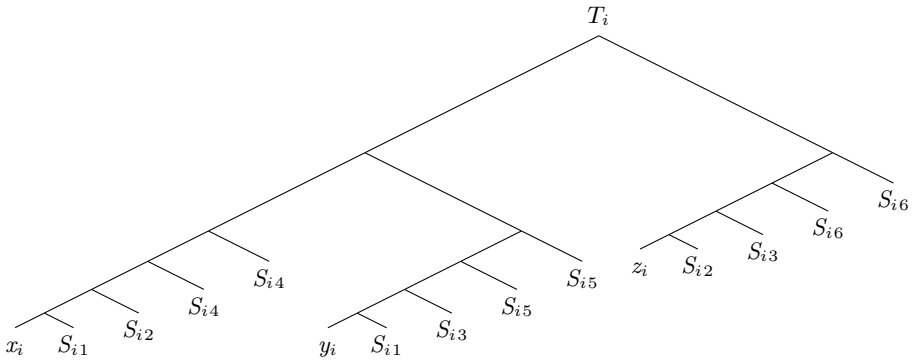
For $1 \leq j \leq m$, let L_j be a caterpillar tree on the leaves ℓ_{j_i} for all i such that ℓ_j is in the clause C_i. For $1 \leq i \leq k$, let S_i be the tree computed as follows:



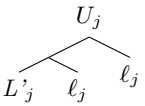
Then, the species tree S is:



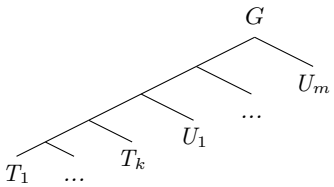
Let now turn to defining the gene tree. For each clause $C_i = (x \vee y \vee z) \in \mathcal{C}$, $1 \leq i \leq k$, let T_i be the following tree:



For $1 \leq j \leq m$, let L'_j be a gene tree which is species label isomorphic to L_j. For $1 \leq j \leq m$, let U_j be the tree computed as follows:



The gene tree G is then:



Notice that for each species $s \in \Sigma$, G contains 2 leaves f such that $s(f) = s$.
We set $M(r(G))$ equal to a matrix of ones of size $|\Sigma| \times 2$. Also recall that $I = \{r(G)\}$. Finally, *Cost* is set to $DL(G, S) + k$.

We next show that $\mathcal{I}$ is a satisfiable instance of the Monotone NAE3SAT problem if (Lemma 9) and only if (Lemma 11) its corresponding instance $\mathcal{I}'$ of DVDLE-BinLR admits a DLE-Reconciliation of cost lower than or equal to *Cost*.

▶ **Lemma 9.** *Let $\mathcal{I}$ be a satisfiable instance of the Monotone NAE3SAT problem. Then its corresponding instance $\mathcal{I}'$ of DVDLE-BinLR admits a DLE-Reconciliation of cost lower than or equal to Cost.*

Proof. See Appendix. ◀

Let $\mathcal{R}'(\langle G, s\rangle, (M, I), S)$ be the optimal DLE-Reconciliation of $\langle G, s, b\rangle$ (with b being a genome labeling consistent with (M, I)) obtained from the optimal DL-Reconciliation of G with S by converting some duplication events into EGT events and by adding some EGT-loss events (this DLE-Reconciliation exists by Lemma 3).

▶ **Lemma 10.** *Let $\mathcal{I}$ be an instance of the Monotone NAE3SAT problem. For its corresponding instance $\mathcal{I}'$ of DVDLE-BinLR, the optimal DLE-Reconciliation $\mathcal{R}'(\langle G, s\rangle, (M, I), S)$ is such that there is at least 1 EGT-loss event in each subtree T_i of G (i.e. $e_E(x, y) = P$ for an edge (x, y) of T_i) for $1 \leq i \leq k$.*

Proof. See Appendix. ◀

▶ **Lemma 11.** *Let $\mathcal{I}$ be an unsatisfiable instance of the* Monotone NAE3SAT *problem. Then its corresponding instance $\mathcal{I}'$ of DVDLE-BinLR does not admit a DLE-Reconciliation of cost equal or lower than Cost.*

Proof. See Appendix. ◀

Note that, by construction, the instance of DVDLE-BinLR in the reduction contains a gene tree with no more than two leaves having the same species label. From this remark, and since Monotone NAE3SAT is NP-complete, lemmas 9 and 11 lead to the following result.

▶ **Theorem 12.** *The DVDLE-BinLR Problem is NP-complete, even if each species label is present at most 2 times in the leaves of the gene tree G.*

As an instance of the DVDLE-BinLR problem is also an instance of the DVDLE-BinL problem, we conclude that the DVDLE-BinL problem is also NP-complete (we can easily show that it is in NP in the same way we showed that the DVDLE-BinLR Problem is in NP).

▶ **Corollary 13.** *The DVDLE-BinL Problem is NP-complete, even if each species label is present at most 2 times in the leaves of the gene tree G.*

4 A general algorithm for the DLE-BinL Problem

A natural heuristic for the DLE-BinL problem for $\langle G, s\rangle$, where G is a binary resolution of an initial multifurcated tree where initial polytomies are reflected by a b-Constraint (M, I), would be to solve each polytomy, i.e. each subtree rooted at a node x of I, individually. In fact, this strategy leads to an exact algorithm for the DL Non-binary Reconciliation problem. However, in the case of DLE-Reconciliation, the $\tilde{b}$ labeling of internal nodes introduces a dependency between polytomies, avoiding the heuristic to be exact in general, i.e. for an arbitrary cost of operations. In this section, we present the general heuristic (Algorithm 1) and show that it is exact in the case of a unitary cost of operations.

Algorithm 1 traverses the tree G in post-order and each time it encounters a node $x \in I$, it "solves" the corresponding subtree $G[x]$ and replaces it by a single leaf, genome labeled appropriately. Once the tree G has been completely traversed, the subtrees are put back in the tree. Notice that on line 13, the algorithm adds a new species to Σ, but does not extend the species labeling $\tilde{s}$ to this new species. The reason is that the new added species is eventually removed from the tree (line 34), i.e. does not remain in the returned reconciliation. Also notice that on line 9, the algorithm adds a new leaf without genome label for which the algorithm will not consider the genome label at any point in the execution. That leaf is also eventually removed from the tree (line 36).

Algorithm 1 calls a function $DLEBinLR(\langle G, s\rangle[x], M(x), S, Bin)$ where $Bin \in \{0, 1\}$, returning an optimal solution of the DLE-BinLR problem such that $\tilde{b}(x) = Bin$. Recall that the DLE-BinLR problem is also NP-complete. In the next section, we will present $DLEBinLR1Species$ which can be substituted to $DLEBinLR$ in Algorithm 1 for a restriction of the problem, where, for each polytomy, genes are b-labeled identically in all but one species.

▶ **Theorem 14.** *Let $\langle G, s\rangle$ be a binary tree, (M, I) be a b-Constraint for $\langle G, s\rangle$, S be a species tree and c be the unitary cost. Then, with the input $(\langle G, s\rangle, (M, I), S)$, Algorithm 1 returns an optimal DLE-Reconciliation of $\langle G, s, b\rangle$ where b is a genome labeling consistent with (M, I).*

Proof. See Appendix. ◀

Algorithm 1 $DLEBinL(\langle G, s\rangle, (M, I), S)$.

```
1  i ← 0
2  for each node x of V(G) \ r(G) in a post-order traversal do
3      M̃(x) ← a zero matrix of size |Σ| × 2
4      if x ∈ I then
5          M'(x) ← M(x) − M̃(x_l) − M̃(x_r)
6          G_{j_0} ← DLEBinLR(⟨G, s⟩[x], M'(x), S, 0)
7          G_{j_1} ← DLEBinLR(⟨G, s⟩[x], M'(x), S, 1)
8          if C(G_{j_0}) == C(G_{j_1}) then
9              Replace the subtree ⟨G, s⟩[x] in G by a new leaf ℓ_i without genome label
10         else
11             label ← arg min_{p∈{0,1}}(C(G_{j_p}));
12             Replace the subtree ⟨G, s⟩[x] in G by a new leaf ℓ_i with s(ℓ_i) ← S_i (where
                S_i is a new species) and b(ℓ_i) ← label;
13             Add the species S_i to Σ;
14             for all x' ∈ I such that x' is a strict ancestor of x do
15                 Add the line [1 − b(ℓ_i), b(ℓ_i)] (corresponding to S_i) to M(x')
16             end
17             for all x'' ∈ I such that x'' is not a strict ancestor of x do
18                 Add the line [0,0] (corresponding to S_i) to M(x'')
19             end
20             if the sibling of x is not in I and has already been visited then
21                 Add the line [0,0] (corresponding to S_i) to M̃(sibling(x))
22             end
23         end
24         M̃(ℓ_i) ← M(x)
25         i ← i + 1
26     else if x is an internal node then
27         M̃(x) ← M̃(x_l) + M̃(x_r)
28     end
29 end
30 M'(r(G)) ← M(r(G)) − M̃(r(G)_l) − M̃(r(G)_r)
31 G ← optimal solution of DLEBinLR(⟨G, s⟩, M'(r(G)), S)
32 for j = i - 1 to 0 do
33     if there is a leaf labeled ℓ_j with a genome label in G then
34         Replace the leaf ℓ_j in G by the tree G_{j_k} where k = b(ℓ_j)
35     else
36         Replace the leaf ℓ_j in G by the tree G_{j_k} where k = b(parent(ℓ_j))
37     end
38 end
39 return ⟨G, s̃, b̃, e_V, e_E⟩
```

5 An exact algorithm for the one-species version of the DLE-BinLR Problem

We consider a restriction of the DLE-BinLR problem where genes are specific to a single genome (the mitochondrial or nuclear genome) in all but one species. In its simplest version where a single species is present, the problem reduces to assigning a multiset of two labels

(a given number of 0s and a given number of 1s) to the leaves of a tree-shape (i.e. a tree with no leaf labels), in a way minimizing 0-1 transitions in the tree. Similar problems on assigning leaves to tree-shapes or to multilabeled trees (MUL-trees) have been considered in the context of other tree distances (Robinson Foulds distance, path distance, maximum agreement subtree), most of them being NP-complete [3, 8]. Here we present an exact polynomial-time algorithm for this restricted version of the DLE-BinLR problem, which we call the DLE-BinLR1Species problem.

Let $\sigma \in \Sigma$ be the only species for which the genes belonging to it are not specific to a single genome. We will call the leaves $\ell \in L(G)$ for which $s(\ell) = \sigma$ *free leaves* and the leaves $\ell \in L(G)$ for which $s(\ell) \neq \sigma$ *fixed leaves*. For a fixed leaf ℓ, $b(\ell)$ is fixed and known in advance, as all leaves whose species label is $s(\ell)$ have the same genome label which is known from the matrix M. The DLE-BinLR1Species problem is then reduced to finding an optimal DLE-Reconciliation for which exactly k free leaves are labeled by 0, where $k = M(r(G))[\sigma, 0]$.

Let $\mathcal{R}_{DL} = \langle G, \tilde{s}, e\rangle$ be an optimal DL-Reconciliation for $\langle G, s\rangle$. From Lemma 3, an optimal DLE-Reconciliation $\mathcal{R}_{DLE} = \langle G, \tilde{s}, \tilde{b}, e_V, e_E\rangle$ with exactly k free leaves labeled by 0 can be obtained from $\mathcal{R}_{DL}$ by converting some duplications into EGTs and adding EGT-Loss events, i.e. a P/A labeling on edges. We define $minCostTransfer(\langle G, \tilde{s}, \tilde{b}, e_V, e_E\rangle) = |e_{V_{EGT}}| * (\tau - \delta) + |e_E| * \rho$. Then recall from Section 2 that, by construction of $\mathcal{R}_{DLE}$, we have:

$$C(\mathcal{R}_{DLE}) = DL(G, S) + minCostTransfer(\langle G, \tilde{s}, \tilde{b}, e_V, e_E\rangle)$$

The problem thus reduces to minimizing $minCostTransfer(\langle G, \tilde{s}, \tilde{b}, e_V, e_E\rangle)$.

We will need to consider the two possible genome labeling $i \in \{0, 1\}$ for the root of G. We therefore denote by $minCostTransfer(\langle G, \tilde{s}, e\rangle, k, i)$ the $minCostTransfer$ function for an optimal DLE-Reconciliation $\mathcal{R}_{DLE}$ with exactly k free leaves labeled by 0 and with the additional constraint that $\tilde{b}(r(G)) = i$.

We are now ready to present Algorithm 2. It proceeds in two phases: (1) a bottom-up phase (Algorithm 3) in which we assign an array of size $2 \times (k + 1)$ to each node x of G where the (i, j)th entry equals $minCostTransfer(\langle G[x], \tilde{s}, e\rangle, j, i)$ where $\langle G[x], \tilde{s}, e\rangle$ is the optimal DL-Reconciliation of $G[x]$ with S; (2) a top-down phase (not given in pseudo-code) in which the algorithm assigns the $\tilde{b}$ labeling of nodes and locates the EGT and EGT-Loss events in the optimal solution. For this purpose, for each entry of $x.array$ of each internal node x, the bottom-up algorithm keeps in memory pointers to the entries of the arrays of the children of x from which the value of the entry was obtained.

▶ **Theorem 15.** *The output of Algorithm 2 is a solution of the DLE-BinLR1Species problem.*

Proof. See Appendix. ◀

▶ **Theorem 16.** *Algorithm 2 computes the solution of the DLE-BinLR1Species problem in $O(nk^2)$ time, where n is the number of leaves of G.*

Proof. For each leaves of G, the associated array is computed in time $O(k)$. For each internal node of G, the associated array is computed in time $O(k^2)$. The time complexity to compute the arrays for all the nodes is then $O(nk^2)$.

Once all the arrays are computed, the algorithm finds the optimal assignation of the internal nodes with a preorder traversal of G in time $O(n)$

We conclude that the time complexity of Algorithm 2 is $O(nk^2)$. ◀

Algorithm 2 $BinLR1Species(\langle G, s\rangle, (M, I), S)$.

```
k ← M(r(G))[σ, 0]
⟨G, s̃_DL, e_DL⟩ ← Optimal DL-Reconciliation of ⟨G, s⟩ with S
Bottom − up(⟨G, s⟩, e_DL, k)
Top − down(⟨G, s⟩, e_DL, k)
return ⟨G, s̃, b̃, e_V, e_E⟩
```

Algorithm 3 $Bottom - up(\langle G, s\rangle, e, k)$.

```
for each node x of G in a post-order traversal do
    x.array ← Array of size 2 × (k + 1)
    if x is a leaf then
        if x is fixed to "0" then
            x.array(0, 0) ← 0
            x.array(i, j) ← ∞ for every values of (i, j) ≠ (0, 0)
        else if x is fixed to "1" then
            x.array(1, 0) ← 0
            x.array(i, j) ← ∞ for every values of (i, j) ≠ (1, 0)
        // Case where x is a free leaf
        else
            x.array(0, 1) ← 0
            x.array(1, 0) ← 0
            x.array(i, j) ← ∞ for every values of (i, j) ≠ (1, 0) and (i, j) ≠ (0, 1)
        end
    else
        for j = 0 to k do
            T_00, T_01, T_10, T_11 ← Arrays of size (j + 1)
            for ℓ = 0 to j do
                // ℓ is the number of free leaves labeled "0" under x_l and
                    j − ℓ is the number of free leaves labeled "0" under x_r
                T_00(ℓ) ← x_l.array(0, ℓ) + x_r.array(0, j − ℓ)
                T_01(ℓ) ← x_l.array(0, ℓ) + x_r.array(1, j − ℓ)
                T_10(ℓ) ← x_l.array(1, ℓ) + x_r.array(0, j − ℓ)
                T_11(ℓ) ← x_l.array(1, ℓ) + x_r.array(1, j − ℓ)
            end
            // Cost of the first transfer
            cost ← ((e(x) == Dup) ? τ − δ : ρ)
            // Case where x is labeled "0"
            x.array(0, j) ←
              min(min(T_00), cost + min(T_01), cost + min(T_10), cost + ρ + min(T_11))
            // Case where x is labeled "1"
            x.array(1, j) ←
              min(cost + ρ + min(T_00), cost + min(T_01), cost + min(T_10), min(T_11))
        end
    end
end
```

See Figure 2 for an example of an execution of Algorithm 2.

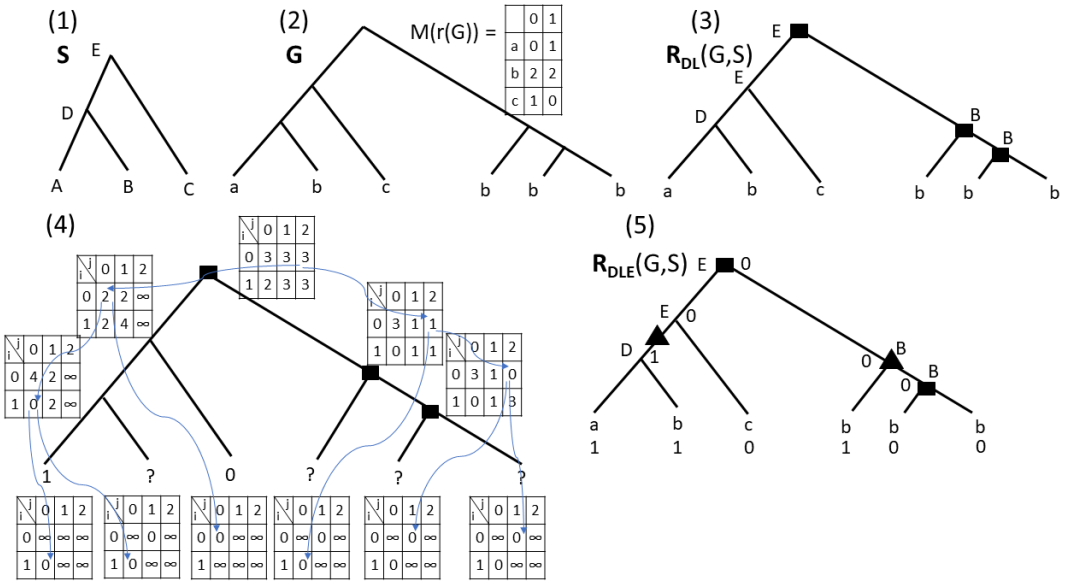


Figure 2 (1) A species tree S on $\Sigma = \{A, B, C\}$; (2) A binary gene tree G where leaves are identified by a species mapping s, and a b-Constraint (M, I) where $I = r(G)$; (3) An optimal DL-Reconciliation of G with S; (4) The tree G accompanied with the arrays computed by Algorithm 3 (we consider here the costs $\delta = \lambda = 1$ and $\rho = \tau = 2$) and the pointers for an optimal solution; (5) The optimal DLE-Reconciliation $\mathcal{R}_{DLE}(G, S)$ of $\langle G, s, b\rangle$ (where b is consistent with (M, I)) returned by Algorithm 2. The cost $minCostTransfer(\mathcal{R}_{DLE}(G, S))$ is 3. Events are represented as in Figure 1.

6 Conclusion

Endosymbiotic gene transfers (EGTs) are important events to be considered in a reconciliation model aiming to infer the evolution of a gene family, given a gene tree for the gene family and a species tree for the species containing the genes. As it is usually difficult, or impossible, to infer a well supported binary tree based on sequence data, it is also important to be able to account for non-binary gene trees. In this paper, we present the first method for DLE reconciliation, i.e. reconciliation accounting for duplications, losses, but also EGTs, for a multifurcated gene tree. It is a natural extension of the DL reconciliation of a multifurcated tree, where we first consider a solution for this problem, i.e. an optimal DL-Reconciliation, which can be obtained efficiently, and then appropriately assign the binary genome labeling (0/1 for mitochondrial/nuclear) to the nodes of the tree, accounting for EGT transfers, in a way minimizing a total DLE (Duplications, Losses and EGT) cost.

We show that the optimal genome labeling assignment step is NP-complete for an arbitrary binary refinement, even for a single polytomy, and even when genes are present in only two copies in each species. We then present two natural heuristics for the general and one-polytomy versions of the problem which are shown to be exact for some restrictions on the model (unitary cost of operations and/or free leaves belonging to a single genome). As explained in the introduction, we argue that these restrictions are biologically relevant. The next step will be to apply our method to the orthologous mitochondrial protein-coding genes (MitoCOGs) dataset [1, 7].

From a theoretical and algorithmic point of view, which is the focus of this paper, many open questions remain. Apart from the fact that a heuristic combining accuracy and time-efficiency should be developed for both the DLE-BinL and DLE-BinLR problems in the

case of a general cost function and an arbitrary number of species presenting an intermediate endosymbiotic integration, a more fundamental question is whether an exact one-step method, considering all the events at once, can be developed. In fact, the complexity results obtained here do not allow to conclude on the complexity of the DLE Non-binary Reconciliation problem. It is indeed not excluded that the polynomial-time PolytomySolver algorithm [9] can be extended for solving a multifurcated tree with a binary labeling of leaves, at least in special cases. In the near future, we will first explore the extension of PolytomySolver to the one species restriction of the model, before considering generalization to an arbitrary number of species.

References

1 Y. Anselmetti, N. El-Mabrouk, M. Lafond, and A. Ouangraoua. Gene tree and species tree reconciliation with endosymbiotic gene transfer. *Bioinformatics*, 37(SI-1):i120–i132, 2021.

2 W.C. Chang and O. Eulenstein. Reconciling gene trees with apparent polytomies. In D.Z. Chen and D. T. Lee, editors, *Proceedings of the 12th Conference on Computing and Combinatorics (COCOON)*, volume 4112 of *Lecture Notes in Computer Science*, pages 235–244, 2006.

3 C. Colijn and G. Plazzotta. A metric on phylogenetic tree shapes. *Systematic Biology*, 67(1):113–126, 2018.

4 D. Durand, B.V. Haldórsson, and B. Vernot. A hybrid micro-macroevolutionary approach to gene tree reconstruction. *Journal of Computational Biology*, 13:320–335, 2006.

5 N. El-Mabrouk and E. Noutahi. *Bioinformatics and Phylogenetics: Seminal Contributions of Bernard Moret*, chapter Gene Family Evolution – An Algorithmic Framework, pages 87–119. Springer International Publishing, t. warnow edition, 2019.

6 M.W. Hahn. Bias in phylogenetic tree reconciliation methods: implications for vertebrate genome evolution. *Genome Biology*, 8(7):R141, 2007.

7 S. Kannan, I. Rogozin, and E. Koonin. MitoCOGs: clusters of orthologous genes from mitochondria and implications for the evolution of eukaryotes. *BMC Evolutionary Biology*, 14(11):1–16, 2014.

8 M. Lafond, N. El-Mabrouk, K.T. Huber, and V. Moulton. The complexity of comparing mutiply-labelled trees by extending phylogenetic-tree metrics. *Theoretical Computer Science*, 760:15–34, 2018.

9 M. Lafond, E. Noutahi, and N. El-Mabrouk. Efficient Non-Binary Gene Tree Resolution with Weighted Reconciliation Cost. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:12, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

10 M. Lafond, K.M. Swenson, and N. El-Mabrouk. An optimal reconciliation algorithm for gene trees with polytomies. In *LNCS*, volume 7534 of *WABI*, pages 106–122, 2012.

11 J. Sabir, R. Jansen, D. Arasappan, et al. The nuclear genome of Rhazya stricta and the evolution of alkaloid diversity in a medically relevant clade of Apocynaceae. *Scientific Reports*, 6(1):33782, 2007.

12 Y. Zheng and L. Zhang. Reconciliation with nonbinary gene trees revisited. *J. ACM*, 64(4), August 2017.

A Appendix

Proof of Lemma 9

Proof. Let $R_{DL} = \langle G, \tilde{s}, e \rangle$ be the optimal DL-Reconciliation of G with S. We recall that, by definition, $C(R_{DL}) = DL(G, S)$. We will show that we can obtain a DLE-Reconciliation R_{DLE} of cost lower than or equal to $Cost$ from R_{DL} by converting some duplication events

into EGT events and by adding EGT-Loss events. Notice that because the costs are unitary, converting a duplication event into an EGT event does not change the cost of the reconciliation. Thus, the cost of R_{DLE} is $C(R_{DL})$ plus the number of EGT-loss events in R_{DLE}.

Let TA be a truth assignment satisfying $\mathcal{C}$ such that the values in each clause are not all equal to each other (we know that such truth assignment exists because $\mathcal{I}$ is a satisfiable instance).

We now construct the genome labeling $\tilde{b}$ (and b) and the mappings e_V and e_E as follows:

Let $e_V = e$. Let $e_E(x, y) = A$ for all edge (x, y) of G.

For all j, $1 \leq j \leq m$, such that ℓ_j is True (resp. False) in TA, we set $\tilde{b}(x) = 1$ (resp. $\tilde{b}(x) = 0$) for each nodes x of the left subtree of U_j.

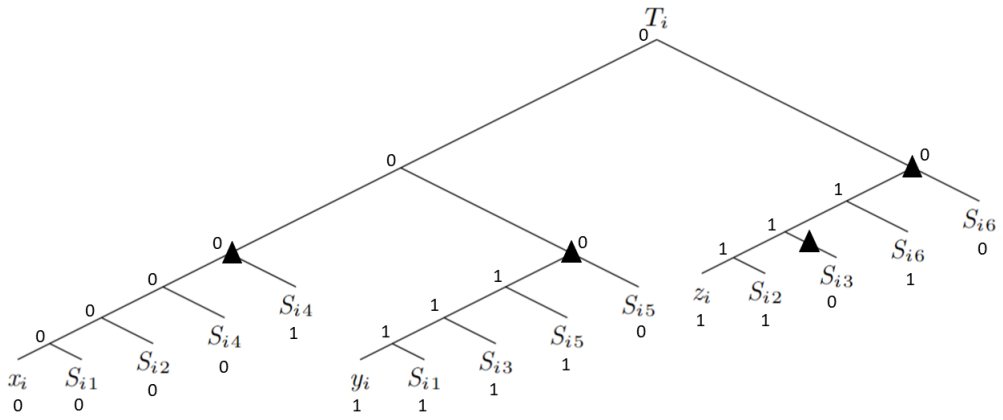
Notice that for each $\sigma \in \Sigma \setminus \{S_{ij} | 1 \leq i \leq k, 1 \leq j \leq 6\}$, we have set the genome label of exactly one of the two leaves of G for which the species label is σ. For each $\sigma \in \Sigma \setminus \{S_{ij} | 1 \leq i \leq k, 1 \leq j \leq 6\}$, we then set the genome label of the leaf with species label σ whose genome label have not been set yet to $1 - i$ where i is the genome label of the other leaf with species label σ.

For each nodes x on the path from the parent of $r(T_1)$ to $r(G)$, we set $\tilde{b}(x) = 0$. We set $\tilde{b}(r(T_i)) = 0$ for $1 \leq i \leq k$ and we set $\tilde{b}(r(U_j)) = 0$ for $1 \leq j \leq m$.

Therefore, there is no EGT-loss event on edges that are not in the subtrees U_j $(1 \leq j \leq m)$ or T_i $(1 \leq i \leq k)$, as all the nodes connected by those edges are labeled by 0.

We now show that no EGT-loss event is required in the subtree U_j of G, for $1 \leq j \leq m$. by construction, all the nodes in the left subtree of U_j have the same genome label i $(i \in \{0, 1\})$ and the node in the right subtree of U_j has the genome label $1-i$. Thus, $\tilde{b}(r(U_j)_l) \neq \tilde{b}(r(U_j)_r)$. Notice that $r(U_j)$ is a duplication node in R_{DL} and recall that $\tilde{b}(r(U_j)) = 0$. We then set $e_V(r(U_j)) = EGT$ which is a transfer from 0 to 1. Therefore, there is no EGT-loss event in the subtree U_j.

We now show that exactly one EGT-loss event is required in the subtree T_i of G, for $1 \leq i \leq k$. Notice that for a clause $C_i = (x \vee y \vee z) \in \mathcal{C}$, x, y and z can't be all equal to each other in TA (because TA is a solution of the instance) and so, by construction, the genome labels of x_i, y_i and z_i in T_i are not all equal to each other. Without loss of generality, let assume that $\tilde{b}(x_i) = 0$, $\tilde{b}(y_i) = 1$ and $\tilde{b}(z_i) = 1$ (the other possible cases are very similar). Then, the following genome labeling $\tilde{b}$ of T_i is correct and requires exactly one EGT-loss event:



We set $e_E(x, y) = P$ where (x, y) is the edge with a triangle on it in the tree above. We also set $e'(lca_{T_i}(\{x_i, S_{i4}\})) = EGT$, $e'(lca_{T_i}(\{y_i, S_{i5}\})) = EGT$ and $e'(lca_{T_i}(\{z_i, S_{i6}\})) = EGT$ (those are the nodes represented by a triangle in the tree above). We can do so because those nodes are duplication nodes in R_{DL}.

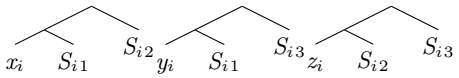
There is then exactly k EGT-loss events in R_{DLE}. Thus, the cost of R_{DLE} is $DL(G,S)+k$ and $C(R_{DLE}) \leq Cost$.

For each leaf x of G, we set $b(x) = \tilde{b}(x)$. Notice that the genome labeling b we construct is consistent with (M,I) as for each $\sigma \in \Sigma$, there is one leaf labeled σ whose genome label is 1 and one leaf labeled σ whose genome label is 0, as needed.

We then obtain a DLE-Reconciliation $R_{DLE} = \langle G, \tilde{s}, \tilde{b}, e_V, e_E \rangle$ of $\langle G, s, b \rangle$ where b is a genome labeling consistent with (M,I) for which $C(R_{DLE}) \leq Cost$ and we conclude that the instance $\mathcal{I}'$ of DVDLE-BinLR admits a DLE-Reconciliation of cost lower than or equal to $Cost$. ◀

Proof of Lemma 10

Proof. For the optimal DLE-Reconciliation $\mathcal{R} = \mathcal{R}'(\langle G, s \rangle, (M,I), S)$, for each clause $C_i = (x \vee y \vee z) \in \mathcal{C}$, $1 \leq i \leq k$, for any genome labeling labeling b consistent with (M,I), there will be at least one EGT-loss event in the three following subtrees of T_i (regardless of the labeling $\tilde{b}$ of the internal nodes of these subtrees):



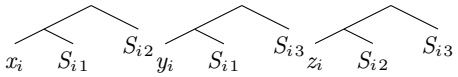
This is the case because there are no duplication node in the DL reconciliation of these subtrees with S (so no EGT events can occur in these subtrees by construction of $\mathcal{R}$) and we know that at least one of these subtrees will not have all its leaves labeled by the same genome label (because two leaves with the same species label can't have the same genome label by construction of the instance) so at least one EGT-loss will be required. ◀

Proof of Lemma 11

Proof. By contradiction, let us suppose that for an unsatisfiable instance $\mathcal{I}$ of the Monotone NAE3SAT problem, its corresponding instance $\mathcal{I}'$ of DVDLE-BinLR does admit a compressed DLE-Reconciliation of cost equal or lower than $Cost$. Let $\mathcal{R} = \mathcal{R}'(\langle G, s \rangle, (M,I), S)$. By lemma 3, $\mathcal{R}$ is optimal and thus $C(\mathcal{R}) \leq DL(G,S) + k$ as $\mathcal{I}'$ does admit a compressed DLE-Reconciliation of cost equal or lower than $Cost = DL(G,S) + k$. By lemma 10, $\mathcal{R}$ is such that there is at least 1 EGT-loss event in each subtree T_i of G for $1 \leq i \leq k$. There is then at least k EGT-loss events in the reconciliation $\mathcal{R}$. As the cost of $\mathcal{R}$ is equal to $DL(G,S)$ plus the number of EGT-loss events in $\mathcal{R}$ (from lemma 4 in [1]), $C(\mathcal{R})$ must be higher than or equal to $DL(G,S)+k$ and we conclude that $C(\mathcal{R}) = DL(G,S)+k$. Thus, there is exactly one EGT-loss event in each subtree T_i of G for $1 \leq i \leq k$ and no EGT-loss event elsewhere in the tree as otherwise $C(\mathcal{R})$ would be higher than $DL(G,S) + k$. In particular, there is no EGT-loss event in the subtrees U_j, $1 \leq j \leq m$, and we can conclude that all nodes in the subtree L'_j, $1 \leq j \leq m$, have the same genome label (there is no EGT event in the subtree L'_j as there is no duplication in the DL-Reconciliation of L'_j with S).

We now define a truth assignment TA as follows: for all $1 \leq j \leq m$, let the variable ℓ_j be true if the genome label of the nodes in L'_j is 1, and let the variable ℓ_j be false otherwise. We now show that TA satisfies $\mathcal{I}$. For each clause $C_i = (x \vee y \vee z) \in \mathcal{C}$, $1 \leq i \leq k$, we need to show that x, y and z are not all equal to each other. Let us suppose by contradiction that this is false, and that there exists a clause $C_i = (x \vee y \vee z) \in \mathcal{C}$ such that x, y and z are all equal to each other. Then, by construction, the genome labels of the leaves x_i, y_i and z_i in the corresponding subtrees T_i are all equal to each other. Then, there is at least 2

EGT-losses events in T_i, as at least two of the following three subtrees of T_i will not have all their leaves labeled by the same genome label and there are no EGT events in those subtrees (by construction) because there are no duplication node in the DL reconciliation of these subtrees with S (this is easy to verify):



This is a contradiction, as there must be exactly one EGT-loss event in the subtree T_i. We then conclude that for each clause $C_i = (x \vee y \vee z) \in \mathcal{C}$, $1 \leq i \leq k$, x, y and z are not all equal to each other. Thus, the truth assignment TA satisfies $\mathcal{I}$, and we conclude by contradiction that if $\mathcal{I}$ is an unsatisfiable instance of the Monotone NAE3SAT problem, then its corresponding instance $\mathcal{I}'$ of DVDLE-BinLR does not admit a compressed DLE-Reconciliation of cost equal or lower than $Cost$. ◀

Proof of Theorem 14

Proof. The proof is by induction on the number of node $x \in V(G)$ such that $x \in I$.

Notice that the DLE-Reconciliation $\langle G, s, b\rangle$ returned by Algorithm 1 is such that b is a genome labeling consistent with (M, I) by construction.

If there is only one node $x \in V(G)$ such that $x \in I$, then this node x is the root of G by definition. The algorithm then returns an optimal solution, as assume that we can solve $DLEBinLR(\langle G, s\rangle, M'(r(G)), S, i)$ (where $M'(r(G)) = M(r(G))$) for $i \in \{0, 1\}$.

If there is more than one node $x \in V(G)$ such that $x \in I$, then the root of G is in I by definition. By induction, we may assume that for each node $x \in V(G) \setminus r(G)$ such that $x \in I$, the resolution of $G[x]$ computed by the algorithm is exact. For each of those subtrees $G[x]$, we then know the possible genome label(s) at the root leading to an optimal resolution of $G[x]$ and the corresponding optimal resolution of $G[x]$. We now give the index 1 to $|I| - 1$ to the elements of $I \setminus r(G)$. For all $1 \leq j \leq |I| - 1$, there is then two cases for $x_j \in I \setminus r(G)$:

Case 1. $G[x_j]$ is such that both $\tilde{b}(x_j) = 0$ and $\tilde{b}(x_j) = 1$ can lead to an optimal resolution of $G[x_j]$.

In that case, Algorithm 1 will remove $G[x_j]$ from G and replace it by a new leaf without genome label. It solves $G(x_j)$ separately and then replace the new leaf in G by the solved $G[x_j]$ (after the rest of G is solved). $G[x_j]$ can be solved separately in that case, because regardless of the genome label of the parent of $G[x_j]$ in an optimal resolution of (the rest of) G we can obtain an optimal resolution of $G[x_j]$ with $r(G[x_j])$ having the same genome label as its parent (and thus we can obtain an optimal solution to the problem by putting the solved $G[x_j]$ with $r(G[x_j])$ having the same genome label as its parent back in G).

Case 2. $G[x_j]$ is such that only $\tilde{b}(x_j) = i_j$ (where $i_j \in \{0, 1\}$) can lead to an optimal resolution of $G[x_j]$. In that case, Algorithm 1 will remove $G[x_j]$ from G and replace it by a new leaf labeled by i.

Then, Algorithm 1 solves $DLEBinLR(\langle G', s\rangle, M'(r(G)), S)$ where G' is the tree obtained by removing $G[x_j]$ for x_j in Case 1, and by replacing $G[x_j]$ for x_j in Case 2 by a new leaf with the appropriate genome labeling and where $M'(r(G))$ is the appropriate matrix of constraints for the genome labeling of the leaves of G'. By construction, it will then return the solution of lowest cost such that $\tilde{b}(x_j) = i_j$, for all x_j in Case 2.

Let's show that this solution is optimal. By contradiction, suppose that there is $x_j \in I$ (x_j in Case 2) such that there is no optimal solution of the problem for which $\tilde{b}(x_j) = i_j$. Then, the optimal solution $\mathcal{R}^*$ of the problem is such that $\tilde{b}(x_j) \neq i_j$. In $\mathcal{R}^*$, if we set $\tilde{b}(x_j) = i_j$ and replace the resolved subtree $G[x_j]$ by the optimal resolution of $G[x_j]$ (that we can obtain because $\tilde{b}(x_j) = i_j$), we obtain a new solution $\mathcal{R}'$ of the problem with at most one more EGT-loss event (on the edge $(parent(x), x)$) and such that the resolution of $G[x_j]$ in $\mathcal{R}'$ has a strictly lower cost than the resolution of $G[x_j]$ in $\mathcal{R}^*$. There is then at least one less event in the resolution of $G[x_j]$ in $\mathcal{R}'$ and as the cost are unitary, the solution $\mathcal{R}'$ we obtain is such that $C(\mathcal{R}') \leq C(\mathcal{R}^*)$ and thus $\mathcal{R}'$ is optimal. Contradiction. We then conclude that there is an optimal solution of the problem for which $\tilde{b}(x) = i$.

Thus, Algorithm 1 returns an optimal solution for the input $(\langle G, s\rangle, \tilde{s}, (M, I), S)$.

We conclude, by induction, that the solution returned by Algorithm 1 is optimal. ◀

Proof of Theorem 15

Proof. Once the optimal arrays are computed for all nodes, the optimal solution is easily reconstructed from the entry $min(r(G).array(0, k), r(G).array(1, k))$ by following the pointers from the root to the leaves.

The key point is therefore showing that the arrays computed by Algorithm 3 are exact, i.e., for each node x, $x.array(i, j)$ is equal to $minCostTransfer(\langle G[x], \tilde{s}, e\rangle, j, i)$ where $\langle G[x], \tilde{s}, e\rangle$ is the optimal DL-Reconciliation of $G[x]$ with S.

The proof is by induction on the height of $G(x)$.

If x is a leaf (either free or fixed), it is easy to see that $x.array$ is correct.

Now if x is an internal node, we assume by induction that $x_l.array$ and $x_r.array$ are correct. By contradiction, let's assume that there is (i, j) such that $x.array(i, j) \neq minCostTransfer(\langle G[x], \tilde{s}, e\rangle, j, i)$. Let $\mathcal{R}$ be the optimal DLE-Reconciliation obtained from the optimal DL-Reconciliation by converting some duplication events into EGT events events and by adding some EGT-loss events (this DLE-Reconciliation exists by Lemma 3) leading to $minCostTransfer(\langle G[x], \tilde{s}, e\rangle, j, i)$. Then, in $\mathcal{R}$, $\tilde{b}(x) = i$, $\tilde{b}(x_l) = \ell_1$ where $\ell_1 \in \{0, 1\}$ and $\tilde{b}(x_r) = \ell_2$ where $\ell_2 \in \{0, 1\}$. Also, as there are j free leaves labeled by 0 under x, the sum of the numbers of free leaves labeled by 0 under x_l and x_r must be equal to j. If the genome labels of the children of x are not the same as i, x is converted as an EGT event if x is a duplication node in the DL-Reconciliation (and possibly an EGT-Loss event is added) and if x is not a duplication node then some EGT-Loss events may be added on the edges between x and its children. As the algorithm considers all possibilities of genome labels for x_l and x_r and all possibilities of number of free leaves labeled by 0 under x_r and x_l leading to j free leaves under x (and considers the optimal assignation of EGT and EGT-loss events for the transfer(s) needed from x to its children), the particular possibility leading to $\mathcal{R}$ will be considered and then $x.array(i, j) = minCostTransfer(\langle G[x], \tilde{s}, e\rangle, j, i)$. Contradiction. Thus, there is no such (i, j) and $x.array$ is exact.

We conclude, by induction, that the arrays computed by Algorithm 3 are exact. ◀

Constructing Founder Sets Under Allelic and Non-Allelic Homologous Recombination

Konstantinn Bonnet
Institute for Medical Biometry and Bioinformatics, Heinrich Heine University, Düsseldorf, Germany

Tobias Marschall[1]
Institute for Medical Biometry and Bioinformatics, Heinrich Heine University, Düsseldorf, Germany

Daniel Doerr[1]
Institute for Medical Biometry and Bioinformatics, Heinrich Heine University, Düsseldorf, Germany

Abstract

Homologous recombination between the maternal and paternal copies of a chromosome is a key mechanism for human inheritance and shapes population genetic properties of our species. However, a similar mechanism can also act between different copies of the same sequence, then called *non-allelic homologous recombination* (NAHR). This process can result in genomic rearrangements – including deletion, duplication, and inversion – and is underlying many genomic disorders. Despite its importance for genome evolution and disease, there is a lack of computational models to study genomic loci prone to NAHR.

In this work, we propose such a computational model, providing a unified framework for both (allelic) homologous recombination and NAHR. Our model represents a set of genomes as a graph, where human haplotypes correspond to walks through this graph. We formulate two founder set problems under our recombination model, provide flow-based algorithms for their solution, and demonstrate scalability to problem instances arising in practice.

2012 ACM Subject Classification Applied computing → Bioinformatics

Keywords and phrases founder set reconstruction, variation graph, pangenomics, NAHR, homologous recombination

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.6

Supplementary Material *Software (Source Code)*: `https://github.com/marschall-lab/hrfs`
archived at `swh:1:dir:32151e14f3faef777e58b7e1af01942431deec26`

Funding This work was supported in part by the National Institutes of Health grant 1U01HG010973 to T.M., by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 956229, and by the BMBF-funded de.NBI Cloud within the German Network for Bioinformatics Infrastructure (de.NBI) (031A532B, 031A533A, 031A533B, 031A534A, 031A535A, 031A537A, 031A537B, 031A537C, 031A537D, 031A538A).

Acknowledgements The authors kindly thank Feyza Yilmaz for providing the haplotype data of the 1p36.13 locus.

1 Introduction

Twenty years ago, at this conference, Esko Ukkonen introduced the problem of inferring founder sets from haplotyped SNP sequences under allelic recombination [30]. Ukkonen's work has since inspired a wealth of research addressing various aspects and applications of founder set reconstruction ranging from the reconstruction of ancestral recombinations and pangenomics to applications in phage evolution [16, 19, 29]. In its original setting, the problem sets out from a given set of m sequences of equal length n, where characters

[1] joint last author

across sequences residing at the same index position correspond to a SNP. It then asks for a smallest set of sequences, called *founder set*, such that each given sequence can be constructed through a series of crossovers between sequences of the founder set, where each segment between two successive recombinations must meet a minimum length threshold. The *Founder Set Reconstruction* problem is NP-hard in general [22], but is solvable in linear time for the special case of founder sets of size two [30, 35]. Since its introduction, various heuristics and approximations have been proposed [35, 24, 25]. A variant of this problem restricts crossovers to coincide at certain positions, thereby decomposing the input sequences into a universally shared succession of blocks. The resulting problem, known as *Minimum Segmentation Problem* is polynomial [26]. In his seminal paper, Ukkonen devised a $O(n^2m)$ algorithm for its solution which has been improved by Norri *et al.* [17] to linear time, i.e. $O(nm)$, capitalizing on recent breakthroughs in data structures [9].

Just like the Founder Set Problem, the vast majority of population genetic analyses and genome-wide association studies have been focused on SNPs in the past decades, neglecting the more complex forms of variation – mostly for technical difficulties in detecting them. In particular, structural variants (SVs), commonly defined as variants of at least 50bp, have posed substantial challenges and studies based on short sequencing reads typically detect less than half of all SVs present in a genome [37]. Recent technological and algorithmic advances help to overcome these limitations [27]. Long read technologies now enable haplotype-resolved *de novo* assembly of human genomes [20], which in turn enables a much more complete ascertainment of SVs [10]. Earlier this year, the first complete telomere-to-telomere assembly of a human genome was announced [18], heralding a new era of genomics where high-quality, haplotype-resolved assemblies of complex repetitive genomic structures become broadly available. Presently, the Human Pangenome Reference Consortium (HPRC), is applying these techniques to generate a large panel of haplotype-resolved genome assemblies from samples of diverse ancestries [33]. These emerging data sets enable studying genetic loci involving duplicated sequence, called *segmental duplications* (SDs), which are amenable to NAHR and are therefore highly mutable and show complicated evolutionary trajectories [13, 31]. The T2T-CHM13 study alone reports over 40 thousand segmental duplications that amount to 202 Mb (6.6% of the human genome) [18].

Interestingly, at loci with highly similar segments arranged in opposite orientations, such as Segment 3 in Figure 1, NAHR can lead to *inversion*, i.e. the reversal of the interior sequence (Segment 4 in Figure 1). Because of being flanked by a pair of copies of the same sequence (cf. Segment 3) that often comprises tens of thousands of bases, such events have been largely undetectable by sequencing technologies with read lengths below the length of the duplicated sequence; in particular by conventional short read sequencing. Recent studies applying multiple technologies reveal that inversions affect tens of megabases of sequence in a typical human genome [7]. Unlike most other classes of genetic variation, inversions are often *recurrent* with high mutation rates, that is, the same events have happened multiple times in human history [21]. Depending on the structures of duplicated sequence at a particular locus, individual human haplotypes can differ in their potential for NAHR. This can have important implications for the risk for a range of genetic disorders caused by NAHR-mediated mutations [21].

In the past two decades, various mathematical models and algorithms to study genome rearrangements have been proposed. These range from the classic reversal [3, 2] and transposition [4] model to composed models for two or more balanced rearrangements [32, 8], to generalized models such as the popular *double cut and join* (DCJ) model [36, 5]. As the research in this field continues, advanced models can additionally accommodate one or more

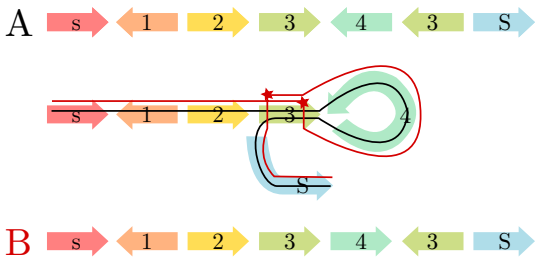


Figure 1 *Illustration of an NAHR-mediated inversion.* Haplotype A (black line) represents the original configuration, while haplotype B (red line) can be derived from A by two recombination events between inverted repeats of genomic marker 3 as indicated by the red stars.

types of unbalanced rearrangements, i.e., deletion, insertion, and duplication [28, 6]. Yet, none of these models adequately considers sequence similarity as a prerequisite for NAHR, which is a key molecular mechanism shaping complex loci in the human genome. In summary, there are now technological opportunities to study the population history of recalcitrant SD loci that are prone to genome rearrangements and relevant to disease, but computational models to facilitate this have so far been lacking.

In this work, we study homologous recombination in a genome model that represents DNA sequences at a level of abstraction where they are already decomposed into genomic markers with assigned homologies. Here, our notion of homology is a synonym for *high DNA sequence similarity*, as we adopt the terminology underlying the concept of homologous recombination. Our model permits recombination events to occur between homologous markers independent of their position within or between haplotypes, as long as the markers' orientations are respected. In other words, a marker can only recombine with a homologous marker alongside the same direction, as illustrated by Figure 1, because a recombination event can only occur between homologous markers if they are aligned to each other. By virtue of recapitulating the underlying molecular mechanism (NAHR), it implicitly allows for all the rearrangements it can give rise to, including deletion, duplication, and inversion.

Marker decomposition and homology assignment can be done in practice with genome graph builders such as MBG [23], minigraph [12], or pggb[2]. In fact, our algorithms are based on *variation graph* or *pangenome graph*, where nodes correspond to homologous DNA segments and edges between segments correspond to observed adjacencies in a given set of haplotypes.

2 Methods

2.1 Preliminaries

A *(genomic) marker* m is an element of the finite universe of markers denoted by $\mathcal{M}$, and is associated with a fragment of a double-stranded DNA molecule. Each marker can be traversed in *forward* and *reverse* direction. A marker in forward orientation (which is the default orientation) is traversed from left to right. Overline notation $\overline{m}$ indicates the reversal of a marker m, which is carried out relative to its orientation, i.e., $\overline{\overline{m}} = m$. Similarly, $\overline{\mathcal{M}}$ represents the set of all reverse oriented markers. We designate two forward oriented markers $\{s, S\} \subseteq \mathcal{M}$ as *terminal markers*. In what follows, we study *terminal sequences*, that is,

[2] `https://github.com/pangenome/pggb`

sequences drawn from the alphabet of oriented markers $\mathcal{M} \cup \overline{\mathcal{M}}$ that start with s or $\overline{S}$, end in S or $\overline{s}$ and do not contain any further terminal markers in between. A terminal sequence can be traversed in forward and reverse direction. A *haplotype* is a terminal sequence that starts with s (*source*) and ends with S (*sink*).

▶ **Example 1.** Consider in the following two sequences of genomic markers A and X drawn from the universe of markers $\mathcal{M} = \{\mathtt{s}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{S}\}$, where $A = \mathtt{s\overline{1}234\overline{3}S}$ and $X = \mathtt{s\overline{1}234\overline{32}1\overline{s}}$. Sequence A starts and ends with terminal markers $\mathtt{s}$ and $\mathtt{S}$, respectively, thus constituting a *haplotype* drawn from $\mathcal{M}$. Conversely, X starts with $\mathtt{s}$ and ends in $\overline{\mathtt{s}}$ and therefore is a terminal sequence, but not a *haplotype*.

Given a sequence A, $|A|$ indicates the length of A which corresponds to the number of A's constituting elements. $\overline{A}$ defines the *reverse complementation* of sequence A, i.e., the simultaneous reversal of the sequence and its constituting elements. The element at the ith position in sequence A is denoted by $A[i]$. A *segment* of sequence A starting at position i and ending at and including position j, is denoted by $A[i..j]$. In particular, $A[..i] := A[1..i]$ and $A[i..] := A[i..|A|]$ denote the *prefix* and *suffix* of A, respectively. The operator "+" indicates the concatenation of two sequences.

▶ **Example 1** (cont'd). The length of A is $|A| = 7$; its reverse complement is $\overline{A} = \mathtt{\overline{S}34\overline{32}1\overline{s}}$; $A[4..6]$ is a segment of A and corresponds to sequence $\mathtt{3\overline{43}}$; The segments $X[..6] = \mathtt{s\overline{1}234\overline{3}}$ and $A[7..] = \mathtt{S}$ are a prefix and a suffix of X and A, respectively; The concatenation of prefix $X[..6]$ and suffix $A[7..]$ results in haplotype $X[..6] + A[7..] = \mathtt{s\overline{1}234\overline{3}S}$.

A *recombination* is an operation that acts on a shared oriented marker m of any two terminal sequences A and B: let $A[i] = B[j] = m$; recombination $\chi(A, B, i, j)$ produces terminal sequence $C = A[..i] + B[j + 1..]$. For a given set of haplotypes $\mathcal{H}$, span($\mathcal{H}$) denotes the *span*, i.e., the set of all *haplotypes* generated by applying χ on haplotypes $\mathcal{H}$ and the resulting terminal sequences. More precisely, let $\mathcal{T}$ be the universe of terminal sequences, defined recursively by $\mathcal{H} \cup \overline{\mathcal{H}} \subseteq \mathcal{T}$ such that for any $A, B \in \mathcal{T}$ with some $A[i] = B[j]$ the recombinant $C = A[i] + B[j + 1]$ and its reverse complement $\overline{C}$ is also in $\mathcal{T}$. Then $span(\mathcal{H}) := \{A \in \mathcal{T} \mid A \text{ is a haplotype}\}$. Accordingly, we also say that "$\mathcal{H}$ is a *generating set* of span($\mathcal{H}$)". Conversely, given any (possibly infinite) set of haplotypes $\mathcal{S}$ and some $\mathcal{H} \subseteq \mathcal{S}$, $\mathcal{H}$ is a generating set of $\mathcal{S}$ iff span($\mathcal{H}$) = $\mathcal{S}$.

▶ **Example 1** (cont'd). Recombination $\chi(A, \overline{A}, 4, 2)$ produces terminal sequence $X = \mathtt{s\overline{1}234\overline{32}1\overline{s}}$. Subsequent recombination $\chi(X, A, 6, 6)$, produces haplotype $B = \mathtt{s\overline{1}234\overline{3}S}$. If $\{A\}$ is a given set of haplotypes, then span($\{A\}$) = $\{A, B\}$.

In this paper, we study the following two problems:

▶ **Problem 1** (Founder Set). *Given a set of haplotypes $\mathcal{H}$, find a generating set $\mathcal{F} \subseteq$ span($\mathcal{H}$) such that $\sum_{A \in \mathcal{F}} |A|$ is minimized.*

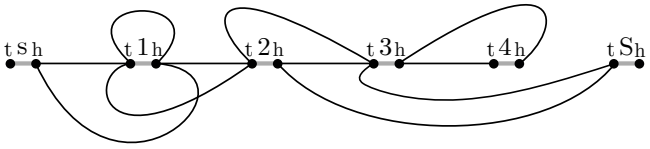
We call a solution to Problem 1 a *founder set* and its members *founder sequences*.

▶ **Problem 2.** *Given a set of haplotypes $\mathcal{H}$, find a founder set $\mathcal{F}$ that minimizes the number of recombinations applied to haplotypes $\mathcal{H}$ and their intermediate terminal sequences in constructing $\mathcal{F}$.*

2.2 Constructing Founder Sets

Variation graph construction. We solve Problem 1 by studying the *variation graph* $G_{\mathcal{H}} = (V, E \cup \overrightarrow{E})$ of the given set of haplotypes $\mathcal{H}$. Graph $G_{\mathcal{H}}$ is an undirected edge-colored multigraph where each edge can have one of two colors corresponding to their membership in edge sets E and $\overrightarrow{E}$. In constructing $G_{\mathcal{H}}$, each marker m of the universe of forward-oriented markers $\mathcal{M}$ is represented by a tuple of its *extremities* $(m^{\mathrm{t}}, m^{\mathrm{h}})$ also called "*tail*" and "*head*" of m, respectively, and its reverse-oriented counterpart $\overline{m}$ is represented as $(m^{\mathrm{h}}, m^{\mathrm{t}})$[3]. Node set V of graph $G_{\mathcal{H}}$ corresponds to the set of all marker extremities, and each marker $m \in \mathcal{M}$ gives rise to one *marker edge* $\{m^{\mathrm{t}}, m^{\mathrm{h}}\} \in \overrightarrow{E}$. Further, any two (not necessarily distinct) nodes $m_1^b, m_2^c \in V$ are connected by one *adjacency edge* $\{m_1^b, m_2^c\} \in E$ iff there exists a sequence $A \in \mathcal{H} \cup \overline{\mathcal{H}}$ with $A = ..m_1 m_2..$ such that $m_1 = (m_1^a, m_1^b)$, $m_2 = (m_2^c, m_2^d)$ and $\{a, b\} = \{c, d\} = \{\mathrm{t}, \mathrm{h}\}$.

▶ **Example 2.** Let $H_1 = \mathtt{s\overline{1}2\overline{34}\overline{3}S}$, $H_2 = \mathtt{s1112\overline{34}\overline{3}S}$, $H_3 = \mathtt{s\overline{1}2\overline{34}\overline{32}\overline{34}\overline{3}S}$, and $H_4 = \mathtt{s\overline{1}2S}$, then the variation graph $G_{\mathcal{H}}$ of $\mathcal{H} = \{H_1, H_2, H_3, H_4\}$ is as follows, with marker edges drawn in gray and adjacency edges in black:



▶ **Proposition 3.** *Let $G_{\mathcal{H}}$ be the variation graph of haplotypes $\mathcal{H}$, and $\mathcal{X}$ the set of all walks between terminal markers s^{t} and S^{h} in $G_{\mathcal{H}}$ with edges alternating between E and $\overrightarrow{E}$, then* $\mathrm{span}(\mathcal{H}) = \mathcal{X}$.

Proof.

⇒ Observe that no recombination can create a new pair of consecutive markers $m_1 m_2$ that is not contained in any sequence $A \in \mathcal{H} \cup \overline{\mathcal{H}}$. Therefore, each haplotype $B \in \mathrm{span}(\mathcal{H})$ is a succession of consecutive markers drawn from sequences in $\mathcal{H} \cup \overline{\mathcal{H}}$, i.e., B can be delineated in $G_{\mathcal{H}}$ by following adjacency edges corresponding to its succession of consecutive markers.

⇐ If each alternating walk $X = (s^{\mathrm{t}}, s^{\mathrm{h}}, \ldots, S^{\mathrm{t}}, S^{\mathrm{h}}) \in \mathcal{X}$ in variation graph $G_{\mathcal{H}}$ corresponds to a haplotype $B \in \mathrm{span}(\mathcal{H})$, then X must be producible through a series of recombinations of haplotypes $\mathcal{H}$ and their recombinants. We show this by construction:

- **a.** Pick some haplotype $A \in \mathcal{H}$ and initialize $i \leftarrow 1$;
- **b.** Let $B \in \mathcal{H} \cup \overline{\mathcal{H}}$ be a sequence such that for some position j, $B[j..j+1] = m_1 m_2$ with $m_1 = X[i..i+1]$ and $m_2 = X[i+2..i+3]$. Then $A \leftarrow \chi(A, B, i/2, j)$.
- **c.** Increase i by 2 and repeat step **b** unless $i = |X| - 3$.

Observe that by construction of the variation graph $G_{\mathcal{H}}$, a suitable sequence $B \in \mathcal{H} \cup \overline{\mathcal{H}}$ must exist in each iteration of step **b**. ◀

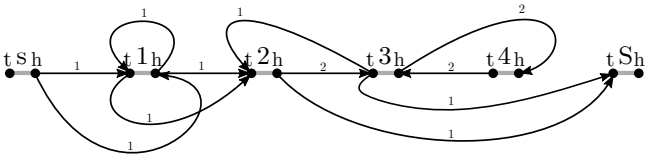
Defining flows on variation graphs. We determine a minimum set of founder sequences by solving a network flow problem in variation graph $G_{\mathcal{H}}$ where flow is allowed to travel along adjacency edges in either direction. In doing so, we find a non-negative flow $\phi : V \times V \to \mathbb{N}$ such that the total flow $\sum_{u,v \in V} \phi(u, v)$ of graph $G_{\mathcal{H}}$ is minimized and satisfies the following constraints:

[3] Our notation is consistent with common practice of illustrating markers as arrows, that, in natural reading direction, face from left (tail of the arrow) to right (head of the arrow).

$$\begin{aligned}
&\forall\, u, v \in V && \phi(u,v) \in \mathbb{N} && \textit{(constrain flow to integer)}\\
&\forall\, (u,v) \in \{(u',v') \mid u', v' \in V : \{u',v'\} \notin E\} && \phi(u,v) = 0 && \textit{(constrain travel of flow)}\\
&\forall\, v \in V && i(v) := \sum_{u \in V} \phi(u,v) && \textit{(incoming flow)}\\
& && o(v) := \sum_{u \in V} \phi(v,u) && \textit{(outgoing flow)}\\
&\forall\, \{u,v\} \in E && \phi(u,v) + \phi(v,u) \geq 1 && \textit{(flow coverage)}\\
& && o(s^{\mathrm{t}}) = i(S^{\mathrm{h}}) = 0 && \textit{(flow direction } s^{\mathrm{t}} \to S^{\mathrm{h}}\textit{)}\\
&\forall\, m \in \mathcal{M} \setminus \{s, S\} && i(m^{\mathrm{t}}) = o(m^{\mathrm{h}}) && \textit{(flow conservation)}\\
& && i(m^{\mathrm{h}}) = o(m^{\mathrm{t}}) &&
\end{aligned}$$

Note that the flow can travel in both directions of an edge $\{u, v\} \in E$ and that $\phi(u, v) = \phi(v, u)$ does not hold true in general. The only node pairs of the graph that are *unbalanced*, i.e., do not satisfy flow conservation, are $(s^{\mathrm{t}}, s^{\mathrm{h}})$ and $(S^{\mathrm{t}}, S^{\mathrm{h}})$.

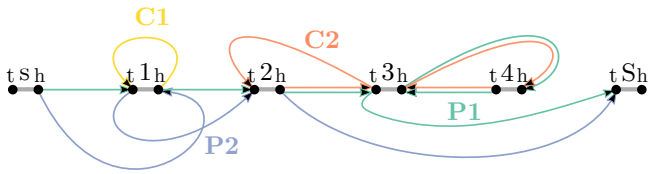
▶ **Example 2** (cont'd). The drawing below illustrates a flow solution on variation graph $G_{\mathcal{H}}$, with the direction and amount of flow along adjacency edges indicated by labeled arrowed arcs.



Deriving haplotypes from flows. By applying the Flow Decomposition Theorem [1, p. 80f], any *flow*, i.e., solution to the above-specified constraints, is decomposable into a set of alternating paths going from source s^{t} to sink S^{h} and a set of alternating cycles. Ahuja *et al.* [1] give a simple and efficient algorithm that does so in polynomial time and which we describe below, adapted to our circumstances. The idea is to perform a random walk in the graph from source to sink or within a cycle, thereby consuming flow along adjacency edges until all flow is depleted. The proof of the algorithm remains unchanged to that given by Ahuja *et al.*, thus is not repeated here.

1. Set $u \leftarrow s^{\mathrm{t}}$.
2. Setting out from current node u, traverse the incident marker edge to some node v, choose any neighbor w of v for which $\phi(v, w) > 1$. Follow the adjacency edge to v and decrease the flow $\phi(v, w)$ by 1. Set $u \leftarrow w$.
3. As long as $u \neq S^{\mathrm{t}}$ do as follows: if u has been visited in the traversal before, then extract the corresponding alternating cycle from the recorded sequence and report it. Proceed with the traversal by repeating step 2.
4. However, if $u = S^{\mathrm{t}}$, follow the marker edge to S^{h} and report the recorded sequence as a path.
5. If s^{h} is incident to edges with positive flow, proceed with step 1. Otherwise, there still might be strictly positive flow remaining in the graph corresponding to unreported cycles. In that case, pick any node $u \leftarrow m^a$ such that for some node w, $\phi(m^b, w) > 0$, $\{a, b\} = \{\mathrm{t}, \mathrm{h}\}$ and $m \in \mathcal{M}$, and proceed with step 2.

▶ **Example 2** (cont'd). The components of the flow solution on variation graph $G_{\mathcal{H}}$ comprise two cycles C1 and C2, and two (s^t, S^h)-paths P1 and P2, as illustrated below.

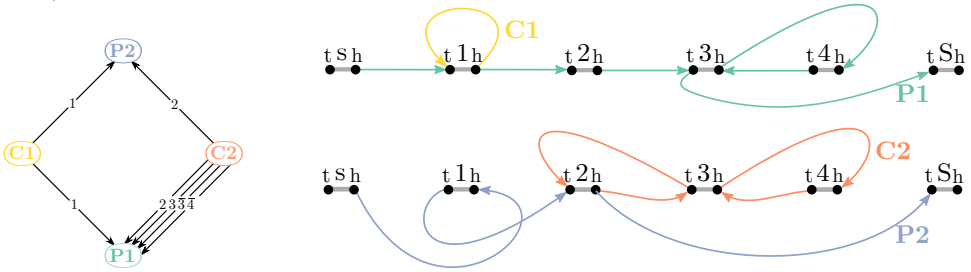


What remains is the integration of cycles into walks that then correspond to the haplotypes of the founder set. The integration is facilitated by a graph structure, the *component graph*. The component graph $G' = (V', E', l)$ is an edge-labeled, directed multigraph, where, in its initial construction, each alternating $(s^{\mathrm{t}}, S^{\mathrm{h}})$-path and each cycle reported during flow decomposition is represented by a distinct node of V'. In the component graph G', each cycle c of the flow decomposition sharing one or more markers with another component c' is connected by one or more directed edges (c, c') to that component, with each edge's label $l(c, c')$ corresponding to one distinct shared marker, oriented according to the their succession in c (which may not be the same as in c'). The component graph is then successively deconstructed until empty as follows:

1. Remove and report all $(s^{\mathrm{t}}, S^{\mathrm{h}})$-walks with in-degree 0 from node set V'[4].
2. Pick a cycle $c \in V'$ with in-degree 0, or, if none such exists, any arbitrary cycle $c \in V'$.
3. Pick an outgoing edge $(c, c') \in E'$ such that c' is a $(s^{\mathrm{t}}, S^{\mathrm{h}})$-walk. If no such c' exists, c is only adjacent to cycles, out of which one c' is picked at will. Let $(m^a, m^b) \leftarrow l(c, c')$, $\{a, b\} = \{\mathrm{t}, \mathrm{h}\}$. If marker m is embedded in c' in same orientation, i.e. $c' = ..m^a m^b..$, then linearize c in m, i.e., $c = m^b c_1..c_{k-1}m^a$, and integrate it into c' such that $c' \leftarrow ..m^a m^b c_1..c_{k-1}m^a m^b..$. Otherwise, integrate the reversed linearization of c, i.e, $c' \leftarrow ..m^b m^a c_{k-1}..c_1 m^b m^a..$. Remove cycle c and its outgoing edges from component graph G'.
4. Proceed with step 1 until no more components remain and all $(s^{\mathrm{t}}, S^{\mathrm{h}})$-walks are reported.

The search for components with in-degree 0 can be efficiently implemented through preorder traversal of G'. Note that each cycle must have at least one outgoing edge and that ultimately all cycles *must be* integrable into a $(s^{\mathrm{t}}, S^{\mathrm{h}})$-walk, otherwise this would imply that $G_{\mathcal{H}}$ contains a disconnected, circular component that is not reachable by an alternating path from source s^{t} to sink S^{h}, thus contradicting the correctness of $G_{\mathcal{H}}$'s construction. The reported $(s^{\mathrm{t}}, S^{\mathrm{h}})$-walks represent the wanted haplotypes of the founder set.

▶ **Example 2** (cont'd). The plot below depicts the component graph of components C1, C2, P1, and P2 (left) and the final two (s^t, S^h)-walks that collectively represent a founder set of $\mathcal{H}$ (right).



[4] By construction, $(s^{\mathrm{t}}, S^{\mathrm{h}})$-walks have out-degree 0, i.e., those with in-degree 0 are singleton in G'.

▶ **Theorem 4.** *Any flow that minimizes the total flow* $\sum_{u,v \in V} \phi(u,v)$ *of variation graph* $G_{\mathcal{H}} = (V, E \cup \overrightarrow{E})$ *of a given set of haplotypes* $\mathcal{H}$ *is equivalent to a solution to Problem 1.*

Proof. It is sufficient to show that every flow is decomposable into a set of haplotypes ($\Rightarrow$) and every founder set represents a valid flow ($\Leftarrow$).

$\Rightarrow$ Any flow of variation graph $G_{\mathcal{H}}$ is decomposable into a set of haplotypes $\mathcal{H}'$, as demonstrated above. Observe that the above-listed flow constraints enforce the derived haplotypes $\mathcal{H}'$ to cover the entire graph $G_{\mathcal{H}}$ and consequently $G_{\mathcal{H}'} = G_{\mathcal{H}}$. This implies that $\text{span}(\mathcal{H}') = \text{span}(\mathcal{H})$, i.e., $\mathcal{H}'$ is a generating set of $\text{span}(\mathcal{H})$. Therefore, the sum of lengths of haplotypes derived from a flow solution is an upper bound of Problem 1.

$\Leftarrow$ Any set of haplotypes $\mathcal{H}' \subseteq \text{span}(\mathcal{H})$ that covers each consecutive pair of markers $m_1 m_2$ in haplotypes $\mathcal{H}$ at least once (either in forward orientation $m_1 m_2$ or in reverse orientation $\overline{m_2 m_1}$) represents a valid flow of $G_{\mathcal{H}}$. To construct a flow from $\mathcal{H}'$, set $\phi(m_1^b, m_2^c)$ to the number of occurrences of consecutive markers $m_1 m_2$ in haplotypes of $\mathcal{H}'$ with $m_1 = (m_1^a, m_1^b)$ and $m_2 = (m_2^c, m_2^d)$, $\{a,b\} = \{c,d\} = \{\text{t},\text{h}\}$. Observe that by construction, flow is integer, travels from source s^{t} to sink S^{h} and satisfies coverage and conservation constraints. ◀

2.3 Minimizing Recombinations in Founder Sequences

We now present an algorithm towards solving Problem 2, i.e., the problem of finding a founder set that minimizes the number of recombinations needed for its construction from a given set of haplotypes $\mathcal{H}$. Our approach is exact under the assumption that the overall multiplicity of each pair of consecutive markers in the founder set of a solution to Problem 2 is known, yet the pair's particular orientation in a founder sequence may be unresolved. To this end, we presume a given function $\hat{\phi}(m_1, m_2)$ acting as oracle for the overall multiplicity of any given pair of consecutive oriented markers $m_1, m_2 \in \mathcal{M} \cup \overline{\mathcal{M}}$[5]. More specifically, $\hat{\phi}(m_1, m_2)$ reports the total number of occurrences of $m_1 m_2$ and $\overline{m_2 m_1}$ in a solution to Problem 2. In addition, we make use of function $\mu(m) := \sum_{m' \in \mathcal{M} \cup \overline{\mathcal{M}}} \hat{\phi}(m, m')$ to retrieve the multiplicity of any marker $m \in \mathcal{M} \cup \overline{\mathcal{M}}$[6]. Our solution makes use of the *flow graph* that is defined in the subsequent paragraph. We calculate a matching in the flow graph that describes a set of founder sequences, each corresponding to a succession of segments of haplotypes $\mathcal{H}$. The objective of the matching is to minimize the total number of these segments across all founder sequences which is equivalent to minimizing the number of recombinations for their construction from haplotype set $\mathcal{H}$.

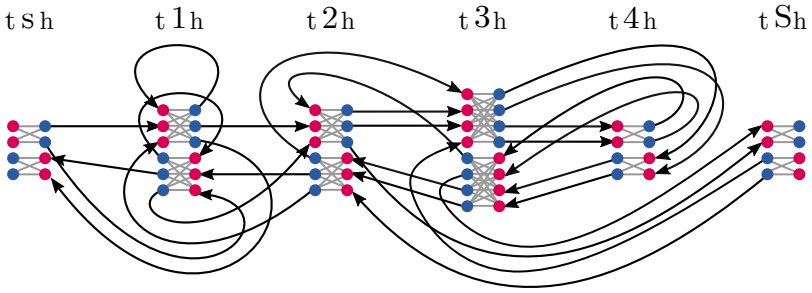
Flow graph construction. The flow graph $G_{\mathcal{H},\hat{\phi}} = (V_{\hat{\phi}}, E_{\hat{\phi}} \cup \overrightarrow{E_{\hat{\phi}}})$ is a directed edge-colored multigraph with adjacency edges $E_{\hat{\phi}}$ and marker edges $\overrightarrow{E_{\hat{\phi}}}$, where each marker extremity m^a, $m \in \mathcal{M}$ and $a \in \{\text{t},\text{h}\}$, gives rise to $2 \cdot \mu(m)$ elements in node set $V_{\hat{\phi}}$, representing $\mu(m)$ many "*in*" (i) and $\mu(m)$ many "*out*" (o) nodes. That is, $V_{\hat{\phi}} = \{i^x_{m^a} \mid m \in \mathcal{M}, a \in \{\text{t},\text{h}\}, x \in 1..\mu(m)\} \cup \{o^x_{m^a} \mid m \in \mathcal{M}, a \in \{\text{t},\text{h}\}, x \in 1..\mu(m)\}$. Each out node $u \in V_{\hat{\phi}} \setminus (\{o^x_{S^{\text{h}}} \mid 1..\mu(S)\} \cup \{o^x_{s^{\text{t}}} \mid 1..\mu(s)\})$ is incident to *one and only one* directed adjacency edge

[5] Our experiments directly use the results of Problem 1 as input for Problem 2. In other words, the multiplicities reported by $\hat{\phi}(m_1, m_2)$ are the number of occurrences of (m_1, m_2) in a solution to Problem 1. This makes our experimental solutions to Problem 2 heuristic.

[6] $\hat{\phi}$ and μ are symmetric w.r.t. the relative orientation of markers, $\hat{\phi}(m_1, m_2) = \hat{\phi}(\overline{m_2}, \overline{m_1})$ and $\mu(m) = \mu(\overline{m})$

(u, v) connecting u to some in node v thereby realizing one occurrence of its representing pair of consecutive oriented markers in a founder sequence. Conversely, each forward-oriented marker $m \in \mathcal{M}$ contributes $\mu(m)^2$ many directed marker edges that connect in/tail nodes with out/head nodes, i.e., $\{(i^x_{m^\mathrm{t}}, o^y_{m^\mathrm{h}}) \mid x, y \in 1..\mu(m)\}$. Analogously, each reverse-oriented marker $\overline{m} \in \overline{\mathcal{M}}$ contributes $\mu(m)^2$ many in/head-to-out/tail-directed marker edges $\{(i^x_{m^\mathrm{h}}, o^y_{m^\mathrm{t}}) \mid x, y \in 1..\mu(m)\}$.

▶ **Example 2** (cont'd). The flow graph $G_{\mathcal{H},\hat{\phi}}$ for the given set of haplotypes $\mathcal{H} = \{\mathtt{s\overline{1}23\overline{43}S}, \mathtt{s1112343\overline{3}S}, \mathtt{s\overline{1}234\overline{32}343\overline{3}S}, \mathtt{s\overline{1}2S}\}$ and a given $\hat{\phi}$ is as follows:



In nodes and out nodes are highlighted in red and blue, respectively. For clarity, the direction of marker edges (gray edges; directed from in to out node) is omitted in the illustration.

Graph decomposition. A perfect matching of marker edges in flow graph $G_{\mathcal{H},\hat{\phi}}$ produces a set of alternating walks and alternating cycles through $G_{\mathcal{H},\hat{\phi}}$, yet only half of the graph is eligible to form a solution to Problem 2. More precisely, for each marker $m \in \mathcal{M}$, exactly half of the number of its associated nodes in $V_{\hat{\phi}}$ must be saturated in the matching that we seek, the other half as well as their incident edges must remain unsaturated. Further, we aim to admit only matchings that consist entirely of alternating $(i^x_{s^\mathrm{t}}, o^y_{S^\mathrm{h}})$-walks, because only those correspond to valid haplotypes of span$(\mathcal{H})$.

At last, we aim to assign to each saturated node $v \in V_{\hat{\phi}}$ a position in some haplotype A of given haplotype set $\mathcal{H}$. That way, we are able to determine whether the incident adjacency edge serves as continuation of the associated segment in A, or whether the incident saturated marker edge implies a recombination between two distinct segments.

The *integer linear program* (ILP) shown in Algorithm 1 implements the above-stated constraints.

Matching constraints. Each edge and node of flow graph $G_{\mathcal{H},\hat{\phi}}$ is associated with binary variables of `x` and `y`, respectively, that determine their saturation in a solution (cf. domains D.1 and D.2). Constraint C.01 ensures that each saturated marker edge is incident to saturated nodes. Perfect matching constraints, i.e., constraints that impose each saturated node being incident to exactly one marker edge, are implemented by constraint C.02. Similarly, constraint C.03 ensures that an adjacency edge is saturated iff its incident nodes are saturated. In other words, constraints C.01-C.03 together ensure that each component of the saturated graph corresponds to an alternating path or cycle component (the latter being prohibited by further constraints). The following two constraints C.04 and C.05 control the overall size of the saturated graph. In doing so, they ensure that, in a solution to Problem 2, the number of saturated nodes and adjacency edges matches the postulated multiplicity of markers $\mu(m)$, $m \in \mathcal{M} \cup \overline{\mathcal{M}}$, and pairs of consecutive markers $\hat{\phi}(m_1, m_2)$, $m_1, m_2 \in \mathcal{M} \cup \overline{\mathcal{M}}$, respectively.

Algorithm 1 An ILP solution to Problem 2.

Objective:

Maximize

$$\sum_{\substack{(i^x_{m^a}, o^{x'}_{m^b}) \in \overrightarrow{E_{\hat{\phi}}}, \\ A[j]=(m^a, m^b)}} \mathtt{t}^{A[j]}_{i^x_{m^a} o^{x'}_{m^b}}$$

Constraints:

(C.01)	$\mathtt{y}_u + \mathtt{y}_v \geq 2\ \mathtt{x}_{uv}$	$\forall\ (u,v) \in \overrightarrow{E_{\hat{\phi}}}$
(C.02)	$\sum_{(u,v)\in\overrightarrow{E_{\hat{\phi}}}} \mathtt{x}_{uv} = \mathtt{y}_u$	$\forall$ in nodes $u \in V_{\hat{\phi}}$
	$\sum_{(u,v)\in\overrightarrow{E_{\hat{\phi}}}} \mathtt{x}_{uv} = \mathtt{y}_v$	$\forall$ out nodes $v \in V_{\hat{\phi}}$
(C.03)	$\mathtt{x}_{uv} = \mathtt{y}_u$	$\forall\ (u,v) \in E_{\hat{\phi}}$
	$\mathtt{x}_{uv} = \mathtt{y}_v$	
(C.04)	$\sum_{x=1}^{\mu(m)} \mathtt{y}_{i^x_{m^a}} + \mathtt{y}_{o^x_{m^a}} = \mu(m)$	$\forall\ m \in \mathcal{M}, a \in \{\mathrm{t}, \mathrm{h}\}$
(C.05)	$\sum_{\substack{x,x' \text{ s.t.} \\ (o^x_{m^b_1}, i^{x'}_{m^c_2}) \in E_{\hat{\phi}}}} \mathtt{x}_{o^x_{m^b_1} i^{x'}_{m^c_2}} = \hat{\phi}(m_1, m_2)$	$\forall\ (m^b_1, m^c_2)$ s.t. $\hat{\phi}(m_1, m_2) > 0$, $m_1 = (m^a_1, m^b_1)$, $m_2 = (m^c_2, m^d_2)$, $\{a,b\} = \{c,d\} = \{\mathrm{t}, \mathrm{h}\}$
(C.06)	$\mathtt{f}_u = \mathtt{f}_v$	$\forall\ (u,v) \in E_{\hat{\phi}}$
(C.07)	$\mathtt{f}_v - \mathtt{f}_u + T\mathtt{x}_{uv} \leq T + 1$	$\forall\ (u,v) \in \overrightarrow{E_{\hat{\phi}}}$
(C.08)	$\mathtt{f}_v = 0$	$\forall v \in \{o^x_{s^{\mathrm{t}}} \mid x \in 1..\mu(s)\} \cup \{i^x_{S^{\mathrm{h}}} \mid x \in 1..\mu(S)\}$
(C.09)	$\sum_{\substack{A\in\mathcal{H} \\ A[j]=m}} \mathtt{c}^{A[j]}_v = 1$	$\forall\ v \in V_{\hat{\phi}}$, v associated with extremities of marker m
(C.10)	$\mathtt{c}^{A[j]}_{o^x_{m^b_1}} = \mathtt{c}^{A[j+1]}_{i^{x'}_{m^c_2}}$	$\forall\ (o^x_{m^b}, i^{x'}_{m^a}) \in E_{\hat{\phi}}$, $A \in \mathcal{H} \cup \overline{\mathcal{H}}$, $i \in 1..\lvert A\rvert - 1$, s.t. $A[j..j+1] = (m^a_1, m^b_1)(m^c_2, m^d_2)$
(C.11)	$\mathtt{x}_{i^x_{m^a} o^{x'}_{m^b}} + \mathtt{c}^{A[j]}_{i^x_{m^a}} + \mathtt{c}^{A[j]}_{o^{x'}_{m^b}} \geq 3\ \mathtt{t}^{A[j]}_{i^x_{m^a} o^{x'}_{m^b}}$	$\forall\ (i^x_{m^a}, o^{x'}_{m^b}) \in \overrightarrow{E_{\hat{\phi}}}$, $A \in \mathcal{H} \cup \overline{\mathcal{H}}$, $i \in 1..\lvert A\rvert$, s.t. $A[j] = (m^a, m^b)$

Domains:

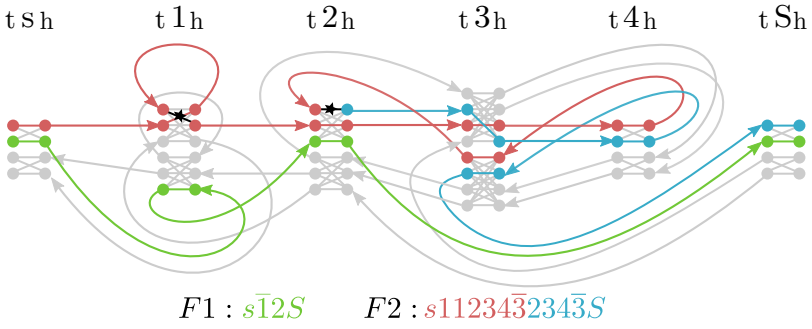
(D.01)	$\mathtt{x}_{uv} \in \{0,1\}$	$\forall\ (u,v) \in E_{\hat{\phi}} \cup \overrightarrow{E_{\hat{\phi}}}$
(D.02)	$\mathtt{y}_v \in \{0,1\}$	$\forall\ v \in V_{\hat{\phi}}$
(D.03)	$1 \leq \mathtt{f}_v \leq T$	$\forall\ v \in V_{\hat{\phi}}$
(D.04)	$\mathtt{c}^{A[j]}_{i^x_{m^a}}, \mathtt{c}^{A[j]}_{o^x_{m^a}}, \mathtt{c}^{A[j]}_{i^x_{m^b}}, \mathtt{c}^{A[j]}_{i^x_{m^b}} \in \{0,1\}$	$\forall\ A \in \mathcal{H} \cup \overline{\mathcal{H}}$, $j \in 1..\lvert A\rvert$, $A[j] = (m^a, m^b)$, $x \in 1..\mu(m)$
(D.05)	$\mathtt{t}^{A[j]}_{i^x_{m^a} o^{x'}_{m^b}} \in \{0,1\}$	$\forall\ A \in \mathcal{H} \cup \overline{\mathcal{H}}$, $j \in 1..\lvert A\rvert$, $A[j] = (m^a, m^b)$, $x \in 1..\mu(m)$

Path constraints. Constraints C.05-C.08 force each component of the saturated graph to start and end in nodes associated with source s^{t} and sink S^{h}, respectively, thereby ruling out any cycles. To this end, they make use of a set of integer variables `f` (cf. Domain D.03) that define an increasing flow within each saturated component that is bounded by constant T corresponding to the total flow of the graph, i.e., $T := \sum_{m \in \mathcal{M}} \mu(m)$. In each saturated marker edge, the flow is increased by 1 while along each adjacency edge, flow is kept constant. This prevents the formation of saturated cycles, because their flow would be infinite. Lastly, constraint C.08 preclude paths from starting in S^{h} or ending in s^{t}, leaving only one option for any saturated component open, that is, the formation of a $(s^{\mathrm{t}}, S^{\mathrm{h}})$-path.

Haplotype assignment. Each node in a solution to the ILP is associated with exactly one position in a haplotype in $\mathcal{H}$, recorded by binary variables `c`. Moreover, any marker edge whose incident pair of nodes is associated with the same position of the same haplotype corresponds to a conserved segment, i.e, no recombination within this marker has taken place. Each marker edge corresponding to a conserved segment contributes a score unit to the objective function. These score units are encoded by binary variables `t` (cf. domain D.05). Constraint C.09 ensures that each marker is associated with exactly one position j in a haplotype A of set $\mathcal{H} \cup \overline{\mathcal{H}}$, while C.10 confines incident nodes of adjacency edges to represent a consecutive marker pair $A[j..j+1]$. At last, constraint C.11 allows `t` variables of marker edges to take on value 1 only if that marker edge is saturated and its incident nodes are associated with the same haplotype position.

By maximizing the sum over `t` variables, the objective minimizes the total number of segments needed to decompose the calculated founder sequences into segments from haplotypes $\mathcal{H} \cup \overline{\mathcal{H}}$ that are delimited by recombination events.

▶ **Example 2** (cont'd). The following plot illustrates a matching that is solution to Algorithm 1 for $G_{\mathcal{H},\hat{\phi}}$. The founder sequences are spelled out on the bottom, colored by haplotype (red, blue and green for haplotypes 2, 3 and 4 respectively). Unsaturated nodes and edges are grayed out, haplotype assignments implied by colored paths. The solution features two recombinations, marked by "⋆" along their associated marker edges.



3 Results

We implemented our methods in the programming language Rust [14] and used Gurobi [11] as the solver. Our software is *open source* and publicly available on `https://github.com/marschall-lab/hrfs`. To run Algorithm 1 on a given set of haplotypes $\mathcal{H}$, we estimated the overall multiplicity $\hat{\phi}(m_1, m_2)$ of pairs of consecutive markers $m_1 m_2$ from a network flow solution to Problem 1 on $\mathcal{H}$. Note that this dispenses Algorithm 1 from being exact in our applications.

All experiments were run on a de.NBI cloud computing machine. For benchmarking purposes, we ran Gurobi single-threaded and recorded wall clock time (in seconds) and *proportional set size* (PSS, in Mb) for memory usage. Optimization time was capped at 30 minutes, beyond which the solver must capitulate and return its best-effort solution found thus far. The threshold for execution time is based upon available compute resources.

3.1 Experimental Data

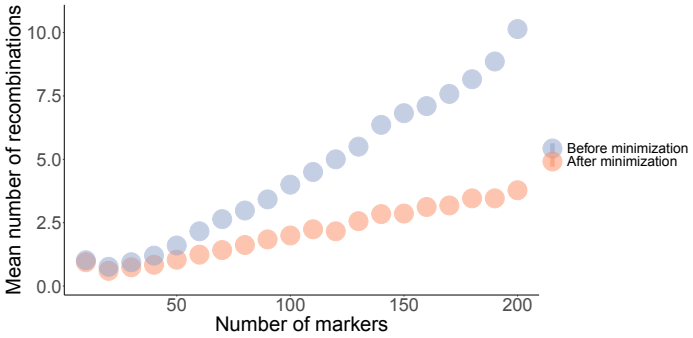


Figure 2 Mean number of recombinations by the size of the graph. Experiments were ran with values ranging from 10 to 200 in for the number of markers, in increments of 10. The ratio of duplications and of inversions was fixed to 10%, and number of haplotypes to 10. Each colored dot represents the mean number of recombinations over 50 replicates for one parameter set, after random assignment trials (blue) and after optimization (red).

We benchmarked the performance of our algorithms by conducting experiments on both simulated data and a real-world data set. The former presumed a simulator, capable of generating haplotypes with duplicated and inverted markers that can produce intricate homologous recombinations while providing control over the degree of complexity. To this end, we implemented our own simulation tool, that constructs a single haplotype sequence sampled at random to serve as seed. This seed sequence is adjustable by the following parameters: (i) number of distinct markers, i.e., the size of its variation graph, (ii) ratio of duplications, i.e., the number of additional edges inducing duplications in a walk of the graph, (iii) ratio of inversions, i.e., the proportion of inverted orientations within the set of duplications, and lastly (iv) the number of haplotypes that are input to subsequent founder set reconstruction. The latter are generated by performing random walks in the seed sequence's variation graph and retaining only those leading from source to sink. In doing so, our simulator does not report nor have knowledge of a true founder set. Our simulator, discussed in more detail in Appendix A.1, enables us to explore various parameterizations that match different situations in biological data.

One important point concerns co-optimality. Problems 1 and 2 do not guarantee a unique solution. In fact, the pool of co-optimal solutions is often large for both problems. One contributing factor to co-optimality are cycles that are shared across multiple haplotypes, because they can be integrated in different orders. Further, the solution does not provide any information that could enable one to generate all co-optimal solutions nor discern between them, making a measure of accuracy challenging, since there is no guarantee that the "correct" founder sequence(s) will be seen in any number of trials.

In addition to simulated data, we applied our methods on a biological data set from the human 1p36.13 locus described by Porubsky *et al.* [21] to demonstrate their computational capabilities in realistic instances.

3.2 Simulation Experiments

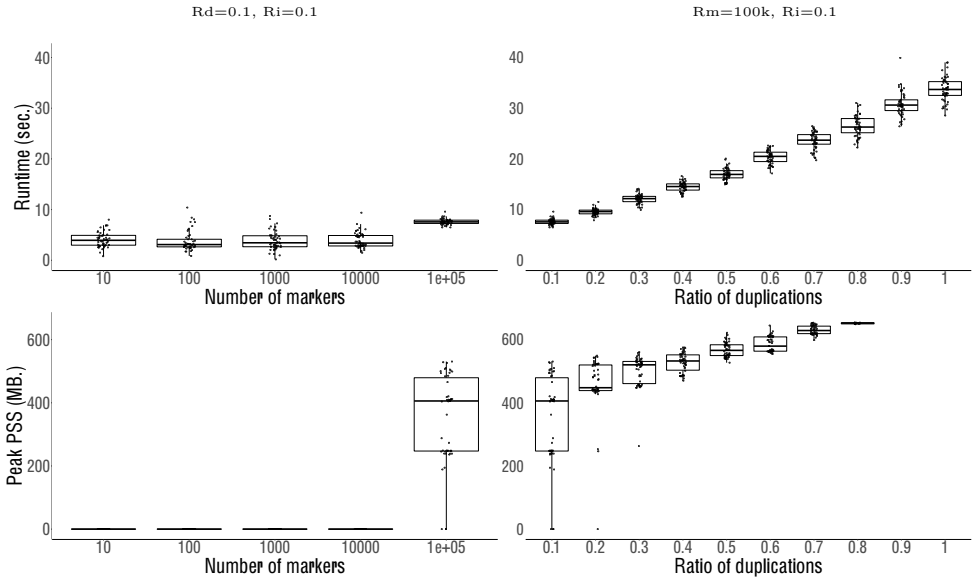


Figure 3 Problem 1, flow computational performance benchmarks. Runtime in seconds (upper panels) and peak PSS in Megabytes (lower panels), as a function of the number of markers (left) and of the ratio of duplications (right). For each experiment, the remaining parameters are fixed as indicated above. The abbreviations read as follows: *Nm*, number of markers; *Rd*, ratio of duplications; and *Ri*, ratio of inverted duplications.

To assess the impact of parameter configurations on the results, we ran a number of different experiments wherein all but one parameters are fixed. A reasonable choice of constants seemed to be 100 distinct markers, 10% of duplications, 10% of inversions and 10 haplotypes, motivated by our data on the 1p36.13 locus (8 markers, 68 haplotypes, 57% of duplications) and statistics compiled by Porubsky *et al.* [21] (6-7% duplications in the whole genome, <1% inversions).

Reduction in number of recombinations. To evaluate the efficacy of our solution to Problem 2, we compared the number of recombinations returned by Algorithm 1 to that in a solution obtained by our network flow algorithm for Problem 1. While the former is the immediate output of Algorithm 1, additional efforts needed to be made in order to retain the latter. In doing so, we estimate the number of recombinations in the flow solution by random assignment of corresponding segments in the original haplotype set and taking the one with the lowest number in 100k trials. Figure 2 summarizes the outcome of this experiment. Over all, Algorithm 1 found a solution with fewer recombinations in all instances but a few where Gurobi returned barely best-effort solutions after reaching the time limit of 30 minutes, all of which exhibited a gap of at least 100%. The parameter settings in those cases were extremal.

Across all experiments and with a fixed ratio of duplications, inversions and number of haplotypes, the mean estimated number of recombinations both in the initial founder set and after minimization increases linearly with the number of markers, by approximately 4.2 and 2.0 per 100 markers respectively, reaching circa 10 and 3.8 for 200 markers. Results for experiments with other variable parameters are shown in Suppl. Figure S1.

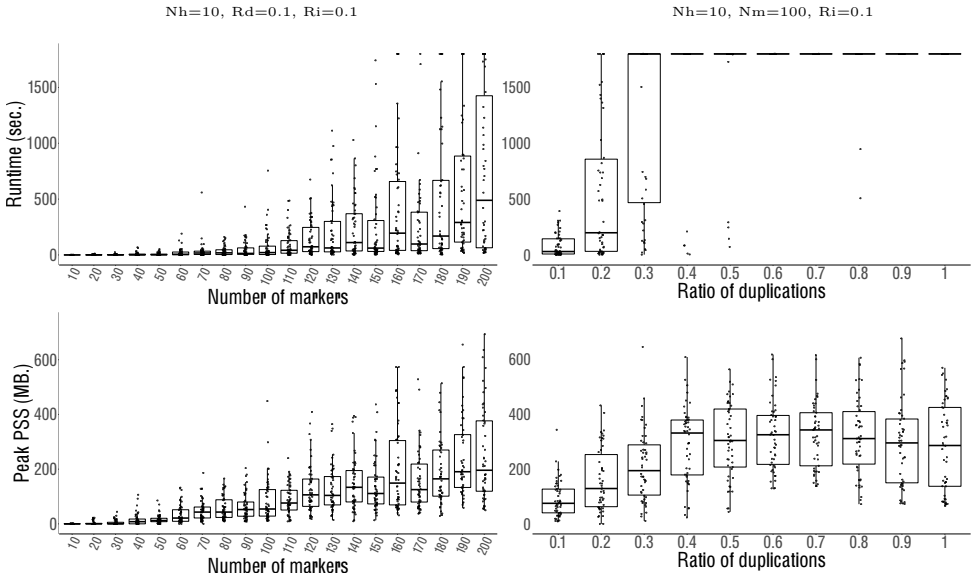


Figure 4 Problem 2, recombinations minimization performance benchmarks. Plots analogous to Figure 3. Runtime in seconds (upper panels) and peak PSS in Megabytes (lower panels), as a function of the number of markers (left) and of the ratio of duplications (right). For each experiment, the remaining parameters are fixed as indicated above. The abbreviations read as follows: *Nh*, number of haplotypes; *Nm*, number of markers; *Rd*, ratio of duplications; and *Ri*, ratio of inverted duplications.

Flow solution benchmark. Computing solutions with our network flow algorithm proved to be in almost all of our experiments near-instantaneous. By varying the number of distinct markers, the algorithm's performance begins to deteriorate only with very large instances beyond 100k distinct markers and becomes excruciating for instances above 1M markers. When varying other parameters, we fixed the number of distinct markers to 100k rather than 100. Under 100k markers, execution completes after a mean wall clock time of 3.4 ± 2.0 seconds. In 95% of all experiments, the solver's runtime was too short to make sufficient measurements for benchmarking memory usage; the maximum PSS for the remaining ones measured at 78MB. Over the 100k mark, both the graph size and duplication ratio begin to reduce performance, with an average runtime of 19.7 ± 8.7s. The ration of inversions on the other hand does not affect performance (Suppl. Figure S3). We measured peak memory consumption at 758MB across all conditions, which also occurred only at the very extremes of 100k distinct markers and a 100% ratio of duplications (Figure 3).

Recombination minimization benchmark. As shown previously, Algorithm 1 successfully reduces the number of recombinations in solutions to Problem 1. However, its runtime increases dramatically with only moderate increments of any but one parameter of our simulator, the ratio of inversions; it does not play any role in performance (Suppl. Figure S2). For the remaining three, going beyond instances of 200 distinct markers, 20% of duplications, or 40 haplotypes typically does not allow for the optimization to finish in a reasonable amount of time (Figure 4, Suppl. Figure S2). A similar but much less pronounced trend is seen with memory usage, which still remains relatively low. Peak memory usage was again observed at extreme parameter values with a PSS of 1072MB with 50 haplotypes.

3.3 Application: Locus 1p36.13

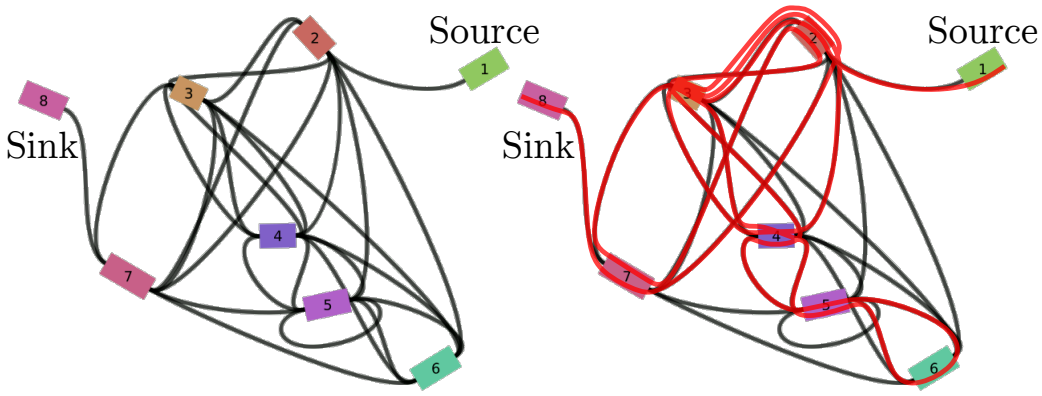


Figure 5 Graphical representation of the variation graph for the 1p36.13 locus data. On the left, a 2D plot rendered by Bandage [34]. Markers are represented as numbered colored rectangles, and the undirected edges connecting them as black curves. Markers 1 and 8 correspond respectively to the source and the sink of the graph. The right plot shows the walk through the graph corresponding to the sequence of haplotype `AFR-NA19036-h1`, a sample of African origin from our experimental data. The sample's sequence in the previously established notation is: $1234\overline{56543273243}278$.

We obtained data from 68 human haplotypes (two per 34 individuals) at the 1p36.13 locus from Porubsky *et al.* [21] and the T2T-CHM13 human reference sequence [18]. The sequences comprise only eight distinct markers, terminal markers included. The sequences are attributed to five super populations, out of which 18 are of African origin (AFR), 16 of Eastern Asian (EAS), 12 of Admixed American (AMR), 12 of European (EUR), and 10 are South Asian (SAS). Their variation graph is densely connected with 26 edges (Figure 5). The 68 haplotypes display a high degree of genetic diversity, with haplotype sequences differing in order, orientation, and copy number of the marker (Suppl. Table T1). Haplotype lengths in terms of the number of markers vary from 15 to 26, with a median of 19.

Our network flow algorithm determined that the data set can be generated from a single founder sequence. Our randomized algorithm for calculation of the minimum number of recombinations in a solution to Problem 1 asserted 15 recombinations after 1M trials, while Algorithm 1 obtained an optimal solution that revealed only 9 recombinations. Minimization completed in 60.3 seconds with a peak PSS of 225MB. Note that there exists multiple other co-optimal solutions; Suppl. Figure S4 is an illustration of one.

4 Conclusion

The advent of sequencing technology and genome assembly methodology to reconstruct full human genomes enables research into previously inaccessible segmental duplication loci. This exciting opportunity entails a demand for explanatory models that can infer evolutionary relationships and histories of complex repetitive genomic regions. In this work, we propose a model capable of explaining a broad range of balanced and unbalanced genome rearrangements. Our experiments on simulated data and on the 1p36.13 locus demonstrate that our algorithmic solutions to the founder set problem and the problem of minimizing recombinations in founder sets are capable of processing realistic instances.

Importantly, the model we are proposing is based on a molecular mechanism with a well-established role in shaping segmental duplication architecture. In our view, many past models of genome rearrangements have not sufficiently captured biological reality and there

is an important need for further research aiming to incorporate knowledge of molecular mechanisms into such models. For instance, we envision future models that additionally include mechanisms like non-homologous end joining (NHEJ) and mobile element insertions. Furthermore, actual rates at which NAHR occurs depend on factors like the length of the duplicated sequence, the sequence similarity, as well as the presence of specific sequence motifs. "Hidden" in our current approach in the construction of the variation graph, we aim to address and model these factors explicitly in future work.

References

1 Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1 edition, February 1993.

2 David A Bader, Bernard ME Moret, and Mi Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. In *1st International Workshop on Algorithms in Bioinformatics (WABI 2001)*, Algorithms in Bioinformatics, pages 365–376, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

3 Vineet Bafna and Pavel A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25(2):272–289, 1996.

4 Vineet Bafna and Pavel A. Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11(2):224–240, 1998.

5 Anne Bergeron, Julia Mixtacki, and Jens Stoye. A unifying view of genome rearrangements. In Philipp Bucher and Bernard M. E. Moret, editors, *6th International Workshop on Algorithms in Bioinformatics (WABI 2006)*, volume 4175 of *Algorithms in Bioinformatics*, pages 163–173, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

6 Leonard Bohnenkämper, Marília DV Braga, Daniel Doerr, and Jens Stoye. Computing the rearrangement distance of natural genomes. *Journal of Computational Biology*, 28(4):410–431, 2021.

7 Mark J P Chaisson, Ashley D Sanders, Xuefang Zhao, Ankit Malhotra, David Porubsky, Tobias Rausch, Eugene J Gardner, Oscar L Rodriguez, Li Guo, Ryan L Collins, Xian Fan, Jia Wen, Robert E Handsaker, Susan Fairley, Zev N Kronenberg, Xiangmeng Kong, Fereydoun Hormozdiari, Dillon Lee, Aaron M Wenger, Alex R Hastie, Danny Antaki, Thomas Anantharaman, Peter A Audano, Harrison Brand, Stuart Cantsilieris, Han Cao, Eliza Cerveira, Chong Chen, Xintong Chen, Chen-Shan Chin, Zechen Chong, Nelson T Chuang, Christine C Lambert, Deanna M Church, Laura Clarke, Andrew Farrell, Joey Flores, Timur Galeev, David U Gorkin, Madhusudan Gujral, Victor Guryev, William Haynes Heaton, Jonas Korlach, Sushant Kumar, Jee Young Kwon, Ernest T Lam, Jong Eun Lee, Joyce Lee, Wan-Ping Lee, Sau Peng Lee, Shantao Li, Patrick Marks, Karine Viaud-Martinez, Sascha Meiers, Katherine M Munson, Fabio C P Navarro, Bradley J Nelson, Conor Nodzak, Amina Noor, Sofia Kyriazopoulou-Panagiotopoulou, Andy W C Pang, Yunjiang Qiu, Gabriel Rosanio, Mallory Ryan, Adrian Stütz, Diana C J Spierings, Alistair Ward, Annemarie E Welch, Ming Xiao, Wei Xu, Chengsheng Zhang, Qihui Zhu, Xiangqun Zheng-Bradley, Ernesto Lowy, Sergei Yakneen, Steven McCarroll, Goo Jun, Li Ding, Chong Lek Koh, Bing Ren, Paul Flicek, Ken Chen, Mark B Gerstein, Pui-Yan Kwok, Peter M Lansdorp, Gabor T Marth, Jonathan Sebat, Xinghua Shi, Ali Bashir, Kai Ye, Scott E Devine, Michael E Talkowski, Ryan E Mills, Tobias Marschall, Jan O Korbel, Evan E Eichler, and Charles Lee. Multi-platform discovery of haplotype-resolved structural variation in human genomes. *Nat. Commun.*, 10(1):1784, April 2019.

8 Zanoni Dias and Joao Meidanis. Genome rearrangements distance by fusion, fission, and transposition is easy. In *spire*, pages 250–253. Citeseer, 2001.

9 Richard Durbin. Efficient haplotype matching and storage using the positional burrows–wheeler transform (pbwt). *Bioinformatics*, 30(9):1266–1272, 2014.

10 Peter Ebert, Peter A Audano, Qihui Zhu, Bernardo Rodriguez-Martin, David Porubsky, Marc Jan Bonder, Arvis Sulovari, Jana Ebler, Weichen Zhou, Rebecca Serra Mari, Feyza Yilmaz, Xuefang Zhao, Pinghsun Hsieh, Joyce Lee, Sushant Kumar, Jiadong Lin, Tobias Rausch, Yu Chen, Jingwen Ren, Martin Santamarina, Wolfram Höps, Hufsah Ashraf, Nelson T Chuang, Xiaofei Yang, Katherine M Munson, Alexandra P Lewis, Susan Fairley, Luke J Tallon, Wayne E Clarke, Anna O Basile, Marta Byrska-Bishop, André Corvelo, Uday S Evani, Tsung-Yu Lu, Mark J P Chaisson, Junjie Chen, Chong Li, Harrison Brand, Aaron M Wenger, Maryam Ghareghani, William T Harvey, Benjamin Raeder, Patrick Hasenfeld, Allison A Regier, Haley J Abel, Ira M Hall, Paul Flicek, Oliver Stegle, Mark B Gerstein, Jose M C Tubio, Zepeng Mu, Yang I Li, Xinghua Shi, Alex R Hastie, Kai Ye, Zechen Chong, Ashley D Sanders, Michael C Zody, Michael E Talkowski, Ryan E Mills, Scott E Devine, Charles Lee, Jan O Korbel, Tobias Marschall, and Evan E Eichler. Haplotype-resolved diverse human genomes and integrated analysis of structural variation. *Science*, February 2021.

11 LLC Gurobi Optimization. Gurobi optimizer reference manual, 2019. URL: `http://www.gurobi.com`.

12 Heng Li, Xiaowen Feng, and Chong Chu. The design and construction of reference pangenome graphs with minigraph. *Genome biology*, 21(1):1–19, 2020.

13 Tomas Marques-Bonet, Santhosh Girirajan, and Evan E Eichler. The origins and impact of primate segmental duplications. *Trends Genet.*, 25(10):443–454, October 2009.

14 Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34(3), pages 103–104. ACM, 2014.

15 Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B Hall, Christopher H Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O Twardziok, Alexander Kanitz, et al. Sustainable data analysis with snakemake. *F1000Research*, 10, 2021.

16 Tuukka Norri, Bastien Cazaux, Saska Dönges, Daniel Valenzuela, and Veli Mäkinen. Founder reconstruction enables scalable and seamless pangenomic analysis. *Bioinformatics*, 37(24):4611–4619, July 2021.

17 Tuukka Norri, Bastien Cazaux, Dmitry Kosolobov, and Veli Mäkinen. Minimum Segmentation for Pan-genomic Founder Reconstruction in Linear Time. In *18th International Workshop on Algorithms in Bioinformatics (WABI 2018)*, volume 113 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:15, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

18 Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V Bzikadze, Alla Mikheenko, Mitchell R Vollger, Nicolas Altemose, Lev Uralsky, Ariel Gershman, Sergey Aganezov, Savannah J Hoyt, Mark Diekhans, Glennis A Logsdon, Michael Alonge, Stylianos E Antonarakis, Matthew Borchers, Gerard G Bouffard, Shelise Y Brooks, Gina V Caldas, Nae-Chyun Chen, Haoyu Cheng, Chen-Shan Chin, William Chow, Leonardo G de Lima, Philip C Dishuck, Richard Durbin, Tatiana Dvorkina, Ian T Fiddes, Giulio Formenti, Robert S Fulton, Arkarachai Fungtammasan, Erik Garrison, Patrick G S Grady, Tina A Graves-Lindsay, Ira M Hall, Nancy F Hansen, Gabrielle A Hartley, Marina Haukness, Kerstin Howe, Michael W Hunkapiller, Chirag Jain, Miten Jain, Erich D Jarvis, Peter Kerpedjiev, Melanie Kirsche, Mikhail Kolmogorov, Jonas Korlach, Milinn Kremitzki, Heng Li, Valerie V Maduro, Tobias Marschall, Ann M McCartney, Jennifer McDaniel, Danny E Miller, James C Mullikin, Eugene W Myers, Nathan D Olson, Benedict Paten, Paul Peluso, Pavel A Pevzner, David Porubsky, Tamara Potapova, Evgeny I Rogaev, Jeffrey A Rosenfeld, Steven L Salzberg, Valerie A Schneider, Fritz J Sedlazeck, Kishwar Shafin, Colin J Shew, Alaina Shumate, Ying Sims, Arian F A Smit, Daniela C Soto, Ivan Sović, Jessica M Storer, Aaron Streets, Beth A Sullivan, Françoise Thibaud-Nissen, James Torrance, Justin Wagner, Brian P Walenz, Aaron Wenger, Jonathan M D Wood, Chunlin Xiao, Stephanie M Yan, Alice C Young, Samantha Zarate, Urvashi Surti, Rajiv C McCoy, Megan Y Dennis, Ivan A Alexandrov, Jennifer L Gerton, Rachel J O'Neill, Winston Timp, Justin M Zook, Michael C Schatz, Evan E Eichler, Karen H Miga, and Adam M Phillippy. The complete sequence of a human genome. *Science*, 376(6588):44–53, April 2022.

19 Laxmi Parida, Marta Melé, Francesc Calafell, Jaume Bertranpetit, and Genographic Consortium. Estimating the ancestral recombinations graph (arg) as compatible networks of snp patterns. *Journal of Computational Biology*, 15(9):1133–1153, 2008.

20 David Porubsky, Peter Ebert, Peter A Audano, Mitchell R Vollger, William T Harvey, Pierre Marijon, Jana Ebler, Katherine M Munson, Melanie Sorensen, Arvis Sulovari, Marina Haukness, Maryam Ghareghani, Peter M Lansdorp, Benedict Paten, Scott E Devine, Ashley D Sanders, Charles Lee, Mark J P Chaisson, Jan O Korbel, Evan E Eichler, Tobias Marschall, and Human Genome Structural Variation Consortium. Fully phased human genome assembly without parental data using single-cell strand sequencing and long reads. *Nat. Biotechnol.*, December 2020.

21 David Porubsky, Wolfram Höps, Hufsah Ashraf, PingHsun Hsieh, Bernardo Rodriguez-Martin, Feyza Yilmaz, Jana Ebler, Pille Hallast, Flavia Angela Maria Maggiolini, William T. Harvey, Barbara Henning, Peter A. Audano, David S. Gordon, Peter Ebert, Patrick Hasenfeld, Eva Benito, Qihui Zhu, Human Genome Structural Variation Consortium (HGSVC), Charles Lee, Francesca Antonacci, Matthias Steinrücken, Christine R. Beck, Ashley D. Sanders, Tobias Marschall, Evan E. Eichler, and Jan O. Korbel. Recurrent inversion polymorphisms in humans associate with genetic instability and genomic disorders. *Cell*, 2022.

22 Pasi Rastas and Esko Ukkonen. Haplotype inference via hierarchical genotype parsing. In Raffaele Giancarlo and Sridhar Hannenhalli, editors, *7th International Workshop on Algorithms in Bioinformatics (WABI 2007)*, Algorithms in Bioinformatics, pages 85–97, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

23 Mikko Rautiainen and Tobias Marschall. MBG: Minimizer-based sparse de Bruijn Graph construction. *Bioinformatics*, 37(16):2476–2478, January 2021.

24 Andrea Roli, Stefano Benedettini, Thomas Stützle, and Christian Blum. Large neighbourhood search algorithms for the founder sequence reconstruction problem. *Computers & Operations Research*, 39(2):213–224, 2012.

25 Andrea Roli and Christian Blum. Tabu search for the founder sequence reconstruction problem: A preliminary study. In Sigeru Omatu, Miguel P. Rocha, José Bravo, Florentino Fernández, Emilio Corchado, Andrés Bustillo, and Juan M. Corchado, editors, *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, pages 1035–1042, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

26 Russell Schwartz, Andrew G Clark, and Sorin Istrail. Methods for inferring block-wise ancestral history from haploid sequences. In *2nd International Workshop on Algorithms in Bioinformatics (WABI 2002)*, Algorithms in Bioinformatics, pages 44–59, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

27 Fritz J Sedlazeck, Hayan Lee, Charlotte A Darby, and Michael C Schatz. Piercing the dark matter: bioinformatics of long-range sequencing and mapping. *Nat. Rev. Genet.*, March 2018.

28 Mingfu Shao, Yu Lin, and Bernard M. E. Moret. An exact algorithm to compute the double-cut-and-join distance for genomes with duplicate genes. *Journal of Computational Biology*, 22(5):425–435, 2015. doi:10.1089/cmb.2014.0096.

29 Krister M Swenson, Paul Guertin, Hugo Deschênes, and Anne Bergeron. Reconstructing the modular recombination history of staphylococcus aureus phages. *BMC bioinformatics*, 14(15):1–9, 2013.

30 Esko Ukkonen. Finding founder sequences from a set of recombinants. In *2nd International Workshop on Algorithms in Bioinformatics (WABI 2002)*, Algorithms in Bioinformatics, pages 277–286, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

31 Mitchell R Vollger, Xavi Guitart, Philip C Dishuck, Ludovica Mercuri, William T Harvey, Ariel Gershman, Mark Diekhans, Arvis Sulovari, Katherine M Munson, Alexandra P Lewis, Kendra Hoekzema, David Porubsky, Ruiyang Li, Sergey Nurk, Sergey Koren, Karen H Miga, Adam M Phillippy, Winston Timp, Mario Ventura, and Evan E Eichler. Segmental duplications and their variation in a complete human genome. *Science*, 376(6588):eabj6965, April 2022.

32 Maria Emilia MT Walter, Zanoni Dias, and Joao Meidanis. Reversal and transposition distance of linear chromosomes. In *Proceedings. String Processing and Information Retrieval: A South American Symposium (Cat. No. 98EX207)*, pages 96–102. IEEE, 1998.

33 Ting Wang, Lucinda Antonacci-Fulton, Kerstin Howe, Heather A Lawson, Julian K Lucas, Adam M Phillippy, Alice B Popejoy, Mobin Asri, Caryn Carson, Mark J P Chaisson, Xian Chang, Robert Cook-Deegan, Adam L Felsenfeld, Robert S Fulton, Erik P Garrison, Nanibaa' A Garrison, Tina A Graves-Lindsay, Hanlee Ji, Eimear E Kenny, Barbara A Koenig, Daofeng Li, Tobias Marschall, Joshua F McMichael, Adam M Novak, Deepak Purushotham, Valerie A Schneider, Baergen I Schultz, Michael W Smith, Heidi J Sofia, Tsachy Weissman, Paul Flicek, Heng Li, Karen H Miga, Benedict Paten, Erich D Jarvis, Ira M Hall, Evan E Eichler, David Haussler, and Human Pangenome Reference Consortium. The human pangenome project: a global resource to map genomic diversity. *Nature*, 604(7906):437–446, April 2022.

34 Ryan R. Wick, Mark B. Schultz, Justin Zobel, and Kathryn E. Holt. Bandage: interactive visualization of de novo genome assemblies. *Bioinformatics*, 31(20):3350–3352, June 2015. doi:10.1093/bioinformatics/btv383.

35 Yufeng Wu and Dan Gusfield. Improved algorithms for inferring the minimum mosaic of a set of recombinants. In Bin Ma and Kaizhong Zhang, editors, *Combinatorial Pattern Matching*, pages 150–161, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

36 Sophia Yancopoulos, Oliver Attie, and Richard Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16):3340–3346, 2005. doi:10.1093/bioinformatics/bti535.

37 Xuefang Zhao, Ryan L Collins, Wan-Ping Lee, Alexandra M Weber, Yukyung Jun, Qihui Zhu, Ben Weisburd, Yongqing Huang, Peter A Audano, Harold Wang, Mark Walker, Chelsea Lowther, Jack Fu, Mark B Gerstein, Scott E Devine, Tobias Marschall, Jan O Korbel, Evan E Eichler, Mark J P Chaisson, Charles Lee, Ryan E Mills, Harrison Brand, and Michael E Talkowski. Expectations and blind spots for structural variation detection from long-read assemblies and short-read genome sequencing technologies. *Am. J. Hum. Genet.*, March 2021.

A Appendix

A.1 Methods for the Simulation Experiments

The simulation experiments were carried out with the help of a new tool developed specifically for it. It generates a single *seed* haplotype, which then serves to construct a variation graph, on which random walks from source to sink are made to generate new haplotypes. The seed haplotype is initially a sequence of unique markers. A rate of duplications determines the number of duplications to add. For each duplication, the marker to duplicate and the position of insertion are sampled at random. The orientation of the duplicates is sampled according to a ratio of inversions. Next, the seed's *variation graph* is built based on its sequence, represented as a walk through the graph. Finally, a given number of unique haplotypes is generated by performing random walks from source to sink in the graph. Essentially, the simulator starts from seed sequence, then generates an observable set of haplotypes and their graph. Because the walks are random, edges not covered by any of the new haplotypes must be pruned in order to respect the properties of a variation graph. The number of markers and haplotypes, and the ratios of duplication and inversion are the simulation parameters. The ratio of duplications (resp. of inversions) is defined as the ratio of the number of duplications to the number of nodes (resp. number of inversions to the number of duplications). All simulation experiments were carried out by running 50 simulations per parameter set, then applying the solutions of Problems 1 and 2 over the generated graph and haplotype set. The simulation experiments are implemented as *Snakemake* [15] workflows which also provide

the benchmarking results then used for evaluation. The data and workflows for the 1p36.13 locus, as well as all simulation experiments are available in the github repository[7] under the `examples` directory.

A.2 Supplementary Figures and Tables

In the following figures, for each of the simulation experiments, performance is measured with regards to a range of values of a single parameter. All others are fixed to a constant value indicated above the given plot. They are labeled as follows: *Nm*, number of markers; *Nh*, number of haplotypes; *Rd*, ratio of duplications; and *Ri*, ratio of inverted duplications. Runtime is measured in seconds of wall clock time, and peak memory usage as the peak proportional set size (PSS) in Megabytes.

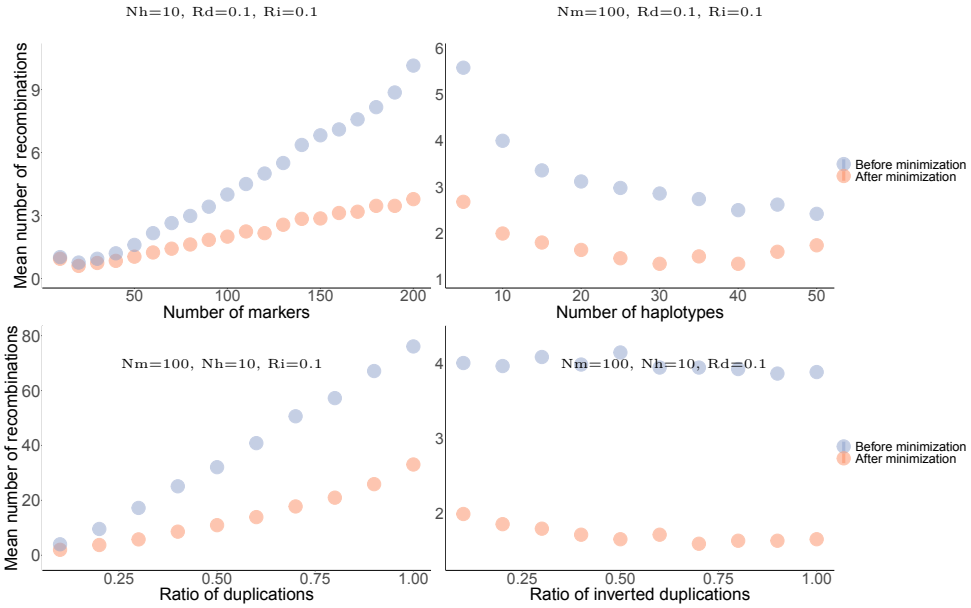


Figure S1 Reduction in the number of recombinations following minimization. The plots show the total number of recombinations before (blue dots) and after (red dots) minimization, as a function of each simulation parameter.

[7] `https://github.com/marschall-lab/hrfs`

Table T1 Haplotype marker sequences used in the 1p36.13 locus analysis, sorted alphabetically. The haplotype labeled *CHM13* is the provided reference. The sequences are in GFA Path format, where > corresponds to traversal in forward direction, and < in reverse direction.

Haplotype	Oriented marker sequence
CHM13	>1>2>3>4>2>3>4>5>6<5<4<3<2>7>8
AFR-HG02011-h1	>1>2>3>4>2>3>4<6<5<4<3<2<4<3<2>7>8
AFR-HG02011-h2	>1>2>3<7<6>3>4>5<6<5<4<3<2<4<3<2>7>8
AFR-HG02587-h1	>1>2>3>4>2>3>4<6<4<3<2>6<5<4<3<2<4<3<2>7>8
AFR-HG02587-h2	>1>2>3>4>2>3<7<6>2>3>4>5<6<5<4<3<2>7>8
AFR-HG03065-h1	>1>2>3>4>5<6>3>4>5>6<4<3<2<4<3>6<4<3<2>7>8
AFR-HG03065-h2	>1>2>3>4>2>3<7>3>4>6<4<3<2<4<3>6<4<3<2>7>8
AFR-HG03371-h1	>1>2>3>4>2>3>4<6>3>4>5<6<4<3<2>6<4<3<2<4<3<2>7>8
AFR-HG03371-h2	>1>2>3>4>2>3<7>3>4>5>6<5<4<3<4<3<2>7>8
AFR-NA19036-h1	>1>2>3<4>5>6<5<4<3<2>7<3<2<4<3<2>7>8
AFR-NA19036-h2	>1>2>3<7>3>4>5>6<4<3<2<4<3>6<4<3<2>7>8
AFR-NA19238-h1	>1>2>3>4>2>3>4<6<4<3<2>6<5<4<3<2<4<3<2>7>8
AFR-NA19238-h2	>1>2>3>4>5<6>2>3>4>5<6<5<4<3<2<4<3<2>7>8
AFR-NA19239-h1	>1>2>3<7>2>3>4>5>6<5<4<3<2>7>8
AFR-NA19239-h2	>1>2>3>4>2>3>4>2>3<7<6>2>3>4>5<6<5<4<3<2<4<3<2>7>8
AFR-NA19240-h1	>1>2>3>4>5<6>2>3>4>5<6<5<4<3<2<4<3<2>7>8
AFR-NA19240-h2	>1>2>3<7>2>3>4>5>6<5<4<3<2>7>8
AFR-NA19983-h1	>1>2>3>4>2>3>4>5>6<5<4<3<2>7<3<2>7>8
AFR-NA19983-h2	>1>2>3>4>2>3>4<6>2>3>4>5<6>2>3>4>6<5<4<3<2<4<3<2>7>8
AMR-GM19650-h1	>1>2>3>4>5<6<5<4<3<2<4<3<2>7<3<2<4<3<2>7>8
AMR-GM19650-h2	>1>2>3>4>2>3>4>5>6<5<4<3<2>7<3<2<4<3<2>7>8
AMR-HG00731-h1	>1>2>3<7>2>3>4>5>6<5<4<3<2>7>8
AMR-HG00731-h2	>1>2>3<7>2>3>4>5<6<5<4<3<2>6<4<3<2>7>8
AMR-HG00732-h1	>1>2>3>4>5>6<4<3<2>6<5<4<3<2>7>8
AMR-HG00732-h2	>1>2>3>4>2>3>4>2>3>4>5>6<5<4<3<2<4<3<2>7>8
AMR-HG00733-h1	>1>2>3<7>2>3>4>5>6<5<4<3<2>7>8
AMR-HG00733-h2	>1>2>3>4>5>6<4<3<2>6<5<4<3<2>7>8
AMR-HG01114-h1	>1>2>3<7>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
AMR-HG01114-h2	>1>2>3>4>2>3>4>5<6<5<4<3<2>6<5<4<3<2>7>8
AMR-HG01573-h1	>1>2>3>4>5>6<5<4<3<2>6<5<4<3<2<4<3<2>7>8
AMR-HG01573-h2	>1>2>3<7>2>3>4>5<4<3<2<5<4<3<2>7>8
EAS-GM00864-h1	>1>2>3<7>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
EAS-GM00864-h2	>1>2>3>4>2>3>4>5>6<5<5<5<4<3<2>7>8
EAS-GM18939-h1	>1>2>3<7>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
EAS-GM18939-h2	>1>2>3<7>2>3>4>6<5<4<3<2>6<4<3<2>7>8
EAS-HG00512-h1	>1>2>3<7>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
EAS-HG00512-h2	>1>2>3>4>5<6>2>3>4>5>6<5<4<3<2>6<4>3>2>7>8
EAS-HG00513-h1	>1>2>3<7>2>3>4>5<6>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
EAS-HG00513-h2	>1>2>3<7>2>3>4>5<6>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
EAS-HG00514-h1	>1>2>3>4>5<6>2>3>4>5>6<5<4<3<2>6<4>3>2>7>8
EAS-HG00514-h2	>1>2>3<7>2>3>4>5>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
EAS-HG01596-h1	>1>2>3<7>2>3>4>5>6<5<4<3<2>7>8
EAS-HG01596-h2	>1>2>3<7>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
EAS-HG02018-h1	>1>2>3<7>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
EAS-HG02018-h2	>1>2>3<7>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
EAS-NA18534-h1	>1>2>3<7>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
EAS-NA18534-h2	>1>2>3>4>5>2>3>4>5<6<5<4<3<2>6<5<4<3<2>7>8
EUR-GM12329-h1	>1>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
EUR-GM12329-h2	>1>2>3>4>2>3>4>5>6<5<4<3<2>7>8
EUR-GM20509-h1	>1>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
EUR-GM20509-h2	>1>2>3<7<6>2>3>4>5>6<5<4<3<2>6>7<3<2<4<3<2>7>8
EUR-HG00096-h1	>1>2>3<7>2>3>4>5<6<5<4<3<2>6<4<3<2>7>8
EUR-HG00096-h2	>1>2>3>4>5>2>3>4<6<5<4<3<2>7>8
EUR-HG00171-h1	>1>2>3>4>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
EUR-HG00171-h2	>1>2>3<7>5>2>3>4>5>2>3>4>6<5<4<3<2>7>8
EUR-HG01505-h1	>1>2>3<7>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
EUR-HG01505-h2	>1>2>3>4>2>3>4>5>2>3>4>5<6<5<4<3<2>7>8
EUR-NA12878-h1	>1>2>3>4>2>3>4>5>6<5<4<3<2>6<5<4<3<2<4<3<2>7>8
EUR-NA12878-h2	>1>2>3>4>2>3>4>5>6<5<4<3<2>6<5<4<3<2>7>8
SAS-GM20847-h1	>1>2>3>4>5<6<5<4<3<2>6<5<4<3<2>6<5<4<3<2>7>8
SAS-GM20847-h2	>1>2>3>4>2>3>4>5>6<5<5<4<3<2>6<4<3<2>7>8
SAS-HG02492-h1	>1>2>3>4>2>3>4>5>6<5<4<3<2>6<5<4<3<2<4<3<2>7>8
SAS-HG02492-h2	>1>2>3>4>2>3>4>2>3>4>5>6<4<3<2>6<5<4<3<2<4<3<2>7>8
SAS-HG03009-h1	>1>2>3<7>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8
SAS-HG03009-h2	>1>2>3>4>5>2>3>4>5<6<5<4<3<2>7>8
SAS-HG03683-h1	>1>2>3>4>2>3>4>5>6<5<4<3<2>7>8
SAS-HG03683-h2	>1>2>3>4>5<6>2>3>4>5>6<5<4<3<2>6<4>3>2>7>8
SAS-HG03732-h1	>1>2>3>4>2>3>4>5>6<5<4<3<2>7>8
SAS-HG03732-h2	>1>2>3>4>5>6<5<4<3<2>6<4<3<2>7>8

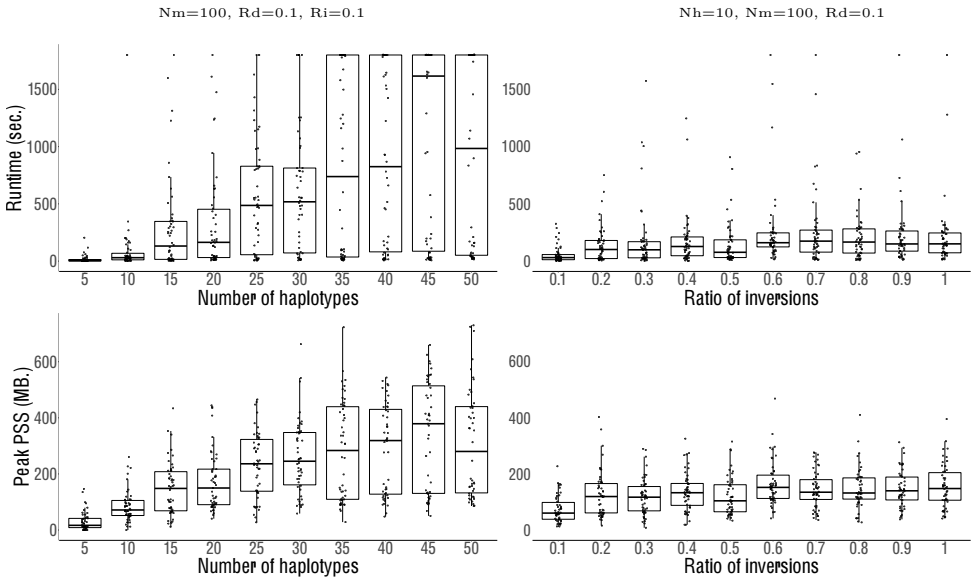


Figure S2 Number of recombinations minimization benchmarks. Runtime (upper panels) and peak PSS (lower panels) as a function of the number of haplotypes (left) and the ratio of inverted duplications (right).

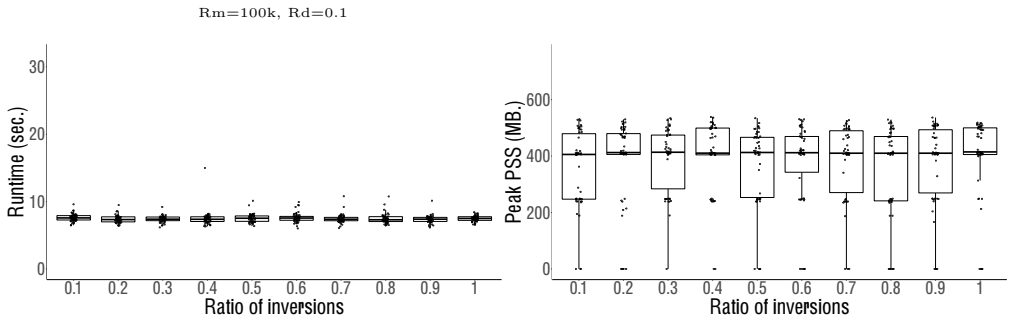


Figure S3 Flow computation performance with a variable ratio of inversions. Runtime (left) and memory usage (right) as a function of this parameter.

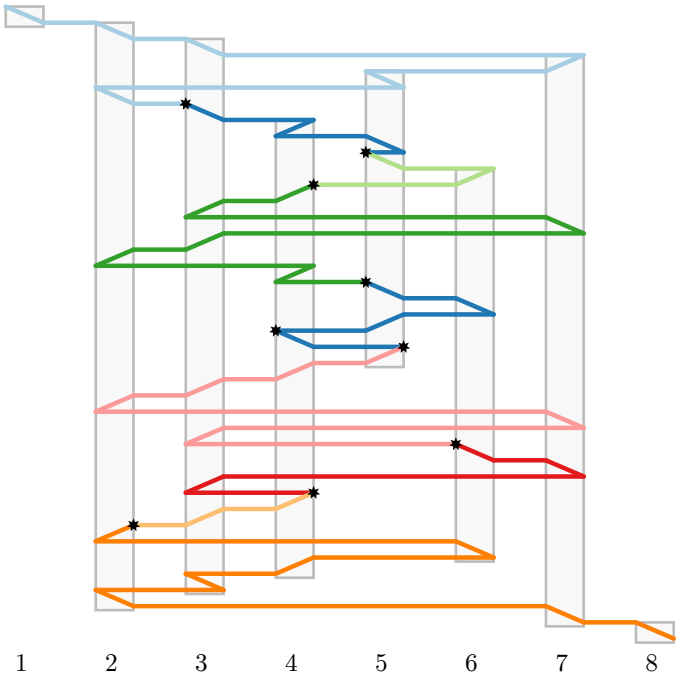


Figure S4 Visualization of a solution to the minimization problem on the 1p36.13 locus. The gray bars correspond to the graph's nodes, labeled 1 to 8. The founder sequence (>1>2>3<7>5>2>3<4>5>5<6<4<3>7<3<2<4>5>6<5>4<5<4<3<2>7<3>6>7<3<4<3<2>6<4>3>2>7>8) is traced from top to bottom. A slanted line indicates the underlying node being traversed; if slanted rightwards, traversal is in forward direction, and if slanted leftwards, traversal is in reverse direction. Colors correspond to different haplotypes. The haplotype sequence is: *EUR-HG00171-h2, AFR-NA19036-h1, SAS-GM20847-h2, AFR-HG03065-h2, AFR-NA19036-h1, AFR-NA19036-h1, AMR-HG01573-h2, AFR-HG02011-h2, AFR-HG03371-h2, SAS-HG03683-h2*. Recombinations are marked with a star.

Automated Design of Dynamic Programming Schemes for RNA Folding with Pseudoknots

Bertrand Marchand
LIX (UMR 7161), Ecole Polytechnique, Institut Polytechnique de Paris, France
LIGM, CNRS, Univ Gustave Eiffel, F77454 Marne-la-vallée France

Sebastian Will
LIX (UMR 7161), Ecole Polytechnique, Institut Polytechnique de Paris, France

Sarah J. Berkemer
LIX (UMR 7161), Ecole Polytechnique, Institut Polytechnique de Paris, France

Laurent Bulteau
LIGM, CNRS, Univ Gustave Eiffel, F77454 Marne-la-vallée France

Yann Ponty[1]
LIX (UMR 7161), Ecole Polytechnique, Institut Polytechnique de Paris, France

Abstract

Despite being a textbook application of dynamic programming (DP) and routine task in RNA structure analysis, RNA secondary structure prediction remains challenging whenever pseudoknots come into play. To circumvent the NP-hardness of energy minimization in realistic energy models, specialized algorithms have been proposed for restricted conformation classes that capture the most frequently observed configurations.

While these methods rely on hand-crafted DP schemes, we generalize and fully automatize the design of DP pseudoknot prediction algorithms. We formalize the problem of designing DP algorithms for an (infinite) class of conformations, modeled by (a finite number of) fatgraphs, and automatically build DP schemes minimizing their algorithmic complexity. We propose an algorithm for the problem, based on the tree-decomposition of a well-chosen representative structure, which we simplify and reinterpret as a DP scheme. The algorithm is fixed-parameter tractable for the tree-width tw of the fatgraph, and its output represents a $\mathcal{O}(n^{tw+1})$ algorithm for predicting the MFE folding of an RNA of length n.

Our general framework supports general energy models, partition function computations, recursive substructures and partial folding, and could pave the way for algebraic dynamic programming beyond the context-free case.

2012 ACM Subject Classification Applied computing → Computational biology; Theory of computation → Dynamic programming

Keywords and phrases RNA folding, treewidth, dynamic programming

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.7

Related Version *Full Version*: `Preprint, including proofs and supplementary material`

Supplementary Material *Software (Source Code)*: `https://gitlab.inria.fr/bmarchan/auto-dp`

Funding This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 101029676, and from the French-Austrian PaRNAssus project (ANR-19-CE45-0023; I 4520-N) supported by the ANR/FWF agencies.

[1] To whom correspondence should be addressed

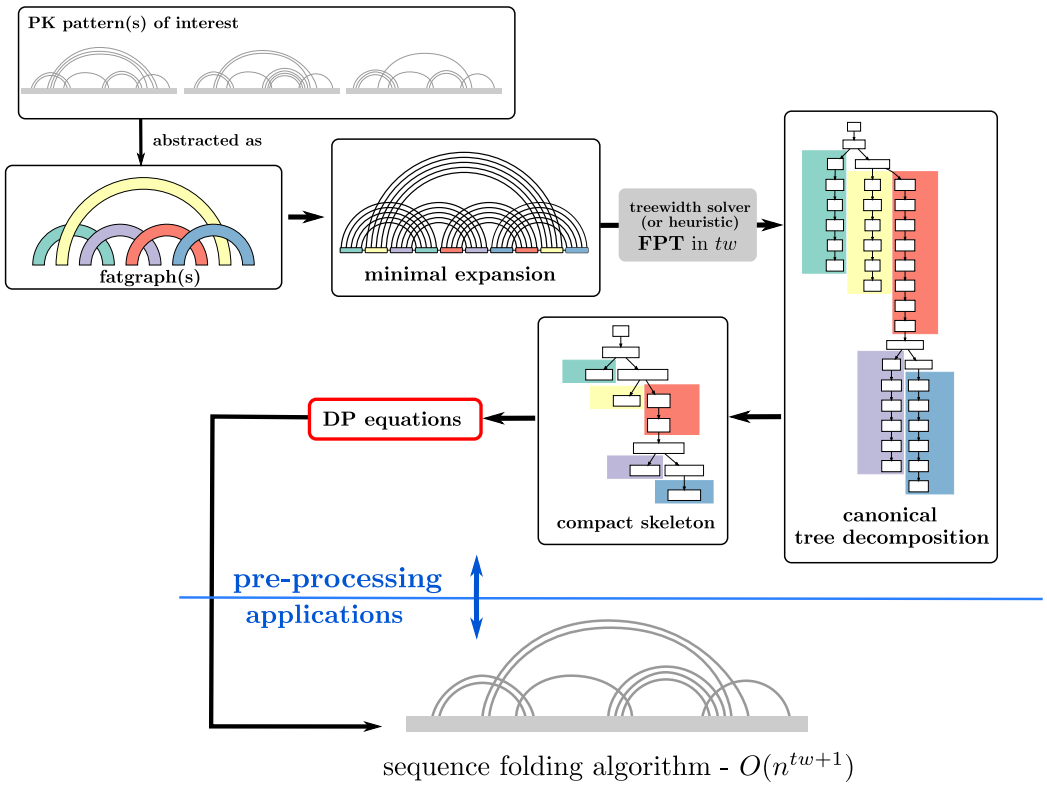


Figure 1 Given a finite number of arbitrary fatgraphs, a dynamic programming scheme for folding (restricted to the family of structures specified by the fatgraphs) is derived from canonical tree decompositions of minimal representative expansions of the helices, for each fatgraph. The workflow gives an overview of the steps of the algorithm. Each step is described in more details in the subsequent sections and figures: see Figure 2 for fatgraphs, Figure 8 and Section 3 for a detailed version of the canonical tree decomposition, Figure 5 for a detailed view of the compact skeleton of the tree decomposition.

1 Introduction

The function of non-coding RNAs is, to a large extent, determined by their structure. Structure prediction algorithms therefore play a crucial role in (bio-)medical and pharmaceutical applications. The basis to determine more complex 3D structures of RNA molecules is set by first accurately predicting their 2D or secondary structures. There exist various RNA folding algorithms that predict an optimal secondary structure as *minimum free energy structure* of the given RNA sequence in suitable thermodynamic models. In the most frequently used methods, this optimization is performed efficiently by a dynamic programming (DP) algorithm, e.g. `mfold` [47], `RNAfold` [23], `RNAstructure` [36]. A recent alternative to predictions based on experimentally determined energy parameters are machine learning approaches that train models on known secondary structures, e.g., `CONTRAfold` [15], `ContextFold` [46], `MXfold2` [40].

However, the most frequently used algorithms (including all of the above ones) optimize solely over pseudoknot-free structures [44], which do not contain crossing base pairs. Although pseudoknots appear in many RNA secondary structures, they have been omitted by initial

prediction algorithms due to their computational complexity [1], and the difficulty to score individual conformations [9]. Nevertheless, many algorithms have been proposed to predict at least certain pseudoknots. These methods are either based on exact DP algorithms such as `pknots-RE` [39], `NUPACK` [14], `gfold` [33], `Knotty` [21] or they use heuristics that don't guarantee exact solutions, e.g., `HotKnots` [35], `IPknot` [41, 40], `Hfold` [20].

Due to the hardness of PK prediction, efficient exact DP algorithms are necessarily restricted to certain categories of pseudoknotted structures. The underlying DP schemes are designed manually, guided by design to either i) support structures that are frequently observed in experimentally resolved structures (declarative categories); or ii) support the largest possible set of conformations, while remaining within a certain complexity (complexity-driven). For most categories, essentially declarative ones, there exists one or several helix arrangements, either observed in experimentally-determined structures or implicitly characterized by graph-theoretical properties (3 non-crossing [34], topologically bounded [33]) that need to be captured. A detailed overview of pseudoknot categories is given in [27]. Similar situations occur for RNA-RNA interactions [2], possibly including several RNA molecules. Interestingly, when more than two RNA strands are considered, existing algorithms restrict the joint conformation to crossing-free interactions [16], further motivating the design of algorithms beyond the case of pseudoknot-free secondary structures.

In this work, we describe classes of pseudoknotted structures as fatgraphs [19, 33, 22, 30], an abstraction of RNA conformations related to RNA shapes [17] or shadows [34, 33]. We formalize the principles underlying the design of DP folding algorithms including pseudoknots, and, at the same time, give a formulation of the computational problem based on the design of DP algorithms. We show how to leverage tree-decompositions, computed on a minimal expansion of the input fatgraph, to automatically derive DP schemes that use as little indices as possible. Our algorithm can be interpreted as a generalization of the algorithms underlying `LiCoRNA` [38] and `gfold` [33] and we propose a parameterized algorithm based on the treewidth (tw) of the underlying fatgraph.

In Section 2, we state our problem and define its input structure abstraction, the fatgraph. Then, we describe helix expansions of the fatgraph and their tree decompositions (Section 3). By minimal helix expansions and a derivation of the tree decomposition to its canonical form, we automatically derive a DP scheme for the folding of pseudoknotted structures (Section 4), using a number of indices equal to the treewidth. Figure 1 outlines the fundamental algorithm. Section 5, discusses extensions to combine multiple fatgraphs, include recursive substructure, and cover realistic energy models.

2 Definitions and main result

We define an *RNA sequence* S as a word of length n over the nucleotides A, C, G and U; moreover an *RNA secondary structure* (potentially, with pseudoknots) ω of S as a set of *base pairs* (i, j) between sequence positions i and j (in $1, ..., n$), such that there is at most one base pair incident to each position. A *diagram* is a graph of nodes 1,...,n (the positions), connecting consecutive positions by directed edges $(i, i+1)$ and moreover connecting positions by arcs, visualizing the *arc-annotation* of the sequence. Typically this is represented drawing the backbone linearly and the arcs on top. RNA secondary structures are naturally interpreted as diagrams.

One of our central concerns is the crossing configuration of arcs in a diagram. We define two arcs (i, j) and (i', j') in a diagram as *crossing* iff $i < i' < j < j'$ or $i' < i < j' < j$. Naturally, this leads to the notion of a conflict graph consisting of all the arcs of a diagram

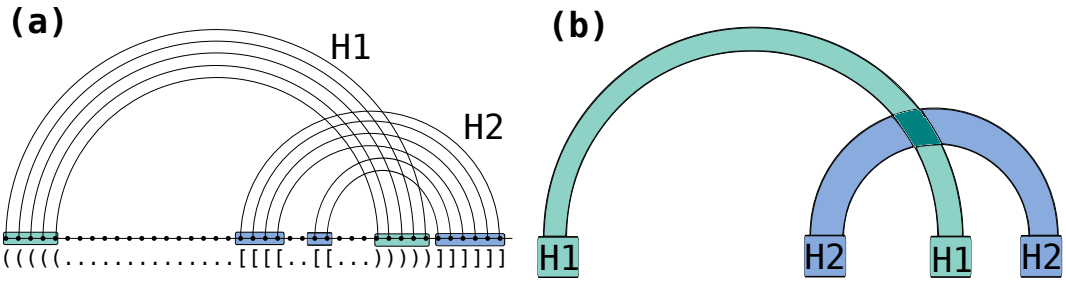


Figure 2 (a) Diagram of a secondary structure with two crossing helices (H1 green, H2 blue). (b) fatgraph corresponding to the above structure such that helices are collapsed into bands and form the shadow of the structure.

and connecting crossing arcs by a conflict edge. Given a potentially conflicted set of base pairs, the associated *RNA structure graph* is the diagram consisting of one vertex per nucleotide, backbone links, and one arc per base pair.

A *fatgraph* [19, 33, 22, 30] is an abstraction of a family of pseudoknotted RNA structures displaying a specific conflict structure. It is typically represented as a *band diagram* (see Figure 1 and Figure 2), in which each band may represent a *helix* of arbitrary size, including bulges. An arc-annotation is said to be an *expansion* of a fatgraph if collapsing nested arcs and contracting isolated bases yields the band diagram of a fatgraph. Given a finite number of fatgraphs, we say a structure is a *recursive expansion* of these fatgraphs if decomposing the structure into conflict-connected components, collapsing nested arcs and contracting isolated bases only yields members of the given fatgraph set. For the purpose of this presentation (where we do not explicitly study structure topology), we moreover identify fatgraphs with their diagrams.

To make the connection to `gfold` [33] explicit, recursive expansions of fatgraphs are equivalently understood in terms of the shadows of a structure. The shadow of an RNA structure (or equivalently, its diagram) is defined in [33] as the diagram obtained by, firstly, removing all unpaired bases and non-crossing structures and, secondly, contracting all stacks (i.e. pairs of arcs between directly consecutive positions) to single arcs. Then, the class of recursive expansions of a set of input fatgraphs Γ is the class of structures, where the shadows of their conflict-connected components are in Γ.

In this paper, we consider a class of RNA folding problems in which the search space is restricted to recursive expansions of a user-specified finite set of fatgraphs. For the sake of simplicity, we first describe minimizing energy in a simple free-energy model $\mathcal{E}$, where the energy of a sequence/structure is obtained by summing the contributions of individual base pairs; moreover, we present the method initially without recursive substructure. Only later, in Section 5, we extend to the full problem in realistic energy models.

▶ **Definition 1** ((Recursive) fatgraph MFE folding problem).
Input: *Finite collection of fatgraphs $\gamma_1, \ldots, \gamma_p$, sequence S*
Output: *Minimum Free Energy (MFE) arc-annotation for S according to free-energy model $\mathcal{E}$, restricting the search to recursive expansions of the input fatgraphs.*

Specifically, we solve the problem of automatic design of such pseudoknot prediction algorithms based on an input set of fatgraphs.

▶ **Definition 2** (Fatgraph folding algorithm design problem).
Input: *Finite collection of fatgraphs $\gamma_1, \ldots, \gamma_p$, sequence S*
Output: *A Dynamic-Programming algorithm that efficiently returns the MFE arc-annotation for S, with respect to free-energy model $\mathcal{E}$, over the recursive expansions of the input fatgraphs.*

Algorithm 1 Pseudocode for the recursive fatgraph folding problem.

Input : Finite number of fatgraphs $\gamma_1, \ldots, \gamma_p$, sequence S, base-pair based energy model $\mathcal{E}$
Output : Best-scoring arc-annotation for S, in the class specified by the fatgraphs

```
foreach fatgraph γ_i do
    Compute minimal expansion G_i of fatgraph γ_i          ▶ Linear time; see Section 3.2
    Find min. width tree decomposition T for G_i           ▶ FPT in tw using classic exact tree dec. algorithm
    Transform T into a canonical form tree dec T'          ▶ Polynomial time; see Section 4.1
    Compute skeleton of T'                                 ▶ Linear time; see Section 4.1
    Derive corresponding DP scheme                         ▶ Linear time; see Section 4.2
end
Use union of DP schemes to find MFE arc-annotation of S    ▶ XP in tw O(n^{tw+1}); See Section 5
```

Defining the treewidth of a fatgraph as the treewidth of its minimal expansion (see Section 3.2), our main result, stated in Algorithm 1, is the existence of an effective algorithm for the fatgraph-folding problem, XP over *tw* the maximum treewidth of the input fatgraphs. Its first step consists in a Fixed-Parameter Tractable (FPT) pre-processing of the input fat graphs, yielding DP equations for folding (see Figure 1), which can be reused to fold any other input sequence.

▶ **Theorem 3** (Main result). *Algorithm 1 solves the fatgraph folding problem in $O(n^{tw+1})$, where tw is the maximum treewidth of the input fatgraphs.*

Since the number of indices used by the DP equation is minimized, the resulting complexities could be seen as optimal within a family of simple DP algorithms. However, a characterization of such a non-trivial family of algorithms would be beyond the scope of this work, and we leave formal proofs of optimality to future work, as briefly discussed in Section 7.

3 Minimal representative expansion of a fatgraph

Our approach builds on the concept of tree decomposition, which we want to leverage to derive decomposition strategies within dynamic programming (DP) schemes. A key challenge is in the fact that tree decompositions are computed for concrete graphs, whereas our objective is to find an algorithm whose search space includes all possible expansions of an input fatgraph.

Fortunately, we find that expanding every helix of a fatgraph to length 5 (i.e. 5 nested BPs) yields a graph which is representative of the fatgraph. Namely, its optimal *tree decomposition*, having *treewidth tw*, trivially generalizes into a tree decomposition for any further expansion, retaining *treewidth tw*. This tree decomposition can finally be reinterpreted into a DP scheme that exactly solves the MFE folding problem in $\mathcal{O}(n^{tw+1})$ complexity.

3.1 Treewidth and tree decompositions

▶ **Definition 4.** *A tree decomposition $\mathcal{T} = (T, \{X_i\}_{i \in V(T)})$ of a graph $G = (V, E)$ is a tree of subsets of vertices of G, called bags, verifying the following conditions:*

- $\forall u \in V \ \exists i \in V(T)$ *such that* $u \in X_i$. *(representing vertices)*
- $\forall (u, v) \in E \ \exists i \in V(T)$ *such that* $\{u, v\} \subset X_i$. *(representing edges)*
- $T_u = \{i \in V(T) \mid u \in X_i\}$ *must be connected.* *(vertex subtree property)*

The *width* of a tree decomposition is the size of its biggest bag minus one, i.e. $\max_{i \in V(T)} |X_i| - 1$. The *treewidth* of a graph G is then the minimum possible width of a tree decomposition of G. Intuitively, the lower the treewidth, the closer G is to being

a tree. Treewidth is NP-HARD to compute [3], but fixed-parameter tractable: there is a $O(f(w) \cdot n)$ algorithm [5] deciding whether $tw(G) \leq w$ given G. Many polynomial heuristics are also known to yield reasonable results [8], and optimized exact solvers have also been developed [43, 18]. Notoriously, a wide variety of hard computational problems can be solved efficiently when restricted to graphs of bounded treewidth [7, 11], including in bioinfomatics [45, 42, 38]. Such is the case of LiCoRNA [38], for pseudoknotted structure-sequence alignment, of which the algorithm presented in this paper can be seen as a generalization.

We will rely in the remainder of this section on some well known-properties for treewidth, which we recall here. First, taking any *minor* of G [24], i.e. performing any sequence or edge contractions, edge deletions and vertex deletions on G can only lower the treewidth. Second, degree-2 vertices can be contracted into their neighbors without changing the treewidth, as quickly stated below (proof in appendix). This implies in particular that any bulge in a helix of an RNA structure graph is inconsequential with respect to treewidth.

▶ **Proposition 5.** *If u is a degree-2 vertex of G with neighbors $\{v, w\}$, and $G_{v \leftarrow u}$ is the graph obtained by contracting u into v in G then $tw(G) = tw(G_{v \leftarrow u})$*

Then, we import from [6] an inequality valid for any *separator* of G. A *separator* is a subset S of vertices of G such that $G \setminus S$ is composed of at least 2 conected components, which we write $\mathcal{C}_G(S)$. We then have:

▶ **Proposition 6.** *If S is a separator of G, then*

$$tw(G) \leq \max_{C \in \mathcal{C}_G(S)} tw(G[C \cup clique(S)])$$

with $G[C \cup clique(S)]$ the subgraph of G induced by $C \cup S$ augmented by edges making S a clique. In case of equality, we say that S is safe.

Proof. Consider, for each $C \in \mathcal{C}_G(S)$, a tree decomposition $\mathcal{T}_C$ of $G[C \cup clique(S)]$. Since these graphs contain S as a clique, each $\mathcal{T}_C$ must have a bag X_C containing S entirely. Consider now the following tree decomposition for G, make a bag out of S, and connect X_C for each C to it. The resulting tree decomposition is valid for G, and its width is the left-hand-side of the inequality. ◀

Let us finish by noting that, in a tree decomposition, any intersection $S = X \cap Y$ of two adjacent bags is always a separator of G. To write down the proofs of the following section in a smoother fashion, we add the following two properties, whose proofs are delayed to the appendix:

▶ **Proposition 7.** *A tree decomposition can always be locally modified such that, for any two adjacent bags X and Y and $S = X \cap Y$:*

- $|S| \leq tw(G)$
- *S is minimal with respect to inclusion, i.e. removing any vertex from S makes it lose its separating properties.*

3.2 Helices of length 5 are sufficient to obtain generalizable tree decompositions

Given an RNA graph (with one vertex per nucleotide and one edge per base pair and backbone link, see Figure 3(a)), we call *perfect helix* a set of directly nested base pairs, resulting in the subgraph depicted on Figure 3(b). We call the number of nested base pairs its *length*,

and denote it with l. With a slight abuse of language, we call such a subgraph a *helix*, even for general graphs. Our main structural result is to show that the treewidth of a graph G does not increase when extending a helix past a length of 5. Its proof relies on the following inequality, involving the graphs $G_{\boxtimes}$ and $G_{\boxslash}$, obtained from G by replacing a helix H with either $\boxtimes$ or $\boxslash$, (see Figure 3(c)).

▶ **Lemma 8.** *Given a graph G and a helix H of length $l \geq 3$ in G, we have:*

$$tw(G_{\boxtimes}) - 1 \leq tw(G_{\boxslash}) \leq tw(G) \leq \max(4, tw\,(G_{\boxtimes}))$$

Proof. To start with, by noticing that the 4 extremities of the helix form a separator S between the inside and the outside of it, we get by Proposition 6 that $tw(G) \leq \max(H \cup clique(S), G_{\boxtimes})$. The graph $H \cup clique(S)$ does not depend on G, and consists of a helix with the 4 extremities forming a clique. With $l \geq 2$, it turns out that this graph has treewidth 4, see Appendix A, hence the inequality.

Next, we notice that $G_{\boxslash}$ is a minor of G when $l \geq 3$. This can be seen by contracting the helix according to the pattern outlined on Figure 3(d) by the green areas (each green area is contracted to the extremity it contains). Therefore, $tw(G_{\boxslash}) \leq tw(G)$.

Finally, let us note that $G_{\boxtimes}$ and $G_{\boxslash}$ only differ by 1 edge, and removing a single edge from a graph can only decrease its treewidth by at most 1. Indeed, suppose that $tw(G_{\boxslash}) < tw(G_{\boxtimes}) - 1$, and consider an optimal tree decomposition $\mathcal{T}$ for $G_{\boxslash}$. Let us denote by u and v the two extremities of the helix not connected in $G_{\boxslash}$. If the subtrees of bags containing respectively u and v do not intersect, then one can just add v to all bags of the tree decomposition, to represent the edge (u, v) while increasing the width by ≤ 1. Therefore $tw(G_{\boxtimes}) - 1 \leq tw(G_{\boxslash})$ and the inequality is complete. ◀

Through the introduction of $G_{\boxtimes}$ and $G_{\boxslash}$ as the two possible graphs to which G is equivalent in terms of treewidth, Lemma 8 already contains the essence of our main structural result, Theorem 9. It will be the basis for generalizing tree decompositions of minimal expansions of a fatgraph to arbitrary helix lengths. Its proof is delayed to Appendix E.

▶ **Theorem 9.** *If H is a helix in G of length $l \geq 5$, then extending the helix to have length $l + 1$ does not increase the treewidth.*

Since bulges in a helix only consist of vertices of degree exactly 2, combining Proposition 5 with Theorem 9 implies that the treewidth of any expansion of a given fatgraph is always smaller than or equal to the treewidth of a minimal expansion where all bands are helices of length exactly 5. As for gaps, arguments similar to the proof of Theorem 9 can show that going from a gap of length 0 to an arbitrary length does not increase the treewidth of a fatgraph expansion. Overall, we formally define the minimal expansion of a fatgraph as:

▶ **Definition 10** (Minimal representative expansion of a fatgraph)**.** *Given a fatgraph γ, its minimal representative expansion consists of:*

- *A perfect helix of length 5 for each band.*
- *No gap between the extremities of two helices*

Such a minimal representative expansion is illustrated in Figure 5(a). For visual clarity, gaps have been kept between consecutive helices, but one can see that the corresponding extremities have the same labels. Given a fatgraph, this RNA structure graph contains all necessary information for formulating DP equations decomposing all RNA structures compatible with the fatgraph.

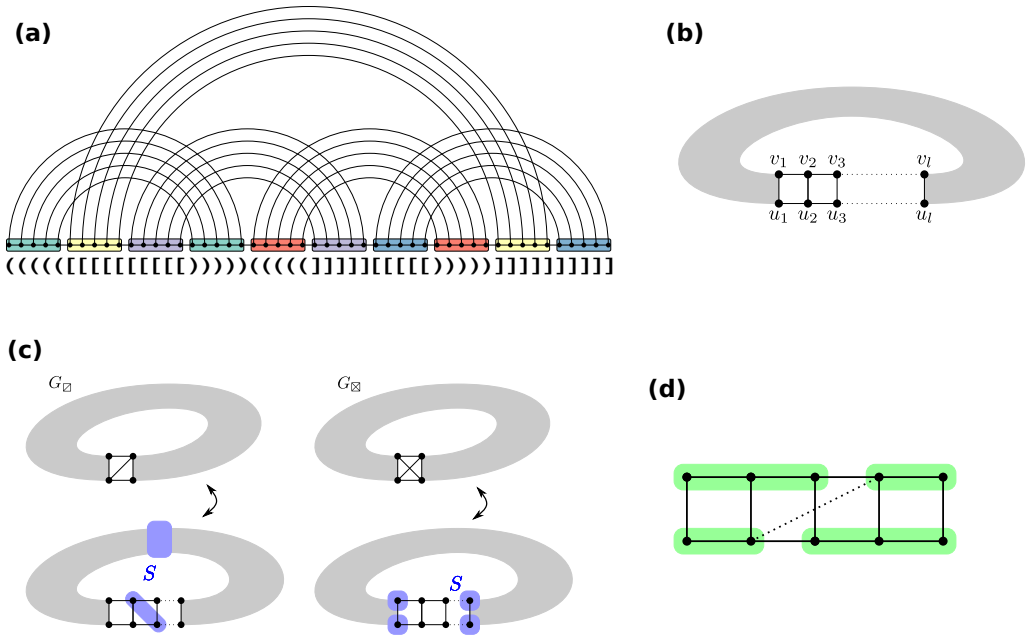


Figure 3 (a) minimal expansion of a fatgraph, with every helix of length 5, and no unpaired base. The associated graph consists of one vertex per base, and one edge per base pair and backbone link. (b) A helix of length l in an RNA graph, as per the latter definition. (c) Given a helix in a graph G, the treewidth of G is either equal to $tw(G_{\boxtimes})$ or $tw(G_{\boxslash})$. Each case is associated with a type of *separator* that can be used to extend the helix, or insert bulges, without changing the treewidth. (d) The dotted line represents a "hop-edge" which, if represented in a given tree decomposition of G, can be used to obtain $G_{\boxtimes}$ as a minor of G, showing that the helix is in the "clique" case.

Interestingly, the two graphs $G_{\boxtimes}$ and $G_{\boxslash}$ that emerge in the proofs as the two graphs G could be equivalent in terms of treewidth, as well as the separators they are associated to (see Figure 3 (c)) are reminiscent of two typical decomposition strategies used into dynamic programming for RNA folding. They suggest, for each helix in a graph, two possible "canonical representations" in terms of tree decomposition, which will be elaborated on in the next section.

4 Tree decompositions of fatgraph expansions as RNA DP algorithms

Starting with a tree decomposition for a minimal representative expansion of a given fatgraph, we first describe in this section how to represent it in a *canonical form*, with each helix represented either in one of two different ways, respectively related to $G_{\boxslash}$ and $G_{\boxtimes}$. The resulting tree decomposition can be further compressed into a *skeleton*, where bags within individual helices are compressed into a single bag.

This tree can then be interpreted as a dynamic programming scheme, in which helices are generated by specializing dynamic programming subroutines. In a sense, the tree decomposition yields automatically a decomposition strategy usable for dynamic programming, of the kind that was hand-crafted in previous approaches [33, 14].

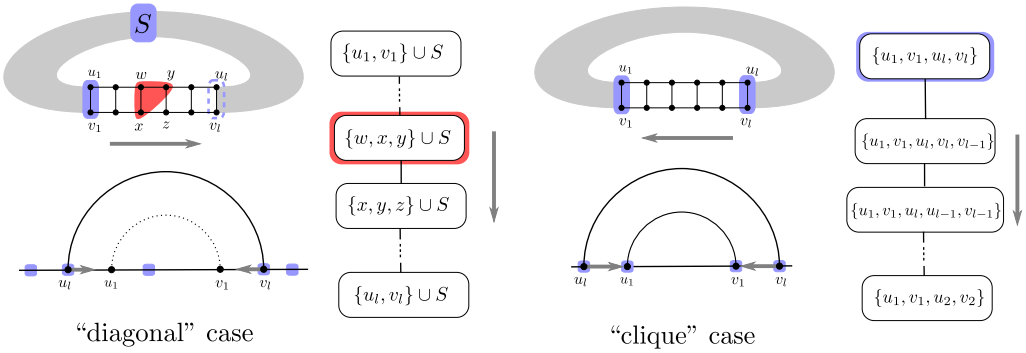


Figure 4 Illustration of the two types of canonical representations for the helices of a graph G.

4.1 Canonical form for tree decompositions

We introduce an additional definition for the sake of presentation: Given an edge $e = (X, Y)$ of a tree decomposition $\mathcal{T}$, we call the $X - side$ of $\mathcal{T}$ the connected component of $T \setminus e$ containing X.

▶ **Definition 11.** *A tree decomposition of an expansion G of a fatgraph is in canonical form if, for each helix H of length l, either:*

- ***Clique case:*** *Helix H is represented by a root bag that contains all 4 extremities of H, connected to a sub-tree-decomposition T_l recursively defined as*

$$T_0^{\boxtimes} = \emptyset \qquad T_l^{\boxtimes} = \{u_1, v_1, u_l, v_l\} \to \{u_1, v_1, u_l, v_{l-1}, v_l\} \to \{u_1, v_1, u_{l-1}, u_l, v_{l-1}\} \to T_{l-1}^{\boxtimes}.$$

- ***Diagonal case:*** *Helix H is represented by a linear series of bags starting with $X_1 = S^* \cup \{u_1, v_1\}$, finishing with $X_{2l+2} = S^* \cup \{u_l, v_l\}$, and such that for $1 < k < l + 1$ $X_{2k} = S^* \cup \{u_{2k-1}, v_{2k-1}, u_{2k}\}$ and $X_{2k+1} = S^* \cup \{v_{2k-1}, u_{2k}, v_{2k}\}$ for k odd.*

The definition above is illustrated on Figure 4. A canonical tree decomposition for a minimum expansion of a fatgraph is also presented on Figure 8. It was obtained through the processing routine that we describe in Algorithm 2 (see Appendix D), applicable to any (optimal or not) tree decomposition. It essentially follows the dichotomy of the proof of Theorem 9. We state its correctness and run-time below, but delay the proof to Appendix E.

▶ **Proposition 12.** *Given G and $\mathcal{T}$, Algorithm 2 outputs a canonical tree decomposition for G, having same width as T, in time $O(N_H \cdot n^3)$, where N_H is the number of helices.*

Note that in a canonical tree decomposition, all vertices and edges internal to a helix of a graph are represented in the canonical sub-tree-decomposition associated to it. All bags outside of these canonical blocks only consist of extremities of helices, or other vertices outside of helices. Ignoring these internal parts, to focus on a more compact "skeleton" of canonical tree decompositions will be the first step towards automatically deriving dynamic programming equations.

▶ **Definition 13.** *The skeleton of a canonical tree decomposition for a graph G, is defined as follows:*

- *All sub-tree-decompositions representing a helix in the "clique" case are replaced with a unique bag containing all extremities of the helix*

- *All sub-tree-decompositions representing a helix in the "diagonal" case are contracted to contain their first and last bags only, denoted as $S \cup \{u_1, v_1\}$ and $S \cup \{u_l, v_l\}$ in Definition 11.*

Figure 5(b) gives an example of such a skeleton.

4.2 Automatic derivation of dynamic programming equations

Given the skeleton of a representative minimal expansion of a fatgraph γ, we describe here how to formulate DP equations for the corresponding folding problem. As mentioned previously, we initially restrict our exposition to a base-pair based model, akin to the one optimized by the seminal Nussinov algorithm [29].

Essentially, we introduce helix DP tables for each helix, and transitional tables for non-helix bags. The variables indexing these tables are called *anchors*. These integer variables each represent a separation point between consecutive (half-)helices. Taken together, a full

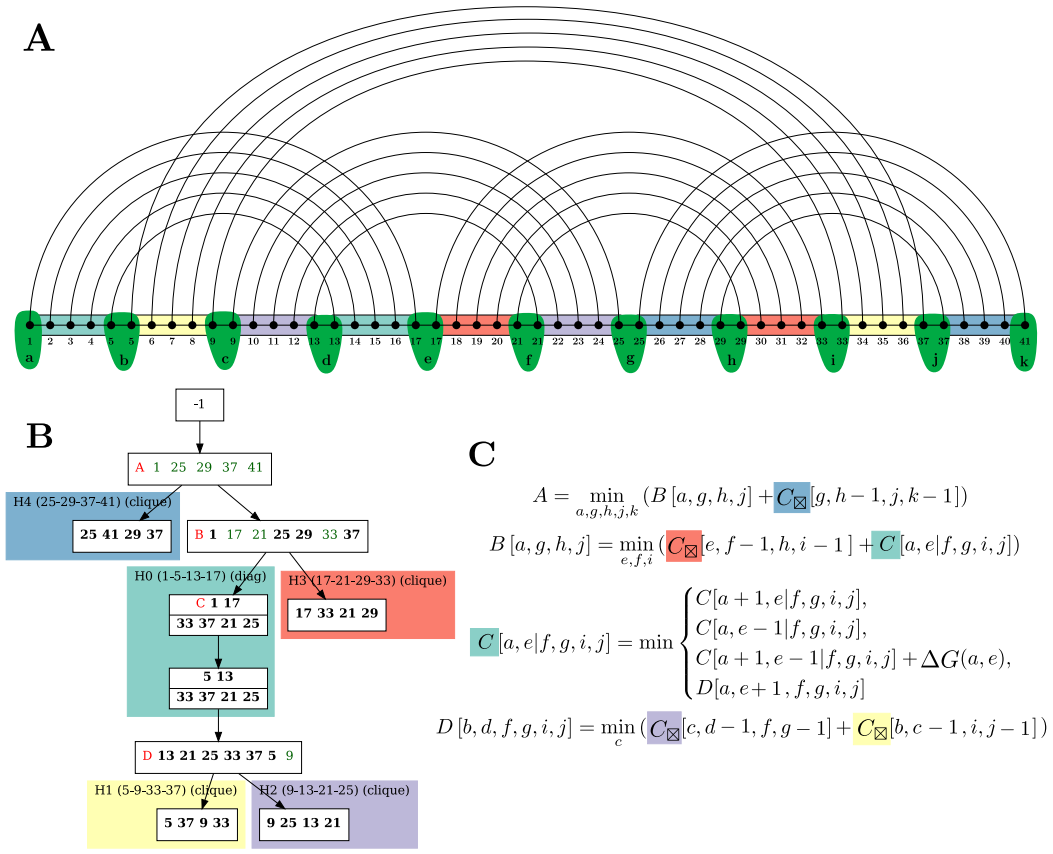


Figure 5 (a) Minimal representative length-5 expansion of the fat graph shown in Figure 1. Anchor variables are highlighted in green. We introduce one such variable per gap between helices. (b) Skeleton of the tree decomposition. White boxes represent transitional bags, introducing/propagating indices, while colored boxes represent helices in the fatgraph (H0 to H4) with associated indices in the input structure. Red letters indicate tables of the dynamic programming algorithm. Green indices are novel indices, absent from a bag's predecessor. (c) DP equations derived from the compact skeleton, involving the anchor variable defined above, and following the rules described in Section 4.2.

set of anchors $(a, b, c, \ldots)$ partitions the sequence into a set of disjoint intervals $[a, b[, [b, c[\ldots$, each associated with one *half-helix*, i.e. one of the subsequences that form a helix. Helix tables will account for the free-energy contributions of concrete base-pairs, while transitional tables will instantiate anchors in a way that remains consistent with previous assignments.

Indeed, owing to the tree decomposition, a skeleton is guaranteed to: i) feature each anchor in some bag; ii) represent each pair of consecutive anchors in at least one bag; iii) propagate anchor values, such that the anchor values within helix tables remain consistent. Due to this observation, non-helix bags can simply propagate previously-assigned anchors, possibly assigning values to novel anchors (if any and constrained to remain consistent with the sequential order) to explore all possible partitions of the input RNA sequence.

Helix tables will predict concrete sets of base pairs and account for their associated free-energy. In order to both prevent the double pairing of certain sequence positions, and to avoid ambiguity, we require (and enforce in the DP rules) that an anchor x, separating the consecutive halves of two helices H and H', implies the pairing of position x to the other half of H', and the pairing of some position $x' < x$ as part of H. In other words, a helix H delimited by anchors i, i', j', and j must pair position i to some position $x \in]j', j[$, and j' to some position $y \in]i, i'[$, implicitly leaving both regions $]y, i'[$ and $]x, j[$ unpaired.

4.2.1 Helix table 1: "Clique" cases

In the skeleton, each bag representing a helix in the "clique" case is associated to the following tables, where $i, i'+1, j'$, and $j+1$ represent the values of the anchors delimiting the helix. The increments on i' and j are here to ensure the presence of gap of length ≥ 1 between two base pairs belonging to different helices. (see also Figure 5(c) for an example of how anchor values are passed to $C_{\boxtimes}$ with a decrement of -1 for the same reason).

A first table $C'_{\boxtimes}$ holds the minimal free-energy of a helix delimited by i, i', j', and j, such that position i is paired to some $x \in]j', j[$ and j' to some position $y \in]i, i'[$. The idea is here to iteratively move the anchor from j to $j-1$, implicitly leaving position j unpaired, until a base pair (i, j) is formed. Once a base pair is created, we transition to another table $C_{\boxtimes}$ which optimizes over helices like $C'_{\boxtimes}$, but additionally allows position i to be left unpaired.

Those two tables can be filled owing to the following recurrences:

$$C'_{\boxtimes}[i, i', j', j] = \min \begin{cases} C'_{\boxtimes}[i, i', j', j-1] & \text{if } j' < j \\ C_{\boxtimes}[i+1, i', j', j-1] + \Delta G_{i,j} & \text{if } (i < i') \wedge (j' < j) \\ \Delta G_{i,j} & \text{if } j = j' \\ +\infty & \text{if no such case apply} \end{cases}$$

and

$$C_{\boxtimes}[i, i', j', j] = \min \begin{cases} C'_{\boxtimes}[i, i', j', j-1] & \text{if } j' < j \\ C_{\boxtimes}[i+1, i', j', j] & \text{if } i < i' \\ C_{\boxtimes}[i+1, i', j', j-1] + \Delta G_{i,j} & \text{if } (i < i') \wedge (j' < j) \\ \Delta G_{i,j} & \text{if } j = j' \\ +\infty & \text{if no such case apply} \end{cases}$$

where $\Delta G_{i,j}$ denote the free-energy contribution of the base-pair (i, j) in the input RNA sequence.

4.2.2 Helix tables 2: "Diagonal" cases

In the skeleton bags representing the diagonal cases, we need to associate a different table to each helix. Indeed, each "diagonal" case associates, to a helix H, a set S of indices, dubbed the *constant anchors*, whose values remain unchanged during the construction of H.

We focus on the case where (i, j) represents the value of the outermost anchor pair (i.e. $[i, j]$ represents the full span of H), leaving to the reader the symmetric case starting from the innermost pair. Note that, in the skeleton, we kept two bags for a "diagonal case" helix. Yet they are associated to a single table, since the helix is created by incrementing two indices only, such that the initial pair of extremities "becomes" the other pair. We need this second bag to know how to map index values to the children tables $\{M_k\}_k$. This value mapping at the end of a diagonal case is illustrated on Figure 6.

Namely, let the cell $D_H[i, j \mid S]$ (resp. $D'_H[i, j \mid S]$) represent the minimum-free energy achieved by the set of helices in the subtree of H, when H is anchored at (i, j) without constraints on i or j (resp. such that i is paired to some position $x \leq j'$). We have:

$$D'_H[i, j \mid S] = \min \begin{cases} D'_H[i, j-1 \mid S] & \text{if } j-1 > i \wedge \forall s \in S,\ j-1 \neq s \\ D_H[i+1, j-1 \mid S] + \Delta G_{i,j} & \text{if } \forall s \in S,\ (i+1 \neq s) \wedge (j-1 \neq s) \end{cases}$$

and

$$D_H[i, j \mid S] = \min \begin{cases} D_H[i+1, j \mid S] & \text{if } i+1 < j \wedge \forall s \in S,\ i+1 \neq s \\ D'_H[i, j-1 \mid S] & \text{if } j-1 > i \wedge \forall s \in S,\ j-1 \neq s \\ D_H[i+1, j-1 \mid S] + \Delta G_{i,j} & \text{if } \forall s \in S,\ (i+1 \neq s) \wedge (j-1 \neq s) \\ \sum_k M_k[I_k] & \text{with } I_k := (\{i, j+1\} \cup S) \cap A_k \end{cases}$$

where A_k denotes the anchors values needed for the k-th child of the diagonal bag.

4.2.3 Transitional tables: Non-helix bags

The general case consists of passing the values of relevant variables onward to the diagonal and clique tables, possibly assigning/propagating anchors that appear in the bag for the first time. Let I_P be the anchors of the parent bag of M in the tree decomposition, we have:

$$M[I_P] = \min_{\substack{\text{Values for all} \\ \text{anchors in } I \setminus I_P}} \sum_k \begin{cases} M_k[I_k] & \text{if } k\text{-th child is transitional} \\ C'_{\boxtimes}[i, i'-1, j', j-1] & \text{if clique, anchored at } (i, i', j', j) \\ D'_{H_k}[i, j-1 \mid S_k] & \text{if diagonal, anchored at } (i, j') \end{cases}$$

where I_k denotes the anchor values from I needed for the k-th child of the bag, and S represents the constant anchors of the k-th child, assumed to be a diagonal.

5 Extensions

The DP scheme, as stated above, only supports conformations that consist of a single pseudoknot configuration, indicated by a fatgraph. Moreover, it forces the first position of the sequence to always form a base pair. Finally, it considers an energy model that is fairly unrealistic in comparison with the current state of the art. In this section, we briefly describe how to extend this fundamental construction in several directions in order to solve the stated algorithm design problem (Def. 2) and consequently the associated folding problem in complex energy models, and discuss the consequences on the complexity.

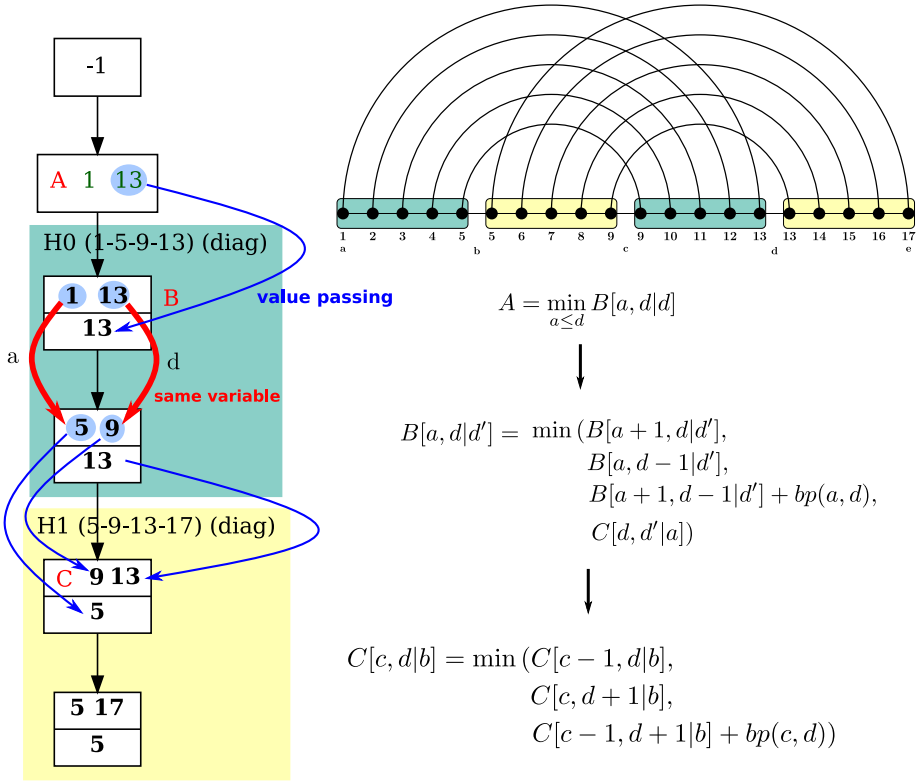


Figure 6 Derivation of DP equations from a skeleton, starting from the canonical tree decomposition of a length-5 expansion for a simple H-type fatgraph. On the left-hand-side, special emphasis is given to explaining how values are mapped at the end of a diagonal case. Extra tables $C'_{\boxtimes}$ and D'_H, needed to ensure unambiguity of the DP scheme, are omitted for the sake of simplicity without adverse consequences to correctness.

5.1 Multiple fatgraphs and integration within 2D folding scheme

Alternative fatgraphs can easily be considered, without added complexity, by simply adding a disjunctive rule at the top level of the DP scheme, such as $\text{MFE}_{\text{PK}} := \min_{i=1}^{p} root_{\gamma_i}$ where $root_{\gamma_i}$ is the top level of the DP scheme for fatgraph γ_i. The associated conformation space then consists of the union of all pseudoknotted structures compatible with one of the fatgraphs.

However, fatgraphs usually represent a structural module rather than a complete RNA conformation. The classic DP scheme for 2D structure energy-minimization can thus be supplemented by additional constructs, enabling the consideration of pseudoknots. Towards that, one needs to access $\text{MFE}_{\text{PK}}(i, j)$, the MFE achieved over a region $[i, j]$ by a conformation compatible with one of the input fat graphs. In other words, one needs an ability to prescribe the span, say $[i, j]$, of the fatgraph occurrence, *i.e.* the values of the extremal anchors, while initiating the dynamic programming.

To ensure this possibility, one simply needs to connect the first and last positions in the minimal fatgraph completion. Indeed, since each arc of the input graph must be represented, any tree decomposition for the completion will feature a bag B including both first and

last position (+ additional anchors $S := \{k_1, k_2, \ldots\}$). Moreover, since a tree decomposition is unordered, B can be arbitrarily used as the root, preceded by a root node restricted to anchors (i, j). This yields the following entry point for the DP of a fatgraph γ:

$$\mathrm{r}_\gamma(i,j) := \min_{i<k_1<k_2<\ldots<j} M_B[i, k_1, k_2, \ldots, j]$$

which can be queried from within a classic DP scheme for the secondary structure.

5.2 Energy models

The extension to more realistic energy models is possible through functions evaluating recursive non-crossing substructure; crossing configuration-specific score contributions; and modifications of the algorithms that fill tables for the clique and diagonal cases. The former enables scoring non-crossing substructure in the Turner model and doesn't require changes beyond our discussion on recursive substructures and performing standard non-crossing free energy minimization. Handling multiple fatgraphs as described by disjunction at the top level enables specific scoring of different crossing configurations.

The latter case concerns the scoring of energy within helix expansions. Firstly, we observe that stacking energy between base pairs of the helix can be accounted for with minimal modification of the helix table recursions and therefore does not change the complexity. For this purpose, one introduces additional 'closed' states of the tables (corresponding to the matrix for closed subsequences in non-crossing free energy minimization). To explicitly score interior loops and bulges, the helix table recursions are extended by a case minimizing over the different loops. Naïvely, this would increase the complexity by a linear factor, which is avoided by bounding the loop size, as common in implemented folding algorithms, or without bounding the size following [25].

5.3 Recursive substructures

Recursive substructures consist of secondary structures/occurrences of fatgraphs that are inserted, both in between and within helices, usually through recursive calls to the (augmented) 2D folding scheme.

To enable the insertion of substructures within an helix requires modifications to the helix clique/diagonal rules that are very similar to the ones enabling support for the Turner energy model. Assuming the presence of a base pair (i, j), An insertion can indeed be performed by delimiting a region $[i, k]$ (resp. $[k, j]$) of arbitrary length, leading to an overall MFE of $\mathrm{MFE}_{\mathrm{SS}}(i, k) + \delta$, where δ is the free-energy contributed by the rest of the helix (possibly accounting for additional terms associated with multiloops).

To allow arbitrary sub-structures to be inserted in the gaps between consecutive helices, one can again modify the minimal helix expansion to distinguish the anchors a, b associated with consecutive helices (instead of merging them into a single anchor in our initial exposition). By connecting a and b, one ensures their simultaneous presence in a tagged bag B, whose DP recurrence is then augmented to include an energy contribution $\mathrm{MFE}_{\mathrm{SS}}(a + 1, b - 1)$.

5.4 Partition functions and ensemble applications

For ensemble applications of our DP schemes, such as computing the partition function [26] and statistical sampling of the Boltzmann ensemble [12], it is imperative for the DP scheme above to be complete and unambiguous [31]. Fortunately, both properties are already guaranteed by our DP schemes. Indeed, intuitively: the completeness is ensured by the

exhaustive investigation of all possible anchor positions, i.e. all possible partitions; the unambiguity is guaranteed by the invariant that assigning a position x to a given anchor (within a transitional or diagonal bag), leads x to be paired within the (half-)helix immediately to its right. Choosing different values for x thus induces different innermost/outermost base pairs for the associated helix, leading to disjoint sets of structures.

From this property, we conclude that the partition function for a fatgraph (or several, possibly recursively and/or within a realistic energy model) can be obtained by simply replacing the $(\min, +, \Delta G)$ terms into $(\sum, \times, e^{\beta \Delta G})$, with $\beta = RT$ being the Boltzmann constant multiplied by some absolute temperature.

6 (Re-)Designing algorithms for specific pseudoknot classes

Our pipeline for automated generation of DP folding equations given a fatgraph has been implemented using `Python` and `Snakemake` [28]. The implementation is freely available at:

`https://gitlab.inria.fr/bmarchan/auto-dp`

Since the algorithms in [33] have been described in terms of a finite number of fatgraphs (called irreducible shadows in the paper), one can directly apply our method to obtain an efficient algorithm that covers the same class as `gfold`, namely **1-structures** that are recursive expansions of the four fatgraphs of genus 1 corresponding to simple PK 'H' `([])`, kissing hairpin 'K' `([](`)`)`, three-knot 'L' `([{)]}]` and 'M' `([{)(]})` (here, represented in *dot-bracket notation*, i.e. corresponding opening and closing brackets correspond to arcs). The maximum complexity of $O(n^6)$ of the four fatgraphs (see Table 1) implies that the automatically derived algorithm covers the class of 1-structures in $O(n^6)$ time – the same complexity as hand-crafted `gfold`. Note that [33] used declarative methods in their algorithm design only to the point of generating grammar rules, which without further optimization yield $O(n^{18})$ (after applying algebraic dynamic programming; ADP [37]). In contrast, our method obtains the optimal complexity in fully automatic fashion. Beyond this re-design of `gfold`, remarkably our method is equally prepared to automatically design a DP algorithm with optimized efficiency for **2-structures**, which are based on all genus 2 fatgraphs. This is remarkable, since the implementation of a practical algorithm has been considered infeasible [33] due to the large number of genus 2 shadows (namely, there are 3472 shadows/fatgraphs), whose grammar rules would have to be optimized by hand. In contrast, due to full automation, our method directly handles even the large number of fatgraphs of genus 2 and yields an efficient, complexity optimized, DP scheme.

Recall that we cover all other pseudoknot classes that are recursive expansions of a finite number of fatgraphs (in the same way as we cover the design of prediction algorithms for 1- and 2-structures). In this way, among the previously existing DP algorithms, we cover the class of **Dirks&Pierce** (D&P) [14], simply by specifying the H-type as single input fatgraph. Consequently, we automatically re-design the D&P algorithm in the same complexity of $O(n^5)$. Even more interestingly, we can design algorithms covering specific (sets of) crossing configurations. This results in an infinite class of efficient algorithms that have not been designed before. Again the complexity of such algorithms is dominated by the most complex fatgraph; where results for interesting ones are given in Table 1. Most remarkably, we design an algorithm optimizing over recursive expansions of kissing hairpins in $O(n^4)$, whereas CCJ [10, 21], which was specifically designed to cover kissing hairpins, requires $O(n^5)$.

A special case, which further showcases the flexibility, is the extension of existing classes by specific crossing configurations. For example, extending D&P by kissing hairpin covers a much larger class while staying in the same complexity. Extending 1-structures by 5-chain

Table 1 Table listing pseudoknot classes, corresponding treewidth and resulting complexity of the folding algorithm. In all cases except the one denoted by (*), the complexity of folding is equal to $O(n^{tw+1})$. For the kissing hairpins case, we are in the specific case where the most complex routine is the alignment of a "clique case" helix, which is done in $O(n^4)$ despite a treewidth of 4. These examples are detailed in the Appendix, Figure 9. The DP equations for each of these examples have been automatically generated by a `Python` implementation of our pipeline, freely available at `https://gitlab.inria.fr/bmarchan/auto-dp`.

name	fatgraph	treewidth	complexity of folding
H-type	(])]	4	$O(n^5)$
kissing hairpins	(])(])	4	$O(n^4)$ (*)
"L" [33]	([{)]}	5	$O(n^6)$
"M" [33]	([{)(]})	5	$O(n^6)$
4-clique	([{<)]}>	5	$O(n^6)$
5-clique	([{<A)]}>a	5	$O(n^6)$
5-chain	({])(][])}]	6	$O(n^7)$

yields a new algorithm with a complexity below of 2-structures (namely only $O(n^7)$ instead of $O(n^8)$ [33]). The complexity of 5-chain is remarkably low, when considering that previously described algorithms covering this configuration take $O(n^8)$ (e.g. `gfold`'s generalization to 2-structures and a hypothetical blow-up of the Rivas and Eddy algorithm [39] to 6-dimensional instead of 4-dimensional DP matrix elements – both of which have never been implemented).

7 Conclusions and discussion

In this work, we provide an algorithm that takes a family of fatgraphs, i.e. pseudoknotted structures, and returns DP equations that efficiently predict arc annotations minimizing the free energy. The DP equations are automatically generated based on an expansion of the fatgraph, designed to capture helices of arbitrary length. The DP tables in the equations use a number of indices smaller than or equal to the treewidth of the minimal expansion. This very general framework recovers the complexity of prior, hand-crafted algorithms, and lays the foundation for a purely declarative approach to RNA folding with pseudoknots.

In addition to the extensions described in Section 5, this work suggests perspectives that will be explored in future work. Indeed, the choice of an optimal decomposition/DP scheme for the input fatgraph can be seen as the automated design of an optimal table strategy in the context of algebraic dynamic programming [32, 4, 37]. This would enable extensions to multiple context free grammars or tree grammars when describing the problem in the ADP framework.

Our automated design of pseudoknot folding algorithms could naturally be extended to RNA–RNA interactions, since the joint conformation of two interacting RNA sequences can be seen as a pseudoknot when concatenating the two structures [13]. More ambitiously, categories of pseudoknots inducing an infinite family of fatgraphs, *e.g.* as covered by the seminal Rivas & Eddy algorithm [39], could be captured by allowing the introduction of recursive gapped structures in prescribed parts of the fatgraph. This could be addressed by adding cliques to the minimal completion graph would ensure the availability of the relevant anchors in some bags of the tree decomposition, allowing to score such, non-contiguous, recursive substructures.

Another avenue for future research includes a proof of optimality, in term of polynomial complexity, for the produced DP algorithms. Of course, it would be far too ambitious (and erroneous) to expect our DP schemes to be optimal within general computational models.

However, it may be possible to prove optimality within a clearly-defined subset of standard implementations of a subset of DP schemes, *e.g.* by contradiction since the existence of a better algorithm would imply the existence of a tree decomposition having smaller width.

References

1 Tatsuya Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Applied Mathematics*, 104(1-3):45–62, 2000.

2 Can Alkan, Emre Karakoç, Joseph H. Nadeau, S. Cenk Sahinalp, and Kaizhong Zhang. RNA–RNA Interaction Prediction and Antisense RNA Target Search. *Journal of Computational Biology*, 13(2):267–282, 2006. doi:10.1089/cmb.2006.13.267.

3 Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in ak-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

4 Sarah J Berkemer, Christian Höner zu Siederdissen, and Peter F Stadler. Algebraic dynamic programming on trees. *Algorithms*, 10(4):135, 2017.

5 Hans L Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25(6):1305–1317, 1996.

6 Hans L Bodlaender and Arie MCA Koster. Safe separators for treewidth. *Discrete Mathematics*, 306(3):337–350, 2006.

7 Hans L Bodlaender and Arie MCA Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008.

8 Hans L Bodlaender and Arie MCA Koster. Treewidth computations i. upper bounds. *Information and Computation*, 208(3):259–275, 2010.

9 Song Cao and Shi-Jie Chen. Predicting RNA pseudoknot folding thermodynamics. *Nucleic Acids Research*, 34(9):2634–2652, January 2006. doi:10.1093/nar/gkl346.

10 Ho-Lin Chen, Anne Condon, and Hosna Jabbari. An O(n^5) algorithm for MFE prediction of kissing hairpins and 4-chains in nucleic acids. *Journal of Computational Biology*, 16(6):803–815, 2009.

11 Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*, volume 1. Springer, 2015.

12 Ye Ding and Charles E. Lawrence. A statistical sampling algorithm for RNA secondary structure prediction. *Nucleic Acids Research*, 31(24):7280–7301, December 2003. doi:10.1093/nar/gkg938.

13 Robert M Dirks, Justin S Bois, Joseph M Schaeffer, Erik Winfree, and Niles A Pierce. Thermodynamic analysis of interacting nucleic acid strands. *SIAM review*, 49(1):65–88, 2007.

14 Robert M Dirks and Niles A Pierce. A partition function algorithm for nucleic acid secondary structure including pseudoknots. *Journal of computational chemistry*, 24(13):1664–1677, 2003.

15 Chuong B Do, Daniel A Woods, and Serafim Batzoglou. CONTRAfold: RNA secondary structure prediction without physics-based models. *Bioinformatics*, 22(14):e90–e98, 2006.

16 Mark E. Fornace, Nicholas J. Porubsky, and Niles A. Pierce. A Unified Dynamic Programming Framework for the Analysis of Interacting Nucleic Acid Strands: Enhanced Models, Scalability, and Speed. *ACS Synthetic Biology*, 9(10):2665–2678, 2020. PMID: 32910644. doi:10.1021/acssynbio.9b00523.

17 Robert Giegerich, Björn Voß, and Marc Rehmsmeier. Abstract shapes of rna. *Nucleic acids research*, 32(16):4843–4851, 2004.

18 Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. *arXiv preprint arXiv:1207.4109*, 2012.

19 Fenix Huang, Christian Reidys, and Reza Rezazadegan. Fatgraph models of RNA structure. *Computational and Mathematical Biophysics*, 5(1):1–20, 2017.

20 Hosna Jabbari and Anne Condon. A fast and robust iterative algorithm for prediction of RNA pseudoknotted secondary structures. *BMC bioinformatics*, 15(1):1–17, 2014.

21 Hosna Jabbari, Ian Wark, Carlo Montemagno, and Sebastian Will. Knotty: efficient and accurate prediction of complex RNA pseudoknot structures. *Bioinformatics*, 34(22):3849–3856, 2018.

22 Martin Loebl and Iain Moffatt. The chromatic polynomial of fatgraphs and its categorification. *Advances in Mathematics*, 217(4):1558–1587, 2008.

23 R Lorenz, SH Bernhart, C Höner Zu Siederdissen, H Tafer, C Flamm, PF Stadler, and IL Hofacker. ViennaRNA Package 2.0. vol. 6. *Algorithms Mol. Biol*, page 26, 2011.

24 László Lovász. Graph minor theory. *Bulletin of the American Mathematical Society*, 43(1):75–86, 2006.

25 R. B. Lyngsø, M. Zuker, and C. N. Pedersen. Fast evaluation of internal loops in RNA secondary structure prediction. *Bioinformatics (Oxford, England)*, 15(6):440–445, June 1999. doi:10.1093/bioinformatics/15.6.440.

26 J. S. McCaskill. The equilibrium partition function and base pair binding probabilities for rna secondary structure. *Biopolymers*, 29(6-7):1105–1119, 1990. doi:10.1002/bip.360290621.

27 Mathias Möhl, Sebastian Will, and Rolf Backofen. Lifting prediction to alignment of RNA pseudoknots. *Journal of Computational Biology*, 17(3):429–442, 2010.

28 Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B Hall, Christopher H Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O Twardziok, Alexander Kanitz, et al. Sustainable data analysis with snakemake. *F1000Research*, 10, 2021.

29 Ruth Nussinov and Ann B Jacobson. Fast algorithm for predicting the secondary structure of single-stranded rna. *Proceedings of the National Academy of Sciences*, 77(11):6309–6313, 1980.

30 Robert Clark Penner, Michael Knudsen, Carsten Wiuf, and Jørgen Ellegaard Andersen. Fatgraph models of proteins. *Communications on Pure and Applied Mathematics*, 63(10):1249–1297, 2010.

31 Yann Ponty and Cédric Saule. A combinatorial framework for designing (pseudoknotted) RNA algorithms. In Teresa M. Przytycka and Marie-France Sagot, editors, *Algorithms in Bioinformatics*, pages 250–269, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

32 Michela Quadrini, Luca Tesei, and Emanuela Merelli. An algebraic language for RNA pseudoknots comparison. *BMC bioinformatics*, 20(4):1–18, 2019.

33 Christian M Reidys, Fenix WD Huang, Jørgen E Andersen, Robert C Penner, Peter F Stadler, and Markus E Nebel. Topology and prediction of RNA pseudoknots. *Bioinformatics*, 27(8):1076–1085, 2011.

34 Christian M Reidys and Rita R Wang. Shapes of RNA pseudoknot structures. *Journal of Computational Biology*, 17(11):1575–1590, 2010.

35 Jihong Ren, Baharak Rastegari, Anne Condon, and Holger H Hoos. HotKnots: heuristic prediction of RNA secondary structures including pseudoknots. *Rna*, 11(10):1494–1504, 2005.

36 Jessica S Reuter and David H Mathews. RNAstructure: software for rna secondary structure prediction and analysis. *BMC bioinformatics*, 11(1):1–9, 2010.

37 Maik Riechert, Christian Höner zu Siederdissen, and Peter F. Stadler. Algebraic dynamic programming for multiple context-free grammars. *Theoretical Computer Science*, 639:91–109, August 2016. doi:10.1016/j.tcs.2016.05.032.

38 Philippe Rinaudo, Yann Ponty, Dominique Barth, and Alain Denise. Tree decomposition and parameterized algorithms for RNA structure-sequence alignment including tertiary interactions and pseudoknots. In *International Workshop on Algorithms in Bioinformatics*, pages 149–164. Springer, 2012.

39 Elena Rivas and Sean R Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *Journal of molecular biology*, 285(5):2053–2068, 1999.

40 Kengo Sato, Manato Akiyama, and Yasubumi Sakakibara. RNA secondary structure prediction using deep learning with thermodynamic integration. *Nature communications*, 12(1):1–9, 2021.

41 Kengo Sato, Yuki Kato, Michiaki Hamada, Tatsuya Akutsu, and Kiyoshi Asai. IPknot: fast and accurate prediction of RNA secondary structures with pseudoknots using integer programming. *Bioinformatics*, 27(13):i85–i93, 2011.

42 Céline Scornavacca and Mathias Weller. Treewidth-based algorithms for the small parsimony problem on networks. In *WABI*, volume 201 of *LIPIcs*, pages 6:1–6:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

43 Hisao Tamaki. Positive-instance driven dynamic programming for treewidth. *Journal of Combinatorial Optimization*, 37(4):1283–1311, 2019.

44 Edwin Ten Dam, Kees Pleij, and David Draper. Structural and functional aspects of RNA pseudoknots. *Biochemistry*, 31(47):11665–11676, 1992.

45 Hua-Ting Yao, Jérôme Waldispühl, Yann Ponty, and Sebastian Will. Taming Disruptive Base Pairs to Reconcile Positive and Negative Structural Design of RNA. In *RECOMB 2021-25th international conference on research in computational molecular biology*, 2021.

46 Shay Zakov, Yoav Goldberg, Michael Elhadad, and Michal Ziv-Ukelson. Rich parameterization improves RNA structure prediction. *Journal of Computational Biology*, 18(11):1525–1542, 2011.

47 Michael Zuker. Mfold web server for nucleic acid folding and hybridization prediction. *Nucleic acids research*, 31(13):3406–3415, 2003.

A Width of a helix closed by a clique

Let us denote by H_l^* the graph corresponding to a helix of length l, with the extremities connected as a clique. This graph appears when considering the possible safety (see Proposition 6) of the extremities as a separator of the graph. We show the following result:

▶ **Lemma 14.** *For $l = 2$, $tw(H_l^*) = 3$, while for $l \geq 3$, $tw(H_l^*) = 4$.*

Proof. For $l = 2$, H_l^* is simply the clique on 4 vertices, and which has a width of 3. For $l \geq 3$, a clique on 5 vertices can be obtained as a minor by contracting the internal part of the helix to one vertex, which ends up being connected to all 4 extremities, which already form a clique. Therefore, $tw(H_l^*) \geq 4$. To obtain the equality, we recursively build a tree decomposition of width ≤ 4, starting with $l = 2$ which we already described. Given a tree decomposition of width ≤ 4 for H_l^*, there has to be a bag X containing all 4 extremities $\{u_1, v_1, u_l, v_l\}$ (see Figure 3(b)). We introduce two new bags: $X' = \{u_1, v_1, u_l, v_l, v_{l+1}\}$ introducing a new vertex v_{l+1}, and $X'' = \{u_1, v_1, u_l, v_{l+1}, u_{l+1}\}$ introducing u_{l+1}. We connect X' to X and X'' to X'. By doing so, we respect the subtree connectivity property for all involved vertices, and build a tree decomposition capable of representing H_{l+1}^*. ◀

B Helix extension close to a separator

Figure 7 shows how, once we have found a separator, associated to an edge of the tree decomposition, separating $\{u_i, v_i\}$ from $\{u_j, v_j\}$ with $i < j$, we can insert new vertices in the helix, extending it while preserving the treewidth. This is used in the proof of Theorem 9, in what corresponds in Section 4 to the "diagonal" case.

C Detailed examples

Figure 8 shows a canonical tree decomposition for the minimal length-5 expansion, shown in the upper half of the figure, for the fatgraph showed in Figure 1. This tree decomposition is optimal, and was computed with [43], a solver that empirically works quite fast on RNA graphs.

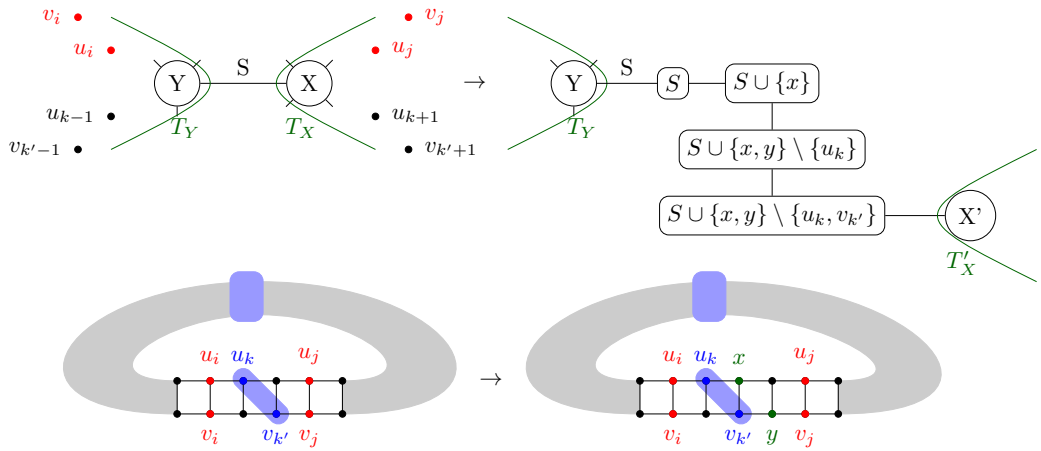


Figure 7 Representation of the local rewriting of a tree decomposition next to a separator S separating to base pairs (u_i, v_i) and (u_j, v_j), in order to extend a helix by one unit, through the introduction of new vertices x and y.

D Transforming a tree decomposition in its canonical form

Algorithm 2 describes how to obtain a canonical tree decomposition for an RNA structure graph, given any valid tree decomposition as input. Interestingly, it can use a sub-optimal tree decomposition obtained from a polynomial heuristic [7] instead of an exponential solver (although [43] is empirically quite efficient on RNA structure graphs).

The run-time and correctness of Algorithm 2 are stated in Proposition 12.

E Delayed proofs

Proof of Proposition 12. Concerning the run-time, enumerating all pairs $1 \leq i < j \leq l$) is quadratic in the length of the helix under consideration, which is $O(n)$ in a general graph, while testing a given edge for separation of u_i, v_i and u_j, v_j takes $O(n)$ (through breadth-first search) for each of the $O(n)$ edges of the tree decomposition. As for its correctness: in all cases of the algorithm, representations of edges outside the helices is not affected by the re-writing, while edges inside the edges are accounted for by the canonical representations. ◀

Proof of Proposition 5. To start with, $G_{v \leftarrow u}$ is a minor of G, therefore $tw(G_{v \leftarrow u}) \leq tw(G)$. Then, given an optimal tree decomposition $\mathcal{T}$ for $G_{v \leftarrow u}$, since (v, w) is an edge of this graph, there has to be a bag X containing both vertices. If $tw(G_{v \leftarrow u}) = 1$, then $X = \{v, w\}$ and can be split into two bags $\{v, u\}$ and $\{u, w\}$ to obtain a tree decomposition for G. If $tw(G_{v \leftarrow u}) \geq 2$, then we can simply connect a new bag $\{u, v, w\}$ and connect it to X to obtain again a valid tree decomposition for G of the same width. Therefore $tw(G) \leq tw(G_{v \leftarrow u})$ and we have the equality. ◀

Proof of Theorem 9. Let us distinguish two cases depending on the treewidth of G. For both of them, we consider an optimal tree decomposition $\mathcal{T}$ of G and show how to modify it into a valid tree decomposition for the extended version of G:

- if $tw(G) \leq 3$ then there has to be a pair i, j $(i \leq j)$ of indices $\in [1, l]$ such that $|i - j| > 1$ and neither u_i, v_i or u_j, v_j are present together in one bag. Indeed, if $\forall i, j \in [1, l]$ there was such an "hop edge" represented, then contracting u_k, v_k together $\forall k$ would yield a

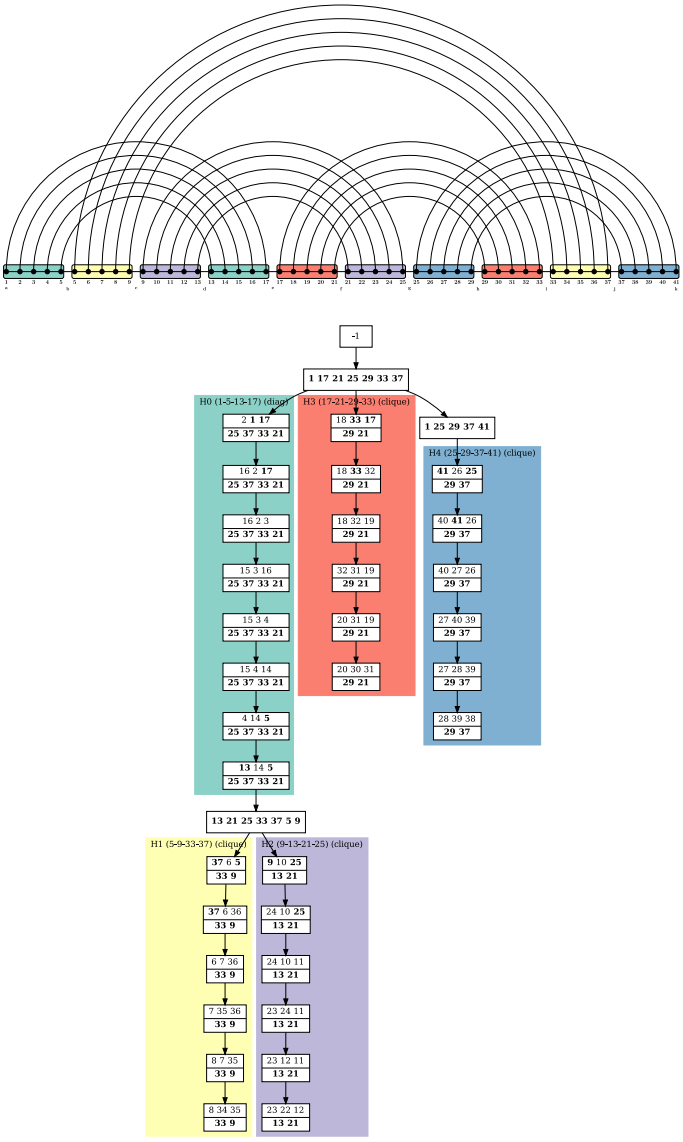


Figure 8 Canonical tree decomposition of the fatgraph given in Figure 1. White boxes represent the bags of the tree decomposition. Number in the bags correspond to the indices of the helices in the fatgraph where number on the bottom are kept while traversing the branch of the decomposition tree. Colored frames indicate the distinct helices (H0 to H4) of the structure.

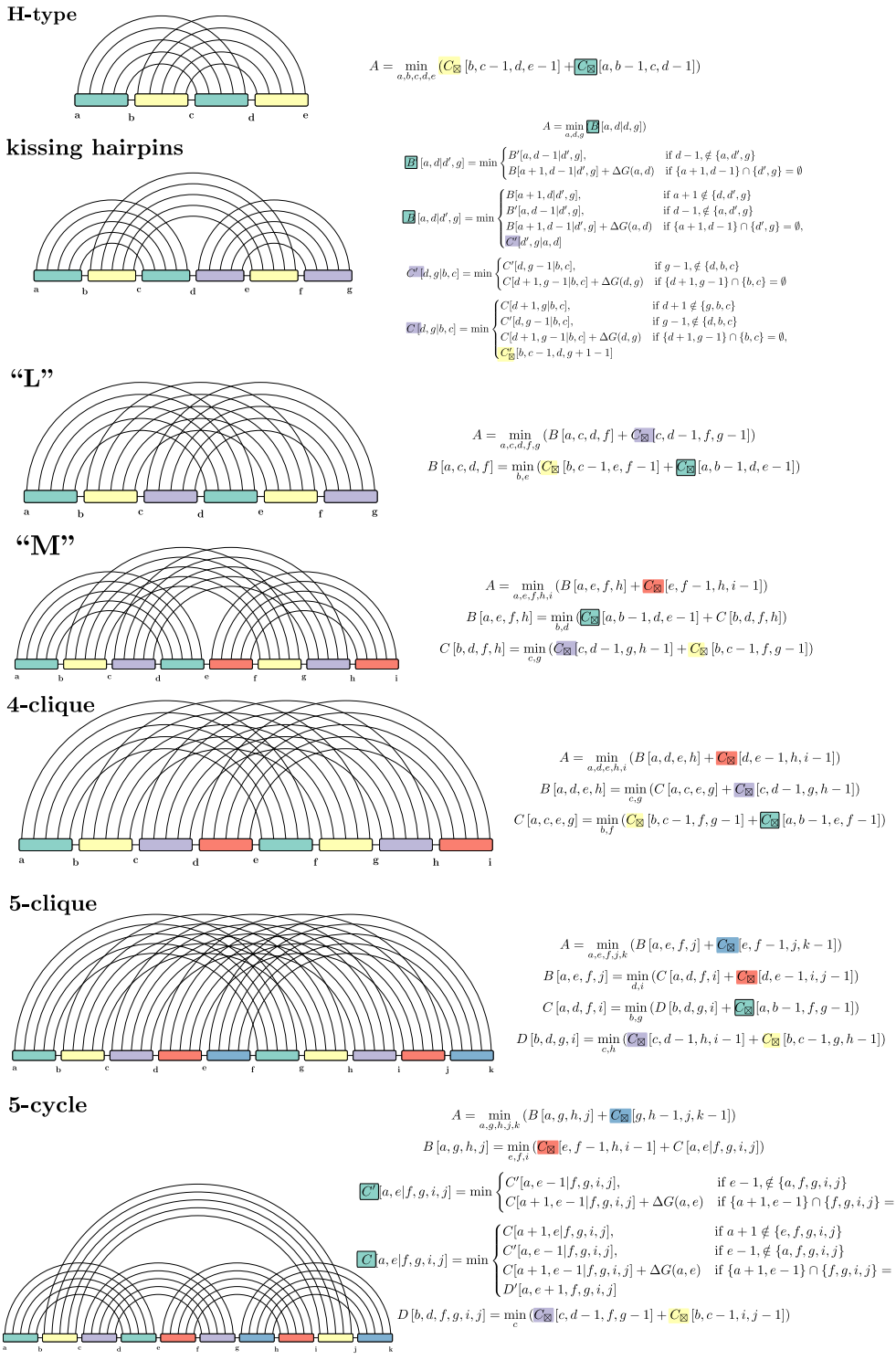


Figure 9 Minimal representative expansions and final equations for the examples of Table 1. The equations have been automatically generated, and the pipeline code is freely available at `https://gitlab.inria.fr/bmarchan/auto-dp`. In particular, the optimal tree decompositions were computed by [43].

Algorithm 2 Algorithm for re-writing a tree decomposition into a canonical one in which every helix of the input graph is represented in a canonical way.

Input : A (not necessarily optimal) tree decomposition $\mathcal{T}$ of a minimal expansion of a fatgraph γ.
Output : A tree decomposition of G in canonical form

if *width*$(\mathcal{T}) \leq 3$ **then**
  foreach *helix* H *in fatgraph* γ **do**
    if $\exists$ *hop-edge represented in* $\mathcal{T}$ **then**
      use hop-edge to obtain a tree dec. for $G_{\boxtimes}$ //▶ (see Fig. 3(d))
      find a bag $X = \{u_1, v_1, u_l, v_l\}$ as $w(T) \leq 3$
      replace X with a "diagonal" canonical representation with $S = \emptyset$.
    else
      find an edge (X, Y) of $\mathcal{T}$ s.t $X \cap Y$ separates u_1, v_1 on the X-side from u_l, v_l on the Y-side
      $\forall i$, replace u_i with u_1 and v_i with v_1 in all bags of the X-side of $\mathcal{T}$
      $\forall j$, replace u_j with u_l and v_j with v_l in all bags of the Y-side of $\mathcal{T}$
      Insert between X and Y the "diagonal" canonical representation for H, with constant part $S = (X \cap Y) \setminus \{u_k, v_k\}_{i \leq k \leq j}$
    end
  end
else
  for *helix* H *in* γ **do**
    if $\exists$ *a hop-edge represented in* $\mathcal{T}$ **then**
      Use the hop-edge to obtain a tree decomposition for $G_{\boxtimes}$
      find a bag containing all extremities and connect $T_l^{\boxtimes}$ to it
    else
      find an edge (X, Y) of $\mathcal{T}$ separating $\{u_1, v_1\}$ and $\{u_l, v_l\}$
      $\forall i$ replace u_i with u_1 and v_i with $v_1$on the X-side of $\mathcal{T}$
      $\forall i$ replace u_i with u_l and v_i with v_l on the Y-side of $\mathcal{T}$
      Insert between X and Y the "diagonal" canonical representation for H, with constant part $S = (X \cap Y) \setminus \{u_k, v_k\}_{1 \leq k \leq l}$
    end
  end
end

clique on 5 vertices, which is forbidden if $tw(G) \leq 3$. Given such a pair i, j of indices, there has to be an edge (X, Y) of the tree decomposition that separates all occurrences of u_i, v_i from all occurrences of u_j, v_j. Let us denote $S = X \cap Y$ the separator associated to that edge. By Proposition 7, S can be assumed to be inclusion minimal, and therefore to contain exactly 2 vertices u_k and $v_{k'}$ such that $|k - k'| \leq 1$ and $i \leq k, k' \leq j$. Such a separator is depicted on Figure 3(c), as well as on Figure 7. On this latter Figure, we also depict the re-writing we perform: we introduce two new vertices x and y to the X-side of the separator, as well as intermediary bags between Y and X that will gradually transform u_k, v'_k into x and y. To be specific, we introduce S as a bag between X and Y, and connect it to X through the series of bags $S \cup \{x\}$, $S \cup \{x, y\} \setminus \{u_k\}$, $S \cup \{x, y\} \setminus \{u_k, v'_k\}$ in the case (w.l.o.g) that $k \leq k'$. In addition, all occurences of u_k in X and beyond in the subtree rooted at X and directed away from S are replaced with x and those of v'_k with y. Since $|S| \leq tw(G)$, such a re-writing does not increase the treewidth, while representing all necessary edges for an extension of the helix by one level.

- if $tw(G) \geq 4$, then we consider two sub-cases depending on whether $\mathcal{T}$ represents any "hop-edge" as depicted on Figure 3(d), i.e. an edge between u_k and v_l or v_k and u_l for $|k - l| > 1$. If any such edge is represented (i.e. there exists a bag containing both endpoints), then by contracting the parts depicted in green on Figure 3 (d) to the extremity they contain (i.e replacing all occurrences of these vertices in the tree decomposition with their corresponding extremity), we obtain a valid tree decomposition for $G_{\boxtimes}$ of width $\leq tw(G)$. By the inequality of Proposition 8, we get that $tw(G) = \max(4, tw(G_{\boxtimes}))$, and the extremities of the helix are a safe separator. There exists therefor an optimal tree decomposition $\mathcal{T}'$ of G which contains S as a bag, separating the helix from the rest of the graph. By Lemma 14, replacing the sub-tree-decomposition of $\mathcal{T}'$ corresponding to the helix with a tree decomposition for a helix longer by 1 unit does not change the width of this sub-tree-decomposition. If there is no such "hop-edge", then there is an edge (X, Y) in the tree decomposition that separates (u_1, v_1) from (u_l, v_l), and to which we can apply the same re-writing as in the case of $tw(G) \leq 3$. ◀

Fast and Accurate Species Trees from Weighted Internode Distances

Baqiao Liu
Department of Computer Science, University of Illinois Urbana-Champaign, IL, USA

Tandy Warnow[1]
Department of Computer Science, University of Illinois Urbana-Champaign, IL, USA

Abstract

Species tree estimation is a basic step in many biological research projects, but is complicated by the fact that gene trees can differ from the species tree due to processes such as incomplete lineage sorting (ILS), gene duplication and loss (GDL), and horizontal gene transfer (HGT), which can cause different regions within the genome to have different evolutionary histories (i.e., "gene tree heterogeneity"). One approach to estimating species trees in the presence of gene tree heterogeneity resulting from ILS operates by computing trees on each genomic region (i.e., computing "gene trees") and then using these gene trees to define a matrix of average internode distances, where the internode distance in a tree T between two species x and y is the number of nodes in T between the leaves corresponding to x and y. Given such a matrix, a tree can then be computed using methods such as neighbor joining. Methods such as ASTRID and NJst (which use this basic approach) are provably statistically consistent, very fast (low degree polynomial time) and have had high accuracy under many conditions that makes them competitive with other popular species tree estimation methods. In this study, inspired by the very recent work of weighted ASTRAL, we present weighted ASTRID, a variant of ASTRID that takes the branch uncertainty on the gene trees into account in the internode distance. Our experimental study evaluating weighted ASTRID shows improvements in accuracy compared to the original (unweighted) ASTRID while remaining fast. Moreover, weighted ASTRID shows competitive accuracy against weighted ASTRAL, the state of the art. Thus, this study provides a new and very fast method for species tree estimation that improves upon ASTRID and has comparable accuracy with the state of the art while remaining much faster. Weighted ASTRID is available at `https://github.com/RuneBlaze/internode`.

2012 ACM Subject Classification Applied computing → Molecular evolution

Keywords and phrases Species tree estimation, ASTRID, ASTRAL, multi-species coalescent, incomplete lineage sorting

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.8

Related Version *Previous Version*: `https://www.biorxiv.org/content/10.1101/2022.05.24.493312v1`

Supplementary Material *Software (Source Code)*: `https://github.com/RuneBlaze/internode` archived at `swh:1:dir:e8b673193c6c6e4a1d11e821fbc9a8ff2e6b74b9`

Funding *Baqiao Liu*: This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Biological and Environmental Research under the Secure Biosystems Design Initiative and by the Laboratory Directed Research and Development (LDRD) program of Sandia National Laboratories, which is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.
Tandy Warnow: NSF grant 2006069

Acknowledgements The authors thank the members of the Warnow lab for insightful comments.

[1] corresponding author

1 Introduction

Species tree estimation is a common task in phylogenomics and is a prior step in many downstream analyses (e.g., estimating divergence, understanding adaptation). Despite the recent increase in the availability of genome-scale data, species tree estimation remains challenging due to gene tree heterogeneity, where gene trees (the evolutionary history of genes) differ from species trees [16]. Among common factors for gene tree heterogeneity, incomplete lineage sorting (ILS), a population-level process modeled statistically by the multi-species coalescent (MSC) [47, 24], is extremely common and well-studied.

A standard approach to species-tree reconstruction under the presence of ILS is to concatenate the alignments of the individual genes and running a maximum likelihood (ML) heuristic on the combined alignment. This simple approach, however, has been established to be statistically inconsistent under the MSC, and can even be positively misleading, converging to the wrong topology with probability 1 as the number of genes increases [43, 42]. Empirically, concatenation can also suffer from degraded accuracy under higher levels of ILS, and can be affected by scalability issues under large data [29, 34]. In response, many accurate ILS-aware methods have since been developed. Those that are most commonly used in practice fall into a class of so-called summary methods, where gene trees are first independently estimated from each genomic region; the inferred gene trees are then used as input to the summary method to "summarize" the input gene trees into a species tree.

In recent years, many accurate summary methods statistically consistent under the MSC have been developed, such as MP-EST [22], NJst [40], ASTRAL [29], ASTRID [49], FASTRAL [9], and wQFM [25]. Many of these methods are scalable to genomic-scale data, and under sufficient gene signal and ILS tend to be more accurate than concatenation [34]. Among these methods, ASTRAL is the most commonly used, compares favorably to other methods in accuracy, and has been successfully applied to many large scale data [33, 55]. However, it is well known that summary methods suffer under inaccurate gene trees [34, 52] as a result of summary methods not explicitly taking gene tree uncertainty into account. Under inaccurate gene trees, it might still be preferable to use concatenation even under substantial ILS [34]. Although co-estimating the gene trees and species trees such as using StarBeast [36] or directly inferring quartets from the alignment and then combining them using SVDquartets+PAUP* [6] circumvents this problem, neither can scale to large data.

This sensitivity of summary methods to gene tree error has motivated approaches pre-processing the gene trees to improve the quality of the signal. Although throwing out inaccurate gene trees generally does not help [34], statistical binning [31, 4] and contracting low-support branches [55] improved accuracy for ASTRAL on many conditions. Nonetheless, these approaches require setting arbitrary thresholds: statistical binning requires a threshold to determine which branches are trustworthy, and contracting low-support branches also requires such a threshold. Suboptimal parameter selection in either case can lead to little accuracy improvement, or even worse, degraded accuracy compared to simply running on the original input [4, 55]. Thus, pragmatically, accurately applying such methods faces the difficulty of parameter selection.

Very recently, Zhang and Mirarab introduced weighted ASTRAL [54]. By directly incorporating gene tree uncertainty into the ASTRAL optimization problem, weighted ASTRAL improved ASTRAL in accuracy under all of their tested conditions. Notably, under conditions where concatenation proved more accurate than unweighted ASTRAL, weighted ASTRAL achieved the largest improvement, shrinking substantially the long known gap [32, 33, 37] between summary methods and concatenation under low gene signal. More

specifically, (unweighted) ASTRAL heuristically searches for a species tree that maximizes the amount of quartet trees (unrooted four-taxon tree) shared with the input gene trees. By using branch support and lengths to weigh the reliability of gene tree quartets, weighted ASTRAL instead heuristically maximizes the weighted agreement with respect to the input gene trees, effectively discounting the contribution of unreliable quartets. Weighted ASTRAL is threshold-free, was shown to be more accurate than running ASTRAL on contracted gene trees [54], and in fact might be the most accurate summary method under ILS that can scale to large datasets.

Here, inspired by weighted ASTRAL, we introduce weighted ASTRID, incorporating gene tree uncertainty into ASTRID. ASTRID, a fast and more accurate variant of NJst, is based on the internode distance, defined by ASTRID as the number of edges between two taxa in a gene tree. We explore variations of this internode distance where branch uncertainty is considered. Notably, ASTRID is shown to have competitive accuracy against ASTRAL [49, 34] while having a much faster running time [49, 9], both of which we hope to generalize to weighted ASTRID when compared against weighted ASTRAL, obtaining a fast alternative to a very accurate method.

The rest of the study is organized as follows. In Section 2 we describe weighted ASTRID and introduce the two ways of weighting the internode distance matrix before providing some theoretical running time bounds. In Section 3 we describe our experimental study, choosing parameters for weighted ASTRID and comparing it to other methods. In Section 4, we present our experimental results showing that support-weighted ASTRID is very fast, is more accurate than ASTRID, has comparable accuracy with weighted ASTRAL, and provides an alternative accurate species-tree inference method robust to low gene signal, whereas branch-length weighted ASTRID has mixed accuracy and is less accurate than support-weighted ASTRID. In Section 5 we summarize our main conclusions and discuss future work.

2 Materials and methods

2.1 Basic definitions

Let n denote the number of taxa and let k denote the number of genes assuming a set of gene trees. Given an unrooted phylogenetic tree T, we denote its leafset by $\mathcal{L}(T)$ and its edge-set $E(T)$. For each edge e in T, deleting e from T partitions the leaves into two sets defined by the two connected components separated by e. Let this bipartition be denoted by π_e for some $e \in E(T)$. We denote the set of bipartitions of T by $C(T) = \{\pi_e \,|\, e \in E(T)\}$, and $C(T)$ uniquely defines the (unrooted) topology of T. A bipartition π_e is said to be trivial if e is incident to a leaf, since in such case $\pi_e \in C(T)$ for any T on the same leafset. Two trees are said to be compatible if the union of their bipartitions can coexist in a single tree. The Robinson-Foulds distance (RF distance) [41] between two trees T and T' on the same leafset is the size of the symmetric difference between the bipartitions of T and T', i.e., $|C(T) \triangle C(T')|$. Given two binary trees T and T', we define the nRF (error) rate as their RF distance normalized by $2n - 6$ (the number of non-trivial bipartitions), obtaining a value that is between 0 and 1.

For taxa $u, v \in \mathcal{L}(T)$, let $P_T(u, v)$ denote the set of edges on the unique path connecting u and v in T. Given an estimated gene tree G, we assume that each internal branch e is associated with a branch support value $s(e)$, some measure of confidence that this branch is correctly estimated, where $s(e) \in [0, 1]$. Note that we say a branch is correctly estimated (in topology) if the bipartition associated with that edge is present in the true gene tree. We

extend this notion of branch support also to pendant edges, where we simply define $s(e) = 1$ if e is incident to a leaf. Similarly, for a true or estimated gene tree G, let $l(e)$ denote the length of a branch, where for estimated gene trees is usually given in substitution units inferred by some maximum likelihood tree inference method.

2.2 Intertaxon-distance based summary method

Under the context of summary methods, given an input of (unrooted) gene trees, our task is to infer an unrooted species tree topology from these input gene trees. A class of summary methods first introduced by Liu and collaborators [23, 21] infers the species tree by first metricizing the gene trees by defining a specific intertaxon distance between leaf nodes on the gene trees, and then for each pair of taxa averages all such distances across the input gene trees. The final averaged distance, represented as a distance matrix, is then fed as input to a distance-based tree inference method, such as UPGMA [28] or neighbor-joining [44], which infers a tree topology that will be the output species tree of this summary method. Here we focus on ASTRID (a faster and more accurate variant of NJst) and specifically describe its most accurate setting under no missing data.

ASTRID metricizes the gene trees by the **internode distance** $d_G(u, v)$ where $u, v \in \mathcal{L}(G)$ is defined to simply be the number of edges in between u, v in G, i.e., $d_G(u, v) = |P_G(u, v)|$. Although this definition from ASTRID differs from the original internode distance from NJst [21], where it was defined as the number of nodes (instead of edges) in between two taxa, this change has essentially no impact for our purposes [2] and generalizes to our later modifications better. Given this measure of internode distance, and assuming that each pair of taxa appears together in some gene tree (no missing data), ASTRID proceeds as follows with the input set of gene trees $\mathcal{G}$:

1. Initialize an $n \times n$ matrix D (n the number of taxa).
2. For each pair of taxa u, v, set $D[u, v]$ to be the empirical mean of $d_G(u, v)$ where G ranges in the input gene trees $\mathcal{G}$ where both u, v are present, i.e., set $D[u, v] = \sum_{G \in \mathcal{G}_{u,v}} d_G(u, v)/|\mathcal{G}_{u,v}|$, where $\mathcal{G}_{u,v}$ is $\{G \,|\, G \in \mathcal{G} \wedge u, v \in \mathcal{L}(G)\}$. This empirical mean is well defined as we assumed $|\mathcal{G}_{u,v}| > 0$.
3. Run FastME's heuristic for balanced minimum evolution [17] (referred to as FastME from here on) on D, outputting an unrooted species tree.

ASTRID is statistically consistent under the MSC when given true gene trees [2], and is in practice very fast while having competitive accuracy [49, 34].

2.3 Weighted ASTRID

In the definition of the internode distance, each branch contributes equally to the internode distance for each pair of taxa it separates. Intuitively, under the realistic assumption that gene trees are estimated with a non-trivial amount of error, some branches will be more reliably estimated than the others. As such it makes sense to assign weights to branches as some confidence of them correctly contributing to the internode distance. For example, because zero-support branches very likely do not contribute correctly to the internode distance, assigning a weight of 0 to such branches likely discounts incorrect contributions to the internode distance. The branch lengths could also be used as such a proxy, because short branches are empirically hard to estimate. Thus, our problem becomes to choose appropriate weighting schemes for the edges based on information already annotated in the gene trees, that is, the branch support and branch lengths. Because branch support is already designed

as some statistical confidence of the correctness of some branch, it seems natural to naively assign the support directly as the weight for each branch. We alternatively explore simply assigning the branch length as the weight. The details are presented as follows.

2.3.1 Distance defined by branch support

We now formally introduce **wASTRID-s** (weighted ASTRID by support), analogous to the naming of weighted ASTRAL by support. As mentioned above, if a branch has low support, it is less likely that this branch contributes correctly to the internode distance, and thus branches with low support should contribute less. Here we try one simple approach, defining each branch's contribution to the internode distance as its support instead of 1, which gives rise to the following definition of $d_G(u, v)$, the new support-weighted intertaxon distance replacing the internode distance from step 2 of ASTRID:

$$d_G(u, v) = \sum_{e \in P_G(u,v)} s(e)$$

In reality, several different measures of support exist with different running time and accuracy trade-offs [3]. Weighted ASTRAL discovered that the approximate Bayesian support [3] of IQ-TREE led to the most accurate species tree reconstruction, although other measures of support also led to accuracy improvements over unweighted ASTRAL. We leave this choice of support as a parameter to be decided later for wASTRID-s.

While ASTRID is statistically consistent under the MSC when given true gene trees [2], the statistical consistency of wASTRID-s only makes sense under *estimated* gene trees because only estimated gene trees can have meaningful branch support. We conjecture that similar to wASTRAL-s (support weighted ASTRAL), wASTRID-s is statistically consistent under the MSC under some probabilistic interpretation of branch support when given estimated gene trees.

2.3.2 Distance defined by branch lengths

Unlike branch support, which is designed to be a measure of statistical confidence on the correctness of a branch, branch lengths can only serve as proxies to such information, where shorter branches are empirically harder to estimate likely as a result of shorter branches containing less information (fewer substitutions) [50]. We do not attempt a complex conversion here, and simply just assign the branch length as the confidence similar to how we use the support values:

$$d_G(u, v) = \sum_{e \in P_G(u,v)} l(e)$$

Notably, this definition of $d_G(u, v)$ coincides with STEAC [23] (motivated by a different perspective of the coalescence time between genes), and potentially under a more accurate setting when paired with the FastME step of ASTRID. In addition, we also explore whether and how to normalize the input branch lengths of the gene trees for the weighting. We name this final algorithm **wASTRID-pl** (weighted ASTRID by path-lengths).

2.4 Running time and fast distance calculation

Clearly, both wASTRID-s and wASTRID-pl retain the original theoretical running time of ASTRID. Recall that n is the number of species and k is the number of gene trees given in the input. For each gene tree, our metricization simply assigns already-computed values

as lengths to each edge; thus calculating the intertaxon distance across all pairs of taxa per gene tree takes $O(n^2)$ time (for normalizing branch lengths in wASTRID-pl, we only explore ways to normalize that does not affect this asymptotic running time). The averaged intertaxon distance thus can be calculated in $O(kn^2)$ time. The running time of FastME (using the balanced minimum evolution heuristic) is $O(n^2 \times \text{diam}(T))$ [8], where $\text{diam}(T)$ is the topological diameter of the output species tree. Assuming the common Yule-Harding distribution [53, 12] or the uniform distribution [10, 1, 27] on the output species tree, the expected diameter is either $\log n$ [10] or $\sqrt{n}$ [8, 10], respectively. Therefore we re-derive the original running time result:

▶ **Theorem 1.** *The averaged intertaxon distance of wASTRID-s and wASTRID-pl can be obtained in $O(kn^2)$ time. The final species tree can be obtained using an additional $O(n^2 \lg n)$ time by FastME assuming the output species tree is under the Yule-Harding distribution (or under the uniform distribution, $O(n^2\sqrt{n})$ time), where n is the number of species and k is the number of genes.*

While the algorithm for the $O(kn^2)$ step can be theoretically uninteresting, we implemented another algorithm for the distance calculation in hope of better in-practice speed. The asymptotically optimal algorithm is easy to devise because the naive algorithm, which given a gene tree, starts a BFS at each leaf to obtain the all-pairs intertaxon distance, is already quadratic time per gene and also asymptotically optimal due to each gene tree having $\binom{n}{2}$ distances. The original ASTRID implementation, in this vein, uses an algorithm which implicitly performs multiple traversals in the tree. For weighted ASTRID, we instead implemented an intertaxon-distance algorithm from TreeSwift [35] based on post-order traversal, through which we hope to achieve better empirical performance due to better cache locality in its simultaneous maintenance of multiple distances from the leaves in an array.

3 Experimental Study

3.1 Overview

We conduct three experiments. In Experiment 1, we explore parameter choices (choice of branch support for wASTRID-s and normalization scheme for wASTRID-pl) for weighted ASTRID. In Experiment 2, we compare the accuracy and running time of weighted ASTRID against other methods on a diverse set of simulated conditions. In Experiment 3, we compare ASTRID, wASTRAL-h, and the best variant of wASTRID (as determined by previous experiments) on the Jarvis et al. [14] avian biological dataset, comparing the quality of the reconstructed species trees.

3.2 Datasets

We assembled a set of diverse data from prior studies (see Table 1), consisting of various simulated conditions with estimated gene trees and one biological dataset ("avian biological") from the avian phylogenomics project [14]. We use the nomenclature of the original ASTRID study and refer to the SimPhy-simulated datasets from the ASTRAL-II study by an "MC" name. The ILS levels of the datasets are measured in average discordance (AD), defined as the average nRF rate between the true species tree and the true gene trees. Same as the original ASTRID study [49], we classify the ILS levels of the datasets into four categories according to their AD percentages, where below 25% is classified as low ILS (L), between 26% and 39% medium ILS (M), between 40% and 59% high ILS (H), and higher AD considered

Table 1 Dataset statistics. The ILS levels of the datasets are categorized according to their AD percentages, where below 25% is low ILS (L), between 26% and 39% mid ILS (M), between 40% and 59% high ILS (H), and higher AD very high ILS (VH). SH-like denotes FastTree default support; BS denotes standard bootstrap support using FastTree or RAxML; aBayes denotes IQ-TREE approximate Bayesian support. (*): training dataset: only 10/50 replicates were used for training.

Dataset	# taxa	# genes	# reps	ILS (AD %)	Branch Support
ASTRAL-II MC2* [33]	201	1000	10	33 (M)	SH-like, BS, aBayes
ASTRAL-III S100 [55]	101	1000	50	46 (H)	aBayes
ASTRAL-II MC3 [33]	201	1000	50	21 (L)	aBayes
ASTRAL-II MC5 [33]	201	1000	50	34 (M)	aBayes
ASTRAL-II MC1 [33]	201	1000	50	69 (VH)	aBayes
ASTRAL-II MC6H [20]	201	1000	50	9 (L)	aBayes
ASTRAL-II MC11H [20]	1001	1000	50	35 (M)	aBayes
Avian 2x [31]	48	1000	20	29 (M)	aBayes
Avian 1x [31]	48	1000	20	47 (H)	aBayes
Avian 0.5x [31]	48	1000	20	60 (VH)	aBayes
Mammalian 2x [31]	37	200	20	21 (L)	aBayes
Mammalian 1x [31]	37	200	20	29 (M)	aBayes
Mammalian 0.5x [31]	37	200	20	50 (H)	aBayes
Jarvis et al. avian [14]	48	14446	1	not known	BS

very high ILS (VH). For the simulated conditions, we subsample the gene trees to size $k = 50, 200, 1000$ except for the mammalian simulation ($k = 50, 100, 200$ instead as only 200 gene trees were provided).

For the measure of support for the datasets, the weighted ASTRAL study provided gene trees reannotated with aBayes support and IQ-TREE inferred branch lengths for the ASTRAL-II and ASTRAL-III datasets. We also reannotated the avian and mammalian simulation with aBayes support and IQ-TREE branch lengths because aBayes was determined as the best measure of support also for wASTRID-s. We use the MC2 condition as the training data for both wASTRID-s and wASTRID-pl, where to explore the choice of branch support for wASTRID-s, we also took the original FastTree [38] inferred trees annotated with the default FastTree SH-like support and also computed a version of the gene trees with standard bootstrap support [11] using 100 FastTree trees.

3.3 Methods

For testing, we compare against ASTRID, ASTRAL, and weighted ASTRAL, with the following settings:

- **ASTRID** (v2.2.1), the base method of weighted ASTRID. We use the original implementation, turning off missing data imputation, as our data has no missing entries in the final averaged matrix.
- **ASTRAL(-III)** (v5.7.8). We do not contract very low support edges in the gene trees because a good threshold can be hard to determine across datasets. In addition, no contraction allows us the fully explore the impact of weighting.
- **wASTRAL-h** (hybrid weighted ASTRAL, v.1.4.2.3). This was the most accurate weighted ASTRAL from the original study, using both branch lengths and support to weigh gene tree quartets. wASTRAL-h supports parallelization, so we run wASTRAL-h with 16 threads.

Many of the datasets have FastTree-inferred gene trees that were reannotated with IQ-TREE approximate Bayesian support. FastTree-inferred trees have polytomies for identical sequences, but these polytomies will be resolved when the trees get reannotated by IQ-TREE, adding false positive edges which may adversely affect the accuracy of the unweighted methods. In these cases, we run the unweighted methods on the original FastTree gene trees.

3.4 Evaluation criterion

For simulated datasets, we compare the topological error rate of the reconstructed species trees using the normalized Robinson-Foulds error (nRF error) with respect to the true species trees. Because all the inferred and true species trees are binary, the nRF error rate is equivalent to the missing branch rate (ratio of branches in the reference tree missing from the reconstructed tree).

On the avian biological dataset, as the true tree is not known, we compare the estimated species trees against prior topologies (wASTRAL tree and published trees). We also compute the local posterior-probability (localPP) branch support [45] for the reconstructed species trees obtained using wASTARL-h to assess the reliability of the branches.

On all datasets, we keep track of the wall-clock running time of the methods, the time taken from consuming the input gene trees (that may have been preprocessed with new branch support values) until outputting the species tree.

3.5 Experimental Environment

All experiments were conducted on the Illinois Campus Cluster, a heterogeneous cluster that has a four-hour running time limit. The heterogeneity of the hardware makes the wall-clock running times not directly comparable across runs, but can still be used to gather obvious running time trends.

4 Results & Discussion

4.1 Experiment 1: Parameter selection

In Figure 1, we explore the choice of branch support among the default FastTree SH-like support, IQ-TREE approximate Bayesian (aBayes) support (normalized to the $[0, 1]$ range), and bootstrap support (100 FastTree replicates) on the training datasets. The best accuracy of wASTRID-s is obtained by using the normalized aBayes support on gene trees. All measures of support, however, improved the species tree estimation error in general. The superiority of (normalized) aBayes support is consistent with the support chosen for weighted ASTRAL, where it was also found superior to SH-like support and bootstrap support. This advantage is even more pronounced when considering that aBayes support can be obtained much faster than bootstrap support [3].

On this training dataset, wASTRID-pl attained the highest accuracy when normalizing the branch lengths in each gene tree by the maximum path length in that gene tree (better than no normalization). Interestingly, while worse than ASTRID with fewer genes $k \in \{50, 200\}$, wASTRID-pl attained higher accuracy than ASTRID when $k = 1000$. However, when comparing wASTRID-s and wASTRID-pl, wASTRID-pl was always less accurate.

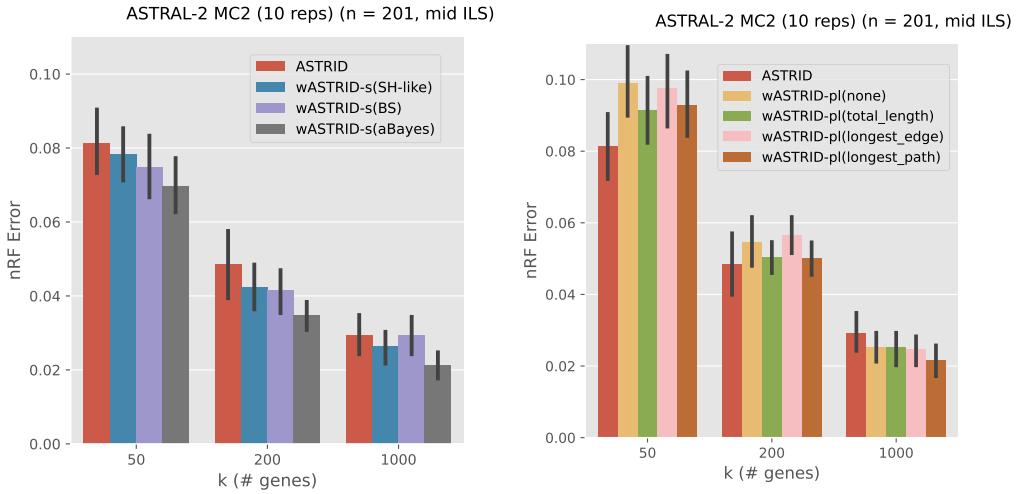


Figure 1 Comparison of the choice of branch support and the choice of branch length normalization strategy for wASTRID-s and wASTRID-pl respectively on the training data, showing the species tree topological error rates (nRF error). "SH-like" is FastTree SH-like support. "aBayes" is IQ-TREE approximate Bayesian support. "BS" is bootstrap support using 100 FastTree trees. The x-axis varies in the number of genes k in the input. Results are shown averaged across ten replicates. Error bars show standard error.

4.2 Experiment 2: Results on simulated datasets

In this experiment, we show four-way comparisons among ASTRID, ASTRAL, weighted ASTRAL (wASTRAL-h), and weighted ASTRID (wASTRID-s). We omit showing the branch-length weighted wASTRID-pl, as it was discovered to be on all datasets less accurate than wASTRID-s. We put an emphasis on the accuracy (nRF error), while later revisiting the problem of running time.

4.2.1 ASTRAL-III S100

This 101-species dataset contains four conditions that varied in the gene tree estimation error (GTEE, measured by the average nRF error between the estimated gene trees and their corresponding true gene trees) by varying the sequence lengths. We show the results of three of the four conditions in Figure 2. In all such figures, we show the unweighted methods (ASTRID, ASTRAL) in dotted lines. On S100, aside from the obvious trend that summary methods become more accurate as k (the number of genes) increases, all methods also improve in accuracy when given more accurate estimated gene trees. These two trends are unsurprising and well-documented across studies for summary methods in general.

More interestingly, the weighted methods (wASTRID-s, wASTRAL-h) are clearly more accurate than their unweighted counterparts, especially at higher levels of GTEE (GTEE $= 0.55, 0.42$). The improvement in accuracy from the weighted methods does not seem to depend on the number of genes, suggesting that the noise brought by low-quality gene trees is not simply resolved by having ample data. This advantage of the weighted methods, however, is smaller as more accurate gene trees are used (GTEE $= 0.31$), as expected.

wASTRID-s clearly improves upon ASTRID on this dataset, with an obvious advantage in the nRF error across all conditions and all numbers of genes. wASTRID-s notably almost matches the accuracy of wASTRAL-h. With $k \in \{200, 1000\}$, no clear benefit exists for using wASTRAL-h on the shown conditions.

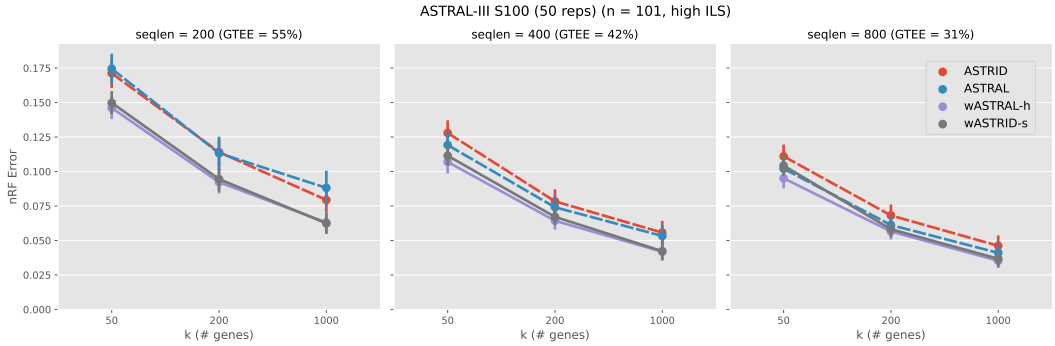


Figure 2 Topological error of species tree across methods on the ASTRAL-III S100 dataset ($n = 101, \text{AD} = 46\%$). Subfigures vary the sequence lengths, affecting the gene tree estimation error (measured in GTEE, the average distance between estimated gene trees and true gene trees). The x-axis varies in the number of genes. Results are shown averaged across 50 replicates with standard error bars. All weighted methods (wASTRID, wASTRAL) ran on gene trees reannotated with IQ-TREE aBayes support branch support and lengths. All methods achieve better accuracy when given more (larger k) or better (lower GTEE) data. Weighted methods are more accurate than unweighted ones. wASTRID-s and wASTRAL-h have almost the same accuracy.

On this dataset, wASTRID-pl (as seen in full results in Appendix Figure 8), similar to trends seen in the training dataset, attains better accuracy than ASTRID when $k = 1000$, but is almost always worse than wASTRID-s. While this better accuracy than ASTRID suggests potential for wASTRID-pl and STEAC-like methods in general, this accuracy disadvantage to wASTRID-s led to us dropping wASTRID-pl from future experiments.

4.2.2 ASTRAL-II SimPhy

This dataset was generated varying the speciation rates, ILS level, and number of taxa, with the "H"-suffixed conditions regenerated from a previous study [20] halving sequence lengths to increase GTEE. We show the results in Figure 3 (only three out of five of the conditions visualized for brevity; see Appendix Figure 9 for the full results).

Across all conditions in this dataset, the weighted methods are more accurate than the unweighted methods. This advantage does not seem to depend on the level of ILS or the number of species. Even under the easiest condition (MC3), wASTRAL-h and wASTRID-s still consistently achieved better accuracy. All methods also performed worse in accuracy as ILS increased, as expected.

While wASTRID-s still consistently improved upon ASTRID in accuracy on this dataset, we also see datasets where wASTRID-s is worse than wASTRAL-h (MC1 as shown here and MC6H as shown in Appendix Figure 9). The relative performance of wASTRID-s and wASTRAL-h seems related to the relative performance of the base methods: MC1 and MC6H are the two conditions that ASTRAL is in general more accurate than ASTRID, but the relative performance of the base methods does not explain the whole picture – for MC1 going to $k = 1000$, ASTRID became more accurate than ASTRAL yet wASTRID-s is still worse than wASTRAL-h. More positively, on the other conditions of this dataset, wASTRAL-h has nearly the same accuracy as wASTRID-s, although wASTRAL-h is marginally more accurate, which might be due to the hybrid weighting of wASTRAL-h also using the branch lengths.

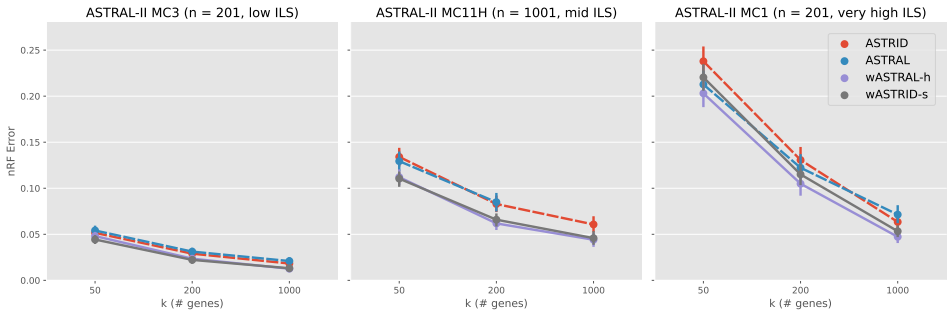


Figure 3 Topological error (nRF error rate) of species tree across methods on selected conditions on the ASTRAL-II SimPhy conditions ($n = 201, 1001, 201$, AD $= 21, 35, 69\%$ respectively). Each subfigure depicts a different model condition. The x-axis varies in the number of genes. Results are shown averaged across 50 replicates with standard error bars. ASTRAL did not finish 24 out of the 50 replicates within four hours for $k = 1000$ on MC11H and thus the data point was omitted. All weighted methods (wASTRID, wASTRAL) were run on gene trees reannotated with IQ-TREE aBayes support branch support and lengths. Weighted methods are more accurate than unweighted ones. wASTRID-s on MC1 was less accurate than wASTRAL-h and otherwise has the same accuracy.

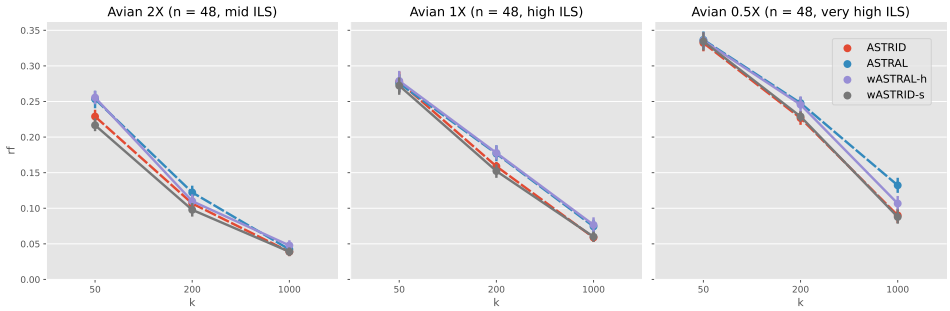


Figure 4 Topological error (nRF error rate) of species tree across methods on the avian simulation ($n = 48$). Each subfigure depicts a different model condition. The x-axis varies in the number of genes. Results are shown averaged across 20 replicates with standard error bars. All weighted methods (wASTRID, wASTRAL) ran on gene trees reannotated with IQ-TREE aBayes support branch support and lengths. ASTRID and wASTRID-s are more accurate than ASTRAL and wASTRAL-h, with a slight accuracy advantage to the weighted methods over the unweighted ones.

On the largest input of these conditions (MC11H, $k = 1000$), ASTRAL did not finish under our four-hour time limit on around half of the replicates (see Appendix Section A.7 for more details), but wASTRAL-h did. We comment on this scalability advantage of wASTRAL-h over ASTRAL later.

4.2.3 Avian and mammalian simulation

These two datasets were generated based on model trees inferred on biological datasets. Both datasets have three conditions with varying ILS by scaling the model tree branch lengths by 2X, 1X, or 0.5X, with shorter branch lengths leading to higher degrees of ILS. Notably, prior results from the ASTRID study [49] showed that ASTRID outperformed ASTRAL on the avian simulation, while on the mammalian simulation ASTRAL was more accurate. Also, the mammalian simulation only has 200 genes available, so we vary k among $50, 100, 200$ unlike the other datasets.

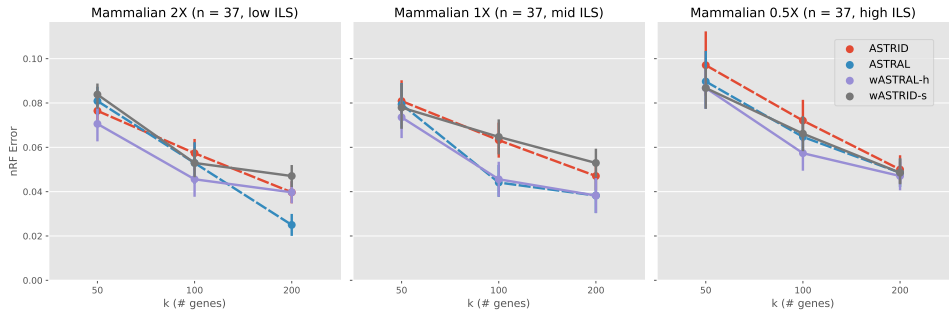


Figure 5 Topological error (nRF error rate) of species tree across methods on the mammalian simulation ($n = 37$). Each subfigure depicts a different model condition. The x-axis varies in the number of genes. Results are shown averaged across 20 replicates with standard error bars. All weighted methods (wASTRID, wASTRAL) ran on gene trees reannotated with IQ-TREE aBayes support branch support and lengths. ASTRAL and wASTRAL-h are more accurate than ASTRID and wASTRID-s. The weighted methods have mixed accuracy compared to the unweighted ones.

On the avian simulation (Figure 4), aside from obvious trends (ILS increases difficulty; more genes leads to more accurate reconstruction), same as the original study, ASTRID is consistently more accurate than ASTRAL. Strangely, although the weighted methods inherit the relative performance of their base methods, in a few cases the weighted methods do not help in accuracy, but they do not erode accuracy either. Even on conditions where the weighted methods improved accuracy, the improvement was small. For example, wASTRAL-h, even though improving upon ASTRAL, is even less accurate than ASTRID, whereas on previously shown data wASTRAL-h was consistently the best in accuracy. This avian simulation does carry substantial GTEE ($> 50\%$), so it is not clear what led to the weighted methods underperforming.

The results for the mammalian simulation (Figure 5) paint a more perplexing picture. On the 2X condition, surprisingly, the weighted methods are less accurate than their unweighted counterparts in general. This trend continues with the 1X condition, where wASTRAL-h only mostly matches ASTRAL in accuracy, and wASTRID-s is worse than ASTRID in accuracy. Only on the 0.5X condition, both weighted methods clearly help in accuracy. wASTRAL-h is clearly better than wASTRID-s on this dataset, but this difference can be explained by the accuracy advantage of ASTRAL on ASTRID. While it is again unclear why the weighted methods underperformed, this dataset is relatively easy compared to the previously shown datasets, with all methods achieving around 0.05 nRF error rate even with $k = 200$, so despite the puzzling relative performance, the difference in accuracy among methods is very small.

4.2.4 Running time

We show the wall-clock running time of the four methods under three representative conditions ($n = 101, 201, 1001$) in Table 2, with a direct comparison of the two most accurate methods visualized in Figure 6. While the heterogeneity of the hardware dilutes the comparability of the running times, clearly wASTRID-s and ASTRID are much faster than wASTRAL-h and ASTRAL, with wASTRID-s on average taking less than 12 seconds even on the largest input, whereas on the same input wASTRAL-h on average takes roughly two hours. In general, wASTRID-s is around two orders of magnitude faster than wASTRAL-h. Although we note that the default flags of both ASTRAL and wASTRAL-h (that we used in the experiments)

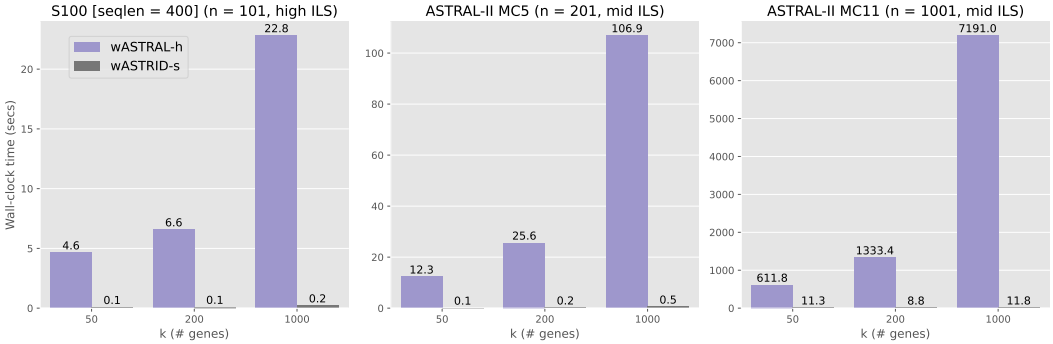


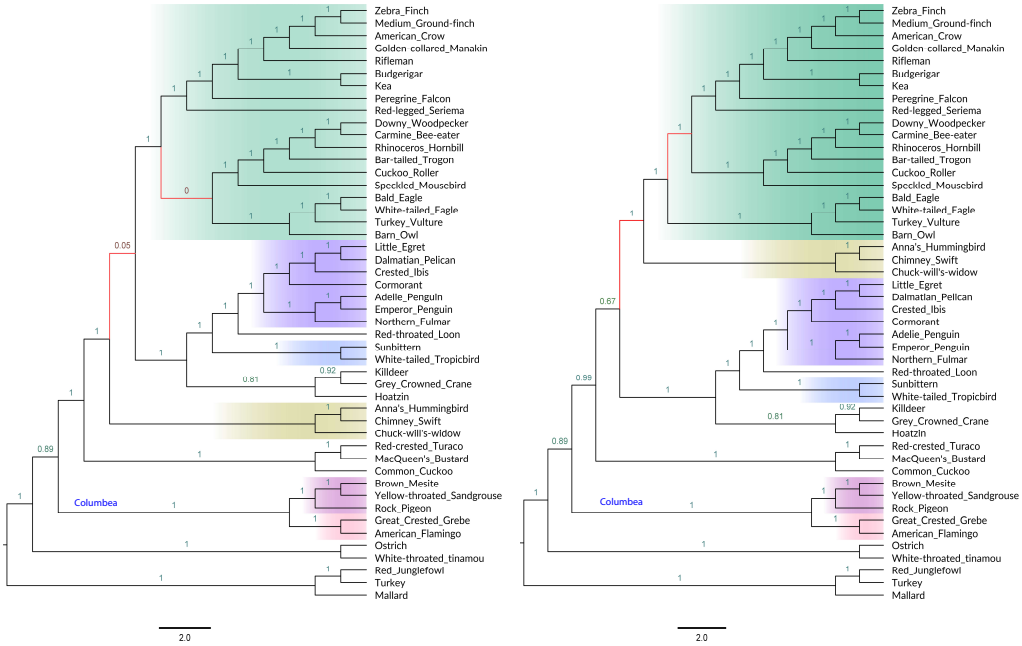
Figure 6 Wall-clock running time (sec) comparison of wASTRID-s and wASTRAL-h on selected representative simulated conditions on $n = 101, 201, 1001$ for k ranging in $50, 200, 1000$. Bars and labels show averages across 50 replicates. wASTRID-s is dramatically faster than wASTRAL-h.

Table 2 Wall-clock running time (sec) across methods on selected representative simulated conditions on $n = 101, 201, 1001$ for k ranging in $50, 200, 1000$. Data points show averages across 50 replicates. ASTRAL did not finish 24 out of the 50 replicates within four hours for $k = 1000$ on MC11H and thus the data point was omitted. The methods sorted by the fastest to the slowest are almost always wASTRID-s, ASTRID, wASTRAL-h, and ASTRAL across all shown conditions. ASTRID and wASTRID-s are much faster than ASTRAL and wASTRAL-h.

Running time (s)	k (# genes)	ASTRAL	ASTRID	wASTRAL-h	wASTRID-s
S100 ($n = 101$)	50	12.1	0.1	4.6	0.1
	200	29.6	0.2	6.6	0.1
	1000	300.7	1.4	22.8	0.2
MC5 ($n = 201$)	50	20.3	0.2	12.3	0.1
	200	57.5	0.6	25.6	0.2
	1000	600.7	1.8	106.9	0.5
MC11H ($n = 1001$)	50	552.9	18.0	611.8	11.3
	200	2059.6	14.9	1333.4	8.8
	1000	–	27.5	7191.0	11.8

also calculate support and lengths for reconstructed species tree, in practice, this is a fast step relative to the species tree reconstruction, and does not affect our running time analysis in any substantial way. On MC11H, wASTRID-s and ASTRID took less time going from $k = 50$ to 200, likely due to the $k = 50$ trees having larger diameters, negatively impacting the FastME step running time which has a linear dependency on the diameter of the output tree.

The weighted methods are faster than their unweighted counterparts. For example on S100 (seqlen = 400), wASTRAL-h under $k = 1000$ is more than ten times faster than ASTRAL, and ASTRAL did not finish around half of the datasets for the largest input (MC11H, $k = 1000$). Aside from the benefit of parallelization (we ran wASTRAL-h using 16 threads, but off-the-shelf ASTRAL does not support parallelization), this speed advantage under a large number of genes of wASTRAL-h over ASTRAL can also be attributed to the algorithmic change implemented in wASTRAL-h. The new weighted ASTRAL algorithm removes the in-practice quadratic dependency of ASTRAL's search algorithm on the number



(a) wASTRID-s avian tree.

(b) wASTRAL-h avian tree.

Figure 7 Results on the avian biological dataset ($n = 48, k = 14446$). In (a) and (b), we show the reconstructed species tree topology of wASTRID-s and wASTRAL-h, annotated with local posterior-probability branch support (localPP) computed using wASTRAL-h. The red branches show where the two trees differ. The six well corroborated clades according to Braun and Kimball [5] are displayed by both trees and highlighted in gradients. For wASTRID-s, the red branches also coincide with the only two low support branches. Contracting the very low support branches for wASTRID-s arrives at a topology compatible with wASTRAL-h. wASTRAL-h took around 294 seconds to infer its tree. wASTRID-s was very fast and took 1 second to infer its topology.

of genes [30] and instead has a linear dependency on k for the running time. wASTRID-s is consistently faster (at least two times faster on most conditions) than ASTRID, showing that the new distance calculation algorithm implemented is more efficient than the original one.

4.3 Experiment 3: Results on the avian biological dataset

Jarvis et al. studied the phylogeny of birds using a dataset on 48 taxa using 14446 genes [14]. The original gene trees were annotated with RAxML [46] bootstrap support, which we directly use in our wASTRID-s analysis. This dataset is known to have very low gene resolution, with the average support only 32% [31]. ASTRAL on the original set of gene trees reconstructed a species tree that contained obvious inaccuracies, but contracting low support branches enabled ASTRAL to construct a very plausible topology in agreement with established results [55]. Zhang and Mirarab reanalyzed the original gene trees using wASTRAL-h, and reconstructed the same topology as ASTRAL running on contracted gene trees [54]. In addition to directly comparing against their wASTRAL-h tree, we also compute the number of differing branches between the inferred trees and the two published trees of the Jarvis et al. study: the ExaML-based concatenation tree, called the TENT ("total evidence nucleotide tree"), and the coalescent-based published tree based on binned MP-EST

(MP-EST* tree). We show the results in Figure 7. In addition to showing the topology and the localPP branch support, we also highlight in gradients six clades that are thought to be strongly corroborated [5] for avian phylogeny.

The ASTRID tree (shown in Appendix Figure 10) differed in eight or nine edges with each of the wASTRAL-h, TENT, and MP-EST* trees. In addition, the ASTRID tree did not recover the Columbea clade, which is a clade that has seen strong support in various analyses of this data [14, 55, 39].

Using wASTRID-s, we recovered a topology that is much more in agreement with the other trees, differing in two branches (4.4% of the branches) with the wASTRAL-h tree. It is also in much higher agreement with the published trees (differing in three branches with the TENT, two branches with the MP-EST* tree). Looking closer, the two branches where it differs from the wASTRAL-h tree coincide with the only two very low support branches (no more than 5% in support), and thus contracting the very low support branches arrives at a topology compatible with the wASTRAL-h tree. Both wASTRID-s and wASTRAL-h recover the Columbea-Passerea split, a major conclusion in the original analysis of this data, and even agree on the placement of the hard-to-place Hoatzin.

For running time, both ASTRID and wASTRID-s finished quickly. ASTRID completed in 6.4 seconds, and wASTRID-s completed in 1.1 seconds. Our rerun of wASTRAL-h on this data finished in 294.2 seconds, showcasing the much better scalability of the new weighted ASTRAL optimization algorithm in the number of genes, whereas ASTRAL on the same input took 32 hours in the ASTRAL-III study. In summary, on this dataset, wASTRID-s inferred a more accurate tree compared to ASTRID, is much faster than wASTRAL-h, and is compatible with wASTRAL-h after contraction of two very low support branches.

4.4 Additional remarks

Impact of number of genes, ILS, and GTEE

In all results shown, all methods achieved better accuracy when given more genes, and all methods have decreased accuracy as the degree of ILS or GTEE increases. These trends are well known and quite expected for all summary methods. The number of species does not seem to influence the accuracy of the methods. The relative performance of the methods does shift across different numbers of genes, but there seems to be no reliable predictor of this change across the conditions. All methods seem equally adversely impacted by ILS in accuracy, but for GTEE, the weighted methods are seen to be more robust to the low signal, as by design the weighted methods can extract better signals from the gene trees under high GTEE.

Impact of weighting

Weighting improved accuracy on all the datasets (simulated and biological) except on the mammalian simulation, and the improvement of accuracy tends to be the same degree for both wASTRAL-h and wASTRID-s, with a slight advantage to wASTRAL-h, likely due to the hybrid weighting using more information than support-weighted ASTRID. Even with ample genes or lower levels of GTEE, the weighted methods still in general improved upon the unweighted variants in accuracy. It is not clear how the mammalian simulation (and to some degree, the avian simulation) differs from the other datasets, where the weighted methods could in cases be worse than the original methods in accuracy. The mammalian simulation has a non-trivial amount of GTEE, but has the smallest number of species among the tested datasets and was generated from a different protocol than the ASTRAL simulation

datasets, and all these factors could potentially affect the accuracy. Although the weighted methods are in general faster than their unweighted counterparts, this speed advantage is orthogonal to the weighting and entirely due to faster algorithmic design or implementation.

wASTRAL-h vs. wASTRID-s

ASTRAL and ASTRID are known to be among the most accurate summary methods under ILS, and their relative accuracy is dataset dependent, as also shown in our results. The relative accuracy of their weighted counterparts is clearly influenced by the accuracy of their base methods, as seen in the performance differences in the avian and mammalian simulation, where either wASTRID-s and wASTRAL-h can be the more accurate method. On the ASTRAL simulated datasets (ASTRAL-III S100, ASTRAL-II SimPhy), wASTRID-s and wASTRAL-h have comparable accuracy, with a small advantage to wASTRAL-h, which might be due to wASTRAL-h's more effective hybrid weighting. Somewhat unsurprisingly, on all conditions except the mammalian simulation, wASTRID-s is more accurate than unweighted ASTRAL, even on conditions where ASTRAL is more accurate than ASTRID.

As for running time, wASTRID-s is clearly the winner. Despite wASTRAL-h's improved algorithm and parallelism, wASTRID-s is still two orders of magnitude faster than wASTRAL-h, and hence can provide better scalability on large-scale data. We do not extend this running time discussion to a highly parallelized setting (e.g., 64 cores, common for large-scale data analysis) since wASTRAL-h has not been efficiently parallelized yet (as noted in their future work), so any current discussion likely does not reflect the future parallel efficiency of wASTRAL-h. The wASTRID-s algorithm is trivially parallelizable in its distance calculation (albeit very fast already), but parallelizing the FastME step can be difficult both in design and in implementation, which can be a bottleneck under large n. Intriguingly, wASTRAL-h theoretically can scale better in the number of species than wASTRID-s as wASTRAL-h has a running time that is subquadratic in the number of species, but our results show that at $n = 1001$ this point is far from being reached. Thus wASTRID-s is much faster than wASTRAL-h, while having comparable accuracy.

5 Conclusions

While the estimation of species trees using summary methods, such as ASTRAL, ASTRID, MP-EST, and others, is now commonplace, it is known that gene tree estimation error reduces the accuracy of the estimated species tree. We presented support weighted ASTRID (wASTRID-s), an improvement over ASTRID that incorporates uncertainty in gene tree branches into its estimation of the averaged internode distance. Our work is largely inspired by the recent work of weighted ASTRAL (wASTRAL-h), which improved upon ASTRAL, and we showed that wASTRID-s obtained similar accuracy improvements over ASTRID. The advantage provided by wASTRID-s over ASTRID is most noteworthy under higher degrees of gene tree estimation error. wASTRID-s has very close accuracy to wASTRAL-h and is sometimes more accurate, but overall wASTRAL-h has a small advantage in accuracy. The improvement of wASTRAL-h over wASTRID-s may be due to its weighting also incorporating branch lengths. A branch-length weighted version of ASTRID (wASTRID-pl) has mixed accuracy compared to ASTRID, and does not compete against the support-weighted wASTRID-s in accuracy. In general, wASTRID-s and wASTRAL-h serve as accurate species tree inference methods under ILS and are more robust to GTEE than ASTRID and ASTRAL, two of the most accurate summary methods. Both can scale very well, with wASTRID-s much faster. However, the differences in accuracy are dataset dependent, just as the comparison between ASTRAL and ASTRID for accuracy seems dataset dependent.

This study was limited to datasets where the only cause for gene tree discordance with the species tree was ILS and gene tree estimation error. When considering real world datasets that may have other sources of gene tree heterogeneity, such as GDL or HGT, it seems likely that ASTRAL and other quartet-based methods may have an advantage over ASTRID and wASTRID-s, due to the theoretical proofs of statistical consistency for quartet-based methods for those conditions (and the lack of such proofs for distance-based species tree estimation methods) [18, 26, 7, 13].

Although wASTRID-s is highly accurate and very fast, we recommend using wASTRID-s in conjunction with wASTRAL-h and other species tree estimation methods. Due to its speed, the inclusion of wASTRID-s adds little computational burden, and having multiple different approaches for estimating the species tree, each based on a very different (but statistically consistent) technique, can provide insights into what parts of the species tree are most reliably recovered, and which parts may need further data in order for full resolution.

For future work, finding a way to incorporate branch lengths into the branch certainty scoring for wASTRID-s, a hybrid weighting, will improve accuracy and might close the accuracy gap between wASTRID-s and wASTRAL-h under some conditions. Missing data in the final averaged distance matrix has not been addressed, and generalizing the approach from ASTRID [48] for handling missing entries to weighted ASTRID will be important. ASTRID has been used to speed-up and sometimes improve analyses for ASTRAL by assisting in constraining the search space explored by ASTRAL (e.g., FASTRAL [9]), suggesting that a weighted version of ASTRID might provide even better accuracy improvements in FASTRAL. Another direction for future work is species tree estimation in the presence of gene duplication and loss, where gene family trees have multiple copies of species and so are called MUL-trees. The combination of DISCO [51], a method for decomposing the MUL-trees into single-copy gene trees, with ASTRID produced very good accuracy and scalability [51], suggesting that combining DISCO with wASTRID might be even more accurate.

References

1. David J. Aldous. Stochastic Models and Descriptive Statistics for Phylogenetic Trees, from Yule to Today. *Statistical Science*, 16(1):23–34, 2001. URL: https://www.jstor.org/stable/2676778.
2. Elizabeth S. Allman, James H. Degnan, and John A. Rhodes. Species tree inference from gene splits by unrooted star methods. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 15(1):337–342, 2018. doi:10.1109/TCBB.2016.2604812.
3. Maria Anisimova, Manuel Gil, Jean-François Dufayard, Christophe Dessimoz, and Olivier Gascuel. Survey of Branch Support Methods Demonstrates Accuracy, Power, and Robustness of Fast Likelihood-based Approximation Schemes. *Systematic Biology*, 60(5):685–699, October 2011. doi:10.1093/sysbio/syr041.
4. Md Shamsuzzoha Bayzid, Siavash Mirarab, Bastien Boussau, and Tandy Warnow. Weighted Statistical Binning: Enabling Statistically Consistent Genome-Scale Phylogenetic Analyses. *PLOS ONE*, 10(6):e0129183, June 2015. doi:10.1371/journal.pone.0129183.
5. Edward L. Braun and Rebecca T. Kimball. Data types and the phylogeny of neoaves. *Birds*, 2(1):1–22, 2021. doi:10.3390/birds2010001.
6. Julia Chifman and Laura Kubatko. Quartet Inference from SNP Data Under the Coalescent Model. *Bioinformatics*, 30(23):3317–3324, December 2014. doi:10.1093/bioinformatics/btu530.
7. Constantinos Daskalakis and Sébastien Roch. Species trees from gene trees despite a high rate of lateral genetic transfer: A tight bound. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1621–1630. SIAM, 2016.

8 Richard Desper and Olivier Gascuel. Fast and accurate phylogeny reconstruction algorithms based on the minimum-evolution principle. *Journal of Computational Biology*, 9(5):687–705, 2002. PMID: 12487758. doi:10.1089/106652702761034136.

9 Payam Dibaeinia, Shayan Tabe-Bordbar, and Tandy Warnow. FASTRAL: improving scalability of phylogenomic analysis. *Bioinformatics*, 37(16):2317–2324, August 2021. doi:10.1093/bioinformatics/btab093.

10 Péter L. Erdös, Michael A. Steel, László A. Székely, and Tandy J. Warnow. A few logs suffice to build (almost) all trees: Part II. *Theoretical Computer Science*, 221(1):77–118, June 1999. doi:10.1016/S0304-3975(99)00028-6.

11 Joseph Felsenstein. Confidence Limits on Phylogenies: An Approach Using the Bootstrap. *Evolution*, 39(4):783–791, 1985. doi:10.1111/j.1558-5646.1985.tb00420.x.

12 E. F. Harding. The probabilities of rooted tree-shapes generated by random bifurcation. *Advances in Applied Probability*, 3(1):44–77, 1971. doi:10.2307/1426329.

13 Max Hill, Brandon Legried, and Sébastien Roch. Species tree estimation under joint modeling of coalescence and duplication: sample complexity of quartet methods. *arXiv preprint*, 2020. arXiv:2007.06697.

14 Erich D. Jarvis, Siavash Mirarab, Andre J. Aberer, Bo Li, Peter Houde, Cai Li, Simon Y. W. Ho, Brant C. Faircloth, Benoit Nabholz, Jason T. Howard, Alexander Suh, Claudia C. Weber, Rute R. da Fonseca, Jianwen Li, Fang Zhang, Hui Li, Long Zhou, Nitish Narula, Liang Liu, Ganesh Ganapathy, Bastien Boussau, Md. Shamsuzzoha Bayzid, Volodymyr Zavidovych, Sankar Subramanian, Toni Gabaldón, Salvador Capella-Gutiérrez, Jaime Huerta-Cepas, Bhanu Rekepalli, Kasper Munch, Mikkel Schierup, Bent Lindow, Wesley C. Warren, David Ray, Richard E. Green, Michael W. Bruford, Xiangjiang Zhan, Andrew Dixon, Shengbin Li, Ning Li, Yinhua Huang, Elizabeth P. Derryberry, Mads Frost Bertelsen, Frederick H. Sheldon, Robb T. Brumfield, Claudio V. Mello, Peter V. Lovell, Morgan Wirthlin, Maria Paula Cruz Schneider, Francisco Prosdocimi, José Alfredo Samaniego, Amhed Missael Vargas Velazquez, Alonzo Alfaro-Núñez, Paula F. Campos, Bent Petersen, Thomas Sicheritz-Ponten, An Pas, Tom Bailey, Paul Scofield, Michael Bunce, David M. Lambert, Qi Zhou, Polina Perelman, Amy C. Driskell, Beth Shapiro, Zijun Xiong, Yongli Zeng, Shiping Liu, Zhenyu Li, Binghang Liu, Kui Wu, Jin Xiao, Xiong Yinqi, Qiuemei Zheng, Yong Zhang, Huanming Yang, Jian Wang, Linnea Smeds, Frank E. Rheindt, Michael Braun, Jon Fjeldsa, Ludovic Orlando, F. Keith Barker, Knud Andreas Jønsson, Warren Johnson, Klaus-Peter Koepfli, Stephen O'Brien, David Haussler, Oliver A. Ryder, Carsten Rahbek, Eske Willerslev, Gary R. Graves, Travis C. Glenn, John McCormack, Dave Burt, Hans Ellegren, Per Alström, Scott V. Edwards, Alexandros Stamatakis, David P. Mindell, Joel Cracraft, Edward L. Braun, Tandy Warnow, Wang Jun, M. Thomas P. Gilbert, and Guojie Zhang. Whole-genome analyses resolve early branches in the tree of life of modern birds. *Science*, 346(6215):1320–1331, December 2014. doi:10.1126/science.1253451.

15 Alexey M Kozlov, Diego Darriba, Tomáš Flouri, Benoit Morel, and Alexandros Stamatakis. RAxML-NG: a fast, scalable and user-friendly tool for maximum likelihood phylogenetic inference. *Bioinformatics*, 35(21):4453–4455, May 2019. doi:10.1093/bioinformatics/btz305.

16 Laura Salter Kubatko and James H. Degnan. Inconsistency of Phylogenetic Estimates from Concatenated Data under Coalescence. *Systematic Biology*, 56(1):17–24, February 2007. doi:10.1080/10635150601146041.

17 Vincent Lefort, Richard Desper, and Olivier Gascuel. FastME 2.0: A Comprehensive, Accurate, and Fast Distance-Based Phylogeny Inference Program. *Molecular Biology and Evolution*, 32(10):2798–2800, October 2015. doi:10.1093/molbev/msv150.

18 Brandon Legried, Erin K Molloy, Tandy Warnow, and Sébastien Roch. Polynomial-time statistical estimation of species trees under gene duplication and loss. *Journal of Computational Biology*, 28(5):452–468, 2021.

19 Frédéric Lemoine and Olivier Gascuel. Gotree/Goalign: toolkit and Go API to facilitate the development of phylogenetic workflows. *NAR Genomics and Bioinformatics*, 3(3), August 2021. doi:10.1093/nargab/lqab075.

20 Baqiao Liu and Tandy Warnow. Data from scalable species tree inference with external constraints, 2021. University of Illinois at Urbana-Champaign. doi:10.13012/B2IDB-2566000_V1.

21 Liang Liu and Lili Yu. Estimating Species Trees from Unrooted Gene Trees. *Systematic Biology*, 60(5):661–667, October 2011. doi:10.1093/sysbio/syr027.

22 Liang Liu, Lili Yu, and Scott V. Edwards. A maximum pseudo-likelihood approach for estimating species trees under the coalescent model. *BMC Evolutionary Biology*, 10(1):302, October 2010. doi:10.1186/1471-2148-10-302.

23 Liang Liu, Lili Yu, Dennis K. Pearl, and Scott V. Edwards. Estimating Species Phylogenies Using Coalescence Times among Sequences. *Systematic Biology*, 58(5):468–477, October 2009. doi:10.1093/sysbio/syp031.

24 Wayne P. Maddison. Gene Trees in Species Trees. *Systematic Biology*, 46(3):523–536, September 1997. doi:10.1093/sysbio/46.3.523.

25 Mahim Mahbub, Zahin Wahab, Rezwana Reaz, M Saifur Rahman, and Md Shamsuzzoha Bayzid. wQFM: highly accurate genome-scale species tree estimation from weighted quartets. *Bioinformatics*, 37(21):3734–3743, November 2021. doi:10.1093/bioinformatics/btab428.

26 Alexey Markin and Oliver Eulenstein. Quartet-based inference is statistically consistent under the unified duplication-loss-coalescence model. *Bioinformatics*, 37(22):4064–4074, 2021.

27 Andy McKenzie and Mike Steel. Distributions of cherries for two models of trees. *Mathematical Biosciences*, 164(1):81–92, March 2000. doi:10.1016/S0025-5564(99)00060-7.

28 Charles D Michener and Robert R Sokal. A quantitative approach to a problem in classification. *Evolution*, 11(2):130–162, 1957.

29 S. Mirarab, R. Reaz, Md. S. Bayzid, T. Zimmermann, M. S. Swenson, and T. Warnow. ASTRAL: genome-scale coalescent-based species tree estimation. *Bioinformatics*, 30(17):i541–i548, September 2014. doi:10.1093/bioinformatics/btu462.

30 Siavash Mirarab. Species Tree Estimation Using ASTRAL: Practical Considerations. *arXiv:1904.03826 [q-bio]*, October 2019. arXiv:1904.03826.

31 Siavash Mirarab, Md. Shamsuzzoha Bayzid, Bastien Boussau, and Tandy Warnow. Statistical binning enables an accurate coalescent-based estimation of the avian tree. *Science*, 346(6215):1250463, December 2014. doi:10.1126/science.1250463.

32 Siavash Mirarab, Md Shamsuzzoha Bayzid, and Tandy Warnow. Evaluating summary methods for multilocus species tree estimation in the presence of incomplete lineage sorting. *Systematic Biology*, 65(3):366–380, 2016.

33 Siavash Mirarab and Tandy Warnow. ASTRAL-II: coalescent-based species tree estimation with many hundreds of taxa and thousands of genes. *Bioinformatics*, 31(12):i44–i52, June 2015. doi:10.1093/bioinformatics/btv234.

34 Erin K Molloy and Tandy Warnow. To Include or Not to Include: The Impact of Gene Filtering on Species Tree Estimation Methods. *Systematic Biology*, 67(2):285–303, March 2018. doi:10.1093/sysbio/syx077.

35 Naima Moshiri. TreeSwift: A massively scalable Python tree package. *SoftwareX*, 11:100436, January 2020. doi:10.1016/j.softx.2020.100436.

36 Huw A. Ogilvie, Remco R. Bouckaert, and Alexei J. Drummond. StarBEAST2 Brings Faster Species Tree Inference and Accurate Estimates of Substitution Rates. *Molecular Biology and Evolution*, 34(8):2101–2114, August 2017. doi:10.1093/molbev/msx126.

37 Swati Patel, Rebecca T Kimball, and Edward L Braun. Error in phylogenetic estimation for bushes in the tree of life. *J. Phylogenet. Evol. Biol*, 1(2):1–10, 2013.

38 Morgan N. Price, Paramvir S. Dehal, and Adam P. Arkin. FastTree 2 – Approximately Maximum-Likelihood Trees for Large Alignments. *PLOS ONE*, 5(3):e9490, March 2010. doi:10.1371/journal.pone.0009490.

39 Maryam Rabiee and Siavash Mirarab. Forcing external constraints on tree inference using astral. *BMC genomics*, 21(2):1–13, 2020.

40 John A. Rhodes, Michael G. Nute, and Tandy Warnow. NJst and ASTRID are not statistically consistent under a random model of missing data, 2020. doi:10.48550/ARXIV.2001.07844.

41 D. F. Robinson and Leslie R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1):131–147, February 1981. doi:10.1016/0025-5564(81)90043-2.

42 Sébastien Roch, Michael Nute, and Tandy Warnow. Long-branch attraction in species tree estimation: inconsistency of partitioned likelihood and topology-based summary methods. *Systematic Biology*, 68(2):281–297, 2019.

43 Sébastien Roch and Mike Steel. Likelihood-based tree reconstruction on a concatenation of aligned sequence data sets can be statistically inconsistent. *Theoretical Population Biology*, 100:56–62, March 2015. doi:10.1016/j.tpb.2014.12.005.

44 Naruya Saitou and Masatoshi Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, July 1987. doi:10.1093/oxfordjournals.molbev.a040454.

45 Erfan Sayyari and Siavash Mirarab. Fast Coalescent-Based Computation of Local Branch Support from Quartet Frequencies. *Molecular Biology and Evolution*, 33(7):1654–1668, July 2016. doi:10.1093/molbev/msw079.

46 Alexandros Stamatakis. RAxML version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics*, 30(9):1312–1313, January 2014. doi:10.1093/bioinformatics/btu033.

47 Naoyuki Takahata. Gene genealogy in three related populations: consistency probability between gene and population trees. *Genetics*, 122(4):957–966, August 1989. doi:10.1093/genetics/122.4.957.

48 Pranjal Vachaspati. *Large scale phylogenomic estimation*. PhD thesis, University of Illinois at Urbana-Champaign, 2019.

49 Pranjal Vachaspati and Tandy Warnow. ASTRID: Accurate Species TRees from Internode Distances. *BMC Genomics*, 16(10):S3, October 2015. doi:10.1186/1471-2164-16-S10-S3.

50 John J Wiens, Caitlin A Kuczynski, Sarah A Smith, Daniel G Mulcahy, Jack W Sites Jr, Ted M Townsend, and Tod W Reeder. Branch lengths, support, and congruence: testing the phylogenomic approach with 20 nuclear loci in snakes. *Systematic Biology*, 57(3):420–431, 2008.

51 James Willson, Mrinmoy Saha Roddur, Baqiao Liu, Paul Zaharias, and Tandy Warnow. DISCO: Species Tree Inference using Multicopy Gene Family Tree Decomposition. *Systematic Biology*, 71(3):610–629, May 2022. doi:10.1093/sysbio/syab070.

52 Zhenxiang Xi, Liang Liu, and Charles C. Davis. Genes with minimal phylogenetic information are problematic for coalescent analyses when gene tree estimation is biased. *Molecular Phylogenetics and Evolution*, 92:63–71, November 2015. doi:10.1016/j.ympev.2015.06.009.

53 George Udny Yule. A mathematical theory of evolution, based on the conclusions of Dr. J. C. Willis. *Philosophical Transactions of the Royal Society of London. Series B, Containing Papers of a Biological Character*, 213(402-410):21–87, January 1925. doi:10.1098/rstb.1925.0002.

54 Chao Zhang and Siavash Mirarab. Weighting by gene tree uncertainty improves accuracy of quartet-based species trees. *bioRxiv*, 2022. doi:10.1101/2022.02.19.481132.

55 Chao Zhang, Maryam Rabiee, Erfan Sayyari, and Siavash Mirarab. ASTRAL-III: polynomial time species tree reconstruction from partially resolved gene trees. *BMC Bioinformatics*, 19(6):153, May 2018. doi:10.1186/s12859-018-2129-y.

A Commands

A.1 Weighted ASTRID

The commands differ depending on the type of support annotated on the gene trees. For FastTree SH-like, bootstrap support, and aBayes support, the commands for wASTRID-s are respectively:

```
wastrid -i $genes -o $output # FastTree SH-like
wastrid -x 100 -i $genes -o $output # bootstrap
wastrid -n 0.333 -i $genes -o $output # aBayes
```

For the final setting of wASTRID-pl (distance defined by path lengths, with branch lengths normalized by the maximum path-length in the tree), the above commands need to be appended with an additional flag: `-m n-length` (distance using "normalized branch length" instead of the default `-m support`).

A.2 Weighted ASTRAL

We ran hybrid-weighted ASTRAL (v1.4.2.3) using the following command on trees annotated with aBayes support:

```
astral-hybrid -x 1 -n 0.333 -i $genes -o $output
```

A.3 ASTRID

We ran ASTRID (v2.2.1) using the following command:

```
ASTRID -s -i $genes -o $output
```

The `-s` flag is specified to skip the missing data imputation step of ASTRID, as all tested data have no missing data in the averaged internode distance matrix (i.e., each pair of taxa appears together at least once in some gene tree).

A.4 ASTRAL

We ran ASTRAL (v5.7.8) using the following command:

```
java -jar astral.5.7.8.jar -i $genes -o $output
```

A.5 Approximate Bayesian support

We computed IQ-TREE (v2.1.2) aBayes support using the following command:

```
iqtree2 -s $aln -te $gtree -m GTR+G -abayes
```

where `$aln` is the alignment file for the gene tree, and `$gtree` is the path to the gene tree topology.

A.6 Bootstrap support

We computed bootstrap support (on the training dataset) using FastTree (v2.1.11) on bootstrap replicates generated by Goalign (v0.3.5) [19].

The following command was used to generate bootstrap replicates (`$aln` is the original gene alignment):

```
goalign build seqboot -i $aln -S -n 100
```

The FastTree command used for inferring a tree from one bootstrap replicate is (`$bs_aln` is one bootstrap alignment replicate generated by Goalign):

```
FastTree -nt -gtr -nosupport $bs_aln > $output
```

The FastTree bootstrap trees are then randomly resolved to eliminate polytomies, and then mapped by RAxML-NG [15] to the original gene trees using the following command:

```
raxml-ng --support --tree $gtree --bs-trees $bs_trees --bs-metric fbp
```

where `$gtree` is the path to the original gene tree, and `$bs_trees` contains new-line separated newick trees inferred on the bootstrap replicates.

A.7 Failures to complete

ASTRAL failed to complete 24/50 replicates (replicates 2, 4, 6, 8, 9, 11, 13, 15, 16, 17, 20, 23, 27, 28, 29, 30, 32, 36, 38, 39, 40, 41, 48, and 49) on the MC11H condition ($n = 1001$) under $k = 1000$. The 24 log files indicate that ASTRAL has indeed timed out on each of the replicates. An example of the last three lines of the log files is attached:

```
Calculated 700000 weights; time (seconds): 1549
Calculated 800000 weights; time (seconds): 1544
slurmstepd: error: *** JOB 5328647 ON golub041 CANCELLED AT 2022-05-08T05
    :24:48 DUE TO TIME LIMIT ***
```

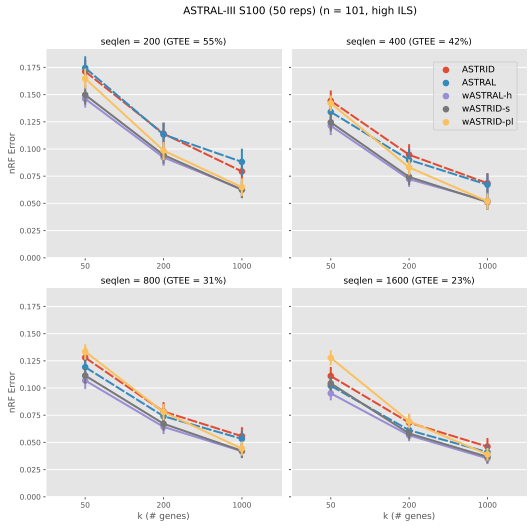


Figure 8 Topological error of species tree across methods on the ASTRAL-III S100 dataset ($n = 101$, AD $= 46$). Results are shown averaged across 50 replicates, with error bars showing the standard error. All weighted methods (wASTRID-s, wASTRID-pl, wASTRAL) ran on gene trees reannotated with IQ-TREE aBayes support branch support and lengths.

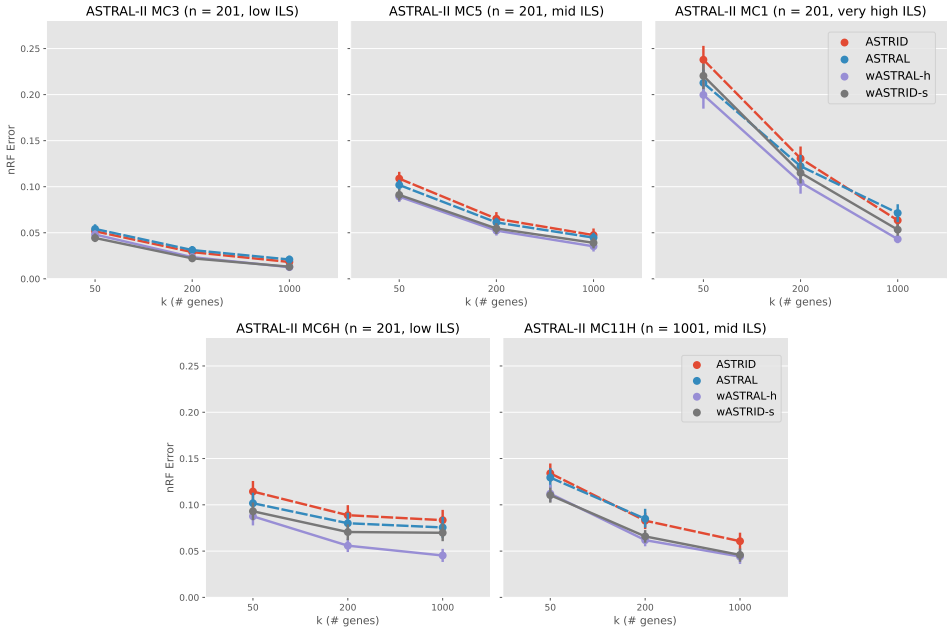


Figure 9 Topological error (nRF error rate) of species tree across methods on selected conditions on the ASTRAL-II SimPhy conditions. Results are shown averaged across 50 replicates with standard error bars. ASTRAL did not finish 24 out of the 50 replicates within four hours for $k = 1000$ on MC11H and thus the data point was omitted. All weighted methods (wASTRID, wASTRAL) were run on gene trees reannotated with IQ-TREE aBayes support branch support and lengths.

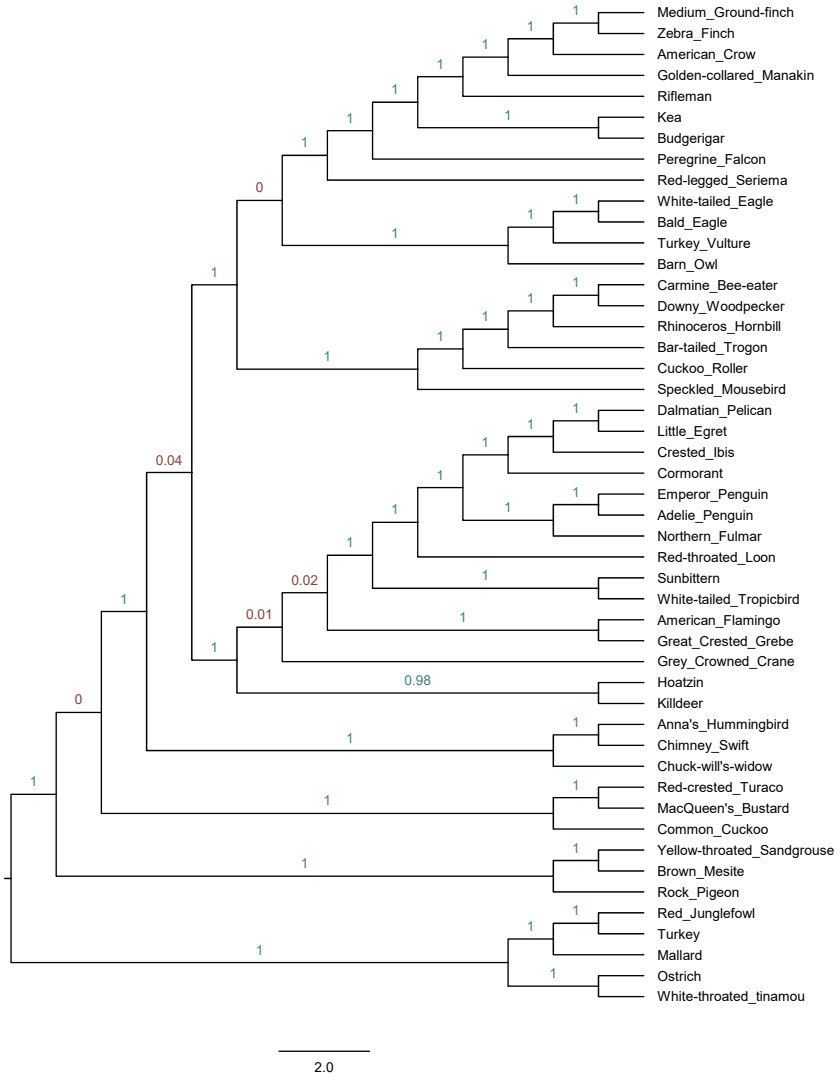


Figure 10 Reconstructed species tree on the Jarvis et al. avian biological data using ASTRID. Branch support values are in localPP values calculated by wASTRAL-h.

On Weighted k-mer Dictionaries

Giulio Ermanno Pibiri
Ca' Foscari University of Venice, Venice, Italy
ISTI-CNR, Pisa, Italy

Abstract

We consider the problem of representing a set of k-mers and their abundance counts, or weights, in compressed space so that assessing membership and retrieving the weight of a k-mer is efficient. The representation is called a *weighted dictionary* of k-mers and finds application in numerous tasks in Bioinformatics that usually count k-mers as a pre-processing step. In fact, k-mer counting tools produce very large outputs that may result in a severe bottleneck for subsequent processing.

In this work we extend the recently introduced SSHash dictionary (Pibiri, *Bioinformatics* 2022) to also store compactly the weights of the k-mers. From a technical perspective, we exploit the order of the k-mers represented in SSHash to encode *runs* of weights, hence allowing (several times) better compression than the empirical entropy of the weights. We also study the problem of reducing the number of runs in the weights to improve compression even further and illustrate a lower bound for this problem. We propose an efficient, greedy, algorithm to reduce the number of runs and show empirically that it performs well, i.e., very similarly to the lower bound. Lastly, we corroborate our findings with experiments on real-world datasets and comparison with competitive alternatives. Up to date, SSHash is the only k-mer dictionary that is exact, weighted, associative, fast, and small.

2012 ACM Subject Classification Applied computing → Bioinformatics

Keywords and phrases K-Mers, Weights, Compression, Hashing

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.9

Supplementary Material *Software (Source Code)*: `https://github.com/jermp/sshash`

Funding This work was partially supported by the projects: MobiDataLab (EU H2020 RIA, grant agreement N°101006879) and OK-INSAID (MIUR-PON 2018, grant agreement N°ARS01_00917).

1 Introduction

Recent advancements in the so-called Next Generation Sequencing (NGS) technology made possible the availability of very large collections of DNA. However, before being able to actually analyze the data at this scale, efficient methods are required to index and search such collections. One popular strategy to address this challenge is to consider short substrings of fixed length k, known as k-mers. Software tools based on k-mers are predominant in Bioinformatics and they have found applications in genome assembly [3, 13], variant calling [15, 41], pan-genome analysis [2, 19], meta-genomics [43], sequence comparison [35, 37, 38], just to name a few but noticeable ones.

For several such applications it is important to quantify how many times a given k-mer is present in a DNA database. In fact, many efficient k-mer counting tools have been developed for this task [8, 23, 17, 34]. The output of these tools is a table where each distinct k-mer in the database is associated to its abundance count, or *weight*. The weights are either exact or approximate. (In this work, we focus on exact weights.) These genomic tables are usually very large and take several GBs – in the range of 40-80 bits/k-mer or more according to recent experiments [12, 22, 23]. Therefore, the tables should be compressed effectively while permitting efficient random access queries in order to be useful for on-line processing tasks. This is precisely the goal of this work. We better formalize the problem as follows.

Let $\mathcal{K}$ be the set of the n distinct k-mers extracted from a given DNA string. In particular, $\mathcal{K}$ can be regarded as a set of n pairs $\langle g, w(g)\rangle$, where g is a k-mer and $w(g)$ is the *weight* of g. Our objective is to build a compressed, weighted, dictionary for $\mathcal{K}$, i.e., a data structure representing the k-mers *and* weights of $\mathcal{K}$ in compressed space such that it is efficient to check the *exact* membership of g to $\mathcal{K}$ and, if g actually belongs to $\mathcal{K}$, retrieve $w(g)$.

In our previous investigation on the problem, we proposed a *sparse and skew hashing* scheme for k-mers (SSHash, henceforth) [25] – a compressed dictionary that relies on k-mer *minimizers* [35] and *minimal perfect hashing* [26, 27] to support fast membership (in both random and streaming query modality) in succinct space. However, we did not consider the weights of the k-mers. In this work, therefore, we enrich the SSHash data structure with the weight information. The main practical result is that, by exploiting the *order* of the k-mers represented in SSHash, the compressed exact weights take only a small extra space on top of the space of SSHash. This extra space is proportional to the number of *runs* (maximal sub-sequences formed by all equal symbols) in the weights and not proportional to the number of distinct k-mers. As a consequence, the weights are represented in a much smaller space than the empirical entropy lower bound.

We also study the problem of reducing the number of runs in the weights and model it as a *graph covering* problem, for which we give an amortized linear-time algorithm. The optimization algorithm effectively reduces the number of runs, hence improving space even further, and almost matches the best possible reduction according to a lower bound.

When empirically compared to other weighted dictionaries that can be either somewhat smaller but much slower or much larger, SSHash embodies a robust trade-off between index space and query efficiency.

2 Related Work

A solution to the weighted k-mer dictionary problem can be obtained using the popular FM-index [11]. The FM-index represents the original DNA string taking the Burrows-Wheeler transform (BWT) [5] of the string. Reporting the weight of a k-mer is solved using the *count* operation of the FM-index which involves $O(k)$ rank queries over the BWT.

Another solution using the BWT is the so-called BOSS data structure [4] that is a succinct representation of the *de Bruijn* graph of the input – a graph where the nodes are the k-mers and the edges model the overlaps between the k-mers. The BOSS data structure has been recently enriched with the weights of the k-mers [12], by delta-encoding the weights on a spanning branching of the graph. Since consecutive k-mers often have equal (or very similar) weights, good space effectiveness is achieved by this technique.

Other solutions, instead, rely on hashing for faster query evaluation compared to BWT-based indexes. For example, both deBGR [22] and Squeakr [23] use a *counting quotient filter* [21] to store the k-mers and the weights. They can either return approximate weights, i.e., wrong answers with a prescribed (low) probability, for better space usage of exact weights at the price of more index space. In any case, the memory consumption of these solutions is not competitive with that of BWT-based ones as they do not employ sophisticated compression techniques and were designed for other purposes, e.g., dynamic updates.

A closely related problem is that of realizing *maps* from k-mers to weights, i.e., data structures that do *not* explicitly represent the k-mers and so return arbitrary answers for out-of-set keys. In the context of this work, we distinguish between such approaches, *maps*, and dictionaries that instead represent *both* the k-mers and the weights. Besides minimal perfect hashing [26, 27], some efficient maps have been proposed and tailored specifically

for genomic counts, such as based on *set-min sketches* [39] and *compressed static functions* (CSFs) [40]. These proposals leverage on the repetitiveness of the weights (low-entropy distributions) to obtain very compact space.

Lastly in this section, we report that other works [18, 14] considered the multi-document version of the problem studied here, that is, how to retrieve a vector of weights for a query k-mer, where each component of the vector represents the weight of the k-mer in a distinct document. Also such count vectors are usually very "regular" (or can be made so by introducing some approximation) [18] and present runs of equal symbols that can be compressed effectively with *run-length encoding* (RLE).

3 Representing Runs of Weights

In this section we describe the compression scheme for the weights that we use in SSHash. Recall that we indicate with $\mathcal{K}$ the set of n distinct $\langle k\text{-mer, weight}\rangle = \langle g, w(g)\rangle$ pairs, that we want to store in a dictionary. With a little abuse of notation, we write "$g \in \mathcal{K}$" for a k-mer g to mean that there is a pair of $\mathcal{K}$ whose k-mer is g. We first highlight the main properties of SSHash that we are going to exploit in the following to obtain good space effectiveness for the weights. (For all the other details concerning the SSHash index, we point the interested reader to our previous work [25].)

From a high-level perspective, SSHash implements the function $h : \Sigma^k \to \{0, 1, \ldots, n\}$, where $n = |\mathcal{K}|$ and Σ^k is the whole set of k-length strings over the DNA alphabet $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$. In particular, $h(g)$ is a unique value $1 \leq i \leq n$ if $g \in \mathcal{K}$; or $h(g) = 0$ otherwise, i.e., if $g \notin \mathcal{K}$. In other words, SSHash serves the same purpose of a minimal and perfect hash function (MPHF) [27] for $\mathcal{K}$ but, unlike a traditional MPHF, SSHash *rejects* alien k-mers. This is possible because the k-mers of $\mathcal{K}$ are actually represented in SSHash whereas the space of a traditional MPHF does not depend on the input keys.

The value $i = h(g)$ for the k-mer $g \in \mathcal{K}$ is the handle of g, or its "hash" code. The hash codes can be used to associate some satellite information to the k-mers such as, for example, the weights themselves using an array $W[1..n]$ where $W[h(g)] = w(g)$. The crucial property of SSHash in which we are interested is that the function h *preserves the relative order* of the k-mers, that is: if $g_1[1..k]$ and $g_2[1..k]$ are two k-mers with $g_1[2..k] = g_2[1..k-1]$ (i.e., g_2 comes immediately after g_1 in a string), then $h(g_2) = h(g_1) + 1$. Therefore, consecutive k-mers, i.e., those sharing an overlap of $k - 1$ symbols, are also given consecutive hash codes. This is achieved in SSHash by pre-processing the input set $\mathcal{K}$ into a so-called *spectrum-preserving string set* (or SPSS) $\mathcal{S}$ – a collection of strings $\mathcal{S} = \{S_1, \ldots, S_m\}$ where each k-mer of $\mathcal{K}$ appears exactly once. We omit the details here on how the collection $\mathcal{S}$ can be built; we only report that there are efficient algorithms for this purpose that also try to minimize the total number of symbols in $\mathcal{S}$, i.e., the quantity $\sum_{i=1}^{m} |S_i|$. One such algorithm is the UST algorithm [33] that we also use to prepare the input for SSHash.

Therefore, once an order $S_1, \ldots, S_m$ for the strings of $\mathcal{S}$ is fixed, then also an order $i = 1, \ldots, n$ for the k-mers g_i is uniquely determined. Let $W[1..n]$ be the sequence of weights in this order. Then, we have: $h(g_i) = i$ and $W[i] = w(g_i)$, for $i = 1, \ldots, n$.

This order-preserving behavior of h induces a property on the sequence of weights $W[1..n]$ that significantly aids compression: W contains *runs*, i.e., maximal sub-sequences of *equal weights*. This is so because consecutive k-mers are very likely to have the same weight due to the high specificity of the strings. This a known fact, also observed in prior work [12, 18, 40]. Here, we are exploiting the order of the k-mers given by SSHash to preserve the natural order of the weights in W. Note that this cannot be achieved by approximate schemes that

```
>5 5 5 5 5 5 5 5 5 5 5 5 5 5
GGTAATGCAGCCAGGGATGCAACGACCGCAACAGAAAAAGCCCG

>4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 4 4 4 4
CAGCTCATTACAGAAAAAATACCGCTCACCGCCCTGCACCGTCAGGTCAATTTCCCTGAGCACCACCCGCGGTGACTGCTCTGATTTAACC

>4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
CAGCTATGCAGGAGACAAGAATCGCCAGCTTACCCGTTACAGCGATACCCGCTGGCATG

>13 13 13 13 13 13 13
TCAGGTGTACGGTGTGCGTAAAGTCTGGCGTCAGTTG
```

Figure 1 An example collection $\mathcal{S}$ of 4 weighted sequences (for $k = 31$) drawn from the genome of *E. Coli* (Sakai strain). With alternating colors we render the change of weight in the runs. There are 111 k-mers in the example but just 6 runs in the weights: $RLW = \langle 5, 14\rangle\langle 4, 18\rangle\langle 2, 8\rangle\langle 1, 31\rangle\langle 4, 33\rangle\langle 13, 7\rangle$. Note that a run can cross the boundary between two (or more) sequences, as it happens for the run $\langle 4, 18\rangle$ which covers completely the third but also the part of the second sequence.

do not represent the k-mers themselves, like a generic MPHF or a CSF. Even if the k-mers were available, those schemes are unable to assign consecutive hashes to consecutive k-mers, actually shuffling the weights at random and, thus, making W very difficult to compress.

It is standard to represent a sequence W featuring r runs of equal symbols using *run-length encoding* (RLE), i.e., W is modeled as a sequence of run-length pairs $RLW = \langle w_1, \ell_1\rangle\langle w_2, \ell_2\rangle \cdots \langle w_r, \ell_r\rangle$ where w_i and ℓ_i are, respectively, the value of the run and the length of the i-th run in W. Figure 1 shows an example of RLW for a collection $\mathcal{S}$ with 4 weighted strings.

Encoding RLW

Let $\mathcal{D}$ be the set of all distinct w_i in RLW. Clearly, $r \geq |\mathcal{D}|$ as we must have at least one run per distinct weight. We store $\mathcal{D}$ using $|\mathcal{D}|\lceil\log_2 max\rceil$ bits where $max \geq 1$ is the largest w_i. We use $\mathcal{D}$ to uniquely represent each w_i in RLW with $\lceil\log_2|\mathcal{D}|\rceil$ bits. Since runs are maximal sub-sequences in W by definition, then $w_i \neq w_{i+1}$ for every $i = 1, \ldots, r-1$ (adjacent weights must be different). Then we take the prefix-sums of the sequence $0, \ell_1, \ldots, \ell_{r-1}$ into an array $L[1..r]$ and encode it with Elias-Fano [9, 10][1]. By construction we have that $\sum_{i=1}^{r} \ell_i = n$ since the runs must cover the whole set of k-mers. So the largest element in L is actually $n - \ell_r$ and we spend at most $r\lceil\log_2(n/r)\rceil + 2r + o(r)$ bits for L. Summing up, we spend at most

$$r \cdot \Big(\lceil\log_2|\mathcal{D}|\rceil + \Big\lceil \log_2\Big(\frac{n}{r}\Big)\Big\rceil + 2 + o(1)\Big) + |\mathcal{D}|\lceil\log_2 max\rceil \text{ bits}$$

for representing RLW on top of the space of SSHash. In conclusion, the weights are represented in space proportional to the number of runs in W (i.e., $r = |RLW|$) and *not* proportional to the number of k-mers, which is n. As a consequence, this space is likely to be considerably less than the empirical entropy $H_0(W)$ as we are going to see with the experiments in Section 5.

[1] Elias-Fano represents a monotone integer sequence $S[1..n]$ with $S[n] \leq U$ in at most $n\lceil\log_2(U/n)\rceil + 2n$ bits. With $o(n)$ extra bits it is possible to decode any $S[i]$ in constant time and support predecessor queries in $O(\log(U/n))$ time. For a complete description of the method, we point the reader to the survey by Pibiri and Venturini [32, Section 3.4]. We also remark that Elias-Fano has been recently used in many compressed, practical, data structures, e.g., inverted indexes [20, 28, 31, 32, 42], tries [24, 29, 30], and full-text indexes [16].

To retrieve the weight $w(g)$ from $i = h(g)$, all that is required is to identify the run containing i. This operation is done in $O(\log(n/r))$ time with a predecessor query over L given that we represent L with Elias-Fano. If the identified run is the j-th run in W, then w_j is retrieved in $O(1)$ from $\mathcal{D}$.

4 Reducing the Number of Runs

In Section 3 we presented an encoding scheme for the k-mer weights whose space is proportional to the *number of runs* in the sequence of weights W. Therefore, in this section we consider the problem of reducing the number of runs in the weights to optimize the space of the encoding.

Rules of the Game

We assume that the strings in $\mathcal{S}$ are *atomic* entities: it is not allowed to partition them into sub-strings (e.g., in correspondance of the runs of weights in the strings). In fact, since the strings are obtained by the UST algorithm [33] with the purpose of *minimizing* the number of nucleotides as we explained in Section 3, breaking them will lead to an increased space usage for the k-mers, actually dwarfing any space-saving effort spent for the weights. With this constraint specified, there are only *two* degrees of freedom that can be exploited to obtain better compression for W: (1) the *order* of the strings, and (2) the *orientation* of the strings. Altering $\mathcal{S}$ using these two degrees of freedom does not affect the correctness nor the (relative) order-preserving property of the function $h : \Sigma^k \to \{0, 1, \ldots, n\}$ implemented by SSHash. In fact, as evident from our description in Section 3, the output of h will still be $\{1, \ldots, n\}$ as the k-mers themselves do *not* change (even when taking reverse-complements into account as they are considered to be identical). What changes is just the absolute order of the k-mers as a consequence of permuting the order of the strings $\{S_1, \ldots, S_m\}$ in $\mathcal{S}$.

Therefore, our goal is to permute the order of the strings in $\mathcal{S}$ and possibly change their orientations to reduce the number of runs in W. We now consider an illustrative example to motivate why both these two operations – those of changing the order and orientation of a string – are important to reduce the number of runs. Refer to Figure 2a which shows an example collection of $m = |\mathcal{S}| = 4$ weighted strings (for $k = 3$). Applying the permutation $\pi = [1, 4, 2, 3]$ as shown in Figure 2b reduces the number of runs by 1 because the run at the junction of string 4 and 2 can be glued. Lastly, applying the *signed* permutation $\pi = [+1, +4, -2, +3]$ as in Figure 2c reduces the number of runs by 3, which is the best possible. Our objective is to compute such a signed permutation π for an input collection of strings, in order to permute $\mathcal{S}$ as shown in Algorithm 1.

Figure 2 also suggests that the final result π solely depends on the weight of the first and last k-mer of each sequence – which we call the *end-point* weights (or just end-points) of a sequence – and not on the other weights nor the nucleotide sequences. Therefore, it is useful to model an input collection $\mathcal{S}$ using a graph, defined as follows.

▶ **Definition 1** (End-point Weight Graph). Given a collection of weighted sequences $\mathcal{S}$, let G be a graph where:

- There is a node u for each sequence of $\mathcal{S}$ and u has two sides – a *left* and a *right* side – respectively labelled with the end-point weights of the sequence.
- There is an edge between any two distinct nodes u and v that have a side with the same weight.

The graph G is called the *end-point weight graph* for $\mathcal{S}$ and indicated with $ewG(\mathcal{S})$.

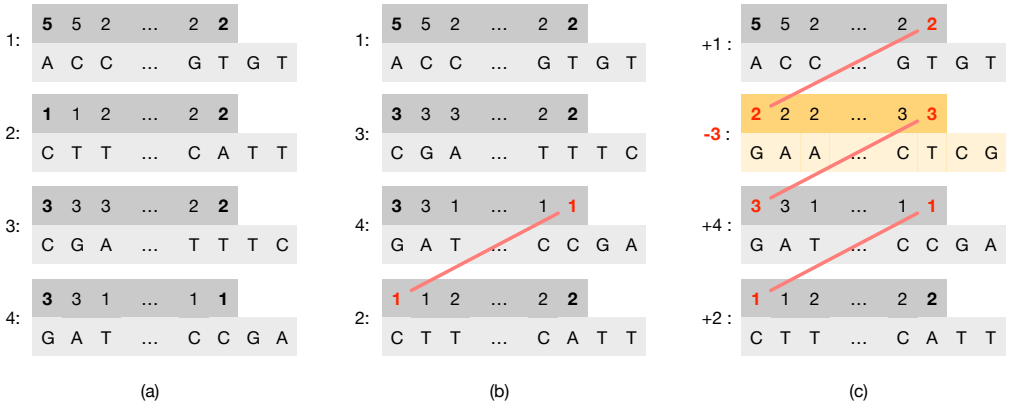


Figure 2 In (a), an example input collection $\mathcal{S}$ of $m = |\mathcal{S}| = 4$ weighted strings (for $k = 3$), where the end-point weights are highlighted in bold font. In (b), the order of the strings is changed according to the permutation $\pi = [1, 4, 2, 3]$ and, as a result, the number of runs is reduced by 1 (the last run in string 4 is glued with the first run of string 2). Lastly, in (c), it is shown that changing the orientation of string 3 (taking the reverse complement of the string and reversing the order of the k-mer weights) makes it possible to glue other two runs. Given that reducing the number of runs by $m - 1$ is the best achievable reduction, the number of runs in (c) is therefore the minimum for the original collection in (a).

Algorithm 1 The algorithm permute takes as input a collection $\mathcal{S} = \{S_1, \ldots, S_m\}$ of weighted strings and a signed permutation π and returns the permuted collection $\tilde{\mathcal{S}} = \pi(\mathcal{S})$. The function reverse takes the reverse-complement of a string and reverse its weights.

```
1 permute(S, π):
2     let S̃ = {S̃_1, ..., S̃_m} be a new collection of empty strings
3     for i = 1; i ≤ m; i = i + 1 :
4         j = π[i]
5         if j < 0 : S̃_{-j} = reverse(S_i)
6         else : S̃_j = S_i
7     return S̃
```

In the following, we indicate a node u in $ewG(\mathcal{S})$ using the identifier (id) of the corresponding sequence of $\mathcal{S}$. Also, we associate to u a $sign \in \{-1, +1\}$ (or orientation), indicating whether the sequence should be reverse-complemented. In summary, a node u in $ewG(\mathcal{S})$ is the 4-tuple $(id, left, right, sign)$.

▶ **Definition 2** (Oriented Path). An *oriented* path in $ewG(\mathcal{S})$ is either a single node (singleton path) or a sequence of nodes $(u_1 \to \cdots \to u_\ell)$ where each consecutive pair of nodes $u_i \to u_{i+1}$ is oriented in such a way that $u_i.right = u_{i+1}.left$, for any $1 \le i < \ell$.

Since we will be interested only in oriented paths, we just refer to them as "paths". For ease of notation, we will indicate a path in our examples as a sequence of signed numbers $(i_1 \to \cdots \to i_\ell)$ where each number represents a node's id and its sign represents the node's $sign$. The first and the last node in the path are called, respectively, the *front* and the *back* of the path. The weights $front.left$ and $back.right$ are the two end-points of the path.

Given this graph model, it follows that the problem of finding a signed permutation π for $\mathcal{S}$ is equivalent to that of computing a (disjoint-node) *path cover* C for $ewG(\mathcal{S})$, i.e., a set of paths in $ewG(\mathcal{S})$ that visit *all* the nodes and where each node belongs to *exactly one* path.

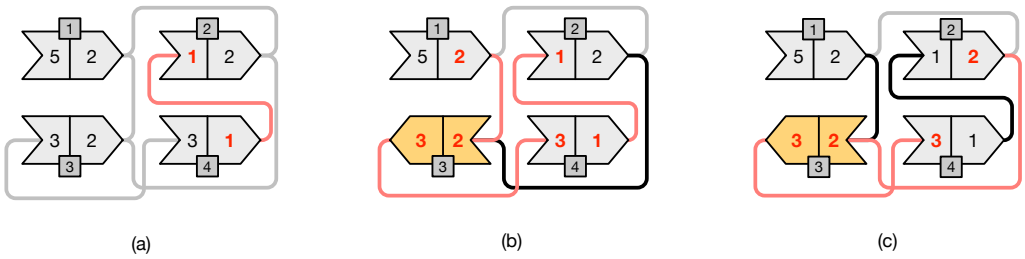


Figure 3 The same example of Figure 2 but modeled using end-point weight graphs. Each node is represented using an arrow-like shape with two-matching sides. Only opposite sides having the same weight can be matched. The numbers inside the shapes represent the end-point weights; the extra darker square contains the node identifier. An arrow oriented from *left-to-right* models a node with *positive* sign; vice versa, an arrow oriented from *right-to-left* models a node with *negative* sign. Gray edges represent edges that *cannot* be traversed without changing the orientation of one of the two connected nodes. Black edges represent edges that can be traversed. Lastly, we highlight in red the edges that belong to paths in a graph cover. The example in (a) corresponds to that of Figure 2b where no node has changed orientation and, therefore, we have three paths in the cover: $(+4 \rightarrow +2)$, $(+3)$, and $(+1)$. Other two different covers are shown in (b) and (c). In (b) the cover contains the single path $(+1 \rightarrow -3 \rightarrow +4 \rightarrow +2)$ and corresponds to the example of Figure 2c where the node 3 was changed orientation from $+$ to $-$ (shown in yellow color). In (c) the cover contains the two paths $(+2 \rightarrow -3 \rightarrow +4)$ and the singleton path $(+1)$.

In fact note that, given a cover C for $ewG(\mathcal{S})$, there is a linear-time reduction from C to π as illustrated in Algorithm 2. Since the cover C is a disjoint-node path cover, the correctness of the algorithm is immediate as well as its complexity of $\Theta(m)$.

Algorithm 2 The algorithm reduce takes as input a path cover C computed for $ewG(\mathcal{S})$ and returns the corresponding signed permutation π. The complexity of the algorithm is $\Theta(m)$ since the number of nodes in $ewG(\mathcal{S})$ is m and each node appears exactly once in C.

```
1 reduce(C):
2     j = 1
3     let π[1..m] be a new array
4     for each path p ∈ C :
5         for each node u ∈ p :
6             π[u.id] = u.sign · j
7             j = j + 1
8     return π
```

Figure 3 illustrates the same example of Figure 2 but with end-point weight graphs. In Figure 3b we would obtain a cover $C = \{(+1 \rightarrow -3 \rightarrow +4 \rightarrow +2)\}$ formed by a single path. In this case the permutation π, following Algorithm 2, would be $\pi[1] = +1$, $\pi[3] = -2$, $\pi[4] = +3$, and $\pi[2] = +4$. This is indeed the same permutation discussed in Figure 2c. Another example: for the graph in Figure 3c, the cover would be $C = \{(+2 \rightarrow -3 \rightarrow +4), (+1)\}$ and the permutation π would be $\pi[2] = +1$, $\pi[3] = -2$, $\pi[4] = +3$, and $\pi[1] = +4$.

4.1 A Lower Bound to the Number of Runs

We showed that changing the order and orientation of the strings in $\mathcal{S}$ can reduce the number of runs in the weights. The crucial question is: by how much? We are interested in deriving a lower bound to the number of runs achievable after applying the signed permutation π to $\mathcal{S}$. Since we modeled the problem of computing π as the problem of finding a path cover C for $ewG(\mathcal{S})$, we reason in terms of $ewG(\mathcal{S})$ and C.

Let $|C|$ be the number of paths in the cover C. Let r_i be the number of runs in S_i and let R be the total number of runs, i.e., $R = \sum_{i=1}^{m} r_i$. Then there are at least $R - m$ runs in $\mathcal{S}$ *regardless* the order of the sequences. Therefore, a straightforward lower bound to the number of runs would be $\max\{|\mathcal{D}|, R - m + 1\}$. This lower bound assumes (very optimistically) that we are able to obtain a cover with a single path, i.e., $|C| = 1$, hence reducing the total number of runs R by $m - 1$ which is the best reduction achievable with m sequences. Note, however, that the bound cannot be lower than $|\mathcal{D}|$ – the number of distinct weights in the input – because, clearly, there must be at least one run per distinct weight value. We would like to improve the bound $\max\{|\mathcal{D}|, R - m + 1\}$ knowing that, in general, we could not be able to form one single path.

We observe that the final number of runs r in the permuted $\mathcal{S}$ will be equal to $R - m + |C|$. In fact, every path in C must begin (resp. end) with a node whose left side (resp. right side) cannot be glued with any other path's side. Therefore, a new run begins with the first node of every path. Since we wish to minimize the quantity $R - m + |C|$, and considering that $R - m$ is constant for a given $\mathcal{S}$, it follows that the problem reduces to that of minimizing $|C|$, the number of paths in the cover. In other words, the problem of minimizing the number of runs r is equivalent to that of finding a minimum-cardinality path cover C for $ewG(\mathcal{S})$.

Therefore our strategy is to give a lower bound to the number of paths $|C|$. To do so, we compute the number of end-point weights, say n_e, that must appear as end-points of the paths (left side of the front node of a path, or right side of the back node of a path). Since a path has exactly two end-points, then it follows that $|C| \geq \lceil n_e/2 \rceil$ and, in turn, that the final number of runs r in $\pi(\mathcal{S})$ is *at least* $R - m + \lceil n_e/2 \rceil \geq \max\{|\mathcal{D}|, R - m + 1\}$.

Since we now focus on the end-point weights of the nodes, in this section we will denote a node $(id, left, right, sign)$ just by its weights $(left, right)$. We start with a preliminary Lemma and a Definition. (See Appendix A for all the proofs omitted from the section.)

▶ **Lemma 3.** Let us consider $d > 0$ equal nodes (w, x). If d is even, then the d nodes can be oriented to form a maximal path of either end-points (w, w) or (x, x). If d is odd, then the path has end-points (w, x).

▶ **Definition 4** (Incidence Set)**.** Given the weight w, a set I_w of nodes where w appears as end-point is called an *incidence set* for w. Let $n(I_w)$ be the number of times w appears in the nodes of I_w. Note that $n(I_w) \geq |I_w|$ because there could be nodes (w, w) in I_w.

▶ **Example 5.** The sets

$$I_w^1 = \{(w,x),(w,y),(w,z),(w,t),(w,l)\}$$
$$I_w^2 = \{(w,w),(w,x),(w,y),(w,z)\}$$
$$I_w^3 = \{(w,w),(w,w),(w,w),(x,w),(w,x)\}$$
$$I_w^4 = \{(w,x),(w,x),(y,w),(w,y),(w,y),(w,z),(w,z),(w,z),(w,z)\}$$

are four different incidence sets for w with $n(I_w^1) = n(I_w^2) = 5$, $n(I_w^3) = 8$, and $n(I_w^4) = 9$. The set $\{(w, x), (x, y)\}$, instead, is not an incidence set for w because the node (x, y) does not have w as an end-point.

Next, we give the following central Lemma that will help in counting the number of weights that must appear as end-points of the paths in C.

▶ **Lemma 6.** Given an incidence set I_w, if $n(I_w)$ is *odd* then only one path will contain w as end-point among all the maximal paths that can be created from the nodes in I_w.

▶ **Example 7.** Let us consider an example for Lemma 6. The sets I_w^1 and I_w^2 in Example 5 are both canonical since all other weights different from w are distinct, except for one single node (w, w) in I_w^2. It is then easy to see that no matter what paths are created, there will always be one extra node that will remain alone. The set I_w^4 in Example 5, with $n(I_w^4) = 9$, is not canonical instead and we have: $D_x = \{(w, x), (w, x)\}$, $D_y = \{(y, w), (w, y), (w, y)\}$, and $D_z = \{(w, z), (w, z), (w, z), (w, z)\}$, with $d_x = |D_x| = 2$, $d_y = |D_y| = 3$, and $d_z = |D_z| = 4$. Since $d_x = 2$, then the nodes in D_x can be collapsed into either (w, w) or (x, x). Since $d_y = 3$, then the nodes in D_y can be collapsed to (w, y). Lastly, since $d_z = 4$, the nodes in D_z can be collapsed to either (w, w) or (z, z). Again, regardless the choices done, I_w^4 can always be reduced to a canonical set.

Let now eW be the set of the distinct end-point weights of $\mathcal{S}$. For every weight $w \in eW$, let I_w^{max} be the incidence set of *maximum-cardinality*, i.e., the one obtained by considering *all* the nodes in $ewG(\mathcal{S})$. We partition eW into three disjoint sets, eW_{odd}, eW_{even}, and eW_{equal}. Based on the properties of I_w^{max}, w belongs to one of the three sets as follows.

- If $n(I_w^{max})$ is odd, then $w \in eW_{odd}$. In this case, we are sure by Lemma 6 that w will appear as end-point of some path in the cover.
- If all the nodes in I_w^{max} are equal to (w, w), then $w \in eW_{equal}$. In this case $n(I_w^{max})$ will always be even and, for Lemma 6, w will appear *twice* as end-points of the same path in the cover.
- If $n(I_w^{max})$ is even but the nodes in I_w^{max} are *not* all equal to (w, w), then $w \in eW_{even}$. In this case, we cannot be sure that w will *not* appear as end-point. If $n(I_w^{max})$ is even *and* I_w^{max} contains *all distinct* nodes, then w will not appear. But if I_w^{max} contains two identical nodes, say of the form (w, x), then w may or may not appear. In fact, as already noted, depending on whether a path is created as $(w, x) \to (x, w)$ or $(x, w) \to (w, x)$, w will appear (in the first case) or not (in the second case). The set I_w^3 from Example 5 illustrates this case.

▶ **Lemma 8.** $|eW_{odd}|$ is even.

Therefore, we obtain the following Corollary.

▶ **Corollary 9.** The number of paths in C is at least $(2|eW_{equal}| + |eW_{odd}|)/2$.

Proof. Immediate by first applying Lemma 6 to the maximal incidence sets, then noting that each each path has exactly two end-points, and lastly taking into account the cardinality of eW_{odd} for Lemma 8. ◀

Let us now compute the lower bound of Corollary 9 on some example graphs.

▶ **Example 10.** First, we re-consider the example graph from Figure 2. In that example we have a set of end-point weights $eW = \{1, 2, 3, 5\}$, where $I_1^{max} = \{(1, 2), (3, 1)\}$, $I_2^{max} = \{(5, 2), (1, 2), (3, 2)\}$, $I_3^{max} = \{(3, 2), (3, 1)\}$, and $I_5^{max} = \{(5, 2)\}$. Since $n(I_2^{max}) = 3$ and $n(I_5^{max}) = 1$, then $eW_{odd} = \{2, 5\}$ and $eW_{equal} = \varnothing$. Therefore we sure that both weights 2 and 5 will appear as end-point weights of some paths in the cover. For Corollary 9, any path cover will contain at least $(2|eW_{equal}| + |eW_{odd}|)/2 = (2 \cdot 0 + 2)/2 = 1$ path. Indeed the cover shown in Figure 2b contains one single path and so, it is optimal, whereas the cover in Figure 2c contains 2 paths and is not optimal.

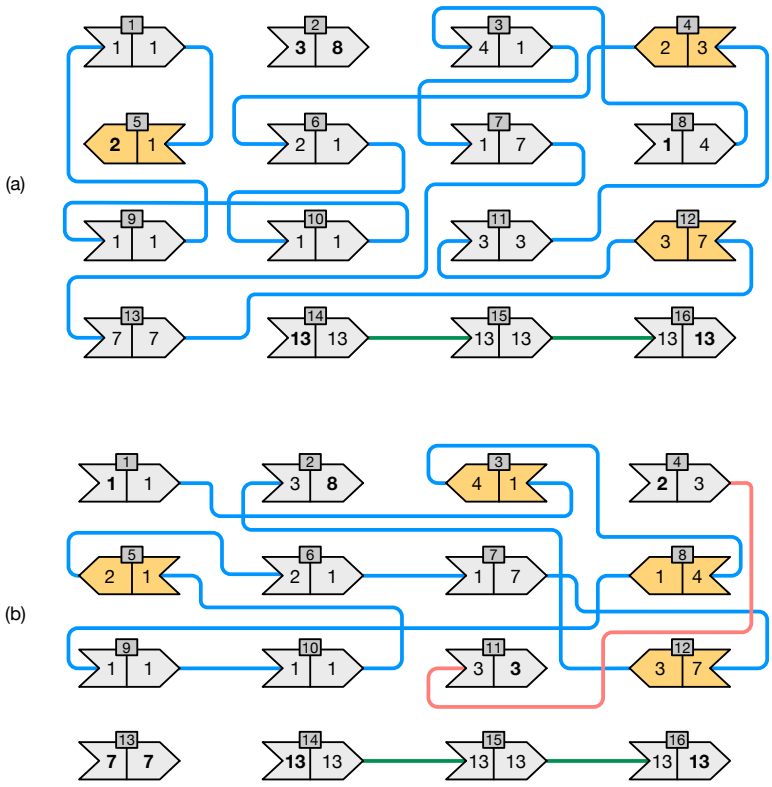


Figure 4 Two different path covers for an example graph with 16 nodes. Nodes linked by edges with the same color belong to the same cover; yellow nodes are those whose orientation was changed. For the graph in the picture, the lower bound in Corollary 9 yields $|C| \geq 3$. The covers in (a) contains 3 paths and is, therefore, optimal. The cover in (b), instead, contains 4 paths.

▶ **Example 11.** We now consider the larger example in Figure 4 for a graph with 16 nodes. In this example we have a set of end-point weights $eW = \{1, 2, 3, 4, 7, 8, 13\}$ and the incidence sets are as follows: $I_1^{max} = \{(1,1), (4,1), (2,1), (2,1), (1,7), (1,4), (1,1), (1,1)\}$; $I_2^{max} = \{(2,3), (2,1), (2,1)\}$; $I_3^{max} = \{(3,8), (2,3), (3,3), (3,7)\}$; $I_4^{max} = \{(4,1), (1,4)\}$; $I_7^{max} = \{(1,7), (3,7), (7,7)\}$; $I_8^{max} = \{(3,8)\}$; $I_{13}^{max} = \{(13,13), (13,13), (13,13)\}$. Since we have $n(I_1^{max}) = 11$, $n(I_2^{max}) = 3$, $n(I_3^{max}) = 5$, and $n(I_8^{max}) = 1$, then $eW_{odd} = \{1, 2, 3, 8\}$ and $|eW_{odd}| = 4$ which is even (Lemma 8). Therefore we are sure that the weights 1, 2, 3, and 8 will appear as end-points of some paths in the cover. The incidence set I_{13}^{max} is made of nodes $(13, 13)$, so $eW_{equal} = \{13\}$ and $|eW_{equal}| = 1$. Also in this case we are sure the weight 13 will appear (twice) as end-point of some path. ($eW_{even} = \{4, 7\}$.) For Corollary 9 we derive that a path cover for the example graph will contain at least $(2 \cdot 1 + 4)/2 = 3$ paths.

Figure 4 shows two different path covers for the same graph, that are $C_a = \{(+8 \to +3 \to +7 \to +13 \to -12 \to +11 \to -4 \to +6 \to +10 \to +9 \to +1 \to -5), (+2), (+14 \to +15 \to +16)\}$ and $C_b = \{(+1 \to -3 \to -8 \to +9 \to +10 \to -5 \to +6 \to +7 \to +12 \to +2), (+4 \to +11), (+13), (+14 \to +15 \to +16)\}$. The cover C_a is optimal for the lower bound as is contains 3 paths; the cover C_b contains 4 paths (1 more than necessary). It is not difficult to see that we cannot find a path cover with less than 3 paths for the graph in Figure 4.

Algorithm 3 The cover algorithm takes an input end-point weight graph $ewG(\mathcal{S})$ and prints a set of paths covering the nodes of $ewG(\mathcal{S})$.

```
1  cover(ewG(S)):
2      incidence = ∅
3      unvisited = ∅
4      for each node u ∈ ewG(S):
5          unvisited.insert(u)
6          incidence[u.left].insert(u)
7          incidence[u.right].insert(u)
8      while unvisited ≠ ∅ :
9          u = unvisited.take()                          ▷ take an unvisited node u
10         p = ∅                                         ▷ a new path
11         while true :
12             extend p with u          ▷ append u to the front or to the back of p
13             unvisited.erase(u)
14             incidence[u.left].erase(u)
15             incidence[u.right].erase(u)
16             if incidence[p.back.right] ≠ ∅ :          ▷ first, try to extend to the right
17                 u = incidence[p.back.right].take()
18             else if incidence[p.front.left] ≠ ∅ :     ▷ then, try to extend to the left
19                 u = incidence[p.front.left].take()
20             else : break                              ▷ p cannot be extended anymore
21         for each u ∈ p :                              ▷ print p
22             print (u.sign, u.id)
```

Lastly, we summarize the main result of this section with the following Theorem.

▶ **Theorem 12.** Let $\mathcal{S}$ be a collection of m weighted strings. Let eW be the set of the distinct end-point weights of the strings in $\mathcal{S}$ and $R = \sum_{i=1}^{m} r_i$, where r_i is the number of runs in the weights of the i-th string of $\mathcal{S}$. Then $\mathcal{S}$ can be permuted to form at least

$$r_{lo} = R - m + (2|eW_{equal}| + |eW_{odd}|)/2 \text{ runs in the weights,} \tag{1}$$

where $eW_{equal} = \{w \in eW \mid I_w^{max} = \{(w, w)\}^d, \text{ for some } d > 0\}$ and $eW_{odd} = \{w \in eW \mid n(I_w^{max}) \text{ is odd}\}$ are defined using the incidence sets for the nodes of $ewG(\mathcal{S})$.

Proof. There are exactly $R - m$ runs in $\mathcal{S}$ that do not depend on the order of the strings in $\mathcal{S}$. Then $\mathcal{S}$ can be permuted as to have $R - m + |C|$ runs where C is a disjoint-node path cover for $ewG(\mathcal{S})$. The theorem follows for Corollary 9 on C. ◀

4.2 Computing a Cover

We showed, via a linear-time reduction (Algorithm 2), that the problem of finding a permutation for $\mathcal{S}$ with the goal of reducing the number of runs in the weights is equivalent to that of computing a path-cover C for the graph $ewG(\mathcal{S})$. In Section 4.1 we also gave a lower bound to the number of runs that our strategy can achieve. The lower bound depends on

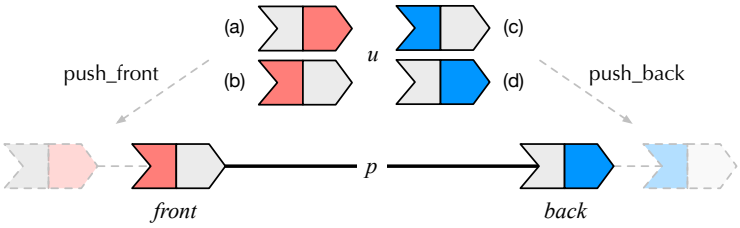


Figure 5 A graphical visualization of line 12 in Algorithm 3 which extends the current path p with a node u. When p is not empty, four different cases can arise, as illustrated in (a), (b), (c), and (d). In cases (b) and (d), the sign of u is changed to match one of the two path's end-points.

the number of paths in C (Corollary 9). In this section we therefore present an algorithm to actually compute a cover C in (optimal) *linear-time* in the number of nodes of $ewG(\mathcal{S})$. Recall that $ewG(\mathcal{S})$ has $m = |\mathcal{S}|$ nodes, so the complexity is $\Theta(m)$.

We recall that the problem of computing a minimum-cardinality path cover for directed graphs is NP-hard. We therefore consider a greedy approach that tries to approximate an optimal solution by extending paths as much as possible to reduce the number of paths in C.

The algorithm is given in Algorithm 3. It manipulates the incidence sets for the end-point weights and a set of unvisited nodes, respectively indicated with *incidence* and *unvisited* in the pseudo-code, and initialized in the lines 4-7. The main loop in the lines 8-22 takes an arbitrary unvisited node and starts a new path from that node. The inner loop in the lines 11-20 greedily tries to extend the current path as much as possible: at every step of the loop, (1) a node is appended to the path, (2) it is erased from the set of unvisited nodes and from the incidence sets where it belongs to, then (3) the next node to append is selected from one of the two incidence sets for the path's end-point weights. When no extension is possible for both ends, the current path is printed in the lines 21-22. In practice, the output can be written to a file, from which the signed permutation π can be derived with Algorithm 2.

If we use hashing to implement the sets *incidence* and *unvisited*, then the operations insert/erase/take are all supported in $O(1)$ on average. Also appending a node to one of the two path's end-points can be done in constant amortized time using a double-ended queue to represent a path. Figure 5 illustrates how the path is extended with a node (line 12). Therefore, the algorithm runs in amortized $\Theta(m)$ time and consumes $\Theta(m)$ space because: (1) at most $2m$ nodes (and at least m) are inserted in *incidence* and exactly m in *unvisited* during the initialization lines 4-7; (2) during the main loop in the lines 8-22, each node is visited, appended to a path, and printed exactly once[2].

5 Experiments

In this section we evaluate the proposed weight compression scheme for SSHash and compare it to several competitive baselines. We first describe our experimental setup.

[2] An important implementation remark: In practice, we implemented the *incidence* sets using a single open-addressing hash-table with a sorted array of $2m$ nodes and an extra array of offsets into the vector to distinguish between the different incidence sets. This makes the algorithm much faster and lighter than a straightforward implementation using a linear-chaining hash-table (e.g., the C++'s `std::unordered_set`), with chains usually implemented as linked lists.

Table 1 Some basic statistics for the datasets used in the experiments, for $k = 31$, such as: number of distinct k-mers (n), number of distinct weights ($|\mathcal{D}|$), largest weight (max), expected weight value (E), and empirical entropy of the weights ($H_0(W)$).

Dataset	n	$\lvert\mathcal{D}\rvert$	$\lceil\log_2 \lvert\mathcal{D}\rvert\rceil$	max	$\lceil\log_2 max\rceil$	E	$H_0(W)$
E-Coli	5,235,781	22	5	27	5	1.05	0.206
S-Enterica-100	13,074,614	587	10	3,483	12	37.47	4.420
Human-Chr-13	90,911,778	806	10	6,354	13	1.08	0.160
C-Elegans	94,006,897	398	9	3,478	12	1.07	0.223

Experiments were run using a server machine equipped with an Intel i9-9940X processor (clocked at 3.30 GHz) and 128 GB of RAM. All the tested software was compiled with gcc 11.2.0 under Ubuntu 19.10 (Linux kernel 5.3.0, 64 bits), using the flags `-O3` and `-march=native`. Our implementation of SSHash is written in C++17 and available at `https://github.com/jermp/sshash`.

All timings were collected using a single core of the processor. The dictionaries are loaded in internal memory before executing queries. For all the experiments, we fix k to 31.

Datasets

We use the following genomic collections: E-Coli and C-Elegans are, respectively, the full genomes of *E. Coli* (Sakai strain) and *C. Elegans* that were also used in the experimentation by Shibuya et al. [40]; S-Enterica-100 is a pan-genome of 100 genomes of *S. Enterica*, collected by Rossi et al. [36]; Human-Chr-13 is the 13-th human chromosome from the genome assembly GRCh38. Table 1 reports some basic statistics for the collections. The weights were collected using the tool BCALM (v2) [7]. In general, note the very low empirical entropy of the weights, $H_0(W)$. This is expected since most k-mers actually appear once for large-enough values of k. Instead, the weights on the pan-genome S-Enterica-100 have much higher entropy due to the fact that many k-mers have weight equal to the number of genomes in the collection (in this specific case, equal to 100). This is useful to test the effectiveness of our encoding on both low- and high- entropy inputs.

Table 2 Space for the weights in SSHash reported in bits/k-mer, *before* and *after* the run-reduction optimization from Section 4. For reference, we also report how many times the achieved space is better than the empirical entropy of the weights $H_0(W)$.

Dataset	$H_0(W)$	*before*		*after*	
E-Coli	0.206	0.017	(12.11×)	0.014	(15.10×)
S-Enterica-100	4.420	0.592	(7.47×)	0.401	(11.02×)
Human-Chr-13	0.160	0.136	(1.18×)	0.107	(1.50×)
C-Elegans	0.223	0.069	(3.23×)	0.055	(4.05×)

Weight Compression in SSHash

We now consider the space effectiveness of the encoding scheme described in Section 3. Table 2 reports the space as average bits per k-mer: we see that, in all cases, the space is well below the empirical entropy lower bound $H_0(W)$ – usually below by several times. The

Table 3 The number of input strings (m) for SSHash as computed by UST [33], the lower bound on the number of runs (r_{lo}) computed using Equation (1) in Theorem 12, and number of actual runs (r) after the optimization. We report the increase, in percentage, of r compared to r_{lo}. The last two columns show the run-time of the path cover Algorithm 3, in total milliseconds (ms) and average nanoseconds per node (ns/node).

Dataset	m	r_{lo}	r		Alg. 3 (ms)	Alg. 3 (ns/node)
E-Coli	2,102	3,723	3,723	(+0.0000%)	0.6	285
S-Enterica-100	150,604	277,649	277,658	(+0.0032%)	53.0	352
Human-Chr-13	266,113	462,175	462,197	(+0.0048%)	94.6	355
C-Elegans	140,452	247,661	247,669	(+0.0032%)	47.1	335

optimization strategy described in Section 4 brings further advantage. (The space shown is comprehensive of the $|\mathcal{D}|\lceil \log_2 max \rceil$ bits used to represent the distinct weights in the collection. Note that this space takes a negligible fraction of the total space since $|\mathcal{D}|$ is very small as reported in Table 1.)

Table 3, instead, shows the performance of the path cover Algorithm 3. As already mentioned in Section 3, the set of strings indexed by SSHash is obtained by building a spectrum-preserving string set (SPSS) from the raw genome, using the algorithm UST [33] over the output of BCALM [7]. (At our code repository `https://github.com/jermp/sshash` we provide further details on how to take these preliminary steps before indexing with SSHash.) The number of strings in each collection, m, determines the run-time of Algorithm 3 whose complexity is $\Theta(m)$. The linear-time complexity is evident from the reported timings and makes the algorithm very fast, taking on average a fraction of a microsecond per node.

The other important point to note is that Algorithm 3 is empirically *optimal* regarding the number of runs given that the final number of runs, r, is only slightly higher than the lower bound r_{lo} computed using Equation (1).

(In Appendix B we report additional experimental results.)

Overall Comparison

In Table 4 we show a comparison between the following weighted dictionaries (see also Section 2; links to the code repositories are included in the References):

- The dBG-FM index [6] based on the popular FM-index [11]. In particular, this representation implements a weighted k-mer dictionary via the *count* query which returns the number of occurrences of a given k-mer in the input. The *count* query, in turn, is implemented using rank queries over the BWT. The dBG-FM implementation has a main trade-off parameter, s, to control the practical performance of rank queries. We test the values $s = 32, 64, 128$.
- The recent cw-dBG [12] dictionary based on the data structure called BOSS [4]. Similarly to an FM-index, also cw-dBG has a trade-off parameter that we vary as $s = 32, 64, 128$. (The authors used $s = 64$ in their own experiments.)
- The *non*-weighted SSHash itself coupled with the fast compressed static function (CSF) tailored for low-entropy distributions, proposed by Shibuya et al. [40]. As reviewed in Section 2, a CSF does not represent the k-mers but just realizes a map from k-mers to their weights. Such map is collision-free only over the set of k-mers that was used to actually build the function. Therefore, we use SSHash as an efficient dictionary for the k-mers and the CSF to represent the weights. The authors proposed two different versions of their approach, BCSF and AMB, with different space/time trade-offs.

Table 4 Dictionary space in average bits/k-mer (bpk) and total MB, and query time in average μsec/k-mer (qtm). For reference, we report in gray color the space and time of SSHash *without* the weight information.

Dictionary	E-Coli			S-Enterica-100			Human-Chr-13			C-Elegans		
	bpk	MB	qtm	bpk	MB	qtm	bpk	MB	qtm	bpk	MB	qtm
dBG-FM, $s = 128$	3.20	2.00	14.73	113.78	177.34	16.47	3.23	34.97	17.40	3.18	35.60	18.05
dBG-FM, $s = 64$	4.02	2.51	7.91	142.25	221.71	11.13	4.07	44.07	11.33	4.01	44.89	10.89
dBG-FM, $s = 32$	5.65	3.53	4.62	198.71	309.72	8.57	5.73	62.15	8.20	5.67	63.49	7.90
cw-dBG, $s = 128$	2.79	1.82	109.13	5.59	9.13	120.72	2.80	31.77	100.88	2.77	32.54	127.86
cw-dBG, $s = 64$	2.86	1.87	70.93	5.74	9.39	85.73	2.86	32.55	73.91	2.84	33.34	84.19
cw-dBG, $s = 32$	2.99	1.96	52.29	6.03	9.85	66.25	2.99	34.02	59.85	2.97	34.87	62.54
SSHash+BCSF	5.07	3.31	0.82	11.12	18.17	0.89	6.15	69.89	1.25	6.00	70.51	1.28
SSHash+AMB	4.90	3.21	1.34	9.27	15.15	1.65	6.08	69.09	1.95	5.88	61.42	1.97
w-SSHash	4.80	3.14	0.37	6.57	10.74	0.48	6.04	68.66	0.84	5.75	67.52	0.85
SSHash	4.79	3.14	0.34	6.17	10.08	0.41	5.93	67.39	0.76	5.69	66.86	0.77

- The weighted SSHash dictionary proposed in this work, which we refer to as w-SSHash in the following, *after* the run-reduction optimization (Table 2 and 3). We use the *regular* index variant of SSHash. The main parameter of the index – the *minimizer* length – is always set to $\lceil \log_4 N \rceil + 1$ where N is the number of nucleotides in the SPSSs of the datasets, following the recommendation given in the previous paper [25]. Therefore, we use the following minimizer lengths: 13, 14, 15, and 15, for respectively, E-Coli, S-Enterica-100, Human-Chr-13, and C-Elegans. Also the AMB algorithm by Shibuya et al. [40] is based on minimizers and we use the same lengths.

We did not compare against deBGR [22] and Squeakr [23] as the authors of cw-dBG showed in their experimentation [12] that both tools take considerably more space than cw-dBG, e.g., one order of magnitude more space. Here, we are interested in a good balance between space effectiveness and query efficiency.

To measure query-time – the time it takes to retrieve the weight $w(g)$ given the k-mer g – we sampled 10^6 k-mers uniformly at random from the collections and use them as queries. We report the mean between 5 measurements. Half of the queries were transformed into their reverse complements to make sure we benchmark the dictionaries in the most general case.

The space of w-SSHash is generally competitive with that of the fastest variant of dBG-FM ($s = 32$), but w-SSHash has (more than) one order of magnitude better query time. Note that on S-Enterica-100 the dBG-FM index is space-inefficient since it redundantly represents many repeated k-mers. Using a higher sampling rate reduces the space of dBG-FM at the price of slowing down query-time; however, the most space-efficient variant tested ($s = 128$) is not even 2× smaller than w-SSHash.

The cw-dBG index is the smallest tested dictionary. Its space effectiveness is comparable to that of dBG-FM $s = 128$, and indeed generally twice as better as that of w-SSHash. The price to pay for this enhanced compression ratio is a significant penalty at query-time. Indeed, w-SSHash can be two order of magnitude faster than cw-dBG. Consider, for example, the two dictionaries built for S-Enterica-100: we have 0.5 vs. 66-120 μs per query.

The two CSFs, BCSF and AMB, make SSHash 2-3× slower than w-SSHash and even consistently larger. This comparison motivates the need for a unified data structure to handle efficiently both the k-mers *and* the weights, like w-SSHash. While the increase in space due to the CSF is not much for the low-entropy datasets because both BCSF and AMB are very space-efficient in those cases, the gap is more evident on S-Enterica-100.

As a last note, observe that there is no significant slowdown in accessing the weights in w-SSHash compared to a simpler membership query (the time reported in shaded color in Table 4), hence proving the RLE-based scheme to be efficient too and not only very effective.

6 Conclusions

In this work we extended the recent SSHash [25] dictionary to also store the weights of the k-mers in compressed format. In particular, we represented the weights using compressed *runs* of equal symbols. While using run-length encoding to compress highly repetitive sequences is not novel per se and indeed a folklore strategy at the basis of many other data structures, this allows to use a very small extra space (e.g., much less than the empirical entropy of the weights) on top of SSHash with only a slight penalty at retrieval time. The crucial point is that it is possible to use run-length encoding because SSHash *preserves the (relative) order* of the k-mers in the indexed sequences. The main practical take-away is, therefore, that SSHash handles weighted k-mer sets in an *exact* manner without noticeable extra costs. Our software is publicly available to encourage its use and reproducibility of results.

We also introduced the concept of *end-point weight graph* (*ewG*) and showed its usefulness in reducing the number of runs in the weights. Precisely, we showed that minimizing the number of runs in a collection of sequences corresponds to the problem of computing a minimum-cardinality path cover for the *ewG* of the sequences. We presented a greedy algorithm that computes a cover in linear-time (in the number of nodes of the graph) and showed that it is empirically almost optimal according to a lower bound on the number of runs. As a result of this optimization, the space spent to represent the weights is unlikely to be improved using run-length encoding.

Although several approaches in the literature [18, 23, 40, 39] also consider *approximate* weights, we did not pursue this direction here as the weights are already encoded space-efficiently in SSHash and in an *exact* way, so there may be no need for approximation.

The distribution of weights in large collections is usually expected to be very skew, i.e., most k-mers actually appear once and few of them repeat many times [40, 39]. A common strategy to save space is then to avoid the representation of the most frequent weight(s). Note that, since we represent runs of weights and not the individual weights, we are already optimizing (potentially very large) sub-sets of weights equal to the most frequent one. That is, run-length encoding is also a good match for such skew distributions.

References

1 Fatemeh Almodaresi, Hirak Sarkar, Avi Srivastava, and Rob Patro. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics*, 34(13):i169–i177, 2018.

2 Uwe Baier, Timo Beller, and Enno Ohlebusch. Graphical pan-genome analysis with compressed suffix trees and the burrows–wheeler transform. *Bioinformatics*, 32(4):497–504, 2016.

3 Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, et al. Spades: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of computational biology*, 19(5):455–477, 2012.

4 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *International Workshop on Algorithms in Bioinformatics (WABI)*, pages 225–235. Springer, 2012.

5 Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer, 1994.

6 Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. On the representation of de Bruijn graphs. In *International conference on Research in computational molecular biology*, pages 35–55. Springer, 2014. URL: `https://github.com/jts/dbgfm`.

7 Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 2016. URL: `https://github.com/GATB/bcalm`.

8 Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. Kmc 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

9 Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.

10 Robert Mario Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT*, 1971.

11 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

12 Giuseppe Italiano, Nicola Prezza, Blerina Sinaimeri, and Rossano Venturini. Compressed weighted de Bruijn graphs. In *32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021)*, volume 191, pages 16:1–16:16, 2021. URL: `https://github.com/nicolaprezza/cw-dBg`.

13 Shaun D Jackman, Benjamin P Vandervalk, Hamid Mohamadi, Justin Chu, Sarah Yeo, S Austin Hammond, Golnaz Jahesh, Hamza Khan, Lauren Coombe, Rene L Warren, et al. Abyss 2.0: resource-efficient assembly of large genomes using a bloom filter. *Genome research*, 27(5):768–777, 2017.

14 Mikhail Karasikov, Harun Mustafa, Gunnar Rätsch, and André Kahles. Lossless indexing with counting de bruijn graphs. *bioRxiv*, 2021.

15 Parsoa Khorsand and Fereydoun Hormozdiari. Nebula: ultra-efficient mapping-free structural variant genotyper. *Nucleic acids research*, 49(8):e47–e47, 2021.

16 Danyang Ma, Simon J Puglisi, Rajeev Raman, and Bella Zhukova. On elias-fano for rank queries in fm-indexes. In *2021 Data Compression Conference (DCC)*, pages 223–232. IEEE, 2021.

17 Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

18 Camille Marchet, Zamin Iqbal, Daniel Gautheret, Mikaël Salson, and Rayan Chikhi. Reindeer: efficient indexing of k-mer presence and abundance in sequencing datasets. *Bioinformatics*, 36(Supplement_1):i177–i185, 2020.

19 Shoshana Marcus, Hayan Lee, and Michael C Schatz. Splitmem: a graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics*, 30(24):3476–3483, 2014.

20 Giuseppe Ottaviano and Rossano Venturini. Partitioned elias-fano indexes. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 273–282, 2014.

21 Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM international conference on Management of Data*, pages 775–787, 2017.

22 Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics*, 33(14):i133–i141, 2017.

23 Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics*, 34(4):568–575, 2018.

24 Raffaele Perego, Giulio Ermanno Pibiri, and Rossano Venturini. Compressed indexes for fast search of semantic data. *IEEE Trans. Knowl. Data Eng.*, 33(9):3187–3198, 2021.

25 Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *Bioinformatics*, 38(Supplement_1):i185–i194, June 2022. doi:10.1093/bioinformatics/btac245.

26 Giulio Ermanno Pibiri and Roberto Trani. Parallel and external-memory construction of minimal perfect hash functions with PTHash. *CoRR*, abs/2106.02350, 2021. arXiv:2106.02350.

27 Giulio Ermanno Pibiri and Roberto Trani. PTHash: Revisiting FCH minimal perfect hashing. In *SIGIR '21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event, Canada, July 11-15, 2021*, pages 1339–1348, 2021.

28 Giulio Ermanno Pibiri and Rossano Venturini. Clustered Elias-Fano indexes. *ACM Trans. Inf. Syst.*, 36(1):2:1–2:33, 2017.

29 Giulio Ermanno Pibiri and Rossano Venturini. Efficient data structures for massive n-gram datasets. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 615–624, 2017.

30 Giulio Ermanno Pibiri and Rossano Venturini. Handling massive N-gram datasets efficiently. *ACM Trans. Inf. Syst.*, 37(2):25:1–25:41, 2019.

31 Giulio Ermanno Pibiri and Rossano Venturini. On optimally partitioning variable-byte codes. *IEEE Trans. Knowl. Data Eng.*, 32(9):1812–1823, 2020.

32 Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression. *ACM Comput. Surv.*, 53(6):125:1–125:36, 2021.

33 Amatur Rahman and Paul Medvedev. Representation of k-mer sets using spectrum-preserving string sets. In *International Conference on Research in Computational Molecular Biology*, pages 152–168. Springer, 2020. URL: https://github.com/medvedevgroup/UST.

34 Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. Dsk: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, 2013.

35 Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

36 Mirko Rossi, Mickael Santos Da Silva, Bruno Filipe Ribeiro-Gonçalves, Diogo Nuno Silva, Miguel Paulo Machado, Mónica Oleastro, Vítor Borges, Joana Isidro, Luis Viera, Jani Halkilahti, Anniina Jaakkonen, Federica Palma, Saara Salmenlinna, Marjaana Hakkinen, Javier Garaizar, Joseba Bikandi, Friederike Hilbert, and João André Carriço. INNUENDO whole genome and core genome MLST schemas and datasets for Salmonella enterica, July 2018. doi:10.5281/zenodo.1323684.

37 Kristoffer Sahlin. Effective sequence similarity detection with strobemers. *Genome research*, 31(11):2080–2094, 2021.

38 Kristoffer Sahlin. Strobemers: an alternative to k-mers for sequence comparison. *bioRxiv*, 2021.

39 Yoshihiro Shibuya, Djamal Belazzougui, and Gregory Kucherov. Set-min sketch: a probabilistic map for power-law distributions with application to k-mer annotation. *Journal of Computational Biology*, 29(2):140–154, 2022.

40 Yoshihiro Shibuya, Djamal Belazzougui, and Gregory Kucherov. Space-efficient representation of genomic k-mer count tables. *Algorithms for Molecular Biology*, 17(1):1–15, 2022. URL: https://github.com/yhhshb/locom.

41 Daniel S Standage, C Titus Brown, and Fereydoun Hormozdiari. Kevlar: a mapping-free framework for accurate discovery of de novo variants. *Iscience*, 18:28–36, 2019.

42 Sebastiano Vigna. Quasi-succinct indices. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 83–92, 2013.

43 Derrick E Wood and Steven L Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology*, 15(3):1–12, 2014.

A Omitted Proofs from Section 4.1

▶ **Lemma 3.** Let us consider $d > 0$ equal nodes (w, x). If d is even, then the d nodes can be oriented to form a maximal path of either end-points (w, w) or (x, x). If d is odd, then the path has end-points (w, x).

Proof. We proceed by induction on d. Base case: if $d = 1$ (odd case), then there is only the singleton path (w, x); if $d = 2$ (even case), then we can either form the path $(w, x) \to (x, w)$ of end-points (w, w) or the path $(x, w) \to (w, x)$ of end-points (x, x). So the base case is verified. Now we assume the Lemma holds true for a generic $d > 2$ and we want to prove it for $d + 1$. If d is even, then $d + 1$ is odd and we can either have a path $(w, w) \to (w, x)$ or a path $(w, x) \to (x, x)$. In both cases the end-points are (w, x). Symmetrically: if d is odd, then $d + 1$ is even and we can either have a path $(w, x) \to (x, w)$ with end-points (w, w) or a path $(x, w) \to (w, x)$ with end-points (x, x). ◀

▶ **Lemma 6.** Given an incidence set I_w, if $n(I_w)$ is *odd* then only one path will contain w as end-point among all the maximal paths that can be created from the nodes in I_w.

Proof. Let us first consider the special case where all the other weights in I_w are distinct, so there are no equal nodes in I_w except for, possibly, nodes of the form (w, w). In this case, we say that I_w is *canonical*. If there are some nodes (w, w), they can be trivially collapsed to a maximal path of end-points (w, w) by Lemma 3. So, without loss of generality, either we have *one* node (w, w) in I_w and $n(I_w) = |I_w| + 1$, or we do not and $n(I_w) = |I_w|$. In this special case, since $n(I_w)$ is odd, it is always possible to create $\frac{n(I_w)-1}{2}$ paths, each having 2 nodes. These paths will *not* contain w as end-point because all the end-points where w appears are used to link the nodes. Therefore, there will be *exactly one* unpaired node where w appears.

Now, observe that we can relax the restriction on the other weights to be all distinct. For every weight $x \neq w$ that appears for $d_x > 1$ times in I_w, there are d_x equal nodes (w, x). Let $D_x \subseteq I_w$ be the set of such nodes (hence, $d_x = |D_x|$). By applying Lemma 3 to the nodes of each set D_x:

- If d_x is even, then the nodes in D_x can be collapsed into a maximal path of end-points (w, w) or (x, x). If the node (w, w) is created, we obtain a new incidence set $I'_w = I_w \setminus D_x \cup \{(w, w)\}$. Instead, if the node (x, x) is created, then $I'_w = I_w \setminus D_x$ since (x, x) cannot be in an incidence set for w. In both cases $n(I'_w)$ will still be odd since we subtract an even number from $n(I_w)$.
- If d_x is odd, the nodes are collapsed into the maximal path of end-points (w, x) and the new incidence set is $I'_w = I_w \setminus D_x \cup \{(w, x)\}$. Again, $n(I'_w)$ will still be odd since we subtract an odd number from $n(I_w)$ but sum one.

After each set D_x is processed in this way, we are left with an incidence set for w that is canonical. ◀

▶ **Lemma 8.** $|eW_{odd}|$ is even.

Proof. Observe that $\sum_{w \in eW} n(I_w^{max})$ is even and equal to $2m$ because we count the occurrences of the weights appearing as end-points of the sequences and each sequence has two end-points. Since $eW = eW_{odd} \cup eW_{even} \cup eW_{equal}$, the above sum can be re-written as

$$\sum_{w \in eW} n(I_w^{max}) = \sum_{w \in eW_{odd}} n(I_w^{max}) + \sum_{w \in eW_{even}} n(I_w^{max}) + \sum_{w \in eW_{equal}} n(I_w^{max}) = 2m.$$

It follows that also

$$\sum_{w \in eW_{odd}} n(I_w^{max}) = 2m - \sum_{w \in eW_{even}} n(I_w^{max}) - \sum_{w \in eW_{equal}} n(I_w^{max})$$

must be even since it is obtained by difference of even quantities. Since each term in the sum $\sum_{w \in eW_{odd}} n(I_w^{max})$ is odd by definition, the whole sum is even if and only if $|eW_{odd}|$ is even, as the sum of an odd number of odd numbers is odd. ◀

Table 5 The performance of Alg. 3 on the datasets Cod, Kestrel, Human, and Bacterial, for which we report the number of distinct k-mers (n) and the number of strings (m) after running UST [33] on the collections. The performance of the algorithm is expressed as: the number of actual runs (r) after the run-reduction optimization in comparison with the lower bound on the number of runs (r_{lo}) computed using Equation (1), and running time (in total sec and average ns/node).

Dataset	n	m	r_{lo}	r		Alg. 3 (sec)	Alg. 3 (ns/node)
Cod	502,465,200	2,406,681	4,183,202	4,183,230	(+0.00067%)	1.2	500
Kestrel	1,150,399,205	682,344	1,140,743	1,140,747	(+0.00035%)	0.3	440
Human	2,505,445,761	13,014,641	22,680,047	22,680,099	(+0.00023%)	7.5	580
Bacterial	5,350,807,438	26,448,286	56,662,230	56,662,304	(+0.00013%)	17.2	650

Table 6 The performance of w-SSHash on the permuted string collections Cod, Kestrel, Human, and Bacterial. We report the empirical entropy of the weights ($H_0(W)$), the dictionary space in average bits/k-mer (bpk) and total GB, and query-time in average μsec/k-mer (qtm). The space is indicated as $x + y$, where x is the space of SSHash (without the weights) and y is the space for the encoding of the weights. In parentheses we report the space reduction of the encoded weights compared to the empirical entropy of the weights.

Dataset	$H_0(W)$	bpk		GB	qtm
Cod	0.441	6.98+0.19	(2.35×)	0.45	1.3
Kestrel	0.089	6.49+0.02	(3.80×)	0.94	1.1
Human	0.453	8.28+0.22	(2.06×)	2.66	1.6
Bacterial	1.890	8.22+0.24	(7.81×)	5.66	1.9

B Additional Experimental Results

In Table 5 and Table 6 we report the performance of Alg. 3 and of w-SSHash on four additional, larger, collections that we also used in our previous work [25], namely the full genomes of *G. Morhua* (Cod), *F. Tinnunculus* (Kestrel), and *H. Sapiens* (Human), and a collection of more than 8000 bacterial genomes (Bacterial) [1]. Precisely, the results in Table 6 are for regular w-SSHash dictionaries with minimizer lengths equal to 17, 17, 20, and 20, for respectively, Cod, Kestrel, Human, and Bacterial.

Accurate k-mer Classification Using Read Profiles

Yoshihiko Suzuki[1]
Okinawa Institute of Science and Technology Graduate University, Okinawa, Japan

Gene Myers[2]
Okinawa Institute of Science and Technology Graduate University, Okinawa, Japan
Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany
Center for Systems Biology Dresden, Dresden, Germany

Abstract

Contiguous strings of length k, called k-mers, are a fundamental element in many bioinformatics tasks. The number of occurrences of a k-mer in a given set of DNA sequencing reads, its k-mer count, has often been used to roughly estimate the copy number of a k-mer in the genome from which the reads were sampled. The problem of estimating copy numbers, called here the *k-mer classification* problem, has been based on simply analyzing the histogram of counts of all the k-mers in a data set, thus ignoring the positional context and dependency between multiple k-mers that appear nearby in the underlying genome. Here we present an efficient and significantly more accurate method for classifying k-mers by analyzing the sequence of k-mer counts along each sequencing read, called a *read profile*. By analyzing read profiles, we explicitly incorporate into the model the dependencies between the positionally adjacent k-mers and the sequence context-dependent error rates estimated from the given dataset. For long sequencing reads produced with the accurate high-fidelity (HiFi) sequencing technology, an implementation of our method, **ClassPro**, outperforms the conventional, histogram-based method in every simulation dataset of fruit fly and human with various realistic values of sequencing coverage and heterozygosity. Within only a few minutes, ClassPro achieves an average accuracy of $> 99.99\%$ across reads without repetitive k-mers and $> 99.5\%$ across all reads, in a typical fruit fly simulation data set with a $40\times$ coverage. The resulting, more accurate k-mer classifications by ClassPro are in principle expected to improve any k-mer-based downstream analyses for sequenced reads such as read mapping and overlap, spectral alignment and error correction, haplotype phasing, and trio binning to name but a few. ClassPro is available at `https://github.com/yoshihikosuzuki/ClassPro`.

2012 ACM Subject Classification Applied computing → Molecular sequence analysis

Keywords and phrases K-mer, K-mer count, K-mer classification, HiFi sequencing

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.10

Acknowledgements We wish to thank Shinichi Morishita, Yuta Suzuki, Bansho Masutani, Ryo Nakabayashi, Charles Plessy, and Michael Mansfield for their feedback and stimulating works. We also thank the Scientific Computing and Data Analysis section of Research Support Division and Communication and Public Relations Division at OIST for providing HPC resources and for proofreading of the manuscript, respectively.

1 Introduction

Long read DNA sequencing technologies are enabling the *de novo* reconstruction of reference quality genomes providing the impetus for projects such as the Vertebrate Genome Project [26], the Darwin Tree of Life Project [30] and the Human Pangenome Project [33], whose goals are to build reference atlases of entire phyla, eco-systems of living creatures, or worldwide

[1] Current affiliation: Department of Computational Biology and Medical Sciences, The University of Tokyo, Japan
[2] Corresponding author

human populations. In addition to dramatic gains in read length, the most recent arrival of long reads with an error rate of only 0.2%, as for example realized by PacBio's HiFi protocol [34], permits haplotype phasing and the resolution of many complex repetitive regions [4, 5, 7, 8, 21] because there is almost always a modest level of heterogeneity between the haplotypes or repeat elements in wild-type animals. However, the goal of a perfect telomere to telomere, phased reconstruction of a multiploid genome is as yet unrealized [11], requiring either better or more data, or better assembly algorithms. In this paper, an initial analysis of a high fidelity shotgun data set delivers precise information about phasing and repetitiveness, that should in principle improve the performance of any downstream assembly method.

In a typical high fidelity data set of a genome G, a collection of reads R is collected so that the genome is covered $c = \sum_{S \in R} |S|/|G|$ times where c is typically 10-30 and the read length $|S|$ averages from 10Kbp to 25Kbp (with current technologies). Note carefully, that c is the **haploid** coverage and G is the phased genome, i.e. it is not a consensus haplotype but the set of all distinct haplotype sequences. For example, for a human genome, G is 6Gbp in length and consists of 46 sequences corresponding to the full diploid set of chromosomes. In the treatment that follows, only R is known, and G is a hypothetical used for definitional purposes.

A k-mer is a string of length k defined over the DNA nucleotides. For any set of DNA sequences X, a k-mer counter such as Meryl [32], Jellyfish [17], KMC [13], or FastK [19], calculates the number of occurrences of each distinct k-mer in X. For a k-mer, α, let $\#_X(\alpha)$ denote the number of times α occurs in X. Reference to "*the* count of α" implicitly refers to $\#_R(\alpha)$, the count in the read data set R. A naive expectation for a shotgun data set R and implied genome G is that $\#_R(\alpha) \sim c \cdot \#_G(\alpha)$ subject to stochastic fluctuations in the arrival of sequencing errors and the read sampling process. This in turn implies that the histogram, $\mathcal{H}_R$, of $\#_R$ (often called the k-mer spectrum of R) will typically have discernible peaks at $0, c, 2c, \cdots$ (Fig. 1a) where the k-mers about 0 are considered to be due to errors in the reads as it is most likely that $\#_G = 0$ for said; i.e. they are not in G. This basic observation has lead to a number of k-mer-based analysis tools. For example, GenomeScope [25] estimates the size, ploidy, heterozygosity, and repeat fraction of G and the error rate of R by fitting a negative binomial mixture model to the histogram $\mathcal{H}_R$. As another example, KAT [16], Merqury [27], and Merfin [6] evaluate the completeness of an assembly of R using a complete table of $\#_R$ and the histogram $\mathcal{H}_R$.

In this paper, we consider the ***k*-mer classification problem** to be that of inferring $\#_G(\alpha)$ for every k-mer α in R. In all previous work of which we are aware, this classification is based solely on the count of α in the context of the histogram $\mathcal{H}_R$; e.g. a k-mer α is deemed an error (i.e. $\#_G(\alpha) = 0$) if the count of α is less than some fixed threshold based on an examination of $\mathcal{H}_R$. Let $\mathcal{H}_{R|v}$ be the histogram or distribution of $\{\ \#_R(\alpha)\ :\ \#_G(\alpha) = v\ \}$, that is, the counts of the k-mers with classification v. Then because of sequencing error and the stochasticity of the Poisson sampling process, the distributions $\mathcal{H}_{R|0}$, $\mathcal{H}_{R|1}$, $\mathcal{H}_{R|2}$, $\cdots$ can and typically do overlap significantly, increasingly so as a function of v and the sequencing error rate. This inseparability implies that the many assemblers (e.g [23, 2, 12, 29, 14, 3, 1]) using k-mers for tasks such as seeding alignments, spectral error correction, or haplotype phasing, are working with classifications (or probabilities of classifications) that are often incorrect as much as 5–10% of the time. So clearly, having highly accurate classifications would improve the performance of all of these systems.

Here we present a method of k-mer classification over a high-fidelity read data set that has a typically accuracy of $> 99.9\%$. We do so by exploiting the contextual information between positionally close k-mer counts along each read $S \in R$. We term the sequence

counts for consecutive k-mers along a read S, a **count profile** (Fig. 1a). In a count profile, neighboring k-mer counts are dependent on each other, providing much stronger statistical leverage than found in the histogram $\mathcal{H}_R$. For example, if a k-mer is an error (classification 0), then typically $O(k)$ neighbors about this k-mer in the profile are also errors. An even more significant observation is that if two consecutive k-mers in the profile have the same classification then their counts will only vary if (a) there has been a read arrival or departure in the underlying sampling of reads or (b) some number of reads that have an error in said k-mer changes. In contrast, if they have different classifications, then there will be a difference on the order of c or more in their counts. As a concrete example, consider a 20× HiFi data set of 15Kbp reads (40× of a diploid genome). A read arrives on average every 375bp and an error occurs every 500bp assuming a 0.2% error rate. One thus expects typically a change of 0, 1 or 2 counts between successive k-mers in the same class and a change on the order of 20 or so counts if the classification changes.

While the read sampling process for PacBio data is to first order Poisson, the likelihood of an error at a given point in a sequence is known to vary widely depending on context. For example, the most common errors that account for $\sim 80\%$ of all the errors in the HiFi sequencing are homopolymer indels [34, 22], followed by copy number errors in dinucleotide satellites, and thereafter those of trinucleotide satellites. It is therefore important to account for this as it can considerably affect the probability of a count transition being due to a classification change versus due to error and read sampling. In other work, the dominance of homopolymer errors was effectively by-passed with homopolymer compression of the reads as in HiCanu [22] and LJA [1]. This approach however does not account for elevated error rates around di- and tri-nucleotide satellites, so in this work we develop a data-driven model of sequence-dependent error.

In this paper, we describe an algorithm and software implementation, ClassPro, that for each read count profile of a diploid genome, classifies every k-mer α in the profile into one of the four types: **error** ($\#_G(\alpha) = 0$), **haploid** ($\#_G(\alpha) = 1$), **diploid** ($\#_G(\alpha) = 2$), and **repeat** ($\#_G(\alpha) \geq 3$). The concept of a count profile has been sporadically seen in previous work [36, 24, 18, 16], but we leverage both stochastic and deterministic properties of a profile to improve k-mer classification. We show empirically that the resulting classifications are highly accurate and so using these in previously studied contexts like error correction, read overlap detection, haplotype phasing, and trio binning should result in significant performance improvements.

2 Preliminaries

2.1 Problem definition and terminology

Throughout the paper, we focus on classifying the k-mers of a single read $S \in R$ of length N. Let the sequence of S, $Seq(S) = s_{-k+1}s_{-k+2}\cdots s_{-1}s_0s_1\cdots s_{N-k}$, and let the $N-k+1$ k-mers of S, $Kmer(S) = \alpha_0\alpha_1\cdots\alpha_{N-k}$, where $\alpha_i = s_{i-k+1}\cdots s_i$. Note that this unusual definition of $Seq(S)$ prefixed by $k-1$ negative indices is for a definitional purpose so that the first k-mer affected by any change to nucleotide s_j is α_j, which will be used in Methods. Lastly, let the k-mer count profile of S, $Count(S) = c_0c_1\cdots c_{N-k}$, where $c_i = \#_R(\alpha_i)$.

For a k-mer α, we call it a haplo-mer iff $\#_G(\alpha) = 1$ and a diplo-mer iff $\#_G(\alpha) = 2$. Note that almost all of the diplo-mers are the homozygous k-mers shared among both alleles, but a very small fraction of them typically consists of paralogous copies of k-mers, especially in repetitive regions. Similarly we call α an error-mer iff $\#_G(\alpha) = 0$ and a repeat-mer iff $\#_G(\alpha) > 2$. (NB: a sequencing error in a read does not necessarily imply that every

k-mer spanning it is an error-mer as it may occur elsewhere in the genome, however this happens rarely, increasingly so as k increases.) Let $\mathcal{T}$ be the set of the four k-mer types, i.e. {E(rror), H(aplo), D(iplo), R(epeat)}. For each k-mer α_i in $Kmer(S)$, let $\tau_i \in \mathcal{T}$ be the true type of α_i, which is unknown, and $t_i \in \mathcal{T}$ be the inferred assignment of a type of α_i by some method. Then, our ultimate objective is to find the best sequence of assignments, $Class(S) = t_0 \cdots t_{N-k}$, such that the number of wrong classifications, i.e. $|\{\ \alpha_i \mid t_i \neq \tau_i\ \}|$, is minimized.

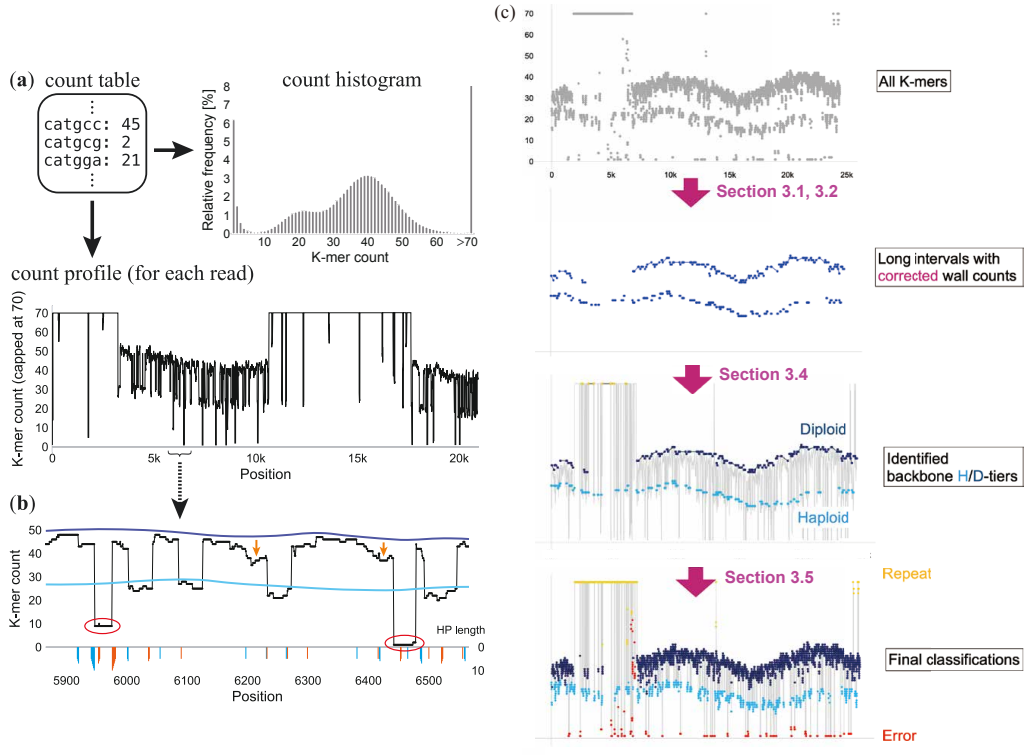


Figure 1 **(a)** Examples of a histogram $\mathcal{H}_R$ and a profile $Count(S)$ of k-mer counts (real HiFi data, $c = 20$ and $k = 40$). In the profile, consecutive k-mer counts are connected with solid lines. **(b)** While a count of 30 is hard to classify based only on $\mathcal{H}_R$, in this region we can conclude it is likely to be haploid rather than diploid because we clearly see two "tiers" of haploid (light blue curve) and diploid (dark blue) segments fluctuating smoothly. The sharp and large count drops (red circles) are caused by sequencing errors occurring in S. In contrast, errors in other reads result in small drops from a tier (orange arrows). The bars at the bottom indicate the homopolymer length for count drop (blue, $\overrightarrow{l}_i^{\mathrm{HP}}$ in Methods) and gain (orange, $\overleftarrow{l}_{i-k+1}^{\mathrm{HP}}$) where only $\geq$ 5bp are depicted. The drop to count 9 at position ~5,950 exemplifies an elevated number of co-occurrences of a homopolymer error. **(c)** The flow of ClassPro. The second and third subplots show reliable intervals instead of k-mers, and the background profile is depicted in gray in the third and last subplots for clarity.

2.2 Anatomy of a k-mer count profile

The intuition that read profiles are effective for the k-mer classification problem is based on two key observations that hold for a typical HiFi dataset: the **coherence principle** and the ***k*-knockout principle**. In brief, the coherence principle is that "adjacent k-mers with the same copy number have similar counts." In other words, the count change between two consecutive k-mers with the same copy number $\#_G$ is relatively much smaller than the

count change due to a copy number change. The k-knockout principle is that a transition from a higher copy number to a lower copy number in the event of a sequencing error or allelic variant lasts for roughly k-or-more k-mers, because $\sim k$ consecutive k-mers share the nucleotide(s) of the event. These two principles create local dependencies among adjacent k-mers that cannot be captured with a histogram.

To elaborate these principles, we consider what a count profile $Count(S)$ should look like from a generative perspective (Fig. 1b). First suppose a read has no errors and is sampled from a region in a diploid genome that is completely homozygous and non-repetitive. Then all the k-mers are diplo-mers and every change between two consecutive counts, c_{i-1} and c_i, is fully explained by the read sampling process. That is, c_i get $+1$ compared to c_{i-1} for every read starting at i and -1 for every read ending at $i-1$. Since the average number of read arrivals per position is $2c/N$ when c is the haploid coverage and N is the read length, for long reads, where $2c/N \ll 1$, the count profile without sequencing errors is very smooth. However, do note that over a large number of bases the counts can drift up and down significantly based on the underlying undulation in the Poisson sampling process, i.e. $|c_i - c_j|$ can be large when $|i-j|$ is large.

Next suppose the read now comes from a heterozygous region where the haplotype it was sampled from varies from its mate in a number of places. Then the k-mers spanning the variant sites will be haplo-mers and those not will be diplo-mers, effectively partitioning the profile into diplo-mer segments and haplo-mer segments. By coherence these segments will be smooth with $O(c)$ jumps between segments. Basically the diplo-mer and haplo-mer segments will create two layers, one roughly twice the height of the other while undulating under the Poisson sampling process. Furthermore, by the knockout principle the haplotype segments are $O(k)$ or longer (in the event two variant sites are less than k bases apart).

Finally consider the case where there are errors both in the read S under consideration and the set O of all the other reads that were sampled from the same region. When the error is in S, the profile of the k-mers containing the error drops to 1 or nearly so, happens roughly once every 500bp for a HiFi data set with a 0.2% error rate, and the profile count stays very low for $O(k)$ counts by the knockout principle. Errors in the other reads O create -1 drops like a read end event but in this case they last for only $O(k)$ consecutive counts before popping back up $+1$ and these fluctuations are much more frequent, occurring every $0.2c$ bases for haplo segments and $0.4c$ for diplo segments, e.g. every 25bp and every 12.5bp when $c = 20\times$.

In summary, if c is not too small and error is not too high, then the transition of counts in a single profile is caused by a combination of the following four factors in order of their possible effect size: (a) copy number changes, (b) sequencing errors in S, (c) sequencing errors in others O, and (d) read sampling fluctuation. The coherence principle implies that the effect sizes of (c) and (d) are generally much smaller than (a) and (b), and the k-knockout principle can be applied to (a), (b) and (c). However, we will only use the knockout principle for the special case of errors, as changes in the underlying repetitiveness of the genome can negate the length of a haplo-mer, diplo-mer, triplo-mer, $\cdots$ run but not so for an error-mer segment.

2.3 The approach

While we have introduced the fundamental components causing the movements in a count profile $Count(S)$, a full statistical model, i.e. $\Pr\{Seq(S), Count(S), Class(S)\}$, that incorporates all of these stochastic factors is very complicated and thus impractical. Our solution

to this challenge is to divide the classification problem into two heuristic parts: we first resolve local dependencies between k-mer counts due to errors using the k-knockout principle, and then identify haplo-/diplo-profiles using the coherence principle (Fig. 1c).

In Section 3.1, we describe how the "knockout length" of an error is precisely determined based on the sequence context and the type of the error, and present how to compute the probabilities of both errors in S and errors in O for each position. Using these error probabilities, we identify the change-points of k-mer classes in $Class(S)$ when the coherence principle breaks. We call these **walls** and split the profile into a set of contiguous segments partitioned by the walls wherein all the k-mers in an **interval** should belong to the same class. The classification by ClassPro is performed on the intervals (instead of the k-mers), and the inferred class ($\in \mathcal{T}$) of an interval is assigned to all the k-mers in the interval at the end. In Section 3.2 we introduce a criterion for selecting potential haplo-/diplo-intervals (called **reliable intervals**) from all the intervals, and estimate their error-free counts at each boundary wall, i.e. the counts that would occur if there were no errors in O (Fig. 1c, second plot). This partition into reliable intervals with corrected wall counts allows us to accurately approximate the complicated transition among all the k-mer counts in $Count(S)$, by only analyzing the count changes at the walls and between the walls.

In the latter part of the divided problem, we first classify only the reliable intervals (Section 3.4; Fig. 1c, third plot) and then do the rest of the intervals while fixing the classification results of the reliable intervals (Section 3.5; Fig. 1c, forth plot). Since the wall counts are corrected for the reliable intervals, at this point the transition between the wall counts of the reliable intervals should be only due to 1) read sampling fluctuation for those having the same class of H or D (i.e. the coherence principle), or 2) copy number changes for those having different classes. Although the optimal classifications for the reliable intervals can be obtained via dynamic programming (D.P.) if all the reliable intervals are haploid or diploid, repeats and errors cannot be handled in the same manner because of the lack of the coherence principle for them. Nevertheless, we employ a heuristics that combines two pseudo-D.P. sweeps in the forward and backward directions and empirically show that it achieves very accurate classifications over various simulation datasets.

3 Methods

3.1 Wall detection: How errors affect the profile

We term a position i in a profile a *wall* if and only if there is a "significant" change between the two counts c_{i-1} and c_i due to a state change (i.e. $t_{i-1} \neq t_i$). We also call a segment $[b..e)$ partitioned by two adjacent walls at b and e an *interval*. The start of an error state in either S or O causes a count drop and the end does a count gain, and below we describe how to determine walls by finding pairs of count drops and gains due to errors while considering positionally variable sequencing error rates due to low-complexity sequences. Another objective here is to evaluate how likely each interval is a product of errors in S.

Let F be a set of the types of sequence features that alter the sequencing error rate. For HiFi reads, we consider three types of low-complexity sequences: the homopolymers (HP; e.g. `aaaa`), the dinucleotide satellites (DS; `ctctct`), and the trinucleotide satellites (TS; `ctgctg`), denoted by $F = \{\text{HP, DS, TS}\}$. For each $f \in F$, let $\overrightarrow{l}_i^f$ and $\overleftarrow{l}_i^f$ be the maximal length of f on $Seq(S)$ up to position $i-1$ and that from i, respectively. For example, if $s_0 \cdots s_8 = \text{agggctcta}$, then $\overrightarrow{l}_4^{\text{HP}} = 3$ (`ggg`) and $\overleftarrow{l}_4^{\text{DS}} = 4$ (`ctct`). For each feature f, let $err^f(l)$ denote the average indel error rate right after f of length l. We estimate $err^f(l)$ directly from a given dataset with HIsim [20], which efficiently and comprehensively

computes the frequency of each error type using a k-mer count table and a user-specified count threshold between erroneous k-mers and normal k-mers. Since accurate estimation of error rates is difficult for large l due to the relatively small number of observations of such long low-complexity sequences in a dataset, we extrapolate the error rates for $l > l_{\max} = 5$ by fitting a quadratic function $err^f(l) = a_2 l^2 + a_1 l + a_0$ to the average estimated error rates for feature lengths up to $l_{\max}$.

Since the first k-mer affected by the start of a sequencing error in either S or O at position i is α_i (see Section 2.1), a count drop event at i, i.e. $c_{i-1} > c_i$, due to an error should depend on the nucleotide sequence up to $i-1$. Likewise, a count gain at i, i.e. $c_{i-1} < c_i$, due to the end of an error depends on the sequence context from $i-k+1$. Therefore, for each position i the sequence context-dependent error rate ε_i is represented as follows for each type of count change, i.e. drop $\searrow$ and gain $\nearrow$:

$$\varepsilon_i^{\searrow}(f) = err^f\left(\overrightarrow{l}_i^f\right)$$
$$\varepsilon_i^{\nearrow}(f) = err^f\left(\overleftarrow{l}_{i-k+1}^f\right)$$

In other words, $\varepsilon_i^{\searrow}$ and $\varepsilon_i^{\nearrow}$ represent the potential error rate that causes a count drop and a gain, respectively, between $i-1$ and i due to a low-complexity indel error of type $f \in F$. Regarding the other "high-complexity" errors other than F, we use $\bar{\varepsilon} = err^{\mathrm{HP}}(1)$ as the error rate of a single event of insertion, deletion, or substitution for both count drop and gain. We consider up to 5 bases for a single high-complexity error. We denote the set of all the possible error types above by Ω.

For each position i in S, let $\Delta_i \in \{\searrow, \nearrow\}$ denote the direction of the count transition between $i-1$ and i. Given a potential wall at i, let c_{in} and c_{out} be the count just inside and outside of the wall, respectively. That is, $(c_{\mathrm{in}}, c_{\mathrm{out}}) = (\min\{c_{i-1}, c_i\}, \max\{c_{i-1}, c_i\})$, which is equal to (c_i, c_{i-1}) if $\Delta_i = \searrow$ and (c_{i-1}, c_i) if $\Delta_i = \nearrow$. We approximate the "error-free" count (i.e. the count if errors do not exist) of c_{in} by c_{out} for each potential wall. Since the sequencing errors should occur independently among the error-free count, the count change between the two consecutive positions, $i-1$ and i, due to an error in S is modeled by a binomial distribution:

$$c_{\mathrm{in}} \sim \mathrm{Binomial}\left(c_{\mathrm{out}}, \varepsilon_i(\omega)\right)$$

where $\varepsilon_i(\omega)$ is the error rate given the type of the error $\omega \in \Omega$. That is, $\varepsilon_i(\omega) = \varepsilon_i^{\Delta_i}(\omega)$ if $\omega \in F$ (i.e. low-complexity errors) and otherwise $\varepsilon_i(\omega) = \bar{\varepsilon}$ for high-complexity errors. With this model, we can compute how likely a count change occurred by the sequencing errors, or how common it is. We define the probability $p_i^S(\omega)$ that a count change at i is caused due to an error in S whose type is ω by using the p-value of the one-sided binomial test:

$$\begin{aligned} p_i^S(\omega) &= \Pr\left\{X \geq c_{\mathrm{in}} \mid c_{\mathrm{out}}, \varepsilon_i(\omega)\right\} \\ &= \mathrm{BinomialTest}\left(c_{\mathrm{in}} \mid c_{\mathrm{out}}, \varepsilon_i(\omega)\right) \end{aligned}$$

We also define the probability $p_i^O(\omega)$ that a count change at i is caused due to the sequencing errors occurring in a subset of O as follows:

$$\begin{aligned} p_i^O(\omega) &= \Pr\left\{X \geq c_{\mathrm{out}} - c_{\mathrm{in}} \mid c_{\mathrm{out}}, \varepsilon_i(\omega)\right\} \\ &= \mathrm{BinomialTest}\left(c_{\mathrm{out}} - c_{\mathrm{in}} \mid c_{\mathrm{out}}, \varepsilon_i(\omega)\right) \end{aligned}$$

An error event makes a pair of count changes, and thus we wish to define the error probabilities for a pair of walls instead of a single count change. The length of an error-interval generated by a single contiguous error event is not always k bp. In HiFi reads, it is

usually smaller than k bp due to the low-complexity indel errors (Fig. 2a). More precisely, given the location i and the type of an error, the length of an error-interval is exactly given by $k+n-m-1$ bp where m and n are specified as follows. First, $m = \overrightarrow{l}^{f}_{i} > 0$ holds for the low-complexity errors of type f and $m = 0$ for the others, because that the error state in a profile can quickly return to the normal state due to the arbitrariness of low-complexity bases in terms of k-mers. Next, n indicates the number of bases in S that are affected by the error. That is, $n = 0$ holds if the error is a deletion in S or an insertion in O, and otherwise, $n(>0)$ is the number of the bases inserted in S, deleted in O, or substituted in S or O. For low-complexity errors of type f, n is the maximal length of the low-complexity sequence from i, i.e. $\overleftarrow{l}^{f}_{i}$. The essential point here is that for a given error type ω, the length of the error-interval caused by a single error event is uniquely determined. Therefore, for each position i, we calculate the probability that the count changes at both of the pair of i and its corresponding position are due to an error in S as the maximum product of probabilities among all possible error types Ω:

$$p_i^S = \begin{cases} \max_{\omega\in\Omega} \left\{ p_i^S(\omega) \cdot p_{i+\pi}^S(\omega) \;\middle|\; \Delta_{i+\pi} = \nearrow \right\} & \text{if } \Delta_i = \searrow \\ \max_{\omega\in\Omega} \left\{ p_{i-\pi}^S(\omega) \cdot p_i^S(\omega) \;\middle|\; \Delta_{i-\pi} = \searrow \right\} & \text{if } \Delta_i = \nearrow \end{cases}$$

where $\pi = k+n-m-1$. There is only one possible combination of m and n for each of the low-complexity errors and are $\eta+1$ for the high-complexity errors up to η bases (because $m=0$ always holds and $n = 0, \cdots, \eta$). Since $|F| = 3$ and $\eta = 5$ here, the total number of cases inspected per position is always a constant of 9. In the same manner, p_i^O is also computed using $p_i^O(\omega)$.

A set of walls is then determined by finding count changes that i) can be explained by errors in S or ii) cannot be explained by errors in O, that is, $\{\ i\ :\ p_i^S > \theta^S$ or $p_i^O < \theta^O\ \}$, where θ^S and θ^O are user-defined parameters (the default value is 10^{-5} for both). In practice, most of the positions in a HiFi read are not walls, and we do not perform the pairing for any position that already does not satisfy the condition. Note that the pairing of a count drop and gain above is applicable only to a single error event. For a long error-interval containing multiple error events in S, it is generally impossible to determine the exact locations and types of the errors within it. To handle this case, for every pair of two consecutive walls at position i and j that cannot be explained by O, as another possible p_i^S $(= p_j^S)$ we also calculate $p_i^S = \max_\omega \left\{p_i^S(\omega)\right\} \max_\omega \left\{p_j^S(\omega)\right\}$. We do not need to consider the multi-error cases for p_i^O because the same set of multiple errors rarely co-occurs among different reads in O. We take unions of overlapping error-intervals, and for each interval I we define the error probability p_I^{E} as the largest p_i^S among the overlapping error-intervals. The number of resulting walls per read is typically on the order of 10–100 (depending on the heterozygosity and repetitiveness), which is much smaller than N.

3.2 Collecting reliable intervals and correcting the wall counts

We now have a set of non-overlapping intervals split by walls, each of which should be comprised only of a single type of k-mer class in $\mathcal{T}$. While we no longer need to consider the dependencies between k-mers due to errors in S across multiple intervals, the dependencies due to errors in O, i.e. a pair of count drop and gain due to errors in O, still exist across multiple intervals. We resolve this by canceling out the decrease in counts due to errors in O at the walls although not all wall counts can be corrected as described below.

From the set of intervals, we extract "reliable" intervals where we can confidently correct the counts at walls and thus use for the determination of a backbone haplo-profile and diplo-profile in the next step (Fig. 2b). Specifically, an interval $I = [b..e)$ is called *reliable*

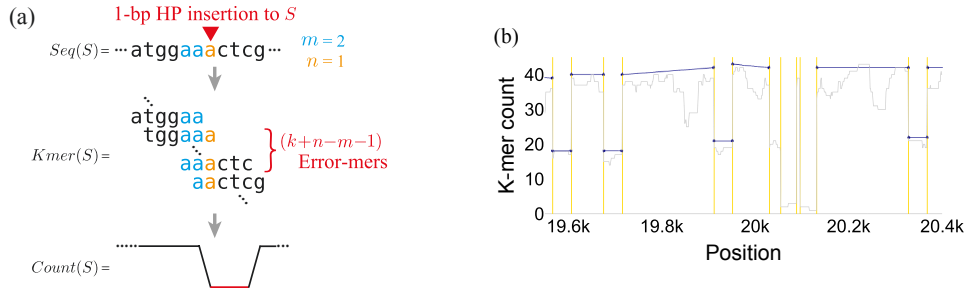


Figure 2 (a) An example of a pair of walls induced by a 1-bp homopolymer insertion event (orange) that occurred in S. Note carefully that the last k-mer, `aactcg`, has a normal count although it contains the inserted base, because of the arbitrariness of the insertion location of low-complexity bases in terms of the k-mer strings. **(b)** Examples of walls (yellow) and reliable intervals (navy) in a small region of a read profile (gray). Each reliable interval is represented and shown by a pair of corrected counts at the walls and a line connecting them.

if and only if i) the error probability p_I^{E} defined above is smaller than the threshold θ^S, ii) the counts at walls are not obviously repetitive, i.e. $\max\{c_b, c_{e-1}\} < \theta^R$, where θ^R is some (loose) threshold for repetitive count, and iii) the length of the interval $e - b$ is at least k. As for the repeat count threshold θ^R, we used the 6σ value of the global diploid coverage while assuming a Poisson distribution of sequencing coverage; that is, $\theta^R = d + 6\sqrt{d}$ where d is the diploid coverage and $\sqrt{d}$ is the standard deviation of Poisson(d). In the last condition we exclude short intervals that can be fully contained within a pair of count drop and gain due to errors in O and thus cannot provide sufficient information for count correction. In contrast, for an interval longer than or equal to k bp, the number of counts decreased at a wall due to errors in O can be estimated because one of the drop-gain pair (i.e. start or end position) of the error state in O that exists across the wall is expected to be contained in the interval in most cases. Another reason for the value of k is because a single SNV results in an interval of length k and this requirement keeps a haplo-interval caused by an SNV as a reliable interval, which avoids over-filtering of intervals.

The counts at both ends of each reliable interval, c_b and c_e, are corrected into $\hat{c}_b$ and $\hat{c}_e$, respectively, using the count changes among $\sim k$ k-mers from b and $\sim k$ k-mers up to e, respectively, based on a logic similar to that in the wall detection:

$$\hat{c}_b \leftarrow c_b + \sum_{i=b+1}^{b+k-1} \max\{c_i - c_{i-1}, 0\} - \sum_{i=b+1}^{b+\max_{f\in F}\left\{\overleftarrow{l}^{f}_{b+k-1}\right\}} \max\{c_{i-1} - c_i, 0\}$$

$$\hat{c}_e \leftarrow c_e + \sum_{i=e-k+1}^{e-1} \max\{c_i - c_{i+1}, 0\} - \sum_{i=e-\max_{f\in F}\left\{\overrightarrow{l}^{f}_{e-k+1}\right\}}^{e-1} \max\{c_{i+1} - c_i, 0\}$$

In both formulae, the first term represents the number of gains/drops within the k bases just after the start position and just before the end position. The second term is the number of gains/drops that actually have a corresponding drop/gain within k bases or less (i.e. drop-gain pairs that are actually contained in the interval) owing to the low-complexity errors.

3.3 Modeling count transition due to the read sampling fluctuation

In addition to sequencing errors, we define a probability of the count change between two positions with some distance due to read sampling fluctuation, i.e. arrivals and exits of the other reads on S. This represents the degree of coherence between k-mers in the same class and is crucial in the next step.

Let u and v ($u < v$) be the locations of two k-mers having the same (non-zero) copy number, i.e. $\#_G(\alpha_u) = \#_G(\alpha_v)(> 0)$. Let $\tilde{c}_u$ and $\tilde{c}_v$ be the error-free counts of α_u and α_v, respectively. For the read length N and the sequencing coverage of the class that the k-mers belong to, i.e. $C = \#_G(\alpha_u) \cdot c$ (c is the global haploid coverage), the distribution of the number of reads starting within a segment $[u..v]$ asymptotically follows Poisson(λ) where $\lambda = (v-u)C/N$ [15]. If we assume that the read departure process is independent from the arrival process, then the distribution of the number of reads ending within $[u..v]$ also follows Poisson(λ). Under the independence, the difference between counts $\tilde{c}_u$ and $\tilde{c}_v$ is modeled by the Skellam distribution [10], which represents the distribution of the difference between two variables independently following a Poisson distribution:

$$\tilde{c}_v - \tilde{c}_u \sim \text{Skellam}\,(\lambda, \lambda)$$

where the shape of the distribution is symmetrical with the mean of 0 given the two variables follow the same Poisson distribution. In practice, this probability is defined when the classes of the two k-mers are deemed identical, i.e. $t_u = t_v$. We thus denote the probability of read sampling fluctuation by $p^{\text{sample}}\,(\tilde{c}_u \to \tilde{c}_v \mid u, v, c^t)$ where c^t is the global coverage of the class $t = t_u (= t_v)$. Using this, we calculate the probability of the transition between haploid intervals and that between diploid intervals, although we cannot use it for repeat intervals because the copy numbers can be different in general between two repeat-mers.

3.4 Classification of the reliable intervals: Finding the backbone haplo- and diplo-profiles

Given a set of reliable intervals, we classify each reliable interval into one of the states $\mathcal{T}$. The main purpose of this step is to detect the backbone haplo- and diplo-profiles using only the corrected counts at the walls for the subsequent classification of the rest of the k-mers (see Fig. 1c). Let $I_i = [b_i..e_i]$ and $T_i \in \mathcal{T}$ denote the i-th interval and its assignment, respectively. Here assigning a specific class to I_i, i.e. $T_i = t'$, means that all the k-mers in I_i are classified as t' in the original profile. The corrected k-mer counts at the two walls, b_i and e_i, of I_i are $\hat{c}_{b_i}$ and $\hat{c}_{e_i}$, respectively.

First, suppose that every reliable interval is either a haplo-interval or a diplo-interval, i.e. $\mathcal{T} = \{\text{H,D}\}$. We assume that for an interval I_i we can estimate the local haploid-coverage and diploid-coverage at e_i given the assignment T_i, and let $\text{cov}[i][s][t]$ be the estimated coverage of class t ($\in \{\text{H,D}\}$) at e_i given $T_i = s$. We also assume that the transition from $T_i = s$ to $T_{i+1} = t$ is determined only by the count transition from $\text{cov}[i][s][t]$ to $\hat{c}_{b_{i+1}}$, i.e. count transition between walls in the sub-profile of class t. Then, (backtracking of) the following dynamic programming using likelihoods of initial classes and transitions gives the classifications of reliable intervals with the maximum likelihood:

$$\begin{aligned} \text{dp}[0][t] &= \Pr\{I_0 \mid T_0 = t\} \\ \text{dp}[i+1][t] &= \max_{s \in \{\text{H,D}\}} \{\text{dp}[i][s] + \Pr\{I_{i+1} \mid T_i = s, T_{i+1} = t\}\} \end{aligned}$$

where

$$\Pr\{I_0 \mid T_0 = t\} = \text{Poisson}\left(\hat{c}_{b_0} \mid c^t\right)$$
$$\Pr\{I_{i+1} \mid T_i = s, T_{i+1} = t\} = p^{\text{sample}}\left(\text{cov}[i][s][t] \to \hat{c}_{b_{i+1}} \mid e_i, b_{i+1}, c^t\right)$$

and c^t is the global coverage of the class t. In practice, $\text{cov}[i][s][t]$ is estimated as follows: i) for $t = s$, then $\hat{c}_{e_i}$ is directly set to $\text{cov}[i][s][t]$, and ii) for $t \neq s$, it is estimated using a linear interpolation using corrected wall counts of the haplo-intervals and diplo-intervals in the best path up to I_i given $T_i = s$.

There actually can exist some repeat-intervals and a small number of error-intervals that are not excluded as unreliable intervals, while the classification categories in the D.P. above cannot be directly extended to $\mathcal{T} = \{\text{E}, \text{H}, \text{D}, \text{R}\}$ because the k-mer counts of different error-intervals and repeat-intervals are generally independent of each other. We thus employ heuristic likelihoods for those intervals as follows. While we cannot use p^{sample} for repeat-intervals, we set the diploid coverage at I_i, $\text{cov}[i][s][\text{D}]$, plus its 2σ (under the assumption of Poisson distribution) as $\text{cov}[i][s][\text{R}]$ and define the likelihood of $T_{i+1} = \text{R}$ given $T_i = s$ as follows:

$$\Pr\{I_{i+1} \mid T_i = s, T_{i+1} = \text{R}\} = \begin{cases} 1 & \text{if } \hat{c}_{b_{i+1}} > \text{cov}[i][s][\text{R}] \\ \text{Binomial}\left(\hat{c}_{b_{i+1}} \mid \text{cov}[i][s][\text{R}], 1 - \bar{\varepsilon}\right) & \text{Otherwise} \end{cases}$$

where $\bar{\varepsilon}$ is the average sequencing error rate. For the probability of $T_{i+1} = \text{E}$ we reuse $p^{\text{E}}_{I_{i+1}}$ that was already computed in Section 3.1.

The classification result can be different in the forward direction and backward direction (where transition from b_{i+1} to e_i is considered instead of transition from e_i to b_{i+1} above) because of the independency of E and R and the estimation of the local coverages. However, we practically obtain accurate classifications by combining the classification result of the pseudo-D.P. in the forward direction and that in the backward direction. Specifically, we find the combined classifications with the maximum likelihood whose prefix is taken from the backward result and suffix is from the forward result, because the coverage estimation tends to become more accurate as the update of the D.P. proceeds.

3.5 Classification of the rest

We finally classify each of the remaining intervals while fixing the assignments of the reliable intervals that are classified as haploid or diploid. Given a focal interval $I = [b..e]$, let $\mathcal{I}_{-I}$ denote the intervals except I, and let $\mathcal{T}_{-I}$ be the assignments of the intervals except T_I, where assignments are initially given to only the reliable intervals. We assign to T_I the class that gives the maximum likelihood computed using the classification results of the other intervals:

$$T_I = \arg\max_t \Pr\{I \mid T_I = t, \mathcal{I}_{-I}, \mathcal{T}_{-I}\}$$

For the case of $I = \text{E}$ we reuse the error probability p^{E}_I, and for the other classes we decompose the probability above into the upstream transition p^+_I (in the direction toward b) and the downstream transition p^-_I (toward e), i.e. $\Pr\{I \mid T_I = t, \mathcal{I}_{-I}, \mathcal{T}_{-I}\} = p^+_I(t) \cdot p^-_I(t)$. For both p^+_I and p^-_I, we consider only the count changes at walls just like the classification of the reliable intervals. As possible events at walls, we consider both read sampling fluctuation and errors in O for $t \in \{\text{H},\text{D}\}$ and only errors in O for $t = \text{R}$. The probability of transition by read sampling fluctuation is calculated between I and the nearest interval J satisfying

$T_I = T_J$ ($\in$ {H,D}) using p^{sample}. The probability by errors in O is computed using the binomial distribution given an estimated coverage and sequencing error rate at a wall of I just like the wall detection (for $t \in$ {H,D}) and the reliable interval classification (for $t =$ R).

3.6 Simulation and real datasets

We adopted two model organisms, fruit fly and human. For the main simulation experiment, we downloaded the latest reference haploid genome sequence (GenBank accession numbers: GCF_000001215.4 [9] for fruit fly and GCA_009914755.3 [21] for human) and a publicly available real HiFi read dataset (BioProject accession numbers: PRJNA573706 for fruit fly and PRJNA586863 [21] for human). For each species we simulated a ground-truth diploid genome sequence from the reference haploid genome sequence using HIsim [20] and then generated synthetic reads using two long-read simulators, Badread [35] and HIsim, both of which build a sequencing error model from a given dataset using short ($\sim$ 10bp) k-mers.

As a baseline of the k-mer classification compared to ClassPro, we performed a histogram-based k-mer classification using GenomeScope with the `-fitted_hist` option [25], where the global thresholds between the four classes (i.e. E,H,D,R) are determined by finding change points of the class that gives the maximum probability according to the GenomeScope inference. We used $k = 40$ unless the value of k is explicitly stated below.

The average overall accuracy of the classifications for a dataset is defined as the number of k-mers with correct classifications divided by the total number of k-mers in the dataset (while regarding multiple k-mers of the same string on different reads or different positions as different k-mers), i.e. $|\{\ \alpha_i \mid t_i = \tau_i\ \}| \ /\ |\{\ \alpha_i \mid \alpha_i \in S, S \in R\ \}|$, and for each combination of parameters we took a harmonic mean of five datasets with different random seeds. In addition, to examine the detailed behavior of the two classification methods, we calculated the average local accuracy of the classifications in each 2Kbp non-overlapping window in the reads as well as the average overall accuracy.

Beyond confirming the expected average behavior, we also explored more practical and realistic cases. First we prepared a 40× simulation dataset of the human major histocompatibility complex (MHC) region by using a publicly available, high-quality diploid assembly of the MHC region [5] as the ground-truth diploid genome and mixing 20× synthetic reads generated from each of the two MHC haplotypes using HIsim. In addition, we downloaded a real 55× HiFi dataset of a diploid human sample HG002/NA24385 (BioProject: PRJNA586863) [37, 21] where an accurate, trio-based diploid assembly (GenBank: GCA_021950905.1 and GCA_021951015.1) [11] is available and can be used as a surrogate of the ground-truth diploid genome, although any missing sequences and false duplicated sequences in the assembly affect the accuracy estimation and thus the "accuracy" should not be perfectly accurate. To try to minimize the effect of the false positive/negative sequences on the accuracy, we ignored a read from accuracy calculation if more than 20% of the k-mers in the read are error-mers or more than 80% are repeat-mers.

4 Results

First we generated HiFi read datasets simulated from a synthetic diploid genome of fruit fly with a small genome size of $\sim$ 160Mbp to deeply investigate ClassPro's performance under various values of sequencing coverage of the reads and heterozygosity of the genome. We used 20×, 25×, 30×, 40×, and 50× as the diploid sequencing coverage, i.e. $2c$. As for the heterozygosity, we specified 0.05%, 0.1%, 0.3%, and 0.5% as the value of the `-p` option of HIsim, which correspond to GenomeScope's estimated heterozygosity of $\sim$ 0.11%, $\sim$ 0.22%, $\sim$

0.66%, and $\sim$ 1.1%, respectively (Fig. S1). We simulated the HiFi reads using two long-read simulators, Badread and HIsim. For each dataset we classified the k-mers in the reads using ClassPro (CP) and GenomeScope (GS; histogram-based method). ClassPro is efficiently parallelized since each read is handled independently, and for example, the degree of the speed up was constantly about 6$\times$ when using 8 threads, regardless of the coverage (Fig. S2). Assuming k is sufficiently small compared to the read length N and both $|\Omega|$ and $|\mathcal{T}|$ (which are always 9 and 4, respectively) are constants, the classification algorithm itself also runs fast in $O(N)$, and thus it takes only $\sim$ 100 seconds wall time to classify a whole 40$\times$ fruit fly dataset using 8 threads on an AMD Epyc 7702 CPU and SSD Lustre system, given a precomputed k-mer count table.

The average overall accuracy of CP was superior than GS in every combination of sequencing coverage and heterozygosity for both Badread datasets and HIsim datasets (Fig. 3, Fig. S3). For example, given a heterozygosity of 0.66% that is close to the estimated heterozygosity value of a real fruit fly dataset [22], in the Badread datasets the overall accuracy of CP exceeds 99.9% (GS=95.1%) when the global coverage per haploid c is 15$\times$ (which is typically called a 30$\times$ dataset) and does 99.99% (GS=96.8%) when $c = 20\times$ (i.e. a 40$\times$ dataset) for normal reads. This indicates that with a typical sequencing coverage ClassPro achieves almost perfect classifications in the most fundamental case where the distinguishment between erroneous, haploid, and diploid k-mers is the only problem. The identification of haploid/diploid k-mers is more difficult for repetitive reads than normal reads without repetitive k-mers in general because the distribution of haploid/diploid k-mers becomes sparse in repetitive regions and thus the number of k-mers that follow the coherence principle decreases. Nevertheless, the overall accuracy of CP for all the reads including repetitive reads (e.g. 99.7% when $c = 20\times$ in the fruit fly dataset above) is still higher than that of GS for only normal reads in every parameter combination. When we calculate the accuracy only with highly repetitive reads in each of which more than 80% of the k-mers are repeat-mers, for example, in a dataset with $c = 20\times$ and 0.22% heterozygosity the accuracy of CP is 98.1% (GS=96.1%), and the false-negative rate of error-mers in such reads is 0.3% (GS=1.0%). This implies that another advantage of CP especially for highly repetitive regions such as centromeres is that we can exclude more false error-mers, which should help singly unique nucleotide k-mers (SUNKs)-based methods (e.g. [3]).

The advantage of CP over GS becomes greater as the heterozygosity gets higher, because the overlap between the distribution of the haploid k-mers and that of the diploid k-mers in the k-mer count histogram (i.e. $\mathcal{H}_{R|1}$ and $\mathcal{H}_{R|2}$ in Introduction) becomes larger in a dataset with a higher genomic diversity. Fig. 3 clearly demonstrates that CP is more robust than GS against haplotype divergence or genome mutation and captures those signals better. We confirmed that CP is robust against the choice of the value of k compared to GS and that the value of k can be either even or odd because we use canonical k-mers in the k-mer counting (Fig. S4).

We further investigated the detailed behavior of the classification in each dataset by calculating the local sequencing coverage and accuracy for every 2Kbp window of the reads in the dataset (Fig. 4). In every parameter combination, the local accuracy of GS drops more quickly than CP when the window coverage deviates from the "sweet spot" that depends on the global coverage c. Thus, the local accuracy of GS classifications can be very low compared to the average overall accuracy not only when the local coverage is considerably lower than c but also when higher. In contrast, CP classifications are consistently better and more robust against changes in coverage than GS especially when the local coverage is higher than c. For example, the window accuracy of CP and GS for normal reads are 99.95%

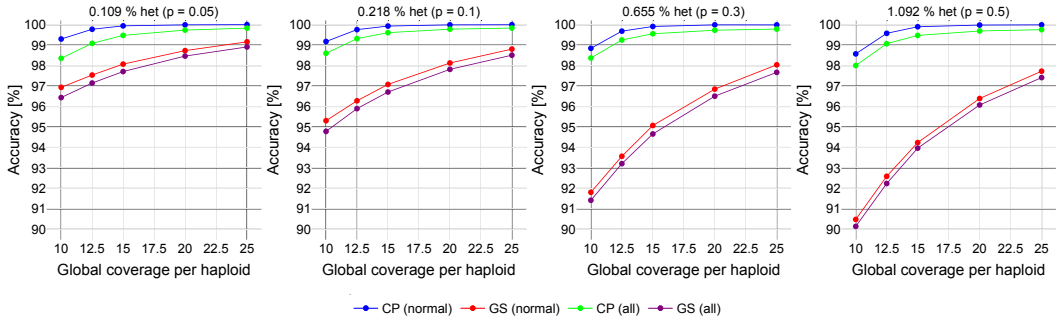


Figure 3 Average overall accuracy between ClassPro and GenomeScope across normal reads without repetitive k-mers and across all reads in the fruit fly simulation datasets by Badread.

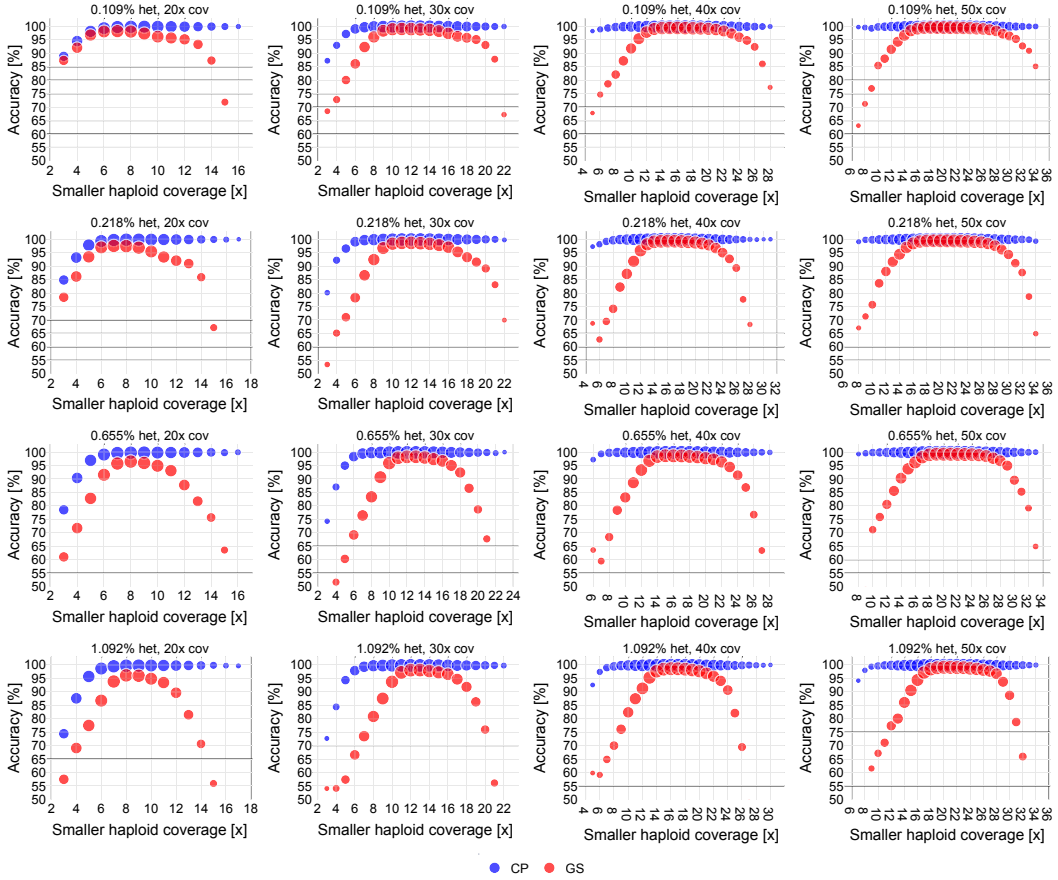


Figure 4 Relationship between local coverage and average accuracy at the resolution of 2Kbp windows across normal reads in the fruit fly simulation datasets. The size of the dots is $\log_2$(# of windows), and only dots of size larger than 100 are drawn. Note that several dots of GS whose accuracy is smaller than 50% are omitted in the plot.

and 25.14%, respectively, when the smaller average haploid coverage is 34× in a dataset with $2c = 50\times$ and $\sim 1.1\%$ heterozygosity. For both CP and GS, the local accuracy with a very small coverage such as 3× becomes worse as the heterozygosity increases because a higher heterozygosity makes the read profile more like a mosaic of haploid k-mers and diploid k-mers and thus requires a more sensitive discrimination between them, which is challenging given 3× per haplotype. The average local accuracy for all reads has the same tendency as normal reads, although there are some fluctuation due to repetitiveness (Fig. S5).

We also evaluated the performance of CP using both simulated and real human HiFi read datasets. We first confirmed that it works with human simulation datasets as well using various sequencing coverages given a typical heterozygosity of $\sim 0.2\%$ (Fig. S6). The accuracy was largely the same as that of the fruit fly dataset with the same heterozygosity: e.g. 99.9% by CP and 97.0% by GS for normal reads in 30× datasets. We then inspected a particular genomic region of interest to researchers. The human MHC region is a highly divergent and repetitive region and thus known to be difficult to accurately assemble [5]. With a simulated 40× dataset generated from a pair of real MHC haplotypes, we confirmed that CP performs well in a difficult-to-assemble region with an accuracy of 99.43% (GS=97.51%). Lastly we applied CP to a real 55× diploid human HiFi dataset while using a high-quality diploid assembly of the same sample as a substitute of the ground-truth genome, and the accuracy of CP was estimated as 99.09% (GS=97.56%). Note carefully that a small amount of remaining missing sequences and false duplicated sequences in the assembly would affect and slightly lower the accuracy estimation.

Note that CP can output different classification results for the same k-mer because it classifies each read independently, while the histogram-based approaches such as GS always classifies the same k-mer into the same class. Nevertheless, by virtue of the high accuracy of CP, the overall consistency of the CP classifications was, for example, over 99.9% in a 40× fruit fly dataset, implying that almost all of the k-mers are consistently classified and the classification results can be used as they are in most applications.

5 Discussion

We developed a novel approach to the k-mer classification problem using k-mer count read profiles, and confirmed that its software implementation, ClassPro, outperforms the conventional, most widely used method based on the k-mer count histogram in every combination of realistic parameter values of sequencing coverage and heterozygosity for two model organisms. The k-mer classification is a fundamental task and used in many sequence analysis programs including error correction, sequence alignment, and genome assembly, and thus the more accurate and robust k-mer classifications by ClassPro promise to help any of such applications boost their performance and accuracy.

The read profiles for ClassPro can be computed by the FastK k-mer counter. It uniquely and very efficiently delivers read profiles as a direct output, whereas with other k-mer counters one is forced to build each profile via a sequence of (relatively more expensive) k-mer table look ups. ClassPro outputs a FastQ-like file of the reads where the QV sequence for each read is replaced with a sequence over the alphabet {E,H,D,R} corresponding to the classification result of the k-mer at each position.

As a demonstration of the power of the improved k-mer classifications by ClassPro, our next target is to incorporate the k-mer classifications into sequence alignment and genome assembly. In the de Bruijn graph approach of genome assembly, ClassPro should offer a better elimination of error-mers for a higher space efficiency and also a more informative guide

for a graph touring. On the other hand, the string graph approach requires the sequence alignments between reads, and the seed selection step is necessary for practical sequence alignment methods. We are currently working on a better seed selection method using the k-mer classification result. Besides that, a more accurate detection and removal of the erroneous k-mers alone would be helpful for trio binning and so on.

Our approach utilizes the positional dependencies between the k-mers. However, it performs the k-mer classification for each read independently. Therefore, it would be natural to think of employing a more complicated data structure capable of handling the k-mer counts of all reads along with their positions simultaneously. The positional de Bruijn graph [28, 31] is apparently the most plausible one of such a representation, although the practical algorithm including the cycle handling due to repeats is not trivial.

The current implementation of ClassPro assumes as input only HiFi reads with an average error rate of $\sim 0.1\%$ (QV30). One direction for future research is to make the method more robust against noisy reads such as Oxford Nanopore reads. Given a sequencing error rate of $\varepsilon\%$, at most $\varepsilon K\%$ bases are expected to be error-mers in each read profile. Therefore, even using the recent Q20+ chemistry with a mean alignment accuracy of QV20 (i.e. 1% error), at most $1\% \times 40 = 40\%$ of the k-mers can be error-mers for a typical Nanopore read given $k = 40$, making a read profile look very erroneous compared to HiFi ($0.1\% \times 40 = 4\%$ error-mers). Moreover, the fluctuation of the haploid and diploid counts by errors in other reads increases as well, i.e. the coherence principle is weakened, making the classification more difficult.

Another possible research target is a utilization of a (not exact but) approximate k-mer counting method for generation and classification of read profiles. Although FastK and ClassPro currently handle only the exact k-mer counting, the whole computation process could be even faster if they can be replaced with an approximate k-mer counting system.

References

1 Anton Bankevich, Andrey V. Bzikadze, Mikhail Kolmogorov, Dmitry Antipov, and Pavel A. Pevzner. Multiplex de Bruijn graphs enable genome assembly from long, high-fidelity reads. *Nature Biotechnology*, 2022. doi:10.1038/s41587-022-01220-6.

2 Jonathan Butler *et al.* ALLPATHS: De novo assembly of whole-genome shotgun microreads. *Genome Research*, 18(5):810–820, 2008.

3 Andrey V. Bzikadze and Pavel A. Pevzner. Automated assembly of centromeres from ultra-long error-prone reads. *Nature Biotechnology*, 38(11):1309–1316, 2020.

4 Haoyu Cheng, Gregory T. Concepcion, Xiaowen Feng, Haowen Zhang, and Heng Li. Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm. *Nature Methods*, 18(2):170–175, 2021.

5 Chen-Shan Chin *et al.* A diploid assembly-based benchmark for variants in the major histocompatibility complex. *Nature Communications*, 11(1):4794, 2020.

6 Giulio Formenti *et al.* Merfin: improved variant filtering and polishing via k-mer validation. *bioRxiv*, 2021. doi:10.1101/2021.07.16.452324.

7 Shilpa Garg. Computational methods for chromosome-scale haplotype reconstruction. *Genome Biology*, 22:101, 2021. doi:10.1186/s13059-021-02328-9.

8 David Heller, Martin Vingron, George Church, Heng Li, and Shilpa Garg. SDip: A novel graph-based approach to haplotype-aware assembly based structural variant calling in targeted segmental duplications sequencing. *bioRxiv*, 2020. doi:10.1101/2020.02.25.964445.

9 Roger A. Hoskins *et al.* The Release 6 reference sequence of the Drosophila melanogaster genome. *Genome Research*, 25(3):445–458, 2015.

10 J. O. Irwin. The frequency distribution of the difference between two independent variates following the same Poisson distribution. *Journal of the Royal Statistical Society*, 100(3):415–416, 1937.

11 Erich D. Jarvis *et al.* Automated assembly of high-quality diploid human reference genomes. *bioRxiv*, 2022. doi:10.1101/2022.03.06.483034.

12 David R. Kelley, Michael C. Schatz, and Steven L. Salzberg. Quake: quality-aware detection and correction of sequencing errors. *Genome Biology*, 11:R116, 2010. doi:10.1186/gb-2010-11-11-r116.

13 Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, May 2017.

14 Sergey Koren *et al.* *De novo* assembly of haplotype-resolved genomes with trio binning. *Nature Biotechnology*, 36(12):1174–1182, 2018.

15 Eric S. Lander and Michael S. Waterman. Genomic mapping by fingerprinting random clones: A mathematical analysis. *Genomics*, 2(3):231–239, 1988.

16 Daniel Mapleson, Gonzalo Garcia Accinelli, George Kettleborough, Jonathan Wright, and Bernardo J Clavijo. KAT: a K-mer analysis toolkit to quality control NGS datasets and genome assemblies. *Bioinformatics*, 33(4):574–576, November 2016.

17 Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

18 Eric Marinier, Daniel G. Brown, and Brendan J. McConkey. Pollux: platform independent error correction of single and mixed genomes. *BMC Bioinformatics*, 16(1):10, 2015.

19 E. W. Myers. FastK. https://github.com/thegenemyers/FASTK, Accessed on 24/06/2022.

20 E. W. Myers. HIsim. https://github.com/thegenemyers/HI.SIM, Accessed on 24/06/2022.

21 Sergey Nurk *et al.* The complete sequence of a human genome. *Science*, 376(6588):44–53, 2022. doi:10.1126/science.abj6987.

22 Sergey Nurk *et al.* HiCanu: accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads. *Genome Research*, 30(9):1291–1305, 2020.

23 Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

24 Nicolas Philippe, Mikaël Salson, Thérèse Commes, and Eric Rivals. CRAC: an integrated approach to the analysis of RNA-seq reads. *Genome Biology*, 14(3):R30, 2013.

25 T. Rhyker Ranallo-Benavidez, Kamil S. Jaron, and Michael C. Schatz. GenomeScope 2.0 and Smudgeplot for reference-free profiling of polyploid genomes. *Nature Communications*, 11(1):1432, 2020.

26 Arang Rhie *et al.* Towards complete and error-free genome assemblies of all vertebrate species. *Nature*, 592(7856):737–746, 2021.

27 Arang Rhie, Brian P. Walenz, Sergey Koren, and Adam M. Phillippy. Merqury: reference-free quality, completeness, and phasing assessment for genome assemblies. *Genome Biology*, 21(1):245, 2020.

28 Roy Ronen, Christina Boucher, Hamidreza Chitsaz, and Pavel Pevzner. SEQuel: improving the accuracy of genome assemblies. *Bioinformatics*, 28(12):i188–i196, 2012.

29 Jared T. Simpson. Exploring genome characteristics and sequence quality without a reference. *Bioinformatics*, 30(9):1228–1235, 2014.

30 The Darwin Tree of Life Project Consortium. Sequence locally, think globally: The Darwin Tree of Life Project. *Proceedings of the National Academy of Sciences*, 119(4):e2115642118, 2022.

31 German Tischler and Eugene W. Myers. Non hybrid long read consensus using local de Bruijn graph assembly. *bioRxiv*, 2017. doi:10.1101/106252.

32 Brian Walenz et al. Meryl. https://github.com/marbl/meryl, Accessed on 24/06/2022.

33 Ting Wang *et al.* The Human Pangenome Project: a global resource to map genomic diversity. *Nature*, 604(7906):437–446, 2022.

34 Aaron M. Wenger *et al.* Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. *Nature Biotechnology*, 37(10):1155–1162, 2019.

35 Ryan R. Wick. Badread: simulation of error-prone long reads. *Journal of Open Source Software*, 4(36):1316, 2019.

36 Xiaohong Zhao *et al.* EDAR: An efficient error detection and removal algorithm for next generation sequencing data. *Journal of Computational Biology*, 17(11):1549–1560, 2010.

37 Justin M. Zook *et al.* Extensive sequencing of seven human genomes to characterize benchmark reference materials. *Scientific Data*, 3(1):160025, 2016.

A Supplementary Figures

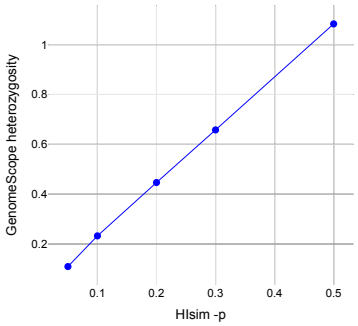


Figure S1 Relationship between HIsim's heterozygosity parameter (`-pX,X` is specified for a value of the x-axis) and the heterozygosity estimated by GenomeScope from the generated fruit fly simulation dataset with 50× coverage.

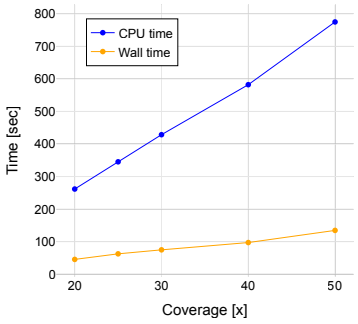


Figure S2 Average computation time of ClassPro for the fruit fly simulation datasets using 8 threads (`-T8` option). Both CPU time (which is the sum of user CPU time and system CPU time) and wall clock time scale linearly with respect to the coverage of the dataset, i.e. the number of k-mers in the dataset, with a consistent speed up of about 6× in wall clock time by the parallelization.

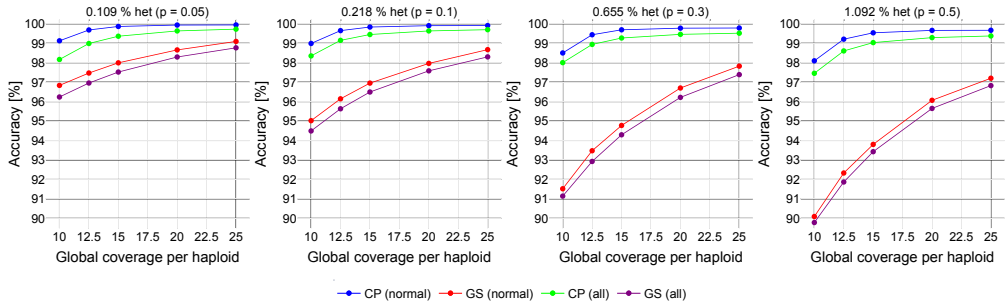


Figure S3 Average overall accuracy between ClassPro and GenomeScope across normal reads without repetitive k-mers and across all reads in the fruit fly simulation datasets by HIsim.

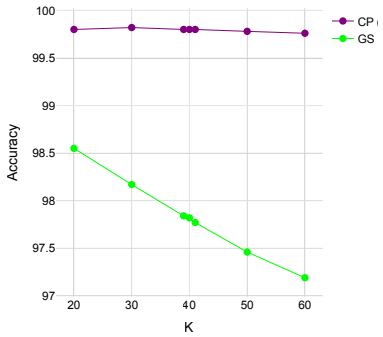


Figure S4 Average overall accuracy with various values of k (=20,30,39,40,41,50,60).

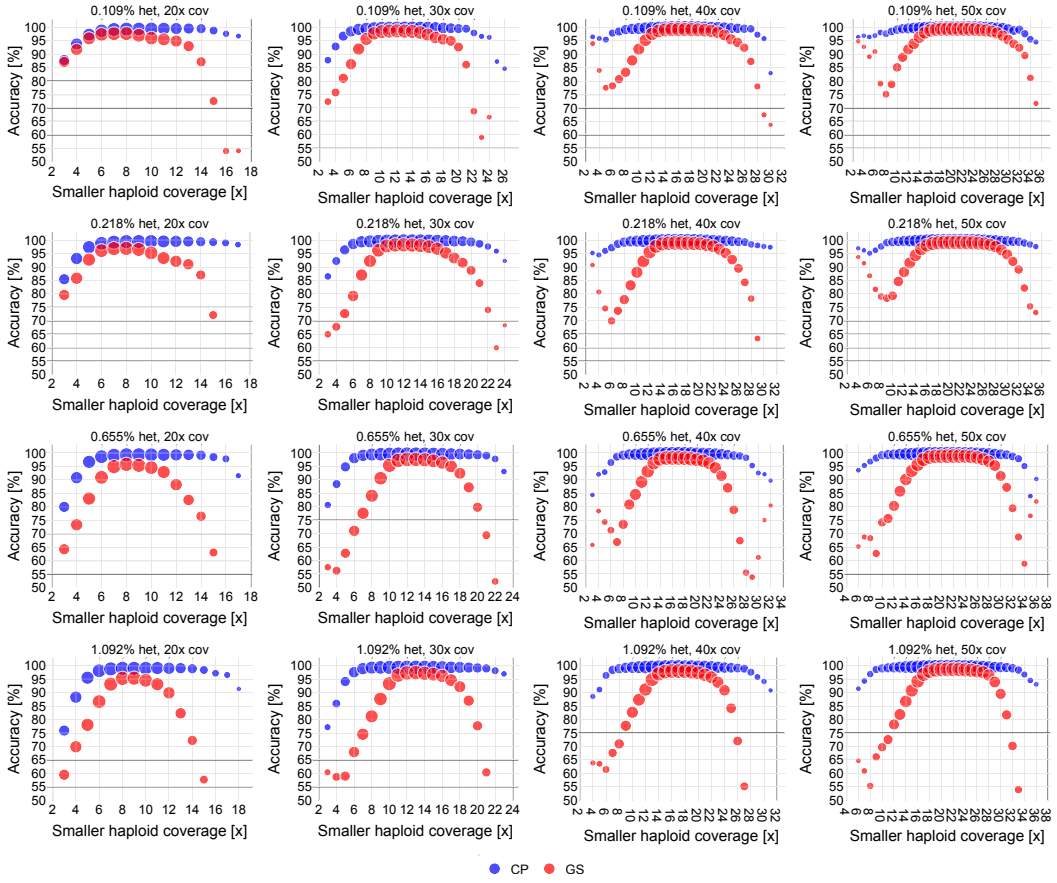


Figure S5 Relationship between local coverage and average accuracy at the resolution of 2Kbp windows across all reads in the fruit fly simulation datasets by Badread.

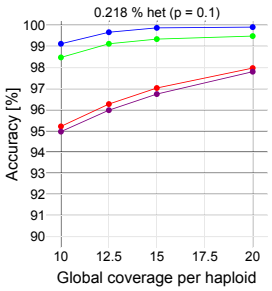


Figure S6 Average overall accuracy between ClassPro and GenomeScope across normal reads without repetitive k-mers and across all reads in the human simulation datasets generated by HIsim using a typical heterozygosity of $\sim 0.2\%$.

New Algorithms for Structure Informed Genome Rearrangement

Eden Ozery
Ben Gurion University of the Negev, Israel

Meirav Zehavi
Ben Gurion University of the Negev, Israel

Michal Ziv-Ukelson
Ben Gurion University of the Negev, Israel

Abstract

We define two new computational problems in the domain of perfect genome rearrangements, and propose three algorithms to solve them. The rearrangement scenarios modeled by the problems consider Reversal and Block Interchange operations, and a PQ-tree is utilized to guide the allowed operations and to compute their weights. In the first problem, Constrained TreeToString Divergence (CTTSD), we define the basic structure-informed rearrangement based divergence measure. Here, we assume that the gene order members of the gene cluster from which the PQ-tree is constructed are permutations. The PQ-tree representing the gene cluster is ordered such that the series of gene IDs spelled by its leaves is equivalent to the reference gene order. Then, a structure-informed gene rearrangement measure is computed between the ordered PQ-tree and the target gene order. The second problem, TreeToString Divergence (TTSD), generalizes CTTSD, where the gene order members are not necessarily permutations and the structure-informed rearrangement based divergence measure is extended to also consider up to d_S and d_T gene insertion and deletion operations, respectively, when modelling the PQ-tree informed divergence process from the reference order to the target order.

The first algorithm solves CTTSD in $O(n\gamma^2 \cdot (m_p \cdot 1.381^\gamma + m_q))$ time and $O(n^2)$ space, where γ is the maximum number of children of a node, n is the length of the string and the number of leaves in the tree, and m_p and m_q are the number of P-nodes and Q-nodes in the tree, respectively. If one of the penalties of CTTSD is 0, then the algorithm runs in $O(nm\gamma^2)$ time and $O(n^2)$ space. The second algorithm solves TTSD in $O(n^2\gamma^2{d_T}^2{d_S}^2m^2(m_p \cdot 5^\gamma\gamma + m_q))$ time and $O(d_Td_Sm(mn + 5^\gamma))$ space, where γ is the maximum number of children of a node, n is the length of the string, m is the number of leaves in the tree, m_p and m_q are the number of P-nodes and Q-nodes in the tree, respectively, and allowing d_T deletions from the tree and d_S deletions from the string. The third algorithm is intended to reduce the space complexity of the second algorithm. It solves a variant of the problem (where one of the penalties of TTSD is 0) in $O(n\gamma^2{d_T}^2{d_S}^2m^2(m_p \cdot 4^\gamma\gamma^2n(d_T + d_S + m + n) + m_q))$ time and $O(\gamma^2nm^2d_Td_S(d_T + d_S + m + n))$ space.

The algorithm is implemented as a software tool, denoted MEM-Rearrange, and applied to the comparative and evolutionary analysis of 59 chromosomal gene clusters extracted from a dataset of 1,487 prokaryotic genomes.

2012 ACM Subject Classification Applied computing → Computational biology

Keywords and phrases PQ-tree, Gene Cluster, Breakpoint Distance

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.11

Supplementary Material The code for our tool, the data used in experiments, and the log file produced by the run of the reported benchmark, can be found on GitHub.
Software (Source Code and Data): `https://github.com/edenozery/MEM-Rearrange`

Funding The research was supported by the Israel Science Foundation (ISF) grants no. 939/18 and 1176/18, and by the Frankel Center for Computer Science at Ben Gurion University.

Acknowledgements Our sincere thanks go to the anonymous WABI 2022 referees who, with their careful reading and incisive comments, helped improve this paper.

1 Introduction

Recent advances in pyrosequencing techniques, combined with global efforts to study infectious diseases, yield huge and rapidly-growing databases of microbial genomes [32, 34]. These big new data statistically empower genomic-context based approaches to functional and evolutionary analysis: the biological principle underlying such analyses is that groups of genes that are located close to each other across many genomes often code for proteins that interact with one another, suggesting a common functional association.

Groups of genes that are co-locally conserved across many genomes are denoted *gene clusters*. The order of the genes in distinct genomic occurrences of a gene cluster may not be conserved. A specific order of the genes of a gene cluster, that is co-linearly conserved across many genomes, is denoted a *gene order* of the gene cluster. The distinct genomes in which a gene order occurs are denoted *instances* of the gene order. Gene clusters in prokaryotic genomes often correspond to (one or several) operons; those are neighbouring genes that constitute a single unit of transcription and translation.

In this paper, our biological goal is to study the evolution of gene clusters in prokaryotes, by computing the divergence between pairs of gene orders that belong to the same gene cluster, based on genome rearrangement scenarios. When defining this computational task as an optimization problem, one needs to take into account that parsimony considerations may not be sufficient: driven by the objective to keep the genome small and efficient, in spite of the high rate of gene shuffling in the prokaryotic genome, only gene orders that are reinforced by conveying some advantage in adaptation will be kept in the genome. This calls for a structure-informed genome rearrangement divergence measure that will interleave parsimony considerations with some learned structural and functional information regarding the gene cluster under study. Such a measure could more accurately assess the degree of divergence from one order of a gene cluster to another, and provide further understanding of gene-context level environmental-specific adaptations [20, 1].

To this end, we propose a new structure-informed rearrangement-based divergence measure and provide efficient algorithms to compute it. According to our approach, information regarding the structure of the gene cluster is learned from the known gene orders of the gene cluster and represented by a PQ-tree (defined in Section 2). The PQ-tree is then utilized to both guide the allowed operations and to compute their weights.

PQ-trees have been advocated as a representation for gene clusters [10, 6, 38]. A PQ-tree describes the possible permutations of a given sequence, and can be constructed in polynomial-time [25]. The PQ-tree representing a given gene cluster describes its hierarchical inner structure and the relations between instances of the gene cluster succinctly, assists in predicting the functional association between the genes in the gene cluster, yields insights into the evolutionary history of the gene cluster, and provides a natural and meaningful way of visualizing complex gene clusters. We refer the reader to Figure 6 (Section 5.2) for two exemplifications of gene clusters and their representative PQ-trees. The figure also illustrates a couple of gene orders per each exemplified gene cluster.

The biological assumptions underlying the representation of gene clusters as PQ-trees is that operons evolve via progressive merging of sub-operons, where the most basic units in this recursive operon assembly are colinearly conserved sub-operons [19]. In the case where an operon is assembled from sub-operons that are colinearly dependent, the conserved gene order could correspond, e.g., to the order in which the transcripts of these genes interact in the metabolic pathway in which they are functionally associated [35]. Thus, rearrangement events that shuffle the order of the genes (or of smaller sub-operons) within this sub-operon could

affect the function of its product. On the other hand, inversion events in which the genes participating in this sub-operon remain colinearly ordered with respect to the transcription order, have less of an affect on the interactions between the sub-operon's gene products.

The case of colinearly conserved sub-operons is represented in the PQ-tree by a Q-node (marked with a rectangle), and by a *Reversal* operation in the corresponding pairwise gene order rearrangement scenario. In the case where an operon is assembled from sub-operons that are not colinearly co-dependent, convergent evolution could yield various orders of the assembled components [19]. This case is represented in the PQ-tree by a P-node (marked with a circle), and by a *Block Interchange* operation in the corresponding pairwise gene order rearrangement scenario.

Background on Structure Informed Rearrangement Scenarios. A generic formulation of genome rearrangement problems is, given two genomes and some allowed edit operations, to transform one genome into the other using a minimum number of edit operations [17, 11, 14, 26]. A famous algorithmic result related to genome rearrangements concerns the problem of sorting signed permutations by Reversals. This problem aims at computing a shortest sequence of Reversals that transforms one signed permutation into another, and can be solved in polynomial time [22, 7, 30]. It was later generalized to handle, still in polynomial time, multichromosomal genomes with linear chromosomes, using rearrangements such as Translocations, chromosome Fusions and Fissions [21, 24]. Then, a general operation called Double Cut-and-Join (DCJ), was introduced in [36] for handling problem instances where the common intervals are organized in a nonlinear structure. A DCJ can be, among others, a Reversal, a Translocation, a Fusion or a Fission, but two consecutive DCJ operations can also simulate a Block-Interchange or a Transposition.

Previous works proposed related forms of structure-informed genome rearrangements by considering rearrangement scenarios on two permutations that preserve their common intervals (groups of co-localized genes). Such scenarios, which may not be shortest among all scenarios, are called perfect [18]. Computing a Reversal scenario of minimum length that preserves a given subset of the common intervals of two signed permutations is NP-hard [18] and several papers have explored this problem, describing families of instances that can be solved in polynomial time [2, 3, 28, 16], fixed parameter tractable algorithms [3, 5], heuristics [8], and an exponential time algorithm (which works also in the more general weighted case) [23]. We note that all the perfect scenarios mentioned above considered only Reversal operations, while for our settings Block-Interchange operations should also be considered.

In [4] the notion of perfect scenario was extended to the Perfect DCJ model, thus capturing additional operations in perfect scenarios, including Cut, Join and Block-Interchange. When considering the perfect rearrangement scenarios that best fit our problem, this is the model that is most relevant to our settings, as the other previous works do not include the Block-Interchange operation. The operations considered by the Perfect DCJ model are very general, which renders the problem computationally intractable in its general setting. Indeed, Berard et al. [4] thus only obtain positive algorithmic results for special cases that, in particular, do not encompass the structure of a PQ-tree, and with a parameter that can often be of the magnitude of the entire input size. For us, the aforementioned special cases are too restricted, and cannot model the problem we have at hand. On the other hand, fortunately, for us, considering Cut and Join operations is an unjustified overhead. Specifically, we seek to model the considered evolutionary scenarios by a formulation that is more specific to our biological problem, in order to increase the divergence measure accuracy as well as tighten the

parameters driving the complexity of the algorithms for the problem. Since, in our problem, we are dealing with prokaryotic gene clusters and the data in our experiment is typically confined to one chromosome per genome, we need not consider Cut and Join operations. In addition, the intervals in prokaryotic gene clusters follow a strongly conserved hierarchy, naturally modeled by the PQ-tree learned from the members of the gene cluster. In terms of divergence measure accuracy, we would like to enforce the PQ-tree structure as a constraint to the considered rearrangements. Furthermore, while the Perfect DCJ model is unweighted (and simply counts the number of DCJ operations applied), we use the PQ-tree as a guide affecting the weights of the applied rearrangement operations. In terms of tightening the parameters driving the complexity of the computation, the PQ-tree constraint enables us to use dynamic programming algorithms and to reduce the parameter from n to the out degree of the tree. In particular, this means that the more hierarchical the input is, the smaller our parameter is likely to be, and the faster our algorithm is – in other words, the running time of our algorithm naturally scales with the amount of structure given by the PQ-tree.

Our Contribution. We propose a new, two-step approach to structure-informed gene order rearrangement: In the first step, given the gene orders of the gene cluster under study, the internal topology properties of a gene cluster are learned from its corresponding gene orders and a PQ-tree is constructed accordingly. Then, in the second step, given a reference gene order and a target gene order, a structure informed rearrangement is computed from the reference to the target, such that colinear dependencies among genes and between sub-operons, as learned by the PQ-tree, are taken into account by the penalties assigned to the rearrangement operations.

To this end, we define two new theoretical problems and propose three algorithms to solve them. In the first problem, denoted Constrained TreeToString Divergence (CTTSD), we define the basic structure-informed rearrangement based divergence measure. Here, we assume that the gene order members of the gene cluster from which the PQ-tree is constructed are permutations. The rearrangement operations considered by this problem include (weighted) Reversals and Block-Interchange operations. In this problem, the PQ-tree representing the gene cluster (Figure 6.A) is ordered such that the series of gene IDs spelled by its leaves is equivalent to the reference gene order (Figure 6.B). Then, a structure-informed, weighted genome rearrangement measure is computed from the ordered PQ-tree to the target gene order (Figure 6.C).

The second problem, denoted TreeToString Divergence (TTSD), is a generalization of the first problem, where the gene order members are not necessarily permutations and the genome rearrangement measure is extended to also consider up to d_S gene insertion operations and up to d_T gene deletion operations.

The first fixed parameter tractable (FPT) algorithm (in Section 3) solves CTTSD in $O(n\gamma^2 \cdot (m_p \cdot 1.381^\gamma + m_q))$ time and $O(n^2)$ space, where γ is the maximum number of children of a node, n is the length of the string and the number of leaves in the tree, and m_p and m_q are the number of P-nodes and Q-nodes in the tree, respectively. If one of the penalties of CTTSD is defined to be 0, then the algorithm runs in $O(nm\gamma^2)$ time and $O(n^2)$ space.

The second FPT algorithm (in the full version) solves TTSD in $O(n^2\gamma^2{d_T}^2{d_S}^2m^2(m_p \cdot 5^\gamma\gamma + m_q))$ time and $O(d_Td_Sm(mn+5^\gamma))$ space, where γ is the maximum number of children of a node, n is the length of the string, m is the number of leaves in the tree, m_p and m_q are the number of P-nodes and Q-nodes in the tree, respectively, and allowing d_T deletions from the tree and d_S deletions from the string.

While our first algorithm is simple and intuitive (based on one dynamic programming and two greedy procedures), for our second algorithm (based on three dynamic programming procedures), more technical ingredients are required. For example, one challenge is the need to compute a vertex cover in a graph that is not fully known by any single entry of our dynamic programming table. Specifically, when we consider a single entry, some of the relevant vertices are not yet processed, and for those that are processed, we cannot store enough information (for the sake of efficiency) so as to deduce which edges exist between them.

The third FPT algorithm (in the full version) is intended to reduce the space complexity of the second algorithm. It solves a variant of the problem (where one of the penalties of TTSD is defined to be 0) in $O(n\gamma^2{d_T}^2{d_S}^2m^2(m_p \cdot 4^\gamma\gamma^2n(d_T + d_S + m + n) + m_q))$ time and $O(\gamma^2nm^2d_Td_S(d_T + d_S + m + n))$ space. This algorithm employs the principle of inclusion-exclusion for the sake of space reduction, which, to the best of our knowledge, is not commonly used in the study of problems in computational biology.

The proposed algorithms are implemented as a software tool, denoted MEM-Rearrange, and applied to the comparative and evolutionary analysis of 59 chromosomal gene clusters extracted from a dataset of 1,487 prokaryotic genomes (in Section 5). Our preliminary results, based on competitive analysis of the 59 gene clusters, indicate that our proposed measure correlates well with an index that is computed by comparing the class composition of the genomic instances of the two compared gene orders. The correlations yielded by our measure show some advantage over the correlations computed for the basic (not structurally informed) Signed Break-point Distance [9], an advantage that grows monotonically with increase in the number of Q-nodes in the PQ-tree. Two of the gene clusters from the benchmark are used to illustrate how our approach can be applied to the study of rearrangement-based adaptations.

Roadmap. The rest of the paper is organized as follows. In Section 2, we formally define the terminology used throughout the paper, and, in particular, the two problems studied in this paper. Some of the more standard definitions are deferred to the full version of this paper. Due to space constraints, our second algorithm, which solves the TTSD problem, is given only in the full version. In Section 3, we present our first algorithm, which solves the CTTSD problem. Due to space constraints, some details – in particular, pseudocode and complexity analysis – are deferred to the full version. Our third algorithm, which reduces the space complexity of our second algorithm to be polynomial in a special case, is also deferred to the full version. In Section 4, we specify the details of our data set construction and experiment. In Section 5, we compare the performance of our proposed rearrangement measure versus that of signed break-point distance on a benchmark of 59 chromosomal gene clusters and provide a detailed exemplification of the difference between the two rearrangement measures on two of the gene clusters from the benchmark.

2 Preliminaries

Let $S = s_1...s_n$ be a string, $S[i] = s_i$, and $S[i : j] = s_i...s_j$.

PQ-Tree: Representing the Pattern. A PQ-tree is a rooted tree with three types of nodes: *P-nodes*, *Q-nodes* and leaves. The children of a P-node can appear in any order, while the children of a Q-node must appear in either left-to-right order or right-to-left order. The possible reordering of the children nodes in a PQ-tree can potentially create many equivalent PQ-trees. Booth and Lueker [10] defined two PQ-trees T and T' as *equivalent* (denoted $T \equiv T'$) if and only if one tree can be obtained by legally reordering the nodes of the other,

as described above. A generalization of their definition, to allow insertions and deletions, is defined as follows. Two PQ-trees T, T' are *quasi-equivalent* with parameter d, denoted by $T \cong_d T'$, if and only if T' can be obtained by (a) randomly permuting the children of the P-nodes of T, (b) reversing the children of the Q-nodes of T, (c) deleting up to d leaves of T. Denote by T_x the subtree of a PQ-tree T rooted in the node x. Denote by $\mathsf{Leaves}(x)$ the set of leaves of T_x, $\mathsf{span}(x) = |\mathsf{Leaves}(x)|$, and for a set of nodes U, $\mathsf{span}(U) = \sum_{v \in U} \mathsf{span}(v)$. Denote by $\mathsf{children}(x)$ the set of children of x, and let $root_T$ be the root node of T.

Given a PQ-tree T, we denote the label of a leaf x by $\mathsf{label}(x)$. The *frontier* T, denoted $F(T)$, is the sequence of labels on the leaves of T read from left to right. In addition, for each node in a PQ-tree (internal node or leaf), we define a unique "color", which will help us distinguish and map between nodes of two quasi-equivalent PQ-trees. Specifically, we use colors to keep track of which operations are performed on a tree when reordering it. From now on, when we say that two PQ-trees T, T' are quasi-equivalent, we assume that the equivalence is with parameter d. In addition, we assume that the PQ-trees T, T' are colored as follows: each color is assigned to one node in T and at most one node in T', and the nodes in T' have the same colors as their corresponding nodes in T. In addition, we say that the *frontier* of T', $F(T')$, is *derived* from T, and we call this a *derivation*. We also say that T' is ordered as $F(T')$. When a string S is derived from T_x, we also say that S is derived from x. Given two quasi-equivalent PQ-trees T, T' and two nodes $x \in T$ and $x' \in T'$, x and x' are *equivalent nodes* if they share the same color.

Break-Point Distances. A *signed string* is a string where each character is assigned a sign ('+' or '-'). Specifically, we suppose to have a function sign that returns the sign of the character in each position in S. A *gene mapping* $\mathcal{M}$ of two strings, $G = g_1, \ldots, g_n$ and $H = h_1, \ldots, h_m$, is a set of pairs $(g_i, h_j) \in G \times H$ such that $g_i = h_j$ and every position in G and H is in exactly one pair in $\mathcal{M}$. Now, we can define the notion of a signed break-point:

▶ **Definition 1** (Signed Break-Point). *Given two signed strings $G = g_1, \ldots, g_n$, $H = h_1, \ldots, h_m$ and a gene mapping $\mathcal{M}$, a* signed break-point *between G and H is a pair of consecutive genes $g_i g_{i+1}$ in G (resp. $h_j h_{j+1}$ in H) such that g_i and g_{i+1} (resp. h_i and h_{i+1}) belong to $\mathcal{M}$, say, $(g_i, h_j), (g_{i+1}, h_k) \in \mathcal{M}$ (resp. $(h_i, g_j), (h_{i+1}, g_k) \in \mathcal{M}$), but neither $k = j + 1$, $\mathsf{sign}_G(i) = \mathsf{sign}_H(j)$ and $\mathsf{sign}_G(i+1) = \mathsf{sign}_H(k)$, nor $k = j - 1$, $\mathsf{sign}_G(i) = -\mathsf{sign}_H(j)$ and $\mathsf{sign}_G(i+1) = -\mathsf{sign}_H(k)$.*

Denote by $\mathsf{NUM_{SBP}}(G, H, \mathcal{M})$ the number of signed break-points of G between G and H with respect to $\mathcal{M}$. The *signed break-point distance* between G and H, denoted by $\mathsf{d_{SBP}}(G, H)$, is the minimum of $\mathsf{NUM_{SBP}}(G, H, \mathcal{M})$ among all gene mappings $\mathcal{M}$. For example, let $S_1 = +a+b+c+d$ and $S_2 = +a-b-d-c$. Then, there exists exactly one gene mapping of S_1 and S_2. The signed break-points of S_1 are the pairs $(a, b), (b, c)$, and $\mathsf{d_{SBP}}(S_1, S_2) = 2$. To accommodate deletions, we will use the notion of signed break-point distance with respect to strings obtained from the given ones after deleting characters.

Problem Preliminaries. Given an internal node x in a PQ-tree T, let $\mathsf{sign}(x)$ be the majority sign of $\mathsf{Leaves}(x)$. If the number of negative signs is equal to the number of positive signs, we abuse notation and consider $\mathsf{sign}(x)$ as $+$ as well as $-$. Given a node x in a PQ-tree, let $S(x)$ denote the signed string of colors of the nodes in $\mathsf{children}(x)$ as they are ordered in the tree (from left to right). For example, consider Figure 1.a where letters represent colors; then, $S(z) = +b+c$, $S(y) = +a+z+d$ and $S(x) = +y+e+f$.

Towards defining the divergence from an (ordered) PQ-tree T to a string S, we define a penalty for taking an action on an internal node of a PQ-tree, denoted by $\Delta_{\mathsf{violation}}$, which is a combination of several types of penalties. The first type concerns large units that "jump" while reordering the children of a P-node x to have the same order as those of its equivalent node x'. If the size of a unit that jumps is t, then we penalize this operation by $(t-1)/2$. In particular, a leaf does not get penalized. To do so, we build a graph $G[x, x']$, whose vertex set is the children of x. Each $c \in \mathsf{children}(x)$ has a weight of $(\mathsf{span}(c) - 1)/2$. This weight represents the penalty for a jump of a large unit; we divide by 2 so that this penalty, even when the unit size is 2, will be smaller than the penalty for a break-point. $G[x, x']$ has an edge for each pair of children that *change their signed order*, defined as follows.

▶ **Definition 2** (Change Of Signed Order). *Let x, x' be two equivalent P-nodes of two quasi-equivalent PQ-trees T and T' with parameter d, $S(x) = c_1, \ldots, c_n$ be the signed string of colors of the nodes in $\mathsf{children}(x)$ that were not deleted, as they ordered in T, $S(x') = c'_1, \ldots, c'_{n'}$ be the signed string of colors of the nodes in $\mathsf{children}(x')$, as they ordered in T', and $\mathcal{M}$ be the gene mapping of $S(x)$ and $S(x')$. Given two nodes y (with color c_i and y' is the equivalent node of y in T') and z (with color c_j and z' is the equivalent node of z in T') in $\mathsf{children}(x)$, say, $j > i$, and such that $(c_i, c'_k), (c_j, c'_t) \in \mathcal{M}$ for some c'_k and c'_t, y and z* change their signed order *if the following are both false: (1) $t > k$, $\mathsf{sign}(y) = \mathsf{sign}(y')$ and $\mathsf{sign}(z) = \mathsf{sign}(z')$ (2) $t < k$, $\mathsf{sign}(y) = -\mathsf{sign}(y')$ and $\mathsf{sign}(z) = -\mathsf{sign}(z')$.*

Let $G = (V, E)$ be a graph with a vertex weight function w. The *minimum weight of a vertex cover* of G is the minimum weight among all vertex covers of G (where a *vertex cover* is a set $S \subseteq V$ such that every edge of G has at least one endpoint in S, and its weight is the sum of weights of its vertices). After building $G[x, x']$, we find the minimum weight of a vertex cover of $G[x, x']$ in order to sum all penalties for the units that jumped while reordering the children of a P-node. Observe that by computing the minimum weight of a vertex cover, we identify a "best" (in terms of penalty) set of nodes that jump. The vertices of the vertex cover itself are considered to be the units that jump.

▶ **Definition 3** ($\Delta^P_{\mathsf{jump}}(x, x')$). *Given two equivalent P-nodes x and x' of two quasi-equivalent PQ-trees with parameter d, and the weight t of a minimum weighted vertex cover of $G[x, x']$, the* jump violation *between x and x', denoted by $\Delta^P_{\mathsf{jump}}(x, x')$, is t.*

Given two equivalent nodes x, x' of two quasi-equivalent PQ-trees T and T' (where x is not deleted), let $\mathsf{isFlipped}(x, x')$ return 1 if x and x' "flipped", and 0 otherwise. That is, if x and x' are leaves, $\mathsf{isFlipped}(x, x') = 1$ if $\mathsf{sign}(x) = -\mathsf{sign}(x')$; otherwise, $\mathsf{isFlipped}(x, x') = 0$. If x and x' are internal nodes, $\mathsf{isFlipped}(x, x') = 1$ if for each child $y \in \mathsf{children}(x)$ that is not deleted, $\mathsf{isFlipped}(y, y') = 1$ where y' is the child of x' equivalent to y, and the order of $\mathsf{children}(x')$ in T' is the reversal of the order of $\mathsf{children}(x)$ in T; otherwise, $\mathsf{isFlipped}(x, x') = 0$. Given an internal node x, denote by $\tilde{\mathcal{S}}_d(x)$ the set of signed strings where $\tilde{S} \in \tilde{\mathcal{S}}_d(x)$ if $\tilde{S}$ is obtained from $S(x)$ by deleting up to d leaves from T_x. Now, we are ready to define the penalty for violation between x and x':

▶ **Definition 4** ($\Delta^d_{\mathsf{violation}}(x, x')$). *Given two equivalent internal nodes x and x' of two quasi-equivalent PQ-trees with parameter d, and input numbers δ^Q_{ord} and δ^Q_{flip}, the* violation *between x and x', denoted by $\Delta^d_{\mathsf{violation}}(x, x')$, is defined as follows.*

- *If x is a P-node, $\Delta^{\mathsf{P}}_{\mathsf{violation}}(x, x') = \min_{\tilde{S} \in \tilde{\mathcal{S}}_d(x)} \mathsf{d}_{\mathsf{SBP}}(\tilde{S}, S(x')) + \Delta^P_{\mathsf{jump}}(x, x')$.*
- *If x is a Q-node, $\Delta^{\mathsf{Q}}_{\mathsf{violation}}(x, x') = \delta^Q_{\mathsf{ord}} \cdot \min_{\tilde{S} \in \tilde{\mathcal{S}}_d(x)} \mathsf{d}_{\mathsf{SBP}}(\tilde{S}, S(x')) + \mathsf{isFlipped}(x, x') \cdot \delta^Q_{\mathsf{flip}}$.*

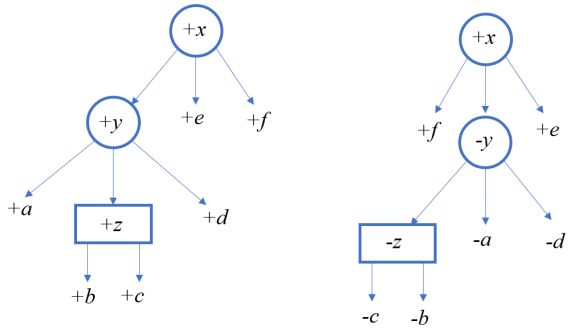


Figure 1 (a) A PQ-tree T. (b) A PQ-tree that is equivalent to T and is ordered as S. Note that a P-node is denoted by a circle and a Q-node is denoted by a rectangle.

When d is clear from the context, we will use the notation $\Delta_{\mathsf{violation}}$ instead. In Definition 4, one of the penalties for reordering the children of a Q-node x is the flip penalty ($\delta^{Q}_{\mathsf{flip}}$), and it is performed if x flipped. Here, notice the case where x flipped and all of its non-deleted children flipped as well (including the P-nodes). Then, we penalise each of the non-deleted children and x as well for flipping, but, the event is flipping only x. Thus, we unnecessarily penalise the non-deleted children. Therefore, we define a "flip correction" procedure:

▶ **Definition 5** ($\mathsf{FlipCorrection}(x, x')$). *Given two equivalent internal nodes x and x' of two quasi-equivalent PQ-trees with parameter d, the* flip correction *between them, denoted by* $\mathsf{FlipCorrection}(x, x')$*, is 0 if not all of* $\mathsf{children}(x)$ *flipped or deleted, and otherwise it is the sum of the flip penalties of all of the Q-nodes and leaves in* $\mathsf{children}(x)$ *that were not deleted.*

Finally, we sum up the violations (and corrections) of all internal nodes to define the divergence from an ordered and labeled PQ-tree to a signed string:

▶ **Definition 6** ($\mathsf{m}_{\mathsf{Diverg}_{d_T,d_S}}(T, S)$). *Let T be an (ordered and leaf-labeled) PQ-tree, and S be a signed string. Let $\mathcal{O}_T$ be the set of all quasi-equivalent T' PQ-trees of T with parameter d_T (i.e. $T \cong_{d_T} T'$) such that T' is ordered as a substring of S obtained by deleting up to d_S characters from S. For any $T' \in \mathcal{O}_T$, let $\mathcal{M}_{T'}$ be the unique*[1] *mapping of each node in T' to its equivalent node in T. Then,*

$$\mathsf{m}_{\mathsf{Diverg}_{d_T,d_S}}(T, S) = \min_{T' \in \mathcal{O}_T} \sum_{(x,x') \in \mathcal{M}_{T'}} (\Delta_{\mathsf{violation}}(x, x') - \mathsf{FlipCorrection}(x, x')).$$

In case $d_T = 0$ and $d_S = 0$, let $\mathsf{m}_{\mathsf{Diverg}}(T, S) = \mathsf{m}_{\mathsf{Diverg}_{d_T,d_S}}(T, S)$.

For example, consider the PQ-tree T in Figure 1, which is ordered as $+a+b+c+d+e+f$, and the signed string $S = +f - c - b - a - d + e$. To order T as S, flip the Q-node z, and pay $\Delta^{\mathsf{Q}}_{\mathsf{violation}}(z, z') = \delta^{Q}_{\mathsf{flip}}$. For the P-node y, swap between a and z, so $\mathsf{d}_{\mathsf{SBP}}(S(y), S(y')) = 1$; hence, $\Delta^{\mathsf{P}}_{\mathsf{violation}}(y, y') = 1$ (there is no jump of a large unit, because the leaf a can jump). Finally, for the P-node x, $\Delta^{P}_{\mathsf{jump}}(x, x') = 0$ (order the children by moving the leaf f to the left-most position), and $\mathsf{d}_{\mathsf{SBP}}(S(x), S(x')) = 1$. Thus, $\Delta^{\mathsf{P}}_{\mathsf{violation}}(x, x') = 1$. In total, $\mathsf{m}_{\mathsf{Diverg}}(T, S) = \Delta^{\mathsf{Q}}_{\mathsf{violation}}(z, z') + \Delta^{\mathsf{P}}_{\mathsf{violation}}(y, y') + \Delta^{\mathsf{P}}_{\mathsf{violation}}(x, x') = \delta^{Q}_{\mathsf{flip}} + 2$.

[1] Note that uniqueness follows from our use of colors.

Problem Definitions

Constrained TreeToString Divergence. The input to CTTSD consists of two signed permutations of length n, $S_1 = \sigma_1 \dots \sigma_n \in \Sigma^n$, $|\Sigma| = n$, such that $\sigma_i \neq \sigma_j$ for all $1 \leq i < j \leq n$, and $S_2 = \lambda_1 \dots \lambda_n \in \Sigma^n$ such that $\lambda_i \neq \lambda_j$ for all $1 \leq i < j \leq n$; a PQ-tree T ordered as S_1 with m_p P-nodes and m_q Q-nodes; and two numbers δ^Q_{ord} and δ^Q_{flip}. The output is $\mathsf{m}_{\mathsf{Diverg}}(T, S_2)$ or "NO" if S_2 cannot be derived from T.

TreeToString Divergence. Generalizing CTTSD, in TTSD we do not assume that the strings are permutations, and we allow deletions. The input to TTSD consists of two signed strings, $S_1 = \sigma_1 \dots \sigma_m \in \Sigma^m_T$ and $S_2 = \lambda_1 \dots \lambda_n \in \Sigma^n_S$; a PQ-tree T ordered as S_1 with m_p P-nodes and m_q Q-nodes; $d_T \in \mathbb{N} \cup \{0\}$, which specifies the number of allowed deletions from T; $d_S \in \mathbb{N} \cup \{0\}$, which specifies the number of allowed deletions from S_2; and two numbers δ^Q_{ord}, δ^Q_{flip} indicating the penalty of the events of changing order and flipping, respectively, a Q-node. The output is $\mathsf{m}_{\mathsf{Diverg}_{d_T,d_S}}(T, S_2)$ or "NO" if S_2 cannot be derived from T with respect to d_T and d_S.

3 Constrained TreeToString Divergence Algorithm

In this section, we present a greedy FPT algorithm to solve CTTSD. Our algorithm consists of three components: the main algorithm, and two procedures called P-Mapping and Q-Mapping. We first present and explain the main algorithm and the procedures. Afterwards, we demonstrate the execution of the algorithm. We remark that, since we consider a string and its reversal (with all signs negated) to be equal, we run the algorithm twice: once on the input string, and once on its reversal.

The Main Algorithm. Recall that the input to CTTSD consists of two signed permutations S_1 and S_2 of length n, two numbers δ^Q_{ord} and δ^Q_{flip}, and a PQ-tree T ordered as S_1. If S_2 can be derived from T, then the output of the algorithm is the divergence from T to S_2, $\mathsf{m}_{\mathsf{Diverg}}(T, S_2)$. Otherwise, the output is "NO" (specifically, the algorithm returns ∞).

The main algorithm constructs a 2-dimensional DP table $\mathcal{A}$ of size $m' \times n$ where $m' = n + m_p + m_q$ is the number of nodes in T. For each node x in T and index ℓ, $\mathcal{A}$ has an entry $\mathcal{A}[x, \ell]$. In the algorithm, for each node x, we keep two indices ℓ and r (denoted by $x.\ell$ and $x.r$ respectively) such that $S_2[x.\ell : x.r]$ is derived from T_x. Then, the purpose of an entry of the DP table, $\mathcal{A}[x, x.\ell]$, is to hold the divergence from the subtree T_x to the substring $S_2[x.\ell : x.r]$ of S_2. That is, $\mathcal{A}[x, x.\ell] = \mathsf{m}_{\mathsf{Diverg}}(T_x, S_2[x.\ell : x.r])$. If any substring of S_2 starting at position ℓ cannot be derived from T_x, then $\mathcal{A}[x, \ell] = \infty$.

Some entries of the DP table define illegal derivations. Such are derivations where the length of the frontier of the subtree is larger than the length of the longest subsrting starting at the specified index ℓ. These entries are called *invalid entries* and their value is defined as ∞ throughout the algorithm. Formally, an entry $\mathcal{A}[x, \ell]$ is invalid if $\mathsf{span}(x) > n - \ell + 1$.

The algorithm first initializes the entries of $\mathcal{A}$ that are meant to hold divergences of derivations of every possible substring of S_2 (a single character) from the leaves of T. Specifically, for a leaf x, if it did not flip, we put 0 in the corresponding entry. If x did flip, we put δ^Q_{flip}. After that, we update $x.\ell$ and $x.r$. As described in the initialization, if $\mathsf{label}(x) = S_2[\ell]$, $S_2[\ell]$ ($S_2[\ell : \ell]$) is derived from $\mathsf{label}(x) = S_2[\ell]$, then we put 0 or δ^Q_{flip} in $\mathcal{A}[x, \ell]$ and we put ℓ in $x.\ell$ and $x.r$.

After the initialization, all other entries of $\mathcal{A}$ are filled as follows. Go over the internal nodes of T in postorder. For every internal node x, go in descending order over every index $1 \leq \ell \leq n$ that can be a start index for the substring of S_2 derived from T_x (in case of invalid

entry, we continue to the next iteration). For every x and ℓ, use the algorithm for P-mapping or Q-mapping according to the type of x. Both algorithms receive the following input: the node x, S_2, start and end indices ℓ, e of a substring of S_2, and the collection of derivations of the children of x (entries of $\mathcal{A}$ that have already been computed and hold the divergence of a derivation). In addition, the Q-Mapping algorithm receives as input the penalty parameters δ^Q_{ord} and δ^Q_{flip}. After being called, both algorithms return the divergence from T_x to $S_2[\ell : e]$, that is, $\mathsf{m}_{\mathsf{Diverg}}(T_x, S_2[\ell : e])$.

Finally, having filled the DP table, $\mathcal{A}[root_T, 1]$ holds the divergence from T to S_2 ($\mathsf{m}_{\mathsf{Diverg}}(T, S_2)$), and so we return $\mathcal{A}[root_T, 1]$.

P-Node Mapping: The Algorithm. Recall that the input consists of a P-node x, a string S_2, two indices ℓ and e, and a set of derivations $\mathcal{D}$. Notice that each value in $\mathcal{D}$ is the divergence from the subtree rooted in a child c of x to $S_2[c.\ell : c.e]$, where $S_2[c.\ell : c.e]$ is a substring of $S_2[\ell : e]$ that is derived from T_c. These values, $\mathcal{A}[c, c.\ell]$ for each $c \in \mathsf{children}(x)$, were calculated in earlier iterations and saved in $\mathcal{D}$. If $S_2[\ell : e]$ can be derived from T_x, then the output of the algorithm is the divergence from T_x to $S_2[\ell : e]$, $\mathsf{m}_{\mathsf{Diverg}}(T_x, S_2[\ell : e])$. Otherwise, the output is "NO" (specifically, the algorithm returns ∞). Denote by T'_x the quasi-equivalent PQ-tree of T_x ordered as $S_2[\ell : e]$. Note that if T'_x exists, then it is unique (because we deal with permutations and forbid deletions).

The algorithm first checks if the interval $[\ell, e]$ can be "completed" by all of the intervals defined by the indices of the children of x. Specifically, we check if there is any order of the children of x, say, ordered as $c_1, \ldots, c_{|\mathsf{children}(x)|} \in \mathsf{children}(x)$, such that $c_1.\ell = \ell$, $c_{|\mathsf{children}(x)|}.e = e$, and for each $1 \leq j \leq |\mathsf{children}(x)| - 1$, $c_j.e + 1 = c_{j+1}.\ell$. If there is no such order, then the interval $[\ell, e]$ cannot be completed, and so $S_2[\ell, e]$ cannot be derived from T_x. In this case, we return ∞. Otherwise, $S_2[\ell : e]$ can be derived from T_x by reordering the children of x according to the unique order that completes the interval $[\ell, e]$. Second, we sum up all of the values in $\mathcal{D}$ (and store the sum in the variable *childrenDist*). Next, we calculate the violation between x and its equivalent node x' in T'_x, $\Delta^{\mathsf{P}}_{\mathsf{violation}}(x, x')$, according to Definition 4 (and store the result in the variable *violation*). Finally, we return the sum of *childrenDist* and *violation*, which is the divergence from T_x to $S_2[\ell : e]$, $\mathsf{m}_{\mathsf{Diverg}}(x, S_2[\ell : e])$ (according to Definition 6).

Q-Node Mapping: The Algorithm. Recall that the input consists of a Q-node x, a string S_2, two indices ℓ and e, a set of derivations $\mathcal{D}$ and penalty parameters δ^Q_{ord} and δ^Q_{flip}. Notice that each value in $\mathcal{D}$ is the divergence from the subtree rooted in a child c of x to $S_2[c.\ell : c.e]$, where $S_2[c.\ell : c.e]$ is a substring of $S_2[\ell : e]$ and is derived from T_c. These values, $\mathcal{A}[c, c.\ell]$ for each $c \in \mathsf{children}(x)$, were calculated in earlier iterations and saved in $\mathcal{D}$. If $S_2[\ell : e]$ can be derived from T_x, then the output of the algorithm is the divergence from T_x to $S_2[\ell : e]$, $\mathsf{m}_{\mathsf{Diverg}}(T_x, S_2[\ell : e])$. Otherwise, the output is "NO" (specifically, the algorithm returns ∞). Denote by T'_x the (unique) quasi-equivalent PQ-tree of T_x ordered as $S_2[\ell : e]$.

The algorithm first checks if the interval $[\ell, e]$ can be "completed consecutively" by all of the intervals defined by the indices of the children of x. Specifically, we check if there is a consecutive order of the children of x, say, ordered as $c_1, \ldots, c_{|\mathsf{children}(x)|} \in \mathsf{children}(x)$, such that $c_1.\ell = \ell$, $c_{|\mathsf{children}(x)|}.e = e$, and for each $1 \leq j \leq |\mathsf{children}(x)| - 1$, $c_j.e + 1 = c_{j+1}.\ell$. As apposed to a P-node, here the order of the children completing the interval $[\ell, e]$ must be consecutive with respect to their indices (the same order as the children of x in T or the reverse order). If there is no such order, then the interval $[\ell, e]$ cannot be completed consecutively, and so $S_2[\ell : e]$ cannot be derived from T_x. In this case, we return ∞. Otherwise, $S_2[\ell : e]$ can

be derived from T_x by keeping the order of the children of x, or flipping it. Second, we sum up all of the values in $\mathcal{D}$ (and store the sum in the variable *childrenDist*). Next, we calculate the violation between x and its equivalent node x' in T'_x, $\Delta^{\mathsf{Q}}_{\mathsf{violation}}(x, x')$, according to Definition 4 (and store the result in the variable *violation*). Afterwards, we calculate the flip correction according to Definition 5 (and store the result in the variable *childrenFlipCorrection*). Finally, we return the sum of *childrenDist* and *violation* minus *childrenFlipCorrection*, which is the divergence from T_x to $S_2[\ell : e]$, $\mathsf{m}_{\mathsf{Diverg}}(x, S_2[\ell : e])$ (according to Definition 6).

Example. Consider the following input: $S_1 = +a+b+c+d+e+f$, $S_2 = +f-c-b-a-d-e$, PQ-tree T ordered as S_1, $\delta^{Q}_{\mathsf{ord}} = 3$ and $\delta^{Q}_{\mathsf{flip}} = 3$. We iterate through the nodes of the tree in post-order, thus we initiate the leaves before their parents. For each leaf $x \in \mathsf{Leaves}(root_T)$, if $\mathsf{label}(x) = S_2[\ell]$, then $\mathcal{A}(x, \ell) = 0$; otherwise, $\mathcal{A}(x, \ell) = \infty$.

Figure 2 describes the PQ-tree T and its quasi-equivalent PQ-tree T' ordered as S_2, after the initialization of the leaves only. Notice that the order of the initialization is in fact different, in postorder; for simplicity, we show the tree where only the leaves are initialized. In addition, in the figures, the equivalent nodes of T and T' are shown as the same nodes. But in the explanations, in order to distinguish between them, for each node $x \in T'$, we denote it by x'. The pair of numbers shown in the figure near a node represent its ℓ and r values. In addition, the sign $+$ or $-$ near a node represents its sign. For example, for $a \in T$, $a.\ell = a.r = 4$ and $\mathsf{sign}(a) = +$. For each internal node, the character assigned to it represents its color.

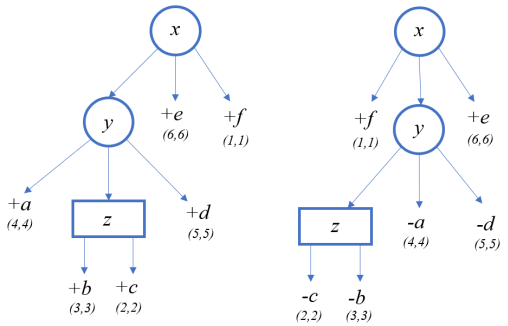


Figure 2 **(a)** PQ-tree T. **(b)** PQ-tree T', which is quasi-equivalent to T and ordered as S_2.

First, consider the iteration where $\mathcal{A}[z, 2]$ is calculated (the values for all other entries with node z is ∞). The intervals of the children of z, $[3, 3]$ and $[2, 2]$, complete the interval $(2, e)$ consecutively where $e = 2+\mathsf{span}(z)-1 = 3$. Thus, the substring $S_2[2 : 3]$ is derived from T_z, and in order to generate it we need to flip the order of $\mathsf{children}(z)$. For this the penalty of flipping, $\delta^{Q}_{\mathsf{flip}}$, is applied, and so $\Delta^{\mathsf{Q}}_{\mathsf{violation}}(z, z') = \delta^{Q}_{\mathsf{flip}} = 3$. $childrenDist = \mathcal{A}[b, 3]+\mathcal{A}[c, 2] = 6$ because $\mathcal{A}[b, 3] = \mathcal{A}[c, 2] = 3$. $\mathsf{FlipCorrection}(z, z') = 6$ because both b and c have been flipped. Therefore, $\mathcal{A}[z, 2] = childrenDist+violation-childrenFlipCorrection = 3$. Now, we update z's indices, $z.\ell = 2$ and $z.r = 3$. Figure 3 describes the PQ-tree T and its quasi-equivalent PQ-tree T' after calculating $\mathcal{A}[z, 2]$.

Next, consider the iteration where $\mathcal{A}[y, 2]$ is calculated (the values for all other entries with node y is ∞). Recall that Figure 3 describes T and T' after calculating the entries of the children of the P-node y. The intervals of the children complete the interval $(2, e)$,

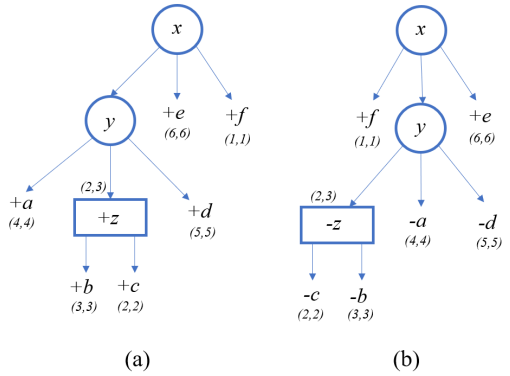


Figure 3 **(a)** PQ-tree T. **(b)** PQ-tree T', which is quasi-equivalent to T and ordered as S_2.

where $e = 2 + \mathsf{span}(y) - 1 = 5$, so we can continue with the iteration. $childrenDist = \mathcal{A}[a,4] + \mathcal{A}[z,2] + \mathcal{A}[d,5] = 9$. $\mathsf{d_{SBP}}(S(y), S(y')) = 1$ where the pair (z,d) is the only singed break-point (note that $S(y) = +a + z + d$, $S(y') = -z - a - d$). $\Delta^P_{\mathsf{jump}}(y,y') = 0$ because to reorder the children of y as $S_2[2:5]$ we can move z only (and its size is 1). Therefore, $\Delta^{\mathsf{P}}_{\mathsf{violation}}(y,y') = \mathsf{d_{SBP}}(S(y), S(y')) + \Delta^P_{\mathsf{jump}}(y,y') = 1$. Then, $\mathcal{A}[y,2] = 10$ and we can update y's indices, $y.\ell = 2$ and $y.r = 5$. Figure 4 describes the PQ-tree T and its quasi-equivalent PQ-tree T' after calculating $\mathcal{A}[y,2]$.

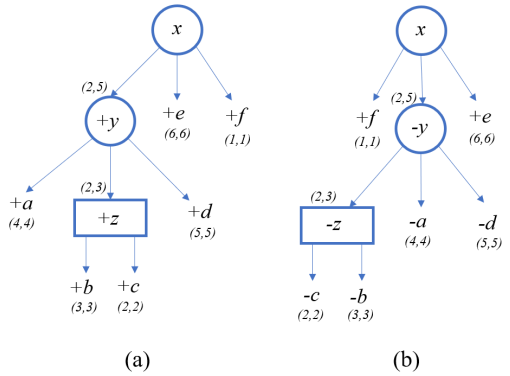


Figure 4 **(a)** PQ-tree T. **(b)** PQ-tree T', which is quasi-equivalent to T and ordered as S_2.

Finally, consider the iteration where $\mathcal{A}[x,1]$ is calculated (the values for all other entries with node x is ∞). Figure 4 describes T and T' after calculating the entries of the children of the P-node x. The intervals of the children complete the interval $(1,e)$, where $e = 1 + \mathsf{span}(x) - 1 = 6$, so we can continue with the iteration. $childrenDist = \mathcal{A}[y,2] + \mathcal{A}[e,6] + \mathcal{A}[f,1] = 10$. $\mathsf{d_{SBP}}(S(x), S(x')) = 2$ where the pairs (y,e) and (e,f) are the signed break-points (note that $S(x) = +y + e + f$, $S(x') = +f - y + e$). $\Delta^P_{\mathsf{jump}}(x,x') = 0$ because to reorder the children of x as S_2 we can move f only (and its size is 1). Therefore, $\Delta^{\mathsf{P}}_{\mathsf{violation}}(x,x') = \mathsf{d_{SBP}}(S(x), S(x')) + \Delta^P_{\mathsf{jump}}(x,x') = 2$. Then, $\mathcal{A}[x,1] = 10 + 2 = 12$ and the algorithm returns 12. Figure 5 describes the PQ-tree T and its quasi-equivalent PQ-tree T' after calculating $\mathcal{A}[x,1]$.

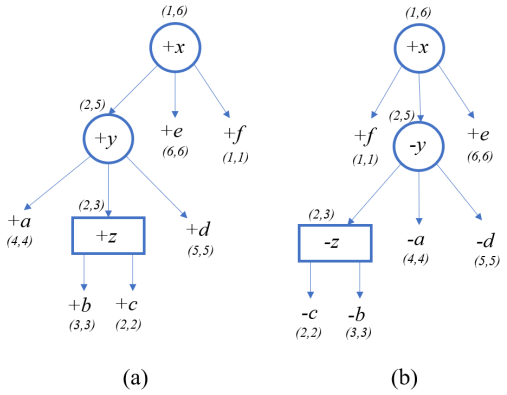


Figure 5 **(a)** PQ-tree T. **(b)** PQ-tree T', which is quasi-equivalent to T and ordered as S_2.

4 Methods and Datasets

Dataset and Gene Cluster Generation. 1,487 fully sequenced prokaryotic strains with COG ID annotations were downloaded from GenBank (NCBI; ver 10/2012). The gene clusters were generated from this data using the tool CSBFinder-S [29]. CSBFinder-S was applied to all the genomes in the dataset after removing their plasmids, using parameters $q = 10$ (a colinear gene cluster is required to appear in at least ten genomes) and $k = 0$ (no insertions are allowed in a colinear gene cluster), resulting in 79,017 colinear gene orders. From these gene orders, only gene orders whose number of distinct COGs is between 4 and 9 were kept, leaving 28,537 gene orders. Next, ignoring strand and gene order information, colinear gene orders that contain the exact same COGs were united to form the generalized set of 91 gene clusters that abide by the requirements that each gene cluster contains at least 3 gene orders and each COG appears only once in each gene order. For each gene cluster, the most abundant gene order was designated as the "reference" (centroid) gene order. Based on this, the clusters were further filtered to keep only 63 gene clusters whose designated reference has instances in at least 30 genomes. Finally, clusters containing one or more gene orders that are identical to the designated reference gene order, in terms of the list of classes in which they have instances, were removed, leaving a benchmark set of 59 gene clusters.

PQ-tree construction. The input PQ-trees for our algorithm where constructed using the tool PQFinder (available on GitHub [37]). PQFinder was applied to each of the gene clusters in the dataset, to build the PQ-tree representing each cluster. In addition, each Q-node with exactly two children, whose height in the tree is greater than 1, was changed to a P-node (in this special case all children of the node were observed in all shuffling options, which in our opinion better fits the syntax of a P-node than that of a Q-node.)

Parameter Settings. In our experiment, we set the parameters of the algorithm as follows. $\delta^Q_{\mathsf{ord}} = 1.5$, $\delta^Q_{\mathsf{flip}} = 0.5$, $d_T = 0$, $d_S = 0$.

5 Results

5.1 Evaluation

In this section we evaluate the accuracy of our approach in measuring the evolutionary divergence between two gene orders that belong to the same gene cluster. To this end, we aim to generate a set of "control" distances, computed from real data, against which the divergence scores computed by our tool can be compared and evaluated.

Recall that in our application, each of the input sequences does not correspond to a specific genomic sequence but rather represents a gene order that occurs in multiple genomes. Furthermore, abundant gene clusters typically display several paralogous occurrences of distinct gene orders, and possibly several paralogous occurrences of a specific gene order, within the same genome. Furthermore, each distinct occurrence of a specific gene order could differ substantially from another occurrence of the same gene order in terms of its encoding genomic sequence, since each COG represents a cluster of genomic sequences that are not identical however are similar enough (possibly based on local sequence similarity) to be clustered to the same orthology group.

Thus, we chose to represent each gene order by the assemblage of its instances, i.e. the set of genomes in which it occurs. Several similarity (or overlap) indices based on presence/absence (incidence) data have been proposed in the literature [27, 12]. A classical and widely used index in comparative assemblage analysis is the Jaccard index [12]. In our comparative evaluation the instance assemblages are used to estimate divergence rather than similarity, and therefore we use the inverse Jaccard Index as an estimator of the instance assemblage based divergence between two gene orders.

Our divergene measure was evaluated, per each cluster, as follows: first, we applied our approach (Alg. 1) to measure the structure informed divergence from the cluster's designated reference (explained in Section 4) to each of the other gene orders. Then, we calculated the Inverse Jaccard based distance from the set of instances of the reference gene order to the sets of instances of each of the other gene orders. In order to tolerate the noise due to inter-specie and inter-genus horizontal transfer of gene orders, we first converted the assemblages of genomes to the assemblages of (taxonomic) classes to which these genomes belong. This resulted in two series of scores, which were then subjected to the computation of (weighted average) Spearman and Pearson correlations between them. The same evaluation procedure was repeated per each cluster, this time using the signed break-point distance instead of our structure-informed divergence measure.

The results are given in Table 1. The 59 gene clusters were distributed to three groups in increasing level of colinear component dependency, according to the number of Q-nodes in their representative PQ-trees. This yielded 8 gene clusters whose representative tree has no Q-nodes, 41 gene clusters whose representative tree has one Q-node, and 10 gene clusters whose representative tree has two or more Q-nodes. For each group, the average Spearman and Pearson correlation scores were computed for both divergence measures. The average score for each group, per each measure, was computed as a weighted arithmetic mean, normalized by the size of each cluster in the group.

Table 1 A comparison between our proposed rearrangement measure ($\mathsf{m}_{\mathsf{Diverg}}$) and signed break-point distance ($\mathsf{d}_{\mathsf{SBP}}$), based on their correlation to a taxonomical instance abundance measure.

num of Q-nodes	Correlation	$\mathsf{m}_{\mathsf{Diverg}}$	$\mathsf{d}_{\mathsf{SBP}}$
0	Pearson	0.759	0.761
	Spearman	0.666	0.665
1	Pearson	0.861	0.844
	Spearman	0.745	0.716
2	Pearson	0.924	0.844
	Spearman	0.898	0.823

Table 1 indicates that, in general, the rearrangement-based divergence between a reference gene order and its target gene order correlates well with the divergence in the taxonomic distribution of their corresponding instances, which is an interesting result on its own.

For our proposed measure $\mathsf{m}_{\mathsf{Diverg}}$, the correlation increases with the number of Q-nodes in the representative trees. The advantage obtained by applying our structure-informed rearrangement approach, in comparison to signed break-point distance, also increases with the number of Q-nodes in the PQ-trees representing the gene clusters.

5.2 Examples

ABC transporters are among the most intensively studied gene clusters, as they form the largest group of paralogous genes in bacterial and archaeal genomes [31]. ABC transporters tend to form operons, and several studies indicate that the ancestral ABC transporter operons may have arisen early in evolution, before the speciation of bacteria and archaea [33]. We exemplify the application of the proposed structure-informed rearrangement approach via two ABC transporter gene clusters from our benchmark: a gene cluster encoding for a Dpp dipeptide uptake system, and another gene cluster encoding for a Ugp sugar uptake system. In both examples, the correlation between the $\mathsf{m}_{\mathsf{Diverg}}$ series for the cluster and the corresponding series of taxonomic instance abundance scores, was higher than that obtained for the $\mathsf{d}_{\mathsf{SBP}}$ measure.

The examples are illustrated in Fig. 6. Due to space constraints, we analyze only one rearrangement scenario per example. Additional information can be found in our GitHub directory, where we publish the extensive log information for each of the 59 gene clusters analyzed in our experiment.

Example 1: A gene cluster encoding a dipeptide uptake system, Figure 6.A-E. The Dpp transporter encoded by this gene cluster is composed of two integral membrane proteins (heterodimer TMDs), two peripheral membrane proteins that bind and hydrolyze ATP (heterodimer NBPs), and a periplasmic (or lipoprotein) substrate-binding protein (SBP) that delivers the substrate to a core importer (Figure 6.H). TMDs and NBDs dimerize and assemble the minimal unit of an importer, with the SBP as the fifth component of the complex.

The signed break-point distance for the reference and target gene orders is 2, based on the break-points (1,2) and (3,4). However, there is no experimental evidence for any special, asymmetric interactions between the products of genes 1 (SBP) and 2 (TMD1). On the contrary, it is well-known that the two lobes of the SBP interact with both of the TMDs [15] after a substrate has been bound by the SBP. Similary, there is no biological basis for assuming an assymmetric interaction between the products of the two genes 3 (TMD2) and 4 (NBP1) that would require direct positional adjacency.

Previous studies indicate that the importer complex is composed of two heterodimer components: NBP1-NBP2, and TMD1-TMD2 [15]. However, this structural information is not taken into account by the signed break-point distance computation. Thus, the break-point distance seems to poorly model the gene order divergence from the reference, ancestral gene order of this gene cluster to the target, rearranged copy.

Interestingly, when considering the conserved structure of the gene orders in this family (represented by the PQ-tree in Figure 6.A), the pattern by which the second gene order diverges from the first gene order becomes evident: the first gene order spells an "outside-in" transcription (and possibly assembly) order of the components of the complex (Figure 6.D), while the second gene order spells an "inside-out" transcription (and possibly assembly) order. This observation is better reflected by the $\mathsf{m}_{\mathsf{Diverg}}$ score for this gene order pair, which is 2.5. This score is obtained by assigning a cost of 1 for a break-point between two Q-nodes

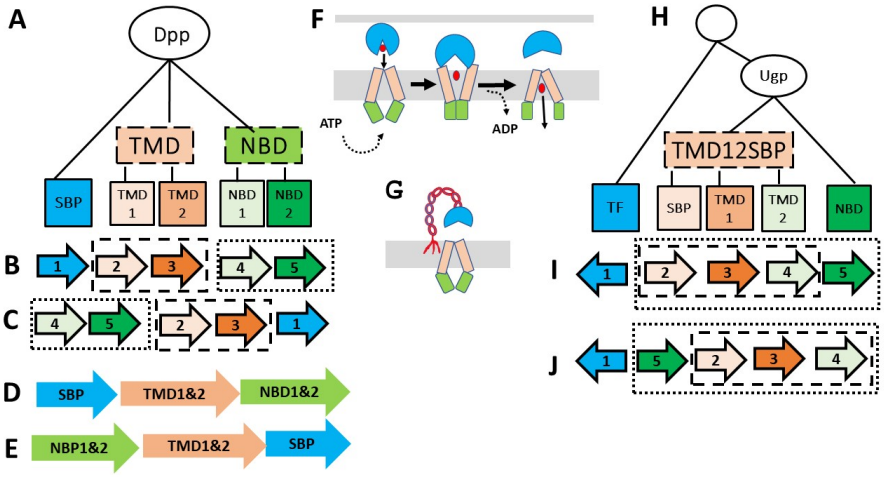


Figure 6 Examples from two ABC transporter gene clusters. **A,B,C.** Reference and target gene orders from the dipeptide uptake gene cluster DppBCD-GsiA-DdpA, and the representative PQ-tree for the corresponding gene cluster. **D,E.** Structure-informed encoding of the gene orders from B and C, respectively, reveals that the transcription order of their structural sub-components has been reversed by the re-arrangement. **F,G.** Substrate binding and import in ABC transporter in gram negative bacteria versus gram positive bacteria, respectively. **H,I,J.** Reference and target gene orders from the sugar uptake system UgpABE-MalK with the corresponding transcription factor PurR, and the representative PQ-tree for the corresponding gene cluster.

(representing the heterodimers TMD and NBD), increased by an additional penalty of 0.5 for a "jump" of a component of size 2, plus a cost of 1 for a break-point between the SBP gene and the Q-node representing TMD12. Thus, our structure-informed rearrangement approach is able to utilize the PQ-tree to capture the order reversal of the dimer-scale components of the encoded uptake machine.

When considering the taxonomical distribution of these two gene orders, we note that the reference order has instances in 263 genomes from our dataset: it is widely spread and spans both archea and bacteria, with many instances in both gram positive and gram negative bacteria. The second gene order, on the other hand, has 53 instances, 50 of which are confined to gram positive bacilli. Gram positive bacteria lack an outer membrane and consequently have no periplasm, and therefore their SPBs are lipoproteins that are tethered to the external surface of the cell membrane (Figure 6.G), anchoring the binding protein in the proximity of the external face of the TMDs of the transport system. To obtain such proximity, the TMDs need to be integrated to the cell membrane prior to the anchoring of the SBPS within their proximity. This could perhaps explain why a Dpp operon with "inside-out" transcription confers adaptation in some gram positive bacteria.

Example 2: A gene cluster encoding a sugar uptake system, and the transcription factor that regulates it. Figure 6.H-J. Here, the signed break-point distance between the reference gene order (Figure 6.I) and the target gene order (Figure 6.J) is 2, due to the break-points (1,2) and (4,5). In contrast, the $\mathsf{m}_{\mathsf{Diverg}}$ score is only 1. Using the structural information encoded by the PQ-tree, our engine recognizes that the break-point (1,2) may be functionally irrelevant, since the rearrangement events are confined to the P-node encapsulating the sub-cluster that encodes the uptake system machine, including gene 2 which codes for the

NBD MalK. Within the P-node encapsulating the Ugp operon, the NBD is separate from the rest of the genes in the transporter, which are confined to the Q-node, due to stochiometry (MalK is a homodimer [13]). Thus, it is realized by the PQ-tree, and conveyed via structure-informed re-arrangement, that the positional order between the uptake system operon and the transcription factor that regulates it (gene 1), remains intact during the rearrangement from the reference gene order to the target one.

References

1. Eric Alm, Katherine Huang, and Adam Arkin. The evolution of two-component systems in bacteria reveals different strategies for niche adaptation. *PLoS computational biology*, 2(11):e143, 2006.
2. Severine Bérard, Anne Bergeron, and Cedric Chauve. Conservation of combinatorial structures in evolution scenarios. In *RECOMB Workshop on Comparative Genomics*, pages 1–14. Springer, 2004.
3. Severine Bérard, Anne Bergeron, Cedric Chauve, and Christophe Paul. Perfect sorting by reversals is not always difficult. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4:4–16, 2007. doi:10.1145/1229968.1229972.
4. Sèverine Bérard, Annie Chateau, Cedric Chauve, Christophe Paul, and Eric Tannier. Computation of perfect dcj rearrangement scenarios with linear and circular chromosomes. *Journal of Computational Biology*, 16(10):1287–1309, 2009.
5. Severine Bérard, Cedric Chauve, and Christophe Paul. A more efficient algorithm for perfect sorting by reversals. *Information processing letters*, 106(3):90–95, 2008.
6. Anne Bergeron, Yannick Gingras, and Cedric Chauve. Formal models of gene clusters. *Bioinformatics Algorithms: Techniques and Applications*, 8:177–202, 2008. doi:10.1002/9780470253441.ch8.
7. Anne Bergeron, Julia Mixtacki, and Jens Stoye. Mathematics of evolution and phylogeny, chapter the inversion distance problem, 2005.
8. Matthias Bernt, Daniel Merkle, Kai Ramsch, Guido Fritzsch, Marleen Perseke, Detlef Bernhard, Martin Schlegel, Peter F Stadler, and Martin Middendorf. Crex: inferring genomic rearrangements based on common intervals. *Bioinformatics*, 23(21):2957–2958, 2007.
9. Guillaume Blin, Cedric Chauve, and Guillaume Fertin. The breakpoint distance for signed sequences. In *Proc. 1st Algorithms and Computational Methods for Biochemical and Evolutionary Networks (CompBioNets)*, pages 3–16, 2004.
10. Kellogg S Booth and George S Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, 1976.
11. Marília DV Braga, Marie-France Sagot, Celine Scornavacca, and Eric Tannier. Exploring the solution space of sorting by reversals, with experiments and an application to evolution. *IEEE/ACM transactions on computational biology and bioinformatics*, 5(3):348–356, 2008.
12. Anne Chao, Robin L Chazdon, Robert K Colwell, and Tsung-Jen Shen. Abundance-based similarity indices and their estimation when there are unseen species in samples. *Biometrics*, 62(2):361–371, 2006.
13. Jue Chen, Gang Lu, Jeffrey Lin, Amy L Davidson, and Florante A Quiocho. A tweezers-like motion of the ATP-binding cassette dimer in an ABC transport cycle. *Molecular cell*, 12(3):651–661, 2003.
14. Aaron E Darling, István Miklós, and Mark A Ragan. Dynamics of genome rearrangement in bacterial populations. *PLoS genetics*, 4(7):e1000128, 2008.
15. Amy L Davidson, Elie Dassa, Cedric Orelle, and Jue Chen. Structure, function, and evolution of bacterial ATP-binding cassette systems. *Microbiology and molecular biology reviews*, 72(2):317–364, 2008.

16 Yoan Diekmann, Marie-France Sagot, and Eric Tannier. Evolution under reversals: Parsimony and conservation of common intervals. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(2):301–309, 2007.

17 Guillaume Fertin, Anthony Labarre, Irena Rusu, Stéphane Vialette, and Eric Tannier. *Combinatorics of genome rearrangements*. MIT press, 2009.

18 Martin Figeac and Jean-Stéphane Varré. Sorting by reversals with common intervals. In *International Workshop on Algorithms in Bioinformatics*, pages 26–37. Springer, 2004.

19 Marco Fondi, Giovanni Emiliani, and Renato Fani. Origin and evolution of operons and metabolic pathways. *Research in Microbiology*, 160(7):502–512, 2009. doi:10.1016/j.resmic.2009.05.001.

20 Yair E Gatt and Hanah Margalit. Common adaptive strategies underlie within-host evolution of bacterial pathogens. *Molecular biology and evolution*, 38(3):1101–1121, 2021.

21 Sridhar Hannenhalli and Pavel A Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). In *Proceedings of IEEE 36th annual foundations of computer science*, pages 581–592. IEEE, 1995.

22 Sridhar Hannenhalli and Pavel A Pevzner. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM (JACM)*, 46(1):1–27, 1999.

23 Tom Hartmann, Matthias Bernt, and Martin Middendorf. An exact algorithm for sorting by weighted preserving genome rearrangements. *IEEE/ACM transactions on computational biology and bioinformatics*, 16(1):52–62, 2018.

24 Géraldine Jean and Macha Nikolski. Genome rearrangements: a correct algorithm for optimal capping. *Information Processing Letters*, 104(1):14–20, 2007.

25 Gad M Landau, Laxmi Parida, and Oren Weimann. Gene proximity analysis across whole genomes via PQ trees1. *Journal of Computational Biology*, 12(10):1289–1306, 2005.

26 Claire Lemaitre, Marilia DV Braga, Christian Gautier, Marie-France Sagot, Eric Tannier, and Gabriel AB Marais. Footprints of inversions at present and past pseudoautosomal boundaries in human sex chromosomes. *Genome biology and evolution*, 1:56–66, 2009.

27 Anne E Magurran. Measuring biological diversity. *Current Biology*, 31(19):R1174–R1177, 2021.

28 Marie-France Sagot and Eric Tannier. Perfect sorting by reversals. In *International Computing and Combinatorics Conference*, pages 42–51. Springer, 2005.

29 Dina Svetlitsky, Tal Dagan, and Michal Ziv-Ukelson. Discovery of multi-operon colinear syntenic blocks in microbial genomes. *Bioinformatics*, 2020. doi:10.1093/bioinformatics/btaa503.

30 Eric Tannier, Anne Bergeron, and Marie-France Sagot. Advances on sorting by reversals. *Discrete Applied Mathematics*, 155(6-7):881–888, 2007.

31 Roman L Tatusov, Arcady R Mushegian, Peer Bork, Nigel P Brown, William S Hayes, Mark Borodovsky, Kenneth E Rudd, and Eugene V Koonin. Metabolism and evolution of Haemophilus influenzae deduced from a whole-genome comparison with Escherichia coli. *Current biology*, 6(3):279–291, 1996.

32 Tatiana Tatusova, Stacy Ciufo, Boris Fedorov, Kathleen O'Neill, and Igor Tolstoy. Refseq microbial genomes database: new representation and annotation strategy. *Nucleic Acids Research*, 42(D1):D553–D559, 2014. doi:10.1093/nar/gkt1274.

33 Kentaro Tomii and Minoru Kanehisa. A comparative analysis of ABC transporters in complete microbial genomes. *Genome research*, 8(10):1048–1059, 1998.

34 Alice R Wattam, David Abraham, Oral Dalay, Terry L Disz, Timothy Driscoll, Joseph L Gabbard, Joseph J Gillespie, Roger Gough, Deborah Hix, Ronald Kenyon, et al. Patric, the bacterial bioinformatics database and analysis resource. *Nucleic Acids Research*, 42(D1):D581–D591, 2014. doi:10.1093/nar/gkt1099.

35 Jonathan N Wells, L Therese Bergendahl, and Joseph A Marsh. Operon gene order is optimized for ordered protein complex assembly. *Cell reports*, 14(4):679–685, 2016.

36 Sophia Yancopoulos, Oliver Attie, and Richard Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16):3340–3346, 2005.

37 G. R. Zimerman. The PQFinder tool. `www.github.com/GaliaZim/PQFinder`.

38 Galia R Zimerman, Dina Svetlitsky, Meirav Zehavi, and Michal Ziv-Ukelson. Approximate search for known gene clusters in new genomes using PQ-trees. *arXiv preprint arXiv:2007.03589*, 2020.

Fast Gapped k-mer Counting with Subdivided Multi-Way Bucketed Cuckoo Hash Tables

Jens Zentgraf
Department of Computer Science, Saarland University, Saarbrücken, Germany
Center for Bioinformatics, Saarland University, Saarbrücken, Germany
Saarbrücken Graduate School of Computer Science, Saarland University, Saarbrücken, Germany

Sven Rahmann
Department of Computer Science, Saarland University, Saarbrücken, Germany
Center for Bioinformatics, Saarland University, Saarbrücken, Germany

Abstract

Motivation. In biological sequence analysis, alignment-free (also known as k-mer-based) methods are increasingly replacing mapping- and alignment-based methods for various applications. A basic step of such methods consists of building a table of all k-mers of a given set of sequences (a reference genome or a dataset of sequenced reads) and their counts. Over the past years, efficient methods and tools for k-mer counting have been developed. In a different line of work, the use of gapped k-mers has been shown to offer advantages over the use of the standard contiguous k-mers. However, no tool seems to be available that is able to count gapped k-mers with the same efficiency as contiguous k-mers. One reason is that the most efficient k-mer counters use *minimizers* (of a length $m < k$) to group k-mers into buckets, such that many consecutive k-mers are classified into the same bucket. This approach leads to cache-friendly (and hence extremely fast) algorithms, but the approach does not transfer easily to gapped k-mers. Consequently, the existing efficient k-mer counters cannot be trivially modified to count gapped k-mers with the same efficiency.

Results. We present a different approach that is equally applicable to contiguous k-mers and gapped k-mers. We use multi-way bucketed Cuckoo hash tables to efficiently store (gapped) k-mers and their counts. We also describe a method to parallelize counting over multiple threads without using locks: We subdivide the hash table into independent subtables, and use a producer-consumer model, such that each thread serves one subtable. This requires designing Cuckoo hash functions with the property that all alternative locations for each k-mer are located in the same subtable. Compared to some of the fastest contiguous k-mer counters, our approach is of comparable speed, or even faster, on large datasets, and it is the only one that supports gapped k-mers.

2012 ACM Subject Classification Applied computing → Molecular sequence analysis; Applied computing → Bioinformatics; Theory of computation → Bloom filters and hashing; Theory of computation → Data structures design and analysis

Keywords and phrases gapped k-mer, k-mer, counting, Cuckoo hashing, parallelization

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.12

Supplementary Material Our software `hackgap` (**ha**sh-based **c**ounting of ***k***-mers with **gap**s) is available on gitlab under the MIT license.
Software (Source Code): `https://gitlab.com/rahmannlab/hackgap`
archived at `swh:1:dir:2a19b268091ab679be5cc1424153dbdcfdb52433`

Acknowledgements We thank Rajesh Moturu for running preliminary experiments with gapped k-mers.

1 Introduction

Counting k-mers (short pieces of DNA sequences of length k) in reference genomes (FASTA files) or sequenced samples (FASTQ files) is a basic step of modern DNA sequence analysis, and many applications depend on efficient k-mer counting tools. For example, genome assembly is often done using the so-called De Bruijn graph (of which there are many variations), which represents all sufficiently frequent k-mers in the sample, recently even for several values of k [1]. RNA transcript quantification can be done by examining the k-mers in each read and assigning a set of possible genes or transcripts to each read based on k-mer content [2]. The abundance of each species in a metagenomic sample can also be estimated by counting k-mers in the reads and assigning them to (groups of) species [10, 16, 21]. Because k-mer counting is such an ubiquitous task, many methods tools for precisely this purpose have been developed, published and reviewed [5, 8, 6, 12].

The purpose of a k-mer counter is to output a mapping $\mathit{count} : \Sigma^k \supset X \to \{1, \ldots, C\}$, where Σ is the (DNA) alphabet, X is the set of k-mers that are present in the sample under consideration, and C is the maximum implemented count, such that all values $\geq C$ will be reported as C. Typically, $C = 2^v - 1$ if v bits are used to represent counters. The final mapping can be stored in different ways, and each tool uses its own disk-based representation.

A recent review [12], and also our own experience, indicate that KMC3 [8, 5] and Gerbil [6] are among the fastest tools in practice. However, these tools also have their limitations: They are specialized towards k-mer counting and do not offer much other functionality (although they have accompanying complementary tools that allow to perform multiset operations on several output files with counts). They only support contiguous k-mers and do not extend towards gapped k-mers (Section 2.1). This limitation is not easily fixed, because their data structures (and their efficiency) depend on concepts related to *minimizers* and *super-k-mers*, sequences of consecutive k-mers that share the same minimizer (Section 2.2). On the other hand, it has been shown several times that gapped k-mers offer a number of advantages over contiguous k-mers in many applications, in particular as seeds for alignments [17], but also for metagenomic classification [3].

In the above-mentioned application examples (assembly, transcript quantification, metagenomics), having *unique* k-mers (that appear at only one location in the genome, in only one gene or transcript, in only one species' genome) will facilitate the analysis. If there are not enough unique k-mers, the k-mer counts will have to be disentangled to provide accurate estimates in the examples above. On the other hand, if most k-mers are unique, such a step may not be necessary, and one may even be able to ignore non-unique k-mers entirely. Consequently, we focus on $k \in \{21, \ldots, 31\}$, because for these k, most of the k-mers are unique in a mammalian genome, i.e., they occur at only one position in the genome. Using gapped k-mers with specific masks may further increase the number of unique (gapped) k-mers in comparison to contiguous k-mers (see Section 4.4).

Because of the wide applicability of gapped k-mers, it would be helpful to have a method and a tool that allows counting them with comparable efficiency as for contiguous k-mers. The challenge is that the concepts of minimizers and super-k-mers do not directly extend to gapped k-mers. Therefore, we present an alternative methodology, using multi-way bucketed Cuckoo hashing with quotienting. After introducing the necessary background and terminology (Section 2), we present our hashing strategy and its parallelization using subtables (Section 3). We evaluate the approach on several datasets, both reference genomes and large sequence-read datasets, and compare (for the contiguous case) with Gerbil and KMC3 (Section 4).

2 Background and Basic Definitions

We first introduce necessary definitions (k-mers, canonical codes of k-mers, gapped k-mers, masks, shapes and κ-mers) in Section 2.1. Then, we review efficient k-mer counting strategies based on minimizers and super-k-mers (Section 2.2) and explain why these do not easily extend to gapped k-mers. Finally, we introduce Cuckoo hashing (Section 2.3) and its practical variations, a basis for our method.

2.1 Terminology

By $[n]$, we denote the set of integers $\{0, \ldots, n-1\}$. Indexing of strings starts at 0 (not at 1). We also write $u[0:k] := u_0 u_1 \ldots u_{k-1}$ (slice notation *excluding* the last index).

▶ **Definition 1** (k-mer). *A DNA sequence is a string of any finite length over the DNA alphabet $\Sigma := \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ of size 4. A k-mer (of a sequence s) is any substring of length k.*

There are 4^k different DNA k-mers. However, a DNA molecule u is equivalent to its reverse complement $rc(u)$, i.e., the reversed string with `A` replaced by `T` and `C` replaced by `G` (and vice versa). Here $rc : \Sigma^k \to \Sigma^k$ maps a k-mer u to its reverse complement. For example, for $k = 5$, the sequence `AACTG` is equivalent to `CAGTT`. Therefore, for odd k, there are only $4^k/2$ equivalence classes (DNA molecules). When k is even, a k-mer's reverse complement may be equal to the k-mer itself (e.g., for `GATATC`), and this happens for exactly $4^{k/2}$ k-mers, so there are $(4^k - 4^{k/2})/2 + 4^{k/2} = (4^k + 4^{k/2})/2$ different DNA molecules of length k. To avoid issues of self-reverse-complementarity, we work with odd values of k.

▶ **Definition 2** (Canonical integer encoding of a k-mer). *Each DNA nucleotide may be encoded with two bits (typically, $\mathtt{A} \mapsto (00)_2 = 0$, $\mathtt{C} \mapsto (01)_2 = 1$, $\mathtt{G} \mapsto (10)_2 = 2$, $\mathtt{T} \mapsto (11)_2 = 3$). This allows encoding a k-mer u uniquely (for fixed k) as a base-4 number $c = enc(u)$ with $0 \le c < 4^k$. To ensure that a k-mer and its reverse complement (the same DNA molecule) are represented by the same integer, we represent both by the* larger *of the two encodings; this is called the* canonical code *$cc(u)$ of the k-mer u: $cc(u) := \max\{enc(u), enc(rc(u))\}$.*

Often, the *smaller* value of the two encodings is used as the canonical code, but our choice avoids confusion with minimizers; see below.

▶ **Example 3.** For $k = 5$, we have $enc(\mathtt{AAGCG}) = (00212)_4 = (00|00|10|01|10)_2 = 38$, and $enc(rc(\mathtt{AAGCG})) = enc(\mathtt{CGCTT}) = (12133)_4 = 415$, so $cc(\mathtt{AAGCG}) = cc(\mathtt{CGCTT}) = 415$.

We have defined k-mers as (contiguous) substrings of a string s. However, we can also "extract" k characters from a larger window of size w, according to a specific pattern, called a *mask*, of *shape* (w, k). The resulting k-tuple of characters is called a *gapped k-mer* (for the given mask), or also a *spaced seed*.

▶ **Definition 4.** *Given two integers $w \ge k \ge 2$, a mask of* shape (w, k) *is a string μ of length w over the alphabet $\{\mathtt{\#}, \mathtt{\_}\}$ that contains exactly k times the character `#` and $w-k$ times the character `_`. The positions marked `#` are called* significant*, and the positions marked `_` are called* insignificant *or* gaps*. We call k the* weight *of the mask and w its* width*. The mask μ may equivalently be defined as a bit vector of length w with k ones. It can also be represented as the set κ of the significant positions: $\kappa = \{j : 0 \le j < w \text{ and } \mu_j = \mathtt{\#}\}$.*

The masks we consider are subject to two technical conditions:

Table 1 Different masks of shape $(w, k) = (31, 25)$ and their properties. Column "max d_H" refers to the maximum Hamming distance tolerated in a read of length 100 bp, such that there is guaranteed to be *at least one* κ-mer unaffected by the substitutions, even if their positions "conspire" to affect as many κ-mers as possible. Column "intact κ-mers" specifies *how many* κ-mers are guaranteed to be unaffected by substitutions in the worst case at the maximum possible Hamming distance. Column "covered bp" specifies *how many basepairs* intact κ-mers are guaranteed to cover.

ID	mask μ	max d_H	intact κ-mers	covered bp
k25 = m1	`#########################`	3	1	25
m2	`####_####_###_###_###_####_####`	4	2	32
m3	`####_###_###_#####_###_###_####`	4	2	32
m4	`###_##_#####_#####_#####_##_###`	4	3	34

1. We require that the first and last mask positions are significant: $\mu_0 = \mu_{w-1} =$ `#`; otherwise, we could just shorten the mask and obtain equivalent results, except for boundary effects.
2. We require that the mask is *symmetric*, so we do not need to distinguish (in terms of the mask) between the forward and reverse complementary DNA sequence.

For a given mask μ or equivalent set κ, we say that a string u of length k is a *μ-mer* or *κ-mer* of a sequence s, if $u = (s_{i+j})_{j\in\kappa}$ for some starting position i.

Clearly, a κ-mer of shape (w, k) with a symmetric mask has a canonical code just like a regular k-mer by only considering the significant k positions.

▶ **Example 5.** Consider $(w, k) = (7, 3)$ with $\mu =$ `#__#__#` or $\kappa = (0, 3, 6)$. Let $s :=$ `TACAGATATA`. We extract and encode the κ-mers as follows:

```
TACAGATATA
T__A__T       cc(TAT) = max(enc(TAT), enc(ATA)) = 51
 A__G__A      cc(AGA) = max(enc(AGA), enc(TCT)) = 55
  C__A__T     cc(CAT) = max(enc(CAT), enc(ATG)) = 19
   A__T__A    cc(ATA) = max(enc(ATA), enc(TAT)) = 51
```

Design of masks. For a symmetric mask with odd k, also w must be odd, and the middle position must be significant. For a given shape (w, k) with odd w and k, there are $\binom{(w-3)/2}{(k-3)/2}$ symmetric masks that satisfy our technical constraints (the first, middle and last position must be significant positions; we may freely distribute half of the remainder of the significant positions in the first half of the mask between first and middle position). For example, for $(w, k) = (31, 25)$, there are $\binom{14}{11} = \binom{14}{3} = 364$ different masks to consider.

There is a rich body of literature on the design of good or optimal masks with regard to a given objective function [18, 7, 11], with early works dating back to the first ISMB conference in 1993 [4]. An extensive bibliography on the topic is maintained by Laurent Noé[1].

Table 1 shows the contiguous 25-mer mask and three different $(31, 25)$-shaped masks and some of their properties. For example, if we distribute 4 substitutions (SNVs or sequencing errors) in a read of length 100 in a "conspiring" manner (as to modify as many κ-mers as possible), it is possible to distribute them in a way (equidistantly) to modify all of the contiguous 25-mers. Even at the lower Hamming distance of 3, only a single 25-mer is guaranteed to be unchanged. However, if we use mask m4, independently of the distribution

[1] `https://sites.google.com/view/laurentnoe/spaced-seeds`

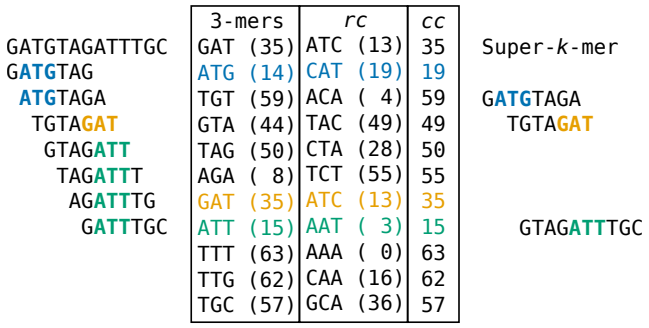


Figure 1 Illustration of a string `CGTTGATCATTTG`, its 7-mers and 3-mers, the canonical codes of the 3-mers, the minimizers for $m = 3$ using the identity permutation $\pi = \mathrm{id}$ and the resulting super-k-mers. We observe that minimizers frequently do not change as we advance the k-mer window; this yields super-k-mers of length up to $2k - m$.

of the 4 substitutions, at least 3 intact κ-mers remain, and at least 34 bp are covered by intact κ-mers. These properties have been determined by combinatorial optimization (unpublished own work).

2.2 Strategies for Efficient k-mer Counting

The k-mer counting tools KMC3 [8] and Gerbil [6] have repeatedly been evaluated to be among the fastest such tools [12]. They rely on the concepts of *minimizers* and *super-k-mers* for efficient counting. Here, we briefly review these ideas. Unfortunately, they do not easily transfer to counting gapped k-mers, as we explain below. Therefore, we propose a different strategy in this work based on a variant of Cuckoo hashing, whose basics we review in the next subsection. We first define the concept of a minimizing m-mer within a k-mer for $m \leq k$.

▶ **Definition 6** (Minimizer). *Let $1 \leq m \leq k$ be two integers. Let $u = u[0:k]$ be a DNA k-mer. Let $V = (v_0, \ldots, v_{k-m}) := (u[0:m], u[1:m+1], \ldots, u[k-m:k])$ be the $k-m+1$-long sequence of m-mers in u. Let $\pi : [4^m] \to [4^m]$ be a permutation on the set of encoded m-mers (for simplicity, we may think of the identity permutation). Consider the sequence of permuted canonical m-mer codes $P = (p_0, \ldots, p_{k-m}) := (\pi(cc(v_0)), \pi(cc(v_1)), \ldots, \pi(cc(v_{k-m})))$. We say that v_j is a* minimizer *(or minimizing m-mer) of u if $p_j = \min P$. In other words, a minimizer is an m-mer with the smallest π-value in u.*

The definition above can be relaxed in the sense that instead of a (bijective) permutation π, a hash function can be used, which may result in non-uniqueness of minimizers.

Adjacent k-mers frequently share the same minimizer, which moves through the k-mer from right to left. Such adjacent k-mers with the same minimizer may be grouped into a so-called *super-k-mer* of some length ℓ with $k \leq \ell \leq 2k - m$. This is illustrated in Figure 1.

The strategy of KMC can be described (somewhat simplified) as follows: The input files are read. For each k-mer in sequence, the minimizer is computed, and the k-mer is appended to one of 512 buffers, which is computed as a function of the minimizer (the default number of buffers can be changed by command line). This means that adjacent k-mers are frequently stored in the same buffer, which is cache-efficient. If a buffer is full, it is sorted, uniquified and counted, and then appended to a corresponding temporary file. After all k-mers have been read, each temporary file is again sorted, uniquified and counted. This can be done independently and hence in parallel for each temporary file, subject to memory and CPU

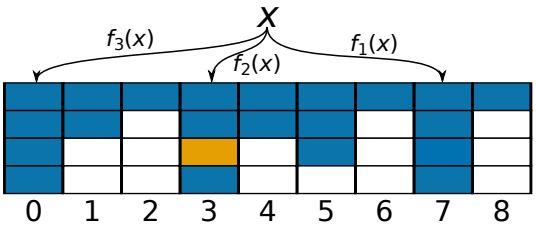


Figure 2 Illustration of multi-way bucketed Cuckoo hashing with $h = 3$ hash functions and bucket size $b = 4$. If all possible $hb = 12$ slots to store a key x are already occupied (blue/orange), a random one of these slots is picked (orange) and removed to make room for x. The removed element is then inserted at an alternative location.

core constraints. Finally, filters are applied to remove too frequent or too rare k-mers, and the remaining k-mers, together with their counts, are written into an output database in a custom binary format.

The strategy of Gerbil is slightly different, but also relies on the cache-friendliness of super-k-mers. The output file of Gerbil is larger but easier to use than that of KMC3: The k-mers and counters are given as byte sequences with as many bytes per k-mer and counter as required.

The purpose of this section is to highlight that the main reason for the speed of some of the fastest existing k-mer counters relies on exploiting the cache-friendliness of grouping many adjacent k-mers into a super-k-mer. In addition, for computing hash values of adjacent contiguous k-mers (or m-mers), a *rolling* hash function may be used, such as ntHash [14], which updates hash values of k-mers in constant time instead of time proportional to k.

Neither idea (super-k-mers, rolling hash functions) is directly applicable if one uses gapped k-mers instead of contiguous ones. this is why these and other tools only support contiguous k-mers.

2.3 Multi-way Bucketed Cuckoo Hashing

The method we propose in Section 3 is not based on minimizers or super-k-mers, but on directly building an in-memory hash table with canonical codes of k-mers as keys and their counts as values. We previously found multi-way bucketed Cuckoo hashing [23, 24] to perform well in terms of both low memory usage (due to a high load factor) and reasonably fast insertion, update and lookup times, so we briefly review their basics.

Consider a hash table of p buckets, where each bucket has space for $b \geq 1$ key-value pairs; we also say that a bucket consists of b slots. Thus, the total capacity of the table is $n := pb$. We use $h \geq 2$ hash functions $f_1, \ldots, f_h : \Sigma^k \to [p]$; each hash function maps each k-mer (key) u to one of the p buckets. Thus, in general, there are hb possible slots where a given key may be stored (see Figure 2). This hashing strategy is referred to as h-way bucketed Cuckoo hashing with bucket size b, or (h, b)-*Cuckoo hashing* for short. It is designed to provide fast lookups and acceptable insertion time. The main idea is that the contents of a small bucket ($b \in \{3, \ldots, 10\}$) are contained within a single cache line, so searching a bucket will incur only a single cache miss.

Lookup. When searching for a k-mer u with canonical code c, we first examine every slot in bucket $f_1(c)$. If u is not found (but the bucket is full), the search continues in bucket $f_2(c)$, and then in $f_3(c), \ldots, f_h(c)$. In this way, a lookup examines between 1 and h buckets, resulting in between 1 and h cache misses. Depending on the load factor of the hash table

(number of slots with keys in relation to $n = hb$), many searches only need to examine a single bucket and incur only a single cache miss. For example, we can experimentally observe that with $h = 3$, $b = 6$ and a load factor around 83%, we need to access 1.16 different buckets on average for a successful search; so many keys are stored in their "first" bucket $f_1(c)$.

Insertion by random walk. While lookup is simple and restricted to h different buckets, the insertion of a new element u can be slightly more complex. Let us start with the easy case. If any of the b slots in bucket $f_1(c)$ is available, u is inserted there. Otherwise, we repeat the attempt for buckets $f_2(c), \ldots, f_h(c)$. As the table gets fuller, it may happen that all hb slots for u are already occupied. In this case, we pick a random one of these hb keys (call it v) and remove it to insert u in its place (hence the name "Cuckoo" hashing). Now we re-start the insertion process for v instead of u, hoping that one of its hb slots is still free. The process may repeat until we either find a free slot, or we report failure (after a pre-determined number of attempts). When picking a random element to remove to insert v, we ensure not to remove again the just previously inserted u, but to pick a different slot.

Alternatively, if all keys are known in advance, it is possible (but computationally expensive) to compute an optimal placement, such that the average number of bucket accesses for successful searches is minimized [25].

Properties of (h, b)-Cuckoo hashing. Walzer [22] analyzed the theoretical load limits of (h, b)-Cuckoo hashing. For classical Cuckoo hashing ($h = 2$ and $b = 1$), the load limit is 1/2 (meaning that if the hash table is only slightly more than 50% full, it will not be possible to insert all keys), but the load limit for (3,4)-Cuckoo hashing is above 99.9%. With random walk insertion (using a finite number of steps), these limits cannot be achieved, and performance degrades as the actual load approaches the limit, so a "safety margin" of 5% should be used in practice. Thus in practice, to obtain high loads, $h = 3$ hash functions suffice, even for small bucket sizes, and in this work, we only use $h = 3$.

3 A k-mer Counter Based on (h, b)-Cuckoo Hashing with Subtables

3.1 Design Goals for Counter Data Structures

A data structure that supports counting the number of occurrences of keys (here, integer-encoded k-mers) must support the following operations very efficiently:

- insertion of a new key with counter value 1,
- test whether a given key is present,
- retrieval of the counter value of a given key if present, or zero otherwise,
- update of the counter value (typically, increment by 1) for a given key.

In contrast, deletions of existing keys need not be supported. Since (h, b)-Cuckoo hash tables (see Section 2.3) support the required operations quite efficiently, they form a good basis for designing a k-mer counter.

Other desirable properties of the data structure are

- the ability to grow dynamically (if the final number of keys to be stored is initially unknown),
- its support of parallel writes, as the sizes of datasets often range in the hundreds of gigabytes.

Our current implementation does not support dynamic growth, but in most applications, a good estimate of the number of keys is known can be quickly estimated [15].

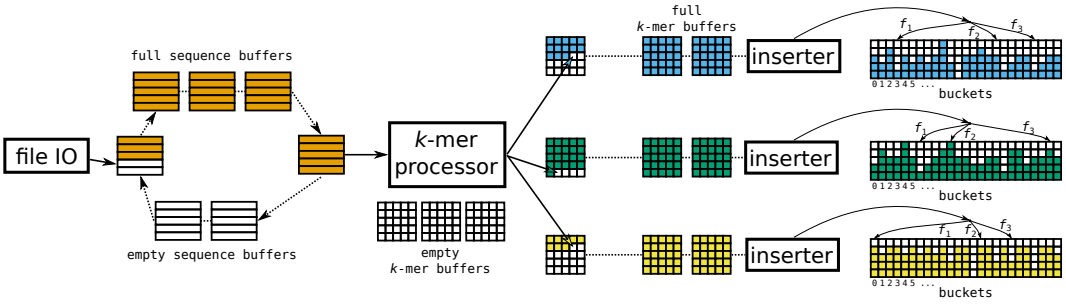


Figure 3 Producer-consumer architecture: An I/O thread reads a large chunk of an input file into an available sequence buffer. A producer thread (the k-mer processor) computes the canonical code c of each k-mer in the sequence buffer, and determines into which subtable t it shall be inserted, by using a simple invertible hash function g_0, such that $(t, s) = (g_0(c) \bmod T,\ g_0(c) /\!\!/ T)$. The quotient s is placed into a k-mer buffer that only thread t will read for insertion into subtable $t \in [T]$.

Supporting parallel write access to hash tables from different threads is non-trivial if one wants to avoid data races. One possibility is to use locks (of individual buckets or larger regions of the hash table), although this approach is rather costly in terms of computational overhead. A different, less costly, option is to design lock-free approaches using atomic CPU operations such as CAS (compare-and-swap), as was implemented for the Jellyfish k-mer counter [13]. However, these atomic CPU instructions operate on entire memory words (e.g., a 64-bit integer). As we use bit packing to save space (see Section 3.3 below), it is possible that one key/value pair extends over several (memory-aligned) words in our hash table implementation; so we opted for a different approach that offers additional advantages.

3.2 Parallelizing Key Insertions into (h, b)-Cuckoo Hash Tables

Given T threads, the key idea is to divide the hash table into T independent subtables, such that each subtable is served (written to) by a specific single thread.

As we read a (gapped) k-mer u from the sequences in the dataset, a *producer thread* computes its canonical code $c = cc(u)$ and, using a simple *bijective* hash function g_0, partitions it into a subtable number $t = g_0(c) \bmod T \in \{0, 1, \ldots, T-1\}$ and a *subkey* $s = g_0(c) /\!\!/ T$, where $/\!\!/$ denotes integer floor division. Given (t, s), we can recover $g_0(c) = s \cdot T + t$, and since g_0 is bijective, we can recover c. For this to work efficiently, g_0 must be easily invertible (see below).

The producer thread inserts the computed s into a buffer that will be read only by the *consumer thread* for subtable t. Several such buffers exist for each consumer thread, and the producer fills them in a round-robin fashion, marking a buffer as "ready-to-read" if it is full and can be processed by the consumer. It then selects a new (ready-to-write) buffer for consumer t to fill next. If no ready-to-write buffer is available, the producer must wait, stalling the whole counting process. Conversely, if a consumer thread has inserted all keys from one buffer, it marks it as "ready-to-write" for the producer and picks the next "ready-to-read" buffer that is available for it (or waits for one to become available). This producer-consumer approach is shown in Figure 3.

The "ready-to-read" and "ready-to-write" messages are sent by atomic volatile stores into and loads from a small integer array accessible from all threads. If a thread has to wait, it keeps monitoring the states of all of its buffers and consumes CPU time, but in an energy-saving state executing the `pause` instruction between checks.

Table 2 Double quotienting: A k-mer u's canonical code c is partitioned into a subtable number t and the first "quotient", the subkey s. The subkey is then partitioned (for each hash function h_j into the bucket address $f_j(s)$ and the second "quotient" $q_j(s)$. Only this quotient $q_j(s)$ needs to be stored (together with j) at address $f_j(s)$ in subtable t in order to reconstruct c.

$$\text{key } u \in \Sigma^k \to \text{ canonical code } c \in [4^k] \quad \longleftrightarrow \quad (t, s) = (g_0(c) \bmod T,\ g_0(c) /\!\!/ T)$$

$$\text{subkey } s \quad \longleftrightarrow \quad (f_j(s),\ q_j(s),\ j) = (g_j(s) \bmod p,\ g_j(s) /\!\!/ p,\ j) \text{ for } j = 1, \ldots, h$$

Hash functions. The h functions $f_1, \ldots, f_h$ that select the possible buckets for a k-mer u are functions of u's subkey s, and they map each possible subkey onto the address space of a subtable $[p_{\text{sub}}]$. Each subtable contains p_{sub} buckets, so the full hash table contains $p = Tp_{\text{sub}}$ buckets.

For the f_j, we use the same structure as for the initial hash function f_0: In general, we take $f_j(s) = g_j(s) \bmod p_{\text{sub}}$ with a function g_j that is bijective on $[2^\sigma]$, where σ is the bit width of subkeys. We also compute the quotient $q_j(s) = g_j(s) /\!\!/ p_{\text{sub}}$, so we can recover s from its address (bucket number) $f_j(s)$ and quotient $q_j(s)$ if $g_j(s)$ is efficiently invertible.

For the bijective functions g_j, $j = 1, \ldots, h$ we use affine functions of the form

$$G_{a,b}(s) := [a \cdot (\text{rot}(s) \text{ xor } b)] \bmod 2^\sigma,$$

where rot performs a cyclic rotation by half the bit width of s, and b is a σ-bit offset, and a is an odd multiplier. Picking a "random" hash function means picking random values for a and b. Such a $G_{a,b}$ is invertible on $[2^\sigma]$ because a is odd; so $\gcd(a, 2^\sigma) = 1$. Therefore, there exists a unique multiplicative inverse a' such that $aa' = 1 \pmod{2^\sigma}$. This inverse a' can be pre-computed efficiently using the extended Euclidean algorithm. It follows that, given $i = f_j(s)$ and $q = q_j(s)$, we can recover s by first computing $x = q \cdot p_{\text{sub}} + i$ and then $s = G^{-1}_{a,b}(x) = [(a' \cdot x) \bmod 2^\sigma] \text{ xor } b$.

3.3 Space-Saving Techniques

As we do not use temporary files and do only in-memory computations, we must take care not to waste space unnecessarily. Our main ingredients for space efficiency are

1. a high load of the hash table,
2. bit packing,
3. storing quotients instead of full keys in the hash table.

A high load rate is achieved by choosing large enough h (number of hash functions) and b (bucket size); we already mentioned that $h \geq 3$ and $b \geq 4$ allow loads above 99.9%. By "bit packing" we mean that we view the hash table as a bit array instead of a byte or uint64 array: If a key-value pair needs 31 bits, we only store 31 bits and do not round up to 32 bits. This sometimes requires reading a single value from two adjacent memory cells, but the CPU efficiently handles bit manipulations. The quotienting technique reduces the storage requirement of every single key-value pair and is explained below.

Double quotienting. Instead of storing the full key-value pair (canonical k-mer code c and its counter) for each k-mer, we only store a part of the key that, however, allows to reconstruct the exact value of c. This is possible because part of the information is stored in the address (subtable t, bucket i) itself. As described above and visualized in Table 2, we

first split the canonical code c into (t, s), so we only need to store s within subtable t (which requires fewer bits than the full c). Because s is a quotient ($s = g_0(c) /\!\!/ T$), this is referred to as *quotienting*. The same strategy is applied again to s where we compute the bucket numbers $i_j = g_j(s) \bmod p_{\text{sub}}$ and the quotients $q_j = g_j(s) /\!\!/ p_{\text{sub}}$ for $j = 1, \ldots, h$. Because we use h hash functions, we must also store *which* hash function j we used (2 bits for $h = 3$).

Importantly, quotienting is only possible when using efficiently invertible hash functions $g_0, g_1, \ldots, g_h$, as described above.

▶ **Example 7.** We store 25-mers (50 bits each) and counters between 0 and 255 (8 bits each), requiring 58 bits per key-value pair. (Technically, 49 bits suffice for storing canonical 25-mers, using a minimal perfect hash function that maps the valid canonical 50-bit codes to $[2^{49}]$. Such functions are folklore for odd k and have been communicated to us at DSB 2022 by Roland Wittler for even k.)

Assume that we use $T = 9$ subtables, each with $p_{\text{sub}} = 83\,333\,335$ buckets of size $b = 4$, for a total of $p = Tp_{\text{sub}} = 750\,000\,015$ buckets and $n = bp = 3\,000\,000\,060$ slots. Then, $t \in [9] = \{0, \ldots, 8\}$, and each subkey $s < 4^k/T$ can be represented in 47 bits (46.83 in theory). Moreover, $f_j(s) \in [p_{\text{sub}}]$, and each quotient $q_j(s) < 2^{47}/p_{\text{sub}}$ can be represented in $\lceil \log_2(2^{47}/p_{\text{sub}}) \rceil = 21$ bits.

Instead of 50 bits per key, we only need to store 21 bits per key, plus 2 bits to remember the hash function j (1, 2, or 3), plus the 8 bits for counters up to 255. So we need 31 bits (instead of 58) per key-value pair.

The more buckets we have, the fewer bits per k-mer are needed (when p_{sub} doubles, we have 1 bit less in the quotient). Using smaller values of b also increases p_{sub} (for a constant number of slots in the hash table). However, using smaller b also decreases the load threshold and requires storing more keys at their second and third hash choices.

3.4 Just-in-Time Compilation

An important aspect of our implementation (not of the methodology itself) is that we just-in-time compile all methods of the hash table *after* we know its parameters. This is achieved using Python as our main implementation language, together with the `numba` package [9] that allows compiling a subset of Python at run-time.

As we invoke affine hash functions of the form $c \mapsto (a(c \text{ xor } b)) \bmod p$ billions to trillions of times, using mod and integer division ($/\!\!/$) operations, there is a significant difference in running time if the modulus or divisor p is a variable or a compile-time constant. If it is a compile-time constant, as in our case, the compiler can replace the general and expensive `divmod` operations by specialized faster code for the given value of p. The same optimization potential exists for integer-encoding a gapped k-mer with a fixed mask that becomes known after the program starts but before compilation.

4 Computational Experiments

4.1 Datasets and Experimental Setup

We present results on different datasets, which comprise the recent telomere-to-telomere (t2t) human genome [19], and several whole-genome sequencing samples from humans and pigs. Sources and statistics are given in Table 3. The datasets were chosen because they exhibit different properties.

Table 3 Datasets for evaluation and their properties (Ref.: literature reference; Gbp: Giga basepairs in the dataset; size [GB]: total file size on disk for the dataset; distinct-25: number of distinct 25-mers in the dataset; for gapped k-mers, the number is higher). URLs for obtaining the data are given in Appendix A.

Name	Ref.	type	Gbp	size [GB]	distinct-25
t2t human genome	[19]	1 `.fasta.gz`	3.117	0.916	2 391 456 540
Göttingen minipigs	[20]	2 `.fastq`	36.788	126.034	2 892 672 435
(10 minipig samples)		(each sample)	to 42.540	to 145.458	to 3 501 660 208
GIAB Ashkenazim trio	[26]	12 `.fastq.gz`	314.554	220	14 297 937 824

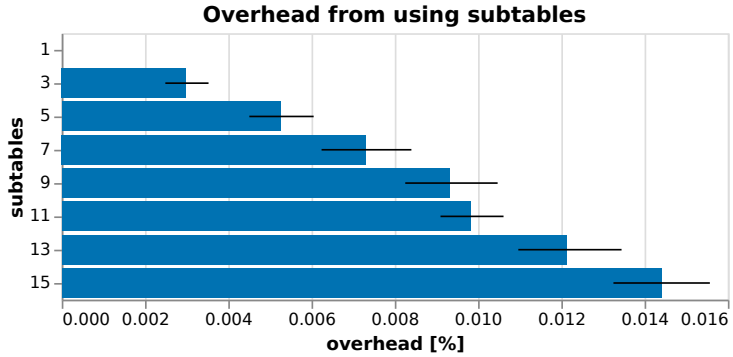


Figure 4 Load overhead from using different numbers of subtables (means and standard deviations from 20 random samples of hash functions). All overheads are less than $0.02\% = 2 \cdot 10^{-4}$.

- The t2t human genome is a small dataset in gzipped FASTA format with only approx. 2.4 billion distinct 25-mers, most of which occur only once.
- The 10 individual Göttingen minipig WGS samples are of moderate size (15x coverage, 2x100 bp Illumina paired-end, uncompressed FASTQ) and their 25-mers and counts conveniently fit into main memory. Many k-mers occur multiple times, as expected for 15x coverage samples, and in each individual, there is a limited number of rare k-mers (e.g., singletons). Each individual was counted separately.
- The GIAB Ashkenazim trio dataset consists of three individual humans from one family (mother, father, child), mate-pair sequenced with 2x125 bp. We counted (gapped) 25-mers from all files simultaneously, resulting in a table that just fits into 64 GB of main memory (14.3 billion 25-mers in 55 GB).

We use $k = 25$ and different symmetric masks of shape $(k, w) = (25, 31)$ with 6 insignificant positions (see Table 1). To evaluate k-mer uniqueness, we also report results on more different values of k that are suitable for mammalian genomes.

We compared KMC3 [8], Gerbil [6] and our tool `hackgap` on a PC workstation with an AMD Ryzen 9 5950X (4.9 GHz) processor with 16 cores (32 threads due to hyperthreading) and 64 GB of DDR4 RAM (3200 MHz, CL22). During evaluations, other activities on the workstation were suspended. All files (inputs, outputs and temporary files) were stored on an internal 16 TB HDD (SATA 3.3 with 6 Gbit/s, 7200 rpm). The exact command lines for calling each tool can be found in Appendix B.

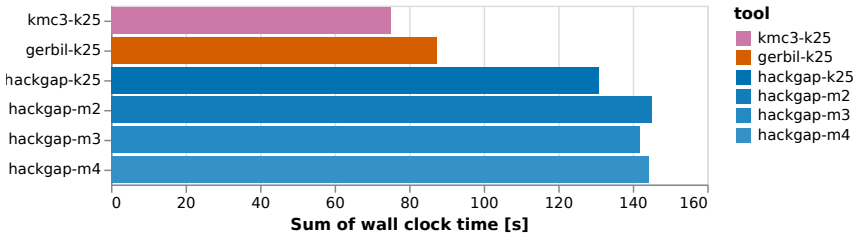


Figure 5 Wall clock times to count 25-mers in the t2t genome for different tools, and, using our `hackgap` tool only, for different $(w, k) = (31, 25)$-shaped masks m_2, m_3, m_4 (cf. Table 1).

4.2 Load Variation among Subtables

We first consider the question whether the partitioning of a large hash table into subtables leads to uneven loads in the subtables, i.e., whether using simple affine invertible hash functions as described above results in some of the subtables being (much) fuller than others. The performance of parallel k-mer counting is limited by the load of the fullest subtable.

Therefore, we chose 20 sets of random hash functions and distributed the 25-mers of the t2t genome into different numbers of subtables. We measured the *overhead*, which is the ratio between the size of the largest subtable and the mean size of all subtables, expressed in per cent above 100%. (A ratio of 1.0007 would be an overhead of 0.07%.) We expect a trend towards larger overheads for more subtables, just because of random fluctuations. The results in Figure 4 confirm this expectation, and they also show that the overhead is extremely small ($\leq 0.016\% = 1.6 \cdot 10^{-4}$) for up to 15 subtables.

4.3 Results on the t2t Genome

We counted the 25-mers in the t2t genome (from its gzipped FASTA file) using KMC3, Gerbil and `hackgap` and measured the wall clock running time. The reference contains a relatively small number of distinct 25-mers (2.4G), and most of the 25-mers occur only once (see also Section 4.4). The final hash table fits into 12.5 GB of memory (with a higher fill rate, it can even fit into 9 GB); each tool was given 16 GB of memory to work. Both KMC3 and Gerbil were given up to 16 cores, but did not use them for most of the time. We used 5 subtables for `hackgap` (6 threads).

The results can be seen in Figure 5. KMC3 was fastest, with Gerbil being almost as fast. Our tool `hackgap` took almost twice as long, but is the only one that supports gapped k-mers, which do not take much longer to count. One reason for the (comparatively) slower performance of `hackgap` may be that the FASTA reader is currently implemented in pure Python and not very efficient. Performance is better on FASTQ files (see Section 4.5 below).

4.4 Effect of Masks on Unique k-mers

As discussed in the introduction, unique k-mers (that appear only once in a reference genome and thus uniquely identify a location in the genome) are most helpful for alignment-free methods. An even stronger notion is the one of *strongly unique* k-mers; these are unique k-mers who additionally have no Hamming-distance 1 neighbor in the genome. Formally, a genomic k-mer $u \in \Sigma^k$ is strongly unique if $count(u) = 1$ and $d_{\text{Hamming}}(u, v) \geq 2$ and $d_{\text{Hamming}}(u, rc(v)) \geq 2$ for all genomic k-mers $v \neq u$. Strong uniqueness of u can be checked by computing the canonical codes of all $3k$ Hamming-distance-1 neighbors of u and querying the hash table for them, but there are also faster methods [24].

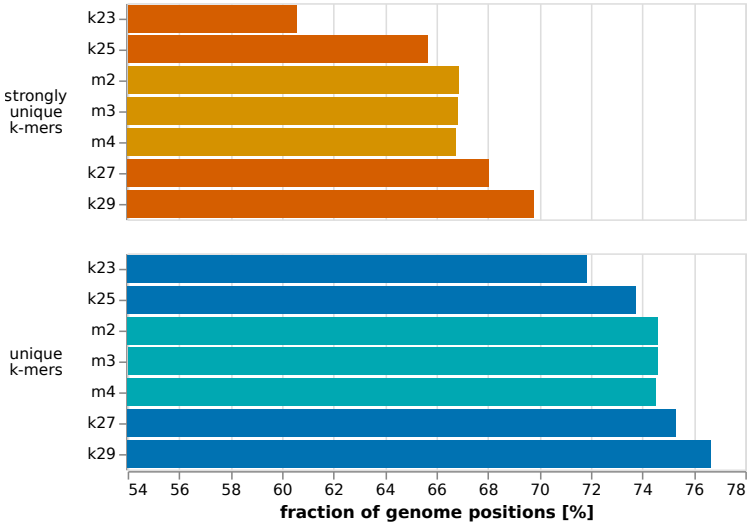


Figure 6 Fraction of genomic starting positions of *unique* and *strongly unique* k-mers begin, for different values of k and a selection of $(w, k) = (31, 25)$-shaped masks.

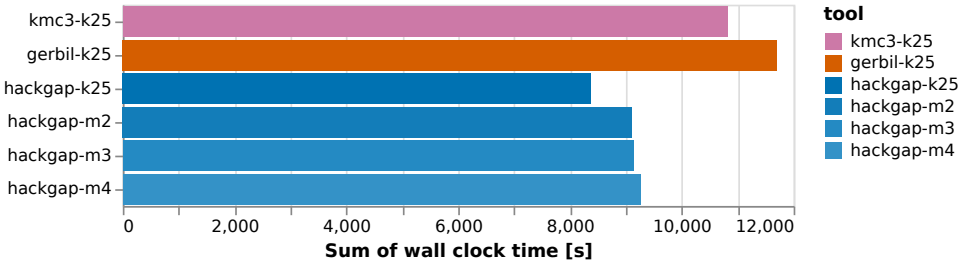


Figure 7 Wall clock times to count 25-mers in whole genome samples of 10 Göttingen minipigs (FASTQ, uncompressed) for KMC3 and Gerbild (given 16 GB RAM and up to 24 threads), and, using our `hackgap` tool only (given 16 GB RAM and using 5 subtables or 7 threads), for different $(31, 25)$-shaped masks, as given in Table 1.

We investigated the effect of varying k and using gapped k-mers with different masks (Table 1) on the fraction of positions in the genome where we find (strongly) unique k-mers. The results are shown in Figure 6. Clearly, the fraction of unique and strongly unique k-mers increases with k. Moreover, the three $(31, 25)$-shaped masks achieve a higher fraction of (strongly) unique genome positions than for the contiguous 25-mers (but less than 27-mers).

Currently, `hackgap` is the only k-mer counter that efficiently supports identifying strongly unique gapped k-mers.

4.5 Results on Minipig Whole Genome Samples

We counted 25-mers in each of 10 Göttingen minipig whole genome samples (unzipped FASTQ files) using KMC3, Gerbil and `hackgap` and measured the wall clock running time. Each sample had an average coverage of approximately 15x and thus contained many repeated k-mers, but also several unique k-mers (due to sequencing errors or technical artefacts).

The final hash table fits into 14 GB of memory for each minipig sample, so each tool was given 16 GB of memory. Both KMC3 and Gerbil were given up to 24 threads, but did not use them for most of the time. We used 5 subtables for `hackgap` (7 threads).

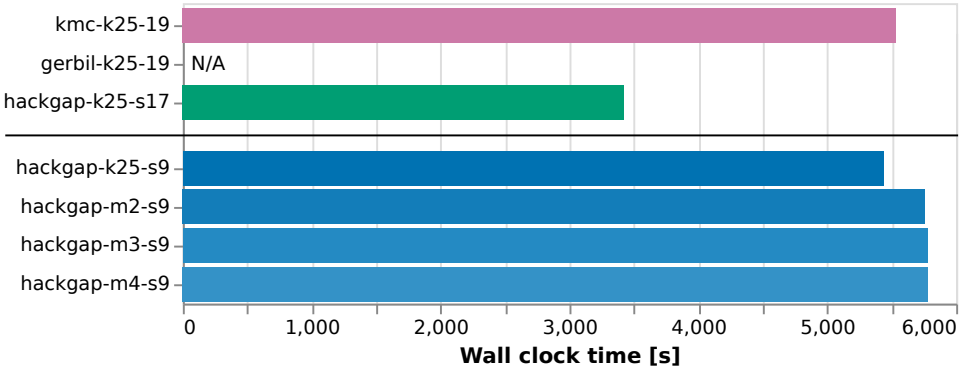


Figure 8 Wall clock times to count 25-mers in a combined whole genome sample of 3 members of an Ashkenazim family (gzipped FASTQ) for different tools, and, using our `hackgap` tool only, for different $(31, 25)$-shaped masks, as given in Table 1. Each tool was given 55 GB of RAM. On all of our test machines, Gerbil exited with an "Unknown error".

The results can be seen in Figure 7. Surprisingly, `hackgap` was the fastest tool overall and also on each single dataset. Counting gapped k-mers takes slightly longer for `hackgap` than counting contiguous k-mers (due to some optimizations not being applicable), but was still faster than KMC3 or Gerbil. This makes `hackgap` the fastest k-mer counter on medium-sized FASTQ files (whose k-mer hash table fits into main memory) and the only one that can count gapped k-mers efficiently.

On the other hand, due to our "busy waiting" (spin lock) implementation, our approach currently consumes unnecessarily much CPU time (details in Table 4) in Appendix C).

4.6 Results on a Large Human Whole Genome Sample

We furthermore picked a large dataset, the Genome-in-a-bottle Ashkenazim trio (three individuals of one family, HG002, HG003, HG004) and counted 25-mers for all three samples together. The resulting hash table barely fits into main memory (with the operating system and necessary systems processes running) and needs 55 GB. The KMC output files even need 68 GB. Gerbil was unable to process the dataset and exited with an "Unknown Error" on all of our test machines.

The results can be seen in Figure 8. With 17 subtables (19 active threads), we count everything in less than 1 hour (even less than 3500 s) wall-clock time, where KMC3 needs 5500 s. When we used gapped 25-mers, we do not benefit from more than 9 subtables (11 threads; the k-mer encoder becomes the bottleneck), but we achieve running times comparable to that of KMC3 with 19 threads. Using gapped k-mers needs approximately 5% more time than using contiguous k-mers.

We additionally ran KMC3 in "memory-only" mode (on a different server with 1 TB of memory). This mode does not write temporary files to disk, and on this data set saved 14% of the running time, but required 433 GB of RAM, whereas our approach is "memory-only" by design, and the allocated 55 GB are sufficient.

5 Discussion and Conclusion

Summary of contributions. We have demonstrated that, in contrast to common belief, efficient k-mer counting does not need to depend on the concepts of minimizers and super-k-mers. Three-way subdivided bucketed Cuckoo hash tables can show the same performance,

if used with appropriate parameters. Having such an alternative paves the way for efficient gapped k-mer counting, for which we provide one of the first efficient tools written in partially jit-compiled Python. Surprisingly, it can even be slightly faster than two of the fastest known k-mer counters (KMC3 and Gerbil) on large gzipped FASTQ datasets.

Using our approach, the running time for gapped k-mers is consistently slightly slower than for contiguous k-mers (by approximately 5% to 10%). This observation can be explained by the more involved integer encoding of a sequence of gapped k-mers compared to a sequence of contiguous k-mers, where two adjacent k-mers overlap by $k-1$ characters and efficient bit-shifting rolling hash functions can be (and are) used. However, this appears to be a relatively small cost for the gained ability to work with gapped k-mers. Also, the increase in time may be relatively small because the mask is a compile-time constant for the jit-compilation during runtime, the compiler can unroll the loop over the significant positions.

On a technical level, we have parallelized Cuckoo hash tables by subdividing them into subtables and designing the hash functions such that all possible locations of a key lie in the same subtable. We also showed that the keys are evenly distributed among the subtables (for an odd number of subtables), even with very simple affine (and even linear) hash functions.

Advantages and limitations of different approaches. Our implementation currently has a number of limitations, some of which we hope to remove in the near future.

Tools like KMC3 and Gerbil that use temporary files whose size is only limited by the available hard drive space do not need to know the number of distinct k-mers beforehand. Our approach, however, needs to allocate a hash table of sufficient size from the beginning; so we need at least a good estimate of the number of distinct k-mers to store. This may be an issue on a completely unknown dataset, but on a mammalian genome dataset, we can usually get a good estimate from the FASTQ file sizes. An alternative is to use a fast cardinality estimation tool like ntCard [15].

Our implementation is currently limited to $k \leq 32$ because we internally use 64-bit integers. For most applications in genomics, this should be sufficient, but in particular Gerbil efficiently supports much larger values of k.

The fact that our method is not disk-based (i.e., writes no temporary files) can be seen as advantage or disadvantage: The number of distinct k-mers we can store is limited by the main memory size. (The FASTQ datasets can be huge, as long as the number of distinct k-mers stays bounded.) In contrast, both KMC and Gerbil use large temporary space on the hard drive (KMC up to 350 GB for the GIAB dataset). While this is not a problem with today's drive sizes, it contributes to drive wear, which may especially affect SSDs which allow a limited number of writes during their lifetime.

If the dataset contains several input files, both KMC and Gerbil may read them in parallel. Our file IO is currently implemented in pure Python and becomes a bottleneck. We observed that we can feed up to 17 threads with contiguous k-mers, but only 9 threads with gapped k-mers, where our k-mer processor must do more work to compute canonical codes from the input sequence buffer. On the other hand, our input can be any stream, such as an anonymous pipe, while KMC3 and Gerbil only accept regular files.

Perspectives. We believe that the subdivided Cuckoo hashing approach may be further improved, as suggested above. The most important bottleneck currently is FASTA and FASTQ file I/O, which is solved much better in KMC3 and Gerbil and one of many reasons of their high speed. Further ongoing work concerns the implementation of pre-filtering steps to catch a large number of rare k-mers, which consume a large amount of memory in our hash

table but typically are of no interest to the user. Lastly, we are investigating the possibility of developing a hybrid method between Cuckoo hashing and minimizers and how the concept of minimizers may be transferred to gapped k-mers.

References

1 A. Bankevich, A. V. Bzikadze, M. Kolmogorov, D. Antipov, and P. A. Pevzner. Multiplex de Bruijn graphs enable genome assembly from long, high-fidelity reads. *Nat Biotechnol*, February 2022.

2 N. L. Bray, H. Pimentel, P. Melsted, and L. Pachter. Near-optimal probabilistic RNA-seq quantification. *Nat. Biotechnol.*, 34(5):525–527, May 2016. Erratum in Nat. Biotechnol. 34(8):888 (2016).

3 Karel Břinda, Maciej Sykulski, and Gregory Kucherov. Spaced seeds improve k-mer-based metagenomic classification. *Bioinformatics*, 31(22):3584–3592, July 2015. doi:10.1093/bioinformatics/btv419.

4 Andrea Califano and Isidore Rigoutsos. FLASH: a fast look-up algorithm for string homology. In Lawrence Hunter, David B. Searls, and Jude W. Shavlik, editors, *Proceedings of the 1st International Conference on Intelligent Systems for Molecular Biology, Bethesda, MD, USA, July 1993*, pages 56–64. AAAI, 1993. URL: http://www.aaai.org/Library/ISMB/1993/ismb93-007.php.

5 S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, May 2015.

6 M. Erbert, S. Rechner, and M. Müller-Hannemann. Gerbil: a fast and memory-efficient k-mer counter with GPU-support. *Algorithms Mol Biol*, 12:9, 2017.

7 Lars Hahn, Chris-André Leimeister, Rachid Ounit, Stefano Lonardi, and Burkhard Morgenstern. rasbhari: Optimizing spaced seeds for database searching, read mapping and alignment-free sequence comparison. *PLoS Comput. Biol.*, 12(10), 2016. doi:10.1371/journal.pcbi.1005107.

8 M. Kokot, M. Dlugosz, and S. Deorowicz. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, September 2017.

9 Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a LLVM-based python JIT compiler. In Hal Finkel, editor, *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015*, pages 7:1–7:6. ACM, 2015. doi:10.1145/2833157.2833162.

10 J. Lu, F. P. Breitwieser, P. Thielen, and S. L. Salzberg. Bracken: estimating species abundance in metagenomics data. *PeerJ Computer Science*, 3:e104, 2017. doi:10.7717/peerj-cs.104.

11 Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *Bioinform.*, 18(3):440–445, 2002. doi:10.1093/bioinformatics/18.3.440.

12 Swati C Manekar and Shailesh R Sathe. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience*, 7(12), October 2018. giy125. doi:10.1093/gigascience/giy125.

13 Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, January 2011. doi:10.1093/bioinformatics/btr011.

14 H. Mohamadi, J. Chu, B. P. Vandervalk, and I. Birol. ntHash: recursive nucleotide hashing. *Bioinformatics*, 32(22):3492–3494, November 2016.

15 H. Mohamadi, H. Khan, and I. Birol. ntCard: a streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, 33(9):1324–1330, May 2017.

16 A. Müller, C. Hundt, A. Hildebrandt, T. Hankeln, and B. Schmidt. MetaCache: context-aware classification of metagenomic reads using minhashing. *Bioinformatics*, 33(23):3740–3748, December 2017.

17 L. Noé. Best hits of 11110110111: model-free selection and parameter-free sensitivity calculation of spaced seeds. *Algorithms Mol Biol*, 12:1, 2017.

18 Laurent Noé. Best hits of 11110110111: model-free selection and parameter-free sensitivity calculation of spaced seeds. *Algorithms Mol. Biol.*, 12(1):1:1–1:16, 2017. doi:10.1186/s13015-017-0092-1.

19 S. Nurk, S. Koren, A. Rhie, M. Rautiainen, A. V. Bzikadze, A. Mikheenko, M. R. Vollger, N. Altemose, L. Uralsky, A. Gershman, S. Aganezov, S. J. Hoyt, M. Diekhans, G. A. Logsdon, M. Alonge, S. E. Antonarakis, M. Borchers, G. G. Bouffard, S. Y. Brooks, G. V. Caldas, N. C. Chen, H. Cheng, C. S. Chin, W. Chow, L. G. de Lima, P. C. Dishuck, R. Durbin, T. Dvorkina, I. T. Fiddes, G. Formenti, R. S. Fulton, A. Fungtammasan, E. Garrison, P. G. S. Grady, T. A. Graves-Lindsay, I. M. Hall, N. F. Hansen, G. A. Hartley, M. Haukness, K. Howe, M. W. Hunkapiller, C. Jain, M. Jain, E. D. Jarvis, P. Kerpedjiev, M. Kirsche, M. Kolmogorov, J. Korlach, M. Kremitzki, H. Li, V. V. Maduro, T. Marschall, A. M. McCartney, J. McDaniel, D. E. Miller, J. C. Mullikin, E. W. Myers, N. D. Olson, B. Paten, P. Peluso, P. A. Pevzner, D. Porubsky, T. Potapova, E. I. Rogaev, J. A. Rosenfeld, S. L. Salzberg, V. A. Schneider, F. J. Sedlazeck, K. Shafin, C. J. Shew, A. Shumate, Y. Sims, A. F. A. Smit, D. C. Soto, I. Sović, J. M. Storer, A. Streets, B. A. Sullivan, F. Thibaud-Nissen, J. Torrance, J. Wagner, B. P. Walenz, A. Wenger, J. M. D. Wood, C. Xiao, S. M. Yan, A. C. Young, S. Zarate, U. Surti, R. C. McCoy, M. Y. Dennis, I. A. Alexandrov, J. L. Gerton, R. J. O'Neill, W. Timp, J. M. Zook, M. C. Schatz, E. E. Eichler, K. H. Miga, and A. M. Phillippy. The complete sequence of a human genome. *Science*, 376(6588):44–53, April 2022.

20 C. Reimer, C. J. Rubin, A. R. Sharifi, N. T. Ha, S. Weigend, K. H. Waldmann, O. Distl, S. D. Pant, M. Fredholm, M. Schlather, and H. Simianer. Analysis of porcine body size variation using re-sequencing data of miniature and large pigs. *BMC Genomics*, 19(1):687, September 2018.

21 Z. Sun, S. Huang, M. Zhang, Q. Zhu, N. Haiminen, A. P. Carrieri, Y. Vázquez-Baeza, L. Parida, H. C. Kim, R. Knight, and Y. Y. Liu. Challenges in benchmarking metagenomic profilers. *Nat Methods*, 18(6):618–626, June 2021.

22 Stefan Walzer. Load thresholds for cuckoo hashing with overlapping blocks. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, volume 107 of *LIPIcs*, pages 102:1–102:10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.ICALP.2018.102.

23 Jens Zentgraf and Sven Rahmann. Fast lightweight accurate xenograft sorting. In Carl Kingsford and Nadia Pisanti, editors, *20th International Workshop on Algorithms in Bioinformatics, WABI 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 172 of *LIPIcs*, pages 4:1–4:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.WABI.2020.4.

24 Jens Zentgraf and Sven Rahmann. Fast lightweight accurate xenograft sorting. *Algorithms Mol. Biol.*, 16(1):2, 2021. doi:10.1186/s13015-021-00181-w.

25 Jens Zentgraf, Henning Timm, and Sven Rahmann. Cost-optimal assignment of elements in genome-scale multi-way bucketed cuckoo hash tables. In Guy E. Blelloch and Irene Finocchi, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*, pages 186–198. SIAM, 2020. doi:10.1137/1.9781611976007.15.

26 J. M. Zook, D. Catoe, J. McDaniel, L. Vang, N. Spies, A. Sidow, Z. Weng, Y. Liu, C. E. Mason, N. Alexander, E. Henaff, A. B. McIntyre, D. Chandramohan, F. Chen, E. Jaeger, A. Moshrefi, K. Pham, W. Stedman, T. Liang, M. Saghbini, Z. Dzakula, A. Hastie, H. Cao, G. Deikus, E. Schadt, R. Sebra, A. Bashir, R. M. Truty, C. C. Chang, N. Gulbahce, K. Zhao, S. Ghosh, F. Hyland, Y. Fu, M. Chaisson, C. Xiao, J. Trow, S. T. Sherry, A. W. Zaranek, M. Ball, J. Bobe, P. Estep, G. M. Church, P. Marks, S. Kyriazopoulou-Panagiotopoulou, G. X. Zheng, M. Schnall-Levin, H. S. Ordonez, P. A. Mudivarti, K. Giorda, Y. Sheng, K. B. Rypdal, and M. Salit. Extensive sequencing of seven human genomes to characterize benchmark reference materials. *Sci Data*, 3:160025, June 2016.

A Dataset URLs

t2t human genome. `https://s3-us-west-2.amazonaws.com/human-pangenomics/T2T/CHM13/assemblies/analysis_set/chm13v2.0.fa.gz`

Minipigs. We use runs ERR2744277 through ERR2744286 from `https://www.ebi.ac.uk/ena/browser/view/PRJEB2765`

GIAB Ashkenazim Trio. We use the 6kb matepair data from `https://github.com/genome-in-a-bottle/giab_data_indexes/blob/master/AshkenazimTrio/sequence.index.AJtrio_Illumina_6kb_matepair_wgs_08032015`

B Executed Commands

In the command lines below, replace `{input}` with the name(s) of the (gzipped) FASTA / FASTQ / TXT input file. A text (TXT) file must contain one file name per line. Similarly, replace `{output}` by the output file name (a huge file containing a hash table or other data structure with k-mers and counts), and `{report.json}` (KMC only) by the name of a report file. Also, replace `{mask}` with the mask (strings in Table 1) (`hackgap` only) and `{tmpdir/}` with the name of a temporary directory (with up to 500 GB of working space).

B.1 t2t Human Genome Dataset

B.1.1 Time Benchmark

```
hackgap count  -v count 256 -n 3000000000 --nostats --fasta {input}\
    -p 6 --mask {mask} --subtables 5 -o {output}

kmc -v  -k25 -ci1 -m16 -sm -fm -t24 -j{report.json} @{input.txt}\
    {output} {tmpdir/}

gerbil -t 24 -i -k 25 -e 16G -l 1 {input.txt} {tmpdir/} {output}
```

B.1.2 Subtable Load Overhead

We ran the following command with 20 different combinations of hash functions; one(!) example is shown below. We varied the number of subtables between 3 and 15 (odd numbers only). Replace `{ST}` by the appropriate number.

```
hackgap count  -v count 3  -n 3000000000 --substatistics --fasta {input}\
    -p 6 --mask {mask} --subtables {ST} -o {output}\
    --hashfunctions linear73198857:linear11669733:linear79255563:linear56630389
```

B.1.3 Unique and Strongly Unique k-mers

The following command was executed for contiguous k-mers for $k \in \{23, 25, 27, 29\}$ and masks `m2, m3, m4` from Table 1.

```
hackgap count -v  count 3 --strong  -n 3000000000 --fasta {input}\
    -p 6 --mask {mask} --subtables 15 -o {output}
```

B.2 Minipig Data

The following commands were executed for each of the 10 minipig samples separately. To compare running times, the times were summed over all individual pigs. We used 5 subtables (7 threads) and made up to 24 threads available to KMC3 and Gerbil. The memory limit was set to 16 GB which conveniently fits the k-mers and their counts.

```
hackgap count -v count 256 -n 3_500_000_000 --nostats --fastq {input}\
    -p 6 --mask {mask} --subtables 5 -o {result}

kmc -v -k25 -ci1 -m16 -sm -fq -t24 -j{report.json} @{input.txt}\
    {output} {tmpdir/}

gerbil -t 24 -i -k 25 -e 16G -l 1 {input.txt} {tmpdir/} {output}
```

B.3 GIAB Dataset

We executed `hackgap` with both 17 (for ungapped 25-mers, comparison among tools) and 9 subtables (comparison among gapped and ungapped masks, `hackgap` only). The reason is that our k-mer processor can serve 17 inserter threads sufficiently fast when it processes ungapped k-mers but only 9 when it processes gapped k-mers because encoding takes more time.

We gave KMC and Gerbil the same number of active threads as for our `hackgap` (19 threads for 17 subtables). Unfortunately, gerbil crashed in each case on this dataset (although it processed the smaller datasets perfectly).

The memory limit was set to 55 GB, which is sufficient to hold our k-mer table with counts.

Command lines for 17 subtables (19 threads), contiguous k-mers.

```
hackgap count -v count 256  -n 14297937824 --fill 0.88 --nostats\
    --fastq {input} -p 7 -k 25 --subtables 17 -o {output}

kmc -v -k25 -ci1 -m55 -sm -fq -t19 -j{report.json} @{input.txt}
{output} {tmpdir/}

gerbil -t 19 -i -k 25 -e 55G -l 1 {input} {tmpdir/} {output}
```

Command lines for 9 subtables (11 threads).

```
hackgap count -v count 256  -n 14297937824 --fill 0.94 --nostats\
    --fastq {input} -p 10 --mask {mask} --subtables 9 -o {output}
```

C CPU time and memory usage

With a comparable number of threads, or even fewer threads, our tool `hackgap` achieves faster wall-clock running times than KMC3 or Gerbil.

However, we noted that KMC3 and Gerbil only rarely use all threads given to them. This actually results in considerable lower CPU times than (number of threads) × (wall-clock time), wheras for `hackgap` all threads are busy (and possibly busy waiting with spin locks) for

Table 4 CPU time and actually used memory in our experiments.

Tool	mask	Dataset	CPU time [s]	Memory [MB]
hackgap	m2	t2t	838.09	13147.30
hackgap	m3	t2t	784.31	13139.45
hackgap	m4	t2t	836.24	13311.99
hackgap	k25	t2t	700.02	13138.26
kmc	k25	t2t	300.97	14916.35
gerbil	k25	t2t	276.89	953.88
hackgap	m2	Minipig	56168.94	15084.040
hackgap	m3	Minipig	56341.69	15083.906
hackgap	m4	Minipig	57177.59	15084.407
hackgap	k25	Minipig	51115.41	15083.565
kmc	k25	Minipig	15927.13	10156.220
gerbil	k25	Minipig	28691.98	2412.909
hackgap-s9	m2	GIAB	58216.93	56500.38
hackgap-s9	m3	GIAB	58482.28	56502.34
hackgap-s9	m4	GIAB	58439.08	56499.67
hackgap-s9	k25	GIAB	54730.04	56499.55
hackgap-s17	k25	GIAB	56847.42	56540.94
kmc	k25	GIAB	10913.06	52422.50
gerbil	k25	GIAB	error	error

most of the time. The measured numbers are given in Table 4. However, we must point out, although our CPU times seem comparatively high, the busy waiting uses the energy-efficient SSE2 `pause` instruction (x86_64, amd64) or spin lock `hint` instruction (arm64), so the energy consumption is less than that of an actually busy CPU.

We also noted that KMC3 tends to use all of the memory assigned to it, Gerbil uses much less memory (as reported by `/usr/bin/time -v`, maximum resident set size).

A Linear Time Algorithm for an Extended Version of the Breakpoint Double Distance

Marília D. V. Braga[1]
Faculty of Technology and Center for Biotechnology (CeBiTec), Bielefeld University, Germany

Leonie R. Brockmann
Faculty of Technology and Center for Biotechnology (CeBiTec), Bielefeld University, Germany

Katharina Klerx
Faculty of Technology and Center for Biotechnology (CeBiTec), Bielefeld University, Germany

Jens Stoye
Faculty of Technology and Center for Biotechnology (CeBiTec), Bielefeld University, Germany

Abstract

Two genomes over the same set of gene families form a *canonical* pair when each of them has exactly one gene from each family. A genome is *circular* when it contains only circular chromosomes. Different distances of canonical circular genomes can be derived from a structure called *breakpoint graph*, which represents the relation between the two given genomes as a collection of cycles of even length. Then, the breakpoint distance is equal to $n - c_2$, where n is the number of genes and c_2 is the number of cycles of length 2. Similarly, when the considered rearrangements are those modeled by the *double-cut-and-join* (DCJ) operation, the rearrangement distance is $n - c$, where c is the total number of cycles.

The distance problem is a basic unit for several other combinatorial problems related to genome evolution and ancestral reconstruction, such as *median* or *double distance*. Interestingly, both median and double distance problems can be solved in polynomial time for the breakpoint distance, while they are NP-hard for the rearrangement distance. One way of exploring the complexity space between these two extremes is to consider a σ_k distance, defined to be $n - (c_2 + c_4 + \ldots + c_k)$, and increasingly investigate the complexities of median and double distance for the σ_4 distance, then the σ_6 distance, and so on. While for the median much effort was done in our and in other research groups but no progress was obtained even for the σ_4 distance, for solving the double distance under σ_4 and σ_6 distances we could devise linear time algorithms, which we present here.

2012 ACM Subject Classification Applied computing → Bioinformatics

Keywords and phrases Comparative genomics, genome rearrangement, breakpoint distance, double-cut-and-join (DCJ) distance, double distance

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.13

Acknowledgements We would like to thank Cedric Chauve for bringing our attention to the class of σ_k distances as a means for studying the hardness bound between the breakpoint distance and the DCJ distance in combinatorial problems related to genome evolution. Thanks also to Eloi Araujo, Daniel Doerr and Fábio H. V. Martinez for helping us studying the median problem under this class.

1 Introduction

In genome comparison, the most elementary problem is that of computing a *distance* between two given *genomes* [10], each one being a set of *chromosomes*. Usually a high-level view of a chromosome is adopted, in which each chromosome is represented by a sequence of oriented *genes* and the genes are classified into *families*. The simplest model in this setting

[1] Corresponding author

is the *breakpoint* model, whose distance consists of somehow quantifying the points of dissimilarity between the two genomes, a *point* in a genome being the oriented neighborhood between two genes in one of its chromosomes [11]. Other models rely on large-scale genome *rearrangements*, such as inversions, translocations, fusions and fissions, yielding distances that correspond to the minimum number of rearrangements required to transform one genome into another [6, 7, 12].

Independently of the underlying model, the distance problem is a basic unit for several other combinatorial problems related to genome evolution and ancestral reconstruction [11]. The *median* problem, for example, has three genomes as input and asks for an ancestor genome that minimizes the sum of its distances to the three given genomes. Other models are related to the *whole genome duplication* (WGD) event [5]. Let the *doubling* of a genome duplicate each of its chromosomes. The *double distance* is the problem that has a *duplicated* genome and a *singular* genome as input and computes the distance between the former and a doubling of the latter. The *halving* problem has a duplicated genome as input and asks for a singular genome whose double distance to the given duplicated genome is minimized. Finally, the *guided halving* problem has a duplicated and a singular genome as input and asks for another singular genome that minimizes the sum of its double distance to the given duplicated genome and its distance to the given singular genome.

In this work, we assume that all considered genomes contain only circular chromosomes and are therefore *circular*. Our study relies on the *breakpoint graph*, a structure that represents the relation between two given genomes [2]. When the two genomes are over the same set of gene families and form a *canonical* pair, that is, when each of them has exactly one gene from each family, their breakpoint graph is a collection of cycles of even length. If we call *k-cycle* a cycle of length k and assume that both genomes have n genes, their breakpoint distance is equal to $n - c_2$, where c_2 is the number of 2-cycles [11]. Similarly, when the considered rearrangements are those modeled by the *double-cut-and-join* (DCJ) operation [12], the rearrangement distance is $n - c$, where c is the total number of cycles [3].

While the halving problem under both breakpoint and rearrangement distances can be solved in polynomial time [1, 5, 9, 11], median, double distance and guided halving problems can be solved in polynomial time only under the breakpoint distance, but are NP-hard under the rearrangement distance [11]. One way of exploring the complexity space between these two extremes is to consider a σ_k distance [4], defined to be $n - (c_2 + c_4 + \ldots + c_k)$, and increasingly investigate the complexities of median, guided halving and double distance under the σ_4 distance, then under the σ_6 distance, and so on. Note that the σ_2 distance is the breakpoint distance and the σ_∞ distance is the DCJ distance. To the best of our knowledge, the guided halving problem has not been studied for this class of problems, while for the median under σ_4 distance much effort has been done in our group and in other research groups (e.g. [4]) but no progress was obtained so far.

In contrast, for the double distance, while σ_8 and higher were not yet studied, we succeeded in devising efficient algorithms for σ_4 and σ_6. Our results, which we present here, are built on a variation of the breakpoint graph, called *ambiguous breakpoint graph* [11] and have three main parts. First we show that in any σ_k double distance, including the NP-hard DCJ double distance, all 2-cycles are fulfilled, meaning that the common adjacencies between the compared genomes are always preserved. Then we show that the σ_4 double distance can be computed by a greedy linear time algorithm. Finally we present a non-greedy but still linear time algorithm for the σ_6 double distance.

Recall that the results we present in this work consider only circular genomes, for which the underlying breakpoint graph is more regular and composed of cycles only. With linear chromosomes the graph also includes paths that may be of odd or of even length. Often the

studies of genomic distances for genomes with linear chromosomes can be adapted to circular genomes and *vice-versa* [8], and we believe the same could be done for the problems that we study here, as we discuss in the end of the paper.

2 Definitions and background

A *chromosome* is an oriented DNA molecule and can be either linear or circular. We represent a chromosome by its sequence of genes, where each *gene* is an oriented DNA fragment. We assume that each gene belongs to a *family*, which is a set of homologous genes. A gene that belongs to a family $\mathtt{X}$ is represented by the symbol $\mathtt{X}$ itself if it is read in direct orientation or by the symbol $\overline{\mathtt{X}}$ if it is read in reverse orientation. For example, the circular string $(\mathtt{1}\,\overline{\mathtt{3}}\,\mathtt{2})$ (flanked by parentheses) represents a circular chromosome K, shown in Figure 1, composed of three genes from the families $\mathtt{1}$, $\mathtt{2}$ and $\mathtt{3}$. Note that K can be equally represented by any circular rotation of the given string and additionally by $(\overline{\mathtt{2}}\,\mathtt{3}\,\overline{\mathtt{1}})$ or any of its circular rotations.

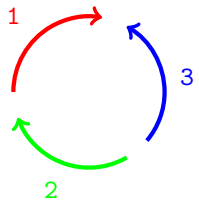


Figure 1 Graphical representation of circular chromosome $K = (\mathtt{1}\,\overline{\mathtt{3}}\,\mathtt{2})$.

We can also represent a gene from family $\mathtt{X}$ referring to its *extremities* $\mathtt{X}^h$ (head) and $\mathtt{X}^t$ (tail). For example, we could represent the circular chromosome K above by $(\mathtt{1}^t\mathtt{1}^h\mathtt{3}^h\mathtt{3}^t\mathtt{2}^t\mathtt{2}^h)$ or $(\mathtt{2}^h\mathtt{2}^t\mathtt{3}^t\mathtt{3}^h\mathtt{1}^h\mathtt{1}^t)$, or by any of their circular rotations. Recall that a gene is an *occurrence* of a family, therefore distinct genes from the same family are represented by the same symbol. The *adjacencies* in a chromosome are the neighboring extremities of distinct genes. In the given example, the adjacencies in K are $\{\mathtt{1}^h\mathtt{3}^h, \mathtt{3}^t\mathtt{2}^t, \mathtt{2}^h\mathtt{1}^t\}$. Note that an adjacency has no orientation, that is, an adjacency between extremities $\mathtt{1}^h$ and $\mathtt{3}^h$ can be equally represented by $\mathtt{1}^h\mathtt{3}^h$ and by $\mathtt{3}^h\mathtt{1}^h$. In the particular case of a single-gene circular chromosome, e.g. $(\mathtt{4})$, an adjacency exceptionally occurs between the extremities of the same gene (here $\mathtt{4}^h\mathtt{4}^t$).

A *genome* is then a set of chromosomes and we denote by $\mathcal{F}(\mathbb{G})$ the set of gene families that occur in the chromosomes of genome $\mathbb{G}$. In addition, we denote by $\mathcal{A}(\mathbb{G})$ the multiset of adjacencies that occur in the chromosomes of $\mathbb{G}$. A genome $\mathbb{S}$ is called *singular* if each gene family occurs exactly once in $\mathbb{S}$. Similarly, a genome $\mathbb{D}$ is called *duplicated* if each gene family occurs exactly twice in $\mathbb{D}$. The two occurrences of a family in a duplicated genome are called *paralogs*. A *doubled* genome is a special type of duplicated genome in which each adjacency occurs exactly twice. These two copies of the same adjacency in a doubled genome are called *paralogous adjacencies*. Observe that distinct doubled genomes can have exactly the same adjacencies, as we show in Table 1, where we also give examples of singular and duplicated genomes.

2.1 Comparing canonical genomes

Two genomes $\mathbb{S}_1$ and $\mathbb{S}_2$ are said to be a *canonical pair* when they are singular and $\mathcal{F}(\mathbb{S}_1) = \mathcal{F}(\mathbb{S}_2)$, that is, when singular genomes $\mathbb{S}_1$ and $\mathbb{S}_2$ have exactly the same gene families. Denote by $\mathcal{F}_*$ the set of families occurring in canonical genomes $\mathbb{S}_1$ and $\mathbb{S}_2$. For example, genomes $\mathbb{S}_1 = \{(\mathtt{1}\,\overline{\mathtt{3}}\,\mathtt{2})\,(\mathtt{4})\}$ and $\mathbb{S}_2 = \{(\mathtt{1}\,\mathtt{2})\,(\mathtt{3}\,\overline{\mathtt{4}})\}$ are canonical with $\mathcal{F}_* = \{\mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}\}$.

Table 1 Examples of a singular, a duplicated and two doubled genomes, with their sets of families and their multisets of adjacencies. Note that the doubled genomes $\mathbb{B}_1$ and $\mathbb{B}_2$ have exactly the same adjacencies.

Singular genome (each family occurs once)	$\mathbb{S} = \{(1\,\overline{3}\,2)\,(4)\}$	$\begin{cases}\mathcal{F}(\mathbb{S}) = \{1,2,3,4\}\\ \mathcal{A}(\mathbb{S}) = \{1^h3^h, 3^t2^t, 2^h1^t, 4^h4^t\}\end{cases}$
Duplicated genome (each family occurs twice)	$\mathbb{D} = \{(1\,2\,\overline{3}\,1\,\overline{3}\,2)\}$	$\begin{cases}\mathcal{F}(\mathbb{D}) = \{1,2,3\}\\ \mathcal{A}(\mathbb{D}) = \{1^h2^t, 2^h3^h, 3^t1^t, 1^h3^h, 3^t2^t, 2^h1^t\}\end{cases}$
Doubled genomes (each adj. occurs twice)	$\mathbb{B}_1 = \{(1\,2\,3)\,(1\,2\,3)\}$ $\mathbb{B}_2 = \{(1\,2\,3\,1\,2\,3)\}$	$\begin{cases}\mathcal{F}(\mathbb{B}_i) = \{1,2,3\}\\ \mathcal{A}(\mathbb{B}_i) = \{1^h2^t, 2^h3^t, 3^h1^t, 1^h2^t, 2^h3^t, 3^h1^t\}\end{cases}$

2.1.1 Breakpoint distance

A simple way of comparing canonical genomes $\mathbb{S}_1$ and $\mathbb{S}_2$ is by searching for their *common adjacencies*, which occur in both $\mathbb{S}_1$ and $\mathbb{S}_2$. For circular canonical genomes $\mathbb{S}_1$ and $\mathbb{S}_2$ the *breakpoint distance*, denoted by $\mathrm{d_{BP}}$, can be computed as follows [11]:

$$\mathrm{d_{BP}}(\mathbb{S}_1, \mathbb{S}_2) = n_* - |\mathcal{A}(\mathbb{S}_1) \cap \mathcal{A}(\mathbb{S}_2)|, \quad \text{where } n_* = |\mathcal{F}_*|.$$

For $\mathbb{S}_1 = \{(1\,\overline{3}\,2)\,(4)\}$ and $\mathbb{S}_2 = \{(1\,2)\,(3\,\overline{4})\}$, the common adjacencies are $\mathcal{A}(\mathbb{S}_1) \cap \mathcal{A}(\mathbb{S}_2) = \{1^t2^h\}$. Since $n_* = 4$, their breakpoint distance is $\mathrm{d_{BP}}(\mathbb{S}_1, \mathbb{S}_2) = 3$.

2.1.2 DCJ distance and breakpoint graph

Another way of comparing two genomes is by searching for the mininum number of rearrangements transforming one genome into the other. A very useful model for this task is called *double cut and join* (DCJ) [12]. Basically, given a genome, a DCJ is the operation that breaks two adjacencies and rejoins the open extremities in a different way.

For example, consider the chromosome $K = (1\,2\,3\,4)$ and a DCJ that cuts K between genes 1 and 2 and between genes 3 and 4, creating segments $\bullet 2\,3\bullet$ and $\bullet 4\,1\bullet$ (where the symbols $\bullet$ represent the open ends). If we join the first with the third and the second with the fourth open end, we get $K' = (1\,\overline{3}\,\overline{2}\,4)$, that is, the described DCJ operation is an inversion transforming K into K'. Besides inversions, in circular genomes a DCJ operation can also represent a circular fission or a circular fusion. The *DCJ distance* $\mathrm{d_{DCJ}}$ is then the minimum number of DCJ operations that transform one genome into the other.

The DCJ distance can be easily computed with the help of the following structure. Given two canonical circular genomes $\mathbb{S}_1$ and $\mathbb{S}_2$, their *breakpoint graph* $BG(\mathbb{S}_1, \mathbb{S}_2) = (V, E)$ is a multigraph representing the adjacencies of $\mathbb{S}_1$ and $\mathbb{S}_2$ [2]. The vertex set V comprises, for each family $\mathtt{X}$ in $\mathcal{F}_*$, one vertex for the extremity $\mathtt{X}^h$ and one vertex for the extremity $\mathtt{X}^t$. The edge multiset E represents the adjacencies. For each adjacency in $\mathbb{S}_1$ there exists one $\mathbb{S}_1$-edge in E linking its two extremities. Similarly, for each adjacency in $\mathbb{S}_2$ there exists one $\mathbb{S}_2$-edge in E linking its two extremities. An example is given in Figure 2.

The breakpoint graph of canonical circular genomes is a simple collection of cycles of even length, where each cycle alternates between $\mathbb{S}_1$-edges and $\mathbb{S}_2$-edges. We call *k-cycle* a cycle of length k. Note that a common adjacency of $\mathbb{S}_1$ and $\mathbb{S}_2$ corresponds to a 2-cycle in $BG(\mathbb{S}_1, \mathbb{S}_2)$. Furthermore, if $\mathbb{S}_1 = \mathbb{S}_2$, their breakpoint graph is composed of exactly n_* 2-cycles. Otherwise, if $\mathbb{S}_1 \neq \mathbb{S}_2$, their breakpoint graph is composed of $c < n_*$ cycles. It has

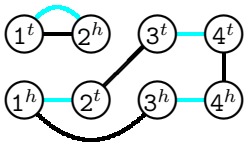


Figure 2 Breakpoint graph of genomes $\mathbb{S}_1 = \{ (1\,\overline{3}\,2)\,(4) \}$ and $\mathbb{S}_2 = \{ (1\,2)\,(3\,\overline{4}) \}$. The colors distinguish the edge types: $\mathbb{S}_1$-edges are drawn in black and $\mathbb{S}_2$-edges are drawn in blue.

been shown that one DCJ operation can create at most one new cycle. Such a DCJ is called a *split DCJ*. Moreover, it was proven that it is possible to transform one genome into another with split DCJs only. Therefore, the DCJ distance of two genomes $\mathbb{S}_1$ and $\mathbb{S}_2$ can be directly derived from their breakpoint graph [3]:

$$\mathrm{d}_{\mathrm{DCJ}}(\mathbb{S}_1, \mathbb{S}_2) = n_* - c, \quad \text{where } n_* = |\mathcal{F}_*| \text{ and } c \text{ is the number of cycles in } BG(\mathbb{S}_1, \mathbb{S}_2).$$

If $\mathbb{S}_1 = \{(1\,\overline{3}\,2)\,(4)\}$ and $\mathbb{S}_2 = \{(1\,2)\,(3\,\overline{4})\}$, then $n_* = 4$ and $c = 2$ (see Figure 2). Consequently, their DCJ distance is $\mathrm{d}_{\mathrm{DCJ}}(\mathbb{S}_1, \mathbb{S}_2) = 2$.

2.1.3 The class of σ_k distances

For $k = 2, 4, 6, \ldots$, we denote by c_k the number of k-cycles in $BG(\mathbb{S}_1, \mathbb{S}_2)$ and by σ_k the cumulative sums $\sigma_k = c_2 + c_4 + \ldots + c_k$. Since a common adjacency of genomes $\mathbb{S}_1$ and $\mathbb{S}_2$ corresponds to a 2-cycle in $BG(\mathbb{S}_1, \mathbb{S}_2)$, their breakpoint distance can be rewritten as

$$\mathrm{d}_{\mathrm{BP}}(\mathbb{S}_1, \mathbb{S}_2) = n_* - c_2 = n_* - \sigma_2.$$

Similarly, since we have $c = c_2 + c_4 + \ldots + c_\infty$, the DCJ distance can be rewritten as

$$\mathrm{d}_{\mathrm{DCJ}}(\mathbb{S}_1, \mathbb{S}_2) = n_* - (c_2 + c_4 + \ldots + c_\infty) = n_* - \sigma_\infty.$$

Generalizing the formulas above, we can define the class σ_k-DIST of σ_k distances of two canonical circular genomes $\mathbb{S}_1$ and $\mathbb{S}_2$, for $k = 2, 4, 6, \ldots, \infty$, as follows:

$$\mathrm{d}_{\sigma_k}(\mathbb{S}_1, \mathbb{S}_2) = n_* - (c_2 + c_4 + \ldots + c_k) = n_* - \sigma_k.$$

2.2 Comparing a singular to a duplicated genome

Let $\mathbb{S}$ be a circular singular genome and $\mathbb{D}$ be a circular duplicated genome such that $\mathcal{F}(\mathbb{S}) = \mathcal{F}(\mathbb{D})$. Note that the number of adjacencies in $\mathbb{D}$ is twice the number of adjacencies in $\mathbb{S}$. In order to search for common adjacencies of $\mathbb{S}$ and $\mathbb{D}$ or to transform one genome into the other with DCJ operations, we need to somehow equalize the contents of these genomes. This can be done by *doubling* the singular genome $\mathbb{S}$. This rearrangement operation mimics a *whole genome duplication* of $\mathbb{S}$ and consists of doubling each adjacency of $\mathbb{S}$. However, when $\mathbb{S}$ is circular, it is not possible to find a unique layout of its chromosomes after the doubling: indeed, each circular chromosome of $\mathbb{S}$ can be doubled into two identical circular chromosomes, or the two copies are concatenated to each other in a single circular chromosome. Therefore, the doubling of a circular genome $\mathbb{S}$ results in a set of doubled genomes denoted by $2{\cdot}\mathbb{S}$. Note that $|2{\cdot}\mathbb{S}| = 2^r$, where r is the number of (circular) chromosomes in $\mathbb{S}$. For example, if $\mathbb{S} = \{(1\,2\,3)\}$, then $2{\cdot}\mathbb{S} = \{\mathbb{B}_1, \mathbb{B}_2\}$ with $\mathbb{B}_1 = \{(1\,2\,3)\,(1\,2\,3)\}$ and $\mathbb{B}_2 = \{(1\,2\,3\,1\,2\,3)\}$. Since all genomes in $2{\cdot}\mathbb{S}$ have exactly the same multiset of adjacencies, we can refer to its adjacencies as $\mathcal{A}(2{\cdot}\mathbb{S}) = \mathcal{A}(\mathbb{B})$ where $\mathbb{B}$ is any doubled genome in $2{\cdot}\mathbb{S}$.

Each family in a duplicated genome can be $\binom{\mathtt{a}}{\mathtt{b}}$-*singularized* by adding the index $\mathtt{a}$ to one of its occurrences and the index $\mathtt{b}$ to the other. A duplicated genome can be entirely singularized if each of its families is singularized. Let $\mathfrak{S}^{\mathtt{a}}_{\mathtt{b}}(\mathbb{D})$ be the set of all possible genomes obtained by all distinct ways of $\binom{\mathtt{a}}{\mathtt{b}}$-singularizing the duplicated genome $\mathbb{D}$. Similarly, we denote by $\mathfrak{S}^{\mathtt{a}}_{\mathtt{b}}(2{\cdot}\mathbb{S})$ the set of all possible genomes obtained by all distinct ways of $\binom{\mathtt{a}}{\mathtt{b}}$-singularizing each doubled genome in the set $2{\cdot}\mathbb{S}$.

2.2.1 Breakpoint double distance

The *breakpoint double distance* of $\mathbb{S}$ and $\mathbb{D}$, denoted by $\mathrm{d}^2_{\mathrm{BP}}(\mathbb{S},\mathbb{D})$, is defined as follows [11]:

$$\mathrm{d}^2_{\mathrm{BP}}(\mathbb{S},\mathbb{D}) = \mathrm{d}^2_{\mathrm{BP}}(\mathbb{S},\check{\mathbb{D}}) = \min_{\mathbb{B}\in\mathfrak{S}^{\mathtt{a}}_{\mathtt{b}}(2\cdot\mathbb{S})}\{\mathrm{d}_{\mathrm{BP}}(\mathbb{B},\check{\mathbb{D}})\},\ \text{where } \check{\mathbb{D}} \text{ is any genome in } \mathfrak{S}^{\mathtt{a}}_{\mathtt{b}}(\mathbb{D}).$$

Observe that $\mathrm{d}^2_{\mathrm{BP}}(\mathbb{S},\check{\mathbb{D}}) = \mathrm{d}^2_{\mathrm{BP}}(\mathbb{S},\check{\mathbb{D}}')$ for any $\check{\mathbb{D}},\check{\mathbb{D}}' \in \mathfrak{S}^{\mathtt{a}}_{\mathtt{b}}(\mathbb{D})$.

Although the search space of this optimization problem can be huge, the solution can be found easily with a greedy algorithm [11]: Each adjacency of $\mathbb{D}$ that occurs in $\mathbb{S}$ can be fulfilled. If an adjacency that occurs twice in $\mathbb{D}$ also occurs in $\mathbb{S}$, it can be fulfilled twice in any genome from $2{\cdot}\mathbb{S}$. Then,

$$\mathrm{d}^2_{\mathrm{BP}}(\mathbb{S},\mathbb{D}) = 2n_* - |\mathcal{A}(2{\cdot}\mathbb{S}) \cap \mathcal{A}(\mathbb{D})|,\ \text{where } n_* = |\mathcal{F}(\mathbb{S})|.$$

2.2.2 DCJ double distance and ambiguous breakpoint graph

Extending the ideas above to the DCJ model, the formulation of the *DCJ double distance* follows:

$$\mathrm{d}^2_{\mathrm{DCJ}}(\mathbb{S},\mathbb{D}) = \mathrm{d}^2_{\mathrm{DCJ}}(\mathbb{S},\check{\mathbb{D}}) = \min_{\mathbb{B}\in\mathfrak{S}^{\mathtt{a}}_{\mathtt{b}}(2\cdot\mathbb{S})}\{\mathrm{d}_{\mathrm{DCJ}}(\mathbb{B},\check{\mathbb{D}})\},\ \text{where } \check{\mathbb{D}} \text{ is any genome in } \mathfrak{S}^{\mathtt{a}}_{\mathtt{b}}(\mathbb{D}).$$

Here the solution space cannot be explored greedily. In fact, computing the DCJ double distance of circular genomes $\mathbb{S}$ and $\mathbb{D}$ is an NP-hard problem [11]. However, an interesting relation between its solutions and a modified breakpoint graph was established.

Given a singular genome $\mathbb{S}$ and a duplicated genome $\mathbb{D}$, their *ambiguous breakpoint graph* $ABG(\mathbb{S},\check{\mathbb{D}}) = (V,E)$ is a multigraph representing the adjacencies of any element in $\mathfrak{S}^{\mathtt{a}}_{\mathtt{b}}(2{\cdot}\mathbb{S})$ and a genome $\check{\mathbb{D}}$ in $\mathfrak{S}^{\mathtt{a}}_{\mathtt{b}}(\mathbb{D})$. The vertex set V comprises, for each family $\mathtt{X}$ in $\mathcal{F}(\mathbb{S})$, the two pairs of *paralogous vertices* $\mathtt{X}^h_{\mathtt{a}}$, $\mathtt{X}^h_{\mathtt{b}}$ and $\mathtt{X}^t_{\mathtt{a}}$, $\mathtt{X}^t_{\mathtt{b}}$. We can use the notation $\hat{u}$ to refer to the paralogous counterpart of a vertex u. For example, if $u = \mathtt{X}^h_{\mathtt{a}}$, then $\hat{u} = \mathtt{X}^h_{\mathtt{b}}$.

The edge set E represents the adjacencies. For each adjacency in $\check{\mathbb{D}}$ there exists one $\check{\mathbb{D}}$-edge in E linking its two extremities. The $\mathbb{S}$-edges represent all adjacencies occurring in all genomes from $\mathfrak{S}^{\mathtt{a}}_{\mathtt{b}}(2{\cdot}\mathbb{S})$: For each adjacency $\gamma\beta$ of $\mathbb{S}$, we have the *pair of paralogous edges* $\mathcal{P}(\gamma\beta) = \{\gamma_{\mathtt{a}}\beta_{\mathtt{a}}, \gamma_{\mathtt{b}}\beta_{\mathtt{b}}\}$ and the *complementary pair of paralogous edges* $\widetilde{\mathcal{P}}(\gamma\beta) = \{\gamma_{\mathtt{a}}\beta_{\mathtt{b}}, \gamma_{\mathtt{b}}\beta_{\mathtt{a}}\}$. Note that $\widetilde{\widetilde{\mathcal{P}}}(\gamma\beta) = \mathcal{P}(\gamma\beta)$. The *square* of $\gamma\beta$ is then $\mathcal{Q}(\gamma\beta) = \mathcal{P}(\gamma\beta) \cup \widetilde{\mathcal{P}}(\gamma\beta)$. The $\mathbb{S}$-edges in the ambiguous breakpoint graph are therefore the squares of all adjacencies in $\mathbb{S}$. The number of squares obviously equals $|\mathcal{F}(\mathbb{S})|$. Again, we can use the notation $\hat{e}$ to refer to the paralogous counterpart of an $\mathbb{S}$-edge e. For example, if $e = \gamma_{\mathtt{a}}\beta_{\mathtt{a}}$, then $\hat{e} = \gamma_{\mathtt{b}}\beta_{\mathtt{b}}$. An example of an ambiguous breakpoint graph is shown in Figure 3 (a).

Resolving a square $\mathcal{Q}(\cdot) = \mathcal{P}(\cdot) \cup \widetilde{\mathcal{P}}(\cdot)$ corresponds to *choosing* in the ambiguous breakpoint graph either the edges from $\mathcal{P}(\cdot)$ or the edges from $\widetilde{\mathcal{P}}(\cdot)$, while the complementary pair is *masked*. Resolving all squares is called *disambiguating* the ambiguous breakpoint graph. If we number the squares of $ABG(\mathbb{S},\check{\mathbb{D}})$ from 1 to $n_* = |\mathcal{F}(\mathbb{S})|$, a *disambiguation* can be

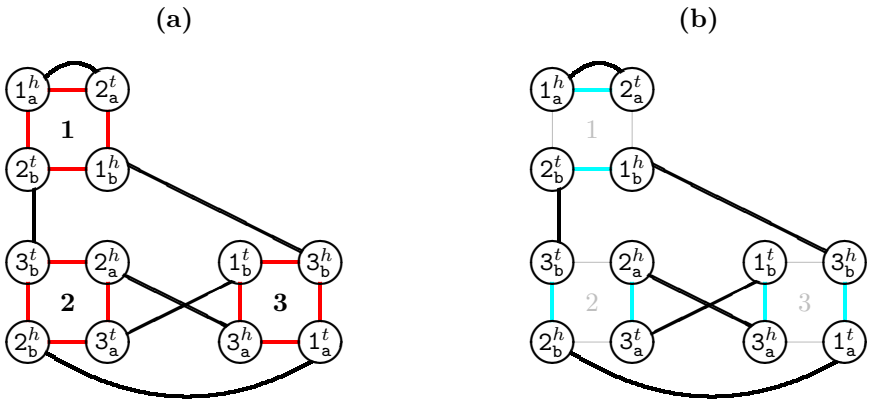


Figure 3 (a) Ambiguous breakpoint graph $ABG(\mathbb{S}, \check{\mathbb{D}})$ for genomes $\mathbb{S} = \{(1\,2\,3)\}$ and $\check{\mathbb{D}} = \{(1_a\,2_a\,\overline{3}_a\,1_b\,\overline{3}_b\,2_b)\}$. The colors distinguish the edge types: $\check{\mathbb{D}}$-edges are drawn in black and $\mathbb{S}$-edges (squares) are drawn in red. (b) Induced breakpoint graph $BG(\tau, \check{\mathbb{D}})$ in which all squares are resolved by the disambiguation $\tau = (\{1^h_a 2^t_a, 1^h_b 2^t_b\}, \{2^h_a 3^t_a, 2^h_b 3^t_b\}, \{3^h_a 1^t_b, 3^h_b 1^t_a\})$, resulting in three cycles (a 2-, a 4- and a 6-cycle). This is also the breakpoint graph of $\check{\mathbb{D}}$ and $\mathbb{B} = \{(1_a\,2_a\,3_a\,1_b\,2_b\,3_b)\} \in \mathfrak{S}^a_b(2\cdot\mathbb{S})$.

represented by a tuple $\tau = (\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_{n_*})$, where each $\mathcal{L}_i$ contains the pair of paralogous edges (either $\mathcal{P}_i$ or $\widetilde{\mathcal{P}}_i$) that are chosen (kept) in the graph for square $\mathcal{Q}_i$. The graph induced by τ is a simple breakpoint graph, which we denote by $BG(\tau, \check{\mathbb{D}})$. Figure 3 (b) shows an example.

Computing the DCJ double distance of $\mathbb{S}$ and $\mathbb{D}$ is equivalent to finding a disambiguation τ so that the number of cycles in $BG(\tau, \check{\mathbb{D}})$ is maximized [11]. As already mentioned, this problem is NP-hard.

2.2.3 The class of σ_k double distances

We can now define the class σ_k-2Dist of σ_k double distances of a singular circular genome $\mathbb{S}$ and duplicated circular genome $\mathbb{D}$ for $k = 2, 4, 6, \ldots$ as follows:

$$\mathrm{d}^2_{\sigma_k}(\mathbb{S}, \mathbb{D}) = \mathrm{d}^2_{\sigma_k}(\mathbb{S}, \check{\mathbb{D}}) = \min_{\mathbb{B} \in \mathfrak{S}^a_b(2\cdot\mathbb{S})} \{\mathrm{d}_{\sigma_k}(\mathbb{B}, \check{\mathbb{D}})\}, \text{ where } \check{\mathbb{D}} \text{ is any genome in } \mathfrak{S}^a_b(\mathbb{D}).$$

The complexity of (breakpoint) σ_2-2Dist being linear and the complexity of (DCJ) σ_∞-2Dist being NP-hard, the goal of our research is to increasingly determine, for $k = 4, 6, \ldots$, the complexity of each σ_k-2Dist. In this process we search for unveiling the boundary in which the complexity changes from polynomial to NP-hard: if for some $k \geq 4$ the complexity is found to be NP-hard, it is very likely that the complexity is also NP-hard for any $k' > k$.

So far we accomplished this task for the σ_4 and the σ_6 double distances, showing that both problems can be solved in linear time, as we will explain in the next section.

3 Solving the σ_k double distance problem

Similarly to the DCJ double distance, each σ_k-2Dist of $\mathbb{S}$ and $\mathbb{D}$ can be computed by finding a disambiguation τ of $ABG(\mathbb{S}, \check{\mathbb{D}})$ so that the number of cycles of length at most k in the resulting breakpoint graph $BG(\tau, \check{\mathbb{D}})$ is maximized. We call the latter problem σ_k-Max.

In order to solve σ_k-Max, one idea is to visit $ABG(\mathbb{S}, \check{\mathbb{D}})$ and search for candidate cycles. For describing how the graph can be screened, we need to introduce the following concepts. Two $\mathbb{S}$-edges in $ABG(\mathbb{S}, \check{\mathbb{D}})$ are *incompatible* when they belong to the same square and are not

paralogous. A cycle in $ABG(\mathbb{S}, \check{\mathbb{D}})$ is *valid* when it does not contain any pair of incompatible edges. Note that a valid cycle necessarily alternates $\mathbb{S}$-edges and $\check{\mathbb{D}}$-edges. Two valid cycles $C \neq C'$ in $ABG(\mathbb{S}, \check{\mathbb{D}})$ are either *intersecting*, when they share at least one edge, or *disjoint*.

It is obvious that any disambiguation τ of $ABG(\mathbb{S}, \check{\mathbb{D}})$ is composed of disjoint valid cycles. Let the *k-score* of τ, denoted by $\sigma_k(\tau)$, be the number of cycles of length at most k in $BG(\tau, \check{\mathbb{D}})$. The *k-score* of $ABG(\mathbb{S}, \check{\mathbb{D}})$ is the score of an optimal disambiguation for σ_k-MAX. The *switching* operation of the i-th element of a disambiguation $\tau = (\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_i \ldots, \mathcal{L}_{n_*})$ is denoted by $\widetilde{s}(\tau, i)$ and replaces value $\mathcal{L}_i$ by $\widetilde{\mathcal{L}_i}$ resulting in $\tau' = (\mathcal{L}_1, \mathcal{L}_2, \ldots, \widetilde{\mathcal{L}_i} \ldots, \mathcal{L}_{n_*})$. A choice of paralogous edges resolving a given square $\mathcal{Q}_i$ can be *fixed* for any disambiguation, meaning that the pair assigned to $\mathcal{Q}_i$ can no longer be switched. In this case, $\mathcal{Q}_i$ is itself said to be *fixed*.

3.1 Common adjacencies are preserved in any σ_k double distance

Let τ be an optimal disambiguation for σ_k-MAX of $ABG(\mathbb{S}, \check{\mathbb{D}})$. If a cycle $C \in BG(\tau, \check{\mathbb{D}})$ is disjoint from any cycle distinct from C in any other optimal disambiguation, then C must be part of all optimal disambiguations and is itself said to be *optimal*.

▶ **Lemma 1.** *For any σ_k-MAX, all existing 2-cycles in $ABG(\mathbb{S}, \mathbb{D})$ are optimal.*

Proof. The proof is sketched in Figure 4. It is clear that any 2-cycle C in $ABG(\mathbb{S}, \mathbb{D})$ is valid. Suppose that an optimal disambiguation τ induces a cycle $D \neq C$, such that C and D intersect. Note that τ cannot induce C. Since two 2-cycles cannot intersect with each other, it is clear that $|D| > 2$. Let $\mathcal{Q}_i$ be the square containing the $\mathbb{S}$-edge that is present in C and let $\tau' = \widetilde{s}(\tau, i)$. The disambiguation τ' induces the same cycles as τ, except that D is split into and replaced by C and D'. Note that $|C| = 2 \leq k$ and $|D'| = |D| - 2$, therefore we have $\sigma_k(\tau') > \sigma_k(\tau)$, contradicting the assumption that τ is optimal. ◀

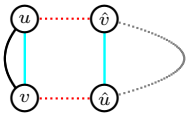


Figure 4 Illustration of the optimality of every 2-cycle. The gray path connecting vertices $\hat{v}$ and $\hat{u}$ is necessarily odd with length at least one and alternates $\check{\mathbb{D}}$- and $\mathbb{S}$-edges. The 2-cycle $C = (uv)$ intersects the longer cycle $D = (u\hat{v} \ldots \hat{u}v)$. Any disambiguation containing (red edges) $\widetilde{\mathcal{P}} = \{u\hat{v}, \hat{u}v\}$ induces D and can be improved by switching $\widetilde{\mathcal{P}}$ to (blue edges) $\mathcal{P} = \{uv, \hat{u}\hat{v}\}$, inducing, instead of D, the 2-cycle C and the cycle $D' = (\hat{v} \ldots \hat{u})$ (which is shorter than D).

This lemma is a generalization of the (breakpoint) σ_2-MAX and guarantees that all common adjacencies are preserved in any σ_k-2DIST, including the NP-hard (DCJ) σ_∞-2DIST. A square that has at least one $\mathbb{S}$-edge in a 2-cycle is called a $\{2\}$-*square*. From now on we assume that these $\{2\}$-squares are fixed so that all existing 2-cycles are induced.

3.2 A linear time greedy algorithm for the σ_4 double distance

Differently from 2-cycles, two valid 4-cycles can intersect with each other. However, two intersecting 4-cycles are always part of two co-optimal disambiguations of σ_4-MAX.

▶ **Lemma 2.** *Any valid 4-cycle that is disjoint from a 2-cycle in $ABG(\mathbb{S}, \mathbb{D})$ is induced by an optimal disambiguation of σ_4-MAX.*

Proof. All possible patterns are represented in Figure 5: (a) two co-optimal valid 4-cycles within a single square; (b) – (d) a valid 4-cycle C (in the center) connecting two squares and the three distinct possibilities of linking the four open ends. In all cases the valid 4-cycle C is either optimal or co-optimal. ◀

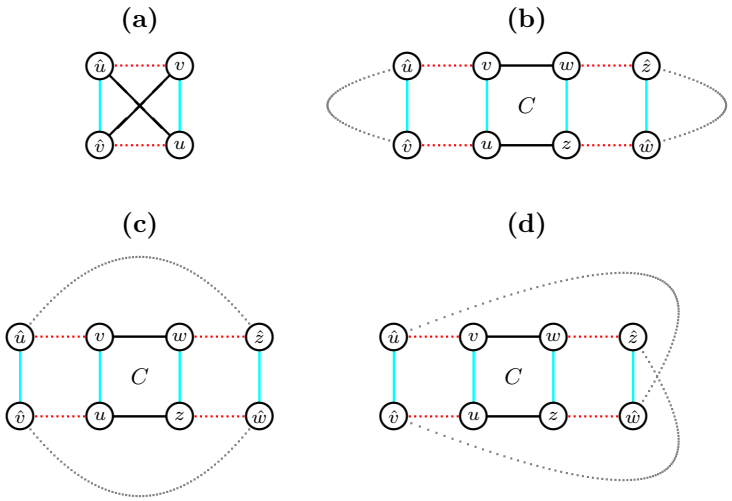


Figure 5 Illustration of the σ_4-MAX co-optimality of every valid 4-cycle not intersecting a 2-cycle. In this picture, each gray path is necessarily odd with length at least one and alternates $\check{\mathbb{D}}$- and $\mathbb{S}$-edges. In (a) any optimal solution includes either the blue edges inducing 4-cycle $(uv\hat{v}\hat{u})$ or the red edges inducing 4-cycle $(u\hat{v}v\hat{u})$. Parts (b) – (c) show the 4-cycle $C = (uvwz)$ in the center, induced by blue edges. In (b) it is easy to see that any optimal solution is induced by the blue edges and includes, besides the cycle C, cycles $(\hat{u} \dots \hat{v})$ and $(\hat{w} \dots \hat{z})$. In (c) an optimal solution includes 4-cycle C and cycle $C' = (\hat{u}\hat{v} \dots \hat{w}\hat{z} \dots)$. If the connection between $\hat{v}$ and $\hat{w}$ is a single edge, then another optimal solution is induced by the red edges, including 4-cycle $D = (u\hat{v}\hat{w}z)$ and cycle $D' = (v\hat{u} \dots \hat{z}w)$. And if additionally the connection between $\hat{u}$ and $\hat{z}$ is a single edge, then both C' and D' are also 4-cycles. In (d) any optimal solution is induced by the blue edges and includes 4-cycle C and cycle $(\hat{u}\hat{v} \dots \hat{z}\hat{w} \dots)$, which is also a 4-cycle when the connections between $\hat{v}$ and $\hat{z}$ and between $\hat{u}$ and $\hat{w}$ are single edges.

A consequence of this lemma is that an optimal disambiguation of σ_4-MAX can be obtained greedily: After fixing the squares containing edges that are part of 2-cycles, traverse the remainder of the graph and, for each valid 4-cycle C that is found, fix the square(s) containing $\mathbb{S}$-edges that are part of C. When this part is accomplished the remaining squares can be fixed arbitrarily.

3.3 A linear time algorithm for the σ_6 double distance

In this section we may refer to a valid 4- or 6-cycle as a $\{4..6\}$-cycle. It is easy to see that $\{4..6\}$-cycles can intersect with each other. Moreover, for the σ_6-MAX, not every $\{4..6\}$-cycle is induced by at least one optimal disambiguation. For that reason, a greedy algorithm does not work here. Still, we can solve σ_6-MAX in linear time, as we will describe in the following.

Pruning the ambiguous breakpoint graph

We proceed with a preprocessing in which from $ABG(\mathbb{S}, \check{\mathbb{D}})$ first all edges are removed that are incompatible with the existing 2-cycles, and then all remaining edges that cannot be part of a $\{4..6\}$-cycle. This results in a $\{6\}$-*pruned* ambiguous breakpoint graph $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$.

The first step is easily achieved by a simple graph traversal in which for each $\check{\mathbb{D}}$-edge uv it is tested whether both ends connect to the same $\mathbb{S}$-edge uv. If this is the case, the two incident $\mathbb{S}$-edges $u\hat{v}$ and $v\hat{u}$ are removed from the graph, separating the 2-cycle (uv). Then, in the second step, for any remaining edge e, its 6-neighborhood (which has constant size in a graph of degree at most three) is exhaustively explored for the existence of a $\{4..6\}$-cycle involving e. If no such cycle is found, e is deleted. Each of these two steps clearly takes linear time $O(|ABG(\mathbb{S}, \check{\mathbb{D}})|)$, and what remains is exactly the desired graph $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$.

Note that, for any square $\mathcal{Q}_i$ with $1 \leq i \leq n_*$, graph $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$ might contain either (a) all edges, or (b) only three edges, or (c) only two edges each one being from a distinct pair of paralogous edges, or (d) only two edges being from the same pair of paralogous edges, or (e) a single edge, or (f) no edge (see Figure 6). In cases (a), (b) and (c), $\mathcal{Q}_i$ is still ambiguous, while for cases (d), (e) and (f) $\mathcal{Q}_i$ is already resolved. We assume that these resolved squares are fixed according to the remaining paralogous edges in cases (d) and (e) or arbitrarily in case (f).

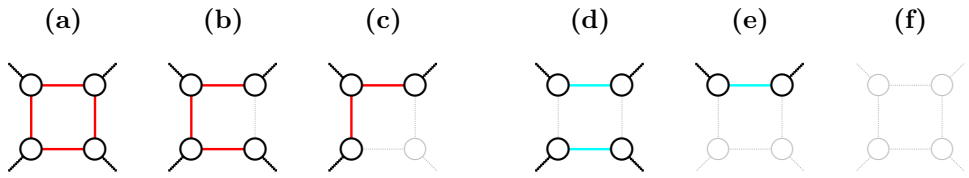


Figure 6 Possible (partial) squares of prunned $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$. Shadowed parts represent the removed elements: $\mathbb{S}$-edges that are incompatible with 2-cycles and/or edges and vertices that cannot be part of a $\{4..6\}$-cycle. Cases (a), (b) and (c) are ambiguous, cases (d) and (e) are resolved and case (f) is arbitrarily resolved.

Let the 6-*score* of $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$ be the score of a disambiguation that maximizes $c_2 + c_4 + c_6$ in $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$. Obviously, a disambiguation giving the 6-score of $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$ is an optimal disambiguation for σ_6-MAX. Therefore we can search for optimally resolving the remaining ambiguous squares by analyzing the smaller pruned graph $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$. Furthermore, the problem can be solved independently for each of the connected components of $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$, so that the result of σ_6-MAX is the sum $\sum_{G \in ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})} \sigma_6(G)$, where $\sigma_6(G)$ is the 6-score (maximum number of disjoint 2- and $\{4..6\}$-cycles) of component G.

Describing the connected components of the pruned graph

We will now describe the properties of the pruned graph $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$. An $\mathbb{S}$-edge (respectively $\check{\mathbb{D}}$-edge) that is present in $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$ is called an $\mathbb{S}_{\{6\}}$-edge (respectively $\check{\mathbb{D}}_{\{6\}}$-edge). Any square that is still ambiguous in $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$ is called a $\{6\}$-*ambiguous square*. A $\{6\}$-ambiguous square $\mathcal{Q}_i$ is a $\{6\}$-*neighbor* of another $\{6\}$-ambiguous square $\mathcal{Q}_j$ when a vertex of $\mathcal{Q}_i$ is connected to a vertex of $\mathcal{Q}_j$ by a $\check{\mathbb{D}}_{\{6\}}$-edge.

▶ **Proposition 3.** *Each connected component G of $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$ is of one of the two types:*
1. Ambiguous*: G includes at least one $\{6\}$-ambiguous square and no 2-cycle;*
2. Resolved (trivial)*: G is a simple valid 2-, 4- or 6-cycle;*

Proof. By construction all 2-cycles are disconnected from the other components of the graph $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$. Therefore, if a component G of $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$ has a $\{6\}$-ambiguous square, G cannot include any 2-cycle. Now let G be a connected component that does not include a $\{6\}$-ambiguous square. Then any vertex in G has exactly one incident $\mathbb{S}$-edge and one incident $\check{\mathbb{D}}$-edge. Therefore G must be a simple valid 2-, 4- or 6-cycle. ◀

Let $\mathcal{R}$ be the set of resolved and $\mathcal{B}$ be the set of ambiguous components of $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$. The result of σ_6-MAX can be then computed with the formula $|\mathcal{R}| + \sum_{G \in \mathcal{B}} \sigma_6(G)$.

Computing the 6-score of an ambiguous component of the pruned graph

For solving σ_6-MAX, we will now describe the only missing part: a procedure that computes the 6-score of an ambiguous component $G \in \mathcal{B}$.

▶ **Proposition 4.** *Any $\check{\mathbb{D}}_{\{6\}}$-edge is part of either one or two (intersecting) $\{4..6\}$-cycles.*

Proof. By construction any $\check{\mathbb{D}}_{\{6\}}$-edge is part of at least one $\{4..6\}$-cycle. The remainder of the proof can be found in Figure 7 and in supplementary Figures 8-10 (in the appendix), which display all possible patterns showing a $\check{\mathbb{D}}$-edge in two distinct intersecting $\{4..6\}$-cycles which themselves do not intersect 2-cycles. In all cases an exhaustive search shows that the same $\check{\mathbb{D}}$-edge cannot be part of a third $\{4..6\}$-cycle. ◀

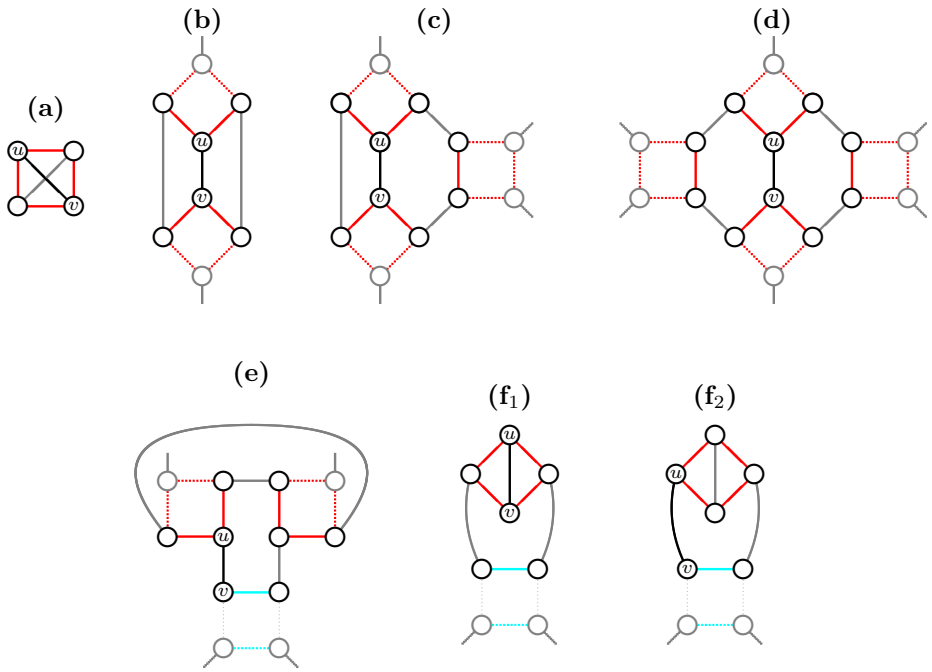


Figure 7 Patterns showing a $\check{\mathbb{D}}$-edge uv in two distinct intersecting $\{4..6\}$-cycles which themselves do not intersect 2-cycles. (a) The edge uv is part of two 4-cycles within the same square. (b) – (d) The edge uv connects two distinct squares and is part of two $\{4..6\}$-cycles whose intersection is only uv. (e) The edge uv is part of two 6-cycles whose intersection is a 3-path starting in uv. (f_1) The edge uv connects vertices of the same square and is part of two 6-cycles. (f_2) The edge uv is one of the other two $\check{\mathbb{D}}$-edges in the 6-cycles of (f_1). In all cases an exhaustive search shows that uv cannot be part of a third $\{4..6\}$-cycle (see supplementary Figures 8-10 in the appendix). Furthermore, in each one of the cases (e) – (f_2) one square (marked in blue) is clearly fixed: if this square could be switched, this would merge each of the two existing 6-cycles into a longer cycle.

▶ **Proposition 5.** *Any $\mathbb{S}_{\{6\}}$-edge of a $\{6\}$-ambiguous square $\mathcal{Q}_i$ is part of exactly one $\{4..6\}$-cycle.*

Proof. If an $\mathbb{S}_{\{6\}}$-edge e is in a $\{6\}$-ambiguous square $\mathcal{Q}_i$, it "shares" the same $\check{\mathbb{D}}_{\{6\}}$-edge d with another $\mathbb{S}_{\{6\}}$-edge e' from the same square $\mathcal{Q}_i$. In this case the $\check{\mathbb{D}}_{\{6\}}$-edge d is part of exactly two $\{4..6\}$-cycles and each of the $\mathbb{S}_{\{6\}}$-edges e and e' can be part of only one $\{4..6\}$-cycle. ◀

The proposition above immediately implies the following:

▶ **Corollary 6.** *Choosing an $\mathbb{S}_{\{6\}}$-edge e of a $\{6\}$-ambiguous square $\mathcal{Q}_i$ (and its paralogous edge $\hat{e}$) implies a unique disambiguation of all $\{6\}$-neighbors of $\mathcal{Q}_i$.*

Consider an ambiguous component G of $ABG_{\{6\}}(\mathbb{S}, \check{\mathbb{D}})$ and denote by τ_G a disambiguation including only the $\{6\}$-ambiguous squares of G. Now let τ_G be obtained with Algorithm 1, using a recursion of the statement of Corollary 6, as described in Algorithm 2. Since the recursion first tests whether each neighbor was already resolved or fixed (lines 1 and 8), it visits and resolves each $\{6\}$-ambiguous square of G exactly once. The resolving procedure itself visits the 6-neighborhood of up to two $\mathbb{S}_{\{6\}}$-edges and can be done in constant time. Therefore the whole recursive procedure takes linear time $O(m)$, where m is the number of $\{6\}$-ambiguous squares in G.

Algorithm 1 StraightComponentDisambiguation.

Input: A component G whose $\{6\}$-ambiguous squares are numbered $\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_m$
Output: A disambiguation τ_G of G

1: $e \leftarrow$ any $\mathbb{S}_{\{6\}}$-edge in $\mathcal{Q}_1$;
2: $\tau_G[1] \leftarrow \{e, \hat{e}\}$;
3: **for** $i \leftarrow 2, \ldots, m$ **do** $\tau_G[i] \leftarrow \emptyset$;
4: ResolveNeighbors(τ_G, e); /* recursive procedure */
5: **if** $\hat{e}$ is an $\mathbb{S}_{\{6\}}$-edge **then** /* the paralogous $\mathbb{S}$-edge $\hat{e}$ is also in G */
6: ResolveNeighbors$(\tau_G, \hat{e})$; /* recursive procedure */
7: **return** τ_G

Algorithm 2 ResolveNeighbors.

Input: A partially filled disambiguation τ_G and an $\mathbb{S}$-edge uv of component G
/* $\mathbb{S}$-edge uv is adjacent to two $\check{\mathbb{D}}_{\{6\}}$-edges uz and vw */

1: **if** vertex z is not in a resolved or fixed square **then**
2: $i \leftarrow$ index in τ_G of square containing z;
3: $e \leftarrow \mathbb{S}$-edge zx of $\mathcal{Q}_i$ forming a $\{4..6\}$-cycle with uv and uz;
4: $\tau_G[i] \leftarrow \{e, \hat{e}\}$;
5: ResolveNeighbors(τ_G, e);
6: **if** $\hat{e}$ is an $\mathbb{S}_{\{6\}}$-edge **then**
7: ResolveNeighbors$(\tau_G, \hat{e})$;
8: **if** vertex w is not in a resolved or fixed square **then**
9: $j \leftarrow$ index in τ_G of square containing w;
10: $f \leftarrow \mathbb{S}$-edge wy of $\mathcal{Q}_j$ forming a $\{4..6\}$-cycle with uv and vw;
11: $\tau_G[j] \leftarrow \{f, \hat{f}\}$;
12: ResolveNeighbors(τ_G, f);
13: **if** $\hat{f}$ is an $\mathbb{S}_{\{6\}}$-edge **then**
14: ResolveNeighbors$(\tau_G, \hat{f})$;
15: **return**

Let $\widetilde{s}(\tau_G)$ be the disambiguation obtained by switching all squares in τ_G. Examples of the two disambiguations τ_G and $\widetilde{s}(\tau_G)$ are given in supplementary Figures 11, 12 and 13 in the appendix. Now denote by τ_G^* a disambiguation with highest 6-score among τ_G and $\widetilde{s}(\tau_G)$.

▶ **Corollary 7.** *The disambiguation τ_G^* is optimal.*

4 Discussion and open problems

Several combinatorial problems related to genome evolution and ancestral reconstruction, including median, guided halving and double distance, have the distance problem as a basic unit. Interestingly, for circular genomes these three problems can be solved in polynomial time when they are built upon the breakpoint distance, while they are NP-hard when they are built upon the (rearrangement) DCJ distance.

Our study started as an exploration of the complexity space of the double distance between these two extremes. Therefore we considered a new class of genomic distance measures called σ_k distances, for $k = 2, 4, 6, \ldots, \infty$, which are between the breakpoint (σ_2) and the DCJ (σ_∞) distance. In this work we presented the results of investigating the complexity of the double distance first for the σ_4, then for the σ_6 distance, assuming that the given genomes have only circular chromosomes. For both cases we devised linear time algorithms, built on a variation of the breakpoint graph called ambiguous breakpoint graph.

The breakpoint graph of genomes with linear chromosomes includes, besides the cycles of even length, paths of odd and even length. It is known that the breakpoint double distance of genomes with linear chromosomes can also be solved in linear time [11]. Furthermore, we know that each linear chromosome in the singular genome $\mathbb{S}$ "removes" from the ambiguous breakpoint graph one ambiguous square (corresponding to the four vertices that will then be at the end of paths), reducing the number of choices to be made. We have not yet worked out the details, but we see no reason why our linear time algorithms for the σ_4 and the σ_6 double distances cannot be extended to take genomes with linear chromosomes into account.

More far-reaching, we conjecture that, if for some $k \geq 8$ the complexity of the σ_k double distance is found to be NP-hard, the complexity is also NP-hard for any $k' > k$: as k grows each edge of the ambiguous breakpoint graph may be part of a larger number of valid cycles of length at most k, making the description of the combinatorial space more complex. We expect that by finding the smallest k for which the σ_k double distance is NP-hard we will be able to confirm our conjecture. In any case, the natural next step in our research is to study the σ_8 double distance.

A more challenging avenue of research is doing the same exploration for both median and guided halving problems under the class of σ_k distances. In both cases it seems possible to adopt variations of the breakpoint graph. To the best of our knowledge, the guided halving problem has not yet been studied for any σ_k distance, while for the median much effort for the σ_4 distance has been done but no progress was obtained so far. A reason for this difference of progress between double distance and median is probably related to the underlying approaches. While the double distance can be solved by *removing* paralogous edges from the ambiguous breakpoint graph, solving the median requires *adding* new edges (representing the adjacencies of the median genome) to an extended breakpoint graph, and the combinatorial space of the distinct possibilities of doing that could not yet be described.

References

1 Max Alekseyev and Pavel A. Pevzner. Colored de Bruijn graphs and the genome halving problem. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(1):98–107, 2008. doi:10.1109/TCBB.2007.1002.

2 Vineet Bafna and Pavel A. Pevzner. Genome rearrangements and sorting by reversals. In *Proceedings of FOCS 1993*, pages 148–157, 1993. doi:10.1109/SFCS.1993.366872.

3 Anne Bergeron, Julia Mixtacki, and Jens Stoye. A unifying view of genome rearrangements. In *Proceedings of WABI 2006*, volume 4175 of *LNBI*, pages 163–173, 2006. doi:10.1007/11851561_16.

4 Cedric Chauve. Personal communication in Dagstuhl Seminar no. 18451 - Genomics, Pattern Avoidance, and Statistical Mechanics, November 2018.

5 Nadia El-Mabrouk and David Sankoff. The reconstruction of doubled genomes. *SIAM Journal on Computing*, 32(3):754–792, 2003. doi:10.1137/S0097539700377177.

6 Sridhar Hannenhalli and Pavel A. Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). In *Proceedings of FOCS 1995*, pages 581–592. IEEE Press, 1995. doi:10.1109/SFCS.1995.492588.

7 Sridhar Hannenhalli and Pavel A. Pevzner. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM*, 46(1):1–27, 1999. doi:10.1145/300515.300516.

8 João Meidanis, Maria Emília M. T. Walter, and Zanoni Dias. Reversal distance of signed circular chromosomes. Technical Report IC–00–23, Institute of Computing, University of Campinas, Brazil, 2000. URL: www.ic.unicamp.br/~reltech/2000/00-23.pdf.

9 Julia Mixtacki. Genome halving under DCJ revisited. In *Proceedings of COCOON 2008*, volume 5092 of *LNCS*, pages 276–286. Springer Verlag, 2008. doi:10.1007/978-3-540-69733-6_28.

10 David Sankoff. Edit distance for genome comparison based on non-local operations. In *Proceedings of CPM 1992*, volume 644 of *LNCS*, pages 121–135, 1992. doi:10.1007/3-540-56024-6_10.

11 Eric Tannier, Chunfang Zheng, and David Sankoff. Multichromosomal median and halving problems under different genomic distances. *BMC Bioinformatics*, 10:120, 2009. doi:10.1186/1471-2105-10-120.

12 Sophia Yancopoulos, Oliver Attie, and Richard Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16):3340–3346, 2005. doi:10.1093/bioinformatics/bti535.

A Supplementary figures

In Figures 8, 9 and 10 we show the exhaustive exploration of the most complex patterns (b) – (d) of Figure 7, showing that uv (the black edge connecting black vertices) can only be part of two intersecting $\{4..6\}$-cycles. Some patterns have very similar structure. If this is the case for patterns P and P' and when the distance (length of shortest path) from uv to the open ends in P' is bigger than in P, we say that P' is *worse* than P. In other words, if uv cannot be in a third $\{4..6\}$-cycle in P, the same is true for P'.

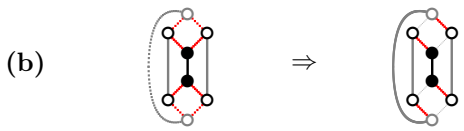


Figure 8 Exhaustive exploration of the pattern from Figure 7 (b). The only possibility to be explored is by connecting the top to the bottom vertex, and the smallest third cycle including edge uv in this scenario is an 8-cycle. (A symmetric case was omitted.)

In Figures 11, 12 and 13 we give examples of running Algorithm 1 and computing a straight disambiguation for three distinct ambiguous components.

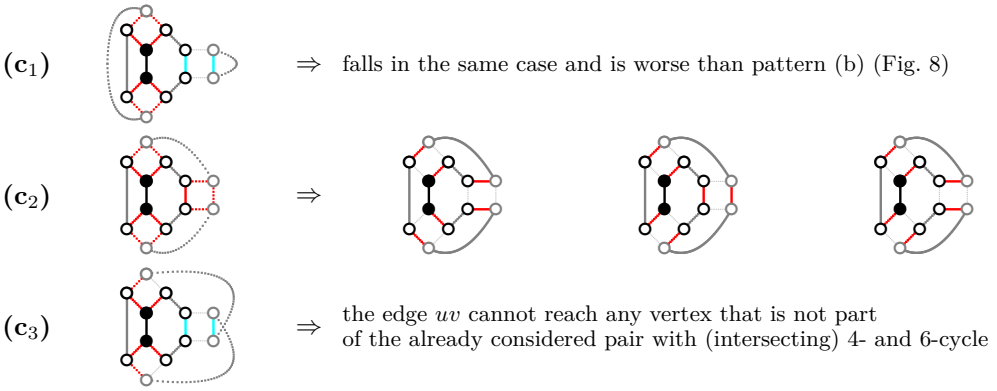


Figure 9 Exhaustive exploration of the pattern from Figure 7 (c) with the three possibilities of connecting the four open ends. Only (c_2) needs to be further explored and no alternative gives a third {4..6}-cycle. (Symmetric cases were omitted.)

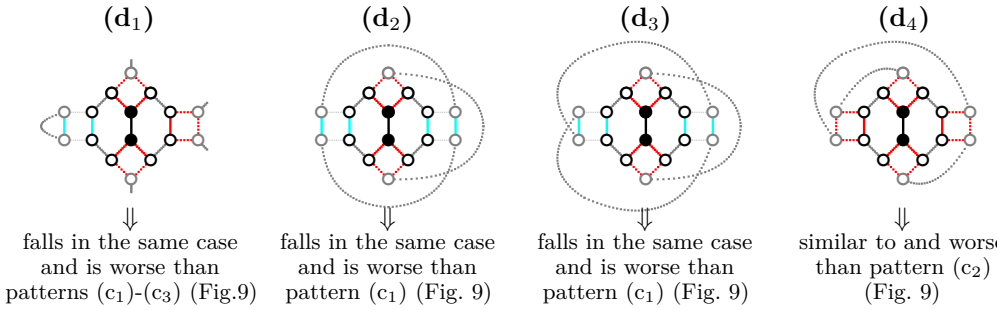


Figure 10 Exhaustive exploration of the pattern from Figure 7 (d). Part (d_1) falls in the same case and is worse than pattern (c). Parts (d_2) – (d_4) display the other possibilities of connecting the open ends and are worse than smaller scenarios already explored. (Symmetric cases were omitted.)

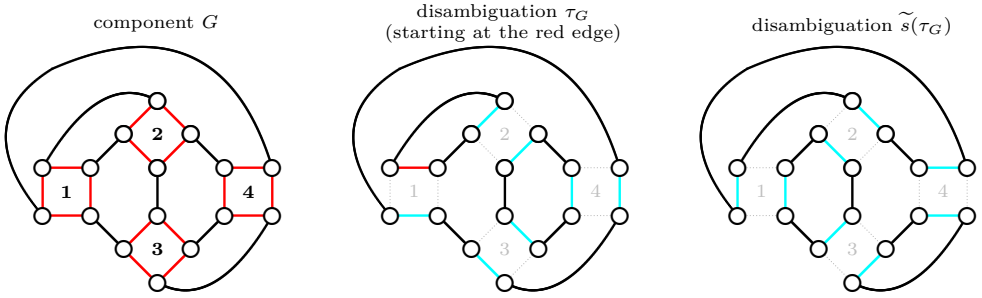


Figure 11 Example of the two best candidate disambiguations of a component of a {6}-pruned ambiguous breakpoint graph. Both candidates are optimal, inducing one 4- and two 6-cycles.

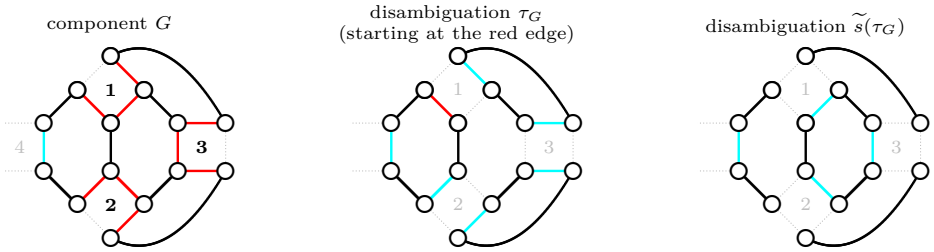


Figure 12 Example of the two best candidate disambiguations of a component G of a $\{6\}$-pruned ambiguous breakpoint graph. While τ_G is optimal and induces two 4-cycles and one 6-cycle, the alternative $\widetilde{s}(\tau_G)$ induces only a single 6-cycle.

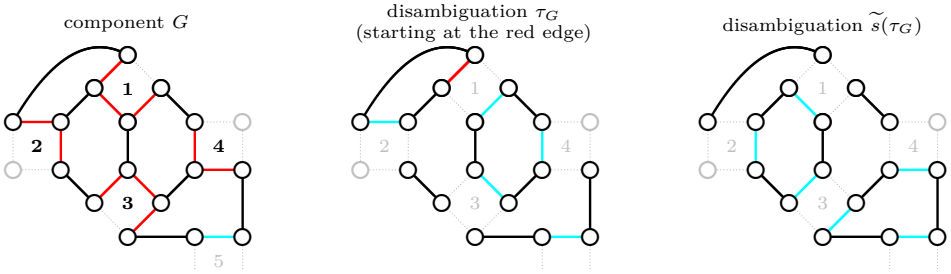


Figure 13 Example of the two best candidate disambiguations of a component G of a $\{6\}$-pruned ambiguous breakpoint graph. Both candidates are optimal: τ_G induces one 4- and one 6-cycle, while the alternative $\widetilde{s}(\tau_G)$ induces two 6-cycles.

Efficient Reconciliation of Genomic Datasets of High Similarity

Yoshihiro Shibuya
LIGM, Université Gustave Eiffel, Marne-la-Vallée, France

Djamal Belazzougui
CAPA, DTISI, Centre de Recherche sur l'Information Scientifique et Technique, Algiers, Algeria

Gregory Kucherov
LIGM, CNRS, Université Gustave Eiffel, Marne-la-Vallée, France

Abstract

We apply Invertible Bloom Lookup Tables (IBLTs) to the comparison of k-mer sets originated from large DNA sequence datasets. We show that for similar datasets, IBLTs provide a more space-efficient and, at the same time, more accurate method for estimating Jaccard similarity of underlying k-mer sets, compared to MinHash which is a go-to sketching technique for efficient pairwise similarity estimation. This is achieved by combining IBLTs with k-mer sampling based on syncmers, which constitute a context-independent alternative to minimizers and provide an unbiased estimator of Jaccard similarity. A key property of our method is that involved data structures require space proportional to the difference of k-mer sets and are independent of the size of sets themselves. As another application, we show how our ideas can be applied in order to efficiently compute (an approximation of) k-mers that differ between two datasets, still using space only proportional to their number. We experimentally illustrate our results on both simulated and real data (*SARS-CoV-2* and *Streptococcus Pneumoniae* genomes).

2012 ACM Subject Classification Applied computing

Keywords and phrases k-mers, sketching, Invertible Bloom Lookup Tables, IBLT, MinHash, syncmers, minimizers

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.14

Related Version *Full Version*: `https://doi.org/10.1101/2022.06.07.495186`

Supplementary Material *Software (Source Code)*: `https://github.com/yhhshb/km-peeler.git`

1 Introduction

Alignment-free methods became a prevalent paradigm in computational analysis of modern genomic datasets. However, despite being faster than their alignment-based counterparts, algorithms based on k-mer sets are starting to struggle when applied to the large datasets produced nowadays [20, 13, 16]. To deal with this issue, a considerable effort has been put to developing optimized data structures, with succinct solutions [25, 23, 16] and approximate membership data structures [29, 13, 2, 14, 3] being two examples.

In recent years, sketching techniques have been gaining increasing attention thanks to their capacity of drastically decreasing space usage. MinHash is probably the most well-known representative of this family of algorithms. Application of MinHash to comparison of DNA sequence datasets was pioneered in Mash software [24] and subsequently used in several other tools. With this approach, input datasets are transformed into smaller "sketches" on which subsequent comparisons are performed. In short, sequences are first fragmented into their constituent k-mers which are then hashed, with each sketch storing only s minimum values, with s defined by the user. The fraction of shared hashes between two sketches is an unbiased estimator of the Jaccard similarity index [4]. A MinHash sketch can thus be viewed

as a sample of the set of k-mers of the sequence it represents. Given that s is much smaller than the genome length, working with the sampled hashes leads to fast pairwise comparisons using small memory. However, when two sequences are close and share most of their k-mers, MinHash sketches of small size are not able to reliably estimate their degree of similarity since differences are likely to be missed during sampling.

In this work, we propose an alternative approach to evaluate the difference in k-mer composition of two related datasets. Our method relies on Invertible Bloom Lookup Table (IBLT) data structure [12, 10] which is an extension of Bloom filters, supporting deletions of items and, most importantly, enumeration (with high probability) of stored items. One of the applications of IBLT is reconciliation of two sets of items: in a scenario considered in [12], a set A is stored in an IBLT which is then transmitted to the holder of another set B. By screening B against the IBLT of A it is possible to recover the items $A \setminus B$ and $B \setminus A$, with high probability. This is done through the so-called *peeling* procedure [7].

In this paper we make one step further: inspired by ideas of [26], we recover both $A \setminus B$ and $B \setminus A$ from IBLTs of A and B, rather than from an IBLT of one of them and the whole other set. Furthermore, a crucial property is that the *size of these IBLTs is bounded in terms of the symmetric difference size* $(A \setminus B) \cup (B \setminus A)$ rather than the size of the original sets. This provides a key to the efficiency of our solution when input sets are similar: even if input sets are very big, their difference can be recovered using a data structure (sketch) whose size is proportional to the size of the difference of those sets rather than of the sets themselves. Estimating the symmetric difference allows us to estimate the Jaccard similarity, using information about the sizes of input sets. Thus, whereas close datasets require larger MinHash sketches to be properly compared, our method, on the contrary, requires smaller memory.

Another ingredient of our solution is k-mer sampling. Intuitively, since two adjacent k-mers share $k-1$ bases, the information stored in the set of all k-mers appears highly redundant. One popular method of sampling k-mers from genomic sequences is based on minimizers [28]. Under this technique, consecutive sampled k-mers are within a bounded distance from each other and therefore no large portion of the sequence can remain unsampled. Another favorable property is that similar regions are likely to yield similar samples of minimizers. However, it has recently been shown that estimating Jaccard similarity based on minimizer sampling leads to a bias [1]. Here we propose to replace minimizers by syncmers [8]. Syncmers provide another way of k-mer sampling which has certain advantages over minimizers. As opposed to minimizers, syncmers are not context-dependent: for a k-mer to be a syncmer depends on the k-mer alone regardless the context where it occurs, and, under standard randomness assumptions on involved hash functions, all k-mers have equal chance to be a syncmer. As a consequence, syncmer sampling leads to an unbiased estimate of Jaccard similarity, as the fraction of syncmers among shared k-mers (intersection) is expected to be the same as that among all k-mers (union). We experimentally validate that this is indeed the case.

By combining syncmer sampling with IBLTs, we obtain a space-efficient method for accurately estimating Jaccard similarity for similar datasets. For datasets of high similarity, the proposed method is superior to the popular MinHash algorithm [24], both in terms of memory and precision. We also propose an application of this technique to retrieve k-mers that differ between two given datasets. Our method computes a superset of those k-mers with a limited number of spurious k-mers. In particular, under the assumption that each k-mer occurs once, our method computes the exact set differences between involved k-mer sets. We validate our algorithms on both simulated data and on real datasets made of *SARS-CoV-2* and *Straphilococcus Pneumoniae* genomes.

2 Technical preliminaries

We consider DNA alphabet $\Sigma = \{A, C, G, T\}$ even though our algorithms can be easily generalized. Given a string $S \in \Sigma^*$, we use the notation $S[i, k]$ to indicate the substring of length k starting at position i called a k-mer. The k-mer set K_S of S is the set of k-mers $S[i, k]$ for $i \in [0, |S| - k + 1]$.

2.1 Minimizers

Independently introduced in [28] and [30], minimizers are defined by a triplet of parameters (k, w, h), where k is the k-mer length, w a window size, and h a function defining an order on k-mers. h is usually chosen to be an appropriately defined hash function, the lexicographical order is rarely used in practice due to its poor statistical properties.

Each window $S[i, w + k - 1]$ defines a minimizer which is the minimal k-mer among w k-mers occurring in $S[i, w + k - 1]$ w.r.t. the order given by h. Two neighboring minimizers are thus separated by at most w positions making it impossible to have large stretches of the original sequence not covered by any minimizers.

Since two neighboring windows at positions i and $i + 1$ are likely to share their minimizer, minimizers provide a way to sample k-mers from a sequence with bounded distance between consecutive sampled k-mers. An advantage of this sampling strategy is that similar sequences will likely have similar lists of minimizers, which makes it useful for mapping algorithms [19, 15]. Under reasonable assumptions, the density of minimizers, i.e. fraction of sampled k-mers, is $\frac{2}{w+1}$ [28, 8]. If minimizer positions in the original sequence are not important, they can be discarded and the resulting k-mer multiset can be reduced to a simple k-mer set.

2.2 Syncmers

Minimizers are susceptible to mutations of any base of their window [8]. That is, a k-mer may cease to be a minimizer if a modified base occurs not only inside this k-mer, but also in its close neighborhood. Sampling with a higher density alleviates this problem but it reduces the advantages of the methods because more minimizers are selected. Methods to generate minimizer indices with the best possible density exist [6, 9] but they are usually offline algorithms, limiting their potential applications outside alignment.

Syncmers are a family of alternative methods to minimizers that does not suffer from this issue [8]. Similarly to minimizers, syncmers are defined using a triplet of parameters (k, z, h) where $z < k$ is used to decompose each k-mer into its constituent z-mers and h defines an order over them. A k-mer q is a *syncmer* (called *closed syncmers* in [8]) iff its minimal z-mer occurs as a prefix (position $i = 0$) or as a suffix (position $i = k - z + 1$) of q. Thus, a syncmer is defined by its sequence alone, regardless the context in which it occurs. For this reason, syncmer sampling has been shown to be more resistant to mutations and then to improve the sensitivity of alignment algorithms [8].

Similar to minimizers, consecutive syncmers occur at a bounded distance. More precisely, consecutive syncmers must overlap by at least z characters and therefore "pave" the sequence without gaps. The fraction of syncmers among all k-mers is estimated to be $\frac{2}{k-z+1}$ [8].

2.3 Invertible Bloom Lookup Tables

Invertible Bloom Lookup Tables (IBLT) [10, 12] are a generalization of Bloom filters for storing a set of elements (keys), drawn from a large universe, possibly associated with attribute values. In contrast to Bloom filters, in addition to insertions, IBLTs also support

deletion of keys as well as listing. The latter operation succeeds with high probability (w.h.p.) depending on the number of stored keys relative to the size of the data structure. An important property is that this probability depends only on the number of keys stored at the moment of listing, and not across the entire lifespan of the data structure. Thus, at a given time, an IBLT can store a number of keys greatly exceeding the threshold for which it was built, returning to be fully functional whenever a sufficient number of deletions has taken place. Note also that IBLTs, in their basic version, don't support multiple insertions of the same key.

An IBLT is an array T of m buckets together with r hash functions $h_1, \ldots, h_r$ mapping a key universe U (in our case, k-mers or strings) to $[0..m-1]$ and an additional global hash function h_e on U. Each bucket $T[i], i \in [0..m-1]$, contains three fields: a counter $T[i].C$, a key field $T[i].P$ and a hash field $T[i].H$, where C counts the number of keys hashed to bucket i, P stores the XOR-sum of the keys (in binary representation) hashed to bucket i, and H contains the XOR-sum of hashes produced by h_e on keys.

Adding a key p to the IBLT is done as follows. For each $j \in \{1, \ldots, r\}$, we perform $T[h_j(p)].C = T[h_j(p)].C + 1$, $T[h_j(p)].P = T[h_j(p)].P \oplus p$, and $T[h_j(p)].H = T[h_j(p)].H \oplus h_e(p)$, where $\oplus$ stands for XOR. Given that XOR is the inverse operation of itself, deletion of p is done similarly except that $T[h_j(p)].C = T[h_j(p)].C - 1$.

Listing the keys held in an IBLT is done through the process of peeling working recursively as follows. If for some i we have $T[i].C = 1$, payload field $T[i].P$ is supposed to contain a single key p. Field H is not strictly necessary, it acts as a "checksum" to verify that p is indeed a valid key by checking if $h_e(T[h_j(p)].P) = T[h_j(p)].H$. This check is used to avoid the case when $T[i].C = 1$ whereas $T[i].P$ is not a valid key, which can result from extraneous deletions of keys not present in the data structure. In Section 3.2 we will elaborate on the role of this field in our framework. If the check holds, key p can be reported and deleted (peeled) from the IBLT. Updating hash sums and counters is done in a similar way: $T[h_j(p)].H = T[h_j(p)].H \oplus h_e(p)$ and $T[h_j(p)].C = T[h_j(p)].C - 1$. The procedure continues until all counters $T[i].C$ are equal to zero.

At each moment, an IBLT is associated to a r-hypergraph where nodes are buckets and edges correspond to stored keys with each edge including the buckets a key is hashed to. Listing the keys contained in an IBLT then relies on the peelability property of random hypergraphs [7, 22]. Assume our hash functions are fully random. Then it is known that for $r \geq 3$, a random r-hypergraph with m nodes and n edges is peelable w.h.p. iff $m \geq c_r n$ where c_r is a constant peelability threshold. The first values of c_r are $c_3 \approx 1.222, c_4 \approx 1.295, c_5 \approx 1.425, \cdots$ [12]. Thus, allocating

$$m = n(c_r + \varepsilon), \tag{1}$$

buckets, for $\varepsilon > 0$, for storing n keys guarantees successful peeling with high probability.

2.4 MinHash sketching

MinHash sketching was introduced in [4] as a method to estimate Jaccard similarity between two sets, applied to document comparison. In bioinformatics, MinHash was first applied in Mash software [24] and then successfully used in a number of other tools. Assume we are given a universe U and an order on U defined via a hash function h. For a set $A \subset U$, the bottom-s MinHash sketch of A, denoted $\mathbb{S}(A)$, is the set of s minimal elements of A (or their

hashes), where s is a user-defined parameter. The Jaccard similarity index between two sets A and B, $J(A,B) = |A \cap B|/|A \cup B|$, can then be estimated from the sketches of A and B, namely

$$|\mathbb{S}(A \cap B) \cap \mathbb{S}(A) \cap \mathbb{S}(B)|/|\mathbb{S}(A \cup B)| \tag{2}$$

is an unbiased estimator of $J(A,B)$.

The Jaccard similarity between the k-mer sets of two dataset constitutes a biologically relevant measure of their similarity. In particular, if involved datasets are genomic sequences, this measure allows one to estimate the mutation rate between the sequences [11, 24].

3 Methods

3.1 Set reconciliation from two IBLTs

Invertible Bloom Lookup Tables can be used to achieve set reconciliation between two sets A and B, that is to recover sets $A \setminus B$ and $B \setminus A$. Under a scenario described in [12], the holder of A stores it in an IBLT T_A which is then transmitted to the holder of B. Elements of B are then deleted from T_A. In the resulting IBLT, P-fields with $T_A[i].C = 1$ correspond to elements of $A \setminus B$ and those with $T_A[i].C = -1$ to $B \setminus A$. The peeling process is applied to either of such fields. Whenever $T_A[i].C = 1$, we delete $p = T_A[i].P$ from T_A on condition that $h_e(p) = T_A[i].H$. Similarly, whenever $T_A[i].C = -1$, we add (XOR) $p = T_A[i].P$ to T_A on condition that $h_e(p) = T_A[i].H$. The process lists all elements of both $A \setminus B$ and $B \setminus A$ w.h.p.

Inspired by work [26], we modify the above scheme in order to recover the symmetric difference between A and B from their respective IBLTs T_A and T_B, rather than from the IBLT of one set and the whole other set. To do this, we define T_A and T_B to be of the same size and to use the same hash functions. We then compute the difference of T_A and T_B, denoted T_{A-B} and defined through $T_{A-B}[i].C = T_A[i].C - T_B[i].C$, $T_{A-B}[i].P = T_A[i].P \oplus T_B[i].P$, and $T_{A-B}[i].H = T_A[i].H \oplus T_B[i].H$. Information about elements of $A \cap B$ is "cancelled out" in T_{A-B}, that is, T_{A-B} holds elements of $(A \setminus B) \cup (B \setminus A)$. Peeling then proceeds as usual, listing both $A \setminus B$ and $B \setminus A$ with the distinction made possible by looking at the sign of C.

A remarkable property of this scheme is that it allows one to recover set differences using a space proportional to the size of those differences regardless the size of the involved sets. Indeed, for the peeling process to succeed w.h.p., it is sufficient that the size of T_{A-B} be $O(n)$ where $n = |(A \setminus B) \cup (B \setminus A)|$ (see (1)). This is particularly suitable for the bioinformatics framework where we are often dealing with highly similar datasets, such as genomes of different individuals or closely related species.

3.2 Making buckets lighter

In the above scheme of IBLT difference, the H field becomes important as the case $T_{A-B}[i].C = 1$ (or $T_{A-B}[i].C = -1$) can occur due to a spurious "cancelling out" of distinct keys. However, to save space, we propose to get rid of the H field and replace the "checksum" verification by another test: if $T_{A-B}[i].C = 1$ (resp. $T_{A-B}[i].C = -1$), we check whether $p = T_{A-B}[i].P$ is a valid key by checking if $h_j(p) = i$ for one of $j \in [1..r]$. This allows us to save space at the price of additional verification time. This technique works particularly well for large IBLTs but it becomes less effective for small ones, as the "false positive" probability is proportional to the size of the table.

3.3 Combining sampling and IBLTs for Jaccard similarity estimation

We now turn to our main goal: estimating Jaccard similarity of two k-mer sets using IBLTs. The common approach uses MinHash sketching as described in [24] (see Section 2.4). However, MinHash requires larger sketches to measure similarity of close datasets. One possible idea could be to store MinHash sketches in IBLTs in hope to use them for estimating Jaccard similarity through the IBLT-difference scheme from the previous section. This, however, runs into an obstacle due to the fact that applying (2) requires knowledge of k-mers belonging to the sketch intersection, and not only to sketch differences.

Rather than working with the entire sets of k-mers, we resort to sampling. It is known that sampling minimizers incurs a bias in estimating Jaccard similarity [1]. Instead, we propose to use syncmers, which don't suffer from being context-dependent thus resulting in an unbiased estimator of Jaccard similarity.

To justify the use of syncmers, we test a standard hash-based sampling, also providing an unbiased estimate of Jaccard similarity as well. To sample with a given sampling rate $1/\nu$, hash-based sampling uses a random hash function $h : \Sigma^k \to [0..\nu - 1]$ with good statistical properties, and samples a k-mer q iff $h(q) = 0$.

Our approach consists in storing sampled k-mers in IBLTs and apply the IBLT-difference technique to recover set differences. Then, Jaccard similarity is estimated by

$$J(A, B) = \frac{|A| - |A \setminus B|}{|A| + |B \setminus A|} = \frac{|B| - |B \setminus A|}{|B| + |A \setminus B|}. \tag{3}$$

Note that cardinalities $|A|$ and $|B|$ can be easily retrieved from respective IBLTs T_A and T_B by summing all counter values and dividing by r.

3.4 IBLT dimensioning with syncmers

Dimensioning an IBLT holding syncmers requires estimating the expected number of differences in the set difference of involved k-mer sets. Assuming that input datasets are close genomic sequences of size L related by a mutation rate bounded by p_m and that k is sufficiently large so that k-mer occurrences are unique, we can estimate the set difference. Each mutation results in $2k$ k-mers in the set difference (k k-mers on each side), and therefore the size of set difference is estimated to be $2kp_mL$. Taking into account density $\frac{2}{k-z+1}$ of syncmers (Section 2.2), we obtain the estimation

$$n = \frac{4kLp_m}{k - z + 1}. \tag{4}$$

3.5 Approximating k-mer set differences

The method of Section 3.3 allows estimating Jaccard similarity on k-mers by Jaccard similarity on syncmers. Here we describe how we can extend these ideas in order to recover *all* k-mers from $K(S_1) \setminus K(S_2)$ and $K(S_2) \setminus K(S_1)$, where S_1, S_2 are input datasets and $K(S)$ denotes the set of k-mers of a dataset S.

Note first that a straightforward way of doing this, through IBLTs of $K(S_1)$ and $K(S_2)$, requires a considerable space because a single mutation generates a difference of k k-mers. Using syncmers, we can "pack" k-mers into longer strings, compute the differences and then recover k-mers from them. The set of recovered k-mers, however, will be a superset of exact differences.

To achieve this, instead of storing syncmers, we store in IBLTs *extended* syncmers of length $2k - z$. Extended syncmers are obtained by extending each syncmer to the right by $k - z$ bases. Since successive syncmers overlap by at least z bases, this ensures that each k-mer belongs to at least one extended syncmer.

By applying the IBLT-difference technique (Section 3.3), we obtain the extended syncmers that differ between the two datasets, from which we extract k-mers and discard those shared between the two obtained sets. It may still happen that the sets we obtain are *supersets* of exact differences, due to the fact that an extended syncmer can contain a k-mer which belongs to another extended syncmer common to both datasets. However, we state that for a sufficiently large k, the fraction of common k-mers in those sets will be small enough, which we illustrate experimentally in Section 4.5. In the extreme case where each k-mer occurs once, our method computes exact k-mer set differences.

4 Results

To validate our ideas, we performed experiments on simulated sequences as well as on two real-life datasets:

- `covid`: subsample of 50 *SARS-CoV-2* genomes[1]. Sequence names are provided in Table 2.
- `spneu`: subsample of 28 *Streptococcus Pneumoniae* genomes from [5] whose names are reported in Table 3. The subsample has been chosen to contain very close strains, with pairwise mutation rates between them not exceeding 0.0005.

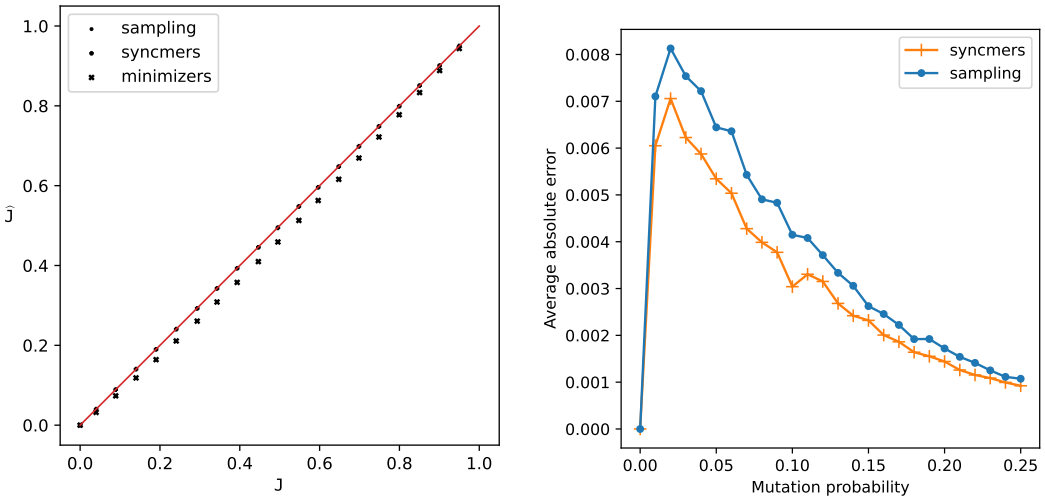
4.1 Comparison of different sampling approaches

Random sampling, minimizers and syncmers have been compared by computing Jaccard similarities between pairs of synthetic sequences. Each pair is constructed by first generating a uniform random sequence of length L and then mutating it through independent substitutions. Points in Figure 1a are averages over $T = 500$ independent trials. For fairness of comparison, parameters for uniform sampling, minimizers and syncmers have been chosen to guarantee the same sampling rate $1/\nu$. We know that $c_s \approx 2/(k-z+1)$, $c_m \approx 2/(w+1)$ and $c_{ns} \approx 1/\nu$ are the densities of syncmers, minimizers and random sampling, respectively. Thus, given parameters k and z, setting the minimizer window length as $w = k - z$ and choosing a sampling rate $1/\nu = c_s$ ensures about the same number of sampled k-mers for all algorithms. As Figure 1a shows, syncmers do not have the previously reported biased behaviour of minimizers [1], but they seem to be comparable to random sampling. However, as shown in Figure 1b, random sampling is subject to larger errors than syncmers, due to less uniform distribution along the sequence. For these reasons, we choose syncmer sampling as the mean to reduce IBLT memory in Section 4.3.

4.2 Space performance of IBLTs

In order to demonstrate the space efficiency of IBLTs in our framework, we compare them against a solution based on KMC k-mer counting software [17]. KMC provides an efficient way for storing, manipulating and querying sets of k-mers. Unlike other counting tools (Jellyfish [21] or DSK [27]), KMC allows easy sorting of its output which leads to an efficient way to compute Jaccard similarity.

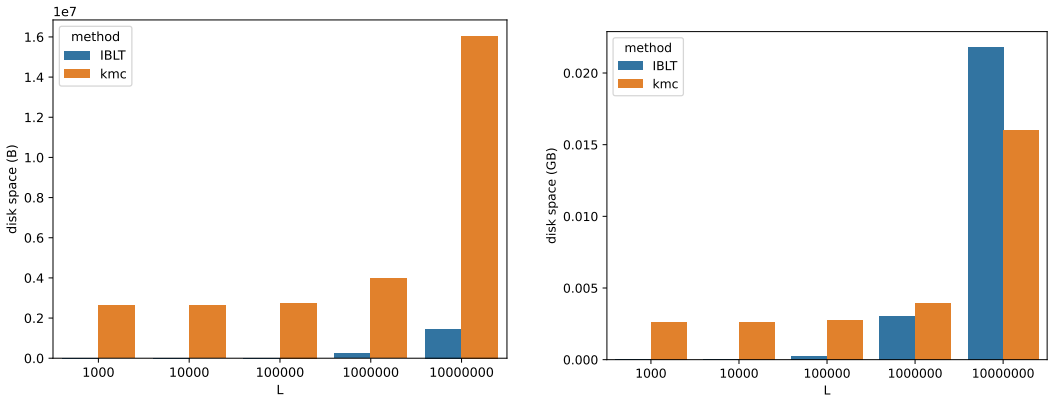
[1] `https://www.ncbi.nlm.nih.gov/datasets/coronavirus/genomes/`



(a) Minimizers present a non-negligible bias as opposed to syncmers and random sampling which are unbiased (and overlap in the plot). Each measurement was repeated 500 times on random sequences of length $L = 10\text{K}$ with $k = 15$, $w = 11$ (for minimizers) and $z = 4$ (for syncmers). Sampling rate is given by $1/\nu = 2/(k - z + 1) = 1/6$.

(b) Absolute average errors for syncmers and random sampling, as a function of mutation rate. Parameters are unchanged from Figure 1a since both plots are generated from the same data.

Figure 1 Comparison between random sampling, minimizers and syncmers.



(a) $p_m = 0.001$.

(b) $p_m = 0.01$.

Figure 2 Space taken by IBLTs depends on the similarity between stored sets. For very similar sequences (mutation rate $p_m = 0.001$, Figure 2a), IBLTs are more space-efficient than KMC. Their advantage appears reduced for increased p_m and large sequences (Figure 2b).

We compared memory taken by IBLTs vs. KMC databases for storing syncmers issued from two similar sequences. For this, we applied the same procedure as in Section 4.1: mutating a random sequence of length L with mutation probability p_m. Sampled syncmers from both sequences are stored respectively in IBLTs and KMC databases. Figure 2 reports average space taken by the two data structures. Each bar is the average over $T = 100$ trials, except for case $L = 10\text{M}$ for which $T = 10$. IBLTs were dimensioned (see (1)) to guarantee peelability of all T sketches with high probability.

Figure 2a clearly demonstrates the advantage of IBLTs when the mutation rate is small. For larger p_m and long sequences, the number of differences reach a point where exact data structures become preferable, as illustrated by Figure 2b for $p_m = 0.01$ and sequences of length 10M.

In our experiments, subtracting one IBLTs from another is dominated by the time taken to load/save the sketches, and not by performing the actual difference. Even in more complex scenarios, subtraction remains a very simple operation that can be performed by accessing one bucket at a time in any given order. On the other hand, the amount of time required by listing the content of an IBLT varies greatly and depends on the set of items stored in it.

4.3 Accuracy of Jaccard similarity estimation from IBLTs of syncmers

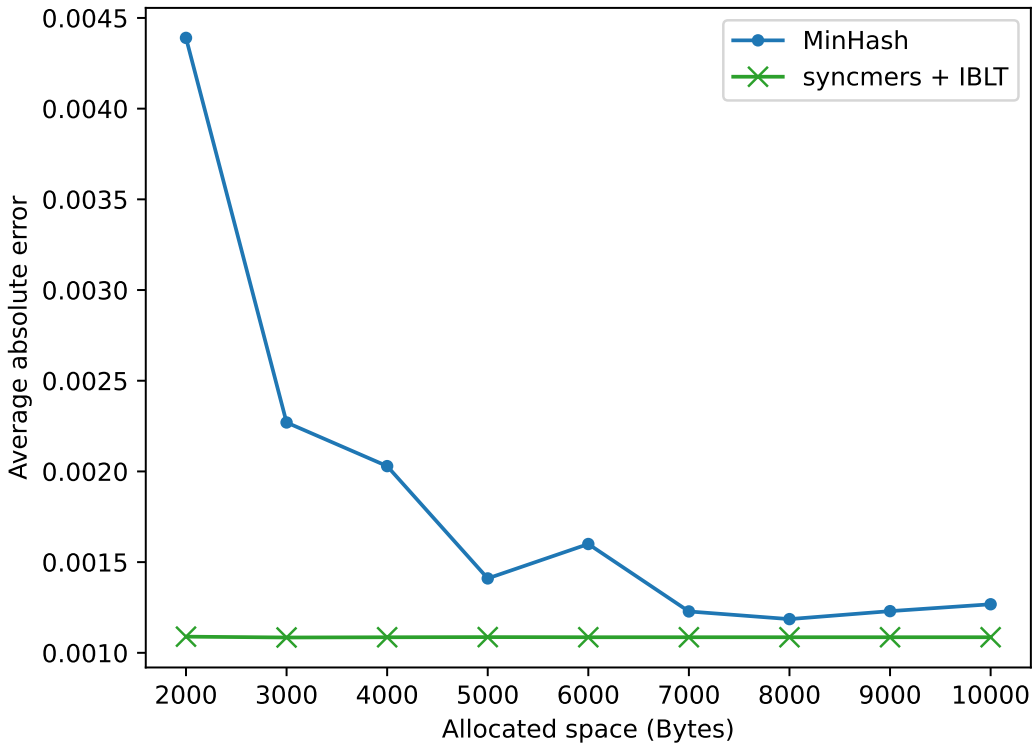


Figure 3 Comparison between IBLTs and MinHash for computing pairwise Jaccard on the `covid` dataset. The x-axis reports the amount of space allocated for each sketch while the y-axis reports the average absolute error. $k = 15$ and $z = 4$ in all tests. Sketch size for MinHash and table size for IBLTs are chosen to fit the allocated memory.

Figures 3 and 4 report comparisons of both IBLTs and MinHash sketches on `covid` and `spneu` datasets respectively. Both plots show the average absolute error of Jaccard estimate computed over all pairs of sequences of the respective dataset. Exact Jaccard similarities computed over the full k-mer sets are used as ground truth. MinHash sketches (line MinHash in the plots) were implemented using Mash [24]. All sketch sizes (in bytes) are fixed beforehand with both MinHash sketches and IBLTs dimensioned accordingly in order to fit the allocated memory. The number of bits allocated for payload field P in our IBLT implementation is set to be the minimum multiple of 8 larger than or equal to $2k$. As Mash [24] uses 32- or 64-bit hashes, we used $k = 15$ in our experiments in order to force both methods to use 32-bit representations.

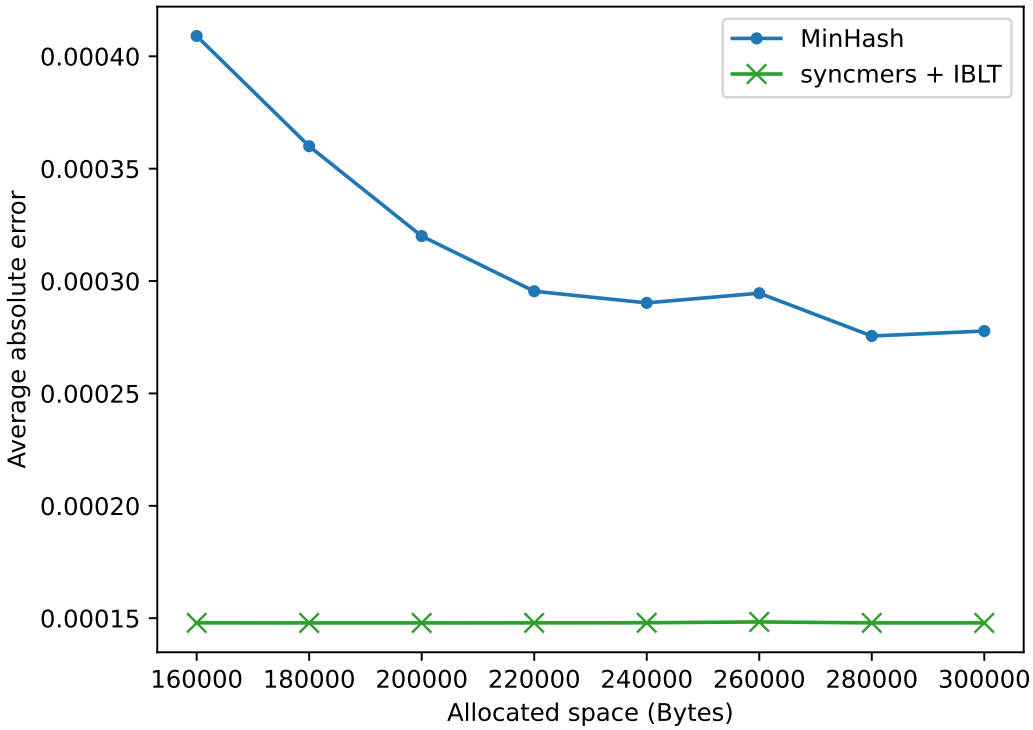


Figure 4 Comparison between IBLTs and MinHash for computing pairwise Jaccard on the `spneu` dataset with the same setting as Figure 3.

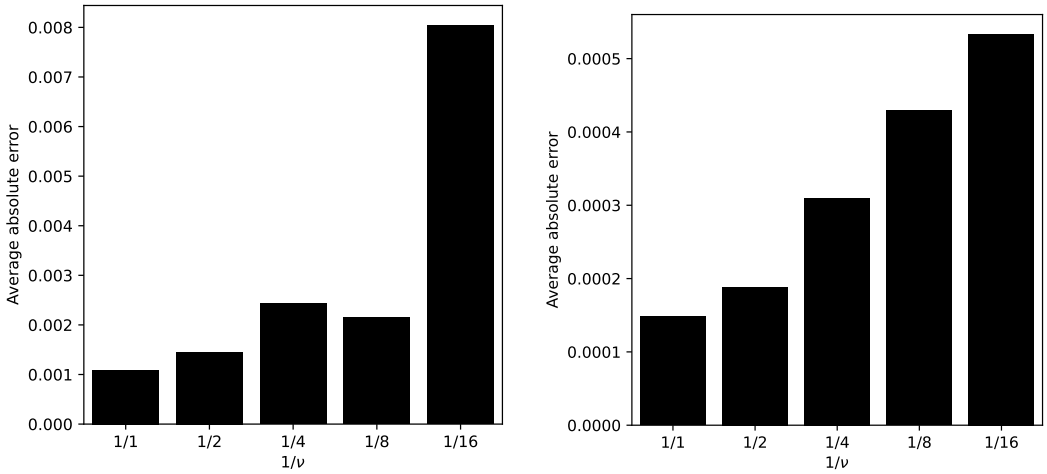
In all experiments, IBLTs storing syncmers (line SYNCMERS + IBLT) showed the best precision. For covid genomes (Figure 3), full MinHash sketches become competitive for larger sketch sizes. Unlike MinHash, the average error of IBLTs remains constant across all reported cases because over-dimensioning only increases the probability of successful listing. For the `spneu` dataset (Figure 4), MinHash errors are about twice those of IBLTs across all allocated sketch sizes confirming that IBLTs are more memory-efficient. The general conclusion is that if sequences to be compared are highly similar, IBLTs storing syncmers are more efficient than MinHash sketches, with the latter being better suited to quickly provide an overview over more diverging datasets.

4.4 Sampling syncmers for further space reductions

Since syncmer sampling rate ($\frac{2}{k-z+1}$) cannot be made arbitrarily small for a given k, we also tested the effect of additional downstream sampling of syncmers, before inserting them into IBLTs. To this end, Figure 5 reports a comparison of syncmers sampled with different sampling rates $1/\nu$. We observe that downstream sampling of syncmers comes at the cost of decreased precision for both datasets (Figure 5a and 5b), but it might be useful to further reduce space.

4.5 Experiments on approximating k-mer set differences

We tested the method from Section 3.5 of approximating k-mer set differences on both the `covid` dataset and on two random datasets. Each random dataset contains 50 sequences of length 30000 obtained by first generating a uniform random sequence which is then mutated 49 times using a mutation probability p_m.



(a) `covid` dataset. **(b)** `spneu` dataset.

Figure 5 Effect of sampling syncmers before IBLT insertion on the average absolute error. $1/\nu$ is the compression rate used for sampling syncmer sets before IBLT insertion. $\nu = 1$ means no sampling (full syncmer sets).

Recall that the method of Section 3.5 allows one to compute a *superset* of the symmetric difference $(K(S_1) \setminus K(S_2)) \cup (K(S_2) \setminus K(S_1))$ of sets of k-mers occurring in datasets S_1 and S_2. Here we measure the precision of this method, that is the number of spurious k-mers found by the algorithm. Those are k-mers actually belonging to $K(S_1) \cap K(S_2)$ but output by the algorithm as if they belong to $(K(S_1) \setminus K(S_2)) \cup (K(S_2) \setminus K(S_1))$.

Table 1 summarizes the experiments. Columns "diff" and "err" show the average/maximum cardinality of the true set difference and spurious k-mers, respectively, over all pairs of sequences. In the case of random datasets, sequences were generated with mutation probabilities $p_m = 0.01$ and $p_m = 0.001$.

Table 1 True size of symmetric difference of k-mer sets and its overestimate. For each experiment, "diff" is the average/maximum size of the true symmetric difference, and "err" is the average/maximum number of spurious k-mers reported as being in the symmetric difference. p_m is the mutation probability used to generate sequences from a random one.

	`covid`		random			
			$p_m = 0.001$		$p_m = 0.01$	
	diff	err	diff	err	diff	err
average	325.29	11.03	1708.78	60.03	15342.81	357.57
max	661	31	2396	110	17047	486

We observe that the number of spurious k-mers remains small, on average within about 3% of the true set difference size.

5 Conclusions

To the best of our knowledge, our work is the first to apply Invertible Bloom Lookup Tables to k-mer processing for alignment-free comparison of DNA sequence datasets. We showed that whenever involved datasets are similar enough and their similarity can be bounded *a priori*, IBLTs lead to a more space-efficient and, at the same time, more accurate method

for estimating Jaccard similarity of underlying k-mer sets. This is achieved by combining IBLTs with k-mer sampling via syncmers. As opposed to minimizers, syncmers provide an unbiased estimator of Jaccard index, which was confirmed in our experiments. At the same time, syncmer sampling is shown to lead to a more concentrated estimator than the straightforward hash-based sampling. Thus, IBLTs combined with syncmers constitute a powerful alternative to MinHashing for estimating Jaccard similarity for similar datasets. Note that in the context of viral/bacterial pan-genomics, dealing with similar datasets is a predominant situation in bioinformatics. In particular, the number of closely related bacterial and viral strains is rapidly growing.

As another application of IBLTs, we are able to approximately compute differences of underlying k-mer sets in small space. This opens new prospects as k-mers proper to a dataset can be used to infer information about genetic variation, specific mutation, etc. Note that MinHash is designed to only estimate similarity and is not capable of providing information about actual differences. We also believe that by using additional space-efficient data structures this method can be extended to compute *exact* set differences on more complex datasets and we plan to explore this in our future work.

Our ideas may have further useful applications, for example to reconciliation of datasets located on remote computers, in which case IBLTs could avoid transmitting entire datasets (similar to a scenario described in [12]). Another example is a selection of sufficiently diverse datasets avoiding redundancy, as e.g. [18]. Note finally that IBLTs may also act as filters for filtering out dissimilar datasets: in this case, non-peelability of the difference IBLT is an indicator of dissimilarity.

References

1 Mahdi Belbasi, Antonio Blanca, Robert S. Harris, David Koslicki, and Paul Medvedev. The minimizer Jaccard estimator is biased and inconsistent. *bioRxiv*, 2022. doi:10.1101/2022.01.14.476226.

2 Timo Bingmann, Phelim Bradley, Florian Gauger, and Zamin Iqbal. COBS: A Compact Bit-Sliced Signature Index. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval*, Lecture Notes in Computer Science, pages 285–303, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-32686-9_21.

3 Phelim Bradley, Henk C Den Bakker, Eduardo P. C. Rocha, Gil McVean, and Zamin Iqbal. Ultra-fast search of all deposited bacterial and viral genomic data. *Nature biotechnology*, 37(2):152–159, February 2019. doi:10.1038/s41587-018-0010-1.

4 Andrei Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 21–29, June 1997. doi:10.1109/SEQUEN.1997.666900.

5 Karel Břinda, Alanna Callendrello, Kevin C. Ma, Derek R. MacFadden, Themoula Charalampous, Robyn S. Lee, Lauren Cowley, Crista B. Wadsworth, Yonatan H. Grad, Gregory Kucherov, Justin O'Grady, Michael Baym, and William P. Hanage. Rapid inference of antibiotic resistance and susceptibility by genomic neighbour typing. *Nature Microbiology*, 5(3):455–464, March 2020. doi:10.1038/s41564-019-0656-6.

6 Dan DeBlasio, Fiyinfoluwa Gbosibo, Carl Kingsford, and Guillaume Marçais. Practical universal k-mer sets for minimizer schemes. In *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, BCB '19, pages 167–176, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3307339.3342144.

7 Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via xorsat. In *Proceedings of the 37th International Colloquium Conference on Automata, Languages and Programming*, ICALP'10, pages 213–225, Berlin, Heidelberg, 2010. Springer-Verlag.

8 Robert Edgar. Syncmers are more sensitive than minimizers for selecting conserved k-mers in biological sequences. *PeerJ*, 9:e10805, February 2021. doi:10.7717/peerj.10805.

9 Barış Ekim, Bonnie Berger, and Yaron Orenstein. A Randomized Parallel Algorithm for Efficiently Finding Near-Optimal Universal Hitting Sets. In Russell Schwartz, editor, *Research in Computational Molecular Biology*, Lecture Notes in Computer Science, pages 37–53, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-45257-5_3.

10 David Eppstein and Michael T. Goodrich. Straggler identification in round-trip data streams via Newton's identities and invertible Bloom filters. *IEEE Transactions on Knowledge and Data Engineering*, 23(2):297–306, 2011. doi:10.1109/TKDE.2010.132.

11 Huan Fan, Anthony R. Ives, Yann Surget-Groba, and Charles H. Cannon. An assembly and alignment-free method of phylogeny reconstruction from next-generation sequencing data. *BMC Genomics*, 16(1):522, July 2015. doi:10.1186/s12864-015-1647-5.

12 Michael T. Goodrich and Michael Mitzenmacher. Invertible Bloom lookup tables, 2011. doi:10.48550/ARXIV.1101.2245.

13 Gaurav Gupta, Minghao Yan, Benjamin Coleman, R. A. Leo Elworth, Todd Treangen, and Anshumali Shrivastava. Sub-linear Sequence Search via a Repeated And Merged Bloom Filter (RAMBO): Indexing 170 TB data in 14 hours. *arXiv:1910.04358 [cs, q-bio]*, December 2019. arXiv:1910.04358.

14 Robert S Harris and Paul Medvedev. Improved representation of sequence Bloom trees. *Bioinformatics*, 36(3):721–727, August 2019. doi:10.1093/bioinformatics/btz662.

15 Chirag Jain, Luis M. Rodriguez-R, Adam M. Phillippy, Konstantinos T. Konstantinidis, and Srinivas Aluru. High throughput ANI analysis of 90K prokaryotic genomes reveals clear species boundaries. *Nature Communications*, 9(1):5114, November 2018. doi:10.1038/s41467-018-07641-9.

16 Mikhail Karasikov, Harun Mustafa, Gunnar Rätsch, and André Kahles. Lossless indexing with counting de bruijn graphs. *bioRxiv*, 2022. doi:10.1101/2021.11.09.467907.

17 Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, May 2017. doi:10.1093/bioinformatics/btx304.

18 Nathan LaPierre, Mohammed Alser, Eleazar Eskin, David Koslicki, and Serghei Mangul. Metalign: efficient alignment-based metagenomic profiling via containment min hash. *Genome Biology*, 21(1):242, September 2020. doi:10.1186/s13059-020-02159-0.

19 Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, May 2018. doi:10.1093/bioinformatics/bty191.

20 Camille Marchet, Christina Boucher, Simon J. Puglisi, Paul Medvedev, Mikaël Salson, and Rayan Chikhi. Data structures based on k-mers for querying large collections of sequencing data sets. *Genome Research*, 31(1):1–12, January 2021. doi:10.1101/gr.260604.119.

21 Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764, March 2011. doi:10.1093/bioinformatics/btr011.

22 Michael Molloy. The pure literal rule threshold and cores in random hypergraphs. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '04, pages 672–681, USA, January 2004. Society for Industrial and Applied Mathematics.

23 Martin D Muggli, Bahar Alipanahi, and Christina Boucher. Building large updatable colored de Bruijn graphs via merging. *Bioinformatics*, 35(14):i51–i60, July 2019. doi:10.1093/bioinformatics/btz350.

24 Brian D. Ondov, Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren, and Adam M. Phillippy. Mash: fast genome and metagenome distance estimation using MinHash. *Genome Biology*, 17(1):132, June 2016. doi:10.1186/s13059-016-0997-x.

25 Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *bioRxiv*, 2022. doi:10.1101/2022.01.15.476199.

26 Ely Porat and Ohad Lipsky. Improved sketching of hamming distance with error correcting. In Bin Ma and Kaizhong Zhang, editors, *Combinatorial Pattern Matching*, pages 173–182, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

27 Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. DSK: k-mer counting with very low memory usage, March 2013. doi:10.1093/bioinformatics/btt020.

28 Michael Roberts, Wayne Hayes, Brian R. Hunt, Stephen M. Mount, and James A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, December 2004. doi:10.1093/bioinformatics/bth408.

29 Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. Using cascading Bloom filters to improve the memory usage for de Brujin graphs. *BMC Algorithms for Molecular Biology*, 9(1):2, 2014. URL: http://www.almob.org/content/9/1/2.

30 Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 76–85, San Diego, California, June 2003. Association for Computing Machinery. doi:10.1145/872757.872770.

A Appendix

Datasets

Table 2 Names of covid genomes used for Figure 3.

BS001151.1	LR877722.1	LR883214.1	MT520216.1
MT706180.1	MT757082.1	MT800758.1	MT834020.1
MT970159.1	MT971010.1	MT973151.1	MW064390.1
MW064919.1	MW064981.1	MW153809.1	MW153954.1
MW154711.1	MW156712.1	MW184416.1	MW184648.1
MW190904.1	MW190957.1	MW191020.1	MW191146.1
MW206148.1	MW276931.1	MW321243.1	MW321430.1
MW593629.1	MW631874.1	MW669599.1	MW681303.1
MW681489.1	MW693959.1	MW696216.1	MW702101.1
MW708072.1	MW708184.1	MW708826.1	MW720341.1
MW733722.1	MW738615.1	MW749542.1	MW776764.1
MW820211.1	MW850083.1	MW863243.1	MW868532.1
MW868533.1	MW871079.1		

Table 3 Names of *S.Pneumoniae* genomes used for Figure 4.

BZ2I7.fa	R34-3087.fa	007649.fa	R34-3097.fa
4PYM0.fa	JBYFY.fa	T8Z8O.fa	R34-3044.fa
O61U7.fa	81LMX.fa	O0RHB.fa	R34-3083.fa
R34-3025.fa	WAMFH.fa	O8I1E.fa	R34-3164.fa
CCV1H.fa	0U64I.fa	6893Z.fa	1VDX8.fa
R34-3074.fa	R34-3227.fa	LS3OB.fa	UTEDZ.fa
REAOU.fa	R34-3229.fa	067094.fa	4K4C9.fa

WGSUniFrac: Applying UniFrac Metric to Whole Genome Shotgun Data

Wei Wei
The Pennsylvania State University, University Park, PA, USA

David Koslicki[1]
Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA, USA
Department of Biology, The Pennsylvania State University, University Park, PA, USA
Huck Institutes of the Life Sciences, The Pennsylvania State University, University Park, PA, USA

Abstract

The UniFrac metric has proven useful in revealing diversity across metagenomic communities. Due to the phylogeny-based nature of this measurement, UniFrac has historically only been applied to 16S rRNA data. Simultaneously, Whole Genome Shotgun (WGS) metagenomics has been increasingly widely employed and proven to provide more information than 16S data, but a UniFrac-like diversity metric suitable for WGS data has not previously been developed. The main obstacle for UniFrac to be applied directly to WGS data is the absence of phylogenetic distances in the taxonomic relationship derived from WGS data. In this study, we demonstrate a method to overcome this intrinsic difference and compute the UniFrac metric on WGS data by assigning branch lengths to the taxonomic tree obtained from input taxonomic profiles. We conduct a series of experiments to demonstrate that this WGSUniFrac method is comparably robust to traditional 16S UniFrac and is not highly sensitive to branch lengths assignments, be they data-derived or model-prescribed.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Applied computing → Bioinformatics; Applied computing → Computational genomics

Keywords and phrases UniFrac, beta-diversity, Whole Genome Shotgun, microbial community similarity

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.15

Supplementary Material
Software (Prototype of WGSUniFrac): `https://github.com/KoslickiLab/WGSUniFrac`, archived at `swh:1:dir:d4a54046a885b69bdfdd5ca37d336ff7e51eace2`
Software (to reproduce results of this paper): `https://github.com/KoslickiLab/WGSUniFrac-reproducibles`, archived at `swh:1:dir:16f79da3471f763bd5649d1d40467ea47f3b0a9f`

Funding *Wei Wei*: NSF Grant No. 2029170
David Koslicki: NSF Grant No. 2029170

1 Introduction

The study of microbial composition and diversity has demonstrated its value in both clinical [13, 9, 6] and environmental [41] studies. Within-sample diversity (known also as alpha-diversity) metrics, such as the Shannon index and Simpson diversity, have been used to evaluate and quantify microbial diversity in various settings [24]. In contrast, between-sample (or, beta-diversity) measurements allow measurement and analysis of differences across multiple samples, giving insights to their significance [55, 19, 56]. Among the most frequently utilized beta-diversity metrics is UniFrac [31, 32, 30, 16, 37, 53].

[1] Corresponding author

UniFrac measures the phylogenetic differences between two microbial communities by calculating the fraction of branch lengths unique to one of the two communities on a phylogenetic tree that has been annotated with the predicted abundances of organisms in the two communities [33]. This computation is established on the intuition that the degree to which two communities or environments differ is positively correlated to the degree of difference in the evolutionary path undergone that resulted in the observed divergence: the longer the evolutionary path, the more divergent [31]. Since its introduction in 2005, the UniFrac distance has been widely applied [55, 16, 13]. Its strengths over other beta-diversity measures has been demonstrated [28] and its robustness has stood the test of time [32]. Over time, the UniFrac metric has undergone a series of developments ranging from conceptual understanding and application to computation efficiency. The variation of weighted UniFrac was introduced two years after the introduction of the original unweighted version [33]. Fast UniFrac made its debut in 2010, improving the speed of UniFrac computation, hence expanding its application to larger datasets [17]. In 2012, the understanding of the UniFrac distance being equivalent to the earth mover's distance was brought to light [14], based on which an exact linear-time computation algorithm, EMDUniFrac, was later developed [35] and then later implemented in Striped UniFrac [37]. All these demonstrate the popularity and potential of the UniFrac metric.

In this paper, we discuss the possibility of applying the UniFrac metric to a new type of data: whole genome shotgun metagenomic samples. Traditionally, UniFrac has been employed almost exclusively in the analysis of 16S rRNA sequencing data. The 16S rRNA sequencing method involves amplification and sequencing of the 16S small subunit ribosomal RNA which contains both highly conserved and variable regions, leading to a simple and cost effective "fingerprinting" approach to inferring microbial composition [47, 48]. An alternative approach to 16S rRNA sequencing is whole genome shotgun sequencing (WGS). Despite requiring more effort and cost, the advantages of WGS analysis are also apparent: higher accuracy, sensitivity, and access to the entirety of the genetic material in a given sample [48]. Additionally, WGS data are becoming more frequently utilized by clinicians and biologists [5, 3] due in part to the ever-decreasing price.

Though UniFrac is widely employed in the analysis of 16S rRNA and other amplicon studies, it has yet to find its application in WGS metagenomic data. While 16S rRNA and other amplicon sequencing approaches naturally have a single gene to build a phylogeny with, there is no consensus in the metagenomic community on how to best construct a phylogenetic tree from WGS data, with approaches ranging from a variety of single gene approaches [29, 42, 51], whole genome alignment approaches [54, 15], to k-mer based similarity techniques [43, 27, 46]. As such, researchers have primarily focused on utilizing taxonomic trees instead of phylogenetic trees due to the relative ease of identifying taxa present in a sample [44, 50, 39]. Since UniFrac was originally intended for usage on a phylogenetic tree, this difference in underlying tree structure in amplicon studies versus WGS studies explains why UniFrac has not been used in WGS metagenomic analyses. In particular, the absence of phylogenetic relationship among taxa in a taxonomic tree, as well as evolutionary distances reflected in branch lengths, hinders the direct computation of UniFrac. Even so, the robustness of UniFrac demonstrated in numerous amplicon studies motivates the endeavor to overcome this intrinsic difficulty and extend its application to WGS data.

In this paper, we demonstrate that by assigning branch lengths to the corresponding taxonomic tree, UniFrac can be applied to WGS data and achieve reasonable robustness. We call this extension WGSUniFrac. We investigated the effect of branch lengths assignments on the computational power of WGSUniFrac, laying the foundation of extending the application of UniFrac to more general structures. A summary of how WGSUniFrac works is shown in

Figure 1. Code implementing a prototype of WGSUniFrac is available at `https://github.com/KoslickiLab/WGSUniFrac` while the results presented in this paper can be reproduced using the code at `https://github.com/KoslickiLab/WGSUniFrac-reproducibles`.

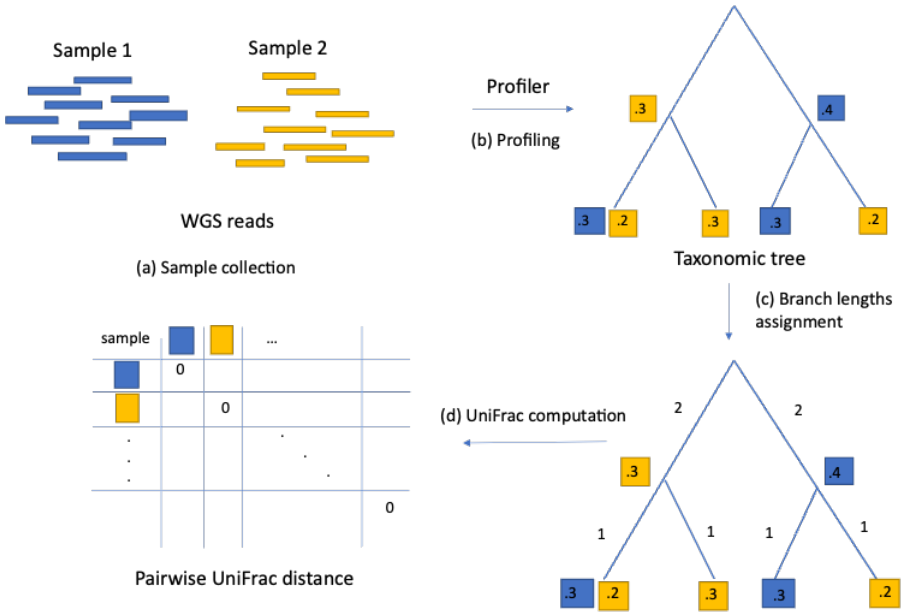


Figure 1 An illustration of the WGSUniFrac workflow. (a) WGS Metagenomic samples are collected. (b) Each sample is converted to its corresponding taxonomic profile using a profiler of choice. Each profile contains the relative abundances of all the taxa present in the sample at all taxonomic levels. The collection of all profiles form a taxonomic tree. (c) Branch lengths are assigned to the taxonomic tree according to branch lengths function specified. In this case, the branches are assigned lengths inversely proportional to their distance from the root. (d) Pairwise UniFrac values of all samples are computed using the EMDUniFrac algorithm.

2 Methods

The UniFrac metric was first defined in 2005 by Lozupone et al. as the fraction of branch lengths unique to only one of the two communities being compared on a phylogenetic tree [31]. This original version of UniFrac (also known as the unweighted UniFrac) is a qualitative measure that decides if two communities differ significantly based on if the computed UniFrac is greater than what would be expected by chance [31]. The weighted UniFrac metric was introduced soon after to offer insights to the degree of differences by taking into consideration the relative abundances of the organisms [33], and the original computation is given by:

$$u = \sum_i^n b_i \times \left| \frac{A_i}{A_T} - \frac{B_i}{B_T} \right| \tag{1}$$

where n is the total number of branches on the tree, b_i is the length of branch i, A_i and B_i represent the number of sequences descended from branch i in communities A and B respectively, and A_T and B_T are the respective total number of sequences for the purpose of normalizing the abundances in the case of uneven sample sizes for communities A and B [33]. The original UniFrac was only intended for an application on phylogenetic trees reflecting the evolutionary relationship amongst the organisms and on which all the abundances are found on the leaf nodes.

In a previous study, it has been demonstrated that the weighted UniFrac distance is equivalent to the Kantorovich-Rubinstein metric, also known as the earth mover's distance [14]. Under this definition, instead of building a phylogenetic tree from scratch using the samples, a pre-existing reference tree can be used [14]. By mapping the reads to the appropriate nodes on the reference tree through comparative methods, the information of relative abundances gets incorporated into the tree. The equivalence with the earth mover's distance then allows us to view the UniFrac distance in a new light: viewing the relative abundances as piles of sand, the UniFrac can be defined as the minimum amount of work required to move the sand from the configuration of one sample to match that of the other, with the amount of work being defined as mass multiplied by the total distance traveled along the tree branches [14]. This gives us an alternative formulation of UniFrac which will be described below.

Let T be a rooted tree with n nodes ordered from leaves to the root ρ representing organisms and branch lengths proportional to evolutionary distances. For a node i in T, define depth(i) as the number of branches on the shortest path from i to the root node. We impose a partial ordering on the set of all nodes in T in terms of depth: a node i is below a node j if depth(i) > depth(j). Represent a branch length by $l(i)$, indicating the weight on the branch connecting node i to its ancestor $a(i)$. Let P and Q be vectors of probability distribution on the tree with non-negative entries summing up to 1, representing the relative abundance of each organism/taxa on the tree in the two input samples respectively, ordered from leaves to the root. Given a node i in T, let T_i be a subtree of T not containing ρ obtained by deleting $(i, a(i))$. Define w_i to be an indicator function that represents a subtree rooted at node i such that the j-th entry of w_i equals 1 if $(j, a(j))$ is a node in the subtree rooted at i, and 0 otherwise. I.e.

$$w_i(j) = \begin{cases} 1 & \text{if } j \text{ is a node on } T_i \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Now let W be an $n \times n$ matrix with column i given by w_i and each row j scaled by $l(j, a(j))$. The UniFrac distance (1) can then be represented equivalently as [34, Lemma 2.2.1], [38, Suppl. pg 10]

$$||W(P - Q)||_{L_1}. \tag{3}$$

This formulation not only allows the exact UniFrac distance to be computed in linear time [35] but also allows UniFrac to be computed on any tree, not necessarily a phylogenetic one. This allows us to draw one step closer to the application of UniFrac on WGS data, with which a phylogenetic tree is in general impossible to be built, but a taxonomic tree instead. The only obstacle of a direct application lies in the absence of branch lengths $l(i)$ on taxonomic trees. As a solution we incorporate the assignment of branch lengths according to a given branch lengths function into the algorithm of WGSUniFrac (Algorithm 1) prior to the computation of UniFrac with the EMDUniFrac implementation.

In general, taxonomic trees do not have a natural notion of "branch lengths" as in a phylogenetic tree. As such, we can impose a functional form for the branch $l(i) = f(i, a(i))$ where $f(i, a(i))$ is some function that assigns lengths to branches based on some biologically reasonable form. For example, in the Results section below, we chose $f(i, a(i)) := \text{depth}(i)^k$ for $k \in \mathbb{Z}$. Defining f in this way means branch lengths are assigned uniformly at each depth, with lengths increasing (or decreasing, depending on the sign of k) the further the branches are from the root.The exploration of other values of k and their impact on the performance of WGSUniFrac can be found under the Results section. One can also imagine a data-derived

Listing 1 WGSUniFrac Algorithm where P and Q are probability vectors with entries representing relative abundances summing up to 1, T being the taxonomic tree, and f being a function that maps a branch to its length.

```
1 Input: P, Q, f, T
2 Initialization: M = P - Q, unifrac = 0
3 for i in 1 ... |T| do #Ordered from the leaves to the root
4     v = M[i]
5     M[a(i)] = M[a(i)] + v
6     l(i) = f(i,a(i))
7     unifrac = unifrac + l(i) * |v|
8 return unifrac
```

definition of the branch lengths if given access to, say, the rate of accumulation of mutations for an organism belonging to the taxonomic clade defined by the node i. In this exposition, the exact form of f does not impact the algorithm we describe.

We now give a complete description of the WGSUniFrac algorithm below. Given a rooted tree T with nodes ordered from leaves to the root, represented by an edge set $E = \{(i, a(i))\}$ for $i \in T$, with $a(i)$ being the ancestor of node i; probability distribution vectors P and Q representing relative abundances in two samples respectively. For $i \in T$, let $l(i) = f(i, a(i))$ for some function f which the user specifies.

This algorithm runs in linear time with respect to the number of nodes. We also give a simple proof that this algorithm does indeed calculate the UniFrac as formulated in equation 3.

▷ Claim 1. Algorithm 1 computes the UniFrac as formulated in equation 3.

Proof. Consider the matrix W in equation 3. Let L be a vector with the ith entry being $l(i)$ and $\overline{W}$ be the skeleton matrix of W such that $\overline{W}_{ij} = 1$ if $W_{ij} \neq 0$ and $\overline{W}_{ij} = 0$ otherwise. Also, for simplicity of comparison, let $M = P - Q$ as in the algorithm. With these notations, (1) can be rewritten as $||L \cdot (\overline{W}M)||_{L_1}$ ($\cdot$ denotes the dot product).

By the construction of $\overline{W}$, for a given row i, $\overline{W}_{ij} = 1$ if and only if $j = i$ or node j is an ancestor of i on the tree. It is then easy to observe that line 4-6 of Algorithm 1 computes $\overline{W}M$. The scaling of $\overline{W}M$ by taking the dot product with L, followed by computing the L_1 distance, is done in line 7. ◁

3 Results

3.1 On taxonomic data converted from phylogenetic data

To test the hypothesis that assigning branch lengths to a taxonomic tree allows computation of UniFrac that reflects beta-diversity using only WGS data, we begin with the most ideal scenario: one in which the taxonomic profile of the WGS data exactly reflects the phylogenetic profile of the 16S rRNA data. To this end we constructed the most ideal taxonomic profiles as follows: using the mapping file provided in the Greengenes database [12] that maps 16S OTUs with a known phylogenetic tree to their corresponding NCBI taxonomic IDs (taxIDs), we converted a phylogenetic sample to its taxonomic counterpart by simply changing the ID type while maintaining the relative abundance of each species. Using the lineage information associated with the taxID of each species in NCBI, we constructed the full taxonomic profile with the ranks of superkingdom, phylum, class, order, family, genus, and species, representing the taxonomic relations among the species.

Since UniFrac is frequently used to observe qualitative difference in samples when partitioned by certain metadata variables and viewed on a Principal Coordinates Analysis (PCoA) plot, we evaluated the performance of UniFrac computed on such a taxonomic profile based on the hypothesis that if the method makes biological sense, the clustering of samples in the WGS data should agree with that using 16S data. As such, we assessed the performance of WGSUniFrac by observing the clustering of samples under PCoA in comparison to that of their 16S counterparts, as well as quantitatively evaluated the clustering quality with commonly used clustering evaluation metrics.

To better observe the clusters, we created a simple model to mimic samples collected from two distinct environments with the aid of the given phylogenetic tree. To create samples from an environment, we first select a random leaf node on the phylogenetic tree and call it a pivot node. We then randomly selected a fixed number of nodes sufficiently close to the pivot node first selected. To create samples from the other environment, we select a second pivot node sufficiently far away from the first node chosen, and create samples in the same manner centering on the second pivot node. For simplicity of computation, when the distance between two leaf nodes was considered, instead of considering the actual distance in the sense of total branch lengths separating the two nodes, we considered the position of the second node in a list of all nodes ranked according to distance with respect to the first node. For instance, instead of considering "nodes within x units of branch length from node 1", we would consider "nodes among the y (for example, 500) nodes closest to node 1". Throughout this paper, we will call this aforementioned value y the "range" of an environment. The distance between the two pivot nodes is also defined in this manner, which we will call "dissimilarity" in this paper (refer to Figure S1). This proxy of replacing the actual distance by the relative position of a node in a list of ranked nodes may very likely result in nonlinearity in the relationship between clustering score and the range or dissimilarity setting, as well as greater variability among repeated experiments having identical range or dissimilarity setting. Nonetheless, it greatly simplifies the calculation and it should not affect the general trend that the greater the dissimilarity and the smaller the range, the more tightly clustered the samples would be on the given phylogenetic tree.

To respectively test the effect of range and dissimilarity on the quality of clustering, we first fixed the dissimilarity to be the maximum (35,461) and generated data across ranges 200, 500, 1,000, 5,000, 10,000, 15,000 and 20,000, and then generated data with dissimilarities 800, 900, 1,000, 5,000, 10,000, 20,000, 30,000 and maximum respectively for a fixed range of 500. We generated 100 replicates for each of these setups, each consisting of 25 samples for each environment, with 200 organisms approximately exponentially distributed in relative abundances in each sample. The quality of clustering for each replicate was assessed with the Silhouette Index [49].

In this experiment, the branches of the taxonomic tree were set to the reciprocal of the depth of the branch in the tree (i.e. 1/ distance from root node); we investigate other branch length specifications subsequently. Figure 2 shows the overall results of this experiment, with the trends demonstrated by the plots being expected and intuitive. Namely, the higher the dissimilarity, the greater the differences between samples from the two environments, resulting in more distinguishable clusterings (reflected in higher Silhouette scores). On the other hand, increasing range indirectly decreases dissimilarity by spreading out the clusters/environments, resulting in a decreasing trend of clustering quality. It is noteworthy that these trends were observed in both WGSUniFrac and 16S UniFrac with similar sensitivity. The same trend was observed when other clustering metrics are used (Figure S2). It is also interesting to note that it appears WGSUniFrac is less sensitive to changes in range compared to 16S UniFrac.

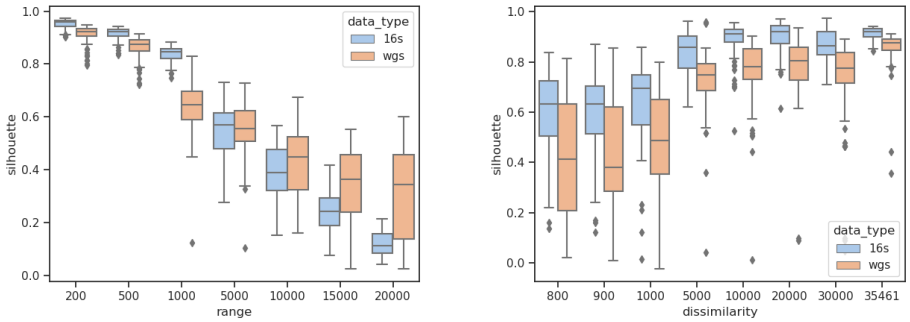


Figure 2 A comparison between the Silhouette scores computed using 16S data and WGS data under different settings of range and dissimilarity. Higher Silhouette score indicates better clustering. Left: Clustering quality of simulated 16S and WGS samples by environments given different within-sample diversity. X-axis (range) indicates the degree of phylogenetic diversity in each sample. Right: Clustering quality of simulated 16S and WGS samples by environments given different degrees of between-sample dissimilarity. X-axis (dissimilarity) indicates the degree of difference among the two simulated community.

3.2 Insensitivity to model or data derived branch length assignment

3.2.1 Model-based branch length assignment

Since the consensus on how branch lengths should be assigned, if it ever exists, has yet to be established, in this section we examine the impact of different branch lengths assignments on WGSUniFrac performance. We first investigated three major categories of branch lengths assignment with respect to the depth of the tree: increasing, constant, decreasing. To this end we defined a branch lengths function to compute the length of a branch located x nodes away from the root, denoted by $l(x)$, by $l(x) = x^k$ for some integer k. In other words, the only factor we take into consideration was the depth of the branch in the tree. We first compared the results by repeating the experiment in the previous section with k set to -1, 0, and 1, resulting in decreasing, constant, and increasing branch lengths respectively, when viewed from the root to the leaves.

From Figure 3, the branch length function x^{-1} yielded the best performance, followed by constant branch length assignment, while assigning branch lengths proportional to tree levels yielded the worst result. This is consistent with the observation that organismal similarity increases as one moves to lower taxonomic rank.

Upon establishing the general relationship between the branch lengths and the depths of the tree, we then examined how sensitive the performance is with respect to fine-tuning of k by setting k to be -2, -1.5 and -0.5 and repeating the procedure. The results are shown in Supplementary Figure S2, in which we observed an improvement of WGSUniFrac in comparison to the 16S UniFrac with respect to increasing magnitude of k (i.e. more negative). This improvement is much more drastic with respect to range than with respect to dissimilarity. In other words, the within-sample diversity is more sensitive to the fine-tuning of ratios between branch lengths. In terms of dissimilarity, which is an intuitive reflection of beta diversity, the improvement in comparison to 16S UniFrac is far less apparent, especially when dissimilarity is small. As such, we conjecture that the magnitude of k does not have a significant effect on detecting beta diversity, although it can be suggestive that WGSUniFrac may potentially be more robust than 16S UniFrac when within-sample diversity is large.

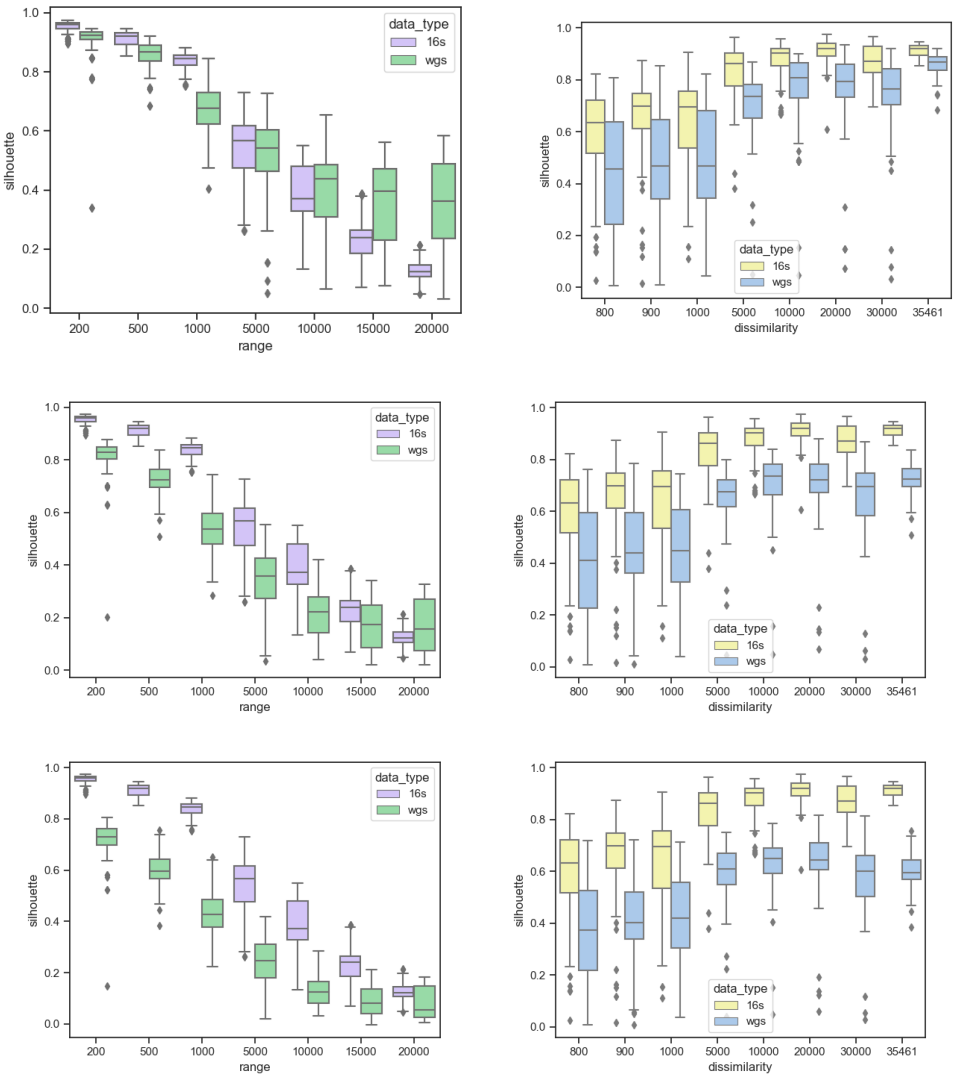


Figure 3 The effect of branch lengths choice. From top to bottom: $k = -1$ (decreasing branch lengths down the tree), $k = 0$ (uniform branch length), $k = 1$ (increasing branch lengths down the tree).

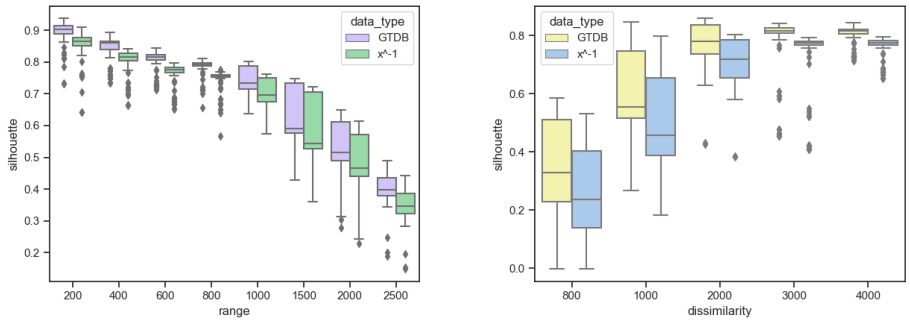


Figure 4 A comparison between the Silhouette scores computed using the GTDB tree and that using the transformed tree with branch lengths reassigned according to branch length function x^{-1}.

However, it should be noted that the clustering quality decreases if the value of k creates edge lengths on a taxonomic tree that deviates too much from what is biologically reasonable, as can be seen in Supplementary Figure S4.

For the subsequent experiments, we only considered the branch length function x^{-1} in all calculations unless otherwise stated and we revisit the effect of branch lengths selection in Section 3.4 below.

3.2.2 Branch lengths specified with data derived phylogeny-aware taxonomy

In this section, we further examine the robustness of WGSUniFrac with the aid of data obtained from the Genome Taxonomy Database (GTDB), a database providing taxonomic trees with topology and branch lengths based on protein phylogeny [45]. As a basis of comparison, we used the bac120 tree from GTDB, which is a tree with branch lengths reflecting the phylogenetic information as inferred from the concatenation of 120 marker genes [45].

To assess the impact of branch length specification, we first investigated the performance of WGSUniFrac when the actual branch lengths on the bac120 tree were replaced by the assignment according to the x^{-1} function, following the same experimental setup in Section 3.1. The results are shown in Figure 4.

From Figure 4, it can be noted that the behavior of UniFrac computed using the transformed tree closely mimics the original bac120 tree from GTDB, though slightly inferior in all cases. Though a different type of tree was used and different types of data were compared, the nature of this experiment was, in actuality, very similar to that in section 3.1. In both cases, we tested how robust UniFrac would remain when a phylogenetic tree of finely annotated branch lengths was replaced by one that only reflected a general trend instead of having finely labeled branches. The stories told in the two cases were also similar: phylogenetic information does add quality to UniFrac, though UniFrac still reflects general trends without it. In fact, a tree reflecting a general trend among the organisms is sufficient for UniFrac to offer decent insights into beta diversity.

We next investigated the effect of difference in taxonomic topology on UniFrac. According to the authors of GTDB, more than half of the genomes in GTDB had changes in their existing taxonomy [45], resulting in significant differences in the GTDB taxonomy and the existing NCBI taxonomy. As such, among around 4,979 organisms having both complete GTDB and

NCBI taxonomy, we selected 200 for each sample according to the protocol in section 2.1. For each sample, we generated taxonomic profiles according to GTDB taxonomy and NCBI taxonomy respectively, each having identical organisms and relative abundance distribution. For both taxonomies, we used the branch lengths function x^{-1}. Fixing dissimilarity to be 4000 nodes apart on the GTDB tree, we created samples with varying values of range, ranging from 200 where nodes from two environments were most tightly clustered, to 2,500 where the two environments were slightly overlapping. Similarly, to test the performance under different values of dissimilarity, we fixed the range to be 600 and generated samples having dissimilarities ranging from 800, where the two environments were relatively similar, to 4,000, where the two environments were highly distinct. Each of these setups was repeated 100 times. The results are shown in Figure 5.

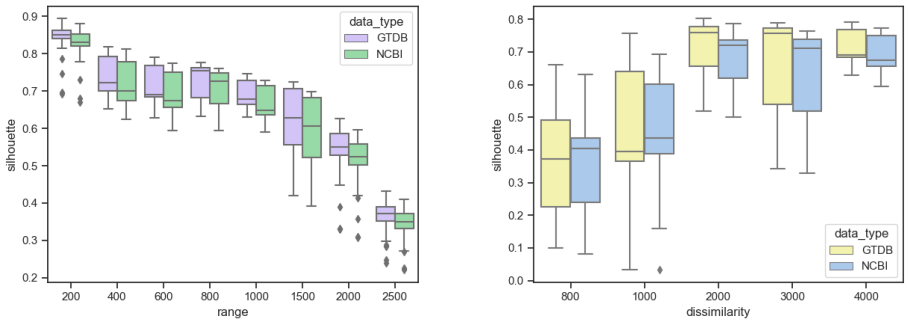


Figure 5 A comparison between the Silhouette scores computed using the GTDB taxonomy and NCBI taxonomy.

Even with differing underlying taxonomic tree topology, we observed highly similar behavior of UniFrac when using the GTDB taxonomy and when using the NCBI taxonomy. In some cases, specifically when dissimilarity was relatively small, the NCBI taxonomy appeared to yield slightly better performance when WGSUniFrac was applied. In most other cases, GTDB taxonomy yielded slightly better overall results, which agreed with previous experiments where 16S data yielded better overall results. This is due to the GTDB taxonomy being more consistent with 16S-derived taxonomy compared to the NCBI taxonomy. Nonetheless, the similarity in performance between the approaches using the GTDB taxonomy and the NCBI taxonomy, together with the previous experiment, suggest that neither the granularity of the branch lengths nor the taxonomic topology is a significant limiting factor to the application of UniFrac, supporting our hypothesis.

3.3 On simulated reads

In the previous section, it has been demonstrated that WGSUniFrac is able to cluster samples according to environments in the most ideal situation in which both the 16S OTU tables and WGS profiles were created without the consideration of sequencing errors and profiling biases, which are common in real-world applications. In addition, different profiling methods and taxonomic classification methods may produce different results both between 16S and WGS data and within the same data type [39, 50, 25].

To answer the question if WGSUniFrac would remain robust under a more realistic setting, in this section we investigate the performance of WGSUniFrac on profiles produced from simulated reads. We also increased the complexity of the experimental setup by testing not only with two environments but also with five.

We used Grinder [4] to simulate both 16S amplicon reads and WGS reads with sequencing protocols similar to those of common modern-day sequencing platforms as much as possible while maintaining computation efficiency (see Supplementary Experimental setup details). We used the built-in Dada2 [7] plugin in QIIME [8] to infer taxonomic feature tables from 16S amplicon reads and mOTUs [40] to generate taxonomic profiles from the simulated WGS reads. We then calculated and compared UniFrac and WGSUniFrac respectively on the results.

Following a similar approach as section 3.1, the following setups were conducted twice, one using two environments and the other using five: Fixing the range to be 500, we generated experiments having dissimilarities 1,000 to 6,000 in steps of 1,000; fixing dissimilarity to be 4,000, generate experiments with range 200, 1,000, 2,000, 3,000. Each of these combinations was repeated five times with organisms chosen at random. The results are summarized in Table 1 and Figure 6.

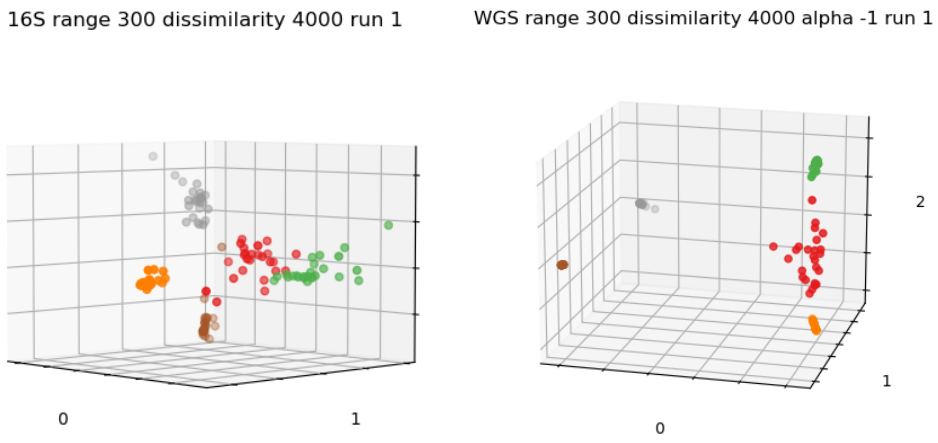


Figure 6 An instance of the comparison between PCoA plots produced using 16S and WGS data with range 300 and dissimilarity 4,000, colors depicting environments. Left: 16S UniFrac. Right: WGSUniFrac.

Table 1 Mean Silhouette Indices for 16S and WGS clusterings by pairwise UniFrac. Higher Silhouette index indicates better clustering of environments.

	2 Environments	5 Environments
16S	0.226	0.051
WGS	0.562	0.206

It was somewhat surprising that the mean Silhouette scores significantly favored the WGS approach in contrast to the 16S approach, which was expected to have better performance. This could be due to the intrinsic differences in simulation protocols and tools used. It has also been pointed out that abundance profiling has much better accuracy when WGS data is used compared to when 16S data is used [25]. This might potentially explain the poor performance of 16S data when inferring of abundances from reads was involved, which also shows the limitation of 16S data and motivates our endeavor to explore a good metric that can be applied to WGS data. Still, an average score of 0.562 allowed us to believe that UniFrac can be applied to WGS data even in the presence of sequencing errors and noises.

3.4 On real WGS studies

While running the experiment on simulated reads allowed a glimpse of the feasibility and performance of WGSUniFrac in a more realistic setting, the real-world situation is still much more complex. For instance, the organisms involved in the previous experiments all come from one single phylogenetic tree [36, 12]. In each experiment setting, organisms were selected to simulate distinct environments, with each sample consisting of the exact same number of organisms with relative abundances distributed over a near-ideal exponential distribution. Also, in order to have a fair comparison with 16S UniFrac, combined with limitations of tools in read simulation and profiling processes, compromises such as limiting read lengths were made, further impacting the resemblance between the simulated data and potential real world data.

As such, we proceeded to test WGSUniFrac on real world studies using human whole genome shotgun data. It has been observed and reported in various 16S studies that metagenomic samples collected at different body sites of a human significantly differ [20, 26, 11]. We investigated if this property could be captured using WGS data alone by investigating if samples can be clustered depending on the site of collection.

Using the HumanMetagenomeDB database [23], a curated database for human WGS metagenomic data, we searched for metagenomic projects with specified body sites. To minimize the effect of differences in sampling and sequencing protocols in different studies, we limited our search to studies originating from the Sequence Read Archive (SRA), sequenced using ILLUMINA, and with number of sequences 10 million and above. Among these, we considered only paired-end data and applied the same quality control to all samples prior to profiling to maintain consistency across samples as much as possible (See Supplementary Materials section: Experimental Setup Details). The samples were then converted to taxonomic profiles using mOTUs [40]. Among these profiles, we removed those containing less than 100 species. The resulting PCoA plots are shown below. To eliminate the potential bias that the samples might be clustering by studies instead of by body sites, as most studies involved one single body site each, we also produced the PCoA plot colored according to project ID for each category as a comparison.

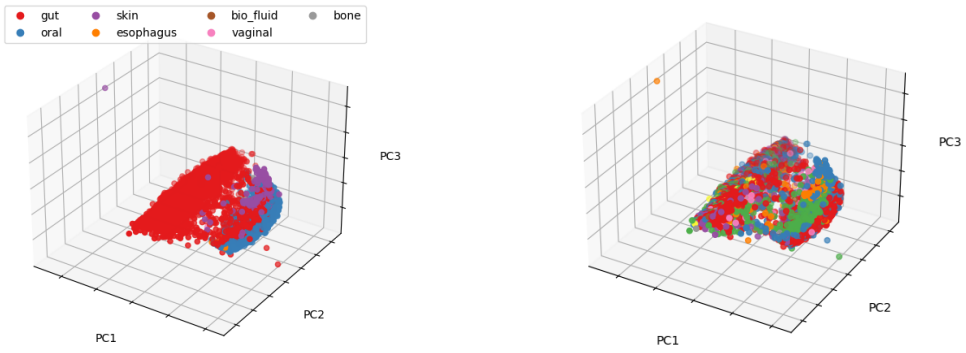


Figure 7 Left: samples colored by body sites. Right: samples colored by study IDs.

From Figure 7, we can see that samples were clustered with reasonable sensitivity according to body sites rather than by study, despite the varying protocols across studies, a demonstration of the robustness of WGSUniFrac in real-world applications.

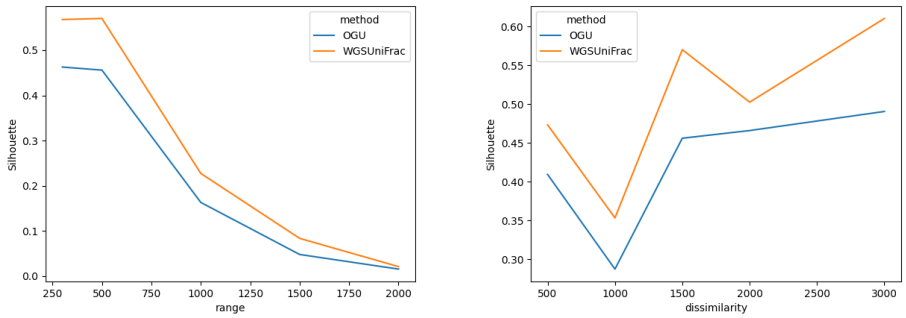


Figure 8 A comparison between the average Silhouette scores computed using OGU method and WGSUniFrac method under different settings of range and dissimilarity. Higher Silhouette score indicates better clustering.

At this point we revisit the open problem of branch lengths function selection in section 1, using these real data. Since the number of data points were massive, for the ease of observing patterns, we stratified the profiles into three categories and analyse them separately: low diversity (containing 100 to 200 species), medium diversity (200 to 300 species), and high diversity (300 species and above). For each of these categories, we produced PCoA plots using branch lengths functions x^{-1} and x^{-2} respectively. The results are shown in Supplementary Figure S3.

A careful examination of the plots shows that changing the k value from -1 to -2 in the branch lengths function x^k only resulted in scaling of the clusters. Specifically, it only clustered more tightly what had already been clustered and revealed no additional information. Hence, there is no strong reason that -2 should be favored over -1. The user could potentially decide on the magnitude of k depending on the alpha diversity of the samples, if this information is known.

3.5 Comparison with the OGU method

In this section, we compare the performance of our WGSUniFrac method with the recently published OGU method [57], which provides an alternative way for similarity metrics such as UniFrac to be computed on WGS data by defining the operational genomic unit (OGU). The fundamental difference between our method and the OGU method is that the OGU method is not taxonomic-based while WGSUniFrac is.

We followed a similar protocol as that in section 3.3, simulating reads from two environments using a randomly selected subset of 3,000 species of the Web of Life (WoL) database [58] as reference genomes. The distance matrices for OGU method were produced following the woltka workflow suggested by the authors [2]. The clustering quality using these matrices were compared with that using our method. Each experimental setup was repeated five times and the average Silhouette score was computed for each experimental setup. The results are shown in Figure 8.

From Figure 8, the clustering quality of WGSUniFrac method exceeds that of the OGU method in every setting, demonstrating the robustness of the WGSUniFrac method and the value of the presence of taxonomic structure. Further, unlike the OGU method that requires the presence of a phylogenetic tree, such as the Web of Life tree [58] in this case, WGSUniFrac can be applied with only the taxonomic profiles, which can be easily obtained

directly from WGS reads using a profiler, giving WGSUniFrac more flexibility and adaptivity. This simplicity of the workflow also gives WGSUniFrac computational advantage, making it much more efficient and straightforward than the current OGU workflow [2].

4 Discussions

Up to this point, we have tested the performance of WGSUniFrac in comparison to the traditional UniFrac metric applied to 16S data under various settings, ranging from the most ideal scenario to real-world data. Under the most ideal scenario, where samples with a phylogenetic classification were directly compared to the corresponding taxonomic classification, WGSUniFrac exhibited comparable ability to distinguish samples from different environments under various parameter settings, providing evidence for the hypothesis that UniFrac can be applied to WGS data simply by assigning branch lengths to a taxonomic tree without significant loss of information on beta-diversity. We then further investigated the effects of different branch length assignments and reached the conclusion that having branch lengths inversely proportional to the height of the taxonomic tree best capitulated the expected clustering trend, while fine-tuning of the magnitude of this proportion did not seem to reveal additional information.

A more detailed investigation of the effect of differences in branch lengths assignments was conducted using the GTDB data, with which we investigated the effect of phylogenetic information both in terms of branch lengths and topology. The results showed that neither the decrease in the resolution of branch lengths nor the change of topology from that of GTDB taxonomy to the conventional NCBI taxonomy significantly decreased the quality of clustering.

The results were slightly puzzling when read simulation was involved in the second part of the experiments, with WGSUniFrac outperforming 16S UniFrac in most cases. We conjecture that this was due to the limitation of simulation and profiling tools and the intrinsic differences in data preparation protocols between 16S and WGS data. The poor performance of 16S UniFrac when sequencing errors were involved demonstrated the potential superiority of WGSUniFrac in real applications. However, further studies are needed to confirm this conjecture. The limitation of efficient read simulation tools that simulate both 16S rRNA and WGS data impeded our further investigation into this matter.

It was perhaps most interesting to evaluate the performance of WGSUniFrac on real data. To this end we tested the ability of WGSUniFrac in recapitulating a known phenomenon previously demonstrated by UniFrac applied on 16S data. Though the lack of corresponding 16S counterparts made a direct comparison to 16S UniFrac impractical, the PCoA plots did clearly demonstrated the ability of WGSUniFrac in clustering metagenomic samples according to body sites, confirming also in this process that the the differences among samples from different body sites are more prominent than the differences of the same body sites across individuals.

It is also noteworthy that except the experiment in section 3.4 where observations were made purely on WGS data without a quantitative or qualitative "ground truth" to compare to, most of the experiments used 16S data as a reference of comparison. However, this was simply because the UniFrac metric was originally designed to be used data with phylogenetic information, which was typically available when 16S data is employed, not necessarily that the 16S phylogeny is indeed the gold standard. In fact, limitations of 16S data in taxonomic classification have been reported in previous studies [48, 25] which undermines the use of 16S as the standard reference. In addition, such as in the case of GTDB, there have been methods

capable of producing phylogenetically consistent taxonomy, and has been shown in the experiments above to yield better results than taxonomy without the additional phylogenetic information. This shows that WGSUniFrac will likely prove itself to be increasingly useful as better methods to uncover the "real" taxonomic classification in WGS data emerge.

5 Conclusion

In this paper, we provided an algorithm for UniFrac to be computed directly on WGS data by assigning branch lengths to taxonomic profiles. Though branch lengths assignments remain an open area of exploration, the insensitivity of the performance of WGSUniFrac to branch lengths assignments demonstrated in our experiments is a strong advocate of the potential of WGSUniFrac. Overall, our study demonstrated that UniFrac can be freed from requiring phylogenetic trees and can find its application in a much wider range of data.

References

1 Cami-challenge. https://github.com/CAMI-challenge/contest_information/blob/master/file_formats/CAMI_TP_specification.mkd, 2015.

2 woltka. https://github.com/qiyunzhu/woltka/blob/master/doc/ogu.md, commit = 7ef8318, 2022.

3 Johanne Ahrenfeldt, Carina Skaarup, Henrik Hasman, Anders Gorm Pedersen, Frank Møller Aarestrup, and Ole Lund. Bacterial whole genome-based phylogeny: construction of a new benchmarking dataset and assessment of some existing methods. *BMC Genomics*, 18(1):19, 2017. doi:10.1186/s12864-016-3407-6.

4 Florent E. Angly, Dana Willner, Forest Rohwer, Philip Hugenholtz, and Gene W. Tyson. Grinder: a versatile amplicon and shotgun sequence simulator. *Nucleic Acids Research*, 40(12):e94–e94, 2012. doi:10.1093/nar/gks251.

5 Francois Balloux, Ola Brønstad Brynildsrud, Lucy van Dorp, Liam P. Shaw, Hongbin Chen, Kathryn A. Harris, Hui Wang, and Vegard Eldholm. From theory to practice: Translating whole-genome sequencing (wgs) into the clinic. *Trends in Microbiology*, 26(12):1035–1048, 2018. doi:10.1016/j.tim.2018.08.004.

6 Sébastien Boutin, Simon Y. Graeber, Michael Weitnauer, Jessica Panitz, Mirjam Stahl, Diana Clausznitzer, Lars Kaderali, Gisli Einarsson, Michael M. Tunney, J. Stuart Elborn, Marcus A. Mall, and Alexander H. Dalpke. Comparison of microbiomes from different niches of upper and lower airways in children and adolescents with cystic fibrosis. *PLoS ONE*, 10(1):e0116029, 2015. doi:10.1371/journal.pone.0116029.

7 Benjamin J Callahan, Paul J McMurdie, Michael J Rosen, Andrew W Han, Amy Jo A Johnson, and Susan P Holmes. Dada2: High-resolution sample inference from illumina amplicon data. *Nature Methods*, 13(7):581–583, 2016. doi:10.1038/nmeth.3869.

8 J Gregory Caporaso, Justin Kuczynski, Jesse Stombaugh, Kyle Bittinger, Frederic D Bushman, Elizabeth K Costello, Noah Fierer, Antonio Gonzalez Peña, Julia K Goodrich, Jeffrey I Gordon, and et al. Qiime allows analysis of high-throughput community sequencing data. *Nature Methods*, 7(5):335–336, 2010. qiime citation. doi:10.1038/nmeth.f.303.

9 Alexander L. Carlson, Kai Xia, M. Andrea Azcarate-Peril, Samuel P. Rosin, Jason P. Fine, Wancen Mu, Jared B. Zopp, Mary C. Kimmel, Martin A. Styner, Amanda L. Thompson, Cathi B. Propper, and Rebecca C. Knickmeyer. Infant gut microbiome composition is associated with non-social fear behavior in a pilot study. *Nature Communications*, 12(1):3294, 2021. doi:10.1038/s41467-021-23281-y.

10 Shifu Chen, Yanqing Zhou, Yaru Chen, and Jia Gu. fastp: an ultra-fast all-in-one fastq preprocessor. *Bioinformatics*, 34(17):i884–i890, 2018. doi:10.1093/bioinformatics/bty560.

11 Elizabeth K. Costello, Erica M. Carlisle, Elisabeth M. Bik, Michael J. Morowitz, and David A. Relman. Microbiome assembly across multiple body sites in low-birthweight infants. *mBio*, 4(6):e00782–13, 2013. doi:10.1128/mbio.00782-13.

12 T. Z. DeSantis, P. Hugenholtz, N. Larsen, M. Rojas, E. L. Brodie, K. Keller, T. Huber, D. Dalevi, P. Hu, and G. L. Andersen. Greengenes, a chimera-checked 16s rrna gene database and workbench compatible with arb. *Applied and Environmental Microbiology*, 72(7):5069–5072, 2006. doi:10.1128/aem.03006-05.

13 Young-Gyu Eun, Jung-Woo Lee, Seung Woo Kim, Dong-Wook Hyun, Jin-Woo Bae, and Young Chan Lee. Oral microbiome associated with lymph node metastasis in oral squamous cell carcinoma. *Scientific Reports*, 11(1):23176, 2021. doi:10.1038/s41598-021-02638-9.

14 Steven N. Evans and Frederick A. Matsen. The phylogenetic kantorovich–rubinstein metric for environmental sequence samples. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 74(3):569–592, 2012. doi:10.1111/j.1467-9868.2011.01018.x.

15 Stéphane Guindon, Jean-François Dufayard, Vincent Lefort, Maria Anisimova, Wim Hordijk, and Olivier Gascuel. New algorithms and methods to estimate maximum-likelihood phylogenies: Assessing the performance of phyml 3.0. *Systematic Biology*, 59(3):307–321, 2010. doi:10.1093/sysbio/syq010.

16 Micah Hamady, Catherine Lozupone, and Rob Knight. Fast unifrac: facilitating high-throughput phylogenetic analyses of microbial communities including analysis of pyrosequencing and phylochip data. *The ISME journal*, 4(1):17–27, 2010.

17 Micah Hamady, Catherine Lozupone, and Rob Knight. Fast unifrac: facilitating high-throughput phylogenetic analyses of microbial communities including analysis of pyrosequencing and phylochip data. *The ISME Journal*, 4(1):17–27, 2010. doi:10.1038/ismej.2009.97.

18 Jaime Huerta-Cepas, François Serra, and Peer Bork. Ete 3: Reconstruction, analysis, and visualization of phylogenomic data. *Molecular Biology and Evolution*, 33(6):1635–1638, 2016. ete3 package. doi:10.1093/molbev/msw046.

19 Luisa W. Hugerth and Anders F. Andersson. Analysing microbial community composition through amplicon sequencing: From sampling to hypothesis testing. *Frontiers in Microbiology*, 8:1561, 2017. doi:10.3389/fmicb.2017.01561.

20 Curtis Huttenhower, Dirk Gevers, Rob Knight, Sahar Abubucker, Jonathan H. Badger, Asif T. Chinwalla, Heather H. Creasy, Ashlee M. Earl, Michael G. FitzGerald, Robert S. Fulton, and et al. Structure, function and diversity of the healthy human microbiome. *Nature*, 486(7402):207–214, 2012. doi:10.1038/nature11234.

21 Stefan Janssen, Daniel McDonald, Antonio Gonzalez, Jose A. Navas-Molina, Lingjing Jiang, Zhenjiang Zech Xu, Kevin Winker, Deborah M. Kado, Eric Orwoll, Mark Manary, and et al. Phylogenetic placement of exact amplicon sequences improves associations with clinical information. *mSystems*, 3(3):e00021–18, 2018. doi:10.1128/msystems.00021-18.

22 Jonathan Kans. Entrez direct: E-utilities on the unix command line - entrez programming utilities help - ncbi bookshelf, April 2013. URL: https://www.ncbi.nlm.nih.gov/books/NBK179288/.

23 Jonas Coelho Kasmanas, Alexander Bartholomäus, Felipe Borim Corrêa, Tamara Tal, Nico Jehmlich, Gunda Herberth, Martin von Bergen, Peter F Stadler, André Carlos Ponce de Leon Ferreira de Carvalho, and Ulisses Nunes da Rocha. Humanmetagenomedb: a public repository of curated and standardized metadata for human metagenomes. *Nucleic Acids Research*, 49(D1):gkaa1031–, 2020. doi:10.1093/nar/gkaa1031.

24 C. J. Keylock. Simpson diversity and the shannon–wiener index as special cases of a generalized entropy. *Oikos*, 109(1):203–207, 2005. doi:10.1111/j.0030-1299.2005.13735.x.

25 Lusine Khachatryan, Rick H. de Leeuw, Margriet E.M. Kraakman, Nikos Pappas, Marije te Raa, Hailiang Mei, Peter de Knijff, and Jeroen F.J. Laros. Taxonomic classification and abundance estimation using 16s and wgs – A comparison using controlled reference samples. *Forensic Science International: Genetics*, 46:102257, 2020. doi:10.1016/j.fsigen.2020.102257.

26 Omry Koren, Aymé Spor, Jenny Felin, Frida Fåk, Jesse Stombaugh, Valentina Tremaroli, Carl Johan Behre, Rob Knight, Björn Fagerberg, Ruth E. Ley, and et al. Human oral, gut,

and plaque microbiota in patients with atherosclerosis. *Proceedings of the National Academy of Sciences*, 108:4592–4598, 2011. doi:10.1073/pnas.1011383107.

27 David Koslicki and Daniel Falush. Metapalette: A k-mer painting approach for metagenomic taxonomic profiling and quantification of novel strain variation. *bioRxiv*, page 039909, 2016. doi:10.1101/039909.

28 Chao Liang, Han-Chi Tseng, Hui-Mei Chen, Wei-Chi Wang, Chih-Min Chiu, Jen-Yun Chang, Kuan-Yi Lu, Shun-Long Weng, Tzu-Hao Chang, Chao-Hsiang Chang, Chen-Tsung Weng, Hwei-Ming Wang, and Hsien-Da Huang. Diversity and enterotype in gut bacterial community of adults in taiwan. *BMC Genomics*, 18(Suppl 1):932, 2017. doi:10.1186/s12864-016-3261-6.

29 Kevin Liu, Sindhu Raghavan, Serita Nelesen, C. Randal Linder, and Tandy Warnow. Rapid and accurate large-scale coestimation of sequence alignments and phylogenetic trees. *Science*, 324(5934):1561–1564, 2009. doi:10.1126/science.1171243.

30 Catherine Lozupone, Micah Hamady, and Rob Knight. Unifrac–an online tool for comparing microbial community diversity in a phylogenetic context. *BMC bioinformatics*, 7(1):1–14, 2006.

31 Catherine Lozupone and Rob Knight. Unifrac: a new phylogenetic method for comparing microbial communities. *Applied and Environmental Microbiology*, 71(12):8228–8235, 2005. doi:10.1128/aem.71.12.8228-8235.2005.

32 Catherine Lozupone, Manuel E Lladser, Dan Knights, Jesse Stombaugh, and Rob Knight. Unifrac: an effective distance metric for microbial community comparison. *The ISME Journal*, 5(2):169–172, 2011. doi:10.1038/ismej.2010.133.

33 Catherine A. Lozupone, Micah Hamady, Scott T. Kelley, and Rob Knight. Quantitative and qualitative β diversity measures lead to different insights into factors that structure microbial communities. *Applied and Environmental Microbiology*, 73(5):1576–1585, 2007. doi:10.1128/aem.01996-06.

34 Jason McClelland. Wasserstein β-diversity metrics over graphs: Derivation, efficient computation and application, 2018.

35 Jason McClelland and David Koslicki. Emdunifrac: exact linear time computation of the unifrac metric and identification of differentially abundant organisms. *Journal of Mathematical Biology*, 77(4):935–949, 2018. doi:10.1007/s00285-018-1235-9.

36 Daniel McDonald, Morgan N Price, Julia Goodrich, Eric P Nawrocki, Todd Z DeSantis, Alexander Probst, Gary L Andersen, Rob Knight, and Philip Hugenholtz. An improved greengenes taxonomy with explicit ranks for ecological and evolutionary analyses of bacteria and archaea. *The ISME Journal*, 6(3):610–618, 2012. doi:10.1038/ismej.2011.139.

37 Daniel McDonald, Yoshiki Vázquez-Baeza, David Koslicki, Jason McClelland, Nicolai Reeve, Zhenjiang Xu, Antonio Gonzalez, and Rob Knight. Striped unifrac: enabling microbiome analysis at unprecedented scale. *Nature methods*, 15(11):847–848, 2018.

38 Daniel McDonald, Yoshiki Vázquez-Baeza, David Koslicki, Jason McClelland, Nicolai Reeve, Zhenjiang Xu, Antonio Gonzalez, and Rob Knight. Striped unifrac: enabling microbiome analysis at unprecedented scale. *Nature Methods*, 15(11):847–848, 2018. doi:10.1038/s41592-018-0187-8.

39 F. Meyer, A. Fritz, Z.-L. Deng, D. Koslicki, A. Gurevich, G. Robertson, M. Alser, D. Antipov, F. Beghini, D. Bertrand, and et al. Critical assessment of metagenome interpretation - the second round of challenges. *bioRxiv*, page 2021.07.12.451567, 2021. doi:10.1101/2021.07.12.451567.

40 Alessio Milanese, Daniel R Mende, Lucas Paoli, Guillem Salazar, Hans-Joachim Ruscheweyh, Miguelangel Cuenca, Pascal Hingamp, Renato Alves, Paul I Costea, Luis Pedro Coelho, and et al. Microbial abundance, activity and population genomic profiling with motus2. *Nature Communications*, 10(1):1014, 2019. doi:10.1038/s41467-019-08844-4.

41 Vanessa Moura, Iris Ribeiro, Priscilla Moriggi, Artur Capão, Carolina Salles, Suleima Bitati, and Luciano Procópio. The influence of surface microbial diversity and succession on

microbiologically influenced corrosion of steel in a simulated marine environment. *Archives of Microbiology*, 200(10):1447–1456, 2018. doi:10.1007/s00203-018-1559-2.

42 Nam-phuong Nguyen, Siavash Mirarab, Bo Liu, Mihai Pop, and Tandy Warnow. Tipp: taxonomic identification and phylogenetic profiling. *Bioinformatics*, 30(24):3548–3555, 2014. doi:10.1093/bioinformatics/btu721.

43 Brian D. Ondov, Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren, and Adam M. Phillippy. Mash: fast genome and metagenome distance estimation using minhash. *Genome Biology*, 17(1):132, 2016. doi:10.1186/s13059-016-0997-x.

44 Donovan H. Parks, Maria Chuvochina, Pierre-Alain Chaumeil, Christian Rinke, Aaron J. Mussig, and Philip Hugenholtz. A complete domain-to-species taxonomy for bacteria and archaea. *Nature Biotechnology*, 38(9):1079–1086, 2020. doi:10.1038/s41587-020-0501-8.

45 Donovan H Parks, Maria Chuvochina, David W Waite, Christian Rinke, Adam Skarshewski, Pierre-Alain Chaumeil, and Philip Hugenholtz. A standardized bacterial taxonomy based on genome phylogeny substantially revises the tree of life. *Nature Biotechnology*, 36(10):996–1004, 2018. doi:10.1038/nbt.4229.

46 N. Tessa Pierce, Luiz Irber, Taylor Reiter, Phillip Brooks, and C. Titus Brown. Large-scale sequence comparisons with sourmash. *F1000Research*, 8:1006, 2019. doi:10.12688/f1000research.19675.1.

47 Rachel Poretsky, Luis M. Rodriguez-R, Chengwei Luo, Despina Tsementzi, and Konstantinos T. Konstantinidis. Strengths and limitations of 16s rrna gene amplicon sequencing in revealing temporal microbial community dynamics. *PLoS ONE*, 9(4):e93827, 2014. doi:10.1371/journal.pone.0093827.

48 Ravi Ranjan, Asha Rani, Ahmed Metwally, Halvor S. McGee, and David L. Perkins. Analysis of the microbiome: Advantages of whole genome shotgun versus 16s amplicon sequencing. *Biochemical and Biophysical Research Communications*, 469(4):967–977, 2016. doi:10.1016/j.bbrc.2015.12.083.

49 Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987. doi:10.1016/0377-0427(87)90125-7.

50 Alexander Sczyrba, Peter Hofmann, Peter Belmann, David Koslicki, Stefan Janssen, Johannes Dröge, Ivan Gregor, Stephan Majda, Jessika Fiedler, Eik Dahms, and et al. Critical assessment of metagenome interpretation—a benchmark of metagenomics software. *Nature Methods*, 14(11):1063–1071, 2017. doi:10.1038/nmeth.4458.

51 Nicola Segata, Levi Waldron, Annalisa Ballarini, Vagheesh Narasimhan, Olivier Jousson, and Curtis Huttenhower. Metagenomic microbial community profiling using unique clade-specific marker genes. *Nature Methods*, 9(8):811–814, 2012. doi:10.1038/nmeth.2066.

52 Wei Shen and Hong Ren. Taxonkit: A practical and efficient ncbi taxonomy toolkit. *Journal of Genetics and Genomics*, 48(9):844–850, 2021. doi:10.1016/j.jgg.2021.03.006.

53 Nathan G. Swenson. Phylogenetic beta diversity metrics, trait evolution and inferring the functional beta diversity of communities. *PLoS ONE*, 6(6):e21264, 2011. doi:10.1371/journal.pone.0021264.

54 Marie Touchon, Claire Hoede, Olivier Tenaillon, Valérie Barbe, Simon Baeriswyl, Philippe Bidet, Edouard Bingen, Stéphane Bonacorsi, Christiane Bouchier, Odile Bouvet, and et al. Organised genome dynamics in the escherichia coli species results in highly diverse adaptive paths. *PLoS Genetics*, 5(1):e1000344, 2009. doi:10.1371/journal.pgen.1000344.

55 Gary D. Wu, Jun Chen, Christian Hoffmann, Kyle Bittinger, Ying-Yu Chen, Sue A. Keilbaugh, Meenakshi Bewtra, Dan Knights, William A. Walters, Rob Knight, and et al. Linking long-term dietary patterns with gut microbial enterotypes. *Science*, 334(6052):105–108, 2011. doi:10.1126/science.1208344.

56 Alexandra Zhernakova, Alexander Kurilshikov, Marc Jan Bonder, Ettje F. Tigchelaar, Melanie Schirmer, Tommi Vatanen, Zlatan Mujagic, Arnau Vich Vila, Gwen Falony, Sara Vieira-

Silva, and et al. Population-based metagenomics analysis reveals markers for gut microbiome composition and diversity. *Science*, 352(6285):565–569, 2016. doi:10.1126/science.aad3369.

57 Qiyun Zhu, Shi Huang, Antonio Gonzalez, Imran McGrath, Daniel McDonald, Niina Haiminen, George Armstrong, Yoshiki Vázquez-Baeza, Julian Yu, Justin Kuczynski, Gregory D. Sepich-Poore, Austin D. Swafford, Promi Das, Justin P. Shaffer, Franck Lejzerowicz, Pedro Belda-Ferre, Aki S. Havulinna, Guillaume Méric, Teemu Niiranen, Leo Lahti, Veikko Salomaa, Ho-Cheol Kim, Mohit Jain, Michael Inouye, Jack A. Gilbert, and Rob Knight. Phylogeny-aware analysis of metagenome community ecology based on matched reference genomes while bypassing taxonomy. *mSystems*, pages e00167–22, 2022. doi:10.1128/msystems.00167-22.

58 Qiyun Zhu, Uyen Mai, Wayne Pfeiffer, Stefan Janssen, Francesco Asnicar, Jon G. Sanders, Pedro Belda-Ferre, Gabriel A. Al-Ghalith, Evguenia Kopylova, Daniel McDonald, Tomasz Kosciolek, John B. Yin, Shi Huang, Nimaichand Salam, Jian-Yu Jiao, Zijun Wu, Zhenjiang Z. Xu, Kalen Cantrell, Yimeng Yang, Erfan Sayyari, Maryam Rabiee, James T. Morton, Sheila Podell, Dan Knights, Wen-Jun Li, Curtis Huttenhower, Nicola Segata, Larry Smarr, Siavash Mirarab, and Rob Knight. Phylogenomics of 10,575 genomes reveals evolutionary proximity between domains bacteria and archaea. *Nature Communications*, 10(1):5477, 2019. doi:10.1038/s41467-019-13443-4.

A Appendix

A.1 Experimental setup details

All computations of UniFrac of 16S data were done using the "beta-phylogenetics" function in `Qiime2` [8]. All profiling of WGS reads into profiles were performed using `mOTUs2` [40] with the parameter "precision."

A.1.1 On taxonomic data converted from phylogenetic data

We used the `99_otus` dataset from the `gg_13_5_otus` data package downloaded from Greengenes database [12]. We converted the phylogenetic tree into its corresponding taxonomic tree using the mapping file provided in the `ete3` python package [18] that maps OTUs to taxonomic IDs and the taxonomic lineage provided in NCBI Taxonomy database. We considered only OTUs in the `99_otus` phylogenetic tree that map to taxonomic IDs with a complete lineage of the ranks superkingdom, kingdom, phylum, class, order, family, genus, and species as can be retrieved from the NCBI Taxonomy database. There are 35,461 such OTUs in total. With respect to each of these OTUs, we computed its phylogenetic distance on the tree (using the "get_distance" method in the `ete3` module) from all the other OTUs and obtained a list of OTUs ranked by proximity.

A.1.2 Comparison with phylogeny-aware taxonomy

The GTDB data were obtained from `https://data.gtdb.ecogenomic.org` with release 202. We obtained the `bac120` taxonomic tree together with the corresponding taxonomy. The taxonomic ID for each of the organism in the `bac120` taxonomy file was retrieved using `TaxonKit` [52]. Species without a matching taxonomic ID in any part of the lineage were removed. There were approximately 4,900 species remaining after this process. The general approach for this part of the experiment is highly similar to that of the section above, with the original GTDB tree playing the role of the phylogenetic tree. There were 100 repeats for each combination of range and dissimilarity shown.

For the first part of the experiment, we selected species from the `bac120` taxonomic tree according to the protocols above and treated these samples as 16S samples, computing pairwise UniFrac distance matrix using `Qiime2` [8]. For each sample, its corresponding

taxonomic profile was generated, following the GTDB taxonomy as provided in the taxonomy file obtained from the database. The UniFrac distance matrix for each sample was computed using our method with the branch length function $l(x) = x^{-1}$, where x is the depth of the tree a branch belongs to, counted from the root.

For the second part of the experiment, the profiles using GTDB taxonomy were used as a reference. For each of these profiles, the species were singled out and for each species, the taxonomic path was reconstructed by retrieving the lineage from NCBI using the `ete3` python package [18], thus creating a second set of profiles differing from the first set only in taxonomic path. The UniFrac matrices of these GTDB profiles and NCBI profiles were compared, using the same branch length function of $l(x) = x^{-1}$, such that the differences in the results were solely accountable by the difference in taxonomy and nothing else.

A.1.3 On simulated reads

To evaluate the applicability of UniFrac on more realistic data, we tested our method on simulated reads. Both simulations of 16S amplicon libraries and of WGS libraries were done using `Grinder` [4]. For the 16S part, we used the reference genomes `99_otus.fasta` provided in the same `gg_13_5_otus` package from Greengenes as the first part of the experiment. With the aid of the mapping file provided that maps OTUs to NCBI accessions, we used the `esearch` and `efetch` functions in `Entrez Direct` [22] to extract the whole genome of each organism present in the 16S reference genomes, if it existed. To simulate amplicon sequencing reads, we use the forward primer sequence `AAACTYAAAKGAATTGRCG` as suggested by `Grinder`. Both the amplicon sequencing and WGS sequencing were single-end, with read length 150bp, 4th degree polynomial error model parameters suggested by `Grinder` and the default 80:20 substitution:indel error ratio, 5× coverage for 16S reads and a total read number of 1,000,000 for WGS reads.

The resulting 16S libraries were denoised using `Qiime2` plugin `dada2` [7], with phylogenetic tree built using `Qiime2` plugin fragment-insertion `SEPP` method [21], and finally converted to pairwise UniFrac distance matrix. The WGS libraries were profiled using `mOTUs` [40] into CAMI format [1] profiles, from which the pairwise UniFrac matrix was computed for each experiment.

Using the same protocol in "environment" creation as the first part of the study described above, with the restriction to only organisms with an WGS reference sequence available on NCBI (around 6000 in total), we simulated either two or five environments for each experiment with varying combinations of range and dissimilarity. Each experiment was repeated five times.

A.1.4 On real-world studies

We used the HumanMetagenomeDB [23] to filter and select human whole genome shotgun SRA data from nine body parts with number of sequences within 10 to 437 million (the maximum number in the HumanMetagenomeDB database) and sequenced using `Illumina`, which came out to be 12,261 samples in total. Among them, we selected only studies that were paired-end. For these paired-end reads, we performed a quality control using `fastp` [10], after which each sample was profiled using `mOTUs` [40]. Among the profiles we removed those having too few species (less than 100).

A.2 Supplementary Figures

A.2.1 Experiment design illustration

Figure S1 provides a conceptual idea of the experimental design adopted in most of the simulated experiments presented in this paper. The dissimilarity can be considered to be the dissimilarity between the center of the two circles labeled "range".

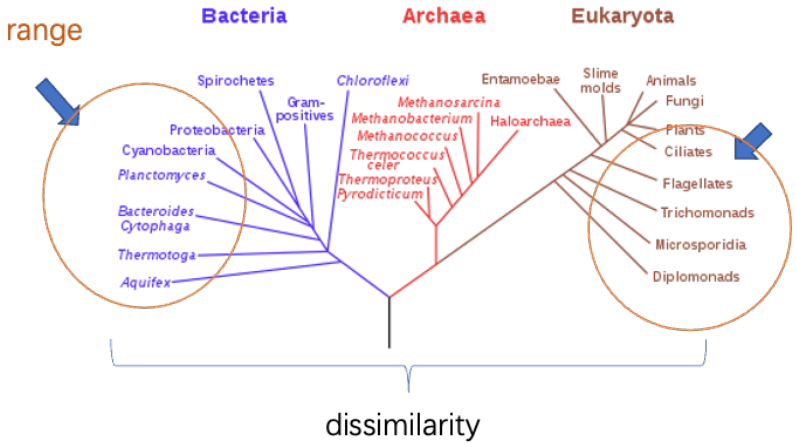


Figure S1 An illustration demonstrating the concept of range and dissimilarity.

A.2.2 Clustering quality measured using different metrics

Figure S2 Shows the results of Section 3.1 measured in clustering quality metrics other than the Silhouette score as presented in Figure 2.

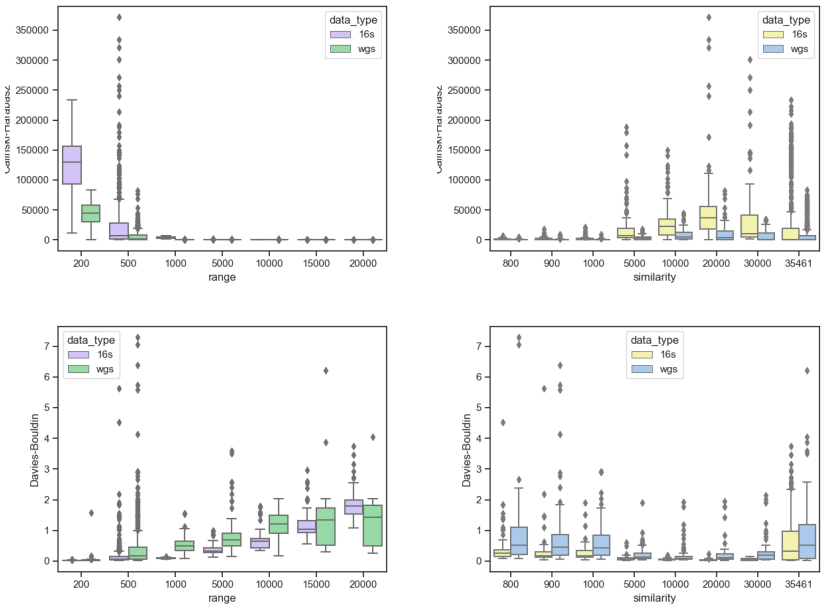


Figure S2 Clustering quality measured using different metrics. Top panel: Calinski-Harabasz index. Bottom panel: Davies-Bouldin index.

A.3 Effect of branch length function further demonstrated using real data

Figure S3 shows a comparison between PCoA plots produced using branch lengths function x^{-1} versus branch lengths function x^{-2} on real WGS data from different body sites. The plots suggest that x^{-2} allows clusters to cluster more tightly, but otherwise not offering other significant insights, demonstrating the robustness of WGSUniFrac.

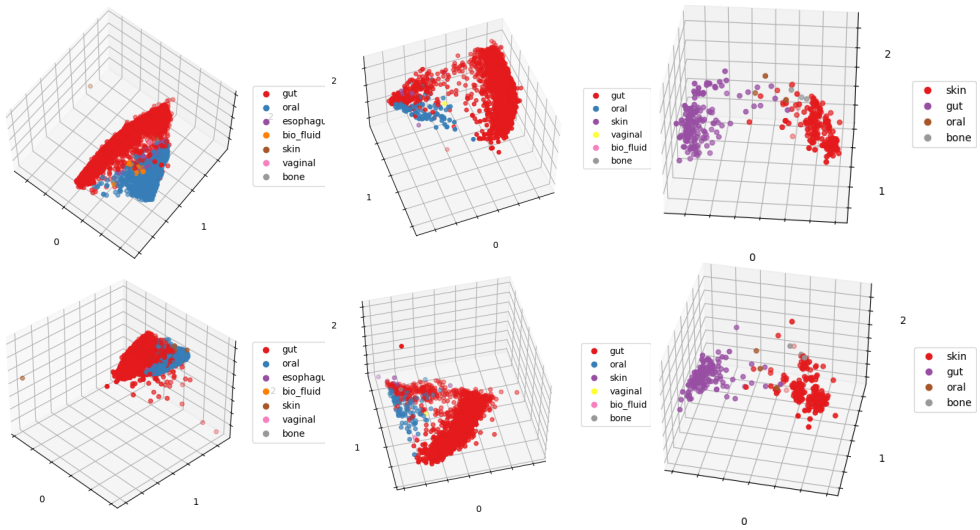


Figure S3 From left to right: low diversity, medium diversity, high diversity. Top: branch lengths function x^{-1}, bottom: branch lengths function x^{-2}.

A.3.1 The caveat of using non-biologically reasonable branch lengths assignments

Figure S4 is produced using one of the raw data used in Section 3.5 with range 1,500 and dissimilarity 1,500 among 3,000 organisms. WGSUniFrac was applied on profiles generated from these data using branch lengths functions ranging from x^{-4} to x^4. The results show that clustering quality decreases as the exponent increases. This suggests that though WGSUniFrac is largely insensitive to branch lengths assignment methods, be it data-driven or model-based, the user should avoid using branch lengths assignments that create taxonomic trees that are simply too far from the reality.

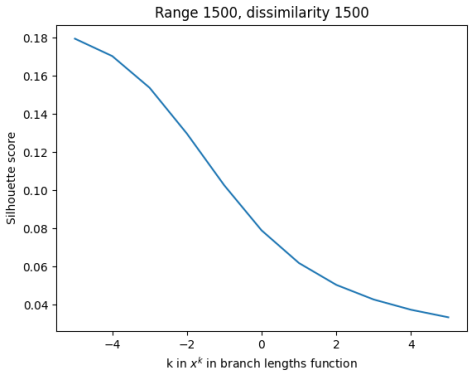


Figure S4 A plot showing the effect of k on clustering quality. As k increases, the topological structure becomes less and less biologically reasonable, resulting in a decrease in clustering quality.

Reconstructing Phylogenetic Networks via Cherry Picking and Machine Learning

Giulia Bernardini
University of Trieste, Italy
CWI, Amsterdam, The Netherlands

Leo van Iersel
Delft Institute of Applied Mathematics, Delft University of Technology, The Netherlands

Esther Julien
Delft Institute of Applied Mathematics, Delft University of Technology, The Netherlands

Leen Stougie
CWI and Vrije Universiteit, Amsterdam, The Netherlands
Erable, France

Abstract

Combining a set of phylogenetic trees into a single phylogenetic network that explains all of them is a fundamental challenge in evolutionary studies. In this paper, we apply the recently-introduced theoretical framework of cherry picking to design a class of heuristics that are guaranteed to produce a network containing each of the input trees, for practical-size datasets. The main contribution of this paper is the design and training of a machine learning model that captures essential information on the structure of the input trees and guides the algorithms towards better solutions. This is one of the first applications of machine learning to phylogenetic studies, and we show its promise with a proof-of-concept experimental study conducted on both simulated and real data consisting of binary trees with no missing taxa.

2012 ACM Subject Classification Applied computing → Computational biology

Keywords and phrases Phylogenetics, Hybridization, Cherry Picking, Machine Learning, Heuristic

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.16

Supplementary Material *Software (Source Code)*: `https://github.com/estherjulien/HybridML`, archived at `swh:1:dir:da2304df7fc11a746b630fcbe73fbbe3e2ea655d`

Funding This paper received funding from the Netherlands Organisation for Scientific Research (NWO) under project OCENW.GROOT.2019.015 "Optimization for and with Machine Learning (OPTIMAL)".
Giulia Bernardini: MUR-FSE REACT EU – PON R&I 2014-2020
Leen Stougie: The Netherlands Organisation for Scientific Research (NWO) under Gravitation-grant NETWORKS-024.002.003

Acknowledgements The authors thank Remie Janssen for providing ideas and preliminary code for the randomised heuristics, and Yukihiro Murakami for the inspiring discussions.

1 Introduction

Phylogenetic networks describe the evolutionary relationships between, for example, genes, genomes, or species. One of the first and most natural approaches to construct phylogenetic networks is to build a network from a set of gene trees. In the absence of incomplete lineage sorting, the constructed network is naturally required to "display", or embed, each of the gene trees. In addition, following the parsimony principle, a network assuming a minimum

number of reticulate evolutionary events (like hybridization or lateral gene transfer) is often sought. Unfortunately, the associated computational problem, called Hybridization, is NP-hard even for two binary input trees with equal leaf sets [4].

For a long time, research on this topic was mostly restricted to inputs consisting of two trees. Proposed algorithms for multiple trees were either completely impractical or ran in reasonable time only for very small numbers of input trees. This situation changed drastically with the introduction of so-called cherry-picking sequences [10]. This theoretical set-up opened the door to solving instances consisting of many input trees, like most practical datasets have. Indeed, a recent paper showed that this technique can be used to solve instances with up to 100 input trees to optimality [19], although it was restricted to binary trees all having the same leaf set and to so-called "tree-child" networks. Moreover, its running time has a (strong) exponential dependence on the number of reticulate events.

In this paper, we show significant progress towards a fully practical method by developing heuristics that are based on cherry picking and machine learning. Admittedly, our method is not yet widely applicable since it is still restricted to binary trees. However, our set-up is made in such a way that it can be extended to general trees, and we plan to do so in future work. Furthermore, although our method can theoretically be applied to trees with different leaf sets, in this paper we only conduct experiments on input trees with equal leaf sets. Still, we see our current method already as a breakthrough as it scales well with number of trees, number of taxa and number of reticulations. In fact, we experimentally show that it can easily handle sets of 100 trees in reasonable time (13 minutes on average for sets consisting of trees with 100 leaves each). We also have a faster version of the heuristic that already finds feasible solutions in 8 seconds for the same instances. As the running time depends at most quadratically on the input size, and linearly on the output number of reticulations, we expect it to be able to solve much larger instances still in a reasonable amount of time. In addition, our method is not restricted to tree-child networks.

The method we present is one of the first applications of machine learning in phylogenetics, and shows its promise. In particular, for the networks generated in our simulation study, it shows that features of interest of the networks can be identified with very high test accuracy (99.8%) purely based on the trees displayed by the networks. It is important to note at this point that no method is able to reconstruct any specific network from displayed trees as networks are, in general, not uniquely determined by the trees they display [12].

We focus on *orchard* networks (also called *cherry picking* networks), which are precisely those networks that can be drawn as a tree with additional horizontal arcs [17]. Such horizontal arcs can for example correspond to lateral gene transfer events (LGT). Nevertheless, orchard networks are applicable more broadly because also networks in which reticulations represent hybridization or recombination can be orchard networks. In particular, the orchard networks class is much bigger than the class of tree-child networks.

Related work. Previous practical algorithms for Hybridization include PIRN [21], PIRNs [11] and Hybroscale [1], exact methods that are only applicable to (very) small numbers of trees and/or to trees that can be combined into a network with a (very) small reticulation number.

The theoretical framework of cherry picking was introduced in [7] (for the restricted class of temporal networks) and [10] (for the class of tree-child networks) and was turned into algorithms for reconstructing tree-child [19] and temporal [5] networks. These methods can handle instances containing many trees but do not scale well with the number of reticulations, due to an exponential dependence. Other methods such as PhyloNet [20] and

PhyloNetworks [16] also construct networks from trees but have different premises and use completely different models. The class of orchard networks, which is based on cherry picking, was introduced in [15] and independently (as cherry-picking networks) in [8], although their practical relevance, as trees with added horizontal edges, was only discovered later [17].

The applicability of machine-learning techniques to Phylogenetics has not yet been fully explored, and to the best of our knowledge existing work is mainly limited to phylogenetic trees inference [2, 22] and to testing evolutionary hypotheses [9].

Our contributions. We introduce the CPH class of heuristics to combine a set of binary phylogenetic trees into a single binary phylogenetic network based on cherry picking. Specifically, we address the Hybridization problem, asking to combine a set of trees in a single network with the minimum possible reticulation number.

We define and analyse several heuristics in the CPH class, all of which are guaranteed to produce feasible solutions to Hybridization and all of which can handle instances of practical size. Two of the methods we propose are simple randomised heuristics that showed to be extremely fast and to produce good solutions when run multiple times.

The main contribution of this paper consists in a machine-learning model that potentially captures essential information about the structure of the input set of trees. We trained the model over an extensive set of synthetically generated data, and applied it to guide our algorithms towards better solutions. In proof-of-concept experiments we show that the two machine-learned heuristics we design yield satisfactory results when applied to both synthetically generated and real data.

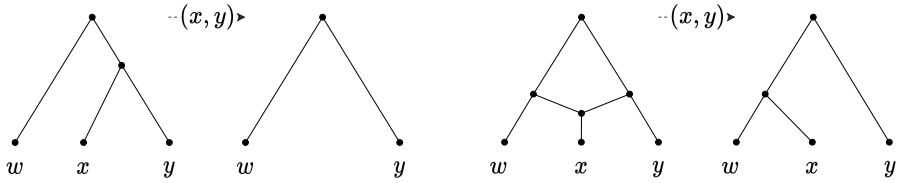
2 Preliminaries

A *phylogenetic network* $N = (V, E, X)$ on a set of taxa X is a directed acyclic graph (V, E) with a single *root* with in-degree 0 and out-degree 1, and the other nodes with either (i) in-degree 1 and out-degree $k > 1$ (*tree nodes*); (ii) in-degree $k > 1$ and out-degree 1 (*reticulations*); or (iii) in-degree 1 and out-degree 0 (*leaves*). The leaves of N are biunivocally labelled by X. A surjective map $\ell : E \to \mathbb{R}^{\geq 0}$ may assign a nonnegative *branch length* to each edge of N. We will denote by $[1, n]$ the set of integers $\{1, 2, ..., n\}$.

Throughout this paper, we will only consider binary networks (with $k = 2$), and we will identify the leaves with their labels. We will also often drop the term "phylogenetic", as all the networks considered in this paper are phylogenetic networks. The *reticulation number* $r(N)$ of a network N is $\sum_{v \in V} \max(0, d^-(v) - 1)$, where $d^-(v)$ is the in-degree of v. A network T with $r(T) = 0$ is a *phylogenetic tree*. It is easy to verify that binary networks with $r(N)$ reticulations have $|X| + r(N) - 1$ tree nodes.

Cherry-picking. We denote by $\mathcal{N}$ a set of networks and by $\mathcal{T}$ a set of trees. An ordered pair of leaves (x, y), $x \neq y$, is a *cherry* in a network if x and y have the same parent; (x, y) is a *reticulated cherry* if the parent $p(x)$ of x is a reticulation, and $p(y)$ is a tree node and a parent of $p(x)$ (see Figure 1). A pair is *reducible* if it is either a cherry or a reticulated cherry.

Reducing (or *picking*) a cherry (x, y) in a network N is the action of deleting x and replacing the two edges $(p(p(x)), p(x))$ and $(p(x), y)$ with a single edge $(p(p(x)), y)$ (see Figure 1a). If N has branch lengths, the length of the new edge is $\ell(p(p(x)), y) = \ell(p(p(x)), p(x)) + \ell(p(x), y)$. A reticulated cherry (x, y) is reduced (picked) by deleting the edge $(p(y), p(x))$ and replacing the other edge $(z, p(x))$ incoming to $p(x)$, and the consecutive edge $(p(x), x)$, with a single



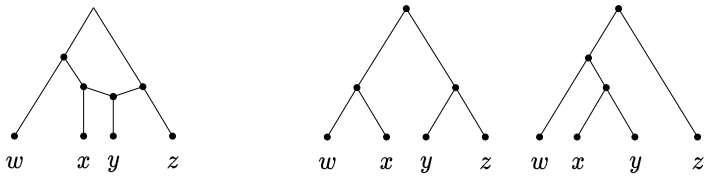
(a) Cherry picking. **(b)** Reticulated cherry picking.

Figure 1 Cherry (x, y) is picked in two different networks. In Fig. 1a (x, y) is a cherry, and in Fig. 1b (x, y) is a reticulated cherry. After picking, degree-two nodes are replaced by a single edge.

edge (z, x). The length of the new edge is $\ell(z, x) = \ell(z, p(x)) + \ell(p(x), x)$ (if N has branch lengths). Reducing a non-reducible pair has no effect on N. In all cases, the resulting network is denoted by $N_{(x,y)}$: we say that (x, y) affects N if $N \neq N_{(x,y)}$.

Any sequence $S = (x_1, y_1), \ldots, (x_n, y_n)$ of ordered leaf pairs, with $x_i \neq y_i$ for all i, is a *partial cherry-picking sequence*; S is a cherry-picking sequence (CPS) if, for each $i < n$, $y_i \in \{x_{i+1}, \ldots, x_n, y_n\}$. Given a network N and a (partial) CPS S, we denote by N_S the network obtained by reducing in N each element of S, in order. We say that S fully reduces N if N_S consists of the root with a single leaf. N is an *orchard network* (ON) if there exists a CPS that fully reduces it. If S fully reduces all $N \in \mathcal{N}$, we say that S fully reduces $\mathcal{N}$. In particular in this paper we will be interested in CPS that fully reduce a set of trees.

Hybridization. The Hybridization problem can be thought of as the computational problem of combining a set of phylogenetic trees into a network with the smallest possible reticulation number, that is, to find a network that displays each of the input trees in the sense specified by Definition 1. See Figure 2 for an example. The definition describes not only what it means to display a tree but also to display another network, which will be useful later.



(a) Network N. **(b)** Input tree set $\mathcal{T}$.

Figure 2 The two trees in Fig. 2b are displayed in the network of Fig. 2a.

▶ **Definition 1.** *Let $N = (V, E, X)$ and $N' = (V', E', X')$ be networks on the sets of taxa X and $X' \subseteq X$, respectively. The network N' is* displayed *in N if there is an* embedding *of N' in N: an injective map of the nodes of N' to the nodes of N, and of the edges of N' to edge-disjoint paths of N, such that the mapping of the edges respects the mapping of the nodes, and the mapping of the nodes respects the labelling of the leaves.*

We call *exhaustive* a tree displayed in $N = (V, E, X)$ with the whole X as leaf set. Note that Definition 1 only involves the topologies of the networks, disregarding possible branch lengths. In the following problem definition, the input trees may or may not have branch lengths, and the output is a network without branch lengths. We allow branch lengths for the input because they will be useful for the machine-learned heuristics of Section 4.

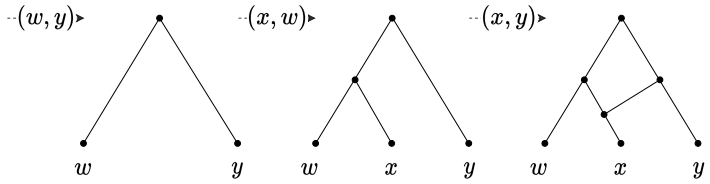


Figure 3 The ON reconstructed from the sequence $S = (x, y), (x, w), (w, y)$. The pairs are added to the network in reverse order: if the first element of a pair is not yet in the network, it is added as a cherry with the second element (see the pair (x, w)). Otherwise, a reticulation is added above the first element with an incoming edge from a new parent of the second element (see the pair (x, y)).

Hybridization
Input: A set of phylogenetic trees $\mathcal{T}$ on a set of taxa X.
Output: A network displaying $\mathcal{T}$ with minimum possible reticulation number.

3 Solving the Hybridization Problem via Cherry-Picking Sequences

We will develop heuristics for the Hybridization problem using cherry-picking sequences that fully reduce the input trees, leveraging the following result by Janssen and Murakami.

▶ **Theorem 2** ([8], Theorem 3). *Let N be a binary orchard network, and N' a (not necessarily binary) orchard network on sets of taxa X and $X' \subseteq X$, respectively. If a minimum-length CPS S that fully reduces N also fully reduces N', then N' is displayed in N.*

For binary ON, the following lemma, which is a special case of [8, Lemma 1], also holds.

▶ **Lemma 3.** *Let N be a binary orchard network, and let (x, y) be a reducible pair of N. Then reducing (x, y) and then adding it back to $N_{(x,y)}$ results in N.*

Theorem 2 and Lemma 3 provide the following approach for finding a feasible solution to Hybridization: find a CPS S that fully reduces all the input trees, and then reconstruct the unique binary orchard network N for which S is a minimum-length CPS. N can be reconstructed from S using one of the methods underlying Lemma 3 proposed in the literature, e.g., in [8] (illustrated in Figure 3) or in [19]. In doing so, the number of reticulations of the resulting N is $r(N) = |S| - |X| + 1$ [18]. In the next section we focus on the first part of the heuristic: producing a CPS that fully reduces a given set of phylogenetic trees.

3.1 Randomised Heuristics

We define a class of randomised heuristics that construct a CPS by picking one reducible pair of the input set $\mathcal{T}$ at a time and by appending this pair to a growing partial sequence, as described in Algorithm 1 (the two subroutines PickNext and CompleteSeq will be described in details below). We call this class CPH (for Cherry-Picking Heuristics). Recall that $\mathcal{T}_S$ denotes the set of trees $\mathcal{T}$ after reducing all trees with a (partial) CPS S.

Algorithm 1 CPH.

INPUT: A set $\mathcal{T}$ of phylogenetic trees
OUTPUT: A CPS reducing $\mathcal{T}$.

1: $S \leftarrow \emptyset$;
2: **while** there is a reducible pair in $\mathcal{T}_S$ **do**
3: $(x, y) \leftarrow \mathsf{PickNext}(\mathcal{T}_S)$;
4: $S \leftarrow S \circ (x, y)$;
5: Reduce (x, y) in all trees of $\mathcal{T}_S$;
6: $S \leftarrow \mathsf{CompleteSeq}(S)$;
7: **return** S;

The while loop at lines 2-5 produces, in general, a partial CPS S, as shown in Example 4. To make it into a CPS, the subroutine CompleteSeq at line 6 appends at the end of S a sequence S' of pairs such that each second element in a pair of $S \circ S'$ is a first element in a later pair (except for the last one), as required by the definition of CPS. These additional pairs do not affect the trees in $\mathcal{T}$, that are already fully reduced by S. Algorithm 2 in Appendix A describes a procedure CompleteSeq that runs in time linear in the length of S.

▶ **Example 4.** Let $\mathcal{T}$ consist of the two 2-leaf trees (x, y) and (w, z). A partial CPS at the end of the while loop in Algorithm 1 could be, e.g., $S = (x, y), (w, z)$. The two trees are both reduced to one leaf, so there are no more reducible pairs, but S is not a CPS. To make it into a CPS it suffices to append either pair (y, z) or pair (z, y): e.g., $S \circ (y, z) = (x, y), (w, z), (y, z)$ is a CPS, and it still fully reduces the two input trees.

The class of heuristics described in Algorithm 1 is concretised in different heuristics depending on the function PickNext at line 3, which is used to choose a reducible pair at each iteration. To formulate them below we need to introduce the notions of height pair and trivial pair. Let N be a network with branch lengths and let (x, y) be a reducible pair in N. The *height pair* of (x, y) in N is a pair $(h_x^N, h_y^N) \in \mathbb{R}^2_{\geq 0}$, where $h_x^N = \ell(p(x), x)$ and $h_y^N = \ell(p(y), y)$ if (x, y) is a cherry, and $h_x^N = \ell(p(y), p(x)) + \ell(p(x), x)$ and $h_y^N = \ell(p(y), y)$ if (x, y) is a reticulated cherry. The *height* $h_{(x,y)}^N$ of (x, y) is the average $(h_x^N + h_y^N)/2$ of h_x^N and h_y^N.

Let $\mathcal{T}$ be a set of trees whose leaf sets are subsets of a set of taxa X. An ordered leaf pair (x, y) is a *trivial pair* of $\mathcal{T}$ if it is reducible in each $T \in \mathcal{T}$ that contains both x and y, and there is at least one tree in which it is reducible. We define the following three heuristics in the CPH class, resulting from as many possible implementations of PickNext.

Rand. Function PickNext picks uniformly at random a reducible pair of $\mathcal{T}_S$.

LowPair. Function PickNext picks a reducible pair (x, y) with the lowest average of values $h_{(x,y)}^T$ over all $T \in \mathcal{T}_S$ in which (x, y) is reducible (ties are broken randomly).

TrivialRand. Function PickNext picks a trivial pair if there exists one, and otherwise picks a reducible pair of $\mathcal{T}_S$ uniformly at random.

▶ **Theorem 5.** *Algorithm 1 computes a CPS that fully reduces $\mathcal{T}$, for any function PickNext that picks, in each iteration, a reducible pair of $\mathcal{T}_S$.*

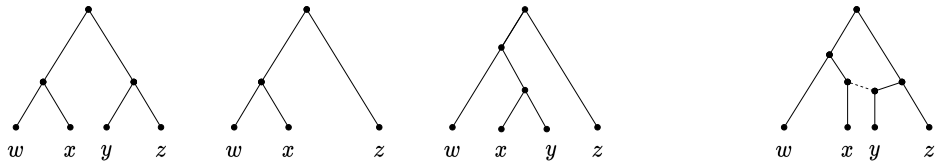
Proof. The sequence S is initiated as an empty sequence. Then, each iteration of the while loop (lines 2-5) of Algorithm 1 appends one pair to S that is reducible in at least one of the trees in $\mathcal{T}$, and reduces it in all trees. Hence, in each iteration the total size of $\mathcal{T}_S$ is reduced, so the algorithm finishes in finite time. Moreover, at the end of the while loop, each tree in $\mathcal{T}_S$ is reduced, thus the partial CPS S reduces $\mathcal{T}_S$. As CompleteSeq only appends pairs at the end of S, the result of this subroutine still reduces all trees in $\mathcal{T}_S$. ◀

In Section 5 we experimentally show that TrivialRand produces the best results among the proposed randomised heuristics. Rand is the fastest heuristic, its running time being trivially bounded by $\mathcal{O}(|\mathcal{T}|^2|X|)$ (see Lemma 9 in Appendix A). In the next section we introduce a further heuristic step for TrivialRand which improves the quality of the output even more.

3.2 Improving Heuristic TrivialRand via Tree Expansion

With respect to a trivial pair (x, y) each tree T in the set is of one of the following types: (i) (x, y) is a reducible pair of T; or (ii) neither x nor y are leaves of T; or (iii) y is a leaf of T but x is not; or (iv) x is a leaf of T but y is not.

Suppose that at some iteration of TrivialRand, the subroutine PickNext returns the trivial pair (x, y). Then, before reducing (x, y) in all trees, we do the following extra step: for each tree of type (iv), replace leaf x with cherry (x, y). We call this operation the *tree expansion* of $\mathcal{T}$: see Figure 4c. The effect of this step is that, after reducing (x, y), leaf x disappears from the set of trees, which would have not necessarily been the case before, because of trees of type (iv). The tree expansion followed by the reduction of (x, y) can, alternatively, be seen as relabelling leaf x in any tree of type (iv) by y.



(a) Input tree T. **(b)** $T_{S^1} = T_{(y,z)}$. **(c)** Expand T_{S^1} with (x, y). **(d)** Network N_T^{exp}.

Figure 4 Tree expansion of T **(a)** with the trivial cherry (x, y) of $\mathcal{T}_{(y,z)}$. **(b)** After picking cherry (y, z), leaf y is missing in T_{S^1}. **(c)** Leaf x is replaced by the cherry (x, y). After completion of the heuristic, we have $S_T = (y, z), (x, y), (y, w), (w, z)$. **(d)** The network N_T^{exp} reconstructed from S_T. Note that the input tree T is displayed in N_T^{exp} (solid edges).

To guarantee that a CPS S produced with tree expansion implies a feasible solution for Hybridization, we must show that the network N reconstructed from S displays all the trees in the input set $\mathcal{T}$. We prove that indeed this is the case with the following lemma.

▶ **Lemma 6.** *Let S be the CPS outputted by TrivialRand with input $\mathcal{T}$ and let S contain $j \geq 0$ trivial pairs. Then the network reconstructed from S displays all the trees in $\mathcal{T}$.*

Proof. We define the *network expansion* of $\mathcal{T}$ with a trivial pair, denoted by $\mathcal{T}^{exp}$, as follows. Let S^{i-1} be the partial CPS constructed in the first $i-1$ steps of TrivialRand, and let i be the first step in which we pick a trivial pair (x, y). For each $T \in \mathcal{T}$ that is reduced by S^{i-1} to a tree $T_{S^{i-1}}$ of type (iv) for (x, y), let S_T^{i-1} be the subsequence of S^{i-1} consisting only of the pairs that subsequently affect T. We use the partial CPS $S_T^{i-1} \circ (x, y)$ to reconstruct a network N_T^{exp} with a method underlying Lemma 3, starting from $T_{S^{i-1}}$: see Figure 4d. For trees of type (i)-(iii), $N_T^{exp} = T$. The set $\mathcal{T}^{exp}$ then consists of networks N_T^{exp} for all $T \in \mathcal{T}$. Note that, by construction and Lemma 3, all the elements of $\mathcal{T}^{exp}_{S^{i-1} \circ (x,y)}$ are trees.

We can generalise this notion to multiple trivial pairs: we denote by $\mathcal{T}^{exp(j)}$ the network expansion of $\mathcal{T}$ with the j-th trivial pair, (w, z) say, and suppose it is added to the partial CPS S at the k-th step. Consider a tree $T' \in \mathcal{T}^{exp(j-1)}_{S^{k-1}}$ of type (iv) for (w, z), and let $N_T^{exp(j-1)} \in \mathcal{T}^{exp(j-1)}$ be the network it originated from. Let S_T^{k-1} be the subsequence of

S^{k-1} consisting only of the pairs that subsequently affected $N_T^{exp(j-1)}$. Then $N_T^{exp(j)}$ is the network reconstructed from $S_T^{k-1} \circ (w, z)$, starting from T'. For trees of $\mathcal{T}_{S^{k-1}}^{exp(j-1)}$ that are of type (i)-(iii) for (w, z), we define $N_T^{exp(j)} = N_T^{exp(j-1)}$. The elements of $\mathcal{T}^{exp(j)}$ are all networks $N_T^{exp(j)}$. For completeness, we define $\mathcal{T}^{exp(0)} = \mathcal{T}$ and $\mathcal{T}^{exp(1)} = \mathcal{T}^{exp}$.

By construction, S fully reduces all the networks in $\mathcal{T}^{exp(j)}$, thus the network N reconstructed from S displays all of them by Theorem 2. We prove that $N_T^{exp(j)}$ displays T for all $T \in \mathcal{T}$, and thus N displays the original tree set $\mathcal{T}$ too, by induction on j.

In the base case we pick $j = 0$ trivial pairs, so the statement is true by Theorem 2. Now let $j > 0$. The induction hypothesis is that each network $N_T^{exp(j-1)} \in \mathcal{T}^{exp(j-1)}$ displays the tree T it originates from. Let (w, z) be the j-th trivial pair, added to the sequence at position k. Let $T' \in \mathcal{T}_{S^{k-1}}^{exp(j-1)}$ be a tree of type (iv) for (w, z), and let $N_T^{exp(j-1)}$ be the network it originates from. Then there are two possibilities: either z is a leaf of $N_T^{exp(j-1)}$ or it is not. In case it is not, then adding (w, z) to $N_T^{exp(j-1)}$ does not create any new reticulation, and clearly $N_T^{exp(j)}$ keeps displaying T. If z does appear in $N_T^{exp(j-1)}$, then it must have been reduced by a pair (z, v) of S^{k-1} (otherwise T' would not be of type (iv)). Then the network $N_T^{exp(j)}$ has an extra reticulation, created with the insertion of (z, v) at some point after (w, z) during the backwards reconstruction. In both cases, by [8, Lemma 10] $N_T^{exp(j-1)}$ is displayed in $N_T^{exp(j)}$, and thus by the induction hypothesis T is displayed too. ◀

3.3 Good Cherries in Theory

By Lemma 3, the binary network N reconstructed from a CPS S is such that S is of minimum length for N. By Theorem 2, if S, in turn, fully reduces $\mathcal{T}$, then N displays all the trees in $\mathcal{T}$. Depending on S, though, N is not necessarily an optimal network (that is, with minimum reticulation number) among the ones displaying $\mathcal{T}$.

Let $\mathsf{OPT}(\mathcal{T})$ denote the set of networks that display $\mathcal{T}$ with the minimum possible reticulation number. Ideally, we would like to produce a CPS fully reducing $\mathcal{T}$ that is also a minimum-length CPS fully reducing some network of $\mathsf{OPT}(\mathcal{T})$. In other words, we aim to find a CPS $\tilde{S} = (x_1, y_1), \ldots, (x_n, y_n)$ such that, for any $i \in [1, n]$, (x_i, y_i) is a reducible pair of $\tilde{N}_{\tilde{S}^{i-1}}$, where $\tilde{S}^0 = \emptyset$, $\tilde{S}^k = (x_1, y_1), \ldots, (x_k, y_k)$ for all $k \in [1, n]$, and $\tilde{N} \in \mathsf{OPT}(\mathcal{T})$.

Let $S = (x_1, y_1), \ldots, (x_n, y_n)$ be a CPS fully reducing $\mathcal{T}$ and let $\mathsf{OPT}_{S^k}(\mathcal{T})$ consist of all networks $N \in \mathsf{OPT}(\mathcal{T})$ such that each pair (x_i, y_i), $i \in [1, k]$, is reducible in $N_{S^{i-1}}$.

▶ **Lemma 7.** *A CPS S reducing $\mathcal{T}$ reconstructs an optimal network $\tilde{N}$ if and only if each pair (x_i, y_i) of S is reducible in $\tilde{N}_{S^{i-1}}$, for all $i \in [1, n]$.*

Proof. ($\implies$) By Lemma 3, S is a minimum-length CPS for the network $\tilde{N}$ that is reconstructed from it; and a CPS $C = (w_1, z_1), \ldots, (w_n, z_n)$ reducing a network N is of minimum length precisely if, for all $j \in [1, n]$, (w_j, z_j) is a reducible pair of $N_{C^{j-1}}$ (otherwise the pair (w_j, z_j) could be removed from C and the new sequence would still reduce N).
($\impliedby$) If all pairs of S affect some optimal network $\tilde{N}$, then S is a minimum-length CPS for $\tilde{N}$, thus $\tilde{N}$ is reconstructed from S (and it displays $\mathcal{T}$ by Theorem 2). ◀

Lemma 7 implies that if some pair (x_i, y_i) of S does not reduce any network in $\mathsf{OPT}_{S^{i-1}}(\mathcal{T})$, then the network reconstructed from S is not optimal: see Example 8.

▶ **Example 8.** Consider the set $\mathcal{T}$ of Figure 2b: $S = (y, x), (y, z), (w, x), (x, z)$ is a CPS that fully reduces $\mathcal{T}$ and consists only of pairs successively reducible in the network N of Fig. 2a, thus it reconstructs it by Lemma 3. Now consider (w, x), which is reducible in $\mathcal{T}$ but not in N, and pick it as first pair, to obtain e.g. $S' = (w, x), (y, z), (y, x), (w, x), (x, z)$. The network N' reconstructed from S', depicted in Figure 5, has $r(N') = 2$, whereas $r(N) = 1$.

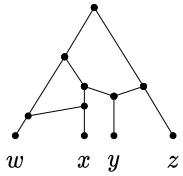


Figure 5 The network N' of Example 8.

Suppose we are incrementally constructing a CPS $S = (x_1, y_1), \ldots, (x_n, y_n)$ for $\mathcal{T}$ with some heuristic in the CPH class. If we had an oracle that at each iteration i told us if a reducible pair (x, y) of $\mathcal{T}_{S^{i-1}}$ were a reducible pair in some $N \in \mathsf{OPT}_{S^{i-1}}(\mathcal{T})$, then, by Lemma 7, we could solve Hybridization optimally. Unfortunately no such exact oracle can exist (unless $P = NP$). However, in the next section we exploit this idea to design machine-learned heuristics in the CPH framework.

4 Predicting Good Cherries via Machine Learning

In this section, we present a supervised machine-learning classifier that (imperfectly) simulates the ideal oracle described at the end of Section 3.3. The goal is to predict, based on $\mathcal{T}$, whether a given cherry of $\mathcal{T}$ is a reducible pair in a network N displaying $\mathcal{T}$ with a close-to-optimal number of reticulations, without knowing N. Based on Lemma 7, we then exploit the output of the classifier to define new functions PickNext, that in turn define new machine-learned heuristics in the class of CPH (Algorithm 1).

Specifically, we train a random forest classifier on data that encapsulates information on the cherries in the tree set. Each reducible pair in $\mathcal{T}_S$ is represented by one data point. Each data point is a pair $(\mathbf{F}, \mathbf{c})$, where $\mathbf{F}$ is an array of the features of cherry (x, y) and $\mathbf{c}$ is an array containing the probability that the cherry belongs to each of the possible classes described below. Recall that cherries are ordered pairs, so (x, y) and (y, x) give rise to two distinct data points. The classification model learns the association between $\mathbf{F}$ and $\mathbf{c}$.

The true class of a cherry (x, y) of $\mathcal{T}$ depends on whether, for the (unknown) network N that we aim to reconstruct: (class 1) (x, y) is a cherry of N; (class 2) (x, y) is a reticulated cherry of N; (class 3) (x, y) is not reducible in N, but (y, x) is a reticulated cherry; or (class 4) neither (x, y) nor (y, x) are reducible in N. Thus, for the data point of a cherry (x, y), $\mathbf{c}[i]$ contains the probability that (x, y) is in class i, and $\mathbf{c}[1] + \mathbf{c}[2]$ gives the predicted probability that (x, y) is reducible in N. We define the following two heuristics in the CPH framework.

ML. Given a threshold $t \in [0, 1)$, function PickNext picks the cherry with the highest predicted probability of being reducible in N, if this probability is at least t; or a random cherry if none of them have a probability of being reducible above the threshold.

TrivialML. Function PickNext picks a random trivial pair, if there exists one; otherwise it uses the same rules as ML.

In both cases, whenever a trivial pair is picked, we do tree expansion, as described in Section 3.2. Note that if $t = 0$, since the predicted probabilities are never exactly 0, ML is fully deterministic. In Section 5 we study the performance of ML with different thresholds.

Table 1 Features of a cherry (x, y). Features 6-12 can be computed for both branch lengths and unweighted branches. We refer to these two options as *distance* and *topological distance*, respectively.

Num	Feature name	Description
1	Cherry in tree	Ratio of trees that contain cherry (x, y)
2	Trivial	Ratio of trees with both leaves x and y that contain cherry (x, y)
3	Leaves in tree	Ratio of trees that contain both leaves x and y
4	New cherries	Number of new cherries of $\mathcal{T}$ after picking cherry (x, y)
5	Before/after	Ratio of the number of cherries of $\mathcal{T}$ before/after picking cherry (x, y)
Features measured by distance (d) and topology (t)		
$6_{d,t}$	Cherry depth	Avg over trees with (x, y) of ratios "depth of (x, y) in the tree/depth of tree"
$7_{d,t}$	Tree depth	Avg over trees with (x, y) of ratios "depth of tree/max depth of trees with (x, y)"
$8_{d,t}$	Leaf distance	Avg over trees with x and y of ratios "x-y leaf distance/depth of the tree"
$9_{d,t}$	LCA distance	Avg over trees with x and y of ratios "x-LCA(x, y) distance/y-LCA(x, y) distance"
$10_{d,t}$	Leaf depth x	Avg over trees with x and y of ratios "root-x distance/depth of tree"
$11_{d,t}$	Leaf depth y	Avg over trees with x and y of ratios "root-y distance/depth of tree"
$12_{d,t}$	Depth x/y	Avg over trees with x and y of ratios "root-x distance and root-y distance"

To assign a class to each cherry, we define 19 features, summarised in Table 1, that may capture essential information about the structure of the set of trees, and that can be efficiently computed and updated at every iteration of the heuristics.

The *depth* (resp. *topological* depth) of a node u in a tree T is the total branch length (resp. the total number of edges) on the root-to-u path; the depth of T is the maximum depth of any leaf of T; the depth of a cherry (x, y) is the depth of the common parent of x and y. The (topological) leaf distance between x and y is the total branch length of the path from the parent of x to the lowest common ancestor of x and y, denoted by LCA(x, y), plus the total length of the path from the parent of y to LCA(x, y) (resp. the total number of edges on both paths). In particular, the leaf distance between the leaves of a cherry is zero.

Features 1-5 can be initially computed for all cherries of $\mathcal{T}$ with two bottom-up traversals of all trees: one for identifying and storing all cherries of $\mathcal{T}$, the second for actually computing the features. Picking a cherry then entails recomputing features 1-2 and 4-5 for up to $|\mathcal{T}|$ data points ($|\mathcal{T}|$ denotes the number of trees), as this can result in up to one new cherry in each tree; and recomputing feature 3 for up to $|X|$ cherries, where X is the set of taxa of $\mathcal{T}$.

Features $6_{d,t}$ to $12_{d,t}$ can also be initially computed with a traversal of $\mathcal{T}$, made efficient by preprocessing each tree to allow constant-time LCA queries [6] and by storing the depth (both topological and w.r.t. branch lengths) of each node of each tree. This preprocessing also allows for efficient updates of each feature. We defer a thorough analysis of the time complexity of computing and updating the features to a future full version of this paper.

Obtaining Training Data. The high-level idea to obtain training data is to first generate a phylogenetic network N; then to extract a subset $\mathcal{T}$ of the trees displayed in N; and finally, to iteratively choose a random reducible pair (x, y) of N, to reduce it in $\mathcal{T}$ as well as in N, and to label the remaining cherries of $\mathcal{T}$ with one of the four classes defined in Section 4.

We generate phylogenetic networks with branch lengths and up to 9 reticulations using the LGT (lateral gene transfer) network generator of [14] for binary orchard networks. For each such network N, we then generate the set $\mathcal{T}$ consisting of all the exhaustive trees displayed in N. Although N is not guaranteed to be an optimal network for $\mathcal{T}$, we expect it to be reasonably close to an optimal one, because we remove redundant reticulations when we generate it and because the trees in $\mathcal{T}$ cover all the edges of N. In particular, $r(N)$ provides an upper bound estimate on the minimum possible number of reticulations of any network displaying $\mathcal{T}$. We will thus use it as a reference value for assessing the quality of our results on synthetic data.

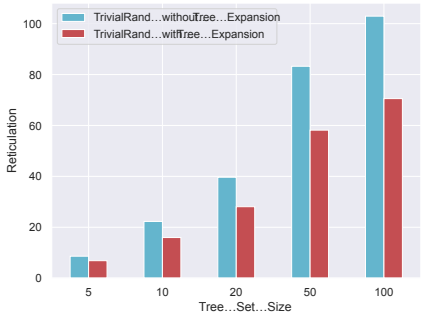


Figure 6 TrivialRand with and without tree expansion. The height of the bars is the average reticulation number over each group, obtained by selecting the best of 200 runs for each instance.

5 Experiments

The code of all our heuristics, available at `https://github.com/estherjulien/HybridML`, is written in Python. All experiments ran on an Intel Xeon Gold 6130 CPU @ 2.1 GHz with 96 GB RAM. We conducted experiments on both synthetic and real data, comparing the performance of Rand, TrivialRand, ML and TrivialML, using threshold $t = 0$. Although our methods can be applied to trees with different leaf sets, we restrict this proof-of-concept study to trees with the same set of leaves, and defer more complete experiments to future extensions. Similar to the training data, we generated two synthetic datasets by first growing a binary orchard network N using [14], and then extracting $\mathcal{T}$ as a subset of the exhaustive trees displayed in N. We provide details on each dataset in Section 5.2.

We start by analysing the usefulness of tree expansion, the heuristic rule described in Section 3.2. We synthetically generated 112 instances for each tree set size $|\mathcal{T}| \in \{5, 10, 20, 50, 100\}$ (560 in total), all consisting of trees with 20 leaves each, and grouped them by $|\mathcal{T}|$; we then ran TrivialRand 200 times (both with and without tree expansion) on each instance, selected the best output for each of them, and finally took the average of these results over each group of instances. The results are in Figure 6, showing that the use of tree expansion brought the output reticulation number down by at least 16% (for small instances) and up to 40% for the larger instances. We consistently chose to use this rule in all the heuristics that detect trivial cherries, namely, TrivialRand, TrivialML, and ML (although ML does not explicitly favour trivial cherries, it does check whether a selected cherry is trivial using feature number 2).

5.1 Prediction Model

The random forest is implemented with Python's `scikit-learn` [13] package using default settings. We evaluated the performance of our trained random forest models on different datasets in a holdout procedure: namely, we removed 10% of the data from each training dataset, trained the models on the remaining 90% and used the holdout 10% for testing. The accuracy was assessed by assigning to each test data point the class with the highest predicted probability, and comparing it with the true class. Before training the models, we balanced each dataset so that each class had the same number of representatives.

Each training dataset differed in terms of the number M of networks used for generating it and the number of leaves of the networks. For each dataset, the number L of leaves of each generated network was uniformly sampled from $[2, \max L]$, where $\max L$ is the maximum

number of leaves per network. We constructed the networks using the LGT generator of [14]. This generator has three parameters: n for the number of steps, α for the probability of lateral gene transfer events, and β for regulating the size of the biconnected components of the network (called *blobs*). The combination of these parameters determines the level (maximum number of reticulations per blob), the number of reticulations, and the number of leaves of the output network. In our experiments, α was uniformly sampled from $[0.1, 0.5]$ and $\beta = 1$. See [14] for more details.

Each generated network gave rise to a number of data points: the total number of data points per dataset is shown in Table 5 in Appendix C. Each row of Table 5 corresponds to a dataset on which the random forest can be trained, obtaining as many ML models. We tested all the models on all the synthetically generated instances: we show these results in Figure 14 in Appendix D. In Section 5.2 we will report the results obtained for the best performing model for each type of instance.

Among the advantages of using a random forest as prediction model there is the ability of computing feature importance, shown in Table 6 in Appendix C. Some of the most useful features for a cherry (x, y) seem to be "Trivial" (the ratio of the trees containing both leaves x and y in which (x, y) is a cherry) and "Cherry in tree" (the ratio of trees that contain (x, y)). This was not unexpected, as these features are well-suited to identify trivial cherries.

'Leaf distance' (the average, over all trees containing both leaves x, y, of the ratio between the x-to-y distance and the depth of the tree) and "Depth x/y" (the average, over all trees containing both leaves x, y, of the ratio between the x-to-root and the y-to-root distances) are also two important features. The rationale behind these features was to try to identify reticulated cherries. This was also the idea for the feature "Before/after", but this has, surprisingly, a very low importance score. In future extensions of this work we plan to conduct a thorough analysis on whether some of the seemingly least important features can be removed without affecting the quality of the results.

5.2 Experimental Results

We assessed the performance of our heuristics on instances of three types: Full Tree Set (FTS), Restricted Tree Set (RTS), and real data. FTS and RTS are synthetically generated. We generated the FTS instances much like we did for the training data: we first grew a network with the LGT generator and then extracted all the exhaustive trees displayed in the network. We generated FTS data for different combinations of the following parameters: $L \in \{20, 50, 100\}$ (number of leaves per tree) and $R \in \{5, 6, 7\}$ (reticulation number of the original network). Note that, for FTS instances, $|\mathcal{T}| = 2^R$.

For generating RTS instances, we also started by growing LGT networks, but we then extracted only a subset of the exhaustive trees from each of them, up to a certain amount $|\mathcal{T}| \in \{20, 50, 100\}$; the other parameter for RTS instances is the number of leaves $L \in \{20, 50, 100\}$, while the number of reticulations is uniformly sampled in the ranges $[5, 25]$, $[6, 30]$, $[7, 35]$ for $|\mathcal{T}| = 20$, $|\mathcal{T}| = 50$ and $|\mathcal{T}| = 100$, respectively. We generated 112 FTS instances and as many RTS instances for each possible combination of the parameters: by *instance group* we indicate all the (FTS or RTS) instances generated for one parameter pair.

The real-world dataset we tested our methods on consists of gene trees on homologous gene sets found in bacterial and archaeal genomes. This dataset was originally constructed in [3] and made binary in [19]. We extracted a subset of instances (Table 2) from the binary dataset, for every combination of parameters $L \in \{20, 50, 100\}$ and $|\mathcal{T}| \in \{10, 20, 50, 100\}$.

Table 2 Number of real data instances for each group (combination of parameters L and $|\mathcal{T}|$).

L	$\|\mathcal{T}\|$				Tot. Trees
	10	20	50	100	
20	50	50	50	50	1684
50	20	20	20	20	290
100	5	5	1	0	53

For the synthetically generated FTS and RTS datasets, we evaluated the performance of each heuristic in terms of the output number of reticulations, comparing it with the number of reticulations of the network N from which we extracted $\mathcal{T}$. Indeed, although N is not guaranteed to be an optimal network for $\mathcal{T}$, $r(N)$ clearly provides an estimate (from above) of the optimal value, and thus we used it as a reference value for our experimental evaluation.

For real data we used a different reference value. In the absence of the natural estimate on the optimal number of reticulations provided by the starting network, on real data we evaluated the performance of the heuristics using the best result obtained by running TrivialRand 1000 times as reference value. We also compared our results with the algorithm from [19], denoted by TreeChild, and the algorithm from [1], denoted by Hybroscale, using the same datasets that were used to test the two methods in [19], which consist of rather small instances ($|\mathcal{T}| \leq 8$) .

Full Tree Set (FTS) instances. We evaluated the output of ML with threshold $t = 0$ on all the random forest models described in Table 5. The model that gave the best results for datasets of this type is the one trained with parameters $\max L = 100, M = 500$ (see Figure 14a), so we chose to use this model for both ML and TrivialML, and compare them to Rand and TrivialRand. We ran the machine-learned heuristics once for each instance, and then averaged the results within each group. The randomised heuristics Rand and TrivialRand were run $\min\{x(I), 1000\}$ times for each instance I, where $x(I)$ is the number of runs that can be executed in the same time as one run of ML on the same instance.

In Figure 7 we summarise the results for the four heuristics. Solid bars represent the ratio of the average reticulation number to the reference value, for each instance group and for each of the four heuristics. Dashed bars represent the ratio of the average (within each group) of the best of the $\min\{x(I), 1000\}$ runs for each instance I to the reference value. The machine-learned heuristics ML and TrivialML seem to perform very similarly, the best one (on average) being one or the other depending on the instance group. The fully randomised heuristic, Rand, always performed much worse than all the others, and we omitted the results for LowPair because they were at least 44% worse, on average, than the ones for Rand.

While just one run of ML or TrivialML gave better results than the average of multiple runs of TrivialRand (up to 64% on average), picking the best output over all the runs of TrivialRand seems to give the best results for FTS instances, TrivialRand being better than ML by up to 25% on average for instances obtained from networks with 20 leaves and 5 reticulations and yielding results only slightly better than ML and TrivialML in the rest of instance groups (ML being better by 3% on average than the best output of TrivialRand only for the instance group (100,7)).

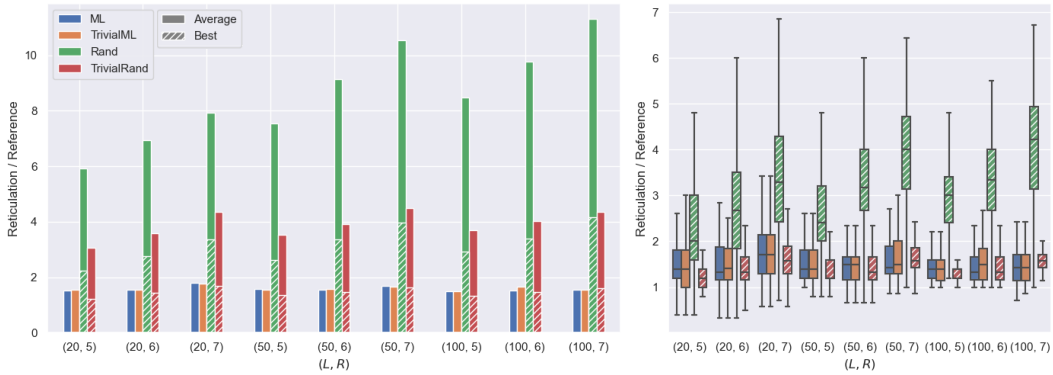


Figure 7 Results for FTS instances for each combination of the parameters (L, R) and distribution of the results per instance group (boxplots on the right, for the randomized heuristics only considering the best result over all runs on each instance).

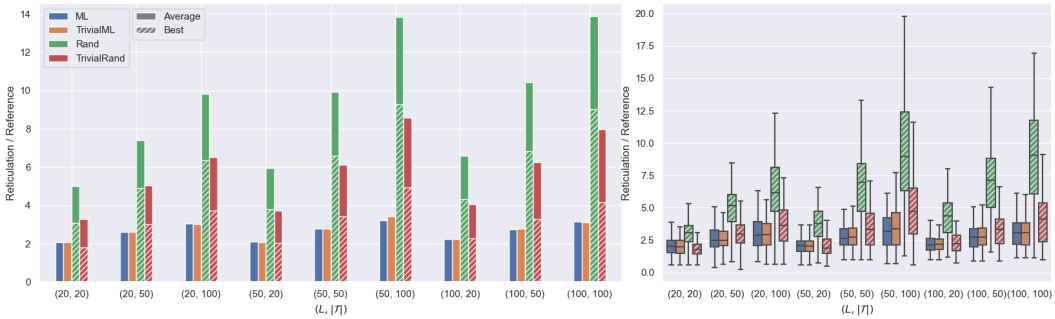


Figure 8 Results for RTS instances for each combination of parameters L and $|\mathcal{T}|$ and distribution of the results per instance group (boxplots on the right, for the randomized heuristics only considering the best result over all runs on each instance).

Restricted Tree Set (RTS) instances. The ML-model that gave the best results for RTS datasets is the one trained with parameters $\max L = 100, M = 10$ (Figure 14b in Appendix D). The results in Figure 8 were obtained with the same procedure as for FTS. Comparing Figures 7 and 8 it is clear that, for all the heuristics, it was more difficult to reconstruct N when the input trees were not the totality of the exhaustive trees displayed in N.

In particular, whereas running TrivialRand several times and picking the best output appeared to be the winning strategies for most of the FTS instances (although ML and TrivialML performed almost as good), this is not the case for RTS, where it seems that machine learning often makes up for the lack of information in the input better than the randomised strategies. For RTS instances, the result of ML was better than the best result of TrivialRand by up to 34% on average (for instances of 100 trees obtained from networks with 50 leaves), the difference being more marked for seemingly "difficult" instance groups, for which all the tested heuristics gave results diverging more from the reference value. The best result of TrivialRand was (slightly) better than the result of ML only for the instance group (20,20).

Real Data. We conducted two sets of experiments on real data, using the ML model trained on the dataset with parameters $\max L = 100, M = 10$. For sufficiently small instances, we compared the results of our heuristics with the results of two existing tools for reconstructing

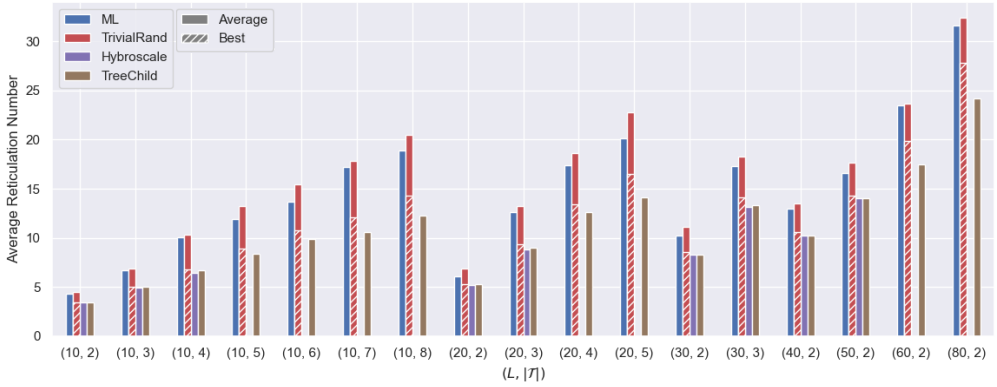


Figure 9 Comparison of ML, TrivialRand, Hybroscale, and TreeChild on real data. The distribution of the results is shown in Appendix B (Figure 12).

networks from binary trees: TreeChild [19] and Hybroscale [1]. Hybroscale is an exact method performing an exhaustive search on the networks displaying the input trees, therefore it can only handle reasonably small instances in terms of number of input trees. TreeChild is a fixed-parameter (in the number of reticulations of the output) exact algorithm that reconstructs the best *tree-child* network, a restricted class of phylogenetic networks, and due to its fast-growing computation time cannot handle large instances either.

We tested ML and TrivialRand against Hybroscale and TreeChild using the same dataset used in [19], in turn taken from [3]. The dataset consists of ten instances for each possible combination of the parameters $|\mathcal{T}| \in [2, 8]$ and $L \in \{10, 20, 30, 40, 50, 60, 80, 100, 150\}$. In Figure 9 we show results only for the instance groups for which Hybroscale or TreeChild could output a solution within 1 hour, consistent with the experiments in [19]. As a consequence of Hybroscale and TreeChild being exact methods (TreeChild only for a restricted class of networks), they performed better than both ML and TrivialRand on all instances they could solve, although the best results of TrivialRand are often close (no worse than 14%) and sometimes match the optimal value. Table 4 in the appendix summarises the running time of all the methods on all instance groups.

The main advantage of our heuristics is that they can handle much larger instances. In Figure 10 we show the results obtained for the instance groups of Table 2. Like for FTS and RTS data, Rand and TrivialRand are executed $\min\{x(I), 1000\}$ times for each instance I, $x(I)$ being the time required for executing ML once on instance I. The results are then averaged within each group and divided by the reference value that we obtained by running TrivialRand 1000 times on each instance and taking the best outputs. For this reason, in contrast with the synthetic datasets, for which we had more accurate estimates, the results of ML and TrivialML are even better (up to 10% on average) than the reference value, for some instance groups. The results essentially confirm the ones we obtained for RTS instances: the best outputs of TrivialRand are quite close to the outputs of ML (and TrivialML), being up to 15% worse and up to 9% better on average, depending on the instance group.

Effect of the Threshold on ML. We tested the effectiveness of adding a threshold $t > 0$ to ML on different types of data (FTS, RTS and real). For these experiments we used a subset of the instance groups, consisting of trees obtained using the parameters $R = 5$ for

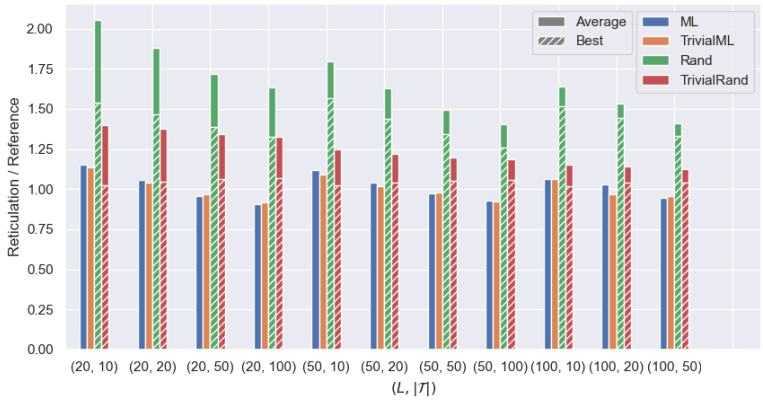


Figure 10 Results for the real instance class for each combination of parameters L and $|\mathcal{T}|$. The distribution of the results is shown in Appendix B (Figure 13.)

FTS, $|\mathcal{T}| = 20$ for RTS, $|\mathcal{T}| = 10$ for the real instances, and $L \in \{20, 50, 100\}$ for all three data types. We ran ML ten times for each threshold $t \in \{0, 0.1, 0.3, 0.5, 0.7\}$ on each instance, took the lowest reticulation number, and averaged these results within each instance group.

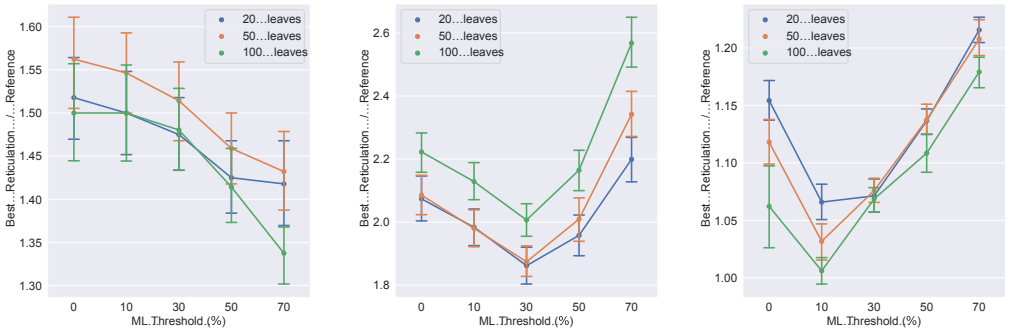
The results are shown in Figure 11. For all types of data a low threshold $t = 0.1$ is beneficial, intuitively indicating that when the probability of a pair being reducible is close to zero it gives no meaningful indication, and thus random choices among these pairs are more suited. For all but one instance group (5 real data instances each consisting of 10 trees with 100 leaves) this is true up to $t = 0.3$, and for all the synthetic datasets even up to $t = 0.5$. The seemingly best value for the threshold, though, is different for each type of instances.

The FTS instances seem to benefit from quite high values of t, the best being the highest tested value, $t = 0.7$. This is consistent with what we observed before: some randomness in the methods seems to be very effective. For the synthetic but less informative RTS instances, the best threshold seems to be around $t = 0.3$, while very high values ($t = 0.7$) are counterproductive. This is also coherent with the fact that the randomised heuristics are less effective on this types of instances. For real data, the best option seems to be around $t = 0.1$.

These experiments should be seen as an indication that the use of an appropriate threshold may improve the performance of the machine-learned heuristics, at the price of running them multiple times. We defer a more thorough analysis to future extensions of this work.

6 Discussion

Our contributions are twofold: first, we presented the first methods that allow reconstructing a phylogenetic network from a large set of large binary phylogenetic trees. Second, we show the promise of the use of machine learning in this context. Our experimental studies indicate that repeated runs of simple and fast randomised cherry-picking heuristics are quite effective in most of the cases, but machine-learned strategies bring better results in the most difficult instances. Furthermore, preliminary experiments indicate that their performance can even be improved by introducing appropriate thresholds, in fact mediating between random choices and predictions. In addition, our current implementations are in Python and hence not optimised for speed, but faster implementations could make machine-learned heuristics with nonzero thresholds even more effective.



(a) FTS Instances. **(b)** RTS Instances. **(c)** Real Instances.

Figure 11 The reticulation number when running ML with different thresholds for the three instances classes. Each instance (in total 112) is run 10 times, where the lowest reticulation value of these runs is selected. At each point, a 68% confidence interval is also shown.

For future work we would like to evaluate if there is any relationship between the accuracy of the machine-learned models and the probability of the heuristic being able to reconstruct an optimal network. We will also investigate if the use of other parameters to introduce some randomness in strategies using machine learning can further improve the results. Finally, we plan to test our methods on trees with different leaf sets and to extend them to nonbinary trees.

References

1 Benjamin Albrecht. Computing all hybridization networks for multiple binary phylogenetic input trees. *BMC bioinformatics*, 16(1):1–15, 2015.

2 Dana Azouri, Shiran Abadi, Yishay Mansour, Itay Mayrose, and Tal Pupko. Harnessing machine learning to guide phylogenetic-tree search algorithms. *Nature communications*, 12(1):1–9, 2021.

3 Robert G Beiko. Telling the whole story in a 10,000-genome world. *Biology Direct*, 6(1):1–36, 2011.

4 Magnus Bordewich and Charles Semple. Computing the minimum number of hybridization events for a consistent evolutionary history. *Discrete Applied Mathematics*, 155(8):914–928, 2007.

5 Sander Borst, Leo van Iersel, Mark Jones, and Steven Kelk. New FPT algorithms for finding the temporal hybridization number for sets of phylogenetic trees. *Algorithmica*, 2022.

6 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

7 Peter J Humphries, Simone Linz, and Charles Semple. Cherry picking: a characterization of the temporal hybridization number for a set of phylogenies. *Bulletin of mathematical biology*, 75(10):1879–1890, 2013.

8 Remie Janssen and Yukihiro Murakami. On cherry-picking and network containment. *Theoretical Computer Science*, 856:121–150, 2021.

9 Sudhir Kumar and Sudip Sharma. Evolutionary sparse learning for phylogenomics. *Molecular Biology and Evolution*, 38(11):4674–4682, 2021.

10 Simone Linz and Charles Semple. Attaching leaves and picking cherries to characterise the hybridisation number for a set of phylogenies. *Advances in Applied Mathematics*, 105:102–129, 2019.

11 Sajad Mirzaei and Yufeng Wu. Fast construction of near parsimonious hybridization networks for multiple phylogenetic trees. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 13(3):565–570, 2015.

12 Fabio Pardi and Celine Scornavacca. Reconstructible phylogenetic networks: do not distinguish the indistinguishable. *PLoS computational biology*, 11(4):e1004135, 2015.

13 F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

14 Joan Carles Pons, Celine Scornavacca, and Gabriel Cardona. Generation of level-k LGT networks. *IEEE/ACM transactions on computational biology and bioinformatics*, 17(1):158–164, 2019.

15 Charles Semple and Gerry Toft. Trinets encode orchard phylogenetic networks. *Journal of Mathematical Biology*, 83(3):1–20, 2021.

16 Claudia Solís-Lemus, Paul Bastide, and Cécile Ané. Phylonetworks: a package for phylogenetic networks. *Molecular biology and evolution*, 34(12):3292–3298, 2017.

17 Leo van Iersel, Remie Janssen, Mark Jones, and Yukihiro Murakami. Orchard networks are trees with additional horizontal arcs. *Bulletin of Mathematical Biology*, 2022. to appear.

18 Leo van Iersel, Remie Janssen, Mark Jones, Yukihiro Murakami, and Norbert Zeh. A unifying characterization of tree-based networks and orchard networks using cherry covers. *Advances in Applied Mathematics*, 129:102222, 2021. doi:10.1016/j.aam.2021.102222.

19 Leo van Iersel, Remie Janssen, Mark Jones, Yukihiro Murakami, and Norbert Zeh. A practical fixed-parameter algorithm for constructing tree-child networks from multiple binary trees. *Algorithmica*, 84:917–960, 2022.

20 Dingqiao Wen, Yun Yu, Jiafan Zhu, and Luay Nakhleh. Inferring phylogenetic networks using phylonet. *Systematic biology*, 67(4):735–740, 2018.

21 Yufeng Wu. Close lower and upper bounds for the minimum reticulate network of multiple phylogenetic trees. *Bioinformatics*, 26(12):i140–i148, 2010.

22 Tujin Zhu and Yunpeng Cai. Applying neural network to reconstruction of phylogenetic tree. In *ICMLC 2021: 13th International Conference on Machine Learning and Computing, Shenzhen China, 26 February, 2021- 1 March, 2021*, pages 146–152. ACM, 2021. doi:10.1145/3457682.3457704.

A Omitted Pseudocode and Proofs

Algorithm 2 summarises a procedure to complete a partial CPS S to obtain a CPS S'.

Algorithm 2 CompleteSeq.

INPUT: A partial CPS $S = (x_1, y_1), \ldots, (x_n, y_n)$ that reduces $\mathcal{T}$
OUTPUT: A CPS S' for $\mathcal{T}$.

$C \leftarrow \emptyset$; $P \leftarrow \emptyset$;
for $i = n, \ldots, 1$ **do**
  if $y_i \notin C$ **then**
    $P \leftarrow P \cup \{y_i\}$;
  $C \leftarrow C \cup \{x_i, y_i\}$;
$S' \leftarrow S$;
while $|P| > 1$ **do**
  Let r_1 and r_2 be two arbitrary elements of P;
  $S' \leftarrow S' \circ (r_1, r_2)$;
  $P \leftarrow P \setminus \{r_1\}$;
return S'

▶ **Lemma 9.** *The running time of a naive implementation of* Rand *is* $\mathcal{O}(|\mathcal{T}|^2|X|)$.

Proof. An upper bound for the length of the sequence is $(|X|-1)|\mathcal{T}|$ as each tree can individually be fully reduced using at most $|X|-1$ pairs. Hence, the while loop of Algorithm 1 is executed at most $(|X|-1)|\mathcal{T}|$ times. Furthermore, choosing a random reducible pair takes $\mathcal{O}(1)$ time, and reducing a pair in all trees in $\mathcal{T}$ takes $\mathcal{O}(|\mathcal{T}|)$ time, as it takes constant time within each tree. Combining this with the fact that CompleteSeq takes $\mathcal{O}(|S|) = \mathcal{O}(|X||\mathcal{T}|)$ time, we conclude that Rand takes no more than $\mathcal{O}(|\mathcal{T}|^2|X|)$ time. ◀

B Runtimes and Omitted Plots

Table 3 Runtimes (in seconds) of ML for instance types FTS, RTS, and real, for all instance groups and for different parameter combinations. The number of leaves per tree are given in column "L". For FTS instances, the size of the tree set is determined by R (reticulation number of the original network). For RFT and real instances, this is simply denoted by $|\mathcal{T}|$.

L	FTS – R			RTS – $\lvert\mathcal{T}\rvert$			Real – $\lvert\mathcal{T}\rvert$			
	5	6	7	20	50	100	10	20	50	100
20	18.86	32.80	63.23	21.60	49.10	87.97	55.24	96.49	216.15	419.84
50	62.28	115.78	227.26	55.74	125.80	251.10	65.80	120.33	303.40	649.32
100	204.20	375.73	744.92	146.45	338.48	661.54	98.84	172.26	407.26	-

Table 4 Runtimes of CPH (same for ML and TrivialRand), Hybroscale, and TreeChild in seconds, for combinations of parameters L (leaves per tree) and $|\mathcal{T}|$ (number of trees).

L	$\lvert\mathcal{T}\rvert$	CPH	Hybroscale	TreeChild
10	2	3.404	0.211	0.009
	3	3.668	6.206	0.008
	4	4.844	251.880	0.073
	5	5.308	-	0.556
	6	5.851	-	4.084
	7	6.762	-	12.870
	8	7.350	-	249.799
20	2	5.591	0.248	0.008
	3	7.194	7.991	0.021
	4	8.897	-	0.943
	5	9.604	-	0.858
30	2	8.373	0.393	0.025
	3	10.348	114.656	0.173
40	2	9.276	0.365	0.016
50	2	11.794	0.729	0.038
60	2	15.043	-	463.171
80	2	19.926	-	1372.930

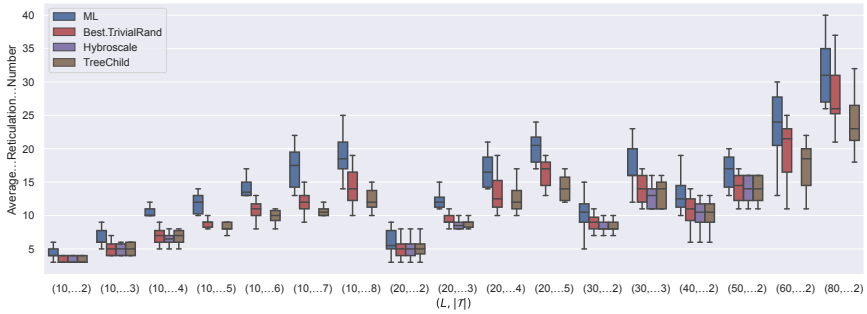


Figure 12 Boxplots for the distribution of the results of Figure 9.

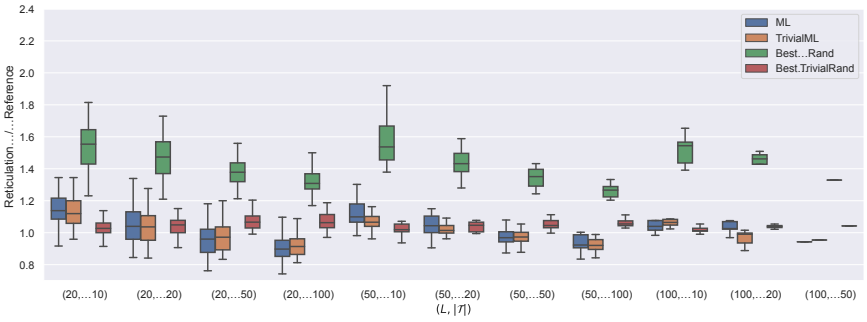


Figure 13 Boxplot for the distribution of the results of Figure 10.

C Random Forest Models

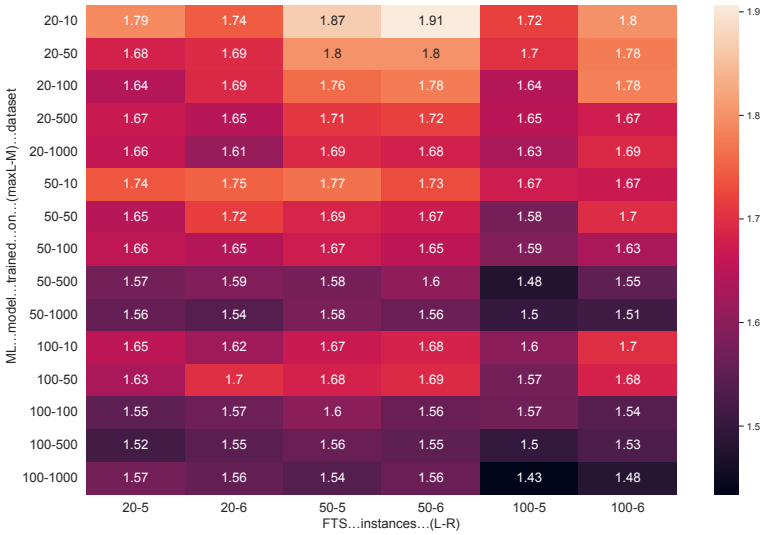
Table 5 Trained random forest models on different datasets for different combinations of max L (maximum number of leaves per network) and M (number of total networks). Each row in the table represents one model. For each model, the testing accuracy is given under "Accuracy", the total number of data points retrieved from all M networks is given under "Num. data". Each dataset is split for training and testing ($90\% - 10\%$). The training duration for the random forest is given in column "Train time" and the time needed to generate the training data is given in column "Datagen time", in hours per core (we used 16 cores in total).

max L	M	Accuracy	Num. data	Train time (min)	Datagen time (hour/core)
20	10	0.968	1,896	00:00	00:00:48
	50	0.984	6,312	00:01	00:02:23
	100	0.983	12,060	00:02	00:04:13
	500	0.983	59,236	00:13	00:23:33
	1000	0.979	114,268	00:27	00:45:19
50	10	0.986	3,556	00:00	00:03:04
	50	0.995	21,228	00:02	00:17:11
	100	0.992	44,136	00:05	00:37:52
	500	0.995	218,256	00:49	02:26:56
	1000	0.995	417,624	02:36	04:29:13
100	10	0.997	14,224	00:02	00:19:19
	50	0.997	42,628	00:07	00:55:42
	100	0.998	105,540	00:22	01:58:03
	500	0.998	529,708	03:59	08:35:33
	1000	0.998	1,098,280	08:09	16:48:45

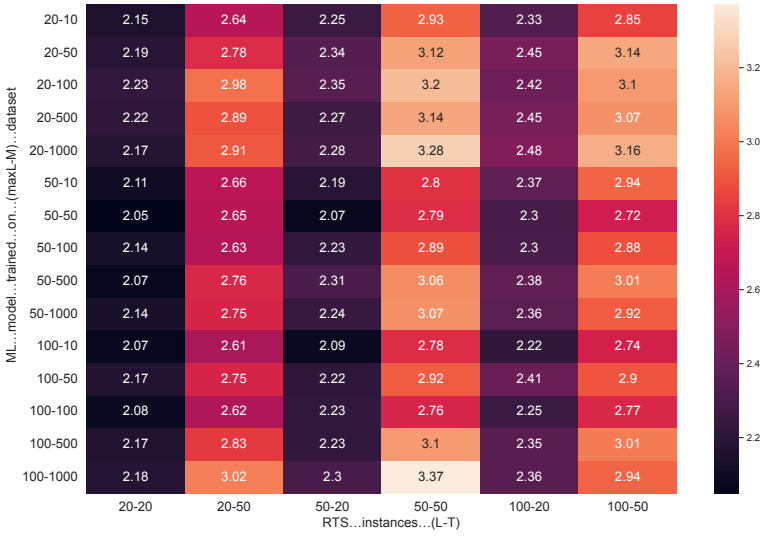
Table 6 Feature importances of random forest trained on the biggest dataset ($M = 1000$ and max $L = 100$). A higher "Importance" corresponds to that feature having more effect on the trained model. The values sum up to one. The descriptions of the features are given in Table 1.

Features		Importance
Trivial		0.183
Leaf distance	(t)	0.158
Cherry in tree		0.152
Leaf distance	(d)	0.127
Depth x/y	(d)	0.095
LCA distance	(t)	0.050
Depth x/y	(t)	0.049
LCA distance	(d)	0.036
Cherry depth	(t)	0.030
Leaf depth y	(t)	0.018
Leaf depth x	(t)	0.017
Leaf depth y	(d)	0.015
Leaf depth x	(d)	0.015
Cherry depth	(d)	0.014
New cherries		0.013
Tree depth	(d)	0.011
Tree depth	(t)	0.009
Before/after		0.005
Leaves in tree		0.003

D Heuristic Performance of ML Models



(a) FTS Instances.



(b) RTS Instances.

Figure 14 Results for ML with the random forest model trained on each of the datasets given in Table 5. For each training dataset, identified by the parameter pair max L-M, the value shown in the heatmap is the average, within each instance group, of the reticulation number found by ML divided by the reference value. We used a group of 112 instances for each combination of parameters $L \in \{20, 50, 100\}$ and $R \in \{5, 6\}$ (for FTS), and $L \in \{20, 50, 100\}$ and $|\mathcal{T}| \in \{20, 50\}$ (for RTS).

Feasibility of Flow Decomposition with Subpath Constraints in Linear Time

Daniel Gibney[1]
Georgia Institute of Technology, Atlanta, GA, USA

Sharma V. Thankachan
North Carolina State University, Raleigh, NC, USA

Srinivas Aluru
Georgia Institute of Technology, Atlanta, GA, USA

Abstract

The decomposition of flow-networks is an essential part of many transcriptome assembly algorithms used in Computational Biology. The addition of subpath constraints to this decomposition appeared recently as an effective way to incorporate longer, already known, portions of the transcript. The problem is defined as follows: given a weakly connected directed acyclic flow network $G = (V, E, f)$ and a set $\mathcal{R}$ of subpaths in G, find a flow decomposition so that every subpath in $\mathcal{R}$ is included in some flow in the decomposition [Williams et al., WABI 2021]. The authors of that work presented an exponential time algorithm for determining the feasibility of such a flow decomposition, and more recently presented an $O(|E| + L + |\mathcal{R}|^3)$ time algorithm, where L is the sum of the path lengths in $\mathcal{R}$ [Williams et al., TCBB 2022]. Our work provides an improved, linear $O(|E| + L)$ time algorithm for determining the feasibility of such a flow decomposition. We also introduce two natural optimization variants of the feasibility problem: (i) determining the minimum sized subset of $\mathcal{R}$ that must be removed to make a flow decomposition feasible, and (ii) determining the maximum sized subset of $\mathcal{R}$ that can be maintained while making a flow decomposition feasible. We show that, under the assumption P $\neq$ NP, (i) does not admit a polynomial-time $o(\log |V|)$-approximation algorithm and (ii) does not admit a polynomial-time $O(|V|^{\frac{1}{2}-\varepsilon} + |\mathcal{R}|^{1-\varepsilon})$-approximation algorithm for any constant $\varepsilon > 0$.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Flow networks, flow decomposition, subpath constraints

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.17

Funding This research is supported in part by the U.S. National Science Foundation (NSF) grants CCF-1704552, CCF-1816027 and CCF-2146003.

1 Introduction

The decomposition of flow networks is both a classical problem in Computer Science and an important tool in Computational Biology, where it is used for transcriptome assembly [2, 8, 13, 15, 16, 17, 21]. It was in this last context that the idea of adding subpath constraints to the decomposition of flow networks arose. These constraints allow one to incorporate longer, known portions of a transcript by enforcing that every path in a provided subpath constraint set is a subpath in the flow decomposition (details given in Section 1.2). Williams *et al.* defined the problem of determining whether a flow decomposition under a set of subpath constraints is feasible and, if so, finding a flow decomposition with the minimum number of paths [21, 22]. Minimizing the size of a flow decomposition under subpath constraints is NP-hard due to the same problem being NP-hard without added constraints, even on

[1] Corresponding author

DAGs [19]. For determining the feasibility of a flow decomposition, Williams *et al.* in [21] presented an algorithm running in time exponential in the total overdemand, where the total overdemand is a quantity related to the flows assigned to every edge and the number of subpath constraints. They also posed the question of whether the problem can be solved in polynomial time. The same authors recently answered this question by providing an algorithm that runs in time $O(|E| + L + |\mathcal{R}|^3)$ on a flow network with $|V|$ vertices, $|E|$ edges, and a subpath constraint set $\mathcal{R}$ where the total sum of the subpath lengths is L [22]. A primary contribution of this work is an algorithm that solves the feasibility problem with an improved, linear time complexity that is $O(|E| + L)$.

We will also consider two optimization versions of this problem: minimizing the number of constraints removed while making a flow decomposition feasible and, its dual, maximizing the number of constraints maintained while making a flow decomposition feasible. When applied to transcriptome assembly, the vertices are often used to represent k-mers and the flows assigned to edges correspond to levels of support for the k-mers being adjacent (frequency of the corresponding $k+1$-mers). The paths found through flow decomposition are interpreted as assembled transcripts. As mentioned above, the idea of adding subpath constraints to the flow decomposition comes from trying to incorporate knowledge of longer portions of transcripts [1, 7, 23]. In the case where a flow decomposition under the constraints is not feasible, a natural next step is to eliminate a small number of these constraints in order to make a flow decomposition possible. This would correspond to rejecting some subset of the longer transcripts used for subpath constraints as being either erroneous or not present in the transcripts that caused the given flow network.

Our work studies the inapproximability of these optimization problems. For the problem of removing a minimal number of subpath constraints, we show that a polynomial-time $o(\log |V|)$-approximation algorithm does not exist under the assumption that P $\neq$ NP. We then show that the minimization problem's dual, maximizing the number of subpath constraints maintained, is more difficult to approximate. For the maximization version, there does not exist a polynomial-time approximation algorithm with an approximation factor that is $O(|\mathcal{R}|^{1-\varepsilon} + |V|^{\frac{1}{2}-\varepsilon})$ for any constant $\varepsilon > 0$, assuming P $\neq$ NP.

1.1 Related work

Research on flow decomposition under subpath constraints began with work on finding a minimum path cover (MPC) under subpath constraints. MPC with subpath constraints was initially introduced by Bao et al. as a tool for RNA-sequence assembly [1]. It received further study by Rizzi et al., who considered different weighted cases of the problem and provided polynomial-time algorithms, as well as hardness results for some variations of the problem [14]. Subsequent work has applied these algorithms to assembly [7]. A generalization from path cover to flow decomposition was introduced by Williams et al. [22]. There, the authors observed the deficiency of path decomposition in terms of failing to incorporate frequency information. They propose flow networks and flow decomposition with the minimum number of paths as an improvement. To circumvent this problem's NP-hardness (even without path constraints [19]), the authors provide an FPT algorithm and heuristics as potential solutions.

In our work, we will not be concerned with the problem of finding a minimum number of paths, but rather with the feasibility problem. By avoiding the minimality condition, the problem becomes computationally tractable. Another recent line of related research that avoids this optimization criterion and leads to a polynomial-time solvable problem is finding paths in flow networks that are "safe". These paths are safe in that they appear as subpaths in every flow decomposition. Such safe paths can be enumerated in polynomial time [5, 6, 10].

1.2 Preliminaries

Like in [21, 22], we will only work with directed acyclic graphs (DAGs) in this paper. The reasons for this stem from its application to transcriptome assembly, where the flow networks constructed from a reference genome are DAGs. We define flow networks accordingly. Also, the definition of a flow decomposition here differs from the standard definition in several ways, including that there is no capacity function and that flows are given as part of the flow network itself rather than the decomposition. However, for consistency with [21, 22] we will maintain this terminology.

▶ **Definition 1** (Flow network). *A flow network $G = (V, E, f)$ consists of a (weakly) connected DAG $G = (V, E)$ where V contains special source and sink vertices s and t, and a function $f : E \to \mathbb{N}$ called the flow. The flow satisfies that for all $v \in V \setminus \{s, t\}$, $\sum_{(u,v)\in E} f((u, v)) = \sum_{(v,w)\in E} f((v, w))$. In addition, vertex s has in-degree 0 and vertex t has out-degree 0.*

For an edge $e \in E$, we will refer to $f(e)$ as the flow assigned to e.

▶ **Definition 2** (Flow decomposition). *A flow decomposition $(\mathcal{P}, w)$ of a flow network G is a set of st-paths $\mathcal{P} = \{P_1, ..., P_{|\mathcal{P}|}\}$ and natural numbers $w = \{w_1, ..., w_{|\mathcal{P}|}\}$, called weights, such that if $e \in E$ is contained exactly in the paths $\{P_{i_1}, ..., P_{i_j}\} \subseteq \mathcal{P}$, then $\sum_{h=1}^{j} w_{i_h} = f(e)$.*

▶ **Definition 3** (Subpath Constraints). *A set of subpath constraints $\mathcal{R} = \{R_1, ..., R_{|\mathcal{R}|}\}$ for a given flow network G is a set of simple paths in G such that for all distinct $R_i, R_j \in \mathcal{R}$, we have R_i is not a subpath of R_j.*

The requirement that no subpath constraint is a subpath of another subpath constraint is a property that will be used in the proof that our linear time algorithm is correct.

▶ **Problem 4** (Flow Decomposition with Subpath Constraints (FDSC)). *An instance $(G, \mathcal{R})$ of FDSC consists of a flow network G and a set of subpath constraints $\mathcal{R}$. The problem is to determine if there exists a flow decomposition $(\mathcal{P}, w)$ of G such that for all subpath constraints $R \in \mathcal{R}$, there exists a path $P \in \mathcal{P}$ where R is a subpath of P.*

If such a flow decomposition exists, we say the instance of FDSC is *feasible*, and we otherwise say it is *infeasible*.

Two optimization variants of this problem are defined below. We consider a solution to the minimization problem to have as the objective value the number of subpath constraints removed from $\mathcal{R}$, and a solution to the maximization problem as the number of subpath constraints maintained.

▶ **Problem 5** (Minimum Subpath Constraint Removal). *Given an instance of FDSC $(G, \mathcal{R})$ and integer $k \geq 0$, determine if there exists a subset of $\mathcal{R}' \subseteq \mathcal{R}$, such that $|\mathcal{R}'| \leq k$ and $(G, \mathcal{R} \setminus \mathcal{R}')$ is feasible.*

▶ **Problem 6** (Maximum Subpath Constraint Retention). *Given an instance of FDSC $(G, \mathcal{R})$ and integer $k \geq 0$, determine if there exists a subset of $\mathcal{R}' \subseteq \mathcal{R}$, such that $|\mathcal{R}'| \geq k$ and $(G, \mathcal{R}')$ is feasible.*

The following definitions will be used in the remainder of this work.

▶ **Definition 7** (Union of Two Paths). *We say two paths $v_{i_1}v_{i_2}...v_{i_k}$ and $v_{j_1}v_{j_2}...v_{j_{k'}}$ can be unioned if there exists suffix-prefix overlap between $v_{i_1}v_{i_2}...v_{i_k}$ and $v_{j_1}v_{j_2}...v_{j_{k'}}$ or between $v_{j_1}v_{j_2}...v_{j_{k'}}$ and $v_{i_1}v_{i_2}...v_{i_k}$ when viewed as strings, i.e., there exists indices i_h and j_g such that either (i) $v_{i_h}v_{i_{h+1}}...v_{i_k} = v_{j_1}v_{j_2}...v_{j_g}$ or (ii) $v_{i_1}v_{i_2}...v_{i_h} = v_{j_g}v_{j_{g+1}}...v_{j_{k'}}$. The union of these two paths in case (i) is $v_{i_1}...v_{i_{h-1}}v_{j_1}...v_{j_{k'}}$, and in case (ii) is $v_{j_1}...v_{j_{g-1}}v_{i_1}...v_{i_k}$. We use $R_i \cup R_j$ to denote the union of subpath constraints R_i and R_j.*

▶ **Definition 8** (Compatible Subpath Constraints). *We say that two subpath constraints R_i and R_j are compatible if R_i and R_j are either vertex disjoint or can be unioned. This is denoted as $R_i \sim R_j$. If two subpath constraints R_i and R_j are not compatible, they are considered incompatible and this is denoted as $R_i \not\sim R_j$.*

Note that by our definition of unioning and compatible subpath constraints, we cannot union two subpath constraints R_i and R_j if they are vertex disjoint, despite them being compatible.

We refer to a set of subpath constraints that can be formed by starting with $\mathcal{R}$ and repeatedly applying zero or more unions as a *unioning of* $\mathcal{R}$. For a particular unioning of $\mathcal{R}$, say $\mathcal{R}'$, and edge $e \in E$, we let $|\mathcal{R}'(e)|$ denote the number of subpath constraints in $\mathcal{R}'$ containing e. We say a subpath constraint set $\mathcal{R}'$ is *maximally unioned* if for all $R_i, R_j \in \mathcal{R}'$, R_i cannot be unioned with R_j. Lastly, we define the length of a path as the number of edges contained in the path.

2 Linear Time Algorithm for FDSC

Our approach will be different from the one taken in [22], although both will result in greedy algorithms. In that work, a graph known as the constraint graph is constructed from the given instance of FDSC. The construction of the constraint graph causes an increased time complexity relative to the algorithm presented here. Also, on the constraint graph, it is possible to derive an equivalent result to Lemma 12, albeit in terms of a path cover on the constraint graph (Lemma 16 in [22]). In the interest of avoiding having to prove that desired equivalence holds outside of the language of the constraint graph, we provide our own proof.

2.1 Equivalence with Finding a Satisfactory Unioning of Constraints

Lemmas 9 - 12 essentially reduce FDSC to the problem of finding an satisfactory unioning of the subpath constraint set.

▶ **Lemma 9** ([21], Corollary 9). *For an FDSC instance $(G, \mathcal{R})$, if for all $e \in E$, $|\mathcal{R}(e)| \leq f(e)$, then $(G, \mathcal{R})$ is feasible.*

▶ **Lemma 10.** *If an instance of FDSC $(G, \mathcal{R})$ is feasible, then there exists a feasible instance of FDSC $(G, \mathcal{R}')$ where $\mathcal{R}'$ is a maximal unioning of $\mathcal{R}$.*

Proof. We can consider the flow decomposition $(\mathcal{P}, w)$ for $(G, \mathcal{R})$ as consisting of (possibly overlapping[2]) st-paths all of weight 1. For duplicate paths we only work with one representative path. For every subpath constraint $R \in \mathcal{R}$, we arbitrarily assign R to one of the representative st-paths $P \in \mathcal{P}$ in the flow decomposition that contains R as a subpath and say that P *satisfies* R.

We first iterate through the representative st-paths in $\mathcal{P}$ in any order. For each representative path, $P \in \mathcal{P}$ we union all non-disjoint subpath constraints satisfied by P. Let the resulting unioning of $\mathcal{R}$ be denoted as $\mathcal{R}'$. We next repeat the following steps until the set of subpath constraints is maximally unioned:

Suppose there exists two compatible, non-disjoint subpath constraints $R_i, R_j \in \mathcal{R}'$ that are not unioned, with the prefix of R_j overlapping with the suffix of R_i. See Figure 1. Let R_i be satisfied by path $P_1 \in \mathcal{P}$ and R_j be satisfied by path $P_2 \in \mathcal{P}$, both having weight 1.

[2] This is permitted under the definition of flow decomposition.

Figure 1 The procedure for swapping portions of paths described in the proof of Lemma 10.

Then there exists some vertex v in common between R_i, R_j, P_1, and P_2. Let $P_1[x, y]$ be the portion of P_1 from vertex x to vertex y (both inclusive) and $P_2[x, y]$ be defined similarly. We next,

1. Remove the paths P_1 and P_2 from $\mathcal{P}$;
2. If not already existing, add the paths $P'_1 = P_1[s, v] \circ P_2[v, t]$ and $P'_2 = P_2[s, v] \circ P_1[v, t]$ to $\mathcal{P}$, where $\circ$ applies concatenation (including v only once);
3. Union R_i and R_j and assign $R_i \cup R_j$ as being satisfied by the representative for P'_1. Additionally, any subpath constraints satisfied by $P_1[s, v]$ or $P_2[v, t]$ are assigned to the representative for P'_1 and any subpath constraints satisfied by $P_2[s, v]$ or $P_1[v, t]$ are assigned to the representative for P'_2 .

For all edges, the sums of path weights are not modified since both P_1 and P_2 had weight 1. Furthermore, other subpath constraints remain satisfied. This is since if a subpath constraint contained v and was being satisfied by either P_1 or P_2, it would have been unioned with R_i or R_j respectively when forming $\mathcal{R}'$. It also still holds that all non-disjoint subpath constraints that are satisfied by the same representative st-path in the decomposition are unioned. Hence, we can repeat Steps 1-3 until the resulting set of subpath constraints is maximally unioned. ◀

▶ **Lemma 11.** *If $\mathcal{R}$ is a maximally unioned subpath constraint set, then $(G, \mathcal{R})$ is infeasible iff there exists an edge $e \in E$ such that $|\mathcal{R}(e)| > f(e)$.*

Proof. First consider when $|\mathcal{R}(e)| > f(e)$. Since $\mathcal{R}$ is maximally unioned, every subpath constraint in $\mathcal{R}(e)$ must be satisfied by a distinct path. This implies there must be at least $|\mathcal{R}(e)|$ distinct paths containing e, each with weight at least 1, making the sum of these exceed $f(e)$. Hence, $(G, \mathcal{R})$ is infeasible. In the other direction, if for all $e \in E$, $|\mathcal{R}(e)| \leq f(e)$, then we can directly apply Lemma 9 to obtain that a flow decomposition exists that satisfies the subpath constraints in $\mathcal{R}$. ◀

The main result for this section is Lemma 12.

▶ **Lemma 12.** *For a given FDSC instance $(G, \mathcal{R})$, there exists a flow decomposition satisfying $\mathcal{R}$ iff there exists a unioning $\mathcal{R}'$ of $\mathcal{R}$ where $|\mathcal{R}'(e)| \leq f(e)$ for all $e \in E$.*

Proof. If there exists a flow decomposition satisfying $\mathcal{R}$, then, by Lemma 10, there exists a feasible instance $(G, \mathcal{R}')$ where $\mathcal{R}'$ is maximally unioned. By Lemma 11 we have that $|\mathcal{R}'(e)| \leq f(e)$ for all $e \in E$. In the other direction, if there exists a unioning of $\mathcal{R}$, say $\mathcal{R}'$, such that $|\mathcal{R}'(e)| \leq f(e)$ for all $e \in E$, then, by Lemma 9, there exists a valid flow decomposition for $(G, \mathcal{R}')$. Since the same flow decomposition used for $(G, \mathcal{R}')$ satisfies all of the constraints in $\mathcal{R}$, the instance $(G, \mathcal{R})$ must be feasible as well. ◀

As the first step in determining whether such a unioning exists, we topologically sort the vertices in V. We will henceforth consider the vertices v_1, v_2, ..., $v_{|V|}$ to be in sorted order from smallest to largest. A given constraint $R_i \in \mathcal{R}$ can now be written as $R_i = v_{i_1} v_{i_2} ... v_{i_{|R_i|}}$

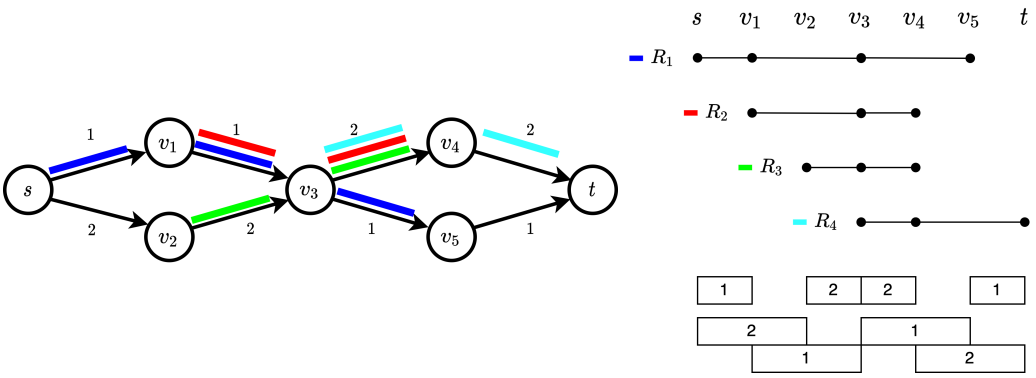


Figure 2 (Left) A flow network with four subpath constraints: $R_1 = s\,v_1\,v_3\,v_5$ (dark blue), $R_2 = v_1\,v_3\,v_4$ (red), $R_3 = v_2\,v_3\,v_4$ (green), and $R_4 = v_3\,v_4\,t$ (light blue). Flows for each edge are indicated by the numbers adjacent to the edge. (Right) The same subpath constraints and edge flows, but in the visualization style used in this work.

where these vertices are also considered to be in sorted order from smallest to largest. Using this ordering we can compare the max and min values of a subpath constraint R_i to the max and min values of another subpath constraint R_j.

We can also now more easily visualize an instance of FDSC. An example of this visual representation is shown in Figure 2. This is done by creating for each subpath constraint $R \in \mathcal{R}$ a set of $|R|$ horizontal dots, each positioned horizontally so as to correspond to vertex indices on the number line. For every subpath constraint, we connect each of its dots by a line. In modeling our problem we fix the horizontal position of each set of dots and lines formed from R, but allow their vertical position to be simultaneously shifted. Two non-disjoint constraints can be unioned if we can place one subpath constraint representation on top of the other without a line being placed on top of a dot (or vice versa). At the bottom of this representation we indicate the flow $f((v_\alpha, v_\beta))$ with a box spanning the interval $[\alpha, \beta]$ and containing $f((v_\alpha, v_\beta))$. We can now recast the feasibility problem as trying to find a way to vertically shift these subpath constraint representations on top of one another such that for every interval $[\alpha, \beta]$ the number of lines spanning it is at most $f((v_\alpha, v_\beta))$. This problem representation removes unnecessary information present in the original FDSC instance, such as edges not containing any subpath constraints. In particular, this makes the reduction used in Section 3.1 easier to visualize.

2.2 Our Greedy Algorithm

We first present a high-level overview of the algorithm that forms a subpath constraint set $\mathcal{R}_G$ by unioning subpath constraints in $\mathcal{R}$. As such, we will denote each subpath constraint in $\mathcal{R}_G$ as a subset of the subpath constraints in $\mathcal{R}$, implying that this subset is to be unioned to form a new subpath constraint. This is merely a notational convenience, for example, we denote the unioned subpath constraints $R_{i_1} \cup R_{i_2} \cup ... \cup R_{i_k}$ as $\{R_{i_1}, R_{i_2}, ..., R_{i_k}\}$. These subsets of $\mathcal{R}$ are disjoint (no subpath constraint appears in two sets) and only contain subpath constraints that are compatible.

Our greedy algorithm *simultaneously minimizes* the number of subpath constraints in $\mathcal{R}_G$ containing any given edge, across all edges. As a result, to determine feasibility, we only need to check for every edge $e \in E$ the number of subpath constraints in $\mathcal{R}_G$ containing e. Implementation details are given in Section 2.4.

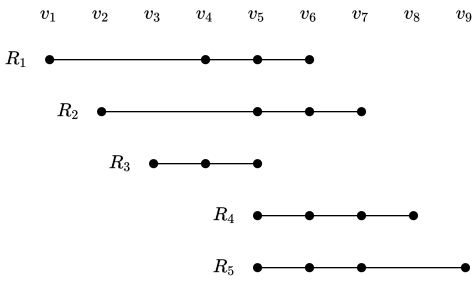


Figure 3 An example sorted set of subpath constraints $\mathcal{R} = \{R_1, R_2, R_3, R_4, R_5\}$ where $R_1 = v_1\, v_4\, v_5\, v_6$, $R_2 = v_2\, v_5\, v_6\, v_7$, $R_3 = v_3\, v_4\, v_5$, $R_4 = v_5\, v_6\, v_7\, v_8$, $R_5 = v_5\, v_6 v_7\, v_9$.

High-Level Algorithm

Recall that all vertices are topologically sorted and indexed according to this order. Sort $\mathcal{R}$ in ascending order according to each subpath constraint's minimum vertex with ties broken arbitrarily. Let R_1, R_2, ..., $R_{|\mathcal{R}|}$ denote the subpath constraints in $\mathcal{R}$ in sorted order from smallest to largest. We next create a new subpath constraint set $\mathcal{R}_G$ that is initially empty. Processing $\mathcal{R}$ in this sorted order from smallest to largest, on iteration i, let

$$\mathcal{R}_{G|\sim i} = \{R \in \mathcal{R}_G : R \sim R_i \text{ and } R \cap R_i \neq \emptyset\}.$$

If $\mathcal{R}_{G|\sim i}$ is non-empty, we update $\mathcal{R}_G$ by unioning R_i with a $R_j \in \mathcal{R}_{G|\sim i}$ that has the largest last vertex, i.e. $j \in \arg\max_h \{\max R_h : R_h \in \mathcal{R}_{G|\sim i}\}$ where $\max R_h$ returns the largest index of any vertex in R_h. If $\mathcal{R}_{G|\sim i}$ is empty, then we add $\{R_i\}$ to $\mathcal{R}_G$.

As an example, consider the subpath constraints $\mathcal{R}$ shown in Figure 3. We create an initially empty set of subpath constraints $\mathcal{R}_G = \emptyset$.

- On iteration $i = 1$, we make $\mathcal{R}_G = \{\{R_1\}\}$.
- On iteration $i = 2$, we make $\mathcal{R}_G = \{\{R_1\}, \{R_2\}\}$.
- On iteration $i = 3$, we make $\mathcal{R}_G = \{\{R_1\}, \{R_2\}, \{R_3\}\}$.
- On iteration $i = 4$, we have to decide whether to union R_4 with $\{R_1\}$, $\{R_2\}$, or $\{R_3\}$. Since $\max\{R_2\} = v_7 > \max\{R_1\} = v_6 > \max\{R_3\} = v_5$, we make $\mathcal{R}_G = \{\{R_1\}, \{R_2, R_4\}, \{R_3\}\}$.
- On iteration $i = 5$, we make $\mathcal{R}_G = \{\{R_1, R_5\}, \{R_2, R_4\}, \{R_3\}\}$.

2.3 Proof of Correctness

Let $\text{OPT}(e)$ denote the minimum number of subpath constraints containing $e \in E$ across all possible ways of unioning $\mathcal{R}$. We claim that after completing all $|\mathcal{R}|$ iterations of the above algorithm, for all edges $e \in E$, we have $|\mathcal{R}_G(e)| = \text{OPT}(e)$. **For the rest of Section 2.3 we fix the edge $e = (v_\alpha, v_\beta)$.** Note that since our greedy algorithm is independent of the choice of e, proving the above claim for e proves it for an arbitrary edge. The proof of the claim involves an exchange argument. For the sake of completeness, we show that an optimal solution can be obtained by iterating over $\mathcal{R}$ in the same order as our greedy algorithm.

▶ **Lemma 13.** *There exists a sequence of unions of subpath constraints in $\mathcal{R}$ that creates a set of subpath constraints $\mathcal{R}'$ such that: (i) $|\mathcal{R}'(e)| = \text{OPT}(e)$, (ii) the process starts with $\mathcal{R}' = \emptyset$ and on the i^{th} iteration, for $i = 1$ to $|\mathcal{R}|$: either unions $R_i \in \mathcal{R}$ with some $R' \in \mathcal{R}'$ created from the previous $i - 1$ iterations, or adds $\{R_i\}$ to $\mathcal{R}'$. Note that this process iterates through $\mathcal{R}$ in the same sorted order as our greedy algorithm.*

Proof. Let $\mathcal{R}''$ be a unioning of $\mathcal{R}$ such that $|\mathcal{R}''(e)| = \text{OPT}(e)$ (not necessarily applying unions in the order described above). Then $\mathcal{R}''$ has a representation of the form $\mathcal{R}'' = \{R''_1, R''_2, ..., R''_k\}$, where each R''_i is the union of some subset of $\mathcal{R}$. We construct a solution as follows: We first create an empty set $\mathcal{R}'$ and process $\mathcal{R}$ in sorted order. When processing R_i, assume that R_i is contained in the subpath constraint $R'' \in \mathcal{R}''$.

- If no previously processed subpath constraint in $\mathcal{R}$ is in R'', we add $\{R_i\}$ to $\mathcal{R}'$.
- If some subset of previously processed subpath constraints in $\mathcal{R}$ are in R'', then we assume inductively that they have already been unioned together, and now union R_i with this partial subset of R''. This is always possible since R_i cannot be vertex disjoint with the partial subset of R''. If it were, then $\min R_i$ would be larger than the maximum vertex in the partial subset of R''. This, together with the sorted order in which $\mathcal{R}$ is being processed, would imply that R'' cannot be a single subpath constraint.

After iterating over all of $\mathcal{R}$, $\mathcal{R}'$ is identical to $\mathcal{R}''$ and the sequence of unions used to obtain it satisfies the stated conditions. ◀

We denote the sequence of unions used in Lemma 13 as the OPT(e)-solution. Let i be the first iteration where our greedy algorithm and the OPT(e)-solution perform a different action on $R_i \in \mathcal{R}$. We will show how to modify the OPT(e)-solution in such a way that it:

- matches our greedy algorithm on the i^{th} iteration;
- does not modify the unions made in earlier iterations;
- has a valid sequence of unions in future iterations;
- does not increase the number of subpath constraints containing e.

Let $\mathcal{R}'$ be the partially constructed set of subpath constraints just prior to the i^{th} iteration resulting from the OPT(e)-solution, and suppose in our greedy algorithm we union R_i with $\{R_{j_1}, ..., R_{j_h}\} \in \mathcal{R}'$ instead of $\{R_{i_1}, ..., R_{i_k}\} \in \mathcal{R}'$ as is done in the OPT(e)-solution. Let the set of subpath constraints unioned with R_i on later iterations in the OPT(e)-solution be denoted as $\{R_{i_{k+1}}, ..., R_{i_t}\}$. Similarly, let the set of subpath constraints in $\mathcal{R}$ unioned with $\{R_{j_1}, ..., R_{j_h}\}$ in later iterations in the OPT(e)-solution be denoted as $\{R_{j_{h+1}}, ..., R_{j_s}\}$. The modification we make to the (completed) OPT(e)-solution is to remove the subpath constraints

$$\{R_{j_1}, ..., R_{j_h}, R_{j_{h+1}}, ..., R_{j_s}\} \text{ and } \{R_{i_1}, ..., R_{i_k}, R_i, R_{i_{k+1}}, ..., R_{i_t}\},$$

then, in the case where $R_{i_k} \cap R_{j_{h+1}} \neq \emptyset$, we add the subpath constraints

$$\{R_{i_1}, ..., R_{i_k}, R_{j_{h+1}}, ..., R_{j_s}\} \text{ and } \{R_{j_1}, ..., R_{j_h}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}.$$

In the case where $R_{i_k} \cap R_{j_{h+1}} = \emptyset$, we instead add

$$\{R_{i_1}, ..., R_{i_k}\}, \{R_{j_{h+1}}, ..., R_{j_s}\}, \text{ and } \{R_{j_1}, ..., R_{j_h}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}.$$

▶ **Lemma 14.** *The modified solution is always valid (no two incompatible subpath constraints are unioned in the modified solution).*

Proof. First, we show that every subpath constraint in the set $\{R_{i_{k+1}}, ..., R_{i_t}\}$ is compatible with every subpath constraint in $\{R_{j_1}, ..., R_{j_h}\}$ (we already know R_i is compatible with $\{R_{j_1}, ..., R_{j_h}\}$ since our greedy algorithm chose it). Suppose for the sake of contradiction that some $R_{j_u} \in \{R_{j_1}, ..., R_{j_h}\}$ is incompatible with some $R_{i_v} \in \{R_{i_{k+1}}, ..., R_{i_t}\}$. See Figure 4 (Left). Because of the sorted order, $\min R_{j_u} \leq \min R_i \leq \min R_{i_v}$. Combining this with

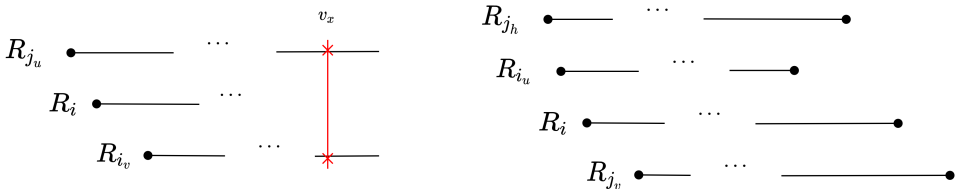


Figure 4 (Left) A visualization of the argument in Lemma 14 that every subpath constraint in $\{R_{i_{k+1}}, ..., R_{i_t}\}$ is compatible with every subpath constraint in $\{R_{j_1}, ..., R_{j_h}\}$. The only relative position that an incompatibility could occur is indicated in red. However, such an incompatibility can not occur since $\max R_i > \max R_{j_u}$. (Right) Every subpath constraint in $\{R_{j_{h+1}}, ..., R_{j_s}\}$ is compatible with every subpath constraint in $\{R_{i_1}, ..., R_{i_k}\}$. Here the relative end positions make an incompatibility impossible as well.

$R_{j_u} \sim R_i$, and $R_i \sim R_{i_v}$, we have that $R_{j_u} \not\sim R_{i_v}$ implies an incompatibility at some vertex v_x where $v_x > \max R_i$ and $v_x < \max R_{j_u}$. However, this implies $\max R_i < \max R_{j_u}$, making $R_i \subset R_{j_u}$, a contradiction.

Next we show that every subpath constraint in the set $\{R_{j_{h+1}}, ..., R_{j_s}\}$ is compatible with every subpath constraint in $\{R_{i_1}, ..., R_{i_k}\}$. Suppose for the sake of contradiction that $R_{i_u} \in \{R_{i_1}, ..., R_{i_k}\}$ is incompatible with $R_{j_v} \in \{R_{j_{h+1}}, ..., R_{j_t}\}$. See Figure 4 (Right). Because our greedy algorithm unions R_i with the subpath constraint having the largest max value, $\max R_{j_h} > \max R_{i_u}$. Also, by the sorted order, we have $\min R_{j_h} < \min R_{j_v}$. Since $R_{j_h} \sim R_{j_v}$, the only way that $R_{i_u} \not\sim R_{j_v}$ is if the incompatibility happens at a vertex greater than $\max R_{j_h}$. However, since $\max R_{j_h} > \max R_{i_u}$, this is not possible. ◀

▶ **Lemma 15.** *The modified solution has at most the same number of subpath constraints containing e as the unmodified* OPT(e)*-solution.*

Proof.

Case 1. $R_{i_k} \cap R_{j_{h+1}} \neq \emptyset$. Recall that the modified solution is

$$\{R_{i_1}, ..., R_{i_k}, R_{j_{h+1}}, ..., R_{j_s}\} \text{ and } \{R_{j_1}, ..., R_{j_h}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}.$$

If both $\{R_{j_1}, ..., R_{j_h}, R_{j_{h+1}}, ..., R_{j_s}\}$ and $\{R_{i_1}, ..., R_{i_k}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}$ contain e in the unmodified OPT(e)-solution, then the number of subpath constraints containing e can not increase in the modified solution. Hence, we only need to consider when the following assumption holds:

▷ Assumption 16. Only one of

$$\{R_{j_1}, ..., R_{j_h}, R_{j_{h+1}}, ..., R_{j_s}\} \text{ and } \{R_{i_1}, ..., R_{i_k}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}$$

contains edge e.

We will show that Assumption 16 implies that only one of $\{R_{i_1}, ..., R_{i_k}\}$, $\{R_i, R_{i_{k+1}}, ...R_{i_t}\}$, $\{R_{j_1}, .., R_{j_h}\}$, $\{R_{j_{h+1}}, ..., R_{j_s}\}$ contains e. This in turn implies that only one of

$$\{R_{i_1}, ..., R_{i_k}, R_{j_{h+1}}, ..., R_{j_s}\} \text{ and } \{R_{j_1}, ..., R_{j_h}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}$$

contains e in the modified solution.

Observe that only a consecutively ordered subset of $\{R_{j_1}, ..., R_{j_h}, R_{j_{h+1}}, ..., R_{j_s}\}$ can contain subpath constraints containing e. To see this, consider R_{j_u}, R_{j_v}, R_{j_w} where $j_u < j_v < j_w$ and R_{j_u} and R_{j_w} contain $e = (v_\alpha, v_\beta)$. We have $\min R_{j_v} \leq \min R_{j_w} \leq$

v_α, and $v_\beta \leq \max R_{j_u} \leq \max R_{j_v}$. These bounds, combined with the compatibility of all three, imply R_{j_v} must contain $e = (v_\alpha, v_\beta)$ as well. The same argument holds for $\{R_{i_1}, ..., R_{i_k}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}$. This observation implies we just need to investigate the two cases below as possible examples where more than one of $\{R_{i_1}, ..., R_{i_k}\}$, $\{R_i, R_{i_{k+1}}, ...R_{i_t}\}$, $\{R_{j_1}, .., R_{j_h}\}$, $\{R_{j_{h+1}}, ..., R_{j_s}\}$ contains e.

(i) Both R_{j_h} and $R_{j_{h+1}}$ contain $e = (v_\alpha, v_\beta)$: We claim that R_i must also contain e. This is since $\min R_i \leq \min R_{j_{h+1}} \leq v_\alpha$, $v_\beta \leq \max R_{j_h} \leq \max R_i$, and $R_{j_h} \sim R_i$. Hence, both $\{R_{j_1}, ..., R_{j_h}, R_{j_{h+1}}, ..., R_{j_s}\}$ and $\{R_{i_1}, ..., R_{i_k}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}$ must contain e, contradicting Assumption 16.

(ii) Both R_{i_k} and R_i contain $e = (v_\alpha, v_\beta)$: We claim that R_{j_h} must contain e. This is since $\min R_{j_h} \leq \min R_i \leq v_\alpha$, $v_\beta \leq \max R_{i_k} \leq \max R_{j_h}$ (where the last inequality is due to our greedy algorithm's selection process), and $R_i \sim R_{j_h}$. Hence, both $\{R_{j_1}, ..., R_{j_h}, R_{j_{h+1}}, ..., R_{j_s}\}$ and $\{R_{i_1}, ..., R_{i_k}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}$ must contain e, contradicting Assumption 16.

This completes the proof for Case 1.

Case 2. $R_{i_k} \cap R_{j_{h+1}} = \emptyset$. Recall that the modified solution is

$$\{R_{i_1}, ..., R_{i_k}\}, \{R_{j_{h+1}}, ..., R_{j_s}\}, \text{ and } \{R_{j_1}, ..., R_{j_h}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}.$$

If e is contained in both $\{R_{j_1}, ..., R_{j_h}, R_{j_{h+1}}, ..., R_{j_s}\}$ and $\{R_{i_1}, ..., R_{i_k}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}$ in the unmodified OPT(e)-solution, then, since $\{R_{i_1}, ..., R_{i_k}\}$ and $\{R_{j_{h+1}}, ..., R_{j_s}\}$ are vertex disjoint, only one of them can contain e. Hence, the number of subpath constraints containing e in the modified solution cannot increase. Again, we only need to consider when Assumption 16 holds and, by the same arguments used when $R_{i_k} \cap R_{j_{h+1}} \neq \emptyset$, we must only look at cases (i) and (ii) from above.

- In (i), R_{i_k} cannot contain e since it is disjoint from $R_{j_{h+1}}$. Combined with the sorted order and $R_{j_{h+1}}$ containing e, this ensures $\{R_{i_1}, ..., R_{i_k}\}$ does not contain e. The previous argument for (i) then applies to $\{R_{j_{h+1}}, ..., R_{j_s}\}$ and $\{R_{j_1}, ..., R_{j_h}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}$ and shows both $\{R_{j_1}, ..., R_{j_h}, R_{j_{h+1}}, ..., R_{j_s}\}$ and $\{R_{i_1}, ..., R_{i_k}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}$ contain e.
- In (ii), $R_{j_{h+1}}$ does not contain e since it is disjoint from R_{i_k}. Combined with the sorted order and R_{i_k} containing e, this ensures $\{R_{j_{h+1}}, ..., R_{j_s}\}$ does not contain e. The previous argument for (ii) then applies to $\{R_{i_1}, ..., R_{i_k}\}$ and $\{R_{j_1}, ..., R_{j_h}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}$ and shows both $\{R_{j_1}, ..., R_{j_h}, R_{j_{h+1}}, ..., R_{j_s}\}$ and $\{R_{i_1}, ..., R_{i_k}, R_i, R_{i_{k+1}}, ..., R_{i_t}\}$ contain e. ◀

Lemmas 14 and 15 imply that the modified sequence of unions remains optimal with respect to the edge e (it becomes a different OPT(e)-solution, but can still be formed by iterating through $\mathcal{R}$ in the same sorted order). Hence, we can repeat the same swapping procedure on the modified solution for the next iteration where a discrepancy occurs with our greedy algorithm.

2.4 Linear Time Implementation via Suffix-Prefix Overlap

We now show how to implement our greedy algorithm via reducing it to suffix prefix overlap, a classical problem in computational biology with an efficient solution [4].

Define the map str that maps an arbitrary subpath constraint $R_i = v_{i_1} v_{i_2} ... v_{i_{|R_i|}}$ onto the string $\text{str}(R_i) = i_1 i_2 ... i_{|R_i|}$. The linear time implementation of our greedy algorithm first performs a linear time sort on $\text{str}(\mathcal{R}) = \{\text{str}(R) : R \in \mathcal{R}\}$ using the subpath constraint's minimum values as keys, with ties broken arbitrarily. Let the sorted strings be denoted $\text{str}(R_1)$,

..., $\text{str}(R_{|\mathcal{R}|})$. We concatenate a unique symbol $\$_i$ to the end of $\text{str}(R_i)$, $1 \leq i \leq |\mathcal{R}|$. Then we concatenate these strings together to form the string $T = \text{str}(R_1)\$_1 ... \$_{|\mathcal{R}|-1} \text{str}(R_{|\mathcal{R}|})\$_{|\mathcal{R}|}$ and construct a suffix tree ST over T. Briefly speaking, the suffix tree ST is a compact trie constructed from all suffixes of T. For a given suffix of T, there exists a distinct leaf in ST where the labels assigned to the edges on the root-to-leaf path match the corresponding suffix of T when concatenated. Suffix trees can be constructed in linear time [11, 18, 20], even for strings over integer alphabets [3], like the one used here.

Next, we describe how the suffix tree ST is used to solve the FDSC instance. Let r denote the root of ST. We preprocess ST by marking every node with a branch whose edge has a label that starts with a \$-symbol (including r). We keep pointers from every leaf in ST to its closest marked ancestor. See Figure 5. This preprocessing can be done in linear time via one traversal of ST. We then iterate from $i = 1$ to $|\mathcal{R}|$, and on the i^{th} iteration we start at the leaf ℓ corresponding to the suffix $\text{str}(R_i)\$_i ... \text{str}(R_{|\mathcal{R}|})\$_{|\mathcal{R}|}$ in T. Let ℓ's closest marked ancestor be u.

- If $u = r$, we record that R_i is the start of a new subpath constraint.
- If $u \neq r$ and is marked due to some $\$_h$ we record that R_i should be unioned with R_h. Note that u cannot be marked due to $\$_i$, since this would imply that $\text{str}(R_i)$ occurs as a substring twice in T, which would, in turn, imply R_i is completely contained in some other subpath constraint.

This works since for any $j > i$, R_j will either not contain $\min R_i$, or will have an incompatibility before the $\max R_j$ position. This causes a mismatch between the suffix starting at $\text{str}(R_i)$ and the suffix starting at $\text{str}(R_j)$ prior to the $\$_j$ symbol in the second suffix. Consequently no node marked due to $\$_j$ in ST will be found on the ru-path. Therefore, the lowest $\$_h$ occurring on the path matching R_i in ST indicates R_h where $h < i$, $R_h \sim R_i$, $R_h \cap R_i \neq \emptyset$, and $\max R_h$ is largest.

Once this is completed for all i iterations, we have obtained a sequence of unions that indicates the subpath constraints in $\mathcal{R}_G$. We create a counter for every edge in E. For all $R' = \{R_{i_1}, R_{i_2}, ..., R_{i_k}\} \in \mathcal{R}_G$, we iterate through the set of subpath constraints R_{i_1}, R_{i_2},..., R_{i_k}. For every edge contained in R', we increment the counter for e only the first time it is encountered in R' (actually unioning the subpath constraints in R' is not required). After processing all subpath constraints in $\mathcal{R}_G$, we check for every edge e whether its counter is at most $f(e)$. If this holds for all $e \in E$, we report that the given instance of FDSC is feasible, otherwise, we report that the instance is infeasible.

The initial topological ordering of V requires $O(|V| + |E|)$ time and since G is weakly connected $|E| \geq |V| - 1$. The construction of the suffix tree, followed by the incrementing and checking of counters to determine feasibility, can be done in time proportional to the total length of the subpath constraints in $\mathcal{R}$. Combining these, we obtain Theorem 17.

▶ **Theorem 17.** *An instance $(G = (V, E), \mathcal{R})$ of FDSC can be solved in $O(|E| + L)$ time, where $L = \sum_{R \in \mathcal{R}} |R|$.*

3 Minimizing or Maximizing the Number of Constraints

3.1 Minimizing Subpath Constraint Removal

▶ **Theorem 18.** *There is no polynomial-time approximation algorithm for Minimum Subpath Constraint Removal with an approximation factor that is $o(\log |V|)$ unless $P = NP$.*

Theorem 18 will be proven using a reduction from the Set Cover problem, which is defined as follows: Given a collection $\mathcal{S}$ of subsets S_1, ..., S_m of a universe $U = \{1, 2, ..., n\}$, determine the minimum number of subsets $S_{i_1}, ..., S_{i_k} \in \mathcal{S}$ such that $\cup_{j=1}^{k} S_{i_j} = U$. Well

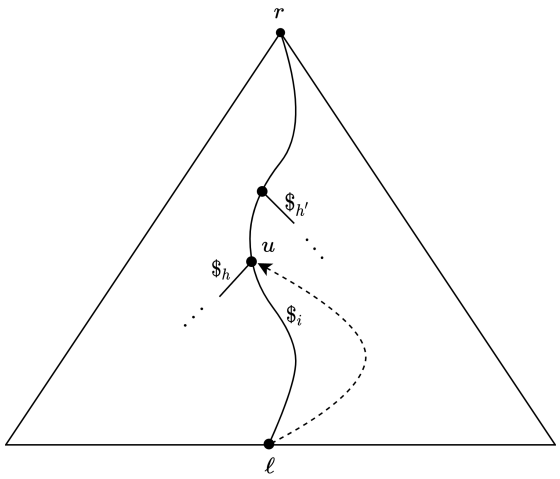


Figure 5 The suffix tree ST for the string $T = \text{str}(R_1)\$_1 ... \$_{|\mathcal{R}|-1}\,\text{str}(R_{|\mathcal{R}|})\$_{|\mathcal{R}|}$ is preprocessed so that every leaf has a pointer to its nearest ancestor that is a marked node. A node is marked if it has a branch whose edge label starts with a \$-symbol.

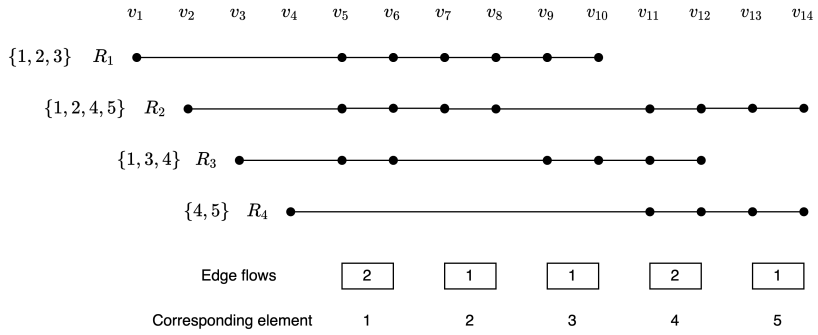


Figure 6 The reduction from the Set Cover instance $\{1,2,3\}, \{1,2,4,5\}, \{1,3,4\}, \{4,5\}$ to an instance of Minimum Subpath Constraint Removal. Only the subpath constraints' relevant corresponding edge flows are shown.

known inapproximability bounds state that no polynomial time $o(\log n)$-approximation algorithm exists assuming P $\neq$ NP. These hold even under the assumption that m and n are polynomially related [9, 12].

Reduction. Let the instance of set cover consist of subsets S_1, ..., $S_m \subseteq U = \{1, 2, ..., n\}$. We construct a flow network G as follows: Let V be initially empty. We first add to V the vertices v_1, v_2,..., v_m, v_{m+1}, ..., v_{m+2n}. For each subpath constraint described below, if an edge specified does not exist in E, we add it to E. Let the set of subpath constraints $\mathcal{R}$ be initially empty. We add to $\mathcal{R}$ the subpath constraints: for $1 \leq i \leq m$, if $S_i = \{x_1, x_2, ..., x_h\}$,

$$R_i = v_i \; v_{m+2x_1-1} \; v_{m+2x_1} \; v_{m+2x_2-1} \; v_{m+2x_2} \; \ldots \; v_{m+2x_h-1} \; v_{m+2x_h}.$$

Next, we describe the flows assigned to each edge. For the edge $e = (v_{m+2j-1}, v_{m+2j})$, $1 \leq j \leq n$, we make $f(e) = |\mathcal{R}(e)| - 1$, i.e., the number of subpath constraints containing that edge minus 1. See Figure 6. For the remaining edges created above, we make the flow

the number of subpath constraints containing that edge. We now create source and sink vertices s and t. For $v \in V \setminus \{s,t\}$ where $\sum_{(u,v)\in E} f((u,v)) < \sum_{(v,w)\in E} f((v,w))$, we add the edge (s,v) and make

$$f((s,v)) = \sum_{(v,w)\in E} f((v,w)) - \sum_{(u,v)\in E} f((u,v)).$$

For $v \in V \setminus \{s,t\}$ where $\sum_{(u,v)\in E} f((u,v)) > \sum_{(v,w)\in E} f((v,w))$, we add the edge (v,t) and make

$$f((v,t)) = \sum_{(u,v)\in E} f((u,v)) - \sum_{(v,w)\in E} f((v,w)).$$

▶ **Lemma 19.** *There exists a set cover of size k for the instance of Set Cover iff there exist k subpath constraints that when deleted make the resulting modified instance of FDSC feasible.*

Proof. First assume there exist a set cover of size k. For each set S_i in the set cover, delete the subpath constraint R_i. Let $\mathcal{R}'$ denote the modified subpath constraint set. For every $x \in \{1,2,...,n\}$, since x is included in some subset taken for the set cover, the number of subpath constraints containing the edge (v_{m+2x-1}, v_{m+2x}) decreases by at least 1. Hence for all $x \in \{1,2,...,n\}$, $|\mathcal{R}'((v_{m+2x-1}, v_{m+2x}))| \leq |\mathcal{R}((v_{m+2x-1}, v_{m+2x}))| - 1 = f((v_{m+2x-1}, v_{m+2x}))$. Since these were the only edges where the number of subpath constraints containing that edge exceeded the flow, and it only exceeded by 1, we now have that for all $e \in E$, $|\mathcal{R}'(e)| \leq f(e)$. By Lemma 12, this suffices to show that $(G, \mathcal{R}')$ is feasible.

In the other direction, assume there exists $\mathcal{R}' \subseteq \mathcal{R}$, such that $|\mathcal{R}'| \geq |\mathcal{R}| - k$ and $(G, \mathcal{R}')$ is feasible. By Lemma 12, this implies $|\mathcal{R}'(e)| \leq f(e)$ for all $e \in E$. Hence, for all $x \in \{1,2,...,n\}$, $|\mathcal{R}'((v_{m+2x-1}, v_{m+2x}))| \leq f((v_{m+2x-1}, v_{m+2x})) = |\mathcal{R}((v_{m+2x-1}, v_{m+2x}))| - 1$. This implies there exists some subpath constraint that was removed and contained edge (v_{m+2x-1}, v_{m+2x}). Hence, if the removed set of subpath constraints is $|\mathcal{R}| \setminus |\mathcal{R}'| = \{R_{i_1}, ..., R_{i_{k'}}\}$, where $k' \leq k$, we have that $S_{i_1} \cup ... \cup S_{i_{k'}} = \{1,2,...,n\}$. ◀

Assuming the instance of set-cover satisfies the condition that m and n are polynomially related, we have $|V| = m + 2n = n^{\Theta(1)}$. In this case, a polynomial-time algorithm providing a $o(\log |V|)$-approximation for the value k, also provides a $o(\log n)$-approximation for set cover. This completes the proof of Theorem 18.

3.2 Maximizing Subpath Constraint Retention

▶ **Theorem 20.** *For every constant $\varepsilon > 0$, there is no polynomial time $O(|V|^{\frac{1}{2}-\varepsilon} + |\mathcal{R}|^{1-\varepsilon})$-approximation algorithm for Maximum Subpath Constraint Retention, unless $P = NP$.*

Theorem 20 is based on a reduction from the Maximum Independent Set problem defined as follows: Given a graph $G = (V, E)$, determine the maximum sized subset $I \subseteq V$ such that no two vertices in I are adjacent, i.e., I is an independent set. We assume that G is connected, making $|E| \geq |V| - 1$.

Reduction. Let $G = (V, E)$ be a given instance of Maximum Independent Set problem where $V = \{u_1, ..., u_{|V|}\}$ and $E = \{e_1, ..., e_{|E|}\}$. We first create a vertex set V' with vertices v_1, ..., $v_{|V|}$, $v_{|V|+1}$, ..., $v_{|V|+2|E|}$. We next construct an edge set E'. Like in the reduction for Minimum Subpath Constraint Removal, if an edge specified by a subpath constraint does not exist, we add it to E'. Let $\mathcal{R}$ be initially empty. For each $u_i \in V$ we add a constraint to $\mathcal{R}$. Suppose u_i is incident to the edges $e_{i_1}, ..., e_{i_h}$. We make

$$R_i = v_i \; v_{|V|+2i_1-1} \; v_{|V|+2i_1} \; v_{|V|+2i_2-1} \; v_{|V|+2i_2} \; \cdots \; v_{|V|+2i_h-1} \; v_{|V|+2i_h}.$$

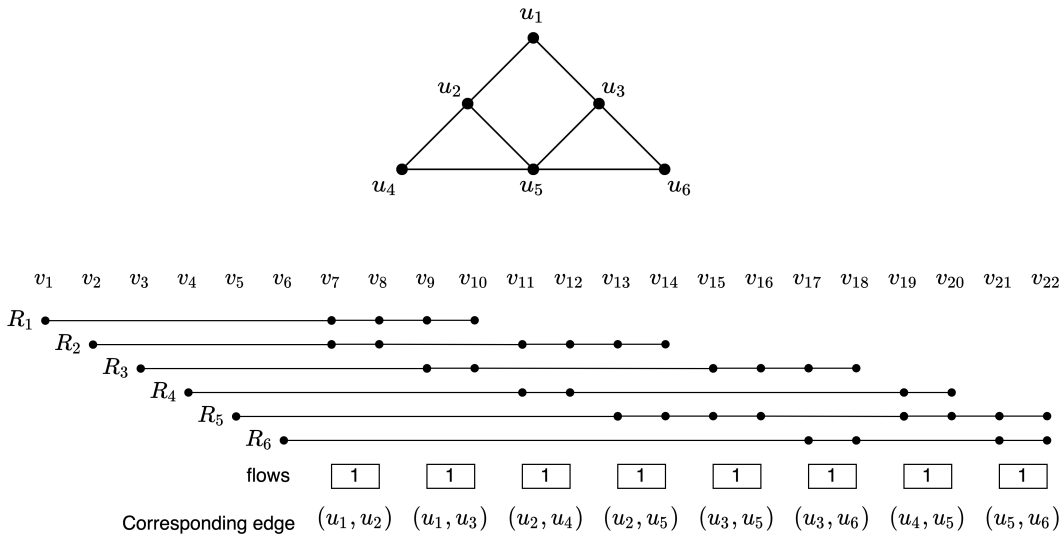


Figure 7 A reduction from the graph above to an instance of Maximum Subpath Constraint Retention. For every vertex there is a corresponding subpath constraint.

For the edge $e = (v_{|V|+2j-1}, v_{|V|+2j}) \in E'$, $1 \leq j \leq |E|$, we make $|f(e)| = 1$. Just as in the reduction for Minimum Subpath Constraint Removal, we make the flows for the remaining edges the number of subpath constraints containing that edge, then add vertices s and t, edges from s to all vertices in $V' \setminus \{s, t\}$, edges from $V' \setminus \{s, t\}$ to t, and assign flows as needed to create a valid flow network. Let G' be the resulting flow network.

▶ **Lemma 21.** *There exists an independent set of size k for the instance of Independent Set iff there exists k subpath constraints that when maintained (and other subpath constraints deleted) make the resulting modified instance of FDSC feasible.*

Proof. First assume there exist an Independent set I of size k. Maintain the subpath constraints corresponding to the vertices in I and delete the remaining subpath constraints. For edge $(v_{|V|+2h-1}, v_{|V|+2h}) \in E'$ that corresponds to an edge $e_h = (u_i, u_j) \in E$, it can not be that both the subpath constraint corresponding u_i and the subpath constraint corresponding to u_j have been maintained, since that would imply that u_i and u_j are both adjacent and in the independent set I. Hence, maintaining only the subpath constraints corresponding to vertices in I must make the resulting instance of FDSC feasible.

In the other direction, assume there exists $\mathcal{R}' \subseteq \mathcal{R}$, such that $|\mathcal{R}'| = k$ and $(G', \mathcal{R}')$ is feasible. By Lemma 12, all edges $e_h = (u_i, u_j) \in E$ have $|\mathcal{R}'((v_{|V|+2h-1}, v_{|V|+2h}))| \leq f(e) = 1$. Hence either R_i or R_j has been removed. This implies that if we take the subset of vertices in V corresponding to the maintained subpath constraints, every edge in E is incident to at most one vertex in this subset. Hence, this subset is an independent set. ◀

For any constant $\varepsilon > 0$ there does not exist a polynomial-time $O(|V|^{1-\varepsilon})$-approximation algorithm for Maximum Independent Set on a graph $G = (V, E)$ unless P = NP [24]. This combined with the above approximation preserving reduction to an FDSC instance $(G' = (V', E'), \mathcal{R})$ where $\mathcal{R} = |V|$, $|V'| = |V| + 2|E| \leq 4|E| \leq 4|V|^2$, and

$$|\mathcal{R}|^{1-\varepsilon} + |V'|^{\frac{1}{2}-\varepsilon} \leq |V|^{1-\varepsilon} + (4|V|^2)^{\frac{1}{2}-\varepsilon} = |V|^{1-\varepsilon} + 4^{\frac{1}{2}-\varepsilon}|V|^{1-2\varepsilon} = O(|V|^{1-\varepsilon}),$$

proves Theorem 20.

4 Open Problems

A question raised by this work is whether feasibility can be determined in polynomial time on cyclic flow networks. A flow decomposition in this setting may be allowed to contain cycles, so the definition of FDSC should make clear whether constraints can also be cyclic. A polynomial-time algorithm for cyclic graphs would have applications to de novo assembly.

References

1. Ergude Bao, Tao Jiang, and Thomas Girke. BRANCH: boosting RNA-Seq assemblies with partial or related genomic sequences. *Bioinform.*, 29(10):1250–1259, 2013. doi:10.1093/bioinformatics/btt127.
2. Elsa Bernard, Laurent Jacob, Julien Mairal, and Jean-Philippe Vert. Efficient RNA isoform identification and quantification from RNA-Seq data with network flows. *Bioinform.*, 30(17):2447–2455, 2014. doi:10.1093/bioinformatics/btu317.
3. Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.
4. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.
5. Shahbaz Khan, Milla Kortelainen, Manuel Cáceres, Lucia Williams, and Alexandru I. Tomescu. Safety and completeness in flow decompositions for RNA assembly. *CoRR*, abs/2201.10372, 2022. arXiv:2201.10372.
6. Shahbaz Khan and Alexandru I. Tomescu. Safety of flow decompositions in dags. *CoRR*, abs/2102.06480, 2021. arXiv:2102.06480.
7. Anna Kuosmanen, Ahmed Sobih, Romeo Rizzi, Veli Mäkinen, and Alexandru I. Tomescu. On using longer RNA-Seq reads to improve transcript prediction accuracy. In James P. Gilbert, Haim Azhari, Hesham H. Ali, Carla Quintão, Jan Sliwa, Carolina Ruiz, Ana L. N. Fred, and Hugo Gamboa, editors, *Proceedings of the 9th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2016) - Volume 3: BIOINFORMATICS, Rome, Italy, February 21-23, 2016*, pages 272–277. SciTePress, 2016. doi:10.5220/0005819702720277.
8. Wei Li, Jianxing Feng, and Tao Jiang. Isolasso: A LASSO regression approach to RNA-Seq based transcriptome assembly. *J. Comput. Biol.*, 18(11):1693–1707, 2011. doi:10.1089/cmb.2011.0171.
9. Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. *J. ACM*, 41(5):960–981, 1994. doi:10.1145/185675.306789.
10. Cong Ma, Hongyu Zheng, and Carl Kingsford. Finding ranges of optimal transcript expression quantification in cases of non-identifiability. *bioRxiv*, 2020.
11. Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.
12. Jelani Nelson. A note on set cover inapproximability independent of universe size. *Electron. Colloquium Comput. Complex.*, 105, 2007. URL: https://eccc.weizmann.ac.il/eccc-reports/2007/TR07-105/index.html.
13. Mihaela Pertea, Geo M Pertea, Corina M Antonescu, Tsung-Cheng Chang, Joshua T Mendell, and Steven L Salzberg. Stringtie enables improved reconstruction of a transcriptome from RNA-Seq reads. *Nature biotechnology*, 33(3):290–295, 2015.
14. Romeo Rizzi, Alexandru I. Tomescu, and Veli Mäkinen. On the complexity of minimum path cover with subpath constraints for multi-assembly. *BMC Bioinform.*, 15(S-9):S5, 2014. doi:10.1186/1471-2105-15-S9-S5.
15. Mingfu Shao and Carl Kingsford. Accurate assembly of transcripts through phase-preserving graph decomposition. *Nature biotechnology*, 35(12):1167–1169, 2017.

16 Mingfu Shao and Carl Kingsford. Theory and a heuristic for the minimum path flow decomposition problem. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 16(2):658–670, 2019. doi:10.1109/TCBB.2017.2779509.

17 Alexandru I. Tomescu, Anna Kuosmanen, Romeo Rizzi, and Veli Mäkinen. A novel min-cost flow method for estimating transcript expression with RNA-Seq. *BMC Bioinform.*, 14(S-5):S15, 2013. doi:10.1186/1471-2105-14-S5-S15.

18 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

19 Benedicte Vatinlen, Fabrice Chauvet, Philippe Chrétienne, and Philippe Mahey. Simple bounds and greedy algorithms for decomposing a flow into a minimal set of paths. *Eur. J. Oper. Res.*, 185(3):1390–1401, 2008. doi:10.1016/j.ejor.2006.05.043.

20 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.

21 Lucia Williams, Alexandru I. Tomescu, and Brendan Mumey. Flow decomposition with subpath constraints. In Alessandra Carbone and Mohammed El-Kebir, editors, *21st International Workshop on Algorithms in Bioinformatics, WABI 2021, August 2-4, 2021, Virtual Conference*, volume 201 of *LIPIcs*, pages 16:1–16:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.WABI.2021.16.

22 Lucia Williams, Alexandru I. Ioan Tomescu, and Brendan Mumey. Flow decomposition with subpath constraints. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pages 1–1, 2022. doi:10.1109/TCBB.2022.3147697.

23 Ting Yu, Zengchao Mu, Zhaoyuan Fang, Xiaoping Liu, Xin Gao, and Juntao Liu. Transborrow: genome-guided transcriptome assembly by borrowing assemblies from different assemblers. *Genome research*, 30(8):1181–1190, 2020.

24 David Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory Comput.*, 3(1):103–128, 2007. doi:10.4086/toc.2007.v003a006.

Prefix-Free Parsing for Building Large Tunnelled Wheeler Graphs

Adrián Goga[1]
Department of Computer Science, Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava, Slovakia

Andrej Baláž
Department of Applied Informatics, Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava, Slovakia

Abstract

We propose a new technique for creating a space-efficient index for large repetitive text collections, such as pangenomic databases containing sequences of many individuals from the same species. We combine two recent techniques from this area: Wheeler graphs (Gagie et al., 2017) and prefix-free parsing (PFP, Boucher et al., 2019).

Wheeler graphs are a general framework encompassing several indexes based on the Burrows-Wheeler transform (BWT), such as the FM-index. Wheeler graphs admit a succinct representation which can be further compacted by employing the idea of tunnelling, which exploits redundancies in the form of parallel, equally-labelled paths called blocks that can be merged into a single path. The problem of finding the optimal set of blocks for tunnelling, i.e. the one that minimizes the size of the resulting Wheeler graph, is known to be NP-complete and remains the most computationally challenging part of the tunnelling process.

To find an adequate set of blocks in less time, we propose a new method based on the prefix-free parsing (PFP). The idea of PFP is to divide the input text into phrases of roughly equal sizes that overlap by a fixed number of characters. The phrases are then sorted lexicographically. The original text is represented by a sequence of phrase ranks (the parse) and a list of all used phrases (the dictionary). In repetitive texts, the PFP representation of the text is generally much shorter than the original since individual phrases are used many times in the parse, thus reducing the size of the dictionary.

To speed up the block selection for tunnelling, we apply the PFP to obtain the parse and the dictionary of the original text, tunnel the Wheeler graph of the parse using existing heuristics and subsequently use this tunnelled parse to construct a compact Wheeler graph of the original text. Compared with constructing a Wheeler graph from the original text without PFP, our method is much faster and uses less memory on collections of pangenomic sequences. Therefore, our method enables the use of Wheeler graphs as a pangenomic reference for real-world pangenomic datasets.

2012 ACM Subject Classification Theory of computation → Theory and algorithms for application domains

Keywords and phrases Wheeler graphs, BWT tunnelling, prefix-free parsing, pangenomic graphs

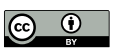
Digital Object Identifier 10.4230/LIPIcs.WABI.2022.18

Related Version *arXiv Version*: `https://arxiv.org/abs/2206.15097`

Supplementary Material *Software (Source Code)*: `https://github.com/fmfi-compbio/pfp_wg` archived at `swh:1:dir:580f9ea6d61e879c69ebb8a8337902e77a3f8a2b`

Funding *Adrián Goga*: VEGA grant 1/0463/20; EU Horizon 2020 grant No. 956229 (ALPACA); Comenius University grant for doctoral students No. 422
Andrej Baláž: VEGA grant 1/0538/22; EU Horizon 2020 grant No. 956229 (ALPACA)

[1] Corresponding author



Acknowledgements We want to thank Travis Gagie for the conception of the idea during his data structures course and helpful remarks throughout the realisation of this project. Our thanks also go to Broňa Brejová for useful advice during the writing process and Uwe Baier for kindly responding to our questions via email. Finally, we thank Lucas Pansani Ramos for the aid he provided in the early days of the project.

1 Introduction

The discovery of Burrows-Wheeler transformation (BWT) [6], a text permutation initially intended for data compression, and the following breakthrough that enhanced it with indexing properties started a new paradigm of text indexing. In this paradigm, the input text can be stored in space not far larger than its empirical entropy while allowing for efficient pattern matching queries. The essence of the BWT, the idea of suffix sorting, has been since enjoyed by a large variety of popular genomic indexing tools. Sequencing read alignment tools, such as BWA [16], Bowtie [14, 15] or CHIC [23] are directly based upon the BWT, while others such as VG [10] use some of its aspects.

As the BWT is an essential tool in processing large textual datasets, it motivates us to further minimize its space requirements. Since its conception, a popular choice has been to use move-to-front (MTF) transformation and subsequently the run-length encoding (RLE), followed by an entropy encoding, e.g. Huffman [6]. Quite recently, in 2018, Baier [2] described another novel source of redundancy in BWTs, yet undetected by the RLE, and proposed a technique called *tunnelling* purposed to deal with this particular redundancy. Tunnelled BWTs are a case of Wheeler graphs [9], a general framework of the text indices based on the idea of suffix sorting.

Tunnelling has been experimentally proven to reduce the size of the resulting BWTs, especially in repetitive texts. The size of the input files is a significant bottleneck of this approach for two reasons. The first one is the fast BWT construction algorithm which requires linear, although non-negligible, memory overhead over the input file itself. The second reason is that tunnelling requires random access to the BWT, rendering it impractical when working in external memory.

The first of these issues could be alleviated by employing a suitable BWT construction algorithm, e.g. one that creates the BWT from a compressed representation of the input, such as the Lempel-Ziv parse by Policriti and Prezza [20] or the Prefix-Free Parsing (PFP) method by Boucher et al. [5]. PFP is especially interesting due to its simplicity and effectiveness, accomplishing to represent large repetitive texts in orders of magnitude smaller space [5]. The second issue is harder to tackle, requiring the attention to either adapt the particular tunnelling algorithm to find the tunnelled BWT from the compressed representation or, when possible, tunnelling the BWT of the compressed representation and then using the information to output a tunnelled BWT of the original input.

The objective of our work is to resolve both problems simultaneously. We have adapted the PFP approach for building BWTs from large repetitive datasets in such a way that it produces tunnelled BWTs. While the resulting BWTs are larger than the original tunnelled BWTs, our approach scales to much larger dataset sizes and could be improved by further post-processing. Our method is not limited to producing tunnelled BWTs, but virtually any Wheeler graphs and hence provides a good starting point for building large pangenomic indexes.

1.1 Preliminaries

In this work, we will use the following notation. Let Σ be the working alphabet, i.e. a finite set of symbols totally ordered by $\preceq$. We denote the size of Σ as $|\Sigma| = \sigma$. A *string* $T = T[1] \dots T[n]$ is a sequence of symbols from Σ. We use the terms *string* and *text* interchangeably, similarly with *symbol* and *character*. A string made of a single character is called *unary*. Furthermore, strings that end with `$` are called *null-terminated*, where `$` is the lexicographically smallest symbol in Σ (i.e. $\$ \preceq c$ for any $c \in \Sigma$) that may only appear at the right end. Moreover, we extend the use of $\preceq$ to label the lexicographic order of strings and write $\alpha \preceq \beta$ if and only if α is lexicographically smaller or equal than β, where $\alpha, \beta \in \Sigma^*$. We will write $\alpha \prec \beta$ if $\alpha \preceq \beta$ and $\alpha \neq \beta$.

The length of the string T will be denoted by $|T| = n$. The string ε is the unique string of length 0. A character of T at position i is denoted by $T[i]$. A substring $T[i \dots j]$ of S is a subsequence $T[i]T[i+1] \dots T[j]$. We assume $T[i \dots j] = \varepsilon$ if $i > j$. The substring $T[1 \dots j]$ is called the j-th *prefix* of T, while the substring $T[j \dots n]$ is called the j-th *suffix*.

For integers i, j we define the set $\{k \mid k \in \mathbb{N};\ i \leq k \leq j\}$ to be the *interval* between i and j and denote it as $[i, j]$.

2 Background

2.1 Burrows-Wheeler transform

The Burrows-Wheeler transform (BWT) [6] is an incredibly influential transformation of text, formed by the concatenation of the letters in the last column of the Burrows-Wheeler matrix, which is created by lexicographically sorting all possible rotations of the original text. This construction is visualized in Figure 1.

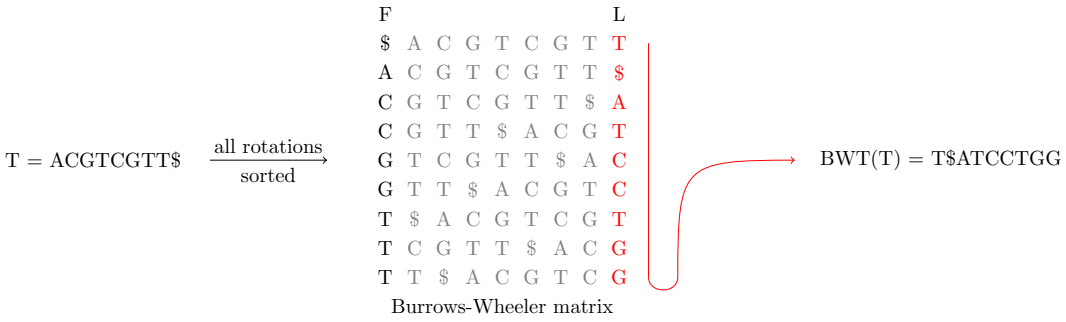


Figure 1 BWT construction. All rotations of the string `ACGTCGTT$` are sorted lexicographically in the Burrows-Wheeler matrix. The last column of Burrows-Wheeler matrix represents the BWT.

The importance of the BWT in string processing is derived from its compressibility, reversibility and usefulness in pattern matching. At first, the improvement in the compressibility of BWT in comparison with the original string is not apparent. However, it stems from empirical observation that similar contexts tend to be preceded by the same characters. The construction of BWT sorts these contexts and therefore tends to place identical characters in runs, which improves the compression.

The reversibility of the BWT is guaranteed by the so-called LF mapping, which uses the observation that the ranks of letters remain the same between the BWT and the first column of the Wheeler matrix F (i.e. the first letter in BWT corresponds to the first letter in F). Since column F can be recovered from BWT by simply sorting the letters of BWT,

the original text can be reconstructed in reverse order by following the LF mapping from the first row of the implicit Wheeler matrix. Furthermore, the LF mapping, together with the suffix array, can be used for efficient pattern matching in the data structure known as FM-index [7].

Although illustrative, the aforementioned construction of BWT is time-inefficient and seldom used in practice. Several efficient methods for construction in linear time exist [13, 5]. This work utilizes prefix-free parsing (PFP), a technique first used for building BWTs in [5]. In comparison with previous methods, the PFP approach allowed the construction of BWT in sublinear space for repetitive data and therefore unlocked a possibility to produce BWTs of big datasets, of which sizes vastly exceed the memory limits of current computers.

2.2 Prefix-free parsing

In prefix-free parsing, the original text T is divided into phrases of variable length, which are stored in the dictionary $\mathcal{D}$. Furthermore, the parse $\mathcal{P}$ is created as a list of lexicographic ranks of phrases, in order in which the phrases appear in the original text. The parse $\mathcal{P}$ and the dictionary $\mathcal{D}$ allow us to reconstruct the original text T.

To perform the division of T into phrases, we define a set of trigger words E, where each trigger word is of length w. Subsequently, we use a sliding window of size w through the text T and each time the sliding window matches a trigger word in E, we terminate the current phrase, add it to the dictionary, and initiate a new phrase. In the end, the dictionary is sorted lexicographically, and the parse is relabeled accordingly. It is noteworthy that the phrases in the dictionary begin and end with a trigger word, and consequently, no phrase is a prefix of another phrase in the dictionary.

This PFP construction is valuable for several reasons. Firstly, only a single pass through the original text is needed to obtain all phrases, and therefore the text can be sequentially read from the disk allowing the processing of large texts.

Secondly, suppose the original text is repetitive. In that case, the parse and the dictionary tend to occupy a much smaller space than the raw representation, and therefore the PFP can be loaded into memory.

Finally, we can use a rolling hash $h : \text{word} \to \{0 \dots p-1\}$ to identify the trigger words. If the hash is equal to zero, we consider it a member of the set of trigger words E. The parameter p then allows us to adjust the lengths of phrases since the expected length of a phrase in a uniformly-random text is p.

An example of a prefix-free parse construction is shown in Example 1.

▶ **Example 1.** Let us have a text T = `#ABDACDABDACDA$`. For the purpose of this example, let the set of trigger words be $E = \{\texttt{A}\}$. Then we have $D = \texttt{\#A}, \texttt{ABDA}, \texttt{ACDA}, \texttt{A\$}$ and $\mathcal{P} = [0, 1, 2, 1, 2, 3]$. Then we lexicographically sort the dictionary phrases to get `#A`, `A$`, `ABDA`, `ACDA` and the remapped parse $\mathcal{P} = [0, 2, 3, 2, 3, 1]$.

2.3 Wheeler graphs

Wheeler graphs (WGs), introduced by Gagie et al. [9], are a class of labelled graphs that generalize the notion of BWT and several of its variations into a unified framework. A formal definition of Wheeler graphs is presented in Definition 2.

▶ **Definition 2** (Wheeler graph). *Let $G = (V, E)$ be a directed graph labeled by $\lambda : E \to \Sigma$. Then G is a Wheeler graph if and only if there is a total order $\leq$ of its vertices, also called the Wheeler order, such that for each pair of edges $e_1 = (u_1, v_1), e_2 = (u_2, v_2)$ the following conditions hold:*

1. *The vertices with zero in-degree precede those with non-zero in-degree,*
2. *If* $\lambda(e_1) \prec \lambda(e_2)$*, then* $v_1 < v_2$*,*
3. *If* $\lambda(e_1) = \lambda(e_2)$ *and* $u_1 < u_2$*, then* $v_1 \leq v_2$*.*

To illustrate this definition, we can visualize the Wheeler graphs by duplicating the vertices and displaying them in two columns in their Wheeler order, as shown in Figure 2. The edge (u, v) will be drawn from vertex u in the right column to vertex v in the left column. Then, the first condition guarantees that the vertices without an incoming edge will form a single interval in the ordering, and that interval will be placed at the top. The second condition guarantees that the vertices with incoming edges labelled by a character c will form an interval, and those intervals appear from top to bottom according to the lexicographic order of the character c. The third condition guarantees that no two edges with the same label c will cross.

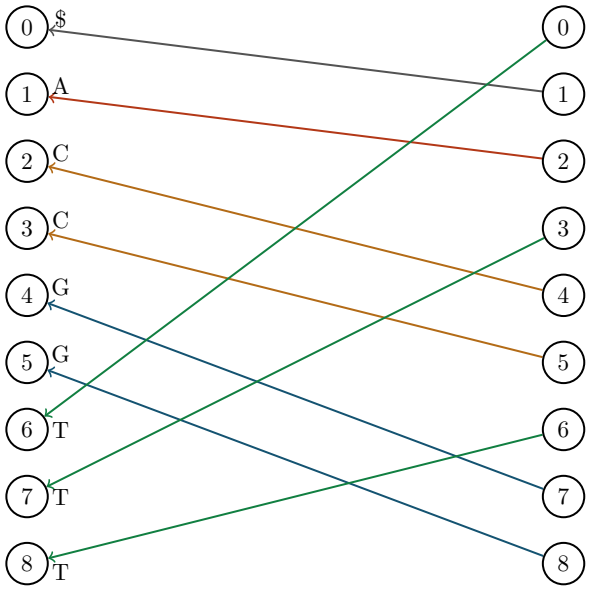


Figure 2 The Wheeler graph visualization. The vertices are doubled and shown in two columns to highlight the conditions of Definition 2. Visualization also reveals the connection between WGs and LF mapping in BWTs.

These conditions give rise to an essential property of WGs, called *path coherence*, which is a generalization of the property from BWTs that enables the LF mapping. Therefore, WGs can be used for efficient pattern matching and to store textual data in a lossless manner. Furthermore, WGs allow for a succinct representation by storing the following data.

- An array C of length σ s.t. $C[c]$ is the number of edges labeled by characters lexicographically smaller than c,
- The string $L = L_1 \dots L_n$, where L_i is the concatenation of the labels of the outgoing edges of vertex v_i,
- The bit vector $O = 10^{d_{out}(v_1)-1} \dots 10^{d_{out}(v_n)-1}$,
- The bit vector $I = 10^{d_{in}(v_1)-1} \dots 10^{d_{in}(v_n)-1}$,

where $d_{in}(v)$ ($d_{out}(v)$) is the in-degree (out-degree) of the vertex v.

In comparison with the plain list-of-edges representation, which uses $\Theta(|E| \log \sigma + |E| \log |V|)$ bits of space, the succinct representation uses only $\Theta(|E| \log \sigma + 2|E| + \sigma \log |E|)$ and together with efficient rank and select queries on the bit vectors O and I supports efficient graph traversing.

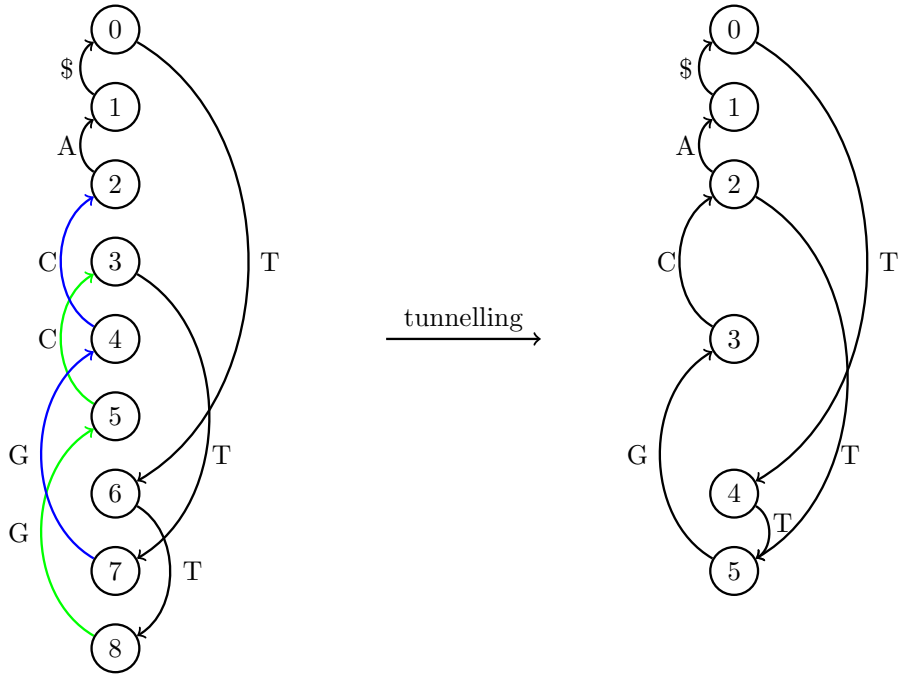


Figure 3 Tunnelling of the WG. Both paths $8-5-3$ (green) and $7-4-2$ (blue) spell `GC` and are merged into a single path in the reduced WG.

Another compelling property of WGs is their reducibility, where a WG representing a particular set of strings can be replaced by a smaller WG representing the same set of strings. One way to accomplish this is by treating the given WG as a Wheeler nondeterministic finite automaton (WNFA), which we subsequently convert to its deterministic variant (WDFA). It has been shown that such conversions for automata that correspond to WGs result only in a linear number of additional states [1]. For a WDFA, an equivalent WDFA with the minimum number of states can then be found by a linear-time algorithm by Alanko et al. [21].

Another particular process of reducing a WG is called *tunnelling* [2] and is illustrated in Figure 3. During the tunnelling, the paths $p_1 = (v_{1,1}, \ldots, v_{1,\ell}), \ldots, p_k = (v_{k,1}, \ldots, v_{k,\ell})$ of which the vertex sets $V_j = \{v_{i,j} \mid 1 \leq i \leq k\}$ are adjacent in the Wheeler order for each $1 \leq j \leq \ell$, and which spell the same substring are identified and a tunnel is formed by merging the vertices from V_j into a single vertex v_j. In general, the task of finding the smallest WG via tunneling was proven to be NP-hard, and therefore multiple heuristic approaches to solve this problem were proposed [4].

3 Methods

We propose to employ PFP as a preprocessing step in building Wheeler graph indexes from large datasets through tunnelling BWTs. We are inspired by the original work by Boucher et al. [5] of building large BWTs using small working memory.

After obtaining the parse $\mathcal{P}$ of a text T, we build the $BWT(\mathcal{P})$ and subsequently find its tunnelling using one of the approaches proposed by Baier and Dede [4]. Since the parse $\mathcal{P}$ is a sequence of lexicographic ranks of dictionary phrases, each character of the parse $\mathcal{P}$ corresponds to multiple characters of T (controlled to a certain degree by the PFP parameter p), and hence serves as a significantly shortened representation of T. Therefore, tunnelling $BWT(\mathcal{P})$ is practical for much larger file sizes.

Under the additional assumption that the input files are repetitive, we can also deduce that at least a certain degree of repetitiveness will also be present in $BWT(\mathcal{P})$, and hence $BWT(\mathcal{P})$ can be significantly shortened by tunnelling. Once we obtain a tunnelled $BWT(\mathcal{P})$ in the form of a Wheeler graph G_P of the parse $\mathcal{P}$, our goal is to efficiently transform it to the Wheeler graph G_T of the text T while retaining the tunnelled paths in an appropriate form.

3.1 Theory

We begin by introducing a vital claim that was already proven by Boucher et al. [5].

▶ **Definition 3** (Suffix set [5]). *Let T be a text of length n. We call the set $Suf(T) = \{T[i \dots n] \mid i \in [1, n]\}$ a suffix set of T.*

▶ **Lemma 4.** *Let T be a text, $\mathcal{P}$ be the parse of the $PFP(T)$ and $x, y \in Suf(T)$ such that $x \prec y$. Furthermore, let α_x, α_y be prefixes of x, y up to the nearest phrase boundaries and β_x, β_y be the rest of x, y. The β_x, β_y correspond to suffixes γ_x, γ_y in the parse $\mathcal{P}$. Then either it holds that $\alpha_x \prec \alpha_y$ or $\alpha_x = \alpha_y$ and $\gamma_x \prec \gamma_y$.*

Proof. The claim follows from the lemmas 1-7 proven by Boucher et al. [5]. ◀

The lemma 4 allows the $BWT(T)$ to be constructed from the parse $\mathcal{P}$ and the dictionary $\mathcal{D}$ by iterating over the suffixes of the phrases from $\mathcal{D} = \{D_1, \dots, D_k\}$ sorted in the lexicographic order and outputting the characters preceding their occurrences in T. As the vertices of Wheeler graphs are totally ordered according to the suffixes of T they represent, we immediately obtain a straightforward generalization of this result for Wheeler graphs in the sense that a Wheeler graph of $\mathcal{P}$ implies a Wheeler graph of T. The exact meaning of this implication is detailed in Def. 5.

▶ **Definition 5** (Expanded graph). *Let T be a string and $G_{\mathcal{P}} = (V_{\mathcal{P}} = [1, m], E_{\mathcal{P}}, \lambda_{\mathcal{P}})$ be a Wheeler graph of $\mathcal{P}$ from the $PFP(T)$, the vertices of which are labeled according to a Wheeler order. Then we define the graph $G' = (V', E', \lambda')$ to be the Expanded graph of T constructed from $G_{\mathcal{P}}$, where*

$$
\begin{aligned}
V_{int} =& \{v_{e,2}, \dots, v_{e,\ell-w} \mid e \in E_{\mathcal{P}}, \ell = |D_{\lambda_{\mathcal{P}}(e)}|\},\\
V' =& V_{\mathcal{P}} \cup V_{int},\\
E' =& \{(v_{e,i}, v_{e,i-1}) \mid e \in E_{\mathcal{P}}, i \in [3, \ell - w], \ell = |D_{\lambda_{\mathcal{P}}(e)}|\} \cup\\
& \{(t, v_{e,\ell-w}) \mid e = (t, u) \in E_{\mathcal{P}}, \ell = |D_{\lambda_{\mathcal{P}}(e)}|\} \cup\\
& \{(v_{e,2}, u) \mid e = (t, u) \in E_{\mathcal{P}}\}
\end{aligned}
$$

$$
\lambda'(e') = \begin{cases} D_{\lambda_{\mathcal{P}}(e)}[i-1] & \text{if } e' = (v_{e,i}, v_{e,i-1}), e \in E_{\mathcal{P}}, \ell = |D_{\lambda_{\mathcal{P}}(e)}|, i \in [3, \ell - w],\\ D_{\lambda_{\mathcal{P}}(e)}[\ell - w] & \text{if } e' = (t, v_{e,\ell-w}), e \in E_{\mathcal{P}}, \ell = |D_{\lambda_{\mathcal{P}}(e)}|,\\ D_{\lambda_{\mathcal{P}}(e)}[1] & \text{if } e' = (v_{e,2}, u), e = (v, u) \in E_{\mathcal{P}}. \end{cases}
$$

The expanded graph $G_T = (V, E, \lambda)$ is therefore obtained by expanding each edge $e = (t, u) \in E_{\mathcal{P}}$ labeled $\lambda_{\mathcal{P}}(e)$ into a path with newly added internal vertices $v_{e,2}, \dots, v_{e,\ell-w}$ so that the concatenation of the labels $\lambda((v_{e,2}, u))\lambda((v_{e,3}, v_{e,2})) \dots \lambda((v_{e,\ell-w}, v_{e,\ell-w-1}))\lambda((t, v_{e,\ell-w})) = D_{\lambda(e)}[1 \dots \ell - w]$, where $\ell = |D_{\lambda(e)}|$. In other words, if we take an edge $e = (t, u) \in E_{\mathcal{P}}$ and an internal vertex

v from the expansion of e in G_T, then starting from v and outputting the edge labels while traversing the reversed edges up to t we get a suffix of $D_{\lambda(e)}[1 \dots \ell - w]$. With the help of Def. 6, we extend this notion to non-internal vertices in Def. 7, which we then use in Thm. 8.

▶ **Definition 6** (Incoming label). *Let $G = (V, E, \lambda)$ be a Wheeler graph. Then for any $v \in V$ we will call the value $i(v) = \lambda((u, v))$ for such $u \in V$ that $(u, v) \in E$ an incoming label of the vertex v.*

▶ **Definition 7** (Phrase suffix). *Let $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}}, \lambda_{\mathcal{P}})$ be a Wheeler graph over a PFP of the string T. Let $G = (V, E, \lambda)$ be the expanded graph of $G_{\mathcal{P}}$. Then for any $v \in V$ we defined $f(v)$ as*

$$f(v) = \begin{cases} D_{i(w)}[i \dots \ell - w], & \text{if } v = v_{e,i}, e = (u, w) \in E_{\mathcal{P}}, i \in [2, \ell - w], \ell = |D_{\lambda_{\mathcal{P}}(e)}| \\ D_{i(v)}[1 \dots \ell - w], & \text{otherwise} \end{cases}$$

Note that in a Wheeler graph, all the incoming edges to a vertex v have to be labelled the same, hence the correctness of Def. 6. Def. 7 deals with two cases – the first one being the internal vertex of the expanded Wheeler graph and a vertex of the $G_{\mathcal{P}}$. Now we are ready to state our core theorem.

▶ **Theorem 8.** *Let $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}})$ be a Wheeler graph over a PFP of the string T. Let $G = (V, E)$ be the expanded graph of $G_{\mathcal{P}}$. Then G is also a Wheeler graph.*

Proof. We prove the claim by showing that there is an ordering of vertices from V such that the conditions from Def. 2 are fulfilled. We define the order as follows. Let $v_1, v_2 \in V$ and $f(v_i)$ a phrase suffix according to Def. 7 for $i = 1, 2$. Then $v_1 \prec v_2$ if and only if $f(v_1) \prec f(v_2)$ or $f(v_1) = f(v_2)$ and $u_1 < u_2$.

We have no vertices of zero in-degree in G, so property 1 is satisfied for any vertex order. We need to show that the vertices in V are sorted according to the suffixes of T they represent. However, this follows from the lemma 4 and the definition of our order. ◀

We note that finding an ordering of the nodes of Wheeler graphs is *NP*-complete in general [11]; however, it can indeed be accomplished in a polynomial time in our restricted case, in which we already have the relative order of the vertices with in-degree and out-degree larger than one[2].

To build the expanded Wheeler graph, we follow an approach similar to that of Boucher et al. [5] for building large BWTs from $\mathcal{P}$ and $\mathcal{D}$. First, we construct a suffix array $\mathrm{SA}_{\mathcal{D}}$ of the concatenation of all phrases from the dictionary $\mathcal{D}$ in the lexicographic order. As we pointed out before, to find the relative order of the nodes u, v in a Wheeler graph, we need to compare the suffixes of T that precede u and v. According to the proof of Theorem 8, it only suffices to compare the suffixes α, β of the phrases in which u, v are located, and if they are equal, we need only to compare their following suffixes, which we can do using the parse $\mathcal{P}$. Having already computed the Wheeler graph $G_{\mathcal{P}}$, we know the lexicographic rank of each suffix among those starting at phrase boundaries.

Hence, as in Boucher et al. [5], we will iterate over the $\mathrm{SA}_{\mathcal{D}}$ while we simultaneously build a succinct representation of G_T, i.e. the array L together with bit vectors I and O. For each phrase $d \in \mathcal{D}$, we save the ranks of vertices $u \in V$ such that $e = (u, v) \in E$ and $\lambda(e) = d$. Moreover, for each such phrase suffix s, we save the labels of the outgoing edges from the vertices $v \in V$ that are suffixed by s.

[2] we even have the relative order of some vertices with both in-degree and out-degree equal to 1, but that is not helpful in general

We will proceed in two passes, each time outputting the entries of the arrays L, I and O for a different subset of nodes. First, we will process those that correspond to vertices suffixed by those elements $s \in \mathcal{S}$ that are either suffix of only a single phrase $d \in \mathcal{D}$ or appear only once in $\mathcal{D}$, leaving placeholders for the rest of the vertices. The vertices corresponding to elements of $\mathcal{S}$ that are suffixes of multiple phrases of $\mathcal{D}$ or are a whole phrase that appears multiple times in T will be processed according to the increasing Wheeler order of their phrase-boundary suffixes. All in all, this proves Theorem 9.

▶ **Theorem 9.** *Let $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}}, \lambda_{\mathcal{P}})$ be a Wheeler graph over a PFP of the string T of length n. Then we can build the expanded Wheeler graph $G = (V, E, \lambda)$ from $G_{\mathcal{P}}$ in $O(n)$ time and workspace proportional to $O(PFP(T))$.*

3.2 Implementation

We have built our approach as an extension of the `bigbwt` – an implementation of the original method by Boucher et al. [5] for building BWTs using PFP. We have used the original PFP component of the `bigbwt` without changes. In contrast with `bigbwt`, the only intermediate file that we need apart from the PFP is the tunnelled BWT of the input. We have used the linear time SACA-K [18] algorithm that only uses $O(1)$ workspace to create the suffix array of $\mathcal{P}$, which is subsequently transformed into the $BWT(\mathcal{P})$. As in `bigbwt`, the suffix array of $\mathcal{D}$ along with the LCP array is created using the gSACAK algorithm [17] and is later used to iterate over the suffixes of the phrases from $\mathcal{D}$ in the increasing lexicographic order.

To find the tunnelling of the $BWT(\mathcal{P})$, we considered several approaches suggested by Baier et al. [2, 4, 3]. Most of them aimed at reducing the compressed size of the run-length encoded BWT. Since our current goal is not to optimize the compressed size of the constructed WG (which is not straightforward to do when only having access to $\mathcal{P}$), we have employed a tunnelling method which does not necessarily minimize the size of the tunnelled BWT after compression, but rather the number of edges in the resulting WG. While being tightly linked with the edge-minimization of the de Bruijn graph constructed from the input text, the prefix intervals in this method do not overlap, and Baier et al. [3] have shown that the method leads to significant reduction of BWT size for repetitive datasets. The only alteration we have made is that we have enabled the processing of large integer alphabets.

The implementation is heavily based on the Succinct Data Structure Library (SDSL) by Simon Gog et al.[12]. Namely, the tunnelled BWT is represented as a wavelet tree that stores the L component, which allows for efficient select queries. The rank and select support for the bitvectors I and O is also supported by SDSL.

Similarly to the implementation of the `bigbwt` [3] [5], we slightly deviate from the method of resolving the ambiguous phrase suffixes; instead of leaving the placeholders to fill in another pass, we use heapsort to merge the given vertices according to the rest of the suffixes. We note that this information is already available in the $G_{\mathcal{P}}$ as the labels of the vertices in $V_{\mathcal{P}}$, so we only need to sort integers instead of actual suffixes. This approach allows us to sequentially write the succinct representation of the resulting WG in a single pass, to some extent alleviating the slow writing process of external memories.

[3] `https://gitlab.com/manzai/Big-BWT`

4 Experiments

We have demonstrated the applicability of our approach using two real-world datasets, 5520 genomes of the Salmonella genus and 1000 copies of human chromosome 19.

The Salmonella dataset was obtained using `ncbi-datasets-cli` tool. Particularly, the command `datasets download genome taxon salmonella -assembly-level complete_genome` was used. The genomic fasta files were concatenated into a single fasta file of size 13GB. The dataset of haplotypes of human chromosome 19 was obtained from the 1000 genomes project [22][4].

From these fasta files, we extracted subsets of increasing sizes to observe the trends of running time and memory consumption, as well as the sizes of the resulting WGs. For the sake of simplicity, we preprocessed the datasets so that we ignored all characters apart from `A`, `C`, `G`, `T`. All of the experiments were run on a machine with Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz and 144GB of operational memory. Every program was executed using a single thread. The running time and peak memory consumption were measured using the `/usr/bin/time -v` command.

The results of the experiments for the copies of the Salmonella genus and chromosome 19 are displayed in Table 1 and 2. Apart from our implementation, we also provide the results for the approach of Baier et al. [3] for comparison. Their approach consists of first constructing a BWT of the input and then tunnelling the constructed BWT. The BWT construction algorithm can be done by either the DIVSUFSORT algorithm [8] or a semi-external variant of the induced sorting algorithm [19] (SE-SAIS), where the former is fast but demanding significantly more memory than the latter, which is slower but memory-efficient. We have used the SE-SAIS since it makes a more fair comparison to our PFP-based algorithm. Our experiments were all ran with the PFP parameters set to $w = 4$ and $p = 50$.

Expectedly, the WGs we produce are larger than those of Baier et al. due to the constraints of PFP, which limits the space of possibilities of tunnelling and treats the removal of any parse character such as having the same benefit, which is clearly not the case from the perspective of the input. We also point out that smaller WGs can be achieved with different combinations of the PFP parameters. Our approach, however, operates within much less memory and shorter running time, which allows it to construct WGs for datasets of unprecedented sizes, e.g. the whole 1000 human genomes project and beyond.

Table 1 Results of the experiments on the Salmonella genomes. The 'time' is the running time in seconds, 'memory' stands for the peak memory usage in MBs during the running time. The file sizes are reported in MBs.

#sequences	size	Baier et al. [3]			PFP		
		time	memory	size	time	memory	size
1	5	8	47	6	5	57	7
10	34	42	118	16	24	223	38
100	238	301	389	55	99	696	174
1000	2619	3431	3128	430	568	2709	1400
5000	12053	18658	14538	1275	1726	4859	4189

[4] dataset available at `http://dolomit.cs.tu-dortmund.de/tudocomp/`

Table 2 Results of the experiments on the copies of chromosome 19. The 'time' is the running time in seconds, 'memory' stands for the peak memory usage in MBs during the running time. The file sizes are reported in MBs. The Baier et al.'s [3] approach for 1000 sequences was terminated after 20 hours.

#sequences	size	Baier et al. [3]			PFP		
		time	memory	size	time	memory	size
1	54	81	341	67	82	597	70
10	533	700	708	87	157	880	296
100	5322	7263	6748	162	644	1319	2123
500	27903	54498	34031	749	2634	4190	13110
1000	53220	—	—	—	5938	8009	29803

5 Conclusion and future work

We have successfully demonstrated that the prefix-free parsing technique can be used to alleviate the computational requirements of the construction of tunnelled BWTs. Our approach allows the use of Wheeler graphs as pangenomic references for huge datasets such as the 1000 Genomes Project, the Vertebrate Genomes Project, the Earth Microbiome Project, and many more.

To this end, we generalized the approach of Boucher et al. [5] devised for the construction of BWTs from large volumes of repetitive data and experimentally showed it can be a good starting point for building Wheeler graphs.

Since our approach treats the PFP parse as any input text, it does not exploit the full information the PFP provides and could benefit from incorporating the lengths of the dictionary phrases into the tunnelling process, potentially even to the point of simulating the tunnelling algorithm on the original text and producing the same output. We leave this for future work.

Another logical next step is to enhance our approach with the ability to output a sufficiently small suffix array sample to allow our Wheeler graphs to locate the occurrences of patterns.

References

1 Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Regular languages meet prefix sorting. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 911–930. SIAM, 2020.

2 Uwe Baier. On Undetected Redundancy in the Burrows-Wheeler Transform. *Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*, 105:3:1–3:15, 2018. doi:10.4230/LIPIcs.CPM.2018.3.

3 Uwe Baier, Thomas Büchler, Enno Ohlebusch, and Pascal Weber. Edge minimization in de Bruijn graphs. In *2020 Data Compression Conference (DCC)*, pages 223–232. IEEE, 2020.

4 Uwe Baier and Kadir Dede. BWT Tunnel Planning is hard but manageable. In *2019 Data Compression Conference (DCC)*, pages 142–151. IEEE, 2019.

5 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms for Molecular Biology*, 14(1):1–15, 2019.

6 Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer, 1994.

7 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st annual symposium on foundations of computer science*, pages 390–398. IEEE, 2000.

8 Johannes Fischer and Florian Kurpicz. Dismantling divsufsort. *arXiv preprint*, 2017. `arXiv:1710.01896`.

9 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical computer science*, 698:67–78, 2017.

10 Erik Garrison, Jouni Sirén, Adam M Novak, Glenn Hickey, Jordan M Eizenga, Eric T Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F Lin, et al. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature biotechnology*, 36(9):875–879, 2018.

11 Daniel Gibney and Sharma V Thankachan. On the complexity of recognizing wheeler graphs. *Algorithmica*, 84(3):784–814, 2022.

12 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014. `doi:10.1007/978-3-319-07959-2_28`.

13 Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International colloquium on automata, languages, and programming*, pages 943–955. Springer, 2003.

14 Ben Langmead. Aligning short sequencing reads with Bowtie. *Current protocols in bioinformatics*, 32(1):11–7, 2010.

15 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9(4):357–359, 2012.

16 Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint*, 2013. `arXiv:1303.3997`.

17 Felipe A Louza, Simon Gog, and Guilherme P Telles. Inducing enhanced suffix arrays for string collections. *Theoretical Computer Science*, 678:22–39, 2017.

18 Ge Nong. Practical linear-time O(1)-workspace suffix sorting for constant alphabets. *ACM Transactions on Information Systems (TOIS)*, 31(3):1–15, 2013.

19 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *2009 data compression conference*, pages 193–202. IEEE, 2009.

20 Alberto Policriti and Nicola Prezza. From LZ77 to the Run-Length Encoded Burrows-Wheeler Transform, and Back. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:10, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.CPM.2017.17`.

21 Jarno Alanko; Nicola Cotumaccio; Nicola Prezza. Linear-time minimization of Wheeler DFAs, 2022. URL: `https://sigport.org/documents/linear-time-minimization-wheeler-dfas`.

22 Nayanah Siva. 1000 genomes project. *Nature biotechnology*, 26(3):256–257, 2008.

23 Daniel Valenzuela and Veli Mäkinen. CHIC: a short read aligner for pan-genomic references. *biorxiv*, page 178129, 2017.

Pangenomic Genotyping with the Marker Array

Taher Mun
Johns Hopkins University, Baltimore MD, USA
Illumina, San Diego, USA

Naga Sai Kavya Vaddadi
Johns Hopkins University, Baltimore MD, USA

Ben Langmead[1]
Johns Hopkins University, USA

Abstract

We present a new method and software tool called `rowbowt` that applies a pangenome index to the problem of inferring genotypes from short-read sequencing data. The method uses a novel indexing structure called the marker array. Using the marker array, we can genotype variants with respect from large panels like the 1000 Genomes Project while avoiding the reference bias that results when aligning to a single linear reference. `rowbowt` can infer accurate genotypes in less time and memory compared to existing graph-based methods.

2012 ACM Subject Classification Applied computing → Computational genomics

Keywords and phrases Sequence alignment indexing genotyping

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.19

Supplementary Material Open source software tool `rowbowt` available at:
Software (Source Code): `https://github.com/alshai/rowbowt`
archived at `swh:1:dir:b9c7b9590a6fe1a7faa105d263ba4240f5acd683`

Funding *Taher Mun*: R01HG011392 and R35GM139602 to BL. Also, NSF-135491
Naga Sai Kavya Vaddadi: R01HG011392 and R35GM139602 to BL
Ben Langmead: R01HG011392 and R35GM139602 to BL. Also, NSF-135491

Acknowledgements We thank Massimiliano Rossi and Travis Gagie for many helpful discussions. We thank Margaret Gagie for her careful editing. Part of this research project was conducted using computational resources at the Maryland Advanced Research Computing Center (MARCC).

1 Introduction

Given DNA sequencing reads from a donor individual, genotyping is the task of determining which alleles the individual has at polymorphic sites. Genotyping from sequencing data, sometimes using low-coverage sequencing data together with genotype imputation, is a common task in human genetics [10] and agriculture [18]. In contrast to variant calling, genotyping is performed with respect to a catalog of known polymorphic sites. For instance, genotyping of a human can be performed with respect to the 1000 Genomes Project call set, which catalogs positions, alleles and allele frequencies for tens of millions of sites [2].

Many existing genotypers start by aligning reads to a single linear reference genome, e.g. the human GRCh38 reference [26]. Because this reference is simply one example of an individual's genome, genotyping is subject to reference bias, the tendency to make mistakes in places where the donor differs genetically from the reference. This was shown in studies of archaic hominids [17], HLA genotypes [4] and structural variants [28]. A similar bias exists

[1] Corresponding author

for methods that extract polymorphic sites along with genomic context, and search for these sequences in the reads [11, 27]. In particular, the bias remains if the flanking sequences are extracted from the reference and so contain only reference alleles.

Reference bias can be avoided by using a *pangenome* reference instead of a single linear reference. A pangenome can take various forms; it can be (a) a generating graph for combinations of alleles, (b) a small collection of linear references indexed separately, or (c) a larger collection of linear reference indexed together in a compressed way. Pangenome graphs (option a) and small collections of linear references (option b) have been studied in recent literature [23, 8, 15, 7]. Variant graphs are effective for genotyping, but have drawbacks in this context. First, haplotype information might be removed when adding variants to the graph, or might be included in the graph but not considered during the read mapping process. This can cause tools like Bayestyper [29] to consider many extraneous haplotype paths through the graph during genotyping, increasing running time. Second, variant graphs can grow exponentially – in terms of the number of paths through the graph – as variants are added, leading to a rapid increase in resource usage and likelihood of ambiguous alignments.

We sought a way to avoid reference bias by indexing and querying many linear references at once while keeping index size and query time low. Such an approach can take full advantage of linkage disequilibrium information in the panel, allowing no recombination events except those occurring in the panel. This avoids mapping ambiguity from spurious recombination events between polymorphic sites [23].

We propose a new structure called the *marker array* that replaces the suffix-array-sample component of the r-index with a structure tailored to the problem of collecting genotype evidence. Here we describe the marker array structure in detail. We compare its space usage and query time to those of the standard r-index and explore how accurately both structures are able to capture markers from a sequencing dataset. Finally, we benchmark it using real whole-genome human sequencing data and compare it to existing genotyping tools in terms of both genotyping accuracy and computational efficiency.

2 Background

r-index

The r-index [14] is a compressed repeat-aware text index that scales with the non-redundant content of a sequence collection. It uses $O(r)$ space where r is the number of same-character *runs* in the Burrows-Wheeler Transform (BWT) of the input text. Past work shows that the r-index can efficiently index collections of long-read-derived human genome assemblies [19] as well as large collections of bacterial genomes [1].

While the main contribution of the r-index was its strategy for storing and using a sample of the suffix array [14], even this sample is large in practice. We propose a new *marker array* structure that replaces the suffix array while retaining its ability to deduce when a read-to-pangenome match provides evidence for a particular allele at a polymorphic site. The design of the marker array flows from three observations. First, we can save space by storing auxiliary information about polymorphic sites ("markers") only at the sites themselves. There are often far fewer sites harboring polymorphism than there are BWT runs. Second, pangenome suffixes starting with the same allele tend to group together in the suffix array, which can be exploited to compress the marker array structure. Third, while a suffix array entry is an offset into the pangenome requiring $O(\log n)$ bits, a marker need only distinguish markers and alleles, and so requires just $O(\log M)$ bits where M is the number of polymorphic sites.

3 Methods

Preliminaries

Consider a string S of length n from ordered alphabet Σ, with operator $\prec$ denoting lexicographical order. Assume S's last character is lexicographically less than the others. Let F be an array of S's characters sorted lexicographically by the suffixes starting at those characters, and let L be an array of S's characters sorted lexicographically by the suffixes starting immediately after them. The list L is the Burrows-Wheeler Transform [5] of S, abbreviated BWT.

The BWT can function as an *index* of S [13]. Given a pattern P of length $m < n$, we seek the number and location of all occurrences of P in S. If we know the range $\mathsf{BWT}(S)[i..j]$ occupied by characters immediately preceding occurrences of a pattern Q in S, we can compute the range $\mathsf{BWT}(S)[i'..j']$ containing characters immediately preceding occurrences of cQ in S, for any character $c \in \Sigma$, since

$$i' = |\{h : S[h] \prec c\}| + |\{h : S[h] = c, h < i\}| \quad (1)$$

$$j' = |\{h : S[h] \prec c\}| + |\{h : S[h] = c, h \leq j\}| - 1\,. \quad (2)$$

The FM Index is a collection of data structures for executing such queries efficiently. It consists of an array C storing $|\{h : S[h] \prec c\}|$ for each character c, plus a rank data structure for $\mathsf{BWT}(S)$, e.g. a wavelet tree, that can quickly tally the occurrences of a character c up to a position of BWT. To locate the offsets of occurrences of P in S, the FM-index can additionally include some form of S's suffix array. The suffix array SA is an array parallel to F containing the offsets of the characters in F. To save space, the FM-index typically keeps only a sample of SA, e.g. a subset spaced regularly across SA or across S.

Let $T = \{T_0, T_1, \ldots, T_n\}$ be a collection of n similar texts where T_0 is the *reference sequence*, and $T_1, \ldots, T_n$ are *alternative sequences*. In the scenarios studied here, a T_i represents a human haplotype sequence, with all chromosomes concatenated, and T_0 represents the GRCh38 primary assembly of the human genome. Each T_i with $i > 0$ is an alternate haplotype taken either from the 1000 Genomes project call set [2] or from the HGSVC project [6, 12], each with chromosomes concatenated in the same order as T_0's. We use the terms "haplotype" and "genome" interchangeably here.

We assume that all the T_i's are interrelated through a multiple alignment, e.g. as provided in a Variant Call Format (VCF) file. The multiple alignment is a matrix with genomes in rows and columns representing genomic offsets. The elements are either bases or gaps. We call a column consisting of identical bases and lacking any gaps a *uniform* column. Any other column is a *polymorphic* column. Figure 1 illustrates a multiply-aligned collection of haplotypes and the concatenated text T.

Marker Array

Let the "marker array" M be an array parallel to the concatenated sequence T marking positions that fall in a polymorphic column in the multiple alignment. Each element of M is a tuple recording the offset i with respect to T_0 where the polymorphism begins, as well as the edit operation describing how the sequence differs from the reference at this locus. Distinct edit operations are given distinct integer identifiers, which are decoded using a separate table E. Identifier 0 is the null operation, denoting that the reference allele appears without edits. For example, say $E = \{1 : X \to C\}$, where $X \to C$ denotes a substitution that replaces the

reference base with C. Then a marker array record $m = (500, 0)$ marks a locus with no edit with respect to reference position $T_0[500]$. A record $m' = (500, 1)$ denotes that a substitution changes that base at $T_0[500]$ to a C. An example is shown in Figure 1 (bottom left).

Consecutive substitutions are collapsed into a single edit in the E table. Insertions and deletions ("indels" for short) are treated somewhat differently; the offset carrying the "mark" is the one just preceding the indel (just to its left) in the multiple alignment. Importantly, the mark covers exactly one position in the genome, even if the insertion/deletion spans many bases. The marked position must come to the left of the indel to ensure that suffixes starting at the marked position include the allele itself. In the multiple alignment in Figure 1 (top left), for example, a deletion with respect to R occurs in the fourth-from-left column, but the marked position is in the third-from-left column.

The marker array MA is a permutation of M such that marks appear in suffix-rank order:

▶ **Definition 1.** *The* marker array MA *is the mapping such that* $\mathsf{MA}[i] = \mathsf{M}[\mathsf{SA}[i]]$.

Thanks to suffix-rank order, identical $\mathsf{M}[i]$'s are often grouped into runs in MA, as seen in Figure 1 (right).

A *marker query* for pattern string q returns all $m \in \mathsf{M}$ overlapped by an occurrence of q in T. We can begin to answer this query using backward search (Equation 1) with $P = q$, giving the maximal SA range $[i..j]$ such that q is a prefix of the suffixes in the range. Having computed $[i..j]$, we know that $\{\mathsf{MA}[i], ..., \mathsf{MA}[j]\}$ contain markers overlapped by q's leftmost character. To recover the markers overlapped by the rest of q's characters, two approaches can be considered, detailed in the following subsections. The FL approach recovers the overlapped markers in a straightforward way but uses $O(|q| \cdot \mathsf{occ})$ time, where occ is the number of times q occurs in T. The heuristic backward-search approach requires only $O(|q|)$ time but is not fully sensitive, i.e. it can miss some overlaps.

FL approach

Say $[i..j]$ is the maximal SA range such that all rows have q as a prefix. We can perform a sequence of FL steps, starting from each row $x \in [i..j]$. An FL step is the inverse of an LF step. That is, if we write an LF mapping step in terms of a rank query

$$i' \quad = \quad |\{h \,:\, S[h] \prec c\}| + \mathsf{BWT}.\mathrm{rank}_{\mathsf{BWT}[i]}(i) \tag{3}$$

where $S.\mathrm{rank}_c(i)$ denotes the number of occurrences of c in S up to but not including offset i, then an FL step inverts this using a select query

$$i \quad = \quad \mathsf{BWT}.\mathrm{select}_{F[i']}(i' - |\{h \,:\, S[h] \prec F[i']\}|) \tag{4}$$

where $S.\mathrm{select}_c(i)$ returns the offset of the $i + 1^{\text{th}}$ occurrence of c in S, i.e. the c of rank i. Whereas LF takes a leftward step with respect to T, FL takes a rightward step.

By starting in each row $x \in [i..j]$ and performing a sequence of $|q| - 1$ FL steps for each, we can visit each offset of T overlapped by an occurrence of q. Checking $\mathsf{MA}[k]$ at each step, where k is the current row, tells us which marker is overlapped, if any. This is slow in practice, both because it requires $O(|q|(j - i + 1))$ FL steps in total, and because each step requires a select query, which is more costly in practice than a rank query.

Heuristic backward-search approach with smearing

Say we perform a backward search starting with the rightmost character of q. At each step we are considering a range $[i..j]$ of SA having a suffix of q as a prefix. Using i and j, we can query $\mathsf{MA}[i..j]$. However, this tells us instances where a *suffix* of q overlaps a marker,

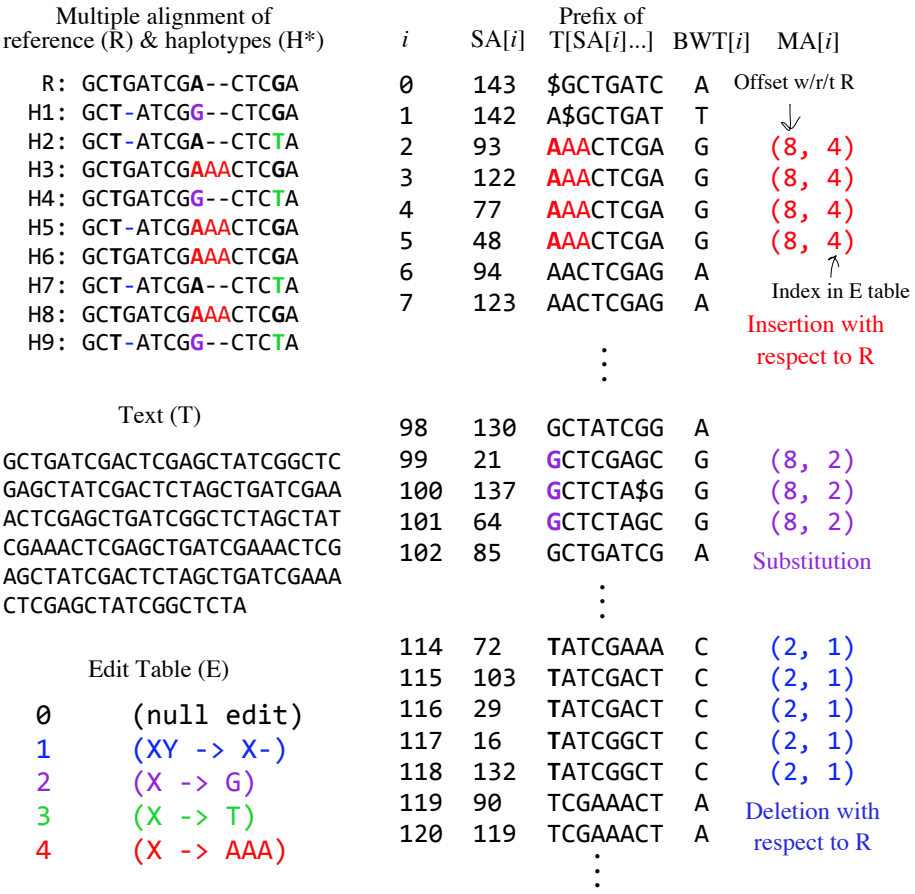


Figure 1 Top left: A multiple alignment for a collection of alternate haplotypes (H1–H9), and a reference sequence (R). Marked bases are in bold and alternate alleles are colored. Middle left: The text T, formed by concatenating rows of the multiple alignment (eliding gaps). Bottom left: The edit table E, with alternate-allele coloring. Right: A partial illustration of the marker array in relation to SA, the relevant suffixes themselves (truncated to fit), and the BWT. Colors and bolding highlight where marked bases and alternate alleles end up in the suffixes.

whereas our goal is to find where the *whole* query q overlaps a marker. If we report overlaps involving trivially short suffixes of q, many would be false positives. We propose to allow but reduce the number of such false positives by augmenting MA:

▶ **Definition 2.** *The augmented marker array* MA^w *is a multimap such that* $\mathsf{MA}^w[i] = [\mathsf{M}[\mathsf{SA}[i]], \mathsf{M}[\mathsf{SA}[i]+1], ..., \mathsf{M}[\mathsf{SA}[i]+w]]$

That is, $\mathsf{MA}^w[i]$ is a (possibly empty) list containing markers overlapping any of the positions $T[i..i+w]$. We call this a "smeared" marker array, since the marks are extended (smeared) to the left by w additional positions. Note that a length-w extension can overlap one or more other marked variants to the left. For this reason, MA^w must be a multimap, i.e. it might associate up to w markers with a given position.

Using MA^w, we adjust the backward-search strategy so that instead of querying MA at each step, we query MA^w every w steps. If w is large enough – e.g. longer than the length at which we see random-chance matches – we can avoid many false positives. More space is required to represent MA^w compared to MA since it is less sparse. However, we expect MA^w to remain run-length compressible for the same reason that MA is.

Genotyping a read

Given a sequencing read, we would like to extract as much genotype information as possible while minimizing computational cost and false-positive genotype evidence. Here we give a heuristic algorithm (Algorithm 1) that handles entire sequencing reads, querying MA^w during the backward-search process as proposed in the previous section. The algorithm proceeds right to left through the read, growing the match by one character if possible. When we can no longer grow the match (i.e. the range $[i..j]$ becomes empty), we reset the range to the all-inclusive range $[0..|T|-1]$ and restart the matching process at the next character. We use the term "extension" to refer to a consecutive sequence of steps that successfully extend a match. Note that this is a heuristic algorithm that does not exhaustively find all half-MEMs between the read and the index, as the MONI algorithm does [25].

As discussed above, the algorithm only checks the marker array every w steps (line 14). As an additional filter, the algorithm only performs a marker-array query when the current suffix-array range size no larger than the number of haplotypes in the index (N_h). A range exceeding that size indicates that we are seeing more than one distinct match in at least one haplotype, meaning that the evidence is ambiguous.

The algorithm tallies evidence as it goes (line 18), but might later choose to ignore that evidence if certain conditions are not satisfied (lines 10 and 27). For example, if the evidence has a conflict – i.e. one match indicates a reference allele at a site but another match found during the same extension finds an alternate allele at that site – then all the evidence is discarded for that extension. Similarly, evidence from an extension is discarded if the tallied sites span multiple chromosomes. Finally, evidence from extensions failing to match at least 80 bp of the read (adjustable with `--min-seed-length` option) is discarded.

We employ other heuristics to minimize mapping time not shown in Algorithm 1. For instance, we avoid wasted effort spent querying the wrong read strand. Specifically: `rowbowt` randomly selects an initial strand of the read to investigate: forward or reverse complement. If an extension from this strand meets the minimum seed-length threshold (80 by default), then the other strand is not considered and analysis of the read is complete. Otherwise, `rowbowt` then goes on to examine the opposite strand of the read.

Sparse marker encoding

We encode the sparse arrays M, MA and MA^w in the following way. Say that array A consists of empty and non-empty elements. We consider A's non-empty elements as falling into one of x maximal runs of identical (and non-empty) elements. Our sparse encoding for A consists of three structures. S is a length-$|A|$ bit vector with 1s at the positions where a run of identical entries in A begins, and 0s elsewhere. E is a similar bit vector marking the last position of each of the x runs. (This variable E is distinct from the E table defined above in "Marker Array.") To find whether an element $A[i]$ is non-empty, we can ask whether we are between two such marks; that is, $A[i]$ is non-empty if and only if $S.\text{rank}_1(i+1) > E.\text{rank}_1(i)$.

X is a length-x array containing the element that is repeated in each of A's non-empty runs, in the order they appear in A. If $A[i]$ is not empty, the element appearing there is given by $X[E.\text{rank}_1(i)]$.

Algorithm 1 Simplified version of `rowbowt` algorithm for matching query string q and compiling genotype evidence. C is an array such that $C[c]$ equals $|\{h : T[h] \prec c\}|$, where T is the original text. N_h is the number of haplotypes in the index. Pseudocode for TallyEvidence and FinalizeEvidence are not given, but the heuristics they use to further filter genotype evidence is described in the text.

```
1: procedure SimpleGenotype(q, w, MA^w, BWT, C, N_h)
2:     i = 0, j = |BWT|
3:     ℓ = 0, k = 0                                          ▷ start new extension
4:     o = |q| − 1
5:     while o ≥ 0 do
6:         c = q[o]
7:         i = C[c] + BWT.rank_c(i)          ▷ backward search, like equations 1 and 2
8:         j = C[c] + BWT.rank_c(j)
9:         if i = j then                                     ▷ no matching substrings
10:            FinalizeEvidence(ℓ)  ▷ ignore tallied evidence if ℓ is small or evidence conflicts
11:            i = 0, j = |BWT|, k = 0, ℓ = 0                ▷ start new extension
12:        else
13:            k = k + 1,   ℓ = ℓ + 1
14:            if k = w then
15:                if (j − i + 1) ≤ N_h then
16:                    for x ∈ [ i .. j ) do
17:                        for site ∈ MA^w[x] do
18:                            TallyEvidence(site)      ▷ increment evidence for allele at site
19:                        end for
20:                    end for
21:                end if
22:                k = 0                                     ▷ check every w steps
23:            end if
24:        end if
25:        o = o − 1
26:    end while
27:    FinalizeEvidence(ℓ)      ▷ ignore tallied evidence if ℓ is small or evidence conflicts
28: end procedure
```

When encoding M or MA, the elements of X are simply tuples. A complication exists for MA^w, since elements are lists of up to w tuples. In this case, we keep an additional bit-vector B of size $|X|$ where 1s denote left-hand boundaries in X that correspond to runs in A. E and B can be used together to access an element in A (Algorithm 2).

Extracting markers from VCF

A Variant Calling Format (VCF) [9] file is used to encode a collection of haplotypes with the variants arranged in order according to a reference genome. In the case of human and other diploid genomes, haplotypes are grouped as pairs. We refer to such a collection of haplotypes as a "panel" and a single haplotype as a "panelist." An VCF entry encodes a variant as a tuple consisting of a chromosome, offset, the allele found in the reference, the alternate allele found in one or more panelists, and a sequence of flags indicating whether each panelist has the reference or alternate version. We start from a VCF file to determine how to populate the marker arrays M, MA and/or MA^w, as well as the edit array E.

Algorithm 2 Access the contents of $A[i]$ in the case where entries of A can be lists.

procedure SPARSEACCESS(S, E, X, B, i)
  if $S.\text{rank}_1(i+1) = E.\text{rank}_1(i)$ **then**
    return $\emptyset$
  end if
  $e = E.\text{rank}_1(i)$
  $j = B.\text{select}_1(e)$
  $k = B.\text{select}_1(e+1)$
  return $\{X[j], ..., X[k-1]\}$
end procedure

A single element of M is a tuple (r, e), where r is an offset in T_0 and e is the edit operation describing how the sequence differs from the reference. As a practical matter, we represent these tuples in a different way that more closely resembles the corresponding VCF records. Specifically, a marker is encoded in a 64-bit word divided into three fields. First, a 12 bit field identifies the chromosome containing the marker. The chromosome ordering is given at the beginning of the VCF file in the "header" section. For example, if "chr1" is the first chromosome in the header, then this chromosome is encoded as `0x000` (using hexadecimal), and if "chr2" is the second chromosome, it is encoded as `0x001`. Second is a 54 bit field encoding the marker's offset within the chromosome. Third is a 4-bit field storing which version of the variant is present, with 0 indicating the reference allele, 1 indicating the 1st alternate allele, 2 the second alternate allele, etc. This 64-bit representation allows for compact storage of markers and easier random access to the marker array.

Diploid genotyping

In a diploid genome, it is possible for both alleles to occur, i.e. for the genotype to be heterozygous. We use an existing approach [21] to compute genotype likelihoods considering all possible diploid genotypes: homozygous reference (2 reference alleles), homozygous alternate (2 alternate alleles), or heterozygous (1 reference, 1 alternate). Let $g \in \{0, 1, 2\}$ denote the number of reference-allele copies at the marked site; e.g. $g = 1$ corresponds to a heterozygous site and $g = 2$ to a homozygous reference site. Let l be the number of times the reference allele was observed in the reads overlapping a particular marked site and let k be the count of all alleles (reference or alternate) observed. Let ϵ be the sequencing error rate. We calculate the genotype likelihood as follows, adopting equation 2 of [21] while setting the ploidy to 2 and adopting a global rather than a per-base error rate:

$$\mathcal{L}(g) = \frac{1}{2}^k [(2-g)\epsilon + g(1-\epsilon)]^l [(2-g)(1-\epsilon) + g\epsilon]^{k-l}$$

To choose the most likely genotype g_{max}, we compute:

$$g_{max} = \text{argmax}_{g \in \{0,1,2\}} \mathcal{L}(g)$$

By default, `rowbowt` uses $\epsilon = 0.01$.

Implementation details

The code for constructing the marker array is implemented in the `pfbwt-f` spoftware package, with repository at `https://github.com/alshai/pfbwt-f`. This repository also contains an efficient implementation of the prefix-free-parse BWT construction algorithm [19]. This software is written in C++17, uses the open-source MIT license, and builds on the Succinct Data Structure Library (SDSL) v3.0 [16].

For querying the marker array, we use the `rowbowt` implementation at `https://github.com/alshai/rowbowt`. This repository contains the open source C++17 implementation of `rowbowt`, distributed under the MIT license. It is also a library, containing algorithms for building and querying indexes containing various structures discussed here, including the run-sampled suffix array, marker array, and others.

4 Results

We evaluated the efficiency and accuracy of our marker-array method for compiling genotype evidence. We first generated multiple series of `rowbowt` indexes covering various settings for three parameters: the window size w for the smeared marker array MA^w, the number of haplotypes indexed, and the minimum allele frequency for marked alleles. The `rowbowt` index consisted of three components: the run-length encoded BWT, the run-sampled suffix array, and the marker array. While we built the sampled suffix array for these experiments, the standard marker-array-based method in `rowbowt` does not require this array.

We generated indexes for collections of $200, 400, 800$, or 1000 human chromosome-21 haplotypes from the 1000 Genomes Phase 3 reference panel [2] based on the $GRCh37$ reference. We generated two sets of indexes: one where the marker array marks all polymorphic sites regardless of frequency (denoted "$AF > 0$"), and another where the marker array marks only those sites where the less common allele occurs in greater than 1% of haplotypes, i.e. has allele frequency over 1% (denoted "$AF > 0.01$"). In all cases, the marker array window size w was set to 19. Each haplotype collection was drawn from a random subset of 500 individuals from the 1000 Genomes Phase 3 panel. The $AF > 0$ panel of 500 haplotypes contained $1,097,388$ polymorphic sites. The $AF > 0.01$ panel of the same haplotypes contained $193,438$ polymorphic sites with allele frequency over 1%. We also included the GRCh37 reference sequence, consisting of all reference alleles, in each collection, corresponding to the reference sequence called T_0 above.

We generated a series of indexes with window size $w \in \{13, 15, 17, 19, 21, 23, 25\}$. We generated two such series: one with no minimum allele frequency ($AF > 0$) and another with a 1% minimum frequency ($AF > 0.01$). Each index was over the same set of 100 haplotypes.

Index size

We measured the size of the three main components of the `rowbowt` index: the augmented marker array, the run-length encoded BWT (RLE BWT) [3] and the run-sampled suffix array ("r-index SA") [14]. Figure 2 plots this measurement for collections of 200, 400, 800 and 1,000 haplotypes for both $AF > 0$ and $AF > 0.01$. All grow linearly with the number haplotypes grows, as expected. For $AF > 0$, the augmented marker array is consistently larger than the run-sampled suffix array ("r-index SA"). For $AF > 0.01$, the augmented marker array is much smaller, approaching the size of the RLE BWT. The $AF > 0$ array is larger because it contains polymorphic sites with infrequent alleles; about 85% of the

marked sites in the $AF > 0$ array have allele frequency under 1%. Further, rare alleles are less likely to form long runs in the augmented marker array, negatively affecting run-length compression.

In the right portion of Figure 2, the RLE BWT and r-index SA have constant size because the w parameter does not affect those data structures. In the left portion of Figure 2, showing size as a function of number of haplotypes, the augmented marker array is almost always larger than the r-index SA for $AF > 0$ as opposed to $AF > 0.01$, except at $w = 13$. The slope of the array size is smaller for $AF > 0.01$ than for $AF > 0$.

Overall, both the value of w and the number of haplotypes in the index cause the augmented marker array to increase in size, but the inclusion of rare alleles ($< 1\%$ allele frequency) has the largest effect on its size.

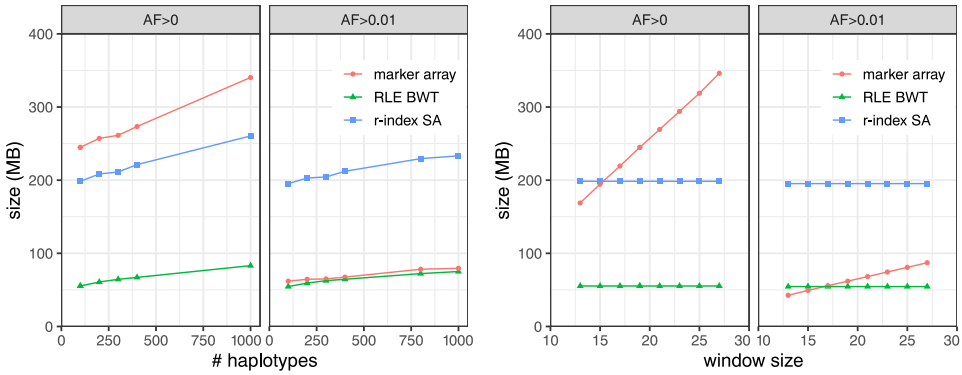


Figure 2 Left: Size of `rowbowt` data structures as a function of the number of haplotypes indexed, and with $w = 19$. "Marker array" refers to the augmented marker array, MA^w. Right: Size of `rowbowt` data structures as a function of the "smearing" window size w, with number of haplotypes fixed at 100. Separate results are shown for when there is no minimum allowed allele frequency ($AF > 0$) and when the minimum frequency is 1% ($AF > 0.01$).

Query time

We next measured query time for the augmented marker array strategy versus the locate-query strategy which uses the run-sampled suffix array. 150bp simulated reads of 25x coverage were generated for one haplotype of HG01498, an individual that is part of the 1000-Genomes panel, but which we excluded from all our indexes. We simulated reads using Mason 2 `mason_simulator` [24] with default options.

In the case of the marker-array strategy, we measured the time required to analyze the reads using the algorithm described above in "Genotyping a read." In the case of the locate-query strategy, the MA^w query was replaced with a two-step process that first performed a locate query with respect to the run-sampled suffix array, then performed a lookup in the M array. To enable this mode, we further augmented the r-index with a representation of M using the sparse encoding described above. To emphasize: the `rowbowt` strategy does not require the run-sampled suffix array or the M array; the MA^w effectively replaces them both.

We repeatedly sampled 10,000 simulated reads and recorded the mean query time over 10 trials. As seen in Figure 3 the augmented marker-array method (labeled "marker") was consistently faster than locate method. This was true for all allele frequencies and window sizes tested. We found that the effect of w and allele-frequency cutoff was more pronounced with the larger reference panel $AF > 0$. For the smaller panel ($AF > 0.01$), query time was mostly invariant to both window size and allele frequency.

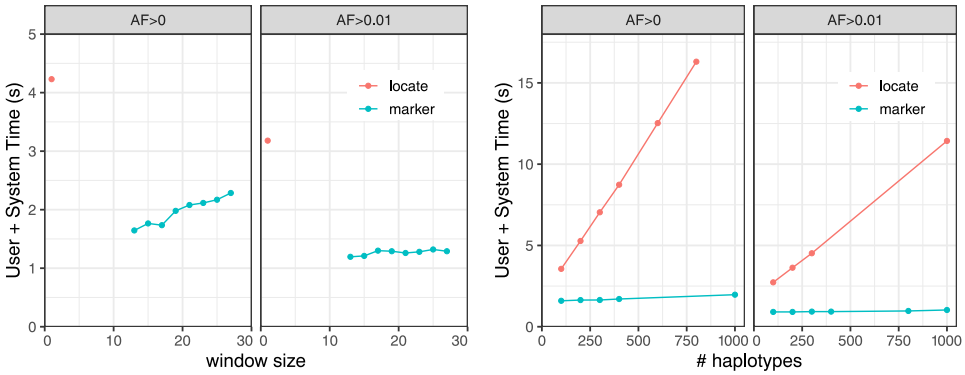


Figure 3 Mean time over 10 trials of aligning 10,000 simulated reads from HG01498 against the augmented marker array (marker) and the r-index suffix array (locate). Experiments are repeated for marker collections including all alleles ($AF > 0$) and for alleles having frequency at least 1% ($AF > 0.01$). Left: The experiment is repeated for various window sizes w, and for 100 haplotypes. Right: The experiment is repeated for different numbers of indexed haplotypes, with $w = 19$.

Genotyping accuracy

We next measured the accuracy of the genotype information gathered using the augmented marker array. We simulated sequencing reads from one haplotype of HG01498 to an average depth of 25-fold coverage. Individual HG01498 was excluded from the indexes. As our "truth" set for evaluation, we use the variant calls in the 1000 Genomes project callset for the same haplotype we simulated from. For simplicity, this experiment treats the genome as haploid. Experiments in the next section will account for the diploid nature of human genomes.

A single marked site can have conflicting evidence, due, for instance, to mismapped reads or sequencing errors. For this evaluation, we make calls simply by finding the frequently observed allele at the site. We ignore any instances of alleles other than the ones noted in the VCF file as the reference and alternate alleles. If the reference and alternate alleles have equal evidence, the reference allele is called.

We calculate precision and recall according to the following formulas. Here, the positive class consists of marked sites that truly have the alternate allele, while the negative class consists of marked sites that truly have the reference allele. We measure:

$$\text{Precision} = \frac{\text{TP}s}{\text{TP}s + \text{FP}s} \qquad \text{Recall} = \frac{\text{TP}s}{\text{TP}s + \text{FN}s}$$

Where TP stands for True Positive, FN stands for False Negative, etc.

Figure 4 shows precision and recall with respect to the number of haplotypes in the index and the minimum allele frequency of the haplotype collection. We observed that the $AF > 0.01$ indexes generally had better precision compared to the $AF > 0$ indexes, though at the expense of recall. Precision and recall generally improve with the addition of more haplotypes to the index. The augmented marker array has similar recall to the locate procedure across all haplotype sizes at the loss of precision. When rare variants are removed from the index ($AF > 0.01$), the gap in precision between the marker array and the locate procedure lessens. This mild (less than 0.1%) loss of precision is expected since algorithm described above in "Genotyping reads" is still prone to some false positives in the earlier part of the extension process.

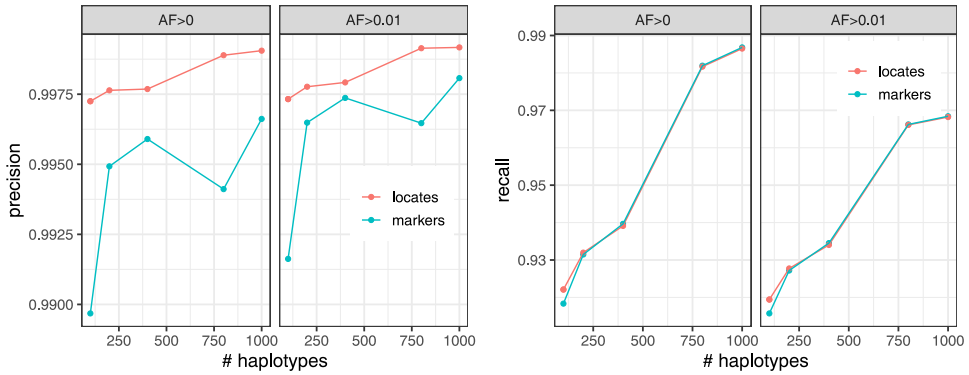


Figure 4 Precision (left) and recall (right) of the calls made when querying 25x simulated reads from HG01498 against the augmented marker array (marker) and the r-index (locate). Stratified by minimum allowed allele frequency (AF) in the index.

Diploid genotyping assessment

To assess diploid genotyping accuracy, we used data from the Human Genome Structural Variation Consortium (HGSVC) [6, 12]. The HGSVC called both simple and complex genetic variants across a panel of 64 human genomes. Calls were made with respect to the GRCh38 primary assembly [26]. For input reads, we subsampled reads from a 30-fold average coverage PCR-free read set provided by the 1000 Genomes Project [2] (accession SRR622457). To create more challenging scenarios for the genotypers, we randomly subsampled the read sets to make smaller datasets of 0.01, 0.05, 0.1, 0.5, 1, 2 and 5-fold average coverage.

We assessed three genotyping methods. The first (`bowtie2_bcftools`) used Bowtie 2 [20] v2.4.2 to align reads to a standard linear reference genome, then used BCFtools v1.13 to call variants (i.e. genotypes) at the marked sites [21]. The second method used the graph-based genotyper `bayestyper` v1.5. The third was `rowbowt`.

Prior to applying `bayestyper`, we built a `bayestyper`-compatible VCF file containing all relevant variants from the HGSVC haplotype panel. For `rowbowt`, we created a `rowbowt` index from the genomes in the HGSVC haplotype panel. In both cases we excluded NA12878's haplotypes from the panel prior to building the index.

When evaluating, we stratified variants by complexity: the "snp" category includes single-nucleotide substitutions, "indel" includes indels no more than 50bp long, and "sv" includes insertion or deletions longer than 50bp, and "all" includes all variant types. More complex structural variants like inversions and chromosomal rearrangements are ignored.

We analyzed the accuracy of `rowbowt`'s diploid genotypes in two ways. First we considered allele-by-allele precision and recall, considering the alternate (ALT) allele calls to be the positive class. Specifically, every diploid genotype called by a method is considered as a pair of individual allele calls. If a given allele call is an alternate (ALT) allele and there is at least one ALT allele present in the true diploid genotype at that site, it counts as a true positive (TP). If the given allele is a reference allele (REF) and there is at least one REF allele in the true diploid genotype, this is a true negative (TN). If the given allele is an ALT but the true genotype is homozygous REF, we count it as a false positive (FP). Finally, if the given allele is REF but the true genotype is homozygous ALT, this is a a false negative (FN).

Second, we considered precision and recall with respect to sites that were either truly heterozygous or called heterozygous. If a heterozygous call made by a method is truly heterozygous, this was counted as a true positive (TP). False positives, false negatives, and true negatives are defined accordingly.

As seen in Figure 5, `rowbowt`'s ALT and HET precision were generally the highest of all the methods across all variant categories, though `bayestyper` sometimes achieved higher ALT/HET precision for indels in the higher-coverage datasets. `rowbowt`'s ALT recall is also higher than the other methods, except for some of the lower coverage measurements in the "sv" category, where `bayestyper` achieved higher ALT recall.

Computational performance

We compared the time and memory usage of each genotyping method, dividing each method into Alignment and Genotyping phases. For each phase, we measured both the wall-clock time elapsed and maximum memory ("maximum resident set") used. Both were measured with the Snakemake tool's `benchmark` directive [22].

Since the tools operate somewhat differently, we define the "Alignment" step differently in each case. For `bowtie2_bcftools`, we define the Alignment step as the process of using `bowtie2` to align sequencing reads to the linear reference genome. For `rowbowt`, we defined the Alignment step as the process of using the `rb_markers` command to genotype the reads using the algorithm described in Methods. For `bayestyper`, we defined the Alignment step as the typical 3-step process of: (a) using the KMC3 software tool to count k-mers in the input reads, (b) using the `bayestyper makeBloom` command to convert k-mer counts to Bloom filters for each sample, and (c) using the `bayestyper cluster` command to identify variant clusters. 16 threads were used during the Alignment phase for `bowtie2_bcftools` and `rowbowt`, while 32 threads were used for `bayestyper`.

For `bowtie2_bcftools`, we define the Genotyping step as the process of using `bcftools call` to call variants from the BAM file output by `bowtie2`. For `rowbowt` We define the Genotyping phase as the process of running the `vc_from_markers.py` script on the output from `rb_markers`. For `bayestyper`, we define the genotyping step as the process of running the `bayesTyper genotype` command. The Genotype phases for both `bowtie2_bcftools` and `rowbowt` do not support multi-threading, so a single thread was used. For the `bayestyper` Genotype phase we used 16 threads.

Figure 6 shows the time taken and peak memory footprint for each method and each phase. We observed that `rowbowt` was consistently faster than the other methods, sometimes by a large margin. We also observed that while `rowbowt` has a higher memory footprint compared to the `bowtie2_bcftools` method, it uses substantially less memory than `bayestyper`, the other pangenome-based method.

5 Discussion

We proposed a family of novel marker array structures, M, MA and MA^w that, together with a pangenome index like the r-index, allow for rapid and memory-efficient genotyping with respect to large pan-genome references. The augmented marker array is smaller and faster to query than the run-sampled suffix array – the usual way to establish where matches fall when querying a run-length compressed index – especially when we limit the set of markers to just alleles at frequency 1% or higher. We further showed that the augmented marker array can replace the sampled suffix array in simple genotyping experiments with moderate sacrifice of precision, and that a marker array based genotyping method outperforms the graph-based Bayestyper method.

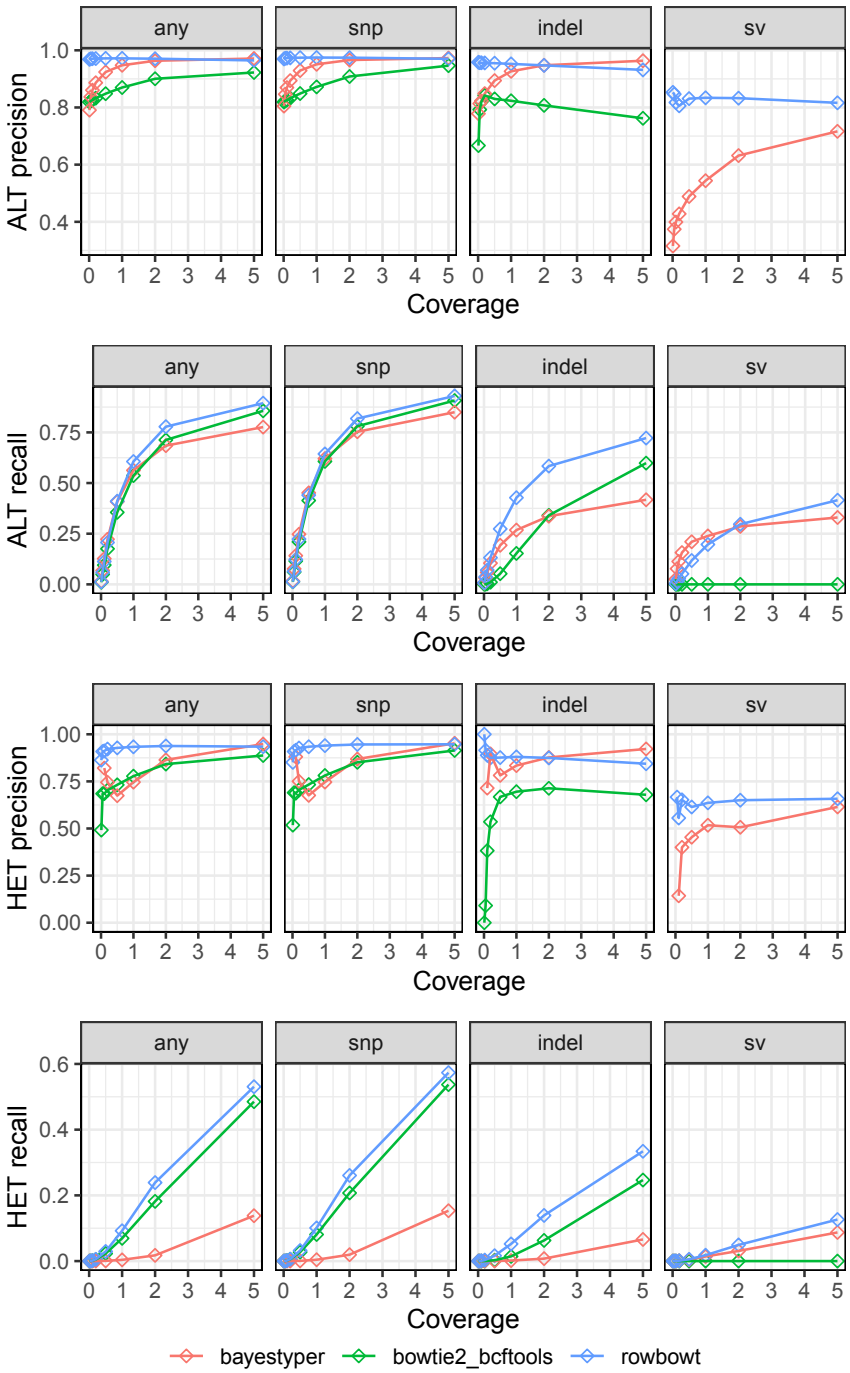


Figure 5 Precision and recall for the three tested genotyping methods, both at the level of individual alleles (ALT precision/recall) and at the level of heterozygous variants (HET precision/recall). Note that the bowtie2_bcftools approach is generally unable to align reads across variants in the "sv" category, leading to low precision and recall.

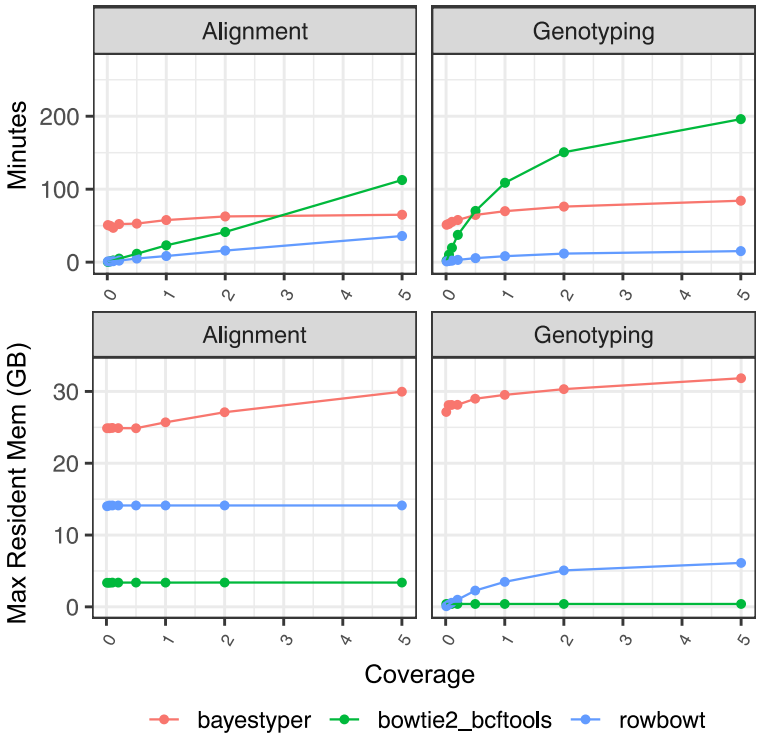


Figure 6 Time for each phase of the impute-first workflow for three methods of alignment/genotyping.

Pan-genome indexes allow for rapid analysis of reads while avoiding reference bias. The indexes used in our experiments consisted of many (up to 65) haplotypes, with none having a higher priority over the others, except in the sense that results were expressed in terms of the standard reference. Our approach preserves all linkage disequilibrium information. This is in contrast to some graph-indexing approaches, which might consider all possible combinations of nearby alleles to be "valid," even if most combinations never co-occur in nature.

While we examined only simple structural variants in the form of insertions and deletions longer than 50 bp, the genotyping method is readily extensible to more complex differences as well. Indeed, as long as we can mark the base or bases just to the left of the variant, we can mark any variant in a way that we can later genotype

The `rowbowt` method can lead to future methods that use information about genotypes to build a personalized reference genome, containing exactly the genotyped alleles. Alignment to a personalized reference have been shown previously to be the best way to avoid reference bias, even more effective than the best pangenome methods [7, 23].

References

1 O. Ahmed, M. Rossi, S. Kovaka, M. C. Schatz, T. Gagie, C. Boucher, and B. Langmead. Pan-genomic matching statistics for targeted nanopore sequencing. *iScience*, 24(6):102696, June 2021.

2 A. Auton, L. D. Brooks, R. M. Durbin, E. P. Garrison, H. M. Kang, J. O. Korbel, J. L. Marchini, S. McCarthy, G. A. McVean, G. R. Abecasis, et al. A global reference for human genetic variation. *Nature*, 526(7571):68–74, October 2015.

3 D. Belazzougui, F. Cunial, T. Gagie, N. Prezza, and M. Raffinot. Flexible Indexing of Repetitive Collections. In Jarkko Kari, Florin Manea, and Ion Petre, editors, *Unveiling Dynamics and Complexity*, volume 10307, pages 162–174. Springer International Publishing, Cham, 2017. Series Title: Lecture Notes in Computer Science.

4 D. Y. Brandt, V. R. Aguiar, B. D. Bitarello, K. Nunes, J. Goudet, and D. Meyer. Mapping Bias Overestimates Reference Allele Frequencies at the HLA Genes in the 1000 Genomes Project Phase I Data. *G3 (Bethesda)*, 5(5):931–941, March 2015.

5 M. Burrows and D.J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

6 M. J. P. Chaisson, A. D. Sanders, X. Zhao, A. Malhotra, D. Porubsky, T. Rausch, E. J. Gardner, O. L. Rodriguez, L. Guo, R. L. Collins, et al. Multi-platform discovery of haplotype-resolved structural variation in human genomes. *Nat Commun*, 10(1):1784, April 2019.

7 N. C. Chen, B. Solomon, T. Mun, S. Iyer, and B. Langmead. Reference flow: reducing reference bias using multiple population genomes. *Genome Biol*, 22(1):8, January 2021.

8 S. Chen, P. Krusche, E. Dolzhenko, R. M. Sherman, R. Petrovski, F. Schlesinger, M. Kirsche, D. R. Bentley, M. C. Schatz, F. J. Sedlazeck, and M. A. Eberle. Paragraph: a graph-based structural variant genotyper for short-read sequence data. *Genome Biol*, 20(1):291, December 2019.

9 P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks, M. A. DePristo, R. E. Handsaker, G. Lunter, G. T. Marth, S. T. Sherry, et al. The variant call format and VCFtools. *Bioinformatics*, 27(15):2156–2158, August 2011.

10 R. W. Davies, M. Kucka, D. Su, S. Shi, M. Flanagan, C. M. Cunniff, Y. F. Chan, and S. Myers. Rapid genotype imputation from sequence with reference panels. *Nat Genet*, 53(7):1104–1111, July 2021.

11 L. Denti, M. Previtali, G. Bernardini, A. Schönhuth, and P. Bonizzoni. MALVA: Genotyping by Mapping-free ALlele Detection of Known VAriants. *iScience*, 18:20–27, August 2019.

12 P. Ebert, P. A. Audano, Q. Zhu, B. Rodriguez-Martin, D. Porubsky, M. J. Bonder, A. Sulovari, J. Ebler, W. Zhou, R. Serra Mari, et al. Haplotype-resolved diverse human genomes and integrated analysis of structural variation. *Science*, 372(6537), April 2021.

13 P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

14 T. Gagie, G. Navarro, and N. Prezza. Optimal-Time Text Indexing in BWT-runs Bounded Space. In *Proceedings of the 29th Annual Symposium on Discrete Algorithms (SODA)*, pages 1459–1477, 2018.

15 E. Garrison, J. Sirén, A. M. Novak, G. Hickey, J. M. Eizenga, E. T. Dawson, W. Jones, S. Garg, C. Markello, M. F. Lin, B. Paten, and R. Durbin. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nat Biotechnol*, 36(9):875–879, October 2018.

16 S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014. doi:10.1007/978-3-319-07959-2_28.

17 T. Günther and C. Nettelblad. The presence and impact of reference bias on population genomic studies of prehistoric human populations. *PLoS Genet*, 15(7):e1008302, July 2019.

18 C. Kim, H. Guo, W. Kong, R. Chandnani, L. S. Shuang, and A. H. Paterson. Application of genotyping by sequencing technology to a variety of crop breeding programs. *Plant Sci*, 242:14–22, January 2016.

19 A. Kuhnle, T. Mun, C. Boucher, T. Gagie, B. Langmead, and G. Manzini. Efficient Construction of a Complete Index for Pan-Genomics Read Alignment. *J Comput Biol*, 27(4):500–513, April 2020.

20 B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nat Methods*, 9(4):357–359, March 2012.

21 H. Li. A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data. *Bioinformatics*, 27(21):2987–2993, November 2011.

22 F. Mölder, K. P. Jablonski, B. Letcher, M. B. Hall, C. H. Tomkins-Tinch, V. Sochat, J. Forster, S. Lee, S. O. Twardziok, A. Kanitz, A. Wilm, M. Holtgrewe, S. Rahmann, S. Nahnsen, and J. Köster. Sustainable data analysis with Snakemake. *F1000Res*, 10:33, 2021.

23 J. Pritt, N. C. Chen, and B. Langmead. FORGe: prioritizing variants for graph genomes. *Genome Biol*, 19(1):220, December 2018.

24 K. Reinert, T. H. Dadi, M. Ehrhardt, H. Hauswedell, S. Mehringer, R. Rahn, J. Kim, C. Pockrandt, J. Winkler, E. Siragusa, G. Urgese, and D. Weese. The SeqAn C++ template library for efficient sequence analysis: A resource for programmers. *J Biotechnol*, 261:157–168, November 2017.

25 M. Rossi, M. Oliva, B. Langmead, T. Gagie, and C. Boucher. MONI: A Pangenomic Index for Finding Maximal Exact Matches. *J Comput Biol*, 29(2):169–187, February 2022.

26 V. A. Schneider, T. Graves-Lindsay, K. Howe, N. Bouk, H. C. Chen, P. A. Kitts, T. D. Murphy, K. D. Pruitt, F. Thibaud-Nissen, D. Albracht, et al. Evaluation of GRCh38 and de novo haploid genome assemblies demonstrates the enduring quality of the reference assembly. *Genome Res*, 27(5):849–864, May 2017.

27 A. Shajii, D. Yorukoglu, Y. William Yu, and B. Berger. Fast genotyping of known SNPs through approximate k-mer matching. *Bioinformatics*, 32(17):i538–i544, September 2016.

28 R. M. Sherman, J. Forman, V. Antonescu, D. Puiu, M. Daya, N. Rafaels, M. P. Boorgula, S. Chavan, C. Vergara, V. E. Ortega, et al. Assembly of a pan-genome from deep sequencing of 910 humans of African descent. *Nat Genet*, 51(1):30–35, January 2019.

29 J. A. Sibbesen, L. Maretty, and A. Krogh. Accurate genotyping across variant classes and lengths using variant graphs. *Nat Genet*, 50(7):1054–1059, July 2018.

Suffix Sorting via Matching Statistics

Zsuzsanna Lipták
Department of Computer Science, University of Verona, Italy

Francesco Masillo
Department of Computer Science, University of Verona, Italy

Simon J. Puglisi
Helsinki Institute for Information Technology (HIIT), Finland
Department of Computer Science, University of Helsinki, Finland

Abstract

We introduce a new algorithm for constructing the generalized suffix array of a collection of highly similar strings. As a first step, we construct a compressed representation of the matching statistics of the collection with respect to a reference string. We then use this data structure to distribute suffixes into a partial order, and subsequently to speed up suffix comparisons to complete the generalized suffix array. Our experimental evidence with a prototype implementation (a tool we call `sacamats`) shows that on string collections with highly similar strings we can construct the suffix array in time competitive with or faster than the fastest available methods. Along the way, we describe a heuristic for fast computation of the matching statistics of two strings, which may be of independent interest.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Generalized suffix array, matching statistics, string collections, compressed representation, data structures, efficient algorithms

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.20

Supplementary Material *Software (Source Code)*: `https://github.com/fmasillo/sacamats`

Funding *Simon J. Puglisi*: Academy of Finland grants 339070 and 351150

1 Introduction

Suffix sorting – the process of ordering all the suffixes of a string into lexicographical order – is the key step in construction of suffix arrays and the Burrows-Wheeler transform, two of the most important structures in text indexing and biological sequence analysis [21, 15, 1]. As such, algorithms for efficient suffix sorting have been the focus of intense research since the early 1990s [16, 23].

With the rise of pangenomics, there is an increased demand for indexes that support fast pattern matching over collections of genomes of individuals of the same species (see, e.g., [8, 24, 25]). With pangenomic collections constantly growing and changing, construction of these indexes – and in particular suffix sorting – is a computational bottleneck in many bioinformatics pipelines. While traditional and well-established suffix sorting tools such as `divsufsort` [17, 7] and `sais` [18, 20] can be applied to these collections, specialised algorithms for collections of similar sequences, perhaps most notably the so-called `BigBWT` program [3], are beginning to emerge.

In this paper we describe a suffix sorting algorithm specifically targeted to collections of highly similar genomes that makes use of the *matching statistics*, a data structure due to Chang and Lawler, originally used in the context of approximate pattern matching [5]. The core device in our suffix sorting algorithm is a novel compressed representation of the matching statistics of every genome in the collection with respect to a designated reference genome, that allows determining the relative order of two arbitrary suffixes, from any of the genomes, efficiently. We use this data structure to drive a suffix sorting algorithm that has a small working set relative to the size of the whole collection, with the aim of increasing

locality of memory reference. Experimental results with a prototype implementation show the new approach to be faster or competitive with state-of-the-art methods for suffix array construction, including those targeted at highly repetitive data. We also provide a fast, practical algorithm for matching statistics computation, which is of independent interest.

The remainder of this paper is structured as follows. The next section sets notation and defines basic concepts. In Section 3 we describe a compressed representation of the matching statistics and a fast algorithm for constructing it. Section 4 then describes how to use the compressed matching statistics to determine the relative lexicographic order of two arbitrary suffixes of the collection. Section 5 describes a complete suffix sorting algorithm. We touch on several implementation details in Section 6, before describing experimental results in Section 7. Reflections and avenues for future work are then offered.

2 Basics

A string T over an ordered alphabet Σ, of size $|\Sigma| = \sigma$, is a finite sequence $T = T[1..n]$ of characters from Σ. We use the notation $T[i]$ for the ith character of T, $|T|$ for its length n, and $T[i..j]$ for the substring $T[i] \cdots T[j]$; if $i > j$ then $T[i..j] = \varepsilon$, where ε is the empty string. The substring (or factor) $T[i..] = T[i..n]$ is called the ith suffix, and $T[..i] = T[1..i]$ the ith prefix of T. We assume throughout that the last character of each string is a special character \$, not occurring elsewhere in T, which is set to be smaller than every character in Σ.

Given a string T, the *suffix array* SA is a permutation of the index set $\{1, \ldots, n\}$ defined by: $SA[i] = j$ if the jth suffix of T is the ith in lexicographic order among all suffixes of T. The *inverse suffix array* ISA is the inverse permutation of SA. The *LCP-array* is given by: $LCP[1] = 0$, and for $i \geq 2$, $LCP[i]$ is the length of the longest common prefix (lcp) of the two suffixes $T[SA[i-1]..]$ and $T[SA[i]..]$ (which are consecutive in lexicographic order). A variant of the LCP array is the *permuted LCP-array*, $PLCP$, defined as $PLCP[i] = LCP[ISA[i]]$, i.e. the lcp values are stored in text order, rather than in SA order. We further define $LCPsum(T) = \sum_{i=1}^{|T|} LCP[i]$. $LCPsum$ can be used as a measure of repetitiveness of strings, since the number of distinct substrings of T equals $(|T|^2 + |T|)/2 - LCPsum(T)$. All these arrays can be computed in linear time in $|T|$, see e.g. [20, 11].

Given the suffix array SA of T and a substring U of T, the indices of all suffixes which have U as prefix appear consecutively in SA. We refer to this interval as *U-interval*: the U-interval is $SA[s..e]$, where $\{SA[s], SA[s+1], \ldots, SA[e-1], SA[e]\}$ are the starting positions of the occurrences of U in T.

Let $\mathcal{C} = \{S_1, \ldots, S_m\}$ be a collection of strings (a set or multiset). The *generalized suffix array* GSA of $\mathcal{C}$ is defined as $GSA[i] = (d, j)$ if $S_d[j..]$ is the ith suffix in lexicographic order among all suffixes of the strings from $\mathcal{C}$, where ties are broken by the document index d. The GSA can be computed in time $\mathcal{O}(N)$, where N is the total length of strings in $\mathcal{C}$ [21].

Let R and S be two strings. The *matching statistics of S with respect to R* is an array MS of length $|S|$, defined as follows. Let U be the longest prefix of suffix $S[i..]$ which occurs in R as a substring, where the end-of-string character \# of R is assumed to be different from, and smaller than that of S. Then $MS[i] = (p_i, \ell_i)$, where $p_i = -1$ if $U = \varepsilon$, and p_i is an occurrence of U in R otherwise, and $\ell_i = |U|$. (Note that p_i is not unique in general.) We refer to U as the *matching factor*, and to the character c immediately following U in S as the *mismatch character*, of position i. For a collection $\mathcal{C} = \{S_1, \ldots, S_m\}$ and a string R, the matching statistics of $\mathcal{C}$ w.r.t. R is simply the concatenation of MS_i's, where MS_i is the matching statistics of S_i w.r.t. R. We will discuss matching statistics in more detail in Section 3.

For an integer array A of length n and an index i, the previous and next smaller values, *PSV* resp. *NSV*, are defined as $PSV(A,i) = \max\{i' < i : A[i'] < A[i]\}$ resp. $NSV(A,i) = \min\{i' > i : A[i'] < A[i]\}$. Note that *PSV* resp. *NSV* is not defined for $i = 1$ resp. $i = n$. In $O(n)$ preprocessing of A, a data structure of size $n\log_2(3+2\sqrt{2}) + o(n)$ bits can be built that supports answering arbitrary *PSV* and *NSV* queries in constant time per query [6].

Let X be a finite set of integers. Given an integer x, the predecessor of x, *pred*(x) is defined as the largest element smaller than x, i.e. $pred_X(x) = \max\{y \in X \mid y \leq x\}$. Using the y-fast trie data structure of Willard [27] allows answering predecessor queries in $O(\log\log|X|)$ time using $O(|X|)$ space.

We are now ready to state our problem:

Problem Statement: Given a string collection $\mathcal{C} = \{S_1, \ldots, S_m\}$ and a reference string R, compute the generalized suffix array *GSA* of $\mathcal{C}$.

We will denote the length of R by n and the total length of strings in the collection by $N = \sum_{d=1}^{m} |S_d|$. As before, we assume that the end-of-string character # of R is strictly smaller than those of the strings in the collection $\mathcal{C}$. We are interested in those cases where $LCPsum_R$ is small and the strings in $\mathcal{C}$ are very similar to R. If no reference string is given in input, we will take S_1 to be the reference string by default.

2.1 Efficient suffix array construction

Currently, the best known and conceptually simplest linear-time suffix array construction algorithm is the SAIS algorithm by Nong et al. [20]. It cleverly combines, and further develops, several ideas used by previous suffix array construction algorithms, among these *induced sorting*, and use of a so-called *type array*, already used in [9, 12] (see also [23]).

Nong et al.'s approach can be summarized as follows: assign a type to each suffix, sort a specific subset of suffixes, and compute the complete suffix array by inducing the order of the remaining suffixes from the sorted subset. There are three types of suffixes, one of which constitutes the subset to be sorted first.

The definition of types is as follows (originally from [12], extended in [20]): Suffix i is *S-type* (smaller) if $T[i..] < T[i+1..]$, and *L-type* (larger) if $T[i..] > T[i+1..]$. An S-type suffix is *S^*-type* if $T[i..]$ is S-type and $T[i-1..]$ is L-type. It is well known that assigning a type to each suffix can be done with a back-to-front scan of the text in linear time.

Now, if the relative order of the S^*-suffixes is known, then that of the remaining suffixes can be induced with two linear scans over the partially filled-in suffix array: the first scan to induce L-type suffixes, and the second to induce S-type suffixes. For details, see [20] or [21].

Another ingredient of SAIS, and of several other suffix array construction algorithms, is what we term the *metacharacter method*. Subdivide the string T into overlapping substrings, show that if two suffixes start with the same substring, then their relative order depends only on the remaining part; assign metacharacters to these substrings according to their rank (w.r.t. the lexicographic order, or some other order, depending on the algorithm), and define a new string on these metacharacters. Then the relative order of the suffixes of the new string and the corresponding suffixes starting with these specific substrings will coincide. In SAIS [20], so-called LMS-substrings are used, while a similar method is applied in prefix-free-parsing (PFP) [3]. Here we will apply this method using substrings starting in special positions which we term insert-heads, see Sections 4 and 5 for details.

3 Compressed matching statistics

Let R, S be two strings over Σ and MS be the matching statistics of S w.r.t. R. Let $MS[i] = (p_i, \ell_i)$. It is a well known fact that if $\ell_i > 0$, then $\ell_{i+1} \geq \ell_i - 1$. This can be seen as follows. Let U be the matching factor of position i, and p_i an occurrence of U in R. Then $U' = U[2..\ell_i]$ is a prefix of $S[i+1..]$ of length $\ell_i - 1$, which occurs in position $p_i + 1$ of R.

Let us call a position j a *head* if $\ell_j > \ell_{j-1} - 1$, and a sequence of the form $(x, x-1, x-2, \ldots)$, of length at most $x - 1$, a *decrement run*, i.e. each element is one less than the previous one. Using this terminology, we thus have that the sequence $L = (\ell_1, \ell_2, \ldots, \ell_n)$ is a concatenation of decrement runs, i.e. L has the form $(x_1, x_1 - 1, x_1 - 2, \ldots, x_2, x_2 - 1, x_2 - 2, \ldots, \ldots, x_k, x_k - 1, x_k - 2, \ldots)$, with each $x_j = \ell_j$ for some head j. We can therefore store the matching statistics in compressed form as follows:

▶ **Definition 1** (Compressed matching statistics). *Let R, S be two strings over Σ, and MS be the matching statistics of S w.r.t. R. The* compressed matching statistics (*CMS*) of S w.r.t. R *is a data structure storing $(j, MS[j])$ for each head j, and a predecessor data structure on the set of heads H.*

We can use *CMS* to recover all values of *MS*:

▶ **Lemma 2.** *Let $1 \leq i \leq |S|$. Then $MS[i] = (p_j + k, \ell_j - k)$, where $j = pred_H(i)$ and $k = i - j$.*

Proof. Let ℓ_i be the length of the matching factor of i. Since there is a matching factor of length ℓ_j starting in position j in S, this implies that $\ell_i \geq \max(0, \ell_j - k)$. If ℓ_i was strictly greater than $\ell_j - k$, this would imply the presence of another head between j and i, in contradiction to $j = pred_H(i)$. Since an occurrence of the matching factor U_j of j starts in position p_j of R, therefore the matching factor $U' = U[k+1..\ell_j]$ of i has an occurrence at position $p_j + k$. ◀

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
R	T	G	A	T	G	G	C	A	C	A	G	A	T	A	C	T	#
S	G	A	T	G	G	C	A	C	A	T	T	G	A	T	G	G	$
p_i	2	3	4	5	6	7	8	9	12	13	1	2	3	4	5	6	-1
ℓ_i	9	8	7	6	5	4	3	2	2	1	6	5	4	3	2	1	0
head	✓								✓		✓						
q_i	2	3	4	5	6	7	8	9	3	4	1	2	3	4	5	11	17
i-head	✓								✓		✓					✓	✓

Figure 1 Example for the matching statistics and the data for the *CMS* and the *eCMS*. In the first two rows, we give *MS* of S w.r.t. R, where $MS[i] = (p_i, \ell_i)$. In row 3, we mark the heads (for the *CMS*). In rows 4, we give the position q_i, defined by $ip(i)$, i.e. $q_i = SA_R[ip(i)]$, where $ip(i)$ is the insert-point of suffix $S[i..]$ in the suffix array of R. In row 5, we mark the insert-heads (for the *eCMS*).

▶ **Example 3.** Consider the reference $R =$ `TGATGGCACAGATACT` and $S =$ `GATGGCACATTGATGG`. The *CMS* of S w.r.t. R is: $(1, 2, 9), (9, 12, 2), (11, 1, 6)$, see Figure 1.

From Lemma 2 and the properties of the predecessor data structure on the set of heads we get:

	i	SA_R	$R[SA_R[i]..]$
	1	17	`#`
	2	8	`ACAGATACT#`
	3	14	`ACT#`
	4	10	`AGATACT#`
	5	12	`ATACT#`
→	6	3	`ATGGCACAGATACT#`
	7	7	`CACAGATACT#`
	8	9	`CAGATACT#`
	9	15	`CT#`
	10	11	`GATACT#`
	11	2	`GATGGCACAGATACT#`
	12	6	`GCACAGATACT#`
	13	5	`GGCACAGATACT#`
	14	16	`T#`
	15	13	`TACT#`
→	16	1	`TGATGGCACAGATACT#`
	17	4	`TGGCACAGATACT#`

Figure 2 Details of computation of the matching statistics from Figure 1. We highlight in blue the matching factors for the indices $i = 9$ (matching factor `AT`, mismatch character `T`) and 11 (matching factor `TGATGG`, mismatch character \$). The arrows represent the insert-points.

▶ **Proposition 4.** *Let R, S be two strings over Σ. We can store the matching statistics of S w.r.t. R in $\mathcal{O}(\chi)$ space such that any entry $MS[i]$, for $1 \leq i \leq |S|$, can be accessed in $\mathcal{O}(\log\log\chi)$ time, where $\chi = |H|$ is the number of heads.*

For some statistics on the number χ of heads, see the end of Sec. 3.1.

3.1 Enhancing the CMS

Let R, S be two strings over Σ, and MS the matching statistics of S w.r.t. R. We now assume that all characters that occur in S also occur in R (see Sec. 6). Let SA_R be the suffix array of R. For position i of S, let $U \neq \varepsilon$ be the matching factor and c the mismatch character of i. We want to compute the position that the suffix $S[i..]$ would have in SA_R if it was present. To this end, we define the *insert point* of i, $ip(i)$, as follows:

$$ip(i) = \begin{cases} 1 & \text{if } U = \varepsilon, \\ \max\{j \mid U \text{ occurs in } SA_R[j] \text{ and } R[SA_R[j]..] < Uc\} & \text{if this set is non-empty,} \\ \min\{j \mid U \text{ occurs in } SA_R[j]\} & \text{otherwise.} \end{cases}$$

In other words, the insert point is the lexicographic rank, among all suffixes of R, of the next smaller occurrence of U in R if such an occurrence exists, and of the smallest occurrence of U in R otherwise. Note that case 1 (where $U = \varepsilon$) only happens for end-of-string characters. The insert point is well-defined for every i because # is smaller than all other characters, including other end-of-string characters. Observe that the insert point of i always lies within the U-interval of SA_R. For an example, see Fig. 2.

We will later use the insert points to bucket suffixes. First we need to slightly change the definition of our compressed matching statistics. We will add more information to the heads: we add the mismatch character and replace the position entry p_i, which gives just some occurrence of the matching factor, by the specific occurrence q_i given by the insert point. This will imply adding more heads, so our data structure may increase in size.

To this end, we define j to be an *insert-head* if $SA_R[ip(j)] \neq SA_R[ip(j-1)] + 1$. Note that, in particular, all heads are also insert-heads, but it is possible to have insert-heads j which are not heads, namely where $\ell_j = \ell_{j-1} - 1$.

▶ **Definition 5** (Enhanced compressed matching statistics)**.** *Let R, S be two strings over Σ. Define the* enhanced matching statistics *as follows: for each $1 \leq i \leq |S|$, let $ems(i) = (q_i, \ell_i, x_i, c_i)$, where $q_i = SA_R[ip(i)]$, ℓ_i is the length of the matching factor U of i, c_i is the mismatch character, and $x_i \in \{S, L\}$ indicates whether Uc_i is smaller (S) or greater (L) than $R[q_i..]$. The* enhanced compressed matching statistics (*eCMS*) of S w.r.t. R *is a data structure storing $(j, ems(j))$ for each insert-head j, and a predecessor data structure on the set of insert-heads H'.*

▶ **Example 6.** Continuing with Example 3, the enhanced *CMS* of S w.r.t. R is: $(1, 2, 9, L, \mathtt{T})$, $(9, 3, 2, L, \mathtt{T})$, $(11, 1, 6, S, \$)$, $(16, 11, 1, S, \$)$, $(17, 17, 0, L, \$)$, see Figure 1.

We will need some properties of the insert point in the following:

▶ **Observation 7.** *Let $ip(i)$ be the insert point of i, and $ems(i) = (q_i, \ell_i, x_i, c_i)$.*

1. *$ip(i) = ip(i')$ if and only if $q_i = q_{i'}$,*
2. *if $x_i = S$ then $R[SA_R[ip(i) - 1]..] < S[i..] < R[SA_R[ip(i)]..] = R[q_i..]$,*
3. *if $x_i = L$ then $R[q_i..] = R[SA_R[ip(i)]..] < S[i..] < R[SA_R[ip(i) + 1]..]$.*

The enhanced CMS can be used in a similar way as the CMS to recover the enhanced matching statistics (including the matching statistics) of each i. Denote by *i-head*(i) the next insert-head to the left of i, i.e. *i-head*$(i) = \max\{j \leq i \mid j \text{ is an insert-head}\}$. Note that *i-head*$(i) = pred_{H'}(i)$.

▶ **Lemma 8.** *Let $1 \leq i \leq |S|$, let eCMS be the enhanced CMS of S w.r.t. R. Let $j =$ i-head(i), $k = i - j$, and $ems(j) = (q_j, \ell_j, x_j, c_j)$. Then $ems(i) = (q_j + k, \ell_j - k, x_j, c_j)$, and $ip(i) = ISA_R[q_j + k]$. In particular, $q_j + k$ is an occurrence and $\ell_j - k$ is the length of the matching factor of i (in other words, the matching statistics entry $MS[i]$).*

Proof. Analogous to Lemma 2, resp. straightforward from the definitions. ◀

Similarly to the *CMS* (cp. Prop. 4), the enhanced *CMS* allows access to all values for every index i, using space $\mathcal{O}(\chi')$ and time $\mathcal{O}(\log\log\chi')$, where $\chi' = |H'|$ is the number of insert-heads. Again, this is due to the fact that the predecessor data structure on the set H' of insert-heads allows retrieving $pred_{H'}(i) =$ *i-head*(i) in $\mathcal{O}(\log\log|H'|)$ time, and the values of $ems(i)$ can then be computed in $\mathcal{O}(1)$ time (Lemma 8).

We close this subsection by remarking that for a collection of similar genomes, one can expect the number of heads to be small. Indeed, on a 500MB viral genome data set (see Section 7) containing approximately 10,000 SARS-cov2 genomes, we observed the number of heads to be 5,326,226 (100x less than the input size) and the number of insert heads to be 6,537,294.

3.2 Computing the CMS

It is well known that the matching statistics of S w.r.t. R can be computed in time $O(|R| + |S|\log\sigma)$ and $O(|R|)$ space by using, for example, the suffix tree of R, as described in Chang and Lawler's original paper [5]. Since then, several authors have described similar algorithms

for computing matching statistics, all focussed on reducing space requirements via the use of compressed indexes instead of the suffix tree [1, 22, 2]. These algorithms all incur the slowdowns typical of compressed data structures.

In our setting, where end-to-end runtime is the priority, it is the speed at which the matching statistics can be computed (rather than working space) that is paramount. Moreover, because the size of the reference is generally small relative to the total length of all the strings $S_i \in \mathcal{C}$, we have some freedom to use large index data structures on R to compute the matching statistics, without overall memory usage getting out of hand. With these factors in mind, we take the following approach to computing CMS. The algorithm is similar to that of Chang and Lawler, but makes use of array-based data structures rather than the suffix tree.

Recall that, given the suffix array SA_R of string R and a substring Y of R, the Y-interval is the interval $SA_R[s..e]$ that contains all suffixes having Y as a prefix.

▶ **Definition 9** (Right extension and left contraction). *For a character c and a string Y, the computation of the Yc-interval from the Y-interval is called a* right extension *and the computation of the Y-interval from cY-interval is called a* left contraction.

We remark that a left contraction is equivalent to following a (possibly implicit) suffix link in the suffix tree of R and a right extension is a downward movement (either to a child or along an edge) in the suffix tree of R.

Given a Y-interval, because of the lexicographical ordering on the SA_R, we can implement a right extension to a Yc-interval in $O(\log |R|)$ time by using a pair of binary searches (with c as the search key), one to find the lefthand end of the Yc-interval and another to find the righthand end. If a right extension is empty then there are no occurrences of Yc in R, but we can have the binary search return to us the insert point where it would have been in SA_R.

On the other hand, given a cY-interval, $SA_R[s..e]$, we can compute the Y-interval (i.e. perform a left contraction) in the following way. Let the target Y-interval be $SA_R[x..y]$. Observe that both $SA_R[s] + 1$ and $SA_R[e] + 1$ must be inside the Y-interval, $SA_R[x..y]$ – that is, $s' = ISA_R[SA_R[s] + 1] \in [x..y]$ and $e' = ISA_R[SA_R[e] + 1] \in [x..y]$. To finish computing $SA_R[x..y]$ from $SA_R[s'..e']$ there are two cases to consider. Firstly, if $s' = e'$ and $|Y| > LCP_R[s']$, then $SA_R[s']$ is the only occurrence of Y and we are done (Y-interval is a singleton). Alternatively, $s' \neq e'$ and we compute $SA_R[x..y]$ using NSV/PSV queries on LCP_R, in particular $SA_R[x..y] = SA_R[PSV(LCP_R, s')..NSV(LCP_R, e')]$.

With these ideas in place, we are ready to describe the matching statistics algorithm. We first compute SA_R, ISA_R, and LCP_R for R and preprocess LCP_R for NSV/PSV queries. The elements of the MS will be computed in left-to-right order, $MS[1], MS[2], \ldots, MS[|S|]$. Note that this makes it trivial to save only the heads (or iheads) and so compute the CMS (or eCMS) instead. To find $MS[1]$ use successive right extensions starting with the interval $SA_R[1..|R|]$, searching with successive characters of $S[1..]$ until the right extension is empty, at which point we know ℓ_1 and p_1. At a generic step in the algorithm, immediately after computing $MS[i]$, we know the interval $SA_R[s_i..e_i]$ containing all the occurrences of $R[p_i..p_i + \ell_i - 1]$. To compute $MS[i + 1]$ we first compute the left contraction of $SA_R[s_i..e_i]$, followed by as many right contractions as possible until ℓ_{i+1} and p_{i+1} are known.

When profiling an implementation of the above algorithm, we noticed that very often the sequence of right extensions ended with a singleton interval (i.e., an interval of size one) and so was the interval reached by the left contraction that followed. In terms of the suffix tree, this corresponds to the match between R and the current suffix of S_i being inside a leaf branch. This frequently happens on genome collections because each sequence is likely to have much longer matches with other sequences (in this case with R) than it does with itself (a single genome tends to look fairly random, at least by string complexity measures).

A simple heuristic to exploit this phenomenon is to compare ℓ_i to the maximum value in the entire LCP_R array of R immediately after $MS[i]$ has been computed. If $\ell_i - 1 > \max(LCP_R)$ then $ISA_R[p_i + 1]$ will also be inside a leaf branch (i.e., the left contraction will also be a singleton interval), and so the left contraction can be computed trivially as $ISA_R[p_i + 1]$ – with no subsequent NSV/PSV queries or access to LCP_R required to expand the interval. Although this gives no asymptotic improvement, there is potential gain from the probable cache miss(es) avoided by not making random accesses to those large data structures.

On a viral genome data set (see Section 7), $\max(LCP_R)$ was 14, compared to an average ℓ_i value of over $1,100$, and this heuristic saved lots of computation. On a human chromosome data set, however, $\max(LCP_R)$ was in the hundreds of thousands, and so we generalized the trick in the following way. We divide the LCP array up into blocks of size b and compute the minimum of each block. These minima are stored in an array M of size $|R|/b$, and b is chosen so that M is small enough to comfortably fit in cache. Now, when transitioning from $MS[i]$ to $MS[i+1]$, if $\ell_i > M[ISA_R[p_i + 1]/b]$ then there is a single match corresponding to $MS[i+1]$, which we compute with right extensions. This generalized form of the heuristic has a consistent and noticeable effect in practice. For a 500MB viral genome data set its use reduced CMS computation from 12.23 seconds to 2.34 seconds. On the human chromosome data set the effect is even more dramatic: from 76.50 seconds down to 7.14 seconds.

4 Comparing two suffixes via the enhanced CMS

We will now show how to use the enhanced CMS of the collection $\mathcal{C}$ w.r.t. R to define a partial order on the set of suffixes of strings in $\mathcal{C}$ (Prop. 12), and how to break ties when the entries are identical (Lemma 13). These results can then be used either directly to determine the relative order of any two of the suffixes (Prop. 14), or as a way of inducing the complete order once that of the subset of the insert-heads has been determined (Prop. 15).

We will prove Prop. 12 via two lemmas. Recall that in the *eCMS* we only have the entries referring to the insert-heads; however, Lemma 8 tells us how to compute them for any position.

▶ **Lemma 10.** *Let* $1 \le d, d' \le m$ *and* $1 \le i \le |S_d|$, $1 \le i' \le |S_{d'}|$. *If* $ip(d,i) < ip(d',i')$, *then* $S_d[i..] < S_{d'}[i'..]$.

Proof. If $ip(d',i') - ip(d,i) > 1$, then there exists an index j s.t. $ip(d,i) < j < ip(d',i')$, and therefore $S_d[i..] < R[SA_R[ip(d,i)+1]..] \le R[SA_R[j]..] \le R[SA_R[ip(d',i')-1]..] < S_{d'}[i'..]$. Now let $ip(d',i') = ip(d,i) + 1$. If $x_{d,i} = S$, then $S_d[i..] < R[SA_R[ip(d,i)]..] = R[SA_R[ip(d',i')-1]..] < S_{d'}[i'..]$, by Obs. 7. Similarly, if $x_{d',i'} = L$, then $S_d[i..] < R[SA_R[ip(d,i)+1]..] = R[SA_R[ip(d',i')]..] < S_{d'}[i'..]$. Finally, let $x_{d,i} = L$ and $x_{d',i'} = S$. Then $R[SA_R[ip(d,i)]..] < S_d[i..], S_{d'}[i'..] < R[SA_R[ip(d,i)+1]..] = R[SA_R[ip(d',i')]..]$. Let U be the matching factor of (d,i), U' that of (d',i'), and $V = lcp(U,U')$, the longest common prefix of the two. V cannot be equal to U' because then U' would be a proper prefix of U, but $ip(d',i')$ is the smallest occurrence in R of U'. If $V = U$, then U is a proper prefix of U', and by definition of $ip(d',i')$, the character following U in U' is strictly greater than the mismatch character c_i of (d,i). Finally, if V is a proper prefix both of U and of U', then the character following V in U is smaller than the one following V in U', therefore $U < U'$. Since U is a prefix of $S_d[i..]$ and U' is a prefix of $S_{d'}[i'..]$, and neither is prefix of the other, this implies $S_d[i..] < S_{d'}[i'..]$. ◀

▶ **Lemma 11.** *Let $1 \leq d, d' \leq m$ and $1 \leq i \leq |S_d|$, $1 \leq i' \leq |S_{d'}|$, and $ip(d,i) = ip(d',i')$.*
1. *If $\ell_{d,i} < \ell_{d',i'}$ and $x_{d,i} = S$, then $S_d[i..] < S_{d'}[i'..]$.*
2. *If $\ell_{d,i} < \ell_{d',i'}$ and $x_{d,i} = L$, then $S_{d'}[i'..] < S_d[i..]$.*
3. *If $\ell_{d,i} = \ell_{d',i'}$ and $x_{d,i} = S$ and $x_{d',i'} = L$, then $S_d[i..] < S_{d'}[i'..]$.*
4. *If $\ell_{d,i} = \ell_{d',i'}$ and $x_{d,i} = x_{d',i'}$ and $c_{d,i} < c_{d',i'}$, then $S_d[i..] < S_{d'}[i'..]$.*

Proof.

1., 2. Let U be the matching factor of i, and U' that of i'. Since $\ell_{d,i} < \ell_{d',i'}$, this implies that U is a proper prefix of U'. If $x_{d,i} = S$, then the mismatch character $c_{d,i}$ is smaller than the character following U in U', therefore $S_d[i..] < S_{d'}[i'..]$. If $x_{d,i} = L$, then it is greater, and thus $S_{d'}[i'..] < S_d[i..]$.

3. follows directly from Observation 7, since now $S[i..] < R[SA_R[ip(i)]..] < S[i'..]$.

4. Now both suffixes start with the same matching factor U, followed by different mismatch characters, which define their relative order. ◀

These two lemmas in fact imply the following:

▶ **Proposition 12.** *The conditions of Lemmas 10 and 11 result in a partial order of the suffixes of strings in $\mathcal{C}$, of which the lexicographic order is a refinement.*

What happens if two suffixes $S_d[i..]$ and $S_{d'}[i'..]$ have the same values of the enhanced matching statistics, i.e. $ems(d,i) = ems(d',i')$? The next lemma says that in this case, the relative order of the two suffixes is decided by the relative order of the heads preceding their respective mismatch characters.

▶ **Lemma 13.** *Let $1 \leq d, d' \leq m$ and $1 \leq i \leq |S_d|$, $1 \leq i' \leq |S_{d'}|$. If $ip(d,i) = ip(d',i')$, $\ell_{d,i} = \ell_{d',i'}$, $x_{d,i} = x_{d',i'}$, and $c_{d,i} = c_{d',i'}$, then $S_d[i..] < S_{d'}[i'..]$ if and only if $S_d[j..] < S_{d'}[j'..]$, where $(d,j) = i\text{-}head(d, i + \ell_i)$ and $(d',j') = i\text{-}head(d', i' + \ell_{i'})$.*

Proof. We will prove that the relative position of the insert-head of i's and i''s mismatch character is the same, i.e. that $j - i = j' - i'$. The claim then follows.

First note that $j > i$. This is because the matching factor of position i ends in position $i + \ell_{d,i} - 1$, so there must be a new insert-head after i and at most at $i + \ell_{d,i}$, the position of the mismatch character. Similarly, $j' > i'$. The fact that $j = i\text{-}head(i + \ell_{d,i})$ implies that there is a matching factor starting in position j which spans the mismatch character $c = c_{d,i} = c_{d',i'}$. Let's write Vc for the prefix of length $i + \ell_{d,i} - j$ of this matching factor. V is a suffix of the matching factor U of position i, but Vc is not. However, Vc is also a prefix of $S_{d'}[i'..]$. Therefore, $j' = i' + (j - i)$ is also an insert-head in $S_{d'}$. An analogous argument shows that any insert-head between i' and $i' + \ell_{d',i'}$ in $S_{d'}$ is also an insert-head in S_d, in the same relative position. ◀

▶ **Proposition 14.** *Let $R, S_1, \ldots, S_m$ be strings over Σ. Using the enhanced CMS of $\mathcal{C} = \{S_1, \ldots, S_m\}$ w.r.t. R, we can decide, for any $1 \leq d, d' \leq m$ and $1 \leq i \leq |S_d|$, $1 \leq i' \leq |S_{d'}|$, the relative order of $S_d[i..]$ and $S_{d'}[i'..]$ in $\mathcal{O}(\log\log\chi' \cdot \max_d\{\text{no. of insert-heads of } S_d\})$ time.*

Proof. Let $(d,j) = i\text{-}head(d, i + \ell_i)$ and $(d',j') = i\text{-}head(d', i' + \ell_{i'})$. From Lemma 8 we get the four *eCMS*-entries of (d,i) and (d',i'), namely the insert positions q_i resp. $q_{i'}$, the length of the matching factor, whether the mismatch characters is smaller or larger, and the mismatch character itself. If any of these differ for the two suffixes, then Lemmas 10 and 11 tell us their relative order. This check takes $\mathcal{O}(1)$ time. Otherwise, Lemma 13 shows that the relative order is determined by the next relevant heads. Iteratively applying the three lemmas, in the worst case, takes us through all heads for the strings S_d and $S_{d'}$. ◀

Instead of using Prop. 14, we will use these lemmas in the following way. We will first sort only the insert-heads. The following proposition states that this suffices to determine the order of any two suffixes in constant time.

▶ **Proposition 15.** *Given the insert-heads in sorted order, the relative order of any two suffixes can be determined in $\mathcal{O}(\log\log\chi')$ time, where χ' is the number of insert-heads.*

Proof. Follows from Lemmas 10, 11, and 13, since all checks take constant time, and each of the two predecessor queries take $\mathcal{O}(\log\log\chi')$ time. ◀

5 Putting it all together

A high-level view of our algorithm is as follows. We first partially sort the insert-heads, then use this partial sort to generate a new string, whose suffixes we sort with an existing suffix sorting algorithm. This gives us a full sort of the insert heads. We then use this to sort the S^*-suffixes of the collection. Finally, we induce the remaining suffixes of the collection using the S^*-suffixes. We next give a schematic description of the algorithm.

Algorithm 1 Algorithm for computing the GSA of string collection $\mathcal{C}$.

input: string collection $\mathcal{C}$, reference string R
output: the *GSA* of $\mathcal{C}$

- **Phase 1 - Augmenting and constructing data structures on R:** Preprocess R ("augmenting", see Sec. 6). Compute the data structures $SA_R, ISA_R, PLCP_R, LCP_R$ and the RMQ-data structure for PSV- and NSV-queries on LCP_R.
- **Phase 2 - Computing the *eCMS*:** Compute the *eCMS* of $\mathcal{C}$, as described in Sec. 3.2.
- **Phase 3 - Bucketing:** Identify the S^*-suffixes in $\mathcal{C}$ via a backward linear scan of $\mathcal{C}$. Bucket S^*-suffixes i according to $ip(i)$, computed using the *eCMS* (Lemma 8).
- **Phase 4 - Sorting the insert-heads:**
  - bucket the insert-heads according to their insert point;
  - for each bucket B, partially sort B, according to Lemmas 10 and 11;
  - rename insert-heads according to lexicographic rank of substring stretching up to the mismatch character (metacharacters are $S_d[j..j+\ell_{d,j}]$);
  - generate new string C as concatenation of these metacharacters;
  - compute the suffix array of C, map back to corresponding suffixes of $\mathcal{C}$.
- **Phase 5 - Fully sorting the S^*-suffixes:** for each bucket B from Phase 3, sort B, according to Lemmas 11 and 13
- **Phase 6 - Inducing the *GSA*:** With two scans, induce L-suffixes, induce S-suffixes.

We next give a worst-case asymptotic analysis of the algorithm.

▶ **Proposition 16.** *Algorithm 1 computes the GSA of a string collection $\mathcal{C}$ of total length N in worst-case time $\mathcal{O}(N\log N)$.*

Proof. Let $|R| = n$. Phase 1 takes $\mathcal{O}(n+N)$ time, since constructing all data structures on R can be done in linear time in n and scanning the collection $\mathcal{C}$ takes time $\mathcal{O}(N)$. Phase 2 takes time $\mathcal{O}(N\log n)$ using the algorithm from Sec. 3.2. In Phase 3, identifying the S^* suffixes, takes time $\mathcal{O}(N)$. Since at this point, the *eCMS* is in text-order, identifying *i-head*(i) takes constant time, also computing the insert-point takes constant time, so altogether $\mathcal{O}(N)$

time. In Phase 4, all steps are linear in χ', the number of insert-heads, including the partial sort of the buckets, since this can be done with radix-sort (three passes over each bucket), so this phase takes time $\mathcal{O}(\chi')$. Phase 5 takes time $\mathcal{O}(|B| \log |B|)$ for each bucket B, thus $\mathcal{O}(N \log |B_{\max}|)$ for the entire collection, where $B_{\max}$ is a largest bucket. Since all strings in the collection are assumed to be highly similar to the reference, the size of the buckets can be expected to vary around the number of strings in the collection m; however, in the worst case the largest bucket can be $\Theta(N)$. Finally, Phase 6 takes linear time $\mathcal{O}(N)$. Altogether, the running time is dominated by Phase 5, $\mathcal{O}(N \log N)$. ◀

6 Implementation details

In Phase 1, the augmentation step involves, for every character c not occurring in R but occurring in $\mathcal{C}$, appending c^{n_c} to R, where n_c is the length of the longest run of c in $\mathcal{C}$. This avoids having 0-length entries in the matching statistics and is necessary in order to have a well defined ip.

To compute SA_R in Phase 1, we use `sais` [18] as implemented by Yuta Mori, a well engineered version of SAIS [20], which was chosen due to its consistent speed on many different inputs. For the computation of $PLCP_R$ and LCP_R we use the Φ method [11]. This is the fastest method to compute the LCP array we know of. We constructed the data structure of Cánovas and Navarro [4] for NSV/PSV queries on the LCP array, as it has low space overheads and was fast to query and initialize.

For the predecessor data structure, we use the following two-layered approach in practice (rather than [27]). We sample every bth head starting position and store these in an array. In a separate array we store a differential encoding of all head positions. The array of differentially encoded starting positions takes 32 bits per entry. Predecessor search for a position x proceeds by first binary searching in the sampled array to find the predecessor sample at index i of that array. We then access the differentially encoded array starting at index ib and scan, summing values until the cumulative sum is greater than x, at which point we know the predecessor. This takes $O(\chi'/b + b)$ time, where χ' is the number of insert-heads.

For Phase 4, when we have to sort C (the concatenation of metacharacters representing partially sorted heads), we use a SACA-K implementation that handles integer alphabets [13]. This choice was made because of this algorithm's low space requirement, in particular, $\mathcal{O}(K)$, where K is the number of distinct *ems*-entries of insert-heads in H' (note $K = O(\chi')$).

7 Experiments

We implemented our algorithm for computing the generalized suffix array in C++. Our prototype implementation, `sacamats`, is available at `https://github.com/fmasillo/sacamats`. The experiments were conducted on a laptop equipped with 16GB of RAM DDR4-2400MHz and an Intel(R) Core(R) i5-8250U@3.4GHz with 6MB of cache. The operating system is Ubuntu 20.04 LTS, the compiler used is `g++` version 9.4.0 with options `-std=c++17 -O3 -funroll-loops` enabled.

In the following experiments, we compare `sacamats` to two well known suffix array construction tools, both implementations by Yuta Mori [18, 17]. The first, `sais`, is an implemenation of the well-known SAIS algorithm by Nong et al. [20]; the second, `divsufsort` [7], is perhaps the most widely used tool for suffix array construction. We also compare against

`gsufsort` [14], which is an extension of the SACA-K algorithm [19] to a collection of strings, and to `bigBWT` [3], a tool computing the BWT and the suffix array, designed specifically for highly repetitive data.

7.1 Datasets

For our tests, we used two publicly available datasets, one consisting of copies of human chromosome 19 from the 1000 Genomes Project [26], and another of copies of SARS-CoV2 genomes taken from NCBI Datasets[1]. For both datasets we selected subsets of different sizes in order to study the scalability of our algorithm. The sizes are 250MB, 500MB, 800MB and 1GB. More information can be found in Table 1.

We observe that on both datasets the number of i-heads is around 100x less than the input size, and on `chr19` it is 8x less than the number of BWT runs.

Table 1 Datasets used in experiments. In column 3, we specify the alphabet size σ, in column 4 the number r of runs of the BWT, in column 5 the number of S^*-suffixes, and column 6 the number of insert-heads. In our experiments we use prefixes of each dataset up to 1GB. The last three columns refer to the 500MB prefix.

Name	Description	σ	r	no. of S^*-suffixes	no. of i-heads
`chr19`	Human Chromosome 19	5	32 018 267	129 129 636	4 220 033
`sars-cov2`	SARS-CoV2 genome	14	351 596	143 588 463	6 537 294

7.2 Results

In Figures 3 and 4, information about running time for both datasets is displayed. The line plot represents a direct comparison of different algorithms, whereas the stacked bar plot is to visualize how much each phase of `sacamats` takes w.r.t. the total running time (cp. Sec. 5).

These tools all produce slightly different outputs: `sais` and `divsufsort` output the SA, `gsufsort` and `sacamats` the GSA, and `bigBWT` both the BWT and the SA. Because of these differences, if one were to write to disk each result, the running time would be affected accordingly by the size of the output. Therefore, we only compare the building time, i.e. the time spent constructing the SA and storing it in a single array in memory, without the time spent writing it to disk. For this reason, we made slight changes to the `bigBWT` code to enable storing the SA in main memory.

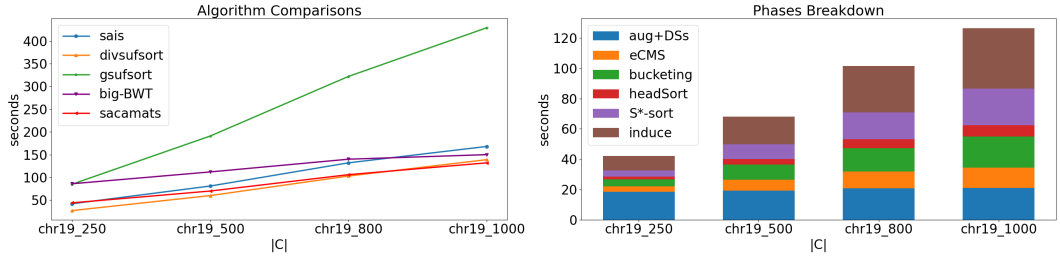
By looking at the line plots, one can see that `sacamats` is competitive in both scenarios, i.e., it is faster than all tools on `sars-cov2`, except `bigBWT`. The same is true for `chr19`, where it is the fastest method, especially on larger inputs, but here the main competitor becomes `divsufsort`. More precisely, for the first dataset (`chr19`) and considering 1GB of data, `sacamats` takes less than a third of the time of `gsufsort`, is 20% faster than `sais`, 12% faster than `bigBWT`, and 5% faster than `divsufsort`. For the second dataset (`covid`), `sacamats` takes again less than a third of the time of `gsufsort`, is 37% faster than `divsufsort`, 16% faster than `sais`, and 30% slower than `bigBWT`.

Shifting our attention to the stacked bar plots, Figure 3b indicates that a lot of time is spent in the first phase, consisting in the augmentation of R and the construction of various data structures for the augmented version of R. In the setting of DNA strings it is not too

[1] `https://www.ncbi.nlm.nih.gov/datasets/coronavirus/genomes/`

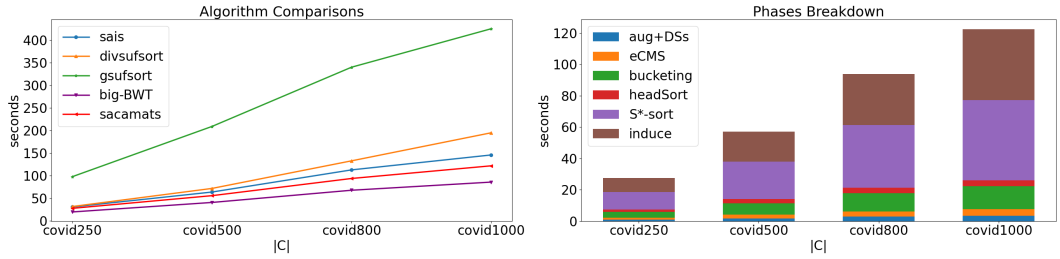
hard to think that the augmentation process will not elongate R, due to the very restricted alphabet. If the application lends itself to it, one could compute beforehand all the data structures listed in Phase 1, gaining roughly 20 seconds of run time. In our experiment on `chr19` we would then be clearly the best algorithm, further distancing from the others. Alternatively, the common method of replacing `N` symbols with random nucleotide symbols would be another way to speed up this phase.

Finally, we comment on memory usage, which is highest for `sacamats` and `gsufsort` at 8 bytes per input symbol, and 4 bytes per input symbol for `divsufsort` and `sais`, and `bigBWT` (including the 4 bytes per input symbol of the SA when it is saved in memory, see above). We have not yet optimized for memory usage and note that a semi-external implementation of our approach, in which buckets reside on disk, presents itself as an effective way to reduce main memory usage. In all phases, the actual working set – the amount of data active in main memory – is small (for the most part, proportional to the number of i-heads), and other authors have shown that the inducing phase is amenable to external memory, too [10]. We leave these optimizations as future work.



(a) Running time comparison. **(b)** Phases breakdown, see Sec. 5 for details.

Figure 3 Experiments on different subsets of copies of Human Chromosome 19.



(a) Running time comparison. **(b)** Phases breakdown.

Figure 4 Experiments on different subsets of SARS-CoV2 genomes.

8 Conclusion

We have presented a new algorithm for computing the generalized suffix array of a collection of highly similar strings. It is based on a compressed representation of the matching statistics, and on efficient handling of string comparisons. Our experiments show that a relatively straightforward implementation of the new algorithm is competitive with the fastest existing suffix array construction algorithms on datasets of highly similar strings, as are common in computational biology applications.

A byproduct of our suffix sorting algorithm is a heuristic for fast computation of the matching statistics of a collection of highly similar genomes w.r.t. a reference sequence, which is of independent interest. We also envisage uses for our compressed matching statistics (CMS) data structure beyond the present paper, for example as a tool for sparse suffix sorting, or for distributed suffix sorting in which the CMS is distributed to all sorting nodes together with a lexicographic range of the suffixes that each particular node is responsible for sorting. From the CMS alone, each node can extract the positions of its suffixes and then sort them with the aid of the CMS.

We believe there to be a great deal of room for further practical improvements, both through algorithm engineering and parallelism. Interestingly, in an initial attempt along the second line, simply assigning S^*-suffix buckets to one of four different sorting threads reduces runtime significantly, for example, from 122 to 89 seconds on the 1GB `sars-cov2` dataset.

Further studies will be conducted on how the size of *eCMS* impacts on the competitiveness of our tool.

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.

2 Djamal Belazzougui, Fabio Cunial, and Olgert Denas. Fast matching statistics in small space. In *Proc. 17th International Symposium on Experimental Algorithms (SEA)*, volume 103 of *LIPIcs*, pages 17:1–17:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

3 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms Mol. Biol.*, 14(1):13:1–13:15, 2019.

4 Rodrigo Cánovas and Gonzalo Navarro. Practical compressed suffix trees. In *Proc. of the 9th International Symposium Experimental Algorithms, SEA 2010*, volume 6049 of *LNCS*, pages 94–105. Springer, 2010.

5 William I. Chang and Eugene L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994.

6 Johannes Fischer. Combined data structure for previous- and next-smaller-values. *Theor. Comput. Sci.*, 412(22):2451–2456, 2011.

7 Johannes Fischer and Florian Kurpicz. Dismantling divsufsort. In *Proc. of the Prague Stringology Conference 2017*, pages 62–76. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017.

8 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020.

9 Hideo Itoh and Hozumi Tanaka. An efficient method for in memory construction of suffix arrays. In *Proc. of the 6th International Symposium on String Processing and Information Retrieval and the 5th International Workshop on Groupware, (SPIRE/CRIWG)*, pages 81–88. IEEE Computer Society, 1999.

10 Juha Kärkkäinen, Dominik Kempa, Simon J. Puglisi, and Bella Zhukova. Engineering external memory induced suffix sorting. In *Proc. of the 19th Workshop on Algorithm Engineering and Experiments, ALENEX 2017*, pages 98–108. SIAM, 2017.

11 Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In *Proc. of the 20th Annual Symposium on Combinatorial Pattern Matching, CPM 2009*, volume 5577 of *LNCS*, pages 181–192. Springer, 2009.

12 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005.

13 Felipe A. Louza, Simon Gog, and Guilherme P. Telles. Inducing enhanced suffix arrays for string collections. *Theor. Comput. Sci.*, 678:22–39, 2017.

14 Felipe A. Louza, Guilherme P. Telles, Simon Gog, Nicola Prezza, and Giovanna Rosone. gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections. *Algorithms Mol. Biol.*, 15(1):18, 2020.

15 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015. doi:10.1017/CBO9781139940023.

16 U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

17 Yuta Mori. Code for divsufsort. URL: `https://github.com/y-256/libdivsufsort`.

18 Yuta Mori. Code for sais-lite. URL: `https://sites.google.com/site/yuta256/sais`.

19 Ge Nong. Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.*, 31(3):15, 2013.

20 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.

21 Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013. URL: `http://www.oldenbusch-verlag.de/`.

22 Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *Proc. of the 17th International Symposium on String Processing and Information Retrieval, SPIRE 2010*, volume 6393 of *LNCS*, pages 347–358. Springer, 2010.

23 Simon J. Puglisi, William F. Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2):4, 2007.

24 Simon J. Puglisi and Bella Zhukova. Relative lempel-ziv compression of suffix arrays. In *Proc. of the 27th International Symposium on String Processing and Information Retrieval, SPIRE 2020*, volume 12303 of *LNCS*, pages 89–96. Springer, 2020.

25 Massimiliano Rossi, Marco Oliva, Paola Bonizzoni, Ben Langmead, Travis Gagie, and Christina Boucher. Finding maximal exact matches using the r-index. *J. Comput. Biol.*, 29(2):188–194, 2022.

26 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–74, 2015.

27 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.

A Maximum Parsimony Principle for Multichromosomal Complex Genome Rearrangements

Pijus Simonaitis
Department of Computer Science, Princeton University, Princeton, NJ, USA

Benjamin J. Raphael
Department of Computer Science, Princeton University, Princeton, NJ, USA

Abstract

Motivation. Complex genome rearrangements, such as chromothripsis and chromoplexy, are common in cancer and have also been reported in individuals with various developmental and neurological disorders. These mutations are proposed to involve simultaneous breakage of the genome at many loci and rejoining of these breaks that produce highly rearranged genomes. Since genome sequencing measures only the novel adjacencies present at the time of sequencing, determining whether a collection of novel adjacencies resulted from a complex rearrangement is a complicated and ill-posed problem. Current heuristics for this problem often result in the inference of complex rearrangements that affect many chromosomes.

Results. We introduce a model for complex rearrangements that builds upon the methods developed for analyzing simple genome rearrangements such as inversions and translocations. While nearly all of these existing methods use a maximum parsimony assumption of minimizing the number of rearrangements, we propose an alternative maximum parsimony principle based on minimizing the number of chromosomes involved in a rearrangement scenario. We show that our model leads to inference of more plausible sequences of rearrangements that better explain a complex congenital rearrangement in a human genome and chromothripsis events in 22 cancer genomes.

2012 ACM Subject Classification Applied computing → Bioinformatics

Keywords and phrases Genome rearrangements, maximum parsimony, cancer evolution, chromothripsis, structural variation, affected chromosomes

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.21

Supplementary Material *Software*: `https://github.com/raphael-group/MICRO`

Funding This work was supported by grants U24CA248453 from the US National Cancer Institute (NCI).

1 Introduction

Genome rearrangements transform a genome by breaking two or more genomic loci and joining the resulting chromosomal segments in a different order. The most common and most well-studied genome rearrangements are *simple* rearrangements such as translocations and inversions (reversals) that break a genome at two locations and join the resulting free ends. More recently, *complex* rearrangements [28] that involve simultaneous breaking and joining at several loci – sometimes up to hundreds of loci – have been reported. These complex rearrangements include chromothripsis [33] and chromoplexy [5], which were reported in more than 25% of the cancer patients in recent pan-cancer genome sequencing studies [36, 12, 16]. Complex rearrangements have also been reported in patients harboring congenital and developmental disorders [31, 40] as well as seemingly healthy individuals [13]. A number of

putative mechanisms for complex rearrangements have been proposed and experimentally induced in cell lines [35, 41, 22], but the precise mechanisms that lead to these mutations occurring in human cells remain largely unknown [27].

Inferring complex rearrangements from genome sequencing data is a difficult problem, since sequencing data measures only *novel adjacencies* – pairs of loci that are adjacent in the sequenced genome but distant in the human reference genome – and determining which combination of these novel adjacencies constitute a complex rearrangement vs. multiple simple rearrangements is a complicated and ill-posed problem. Multiple methods have been developed to predict complex rearrangements from sequencing data [5, 12, 7, 23, 16, 21]. These methods use various heuristics to identify clusters of novel adjacencies whose ends, or *extremities*, are close together on the human reference genome. However, the sensitivity and specificity of these methods in distinguishing one-off complex rearrangements from progressive sequences of simple rearrangements remains a source of debate [24, 19, 26].

There is an extensive literature studying sequences of genome rearrangements, and finding the *minimum* number of genome rearrangements that transform one genome into another. Most of this work focuses on simple rearrangements and analyses sequences of inversions [18], sequences of inversions and translocations [17, 29], or sequences of *double-cut-and-join* operations [37, 32, 39], also called 2-breaks, that break a genome and two loci and join the resulting free ends. A complex genome rearrangement can be modeled by a *k-break* that breaks a genome at k loci and joins back the resulting chromosomal strands thus introducing k novel adjacencies [4, 19, 10], a generalization of a *double-cut-and-join*.

Computing the minimum number of rearrangements that transform one genome into another follows the principle of maximum parsimony, or finding the simplest explanation for the data. However, once complex rearrangements are allowed, it is unclear how to define *simplest*. For example, it may be possible to transform one genome into another with a *single* k-break, but using an extremely large value of k. Indeed complex rearrangements involving more than a hundred simultaneous breaks in cancer genomes have been proposed [12, 16]. However, explaining data with a single arbitrary complex rearrangement is in a sense trivial: this explanation is parsimonious in one criterion (minimizing number of rearrangements) but not parsimonious under another criterion (minimizing k, or complexity of allowed operations). It is generally unknown what values of k are reasonable to analyze complex rearrangements using k-breaks. Thus, there is a gap between the parsimonious k-break scenarios studied in the genome rearrangement literature – where the values of k are relatively small and a minimum sequence of rearrangements is computed – and the arbitrary complex rearrangements described in the cancer genomics literature – where the value of k is unbounded leading to explanations of the data with a single, arbitrarily complicated rearrangement. These two approaches are in a sense two extremes and a natural question is whether there is an intermediate between these extremes.

In this paper, we propose an alternative maximum parsimony principle for studying complex rearrangements that involve multiple chromosomes. Specifically, based on biological knowledge of the mechanisms of complex rearrangements, we propose that a complex rearrangement might be unlikely to simultaneously break a large number of chromosomes. Supporting this approach are two non-exclusive cellular mechanisms that have been proposed to explain a shattering of one or a few chromosomes followed by a random joining of the resulting chromosomal segments [28]. First, defects in chromosomal segregation or formation of acentric chromosomes might lead to a physical isolation and rearrangement of one or a few chromosomes in an aberrant nuclear structure called micronucleus [41]. Second, a dicentric chromosome formed after an end-to-end fusion or a translocation between

two chromosomes might get shattered during mitosis [35]. Following these observations, we propose that the *chromosome number*, the maximum number of chromosomes broken by a k-break in a sequence of rearrangements, is a useful statistic for studying complex rearrangements. Further, we propose that minimizing the chromosome number provides an alternative maximum parsimony criterion for evaluating sequences of simple and complex rearrangements that transform one genome into another.

We derive two algorithms to compute the minimum chromosome number of rearrangement scenarios under the *infinite sites assumption* (ISA) [25, 15, 2], also known as the *constraint of no breakpoint reuse* [3], where a genomic locus is involved in at most one genome rearrangement in a sequence of genome rearrangements transforming one genome into another. The first algorithm computes the minimum chromosome number for rearrangement scenarios between two genomes. Unfortunately, current DNA sequencing technologies do not yield complete genomes, but rather measure a set of novel adjacencies that are present in this genome. This measured set is often missing novel adjacencies that are present in the sequenced genome [8], and for sequencing data from bulk tumor this set might be a superposition of novel adjacencies from multiple cancer clones in the tumor [2]. Thus, the second algorithm computes the minimum chromosome number between a genome and a set of novel adjacencies.

We apply our first algorithm to five human genomes that were proposed to harbor congenital complex rearrangements [14, 11, 13, 30] and our second algorithm to 252 cancer genomes that were identified to harbor chromothripsis events [16]. For one of the human genomes and for 22 of the cancer genomes, we derive alternative sequences of rearrangements with lower chromosome number demonstrating that multichromosomal complex rearrangements may be less complicated than previously described.

2 Methods

2.1 Multi-breaks and rearrangement scenarios

Let A be a reference genome and let B be a rearranged genome that is derived from A by a sequence of simple and complex rearrangements. Here, a *genome* is defined as a set of linear and circular DNA molecules called *chromosomes*, each chromosome is partitioned into a sequence of unique directed *synteny blocks*, and pairs of consecutive blocks are separated by *breakpoint regions*. The two *endpoints* of a synteny block are its *extremities*, and an *adjacency* is an unordered pair of extremities separated by a breakpoint region. A *telomere* is an extremity incident to an end of a linear chromosome, and two genomes are *co-tailed* if their sets of telomeres are equal.

A *rearrangement* is a k-break for $k \geq 2$ that breaks a genome at k breakpoint regions and joins the resulting chromosomal fragments back thus forming k new breakpoint regions and modifying the order of the synteny blocks [4]. Specifically, let A be a genome, let $\alpha = \{\{u_1, u_2\}, \ldots, \{u_{2k-1}, u_{2k}\}\}$ be a subset of its adjacencies, and let β be a set $\{\{u_{\sigma(1)}, u_{\sigma(2)}\}, \ldots, \{u_{\sigma(2k-1)}, u_{\sigma(2k)}\}\}$ disjoint from α with σ being a permutation of a set $\{1, \ldots, 2k\}$. An ordered pair $\tau = (\alpha, \beta)$ is a *k-break* and we say that it *transforms* A into a genome in which adjacencies α are replaced with adjacencies β. A *multi-break* is a k-break for $k \geq 2$. See Figure 1 for an example of two co-tailed genomes that contain the same synteny blocks.

Given a pair A and B of co-tailed genomes with the same synteny blocks, there is a single multi-break that transforms A into B, formalized in the following lemma.

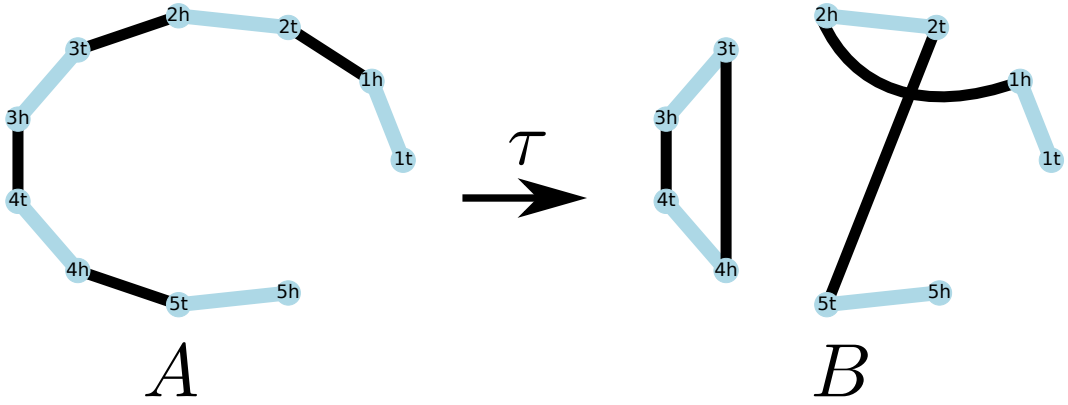


Figure 1 Two co-tailed genomes, A and B, partitioned into five synteny blocks (blue lines) and a 3-break τ transforming A into B. Genome A consists of a single linear chromosome with adjacencies (black lines) $\{\{1_h, 2_t\}, \{2_h, 3_t\}, \{3_h, 4_t\}, \{4_h, 5_t\}\}$. Genome B consists of a circular and a linear chromosomes with adjacencies $\{\{1_h, 2_h\}, \{2_t, 5_t\}, \{3_h, 4_t\}, \{4_h, 3_t\}\}$. A 3-break $\tau = (\alpha, \beta)$ with $\alpha = \{\{1_h, 2_t\}, \{2_h, 3_t\}, \{4_h, 5_t\}\}$ and $\beta = \{\{1_h, 2_h\}, \{2_t, 5_t\}, \{4_h, 3_t\}\}$ transforms A into B.

▶ **Lemma 1.** *Suppose that A and B are co-tailed genomes that contain the same synteny blocks, $E(A)$ are the adjacencies of A that are absent from B, and $E(B)$ are the adjacencies of B that are absent from A. The multi-break $\tau_t = (E(A), E(B))$ is the unique multi-break that transforms A into B.*

The proof of Lemma 1 along with all the other proofs is to be found in Appendix A. We call $\tau_t = (E(A), E(B))$ the *trivial multi-break* for A and B. For now we assume that genomes A and B are co-tailed and contain the same synteny blocks, however this assumption will be relaxed in Section 2.4.

A multi-break *scenario* $\mathbf{T}$ for genomes A and B is a sequence $(\tau_1, \ldots, \tau_l)$ of multi-breaks transforming A into B. Following previous work [4], we say that $\mathbf{T}$ is a *k-break scenario* if all the multi-breaks in $\mathbf{T}$ break at most k adjacencies; i.e., $\tau_i = (\alpha_i, \beta_i)$ where $|\alpha_i| \leq k$ for $i \in \{1, \ldots, l\}$. Appealing to the principle of parsimony, a common problem studied in the genome rearrangement literature is to find the *most parsimonious* rearrangement scenario, or the rearrangement scenario with the fewest number of mutations. Along these lines, a dynamic programming algorithm that is polynomial in k was previously proposed for finding a parsimonious k-break scenario for A and B, and applied to human and mouse genomes with $k = 3$ [4, 3]. However, when examining complex rearrangements, it is unclear what values of k to allow in a k-break scenario. If the number of breaks is unbounded, then, as shown in Lemma 1, the trivial multi-break τ_t transforms A into B and comprises the *trivial multi-break scenario* $\mathbf{T}_t = (\tau_t)$ for A and B. Thus, there is a gap between the parsimonious k-break scenarios considered in the genome rearrangement literature where the number of breaks k is supposed to be ≤ 3 for all the practical purposes, and the multi-break scenarios assumed in the cancer genomics literature, where the number of breaks is unbounded. We are interested in what biologically motivated constraints might replace the number k of breaks in the study of complex rearrangements.

We build upon existing work in cancer genomics and genome rearrangement literature and suppose that evolution by genome rearrangements respects the *Infinite Sites Assumption (ISA)* [25, 2], also known as the *constraint of no breakpoint reuse* [4]. A multi-break scenario $\mathbf{T}$ is said to be a *ISA multi-break scenario* if an adjacency joined by a multi-break in $\mathbf{T}$ is not broken, or *reused*, by any of the subsequent rearrangements in that scenario. Let $\mathcal{I}(A, B)$ be

the set of ISA multi-break scenarios transforming A into B. Note that the trivial multi-break scenario $\mathbf{T}_t = (\tau_t)$ is a ISA multi-break scenario for A and B thus ensuring that $\mathcal{I}(A, B)$ is not empty. We say that a multi-break τ is a *ISA multi-break* for A and B if τ appears in some ISA multi-break scenario that transforms A into B. Let $\mathcal{T}(A, B)$ be the set of ISA multi-breaks for A and B. The ISA is based on two underlying assumptions. First, that the probability of a region to be broken by a rearrangement is proportional to the number of base pairs spanned by this region or its *length* [9]. And second, that breakpoint regions are so short that they are unlikely to be reused.

2.2 The chromosome number of a multi-break scenario

We propose to evaluate multi-break scenarios according to the number of chromosomes broken by the multi-breaks. More specifically, given a genome A and a multi-break $\tau = (\alpha, \beta)$ that transforms A, we say that a chromosome of a genome A is *broken* by $\tau = (\alpha, \beta)$ if α includes an adjacency from that chromosome. Let $c(A, \tau)$ be the number of chromosomes broken by τ in A. Let $\mathbf{T} = (\tau_1, \ldots, \tau_l)$ be a multi-break scenario transforming genome $A = A_1$ into genome $B = A_{l+1}$, where the multi-break τ_i transforms genome A_i into genome A_{i+1} for $i = 1, \ldots, l$. Let $c(\mathbf{T}, \tau_i) = c(A_i, \tau_i)$ be the number of chromosomes broken by τ_i *in* $\mathbf{T}$. We define $c^*(\mathbf{T})$, the *chromosome number* of $\mathbf{T}$, to be the maximum number of chromosomes broken by any multi-break in $\mathbf{T}$; i.e., $c^*(\mathbf{T}) = \max_{1 \leq i \leq l} c(\mathbf{T}, \tau_i)$. Let $\mathcal{I}(A, B)$ be the set of ISA multi-break scenarios that transform A into B. We aim to find the minimum chromosome number $c(A, B) = \min_{\mathbf{T} \in \mathcal{I}(A,B)} c^*(\mathbf{T})$ of a ISA multi-break scenario transforming A into B, described formally in the following problem.

▶ **Problem 1** (The Minimum Chromosome Number or MCN problem). *Given genomes A and B find the minimum chromosome number $c(A, B) = \min_{\mathbf{T} \in \mathcal{I}(A,B)} c^*(\mathbf{T})$, over all ISA multi-break scenarios that transform A into B.*

We say that the MCN problem is *trivial* for genomes A and B if $c(A, B) = c(\mathbf{T}_t)$, where $\mathbf{T}_t$ is the trivial ISA multi-break scenario containing a single multi-break. The simplest non-trivial examples of the MCN problem are for genomes A and B that admit exactly three ISA multi-break scenarios: $\mathbf{T}_t$, $\mathbf{T} = (\tau, \tau')$ and $\mathbf{T}' = (\tau', \tau)$ (See Section 2.3). In this case it can happen that both $\mathbf{T}$ and $\mathbf{T}'$ but not $\mathbf{T}_t$ have the minimum chromosome number $c(A, B)$ (Figure 2, top), it can also happen that only $\mathbf{T}$ but not $\mathbf{T}'$ and $\mathbf{T}_t$ has the minimum chromosome number $c(A, B)$ (Figure 2, bottom).

To solve the MCN problem we first partition the chromosomes of A and B into subsets $\{A_1, \ldots, A_m\}$ and $\{B_1, \ldots, B_m\}$ such that $c(A, B) = \max_{i \leq m} c(A_i, B_i)$. We perform this partition with a help of the *chromosome graph* $C(A, B)$ whose vertices are the block extremities, and edges are the adjacencies of A, the adjacencies of B and the synteny blocks. Note that for a chromosome h of A or B all the synteny blocks and adjacencies of h belong to the same connected component H of the chromosome graph $C(A, B)$ (Figure 3). We say that chromosome h is *included* in component H.

▶ **Theorem 1.** *Let $\{H_1, \ldots, H_m\}$ be the connected components of the chromosomes graph $C(A, B)$. Let A_i (resp. B_i) be the genome consisting of the chromosomes of A (resp. B) that are in H_i. Then A_i and B_i are co-tailed and contain the same synteny blocks. Further, the minimum chromosome number $c(A, B) = \max_{i \leq m} c(A_i, B_i)$.*

We find the minimum chromosome number $c(A_i, B_i) = \min_{\mathbf{T} \in \mathcal{I}(A_i, B_i)} c^*(\mathbf{T})$ by applying to A_i one by one all the ISA multi-break scenarios for A_i and B_i. We iterate over these scenarios with a help of a bijection introduced in Section 2.3 between $\mathcal{I}(A_i, B_i)$ and a set that we can enumerate.

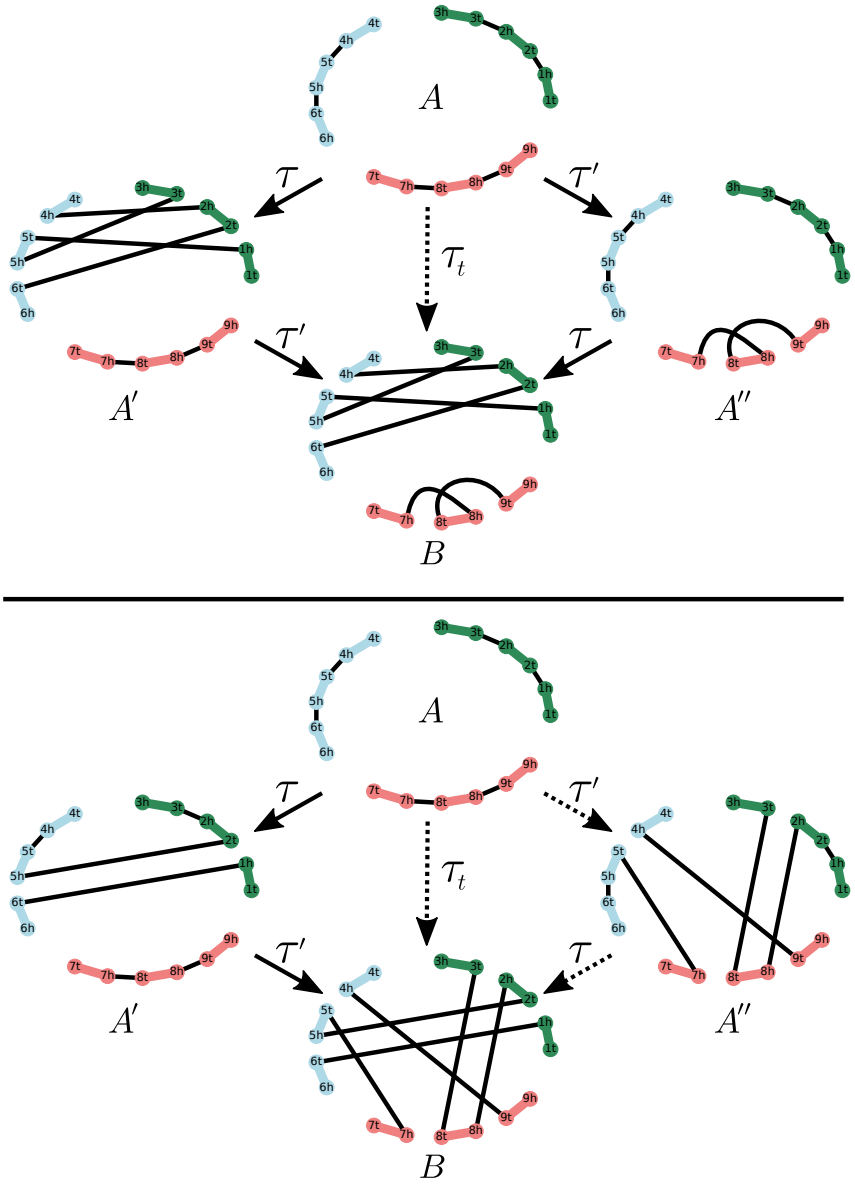


Figure 2 Two examples of genomes A and B for which the Minimum Chromosome Number problem is non-trivial. In both cases there exist three ISA multi-break scenarios $\mathbf{T}_t = (\tau_t)$, $\mathbf{T} = (\tau, \tau')$ and $\mathbf{T}' = (\tau', \tau)$. Solid arrows indicate scenarios with the minimum chromosome number $c(A, B)$, while dashed arrows indicate scenarios with chromosome number greater than $c(A, B)$. (Top) Genomes A and B for which both $\mathbf{T}$ and $\mathbf{T}'$ have the minimum chromosome number; i.e., $c^*(\mathbf{T}) = c^*(\mathbf{T}') = c(A, B) = 2$, while $c^*(\mathbf{T}_t) = 3$. (Bottom) Genomes A and B for which only $\mathbf{T}$ has the chromosome number equal to the minimum chromosome number; i.e., $c^*(\mathbf{T}) = c(A, B) = 2$, while $c^*(\mathbf{T}_t) = c^*(\mathbf{T}') = 3$. Note that in this case τ' breaks two chromosomes in $\mathbf{T}$ but three in $\mathbf{T}'$. Thus, the number of chromosomes broken by a multi-break can vary across the ISA multi-break scenarios.

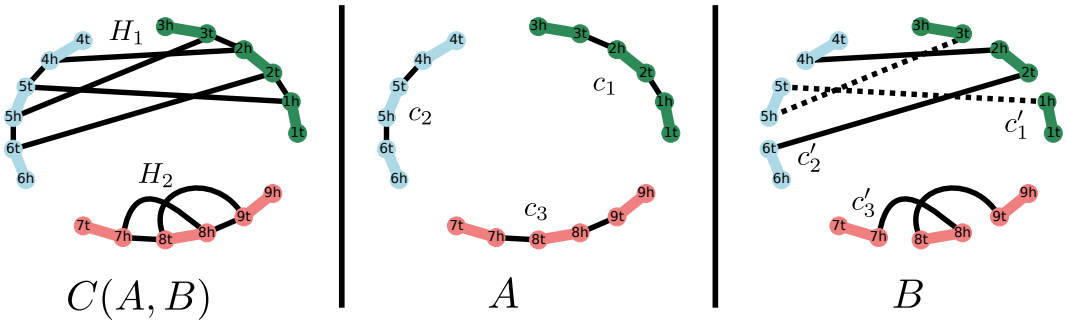


Figure 3 (Left) The chromosome graph $C(A, B)$ has two connected components H_1 and H_2 that respectively include subsets of chromosomes $A_1 = \{c_1, c_2\}$ and $A_2 = \{c_3\}$ of genome A, and $B_1 = \{c'_1, c'_2\}$ and $B_2 = \{c'_3\}$ of genome B. (Right) The adjacencies of a chromosome c'_1 in genome B are dashed in order to distinguish them from the adjacencies of a chromosome c'_2. Note that genomes A_1 and B_1 are co-tailed and contain the same synteny blocks, and similarly for A_2 and B_2.

2.3 Enumeration of ISA multi-break scenarios

In this section we describe a bijection involving ISA multi-break scenarios and the *breakpoint graph* [6, 4], a data structure that is routinely used for a pairwise comparison of genomes, and is defined as follows. The *breakpoint graph* $G(A, B)$ is a 2-edge-colored graph whose vertices are the block extremities, and black and gray edges are respectively the adjacencies of the genomes A and B. Every vertex in $G(A, B)$ is incident to either one black and one gray edge, or to no edges at all. This means that all the non-empty connected components of $G(A, B)$ are alternating cycles whose edges alternate between black and gray. In what follows we say that a *cycle* of $G(A, B)$ is an alternating cycle with at least four edges. It turns out that the ISA multi-break scenarios are related to the set $\mathcal{C}(A, B)$ of the subsets of the cycles of the breakpoint graph $G(A, B)$. Let P be an element of $\mathcal{C}(A, B)$, and let α and β be respectively the black and the gray edges in P. In the proof of Lemma 2 we show that $\tau(P) = (\alpha, \beta)$ is a multi-break. Similarly, let $\mathcal{P}(A, B)$ be the set of the ordered partitions of the cycles of the breakpoint graph, and let $\mathbf{P} = (P_1, \ldots, P_l)$ be an element in $\mathcal{P}(A, B)$. In Lemma 2 we establish that a sequence of multi-breaks $\mathbf{T}(\mathbf{P}) = (\tau(P_1), \ldots, \tau(P_l))$ is a ISA multi-break scenario for A and B (Figure 4).

▶ **Lemma 2.** *For genomes A and B, the function $\mathbf{T} : \mathcal{P}(A, B) \to \mathcal{I}(A, B)$ between the set $\mathcal{P}(A, B)$ of the ordered partitions of the cycles of the breakpoint graph $G(A, B)$ and the set $\mathcal{I}(A, B)$ of the ISA multi-break scenarios for A and B is a bijection.*

Due to Lemma 2, all the ISA multi-break scenarios for A and B perform the same total number of breaks and contain at most m multi-breaks, where m is the number of the cycles of the breakpoint graph. Given these observations one might expect that there always exists a ISA multi-break scenario for A and B with the minimum chromosome number that contains exactly m multi-breaks, however note that this is not the case for the genomes presented in Figure 4.

2.4 The chromosome number of a complex rearrangement

Current high-throughput DNA sequencing technologies do not measure the rearranged genome B, but rather measure only a set $\mathcal{B}$ of novel adjacencies derived from a sequencing sample. This set $\mathcal{B}$ of novel adjacencies may not correspond to a set of novel adjacencies of a

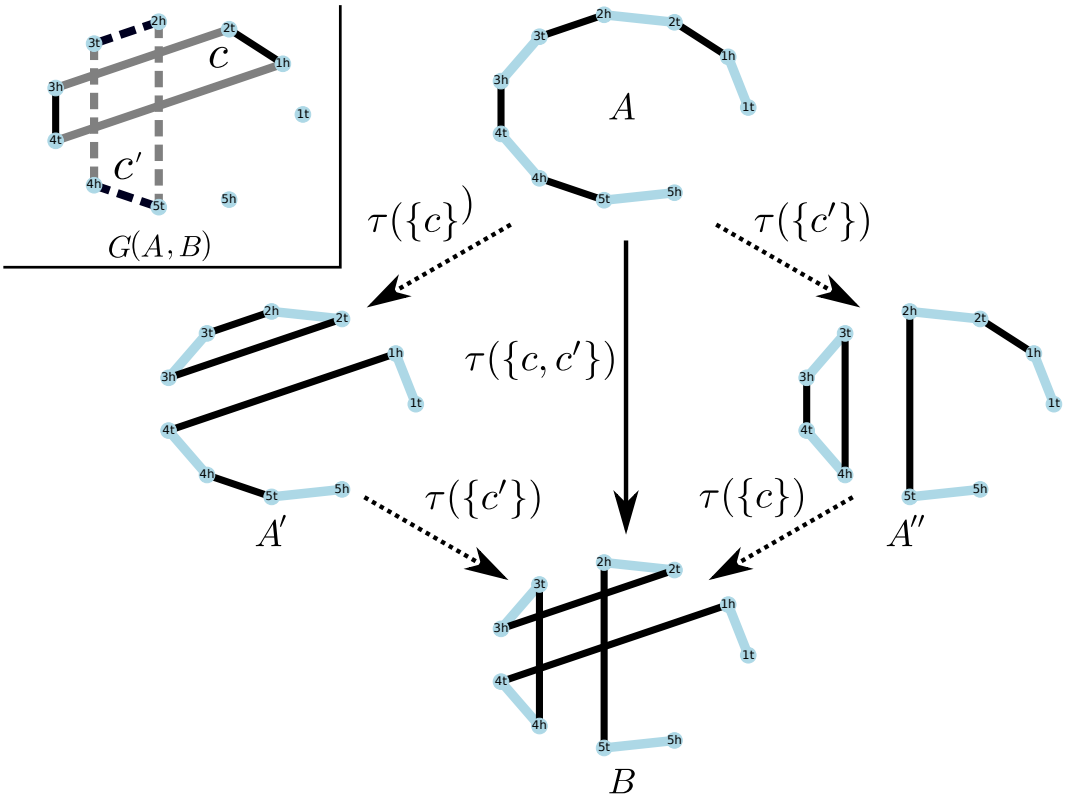


Figure 4 (Left) The breakpoint graph $G(A, B)$ has two cycles c (solid lines) and c' (dashed lines) that admit three ordered partitions $\mathbf{P} = (\{c\}, \{c'\})$ $\mathbf{P}' = (\{c'\}, \{c\})$ and $\mathbf{P}_t = (\{c, c'\})$. (Middle) ISA multi-break scenarios $\mathbf{T}(\mathbf{P}) = (\tau(\{c\}), \tau(\{c'\}))$, $\mathbf{T}(\mathbf{P}') = (\tau(\{c'\}), \tau(\{c\}))$ and $\mathbf{T}_t = \mathbf{T}(\mathbf{P}_t) = (\tau(\{c, c'\}))$ for A and B. The minimum chromosome number is $c(A, B) = 1 = c^*(\mathbf{T}_t)$, while $c^*(\mathbf{T}(\mathbf{P})) = c^*(\mathbf{T}(\mathbf{P}')) = 2$.

unique genome B; for example, $\mathcal{B}$ might be missing some adjacencies or include erroneous adjacencies [1]. The set $\mathcal{B}$ might also include adjacencies from multiple different genomes present in the sample; for example, DNA sequencing data from a bulk tumor is often a mixture of the genomes of different subclones [2]. What is more, a complex rearrangement might not be a multi-break; for example, chromothripsis can delete synteny blocks and chromoanasynthesis can amplify synteny blocks [28]. Finally, even if we were to obtain a rearranged genome B, it might not be co-tailed with the reference genome A. These observations above limit the scope of the Minimum Chromosome Number (MCN) problem.

Below, we introduce the Minimum Chromosome Number of a Complex Rearrangement (MCNR) problem that overcomes the limitations of the Minimum Chromosome Number (MCN) problem. In the MCNR problem our input no longer consists of co-tailed genomes A and B with the same synteny blocks, but of a reference genome A, a set $\mathcal{B}$ of novel adjacencies and a subset $\beta \subseteq \mathcal{B}$ of novel adjacencies that are proposed to result from a complex rearrangement. This input is motivated by the cancer genomics literature which which identifies such subsets $\beta \subseteq \mathcal{B}$ [5, 23, 12, 21, 16]. We propose to evaluate $(A, \mathcal{B}, \beta)$ according to the number of chromosomes broken in an intermediate genome by the complex rearrangement that introduced novel adjacencies β.

A widely reported measure of the "complexity" of a complex rearrangement is the number of reference chromosomes that are *affected by* or *involved in* the adjacencies β introduced by this rearrangement [33, 5, 36, 12, 7]. A chromosome is said to be affected by β if that chromosome includes a block extremity incident to an adjacency in β. Unlike in Section 2.1, in the cancer genomics literature the complex rearrangement is not supposed to be a multi-break; however, note that if there exists a subset α of the adjacencies of the reference genome A such that $\tau = (\alpha, \beta)$ is a multi-break, then the number of chromosomes broken by τ in the reference genome A is equal to the number of chromosomes affected by α in A, and by β in A. In the previous work only the number of chromosomes affected by β in the reference genome was analyzed, however, as it was briefly mentioned by Cortés-Ciriano et al. [12], the complex rearrangement could have rearranged an intermediate genome. Here, we aim to find $c(A, B, \beta)$, the *chromosome number* of β, that is the minimum number of chromosomes affected by β in an intermediate genome under the infinite sites assumption.

First, we formally define the notion of a ISA intermediate genome. As above, the *breakpoint graph* $G(A, \mathcal{B})$ is the 2-edge-colored graph whose black and gray edges are respectively the adjacencies of A and $\mathcal{B}$.

▶ **Definition 1** (ISA intermediate genome). *A multi-break $\tau = (\alpha, \beta)$ is a* ISA multi-break *for a genome A and a set of adjacencies $\mathcal{B}$, if α and β are respectively black and gray edges of a subset of the cycles of the breakpoint graph $G(A, \mathcal{B})$. A genome A' is a* ISA intermediate genome *for A, $\mathcal{B}$ and a subset $\beta \subseteq \mathcal{B}$, if $A = A'$ or if it can be obtained from A by a ISA multi-break $\tau = (\alpha', \beta')$ for A and $\mathcal{B}$ that satisfies $\beta \cap \beta' = \emptyset$.*

Using this definition, we define the following problem.

▶ **Problem 2** (The Minimum Chromosome Number of a Complex Rearrangement or MCNR problem). *Given a genome A, a set of adjacencies $\mathcal{B}$, and a subset $\beta \subseteq \mathcal{B}$, find the chromosome number $c(A, \mathcal{B}, \beta)$ defined as the minimum number of chromosomes affected by β over all ISA intermediate genomes for A, $\mathcal{B}$ and β.*

We say that the MCNR problem is *trivial* if $c(A, \mathcal{B}, \beta)$ is equal to the number of chromosomes affected by β in the reference genome A. The simplest non-trivial example of the MCNR problem is for a triplet $(A, \mathcal{B}, \beta)$ that admits two ISA intermediate genomes A and A' where β affects fewer chromosomes in A' than in A (Figure 5).

To solve the MCNR problem we first partition the chromosomes of A and the adjacencies in $\mathcal{B}$ into subsets $\{A_1, \ldots, A_m\}$ and $\{\mathcal{B}_1, \ldots, \mathcal{B}_m\}$ such that $c(A, \mathcal{B}, \beta) = \Sigma_{i=1}^{m} c(A_i, \mathcal{B}_i, \mathcal{B}_i \cap \beta)$. We perform this partition with a help of the *chromosome graph* $C(A, \mathcal{B})$ whose vertices are the block extremities, and edges are the adjacencies of A, the adjacencies in $\mathcal{B}$ and the synteny blocks.

▶ **Theorem 2.** *Let $\{H_1, \ldots, H_m\}$ be the connected components of the chromosomes graph $C(A, \mathcal{B})$. Let A_i be the genome consisting of the chromosomes of A that are in H_i, and let $\mathcal{B}_i$ be the adjacencies in $\mathcal{B}$ that are in H_i. Then the chromosome number $c(A, \mathcal{B}, \beta)$ is equal to $\Sigma_{i=1}^{m} c(A_i, \mathcal{B}_i, \mathcal{B}_i \cap \beta)$.*

We find the chromosome number $c(A_i, \mathcal{B}_i, \mathcal{B}_i \cap \beta)$ by iterating over all the ISA intermediate genomes for $(A_i, \mathcal{B}_i, \mathcal{B}_i \cap \beta)$. We perform this step via a bijection that, by definition, exists between the ISA intermediate genomes for $(A_i, \mathcal{B}_i, \mathcal{B}_i \cap \beta)$ and the subsets of the cycles of the breakpoint graph $G(A_i, \mathcal{B}_i)$ that do not contain gray edges from $\mathcal{B}_i \cap \beta$.

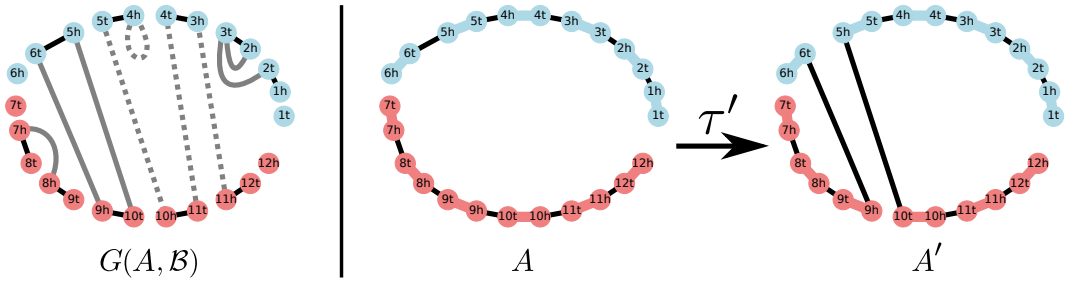


Figure 5 A triplet $(A, \mathcal{B}, \beta)$ for which the Minimum Chromosome Number of a Complex Rearrangement problem is non-trivial. (Left) The breakpoint graph $G(A, \mathcal{B})$ with a subset $\beta \subset \mathcal{B}$ of novel adjacencies (dashed lines) that affects both reference chromosomes. Note that multiple novel adjacencies sharing the same block extremity (3_t) and self-loops (4_h) might occur in both $\mathcal{B}$ and β. (Right) A ISA intermediate genome A' in which β affects a single chromosome. Note that the breakpoint graph $G(A, \mathcal{B})$ contains a single cycle, and A together with A' are the only ISA intermediate genomes for A and $\mathcal{B}$.

3 Results

3.1 The minimum chromosome number of a congenital complex rearrangement

We solved the Minimum Chromosome Number problem for five human genomes harboring congenital complex rearrangements that affect at least three chromosomes [14, 11, 13, 30]. All five genomes are co-tailed with the reference genome. For one of these genomes, the human genome labeled *Case 1* in Eisfeldt et al. [14], we identified a ISA multi-break scenario with a lower chromosome number than previously suggested. Specifically, the published analysis suggested that this genome resulted from a single complex rearrangement that broke chromosomes 2, 8 and 15. In contrast, we derived a ISA multi-break scenario consisting of a multi-break breaking chromosomes 2 and 8 followed by a multi-break breaking chromosome 15 and one of the previously rearranged chromosomes (Figure 6).

For the other four genomes – specifically *Case TL010* and *Case UTR22* from Collins et al. [11], the genome in Eisfeldt et al. [13] and the genome in Plesser et al. [30] – we computed the minimum chromosome number of a ISA multi-break scenario to be equal to 4, which is the same as in the published analysis. Note that to analyze the genome described in Eisfeldt et al. [13], we enumerate 75 ISA multi-break scenarios, the most of the five genomes analyzed.

3.2 The minimum chromosome number of a chromothripsis in cancer

Chromothripsis, is one the most studied types of complex rearrangements in cancer genomes, and is defined as a shattering of *one or a few chromosomes* followed by a random joining of the resulting chromosomal fragments [33, 28]. Some analyses have reported chromothripsis events that include novel adjacencies containing extremities from as many as eighteen reference chromosomes [16, 12, 36]. This seems like an extremely large number of chromosomes to be involved in a simultaneous event, and the current understanding of the molecular mechanisms of genome rearrangements do not indicate simultaneous rearrangements involving more than three chromosomes [41, 35].

To evaluate this discrepancy, we analyzed 252 cancer genomes identified by Hadi et al. [16] as harboring a chromothripsis event that affects at least two reference chromosomes. These genomes form a subset of the 2778 cancer genomes from multiple cancer types with

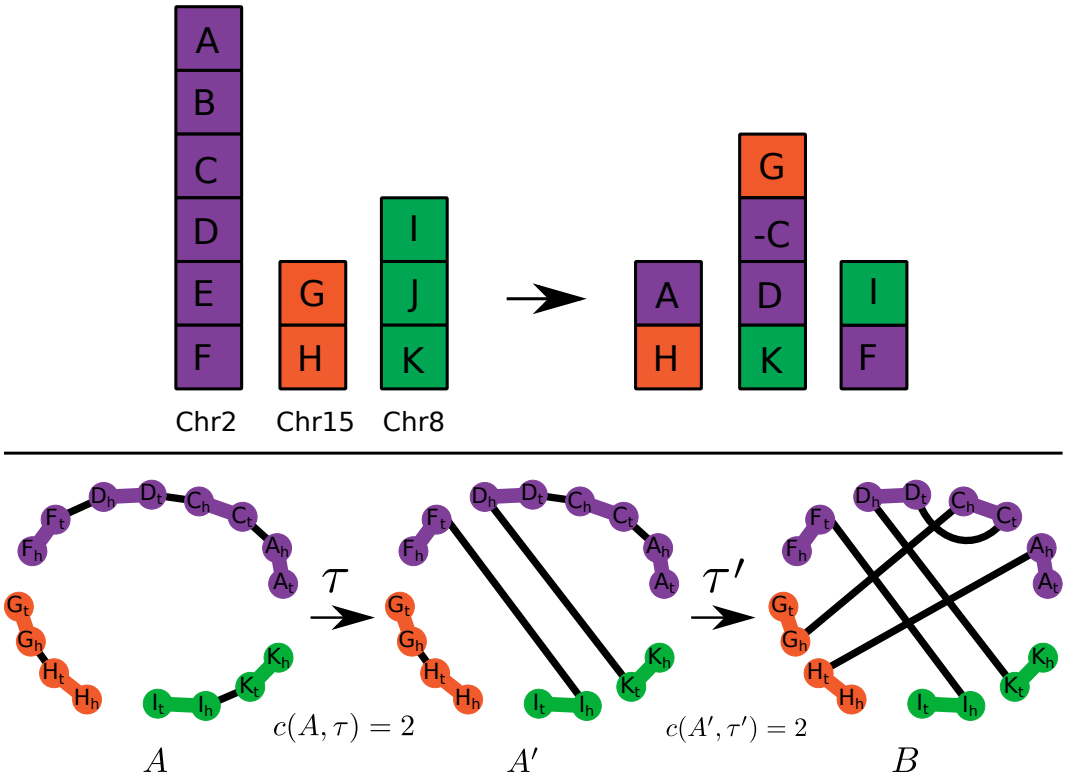


Figure 6 Congenital complex rearrangement in human genome *Case 1* from Eisfeldt et al. [14]. (Top left) Reference chromosomes 2 (purple), 8 (green), and 15 (orange) are partitioned into eleven synteny blocks `A` to `K` in accordance with notation published by Eisfeldt et al. [14]. (Top right) Rearranged chromosomes lack synteny blocks `B`, `E` and `J`, and the direction of the block `C` is inverted as indicated by the minus sign. The published analysis suggested that reference chromosomes 2, 8 and 15 were transformed into the rearranged ones by a single complex rearrangement. (Bottom) Genome A is obtained from the reference chromosomes by removing the blocks `B`, `E` and `J`, while genome B corresponds to the rearranged chromosomes. A ISA multi-break scenario (τ, τ') for A and B has the chromosome number equal to two, which contrasts with the published complex rearrangement that simultaneously breaks three chromosomes.

whole-genome sequencing data that are available through gGnome.js portal [16]. In particular, we analyzed a total of 288 triplets $(A, \mathcal{B}, \beta)$ generated from a data structure called the *JaBbA graph* described in [16] (see Section 3.2.3 for further details). These triplets consist of a reference genome A, a set $\mathcal{B}$ of novel adjacencies derived from a cancer sample, and a subset $\beta \subseteq \mathcal{B}$ identified as introduced by a chromothripsis event by Hadi et al. [16]. Since some cancer genomes were identified to harbor multiple chromothripsis events, the number of triplets, 288, is larger than the number 252 of genomes.

3.2.1 Chromothripsis event breaks less chromosomes than it affects

We solved the Minimum Chromosome Number of a Complex Rearrangement problem for all the 288 triplets $(A, \mathcal{B}, \beta)$. For 5 triplets, we identified a ISA intermediate genome in which β affects fewer chromosomes than in the reference genome. This illustrates that a chromothripsis event could have broken fewer chromosomes in an intermediate genome than it affects in the reference.

One such triplet $(A, \mathcal{B}, \beta)$ is from a prostate adenocarcinoma sample `PR-3042` (Figure 7) in which β[1] affects chromosomes 4, 5 and 10. In this sample we found a ISA 3-break $\tau' = (\alpha', \beta')$ that breaks chromosomes 4 and 5, and transforms the reference genome A into a genome in which β affects two chromosomes. This suggests that the chromothripsis event could have broken two rearranged chromosomes instead of the three reference chromosomes affected by β.

3.2.2 The number of affected chromosomes is overestimated

Since identifying a subset $\beta \subseteq \mathcal{B}$ of the novel adjacencies introduced by a chromothripsis event is challenging, we further analyzed the 288 triplets $(A, \mathcal{B}, \beta)$ considering the possibility that false positives in β might lead to an overestimation of the number of the affected reference chromosomes. For 17/288 triplets we found a multi-break $\tau' = (\alpha', \beta')$ such that $\beta \setminus \beta'$ affects fewer reference chromosomes than β and the ratio $\frac{|\beta \cap \beta'|}{|\beta|}$ is less than 0.2. The latter property ensures that only a small fraction of the adjacencies in β are proposed to be false positives, while the former establishes that the multi-break τ' and a chromothripsis event introducing adjacencies $\beta \setminus \beta'$ is a simpler evolutionary explanation than a chromothripsis event introducing adjacencies β.

One such triplet $(A, \mathcal{B}, \beta)$, shown in Figure 8, is from a Barrett's esophagus sample `740_T2_35_24253` in which β affects chromosomes 2, 3, 4, 17 and 22. In this sample we found a ISA 2-break $\tau' = (\alpha', \beta')$ that breaks chromosomes 4 and 17, and a ISA 3-break $\tau'' = (\alpha'', \beta'')$ that breaks chromosomes 2, 3, and 4, such that $\beta \setminus (\beta' \cup \beta'')$ only affects chromosomes 4 and 22. This suggests that the chromothripsis event could have broken two chromosomes (4 and 22) instead of five chromosomes (2, 3, 4, 17 and 22) affected by β.

3.2.3 Processing JaBbA graphs

A JaBbA graph is a data structure that stores synteny blocks (called *intervals*), reference adjacencies (called *REF connections*) and novel adjacencies (called *ALT connections*). See for example the JaBbA graph of a Barrett's esophagus sample `740_T2_35_24253` downloaded from gGnome.js portal [16].

A reference genome A_J and a set of novel adjacencies $\mathcal{B}_J$ can be immediately retrieved from the JaBbA graph, however the breakpoint graph $G(A_J, \mathcal{B}_J)$ thus obtained would have almost no cycles. One technical reason for this is that the synteny blocks are identified with single-nucleotide precision in JaBbA graphs and the breakpoint regions between adjacent synteny blocks are empty. However genome rearrangements oftentimes result in duplications or deletions of the regions surrounding chromosomal breaks [23], and in the genome rearrangement literature [29, 2] such regions are usually interpreted as non-empty breakpoint regions instead of synteny blocks. We thus identify synteny blocks potentially deleted or duplicated by genome rearrangements, and incorporate this information to obtain genome graph A and novel adjacencies $\mathcal{B}$ used in our analysis (see Figure 9 for further details).

3.2.4 Summary

We presented two examples for how a chromothripsis event could have broken fewer chromosomes than it affects in the reference genome. The analyzed chromothripsis events in the gGnome.js portal affect up to seven chromosomes, and in future work it would be of interest to analyze the chromothripsis events identified by Cortés Ciriano et al. [12] that affect up to eighteen chromosomes.

[1] Links to gGnome.js portal [16] for visualizing the JaBbA graphs were tested to work on Google Chrome.

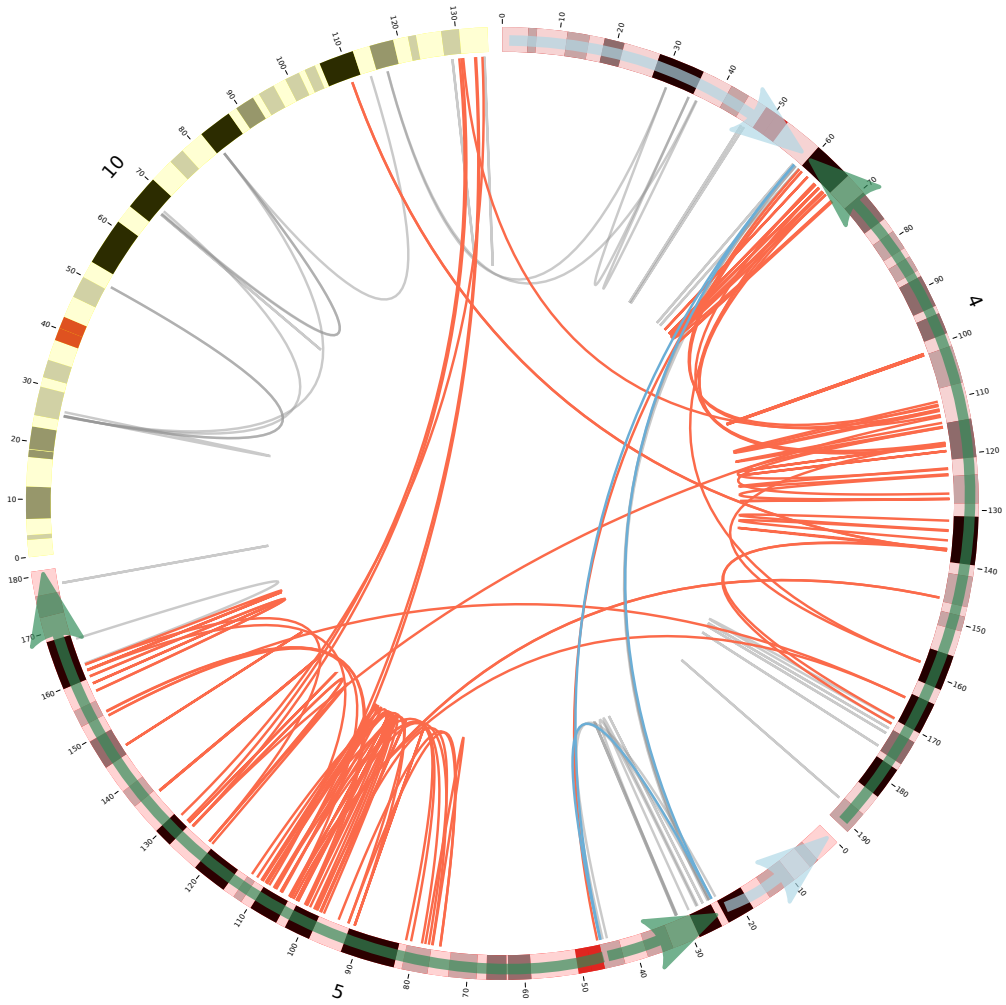


Figure 7 A Circos plot [20] of chromosomes 4, 5, and 10 from prostate adenocarcinoma sample PR-3042. Arcs in the plot indicate the novel adjacencies $\mathcal{B}$. Red arcs are the adjacencies β identified as introduced by a chromothripsis event by Hadi et al. [16], the three blue arcs are the adjacencies β' introduced by a ISA 3-break $\tau' = (\alpha', \beta')$ found by our method, and gray arcs are the remaining adjacencies $\mathcal{B} \setminus (\beta \cup \beta')$. The multi-break τ' transforms chromosomes 4 and 5 into the rearranged chromosomes whose segments are indicated by green and blue arrows. Red arcs β affect all three chromosomes (4, 5, and 10) in the reference genome, but only two chromosomes (10 and the green rearranged chromosome) in the rearranged genome. We suggest that the multi-break τ' preceded the chromothripsis event that broke two chromosomes.

4 Discussion

In this work, we introduce a unified model for simple and complex genome rearrangements where each mutation is modeled as a multi-break. Within this model we formulate a novel maximum parsimony principle based on minimizing the number of chromosomes broken by a rearrangement. We formulate the problem of minimizing the chromosome number for the case of a pair of genomes and for the case of a genome and a set of novel adjacencies derived from a sequencing sample. We present exact algorithms to solve both problems under the

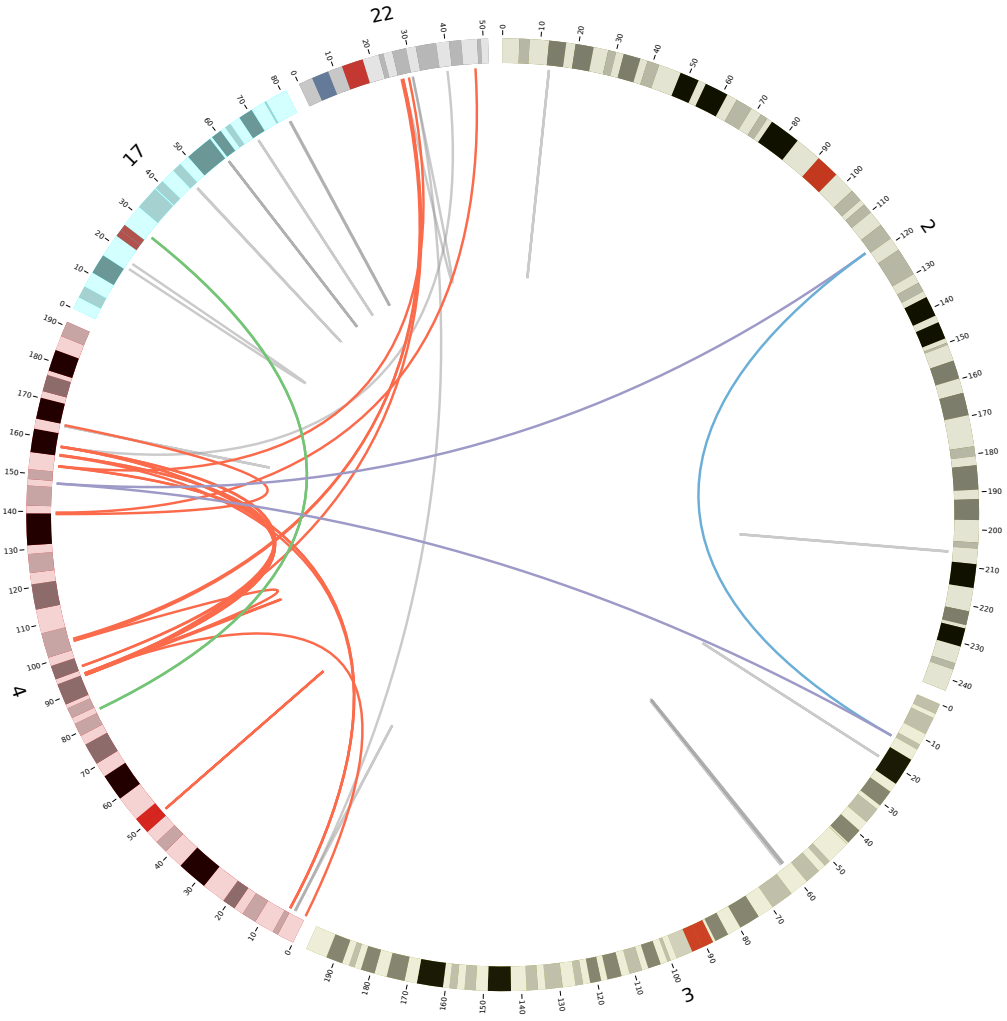


Figure 8 A Circos plot [20] of chromosomes 2, 3, 4, 17, and 22 from a Barrett's esophagus sample `740_T2_35_24253`. Arcs in the plot indicate the novel adjacencies $\mathcal{B}$. A subset $\beta \subseteq \mathcal{B}$, affecting chromosomes 2, 3, 4, 17, and 22, is identified as introduced by a chromothripsis event by Hadi et al. [16], however our analysis partitions β into three subsets: two green arcs β' (indistinguishable in this plot) introduced by a ISA 2-break $\tau' = (\alpha', \beta')$; two purple arcs introduced by a ISA 3-break $\tau'' = (\alpha'', \beta'')$; and nineteen red arcs $\beta \setminus (\beta' \cup \beta'')$. The blue arc is introduced by the 3-break τ'' in addition to the two purple arcs, while gray arcs are the remaining adjacencies $\mathcal{B} \setminus (\beta \cup \beta' \cup \beta'')$. We suggest that the chromothripsis event only introduced the red arcs $\beta \setminus (\beta' \cup \beta'')$ affecting chromosomes 4 and 22, while β' and β'' were introduced instead by the ISA multi-breaks τ' and τ''.

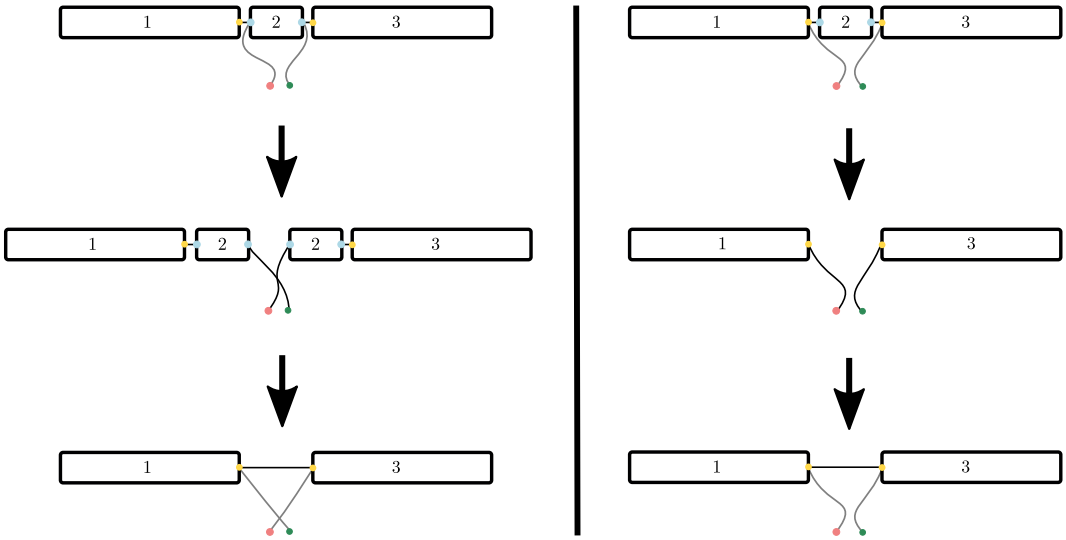


Figure 9 (Top left) A portion of a reference genome (black) and novel adjacencies (gray) retrieved from a JaBbA graph with synteny block 2 being *duplicated by a rearrangement*. We say that a synteny block is duplicated by a rearrangement if it contains less than 1kbp, both of its extremities (blue) are incident to separate novel adjacencies (gray), neither of the adjacent extremities (yellow) is incident to a novel adjacency, and the copy numbers provided in the JaBbA graph of blocks 1 and 3 are lower than the copy number of block 2. (Middle left) We assume that synteny block 2 got duplicated by a rearrangement that resulted in the depicted local organization of a genome. (Bottom left) A portion of the updated reference genome and novel adjacencies that are used in our analysis. (Top right) A portion of a reference genome (black) and novel adjacencies (gray) retrieved from a JaBbA graph with synteny block 2 being *deleted by a rearrangement*. We say that a synteny block is deleted by a rearrangement if it contains less than 100kbp, neither of its extremities (blue) is incident to a novel adjacency, both adjacent extremities (yellow) are incident to separate novel adjacencies (gray), neither of the neighboring synteny blocks (1 and 3) is duplicated by a rearrangement, and the copy numbers provided in the JaBbA graph of blocks 1 and 3 are greater than the copy number of block 2. (Middle right) We assume that synteny block 2 got deleted by a rearrangement that resulted in the depicted local organization of a genome. (Bottom right) A portion of the updated reference genome and novel adjacencies that are used in our analysis.

infinite sites assumption and apply these algorithms to analyze 5 human genomes harboring congenital complex rearrangements and 252 cancer genomes harboring chromothripsis events. For one human genome and 22 cancer genomes we compute multi-break scenarios containing complex rearrangements that affect fewer chromosomes than previously reported.

While multi-breaks have previously been used to model complex genome rearrangements [4, 10], to our knowledge the present work is the first to use the number of chromosomes broken by a rearrangement as a constraint on a rearrangement scenario. For simple rearrangements, Yin et al. [38] briefly mention the problem of prioritizing intra-chromosomal rearrangements (inversions) over inter-chromosomal rearrangements (translocations); however, as far as we are aware, the problem remains open.

We note a number of limitations and directions for future work. First, the time complexities of the Minimum Chromosome Number and the Minimum Chromosome Number of a Complex Rearrangement problems remain unknown. It would be desirable to derive a more efficient algorithm to compute or approximate the minimum chromosome number. Second, our method is sensitive to missing novel adjacencies – if at least one novel adjacency introduced by a multi-break remains unidentified from the sequencing data, then this multi-break is excluded from our analyses. Such missing novel adjacencies are abundant in short-read sequencing data [1, 8]. It would be helpful to further extend our model to address missing and erroneous novel adjacencies present in real data, although such issues will also be reduced from improved identification of novel adjacencies from long-read sequencing data. Third, extending our genome representation and space of allowed rearrangements would yield more realistic reconstructions. In particular, following the standard approach in the genome rearrangement literature, we analyze a haploid representation of the genome. However, the human genome is diploid and assigning novel adjacencies to the correct chromosomal homolog [2] will be useful for analyzing complex rearrangements and counting the distinct homologous chromosomes involved in these rearrangements. Another extension is to incorporate additional events including duplication and loss of chromosomal regions. Finally, our enumeration algorithm could be used to solve other optimization problems for the ISA multi-break scenarios, such as minimizing the number of circular excisions (e.g. from ecDNA [34]) in a ISA multi-break scenario or maximizing the number of inversions.

References

1 Sergey Aganezov, Sara Goodwin, Rachel M Sherman, Fritz J Sedlazeck, Gayatri Arun, Sonam Bhatia, Isac Lee, Melanie Kirsche, Robert Wappel, Melissa Kramer, et al. Comprehensive analysis of structural variants in breast cancer genomes using single-molecule sequencing. *Genome research*, 30(9):1258–1273, 2020.

2 Sergey Aganezov and Benjamin J Raphael. Reconstruction of clone-and haplotype-specific cancer genome karyotypes from bulk tumor samples. *Genome research*, 30(9):1274–1290, 2020.

3 Max A Alekseyev and Pavel A Pevzner. Are there rearrangement hotspots in the human genome? *PLoS Computational Biology*, 3(11):e209, 2007.

4 Max A Alekseyev and Pavel A Pevzner. Multi-break rearrangements and chromosomal evolution. *Theoretical Computer Science*, 395(2-3):193–202, 2008.

5 Sylvan C Baca, Davide Prandi, Michael S Lawrence, Juan Miguel Mosquera, Alessandro Romanel, Yotam Drier, Kyung Park, Naoki Kitabayashi, Theresa Y MacDonald, Mahmoud Ghandi, et al. Punctuated evolution of prostate cancer genomes. *Cell*, 153(3):666–677, 2013.

6 Vineet Bafna and Pavel A Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25(2):272–289, 1996.

7 Lisui Bao, Xiaoming Zhong, Yang Yang, and Lixing Yang. Mutational signatures of complex genomic rearrangements in human cancer. *bioRxiv*, 2021.

8 Julie M Behr, Xiaotong Yao, Kevin Hadi, Huasong Tian, Aditya Deshpande, Joel Rosiene, Titia de Lange, and Marcin Imielinski. Loose ends in cancer genome structure. *bioRxiv*, 2021.

9 Priscila Biller, Laurent Guéguen, Carole Knibbe, and Eric Tannier. Breaking good: accounting for fragility of genomic regions in rearrangement distance estimation. *Genome Biology and Evolution*, 8(5):1427–1439, 2016.

10 Laurent Bulteau, Guillaume Fertin, Géraldine Jean, and Christian Komusiewicz. Sorting by multi-cut rearrangements. *Algorithms*, 14(6):169, 2021.

11 Ryan L Collins, Harrison Brand, Claire E Redin, Carrie Hanscom, Caroline Antolik, Matthew R Stone, Joseph T Glessner, Tamara Mason, Giulia Pregno, Naghmeh Dorrani, et al. Defining the diverse spectrum of inversions, complex structural variation, and chromothripsis in the morbid human genome. *Genome biology*, 18(1):1–21, 2017.

12 Isidro Cortés-Ciriano, Jake June-Koo Lee, Ruibin Xi, Dhawal Jain, Youngsook L Jung, Lixing Yang, Dmitry Gordenin, Leszek J Klimczak, Cheng-Zhong Zhang, David S Pellman, et al. Comprehensive analysis of chromothripsis in 2,658 human cancers using whole-genome sequencing. *Nature genetics*, 52(3):331–341, 2020.

13 Jesper Eisfeldt, Maria Pettersson, Anna Petri, Daniel Nilsson, Lars Feuk, and Anna Lindstrand. Hybrid sequencing resolves two germline ultra-complex chromosomal rearrangements consisting of 137 breakpoint junctions in a single carrier. *Human Genetics*, pages 1–16, 2020.

14 Jesper Eisfeldt, Maria Pettersson, Francesco Vezzi, Josephine Wincent, Max Käller, Joel Gruselius, Daniel Nilsson, Elisabeth Syk Lundberg, Claudia MB Carvalho, and Anna Lindstrand. Comprehensive structural variation genome map of individuals carrying complex chromosomal rearrangements. *PLoS genetics*, 15(2):e1007858, 2019.

15 Chris D Greenman, Luca Penso-Dolfin, and Taoyang Wu. The complexity of genome rearrangement combinatorics under the infinite sites model. *Journal of Theoretical Biology*, 501:110335, 2020.

16 Kevin Hadi, Xiaotong Yao, Julie M Behr, Aditya Deshpande, Charalampos Xanthopoulakis, Huasong Tian, Sarah Kudman, Joel Rosiene, Madison Darmofal, Joseph DeRose, et al. Distinct classes of complex structural variation uncovered across thousands of cancer genome graphs. *Cell*, 183(1):197–210, 2020.

17 Sridhar Hannenhalli and Pavel A Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). In *Proceedings of IEEE 36th annual foundations of computer science*, pages 581–592. IEEE, 1995.

18 Sridhar Hannenhalli and Pavel A Pevzner. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM (JACM)*, 46(1):1–27, 1999.

19 Marcus Kinsella, Anand Patel, and Vineet Bafna. The elusive evidence for chromothripsis. *Nucleic acids research*, 42(13):8231–8242, 2014.

20 Martin Krzywinski, Jacqueline Schein, Inanc Birol, Joseph Connors, Randy Gascoyne, Doug Horsman, Steven J Jones, and Marco A Marra. Circos: an information aesthetic for comparative genomics. *Genome research*, 19(9):1639–1645, 2009.

21 Yeonghun Lee and Hyunju Lee. Integrative reconstruction of cancer genome karyotypes using InfoGenomeR. *Nature communications*, 12(1):1–13, 2021.

22 Mitchell L Leibowitz, Stamatis Papathanasiou, Phillip A Doerfler, Logan J Blaine, Lili Sun, Yu Yao, Cheng-Zhong Zhang, Mitchell J Weiss, and David Pellman. Chromothripsis as an on-target consequence of CRISPR–Cas9 genome editing. *Nature Genetics*, 53(6):895–905, 2021.

23 Yilong Li, Nicola D Roberts, Jeremiah A Wala, Ofer Shapira, Steven E Schumacher, Kiran Kumar, Ekta Khurana, Sebastian Waszak, Jan O Korbel, James E Haber, et al. Patterns of somatic structural variation in human cancer genomes. *Nature*, 578(7793):112–121, 2020.

24 Pengfei Liu, Ayelet Erez, Sandesh C Sreenath Nagamani, Shweta U Dhar, Katarzyna E Kołodziejska, Avinash V Dharmadhikari, M Lance Cooper, Joanna Wiszniewska, Feng Zhang, Marjorie A Withers, et al. Chromosome catastrophes involve replication mechanisms generating complex genomic rearrangements. *Cell*, 146(6):889–903, 2011.

25 Jian Ma, Aakrosh Ratan, Brian J Raney, Bernard B Suh, Webb Miller, and David Haussler. The infinite sites model of genome evolution. *Proceedings of the National Academy of Sciences*, 105(38):14254–14261, 2008.

26 Layla Oesper, Simone Dantas, and Benjamin J Raphael. Identifying simultaneous rearrangements in cancer genomes. *Bioinformatics*, 34:346–352, 2018.

27 R Gonzalo Parra, Moritz J Przybilla, Milena Simovic, Hana Susak, Manasi Ratnaparkhe, John KL Wong, Verena Koerber, Philipp Mallm, Martin Sill, Thorsten Kolb, et al. Single cell multi-omics analysis of chromothriptic medulloblastoma highlights genomic and transcriptomic consequences of genome instability. *bioRxiv*, 2021.

28 F Pellestor, JB Gaillard, A Schneider, J Puechberty, and V Gatinois. Chromoanagenesis, the mechanisms of a genomic chaos. In *Seminars in Cell & Developmental Biology*. Elsevier, 2021.

29 Pavel Pevzner and Glenn Tesler. Human and mouse genomic sequences reveal extensive breakpoint reuse in mammalian evolution. *Proceedings of the National Academy of Sciences*, 100(13):7672–7677, 2003.

30 Morasha Plesser Duvdevani, Maria Pettersson, Jesper Eisfeldt, Ortal Avraham, Judith Dagan, Ayala Frumkin, James R Lupski, Anna Lindstrand, and Tamar Harel. Whole-genome sequencing reveals complex chromosome rearrangement disrupting NIPBL in infant with Cornelia de Lange syndrome. *American Journal of Medical Genetics Part A*, 182(5):1143–1151, 2020.

31 Claire Redin, Harrison Brand, Ryan L Collins, Tammy Kammin, Elyse Mitchell, Jennelle C Hodge, Carrie Hanscom, Vamsee Pillalamarri, Catarina M Seabra, Mary-Alice Abbott, et al. The genomic landscape of balanced cytogenetic abnormalities associated with human congenital anomalies. *Nature genetics*, 49(1):36–45, 2017.

32 Pijus Simonaitis, Annie Chateau, and Krister Swenson. Weighted minimum-length rearrangement scenarios. In *19th International Workshop on Algorithms in Bioinformatics (WABI)*, pages 13–1, 2019.

33 Philip J Stephens, Chris D Greenman, Beiyuan Fu, Fengtang Yang, Graham R Bignell, Laura J Mudie, Erin D Pleasance, King Wai Lau, David Beare, Lucy A Stebbings, et al. Massive genomic rearrangement acquired in a single catastrophic event during cancer development. *cell*, 144(1):27–40, 2011.

34 Kristen M Turner, Viraj Deshpande, Doruk Beyter, Tomoyuki Koga, Jessica Rusert, Catherine Lee, Bin Li, Karen Arden, Bing Ren, David A Nathanson, et al. Extrachromosomal oncogene amplification drives tumour evolution and genetic heterogeneity. *Nature*, 543(7643):122–125, 2017.

35 Neil T Umbreit, Cheng-Zhong Zhang, Luke D Lynch, Logan J Blaine, Anna M Cheng, Richard Tourdot, Lili Sun, Hannah F Almubarak, Kim Judge, Thomas J Mitchell, et al. Mechanisms generating cancer genome complexity from a single cell division error. *Science*, 368(6488), 2020.

36 Natalia Voronina, John KL Wong, Daniel Hübschmann, Mario Hlevnjak, Sebastian Uhrig, Christoph E Heilig, Peter Horak, Simon Kreutzfeldt, Andreas Mock, Albrecht Stenzinger, et al. The landscape of chromothripsis across adult cancer types. *Nature communications*, 11(1):1–13, 2020.

37 Sophia Yancopoulos, Oliver Attie, and Richard Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16):3340–3346, 2005.

38 Xiao Yin and Daming Zhu. Sorting genomes by reversals and translocations. In *2009 Asia-Pacific Conference on Information Processing*, volume 2, pages 391–394. IEEE, 2009.

39 Ron Zeira and Ron Shamir. Sorting cancer karyotypes using double-cut-and-joins, duplications and deletions. *Bioinformatics*, 37(11):1489–1496, 2021.

40 Cinthya J Zepeda-Mendoza and Cynthia C Morton. The iceberg under water: unexplored complexity of chromoanagenesis in congenital disorders. *The American Journal of Human Genetics*, 104(4):565–577, 2019.

41 Cheng-Zhong Zhang, Alexander Spektor, Hauke Cornils, Joshua M Francis, Emily K Jackson, Shiwei Liu, Matthew Meyerson, and David Pellman. Chromothripsis from DNA damage in micronuclei. *Nature*, 522(7555):179–184, 2015.

A Proofs

A.1 Lemma 1

▶ **Lemma.** *Suppose that A and B are co-tailed genomes that contain the same synteny blocks, $E(A)$ are the adjacencies of A that are absent from B, and $E(B)$ are the adjacencies of B that are absent from A. The multi-break $\tau_t = (E(A), E(B))$ is the unique multi-break that transforms A into B.*

Proof. If genomes A and B are co-tailed and contain the same synteny blocks, then a block extremity is in some adjacency of A if and only if it is in some adjacency of B, and the same property holds for $E(A)$ and $E(B)$. As $E(A) \cap E(B) = \emptyset$ by construction, we obtain that $\tau_t = (E(A), E(B))$ is a multi-break that transforms A into B.

Now if a multi-break $\tau = (\alpha, \beta)$ transforms A into B, then $E(A) \subseteq \alpha$ and $E(B) \subseteq \beta$ as τ has to break the adjacencies in $E(A)$ and introduce the adjacencies in $E(B)$. What is more, if τ breaks an adjacency $e = \{u, v\}$ then, due to the definition of a multi-break, we obtain that it introduces an adjacency $f \neq e$ of B incident to u. However B contains a single adjacency incident to u, thus B does not contain e, which establishes that $\alpha = E(A)$. As $|\alpha| = |\beta|$ and $|E(A)| = |E(B)|$ we obtain that $\beta = E(B)$ and conclude that $\tau = \tau_t$. ◀

A.2 Theorem 1

▶ **Theorem.** *Let $\{H_1, \ldots, H_m\}$ be the connected components of the chromosomes graph $C(A, B)$. Genomes A_i and B_i consisting respectively of the chromosomes of A and B included in a component H_i are co-tailed and contain the same synteny blocks. Also, the minimum chromosome number $c(A, B)$ is equal to $\max_{i \leq m} c(A_i, B_i)$.*

Proof. Let S_i be the synteny blocks, and let U_i be the telomeres of a genome A included in a connected component H_i of the chromosome graph $C(A, B)$ for $i \leq m$. Due to A and B being co-tailed and containing the same synteny blocks, S_i are the synteny blocks, and U_i are the telomeres of both genomes A_i and B_i, which ensures that A_i and B_i are also co-tailed and contain the same synteny blocks.

Let $\mathbf{T}_i$ be a ISA multi-break scenario for A_i and B_i with the chromosome number $c^*(\mathbf{T}_i) = c(A_i, B_i)$ for $i \leq m$. A sequence $\mathbf{T}$ of multi-breaks obtained by concatenating the scenarios $\{\mathbf{T}_1, \ldots, \mathbf{T}_m\}$ is a ISA multi-break scenario for A and B with a chromosome number equal to $\max_{i \leq m} c(A_i, B_i)$, which establishes inequality $c(A, B) \leq c^*(T) = \max_{i \leq m} c(A_i, B_i)$.

We say that a multi-break $\tau = (\alpha, \beta)$ is *split* if α includes edges from more than one connected component of the chromosome graph $C(A, B)$. A multi-break scenario is *splitless* if it does not contain a split multi-break. Let $c_s(A, B)$ be the minimum chromosome number of a splitless ISA multi-break scenario for A and B. In what follows we show that $c(A, B) = c_s(A, B)$ and $c_s(A, B) \geq \max_{i \leq m} c(A_i, B_i)$.

First, let $\mathbf{T}$ be a ISA multi-break scenario for A and B with $c^*(\mathbf{T}) = c(A, B)$. If $\mathbf{T}$ is splitless, then $c(A, B) \geq c_s(A, B)$. Otherwise, let $\tau = (\alpha, \beta)$ be a split ISA multi-break in $\mathbf{T}$ with α and β including adjacencies from the connected components $\{H_{\sigma(1)}, \ldots, H_{\sigma}(l)\}$ of the chromosome graph $C(A, B)$, where $\sigma : \{1, \ldots, l\} \to \{1, \ldots, m\}$ is an injection for $l \leq m$. The scenario $\mathbf{T}$ being a ISA multi-break scenario means that α does not include adjacencies introduced by multi-breaks preceding τ in $\mathbf{T}$, and that β does not include adjacencies broken by the multi-breaks proceeding τ in $\mathbf{T}$, which ensures that α and β are respectively adjacencies of A and B, and thus are included among the edges of the chromosome graph $C(A, B)$. Partition the adjacencies α and β into subsets $\{\alpha_1, \ldots, \alpha_l\}$

and $\{\beta_1, \ldots, \beta_l\}$ where α_i and β_i respectively are the adjacencies of α and β included in a component $H_{\sigma(i)}$ of the chromosome graph. This way $\tau_i = (\alpha_i, \beta_i)$ a ISA multi-break for A and B that is not split, and the multi-break τ then can be replaced in the scenario $\mathbf{T}$ with a sequence of multi-breaks $((\alpha_1, \beta_1), \ldots, (a_l, \beta_l))$ to obtain a ISA multi-break scenario $\mathbf{T}'$ for A and B that has less split multi-breaks than $\mathbf{T}$. Let A' be a genome transformed by τ during the scenario $\mathbf{T}$. By construction, the multi-breaks $\{(\alpha_1, \beta_1), \ldots, (\alpha_l, \beta_l)\}$ break disjoint subsets of chromosomes of A', which ensures that the number of chromosomes broken by a multi-break τ during $\mathbf{T}$ is equal to the sum of the numbers of chromosomes broken by the multi-breaks $\{(\alpha_1, \beta_1), \ldots, (\alpha_l, \beta_l)\}$ during $\mathbf{T}'$, and thus $c^*(\mathbf{T}') \leq c^*(\mathbf{T})$. We proceed until a splitless ISA multi-break scenario for A and B with a chromosome number smaller than or equal to $c^*(\mathbf{T})$ is obtained, thus establishing that $c(A, B) = c^*(\mathbf{T}) \geq c_s(A, B)$. A splitless ISA multi-break scenario is also a ISA multi-break scenario, thus we have that $c(A, B) \leq c_s(A, B)$, and conclude that $c(A, B) = c_s(A, B)$.

Finally, let $\mathbf{T}$ be a splitless ISA multi-break scenario for A and B with $c^*(\mathbf{T}) = c_s(A, B)$, and let $\mathbf{T}_i$ be a subsequence of $\mathbf{T}$ that consists of the multi-breaks that break adjacencies in A_i. The subsequence $\mathbf{T}_i$ is a ISA multi-break scenario for A_i and B_i, and every subset of chromosomes broken by a multi-break in $\mathbf{T}_i$ is also broken by the same multi-break in $\mathbf{T}$, which ensures that $c^*(\mathbf{T}) \geq c^*(\mathbf{T}_i)$. Thus we obtain an inequality $c_s(A, B) = c^*(\mathbf{T}) \geq \max_{i \leq m} c^*(\mathbf{T}_i) \geq \max_{i \leq m} c(A_i, B_i)$, which allows us to conclude that $c(A, B) = \max_{i \leq m} c(A_i, B_i)$. ◀

A.3 Lemma 2

▶ **Lemma.** *For genomes A and B, the function $\mathbf{T} : \mathcal{P}(A, B) \to \mathcal{I}(A, B)$ between the set $\mathcal{P}(A, B)$ of the ordered partitions of the cycles of the breakpoint graph $G(A, B)$ and the set $\mathcal{I}(A, B)$ of the ISA multi-break scenarios for A and B is a bijection.*

Proof. Let $\mathbf{P} = (P_1, \ldots, P_l)$ be an ordered partition of the connected components of the breakpoint graph $G(A, B)$, and let α_i and β_i be respectively black and gray edges in P_i for $i \in \{1, \ldots, l\}$. We start by showing that $\tau(P_i) = (\alpha_i, \beta_i)$ is a multi-break. By construction of the breakpoint graph we have that $\alpha_i \cap \beta_i \subseteq E(A) \cap E(B) = \emptyset$, where $E(A)$ and $E(B)$ are respectively the adjacencies of A that are absent from B and the adjacencies of B that are absent from A. Every vertex in P_i is incident to one black and one gray edge. This ensures that for $\alpha_i = \big\{\{u_1, u_2\}, \ldots, \{u_{2k-1}, u_{2k}\}\big\}$ there exists a permutation σ of a set $\{1, \ldots, 2k\}$ such that $\beta_i = \big\{\{u_{\sigma(1)}, u_{\sigma(2)}\}, \ldots, \{u_{\sigma(2k-1)}, u_{\sigma(2k)}\}\big\}$, which means that $\tau(P_i) = (\alpha_i, \beta_i)$ is a multi-break.

We proceed by showing that $\mathbf{T}(\mathbf{P}) = ((\alpha_1, \beta_1), \ldots, (\alpha_l, \beta_l))$ is a ISA multi-break scenario for A and B. The set $\mathbf{P}$ is an ordered partition of the connected components of $G(A, B)$, thus $\{\alpha_1, \ldots, \alpha_l\}$ and $\{\beta_1, \ldots, \beta_l\}$ respectively partitions $E(A)$ and $E(B)$. This already ensures that $\mathbf{T}(\mathbf{P}) = ((\alpha_1, \beta_1), \ldots, (\alpha_l, \beta_l))$ is a multi-break scenario for A and B. In order to establish that $\mathbf{T}(\mathbf{P})$ is a ISA multi-break scenario for A and B we have to show that an adjacency joined by a multi-break in $\mathbf{T}(\mathbf{P})$ is not broken by a subsequent multi-break in $\mathbf{T}(\mathbf{P})$; i.e., that for $1 \leq i < j \leq l$ we have $\beta_i \cap \alpha_j = \emptyset$. Let $i < j$ be two elements from $\{1, \ldots, l\}$, and let $e = \{u, v\}$ be an adjacency in α_j. Due to (α_j, β_j) being a multi-break, there exists an adjacency $f \neq e$ in β_j that includes u. By construction, we have that $\beta_j \subseteq E(B)$, which means that f is an adjacency of B and, by definition of a genome, f is the single adjacency of B that includes u. As $\{\beta_1, \ldots, \beta_l\}$ partitions $E(B)$ and $i \neq j$, we conclude that β_i does not contain an adjacency that includes u, thus β_i does not include $e = \{u, v\}$. This way we obtain that $\beta_i \cap \alpha_j = \emptyset$, and conclude that $\mathbf{T}(\mathbf{P})$ is a ISA multi-break scenario for A and B.

The function $P \mapsto \tau(P)$ is an injection between the subsets of the connected components of the breakpoint graph and multi-breaks. Let $\mathbf{P} = (P_1, \ldots, P_l)$ and $\mathbf{P'} = (P'_1, \ldots, P'_m)$ be two ordered partitions of the connected components of the breakpoint graph with $\mathbf{T}(\mathbf{P}) = (\tau(P_1), \ldots, \tau(P_l)) = (\tau(P'_1), \ldots, \tau(P'_m)) = \mathbf{T}(\mathbf{P'})$. From the equality $\mathbf{T}(\mathbf{P}) = \mathbf{T}(\mathbf{P'})$ and the injectivity of $P \mapsto \tau(P)$ we obtain that $l = m$ and that $P_i = P'_i$ for $i \in \{1, \ldots, l\}$. This way we obtain that $\mathbf{P} = \mathbf{P'}$, and conclude that $\mathbf{P} \mapsto \mathbf{T}(\mathbf{P})$ is an injection between the ordered partitions of the connected components of the breakpoint graph $G(A, B)$ and the ISA multi-break scenarios for A and B.

It remains to show that $\mathbf{P} \mapsto \mathbf{T}(\mathbf{P})$ is a surjection between the ordered partitions of the connected components of the breakpoint graph $G(A, B)$ and the ISA multi-break scenarios for A and B. Let $\mathbf{T} = ((\alpha_1, \beta_1), \ldots, (\alpha_l, \beta_l))$ be a ISA multi-break scenario for A and B. We start by showing that $\cup_{i=1}^{l} \alpha_i \subseteq E(A)$ and $\cup_{i=1}^{l} \beta_i \subseteq E(B)$. Let i be an element in $\{1, \ldots, l\}$, and let $e = \{u, v\}$ be an adjacency in α_i. Due to $\mathbf{T}$ being a ISA multi-break scenario for A and B, the adjacency e is not joined by any of the multi-breaks preceding (α_i, β_i) in $\mathbf{T}$, which ensures that e is an adjacency of A. In what follows we show that e is not an adjacency of B, and conclude that $e \in E(A)$. Due to (α_i, β_i) being a multi-break, there exists an adjacency $f \neq e$ in β_i that includes u. Due to $\mathbf{T}$ being a ISA multi-break scenario for A and B, the adjacency f is an adjacency of B, as it is not broken by any of the multi-breaks that follow (α_i, β_i) in $\mathbf{T}$. By definition of a genome, f is the single adjacency of B that includes u, which means that e is not an adjacency of B. This way we obtain that $e \in E(A)$, and conclude that $\cup_{i=1}^{l} \alpha_i \subseteq E(A)$. Now, let $e = \{u, v\}$ be an adjacency in β_i. Due to (α_i, β_i) being a multi-break, there exists an adjacency $f \neq e$ in α_i that includes u. We have already shown that $f \in E(A)$, and, as f is the single adjacency of A that includes u, we obtain that $e \in E(B)$, and conclude that $\cup_{i=1}^{l} \beta_i \subseteq E(B)$.

We proceed by showing that $\{\alpha_1, \ldots, \alpha_l\}$ and $\{\beta_1, \ldots, \beta_l\}$ partitions $E(A)$ and $E(B)$. Let e and f be respectively adjacencies in $E(A)$ and $E(B)$. By definition we have that $E(A) \cap E(B) = \emptyset$, thus every adjacency in $E(A)$ must be broken by a multi-break in $\mathbf{T}$, and every adjacency in $E(B)$ must be joined by a multi-break in $\mathbf{T}$. This ensures that there exist $i, j \in \{1, \ldots, l\}$ such that $e \in \alpha_i$ and $f \in \beta_i$. By definition of a genome, the adjacencies of A contain a single copy of e and the adjacencies of B contain a single copy of f. What is more, $\{e\} \cap \cup_{i=1}^{l} \beta_i \subseteq E(A) \cap E(B) = \emptyset$, thus e is not joined by any of the multi-breaks in $\mathbf{T}$, which ensures that it is only broken by (α_i, β_i) in $\mathbf{T}$. Similarly, f is not broken by any of the multi-breaks in $\mathbf{T}$, which ensures that it is only joined by (α_j, β_j) in $\mathbf{T}$. We conclude that $\{\alpha_1, \ldots, \alpha_l\}$ and $\{\beta_1, \ldots, \beta_l\}$ respectively partitions $E(A)$ and $E(B)$.

Let P_i be a subgraph of $G(A, B)$ induced by adjacencies α_i and β_i for $i \in \{1, \ldots, l\}$. We will show that P_i is a subset of the connected components of $G(A, B)$. We do this by establishing that for every vertex u in P_i all the edges incident to u in $G(A, B)$ are also present in P_i. A vertex u of P_i is incident to one black and one gray edge in $G(A, B)$. By construction of P_i, u is in some adjacency in $\alpha_i \cup \beta_i$, however due to (α_i, β_i) being a multi-break, we obtain that u is both in some adjacency in α_i and in some adjacency in β_i. This ensures that P_i contains one black and one gray edge incident to u, and thus that all the edges incident to u in $G(A, B)$ are also present in P_i. This way we conclude that P_i is a subset of the connected components of $G(A, B)$.

Finally, due to $\{\alpha_1, \ldots, \alpha_l\}$ and $\{\beta_1, \ldots, \beta_l\}$ respectively partitioning $E(A)$ and $E(B)$, we obtain that $\mathbf{P} = (P_1, \ldots, P_l)$ is an ordered partition of the connected components of the breakpoint graph $G(A, B)$. By construction of $\mathbf{P}$, we have that $\mathbf{T}(\mathbf{P}) = \mathbf{T} = ((\alpha_1, \beta_1), \ldots, (\alpha_l, \beta_l))$, which ensures that $\mathbf{P} \mapsto \mathbf{T}(\mathbf{P})$ is a surjection between the ordered partitions of the connected components of the breakpoint graph $G(A, B)$ and the ISA multi-break scenarios for A and B. ◀

A.4 Theorem 2

▶ **Theorem.** *Let $\{H_1, \ldots, H_m\}$ be the connected components of the chromosomes graph $C(A, \mathcal{B})$. If A_i is a genome consisting of the chromosomes of A included in H_i, and $\mathcal{B}_i$ are the adjacencies in $\mathcal{B}$ included in H_i, then the chromosome number $c(A, \mathcal{B}, \beta)$ is equal to $\Sigma_{i=1}^{m} c(A_i, \mathcal{B}_i, \mathcal{B}_i \cap \beta)$.*

Proof. Let A' be a ISA intermediate genome for A and $\mathcal{B}$ in which β affects $c(A, \mathcal{B}, \beta)$ chromosomes. A genome A'_i that consists of the chromosomes of A' included in a connected component H_i of the chromosome graph $C(A, \mathcal{B})$ is a ISA intermediate genome for A_i and $\mathcal{B}_i$. What is more, $c(A, \mathcal{B}, \beta) = \Sigma_{i=1}^{m} c_i$ where c_i is the number of chromosomes affected in A'_i by β. By construction of the chromosome graph, if an adjacency $e \in \beta$ affects a chromosome in A_i, then $e \in \mathcal{B}_i \cap \beta$, which ensures that the numbers of chromosomes affected by $\mathcal{B}_i \cap \beta$ in A'_i is equal to c_i. This allows us to conclude that $c(A, \mathcal{B}, \beta) \geq \Sigma_{i=1}^{m} c(A_i, \mathcal{B}_i, \mathcal{B}_i \cap \beta)$.

Let A'_i be a ISA intermediate genome for A_i and $\mathcal{B}_i$ in which $\mathcal{B}_i \cap \beta$ affects $c(A_i, \mathcal{B}_i, \mathcal{B}_i \cap \beta)$ chromosomes for $i \leq m$. A genome A' that consists of the union of the chromosomes of the genomes $\{A'_1, \ldots, A'_m\}$ is a ISA intermediate genome for A and $\mathcal{B}$. Whats is more, β affects $\Sigma_{i=1}^{m} c(A_i, \mathcal{B}_i, \mathcal{B}_i \cap \beta)$ chromosomes in A', thus we have that $c(A, \mathcal{B}, \beta) \leq \Sigma_{i=1}^{m} c(A_i, \mathcal{B}_i, \mathcal{B}_i \cap \beta)$, and finally conclude that $c(A, \mathcal{B}, \beta) = \Sigma_{i=1}^{m} c(A_i, \mathcal{B}_i, \mathcal{B}_i \cap \beta)$. ◀

Locality-Sensitive Bucketing Functions for the Edit Distance

Ke Chen
Department of Computer Science and Engineering, School of Electronic Engineering and Computer Science, The Pennsylvania State University, University Park, PA, United States

Mingfu Shao
Department of Computer Science and Engineering, School of Electronic Engineering and Computer Science, The Pennsylvania State University, University Park, PA, United States
Huck Institutes of the Life Sciences, The Pennsylvania State University, University Park, PA, United States

Abstract

Many bioinformatics applications involve bucketing a set of sequences where each sequence is allowed to be assigned into multiple buckets. To achieve both high sensitivity and precision, bucketing methods are desired to assign similar sequences into the same bucket while assigning dissimilar sequences into distinct buckets. Existing k-mer-based bucketing methods have been efficient in processing sequencing data with low error rate, but encounter much reduced sensitivity on data with high error rate. Locality-sensitive hashing (LSH) schemes are able to mitigate this issue through tolerating the edits in similar sequences, but state-of-the-art methods still have large gaps. Here we generalize the LSH function by allowing it to hash one sequence into multiple buckets. Formally, a bucketing function, which maps a sequence (of fixed length) into a subset of buckets, is defined to be (d_1, d_2)-sensitive if any two sequences within an edit distance of d_1 are mapped into at least one shared bucket, and any two sequences with distance at least d_2 are mapped into disjoint subsets of buckets. We construct locality-sensitive bucketing (LSB) functions with a variety of values of (d_1, d_2) and analyze their efficiency with respect to the total number of buckets needed as well as the number of buckets that a specific sequence is mapped to. We also prove lower bounds of these two parameters in different settings and show that some of our constructed LSB functions are optimal. These results provide theoretical foundations for their practical use in analyzing sequences with high error rate while also providing insights for the hardness of designing ungapped LSH functions.

2012 ACM Subject Classification Applied computing → Bioinformatics; Applied computing → Computational biology

Keywords and phrases Locality-sensitive hashing, locality-sensitive bucketing, long reads, embedding

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.22

Supplementary Material *Software (Source Code)*: `https://github.com/Shao-Group/lsbucketing`

Funding This work is supported by the US National Science Foundation (DBI-2019797) and the US National Institutes of Health (R01HG011065).

1 Introduction

Comparing a set of given sequences is a common task involved in many bioinformatics applications, such as homology detection [6], overlap detection and the construction of overlap graphs [10, 4, 24], phylogenetic tree reconstruction, and isoform detection from circular consensus sequence (CCS) reads [22], to name a few. The naive all-vs-all comparison gives the most comprehensive information but does not scale well. An efficient and widely-used approach that avoids unnecessary comparisons is *bucketing*: a linear scan is employed to assign each sequence into one or multiple buckets, followed by pairwise comparisons within each bucket. The procedure of assigning sequences into buckets, which we refer to as a

bucketing function, is desired to be both "sensitive", i.e., two similar sequences ideally appear in at least one shared bucket so that they can be compared, and "specific", i.e., two dissimilar sequences ideally appear in disjoint buckets so that they can be exempt from comparison. The criteria of similar/dissimilar sequences are application-dependent; in this work we study bucketing functions for the edit distance (Levenshtein distance).

A simple yet popular bucketing function is to put a sequence into buckets labeled with its own k-mers. The popular seed-and-extend strategy [1, 2] implicitly uses this approach. Various sketching methods such as minimizer [19, 23, 20, 13] and universal hitting set [16, 7] reduce the number of buckets a sequence is assigned to by only considering a subset of representative k-mers. These bucketing methods based on exact k-mer matching enjoyed tremendous success in analyzing next-generation sequencing (NGS) data, but are challenged by the third-generation long-reads sequencing data represented by PacBio [18] and Oxford Nanopore [8] technologies; due to the high error rate, sequences that should be assigned to the same buckets hardly share any identical k-mers (for a reasonably large k such as $k = 21$ with 15% error rate), and therefore results in poor sensitivity.

To address this issue, it is required to be able to recognize similar but not necessarily identical sequences. A general solution is locality-sensitive hashing (LSH) [14, 15] where with high probability, similar sequences are sent into the same bucket (i.e., there is a hash collision), and with high probability dissimilar sequences are sent into different buckets. However, designing locality-sensitive hashing functions for the edit distance is hard; the state-of-the-art method Order Min Hash (OMH) is proved to be a gapped LSH but admits a large gap [14]. Another related approach is embedding the metric space induced by the edit distance into more well-studied normed spaces [3, 17, 24]. However, such an embedding is also hard; for example, it is known that the embedding into L_1 cannot be distortion-free [9]. In addition, there are seeding/sketching methods such as spaced k-mer [5, 11], indel seeds [12], and the more recent strobemer [21] that allow gaps in the extracted seeds to accommodate some edits, but an edit that happens within the chosen seed can still cause mismatches.

It is worth noting that locality-sensitive hashing functions, when interpreted as bucketing functions, assign a sequence into exactly one bucket: buckets are labeled with hash values, and a sequence is put into the single bucket where it is hashed to. In this work, we propose the concept of *locality-sensitive bucketing* (LSB) functions as a generalization of LSH functions by allowing it to assign a sequence into multiple buckets. Formally, a bucketing function, which maps a sequence (of fixed length) into one or more buckets, is defined to be (d_1, d_2)-sensitive if any two sequences within an edit distance of d_1 are mapped into at least one shared bucket, and any two sequences with an edit distance at least d_2 are mapped into disjoint subsets of buckets. While a stochastic definition by introducing a distribution on a family of bucketing functions can be made in a similar way as the definition of LSH functions, here we focus on this basic, deterministic definition. We design several LSB functions for a variety of values of (d_1, d_2) including both ungapped ($d_2 = d_1 + 1$) and gapped ($d_2 > d_1 + 1$) ones. This demonstrates that allowing one sequence to appear in multiple buckets makes the locality-sensitive properties easier to satisfy. Moreover, our lower bound proof shows that any $(1, 2)$-sensitive bucketing function must put each sequence (of length n) into at least n buckets (see Lemma 2), suggesting that certain ungapped locality-sensitive hashing functions, where each sequence is sent to a single bucket, may not exist.

The rest of this paper is organized as follows. In Section 2, we give the precise definition of LSB functions and propose criteria to measure them. In Sections 3 and 4, we design LSB functions using two different approaches, the results are summarized in Section 5. We show experimental studies in Section 6, with a focus on demonstrating the performance of gapped LSB functions. Future directions are discussed in Section 7.

2 Basics of locality-sensitive bucketing (LSB) functions

Given an alphabet Σ with $|\Sigma| > 1$ and a natural number n, let $\mathcal{S}_n = (\Sigma^n, \text{edit})$ be the metric space of all length-n sequences equipped with the Levenshtein (edit) distance. Given a set B of buckets, a bucketing function f maps $\mathcal{S}_n$ to $\mathcal{P}(B)$, the power set of B. This can be viewed as assigning a sequence $\boldsymbol{s}$ of length n to a subset of buckets $f(\boldsymbol{s}) \subset B$. Let $d_1 < d_2$ be two non-negative integers, we say a bucketing function f is (d_1, d_2)-*sensitive* if

$$\text{edit}(\boldsymbol{s}, \boldsymbol{t}) \leq d_1 \implies f(\boldsymbol{s}) \cap f(\boldsymbol{t}) \neq \varnothing, \tag{1}$$

$$\text{edit}(\boldsymbol{s}, \boldsymbol{t}) \geq d_2 \implies f(\boldsymbol{s}) \cap f(\boldsymbol{t}) = \varnothing. \tag{2}$$

We refer to the above two conditions as LSB-properties (1) and (2) respectively. Intuitively, the LSB-properties state that, if two length-n sequences are within an edit distance of d_1, then the bucketing function f guarantees assigning them to at least one same bucket, and if two length-n sequences have an edit distance at least d_2, then the bucketing function f guarantees not assigning them to any shared bucket. In other words, (d_1, d_2)-sensitive bucketing functions perfectly distinguish length-n sequences within distance d_1 from those with distances at least d_2. It is easy to show that if $f : \mathcal{S}_n \to \mathcal{P}(B)$ is a (d_1, d_2)-sensitive bucketing function, then $f(\boldsymbol{s}) \neq \varnothing$ for all $\boldsymbol{s} \in \mathcal{S}_n$. In fact, since $\text{edit}(\boldsymbol{s}, \boldsymbol{s}) = 0 \leq d_1$, the LSB-property (1) implies that $f(\boldsymbol{s}) = f(\boldsymbol{s}) \cap f(\boldsymbol{s}) \neq \varnothing$. If $d_1 = d_2 - 1$ then we say the bucketing function is ungapped; otherwise it is called gapped.

We note that the above definition of LSB functions generalize the (deterministic) LSH functions: if we require that $|f(\boldsymbol{s})| = 1$ for every sequence $\boldsymbol{s} \in \mathcal{S}_n$, i.e., f maps a sequence to a single bucket, then $f(\boldsymbol{s}) \cap f(\boldsymbol{t}) \neq \varnothing$ implies $f(\boldsymbol{s}) = f(\boldsymbol{t})$ and $f(\boldsymbol{s}) \cap f(\boldsymbol{t}) = \varnothing$ implies $f(\boldsymbol{s}) \neq f(\boldsymbol{t})$.

Two related parameters can be used to measure an LSB function: $|B|$, the total number of buckets, and $|f(\boldsymbol{s})|$, the number of different buckets that contain a specific sequence $\boldsymbol{s}$. From a practical perspective, it is desirable to keep both parameters small. We therefore aim to design LSB functions that minimize $|B|$ and $|f(\boldsymbol{s})|$. Specifically, in the following sections, we will construct (d_1, d_2)-sensitive bucketing functions with a variety of values of (d_1, d_2), and analyze their corresponding $|B|$ and $|f(\boldsymbol{s})|$; we will also prove lower bounds of $|B|$ and $|f(\boldsymbol{s})|$ in different settings and show that some of our constructed LSB functions are optimal, in terms of minimizing these two parameters.

The bounds of $|B|$ and $|f(\boldsymbol{s})|$ are closely related to the structure of the metric space $\mathcal{S}_n$. For a sequence $\boldsymbol{s} \in \mathcal{S}_n$, its d-neighborhood, denoted by $N_n^d(\boldsymbol{s})$, is the subspace of all sequences of length n with edit distance at most d from $\boldsymbol{s}$; formally $N_n^d(\boldsymbol{s}) = \{\boldsymbol{t} \in \mathcal{S}_n \mid \text{edit}(\boldsymbol{s}, \boldsymbol{t}) \leq d\}$. The following simple fact demonstrates the connection between the bound of $|f(\boldsymbol{s})|$ and the structure of $\mathcal{S}_n$, which will be used later.

▶ **Lemma 1.** *Let $\boldsymbol{s}$ be a sequence of length n. If $N_n^{d_1}(\boldsymbol{s})$ contains a subset X with $|X| = x$ such that every two sequences in X have an edit distance at least d_2, then for any (d_1, d_2)-sensitive bucketing function f we must have $|f(\boldsymbol{s})| \geq x$.*

Proof. Let f be an arbitrary (d_1, d_2)-sensitive bucketing function. By the LSB-property (2), these x sequences must be assigned to distinct buckets by f. On the other hand, since they are all in $N_n^{d_1}(\boldsymbol{s})$, the LSB-property (1) requires that $f(\boldsymbol{s})$ overlaps with $f(\boldsymbol{t})$ for each sequence $\boldsymbol{t} \in X$. Combined, we have $|f(\boldsymbol{s})| \geq x$. ◀

3 An optimal $(1, 2)$-sensitive bucketing function

In the most general setting of LSB functions, the labels of buckets in B are just symbols that are irrelevant to the construction of the bucketing function. Hence we can let $B = \{1, \ldots, |B|\}$. The remaining of this section studies $(1, 2)$-sensitive bucketing functions in this general case. We first prove lower bounds of $|B|$ and $|f(\boldsymbol{s})|$ in this setting; we then give an algorithm to construct an optimal $(1, 2)$-sensitive bucketing function f that matches these bounds.

▶ **Lemma 2.** *If $f : \mathcal{S}_n \to \mathcal{P}(B)$ is $(1, 2)$-sensitive, then for each $\boldsymbol{s} \in \mathcal{S}_n$, $|f(\boldsymbol{s})| \geq n$.*

Proof. According to Lemma 1 with $d_1 = 1$ and $d_2 = 2$, we only need to show that $N_n^1(\boldsymbol{s})$ contains n different sequences with pairwise edit distances at least 2. For $i = 1, \ldots, n$, let $\boldsymbol{t}^i$ be a sequence obtained from $\boldsymbol{s}$ by a single substitution at position i. If $i \neq j$, then $\boldsymbol{t}^i$ differs from $\boldsymbol{t}^j$ at two positions, namely i and j. Then we must have edit $\left(\boldsymbol{t}^i, \boldsymbol{t}^j\right) \geq 2$ as $\boldsymbol{t}^i$ cannot be transformed into $\boldsymbol{t}^j$ with a single substitution or a single insertion or deletion. Hence, $\left\{\boldsymbol{t}^1, \ldots, \boldsymbol{t}^n\right\}$ forms the required set. ◀

▶ **Lemma 3.** *If $f : \mathcal{S}_n \to \mathcal{P}(B)$ is $(1, 2)$-sensitive, then $|B| \geq n|\Sigma|^{n-1}$.*

Proof. Consider the collection of pairs $H = \{(\boldsymbol{s}, b) \mid \boldsymbol{s} \in \mathcal{S}_n \text{ and } b \in f(\boldsymbol{s})\}$. We bound the size of H from above and below. For an arbitrary sequence $\boldsymbol{s}$, let $b \in f(\boldsymbol{s})$ be a bucket that contains $\boldsymbol{s}$. According to the LSB-property (2), any other sequence in b has edit distance 1 from $\boldsymbol{s}$, i.e., a substitution. Suppose that the bucket b contains two sequences $\boldsymbol{u}$ and $\boldsymbol{v}$ that are obtained from $\boldsymbol{s}$ by a single substitution at different positions. Then edit $(\boldsymbol{u}, \boldsymbol{v}) = 2$ and $f(\boldsymbol{u}) \cap f(\boldsymbol{v}) \neq \varnothing$, which contradicts the LSB-property (2). Therefore, all the sequences in b can only differ from $\boldsymbol{s}$ at some fixed position i. There are $|\Sigma|$ such sequences (including $\boldsymbol{s}$ itself). So each bucket $b \in B$ can appear in at most $|\Sigma|$ pairs in H. Thus $|H| \leq |\Sigma| \cdot |B|$.

On the other hand, for a length-n sequence $\boldsymbol{s}$, its 1-neighborhood $N_n^1(\boldsymbol{s})$ contains $n(|\Sigma|-1)$ other length-n sequences, corresponding to the $|\Sigma| - 1$ possible substitutions at each of the n positions. The LSB-property (1) requires that $\boldsymbol{s}$ shares at least one bucket with each of them. As argued above, each bucket $b \in f(\boldsymbol{s})$ can contain at most $|\Sigma| - 1$ sequences other than $\boldsymbol{s}$. Therefore, $\boldsymbol{s}$ needs to appear in at least $n(|\Sigma| - 1)/(|\Sigma| - 1) = n$ different buckets, and hence at least n pairs in H. So $|H| \geq n|\mathcal{S}_n| = n|\Sigma|^n$. Together, we have $|\Sigma| \cdot |B| \geq n|\Sigma|^n$, or $|B| \geq n|\Sigma|^{n-1}$. ◀

We now construct a bucketing function $f : \mathcal{S}_n \to \mathcal{P}(B)$ that is $(1, 2)$-sensitive using the algorithm given below. It has exponential running time with respect to n but primarily serves as a constructive proof that $(1, 2)$-sensitive bucketing functions exist. Assign to the alphabet Σ an arbitrary order $\sigma : \{1, \ldots, |\Sigma|\} \to \Sigma$. The following algorithm defines the function f:

```
foreach s ∈ S_n do f(s) = ∅
m ← 1   // index of the smallest unused bucket
foreach s = s_1 s_2 ··· s_n ∈ S_n do   // in an arbitrary order
    for i = 1 to n do
        if s_i == σ(1) then   // s_i is the smallest character in Σ
            for j = 1 to |Σ| do
                t ← s_1 ··· s_{i−1} σ(j) s_{i+1} ··· s_n
                f(t) ← f(t) ∪ {m}   // add t to bucket m
            m ← m + 1
```

A toy example of the bucketing function f with $n = 2$ and $\Sigma = \{\sigma(1) = \mathrm{A}, \sigma(2) = \mathrm{C}, \sigma(3) = \mathrm{G}, \sigma(4) = \mathrm{T}\}$ constructed using the above algorithm (where the sequences are processed in the lexicographical order induced by σ) is given below, followed by the contained sequences in the resulting buckets.

$$\begin{array}{llll} f(\mathrm{AA}) = \{1,2\}, & f(\mathrm{AC}) = \{2,3\}, & f(\mathrm{AG}) = \{2,4\}, & f(\mathrm{AT}) = \{2,5\}, \\ f(\mathrm{CA}) = \{1,6\}, & f(\mathrm{CC}) = \{3,6\}, & f(\mathrm{CG}) = \{4,6\}, & f(\mathrm{CT}) = \{5,6\}, \\ f(\mathrm{GA}) = \{1,7\}, & f(\mathrm{GC}) = \{3,7\}, & f(\mathrm{GG}) = \{4,7\}, & f(\mathrm{GT}) = \{5,7\}, \\ f(\mathrm{TA}) = \{1,8\}, & f(\mathrm{TC}) = \{3,8\}, & f(\mathrm{TG}) = \{4,8\}, & f(\mathrm{TT}) = \{5,8\}. \end{array}$$

bucket #	sequences	bucket #	sequences
1	AA, CA, GA, TA	2	AA, AC, AG, AT
3	AC, CC, GC, TC	4	AG, CG, GG, TG
5	AT, CT, GT, TT	6	CA, CC, CG, CT
7	GA, GC, GG, GT	8	TA, TC, TG, TT

▶ **Lemma 4.** *The constructed bucketing function $f : \mathcal{S}_n \to \mathcal{P}(B)$ satisfies: (i) each bucket contains $|\Sigma|$ sequences, (ii) $|f(\boldsymbol{s})| = n$ for each $\boldsymbol{s} \in \mathcal{S}_n$, and (iii) $|B| = n|\Sigma|^{n-1}$.*

Proof. Claim (i) follows directly from the construction (the most inner for-loop). In the algorithm, each sequence $\boldsymbol{s} \in \mathcal{S}_n$ is added to n different buckets, one for each position. Specifically, let $\boldsymbol{s} = s_1 s_2 \cdots s_n$, then $\boldsymbol{s}$ is added to a new bucket when we process the sequence $\boldsymbol{s}^i = s_1 s_2 \cdots s_{i-1} \sigma(1) s_{i+1} \cdots s_n$, $1 \le i \le n$. Hence, $|f(\boldsymbol{s})| = n$. To calculate $|B|$, observe that a new bucket is used whenever we encounter the smallest character $\sigma(1)$ in some sequence $\boldsymbol{s}$. So $|B|$ is the same as the number of occurrences of $\sigma(1)$ among all sequences in $\mathcal{S}_n$. The total number of characters in $\mathcal{S}_n$ is $n|\Sigma|^n$. By symmetry, $\sigma(1)$ appears $n|\Sigma|^{n-1}$ times. ◀

▶ **Lemma 5.** *The constructed bucketing function f is $(1,2)$-sensitive.*

Proof. We show that for $\boldsymbol{s}, \boldsymbol{t} \in \mathcal{S}_n$, $\mathrm{edit}(\boldsymbol{s}, \boldsymbol{t}) \le 1$ if and only if $f(\boldsymbol{s}) \cap f(\boldsymbol{t}) \neq \varnothing$. For the forward direction, $\mathrm{edit}(\boldsymbol{s}, \boldsymbol{t}) \le 1$ implies that $\boldsymbol{s}$ and $\boldsymbol{t}$ can differ by at most one substitution at some position i. Let $\boldsymbol{r}$ be the sequence that is identical to $\boldsymbol{s}$ except at the i-th position where it is substituted by $\sigma(1)$ (it is possible that $\boldsymbol{r} = \boldsymbol{s}$). According to the algorithm, when processing $\boldsymbol{r}$, both $\boldsymbol{s}$ and $\boldsymbol{t}$ are added to a same bucket m. Therefore, $m \in f(\boldsymbol{s}) \cap f(\boldsymbol{t})$.

For the backward direction, let m be an integer from $f(\boldsymbol{s}) \cap f(\boldsymbol{t})$. By construction, all the $|\Sigma|$ sequences in the bucket m differ by a single substitution. Hence, $\mathrm{edit}(\boldsymbol{s}, \boldsymbol{t}) \le 1$. ◀

Combining Lemmas 2–5, we have shown that the above $(1,2)$-sensitive bucketing function is optimal in the sense of minimizing $|B|$ and $|f(\boldsymbol{s})|$. This is summarized below.

▶ **Theorem 1.** *Let $B = \{1, \ldots, n|\Sigma|^{n-1}\}$, there is a $(1,2)$-sensitive bucketing function $f : \mathcal{S}_n \to \mathcal{P}(B)$ with $|f(\boldsymbol{s})| = n$ for each $\boldsymbol{s} \in \mathcal{S}_n$. No $(1,2)$-sensitive bucketing function exists if $|B|$ is smaller or $|f(\boldsymbol{s})| < n$ for some sequence $\boldsymbol{s} \in \mathcal{S}_n$.*

4 Mapping to sequences of length n

We continue to explore LSB functions with different values of d_1 and d_2. Here we focus on a special case where $B \subset \mathcal{S}_n$, namely, each bucket in B is labeled by a length-n sequence. The idea of designing such LSB functions is to map a sequence $\boldsymbol{s}$ to its neighboring sequences that are in B. Formally, given a subset $B \subset \mathcal{S}_n$ and an integer $r \ge 1$, we define the bucketing function $f_B^r : \mathcal{S}_n \to \mathcal{P}(B)$ by

$$f_B^r(\boldsymbol{s}) = N_n^r(\boldsymbol{s}) \cap B = \{\boldsymbol{v} \in B \,|\, \mathrm{edit}(\boldsymbol{s}, \boldsymbol{v}) \le r\} \text{ for each } \boldsymbol{s} \in \mathcal{S}_n.$$

We now derive the conditions for f_B^r to be an LSB function. For any sequence $\boldsymbol{s}$, all the buckets in $f_B^r(\boldsymbol{s})$ are labeled by its neighboring sequences within radius r. Therefore, if two sequences $\boldsymbol{s}$ and $\boldsymbol{t}$ share a bucket labeled by $\boldsymbol{v}$, then $\text{edit}(\boldsymbol{s},\boldsymbol{v}) \le r$ and $\text{edit}(\boldsymbol{t},\boldsymbol{v}) \le r$. Recall that $\mathcal{S}_n$ is a metric space, in particular, the triangle inequality holds. So $\text{edit}(\boldsymbol{s},\boldsymbol{t}) \le \text{edit}(\boldsymbol{s},\boldsymbol{v}) + \text{edit}(\boldsymbol{t},\boldsymbol{v}) \le 2r$. In other words, if $\boldsymbol{s}$ and $\boldsymbol{t}$ are $2r+1$ edits apart, then they will be mapped to disjoint buckets. Formally, if $\text{edit}(\boldsymbol{s},\boldsymbol{t}) \ge 2r+1$, then $f_B^r(\boldsymbol{s}) \cap f_B^r(\boldsymbol{t}) = \varnothing$. This implies that f_B^r satisfies the LSB-property (2) with $d_2 = 2r+1$. We note that this statement holds regardless of the choice of B.

Hence, to make f_B^r a $(d_1, 2r+1)$-sensitive bucketing function for some integer d_1, we only need to determine a subset B so that f_B^r satisfies the LSB-property (1). Specifically, B should be picked such that for any two length-n sequences $\boldsymbol{s}$ and $\boldsymbol{t}$ within an edit distance of d_1, we always have

$$f_B^r(\boldsymbol{s}) \cap f_B^r(\boldsymbol{t}) = (N_n^r(\boldsymbol{s}) \cap B) \cap (N_n^r(\boldsymbol{t}) \cap B) = N_n^r(\boldsymbol{s}) \cap N_n^r(\boldsymbol{t}) \cap B \neq \varnothing.$$

For the sake of simplicity, we say a set of buckets $B \subset \mathcal{S}_n$ is (d_1, r)-*guaranteed* if and only if $N_n^r(\boldsymbol{s}) \cap N_n^r(\boldsymbol{t}) \cap B \neq \varnothing$ for every pair of sequences $\boldsymbol{s}$ and $\boldsymbol{t}$ with $\text{edit}(\boldsymbol{s},\boldsymbol{t}) \le d_1$. Equivalently, following the above arguments, B is (d_1, r)-guaranteed if and only if the corresponding bucketing function f_B^r is $(d_1, 2r+1)$-sensitive. Note that the (d_1, r)-guaranteed set is not a new concept, but rather an abbreviation to avoid repeating the long phrase "a set whose corresponding bucketing function is $(d_1, 2r+1)$-sensitive". In the following sections, we show several (d_1, r)-guaranteed subsets $B \subset \mathcal{S}_n$ for different values of d_1.

4.1 $(2r, r)$-guaranteed and $(2r-1, r)$-guaranteed subsets

We first consider an extreme case where $B = \mathcal{S}_n$.

▶ **Lemma 6.** *Let $B = \mathcal{S}_n$. Then B is $(2r, r)$-guaranteed if r is even, and B is $(2r-1, r)$-guaranteed if r is odd.*

Proof. First consider the case that r is even. Let $\boldsymbol{s}$ and $\boldsymbol{t}$ be two length-n sequences with $\text{edit}(\boldsymbol{s},\boldsymbol{t}) \le 2r$. Then there are $2r$ edits that transforms $\boldsymbol{s}$ to $\boldsymbol{t}$. (If $\text{edit}(\boldsymbol{s},\boldsymbol{t}) < 2r$, we can add in trivial edits that substitute a character with itself.) Because $\boldsymbol{s}$ and $\boldsymbol{t}$ have the same length, these $2r$ edits must contain the same number of insertions and deletions. Reorder the edits so that each insertion is followed immediately by a deletion (i.e., a pair of indels) and all the indels come before substitutions. Because r is even, in this new order, the first r edits contain an equal number of insertions and deletions. Namely, applying the first r edits on $\boldsymbol{s}$ produces a length-n sequence $\boldsymbol{v}$. Clearly, $\text{edit}(\boldsymbol{s},\boldsymbol{v}) \le r$ and $\text{edit}(\boldsymbol{t},\boldsymbol{v}) \le r$, i.e., $\boldsymbol{v} \in N_n^r(\boldsymbol{s}) \cap N_n^r(\boldsymbol{t}) = N_n^r(\boldsymbol{s}) \cap N_n^r(\boldsymbol{t}) \cap B$.

For the case that r is odd. Let $\boldsymbol{s}$ and $\boldsymbol{t}$ be two length-n sequences with $\text{edit}(\boldsymbol{s},\boldsymbol{t}) \le 2r-1$. By the same argument as above, $\boldsymbol{s}$ can be transformed to $\boldsymbol{t}$ by $2r-1$ edits and we can assume that all the indels appear in pairs and they come before all the substitutions. Because r is odd, $r-1$ is even. So applying the first $r-1$ edits on $\boldsymbol{s}$ produces a length-n sequence $\boldsymbol{v}$ such that $\text{edit}(\boldsymbol{s},\boldsymbol{v}) \le r-1 < r$ and $\text{edit}(\boldsymbol{t},\boldsymbol{v}) \le 2r-1-(r-1) = r$. Therefore, $\boldsymbol{v} \in N_n^r(\boldsymbol{s}) \cap N_n^r(\boldsymbol{t}) = N_n^r(\boldsymbol{s}) \cap N_n^r(\boldsymbol{t}) \cap B$. ◀

By definition, setting $B = \mathcal{S}_n$ makes f_B^r $(2r, 2r+1)$-sensitive if r is even and $(2r-1, 2r+1)$-sensitive if r is odd. This provides nearly optimal bucketing performance in the sense that there is no gap (when r is even) or the gap is just one (when r is odd). It is evident from the proof that the gap at $2r$ indeed exists when r is odd because if $\boldsymbol{s}$ can only be transformed to $\boldsymbol{t}$ by r pairs of indels, then there is no length-n sequence $\boldsymbol{v}$ with $\text{edit}(\boldsymbol{s},\boldsymbol{v}) = \text{edit}(\boldsymbol{t},\boldsymbol{v}) = r$.

4.2 Properties of (r, r)-guaranteed subsets

In the above section all sequences in $\mathcal{S}_n$ are used as buckets. A natural question is, can we use a proper subset of $\mathcal{S}_n$ to achieve (gapped) LSB functions? This can be viewed as down-sampling $\mathcal{S}_n$ such that if two length-n sequences $\boldsymbol{s}$ and $\boldsymbol{t}$ are similar, then a length-n sequence is always sampled from their common neighborhood $N_n^r(\boldsymbol{s}) \cap N_n^r(\boldsymbol{t})$.

Here we focus on the case that $d_1 = r$, i.e., we aim to construct B that is (r, r)-guaranteed. Recall that this means for any $\boldsymbol{s}, \boldsymbol{t} \in \mathcal{S}_n$ with edit $(\boldsymbol{s}, \boldsymbol{t}) \le r$, we have $N_n^r(\boldsymbol{s}) \cap N_n^r(\boldsymbol{t}) \cap B \neq \varnothing$. In other words, f_B^r is $(r, 2r+1)$-sensitive. To prepare the construction, we first investigate some structural properties of (r, r)-guaranteed subsets. We propose a conjecture that such sets form a hierarchical structure with decreasing r:

▶ **Conjecture 1.** *If $B \subset \mathcal{S}_n$ is (r, r)-guaranteed, then B is also $(r+1, r+1)$-guaranteed.*

We prove a weaker statement:

▶ **Lemma 7.** *If $B \subset \mathcal{S}_n$ is (r, r)-guaranteed, then B is $(r+2, r+2)$-guaranteed.*

Proof. Let $\boldsymbol{s}$ and $\boldsymbol{t}$ be two length-n sequences with edit $(\boldsymbol{s}, \boldsymbol{t}) \le r+2$; we want to show that $N_n^{r+2}(\boldsymbol{s}) \cap N_n^{r+2}(\boldsymbol{t}) \cap B \neq \varnothing$. Consider a list of edits that transforms $\boldsymbol{s}$ to $\boldsymbol{t}$: skipping a pair of indels or two substitutions gives a length-n sequence $\boldsymbol{m}$ such that edit $(\boldsymbol{s}, \boldsymbol{m}) \le r$ and edit $(\boldsymbol{t}, \boldsymbol{m}) = 2$. Because $\boldsymbol{s}$ and $\boldsymbol{m}$ are within a distance of r and B is (r, r)-guaranteed, we have that $N_n^r(\boldsymbol{s}) \cap N_n^r(\boldsymbol{m}) \cap B \neq \varnothing$, i.e., there exists a length-n sequence $\boldsymbol{v} \in B$ such that edit $(\boldsymbol{s}, \boldsymbol{v}) \le r$ and edit $(\boldsymbol{m}, \boldsymbol{v}) \le r$. By triangle inequality, edit $(\boldsymbol{t}, \boldsymbol{v}) \le$ edit $(\boldsymbol{t}, \boldsymbol{m}) +$ edit $(\boldsymbol{m}, \boldsymbol{v}) \le r+2$. Hence, we have $\boldsymbol{v} \in N_n^{r+2}(\boldsymbol{t})$. Clearly, $\boldsymbol{v} \in N_n^r(\boldsymbol{s})$ implies that $\boldsymbol{v} \in N_n^{r+2}(\boldsymbol{s})$. Combined, we have $\boldsymbol{v} \in N_n^{r+2}(\boldsymbol{s}) \cap N_n^{r+2}(\boldsymbol{t}) \cap B$. ◀

The next lemma shows that $(1, 1)$-guaranteed subsets have the strongest condition.

▶ **Lemma 8.** *If $B \subset S_n$ is $(1, 1)$-guaranteed, then B is (r, r)-guaranteed for all $r \ge 1$.*

Proof. According to the previous lemma, we only need to show that B is $(2, 2)$-guaranteed. Given two length-n sequences $\boldsymbol{s}$ and $\boldsymbol{t}$ with edit $(\boldsymbol{s}, \boldsymbol{t}) = 2$, consider a list Q of two edits that transforms $\boldsymbol{s}$ to $\boldsymbol{t}$. There are two possibilities:

- If both edits in Q are substitutions, let i be the position of the first substitution.
- If Q consists of one insertion and one deletion, let i be the position of the character that is going to be deleted from $\boldsymbol{s}$.

In either case, let $\boldsymbol{m}$ be a length-n sequence obtained by replacing the i-th character of $\boldsymbol{s}$ with another character in Σ. Then edit $(\boldsymbol{s}, \boldsymbol{m}) = 1$. Because B is $(1, 1)$-guaranteed, there is a length-n sequence $\boldsymbol{v} \in B$ such that edit $(\boldsymbol{s}, \boldsymbol{v}) \le 1$ and edit $(\boldsymbol{m}, \boldsymbol{v}) \le 1$. Observe that either $\boldsymbol{s} = \boldsymbol{v}$ or $\boldsymbol{v}$ is obtained from $\boldsymbol{s}$ by one substitution at position i. So applying the two edits in Q on $\boldsymbol{v}$ also produces $\boldsymbol{t}$, i.e., edit $(\boldsymbol{t}, \boldsymbol{v}) \le 2$. Therefore, $\boldsymbol{v} \in N_n^2(\boldsymbol{s}) \cap N_n^2(\boldsymbol{t}) \cap B$. ◀

Now we bound the size of a $(1, 1)$-guaranteed subset from below.

▶ **Lemma 9.** *If B is (1,1)-guaranteed, then*

$$\textit{(i) for each } \boldsymbol{s} \in \mathcal{S}_n,\ \left|N_n^1(\boldsymbol{s}) \cap B\right| \ge \begin{cases} 1 & \textit{if } \boldsymbol{s} \in B \\ n & \textit{if } \boldsymbol{s} \notin B \end{cases}, \qquad \textit{(ii) } |B| \ge |\mathcal{S}_n|/|\Sigma| = |\Sigma|^{n-1}.$$

Proof. Let $B \subset \mathcal{S}_n$ be an arbitrary $(1,1)$-guaranteed subset. For part (i), because $\boldsymbol{s} \in N_n^1(\boldsymbol{s})$, if $\boldsymbol{s}$ is also in B, then $\boldsymbol{s}$ is in their intersection, hence $\left|N_n^1(\boldsymbol{s}) \cap B\right| \geq 1$. If $\boldsymbol{s} = s_1 s_2 \dots s_n \notin B$, then it must have at least n 1-neighbors $\boldsymbol{v}^i \in B$, one for each position $1 \leq i \leq n$, where $\boldsymbol{v}^i = s_1 \dots s_{i-1} v_i s_{i+1} \dots s_n$, $v_i \neq s_i$. Suppose conversely that this is not the case for a particular i. Let $\boldsymbol{t} = s_1 \dots s_{i-1} t_i s_{i+1} \dots s_n$ where $t_i \neq s_i$. We have $\operatorname{edit}(\boldsymbol{s}, \boldsymbol{t}) = 1$. Also, $N_n^1(\boldsymbol{s}) \cap N_n^1(\boldsymbol{t}) = \{x \in \Sigma \mid s_1 \dots s_{i-1} x s_{i+1} \dots s_n\}$, but none of them is in B (consider the two cases $x = s_i$ and $x \neq s_i$), i.e., $N_n^1(\boldsymbol{s}) \cap N_n^1(\boldsymbol{t}) \cap B = \varnothing$. This contradicts the assumption that B is $(1,1)$-guaranteed.

For part (ii), consider the collection of pairs $H = \left\{(\boldsymbol{s}, \boldsymbol{v}) \,\middle|\, \boldsymbol{s} \in \mathcal{S}_n \text{ and } \boldsymbol{v} \in N_n^1(\boldsymbol{s}) \cap B\right\}$. For all $\boldsymbol{v} \in B$, the number of sequences $\boldsymbol{s} \in \mathcal{S}_n$ with $\operatorname{edit}(\boldsymbol{s}, \boldsymbol{v}) \leq 1$ is $n(|\Sigma| - 1) + 1$. So $|H| = (n(|\Sigma| - 1) + 1)|B|$. On the other hand, part (i) implies that $|H| \geq |B| + n(|\Sigma|^n - |B|)$. Combined, we have $|B| \geq |\Sigma|^{n-1}$, as claimed. ◀

In Section 4.3, we give an algorithm to construct a $(1,1)$-guaranteed subset B that achieves the size $|B| = |\Sigma|^{n-1}$; furthermore, the corresponding $(1,3)$-sensitive bucketing function f_B^1 satisfies $\left|f_B^1(\boldsymbol{s})\right| = 1$ if $\boldsymbol{s} \in B$ and $\left|f_B^1(\boldsymbol{s})\right| = n$ if $\boldsymbol{s} \notin B$. This shows that the lower bounds proved above in Lemma 9 are tight and that the constructed $(1,1)$-guaranteed subset B is optimal in the sense of minimizing both $|B|$ and $\left|f_B^1(\boldsymbol{s})\right|$. Notice that this result improves Lemma 6 with $r = 1$ where we showed that $\mathcal{S}_n$ is a $(1,1)$-guaranteed subset of size $|\Sigma|^n$. According to Lemma 8, this constructed B is also (r,r)-guaranteed. So the corresponding bucketing function f_B^r is $(r, 2r+1)$-sensitive for all integers $r \geq 1$.

4.3 Construction of optimal $(1,1)$-guaranteed subsets

Let $m = |\Sigma|$ and denote the characters in Σ by $c_1, c_2, \dots, c_m$. We describe a recursive procedure to construct a $(1,1)$-guaranteed subset of $\mathcal{S}_n$. In fact, we show that $\mathcal{S}_n$ can be partitioned into m subsets $B_n^1 \sqcup B_n^2 \sqcup \dots \sqcup B_n^m$ such that each B_n^i is $(1,1)$-guaranteed. Here the notation $\sqcup$ denotes disjoint union. The partition of $\mathcal{S}_n$ is built from the partition of $\mathcal{S}_{n-1}$. The base case is $\mathcal{S}_1 = \{c_1\} \sqcup \dots \sqcup \{c_m\}$.

Suppose that we already have the partition for $\mathcal{S}_{n-1} = B_{n-1}^1 \sqcup B_{n-1}^2 \sqcup \dots \sqcup B_{n-1}^m$. Let

$$B_n^1 = \left(c_1 \circ B_{n-1}^1\right) \sqcup \left(c_2 \circ B_{n-1}^2\right) \sqcup \dots \sqcup \left(c_m \circ B_{n-1}^m\right),$$

where $c \circ B$ is the set obtained by prepending the character c to each sequence in the set B. For B_n^2, the construction is similar where the partitions of $\mathcal{S}_{n-1}$ are shifted (rotated) by one such that c_1 is paired with B_{n-1}^2, c_2 is paired with B_{n-1}^3, and so on. In general, for $1 \leq i \leq m$,

$$B_n^i = \left(c_1 \circ B_{n-1}^i\right) \sqcup \left(c_2 \circ B_{n-1}^{i+1}\right) \sqcup \dots \sqcup \left(c_{m-i+1} \circ B_{n-1}^m\right) \sqcup \left(c_{m-i+2} \circ B_{n-1}^1\right) \sqcup \dots \sqcup \left(c_m \circ B_{n-1}^{i-1}\right).$$

Examples of this partition for $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$ and $n = 2, 3$ are shown below.

$B_2^1 = \{\mathrm{AA}, \mathrm{CC}, \mathrm{GG}, \mathrm{TT}\}$
$B_2^2 = \{\mathrm{AC}, \mathrm{CG}, \mathrm{GT}, \mathrm{TA}\}$
$B_2^3 = \{\mathrm{AG}, \mathrm{CT}, \mathrm{GA}, \mathrm{TC}\}$
$B_2^4 = \{\mathrm{AT}, \mathrm{CA}, \mathrm{GC}, \mathrm{TG}\}$

$B_3^1 = \{$AAA, ACC, AGG, ATT, CAC, CCG, CGT, CTA,
GAG, GCT, GGA, GTC, TAT, TCA, TGC, TTG$\}$
$B_3^2 = \{$AAC, ACG, AGT, ATA, CAG, CCT, CGA, CTC,
GAT, GCA, GGC, GTG, TAA, TCC, TGG, TTT$\}$
$B_3^3 = \{$AAG, ACT, AGA, ATC, CAT, CCA, CGC, CTG,
GAA, GCC, GGG, GTT, TAC, TCG, TGT, TTA$\}$
$B_3^4 = \{$AAT, ACA, AGC, ATG, CAA, CCC, CGG, CTT,
GAC, GCG, GGT, GTA, TAG, TCT, TGA, TTC$\}$

Note that each sequence in $\mathcal{S}_n$ appears in exactly one of the subsets B_n^i, justifying the use of the disjoint union notation. (The induction proof of this claim has identical structure as the following proofs of Lemma 10 and 11, so we leave it out for conciseness.) Now we prove the correctness of this construction.

▶ **Lemma 10.** *Each constructed B_n^i is a minimum $(1,1)$-guaranteed subset of $\mathcal{S}_n$.*

Proof. By Lemma 9, we only need to show that each B_n^i is $(1,1)$-guaranteed and has size $|\Sigma|^{n-1} = m^{n-1}$. The proof is by induction on n. The base case $\mathcal{S}_1 = \{c_1\} \sqcup \cdots \sqcup \{c_m\}$ is easy to verify.

As the induction hypothesis, suppose that $\mathcal{S}_{n-1} = \bigsqcup_{j=1}^{m} B_{n-1}^j$, where each B_{n-1}^j is $(1,1)$-guaranteed and has size m^{n-2}. Consider an arbitrary index $1 \le i \le m$. By construction, we have $\left|B_n^i\right| = \sum_{j=1}^{m} \left|B_{n-1}^j\right| = m^{n-1}$. To show that B_n^i is $(1,1)$-guaranteed, consider two sequences $\boldsymbol{s}, \boldsymbol{t} \in \mathcal{S}_n$ with $\text{edit}(\boldsymbol{s}, \boldsymbol{t}) = 1$. If the single substitution happens on the first character, let $\boldsymbol{x} \in \mathcal{S}_{n-1}$ be the common $(n-1)$-suffix of $\boldsymbol{s}$ and $\boldsymbol{t}$. Since $\bigsqcup_{j=1}^{m} B_{n-1}^j$ is a partition of $\mathcal{S}_{n-1}$, $\boldsymbol{x}$ must appear in one of the subsets B_{n-1}^{ℓ}. In B_n^i, it is paired with one of the characters c_k. Let $\boldsymbol{y} = c_k \circ \boldsymbol{x}$, then $\boldsymbol{y} \in B_n^i$. Furthermore, $\boldsymbol{s}$ and $\boldsymbol{t}$ can each be transformed to $\boldsymbol{y}$ by at most one substitution on the first character. Thus, $\boldsymbol{y} \in N_n^1(\boldsymbol{s}) \cap N_n^1(\boldsymbol{t}) \cap B_n^i$.

If the single substitution between $\boldsymbol{s}$ and $\boldsymbol{t}$ does not happen on the first position, then they share the common first character c_k. In B_n^i, c_k is paired with one of the subsets B_{n-1}^{ℓ}. Let $\boldsymbol{s'}$ and $\boldsymbol{t'}$ be $(n-1)$-suffixes of $\boldsymbol{s}$ and $\boldsymbol{t}$, respectively. It is clear that $\text{edit}(\boldsymbol{s'}, \boldsymbol{t'}) = 1$. By the induction hypothesis, B_{n-1}^{ℓ} is $(1,1)$-guaranteed. So there is a sequence $\boldsymbol{x} \in B_{n-1}^{\ell}$ of length $n-1$ such that $\text{edit}(\boldsymbol{s'}, \boldsymbol{x}) \le 1$ and $\text{edit}(\boldsymbol{t'}, \boldsymbol{x}) \le 1$. Let $\boldsymbol{y} = c_k \circ \boldsymbol{x}$, then $\boldsymbol{y} \in B_n^i$ by the construction. Furthermore, $\text{edit}(\boldsymbol{s}, \boldsymbol{y}) = \text{edit}(\boldsymbol{s'}, \boldsymbol{x}) \le 1$ and $\text{edit}(\boldsymbol{t}, \boldsymbol{y}) = \text{edit}(\boldsymbol{t'}, \boldsymbol{x}) \le 1$. Thus, $\boldsymbol{y} \in N_n^1(\boldsymbol{s}) \cap N_n^1(\boldsymbol{t}) \cap B_n^i$. Therefore, B_n^i is $(1,1)$-guaranteed. Since the index i is arbitrary, this completes the proof. ◀

It remains to show that for each $\boldsymbol{s} \in \mathcal{S}_n$, $\left|N_n^1(\boldsymbol{s}) \cap B_n^i\right|$ matches the lower bound in Lemma 9. Together with Lemma 10, this proves that each constructed B_n^i yields an optimal $(1,3)$-sensitive bucketing function in terms of minimizing both the total number of buckets and the number of buckets each length-n sequence is sent to.

▶ **Lemma 11.** *For $\boldsymbol{s} \in \mathcal{S}_n$, each constructed B_n^i satisfies* $\left|N_n^1(\boldsymbol{s}) \cap B_n^i\right| = \begin{cases} 1 & \text{if } \boldsymbol{s} \in B_n^i \\ n & \text{if } \boldsymbol{s} \notin B_n^i \end{cases}$.

Proof. We proceed by induction on n. The base case $n = 1$ is trivially true because $|B_1^i| = 1$ and all single-character sequences are within one edit of each other. Suppose that the claim is true for $n-1$. Consider an arbitrary index i. If $\boldsymbol{s} \in B_n^i$, we show that any other length-n sequence $\boldsymbol{t} \in B_n^i$ has edit distance at least 2 from $\boldsymbol{s}$, namely $N_n^1(\boldsymbol{s}) \cap B_n^i = \{\boldsymbol{s}\}$. Let $\boldsymbol{s'}$ and

$\boldsymbol{t'}$ be the $(n-1)$-suffixes of $\boldsymbol{s}$ and $\boldsymbol{t}$ respectively. According to the construction, if $\boldsymbol{s}$ and $\boldsymbol{t}$ have the same first character, then $\boldsymbol{s'}$ and $\boldsymbol{t'}$ are in the same B^j_{n-1} for some index j. By the induction hypothesis, $\text{edit}(\boldsymbol{s'}, \boldsymbol{t'}) \geq 2$ (otherwise $\left|N^1_{n-1}(\boldsymbol{s'}) \cap B^j_{n-1}\right| \geq 2$), and therefore $\text{edit}(\boldsymbol{s}, \boldsymbol{t}) = \text{edit}(\boldsymbol{s'}, \boldsymbol{t'}) \geq 2$. If $\boldsymbol{s}$ and $\boldsymbol{t}$ are different at the first character, then $\boldsymbol{s'}$ and $\boldsymbol{t'}$ are not in the same B^j_{n-1}, so $\boldsymbol{s'} \neq \boldsymbol{t'}$ (recall that B^j_{n-1} and B^k_{n-1} are disjoint if $j \neq k$), namely $\text{edit}(\boldsymbol{s'}, \boldsymbol{t'}) \geq 1$. Together with the necessary substitution at the first character, we have $\text{edit}(\boldsymbol{s}, \boldsymbol{t}) = 1 + \text{edit}(\boldsymbol{s'}, \boldsymbol{t'}) \geq 2$.

If $\boldsymbol{s} \notin B^i_n$, Lemma 9 and 10 guarantee that $\boldsymbol{s}$ has n 1-neighbors $\boldsymbol{v}^k$ in B^i_n, $k = 1, \ldots, n$, where $\boldsymbol{v}^k$ is obtained from $\boldsymbol{s}$ by a single substitution at position k. Let $\boldsymbol{t} \neq \boldsymbol{s}$ be a 1-neighbor of $\boldsymbol{s}$. Since $\boldsymbol{t}$ can only differ from $\boldsymbol{s}$ by a single substitution at some position ℓ, we know that either $\boldsymbol{t} = \boldsymbol{v}^\ell$ or the edit distance between $\boldsymbol{t}$ and $\boldsymbol{v}^\ell$ is 1. In the latter case, $\boldsymbol{t}$ cannot be in B^i_n otherwise $\left|N^1_n\left(\boldsymbol{v}^\ell\right) \cap B^i_n\right| \geq 2$, contradicting the result of the previous paragraph. Therefore, $N^1_n(\boldsymbol{s}) \cap B^i_n = \left\{\boldsymbol{v}^1, \ldots \boldsymbol{v}^n\right\}$ which has size n. ◀

We end this section by showing that a membership query can be done in $O(n)$ time on the $(1,1)$-guaranteed subset B constructed above (i.e., $B = B^i_n$ for some i). Thanks to its regular structure, the query is performed without explicit construction of B. Consequently, the bucketing functions using B can be computed without computing and storing this subset of size $|\Sigma|^{n-1}$.

Specifically, suppose that we choose $B = B^i_n$ for some fixed $1 \leq i \leq m$. Let $\boldsymbol{s}$ be a given length-n sequence; we want to query if $\boldsymbol{s}$ is in B or not. This is equivalent to determining whether the index of the partition of $\mathcal{S}_n$ that $\boldsymbol{s}$ falls into is i or not. Write $\boldsymbol{s} = s_1 s_2 \ldots s_n$ and let $\boldsymbol{s'} = s_2 \ldots s_n$ be the $(n-1)$-suffix of $\boldsymbol{s}$. Suppose that it has been determined that $\boldsymbol{s'} \in B^j_{n-1}$ for some index $1 \leq j \leq m$, i.e., the sequence $\boldsymbol{s'}$ of length $n-1$ comes from the j-th partition of $\mathcal{S}_{n-1}$. By construction, the index ℓ for which $\boldsymbol{s} \in B^\ell_n$ is uniquely determined by the character $s_1 = c_k \in \Sigma$ and the index j according to the formula $\ell = (j + m + 1 - k) \bmod m$. The base case $n = 1$ is trivially given by the design that $c_p \in B^p_1$ for all $1 \leq p \leq m$. This easily translates into a linear-time algorithm that scans the input length-n sequence $\boldsymbol{s}$ backwards and compute the index ℓ such that $\boldsymbol{s} \in B^\ell_n$. To answer the membership query, we only need to check whether $\ell = i$. We provide an implementation of both the construction and the efficient membership query of a $(1,1)$-guaranteed subset at `https://github.com/Shao-Group/lsbucketing`.

4.4 A $(3,5)$-sensitive bucketing function

Let $B \subset \mathcal{S}_n$ be one of the constructed $(1,1)$-guaranteed subsets. Recall that the resulting bucketing function f^r_B is $(r, 2r+1)$-sensitive for all integers $r \geq 1$; in particular, f^2_B is $(2,5)$-sensitive. We are able to strengthen this result by showing that f^2_B is in fact $(3,5)$-sensitive.

▶ **Theorem 2.** *Let $B \subset \mathcal{S}_n$ be a $(1,1)$-guaranteed subset. The bucketing function f^2_B is $(3,5)$-sensitive.*

Proof. As f^r_B is already proved to be $(2,5)$-sensitive, to show it is $(3,5)$-sensitive, we just need to prove that, for any two sequences $\boldsymbol{s}, \boldsymbol{t} \in \mathcal{S}_n$ with $\text{edit}(\boldsymbol{s}, \boldsymbol{t}) = 3$, $f^2_B(\boldsymbol{s}) \cap f^2_B(\boldsymbol{t}) = N^2_n(\boldsymbol{s}) \cap N^2_n(\boldsymbol{t}) \cap B \neq \varnothing$. If the three edits are all substitutions, then there are length-n sequences $\boldsymbol{x}$ and $\boldsymbol{y}$ such that $\text{edit}(\boldsymbol{s}, \boldsymbol{x}) = \text{edit}(\boldsymbol{x}, \boldsymbol{y}) = \text{edit}(\boldsymbol{y}, \boldsymbol{t}) = 1$. Since B is $(1,1)$-guaranteed, there is a length-n sequence $\boldsymbol{z} \in B$ with $\text{edit}(\boldsymbol{x}, \boldsymbol{z}) \leq 1$ and $\text{edit}(\boldsymbol{y}, \boldsymbol{z}) \leq 1$. By triangle inequality, $\text{edit}(\boldsymbol{s}, \boldsymbol{z}) \leq \text{edit}(\boldsymbol{s}, \boldsymbol{x}) + \text{edit}(\boldsymbol{x}, \boldsymbol{z}) \leq 2$; $\text{edit}(\boldsymbol{t}, \boldsymbol{z}) \leq \text{edit}(\boldsymbol{t}, \boldsymbol{y}) + \text{edit}(\boldsymbol{y}, \boldsymbol{z}) \leq 2$. So $\boldsymbol{z} \in N^2_n(\boldsymbol{s}) \cap N^2_n(\boldsymbol{t}) \cap B$.

If the three edits are one substitution and a pair of indels, then there is a length-n sequence $\boldsymbol{x}$ such that edit $(\boldsymbol{s}, \boldsymbol{x}) = 1$ and edit $(\boldsymbol{x}, \boldsymbol{t}) = 2$ where the two edits between $\boldsymbol{x}$ and $\boldsymbol{t}$ can only be achieved by one insertion and one deletion. Let i be the position in $\boldsymbol{x}$ where the deletion between $\boldsymbol{x}$ and $\boldsymbol{t}$ takes place. Let $\boldsymbol{y}$ be a length-n sequence obtained from $\boldsymbol{x}$ by a substitution at position i, so edit $(\boldsymbol{x}, \boldsymbol{y}) = 1$. Since B is $(1, 1)$-guaranteed, there is a length-n sequence $\boldsymbol{z} \in B$ with edit $(\boldsymbol{x}, \boldsymbol{z}) \leq 1$ and edit $(\boldsymbol{y}, \boldsymbol{z}) \leq 1$. Then edit $(\boldsymbol{s}, \boldsymbol{z}) \leq$ edit $(\boldsymbol{s}, \boldsymbol{x}) +$ edit $(\boldsymbol{x}, \boldsymbol{z}) \leq 2$. Observe that $\boldsymbol{x}$ and $\boldsymbol{z}$ differ by at most one substitution at position i, which will be deleted when transforming to $\boldsymbol{t}$. So the two edits from $\boldsymbol{x}$ to $\boldsymbol{t}$ can also transform $\boldsymbol{z}$ to $\boldsymbol{t}$, namely, edit $(\boldsymbol{t}, \boldsymbol{z}) \leq 2$. Thus, $\boldsymbol{z} \in N_n^2(\boldsymbol{s}) \cap N_n^2(\boldsymbol{t}) \cap B$. ◀

5 Summary of proved LSB functions

We proposed two sets of LSB functions and studied the efficiency of them in terms of $|B|$, the total number of buckets, and $|f(\boldsymbol{s})|$, the number of buckets a specific length-n sequence $\boldsymbol{s}$ occupies. The results are summarized in Table 1.

Table 1 Results on (d_1, d_2)-sensitive bucketing functions of length-n sequences. Entries with $\leq$ show the best known upper bounds. Entries marked with a single star cannot be reduced under the specific bucketing method. Entries marked with double stars cannot be reduced in general. In column B, we use B_n^i to refer to a $(1, 1)$-guaranteed subset constructed in Section 4.3.

(d_1, d_2)-sensitive	B	$\lvert B\rvert$	$\lvert f(\boldsymbol{s})\rvert$	Ref.
$(1, 2)$	$\{1, \ldots, \lvert B\rvert\}$	$n\lvert\Sigma\rvert^{n-1}$**	n**	Theorem 1
$(1, 3)$	$\mathcal{S}_n$	$\lvert\Sigma\rvert^n$	$\lvert N_n^1(\boldsymbol{s})\rvert = (\lvert\Sigma\rvert - 1)n + 1$	Lemma 6
$(1, 3)$	B_n^i	$\lvert\Sigma\rvert^{n-1}$*	$\begin{cases} 1 & \text{if } \boldsymbol{s} \in B \\ k & \text{if } \boldsymbol{s} \notin B \end{cases}$*	Lemma 9–11
$(3, 5)$	B_n^i	$\lvert\Sigma\rvert^{n-1}$	$\leq \lvert N_n^2(\boldsymbol{s})\rvert$	Theorem 2
$(r, 2r+1)$, $r > 1$	B_n^i	$\lvert\Sigma\rvert^{n-1}$	$\leq \lvert N_n^r(\boldsymbol{s})\rvert$	Lemma 8, 10
$(2r-1, 2r+1)$, $r \geq 3$ odd	$\mathcal{S}_n$	$\lvert\Sigma\rvert^n$	$\lvert N_n^r(\boldsymbol{s})\rvert$	Lemma 6
$(2r, 2r+1)$, $r \geq 2$ even	$\mathcal{S}_n$	$\lvert\Sigma\rvert^n$	$\lvert N_n^r(\boldsymbol{s})\rvert$	Lemma 6

6 Experimental results on the gapped LSB functions

Several gapped LSB functions are introduced in Section 4. Now we investigate their behavior at the gap. We pick 3 LSB functions to experiment, corresponding to the rows 2–4 in Table 1. For $d = 1, 2, \ldots, 6$, we generate 100,000 random pairs $(\boldsymbol{s}, \boldsymbol{t})$ of sequences of length 20 with edit distance d. Each one of the picked LSB functions f_B^r is applied and the number of pairs that share a bucket under f_B^r is recorded. The code can be found at `https://github.com/Shao-Group/lsbucketing`. The results are shown in Figure 1.

Recall that Lemma 6 implies $f_{\mathcal{S}_n}^r$ is $(2r-1, 2r+1)$-sensitive when r is odd. The discussion after the proof shows that the gap at $2r$ indeed exists. In particular, if $\boldsymbol{s}$ can only be transformed to $\boldsymbol{t}$ by r pairs of indels, then $N_n^r(\boldsymbol{s}) \cap N_n^r(\boldsymbol{t}) = \varnothing$. On the other hand, if there are some substitutions among the $2r$ edits between $\boldsymbol{s}$ and $\boldsymbol{t}$, then by a similar construction as in the case where r is even, we can find a length-n sequence $\boldsymbol{v}$ such that edit $(\boldsymbol{s}, \boldsymbol{v}) =$ edit $(\boldsymbol{v}, \boldsymbol{t}) = r$. Motivated by the this observation, we further explore the performance of the LSB functions at the gap for different types of edits. Given a gapped LSB function f, for the gap at d, define categories $0, \ldots, \lfloor d/2 \rfloor$ corresponding to the types of edits: a pair of length-n sequences with edit distance d is in the i-th category if they can

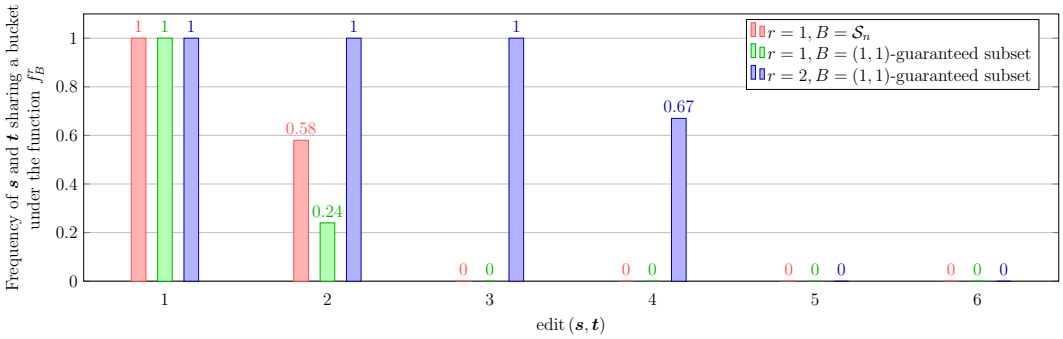


Figure 1 Probabilities (estimated by frequencies) that two sequences share a bucket with respect to their edit distance under three gapped LSB functions (red, green, and blue bars correspond to the rows 2–4 of Table 1).

be transformed to each other with i pairs of indels (and $d - 2i$ substitutions) but not $i - 1$ pairs of indels (and $d - 2i + 2$ substitutions). Figure 2 shows the results for the three LSB functions in Figure 1 at their respective gaps with respect to different types of edits. Observe that the result for $f^1_{\mathcal{S}_n}$ (in red) agrees with our analysis above.

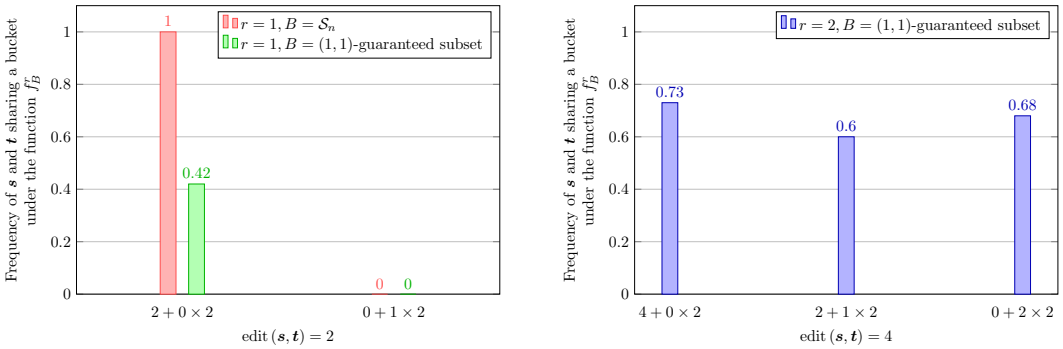


Figure 2 Probabilities (estimated by frequencies) that two sequences share a bucket with respect to their edit type under three gapped LSB functions. The types of edits are labeled in the format $a + b \times 2$ where a is the number of substitutions and b is the number of pairs of indels. Left: two $(1, 3)$-sensitive bucketing functions (rows 2 and 3 of Table 1). Right: the $(3, 5)$-sensitive bucketing function (row 4 of Table 1).

7 Conclusion and Discussion

We introduce locality-sensitive bucketing (LSB) functions, that generalize locality-sensitive hashing (LSH) functions by allowing it to map a sequence into multiple buckets. This generalization makes the LSB functions easier to construct, while guaranteeing high sensitivity and specificity in a deterministic manner. We construct such functions, prove their properties, and show that some of them are optimal under proposed criteria. We also reveal several properties and structures of the metric space $\mathcal{S}_n$, which are of independent interests for studying LSH functions and the edit distance.

Our results for LSB functions can be improved in several aspects. An obvious open problem is to design (d_1, d_2)-sensitive functions that are not covered here. For this purpose, one direction is to construct optimal (r, r)-guaranteed subsets for $r > 1$. As an implication

of Lemma 11, it is worth noting that the optimal $(1, 1)$-guaranteed subset is a maximal independent set in the undirected graph G_n^1 whose vertex set is $\mathcal{S}_n$ and each sequence is connected to all its 1-neighbors. It is natural to suspect that similar results hold for (r, r)-guaranteed subsets with larger r. Another approach is to use other more well-studied sets as buckets and define LSB functions based on their connections with $\mathcal{S}_n$. This is closely related to the problem of embedding $\mathcal{S}_n$ which is difficult as noted in the introduction. Our results in Section 3 suggest a new angle to this challenging problem: instead of restricting our attention to embedding $\mathcal{S}_n$ into metric spaces, it may be beneficial to consider a broader category of spaces that are equipped with a non-transitive relation (here in LSB functions we used subsets of integers with the "have a nonempty intersection" relation). Yet another interesting future research direction would be to explore the possibility of improving the practical time and space efficiency of computing and applying LSB functions.

A technique commonly used to boost the sensitivity of an LSH function is known as the OR-amplification. It combines multiple LSH functions in parallel, which can be viewed as sending each sequence into multiple buckets such that the probability of having similar sequences in one bucket is higher than using the individual functions separately. However, as a side effect, the OR-amplification hurts specificity: the chance that dissimilar sequences share a bucket also increases. It is therefore necessary to combine it with other techniques and choosing parameters to balance sensitivity and specificity is a delicate work. On contrast, the LSB function introduced in this paper achieves a provably optimal separation of similar and dissimilar sequences. In addition, the OR-amplification approach can also be applied on top of the LSB functions as needed.

References

1. Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.
2. Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389–3402, 1997.
3. Z. Bar-Yossef, T.S. Jayram, R. Krauthgamer, and R. Kumar. Approximating edit distance efficiently. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 550–559, 2004.
4. Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature Biotechnology*, 33(6):623–630, 2015.
5. Andrea Califano and Isidore Rigoutsos. FLASH: A fast look-up algorithm for string homology. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 353–359. IEEE, 1993.
6. Junjie Chen, Mingyue Guo, Xiaolong Wang, and Bin Liu. A comprehensive review and comparison of different computational methods for protein remote homology detection. *Briefings in Bioinformatics*, 19(2):231–244, 2018.
7. Dan DeBlasio, Fiyinfoluwa Gbosibo, Carl Kingsford, and Guillaume Marçais. Practical universal k-mer sets for minimizer schemes. In *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics (BCB'19)*, pages 167–176, New York, NY, USA, 2019. Association for Computing Machinery.
8. Miten Jain, Sergey Koren, Karen H Miga, Josh Quick, Arthur C Rand, Thomas A Sasani, John R Tyson, Andrew D Beggs, Alexander T Dilthey, Ian T Fiddes, et al. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature Biotechnology*, 36(4):338–345, 2018.

9 Robert Krauthgamer and Yuval Rabani. Improved lower bounds for embeddings into l_1. *SIAM Journal on Computing*, 38(6):2487–2498, 2009.

10 Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

11 Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

12 Denise Mak, Yevgeniy Gelfand, and Gary Benson. Indel seeds for homology search. *Bioinformatics*, 22(14):e341–e349, 2006.

13 Guillaume Marçais, Dan DeBlasio, and Carl Kingsford. Asymptotically optimal minimizers schemes. *Bioinformatics*, 34(13):i13–i22, 2018.

14 Guillaume Marçais, Dan DeBlasio, Prashant Pandey, and Carl Kingsford. Locality-sensitive hashing for the edit distance. *Bioinformatics*, 35(14):i127–i135, 2019.

15 Samuel McCauley. Approximate similarity search under edit distance using locality-sensitive hashing. In *24th International Conference on Database Theory (ICDT 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

16 Yaron Orenstein, David Pellow, Guillaume Marçais, Ron Shamir, and Carl Kingsford. Designing small universal k-mer hitting sets for improved analysis of high-throughput sequencing. *PLoS Computational Biology*, 13(10):e1005777, 2017.

17 Rafail Ostrovsky and Yuval Rabani. Low distortion embeddings for edit distance. *Journal of the ACM (JACM)*, 54(5):23–es, 2007.

18 Anthony Rhoads and Kin Fai Au. PacBio sequencing and its applications. *Genomics, Proteomics & Bioinformatics*, 13(5):278–289, 2015.

19 Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

20 Michael Roberts, Brian R Hunt, James A Yorke, Randall A Bolanos, and Arthur L Delcher. A preprocessor for shotgun assembly of large genomes. *Journal of Computational Biology*, 11(4):734–752, 2004.

21 Kristoffer Sahlin. Effective sequence similarity detection with strobemers. *Genome Research*, 31(11):2080–2094, 2021.

22 Kristoffer Sahlin, Marta Tomaszkiewicz, Kateryna D Makova, and Paul Medvedev. Deciphering highly similar multigene family transcripts from Iso-Seq data with IsoCon. *Nature Communications*, 9(1):1–12, 2018.

23 Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD (International Conference on Management of Data)*, pages 76–85, 2003.

24 Yan Song, Haixu Tang, Haoyu Zhang, and Qin Zhang. Overlap detection on long, error-prone sequencing reads via smooth q-gram. *Bioinformatics*, 36(19):4838–4845, 2020.

phyBWT: Alignment-Free Phylogeny via eBWT Positional Clustering

Veronica Guerrini[1]
Dipartimento di Informatica, University of Pisa, Italy

Alessio Conte
Dipartimento di Informatica, University of Pisa, Italy

Roberto Grossi
Dipartimento di Informatica, University of Pisa, Italy

Gianni Liti
CNRS UMR 7284, INSERM U 1081, Université Côte d'Azur, France

Giovanna Rosone[1]
Dipartimento di Informatica, University of Pisa, Italy

Lorenzo Tattini
CNRS UMR 7284, INSERM U 1081, Université Côte d'Azur, France

Abstract

Molecular phylogenetics is a fundamental branch of biology. It studies the evolutionary relationships among the individuals of a population through their biological sequences, and may provide insights about the origin and the evolution of viral diseases, or highlight complex evolutionary trajectories.

In this paper we develop a method called phyBWT, describing how to use the extended Burrows-Wheeler Transform (eBWT) for a collection of DNA sequences to directly reconstruct phylogeny, bypassing the alignment against a reference genome or de novo assembly. Our phyBWT hinges on the combinatorial properties of the eBWT positional clustering framework. We employ eBWT to detect relevant blocks of the longest shared substrings of varying length (unlike the k-mer-based approaches that need to fix the length k a priori), and build a suitable decomposition leading to a phylogenetic tree, step by step. As a result, phyBWT is a new alignment-, assembly-, and reference-free method that builds a partition tree without relying on the pairwise comparison of sequences, thus avoiding to use a distance matrix to infer phylogeny.

The preliminary experimental results on sequencing data show that our method can handle datasets of different types (short reads, contigs, or entire genomes), producing trees of quality comparable to that found in the benchmark phylogeny.

2012 ACM Subject Classification Applied computing → Bioinformatics; Mathematics of computing → Combinatorial algorithms

Keywords and phrases Phylogeny, partition tree, BWT, positional cluster, alignment-free, reference-free, assembly-free

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.23

Supplementary Material *Software (Source Code)*: `https://github.com/veronicaguerrini/phyBWT`

Funding Work partially supported by Italian Ministry for Education and Research – PRIN Project n. 20174LF3T8 AHeAD and by University of Pisa - Project no. PRA_2020–2021_26 "Metodi Informatici Integrati per la Biomedica".

[1] Corresponding author

1 Introduction

Molecular phylogenetics are a key tool for understanding the evolutionary relationships among biological sequences. Phylogenies, in the form of rooted or unrooted trees, can be used for several purposes: to reconstruct the ancestry of the species (or other taxa) on the tree of life, to understand the epidemiological dynamics of pathogens, and to identify and study complex evolutionary events such as hybridisation [17, 34], introgression [12], and horizontal gene transfer [33]. Thus, they are successfully employed in almost every branch of biology, including e.g. population genomics and metagenomics, ecology, and biogeography [40].

A vast array of techniques for inferring phylogeny have been developed over the years [37]. Sequence-based methods analyze the DNA or RNA sequences of the taxa, and are based on their similarity or dissimilarity detection. Most of them rely on a distance matrix by computing the pairwise evolutionary distances between every pair of input sequences. Standard algorithms, such as the neighbour-joining algorithm [32], are then applied to the distance matrix to perform the tree reconstruction.

A crucial component is how to compute these evolutionary distances. Sequence alignment is a central task in distance computation, which is performed on either entire sequences or parts of them, with the optional usage of a reference genome. With the advent of new sequencing technologies and the completion of various genome projects, the number of whole-genome sequence data available has increased and a new era for phylogeny started. Owing to the rising cost of the alignment task, alignment-free approaches for quantifying the similarity/dissimilarity between sequences have been introduced: an advantage of these approaches is that they are robust for recombination and shuffling events [36, 35, 42]. As the majority of alignment-free approaches for phylogenetic reconstruction transforms each sequence into a multiset of k-mers, *i.e.* substrings of length k extracted from the input sequences, they can also work directly on the reads obtained from the sequencing platforms, without the need of performing a preliminary assembly of these reads.

In this paper we introduce phyBWT, which combines many features in a single new method to reconstruct a phylogenetic tree. Firstly it is alignment-, assembly-, and reference-free. Second, it does not need a distance matrix as it does not rely on the pairwise comparison of sequences. Third, it exploits the combinatorial properties of the *positional clustering* framework recently introduced in [28], overcoming the limitations of employing k-mers with fixed size k a priori.

The contribution of this paper is twofold, theoretical as well as practical. To the best of our knowledge, phyBWT is the first method that applies the properties of the Extended Burrows-Wheeler Transform (eBWT), employed in the positional clustering, to the idea of decomposition for phylogenetic inference. Moreover, phyBWT not only is oblivious to extra information such as reference sequences or read mappings, but it also avoids any assembly or alignment of input sequences. Finally, phyBWT infers the tree structure by comparing all the sequences simultaneously and efficiently, instead of performing their pairwise comparisons. We present a preliminary experimental evaluation of phyBWT for reconstructing phylogenetic trees using different types of biological sequences (short reads and de novo assemblies/genomes) and different taxa. The phylogenetic trees produced by phyBWT are of high quality according to the benchmarks in the literature.

State of the art

The Burrows-Wheeler Transform (BWT) [9] of a string (and the eBWT of a set of strings [25, 5]) is a suitable permutation of the symbols of the string(s), whose output shows a local similarity, *i.e.* symbols preceding similar contexts tend to occur in clusters. Both transformations

have been intensively studied with important and successful applications in several areas. For instance, the eBWT has been used for defining alignment-free methods based on a pairwise distance matrix [25, 26, 39, 19] in order to build up a phylogenetic tree for mitochondrial DNA genomes. The positional clustering detects "interesting" blocks in the output of the eBWT [25, 5], so that the requirement on the fixed size k in k-mers is relaxed and becomes of variable-order, not fixed a priori, in an adaptive way according to the contexts. This framework has already been used in other bioinformatics tasks, such as for detecting SNPs and INDELs in short-read datasets [29] and for lossy compression of FASTQ datasets [18].

We observe that phyBWT exploits the underlying properties of the eBWT: (i) the clustering effect, *i.e.*, the fact that the eBWT tends to group together equal symbols in the transformed string that occur in similar contexts in the input string collection; (ii) the fact that if a substring x occurs in one or more strings, then the suffixes of the input dataset starting with x-occurrence are close in the sorted list of suffixes. In other words, the greater the number of these substrings shared by two taxa is, the more they are similar.

Although phyBWT does not use the distance matrix, it has some resemblance with split decomposition methods when reconstructing the tree from the information gathered through the eBWT. We recall that split decomposition relies on a solid mathematical ground [2, 4], and has been successfully applied to phylogeny [3]. The idea is to score the possible splits (*i.e.* bipartitions) of the taxa, and assign an isolation index to each split based on the distances in the given matrix. Compatible splits are those with an empty intersection on one of the parts in the splits, and the isolation index is treated as a priority weight in making a (greedy) choice among the splits. Compatible splits induce a tree and vice versa. However, a residual error is generated on real-world data, and a notion of weak split compatibility is preferred to create a weighted phylogeny network instead of a phylogeny tree: the shortest weighted part between any two nodes in this network gives the isolation index in the corresponding split. For ℓ taxa, only $O(\ell^2)$ splits are needed for split decomposition instead of 2^ℓ ones [2].

As the original algorithm in the seminal papers on split decomposition [2, 3] requires $O(\ell^6)$ comparisons, further papers have addressed efficiency and extended these ideas. The recent alignment-free method SANS [38, 31] uses the notions of the split decomposition theory to greedily build a list of weakly compatible splits from which to infer phylogenies. In the list, each split has its own weight computed by counting k-mers that are stored in a colored de Bruijn graph [38] (this has been improved later by hashing [31], leaving the colored de Bruijn graph as input option). The calculated list of splits ordered by weight is then filtered according to two strategies that are described and implemented in the software tool SplitsTree [20]. In our experimental study, we compare the trees obtained by SANS and phyBWT. It should be noted that SANS is also able to reconstruct phylogenetic networks whereas phyBWT focuses on phylogenetic trees only.

As previously mentioned, a plethora of methods have been designed for phylogeny (*e.g.* DBLP reports over 500 papers having "phylogeny" in the title). We refer the reader to [21, 22, 37] for a complete and detailed review of various methods for phylogeny estimation. We briefly mention here that among the alignment-based approaches are character-based methods [40], that generally produce alignments of the input sequences and compare all sequences simultaneously considering one character per time (*e.g.* using maximum parsimony or maximum likelihood). The alignment-free tree reconstruction comes from computing some distribution within and among k-mers by using a distance matrix or not. For instance, the method in [14] reconstructs a phylogeny from whole-genome short-read sequencing data on the basis on a matrix of pairwise genetic distances, without assembling the reads.

2 Notation and background

2.1 Notation

Let s be a string (also called sequence) of length n on the alphabet Σ. We denote the i-th symbol of s by $s[i]$. A *substring* of any s is denoted as $s[i,j] = s[i] \cdots s[j]$, with $s[1,j]$ being called a *prefix* and $s[i,n+1]$ a *suffix* of s. A *k-mer* is a string of length k.

Let $\mathcal{S} = \{s_1, s_2, \ldots, s_\ell\}$ be a collection of ℓ strings. We assume that each string $s_i \in \mathcal{S}$ has length n_i and is followed by a special end-marker symbol $S_i[n_i+1] = \$_i$, which is lexicographically smaller than any other symbol in $\mathcal{S}$, and does not appear in $\mathcal{S}$ elsewhere[2].

2.2 Basic data structures

The *Burrows-Wheeler Transform* (BWT) [9] is a well-known widely used reversible string transformation that can be extended to a collection of strings. Such an extension, introduced in [25], is a reversible transformation whose output string (denoted by $\mathsf{ebwt}(\mathcal{S})$) is a permutation of the symbols of all strings in $\mathcal{S}$. In [5], the authors introduced a variant of this transformation for string collection in which a distinct end-marker is appended to each string, making the collection ordered. Such transformations are known as eBWT or multi-string BWT.

The length of $\mathsf{ebwt}(\mathcal{S})$ is denoted by $N = \sum_{i=1}^{\ell}(n_i+1)$, and $\mathsf{ebwt}(\mathcal{S})[i] = x$, with $1 \leq i \leq N$, if x circularly precedes the i-th suffix $S_j[k, n_j+1]$ (for some $1 \leq j \leq \ell$ and $1 \leq k \leq n_j+1$), according to the lexicographic sorting of the suffixes of all strings in $\mathcal{S}$.

Usually the output string $\mathsf{ebwt}(\mathcal{S})$ is enhanced with the *document array* (DA) and *longest common prefix* (LCP) array of $\mathcal{S}$.

The *document array* of $\mathcal{S}$ (denoted by $\mathsf{da}(\mathcal{S})$) is the array of length N such that $\mathsf{da}(\mathcal{S})[i] = j$, with $1 \leq j \leq \ell$ and $1 \leq i \leq N$, where $\mathsf{ebwt}(\mathcal{S})[i]$ is a symbol of the string s_j.

The *longest common prefix* (LCP) array [24] of $\mathcal{S}$ is the array $\mathsf{lcp}(\mathcal{S})$ of length $N+1$, such that $\mathsf{lcp}(\mathcal{S})[i]$, with $2 \leq i \leq N$, is the length of the longest common prefix between the suffixes associated with the positions i and $i-1$ in $\mathsf{ebwt}(\mathcal{S})$, and $\mathsf{lcp}(\mathcal{S})[1] = \mathsf{lcp}(\mathcal{S})[N+1] = 0$ by default. The set $\mathcal{S}$ can be omitted when it is clear from the context.

The following is an important property of the BWT, and thus of the related data structures DA and LCP, that will be used in our method:

▶ Remark 1. One can obtain the DA of a subset of $\mathcal{S}$ by scanning the DA($\mathcal{S}$). In [5], the authors prove that given a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_\ell\}$ of strings and $\mathsf{ebwt}(\mathcal{S})$, one can obtain the eBWT of a subset $\mathcal{R}$ of $\mathcal{S}$ by removing all the characters not in $\mathcal{R}$, without constructing the eBWT from scratch, as the relative order of suffixes holds. Similarly, one can obtain the LCP of a subset of $\mathcal{S}$ by using the properties of the LCP array.

Let $\mathcal{R} \subset \mathcal{S}$. We denote by $\mathsf{ebwt}(\mathcal{S})|_{\mathcal{R}}$ (resp. $\mathsf{da}(\mathcal{S})|_{\mathcal{R}}$, $\mathsf{lcp}(\mathcal{S})|_{\mathcal{R}}$) the restriction of the data structure $\mathsf{ebwt}(\mathcal{S})$ (resp. $\mathsf{da}(\mathcal{S})$, $\mathsf{lcp}(\mathcal{S})$) to the set of strings $\mathcal{R}$.

2.3 LCP-interval and k-mer vs Positional cluster

The LCP-intervals [1] of lcp-value k, or k-intervals, are maximal intervals $[i,j]$ that satisfy $\mathsf{lcp}(\mathcal{S})[r] \geq k$ for $i < r \leq j$. In other words, the suffixes associated with k-intervals have a common k-mer as prefix.

[2] Note that, in the implementations, one can use a single symbol as end-marker for all strings, but end-markers from different strings are then sorted on the basis of the index and the relative order of the strings in the set they belong to.

In any string collection, thus, LCP-intervals of lcp-value k are in a one-to-one correspondence with the set of all k-mers.

Note that the common prefix w in a LCP-interval is of length at least k, but it could be longer. So, to overcome the limitation of strategies based on LCP-intervals that require to fix the length k, the authors of [28, 29] introduced a new framework called "positional clustering". In this framework the intervals do not depend on a value k fixed a-priori, but they are enclosed between two "local minima" in the LCP-array (thus, their boundaries are data-driven).

Crucially, the length k of the common prefix w of the suffixes inside such intervals is not the same, but it differs interval by interval. Hence, there is no one-to-one correspondence between such intervals and the set of k-mers.

However, as to exclude intervals corresponding to some short random contexts w, one needs to set a minimum length for w, which we denote by k_m.

According to [29], an *eBWT positional cluster* $\text{eBWTCLUST}[i, j]$ is a maximal substring $\mathsf{ebwt}[i, j]$ where $\mathsf{lcp}[r] \geq k_m$, for all $i < r \leq j$, and none of the indices r, $i < r \leq j$, is a *local minimum* of the LCP array.

By definition, we have that:

▶ Remark 2. Any two eBWT positional clusters, $\text{eBWTCLUST}[i, j]$ and $\text{eBWTCLUST}[i', j']$, such that $i \neq i'$ are disjoint, *i.e.* it holds that either $j < i'$ or $j' < i$.

Here, we define a local minimum of the LCP array (of length N) any index i, $1 < i < N$ such that $\mathsf{lcp}[i-1] > \mathsf{lcp}[i]$ and $\mathsf{lcp}[i] < \mathsf{lcp}[i+j]$, where $j > 1$ is the number of adjacent occurrences of the value $\mathsf{lcp}[i]$ from position i. For instance, let $\mathsf{lcp} = [2, 1, 3, 3, 5, 4, 2, 2, 7]$. The local minima are indices 2 and 7.

Note that the above definition differs from that in [29], where local minima in the LCP array (of length N) are detected searching for indices r such that $\mathsf{lcp}[r-1] \geq \mathsf{lcp}[r] < \mathsf{lcp}[r+1]$, for all $1 < r \leq N$. According to such definition, local minima can be detected in any non-increasing sequence where some values are repeated. For instance, for the second occurrence of 4 in the sequence $5, 4, 4, 2$ yields the definition of local minimum. Therefore, the slightly different notion of local minima we use is to maximize the length of the non-increasing sequence described in the following Remark 3.

▶ Remark 3 ([28], Thm 3.3). In any eBWT positional cluster, the lcp-values form a sequence of non-decreasing values followed by a (possibly empty) sequence of non-increasing values.

From the above remark follows that the length l of the longest common prefix shared by the suffixes associated with a eBWT positional cluster $\mathsf{ebwt}[i, j]$ is given by the minimum value in $\mathsf{lcp}[i+1, j]$, which could be simply obtained by taking the minimum between the values $\mathsf{lcp}[i+1]$ and $\mathsf{lcp}[j]$.

In general, if we set the minimum length k_m equal to k, the set of eBWT positional clusters forms a refinement of the set of $\mathsf{ebwt}[i, j]$ with $[i, j]$ LCP-interval of lcp-value k.

In fact, any $\mathsf{ebwt}[i, j]$, where $[i, j]$ is a LCP-interval, can be subdivided in correspondence of the local minima of $\mathsf{lcp}[i, j]$, thus giving rise to a sequence of consecutive eBWT positional clusters (see Figure 1). Clearly, such subdivision depends only on the trend of the LCP values inside the LCP-interval $[i, j]$. Hence, more than one positional cluster can be related to the same LCP-interval, and equivalently, to the same k-mer.

▶ **Example 4** (running example)**.** In Figure 1, we represent the data structures used in our tool (cda, ebwt, lcp), the auxiliary array da and the sorted list of suffixes, for the sake of clarity. The LCP-intervals of lcp-value $k = 1$ correspond to the following intervals: $[4, 10]$, $[11, 17]$, $[18, 28]$, $[29, 34]$. Whereas the horizontal lines delimit eBWTCLUST for $k_{min} = 1$.

i	cda	da	lcp	ebwt	Sorted suffixes
1	1	1	0	A	$
2	2	2	0	T	$
3	3	3	0	T	$
4	1	1	0	C	A$
5	1	1	1	T	ACCA$
6	3	3	2	T	ACGACT$
7	3	3	4	$	ACGAGTACGACT$
8	3	3	2	G	ACT$
9	3	3	1	G	AGTACGACT$
10	2	2	1	T	ATT$
11	1	1	0	**C**	CA$
12	1	1	1	**A**	CCA$
13	3	3	1	**A**	CGACT$
14	3	3	3	**A**	CGAGTACGACT$
15	1	1	2	G	CGTACCA$
16	2	2	4	G	CGTATT$
17	3	3	1	A	CT$
18	3	3	0	C	GACT$
19	3	3	2	C	GAGTACGACT$
20	1	1	1	G	GCGTACCA$
21	2	2	5	G	GCGTATT$
22	1	1	1	**$**	GGCGTACCA$
23	2	2	6	**G**	GGCGTATT$
24	2	2	2	G	GGGCGTATT$
25	2	2	3	$	GGGGCGTATT$
26	1	1	1	C	GTACCA$
27	3	3	4	A	GTACGACT$
28	2	2	3	C	GTATT$
29	2	2	0	T	T$
30	3	3	1	C	T$
31	1	1	1	G	TACCA$
32	3	3	3	G	TACGACT$
33	2	2	2	G	TATT$
34	2	2	1	A	TT$

Figure 1 Extended Burrows-Wheeler Transform (EBWT), LCP array, and the auxiliary data structures DA and CDA for the set $\mathcal{S} = \{\text{GGCGTACCA, ACGAGTACGACT, GGGGCGTATT}\}$.

Note that when $k_{min} = k$, the eBWTclust can refine the LCP-intervals. For example the LCP-interval $[18, 28]$ includes five positional clusters: eBWTclust$[18, 19]$, eBWTclust$[20, 21]$, eBWTclust$[22, 23]$, eBWTclust$[24, 25]$, eBWTclust$[26, 28]$.

3 Method

In this section, we describe the proposed method for building a phylogenetic tree where each leaf, representing an organism, is a set of strings (*e.g.* sequencing reads, contigs, genome).

The idea of our method is to reconstruct the tree through a series of partitions performed on groups of organisms. As these partitions isolate groups of organisms from each other, we proceed in both directions: we divide each part in one direction, and we group the parts together in the other direction. Each part corresponds to a node of the phylogeny tree. When it is not possible to further divide or group together, we draw the edges of the tree from those groups to a node corresponding to their union.

The partitions generated by our method are intended to estimate phylogenetic signals; in particular, each part produces evidence of separations among the input set of nodes.

More formally, we denote the set of leaves as $\mathcal{S} = \{S_1, S_2, \ldots, S_\ell\}$, where each leaf $S_i \in \mathcal{S}$ contains m_i strings including their reverse-complement, *i.e.* $S_i = \{s_{i,1}, \ldots, s_{i,m_i}\}$, where $s_{i,\frac{m_i}{2}}, \ldots, s_{i,m_i}$ are strings in the reverse-complement form. Each node of the tree identifies a set of organisms (*i.e.* it is a subset of $\mathcal{S}$).

Subsection 3.1 describes how the partitioning procedure is iteratively called to infer a phylogeny tree for $\mathcal{S}$, while Subsection 3.2 provides the details of the inner partition algorithm.

The idea behind the partitioning algorithm, described in Subsection 3.2, is to group together nodes whose associated strings share long common substrings of varying length which are *not* present in other nodes, and we interpret the presence of such substrings as a common feature of the group that differentiates it from the others.

To perform partitions we do not use any external information, such as reference sequences or annotations, and in addition, we do not perform assembly or alignment.

Finally, we recall that, in any tree, a *cut* of a given subset of its edges determines a partition of the set of its leaves. Therefore, the notion of partition of $\mathcal{S}$ we use slightly differs from the one of split, as a split of the set of leaves is obtained by removing a single edge from the corresponding tree. Thus, splits are essentially bi-partitions of $\mathcal{S}$.

Algorithm 1 TreeReconstruction($\mathcal{S}$).

input : $\mathcal{S} = \{S_1, \ldots, S_\ell\}$
output : A tree whose leaves are the elements of $\mathcal{S}$

Initialize $\mathcal{Q} \leftarrow \{\{S_1\}, \ldots, \{S_\ell\}\}$
Queue.push($\mathcal{Q}$)
while Queue *is not empty* **do**
  $\{Q_1, \ldots, Q_q\} \leftarrow$ Queue.pop()
  for $i \in \{1, \ldots, q\}$ **do** Create node Q_i if it does not exists
  if $q > 1$ **then**
    $\mathcal{P} \leftarrow$ PARTITION($\{Q_1, \ldots, Q_q\}$)
    $p \leftarrow \mathcal{P}.size()$
    if $p = q$ **then** `/* the partitioning cannot aggregate further */`
      Create node $\mathcal{Q} = \bigcup\{Q_1, \ldots, Q_q\}$ if it does not exists
      Draw edges from the node $\mathcal{Q}$ to nodes Q_i, for all $i \in \{1, \ldots, q\}$
    else
      Queue.push($\{\bigcup P_1, \ldots, \bigcup P_p\}$) `/* link partitions to each other */`
      for $i \in \{1, \ldots, p\}$ **do**
        Queue.push(P_i) `/* link elements within the partition */`

3.1 Partitioning-based tree reconstruction

We here describe a method that, given a partitioning algorithm for sets of strings as a blackbox, is able to reconstruct a tree by applying the partitioning step by step.

The blackbox for partitioning is described in Subsection 3.2, and denoted here by PARTITION. Given a set $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$, the output of PARTITION is a non-empty collection $\mathcal{P}$ of subsets of $\mathcal{Q}$, *i.e.* $\mathcal{P} = \{P_1, \ldots, P_p\}$, such that

1. $\bigcup_{i=1}^{p} P_i = \mathcal{Q}$, *i.e.*, the union of all $P_i \in \mathcal{P}$ is $\mathcal{Q}$.
2. $P_i \cap P_j = \emptyset$ for all $i \neq j$, *i.e.*, every pair of sets in $\mathcal{P}$ has empty intersection.

The input set $\mathcal{Q}$ is a collection of sets of organisms. The key in this procedure is that each part P_i of $\mathcal{Q}$ generated by PARTITION groups sets of organisms that share similarity to each other and divergence from the others. As the trivial partition $\mathcal{P} = \{\mathcal{Q}\}$ provides no significant information, we assume PARTITION is such that each part P_i will be a proper subset of $\mathcal{Q}$.

Formally, each element Q_i will identify a node of the tree, and thus, a set of organisms (*i.e.* a subset of $\mathcal{S}$). In particular, each leaf $Q_i = \{S_i\}$, will be a singleton set containing a single organism, while each internal node Q_i will be the union of the leaves of its subtree.

For the sake of clarity, we explicitly create the nodes of the tree (by their corresponding set of organisms) while reconstructing it (see Algorithm 1).

For convenience, given a collection of sets $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$, we will use the notation $\bigcup \mathcal{Q}$ as a shorthand for $Q_1 \cup \ldots \cup Q_q$, *i.e.*, the set of all elements of the sets of $\mathcal{Q}$.

Algorithm structure. To give an intuitive overview, the algorithm iteratively considers several sets of organisms $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ and aggregates them into *groups of sets* of organisms by means of PARTITION, obtaining a partition $\mathcal{P} = \{P_1, \ldots, P_p\}$ with each $P_i \subset \mathcal{Q}$. At each call of PARTITION, if the output $\mathcal{P}$ is not the finest partition on $\mathcal{Q}$ (*i.e.* $p < q$), then it proceeds in two directions:

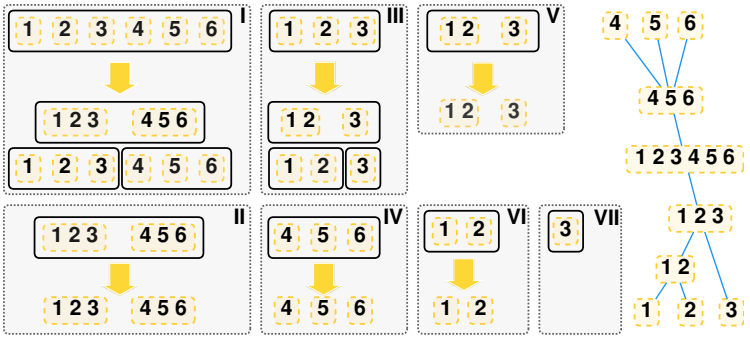


Figure 2 A possible execution of Algorithm 1 on a set of organisms $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$. Each panel shows an iteration of the algorithm: the rectangle at the top is the element taken from the queue, and the arrows points to the elements generated from the queue; dashed sets within the queue elements represent the sets of organisms. Iterations II,IV,V and VI meet the condition in Line 9: no new element of the queue is produced (we only show the result of PARTITION) and edges are generated. To the right is the resulting tree.

- Each part P_i will be a node of the tree (identified by the union of its contained Q_j sets), and the set $\mathcal{P}$ is recursively fed to PARTITION.
- Within each single part P_i, each set $Q_j \in P_i$ is a node, and these nodes will be recursively aggregated by calling PARTITION on P_i to induce a tree structure.

Otherwise, if the output $\mathcal{P}$ is the partition of singletons (*i.e.* each P_i is a singleton), that means PARTITION is not able to further aggregate the elements of $\mathcal{Q}$ (for instance, when called on 1 or 2 sets of organisms), then it traces edges between the node $\mathcal{Q}$ and those corresponding to the sets Q_i.

At the very beginning, the set $\mathcal{Q}$ is made of all singletons corresponding to the leaves, *i.e.*, the single organisms in $\mathcal{S}$, so $\mathcal{Q} = \{\{S_1\}, \ldots, \{S_\ell\}\}$. The set $\mathcal{Q}$ is inserted into a queue. The algorithm will iteratively process elements of the queue and will generate new elements for the queue, as described above (*i.e.*, one with the set $\mathcal{P}$ of parts, and one for each P_i).

The pseudocode in Algorithm 1 describes the method in detail. An example run is provided in Figure 2.

3.2 Inner Partitioning algorithm

In this subsection, we describe the approach we use as inner function within the algorithm described in Subsection 3.1 in order to obtain a partition $\mathcal{P} = \{P_1, \ldots, P_p\}$ of disjoint subsets from $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$, where $q \geq p > 1$.

According to Subsection 3.1, the set $\mathcal{Q}$ to be partitioned comprises some (not necessarily all) organisms, which are distributed among $Q_1, \ldots, Q_q$.

Before describing the inner partitioning procedure, we introduce some notation.

We recall that the set $\mathcal{S}$ contains ℓ sets of strings, as for each organism we can have multiple strings (like reads, contigs, a genome, and so on). So, $\mathcal{S} = \{S_1, S_2, \ldots, S_\ell\}$ and each set $S_i \in \mathcal{S}$ contains m_i strings, *i.e.* $S_i = \{s_{i,1}, \ldots, s_{i,m_i}\}$. Note that the definitions of eBWT, LCP and DA apply also to this case, *e.g.* $\mathsf{ebwt}(\mathcal{S}) = \mathsf{ebwt}(\{S_1, S_2, \ldots, S_\ell\}) = \mathsf{ebwt}(\{s_{1,1}, \ldots, s_{1,m_1}, \ldots, s_{\ell,1}, \ldots, s_{\ell,m_\ell}\})$, $\mathsf{lcp}(\mathcal{S}) = \mathsf{lcp}(\{S_1, S_2, \ldots, S_\ell\}) = \mathsf{lcp}(\{s_{1,1}, \ldots, s_{\ell,m_\ell}\})$ and $\mathsf{da}(\mathcal{S}) = \mathsf{da}(\{S_1, S_2, \ldots, S_\ell\}) = \mathsf{da}(\{s_{1,1}, \ldots, s_{\ell,m_\ell}\})$. These data structures are well known in string processing, so their construction is outside the scope of this paper (see Section 4 for a collection of methods and tools to construct them).

We extend the above, commonly used, notion of DA to *Color Document Array* (CDA), where $\mathsf{cda}(\mathcal{S})[i] = r$ if $\mathsf{da}(\mathcal{S})[i] = u$ and s_u belongs to the set S_r. In other words, we assign a same color to the strings belonging to the same set S_r, so we have a distinct color r for each set $S_r \in \mathcal{S}$.

▶ **Example 5** (running example)**.** In Figure 1, cda coincides with da assuming that each organism is a single string.

Let $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$, where each Q_i represents a node of the phylogeny tree, and thus, it identifies a non-empty set of organisms. We define a map $\chi_{\mathcal{Q}}$ that associates any organism to the element of $\mathcal{Q}$ to which it belongs (if there exists).

▶ **Definition 6.** *Given* $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$*, we define* $\chi_{\mathcal{Q}}$ *from* $\{1, \ldots, \ell\}$ *to* $\{Q_1, \ldots, Q_q\} \cup \{\emptyset\}$*, such that*

$$\chi_{\mathcal{Q}}(r) = \begin{cases} Q_s & \text{if there exists } Q_s \in \mathcal{Q} \text{ s.t. } r \text{ belongs to } Q_s \\ \emptyset & \text{otherwise.} \end{cases}$$

Recall that we denote by eBWTclust$[i, j]$ the concatenation of the symbols in the eBWT associated with the range $[i, j]$ (*i.e.* $\mathsf{ebwt}(\mathcal{S})[i, j]$), where $[i, j]$ is a positional cluster, unless otherwise specified.

Then, for each eBWTclust$[i, j]$, the corresponding interval in the CDA, $\mathsf{cda}(\mathcal{S})[i, j]$, determines the organisms (or colors) to which the symbols in eBWTclust$[i, j]$ belong.

▶ **Definition 7.** *An* eBWTclust$[i, j]$ is $\gamma_{\mathcal{Q}}$*-colored* if $\gamma_{\mathcal{Q}}$ is the set of elements of $\mathcal{Q}$ appearing in $\mathsf{cda}(\mathcal{S})[i, j]$, *i.e.* $\gamma_{\mathcal{Q}} = \{\chi_{\mathcal{Q}}(r) : r \in \mathsf{cda}(\mathcal{S})[i, j]\}$.

Note that if eBWTclust and CDA are restricted to the strings in $\mathcal{Q}$ (see Remark 1), then $\gamma_{\mathcal{Q}}$ contains only non-empty sets.

▶ **Example 8.** Let $\mathcal{Q} = \{\{S_1, S_3, S_4\}, \{S_2\}, \{S_5\}\}$ and eBWTclust$[i, j] = ACAAGT$ with $\mathsf{cda}[i, j] = [1\ 2\ 1\ 1\ 3\ 4]$. Then, eBWTclust$[i, j]$ is $\gamma_{\mathcal{Q}}$-colored and $\gamma_{\mathcal{Q}} = \{\{S_1, S_3, S_4\}, \{S_2\}\}$.

The main idea is to detect and analyze only eBWT positional clusters associated with left-maximal contexts shared by some Q_i:

▶ **Definition 9.** *A* $\gamma_{\mathcal{Q}}$*-colored* eBWTclust$[i, j]$ is *relevant*, if $\mathsf{ebwt}[i, j]$ is not a concatenation of a same symbol (*i.e.* it is not a run) and $1 < \mathsf{card}(\gamma_{\mathcal{Q}}) < q$ holds.

▶ **Example 10** (running example)**.** We highlight in bold, in Figure 1, the relevant eBWTclust, that are eBWTclust$[11, 14]$ and eBWTclust$[22, 23]$. Every other eBWTclust is either a run of a same symbol or the associated cda contains only one color or all of them.

Now, we use the notion of relevant eBWTclust to obtain a partition $\mathcal{P} = \{P_1, \ldots, P_p\}$ of disjoint subsets from $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$, with $q \geq p > 1$. Recall that $p > 1$, because the trivial partition $\mathcal{P} = \{\mathcal{Q}\}$ provides no significant information. The partitioning strategy is summarized in the following three phases:

1. we scan our data structures computed on $\mathcal{S}$, and we detect the relevant eBWTclust of $\text{eBWT}(\mathcal{S})|_Q$ (denoted by eBWTclust$_{\mathcal{Q}}[i, j]$, for some $i < j$);
2. we analyze each eBWTclust$_{\mathcal{Q}}[i, j]$ in order to build a list $\mathfrak{L}(\mathcal{Q})$ of subsets of $\mathcal{Q}$, called *candidate parts*, and we incrementally assign a score to each candidate part;
3. we build $\mathcal{P}$ from the list $\mathfrak{L}(\mathcal{Q})$ by selecting *compatible* candidates with the highest score.

Note that, by using Remark 1, it is easy to verify that the relevant eBWTclust$_{\mathcal{Q}}$ of the first step can be obtained by a linear scan of the input data structures.

At step 2), while analyzing any relevant eBWTclust$_{\mathcal{Q}}[i,j]$, we require that any element Q_s in $\gamma_{\mathcal{Q}}$ is a representative, *i.e.* the number of colors of Q_s appearing in $[i,j]$ is sufficiently large. We denote by τ $(0 < \tau \leq 1)$ such support threshold that determines the minimum required portion for each $Q_s \in \gamma_{\mathcal{Q}}$. Intuitively, the support threshold guarantees that all the elements of $\mathcal{Q}$ appearing in the eBWTclust are sufficiently represented. In fact, when the support threshold approaches 1, all the elements of the subset Q_s considered are required to be in the cluster. In other words, we aim at measuring how similar the shared history of the phylogeny is in terms of shared substrings. On the other hand, when the support threshold approaches 0, at least one of the elements of the subset Q_s is required to be in the cluster considered. Thus, we are observing how similar all the evolution events are, providing two different viewpoints of their phylogeny. More formally,

▶ **Definition 11.** *Let* $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ *and* eBWTclust$_{\mathcal{Q}}[i,j]$ be relevant. Given a support threshold value τ in $(0,1]$, we define the *candidate part of* eBWTclust$_{\mathcal{Q}}[i,j]$, and we denote it by $L_{[i,j]}$, the set $\gamma_{\mathcal{Q}}$ only if it holds $\mathsf{card}(\mathsf{cda}[i,j] \cap Q_s) \geq \tau \cdot \mathsf{card}(Q_s)$, for all $Q_s \in \gamma_{\mathcal{Q}}$.

In general, any relevant $\gamma_{\mathcal{Q}}$-colored eBWTclust$_{\mathcal{Q}}[i,j]$ may not have an associated candidate part $L_{[i,j]}$. That is the case in which some $Q_s \in \gamma_{\mathcal{Q}}$ are not sufficiently represented in the interval $[i,j]$.

The list $\mathfrak{L}(\mathcal{Q})$ of all candidate parts $L_{[i,j]} \subset \mathcal{Q}$ is built up by analyzing all eBWTclust$_{\mathcal{Q}}$. Any eBWTclust$_{\mathcal{Q}}[i,j]$ contributes to the score of its candidate part $L_{[i,j]} \subset \mathcal{Q}$ by the minimum value in $\mathsf{lcp}[i+1,j]$. So, the elements of the list $\mathfrak{L}(\mathcal{Q})$ appear as pairs (L,y), where L is a proper subset of $\mathcal{Q}$ and y its associated score incrementally obtained.

Intuitively, we use the score to determine the order in which the candidate parts must be taken into account to build $\mathcal{P}$. Since $\mathcal{P}$ is a partition of $\mathcal{Q}$, we cannot take all the candidate parts with a high score, but we must select, step by step, only those that are somehow compatible with each other. In fact, by partition definition, if X is a part of $\mathcal{P}$ and Y has non-empty intersection with X, then Y cannot be a part of $\mathcal{P}$.

▶ **Example 12** (running example)**.** Let us consider the first call of PARTITION for the toy example in Figure 1, where $\mathcal{Q} = \{Q_1, Q_2, Q_3\}$ and $Q_i = \{S_i\}$. Let τ be any value in $(0,1]$, eBWTclust$[11,14]$ contributes to increase the score of its candidate part $\{Q_1, Q_3\}$ by 1 (being the minimum lcp-value in $[12,14]$), while eBWTclust$[22,23]$ contributes to increase the score of its candidate part $\{Q_1, Q_2\}$ by 6. So, at the end of the cluster analysis, we have that $\mathcal{L}(\mathcal{Q})$ contains $(\{Q_1, Q_2\}, 6)$, $(\{Q_1, Q_3\}, 1)$. The output partition $\mathcal{P}$ is $\{\{Q_1, Q_2\}, \{Q_3\}\}$, as $\{Q_1, Q_2\}$ has a higher score than $\{Q_1, Q_3\}$.

We use a greedy algorithm to select a list (denoted by L_C) of compatible candidate parts that will constitute the parts of the output partition $\mathcal{P}$.

In our selecting procedure, we choose to consider only elements of $\mathfrak{L}(\mathcal{Q})$ having high scores. In particular, we denote by $\bar{t}$ the number of elements with the highest scores such that the difference with the previous higher score does not form a local minimum and at least t highest scores are considered.

In addition to the list L_C of compatible subsets, during the scan of such $\bar{t}$ subsets with the highest scores, we build a second list (denoted by L_E) of *compatible extensions* of L_C.

Both lists L_C and L_E are initially empty. A subset V is compatible, and we add it to L_C, if it is disjoint from all the elements already in L_C, and moreover, if either it has empty intersection with all the elements in the list L_E, or it is strictly contained in the first element with non-empty intersection. We define a set V compatible extension, and we add it to L_E, if it is a superset of all the elements already in L_C.

Intuitively, the list L_E contains those subsets that warn us from selecting next compatible subsets that separate its elements. In fact, the order in which subsets are processed is given by their score, so when checking if V is compatible, all the elements already in L_C and in L_E show a score higher than V.

The output partition $\mathcal{P}$ contains all the compatible subsets in L_C as parts, and any other element of $\mathcal{Q}$ not appearing in any subset in L_C, as a singleton part.

3.3 Complexity

Partitioning. We observe here how the partitioning procedure described in Subsection 3.2 can be computed in $O(N)$ time and space, where N corresponds to the sum of lengths of all strings within the organisms in $\mathcal{Q}$. Indeed, the eBWT can be computed in linear time in its length, which is N, including the identification of all positional clusters [28]. Given an element of a eBWTCLUST$[i, j]$ and τ, we can determine in $O(1)$ time its color using the CDA; as we can pre-compute the size of all Q_i, this lets us easily determine the $\gamma_{\mathcal{Q}}$-coloring of the cluster and the associated $L_{[i,j]}$ (Definitions 7 and 11) in time proportional to the cluster's length, for a total cost of $O(N)$ over the whole BWT to obtain $\mathcal{L}(\mathcal{Q})$.

While potentially there could be up to $2^q \leq 2^\ell$ candidate parts $L_{[i,j]}$, observe that each positional cluster can in fact define at most one candidate, of size not greater than the length of the cluster; it follows that $\mathcal{L}(\mathcal{Q})$ has $< N$ elements, and the sum of sizes of all $L_{[i,j]}$ is too at most N.

Next, the algorithm sorts $\mathcal{L}(\mathcal{Q})$ by score, which using a bucket sort takes $O(N)$ time. Finally, we need to scan $\mathcal{L}(\mathcal{Q})$ to obtain L_C: using bit-vectors (or other standard data structures) we can keep track in constant time of which Q_i have been added to L_C or to L_E; thus we can check if an $L_{[i,j]}$ is compatible (or if it generates a compatible extension) in time proportional to its size, meaning the total cost of this step is once more $O(N)$, and so are the running time and space requirements of the partitioning procedure.

Tree reconstruction. While we omit a more detailed analysis, it is relatively straightforward to see that each $\mathcal{Q}$ placed in the queue corresponds to one node of the tree (see Line 10 in Algorithm 1).

Since PARTITION always splits the input in at least two parts, the number of internal nodes are at most the number of leaves (ℓ); as such, the complexity is bounded by ℓ times the cost of PARTITION. As described above, the latter is bound by the sum of lengths of the strings in the $\mathcal{Q}$ PARTITION is called upon, which is at most the size of the input $O(N)$ (although this is a worst case). As for the space requirement, it is that of PARTITION plus the maximum size of the queue: the queue holds up to $O(\ell)$ elements (one for each node of the tree), and each has size at most ℓ. The following holds:

▶ **Lemma 13.** *Given a set $\mathcal{S}$ of ℓ organisms, whose total length is N, phyBWT reconstructs a phylogenetic tree for $\mathcal{S}$ in $O(N\ell)$ time and $O(N + \ell^2)$ space.*

We observe that N is the dominant factor in this complexity, as the length of the strings representing a taxon is -in known applications- many orders of magnitude greater than the number ℓ of taxa. Moreover, letting $n = N/\ell$ be the average length of a taxon, the time cost $O(N\ell)$ can be equivalently seen as $O(n\ell^2)$, so quadratic in the number of taxa.

4 Preliminary experiments

In this section we assess our partitioning-based method, phyBWT, for reconstructing phylogenetic trees from short-reads and *de novo* assembled sequences. Indeed, phyBWT is not limited to a particular type of input, and it is able to manage both types of data. However, the diversity in the type of input data needs a tuning of the parameters described in the method section.

In the literature, features similar to phyBWT are shared by the tool SANS [38, 31] which, as our tool, is an alignment- and reference-free approach that is whole-genome based, and in addition, it does not produce a pairwise comparison of the sequences or their characteristics. Differently from phyBWT, SANS is based on the computation of all k-mers, which are either directly extracted or read in a colored de Bruijn graph, and then used to build a list of splits. The first implementation of SANS [38] post-processed the list of splits according to two filtering strategies that are described in the software tool SplitsTree [20]: (i) a greedy weakly approach that allows to display the output as a network, and (ii) a greedy tree approach that displays the output as a tree. In fact, according to the phylogenetic splits model [3], the reconstructed phylogenies are mesh-like graph and they are not restricted to trees.

The latest version[3] of the tool SANS [31] is a stand-alone re-implementation that introduces some new features and improves the runtime and the memory usage. It has mainly three filtering options that allow to limit the output splits in order to reduce the complexity of the network or calculate a subset of the splits representing a tree.

We show experiments carried out by such new version of the tool SANS by applying the filtering approach for drawing trees.

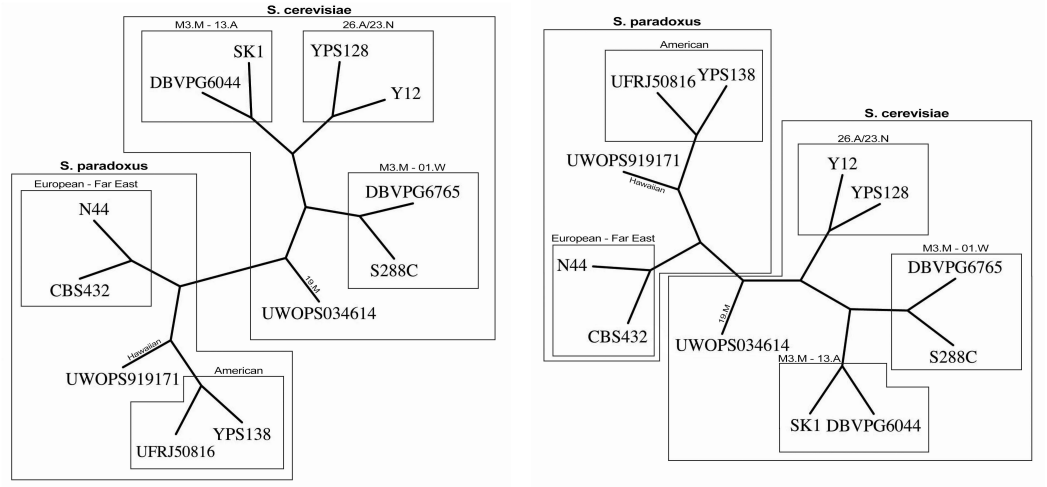
Drawing phylogeny tree. Our tool reconstructs a tree by means of partitions and outputs can be visualized by using well-known existing tools. In this paper, all the trees are drawn by using the PHYLIP package by Joe Felsenstein[4] and manually annotated.

Data Structure building. We observe that we take in input the following data structures eBWT, LCP and DA that can be built independently from our tool, *e.g.* [5, 10, 13, 7, 23, 30, 8]. One also could build them for each S_i, then, one could merge them to get the data structures of the union of some S_i.

Datasets. To show the effectiveness of our method, we have chosen three datasets with a similar number of organisms but with a diverse composition and different length of the strings. More in details, we used three different types of datasets: i) Illumina sequencing data (short reads) for seven *S. cerevisiae* and five *S. paradoxus* strains from the study in [41]; ii) assemblies from 12 species of the genus *Drosophila* from the FlyBase database (largely accepted phylogeny shown in [11]); iii) viral complete genomes from the *Prasinovirus* genus (benchmark phylogeny trees reported in [15]). The datasets ii) and iii) are also analyzed in [38].

Running time. While our implementation of phyBWT is not yet optimized, we observe that the running time of our prototype is dominated by the cost of computing eBWT, LCP and DA. When the latter cost is stripped down, the running time of phyBWT is lower compared

[3] `https://gitlab.ub.uni-bielefeld.de/gi/sans`
[4] `https://evolution.genetics.washington.edu/phylip/`, version 3.698 for 64-bit Windows systems.



(a) Tree by phyBWT. **(b)** Tree by SANS.

Figure 3 Yeasts phylogeny by phyBWT (a) and by SANS (b).

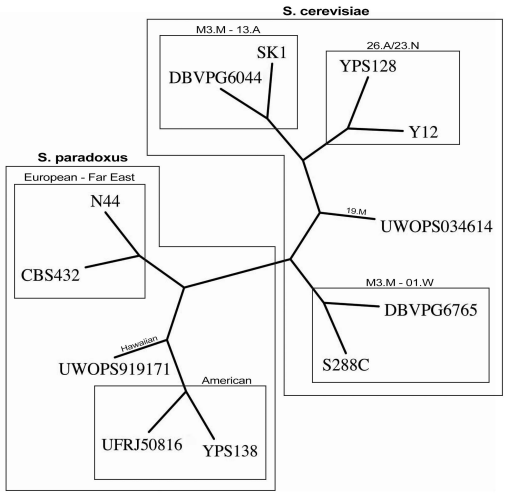
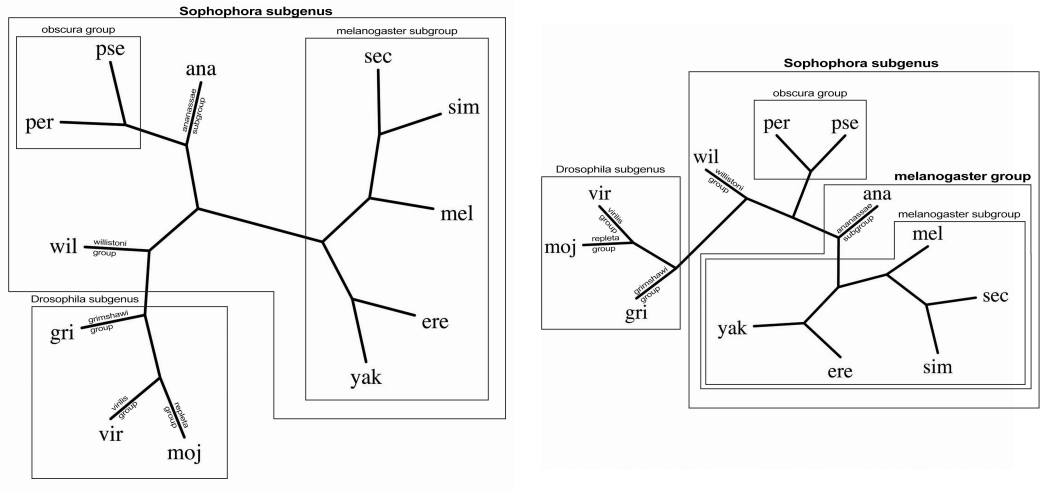


Figure 4 Benchmark phylogeny for the yeasts dataset. Figure redrawn from [41].

to that of SANS. However, as it is often the case, eBWT, LCP and DA (or a subset of them) might be already available from some other applications. For example, for the medium-size Drosophila dataset, phyBWT infers the phylogenetic tree in 2 minutes by using 25 GB of internal memory, and SANS runs in 26 minutes using 28 GB of internal memory.

4.1 Yeasts dataset

This dataset comprises 12 Illumina sequencing experiments obtained from the study in [41], and deposited in the public repository SRA (Short Reads Archive) under accession code PRJNA340312. The 12 datasets from [41] include seven sequencing experiments for the *S. cerevisiae* strains and five for the *S. paradoxus* strains. According to [41], for each sequencing experiment, comprising 151-bp paired-end reads, we performed adaptor-removing and quality-based trimming using trimmomatic [6]. For each sample, we extracted 5 million of 151-bp paired-end reads to form the yeasts dataset with a total size of 26 GB.



(a) Tree by phyBWT **(b)** Benchmark phylogeny. Figure redrawn from [11].

Figure 5 Drosophila phylogeny: (a) by our method; (b) benchmark redrawn from[11].

We ran both phyBWT and SANS on such short reads dataset. The tree depicted in Figure 3a is produced by phyBWT for any $k_m > 13$, $\tau = 0.6$ and $t = 12$ (as the number of taxa). For SANS, we used default parameters that corresponds to a k-mer length of 31, and we set the parameter `-f strict` to output a tree in the Newick format (see Figure 3b).

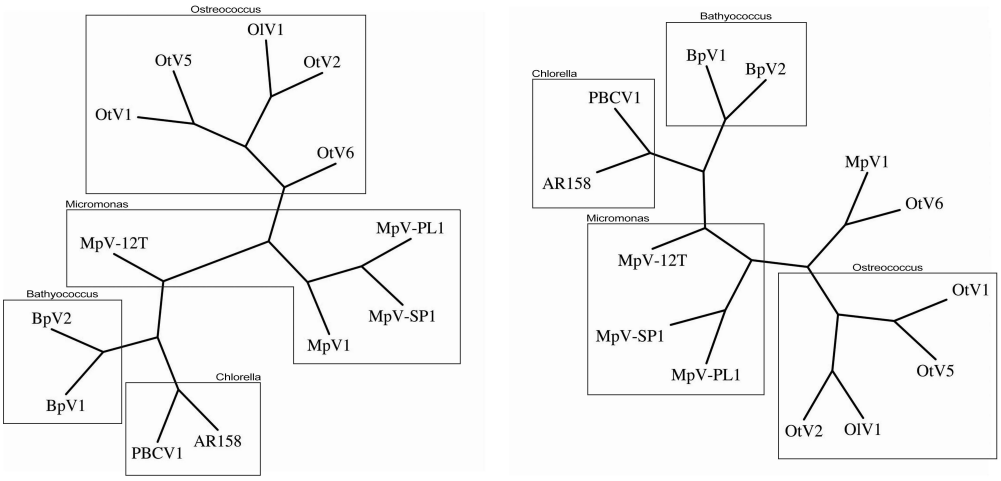
The benchmark tree reported in Figure 4 is the one obtained in the original study [41]. Remarkably, the benchmark was built using nuclear one-to-one orthologs, i.e. blocks of nuclear genes which are shared among (1) the seven *S. cerevisiae*, (3) the five *S. paradoxus* strains sequenced in the study, and (3) six outgroups from the *Saccharomyces* genus.

Both phyBWT and SANS correctly group the *S. cerevisiae* and the *S. paradoxus* strains which show an average whole-genome sequence divergence of $\sim 10\%$. As expected by taking into account the relatively high divergence among *S. paradoxus* strains (0.5% - 4.5%), also the same *S. paradoxus* partition is obtained. On the other hand, a few differences are shown in the *S. cerevisiae* partition which groups strains with a sequence divergence $\sim 0.5\%$. Compared to SANS, phyBWT produces a tree which is closer to the benchmark although the differences with the benchmark shown by both SANS and our method can be explained considering the relatively low divergence among *S. cereviasiae strains* as well as the partially admixed genomes of some of the trains (e.g. S288C and DBVPG6044) [27].

4.2 Drosophila dataset

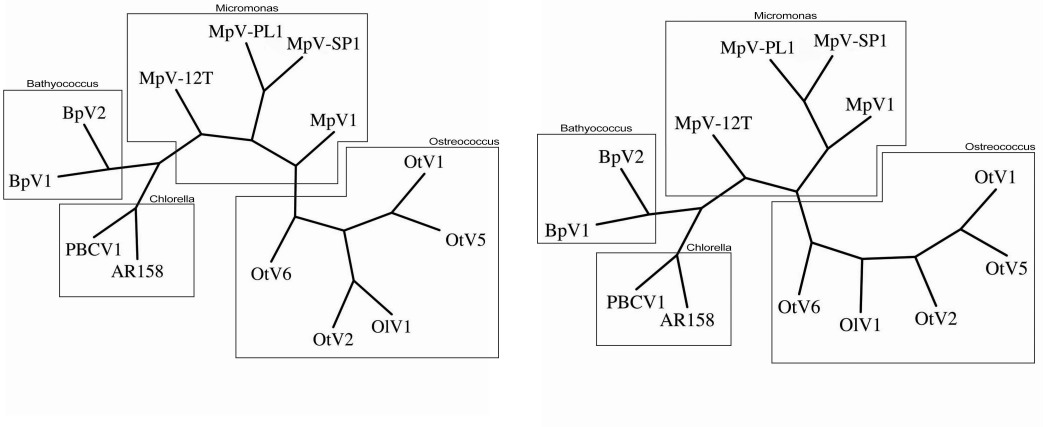
Drosophila data are downloaded from the FlyBase database (`http://flybase.org/`). This dataset includes assemblies from 12 species of the genus *Drosophila*: *D. melanogaster* (mel), *D. ananassae* (ana), *D. erecta* (ere), *D. grimshawi* (gri), *D. mojavensis* (moj), *D. persimilis* (per), *D. pseudoobscura* (pse), *D. sechellia* (sec), *D. simulans* (sim), *D. virilis* (vir), *D. willistoni* (wil), and *D. yakuba* (yak). Nine of these species fall within the Sophophora subgenus, which includes members of the melanogaster, obscura and willistoni groups.

The number of strings for each species varies: it ranges from $1,870$ for *D. melanogaster* to $17,440$ for *D. grimshawi*. The obtained dataset is a medium-sized input with a total number of symbols of more than $2,161$ Mbp.



(a) Tree by phyBWT. **(b)** Tree by SANS.

Figure 6 *Prasinovirus* by phyBWT and by SANS.



(a) Redrawn reference tree [15, Figure 3]. **(b)** Redrawn reference tree [15, Figure 4].

Figure 7 Benchmark phylogeny for *Prasinovirus* dataset. Figures redrawn from [15].

phyBWT produces the tree depicted in Figure 5a for any k_m in $[23, 72]$, $\tau = 0.5$ and $t = 12$ (as the number of taxa). The *Sophophora* subgenus as well as the *Drosophila* subgenus are correctly detected, and inside the *Sophophora* subgenus, the *melanogaster* subgroup is correctly isolated. The only difference with respect to the benchmark tree by [11] is the organism *D. ananassae* that represents the *ananassae* subgroup. Such subgroup is part of the *melanogaster* group together with *D. melanogaster*, *D. sechellia*, *D. simulans*, *D. erecta* and *D. yakuba*. However, our method places *D. ananassae* closer to the *obscura* group rather than the *melanogaster* subgroup. SANS was run with default values as described in [38]. The output tree obtained by setting `-f strict` is topological equivalent to the benchmark reference tree in [11], which has reported in Figure 5b for completeness.

4.3 *Prasinovirus* dataset

This dataset comprises 13 genomes from the viral genus *Prasinovirus* studied in [15] and infecting the genera *Ostreococcus* (OtV1, OtV2, OtV5, OtV6, OlV1), *Bathyococcus* (BpV1, BpV2), *Micromonas* (MpV1, MpV-12T, MpV-PL1, MpV-SP1), and *Chlorella* (PBCV1, AR158).

Phylogenetic reconstruction of viral genomes is challenging due to their small genome size and the high variability of their genome content.

In [15], the authors reported two different phylogenetic trees: one is based on the presence/absence of shared putative genes [15, Figure 3] and the other is a maximum likelihood estimation based on a marker gene (DNA polymerase B) [15, Figure 4]. The two benchmark trees are depicted in Figure 7.

The complete prasinovirus genomes used in this dataset are 213 Kbp on average: in particular, the genome sizes range from 173,350 bp for MpV-SP1 to 205,622 bp for MpV-12T.

phyBWT produces the tree depicted in Figure 6a for $k_m = 13$, $\tau = 0.5$ and $t = 13$ (as the number of the taxa), while Figure 6b depicts the output tree by SANS generated by using parameters `-k 11 -t 130 -f strict` (recommended parameter for such dataset, see `https://gitlab.ub.uni-bielefeld.de/gi/sans`).

The only main difference of the tree produced by SANS with respect to the benchmark trees is that the former shows a subtree with MpV1 and OtV6 as leaves, although MpV1 belongs to Micromonas group and OtV6 belongs to Ostreococcus group.

5 Conclusions and discussion

In this paper, we proposed a new alignment-, assembly- and reference-free partition-based method to build the phylogeny inference of a set of organisms.

To the best of our knowledge, phyBWT is the first method that applies the properties of the Extended Burrows-Wheeler Transform (eBWT) to the idea of decomposition for phylogenetic inference. Our approach is based on the eBWT positional cluster framework introduced in [29], which allowed us to consider longest shared substrings of varying length, unlike k-mer-based approaches such as SANS. We introduce the inner partitioning algorithm based on the eBWT positional cluster, and employ it as a black-box in our novel tree reconstruction algorithm. Specifically, phyBWT does not start from the leaves to group them together in a bottom-up fashion; it does not start from the root and performs a top-down partition; instead, it proceeds in both directions (bottom-up and top-down), according to what is returned by the inner partitioning algorithm.

We tested our method on three sequencing datasets, with short reads and *de novo* assembled sequences. The experimental results show that our algorithm produces trees comparable to the benchmark phylogeny and to the newly introduced tool SANS. We plan to perform algorithm engineering of phyBWT that will better exploit the bounded length of the reads to overcome the computational bottleneck of computing the eBWT.

While the worst-case complexity of the method is competitive with existing methods, there are interesting directions for further optimization, such as using Colored Range Queries [16] to speed up identification of colors in the various clusters, or exploiting the natural predisposition of the method for parallel computation. Our current prototype requires a preprocessing in order to compute some data structures that are at the heart of several text and string algorithms. More efficient tools for computing them can appear in the literature, and we plan to investigate whether the required information can be reduced for instances of our problem.

References

1 M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0.

2 H.-J. Bandelt and A. W. M. Dress. A canonical decomposition theory for metrics on a finite set. *Advances in mathematics*, 92(1):47–105, 1992.

3 H.-J. Bandelt and A. W. M. Dress. Split decomposition: A new and useful approach to phylogenetic analysis of distance data. *Molecular Phylogenetics and Evolution*, 1(3):242–252, 1992. doi:10.1016/1055-7903(92)90021-8.

4 H.-J. Bandelt, K. T. Huber, J. H. Koolen, V. Moulton, and A. Spillner. *Basic Phylogenetic Combinatorics*. Cambridge University Press, 2012. URL: http://www.cambridge.org/de/knowledge/isbn/item6439332/.

5 M.J. Bauer, A.J. Cox, and G. Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483(0):134–148, 2013. doi:10.1016/j.tcs.2012.02.002.

6 A. M Bolger, M. Lohse, and B. Usadel. Trimmomatic: a flexible trimmer for illumina sequence data. *Bioinformatics*, 30(15):2114–20, 2014. doi:10.1093/bioinformatics/btu170.

7 P. Bonizzoni, G. Della Vedova, Y. Pirola, M. Previtali, and R. Rizzi. Multithread Multistring Burrows-Wheeler Transform and Longest Common Prefix Array. *Journal of computational biology*, 26(9):948–961, 2019. doi:10.1089/cmb.2018.0230.

8 C. Boucher, D. Cenzato, Z. Lipták, M. Rossi, and M. Sciortino. Computing the original ebwt faster, simpler, and with less memory. In *SPIRE*, pages 129–142. Springer International Publishing, 2021.

9 M. Burrows and D.J. Wheeler. A Block Sorting data Compression Algorithm. Technical report, DIGITAL System Research Center, 1994.

10 A.J. Cox, F. Garofalo, G. Rosone, and M. Sciortino. Lightweight LCP construction for very large collections of strings. *J. Discrete Algorithms*, 37:17–33, 2016. doi:10.1016/j.jda.2016.03.003.

11 M. A. Crosby, J. L. Goodman, V. B. Strelets, P. Zhang, W. M. Gelbart, and The FlyBase Consortium. FlyBase: genomes by the dozen. *Nucleic Acids Research*, 35(suppl.1):D486–D491, November 2006.

12 M. D'Angiolo, M. De Chiara, J.-X. Yue, A. Irizar, S. Stenberg, K. Persson, A. Llored, B. Barré, J. Schacherer, R. Marangoni, E. Gilson, J. Warringer, and G. Liti. A yeast living ancestor reveals the origin of genomic introgressions. *Nature*, 587(7834):420–425, November 2020.

13 L. Egidi, F. A. Louza, G. Manzini, and G. P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms for Molecular Biology*, 14(1):6:1–6:15, 2019. doi:10.1186/s13015-019-0140-0.

14 H. Fan, A. R Ives, Y. Surget-Groba, and C. H Cannon. An assembly and alignment-free method of phylogeny reconstruction from next-generation sequencing data. *BMC genomics*, 16(1):1–18, 2015.

15 J. F. Finke, D. M. Winget, A. M. Chan, and C. Suttle. Variation in the genetic repertoire of viruses infecting micromonas pusilla reflects horizontal gene transfer and links to their environmental distribution. *Viruses*, 9, 2017.

16 Travis Gagie, Juha Kärkkäinen, Gonzalo Navarro, and Simon J. Puglisi. Colored range queries and document retrieval. *Theoretical Computer Science*, 483:36–50, 2013. doi:10.1016/j.tcs.2012.08.004.

17 B. Gallone, J. Steensels, S. Mertens, M. C. Dzialo, J. L. Gordon, R. Wauters, F. A. Theßeling, F. Bellinazzo, V. Saels, B. Herrera-Malaver, T. Prahl, C. White, M. Hutzler, F. Meußdoerffer, P. Malcorps, B. Souffriau, L. Daenen, G. Baele, S. Maere, and K. J. Verstrepen. Interspecific hybridization facilitates niche adaptation in beer yeast. *Nat Ecol Evol*, 3(11):1562–1575, November 2019.

18 V. Guerrini., F. Louza., and G. Rosone. Lossy Compressor Preserving Variant Calling through Extended BWT. In *BIOSTEC/BIOINFORMATICS*, pages 38–48. INSTICC, SciTePress, 2022. doi:10.5220/0010834100003123.

19 V. Guerrini, F.A. Louza, and G. Rosone. Metagenomic analysis through the extended Burrows-Wheeler transform. *BMC Bioinformatics*, 21, 2020. doi:10.1186/s12859-020-03628-w.

20 D. H. Huson and D. Bryant. Application of Phylogenetic Networks in Evolutionary Studies. *Molecular Biology and Evolution*, 23(2):254–267, October 2005.

21 J. Jansson and W.-K. Sung. *Algorithms for Combining Rooted Triplets into a Galled Phylogenetic Network*, pages 48–52. Springer New York, New York, NY, 2016. doi:10.1007/978-1-4939-2864-4_92.

22 J. Jansson and W.-K. Sung. *Maximum Agreement Supertree*, pages 1224–1227. Springer New York, New York, NY, 2016. doi:10.1007/978-1-4939-2864-4_222.

23 F. A. Louza, Guilherme P. Telles, Simon Gog, Nicola Prezza, and G. Rosone. gsufsort: constructing suffix arrays, lcp arrays and bwts for string collections. *Algorithms for Molecular Biology*, 15, 2020.

24 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *ACM-SIAM SODA*, pages 319–327, 1990.

25 S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows-Wheeler Transform. *Theoret. Comput. Sci.*, 387(3):298–312, 2007.

26 S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. A new combinatorial approach to sequence comparison. *Theory Comput. Syst.*, 42(3):411–429, 2008. doi:10.1007/s00224-007-9078-6.

27 J. Peter, M. De Chiara, A. Friedrich, J.-X. Yue, D. Pflieger, A. Bergström, A. Sigwalt, B. Barre, K. Freel, A. Llored, C. Cruaud, K. Labadie, J.-M. Aury, B. Istace, K. Lebrigand, P. Barbry, S. Engelen, A. Lemainque, P. Wincker, and J. Schacherer. Genome evolution across 1,011 saccharomyces cerevisiae isolates. *Nature*, 556, April 2018. doi:10.1038/s41586-018-0030-5.

28 N. Prezza, N. Pisanti, M. Sciortino, and G. Rosone. SNPs detection by eBWT positional clustering. *Algorithms for Molecular Biology*, 14(1):3, 2019. doi:10.1186/s13015-019-0137-8.

29 N. Prezza, N. Pisanti, M. Sciortino, and G. Rosone. Variable-order reference-free variant discovery with the Burrows-Wheeler transform. *BMC Bioinformatics*, 21, 2020. doi:10.1186/s12859-020-03586-3.

30 N. Prezza and G. Rosone. Space-efficient construction of compressed suffix trees. *Theoretical Computer Science*, 852:138–156, 2021. doi:10.1016/j.tcs.2020.11.024.

31 A. Rempel and R. Wittler. SANS serif: alignment-free, whole-genome-based phylogenetic reconstruction. *Bioinformatics*, 37(24):4868–4870, 2021. doi:10.1093/bioinformatics/btab444.

32 N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular biology and evolution*, 4(4):406–425, 1987.

33 S. M Soucy, J. Huang, and J. P. Gogarten. Horizontal gene transfer: building the web of life. *Nat Rev Genet*, 16(8):472–82, August 2015.

34 L. Tattini, N. Tellini, S. Mozzachiodi, M. D'Angiolo, S. Loeillet, A. Nicolas, and G. Liti. Accurate tracking of the mutational landscape of diploid hybrid genomes. *Mol Biol Evol*, August 2019.

35 S. Vinga. Alignment-free methods in computational biology, 2014.

36 S. Vinga and J. Almeida. Alignment-free sequence comparison—a review. *Bioinformatics*, 19(4):513–523, 2003. doi:10.1093/bioinformatics/btg005.

37 T. Warnow. *Computational Phylogenetics: An Introduction to Designing Methods for Phylogeny Estimation*. Cambridge University Press, 2017.

38 R. Wittler. Alignment- and Reference-Free Phylogenomics with Colored de Bruijn Graphs. In *19th International Workshop on Algorithms in Bioinformatics (WABI 2019)*, volume 143, pages 2:1–2:14, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.WABI.2019.2.

39 L. Yang, X. Zhang, and T. Wang. The Burrows-Wheeler similarity distribution between biological sequences based on Burrows-Wheeler transform. *Journal of Theoretical Biology*, 262(4):742–749, 2010. doi:10.1016/j.jtbi.2009.10.033.

40 Z. Yang and B. Rannala. Molecular phylogenetics: Principles and practice. *Nature reviews. Genetics*, 13:303–14, March 2012. doi:10.1038/nrg3186.

41 J.-X. Yue, J. Li, L. Aigrain, J. Hallin, K. Persson, K. Oliver, A. Bergström, P. Coupland, J. Warringer, M. C. Lagomarsino, G. Fischer, R. Durbin, and G. Liti. Contrasting evolutionary genome dynamics between domesticated and wild yeasts. *Nature Genetics*, 49(6):913–924, 2017. doi:10.1038/ng.3847.

42 A. Zielezinski, S. Vinga, J. Almeida, and W. Karlowski. Alignment-free sequence comparison: Benefits, applications, and tools. *Genome Biology*, 18:186, October 2017. doi:10.1186/s13059-017-1319-7.

Gene Orthology Inference via Large-Scale Rearrangements for Partially Assembled Genomes

Diego P. Rubert
Faculdade de Computação, Universidade Federal de Mato Grosso do Sul, Campo Grande, MS, Brasil
Faculty of Technology and Center for Biotechnology (CeBiTec), Bielefeld University, Germany

Marília D. V. Braga[1]
Faculty of Technology and Center for Biotechnology (CeBiTec), Bielefeld University, Germany

Abstract

Recently we developed a gene orthology inference tool based on genome rearrangements (*Journal of Bioinformatics and Computational Biology* 19:6, 2021). Given a set of genomes our method first computes all pairwise gene similarities. Then it runs pairwise ILP comparisons to compute optimal gene matchings, which minimize, by taking the similarities into account, the weighted rearrangement distance between the analyzed genomes (a problem that is NP-hard). The gene matchings are then integrated into gene families in the final step. Although the ILP is quite efficient and could conceptually analyze genomes that are not completely assembled but split in several contigs, our tool failed in completing that task. The main reason is that each ILP pairwise comparison includes an optimal *capping* that connects each end of a linear segment of one genome to an end of a linear segment in the other genome, producing an exponential increase of the search space.

In this work, we design and implement a heuristic capping algorithm that replaces the optimal capping by clustering (based on their gene content intersections) the linear segments into $m \geq 1$ subsets, whose ends are capped independently. Furthermore, in each subset, instead of allowing all possible connections, we let only the ends of content-related segments be connected. Although there is no guarantee that m is much bigger than one, and with the possible side effect of resulting in sub-optimal instead of optimal gene matchings, the heuristic works very well in practice, from both the speed performance and the quality of computed solutions. Our experiments on real data show that we can now efficiently analyze fruit fly genomes with unfinished assemblies distributed in hundreds or even thousands of contigs, obtaining orthologies that are more similar to FlyBase orthologies when compared to orthologies computed by other inference tools. Moreover, for complete assemblies the version with heuristic capping reports orthologies that are very similar to the orthologies computed by the optimal version of our tool. Our approach is implemented into a pipeline incorporating the pre-computation of gene similarities.

2012 ACM Subject Classification Applied computing → Bioinformatics

Keywords and phrases Comparative genomics, double-cut-and-join, indels, gene orthology

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.24

Supplementary Material Both the original version with optimal capping and the new modified version with heuristic capping can be downloaded from our GitLab server at:
Software (Source Code): `https://gitlab.ub.uni-bielefeld.de/gi/FFGC`

Acknowledgements We thank the anonymous reviewers for their valuable comments.

[1] Corresponding author

1 Introduction

The study of distances and parsimonious evolutionary scenarios based on large-scale genome rearrangements traditionally depends on the pre-computation of *gene families*. Computing such a distance is usually polynomial when genomes have at most one gene per family [4,7,13] or NP-hard otherwise [3,5,8,20,21]. These works adopt several rearrangement models and among the most popular ones is the double-cut-and-join (DCJ) operation [24], which mimics organizational rearrangements, such as inversions, fusions, fissions and translocations.

An alternative (NP-hard) *family-free* setting for genome rearrangement studies was proposed in 2013 [6] and further extended [16,19], in a model that does not require the pre-computation of gene families and, besides DCJ operations, takes into account insertions and deletions of DNA segments, collectively called *indels*. This model is able to infer pairwise orthologs between two genomes directly, simultaneously based on gene similarities and rearrangements. In practice, its optimization function can be solved exactly due to an ILP formulation [19] that is called FF-DCJ-Indel and also reports an optimal matching of orthologs between the two analyzed genomes. (The ILP FF-DCJ-Indel is itself based on the previous formulations for family-based approaches [5,21].)

With these achievements we were able to invert the traditional paradigm of genome rearrangement studies: instead of requiring the gene families to proceed with rearrangement comparisons, it became possible to use rearrangement comparisons for inferring the gene families[2]. Indeed, in our most recent work [18], we did a first attempt of using FF-DCJ-Indel for inferring genome-scale gene families across several species. More precisely, given a set of genomes, our method first computes all pairwise optimal gene matchings, which are integrated into gene families in the second step, resulting in a complete pipeline called DiffMGC, whose inferences displayed good quality in the analysis of completely assembled genomes.

However, although the integrated FF-DCJ-Indel is quite efficient and could conceptually analyze genomes that are not completely assembled but split in several contigs, it failed in completing that task. The main reason is that each ILP pairwise comparison includes an optimal *capping* that must allow the end of any linear segment of one genome to be matched to the end of any linear segment of the other genome. The optimal capping then produces an exponential increase of the search space.

In this work, we design and implement a heuristic capping algorithm that replaces the optimal capping by clustering (based on their gene content intersections) the linear segments into $m \geq 1$ subsets, so that the ends of the linear segments in the same subset S can only be matched to elements of S. Furthermore, in each subset, instead of allowing all possible connections, we let only the ends of content-related segments be connected. Although there is no guarantee that m is much bigger than one, and with the possible side effect of resulting in sub-optimal instead of optimal gene matchings, the heuristic works very well in practice, from both the speed performance and the quality of computed solutions.

We call DiffMGC$^{\approx}_{\text{H}}$ the new complete pipeline adopting the heuristic capping for FF-DCJ-Indel. Our experiments on real data show that we can now efficiently analyze fruit fly genomes with unfinished assemblies distributed in hundreds or even thousands of contigs.

[2] To be more precise, another attempt called MSOAR [22] was made before our studies, the differences being that MSOAR first infers gene families based on similarities and then computes a matching based on a heuristic including structural rearrangements and tandem duplications, while FF-DCJ-Indel takes similarities and rearrangements simultaneously into account for inferring an optimal matching, in a rearrangement model including DCJ and mimicking all content modifications with insertions and deletions of DNA segments. MSOAR was not maintained and is no longer operational, therefore we could never compare its performance to FF-DCJ-Indel.

We compared the gene families inferred by DiffMGC$\tilde{\tilde{\text{H}}}$ to Oma [10,17], ProteinOrtho [14], and Poff [15]. The orthologies inferred by DiffMGC$\tilde{\tilde{\text{H}}}$ are more similar to FlyBase orthologies when compared to orthologies computed by these other inference tools. Moreover, for complete assemblies DiffMGC$\tilde{\tilde{\text{H}}}$ reports orthologies that are very similar to the orthologies computed by DiffMGC, which is the optimal version of our tool.

2 Orthology inference via family-free genome rearrangements

For studying large-scale genome rearrangements a high-level view of a *chromosome* is adopted. In this view each chromosome is represented by a sequence of *genes*. Since each gene is an oriented DNA fragment, we need to distinguish its two possible representations: a gene g is represented by the symbol g itself, if it is read in direct orientation, or by the symbol $\overline{g}$, if it is read in reverse orientation. In our notation, all genes of a linear chromosome are concatenated in a string that can be read in any of the two directions and is flanked by square brackets. As an example, let $C = [\overline{6}1894\overline{4}]$ be a linear chromosome. A *genome* is then a set of chromosomes and can be transformed with the following types of mutations:

1. **Structural rearrangements (DCJ operations):** A *cut* performed on a chromosome C of a genome $\mathbb{A}$ separates two adjacent genes of C. A *double-cut and join* or *DCJ* applied on genome $\mathbb{A}$ is the operation that performs cuts in two different positions of distinct chromosomes or of the same chromosome of $\mathbb{A}$, creating four open ends, and joins these open ends in a different way [4, 24]. For example, let $\mathbb{A} = \{[\overline{6}1894\overline{4}], [3\overline{5}\overline{7}2]\}$, and consider a DCJ that cuts between genes 1 and 8 of its first chromosome and between genes 7 and 2 of its second chromosome, creating segments $\overline{6}1\bullet$, $\bullet 89\overline{4}$, $3\overline{5}\overline{7}\bullet$ and $\bullet 2$ (where the symbols $\bullet$ represent the open ends). If we join the first with the fourth and the second with the third open end, we get $\mathbb{A}' = \{[\overline{6}12], [3\overline{5}\overline{7}89\overline{4}]\}$, that is, the described DCJ operation is a translocation transforming $\mathbb{A}$ into $\mathbb{A}'$. Indeed, a DCJ operation can correspond not only to a translocation but to several structural rearrangements, such as an inversion, a fusion or a fission.
2. **Content-modifying (indel operations):** The content of a chromosome can be modified with *insertions* and with *deletions* of blocks of contiguous genes, collectively called *indel* operations. Note that at most one chromosome can be entirely deleted or inserted at once. As an example, consider the deletion of segment $\overline{7}89$ from chromosome $[3\overline{5}\overline{7}89\overline{4}]$, resulting in chromosome $[3\overline{5}\overline{4}]$. A gene cannot be deleted and then reinserted, nor inserted and then deleted. This restriction prevents the *free lunch* artifact of sorting one genome into the other by simply deleting the chromosomes of the first and inserting the chromosomes of the second, ignoring their common parts.

2.1 Computing an optimal set of orthologs between two genomes

We can represent the pairwise similarities between the genes of genome $\mathbb{A}$ and the genes of genome $\mathbb{B}$ in the so called *gene similarity graph* [6], denoted by $\mathcal{S}(\mathbb{A}, \mathbb{B})$. This is a weighted bipartite graph that has a vertex for each gene in genome $\mathbb{A}$ and a vertex for each gene in genome $\mathbb{B}$. Furthermore, for each pair of genes $g_1 \in \mathbb{A}, g_2 \in \mathbb{B}$, denote by $\sigma(g_1, g_2)$ their *normalized similarity*, a value that ranges in the interval $[0, 1]$. Given a threshold $0 \leq x \leq 1$, if $\sigma(g_1, g_2) \geq x$ there is an edge e connecting g_1 and g_2 in $\mathcal{S}(\mathbb{A}, \mathbb{B})$ whose weight is $w(e) = \sigma(g_1, g_2)$. In addition, to each vertex u of $\mathcal{S}(\mathbb{A}, \mathbb{B})$ we assign a weight $w(u)$ that can be obtained as follows: $w(u) = \max\{\sigma(uv) \mid uv \in \mathcal{S}(A, B)\}$, that is, $w(u)$ is the maximum similarity among the edges incident to the vertex (or gene) u in $\mathcal{S}(\mathbb{A}, \mathbb{B})$.

A matching M from $\mathcal{S}(\mathbb{A},\mathbb{B})$, here also called an *ortholog-set*, defines the tuple $(\mathbb{A},\mathbb{B},M)$, in which every two genes a,b, such that $a \in \mathbb{A}$, $b \in \mathbb{B}$ and $ab \in M$, are considered to be *orthologs*. The *complement* of M, denoted by $\widetilde{M}$, is the set composed of genes whose corresponding vertices in $\mathcal{S}(\mathbb{A},\mathbb{B})$ are M-unsaturated.

The DCJ-indel distance $\mathrm{d}^{\mathrm{ID}}_{\mathrm{DCJ}}(\mathbb{A},\mathbb{B},M)$ is the minimum number of DCJ and indel operations required to transform $\mathbb{A}$ into $\mathbb{B}$ assuming the orthologs given by M and allowing only the genes belonging to the complement $\widetilde{M}$ to be inserted or deleted. It can be computed using an approach relying on the cycles and paths of a graph that represents the structural relation between genomes $\mathbb{A}$ and $\mathbb{B}$ according to the ortholog-set M [7, 19] (this graph is equivalent to a consistent decomposition of the family-free relational graph, described in Section 2.2 and represented in Figure 1 (bottom)). Together with the weights of edges and vertices of $\mathcal{S}(\mathbb{A},\mathbb{B})$, the DCJ-indel distance $\mathrm{d}^{\mathrm{ID}}_{\mathrm{DCJ}}$ allows the computation of the weighted rearrangement distance $\mathrm{wd}^{\mathrm{ID}}_{\mathrm{DCJ}}$ [19]:

$$\mathrm{wd}^{\mathrm{ID}}_{\mathrm{DCJ}}(\mathbb{A},\mathbb{B},\mathcal{S},M) = \mathrm{d}^{\mathrm{ID}}_{\mathrm{DCJ}}(\mathbb{A},\mathbb{B},M) + |M| - w(M) + w(\widetilde{M}).$$

Then, given that $\mathfrak{M}$ is the set of all possible ortholog-sets in $\mathcal{S}(\mathbb{A},\mathbb{B})$, the rearrangement distance between $\mathbb{A}$ and $\mathbb{B}$ is the result of the following optimization:

$$\textsc{Diff}(\mathbb{A},\mathbb{B},\mathcal{S}) = \min_{M\in\mathfrak{M}} \{\mathrm{wd}^{\mathrm{ID}}_{\mathrm{DCJ}}(\mathbb{A},\mathbb{B},\mathcal{S},M)\}\,.$$

Figure 1 shows examples of ortholog-sets and their distances. Denote by $\textsc{DiffM}(\mathbb{A},\mathbb{B},\mathcal{S})$ an *optimal* ortholog-set in $\mathcal{S}(\mathbb{A},\mathbb{B})$, which is an ortholog-set whose rearrangement distance equals $\textsc{Diff}(\mathbb{A},\mathbb{B},\mathcal{S})$. Computing the rearrangement distance $\textsc{Diff}(\mathbb{A},\mathbb{B},\mathcal{S})$ and finding an optimal ortholog-set $\textsc{DiffM}(\mathbb{A},\mathbb{B},\mathcal{S})$ are NP-hard problems [19].

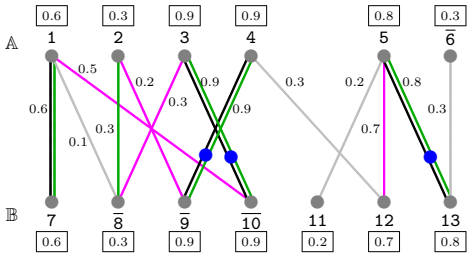
2.2 Family-free relational graph

One approach for solving the NP-hard problems $\textsc{Diff}(\mathbb{A},\mathbb{B},\mathcal{S})$ and $\textsc{DiffM}(\mathbb{A},\mathbb{B},\mathcal{S})$ is by decomposing the following graph.

The *family-free relational graph* $FFR(\mathbb{A},\mathbb{B},\mathcal{S})$, shown in Figure 1 (bottom), represents all possible weighted distances corresponding to all candidate ortholog-sets in $\mathcal{S}(\mathbb{A},\mathbb{B})$ [19]. Given a gene m, denote the extremities of m by m^h (*head*) and m^t (*tail*). The graph $FFR(\mathbb{A},\mathbb{B},\mathcal{S})$ has a set $V(\mathbb{A})$ with a vertex for each of the two extremities of each gene of genome $\mathbb{A}$ and a set $V(\mathbb{B})$ with a vertex for each of the two extremities of each gene of genome $\mathbb{B}$.

The set of edges is partitioned into several subsets:

- Sets $E^{\mathbb{A}}_{\mathrm{adj}}$ and $E^{\mathbb{B}}_{\mathrm{adj}}$ contain adjacency edges connecting adjacent extremities of genes in $\mathbb{A}$ and in $\mathbb{B}$.
- The set E_γ contains, for each edge $ab \in \mathcal{S}(\mathbb{A},\mathbb{B})$, an extremity edge connecting a^t to b^t, and an extremity edge connecting a^h to b^h. To both edges a^tb^t and a^hb^h, that are called *siblings*, we assign the same weight, which corresponds to the similarity of the edge ab in $\mathcal{S}(\mathbb{A},\mathbb{B})$: $w(a^tb^t) = w(a^hb^h) = \sigma(ab)$.
- Sets $E^{\mathbb{A}}_{\mathrm{id}}$ and $E^{\mathbb{B}}_{\mathrm{id}}$ contain indel edges connecting the two extremities of each gene in $\mathbb{A}$ and in $\mathbb{B}$. Each indel edge m^hm^t receives a weight $w(m^hm^t) = \max\{\sigma(mv)|mv \in \mathcal{S}(\mathbb{A},\mathbb{B})\}$, that is, it is the maximum similarity among the edges incident to the gene m in $\mathcal{S}(\mathbb{A},\mathbb{B})$.



`black` ⊂ `green` (black lines next to green lines)

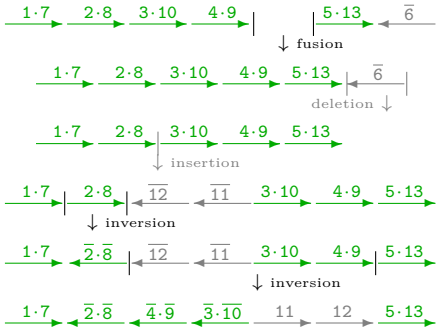
`blue` ⊂ `black` (blue dots on green/black lines)

Ranking of the represented ortholog-sets based on their corresponding distances

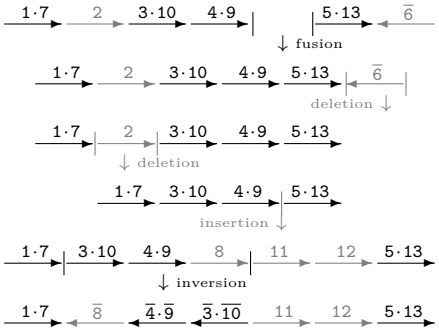
M	$\lvert M\rvert$	$w(M)$	$w(\widetilde{M})$	d_{DCJ}^{ID}	wd_{DCJ}^{ID}
`black`	4	3.2	1.8	5 *	7.6
`green`	5	3.5	1.2	5 *	7.7
`blue`	3	2.6	3.0	5	8.4
`magenta`	4	1.7	2.8	6	11.1

* Rearrangement scenarios are given below, with genes belonging to $\widetilde{M}$ as well as indel operations represented in gray

`green` ortholog-set ($d_{DCJ}^{ID} = 5$):



`black` ortholog-set ($d_{DCJ}^{ID} = 5$):



$FFR(\mathbb{A}, \mathbb{B}, \mathcal{S})$ and decomposition corresponding to the `black` ortholog-set of $\mathcal{S}(\mathbb{A}, \mathbb{B}, \mathcal{S})$

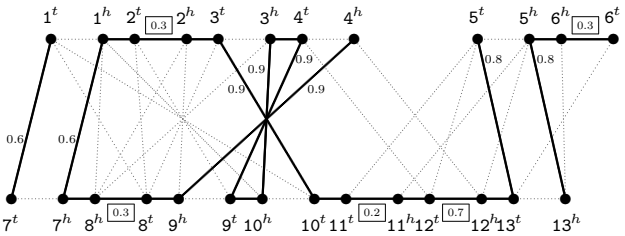


Figure 1 On the top part is displayed the gene similarity graph $\mathcal{S}(\mathbb{A}, \mathbb{B})$ of genomes $\mathbb{A} = \{[1\,2\,3\,4]\ [5\,\overline{6}]\}$ and $\mathbb{B} = \{[7\,\overline{8}\,\overline{9}\,\overline{10}\,11\,12\,13]\}$ and next to it a table with the ranking of four distinct ortholog-sets. On the middle the rearrangement scenarios induced by two of these ortholog-sets are shown. On the bottom part the family-free relational graph $FFR(\mathbb{A}, \mathbb{B}, \mathcal{S})$ is illustrated, highlighting the edges of the decomposition corresponding to the (`black`) ortholog-set $M = \{\{1, 7\}, \{3, 10\}, \{4, 9\}, \{5, 13\}\}$. (This decomposition has two $\mathbb{AB}$-paths, one $\mathbb{AA}$-path and one cycle.) All extremity and indel edges in $FFR(\mathbb{A}, \mathbb{B}, \mathcal{S})$ are weighted according to $\mathcal{S}(\mathbb{A}, \mathbb{B})$ but the weights of edges not derived from M or $\widetilde{M}$ are omitted.

2.3 Consistent decompositions of the family-free relational graph

A *decomposition* of $FFR(\mathbb{A}, \mathbb{B}, \mathcal{S})$ is a collection of vertex-disjoint *components*, that can be cycles and/or paths, covering all vertices of $FFR(\mathbb{A}, \mathbb{B}, \mathcal{S})$. We only consider *consistent* decompositions that correspond to ortholog-sets of $\mathcal{S}(\mathbb{A}, \mathbb{B})$. A set $S \subseteq E_\gamma$ is a *sibling-set* if it is exclusively composed of pairs of siblings and does not contain any pair of incident edges. Thus, a sibling-set S of $FFR(\mathbb{A}, \mathbb{B}, \mathcal{S})$ corresponds to an ortholog-set $M(S)$ of $\mathcal{S}(\mathbb{A}, \mathbb{B})$.

The set of edges $D[S]$ *induced* by a sibling-set S is said to be a *consistent decomposition* of $FFR(\mathbb{A}, \mathbb{B}, \mathcal{S})$ and can be obtained as follows. In the beginning, $D[S]$ is the union of S with the sets of adjacency edges $E_{\text{adj}}^{\mathbb{A}}$ and $E_{\text{adj}}^{\mathbb{B}}$. We then need to determine the *complement*

of the sibling-set S, denoted by $\widetilde{S}$, that is composed of the indel-edges of $FFR(\mathbb{A}, \mathbb{B}, \mathcal{S})$ that must be added to $D[S]$: for each indel edge e, if its two endpoints have degree one or zero in $D[S]$, then e is added to both $\widetilde{S}$ and $D[S]$. (Note that $\widetilde{S}$ is equal to the complement of $M(S)$, while $|S| = 2|M(S)|$ and $w(S) = 2w(M(S))$.) The consistent decomposition $D[S]$ covers all vertices of $FFR(\mathbb{A}, \mathbb{B}, \mathcal{S})$ and is composed of cycles and paths. The paths connect the ends of linear chromosomes in both genomes and can be of three types: either $\mathbb{A}\mathbb{A}$-path, or $\mathbb{B}\mathbb{B}$-path or $\mathbb{A}\mathbb{B}$-path.

The structure of $D[S]$ has all necessary information for computing $\mathrm{wd}^{\mathrm{ID}}_{\mathrm{DCJ}}(\mathbb{A}, \mathbb{B}, \mathcal{S}, M(S))$, therefore we can say that $\mathrm{wd}^{\mathrm{ID}}_{\mathrm{DCJ}}(\mathbb{A}, \mathbb{B}, \mathcal{S}, M(S)) = \mathrm{wd}^{\mathrm{ID}}_{\mathrm{DCJ}}(D[S])$ [19] and modify our optimization problem to $\textsc{Diff}(\mathbb{A}, \mathbb{B}, \mathcal{S}) = \min_{S \in \mathfrak{S}} \{\mathrm{wd}^{\mathrm{ID}}_{\mathrm{DCJ}}(D[S])\}$, where $\mathfrak{S}$ is the set of all possible sibling-sets in $FFR(\mathbb{A}, \mathbb{B}, \mathcal{S})$. Assuming that a decomposition $D[S_\star]$ gives the optimal solution for $\textsc{Diff}(\mathbb{A}, \mathbb{B}, \mathcal{S})$, then $\textsc{DiffM}(\mathbb{A}, \mathbb{B}, \mathcal{S}) = M(S_\star)$.

3 Capping

The end of a linear chromosome is called *telomere*. The telomeres are also the ends of the paths of any consistent decomposition. Therefore, if $\kappa(\mathbb{A})$ is the number of linear chromosomes in $\mathbb{A}$ and $\kappa(\mathbb{B})$ is the number of linear chromosomes in $\mathbb{B}$ the number of paths in any decomposition is $\kappa(\mathbb{A}) + \kappa(\mathbb{B})$. Our ILP is able to capture all necessary properties from the cycles of a decomposition, but cannot handle paths. A way to overcome this problem is by linking all paths of any decomposition with a known technique called *capping* [13].

3.1 Capping a consistent decomposition

The idea of the capping is to split the telomeres into disjoint pairs and then to connect the two elements of each pair, so that all paths are linked into cycles. The only restriction is that a pair cannot contain telomeres from the same genome, therefore, if the numbers of telomeres in the two genomes are different, some *dummy* telomeres need to be created, as we describe in the following.

Suppose that $D[S]$ is any consistent decomposition of $FFR(\mathbb{A}, \mathbb{B}, \mathcal{S})$. For each telomere (vertex) v, add to $D[S]$ a *cap vertex* θ_v and connect v to θ_v by an adjacency edge. Now let $\theta(\mathbb{A})$ (respectively $\theta(\mathbb{B})$) be the set of all cap vertices in $\mathbb{A}$ (respectively in $\mathbb{B}$). Note that, since each linear chromosome has two ends, the cardinalities of these sets must be even. Moreover, if $|\theta(\mathbb{A})| \neq |\theta(\mathbb{B})|$, the cardinalities of these sets must be equalized. Let $p_* = \max\{\kappa(\mathbb{A}), \kappa(\mathbb{B})\}$ and $a_* = |\kappa(\mathbb{A}) - \kappa(\mathbb{B})|$. For equalizing the cardinalities with the minimum number of extra vertices, we need to add $2a_*$ extra cap vertices to the set with smaller cardinality. These extra cap vertices must be split into pairs (arbitrarily chosen) so that the vertices of each pair are connected by a *dummy* adjacency edge in $D[S]$. Denote by $\hat{\theta}(\mathbb{A})$ and $\hat{\theta}(\mathbb{B})$ the sets with equalized cardinalities and let P be a *capping-set*, which is a perfect matching between them: for $\gamma \in \hat{\theta}(\mathbb{A})$ and $\gamma' \in \hat{\theta}(\mathbb{B})$, if $\gamma\gamma' \in P$, then γ and γ' are connected by a *cap edge*. Let $\theta(D[S], P)$ be a *capped decomposition* of $D[S]$ with capping-set P. It is easy to see that $\theta(D[S], P)$ is composed of cycles only.

DCJ-indel optimal capping

So far we explained how to guarantee that all paths in any decomposition are linked into cycles. Note, however, that there are $(2p_*)!$ ways of completely matching the vertices of sets $\hat{\theta}(\mathbb{A})$ and $\hat{\theta}(\mathbb{B})$. For a given decomposition $D[S]$, any of these possibilities, say capping-set

P, would produce a capped decomposition $\theta(D[S], P)$, and capping the same $D[S]$ with distinct capping-sets may produce distinct weighted costs. Let $P_\star$ be an optimal capping-set for $D[S]$, that is, the capped decomposition $\theta(D[S], P_\star)$ has the minimum weighted cost among all capped decompositions of $D[S]$. (There can be several co-optimal capping-sets for the same decomposition $D[S]$ and each optimal capping-set links up to 4 cycles of $D[S]$ into a single cycle [5].) It has been shown that $\mathrm{wd}^{\mathrm{ID}}_{\mathrm{DCJ}}(\theta(D[S], P_\star)) = \mathrm{wd}^{\mathrm{ID}}_{\mathrm{DCJ}}(D[S])$, therefore any consistent decomposition of $D[S]$ with an optimal capping-set is DCJ-indel optimal and preserves the weighted cost of $D[S]$, reporting both $\mathrm{d}^{\mathrm{ID}}_{\mathrm{DCJ}}(\mathbb{A}, \mathbb{B}, M(S))$ and $\mathrm{wd}^{\mathrm{ID}}_{\mathrm{DCJ}}(\mathbb{A}, \mathbb{B}, \mathbb{S}, M(S))$ [19]. Figure 2 (top) highlights an optimal capping of a consistent decomposition.

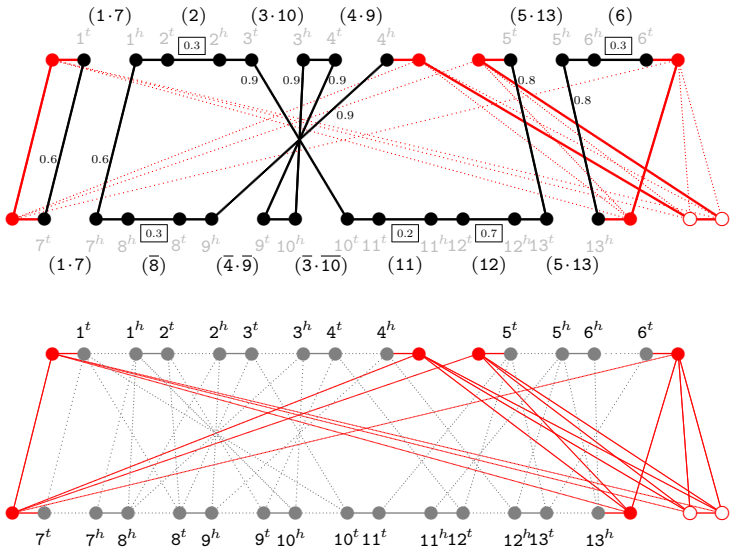


Figure 2 On the top part we show the capping of the decomposition corresponding to the (`black`) ortholog-set $M = \{\{1, 7\}, \{3, 10\}, \{4, 9\}, \{5, 13\}\}$ from the gene similarity graph $\mathbb{S}(\mathbb{A}, \mathbb{B})$ of Figure 1 (bottom). Each red vertex is a cap vertex. Each filled (red) vertex is connected to a telomere (chromosome/path ends). The unfilled vertices represent the extra (equalizing) vertices connected by a dummy adjacency. The capping is a perfect matching of the complete bipartite graph of the cap vertices. The optimal capping for this decomposition is highlighted. It closes each of its paths into a separate cycle. (In general, an optimal capping of a decomposition may link up to 4 paths into a single cycle [5]). On the bottom part is displayed the complete family-free graph $FFR(\mathbb{A}, \mathbb{B}, \mathbb{S})$ optimally capped. Cap edges are unweighted. Weights of extremity and indel edges are omitted.

3.2 Optimally capped family-free relational graph

All consistent decompositions share the same telomeres, therefore a set of capping-sets for one decomposition is also a set of capping-sets of any other decomposition. If we then simply add all possible capping-sets to the family-free relational graph, which implies adding a complete bipartite graph with partite sets $\hat{\theta}(\mathbb{A})$ and $\hat{\theta}(\mathbb{B})$, we guarantee that an optimal solution can be found. Let the so-called optimal capping (represented in Figure 2 (bottom)) of $FFR(\mathbb{A}, \mathbb{B}, \mathbb{S})$ with the minimum number of extra elements be denoted by $\theta_\star(FFR(\mathbb{A}, \mathbb{B}, \mathbb{S}))$ and be defined as follows:

1. Add the set of *cap vertices* $\hat{\theta}(\mathbb{A}) = \theta^1_{\mathbb{A}}, \theta^2_{\mathbb{A}}, \ldots, \theta^{2p_*}_{\mathbb{A}}$ and connect each telomere of genome $\mathbb{A}$ to one of these cap vertices by an adjacency edge added to $E^{\mathbb{A}}_{\text{adj}}$.
2. Similarly, add the set of cap vertices $\hat{\theta}(\mathbb{B}) = \theta^1_{\mathbb{B}}, \theta^2_{\mathbb{B}}, \ldots, \theta^{2p_*}_{\mathbb{B}}$ and connect each telomere of genome $\mathbb{B}$ to one of these cap vertices by an adjacency edge added to $E^{\mathbb{B}}_{\text{adj}}$.
3. Add (arbitrarily chosen) $p_* - \kappa(\mathbb{A})$ dummy adjacency edges to $E^{\mathbb{A}}_{\text{adj}}$ and $p_* - \kappa(\mathbb{B})$ dummy adjacency edges to $E^{\mathbb{B}}_{\text{adj}}$. (Note that only one of the two genomes may have dummy adjacencies.)
4. Connect all cap vertices in $\hat{\theta}(\mathbb{A})$ to all cap vertices in $\hat{\theta}(\mathbb{B})$ with *cap edges*. The set of all cap edges is denoted by E_θ.

Since all $2p_*$ cap vertices in $\mathbb{A}$ are connected to all $2p_*$ cap vertices in $\mathbb{B}$ and any perfect matching of these edges is a valid capping, the search space of our optimization problem is multiplied by $(2p_*)!$. Denote by $\mathfrak{P}$ the set of all possible capping-sets (perfect matchings) between the vertices from $\hat{\theta}(\mathbb{A})$ and $\hat{\theta}(\mathbb{B})$. The optimization problem over $\theta_\star(\mathit{FFR}(\mathbb{A}, \mathbb{B}, \mathcal{S}))$ can be rewritten as $\textsc{Diff}(\mathbb{A}, \mathbb{B}, \mathcal{S}) = \min_{S \in \mathfrak{S}, P \in \mathfrak{P}} \{\text{wd}^{\text{ID}}_{\text{DCJ}}(\theta(D[S], P)\}$. Assuming that an optimal capping of a decomposition $D[S_\star]$ gives the optimal solution for $\textsc{Diff}(\mathbb{A}, \mathbb{B}, \mathcal{S})$, the optimal ortholog-set is $\textsc{DiffM}(\mathbb{A}, \mathbb{B}, \mathcal{S}) = M(S_\star)$. Both problems Diff and DiffM can be solved with the ILP formulation FF-DCJ-Indel [19], which can be found in Appendix A.

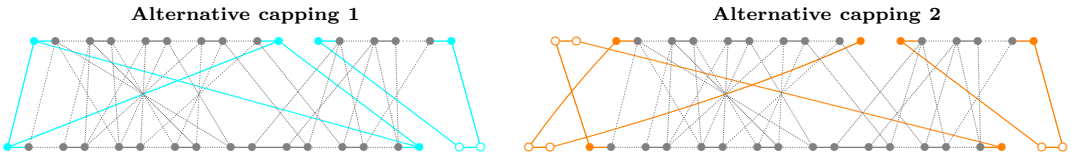
3.3 Integration of pairwise optimal ortholog-sets into gene families

In our previous work [18], the ILP FF-DCJ-Indel solving DiffM (with optimal capping) was integrated in a tool called DiffMGC for inferring gene families across several species. The pipeline, illustrated in Figure 6 of Appendix B, can be summarized as follows: given a set of n genomes, gene similarities and ortholog-sets are computed for all pairwise comparisons and simply integrated into an n-partite graph. The connected components of this graph are the inferred gene families.

4 Heuristic capping

Conceptually our approach can handle partially assembled genomes distributed into several contigs/scaffolds: each of these is a linear segment and could simply be treated as the same object that we so far called chromosome. However, as already explained, the optimal capping multiplies the search space of $\mathit{FFR}(\mathbb{A}, \mathbb{B})$ by $(2p_*)!$ where p_* is the maximum between the number of linear segments in genomes $\mathbb{A}$ and $\mathbb{B}$. This effect makes it unfeasible to analyze genomes with a large number of segments with our ILP over an optimally capped family-free relational graph.

One way of overcoming this issue is by adopting a lighter capping, for example by removing some edges from the complete bipartite graph of $\hat{\theta}(\mathbb{A})$ and $\hat{\theta}(\mathbb{B})$, and/or by partitioning these sets into subsets that are capped independently. In any case it is important to guarantee that a capping is *valid*, that is, it allows to find a capping-set (a perfect matching of the cap vertices). A valid lighter capping may not include the optimal capping-sets, and therefore may not preserve the computed weighted costs. Note, however, that even if the weighted costs are not preserved, the ranking of the ortholog-sets/sibling-sets may not be affected. In Figure 3 we show examples of arbitrary lighter valid cappings and their effects on the sibling-set ranking.



Orthologs-sets and their corresponding costs with the two alternative lighter cappings above

				Optimal		Alternative 1		Alternative 2	
M	$\|M\|$	$w(M)$	$w(\widetilde{M})$	$\mathrm{d}_{\mathrm{DCJ}}^{\mathrm{ID}}$	$\mathrm{wd}_{\mathrm{DCJ}}^{\mathrm{ID}}$	$\mathrm{d}_{\mathrm{DCJ}}^{\mathrm{ID}}$	$\mathrm{wd}_{\mathrm{DCJ}}^{\mathrm{ID}}$	$\mathrm{d}_{\mathrm{DCJ}}^{\mathrm{ID}}$	$\mathrm{wd}_{\mathrm{DCJ}}^{\mathrm{ID}}$
black	4	3.2	1.8	5	7.6	6	8.6	8	10.6
green	5	3.5	1.2	5	7.7	6	8.7	8	10.7
blue	3	2.6	3.0	5	8.4	6	9.4	7	10.4
magenta	4	1.7	2.8	6	11.1	6	11.1	7	12.1

Figure 3 Examples of two arbitrary lighter valid cappings of the family-free relational diagram from Figure 1 (bottom) and their effects on the ranking of ortholog-sets/sibling-sets represented in Figure 1 (top). Both cappings affect the computed distances, but, while the capping shown in the left (cyan) preserves the optimal ranking, the one shown in the right (orange) does not.

4.1 Perfect contig intersection graph with thresholds τ and ϵ

Our goal is therefore to develop a lighter heuristic capping that may potentially preserve the original (optimal) ranking of the best ortholog-sets/sibling-sets. We achieve this by connecting cap vertices only between the telomeres of the linear segments (contigs or chromosomes) that (potentially) share most of their genomic contents. This is because those telomeres have a higher chance of being in the same paths of the best consistent decompositions of the family-free relational graph.

Given two contigs α and β belonging to genomes $\mathbb{A}$ and $\mathbb{B}$, respectively, their *shared genomic content* $\lambda(\alpha, \beta)$ is the number of edges in $\mathcal{S}$ between genes of α and β. Formally, for $\alpha \subseteq \mathbb{A}$, $\beta \subseteq \mathbb{B}$, we have $\lambda(\alpha, \beta) = |\{g_1 g_2 \in \mathcal{S} \mid g_1 \in \alpha, g_2 \in \beta \text{ and } w(g_1 g_2) > 0\}|$. Now let $\mathcal{C}(\mathbb{A}, \mathbb{B}) = (V_{\mathbb{A}}, V_{\mathbb{B}}, E)$ be the bipartite *contig intersection graph* where the vertex sets are $V_{\mathbb{A}} = \{\alpha : \alpha \text{ is a contig in } \mathbb{A}\}$ and $V_{\mathbb{B}} = \{\beta : \beta \text{ is a contig in } \mathbb{B}\}$. Initially the set of edges is $E = \{\alpha\beta \mid \alpha \in V_{\mathbb{A}}, \beta \in V_{\mathbb{B}} \text{ and } \lambda(\alpha, \beta) > 0\}$. Each edge $\alpha\beta$ is weighted such that $w(\alpha\beta) = \lambda(\alpha\beta)$. Then, given a positive integer τ, we remove some edges from E by applying a filtering procedure that simply iterates over $V_{\mathbb{A}} \cup V_{\mathbb{B}}$, keeping in E only the τ edges of highest weights for each vertex. Then, the remaining edges are again filtered out to remove weak relations between contigs, according to another threshold given by a rational value $\epsilon \in [0, 1]$: for each vertex v and the edge e of highest weight incident to v, edges uv of weights below $\epsilon w(e)$ are removed.

Capping attempt induced by the contig intersection graph

The contig intersection graph will now *induce* our capping procedure. The idea is to allow cap connections only between the ends of contigs that are connected in $\mathcal{C}$. Therefore, the contigs that are in the same connected component of $\mathcal{C}$ will be capped together, independently from the contigs that are in other connected components. In other words, the connected components of $\mathcal{C}$ will impose a partitioning of the capping procedure.

Let us then assume that $\mathcal{C}$ has a single connected component. Note that a capping induced by $\mathcal{C}$ can only be valid if its partite sets $V_{\mathbb{A}}$ and $V_{\mathbb{B}}$ are of the same size. This necessary condition also applies and is sufficient for the optimal capping, but here it is not sufficient: even when $V_{\mathbb{A}}$ and $V_{\mathbb{B}}$ are of the same size, since not all connections between the ends of the contigs in $V_{\mathbb{A}}$ and in $V_{\mathbb{B}}$ are present, the induced capping could be invalid (Figure 4 (a)).

Here the necessary and sufficient condition for a valid capping is the existence of a perfect matching in $\mathcal{C}(\mathbb{A}, \mathbb{B})$, as stated in Lemma 2, whose proof relies on a theorem closely related to perfect matchings and demonstrated by Hall [12] in 1935. Denote by $\mathcal{N}(S)$ the neighborhood of a vertex set S, that is, the set of all vertices adjacent to some vertex of S.

▶ **Theorem 1** (Hall's marriage theorem). *Let $G = (U, V, E)$ be a bipartite graph. There exists a matching in G that covers the vertex set V if and only if for each subset $S \subseteq V$, $|S| \leq |\mathcal{N}(S)|$.*

Note that a perfect matching exists in $\mathcal{C}$ if and only if the condition of Theorem 1 holds for both $V_{\mathbb{A}}$ and $V_{\mathbb{B}}$. We can now establish the relation between perfect matchings in $\mathcal{C}$ and the validity of the capping it induces in the family-free relational graph.

▶ **Lemma 2.** *A perfect matching exists in $\mathcal{C}(\mathbb{A}, \mathbb{B})$ if and only if the capping of $FFR(\mathbb{A}, \mathbb{B}, \mathcal{S})$ induced by $\mathcal{C}$ is valid.*

Proof. If a perfect matching M exists in $\mathcal{C}$, for each edge $\alpha\beta$ in M, in the induced capping of $FFR(\mathbb{A}, \mathbb{B}, \mathcal{S})$ the pair of cap vertices connected to the ends of α can be connected in any of the two distinct ways to the pair of cap vertices connected to the ends of β, resulting in a capping-set.

The converse is shown by contraposition. Suppose that a maximum cardinality matching M in $\mathcal{C}$ is not a perfect matching. Therefore, by Hall's marriage theorem, there exists some S in $\mathcal{C}$ such that $|\mathcal{N}(S)| < |S|$. Let S' be the set of cap vertices in the capping induced by $\mathcal{C}$ for all contigs in S. Since the connection of these cap vertices follows $\mathcal{C}$ and each contig has two cap vertices, it is clear that $|S'| = 2|S|$ and $|\mathcal{N}(S')| = 2|\mathcal{N}(S)|$, hence, $|\mathcal{N}(S')| < |S'|$. By the pigeonhole principle, at least 2 cap vertices (because $|\mathcal{N}(S')|$ and $|S'|$ are even numbers) will not be incident to any cap edge, therefore no capping-set exists. ◀

Building the perfect contig intersection graph

We will now describe a procedure for transforming the contig intersection graph $\mathcal{C}(\mathbb{A}, \mathbb{B})$ into a *perfect contig intersection graph* $\widehat{\mathcal{C}}(\mathbb{A}, \mathbb{B})$ that has at least one perfect matching, as shown in Algorithm 1. In the completion loop, dummy contigs are iteratively created until a perfect matching is possible. If a maximum cardinality matching M is found but is not a perfect matching, a Hall violator set S can be found as follows. Let v be a vertex unsaturated by M, then $S = \{v\} \cup \{u \mid u \text{ is reachable from } v \text{ by an } M\text{-alternating path}\}$. Finally, $|S| - |\mathcal{N}(S)|$ dummy contigs are created and connected to each contig in S.

An edge in $\widehat{\mathcal{C}}(\mathbb{A}, \mathbb{B})$ is *matchable* if it is part of at least one perfect matching and *non-matchable* otherwise. Once the completion loop is finished, $\widehat{\mathcal{C}}$ admits at least one perfect matching and its matchable edges can be identified efficiently [23]. The last step of our algorithm is then removing from $\widehat{\mathcal{C}}$ all non-matchable edges. An example of the construction of a perfect contig intersection graph is illustrated in Figure 4 (b).

Search space compared to optimal capping

If the threshold τ is similar to the numbers $\kappa(\mathbb{A})$ and $\kappa(\mathbb{B})$ of contigs in each genome, and in the unlike situation where all contigs from one genome are connected to all contigs from the other genome in $\mathcal{C}$, the heuristic capping induced by $\widehat{\mathcal{C}}$ may be as large as the optimal capping.

The threshold τ is thought to be smaller than $\kappa(\mathbb{A})$ and $\kappa(\mathbb{B})$, effectively reducing the number of capping-sets. We could not yet estimate this reduction as a function of τ, though. As our experimental results with real genomes show (details below, in Section 5), with a small τ the heuristic capping leans to a considerably smaller number of capping-sets in practice.

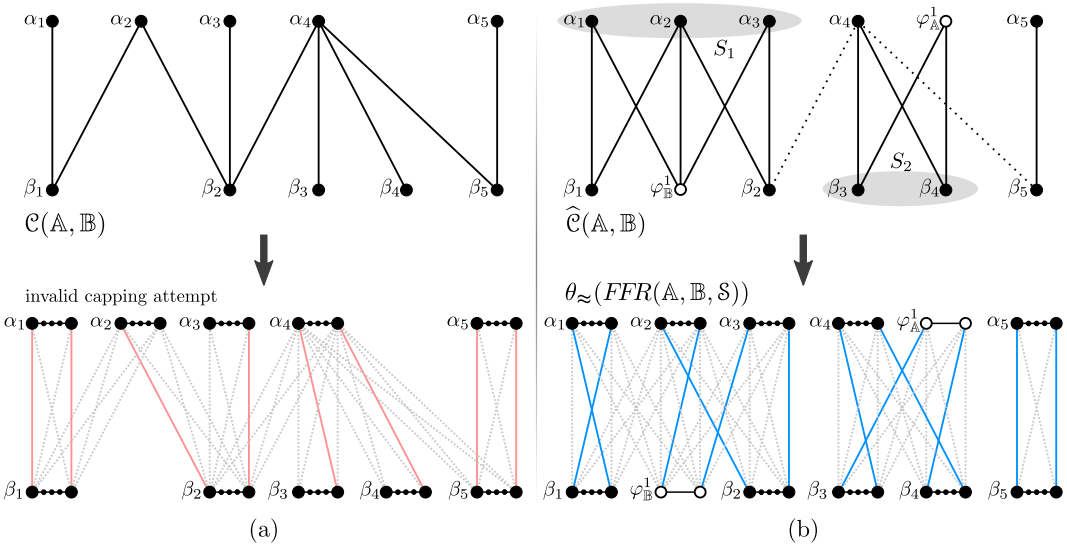


Figure 4 (a) Example of a contig intersection graph $\mathcal{C}(\mathbb{A},\mathbb{B})$. The genomes $\mathbb{A}$ and $\mathbb{B}$ have contigs $\alpha_{1..5}$ and $\beta_{1..5}$, respectively. The capping of $FFR(\mathbb{A},\mathbb{B},\mathcal{S})$ induced by $\mathcal{C}$ is invalid. (b) Transformation of $\mathcal{C}$ into a perfect contig intersection graph $\widehat{\mathcal{C}}(\mathbb{A},\mathbb{B})$: Vertex sets S_1 and S_2 represent Hall violators (among other possibilities) that demand the creation of dummy contigs $\varphi^1_{\mathbb{B}}$ and $\varphi^1_{\mathbb{A}}$, respectively. Dotted edges represent those that are non-matchable and must be removed from $\widehat{\mathcal{C}}$ after the completion is finished. Notice that the component with vertices $\alpha_1,\alpha_2,\alpha_3,\beta_1,\varphi^1_{\mathbb{B}}$ and β_2 is not a complete bipartite subgraph. (In both (a) and (b), for the capped *FFR* only cap vertices, cap edges and dummy adjacencies are represented explicitly, while vertices of gene extremities between cap vertices are represented by a line with small dots. In addition, colored solid edges represent a maximum cardinality matching between cap vertices, while the cap edges not in the matching are dashed grey.)

4.2 Heuristically capped family-free relational graph

The bipartite perfect contig intersection graph $\widehat{\mathcal{C}}(\mathbb{A},\mathbb{B})$ has edge set $\widehat{E}$ and partite sets $\widehat{V}_{\mathbb{A}} = V_{\mathbb{A}} \cup V^{\varphi}_{\mathbb{A}}$ and $\widehat{V}_{\mathbb{B}} = V_{\mathbb{B}} \cup V^{\varphi}_{\mathbb{B}}$, where $V_{\mathbb{A}}$ and $V_{\mathbb{B}}$ are the sets of contigs and $V^{\varphi}_{\mathbb{A}}$ and $V^{\varphi}_{\mathbb{B}}$ the sets of dummy contigs. Recall that the sets $\widehat{V}_{\mathbb{A}}$ and $\widehat{V}_{\mathbb{B}}$ have the same cardinality, which here we denote by $p_{\approx}$. The heuristic capping $\theta_{\approx}$ of the family-free relational graph $FFR(\mathbb{A},\mathbb{B},\mathcal{S})$ induced by $\widehat{\mathcal{C}}(\mathbb{A},\mathbb{B})$ is shown in Figure 4 (b) and described as follows:

1. Add the set of cap vertices $\hat{\theta}(\mathbb{A}) = \theta^1_{\mathbb{A}}, \theta^2_{\mathbb{A}}, \ldots, \theta^{2p_{\approx}}_{\mathbb{A}}$. For $i = 1 \ldots |V_{\mathbb{A}}|$, associate each contig $\alpha_i \in V_{\mathbb{A}}$ to cap vertices $\theta^{2i-1}_{\mathbb{A}}$ and $\theta^{2i}_{\mathbb{A}}$ and connect with adjacency edges one telomere of α_i to $\theta^{2i-1}_{\mathbb{A}}$ and the other to $\theta^{2i}_{\mathbb{A}}$. Note that $2|V^{\varphi}_{\mathbb{A}}|$ cap vertices remain disconnected.
2. Similarly, add cap vertices $\hat{\theta}(\mathbb{B}) = \theta^1_{\mathbb{B}}, \theta^2_{\mathbb{B}}, \ldots, \theta^{2p_{\approx}}_{\mathbb{B}}$. For $j = 1 \ldots |V_{\mathbb{B}}|$, associate each contig $\beta_j \in V_{\mathbb{B}}$ to cap vertices $\theta^{2j-1}_{\mathbb{B}}$ and $\theta^{2j}_{\mathbb{B}}$ and connect with adjacency edges one telomere of β_i to $\theta^{2j-1}_{\mathbb{B}}$ and the other to $\theta^{2j}_{\mathbb{B}}$. Again, $2|V^{\varphi}_{\mathbb{B}}|$ cap vertices remain disconnected.
3. For $i_\circ = 1 \ldots |V^{\varphi}_{\mathbb{A}}|$ and $i = |V_{\mathbb{A}}| + i_\circ$, connect the pair of cap vertices $\theta^{2i-1}_{\mathbb{A}}$ and $\theta^{2i}_{\mathbb{A}}$ by a dummy adjacency edge, associating this pair to the dummy contig $\varphi^{i_\circ}_{\mathbb{A}} \in V^{\varphi}_{\mathbb{A}}$. Similarly, for $j_\circ = 1 \ldots |V^{\varphi}_{\mathbb{B}}|$ and $j = |V_{\mathbb{B}}| + j_\circ$, connect the pair of cap vertices $\theta^{2j-1}_{\mathbb{B}}$ and $\theta^{2j}_{\mathbb{B}}$ by a dummy adjacency edge, associating this pair to the dummy contig $\varphi^{j_\circ}_{\mathbb{B}} \in V^{\varphi}_{\mathbb{B}}$.
4. For each edge $ab \in \widehat{E}$, let $a \in \widehat{V}_{\mathbb{A}}$ and $b \in \widehat{V}_{\mathbb{B}}$ be associated, respectively, to the cap vertices $\theta^{2i-1}_{\mathbb{A}}, \theta^{2i}_{\mathbb{A}} \in \hat{\theta}(\mathbb{A})$ and $\theta^{2j-1}_{\mathbb{B}}, \theta^{2j}_{\mathbb{B}} \in \hat{\theta}(\mathbb{B})$. Connect the four "crossing" pairs of cap vertices $\{\theta^{2i-1}_{\mathbb{A}}, \theta^{2j-1}_{\mathbb{B}}\}$, $\{\theta^{2i}_{\mathbb{A}}, \theta^{2j-1}_{\mathbb{B}}\}$, $\{\theta^{2i-1}_{\mathbb{A}}, \theta^{2j}_{\mathbb{B}}\}$ and $\{\theta^{2i}_{\mathbb{A}}, \theta^{2j}_{\mathbb{B}}\}$ with cap edges. The set of all cap edges is denoted by E_θ.

Algorithm 1 Creates a perfect contig intersection graph from a contig intersection graph.

Input: A contig intersection graph $\mathcal{C}(\mathbb{A}, \mathbb{B}) = (V_{\mathbb{A}}, V_{\mathbb{B}}, E)$
Output: A perfect contig intersection graph $\widehat{\mathcal{C}}(\mathbb{A}, \mathbb{B}) = (\widehat{V}_{\mathbb{A}}, \widehat{V}_{\mathbb{B}}, \widehat{E})$
1: $\widehat{V}_{\mathbb{A}} \leftarrow V_{\mathbb{A}}\,, \quad \widehat{V}_{\mathbb{B}} \leftarrow V_{\mathbb{B}}\,, \quad \widehat{E} \leftarrow E$
2: **while** a maximum cardinality matching M in $\widehat{E}$ is not a perfect matching **do**
3: $\quad S \leftarrow$ a Hall violator set derived from M
4: $\quad \Phi \leftarrow$ dummy contigs $\{\varphi^1, \ldots, \varphi^{|S|-|\mathcal{N}(S)|}\}$
5: $\quad$ **if** $S \subseteq \widehat{V}_{\mathbb{A}}$ **then** $\widehat{V}_{\mathbb{B}} \leftarrow \widehat{V}_{\mathbb{B}} \cup \Phi$ **else** $\widehat{V}_{\mathbb{A}} \leftarrow \widehat{V}_{\mathbb{A}} \cup \Phi$
6: $\quad \widehat{E} \leftarrow \widehat{E} \cup (S \times \Phi)$
7: remove from $\widehat{E}$ all non-matchable edges
8: **return** $\widehat{\mathcal{C}} = (\widehat{V}_{\mathbb{A}}, \widehat{V}_{\mathbb{B}}, \widehat{E})$

Denote by $\mathfrak{P}_{\approx}$ the set of all possible capping-sets (perfect matchings) between the vertices of $\hat{\theta}(\mathbb{A})$ and $\hat{\theta}(\mathbb{B})$. The optimization problem over $\theta_{\approx}(FFR(\mathbb{A}, \mathbb{B}, \mathcal{S}))$ is defined as

$$\textsc{DiffH}^{\approx}(\mathbb{A}, \mathbb{B}, \mathcal{S}) = \min_{S \in \mathfrak{S}, P \in \mathfrak{P}_{\approx}} \{\mathrm{wd}^{\mathrm{ID}}_{\mathrm{DCJ}}(\theta(D[S], P)\}\,.$$

Assuming that a heuristic capping-set of a decomposition $D[S_{\approx}]$ gives the optimal solution for $\textsc{DiffH}^{\approx}(\mathbb{A}, \mathbb{B}, \mathcal{S})$, the best heuristic ortholog-set is $\textsc{DiffMH}^{\approx}(\mathbb{A}, \mathbb{B}, \mathcal{S}) = M(S_{\approx})$. Both problems $\textsc{DiffH}^{\approx}$ and $\textsc{DiffMH}^{\approx}$ can also be solved with the ILP FF-DCJ-Indel, shown in Appendix A.

4.3 Integration of pairwise heuristic ortholog-sets into gene families

The ILP FF-DCJ-Indel solving $\textsc{DiffMH}^{\approx}$ (with heuristic capping) is the core of a new version of our tool, called $\textsc{DiffMGCH}^{\approx}$, for inferring gene families across several species, as illustrated in Figure 6 of Appendix B. Recall that each family is a connected component of the n-partite graph obtained by the simple integration of the computed pairwise ortholog-sets. An *ambiguous* family corresponds to a connected component of the n-partite graph that has more than one gene from the same genome. Otherwise we have a *resolved* family, which can be either *complete*, when it contains one gene per genome, or *incomplete* otherwise. The types of families are shown in Figure 7 of Appendix B.

5 Implementation and experiments

The pipeline DiffMGC (with optimal capping) was previously integrated into the ffgc workflow [11, 19], which includes the pre-computation of gene similarities, allowing therefore the automatic generation of families directly from the genome data. We implemented the new pipeline $\textsc{DiffMGCH}^{\approx}$ (with heuristic capping) as another extension of the same workflow. The implementation and its documentation can be downloaded from our GitLab server at `gitlab.ub.uni-bielefeld.de/gi/FFGC`.

5.1 Computational environment and parameters

We ran experiments in a 2.4GHz multi-core machine. Whenever possible, tasks ran using 8 cores. As an ILP solver, we used CPLEX. The default values of the parameters of DiffMGC and $\textsc{DiffMGCH}^{\approx}$ for the pre-computation of gene similarities (with the help of blast [2]) were kept, except for two of them. The first one is the minimum number of genomes for

which each gene must share some similarity in, set to 1 – otherwise genes not similar to any other gene, which should be still considered in indels, will not appear in genomes. The second one is the stringency threshold, set to $t = 0.8$. Succinctly, the stringency filter [11, 14] prunes edges in $\mathcal{S}(A, B)$ that are adjacent to edges with considerably higher weights based on a threshold. Empirical evidence shows that thresholds higher than $t = 0.8$ may discard relevant gene relationships, while lower values simply increase the ILP running time with small variations in the results found by the solver.

For the capping heuristic, we set $\tau = 3$ (which in average corresponds to half of the number of chromosomes in a completely assembled *Drosophila* genome) and $\epsilon = 0.01$ (a conservative choice for only filtering out from $\widehat{\mathcal{C}}$ very weak edges). Since these (reasonable) choices produced very good results, we did not explore other possibilities, but in a future work we intend to do a systematic study testing the speed and quality performances of the heuristic capping for different values of τ and ϵ.

5.2 Analysis of *Drosophila* genomes

All genome assemblies used in experiments were fetched from NCBI. The FlyBase consortium sequenced, assembled and annotated the genomes of 12 *Drosophilas* with $\sim$ 12,000–16,000 protein-coding genes, however only 11 of those genomes are available on NCBI together with the complete annotation: *D. simulans*, *D. sechellia*, *D. melanogaster*, *D. yakuba*, *D. erecta*, *D. ananassae*, *D. persimilis*, *D. willistoni*, *D. mojavensis*, *D. virilis* and *D. grimshawi*.

Average numbers of cap edges and capping-sets in $\theta_\star$ and in $\theta_\approx$

The analyzed *Drosophila* genomes have 507 contigs on average, therefore each optimally capped family-free relational diagram has 1,014 × 1,014 = 1,028,196 cap edges and an unfeasible total of 1,014! capping-sets on average.

In contrast, considering the perfect contig intersection graphs for all pairwise *Drosophila* comparisons, 99.7% of the components in those graphs have only 1 contig in each part of the graph. In the remaining 0.3%, 80% have 7 or fewer contigs in each part, with the largest component having 76 contigs in each part. The perfect contig intersection graphs have an average of 1,419 edges. For that number of edges, each heuristically capped family-free relational diagram has 2,838 cap edges on average. As the exact number of perfect matchings in arbitrary graphs is not trivial to estimate, we computed an upper limit for the average number of distinct capping-sets: $\sim$ 305!.

Benchmark for our experiments

Reference families were obtained directly from FlyBase (`flybase.org`). Since the set of genes classified in FlyBase is slightly different from the set of genes present with their coding sequences in database files, we filtered out a small portion ($\sim$ 7%) of genes in FlyBase families so that only those present in the databases with their coding sequences were kept. Prior to any comparison of inferred families to FlyBase families, we also filtered out from the inferred families genes not present in FlyBase families.

We perform two distinct comparisons. First, based on a set of three completely assembled *Drosophila* genomes, we compare DiffMGC$\widetilde{\widetilde{\text{H}}}$ families (inferred with the optimal capping) to DiffMGC families (inferred with the heuristic capping) using FlyBase families as reference. Next, based on the complete set of 11 *Drosophilas* including partially assembled genomes, we compare DiffMGC$\widetilde{\widetilde{\text{H}}}$ families to the gene families inferred by other inference tools, again using FlyBase families as reference.

5.2.1 Comparing DiffMGC$^{\approx\approx}_{\text{H}}$ to DiffMGC

We performed a first empirical evaluation on how the heuristic capping could impact running times and the quality of results. We compare families inferred by DiffMGC and DiffMGC$^{\approx\approx}_{\text{H}}$, considering genomes of *D. melanogaster*, *D. simulans* and *D. yakuba*, with 7, 6 and 6 linear chromosomes (after filtering out unlocalized contigs), respectively. DiffMGC cannot deal with large contig numbers and those were the only species assembled by FlyBase at chromosome level available on NCBI.

For DiffMGC$^{\approx\approx}_{\text{H}}$, the largest number of edges in $\widehat{\mathcal{C}}(\mathbb{A}, \mathbb{B})$ was for $A = D.\ melanogaster$ and $B = D.\ yakuba$. In this case, $\widehat{\mathcal{C}}$ has 10 edges distributed among 4 components with 2 vertices (including 1 dummy contig), and 1 component with 6 vertices. That corresponds to at most 11,520 capping-sets in $\theta_{\approx}$, while the three pairwise comparisons with the optimal $\theta_{\star}$ have between 12! and 14! capping-sets. Even so, the running times were very similar between the two approaches, varying from 15 to 30 seconds, probably due to the fact that these three species are phylogenetically closely related and very well assembled and annotated, allowing the solver to quickly identify the best ortholog-sets despite the much higher increase of the search space produced by $\theta_{\star}$.

Quality of inferred families

While 12,406 families were inferred using DiffMGC, 12,405 were inferred using DiffMGC$^{\approx\approx}_{\text{H}}$, and 99.8% of those families are the same. Consequently, only slight variations were found when comparing those families to the FlyBase families – 11,542 and 11,544 families are identical to those of FlyBase for DiffMGC and DiffMGC$^{\approx\approx}_{\text{H}}$, respectively.

5.2.2 Comparing DiffMGC$^{\approx\approx}_{\text{H}}$ to other tools

For comparing the performance of DiffMGC$^{\approx\approx}_{\text{H}}$ against other inference tools, we analyzed the complete dataset with 11 *Drosophila* genomes. Unlocalized contigs were not filtered out, resulting in genomes with 11 to 1,041 contigs.

Homologous families were also inferred by the following tools using the default parameters unless noted otherwise.

ProteinOrtho and Poff. ProteinOrtho [14] compares similarities of gene sequences and clusters them to find significant orthologous groups. To enhance the prediction accuracy, the Poff extension [15] can be used to take into account the relative order of genes as an additional feature for the discrimination of orthologs. We changed the parameter "minimum reciprocal similarity for additional hits" from the default 0.95 to 0.8 – experiments have shown that, as the "stringency threshold" parameter of DiffMGC, higher values might discard relevant gene relationships.

Oma. Based on sequence similarities and on phylogeny, Oma [1] is the underlying tool of the homonym online orthology browser. The standalone tool allows custom genomes to be compared to infer orthologous groups. We have also provided the phylogenetic tree of the 11 *Drosophilas* as input.

Quality of inferred families

Considering FlyBase families as reference, we compare the families inferred by Oma, ProteinOrtho, Poff and DiffMGC$\tilde{\tilde{\text{H}}}$ (Figure 5). All methods inferred more than 12,000 families. Since the number of families alone cannot hint the quality of results, we focus on the intersections with FlyBase families.

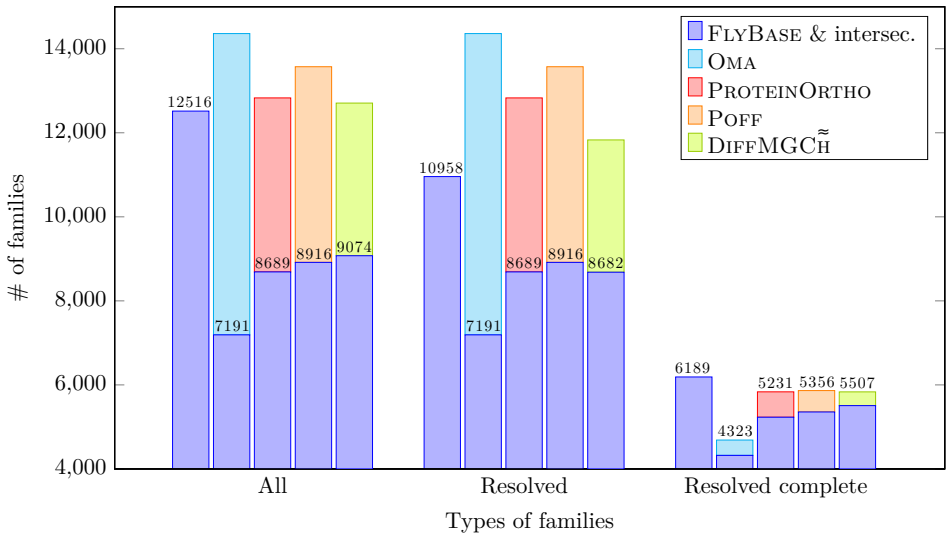


Figure 5 The numbers of all, resolved and resolved complete families in FlyBase, followed by the numbers of families inferred by Oma, ProteinOrtho, Poff and DiffMGC$\tilde{\tilde{\text{H}}}$, respectively. The lower part of each bar represents the intersection between the inferred sets and FlyBase. (For resolved complete families, the numbers of families in the intersections are shown on the top of bars.) More details on the distribution of families inferred by the four methods can be found in Appendix C.

All except DiffMGC$\tilde{\tilde{\text{H}}}$ produced only resolved families. For general and resolved families, 7,191 Oma families and FlyBase families are identical. ProteinOrtho, Poff and DiffMGC$\tilde{\tilde{\text{H}}}$ inferred 8,689, 8,916 and 9,074 families identical to those in FlyBase, respectively, while the intersection with FlyBase for resolved families decreased slightly for DiffMGC$\tilde{\tilde{\text{H}}}$ (8,682). For resolved complete families, however, the intersection of DiffMGC$\tilde{\tilde{\text{H}}}$ and FlyBase families is the largest again, with 5,507 families (89% of the FlyBase set), against, 4,323 (70%), 5,231 (85%) and 5,356 (87%) for Oma, ProteinOrtho and Poff respectively.

We also counted the numbers of gene homologies that are classified as *true positive* (TP), *false positive* (FP) and *false negative* (FN) with a procedure described in Appendix C. We then computed the values of *precision* $\left(\frac{\text{TP}}{\text{TP+FP}}\right)$ and *recall* $\left(\frac{\text{TP}}{\text{TP+FN}}\right)$ for the four methods. Our tool DiffMGC$\tilde{\tilde{\text{H}}}$ had the lowest value for precision (probably due to its unrefined ambiguous families) and the highest value for recall, which seems to be consistent to its high agreement with FlyBase families.

Running times

The running times are dominated by the preprocessing for all tools – the computation of pairwise sequence similarities. This step took more than 200 hours for Oma, which relies on an internal implementation of the Smith-Waterman algorithm, and 30 hours for DiffMGC$\tilde{\tilde{\text{H}}}$, which uses blast [2]. As for ProteinOrtho and its extension Poff, they took 45 minutes for performing sequence alignments with diamond [9].

Having at hand the results of alignments, Oma spent around 1 hour to output the inferred gene families, while ProteinOrtho and Poff spent 10 minutes. DiffMGC$\tilde{\tilde{\text{H}}}$ took 1 hour to postprocess alignments and generate ILPs, then less than 10 minutes to solve most of ILPs, totaling approximately 14 hours. Only 4 of the 55 ILPs reached the time limit of 2 hours, within a gap to the optimal solution of around 0.1% for 3 and 11% for 1 of them. Another round of postprocessing spent 5 minutes to generate families from the solver results.

6 Conclusions and Discussion

We devised and implemented a heuristic capping for improving our recently developed pipeline DiffMGC [18] for inferring gene families based on genome rearrangements. In DiffMGC we adopted an optimal capping including all connections between the ends of linear segments to allow all possible $(2p_*)!$ capping-sets in the input of the ILP FF-DCJ-Indel that infers the DiffM pairwise orthologs. However, due to the heavy optimal capping, FF-DCJ-Indel fails in handling a pair of genomes if one or both of them (with the dimension of a fruit fly genome) are distributed in a hundred contigs.

In contrast, the new pipeline DiffMGC$\tilde{\tilde{\text{H}}}$ adopts a lighter heuristic capping including connections only between linear segments that share gene content leading to a smaller number of capping-sets in the input of the same FF-DCJ-Indel that here infers DiffM$\tilde{\tilde{\text{H}}}$ pairwise orthologs. Data from experiments show that the heuristic capping can indeed be much lighter than the optimal capping, reducing drastically the search space. In the analysis of 11 *Drosophila* (partially assembled) genomes, the number of distinct capping-sets was reduced, on average, from (unfeasible) 1,014! to less than 305!. Although the latter value is still large, the heuristic capping allows FF-DCJ-Indel to efficiently handle a pair of fruit fly genomes where both of them can be distributed in hundreds or even thousands of contigs. The bottleneck of our pipeline is still the ILP pairwise computations that, despite the gain of heuristic capping, solve instances of an NP-hard problem. However, at least for genomes with the dimension of a fruit fly genome, DiffMGC$\tilde{\tilde{\text{H}}}$ lifts the limitation of requiring chromosome-level assembled genomes, expanding to a great extent its applicability.

Not only the genomes in contig-level could be analyzed, but also the quality of the inferred orthologies was very good. The quality evaluation was done by adopting the gene families curated by the FlyBase consortium as a benchmark. The analysis based on a smaller dataset of three completely assembled *Drosophila* genomes compared the previous workflow DiffMGC with the new DiffMGC$\tilde{\tilde{\text{H}}}$ and showed that the gene families inferred by the two pipelines are virtually the same. The heuristic capping in practice did not have a negative impact on the inferred gene families, preserving the original (optimal) orthology relations. A larger experimental study based on 11 *Drosophila* genomes, including partially assembled genomes distributed in several contigs, compared DiffMGC$\tilde{\tilde{\text{H}}}$ to other genome-scale methods, namely Oma, ProteinOrtho and Poff. Our results showed that DiffMGC$\tilde{\tilde{\text{H}}}$ was able to infer the highest number of families and complete families in common with FlyBase, and was very close to the top on the number of incomplete resolved families. Indeed, our

tool had the highest value for recall, which seems to be consistent to its high agreement with FlyBase families. However, for precision DiffMGC$\tilde{\tilde{\text{H}}}$ had the lowest value, probably due to its 877 unrefined ambiguous families, the largest of them including 151 genes.

As a future work we intend to refine our ambiguous families by breaking them into smaller families, so that we can improve our precision without losing in our recall rate. We will also replace BLAST by DIAMOND in our pipeline, bringing its preprocessing running times closer to those of ProteinOrtho and Poff. This will allow us to more efficiently evaluate our tool with datasets including larger genomes. Additionally we intend to do a systematic study testing the speed and the quality performances of the heuristic capping for a range of distinct values of the parameters τ and ϵ, so that we can derive a way to estimate good values for these parameters considering the input datasets.

References

1 Adrian M Altenhoff, Jeremy Levy, Magdalena Zarowiecki, Bartłomiej Tomiczek, Alex Warwick Vesztrocy, Daniel A Dalquen, Steven Müller, Maximilian J Telford, Natasha M Glover, David Dylus, et al. OMA standalone: orthology inference among public and custom genomes and transcriptomes. *Genome Res*, 29(7):1152–1163, 2019.

2 Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, 1990.

3 Sébastien Angibaud, Guillaume Fertin, Irena Rusu, Annelyse Thévenin, and Stéphane Vialette. On the approximability of comparing genomes with duplicates. *J Graph Algo App*, 13(1):19–53, 2009. doi:10.7155/jgaa.00175.

4 Anne Bergeron, Julia Mixtacki, and Jens Stoye. A unifying view of genome rearrangements. In *Proc. of WABI*, volume 4175 of *Lecture Notes in Bioinformatics*, pages 163–173, 2006. doi:10.1007/11851561_16.

5 Leonard Bohnenkämper, Marília D. V. Braga, Daniel Doerr, and Jens Stoye. Computing the rearrangement distance of natural genomes. *J Comput Biol*, 28(4):410–431, 2021. doi:10.1089/cmb.2020.0434.

6 Marília D. V. Braga, Cedric Chauve, Daniel Doerr, Katharina Jahn, Jens Stoye, Annelyse Thévenin, and Roland Wittler. The potential of family-free genome comparison. In C. Chauve, N. El-Mabrouk, and E. Tannier, editors, *Models and Algorithms for Genome Evolution*, volume 19 of *Computational Biology Series*, chapter 13, pages 287–307. Springer Verlag, Berlin, 2013. doi:10.1007/978-1-4471-5298-9_13.

7 Marília D. V. Braga, Eyla Willing, and Jens Stoye. Double cut and join with insertions and deletions. *J Comput Biol*, 18(9):1167–1184, 2011. doi:10.1089/cmb.2011.0118.

8 David Bryant. The complexity of calculating exemplar distances. In David Sankoff and Joseph H. Nadeau, editors, *Comparative Genomics*, volume 1 of *Computational Biology Series*, pages 207–211. Kluver Academic Publishers, London, 2000. doi:10.1007/978-94-011-4309-7_19.

9 Benjamin Buchfink, Chao Xie, and Daniel H. Huson. Fast and sensitive protein alignment using DIAMOND. *Nat Methods*, 12:59–60, 2015.

10 C. Dessimoz, G. Cannarozzi, M. Gil, D. Margadant, A. C. J. Roth, A. Schneider, and G. H. Gonnet. OMA, a comprehensive, automated project for the identification of orthologs from complete genome data: introduction and first achievements. In *Proc. of RECOMB-CG*, volume 3678 of *Lecture Notes in Bioinformatics*, pages 61–72, 2005.

11 Daniel Doerr, Pedro Feijão, and Jens Stoye. Family-free genome comparison. In João C. Setubal, Jens Stoye, and Peter F. Stadler, editors, *Comparative Genomics: Methods and Protocols*, volume 1704 of *Methods in Molecular Biology*, pages 331–342. Springer Nature, New York, 2018. doi:10.1007/978-1-4939-7463-4_12.

12 P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, s1-10(1):26–30, 1935.

13 Sridhar Hannenhalli and Pavel A. Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). In *Proc. of FOCS*, pages 581–592, 1995. doi:10.1109/SFCS.1995.492588.

14 Marcus Lechner, Sven Findeiß, Lydia Steiner, Manja Marz, Peter F. Stadler, and Sonja J. Prohaska. Proteinortho: Detection of (co-)orthologs in large-scale analysis. *BMC Bioinform*, 12(124), 2011.

15 Marcus Lechner, Maribel Hernandez-Rosales, Daniel Doerr, Nicolas Wieseke, Annelyse Thévenin, Jens Stoye, Roland K. Hartmann, Sonja J. Prohaska, and Peter F. Stadler. Orthology detection combining clustering and synteny for very large datasets. *PLoS One*, 9(8:e105015), 2014.

16 Fábio V. Martinez, Pedro Feijao, Marília D. V. Braga, and Jens Stoye. On the family-free DCJ distance and similarity. *Algorithms Mol Biol*, 13(10), 2015. doi:10.1186/s13015-015-0041-9.

17 Alexander C. J. Roth, Gaston H. Gonnet, and Christophe Dessimoz. Algorithm of OMA for large-scale orthology inference. *BMC Bioinform*, 9(518), 2008.

18 Diego P. Rubert, Daniel Doerr, and Marília D. V. Braga. The potential of family-free rearrangements towards gene orthology inference. *J Bioinform Comput Biol*, 19(6):2140014, 2021. doi:10.1142/S021972002140014X.

19 Diego P. Rubert, Fábio V. Martinez, and Marília D. V. Braga. Natural Family-Free Genomic Distance. *Algorithms Mol Biol*, 16(4), 2021. doi:10.1186/s13015-021-00183-8.

20 David Sankoff. Genome rearrangement with gene families. *Bioinformatics*, 15(11):909–917, 1999. doi:10.1093/bioinformatics/15.11.909.

21 Mingfu Shao, Yu Lin, and Bernard Moret. An exact algorithm to compute the double-cut-and-join distance for genomes with duplicate genes. *J Comput Biol*, 22(5):425–435, 2015. doi:10.1089/cmb.2014.0096.

22 Guanqun Shi, Liqing Zhang, and Tao Jiang. MSOAR 2.0: Incorporating tandem duplications into ortholog assignment based on genome rearrangement. *BMC Bioinform*, 11(10), 2010.

23 Tamir Tassa. Finding all maximally-matchable edges in a bipartite graph. *Theoretical Computer Science*, 423:50–58, 2012.

24 Sophia Yancopoulos, Oliver Attie, and Richard Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16):3340–3346, 2005. doi:10.1093/bioinformatics/bti535.

A ILP formulation for family-free DCJ-indel

This ILP was developed in our previous work [19], but there it only considered an optimal capping of a family-free relational graph $FFR(\mathbb{A}, \mathbb{B}, \mathcal{S})$. It is an adaptation of the ILP for computing the DCJ-indel distance of family-based natural genomes, by Bohnenkämper *et al.* [5], which is itself an extension of the ILP for computing the DCJ distance of family-based balanced genomes, by Shao *et al.* [21].

Given a valid capping θ (that here can be $\theta_\star$ or $\theta_\approx$), the general idea is searching for a sibling-set that, together with a capping-set, induces an optimal consistent capped decomposition of the capped diagram $\theta(FFR(A, B, \mathcal{S})) = (V, E)$, where $V = V_\gamma^{\mathbb{A}} \cup V_\gamma^{\mathbb{B}} \cup \hat{\theta}(\mathbb{A}) \cup \hat{\theta}(\mathbb{B})$ and the set of edges comprises all disjoint sets of distinct edge types: $E = E_\gamma \cup E_\theta \cup E_{\text{adj}}^{\mathbb{A}} \cup E_{\text{adj}}^{\mathbb{B}} \cup E_{\text{id}}^{\mathbb{A}} \cup E_{\text{id}}^{\mathbb{B}}$. Therefore the same ILP formulation (shown in Algorithm 2) computes either $\textsc{Diff}(A, B, \mathcal{S})$ (with $\theta_\star$) or $\textsc{Diff}_{\text{H}}^{\approx}(A, B, \mathcal{S})$ (with $\theta_\approx$). A particular feature of this ILP when compared to those from [5] and [21] is that its search space includes all sibling-sets, of any size.

For capturing the properties required for computing the best DCJ-indel weighted cost, whose details can be found in [18, 19], the ILP is distributed in three main parts. Counting indel-free cycles (those without indel edges) makes up the first part, depicted in constraints

Algorithm 2 FF-DCJ-INDEL: ILP for computing the best DCJ-indel weighted cost.

Input: A family-free relational graph $FFR(\mathbb{A}, \mathbb{B}, \mathbb{S})$ with a valid capping $\theta_\star$ or $\theta_\approx$.

$$\text{min} \quad p + \sum_{e \in E_\gamma} x_e - \sum_{1 \leq i \leq |V|} z_i + \sum_{k \in K} s_k + \frac{1}{2} \sum_{e \in E} t_e - \frac{1}{2} \sum_{e \in E_\gamma} w_e x_e + \sum_{e \in E_{\text{id}}} w_e x_e$$

$$\begin{aligned}
\text{s. t.} \quad & x_e = 1 && \forall\, e \in E^{\mathbb{A}}_{\text{adj}} \cup E^{\mathbb{B}}_{\text{adj}} && \text{(C.01)}\\
& \sum_{uv \in E} x_{uv} = 2 && \forall\, u \in V && \text{(C.02)}\\
& x_e = x_d && \forall\, e, d \in E_\gamma,\ e, d \text{ are siblings} && \text{(C.03)}\\
& \left.\begin{aligned} y_i &\leq y_j + i(1 - x_{v_i v_j}) \\ y_j &\leq y_i + j(1 - x_{v_i v_j}) \end{aligned}\right\} && \forall\, v_i v_j \in E && \text{(C.04)}\\
& \left.\begin{aligned} y_i &\leq i(1 - x_{v_i v_j}) \\ y_j &\leq j(1 - x_{v_i v_j}) \end{aligned}\right\} && \forall\, v_i v_j \in E^{\mathbb{A}}_{\text{id}} \cup E^{\mathbb{B}}_{\text{id}} && \text{(C.05)}\\
& i z_i \leq y_i && \forall\, 1 \leq i \leq |V| && \text{(C.06)}\\
& \left.\begin{aligned} r_v &\leq 1 - x_{uv} \\ r_{v'} &\geq x_{u'v'} \end{aligned}\right\} && \begin{aligned}&\forall\, uv \in E^{\mathbb{A}}_{\text{id}}\\ &\forall\, u'v' \in E^{\mathbb{B}}_{\text{id}}\end{aligned} && \text{(C.07)}\\
& \left.\begin{aligned} t_{uv} &\geq r_v - r_u - (1 - x_{uv}) \\ t_{uv} &\geq r_u - r_v - (1 - x_{uv}) \end{aligned}\right\} && \forall\, uv \in E && \text{(C.08)}\\
& \sum_{d \in E^{\mathbb{A}}_{\text{id}},\ d \cap e \neq \emptyset} x_d - t_e \geq 0 && \forall\, e \in E^{\mathbb{A}}_{\text{adj}} && \text{(C.09)}\\
& t_e = 0 && \forall\, e \in E \setminus E^{\mathbb{A}}_{\text{adj}} && \text{(C.10)}\\
& \sum_{e \in E^k_{\text{id}}} x_e - |k| \leq s_k && \forall\, k \in K && \text{(C.11)}
\end{aligned}$$

$$\begin{aligned}
\text{and} \quad & x_e \in \{0, 1\} && \forall\, e \in E && \text{(D.01)}\\
& 0 \leq y_i \leq i && \forall\, 1 \leq i \leq |V| && \text{(D.02)}\\
& z_i \in \{0, 1\} && \forall\, 1 \leq i \leq |V| && \text{(D.03)}\\
& r_v \in \{0, 1\} && \forall\, v \in V && \text{(D.04)}\\
& t_e \in \{0, 1\} && \forall\, e \in E && \text{(D.05)}\\
& s_k \in \{0, 1\} && \forall\, k \in K && \text{(D.06)}\\
& p = p_* \text{ (optimal capping) or } p_\approx \text{ (heuristic capping)} && && \text{(D.07)}
\end{aligned}$$

(C.01)–(C.06), variables and domains (D.01)–(D.03). The second part is for counting transitions (paths between an indel edge in $\mathbb{A}$ and an indel edge in $\mathbb{B}$, described in constraints (C.07)–(C.10), variables and domains (D.04)–(D.05). The last part describes how to count the number of circular singletons (circular chromosomes exclusively composed of indel and adjacency edges) with constraint (C.11), variable and domain (D.06). The objective function of our ILP minimizes the size of the sibling-set (that is twice the size of the ortholog-set), with sum over variables x_e, the number of circular singletons, calculated by the sum over variables s_k, half the overall number of transitions in indel-enclosing (non-singletons) cycles, calculated by the sum over variables t_e, and the weight of all indel edges in the decomposition, given by the sum over their weights $w_e x_e$ for all $e \in E_{\text{id}}$, while maximizing both the number of indel-free cycles, counted by the sum over variables z_i, and half of the weight of the sibling-set. Note that the minimization is not affected by constant p that corresponds to p_* when the graph is optimally capped or to $p_\approx$ when the graph is heuristically capped.

B Pipeline description

Both the previous DiffMGC with optimal capping and the new DiffMGC$\tilde{\tilde{\text{H}}}$ with heuristic capping can be summarized in the following pipeline. Given a set of n genomes, for all pairwise comparisons gene similarities and ortholog-sets are computed with the ILP FF-DCJ-Indel solving either DiffM (for DiffMGC) or DiffM$\tilde{\tilde{\text{H}}}$ (for DiffMGC$\tilde{\tilde{\text{H}}}$). The resulting pairwise ortholog-sets are then simply integrated into an n-partite graph, and the connected components of this graph are the inferred gene families.

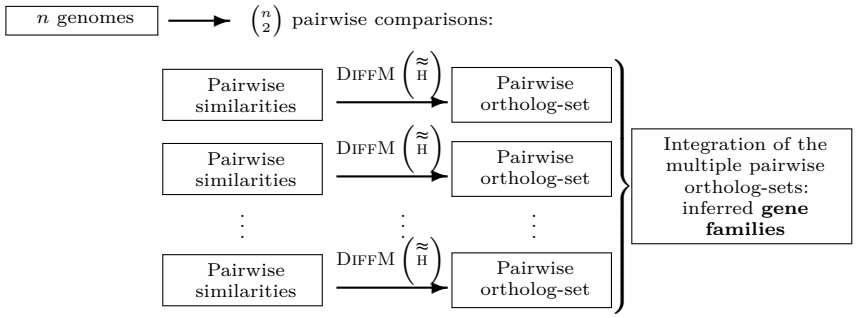


Figure 6 The pipeline of our approach is straightforward: our gene families are the connected components of the n-partite graph derived by the integration of the computed ortholog-sets.

An *ambiguous* family corresponds to a connected component of the n-partite graph that has more than one gene from the same genome. Otherwise we have a *resolved* family, which can be either *complete*, when it contains one gene per genome, or *incomplete* otherwise. Figure 7 illustrates these types of families in a 3-partite graph.

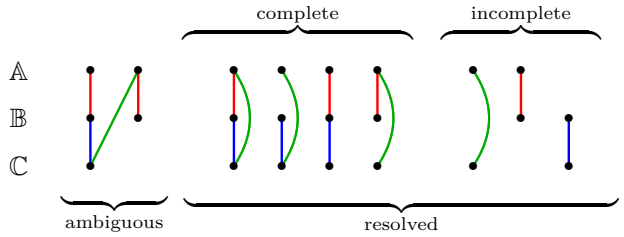


Figure 7 Types of families given by the integration of three ortholog-sets into a 3-partite graph.

C Supplementary information on the analysis of eleven *Drosophilas*

First we show in Figure 8 the distribution of the numbers of resolved FlyBase families (per family size) inferred by each of the four methods.

The numbers of homologies that are classified as *true positive* (TP), *false positive* (FP) and *false negative* (FN) were obtained as follows. Let $\mathcal{H}$ be the subsets of size two of all FlyBase families. For a given method, let $\mathcal{M}$ be the subsets of size two of all inferred families. Then TP is the size of $\mathcal{H} \cap \mathcal{M}$, FN is the size of $\mathcal{H} \setminus \mathcal{M}$ and FP is the size of $\mathcal{M} \setminus \mathcal{H}$. For the four methods we calculated *precision* $\left(\frac{\text{TP}}{\text{TP+FP}}\right)$ and *recall* $\left(\frac{\text{TP}}{\text{TP+FN}}\right)$. The results (Fig. 9) show that DiffMGC$\tilde{\tilde{\text{H}}}$ (with 877 ambiguous families, the largest of size 151) had the lowest precision and the highest recall, which is consistent with its agreement with FlyBase.

Finally, to give an idea of how many families the four methods have in common, we show the pairwise intersections in Table 1.

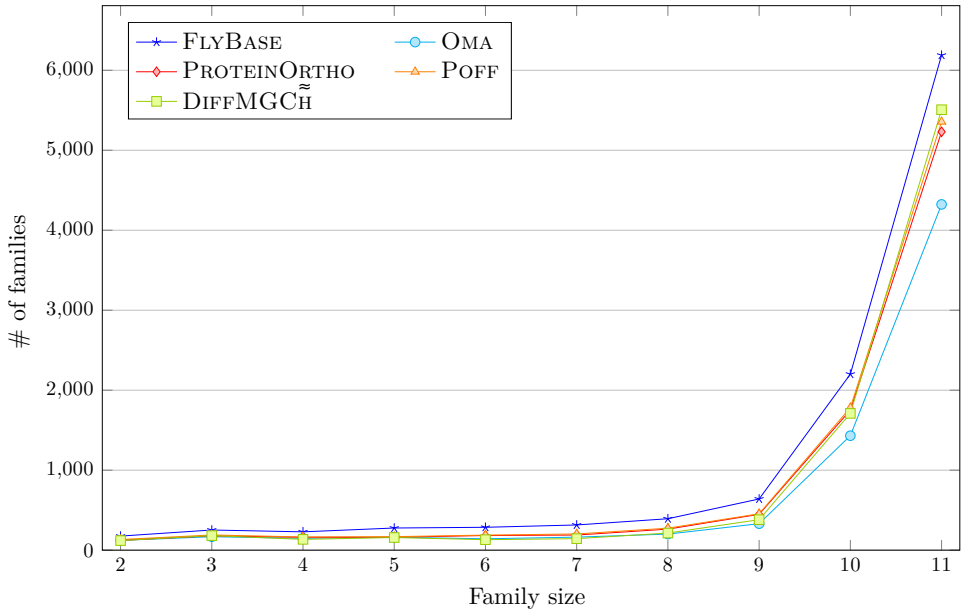


Figure 8 Distribution of resolved families in common with FlyBase families for eleven *Drosophilas*.

Method	TP	FP	FN	precision	recall	F_1-score
Oma	500,500	1,618	114,343	0.997	0.814	0.90
ProteinOrtho	522,462	4,744	92,381	0.991	0.849	0.91
Poff	531,383	3,645	83,460	0.993	0.864	0.92
DiffMGC≈H	561,002	74,801	53,841	0.882	0.912	0.90

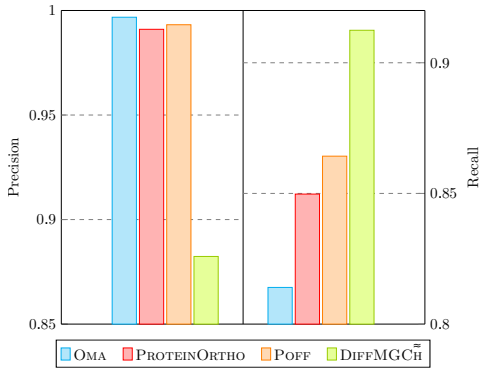


Figure 9 Precision $\left(\frac{TP}{TP+FP}\right)$, recall $\left(\frac{TP}{TP+FN}\right)$ and their harmonic mean F_1-score for Oma, ProteinOrtho, Poff and DiffMGC≈H, based on the dataset with eleven *Drosophilas*.

Table 1 Comparison of inferred families considering eleven *Drosophila* species. Each cell in the diagonal holds the number of families given by a method and the number of those families that are identical to a FlyBase family. The remaining cells show the number of families in common between two methods and the percentages separated by a dot show the proportions with respect to the total number of families given by the method in the corresponding row and column, respectively.

common resolved incomplete families (FlyBase has 4,769 res. incomplete families)

	DiffMGC$_H^{\approx}$	Poff	ProteinOrtho	Oma
DiffMGC$_H^{\approx}$	5,996 100% 3,175 (53%)	3,380 56%.44%	3,209 57%.46%	2,899 48%.30%
Poff	- - -	7,707 100% 3,551 (46%)	5,751 75%.82%	4,235 55%.44%
ProteinOrtho	- - -	- - -	6,995 100% 3,458 (49%)	4,127 59%.43%
Oma	- - -	- - -	- - -	9,673 100% 2,863 (40%)
∩FlyBase				

common resolved complete families (FlyBase has 6,189 res. complete families)

	DiffMGC$_H^{\approx}$	Poff	ProteinOrtho	Oma
DiffMGC$_H^{\approx}$	5,834 100% 5,507 (94%)	5,177 89%.88%	5,026 86%.86%	4,257 73%.91%
Poff	- - -	5,865 100% 5,356 (91%)	5,501 94%.94%	4,468 76%.95%
ProteinOrtho	- - -	- - -	5,835 100% 5,231 (89%)	4,369 75%.93%
Oma	- - -	- - -	- - -	4,688 100% 4,328 (92%)
∩FlyBase				

Toward Optimal Fingerprint Indexing for Large Scale Genomics

Clément Agret[1]
Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
LIRMM, Univ Montpellier, CNRS, Montpellier, France

Bastien Cazaux[1]
Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Antoine Limasset[1,2] ✉
Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Abstract

Motivation. To keep up with the scale of genomic databases, several methods rely on local sensitive hashing methods to efficiently find potential matches within large genome collections. Existing solutions rely on Minhash or Hyperloglog fingerprints and require reading the whole index to perform a query. Such solutions can not be considered scalable with the growing amount of documents to index.

Results. We present NIQKI, a novel structure with well-designed fingerprints that lead to theoretical and practical query time improvements, outperforming state-of-the-art by orders of magnitude. Our contribution is threefold. First, we generalize the concept of Hyperminhash fingerprints in (h,m)-HMH fingerprints that can be tuned to present the lowest false positive rate given the expected sub-sampling applied. Second, we provide a structure able to index any kind of fingerprints based on inverted indexes that provide optimal queries, namely linear with the size of the output. Third, we implemented these approaches in a tool dubbed NIQKI that can index and calculate pairwise distances for over one million bacterial genomes from GenBank in a few days on a small cluster. We show that our approach can be orders of magnitude faster than state-of-the-art with comparable precision. We believe this approach can lead to tremendous improvements, allowing fast queries and scaling on extensive genomic databases.

2012 ACM Subject Classification Applied computing → Bioinformatics

Keywords and phrases Data Structure, Indexation, Local Sensitive Hashing, Genomes, Databases

Digital Object Identifier 10.4230/LIPIcs.WABI.2022.25

Related Version *Full Version*:
`https://www.biorxiv.org/content/10.1101/2021.11.04.467355v1`

Supplementary Material We wrote the NIQKI index as an open-source C++ library under the AGPL3 license. It is designed as a user-friendly tool and comes along with usage samples.
Software (Source Code): `https://github.com/Malfoy/NIQKI`
archived at `swh:1:dir:4b130954e11adff2be9108f45c4181f972604407`

Acknowledgements We want to thank Camille Marchet, Pierre Doignies, organizers and participants of the Bioinformatics: from Algorithms to Applications conference, for their support and discussions on this project. The ANR SEQdigger supported this work.

[1] All authors have contributed equally.
[2] Corresponding author

Introduction

Historically, genomic databases such as GenBank are growing exponentially[3]. Lower sequencing costs and required investment, broader access to sequencing technologies, and breakthrough in genome assembly practices will surely fuel the explosion of available genomes in the near future. While those data are still widely unprocessed, the ability to explore and delve into such rich archives presents countless applications [4, 1]. Allowing to query such databases at a reasonable cost is a growing research subject [3, 13, 15, 16]. To avoid relying on computation-intensive steps, an efficient way to approximate the similarity of two sequences is to represent them as a set of k-mer (sub-word of length k) and compare their k-mer contents. Namely, the fraction of shared k-mer that corresponds to the Jaccard index is a good proxy for Average Nucleotide Identity [14] between genomes. Given the scale of such collections, indexing complete genomes quickly become prohibitively expensive. When searching for broad matches between large genomic sequences such as genomes, dimension reduction techniques can be used to reduce the computational burden.

Minhash [5] is a very resource-efficient technique able to estimate the Jaccard index between two sets. Minhash constructs a sketch of S fingerprints chosen from the hashed elements of a given set of size N (e.g., selecting the smallest hash values). The attractive property of such sketches is that the Jaccard index between them approximates the actual Jaccard index of the two sets they represent. This way, the Jaccard index can be estimated by comparing two sketches of size S that can be orders of magnitude smaller than the actual cardinality of the indexed sets.

Three principal variants of Minhash have been proposed.

- S **hash functions.** Each element is hashed using S distinct hash functions $h_1 \dots h_S$, the sketch is composed of the smallest hash output by each hash function. The sketches can be compared in $\Theta(S)$ and constructing sketches is $\Theta(N.S)$
- S **minimal values.** Each element is hashed using a single hash function h. The sketch comprises the S smallest hashes output by h. The sketch construction is reduced to $\Theta(N.\log(S))$ but the sketch comparison is $\Theta(S.\log(S))$
- S **partitions.** Each element is hashed using a single hash function h. The hash values are split into S partitions according to their first bits. The sketch is composed of the minimal element of each partition. The sketch construction is reduced to $\Theta(N)$ and the sketch comparison to $\Theta(S)$.

While S partition seems optimal, some partitions can be empty if no hashes start with a given prefix. Some process dubbed densification [17] aims to cope with this problem by using other partition data to fill empty partitions.

The other parameter of a Minhash sketch is the size of each stored hash. Usually, a "large" hash size (e.g., 32 bits) is used to avoid the risk of collisions or saturation. Collisions occur when two different keys have the same hash value, leading to false-positive hits. Since the smallest hash values are kept, all fingerprints tend to zero. Once the zero value is reached, it can no longer change. We call such fingerprints saturated. A comparison of saturated sketches can report a large amount of false-positive hits.

[3] `https://www.ncbi.nlm.nih.gov/genbank/statistics/`

However, since the minimal values are kept, the first bits of each fingerprint may contain a large amount of low-informative zeroes. The b-bit Minhash [9] variation takes advantage of this observation and only keeps the b lowest bits to reduce the sketches sizes by reducing the fingerprints sizes. It can either improve the memory footprint of sketches or their accuracy by increasing the number of fingerprints used for the same amount of memory.

Those techniques were successfully applied in bioinformatics to index and compare genomes databases. Mash [14] implemented S minimal values and Bindash [19] implement S partitions, b-bit Minhash with densification. Such works showed that genomes and whole sequencing datasets could be precisely compared by using sketches orders of magnitude smaller than their amount of distinct k-mers leading to tremendous performance improvement when working at a large scale.

An alternative to Minhash fingerprints is Hyperloglog [6] fingerprints. Instead of storing the whole hash, Hyperloglog fingerprints store its log value (e.g., the position of the leading 1). The interest in such fingerprints is twofold. They are tiny: a 64bits hash needs a 6bits Hyperloglog fingerprint (since $2^6 = 64$) and is as hard to saturate as their associated hashes. Six bits seem enough to cover most real-world cases in practice, as 64bit hashes seem unlikely to be saturated. The downside is that their collision rate is very high. This fact can be leveraged by using a large number of fingerprints. Dashing [2] showed that large S partitions sketches of Hyperloglog fingerprints could lead to very precise estimations of Jaccard index between genomes.

Interestingly, another approach tried to benefit from the trailing zeroes by combining the b-bit Minhash and Hyperloglog ideas. Hyperminhash [18] builds a fingerprint from a given hash using 6 bits to encode the first run of zero (which is equivalent to a Hyperloglog fingerprint) and combines them with the lowest 10 bits (which is equivalent to a b-bit Minhash fingerprint) to obtain 16 bits fingerprints. The interest of this sketch is to be very hard to saturate due to the Hyperloglog fingerprint and to limit the number of collisions with the additional Minhash bits. Such fingerprint can be seen as an efficient lossy compression of the hash value that represents more than 16 bits from the hash (if the run length of zeroes is longer than 6).

In bioinformatics, such fingerprint indexes have numerous applications such as very fast clustering or phylogeny construction from large genome collections, finding potentially related genomes from assembled contigs or sequences of interest, quantification of a given gene, strain, or species in databases etc... Their main feature is to find possible matches for a query sequence inside a database, acting as filters to focus only on relevant entries. They do so by efficiently estimating similarities between query and index sequences.

The main bottleneck of existing methods is that a query requires the queried sketch to be compared to all sketches in the database. The query is $\mathcal{O}(N.S)$ with N the number of indexed entries. This cannot be considered scalable regarding the exponential growth of available genomic resources. To cope with this algorithmic problem, we design a structure to perform queries in $\Theta(\#hits)$ where $\#hits$ is the number of fingerprint hits between the queried sketch and the indexed sketches. Since the expected output of a query is a list of genomes identifier associated with a number of hits (or an approximation of the Jaccard index), we can consider our query time as close to optimal.

However, such a structure presents an overhead exponential with the fingerprint size. For this reason, we are highly attentive to using powerful but small fingerprint sizes. We generalize the Hyperminhash fingerprint and introduce (h, m)-Hyperminhash fingerprints that cover both Minhash and Hyperloglog cases and provide an analysis to select the best fingerprint for a given use case.

Methods

Index structure

To generalize the concept of indexing a fixed amount of fingerprints (either Hyperminhash, b-bit Minhash, or Hyperloglog), we call such a tuple of fingerprints a sketch. In the following, we consider bit-vectors of size n and integers between 0 and $2^n - 1$ as equivalent. For the sake of clarity, log() will be used for $\log_2()$ in this document.

Let W, S, and N be three integers, where N will be the number of sketches, S the size of the sketches, which is the number of fingerprints in each sketch, and w the size in bits of each fingerprint. More specifically, a *sketch* s_i of size S is a tuple of S fingerprints where a fingerprint is an element between 0 and $2^W - 1$, i.e. $s_i = f_{i,1} \dots f_{i,S}$ where $\forall j \in \{1, \dots, S\}$, $f_{i,j} \in \{0, \dots, 2^W - 1\}$. We present a new data structure to index a set $B = \{s_1, \dots, s_N\}$ of N sketches where we want to optimize the time complexity of the query $\texttt{find\_hits}(\mathtt{B}, \mathtt{s}', \mathtt{m})$ which corresponds to find all the sketches $s_i = f_{i,1} \dots f_{i,S}$ of B such that $|\{j \in \{1, \dots, S\} \mid f_{i,j} = f'_j\}| \geq m$ with $s' = f'_1 \dots f'_S$ a sketch and m an integer.

Instead of indexing each sketch, we split the set of sketches B by column j between 1 and S. We denote by P_j the sequence of fingerprints $f_{1,j} \dots f_{N,j}$. Unlike previous approaches, our scheme handles each column P_j independently by grouping the indices with the same fingerprint on this column. Indeed, for each column j between 1 and S, we index in a vector the sets $X_{j,0} \dots X_{j,2^W-1}$ where for all $l \in \{0, \dots, 2^W - 1\}$, $X_{j,l} = \{i \in \{1, \dots, n\} \mid f_{i,j} = l\}$. As $X_{j,0} \dots X_{j,2^W-1}$ is a partition of $\{1, \dots, N\}$, our vector have $\mathcal{O}(N + 2^W)$ elements (see Figure 1).

We argue that performing queries by partition presents several advantages. One advantage of this processing partition by column is the possibility to stop queries that can not find high-scoring matches. If we are looking for a minimum of m shared fingerprint among S to report a match, if after $P_{|S|-j}$ no sketches share at least $m - j$ fingerprints, that query can be stopped as no matches can reach the minimal score. If not using the densification technique, empty partitions could also be skipped.

As, for our case, a sketch represents a genome, and the fingerprints come from the set of k-mers of this genome, one possible mechanism presented in [10] inspired by [7] is to insert the k-mers of a given partition in a Bloom filter (or any set structure) to "protect" each partition from alien k-mers. If the selected k-mer is not in the partition set, the partition is skipped as we know that the k-mer was not inserted in the partition and that any fingerprint hit would be a false positive.

We use dense addressing to index our partitions in a trade-off favoring code simplicity and throughput over memory footprint. This way, our index has an overhead of $\Theta(S.2^W)$ as we allocate $S.2^W$ vectors to represent all possible values of each partition. Once the index created, inserting the nth sketch $s_n = f_1 \dots f_S$ is inserting n in the vector $f_i + i.2^w$. The inserted elements cost $\Theta(N.\log(N))$ space and each partition can be queried for $\texttt{find\_hits}(\mathtt{P_j}, \mathtt{f'_j}, \mathtt{1})$ in $\Theta(\#hits)$ which corresponds to $\Theta(|\{i \in \{1, \dots, N\} \mid f_{i,j} = f'_j\}|)$. As $\texttt{find\_hits}(\mathtt{B}, \mathtt{s}', \mathtt{m}) = \{i \in \{1, \dots, N\} \mid \sum_{j=1}^{N} \mathbb{1}_{\texttt{find\_hits}(\mathtt{P_j}, \mathtt{f'_j}, \mathtt{1})}(j) \geq m\}$, by computing all the $\texttt{find\_hits}(\mathtt{P_j}, \mathtt{f'_j}, \mathtt{1})$, we can compute $\texttt{find\_hits}(\mathtt{B}, \mathtt{s}', \mathtt{m})$.

If this approach presents a costly overhead, its interest is that query a sketch performs at worst S "costly" random access. We argue that using relatively small, well-designed fingerprints, our scheme can perform swift queries with a reasonable memory footprint.

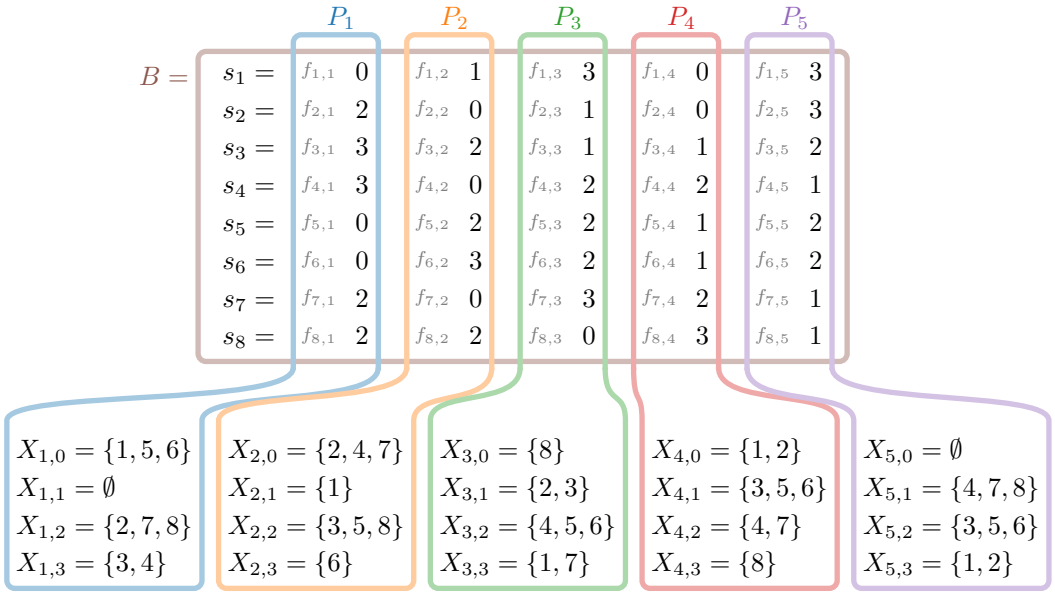


Figure 1 Example of processing partitions of 8 (N) sketches where each sketch has 5 (S) fingerprints of 2 (W) bits. $B = \{01303, 20103, 32112, 30221, 02212, 03212, 20321, 22031\}$ is the set of sketches used as instance where the sketch $s_1 = 01303$ corresponds to the tuple of the 5 fingerprints where the first fingerprint is 0, the second fingerprint is 1, the third fingerprint is 3 and so on. To represent the decomposition by column, we add specific color. For the first column (in blue) P_1 corresponds to all the fingerprints seen as the first position of a sketch of B (the first partition) and $X_{1,vf}$ contain the set of sketch identifiers (positions of the sketches in B) whose first fingerprint value is vf. We obtain $S \times 2^W = 20$ sets $X_{j,l}$ ($1 \leq j \leq S$ and $0 \leq l \leq 2^W - 1$) corresponding to the stored sets.

Optimal Hyperminhash fingerprint

Since our proposed scheme presents an overhead exponential with W, we need to rely on very space-efficient fingerprints and make the most of each bit. When building a fingerprint database, the users tune the sketches' size according to the expected genome sizes they index and the needed precision according to their application. This parameter also changes the fingerprint values as we expect each fingerprint to be the minimum across $\approx \frac{genome\_size}{sketch\_size}$ that we call the sub-sampling value. In the section, we argue that different sub-sampling values call for different fingerprints. This part presents a parametric fingerprint that generalizes Hyperminhash, b-bit minhash, and Hyperloglog. We give a method to fix this parameter to optimize the false positive rate for a given fingerprint size.

For a bitvector/hash B of size n and two integers h and m such that $h + m = W$, the (h, m)-hyperminhash fingerprint of B, denoted by $\texttt{HMH}_{h,m}(B)$, is the bitvector of size $h + m$ where the h first bits correspond to the position to the first one on the prefix of B of size $2^h - 1$ and the m last bits are the last m bits of B, i.e.

$$\texttt{HMH}_{h,m}(B) = \max(2^h + \lfloor \log(B) \rfloor - n, 0) \times 2^m + (B \bmod 2^m)$$

For a set X of bitvectors of size n, the (h, m)-hyperminhash fingerprint of X is the minimum value of the (h, m)-hyperminhash fingerprint of B for all the bitvectors B of X, i.e. $\texttt{HMH}_{h,m}(X) = \min_{B \in X} \texttt{HMH}_{h,m}(B)$.

Our goal is to find the pair h, m where $h+m=W$ which gives us the most space-efficient fingerprint, i.e. the fingerprint which covers the most larger interval of values. One way to compute this space-efficiency is for a threshold ε of $[0,1]$, to compute $c_{h,m} = b_{h,m} - a_{h,m} + 1$ which is the size of the interval $\{a_{h,m}, \ldots, b_{h,m}\}$ where

$$\begin{array}{rcl} a_{h,m} & = & \max_{k\in\{0,\ldots,2^W-1\}}(\mathbb{P}[\texttt{HMH}_{h,m}(X) \leq k] \leq \frac{\varepsilon}{2}) \\ b_{h,m} & = & \min_{k\in\{0,\ldots,2^W-1\}}(\mathbb{P}[\texttt{HMH}_{h,m}(X) \leq k] \geq 1-\frac{\varepsilon}{2}) \end{array}$$

and thus $\mathbb{P}[a_{h,m} \leq \texttt{HMH}_{h,m}(X) \leq b_{h,m}] \leq 1-\varepsilon$.

To begin, if $n < 2^h - 1 + m$, some bits can be in both prefix and suffix of the fingerprint due to the overlap, and thus it is not space-efficient. Indeed, we can prove that the number of fingerprint values that are not reached is equal to $2^m \times (2^h - 1 + m - n)$. For this reason, we take in all the following $n \geq 2^h - 1 + m$.

For an integer k of $\{0, \ldots, 2^{h+m}\}$, we denote by $H(h,m,k)$ the number of $B \in \{0, \ldots, 2^n - 1\}$ such that $\texttt{HMH}_{h,m}(B) = k$. As (h,m)-hyperminhash is an increasing function because if $B \leq B'$, we have $\texttt{HMH}_{h,m}(B) \leq \texttt{HMH}_{h,m}(B')$, by counting, we can show that $H(h,m,k) = 2^{n-2^h+1-m+\max(0,\lfloor\frac{k}{2^m}\rfloor-1)}$. Indeed, each B of $\{0, \ldots, 2^n-1\}$ such that $\texttt{HMH}_{h,m}(B) = k$ has the suffix of length m and the prefix of length $2^h - 1 - \max(0, \lfloor\frac{k}{2^m}\rfloor - 1)$ which are fixed.

As $\mathbb{P}[\texttt{HMH}_{h,m}(X) \leq k] = \sum_{i=0}^{k} \mathbb{P}[\texttt{HMH}_{h,m}(X) = i]$, it is enougth to know the value of $\mathbb{P}[\texttt{HMH}_{h,m}(X) = k]$. As $\sum_{i=0}^{k} H(h,m,i) = 2^{n-2^h+1-m} \times \sum_{i=0}^{k} 2^{\max(0,\lfloor\frac{k}{2^m}\rfloor-1)}$, if $k < 2^m$, $\sum_{i=0}^{k} H(h,m,i) = k \times 2^{n-2^h+1-m}$. Otherwise,

$$\begin{array}{rcl} \sum_{i=0}^{k} H(h,m,i) & = & 2^{n-2^h+1-m} \times \sum_{i=0}^{k} 2^{\max(0,\lfloor\frac{i}{2^m}\rfloor-1)} \\ & = & 2^{n-2^h+1-m} \times \Big(\sum_{i=0}^{2^m-1} 2^{\max(0,\lfloor\frac{i}{2^m}\rfloor-1)} \\ & & + \sum_{i=2^m}^{\lfloor\frac{k}{2^m}\rfloor\times 2^m-1} 2^{\max(0,\lfloor\frac{i}{2^m}\rfloor-1)} + \sum_{i=\lfloor\frac{k}{2^m}\rfloor\times 2^m}^{k} 2^{\max(0,\lfloor\frac{i}{2^m}\rfloor-1)} \Big) \\ & = & 2^{n-2^h+1-m} \times \Big(2^m \times 2^0 + 2^m \times \sum_{i=0}^{\lfloor\frac{k}{2^m}\rfloor-2} 2^i + \sum_{i=\lfloor\frac{k}{2^m}\rfloor\times 2^m}^{k} 2^{\lfloor\frac{k}{2^m}\rfloor-1} \Big) \\ & = & 2^{n-2^h+1-m} \times \Big(2^m + 2^{\lfloor\frac{k}{2^m}\rfloor-1+m} - 2^m) + ((k \bmod 2^m)+1) \times 2^{\lfloor\frac{k}{2^m}\rfloor-1} \Big) \\ & = & 2^{n-2^h-m+\lfloor\frac{k}{2^m}\rfloor} \times \Big(2^m + (k \bmod 2^m) + 1 \Big) \end{array}$$

For the sake of completeness, we added all the calculations of $\sum_{i=0}^{k} H(h,m,i)$, but the only thing one needs to remember is that it can be calculated in constant time. Besides, we have

$$\begin{array}{rcl} \mathbb{P}[\texttt{HMH}_{h,m}(X) = k] & = & \mathbb{P}[\texttt{HMH}_{h,m}(X) > k-1] - \mathbb{P}[\texttt{HMH}_{h,m}(X) > k] \\ & = & \frac{C_{2^n}^{|X|} - \sum_{i=0}^{k-1} H(h,m,i)}{C_{2^n}^{|X|}} - \frac{C_{2^n}^{|X|} - \sum_{i=0}^{k} H(h,m,i)}{C_{2^n}^{|X|}} \end{array}$$

We can extend this formal definition to the probability of interval $\{a, \ldots, b\}$, i.e.

$$\mathbb{P}[a \leq \texttt{HMH}_{h,m}(X) \leq b] = \frac{C_{2^n}^{|X|} - \sum_{i=0}^{a-1} H(h,m,i)}{C_{2^n}^{|X|}} - \frac{C_{2^n}^{|X|} - \sum_{i=0}^{b} H(h,m,i)}{C_{2^n}^{|X|}}$$

By approximating the value of $\mathbb{P}[\texttt{HMH}_{h,m}(X) \leq k]$ by $1 - (\frac{2^n - \sum_{i=0}^{k-1} H(h,m,i)}{2^n})^{|X|}$, we can give an approximation of $a_{h,m}$ and $b_{h,m}$ for a threshold ε of $[0,1]$ in $\mathcal{O}(1)$:

$$a_{h,m} \approx \begin{cases} \lfloor (1-(1-\frac{\varepsilon}{2})^{\frac{1}{|V|}}) \times 2^{m+2^h-1} \rfloor & \text{if } (1-(1-\frac{\varepsilon}{2})^{\frac{1}{|V|}}) < 2^{1-2^h} \\ \lfloor (\log_2(1-(1-\frac{\varepsilon}{2})^{\frac{1}{|V|}}) + 2^h - 1) \times 2^m & \\ +(1-(1-\frac{\varepsilon}{2})^{\frac{1}{|V|}}) \times 2^{m-\log_2(1-(1-\frac{\varepsilon}{2})^{\frac{1}{|V|}})} - 1 \rfloor & \text{Otherwise} \end{cases}$$

and

$$b_{h,m} \approx \begin{cases} \lfloor (1-(\frac{\varepsilon}{2})^{\frac{1}{|V|}}) \times 2^{m+2^h-1} \rfloor & \text{if } (1-(\frac{\varepsilon}{2})^{\frac{1}{|V|}}) < 2^{1-2^h} \\ \lfloor (\log_2(1-(\frac{\varepsilon}{2})^{\frac{1}{|V|}}) + 2^h - 1) \times 2^m & \\ +(1-(\frac{\varepsilon}{2})^{\frac{1}{|V|}}) \times 2^{m-\log_2(1-(\frac{\varepsilon}{2})^{\frac{1}{|V|}})} - 1 \rfloor & \text{Otherwise} \end{cases}$$

As shown in Figure 2a, the choice of the good fingerprint depends on the size of the sub-sampling. By dichotomic search, we can compute each exact value $c_{h,m}$ in $\mathcal{O}(|X| \times W)$ and thus we can find the pair h, m which maximizes $c_{h,m}$ in $\mathcal{O}(|X| \times W^2)$. In practice, we use the approximation value of $c_{h,m}$ to compute an approximate optimal pair h, m in $\mathcal{O}(W)$.

The correspondence between the theoretical optimal fingerprint and the fingerprint with the minimum false positive values in practice justifies the relevance of our study (see Figure 2).

Implementation details

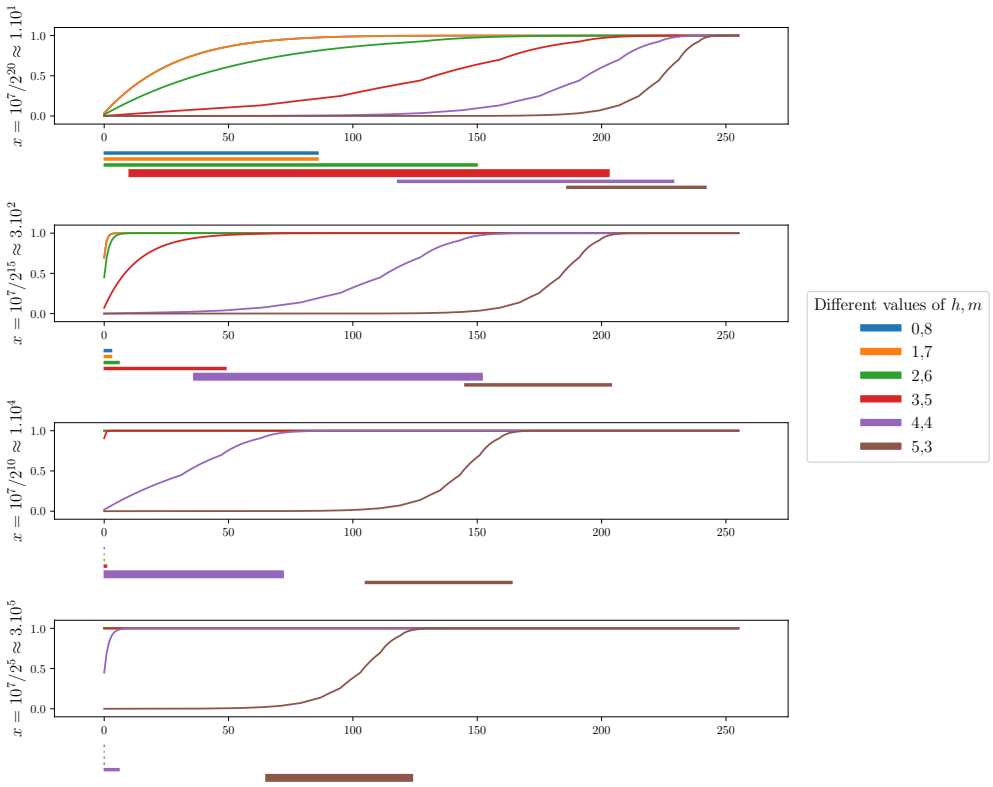
Our proof of concept is dubbed NIQKI (stand for Next Index to Query k-mer Intersection), is open-source and available on Github `https://github.com/Malfoy/NIQKI`. In this section, we detail some practical aspects of this implementation.

Our NIQKI index is technically able to index any kind of fingerprint. In practice, from a given fingerprint size (12 by default), we use the best possible (h,m)HMH fingerprint when the user specifies an expected genome size or a regular Hyperminhash fingerprint if the expert user precise the size of the Hyperloglog fingerprint. We use a default Hyperloglog fingerprint of four if no input is given. We implemented the state-of-the-art densification technique [11] to take care of empty bins.

NIQKI is written in C++ and parallelized with straightforward OpenMP instructions. Each thread inserts or queries a set of sequences independently. A mutex array protects our vectors from double writing for insertion. We chose to use 64-bit hashes for performance purposes as they provide a low collision chance at the hash level in practice while using efficient 64 bits integers. For the same reason, we use a Xorshift [12] hash function are they are found good enough in practice while being incredibly cheap to compute. The sketch sizes are necessarily a power of two for code simplicity and performance aspects. Like Dashing, we ask the user to enter $\log_2(S)$ instead of the actual sketch size.

Our implementation can take any classical sequence format as input: fastq, fasta, or multiline fasta files gzipped or not. The input is either a file of files where each file is an entry to index (or query) or a file where each sequence is a separate entry. We also provide a way to download genomes directly from NCBI from an accession list (that can be generated following NCBI instruction[4]). The corresponding sequences are downloaded and directly inserted into the index in a streaming fashion without any disk operation.

[4] `https://www.ncbi.nlm.nih.gov/genome/doc/ftpfaq/`



(a) Theoretical probabilities and intervals of presence $\{a_{h,m}, \ldots, b_{h,m}\}$.

h, m	$x \approx 1.10^1$	$x \approx 3.10^2$	$x \approx 1.10^4$	$x \approx 3.10^5$
0,8	0.019	0.534	1	1
2,6	0.009	0.289	1	1
3,5	0.006	0.037	0.831	1
4,4	0.011	0.011	0.014	0.029
5,3	0.022	0.022	0.023	0.022
6,2	0.045	0.045	0.045	0.043

(b) False positive values find in practice.

Figure 2 Correspondence between fingerprint with the theoretical maximal interval $\{a_{h,m}, \ldots, b_{h,m}\}$ and the fingerprint with the minimum number of false positive value in practice where $n = 256$, $W = 8$ and the number of k-mers is 10^7 for different values of $S \in \{2^5, 2^{10}, 2^{15}, 2^{20}\}$. 2a shows the different values of $\mathbb{P}[\texttt{HMH}_{h,m}(X) \leq k]$ for all the $k \in \{0, \ldots, 2^W - 1 = 255\}$ and for all pairs h, m such that $n \geq 2^h - 1 + m$ and all values of $x =$ number of k-mers$/S$. Under each plot, we add a rectangle for each interval $\{a_{h,m}, \ldots, b_{h,m}\}$ with the corresponding color. 2b gives the different values of false positive depending of a pair h, m and a subsampling size x.

Indexes can be dumped and loaded from the disk for later use. This allows users to keep a small index on a disk instead of a large file collection and avoid reading all files to reload a given index.

Our implementation delivers a gzip-compressed sparse output to limit disk usage by default. We provide a pretty printer to parse such a file and an option to directly output (larger) human-readable output.

A common use of such a genome index is to compute all pairwise distances between them by comparing the index with itself. An efficient way is to check every vector of our index and increment the score of each genome pair present in the vector. While very fast in theory, this technique requires storing the whole score matrix of size N^2 in memory to be efficient. Henceforth this behavior can be interesting when a large amount of memory is available or when the number of genomes is not too large. Optimization relying on buffer keeping the matrix on disk can dissipate this memory footprint problem at the expense of several passes on the index. In the following benchmarks, the pairwise distances are computed without using the fact that the input is the indexed data itself. Competitors use this information to avoid reading the input file several times or computing the lower-left side of the symmetrical matrix.

Results

Fingerprint impact

As we previously evaluated the impact of the size of the Hyperloglog fingerprint, we now aim to evaluate the impact of the fingerprint size itself. The fingerprint size can be seen as the main parameter of our index and the amount of fingerprint used because our index has a $\theta(S.2^W)$ memory overhead. More importantly, the fingerprint size will impact the amount of false-positive hits where two distinct hashes in compared sketches get the same fingerprint. Such false-positive hits over-estimate the similarity and may negatively impact downstream results by reporting irrelevant matches. To access the false positives found in practice, we created a synthetic dataset of one thousand randomly generated genomes of 10 million bases that share no k-mers. We computed the pairwise distances between those "genomes" and counted the number of hits between unrelated genomes as false positives. We computed sketches of size 4,096 from each ten megabases synthetic genome with fingerprint sizes varying from 4 to 18 with a constant Hyperloglog fingerprint size equal to four. We present the false-positive rates found in practice and the memory used by our index for such an experiment using various fingerprint sizes in Figure 3. As expected, we observe an exponential decrease in the false positive rate as we increase the fingerprint size. We observe that a small fingerprint size can obtain a shallow false-positive rate. From $W = 12$ the rate is below 10^{-3} and from $W = 15$ below 10^{-4}.

Accuracy analysis

To further analyze our approach's precision, we compared the result of NIQKI with state-of-the-art on a real dataset composed of one thousand bacterial genomes from Refseq. In this experiment, we computed the pairwise distances between all genomes with our approach and compared those results with Dashing using many fingerprints (1 million) as the ground truth. In a first experiment reported in Figure 4 we plot the correlation between NIQKI and Dashing estimations using different amounts of fingerprint (namely $S = 65536$,$S = 4096$ and $S = 256$) while using a constant fingerprint size ($W = 12$). As described in the theoretical analysis of the related studies, a smaller amount of fingerprints results in larger error bounds. We observe that NIQKI obtain a strong correlation with Dashing output even with a very low amount of fingerprint. Even small sketches can lead to rough estimations that can be improved using larger sketches where the error bound can be tiny, as in the standard approaches.

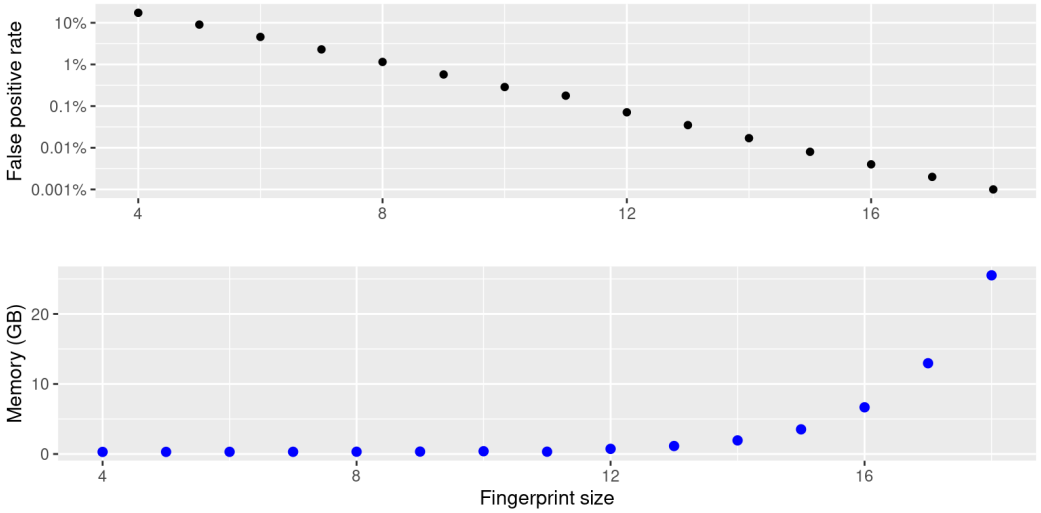
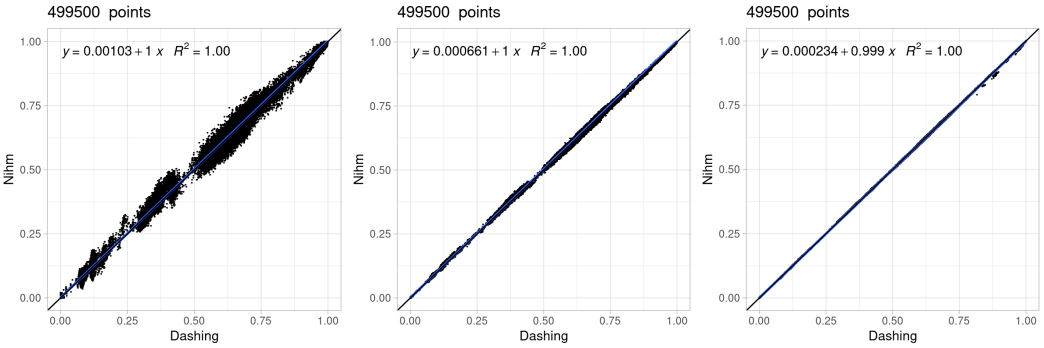


Figure 3 Impact of fingerprint size on false-positive rate and memory usage when indexing and comparing one thousand synthetic genomes of 10 megabases against themselves. Here the Hyperloglog fingerprint is kept constant ($H = 4$) and 4,096 fingerprint are used per genome.



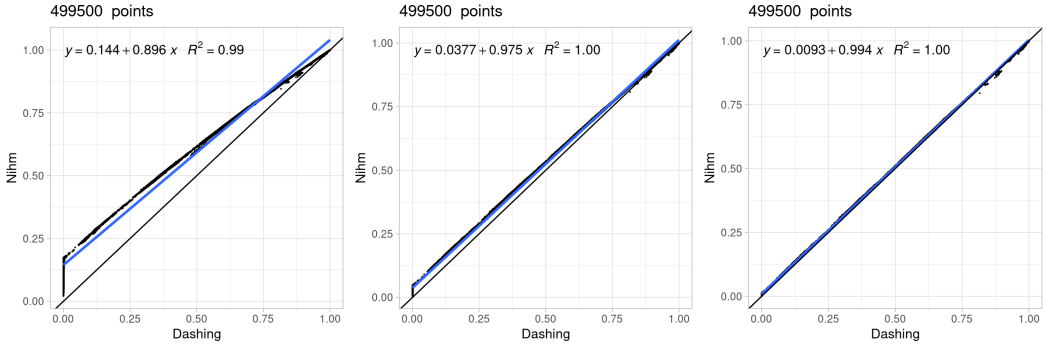
(a) $S = 256$ **(b)** $S = 4096$ **(c)** $S = 65536$

Figure 4 Impact of increasing sketch sizes. We display the correlation between NIQKI with varying sketch sizes and Dashing using one million fingerprints on one thousand bacterial genomes from RefSeq. Here the fingerprint size is kept constant ($W = 12$).

In a second experiment reported in figure5 we used different fingerprint sizes (namely $W = 8$,$W = 6$ and $W = 4$) with a constant fingerprint number($S = 65536$).

Due to the large number of fingerprints, we observe very small error bounds, but due to the false positive rate, we observe that all indices are overestimated. The expected overestimation is $\approx (1 - J) * FP_{Rate}$ where J is the Jaccard index between the two sequences, and the observed pattern goes accordingly with this projection.

This experiment shows that using fingerprints as small as eight bits seems reasonable in practice for such an index. The false positive rate could still impact the results when dealing with a low Jaccard index, and a larger fingerprint should be used. We fixed the default fingerprint size to 12 as it provides very low false positives and a reasonable memory overhead.



(a) $W = 4$ **(b)** $W = 6$ **(c)** $W = 8$

Figure 5 Impact of increasing fingerprint sizes. Correlation between NIQKI with varying fingerprint size and Dashing using one million fingerprints on one thousand bacterial genomes from RefSeq. Here the sketch size is kept constant ($S = 65536$).

Performance analysis

This section displays how our index can scale on synthetic and real datasets compared to the two most used tools of the state-of-the-art Mash and Dashing. All experiments were performed on a single cluster node running with Intel(R) Xeon(R) CPU E5-2420 @1.90GHz with 192GB of RAM and Ubuntu 16.04. with a timeout of 48hours. Bindash was not included in our benchmark because it cannot compute a distance matrix from a file of files directly.

All state-of-the-art tools present a $\mathcal{O}(S.N)$ query time (or $\mathcal{O}(S.\log(S)).N$) if S minimum Minhash is used) as a query sketch have to be compared to each indexed sketch. In contrast, our index presents a query time linear with the number of hits. It can be $\mathcal{O}(S.N)$ in the worst case where all genomes are identical and can be $\mathcal{O}(S)$ in the best-case scenario where an alien entry is entirely dissimilar from the indexed genomes (ignoring false positives).

To show those two regimes, we perform a first "idealistic" benchmark with randomly generated genomes sharing no k-mers that display the best case of our index. We compare those results to a real database composed of GenBank genomes in a second experiment. We downloaded all Genbank bacterial assemblies (1,042,611) and randomly selected genomes from this pool to build such an index. We want to point out that such a database is highly redundant. For instance, it contains 142,568 assemblies of the Escherichia coli organism. It constitutes a stress case for our index as each query associated with Escherichia coli should report a large part of the indexed genomes among its matches.

In Figure 6 we ran Dashing, Mash, and NIQKI to compute pairwise distance on randomly generated genome collection of growing sizes and report the mean CPU time per entry and RAM usage. Because of the quadratic aspect of computing all pairwise distances, we report the mean time required per entry by dividing the total running time by the amount of entry to improve results readability. To provide a fair comparison, we choose parameters such as each tool to allocate the same memory per entry. Mash used 65,536 32 bits Minhash fingerprints, Dashing used 262,144 8 bits Hyperloglog signatures, and NIQKI used 65,536 32 bits genome identifiers and the default fingerprint ($W = 12$). We observe that both Mash and Dashing runtime grow following an expected quadratic evolution as computing all pairwise distances needs $\Theta(S.N^2)$. The growth runtime of NIQKI is roughly linear and can become orders of magnitude faster than state-of-the-art.

In Figure 7 we perform the same experiment on a realistic database constituted from bacterial GenBank genomes. We observe that Mash and Dashing deliver similar performances on synthetic and real datasets. The differences are due to the fact that the mean size of real

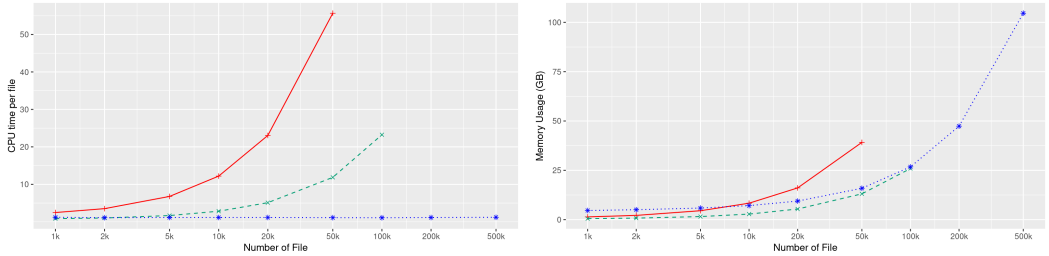


Figure 6 Benchmark on synthetic genomes. We report the CPU time divided by the number of entry and the total memory footprint for various collection sizes. Mash is plot in red, Dashing in green and NIQKI in blue.

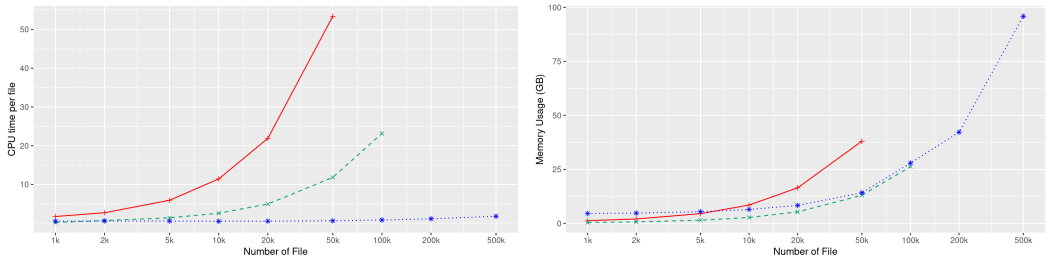


Figure 7 Benchmark on GenBank bacterial genomes. We report the CPU time divided by the number of entry and the total memory footprint for various collection sizes. Mash is plot in red, Dashing in green and NIQKI in blue.

bacterial genomes is around five megabases, while the generated genomes are ten megabases long. NIQKI results on small databases are similar to the synthetic cases, but a superlinear growth can be observed on the largest collections. However, despite this observation NIQKI can still be orders of magnitude faster than state-of-the-art tools while using a comparable amount of memory. For example, the largest Mash experiment used 773 CPU hours on 50k genomes where NIQKI used only 15 CPU hours. The largest Dashing experiment used 645 CPU hours on 100k genomes where NIQKI used only 30 CPU hours. We extrapolate that Dashing or Mash would need more than ten thousand CPU hours to handle 500k genomes while NIQKI used 166 CPU hours. On the downside, because of its large overhead NIQKI uses more memory than the other approaches on small collections and is slightly more memory expensive than Dashing on larger databases.

Indexing Genbank bacterial genomes

To access our approach scaling ability scalability, we choose to index and compute pairwise distance on all bacterial genomes from GenBank. This dataset represents more than one million genomes, counting more than five tera-nucleotides or more than one terabytes of gzipped fasta files. We choose to evaluate the cost of such an operation by varying the number of indexed minimizers that linearly impact our approach's memory cost and the running time. We choose to keep the memory overhead constant ($\mathcal{O}(2^{W+\log_2(S)})$) to ensure a fair memory comparison and raise the fingerprint size accordingly. We report the CPU time and Wall clock time along with the memory usage required for those experiments in Table 1. We observe that we can index and compute pairwise distance on such a database with medium-sized sketches in a few days with a reasonable memory footprint. If the indexing

time is roughly constant and dominated by reading the input, we observe that the query time grows linearly with the sketch size (plus a "reading" time constant cost). We want to recall that most of the query computational time is due to the database's redundancy, which generates many matches for some queries. For comparison, on a simulated dataset of one million random synthetics genomes indexed with 4096 fingerprints, the query time lasted 12 hours instead of 38 hours for the Genbank database.

Table 1 Benchmark on all Genbank bacterial genomes with various sketch sizes.

#Fingerprint	CPU time (seconds)	Wall clock time all / query (hours)	Memory
32	416,245	12 / 6	4,675
128	469,072	12 / 6	5,363
1,024	969,884	17 / 11	11,306
2,048	1,525,982	28 / 22	17,602
4,096	3,383,009	44 / 38	30,021

Conclusions and future work

We showed that using inverted indexes on partitioned sketches leads to algorithmic improvement of fingerprint queries that can reduce running time by orders of magnitude with comparable precision. Theoretically, our proposed structure query is $\mathcal{O}(\#hits)$ compared to $\mathcal{O}(S.N)$ for state-of-the-art. This structure came with a memory cost as our index uses $\mathcal{O}(S(NlogN + 2^W))$ bits instead of $\mathcal{O}(S.N.W)$. We showed that even a straightforward implementation could efficiently index small fingerprints while providing results comparable to the state-of-the-art. Our approach could provide orders of magnitude faster queries on idealistic synthetic databases and real-world redundant databases with comparable memory footprints on large instances. We also demonstrated our index capacity to index and query all bacterial genomes of Genbank (more than one million bacterial genomes) in a matter of days on a small cluster. Our index can be used to index large collections to detect matches between novel query sequences and elements of the collection or to compute pairwise comparisons of all indexed sequences.

While our index can index any fingerprint (Minhash, Hyperloglog, Hyperminhash), we aimed to provide the best possible fingerprints of a given size to limit the number of false positives. To do so, we generalized the concept of Hyperminhash to account for different sizes of Hyperloglog and Minhash fingerprints dubbed (h, m)-HMH fingerprints. Interestingly Minhash, Hyperloglog, and Hyperminhash fingerprints can be seen as particular cases of (h, m)-HMH fingerprints. Given an expected sub-sampling, we can select the (h, m)-HMH fingerprint with the optimal parameter that provides the lowest false positive rate and confirms this choice in practice. While improving false-positive rates, those well-designed fingerprints come without computational or memory costs and could be used to improve existing sketching methods. Sketching methods could either use smaller fingerprints for a desired false positive rate, reducing memory footprint, or reducing their false positive rate without memory or time overhead.

On the practical aspect, our implementation still misses user-friendly features such as computing various metrics used in practice (Mash Index, containment index, or cardinality estimation) using advanced estimation methods. The different partition-based optimization described in the methods could also optimize query time in certain situations. Our running time could be improved by implementing classical optimization techniques such as batched

queries, SIMD parallelism or vectorization, or practical optimization as computing only upper-half identifiers when comparing the index against itself. We mentioned that processing partition by column grants the possibility to stop queries that can not find high-scoring matches, but this feature is not yet implemented in practice. More generally more advanced output filtering technique could benefit our implementation in practice. More importantly, our current implementation uses plain integers as genomes identifiers leading to high memory costs. As our vectors can be seen as lists of increasing integers, those could be compressed with delta-encoding or other high throughput compression technique [8]. The index representation could be highly reduced with a limited impact on the runtime. While our proof of concept implementation is efficient, it tends to allocate a large amount of unused memory because of the heavy use of vectors. Other implementations could be made with different time/memory trade-offs. For example, comparing our approach with a Rank and the select-based index could be interesting.

On the theoretical aspects, our analysis of (h, m)-HMH fingerprint leads us to identify the need for an efficient fingerprint that presents smoother patterns to enable a larger range of possible hashes with or without estimation of the sub-sampling. On the fingerprint indexing problem, we showed that our partition query could be considered optimal as its running time is linear with the size of the output. However, the classical usage is to ask for matches with a number of hits above a certain threshold. It would be interesting to investigate which indexes or algorithmic solutions would provide an optimal answer to this problem.

References

1 Alexandre Almeida, Stephen Nayfach, Miguel Boland, Francesco Strozzi, Martin Beracochea, Zhou Jason Shi, Katherine S Pollard, Ekaterina Sakharova, Donovan H Parks, Philip Hugenholtz, et al. A unified catalog of 204,938 reference genomes from the human gut microbiome. *Nature biotechnology*, 39(1):105–114, 2021.

2 Daniel N Baker and Ben Langmead. Dashing: fast and accurate genomic distances with hyperloglog. *Genome biology*, 20(1):1–12, 2019.

3 Timo Bingmann, Phelim Bradley, Florian Gauger, and Zamin Iqbal. Cobs: a compact bit-sliced signature index. In *International Symposium on String Processing and Information Retrieval*, pages 285–303. Springer, 2019.

4 Phelim Bradley, Henk C Den Bakker, Eduardo PC Rocha, Gil McVean, and Zamin Iqbal. Ultrafast search of all deposited bacterial and viral genomic data. *Nature biotechnology*, 37(2):152–159, 2019.

5 Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997.

6 Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

7 David Koslicki and Hooman Zabeti. Improving minhash via the containment index with applications to metagenomic analysis. *Applied Mathematics and Computation*, 354:206–215, 2019.

8 Daniel Lemire, Leonid Boytsov, and Nathan Kurz. Simd compression and the intersection of sorted integers. *Software: Practice and Experience*, 46(6):723–749, 2016.

9 Ping Li and Christian König. b-bit minwise hashing. In *Proceedings of the 19th international conference on World wide web*, pages 671–680, 2010.

10 Antoine Limasset. Million sequences indexing. In *BMC BIOINFORMATICS*, volume 20. BMC CAMPUS, 4 CRINAN ST, LONDON N1 9XW, ENGLAND, 2019.

11 Tung Mai, Anup Rao, Matt Kapilevich, Ryan Rossi, Yasin Abbasi-Yadkori, and Ritwik Sinha. On densification for minwise hashing. In *Uncertainty in Artificial Intelligence*, pages 831–840. PMLR, 2020.

12 George Marsaglia et al. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.

13 Martin D Muggli, Bahar Alipanahi, and Christina Boucher. Building large updatable colored de bruijn graphs via merging. *Bioinformatics*, 35(14):i51–i60, 2019.

14 Brian D Ondov, Todd J Treangen, Páll Melsted, Adam B Mallonee, Nicholas H Bergman, Sergey Koren, and Adam M Phillippy. Mash: fast genome and metagenome distance estimation using minhash. *Genome biology*, 17(1):1–14, 2016.

15 N Tessa Pierce, Luiz Irber, Taylor Reiter, Phillip Brooks, and C Titus Brown. Large-scale sequence comparisons with sourmash. *F1000Research*, 8, 2019.

16 Will PM Rowe. When the levee breaks: a practical guide to sketching algorithms for processing the flood of genomic data. *Genome biology*, 20(1):1–12, 2019.

17 Anshumali Shrivastava. Optimal densification for fast and accurate minwise hashing. In *International Conference on Machine Learning*, pages 3154–3163. PMLR, 2017.

18 Yun William Yu and Griffin M Weber. Hyperminhash: Minhash in loglog space. *arXiv preprint*, 2017. `arXiv:1710.08436`.

19 XiaoFei Zhao. Bindash, software for fast genome distance estimation on a typical personal laptop. *Bioinformatics*, 35(4):671–673, 2019.