23rd International Workshop on Algorithms in Bioinformatics

WABI 2023, September 4–6, 2023, Houston, TX, USA

Edited by

Djamal Belazzougui

Aïda Ouangraoua

Editors

Djamal Belazzougui
CERIST, Algeria
dbelazzougui@cerist.dz

Aïda Ouangraoua
University of Sherbrooke, Canada
Aida.Ouangraoua@USherbrooke.ca

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 978-3-95977-294-5

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/978-3-95977-294-5.

Publication date
August, 2023

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.2023.0

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

Invited Talk

Papers

23rd International Workshop on Algorithms in Bioinformatics (WABI 2023).
Editors: Djamal Belazzougui and Aïda Ouangraoua
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This proceedings volume contains papers and abstracts presented at the 23rd Workshop on Algorithms in Bioinformatics (WABI 2023), which was held in Houston, Texas, USA, from September 4th to 6th, 2023.

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 serves as a forum for the presentation of new insights about discrete algorithms and machine-learning methods that address important problems in biology, particularly those based on molecular data and phenomena. These algorithms are founded on sound models, exhibit computational efficiency, and 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 for future research. Over the 23 instances of WABI, computational biology has significantly grown in importance. Computational analysis methods, some of which have been advanced significantly over the years at WABI, have proven crucial for the global response to the COVID-19 pandemic and for vaccine development.

WABI 2023 was co-located with ACM-BCB 2023. A total of 44 manuscripts were submitted to WABI 2023, out of which 22 were selected for presentation at the conference. Among these, 19 are included in this proceedings volume as full papers, and 3 are included as extended abstracts. The 22 papers chosen for the conference underwent a rigorous peer review process, involving at least three (most frequently four) independent reviewers per submitted paper. Following this, discussions took place among the WABI Program Committee members. The reviews were conducted by both Program Committee members and additional reviewers who were selected based on their expertise relevant to specific papers. The selected papers encompass a wide range of topics, including phylogenetic trees and networks, cancer phylogenetics, sequence alignment and assembly, gene and genomic-level evolution, genome rearrangement, sequence and genome analysis, RNA structure, pattern matching, data compression, and more. These papers are arranged according to the conference program within this volume. The program also featured an invited talk, two highlight talks, and a poster session.

Extended versions of select papers have been invited for publication in a thematic series in the journal "Algorithms for Molecular Biology" (AMB), published by BioMed Central. A total of 16 extended papers will be submitted to this thematic series.

We extend our gratitude to all the authors of the submitted papers, whose contributions made this conference possible. Special thanks are due to all the members of the WABI 2023 Program Committee and their sub-reviewers for their diligent efforts in reviewing the manuscripts and engaging in comprehensive discussions. These discussions informed the decision-making process and resulted in constructive review reports for the authors. We are grateful to the WABI Steering Committee for their availability, assistance, and guidance. Our thanks go out to all the conference participants, session chairs, and speakers who contributed to a highly successful scientific program. In particular, we express our gratitude to the conference's keynote speaker, Teresa Przytycka (NCBI, NIH), for her presentation titled "Algorithmic Approaches to Study Mutational Processes in Cancer". We extend our thanks to the highlight presentation speakers, Christina Boucher and Travis Gagie, for graciously agreeing to speak at the conference on the topic of "The Theory and Applications of the r-index in Bioinformatics". Furthermore, we would like to express our special appreciation

23rd International Workshop on Algorithms in Bioinformatics (WABI 2023).
Editors: Djamal Belazzougui and Aïda Ouangraoua
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to the ACM-BCB 2023 Organizing Committee for their dedicated efforts and generosity in organizing the event. The EasyChair system was utilized for handling submissions and reviews.

Previous WABI proceedings appeared in LNCS/LNBI volumes as follows: 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). Beginning in 2017, they appeared in LIPIcs volumes: 88 (WABI 2017, Boston), 113 (WABI 2018, Helsinki), 143 (WABI 2019, Boston), 172 (WABI 2020, held virtually in Pisa), 201 (WABI 2021, held virtually in Chicago), and 242 (WABI 2022, Potsdam).

Djamal Belazzougui and Aïda Ouangraoua

WABI 2023 Committees

Steering Committee

Vincent Moulton
University of East Anglia, UK

Nadia Pisanti
University of Pisa, Italy

Mona Singh
Princeton University, USA

Jens Stoye
Bielefeld University, Germany

PC Chairs

Djamal Belazzougui
CERIST, Algeria

Aïda Ouangraoua
University of Sherbrooke, Canada

Program Committee

Tatsuya Akutsu
Kyoto University, Japan

Jasmijn Baaijens
TU Delft, Netherlands

Anne Bergeron
Université du Québec à Montréal, Canada

Paola Bonizzoni
Universita di Milano-Bicocca, Italy

Christina Boucher
University of Florida, USA

Marília Braga
Bielefeld University, Germany

Cédric Chauve
Simon Fraser University, Canada

Daniel Doerr
University of Duesseldorf, Germany

Mohammed El-Kebir
University of Illinois at Urbana-Champaign, USA

23rd International Workshop on Algorithms in Bioinformatics (WABI 2023).
Editors: Djamal Belazzougui and Aïda Ouangraoua
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Nadia El-Mabrouk
University of Montreal, Canada

Anna Gambin
Warsaw University, Poland

Bjarni Halldorsson
deCODE genetics and Reykjavik University, Iceland

Katharina Huber
University of East Anglia, UK

Gregory Kucherov
University of Paris Est, France

Manuel Lafond
Université de Sherbrooke, Canada

Elodie Laine
Sorbonne Université, France

Veli Makinen
University of Helsinki, Finland

Erin Molloy
University of Maryland College Park, USA

Vincent Moulton
University of East Anglia, UK

Francesca Nadalin
EMBL-EBI / ITT, Italy, England

Gonzalo Navarro
University of Chile, Chile

Yann Ponty
CNRS & École poly technique, France

Teresa Przytycka
NCBI, NIH, USA

Nadia Pisanti
University of Pisa, Italy

Solon Pissis
CWI Amsterdam, the Netherlands

Alberto Policriti
University of Udine, Italy

Sven Rahmann
Saarland University, Germany

Knut Reinert
FU Berlin, Germany

Eric Rivals
LIRMM & Université de Montpellier, France

Giovanna Rosone
University of Pisa, Italy

Mikaël Salson
Université de Lille, France

Alexander Schoenhuth
Bielefeld University, Germany

Mingfu Shao
Penn State, USA

Blerina Sinaimeri
Université Lyon I & INRIA, France

Krister Swenson
CNRS & Université de Montpellier, France

Sharma Thankachan
University of Central Florida, USA

Alexandru Tomescu
University of Helsinki, Finland

Hélène Touzet
CNRS & Université de Lille, France

Gianluca Della Vedova
University of Milano-Bicocca, Italy

Tomas Vinar
Comenius University, Slovakia

Jérôme Waldispühl
McGill University, Canada

Tandy Warnow
University of Illinois at Urbana-Champaign, USA

Prudence W. H. Wong
University of Liverpool, UK

Louxin Zhang
National University of Singapore, Singapore

Michal Ziv-Ukelson
Ben Gurion University of the Negev, Israel

Subreviewers

Paniz Abedin

Metin Balaban

Leonard Bohnenkämper

Vladimir Boza

Broňa Brejová

Hannes P. Eggertsson Manuel Caceres

Bastien Cazaux

Isaure Chauvot de Beauchene

Ke Chen

Sriram P. Chockalingam

Nicola Cotumaccio

Mitra Darvish

Mattéo Delabre

Diego Diaz

Travis Gagie

Mathieu Gascon

Nimisha Ghosh

Daniel Gibney

Mauricio Soto Gomez

Simon Gottlieb

Xinyu Gu

Pina Krell

Xiang Li

Ronny Lorenz

Davide Martincigh

Fábio Henrique Viduani Martinez

Roberto Pagliarini

Murray Patterson

Anna Vathrakokoili Pournara

Rangaram D Raghuram

Vaibhav Rajan

Hugues Richard

Diego P. Rubert

Cenk Sahinalp

Johannes Schlüter

Sebastian Schmidt

Johanna Schmitz

Grzegorz Skoraczyński

Francesco Stranieri

Riccardo Vicedomini

Tasfia Zahin

Diego Zea

ACM-BCB Organizing Committee

May D. Wang (general chair)
Georgia Institute of Technology, USA

Byung-Jun Yoon (general chair)
Texas A&M University, USA

Peter L. Elkin (technical program chair)
University at Buffalo, USA

Xiaoning Qian (technical program chair and local chair)
Texas A&M University, USA

Wenyi Wang (technical program chair)
MD Anderson Cancer Center, USA

Yijie Wang (poster chair)
Indiana University, USA

Vicky Yao (local chair)
Rice University, USA

Ananth Kalyanaraman (website)
Washington State University, USA

(and all the ACM-BCB Organizing Committee)

Algorithmic Approaches to Study Mutational Processes in Cancer

Teresa M. Przytycka
National Center for Biotechnology Information, National Library of Medicine,
National Institutes of Health, Bethesda, MD, USA

Abstract

Mutations are the driving force of evolution, yet they underlie many diseases and, in particular, cancer. They are thought to arise from a combination of stochastic errors in DNA processing, naturally occurring DNA damage (e.g., the spontaneous deamination of methylated CpG sites), replication errors, carcinogenic exposures or cancer related aberrations of DNA maintenance machinery. These processes often lead to distinctive patterns of mutations, called "mutational signatures". Starting with the seminal work of Alexandrov at al. [1] several computational approaches have been developed to uncover such mutational signatures. However connecting mutational signatures to mutational processes is not always easy [3].

To gain insights into the relationships between mutational processes and computationally derived somatic mutation patterns (mutational signatures), we developed several complementary approaches that leverage different algorithmic techniques allowing us to link such patterns to their potential causes. For example, to investigate the genetic aberrations associated with mutational signatures, we took a network-based approach considering mutational signatures as phenotypes. Specifically, our analysis aims to answer the following two complementary questions: (i) what are functional pathways whose gene expression activities correlate with the strengths of mutational signatures, and (ii) are there pathways whose genetic alterations might have led to specific mutational signatures? To identify mutated pathways, we adopted an optimization method based on integer linear programming. Analyzing a breast cancer dataset, we identified pathways associated with mutational signatures on both expression and mutation levels. Our analysis captured important differences in the etiology of the APOBEC related signatures and the two clock-like signatures. In particular, it revealed that clustered and dispersed APOBEC mutations may be caused by different mutagenic processes. In addition, our analysis elucidated differences between two age related signatures – one of the signatures is correlated with the expression of cell cycle genes while the other has no such correlation but shows patterns consistent with the exposure to environmental/external processes [4].

Complementing this approach, we also developed a network-based method, named GENESIGNET that constructs an influence/information flow network connecting genes and mutational signatures [2]. The approach leverages sparse partial correlation among other statistical techniques to uncover dominant influence relations between the activities of network nodes. Applying GENESIGNET to cancer data sets, we uncovered important relations between mutational signatures and several cellular processes that can shed light on cancer-related processes. In particular, GENESIGNET exposed a link between the SBS8 signature of unknown etiology and the Nucleotide Excision Repair (NER) pathway.

Linking mutational signatures to molecular features can help understand the etiology and develop personalized cancer therapy. However, due to the complex and dynamic nature of tumor evolution, untangling the cause and effect relationship can be challenging and requires further integrated and comprehensive analyses.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Applied computing → Life and medical sciences

Keywords and phrases Biological Networks, Cancer, Mutational Signatures, DNA Damage, DNA Repair

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.1

Category Invited Talk

23rd International Workshop on Algorithms in Bioinformatics (WABI 2023).
Editors: Djamal Belazzougui and Aïda Ouangraoua; Article No. 1; pp. 1:1–1:2

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

Funding *Teresa M. Przytycka*: Supported by the Intramural Research Program of the National Library of Medicine at the National Institutes of Health, USA.

References

1 Ludmil B Alexandrov, Serena Nik-Zainal, David C Wedge, Samuel Aparicio, Sam Behjati, et al. Signatures of mutational processes in human cancer. *Nature*, 500(7463):415–421, 2013. doi:10.1038/nature12477.

2 B. Amgalan, D. Wojtowicz, Y.-A. Kim, and T. M. Przytycka. Influence network model uncovers relations between biological processes and mutational signatures. *Genome Medicine*, 15, 2023.

3 Y.-A. Kim, M. D. M. Leiserson, P. Moorjani, R. Sharan, D. Wojtowicz, and T. M. Przytycka. Mutational Signatures: From Methods to Mechanisms. *Annu Rev Biomed Data Sci*, 4:189–206, July 2021.

4 Y.-A. Kim, D. Wojtowicz, R. Sarto Basso, I. Sason, W. Robinson, D. S. Hochbaum, M. D. M. Leiserson, R. Sharan, F. Vadin, and T. M. Przytycka. Network-based approaches elucidate differences within APOBEC and clock-like signatures in breast cancer. *Genome Med*, 12(1):52, May 2020.

EMMA: Adding Sequences into a Constraint Alignment with High Accuracy and Scalability

Chengze Shen
Department of Computer Science, University of Illinois Urbana-Champaign, IL, USA

Baqiao Liu
Department of Computer Science, University of Illinois Urbana-Champaign, IL, USA

Kelly P. Williams
Sandia National Laboratories, Livermore, CA, USA

Tandy Warnow[1]
Department of Computer Science, University of Illinois Urbana-Champaign, IL, USA

Abstract

Multiple sequence alignment (MSA) is a crucial precursor to many downstream biological analyses, such as phylogeny estimation [4], RNA structure prediction [8], protein structure prediction [1], etc. Obtaining an accurate MSA can be challenging, especially when the dataset is large (i.e., more than 1000 sequences). A key technique for large-scale MSA estimation is to add sequences into an existing alignment. For example, biological knowledge can be used to form a reference alignment on a subset of the sequences, and then the remaining sequences can be added to the reference alignment. Another case where adding sequences into an existing alignment occurs is when new sequences or genomes are added to databases, leading to the opportunity to add the new sequences for each gene in the genome into a growing alignment. A third case is for *de novo* multiple sequence alignment, where a subset of the sequences is selected and aligned, and then the remaining sequences are added into this "backbone alignment" [5, 6, 10, 3, 7, 11]. Thus, adding sequences into existing alignments is a natural problem with multiple applications to biological sequence analysis.

A few methods have been developed to add sequences into an existing alignment, with MAFFT--add [2] perhaps the most well-known. However, several multiple sequence alignment methods that operate in two steps (first extract and align the backbone sequences and then add the remaining sequences into this backbone alignment) also provide utilities for adding sequences into a user-provided alignment. We present EMMA, a new approach for adding "query" sequences into an existing "constraint" alignment. By construction, EMMA never changes the constraint alignment, except through the introduction of additional sites to represent homologies between the query sequences. EMMA uses a divide-and-conquer technique combined with MAFFT--add (using the most accurate setting, MAFFT-linsi--add) to add sequences into a user-provided alignment. We evaluate EMMA by comparing it to MAFFT-linsi--add, MAFFT--add (the default setting), and WITCH-ng-add. We include a range of biological and simulated datasets (nucleotides and proteins) ranging in size from 1000 to almost 200,000 sequences and evaluate alignment accuracy and scalability. MAFFT-linsi--add was the slowest and least scalable method, only able to run on datasets with at most 1000 sequences in this study, but had excellent accuracy (often the best) on those datasets. We also see that EMMA has better recall than WITCH-ng-add and MAFFT--add on large datasets, especially when the backbone alignment is small or clade-based.

2012 ACM Subject Classification Applied computing → Bioinformatics

Keywords and phrases Multiple sequence alignment, constraint alignment, MAFFT

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.2

Category Abstract

Related Version *Full Version*: https://doi.org/10.1101/2023.06.12.544642 [9]

[1] corresponding author

Supplementary Material bioRxiv paper has additional supplementary materials
Software (Source Code): `https://github.com/c5shen/EMMA`
archived at `swh:1:dir:d3da832ddc0eb1adb17fbfee398750b89da20544`
Dataset: `https://doi.org/10.13012/B2IDB-2567453_V1`
Dataset: `https://doi.org/10.13012/B2IDB-3974819_V1`

Funding This work was funded in part by the US NSF grant 2006069 and by the Laboratory Directed Research and Development program at 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.

References

1 John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, August 2021. Number: 7873 Publisher: Nature Publishing Group. `doi:10.1038/s41586-021-03819-2`.

2 Kazutaka Katoh and Martin C. Frith. Adding unaligned sequences into an existing alignment using MAFFT and LAST. *Bioinformatics*, 28(23):3144–3146, December 2012. `doi:10.1093/bioinformatics/bts578`.

3 Baqiao Liu and Tandy Warnow. WITCH-NG: efficient and accurate alignment of datasets with sequence length heterogeneity. *Bioinformatics Advances*, 3(1):vbad024, January 2023. `doi:10.1093/bioadv/vbad024`.

4 David A Morrison. Multiple sequence alignment for phylogenetic purposes. *Australian Systematic Botany*, 19(6):479–539, 2006.

5 Nam-phuong D. Nguyen, Siavash Mirarab, Keerthana Kumar, and Tandy Warnow. Ultra-large alignments using phylogeny-aware profiles. *Genome Biology*, 16(1):124, June 2015. `doi:10.1186/s13059-015-0688-z`.

6 Minhyuk Park, Stefan Ivanovic, Gillian Chu, Chengze Shen, and Tandy Warnow. UPP2: fast and accurate alignment of datasets with fragmentary sequences. *Bioinformatics*, 39(1):btad007, January 2023. `doi:10.1093/bioinformatics/btad007`.

7 Minhyuk Park and Tandy Warnow. HMMerge: an ensemble method for multiple sequence alignment. *Bioinformatics Advances*, page vbad052, 2023.

8 Bruce A Shapiro, Yaroslava G Yingling, Wojciech Kasprzak, and Eckart Bindewald. Bridging the gap in RNA structure prediction. *Current Opinion in Structural Biology*, 17(2):157–165, 2007.

9 Chengze Shen, Baqiao Liu, Kelly P Williams, and Tandy Warnow. Computing multiple sequence alignments given a constraint subset alignment using EMMA. *bioRxiv*, 2023. `doi:10.1101/2023.06.12.544642`.

10 Chengze Shen, Minhyuk Park, and Tandy Warnow. WITCH: Improved Multiple Sequence Alignment Through Weighted Consensus Hidden Markov Model Alignment. *Journal of Computational Biology*, May 2022. Publisher: Mary Ann Liebert, Inc., publishers. `doi:10.1089/cmb.2021.0585`.

11 Kazunori D. Yamada, Kentaro Tomii, and Kazutaka Katoh. Application of the MAFFT sequence alignment program to large data?reexamination of the usefulness of chained guide trees. *Bioinformatics*, 32(21):3246–3251, November 2016. `doi:10.1093/bioinformatics/btw412`.

BATCH-SCAMPP: Scaling Phylogenetic Placement Methods to Place Many Sequences

Eleanor Wedell
Department of Computer Science, University of Illinois Urbana-Champaign, IL, USA

Chengze Shen
Department of Computer Science, University of Illinois Urbana-Champaign, IL, USA

Tandy Warnow
Department of Computer Science, University of Illinois Urbana-Champaign, IL, USA

Abstract

Phylogenetic placement is the problem of placing one or more query sequences into a phylogenetic "backbone" tree, which may be a maximum likelihood tree on a multiple sequence alignment for a single gene, a taxonomy with leaves labeled by sequences for a single gene [7], or a species tree [4]. When the backbone tree is a tree estimated on a single gene, the most accurate techniques for phylogenetic placement are likelihood-based, and can be computationally intensive when the backbone trees are large [3]. Phylogenetic placement into gene trees occurs when updating existing gene trees with newly observed sequences, but can also be applied in the "bulk" context, where many sequences are placed at the same time into the backbone tree. For example, phylogenetic placement can be used to taxonomically characterize shotgun sequencing reads generated for an environmental sample in metagenomic analysis [7, 2].

The two most well known maximum likelihood phylogenetic placement methods are pplacer [5] and EPA-ng [2]. Of these two, EPA-ng is optimized for scaling the number of query sequences and is capable of placing millions of sequences into phylogenetic trees of up to a few thousand sequences [2], and achieves sublinear runtime in the number of query sequences (see Figure 2 from [1]).

Previously we introduced the SCAMPP framework [8] to enable both pplacer and EPA-ng to perform phylogenetic placement into ultra-large backbone trees, and we demonstrated its utility for placing into backbone trees with up to 200,000 sequences. By using maximum likelihood methods pplacer or EPA-ng within the SCAMPP framework, the resulting placements are more accurate than with APPLES-2 [1], with the most notable accuracy improvement for fragmentary sequences, and are computationally similar for single query sequence placement [8]. However, SCAMPP was designed to incrementally update a large tree, one query sequence at a time, and was not optimized for the other uses of phylogenetic placement, where batch placement of many sequencing reads is required.

Here we introduce BATCH-SCAMPP, a technique that improves scalability in both dimensions: the number of query sequences being placed into the backbone tree and the size of the backbone tree. Furthermore, BATCH-SCAMPP is specifically designed to improve EPA-ng's scalability to large backbone trees. Although BATCH-SCAMPP is based on SCAMPP, it uses a substantially modified design in order to be able to take advantage of EPA-ng's ability to place many query sequences efficiently.

The BATCH-SCAMPP method operates by allowing the input set of query sequences to suggest and then vote on placement subtrees, thus enabling many query sequences to select the same placement subtree. We pair BATCH-SCAMPP with EPA-ng to explore the capability of this approach for scaling to many query sequences. We show that this combination of techniques (which we call BSCAMPP+EPA-ng, or BSCAMPP(e)) not only provides high accuracy and scalability to large backbone trees, matching that of SCAMPP used with EPA-ng (i.e., SCAMPP(e)), but also achieves the goal of scaling sublinearly in the number of query sequences. Moreover, it is much more scalable than EPA-ng and faster than SCAMPP+EPA-ng: when placing 10,000 sequences into a backbone tree of 50,000 leaves, EPA-ng is unable to run due to memory issues, SCAMPP+EPA-ng requires 1421 minutes, and BSCAMPP(e) places all sequences in 7 minutes (all given the same computational resources. Figure 1 gives an example of this performance advantage on the nt78 [6] simulated dataset.

23rd International Workshop on Algorithms in Bioinformatics (WABI 2023).
Editors: Djamal Belazzougui and Aïda Ouangraoua; Article No. 3; pp. 3:1–3:2

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

2012 ACM Subject Classification Applied computing → Bioinformatics

Keywords and phrases Phylogenetic Placement, EPA-ng, Phylogenetics

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.3

Category Abstract

Related Version *Full Version*: https://doi.org/10.1101/2022.10.26.513936

Supplementary Material *Software (Source Code)*: https://github.com/ewedell/BSCAMPP_code archived at swh:1:dir:94380a8834bd8587970f2bd229b5ba62c733ca86

Funding The authors acknowledge the financial support of the Department of Computer Science. EW was supported by a Siebel Scholars scholarship, a SURGE Fellowship, and a Wing Kai Cheng Fellowship. CS was supported by NSF grant 2006069 (to TW).

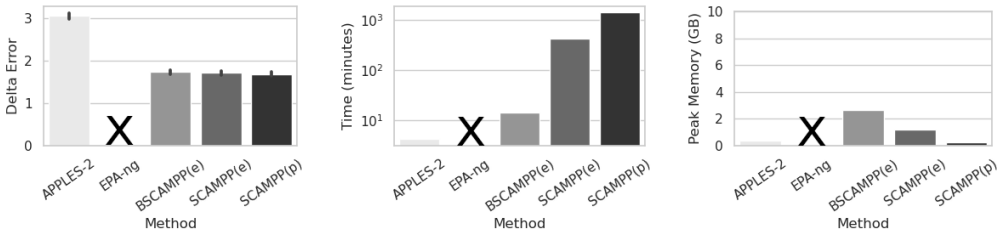


Figure 1 Results on the nt78 dataset (68,132 sequences in the backbone tree, placing 10,000 query sequences). Delta error measures placement error in number of edges, and is averaged across the query sequences. EPA-ng reported out-of-memory issues given 64 GB and 16 cores.

References

1 Metin Balaban, Yueyu Jiang, Daniel Roush, Qiyun Zhu, and Siavash Mirarab. Fast and accurate distance-based phylogenetic placement using divide and conquer. *Molecular Ecology Resources*, 22(3):1213–1227, 2022.

2 Pierre Barbera, Alexey M Kozlov, Lucas Czech, Benoit Morel, Diego Darriba, Tomáš Flouri, and Alexandros Stamatakis. EPA-ng: massively parallel evolutionary placement of genetic sequences. *Systematic Biology*, 68(2):365–369, 2019.

3 Gillian Chu and Tandy Warnow. SCAMPP+FastTree: improving scalability for likelihood-based phylogenetic placement. *Bioinformatics Advances*, 3(1), January 2023. vbad008. doi:10.1093/bioadv/vbad008.

4 Yueyu Jiang, Metin Balaban, Qiyun Zhu, and Siavash Mirarab. DEPP: deep learning enables extending species trees using single genes. *Systematic Biology*, 72(1):17–34, 2023.

5 Frederick A Matsen, Robin B Kodner, and E Virginia Armbrust. pplacer: linear time maximum-likelihood and Bayesian phylogenetic placement of sequences onto a fixed reference tree. *BMC bioinformatics*, 11(1):538, 2010.

6 Morgan N Price, Paramvir S Dehal, and Adam P Arkin. FastTree 2–approximately maximum-likelihood trees for large alignments. *PloS one*, 5(3):e9490, 2010.

7 Nidhi Shah, Erin K. Molloy, Mihai Pop, and Tandy Warnow. TIPP2: metagenomic taxonomic profiling using phylogenetic markers. *Bioinformatics*, 2021. doi:10.1093/bioinformatics/btab023.

8 Eleanor Wedell, Yirong Cai, and Tandy Warnow. SCAMPP: Scaling alignment-based phylogenetic placement to large trees. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pages 1417–1430, 2022. doi:10.1109/TCBB.2022.3170386.

Optimal Subtree Prune and Regraft for Quartet Score in Sub-Quadratic Time

Shayesteh Arasti
Computer Science and Engineering Department, University of California, San Diego, CA, USA

Siavash Mirarab[1]
Electrical and Computer Engineering Department, University of California, San Diego, CA, USA

Abstract

Finding a tree with the minimum total distance to a given set of trees (the median tree) is increasingly needed in phylogenetics. Defining tree distance as the number of induced four-taxon unrooted (i.e., quartet) trees with different topologies, the median of a set of gene trees is a statistically consistent estimator of the species tree under several models of gene tree species tree discordance. Because of this, median trees defined with quartet distance are widely used in practice for species tree inference. Nevertheless, the problem is NP-Hard and the widely-used solutions are heuristics. In this paper, we pave the way for a new type of heuristic solution to this problem. We show that the optimal place to add a subtree of size m onto a tree with n leaves can be found in time that grows quasi-linearly with n and is nearly independent of m. This algorithm can be used to perform subtree prune and regraft (SPR) moves efficiently, which in turn enables the hill-climbing heuristic search for the optimal tree. In exploratory experiments, we show that our algorithm can improve the quartet score of trees obtained using the existing widely-used methods.

2012 ACM Subject Classification Applied computing → Bioinformatics

Keywords and phrases Phylogenetics, Gene tree discordance, Quartet score, Quartet distance, Subtree prune and regraft, Tree search, ASTRAL

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.4

Supplementary Material *Text (Figures)*: `https://doi.org/10.6084/m9.figshare.23582676`

Funding The work was supported by National Health Institute (NIH) grant R35GM142725.
Siavash Mirarab: was supported through National Science Foundation grant IIS 1845967.

1 Introduction

Phylogenetic reconstruction literature has now fully embraced Maddison's insight that gene trees and species trees differ [16], and a plethora of methods exist for accounting for such discordance when inferring a species tree [3]. A prominent biological cause of discordance is incomplete lineage sorting (ILS), a process that can never be fully discounted and is studied using the multi-species coalescent model (MSC) [23, 26]. Under the MSC model, the species tree is the parameter of a distribution that generates true gene trees disagreeing with the species tree due to ILS [20]. In developing methods that account for ILS, an easily obtained result [1] has proved invaluable: Under the MSC distribution parameterized by a four-taxon (i.e., quartet) species tree, the unrooted gene tree topology matching the species tree unrooted topology has a higher (marginal) probability than the two alternative topologies. This observation is what underlines many of the existing methods, including the popular methods ASTRAL [21], SVDQuartets [7], and others [14, 29]. As a result, the decades-old idea of inferring phylogenetic trees by dividing the dataset into quartets [9, 32, 31] has gained increased attention in recent years.

[1] corresponding author

A naive method dividing a set of n taxa into quartets will have one of two problems: it either has to scale with $\Omega(n^4)$, which is impractical for modern datasets, or it has to subsample quartets to an asymptotically smaller size. Neither is ideal, but also, neither is necessary. Take the simple question of computing the fraction of $\binom{n}{4}$ quartet topologies shared between two trees. Naively, we list all topologies and compare them, which requires $\Omega(n^4)$ time. However, a relatively straightforward algorithm [6] can compute the score using a post-traversal of one tree versus each node of the other tree in time that grows with $\Theta(n^2)$. But, we can do even better. As Brodal *et al.* have shown [4], the quartet score can be computed in $O(n \log^2(n))$ using a complex algorithm that uses a sophisticated data structure called Hierarchical Decomposition Tree (HDT) to represent trees. Thus, sub-quadratic (and hence, scalable) analysis using quartets is doable.

Beyond computing quartet distance, we can seek to minimize it. For an input set of k unrooted (gene) trees, the median quartet tree is the (species) tree that maximizes the number of input tree quartets shared with the output tree. Several theoretical and empirical results have shown that the median quartet tree is robust to not just ILS but also HGT [8] or duplication and loss [19, 15, 11], making the search for this tree a fruitful optimization objective. One of the most widely used solutions to this problem is ASTRAL [21], which approaches this NP-Hard [13] problem using a dynamic programming algorithm capable of precisely solving the problem; nevertheless, to achieve scalability, it restricts the search space to the trees that can be built from a predefined set $\mathcal{X}$ of clusters. With this construction, the running time of the latest version, ASTRAL-III [35], becomes $O(|\mathcal{X}|^{3/\log_3(27/4)}nk)$, which is $O((nk)^{2.726})$ because ASTRAL-III restricts $|\mathcal{X}| = O(nk)$; on typical datasets, the scaling seems to be close to n^2k^2 empirically. While ASTRAL is reasonably scalable, it does not take advantage of the HDT data structure proposed by Brodal *et al.* [4] (referred to as B13 henceforth). Moreover, unlike most methods used in practice, ASTRAL does not use a hill-climbing search algorithm and instead uses a dynamic programming approach pioneered by earlier work [10, 5, 33]. This leads to a question: Is the dynamic programming algorithm used in ASTRAL the most efficient and effective way to find the quartet distance median tree or would hill-climbing be more efficient?

Building a hill-climbing heuristic requires the ability to make tree rearrangements. Having computed the score of the current tree, T, computing the quartet score of Nearest Neighbour Interchange (NNI) moves around T is relatively simple and can be done efficiently given some pre-computations. However, NNIs are generally thought to be not enough for efficient search. All leading phylogenetic search tools use Subtree Prune and Regraft (SPR) rearrangements in addition. An SPR move on an *unrooted* query tree is defined on a node u and one of its three neighbors u'; it prunes the subtree pending from the $u \leftrightarrow u'$ edge (including that edge) and grafts back this subtree on another edge $v \leftrightarrow w$, creating two new edges $v \leftrightarrow u$ and $w \leftrightarrow u$. Knowing the quartet score of T, it is far from clear how to compute the score of the tree after the SPR move. Naively, the B13 algorithm can recompute the score in $O(n \log^2(n)k)$ after each move, leading to $O(n^2 \log^2(n)k)$ to find the optimal SPR move for each pruned edge $u \leftrightarrow u'$ and $O(n^3 \log^2(n)k)$ to test all possible SPR moves over a tree. This process of finding optimal SPR moves has to happen numerous times, rendering such an algorithm impractical. Recognizing this difficulty, all existing median quartet tree heuristics use alternatives such as ASTRAL's dynamic programming [21] or graph-based techniques used in weighted quartet max-cut [2].

Recent advances have opened up the way for a hill-climbing quartet optimization approach. Mai and Mirarab [17] showed how to prune and regraft single-taxon subtrees to their *optimal* place in $O(n \log^2(n)k)$ time. As we see, it is trivial to use that algorithm to prune and regraft

a subtree of size m in $O((n-m)m\log(n-m)\log(n)k) = O(n^2\log^2(n)k)$ time. In this paper, we show a key result: a subtree of size m can be pruned and regrafted onto the query tree with $O((n-m)\log(n-m)\log(n)k)$ time complexity, which is nearly independent of the size of the clade m (and is $O(n\log^2(n)k)$ in the worst case). This result allows us to complete a full round of optimal moves (i.e., testing all subtrees to find their optimal placement) in $O(n^2\log^2(n)k)$ time. This improved running time starts to make a hill-climbing strategy viable. After describing the algorithm, which we have implemented and made available (github.com/shayesteh99/Quartet_SPR), we present exploratory experiments showing that using SPR moves, the quartet score obtained by existing tools (ASTER and ASTRAL-III) can be further improved. These results provide us with all the necessary elements for developing a full-fledged hill-climbing quartet score optimizer, a task that future work can easily tackle.

2 Materials and Methods

2.1 Problem Definition and Notations

Let $T = (E, V)$ be a tree rooted at r_T. All leaves of T are labeled and denoted by $L(T)$. We use $v = p(u)$ to denote the parent of u and use $C(u)$ to refer to the subtree that includes vertex u after removing the edge (v, u) from the tree T. The *placement* of a subtree $C(w)$ on the edge $e = (v, u)$ of T is denoted by $\mathcal{P}_T(C(w), u)$ and is obtained by adding a degree-2 node p_u to $e = (v, u)$ to obtain (v, p_u) and (p_u, u), and connecting the subtree $C(w)$ to p_u by adding the edge (p_u, w). A *rooting* of T on $e = (v, u)$ is denoted by T_u and is obtained by dividing the edge $e = (v, u)$ into two edges (r, v) and (r, u) and reversing the direction of all edges in the path from v to r_T. Note T can be multifurcating and let d be the maximum degree among all nodes of T.

A *triplet* is a tree induced from any arbitrary set of three leaves in $L(T)$, and the least common ancestor of these three leaves is called its *anchor*. Similarly, a *rooted quartet* is a tree induced from any arbitrary set of four leaves in $L(T)$, and the *anchor* is the least common ancestor (LCA) of the four leaves. An *unrooted quartet* is the unrooted tree induced by the four leaves (unless otherwise specified, we use "quartet" to refer to an unrooted tree). A triplet on the set of leaves x, y, z is called a *matching* or *shared* triplet in trees T_1 and T_2, if $x, y, z \in L(T_1) \cap L(T_2)$ and it has the same topology in both trees. The triplet score of the trees T_1 and T_2 is defined as the number of matching triplets of T_1 and T_2 and is denoted by $S_T(T_1, T_2)$. Similarly, a quartet on the set of leaves w, x, y, z is called a matching or shared quartet in T_1 and T_2, if $w, x, y, z \in L(T_1) \cap L(T_2)$ and it has the same *unrooted* topology in both trees. The quartet score of the trees T_1 and T_2 is defined as the number of matching quartets between T_1 and T_2 and is denoted by $S_Q(T_1, T_2)$.

We define the Quartet Subtree Pruning and Regrafting (Q-SPR) problem as follows:

▶ **Definition 1** (Q-SPR Problem). *Given a rooted query tree $T' = (E, V)$, a set of arbitrarily rooted reference trees $\mathcal{R} = \{R_1, R_2, ..., R_k\}$, and $u \in V$, find*

$$\arg\max_{w} \sum_{i=1}^{k} S_Q(\mathcal{P}_T(C(u), w), R_i)$$

where T is the tree induced on $L(T') \setminus L(C(u))$.

We let $n = |T'|$ and $m = |C(u)|$, and hence $|T| = n - m$. We shorten the query subtree $C(u)$ to C when clear by context. Note that the solution to Q-SPR is the optimal position for regrafting subtree C after pruning it from the tree T', hence the name (see Figure 1 for a demonstration of the Q-SPR problem).

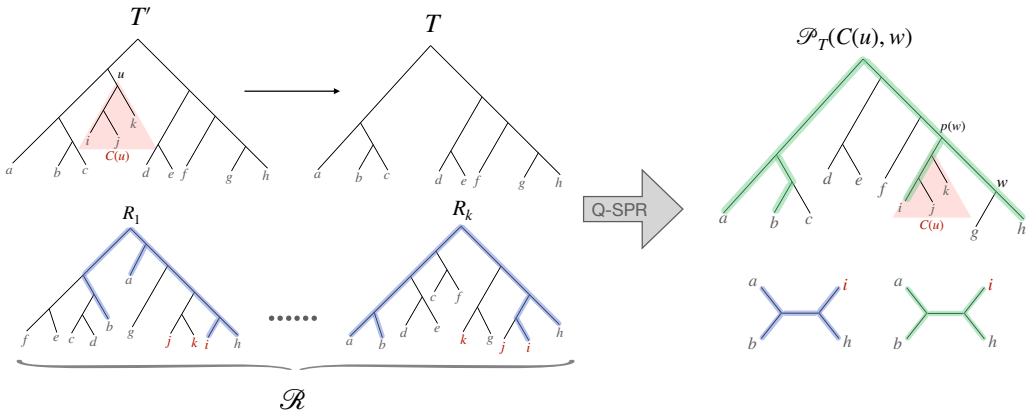


Figure 1 An overview of the Q-SPR problem. Given the query tree T', a set of reference trees $\mathcal{R}$, and a node u, the Q-SPR problem aims to find the optimal placement of $C(u)$ on T by maximizing the number of shared *unrooted* quartets between $\mathcal{P}_T(C(u), w)$ and the reference trees. An example of such quartets is highlighted in both the reference trees and the final placement.

The Q-SPR problem is an extension of the Maximum-matching Quartet Placement (MQP) problem, which is the special case of $m = 1$ and is defined as follows: Given an unrooted tree T, a single taxon q, and a set of unrooted reference trees $\mathcal{R}$, find the placement of taxon q on the tree T, denoted by T_q that maximizes $\sum_{i=1}^{k} S_Q(T_q, R_i)$. MQP is equivalent to the Maximum Triplet Rooting (MTR) problem, defined as follows: Given an unrooted tree T and a set of rooted reference trees $\mathcal{R}$, find a rooting of T on $u \in V(T)$ denoted by T_u that maximizes $\sum_{i=1}^{k} S_T(T_u, R_i)$. To see the equivalence of MQP and MTP [17], consider rooting the reference tree(s) on the query taxon q and then solving the MTR problem. The best placement of q would be at the root of the optimal rooting of the query tree T with respect to the reference tree(s).

2.2 Related work

While the Q-SPR problem is not studied in the past to our knowledge, MQP and MTR have been studied [25]. Most recently, tripVote [17] solved MTR and MQP by extending the B13 method for computing triplet scores. For a single reference tree R, tripVote computes the triplet score between the alternative rooting T_u of T and the rooted reference tree R using:

$$S_T(T_u, R) = S_T(T_{p(u)}, R) - \tau^i_{p(u)} - \tau^r_{p(u)} + \tau^o_u + \tau^r_u \tag{1}$$

where $\tau^i_{p(u)}$ is the number of shared triplets between T and R that are anchored at $p(u)$ in T, $\tau^r_{p(u)}$ is the number of shared triplets between $T_{p(u)}$ and R that are anchored at the root of $T_{p(u)}$, τ^o_u is the number of shared triplets between T_u and R that are anchored at $p(u)$ in T_u, and τ^r_u is the number of shared triplets between T_u and R that are anchored at the root of T_u. Clearly, given the three values τ^i_u, τ^o_u, and τ^r_u for every node $u \in V(T)$, the triplet score of rooting at every node can be computed in $O(1)$ using a pre-order traversal of T.

To compute these values efficiently, tripVote uses the coloring scheme of B13 for the leaf set in T and R. Tree T is traversed in a preorder manner and at each node, we recolor the leaves. Each degree d node u of T with children $c(u) = v_1, \ldots, v_{d-1}$ and parent $v_0 = p(u)$ creates d subtrees: $C(v_1), \ldots, C(v_{d-1})$, and $T - C(u)$. After recoloring based on u, each leaf in the $C(v_i)$ subtree would be assigned the color $i \in [1, d-1]$, and all the leaves in the $T \setminus C(u)$ subtree would have the color 0. The leaves of R are also assigned the same colors as T.

This coloring method enables tripVote to compute τ_u^i, τ_u^o, and τ_u^r in a top-down traversal of the nodes in T, where for each node the colors of the leaves in both trees are updated accordingly. After recoloring τ_u^i, τ_u^o, τ_u^r values can then be computed using predefined counters in the reference tree R in constant time. Colors are updated one leaf at a time by traversing R from that leaf to the root. Thus, each leaf recoloring and updating counters, if done naively, would require $O(H(R))$ operations, where $H(R)$ is the height of R. To prevent a quasi-quadratic total complexity, tripVote represents the reference trees using the HDT data structure used by B13. An HDT (Hierarchical Decomposition Tree) is a balanced binary tree data structure constructed from the nodes in the reference tree R within a time complexity of $O(n)$, where n represents the size of the reference tree. Each node in the HDT, also referred to as a component, consists of several nodes in the reference tree in a way that ensures local balance in the HDT, meaning that each component X in the HDT with m leaves has $O(\log(m))$ height. Thus, updating the HDT counters for each leaf recoloring will have $O(\log(n))$ complexity (see Table S1 for a comprehensive list of the components of the HDT).

tripVote uses two main counters in the HDT that are defined as follows (ρ^X was similar to B13 counters while π_j^X was new):

- ρ^X: The number of triplets that belong to the component X of the HDT and match the triplets anchored at the root of the alternative rooting of the query tree T_u.
- π_j^X: The number of triplets that belong to the component X of the HDT and match the triplets anchored at u in the alternative rooting of the query tree T_{v_j} where v_j corresponds to one of the child nodes of u ($j \in [1, d]$) or the parent node $p(u)$ ($j = 0$) in the query tree.

It immediately follows that HDT counters ρ^X and π_j^X are equivalent to the query tree parameters τ_u^i, τ_u^o, and τ_u^r at the root ($\mathfrak{R}$) of the HDT:

$$\tau_u^i = \pi_0^{\mathfrak{R}} \qquad \tau_u^r = \rho^{\mathfrak{R}} \qquad \tau_{v_j}^o = \pi_j^{\mathfrak{R}} \ .$$

tripVote, similar to B13, keeps $O(d^2)$ auxiliary counters for each HDT component to compute ρ^X and π_j^X efficiently using recursive functions of the colors of the leaves $j \in [0, d]$. Thus, for updating the color of each leaf, only the counters of its ancestors need to get updated, which can be done in $O(d^2 \log(n))$ using a bottom-up traversal of the HDT. Using a smaller-half trick, similar to B13, tripVote ensures $O(n \log(n))$ leaf recoloring in total. As a result, the total complexity of the algorithm is $O(d^2 n \log^2(n))$ for one reference tree and $O(kd^2 n \log^2(n))$ for k reference trees.

The tripVote algorithm can be used to solve the Q-SPR problem. It is easy to see that the optimal solution to Q-SPR can be obtained by placing each of the query taxa on the tree independently, noting the change in quartet score for each query for each branch, and choosing the branch that optimizes the total change at the end. This algorithm requires $O(kd^2(n-m)\log(n-m)\log(n)m)$ time, which is $O(kd^2 n^2 \log^2(n))$ for $m = \Theta(n)$ and becomes impractical for large trees.

2.3 Quartet SPR Placement Algorithm

We now propose an algorithm based on tripVote to solve the Q-SPR problem in a quasi-linear time. TripVote differentiates between the query taxon and the other taxa in the reference tree(s) by rooting the reference tree on the query taxon. However, since the Q-SPR problem has multiple query taxa that can be scattered in the reference tree(s), this approach is not viable. Therefore, we devise a quartet counting method that does not rely on a specific rooting of the reference tree(s). Instead of counting shared triplets among the query tree and the rooted version of the reference tree, we directly count the number of shared quartets

between the query tree and an arbitrary rooting of the reference tree. This method of counting presents new challenges as we have to keep track of the positions of the query taxa in the reference tree(s), and also introduce new counters for counting quartets. Below, we describe solving Q-SPR for one reference tree R; extending the approach to multiple reference trees follows trivially.

2.3.1 Counting Rooted Quartets

We select a new color, -1, for coloring the leaves of the query subtree C to be able to store their information in the counters. These leaves colored -1 are going to be present only in the reference tree, and unlike other leaves, their color is fixed throughout the algorithm.

To find the number of shared quartets between any potential placement of the subtree C on the query tree T and the reference tree R, we only need to consider the quartets that have a single taxon from $L(C)$ and three taxa from $L(T)$. This is true because the topology for quartets with zero or more than one taxon from $L(C)$ does not depend on the placement of C. We refer to the relevant quartets with one taxon from $L(C)$ as *solo* quartets. For each solo quartet, removing the sole taxon from C creates a triplet, which we will refer to as its *associated* triplet.

Since the reference tree is traversed as a rooted tree, to count the number of shared solo quartets between the query tree and the reference tree, we need to consider the rooted version of the quartets. We define the anchor of a rooted quartet as the least common ancestor of all the four taxa in the quartet; we count each quartet when we arrive at its anchor. We adopt the recursion (1) from tripVote to computing the quartet score of a node $u \in V(T)$:

$$S_Q(\mathcal{P}_T(C,u),R) = S_Q(\mathcal{P}_T(C,p(u)),R) - \varphi^i_{p(u)} - \varphi^r_{p(u)} + \varphi^o_u + \varphi^r_u \tag{2}$$

where φ^i_u, φ^r_u, and φ^o_u are defined as follows:

- φ^i_u: The number of shared solo quartets between the placement $\mathcal{P}_T(C,u)$ and R, where the associated triplet is anchored at u in T.
- φ^r_u: The number of shared solo quartets between the placement $\mathcal{P}_T(C,u)$ and R, where the associated triplet is anchored at the root r_{T_u} in the alternative rooting T_u.
- φ^o_u: The number of shared solo quartets between the placement $\mathcal{P}_T(C,u)$ and R, where the associated triplet is anchored at $p(u)$ in the alternative rooting of T_u.

To compute these variables, we first need a definition:

▶ **Definition 2.** *A quartet q is said to belong to a HDT component X, if and only if all four leaves of q belong to X.*

We now update the definitions of the main counters ρ^X and π^X_j associated with the HDT:

- ρ^X: The number of solo quartets that belong to the component X of the HDT and match the solo quartets in the placement $\mathcal{P}_T(C,u)$ where the associated triplet is anchored at the root of the alternative rooting T_u.
- π^X_j: The number of solo quartets that belong to the component X of the HDT and match the solo quartets in the placement $\mathcal{P}_T(C,u)$ where the associated triplet is anchored at u in the alternative rooting of the query tree T_{v_j} where v_j corresponds to one of the child nodes of u ($j \in [1,d]$) or the parent node $p(u)$ ($j = 0$) in the query tree.

For a particular unrooted solo quartet q in the query tree T, we consider every possible rooted topology of q in the reference tree R in order to compute ρ^X and π^X_j for every component X in the HDT. Each rooted quartet is counted exactly once at the anchor of the quartet. Same as in tripVote, the counters ρ^X and π^X_j are associated with the query tree parameters φ^i_u, φ^r_u, and φ^o_u at the HDT root $\mathfrak{R}$:

$$\varphi_u^i = \pi_0^{\mathfrak{R}} \qquad \varphi_u^r = \rho^{\mathfrak{R}} \qquad \varphi_{v_j}^o = \pi_j^{\mathfrak{R}} \ .$$

The ρ^X and π_j^X counters can be computed recursively in a postorder traversal on the HDT components, in an efficient manner:

▶ **Lemma 3.** *The ρ^X and π_j^X counters of the HDT component X can be updated in $O(d^2)$ time assuming the counters for children of X are already calculated.*

Proof. The recursion equation for each component depends on the type of the component (see Table S1). For the I and L types, ρ^X and π_j^X are set to zero as they correspond to a single node in the reference tree R and do not contain any quartets. For a component X with two child components, a quartet belongs to X if it belongs to one of the child components of X or it has at least one leaf from each child component of X. Thus, for any component X of type $IG \to C$, $CC \to C$, or $GG \to G$, with child components X_1 and X_2, we define ρ^X and π_j^X as follows:

$$\begin{aligned} \rho^X &= \rho^{X_1} + \rho^{X_2} + \rho^X_{comb} \\ \pi_j^X &= \pi_j^{X_1} + \pi_j^{X_2} + \pi^X_{comb_j} \end{aligned}$$

where ρ^X_{comb} is the number of quartets that match the criteria for ρ^X and have at least one leaf from each of the child components of X, and $\pi^X_{comb_j}$ is the number of quartets that match the criteria for π_j^X and have at least one leaf from each of the child components of X. Thus, ρ^X_{comb} and $\pi^X_{comb_j}$ of a $IG \to C$ component is set to zero, as an I component in the HDT corresponds to a single internal node in the reference tree and does not contain any leaves. For any other HDT component X, ρ^X_{comb} and $\pi^X_{comb_j}$ are computed with the help of a set of auxiliary HDT counters defined by B13. The rooted quartets counted in ρ^X_{comb} must have either $((i,j),(-1,0))$ or $((0,0),(-1,i))$ unrooted topologies, where $i,j \in [1,d]$. The rooted quartets counted in $\pi^X_{comb_j}$ must have the unrooted topology $((i,i),(-1,k))$ where $i,k \in [0,d]$ and $i,k \neq j$. We provide the full set of equations for the rooted quartets with the topologies $((i,j),(-1,0))$ and $((0,0),(-1,i))$ for ρ^X_{comb} , and $((i,i),(-1,k))$ for $\pi^X_{comb_j}$ in Tables S2, S3, and S4, respectively, noting that iterating through all 62 cases would be cumbersome. Figure 2A demonstrates an example for how ρ^X is computed using the HDT counters and also how φ_u^r is computed using ρ^X.

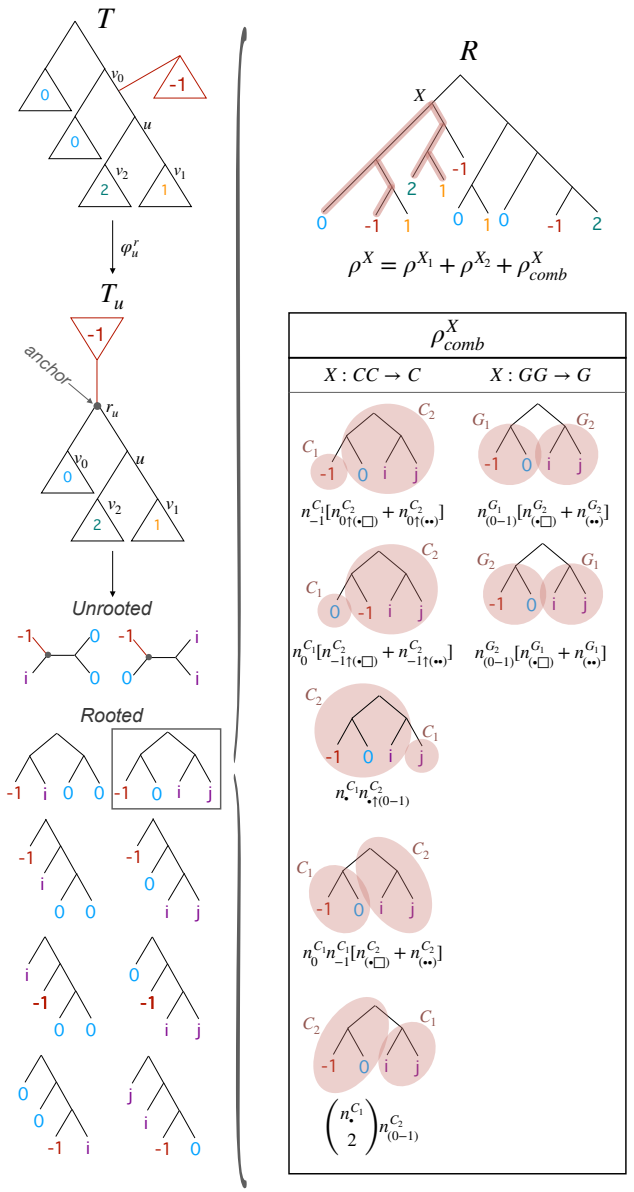
Because we only count resolved quartets and all quartets counted by our algorithm have exactly a single -1, we have $O(d^2)$ counters (i.e., of the form $(0,-1,i,j)$ for $i,j \in [1,d]$) per HDT component. From Tables S2–S4, it can be easily checked that the amortized cost of updating all the counters is constant per counter (most counters are constant while a constant number of counters need $O(d^2)$ time). ◀

To state our final results, we need one more result:

▶ **Lemma 4.** *In a top-down traversal of the query tree, a total of $(n-m)\log(n-m)$ leaf coloring steps are needed.*

Proof. The proof follows the B13 construction. There are $O(n-m)$ nodes in the tree. The smaller-half trick of B13 ensures that on each node, we avoid recoloring leaves in its largest child. This is because when we arrive at a node, the larger child already has the right color and does not need to be updated. Thus, over the entire tree, we need at most $O((n-m)\left(\frac{1}{2}+\frac{1}{2}+\ldots\right)) = O((n-m)\log(n-m))$ leaf recoloring steps. ◀

A) B)



function Q-SPR($T = (V, E), R$)
  $\mathcal{R} \leftarrow$ build-HDT(R)
  $T \leftarrow$ heaviest-child-first(T)
  color(root of T , 1)
  for $u \in$ pre-order (T) **do**
    if u is a leaf **then**
      color(u,0) and continue
    if u is the leftmost child **then**
      color(u,1)
    $v_1, v_2, \ldots, v_{d_u-1} \leftarrow c(u)$
    $\forall 2 \le i < d_u$: color(v_i, i)
    update-counters (Tables S2–S4)
    $\varphi^i_u \leftarrow \pi^{\mathfrak{R}}_0$ and $\varphi^r_u \leftarrow \rho^{\mathfrak{R}}$
    $\forall 1 \le i < d_u : \varphi^o_{v_i} \leftarrow \pi^{\mathfrak{R}}_i$
    $\forall 2 \le i < d_u$: color(v_i, 0)
  for (v, u) in pre-order (T) **do**
    $S_Q(\mathcal{P}_T(C, u), R) =$
$S_Q(\mathcal{P}_T(C, v), R) - \varphi^i_v - \varphi^r_v + \varphi^o_u + \varphi^r_u$
  $u^* = \arg\max_u S_Q(\mathcal{P}_T(C, u), R)$
  return $\mathcal{P}_T(C, u^*)$

Figure 2 A) **An example of how φ^r_u is computed for the node u in the query tree.** φ^r_u is obtained by counting the number of matching quartets resulting from taking a triplet anchored at r_u in T_u and placing one query taxon at the anchor of the triplet. To count the number of matching unrooted quartets in the reference tree R, we consider every possible rooting of the quartet. By definition, φ^r_u can be obtained by updating $\rho^{\mathfrak{R}}$, which is obtained from the demonstrated recursive equation. ρ^X is computed for each HDT component X using the counters of its child components as shown in the table. The computations of ρ^X_{comb} correspond to the first row of the Table S2. B) **Quasi-linear-time algorithm to solve the Q-SPR problem.** `color(`u`,` i`)` colors all the leaves below u with i. d_u is the of degree u.

We now can state our main results.

▶ **Theorem 5.** *The Q-SPR problem can be optimally solved using the algorithm shown in Figure 2B in* $O(kd^2(n-m)\log(n-m)\log(n))$.

Proof. We perform top-down traversal of the query tree and at each node, we recolor *some* leaves and update the HDT counters (see Figure 2B). Each recoloring of the HDT according to a new node of the query tree requires recoloring the corresponding leaf components of the HDT and updating counters of all the ancestors of those leaf components. By construction, the HDT data structure of B13 has a height of $O(\log(n))$. By Lemma 3, each such update takes $O(d^2)$; thus, the total time for recoloring one leaf is $O(d^2\log(n))$. We need a total of $O((n-m)\log(n-m))$ recoloring steps by Lemma 4. Thus, we need $O(d^2(n-m)\log(n-m)\log(n))$ operations in total for recoloring and updating counters. This produces variables $\rho^{\mathfrak{R}}$ and $\pi_j^{\mathfrak{R}}$ at the root of the HDT, which then give us φ_u^i, φ_u^r, and φ_u^o for each edge. We need to repeat this for each reference tree, resulting in $O(kd^2(n-m)\log(n-m)\log(n))$. Then, the quartet score for each node u of the query tree can be trivially computed in a second preorder traversal of the query tree using Equation (2), which takes only $O((n-m)k)$ time. Ultimately, the optimal placement of C can be obtained by finding the branch that has the maximum quartet score. ◀

2.4 Tree search using quartet SPR moves

Using our method, we can perform heuristic tree searches to find the quartet median tree with respect to the reference tree(s) R using SPR moves. At each *round*, we have a current tree T' from the previous round. We draw (without replacement) a branch (v_0, u) uniformly at random from $E(T')$, remove the subtree $C(u)$ from the query tree, and obtain the pruned tree T. The optimal placement of $C(u)$ on T denoted as v^* is obtained by solving the Q-SPR problem. If $v^* \neq u$, we place $C(u)$ on v^* to obtain the improved tree $T^* = \mathcal{P}_T(C(u), v^*)$. If not, we repeat the process until edges of $E(T')$ are exhausted. We then start over the process (i.e., a new round), using T^* as the current tree. A tree is considered final if, in a round, the algorithm performs SPR on every branch of the tree and cannot find an improvement in the quartet score. Since the query tree T' has $2n-1$ branches, and we may have to find the optimal placement for all subtrees, the total running time of performing one round of SPR on every subtree of T' has $O(kd^2n^2\log^2(n))$ complexity. For the starting tree, any tree is valid. In particular, one can be made by inserting taxa into an empty tree with an arbitrary order using the Q-SPR problem. This simple algorithm can be further optimized in several ways, which we leave for discussions and future work.

2.5 Experimental setup

While we leave the development of a fully-featured software for quartet median tree search to future work, here, we present experiments that E1) demonstrate the running time of the tool and test its accurate implementation, and E2) show intriguing results in terms of the effectiveness of SPR moves and the optimality of existing tools. These experiments are based on an implementation of our method, called Q-SPR, publicly available at github.com/shayesteh99/Quartet_SPR. Our code builds on tripVote, which itself builds on the tqDist code [28]. TqDist is a library that calculates the triplet and quartet scores between two trees, utilizing the B13 algorithm.

2.5.1 E1: Running time comparison

We modified tripVote to find the optimal placement of a clade C on a query tree T with respect to the reference tree(s) R. To do so, we place each query taxon $t_i \in L(C)$ independently and obtain the quartet score $S_Q(\mathcal{P}_T(t_i, v), R)$ for every node $v \in V(T)$. To find the total quartet score for the placement of C on v, we aggregate the quartet score obtained for each taxon t_i as follows:

$$S_Q(\mathcal{P}_T(C, v), R) = \mathcal{C} + \sum_{t_i \in L(C)} S_T(T_v, R_{t_i})$$

where $\mathcal{C}$ is a constant (quartets with zero or more than one leaf from C), which we ignore. We can then simply pick the query with optimal placement. The running time of this algorithm is expected to grow linearly with $m = |L(C)|$.

For this experiment, we used an existing 10000-taxon simulated dataset [12] with 10 replicates with gene trees disagreeing with the species tree due to both ILS and Horizontal Gene Transfer (HGT), as simulated by SimPhy [18]. We used the true species tree as the query tree, and the estimated gene trees available from the original publication (estimated using FastTree-II) [24] as the reference set. We randomly selected $\{50, 100, 200, 500, 1000, 10000\}$ taxa for each replicate and pruned the input trees (the query tree and the reference trees) to only contain the selected taxa. We also subsampled the gene trees randomly to obtain $k = 100$ trees per replicate. For each of the replicates, we performed one round of SPR on *every* subtree of the query tree and measured the time each method took to find the optimal placement for each subtree. However, for trees of size 1000 and 10000, we restricted these analyses to subtrees of size at most 70 due to increased running time. In addition to the running times, we computed the quartet score $S_Q(\mathcal{P}_T(C, v), R)$ for every node $v \in V(T)$ and every subtree C, and compared the scores of Q-SPR to the scores of modified tripVote to ensure the correct implementation of Q-SPR.

2.5.2 E2: Improving ASTER trees

In this experiment, we tested the optimality of two leading median tree search heuristics currently in use. We tested ASTRAL-III and ASTER; the latter is a newer algorithm and is shown to be better than ASTRAL-III in the face of the missing data [34]. We used the species trees inferred from the gene trees using these methods as the starting tree to our Q-SPR hill-climbing search. If the initial tree is an optimal tree (even a locally optimal tree), then, no SPR move should improve the quartet score. We asked if this is true, and if not, how much improvement in quartet score, species tree topology, species tree branch length, and branch support can be obtained.

The experiment used an existing Simphy-simulated ILS-only 200-taxon dataset [22] that was modified by Zhang *et al.* [34] to randomly remove $\approx 5\%$ of taxa from each estimated gene tree. For the purpose of this experiment, we focused on four different model conditions in this dataset (tree height $\in \{5 \times 10^5, 2 \times 10^6\}$ corresponding to high and medium ILS, and $k \in \{50, 200\}$). Each model condition provides 50 replicates with a speciation rate of 10^{-6} and 50 replicates with a speciation rate of 10^{-7}, making a total of 100 replicates per condition. The model condition with high ILS and $k = 50$ is regarded as the most challenging model condition where Zhang *et al.* [34] observed the most pronounced differences between optimization methods (ASTRAL-III vs. ASTER) [34]. For each model condition, we ran Q-SPR on both the ASTRAL-III and the ASTER trees. To test the optimality of the starting trees, we report the number of rounds of improvement and the quartet score

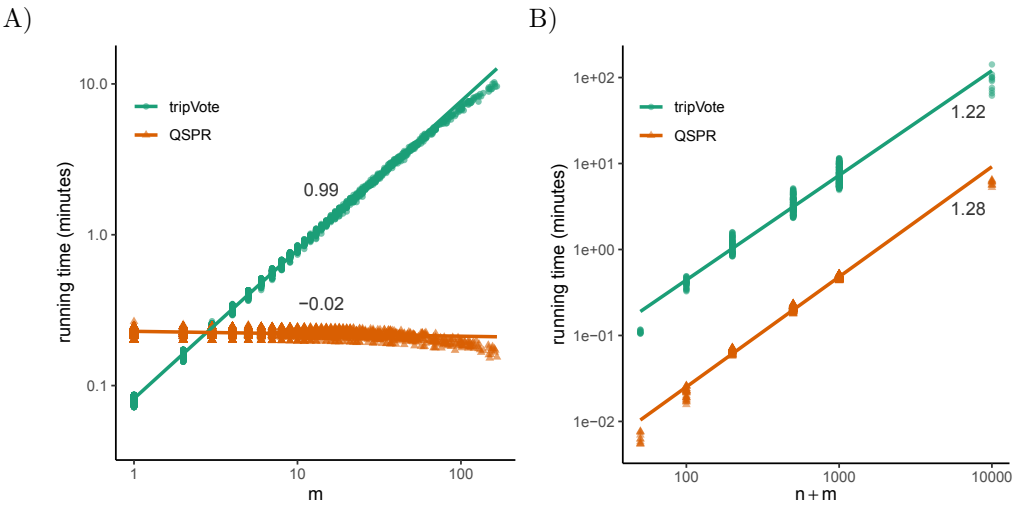


Figure 3 Results of E1. The running time is shown versus the increasing size of (A) the query subtree (m) or (B) the query tree (n). In (A), we fix $n = 500$ and in (B), we ensure $30 < m < 70$. Both axes are in log scale, making the line slope (annotated) an empirical estimate of the asymptotic running time degree. We refer to our adoption of tripVote for Q-SPR as tripVote.

improvement at the end of all rounds. We also report the total running time of Q-SPR for each experiment (not including starting tree inference). To measure accuracy, the Q-SPR and the starting trees were both compared against the known true species tree based on both normalized quartet distance and the normalized Robinson–Foulds (nRF) [27] metric. In addition, the local posterior probability (PP) [30] and the coalescent unit length for each internal branch was computed using ASTRAL-III. This method sets branch lengths to zero when the frequency of the species tree quartet topology is less than $1/3$ among gene tees (unexpected under MSC); note that localPP is $< 1/3$ under those conditions as well. We evaluated support and length for internal branches that matched or differed between the starting and final Q-SPR trees, with a particular focus on the unexpected cases with zero branch length or localPP $< 1/3$.

3 Results

3.1 Running time scaling

Comparing tripVote and Q-SPR, we ensured the two software produced identical results (quartet support of the placement clade) in all cases. The gene trees here are estimated and include polytomies. Our extensive empirical results provide validation that the complex set of recursively defined counters used by Q-SPR (Tables S2–S4) are correct.

Testing the impact of n for subclades of limited range ($30 < m < 70$) on the running time, we observe similar asymptotic behavior between tripVote and Q-SPR (Figure 3 B). The theoretical running time of both methods increases quasi-linearly with respect to the size of the query tree n, which matches our empirical estimate of the log-log slope to be only slightly above 1.0. Regardless of n, for the selected range of m, Q-SPR is always faster than tripVote by a factor of 10 to 31 (mean: 16), which is $\approx m/2.8$; this empirical observation matches the theory that Q-SPR is faster by a factor of $\Theta(m)$.

When we fix $n = 500$ and vary m, we observe the asymptotic advantage of Q-SPR over tripVote (Figure 3A). The running time of Q-SPR is nearly independent of the size of the query subtree (m), as expected, whereas the running time of tripVote increases linearly with m. Regardless of m, the running time of Q-SPR ranges between 9 and 16 seconds (mean: 13.5). In these analyses, Q-SPR is faster than tripVote in most conditions. The main exception is that tripVote is faster for subtrees of size one or two ($m \leq 2$); the likely reason is that tripVote has fewer counters to maintain than Q-SPR. At the other end, for $m > 100$ taxa, Q-SPR is on average 73 times faster than tripVote. The running time of Q-SPR has a slight downward trend. The reason is that increasing m decreases the size of the tree induced by removing the query subtree from the query tree $n-m$. In the SPR setting where n is fixed, as the subtree size increases, the size of the backbone tree $(n - m)$ decreases, which makes Q-SPR faster. The total running time of Q-SPR in this case is $O(k(n-m)\log(n-m)\log(n))$, while the overall running time for tripVote is $O(km(n - m)\log(n - m)\log(n))$.

3.2 Improving ASTER and ASTRAL trees

The quartet score of the starting tree with respect to the gene trees is guaranteed to remain fixed or improve after Q-SPR moves. We saw improvements in 310 out of 800 cases we tested, but there were wide differences depending on the model condition (Figure 4A). For example, 93% and 61% of runs starting from ASTRAL-III improved in quartet score after Q-SPR moves for 50 and 200 genes, respectively. In the other extreme, only 5% of low ILS cases starting from ASTER saw any improvements. The normalized quartet score improvements were as large as 6% in extreme cases (Figure S2) but were below 0.5% in 99% of cases. The mean improved score was 0.11% in the most difficult condition (50 genes, high ILS, starting from ASTRAL-III) and only 0.0001% in the easiest condition (200 genes, low ILS, starting from ASTER). Overall, 13.7% of branches across all trees changed.

The improvements in the optimization score tend to but do not always lead to improved species trees (Figure 4B,C). Overall, only 107 or 157 out of the 310 cases with improved quartet scores had improved accuracy compared to the true species trees in terms of nRF or quartet distance, respectively; in contrast, 137 or 149 cases had lowered accuracy (the rest had equal distance). However, while improving quartet score *can* reduce the accuracy, it never reduces it dramatically (e.g., never more than 7.5% nRF). In contrast, in *some cases*, accuracy improves dramatically after optimization (e.g., in seven cases delta nRF is 10% or more). Reassuringly, there does seem to be a positive correlation between improved score and improved accuracy and many cases with the largest quartet score improvement did have high improvements in accuracy (Figs. 4B and S2). Overall, Q-SPR optimization only slightly improved the nRF and the quartet distance between the estimated and true species trees (e.g., nRF improved by 0.1%).

Examining individual rounds of Q-SPR showed interesting patterns (Figure 4E). First, there is much variation from one replicate to another and from one condition to another, with the number of rounds ranging from 1 to 242 (mean 4.8). The mean number of rounds is as low as 1.5 for easy conditions and as high as 14 for difficult ones. Correspondingly, there is a high level of variation in terms of running times, requiring between 11 and 208 minutes on this 200-taxon dataset (Figure S1). Interestingly, some of the runs with many rounds of quartet score improvement had little or no impact on accuracy. Overall, it appears that improving the quartet score when it is already high leads to very little improvement in accuracy whereas improving the quartet score for challenging datasets with low accuracy can dramatically improve accuracy (Figure 4E).

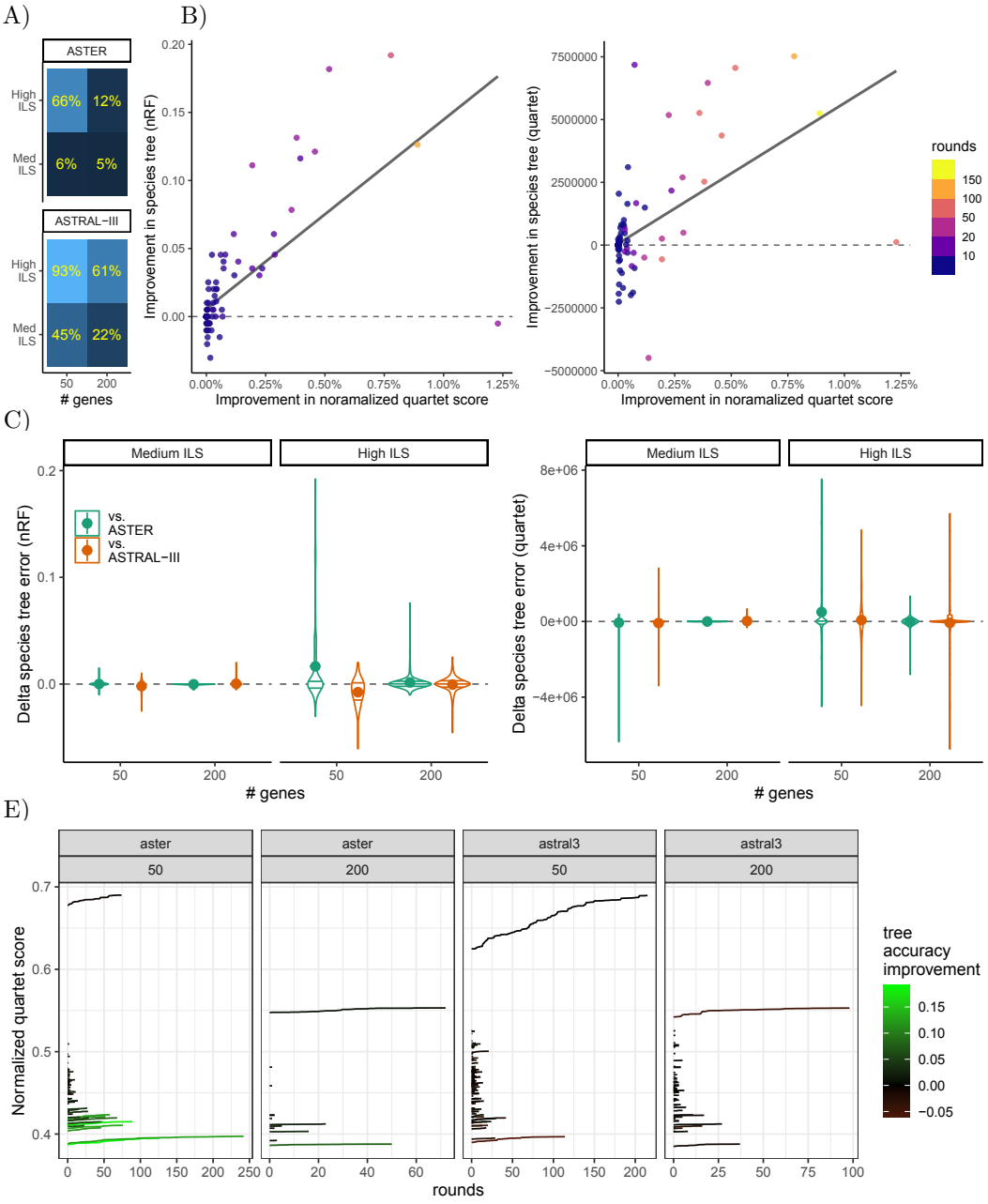


Figure 4 Results for E2. A) Percentage of the replicates with improved quartet score, starting from either ASTER or ASTRAL-III trees. B) Improvement in the quartet score of the Q-SPR algorithm above the ASTER tree (under the High ILS model condition and $k = 50$) with respect to the gene trees versus the improvement in the normalized RF or the quartet distance between the ASTER tree and the true species tree. The number of SPR rounds performed for each replicate is shown in colors. Restricted to high ILS, 50 genes, ASTER; see Figure S2 for all model conditions. C) The difference between the normalized RF and the quartet distance of the Q-SPR tree and the starting tree with respect to the true species tree for all tested conditions. Positive (negative) values indicate an improvement (reduction) in the accuracy of the starting tree. E) The normalized quartet score between the Q-SPR tree at the end of each SPR round and the gene trees under the High ILS (ILS rate $= 500k$) model conditions. The final improvement of the Q-SPR tree with respect to the true species tree is shown in colors.

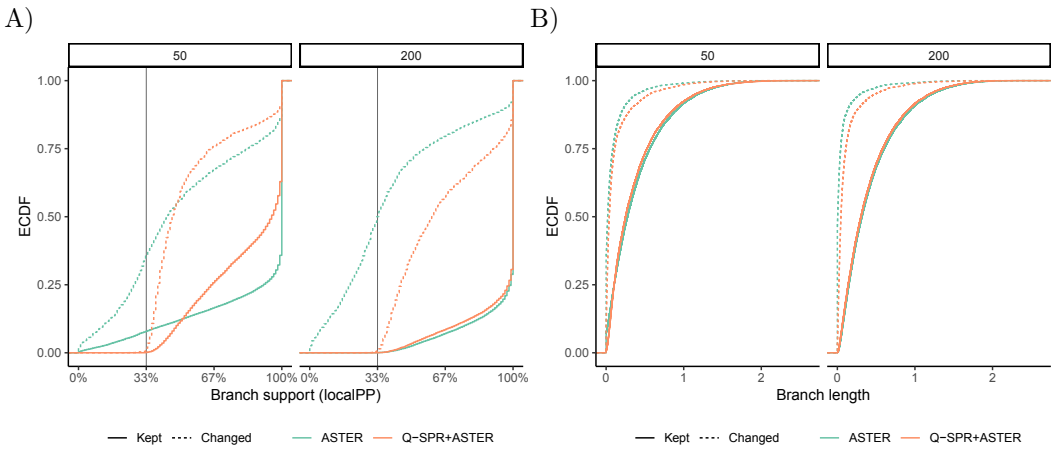


Figure 5 The results of the analysis on the branch lengths and branch supports of the improved tree using Q-SPR and the starting ASTER tree. A) The branch supports for the improved tree (ASTER + Q-SPR) and the starting tree (ASTER). The non-matching branches refer to the branches that are either removed from the ASTER tree or added to the improved tree. The branch support threshold 1/3 is shown by the dotted horizontal line. B) The empirical cumulative density function (ECDF) of the branch lengths of the improved and the starting tree for both the non-matching and matching branches. Shown is only ASTER starting tree with high ILS; see Figure S3 for full results.

Comparing the branch supports of ASTER and Q-SPR trees shows that fewer branches have unexpectedly low support (Figure 5A). Around 10% of branches before Q-SPR had support below $\frac{1}{3}$ and branch length 0. Species tree branches with these properties are problematic as they would have to have less frequency than alternative topologies in the gene trees (inconsistent with MSC). In contrast, only 0.2% of the branches after Q-SPR had support below $\frac{1}{3}$ or 0 branch length. While most of the branches changed by the Q-SPR had low support, some had high support (e.g., 16% had localPP > 0.95). Interestingly, it appears that compared to the starting trees, Q-SPR (indirectly) trades off some reduction in the number of high support branches with having fewer very low support branches.

4 Discussion

We introduce a quasi-linear algorithm for placing a query subtree on a tree. Our algorithm takes as input a set of reference trees that include all or some of the leaves in the query subtree and maximizes the total quartet score between the output tree and the reference tree(s). We demonstrated that the running time of this method improves on the state-of-the-art (tripVote) and is indeed nearly independent of the size of the query subtree. Our algorithm, called Q-SPR, can perform quartet-based subtree prune and regraft (SPR) on a given tree with respect to the reference tree(s). This efficient algorithm for SPR moves provided us with a way to enhance species tree inference starting from the output of methods such as ASTER and ASTRAL, as we demonstrated in our experiments. We showed that for both methods, the quartet score could be often improved. These improvements tended to eliminate super low support branches with zero estimated length (i.e., those that do not match MSC expectations). As such, they would make the resulting trees, and in particular support values on those trees, more interpretable. However, the impact on the overall topological accuracy was small to moderate, especially given enough genes. While improving the quartet score

generally leads to a more precise median tree for an infinite number of gene trees, this is often not the case when dealing with a finite set of gene trees. Thus, imperfect correlations between the quartet score and accuracy mean that further improving the quartet score beyond what heuristics such as ASTER and ASTRAL-III achieve has limited practical value. However, our results indicate that perhaps a faster and better heuristic method can be designed relying solely on hill-climbing using SPRs. This will be the subject of future work.

Our approach can be expanded in many ways. Having an efficient SPR move available, we can now explore the best possible way to apply SPR moves on a starting tree to obtain efficient hill climbing. Our current approach, testing all possible pruning locations at random with no attempt at guessing the best move, can likely be improved. More ambitiously, our theory can be extended to ask if some of the calculations performed for the optimal SPR move of a clade can be reused for adjacent clades, hence reducing running time. Moreover, the much faster NNI moves (which are a special case of SPR) should perhaps also be tried. The choice of the starting tree will also need to be explored. We can, for example, use the widely-used step-wise addition strategy, which incidentally, is also made possible by our algorithm (and tripVote). These improvements need to all be combined with high-performance computing features such as parallelism. Thus, while all the main algorithmic ingredients are ready, a fully-featured heuristic hill-climbing implementation requires further engineering.

References

1. E S Allman, James H. Degnan, and J A Rhodes. Identifying the rooted species tree from the distribution of unrooted gene trees under the coalescent. *J. Math. Biol.*, 62:833–862, 2011.
2. Eliran Avni, Reuven Cohen, and Sagi Snir. Weighted Quartets Phylogenetics. *Systematic Biology*, 64(2):233–242, March 2015. doi:10.1093/sysbio/syu087.
3. Paul D Blischak, Jeremy M Brown, Zhen Cao, Alison Cloutier, Kerry Cobb, Alexandria A DiGiacomo, Deren AR Eaton, Scott V Edwards, Kyle A Gallivan, and Daniel J Gates. *Species Tree Inference: A Guide to Methods and Applications*. Princeton University Press, 2023.
4. Gerth Stølting Brodal, Rolf Fagerberg, Thomas Mailund, Christian N. S. Pedersen, and Andreas Sand. Efficient Algorithms for Computing the Triplet and Quartet Distance Between Trees of Arbitrary Degree. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1814–1832, Philadelphia, PA, January 2013. Society for Industrial and Applied Mathematics. doi:10.1137/1.9781611973105.130.
5. David Bryant and Mike Steel. Constructing Optimal Trees from Quartets. *Journal of Algorithms*, 38(1):237–259, January 2001. doi:10.1006/jagm.2000.1133.
6. David Bryant, John Tsang, Paul E Kearney, and Ming Li. Computing the quartet distance between evolutionary trees. In *Algorithms and Computation. ISAAC 2001*, volume 9(11) of *LNCS*, pages 285–286. Citeseer, 2000.
7. Julia Chifman and Laura S Kubatko. Quartet Inference from SNP Data Under the Coalescent Model. *Bioinformatics*, 30(23):3317–3324, August 2014. Publisher: Oxford Univ Press. doi:10.1093/bioinformatics/btu530.
8. Ruth Davidson, Pranjal Vachaspati, Siavash Mirarab, and Tandy Warnow. Phylogenomic species tree estimation in the presence of incomplete lineage sorting and horizontal gene transfer. *BMC Genomics*, 16(Suppl 10):S1, 2015. doi:10.1186/1471-2164-16-S10-S1.
9. G. F. Estabrook, F. R. McMorris, and C. A. Meacham. Comparison of Undirected Phylogenetic Trees Based on Subtrees of Four Evolutionary Units. *Systematic Biology*, 34(2):193–200, June 1985. doi:10.2307/sysbio/34.2.193.
10. M. T. Hallett and Jens Lagergren. New algorithms for the duplication-loss model. In *Proceedings of the fourth annual international conference on Computational molecular biology - RECOMB '00*, pages 138–146, New York, New York, USA, 2000. ACM Press. doi:10.1145/332306.332359.

11 Max Hill, Brandon Legried, and Sebastien Roch. Species tree estimation under joint modeling of coalescence and duplication: sample complexity of quartet methods. *arXiv*, 2020. arXiv:2007.06697.

12 Yueyu Jiang, Metin Balaban, Qiyun Zhu, and Siavash Mirarab. DEPP: Deep Learning Enables Extending Species Trees using Single Genes. *Systematic Biology*, page 2021.01.22.427808, April 2022. doi:10.1093/sysbio/syac031.

13 Manuel Lafond and Celine Scornavacca. On the Weighted Quartet Consensus problem. *Theoretical Computer Science*, 769:1–17, May 2019. arXiv:1610.00505 Genre: Data Structures and Algorithms. doi:10.1016/j.tcs.2018.10.005.

14 Bret R. Larget, Satish K. Kotha, Colin N. Dewey, and Cécile Ané. BUCKy: Gene tree/species tree reconciliation with Bayesian concordance analysis. *Bioinformatics*, 26(22):2910–2911, November 2010. arXiv:0912.4472 Publisher: Department of Statistics, University of Wisconsin-Madison, WI 53706, USA. ISBN: 03036812. doi:10.1093/bioinformatics/btq539.

15 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, May 2021. doi:10.1089/cmb.2020.0424.

16 Wayne P. Maddison. Gene Trees in Species Trees. *Systematic Biology*, 46(3):523–536, September 1997. doi:10.2307/2413694.

17 Uyen Mai and Siavash Mirarab. Completing gene trees without species trees in sub-quadratic time. *Bioinformatics*, 38(6):1532–1541, March 2022. doi:10.1093/bioinformatics/btab875.

18 Diego Mallo, Leonardo De Oliveira Martins, and David Posada. SimPhy : Phylogenomic Simulation of Gene, Locus, and Species Trees. *Systematic Biology*, 65(2):334–344, March 2016. doi:10.1093/sysbio/syv082.

19 Alexey Markin and Oliver Eulenstein. Quartet-based inference is statistically consistent under the unified duplication-loss-coalescence model. *Bioinformatics*, page btab414, May 2021. doi:10.1093/bioinformatics/btab414.

20 Siavash Mirarab, Luay Nakhleh, and Tandy Warnow. Multispecies Coalescent: Theory and Applications in Phylogenetics. *Annual Review of Ecology, Evolution, and Systematics*, 52(1):247–268, November 2021. doi:10.1146/annurev-ecolsys-012121-095340.

21 Siavash Mirarab, Rezwana Reaz, Md. Shamsuzzoha Bayzid, Théo Zimmermann, M. S. Swenson, and Tandy Warnow. ASTRAL: genome-scale coalescent-based species tree estimation. *Bioinformatics*, 30(17):i541–i548, September 2014. doi:10.1093/bioinformatics/btu462.

22 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.

23 P Pamilo and M Nei. Relationships between gene trees and species trees. *Molecular biology and evolution*, 5(5):568–583, 1988. ISBN: 0737-4038 (Print). URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&dopt=Citation&list_uids=3193878.

24 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. Publisher: Public Library of Science. doi:10.1371/journal.pone.0009490.

25 Maryam Rabiee and Siavash Mirarab. INSTRAL: Discordance-Aware Phylogenetic Placement Using Quartet Scores. *Systematic Biology*, 69(2):384–391, August 2020. doi:10.1093/sysbio/syz045.

26 Bruce Rannala and Ziheng Yang. Bayes estimation of species divergence times and ancestral population sizes using DNA sequences from multiple loci. *Genetics*, 164(4):1645–1656, 2003. Publisher: Department of Medical Genetics, University of Alberta, Edmonton, Alberta T6G 2H7, Canada.

27 DF Robinson and LR Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1-2):131–147, 1981. URL: http://www.sciencedirect.com/science/article/pii/0025556481900432.

28 Andreas Sand, Morten K. Holt, Jens Johansen, Gerth Stølting Brodal, Thomas Mailund, and Christian N. S. Pedersen. tqDist: a library for computing the quartet and triplet distances between binary or general trees. *Bioinformatics*, 30(14):2079–2080, July 2014. doi:10.1093/bioinformatics/btu157.

29 Erfan Sayyari and Siavash Mirarab. Anchoring quartet-based phylogenetic distances and applications to species tree reconstruction. *BMC Genomics*, 17(S10):101–113, November 2016. doi:10.1186/s12864-016-3098-z.

30 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.

31 Sagi Snir, Tandy Warnow, and Satish Rao. Short Quartet Puzzling: A New Quartet-Based Phylogeny Reconstruction Algorithm. *Journal of Computational Biology*, 15(1):91–103, January 2008. doi:10.1089/cmb.2007.0103.

32 Michael Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *Journal of Classification*, 9(1):91–116, January 1992. doi:10.1007/BF02618470.

33 Cuong Than and Luay Nakhleh. Species Tree Inference by Minimizing Deep Coalescences. *PLoS Computational Biology*, 5(9):e1000501, September 2009. doi:10.1371/journal.pcbi.1000501.

34 Chao Zhang and Siavash Mirarab. Weighting by Gene Tree Uncertainty Improves Accuracy of Quartet-based Species Trees. *Molecular Biology and Evolution*, 39(12):msac215, October 2022. doi:10.1093/molbev/msac215.

35 Chao Zhang, Maryam Rabiee, Erfan Sayyari, and Siavash Mirarab. ASTRAL-III: polynomial time species tree reconstruction from partially resolved gene trees. *BMC Bioinformatics*, 19(S6):153, May 2018. doi:10.1186/s12859-018-2129-y.

S1 Supplementary Tables

Table S1 HDT components types. Each component in the HDT corresponds to a part of the reference tree R, with C and G being the super-types in the HDT.

Type	Subtype	Description
I		corresponds to an internal node in the reference tree R. Every I component is a leaf in the HDT
G		corresponds to a set of subtrees in the reference tree R where the roots of the subtrees are siblings. A G component can be one of two types:
	L	corresponds to a leaf node in the reference tree R. Every L component is a leaf in the HDT
	$GG \rightarrow G$	a component of the HDT that is the parent of two other G components
C		corresponds to a connected subset of nodes in the reference tree R where the root of one subset is a descendant of the root of the other subset. A C component can be one of two types:
	$IG \rightarrow C$	a component of the HDT that is the parent of an I and a G component
	$CC \rightarrow C$	a component of the HDT that is the parent of two C components

Table S2 Equations for computing the HDT counter ρ^{X}_{comb} for component types $CC \to C$ and $GG \to G$. Each row shows one rooted topology of the unrooted quartet $((i,j),(-1,0))$. ρ^{X}_{comb} for each component type is the sum over the equations of all the rooted topologies.

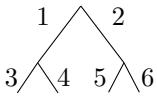
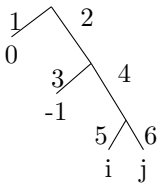
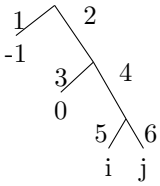
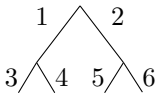
Rooted Quartet		$CC \to C$		$GG \to G$
	1 2 3 4 5, 6	$n_0^{C_1} n_{-1}^{C_1} [n_{(\bullet\square)}^{C_2} + n_{(\bullet\bullet)}^{C_2}]$ $+\binom{n_\bullet^{C_1}}{2} n_{(0-1)}^{C_2}$ $+n_0^{C_1} [n_{-1\uparrow(\bullet\square)}^{C_2} + n_{-1\uparrow(\bullet\bullet)}^{C_2}]$ $+n_{-1}^{C_1} [n_{0\uparrow(\bullet\square)}^{C_2} + n_{0\uparrow(\bullet\bullet)}^{C_2}]$ $+n_\bullet^{C_1} n_{\bullet\uparrow(0-1)}^{C_2}$	1 2	$n_{(0-1)}^{G_1} [n_{(\bullet\square)}^{G_2} + n_{(\bullet\bullet)}^{G_2}]$ $+n_{(0-1)}^{G_2} [n_{(\bullet\square)}^{G_1} + n_{(\bullet\bullet)}^{G_1}]$
	1 2 3 4 5, 6	$+n_0^{C_1} [n_{(-1(\bullet\square))}^{C_2} + n_{(-1(\bullet\bullet))}^{C_2}]$ $+[n_{[-1(\bullet\square)]}^{C_1} + n_{[-1(\bullet\bullet)]}^{C_1}] n_0^{C_2}$ $+n_{-1}^{C_1} [n_{(\bullet\square)\uparrow 0}^{C_2} + n_{(\bullet\bullet)\uparrow 0}^{C_2}]$ $+\binom{n_\bullet^{C_1}}{2} n_{-1\uparrow 0}^{C_2}$ $+n_\bullet^{C_1} n_{\bullet\uparrow -1\uparrow 0}^{C_2}$	1 2	$+n_0^{G_1} [n_{[-1(\bullet\square)]}^{G_2} + n_{[-1(\bullet\bullet)]}^{G_2}]$ $+n_0^{G_2} [n_{[-1(\bullet\square)]}^{G_1} + n_{[-1(\bullet\bullet)]}^{G_1}]$
	1 2 3 4 5, 6	$+n_{-1}^{C_1} [n_{(0(\bullet\square))}^{C_2} + n_{(0(\bullet\bullet))}^{C_2}]$ $+[n_{[0(\bullet\square)]}^{C_1} + n_{[0(\bullet\bullet)]}^{C_1}] n_{-1}^{C_2}$ $+n_0^{C_1} [n_{(\bullet\square)\uparrow -1}^{C_2} + n_{(\bullet\bullet)\uparrow -1}^{C_2}]$ $+\binom{n_\bullet^{C_1}}{2} n_{0\uparrow -1}^{C_2}$ $+n_\bullet^{C_1} n_{\bullet\uparrow 0\uparrow -1}^{C_2}$	1 2	$+n_{-1}^{G_1} [n_{[0(\bullet\square)]}^{G_2} + n_{[0(\bullet\bullet)]}^{G_2}]$ $+n_{-1}^{G_2} [n_{[0(\bullet\square)]}^{G_1} + n_{[0(\bullet\bullet)]}^{G_1}]$
	1 2 3 4 5 6	$+n_\bullet^{C_1} n_{(\bullet(0-1))}^{C_2}$ $+n_{[\bullet(0-1)]}^{C_1} n_\bullet^{C_2}$ $+n_\bullet^{C_1} n_{(0-1)\uparrow\bullet}^{C_2}$ $+n_0^{c_1} n_{-1}^{C_1} [n_{\bullet\uparrow\square}^{C_2} + n_{\bullet\uparrow\bullet}^{C_2}]$ $+n_0^{C_1} [n_{-1\uparrow\bullet\uparrow\square}^{C_2} + n_{-1\uparrow\bullet\uparrow\bullet}^{C_2}]$ $+n_{-1}^{C_1} [n_{0\uparrow\bullet\uparrow\square}^{C_2} + n_{0\uparrow\bullet\uparrow\bullet}^{C_2}]$	1 2	$+n_\bullet^{G_1} n_{[\bullet(0-1)]}^{G_2}$ $+n_\bullet^{G_2} n_{[\bullet(0-1)]}^{G_1}$

Table S3 Equations for computing the HDT counter ρ^X_{comb} for component types $CC \to C$ and $GG \to G$ (Cont.) Each row shows one rooted topology of the unrooted quartet $((0,0),(-1,i))$. ρ^X_{comb} for each component type is the sum over the equations of all the rooted topologies in Table S2 and S3.

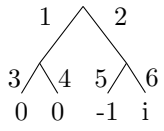
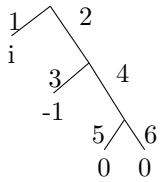
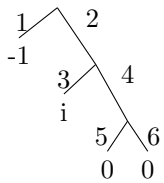
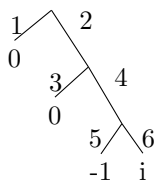
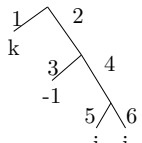
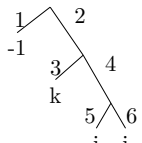
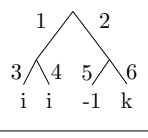
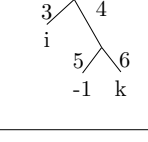
Rooted Quartet		$CC \to C$		$GG \to G$
	1 2 3, 4 5 6	$+\binom{n_0^{C_1}}{2} n^{C_2}_{(-1\bullet)}$ $+n^{C_1}_{-1} n^{C_1}_{\bullet} n^{C_2}_{(00)}$ $+n^{C_1}_{0} n^{C_2}_{0\uparrow(-1\bullet)}$ $+n^{C_1}_{-1} n^{C_2}_{\bullet\uparrow(00)}$ $+n^{C_1}_{\bullet} n^{C_2}_{-1\uparrow(00)}$	1 2	$+n^{G_1}_{(00)} n^{G_2}_{(-1\bullet)}$ $+n^{G_2}_{(00)} n^{G_1}_{(-1\bullet)}$
	1 2 3 4 5, 6	$+n^{C_1}_{\bullet} n^{C_2}_{(-1(00))}$ $+n^{C_1}_{[-1(00)]} n^{C_2}_{\bullet}$ $+n^{C_1}_{-1} n^{C_2}_{(00)\uparrow\bullet}$ $+\binom{n_0^{C_1}}{2} n^{C_2}_{-1\uparrow\bullet}$ $+n^{C_1}_{0} n^{C_2}_{0\uparrow-1\uparrow\bullet}$	1 2	$+n^{G_1}_{\bullet} n^{G_2}_{[-1(00)]}$ $+n^{G_2}_{\bullet} n^{G_1}_{[-1(00)]}$
	1 2 3 4 5, 6	$+n^{C_1}_{-1} n^{C_2}_{(\bullet(00))}$ $+n^{C_1}_{[\bullet(00)]} n^{C_2}_{-1}$ $+n^{C_1}_{\bullet} n^{C_2}_{(00)\uparrow-1}$ $+\binom{n_0^{C_1}}{2} n^{C_2}_{\bullet\uparrow-1}$ $+n^{C_1}_{0} n^{C_2}_{0\uparrow\bullet\uparrow-1}$	1 2	$+n^{G_1}_{-1} n^{G_2}_{[\bullet(00)]}$ $+n^{G_2}_{-1} n^{G_1}_{[\bullet(00)]}$
	1 2 3 4 5 6	$+n^{C_1}_{0} n^{C_2}_{(0(-1\bullet))}$ $+n^{C_1}_{[0(-1\bullet)]} n^{C_2}_{0}$ $+n^{C_1}_{0} n^{C_2}_{(\bullet-1)\uparrow0}$ $+n^{c_1}_{\bullet} n^{C_1}_{-1} n^{C_2}_{0\uparrow0}$ $+n^{C_1}_{-1} n^{C_2}_{\bullet\uparrow0\uparrow0}$ $+n^{C_1}_{\bullet} n^{C_2}_{-1\uparrow0\uparrow0}$	1 2	$+n^{G_1}_{0} n^{G_2}_{[0(-1\bullet)]}$ $+n^{G_2}_{0} n^{G_1}_{[0(-1\bullet)]}$

Table S4 Equations for computing the HDT counter $\pi^X_{comb_j}$ for component types $CC \to C$ and $GG \to G$ Each row shows one rooted topology of the unrooted quartet $((i,i),(-1,k))$. $\pi^X_{comb_j}$ for each component type is the sum over the equations of all the rooted topologies.

Rooted Quartet		$CC \to C$		$GG \to G$
	1 2 3 4 5, 6	$\sum_{i\neq j} n_i^{C_1}(n^{C_2}_{(-1(\bullet\bullet))} - n^{C_2}_{(-1(ii))} - n^{C_2}_{(-1(jj))})$ $+\sum_{i\neq j} n^{C_1}_{[-1(ii)]}(n^{C_2}_{\bullet} - n_i^{C_2} - n_j^{C_2})$ $+n_{-1}^{C_1}(n^{C_2}_{(\bullet\bullet)\uparrow\square} - n^{C_2}_{(jj)\uparrow\bullet} - n^{C_2}_{(\bullet\bullet)\uparrow j})$ $+\sum_{i\neq j} \binom{n_i^{C_1}}{2}(n^{C_2}_{-1\uparrow\bullet} - n^{C_2}_{-1\uparrow i} - n^{C_2}_{-1\uparrow j})$ $+\sum_{i\neq j} n_i^{C_1}(n^{C_2}_{i\uparrow -1\uparrow\bullet} - n^{C_2}_{i\uparrow -1\uparrow j})$	1 2	$\sum_{i\neq j} n^{G_2}_{[-1(ii)]}(n^{G_1}_{\bullet} - n_i^{G_1} - n_j^{G_1})$ $+\sum_{i\neq j} n^{G_1}_{[-1(ii)]}(n^{G_2}_{\bullet} - n_i^{G_2} - n_j^{G_2})$
	1 2 3 4 5, 6	$+n_{-1}^{C_1}(n^{C_2}_{(\bullet(\square\square))} - n^{C_2}_{(j(\bullet\bullet))} - n^{C_2}_{(\bullet(jj))})$ $+(n^{C_1}_{[\bullet(\square\square)]} - n^{C_1}_{[j(\bullet\bullet)]} - n^{C_1}_{[\bullet(jj)]})n_{-1}^{C_2}$ $+\sum_{i\neq j} n_i^{C_1}(n^{C_2}_{(\bullet\bullet)\uparrow -1} - n^{C_2}_{(ii)\uparrow -1} - n^{C_2}_{(jj)\uparrow -1})$ $+\sum_{i\neq j} \binom{n_i^{C_1}}{2}(n^{C_2}_{\bullet\uparrow -1} - n^{C_2}_{i\uparrow -1} - n^{C_2}_{j\uparrow -1})$ $+\sum_{i\neq j} n_i^{C_1}(n^{C_2}_{i\uparrow\bullet\uparrow -1} - n^{C_2}_{i\uparrow j\uparrow -1})$	1 2	$+n_{-1}^{G_1}(n^{G_2}_{[\bullet(\square\square)]} - n^{G_2}_{[j(\bullet\bullet)]} - n^{G_2}_{[\bullet(jj)]})$ $+n_{-1}^{G_2}(n^{G_1}_{[\bullet(\square\square)]} - n^{G_1}_{[j(\bullet\bullet)]} - n^{G_1}_{[\bullet(jj)]})$
	1 2 3, 4 5 6	$+\sum_{i\neq j} \binom{n_i^{C_1}}{2}(n^{C_2}_{(-1\bullet)} - n^{C_2}_{(-1i)} - n^{C_2}_{(-1j)})$ $+\sum_{i\neq j} n_i^{C_1} n_{-1}^{C_1}(n^{C_2}_{(\bullet\bullet)} - n^{C_2}_{(ii)} - n^{C_2}_{(jj)})$ $+\sum_{i\neq j} n_i^{C_1}(n^{C_2}_{i\uparrow(-1\bullet)} - n^{C_2}_{i\uparrow(-1j)})$ $+n_{-1}^{C_1}(n^{C_2}_{\bullet\uparrow(\square\square)} - n^{C_2}_{j\uparrow(\bullet\bullet)} - n^{C_2}_{\bullet\uparrow(jj)})$ $+\sum_{i\neq j} n_i^{C_1}(n^{C_2}_{-1\uparrow(\bullet\bullet)} - n^{C_2}_{-1\uparrow(ii)} - n^{C_2}_{-1\uparrow(jj)})$	1 2	$+\sum_{i\neq j} n^{G_1}_{(ii)}[n^{G_2}_{(-1\bullet)} - n^{G_2}_{(-1i)} - n^{G_2}_{(-1j)}]$ $+\sum_{i\neq j} n^{G_2}_{(ii)}[n^{G_1}_{(-1\bullet)} - n^{G_1}_{(-1i)} - n^{G_1}_{(-1j)}]$
	1 2 3 4 5 6	$+\sum_{i\neq j} n_i^{C_1}(n^{C_2}_{(i(-1\bullet))} - n^{C_2}_{(i(-1j))})$ $+\sum_{i\neq j} (n^{C_1}_{[i(-1\bullet)]} - n^{C_1}_{[i(-1j)]})n_i^{C_2}$ $+\sum_{i\neq j} n_i^{C_1}(n^{C_2}_{(\bullet -1)\uparrow i} - n^{C_2}_{(j-1)\uparrow i})$ $+\sum_{i\neq j} n_i^{C_1} n_{-1}^{C_1}(n^{C_2}_{\bullet\uparrow\bullet} - n^{C_2}_{j\uparrow j} - n^{C_2}_{i\uparrow i})$ $+n_{-1}^{C_1}(n^{C_2}_{\bullet\uparrow\square\uparrow\square} - n^{C_2}_{j\uparrow\bullet\uparrow\bullet} - n^{C_2}_{\bullet\uparrow j\uparrow j})$ $+\sum_{i\neq j} n_i^{C_1}(n^{C_2}_{-1\uparrow\bullet\uparrow\bullet} - n^{C_2}_{-1\uparrow i\uparrow i} - n^{C_2}_{-1\uparrow j\uparrow j})$	1 2	$+\sum_{i\neq j} n_i^{G_1}(n^{G_2}_{[i(-1\bullet)]} - n^{G_2}_{[i(-1j)]})$ $+\sum_{i\neq j} n_i^{G_2}(n^{G_1}_{[i(-1\bullet)]} - n^{G_1}_{[i(-1j)]})$

Leveraging Constraints Plus Dynamic Programming for the Large Dollo Parsimony Problem

Junyan Dai
Department of Computer Science, University of Maryland, College Park, MD, USA

Tobias Rubel
Department of Computer Science, University of Maryland, College Park, MD, USA

Yunheng Han
Department of Computer Science, University of Maryland, College Park, MD, USA

Erin K. Molloy[1]
Department of Computer Science, University of Maryland, College Park, MD, USA

Abstract

The last decade of phylogenetics has seen the development of many methods that leverage constraints plus dynamic programming. The goal of this algorithmic technique is to produce a phylogeny that is optimal with respect to some objective function and that lies within a constrained version of tree space. The popular species tree estimation method ASTRAL, for example, returns a tree that (1) maximizes the quartet score computed with respect to the input gene trees and that (2) draws its branches (bipartitions) from the input constraint set. This technique has yet to be used for classic parsimony problems where the input are binary characters, sometimes with missing values. Here, we introduce the clade-constrained character parsimony problem and present an algorithm that solves this problem in polynomial time for the Dollo criterion score. Dollo parsimony, which requires traits/mutations to be gained at most once but allows them to be lost any number of times, is widely used for tumor phylogenetics as well as species phylogenetics, for example analyses of low-homoplasy retroelement insertions across the vertebrate tree of life. Thus, we implement our algorithm in a software package, called Dollo-CDP, and evaluate its utility in the context of species phylogenetics using both simulated and real data sets. Our results show that Dollo-CDP can improve upon heuristic search from a single starting tree, often recovering a better scoring tree. Moreover, Dollo-CDP scales to data sets with much larger numbers of taxa than branch-and-bound while still having an optimality guarantee, albeit a more restricted one. Lastly, we show that our algorithm for Dollo parsimony can easily be adapted to Camin-Sokal parsimony but not Fitch parsimony.

2012 ACM Subject Classification Applied computing → Molecular evolution

Keywords and phrases phylogenetics, parsimony, Dollo, Camin-Sokal, dynamic programming, constraints

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.5

Supplementary Material *Software (Source Code)*: `https://github.com/molloy-lab/Dollo-CDP`
archived at `swh:1:dir:72c3b66d3db8c6efd1a391e4bd5f2d0acd4b438b`
Software (Source Code and Data): `https://github.com/molloy-lab/dollo-study`
archived at `swh:1:dir:a7379005ab0e3f94a3f7105f2d6fb8f2f51844c8`

Funding This work was financially supported by the State of Maryland. All computational experiments were performed on the compute cluster for the Center for Bioinformatics and Computational Biology at the University of Maryland, College Park.

Acknowledgements We thank the anonymous reviewers for constructive feedback.

[1] Corresponding author

1 Introduction

The last decade of phylogenetics has seen the development of many methods that leverage constraints plus dynamic programming (CDP). The goal of CDP is to produce a phylogeny that is optimal with respect to some objective function and that lies within a constrained version of tree space. To our knowledge, the first method based on CDP was introduced in 2000 by Hallet and Lagergren [20] for gene tree parsimony, which seeks a species tree that minimizes the number of events (e.g., duplications and losses) needed to explain the input gene trees (also see the related results presented at WABI 2017 [1, 2]). Since its introduction, CDP has been leveraged for a variety of optimization problems, including minimizing deep coalescence [42], maximizing quartet support [6, 29] (see [44] for extensions to multi-copy genes), and maximizing bipartition support [37] (see [32] for extensions to multi-copy genes). All of these optimization problems take gene trees as input and seek a species tree that minimizes the dissimilarity between it and the input gene trees or alternatively maximizes the similarity (note that species trees depict the evolutionary history of species, whereas gene trees depict the evolutionary history of orthologous genomic regions, referred to as genes).

That CDP is so widely utilized in phylogenetics is likely due to it being possible to build effective constraints in practice. The constraints are effective if (nearly) optimal solutions lie within the constrained solution space and this space is small enough to enable efficient running times. As an example, the species tree estimation method `ASTRAL` solves the *bipartition-constrained maximum quartet support supertree problem* [29]. The first version of `ASTRAL` [29] formed the constrained solution space from the set of all bipartitions (i.e., branches) found in the input gene trees. The newer versions of `ASTRAL` [30, 43] allow extra bipartitions to be included as constraints with the goal of improving accuracy. `FASTRAL` [10], on the other hand, aggressively limits the number of bipartitions added with the goal of improving runtime. Overall, the popularity of the `ASTRAL` family of methods is likely due to their speed and accuracy on practical inputs.

Given the success of CDP thus far, we explore the use of this technique for traditional parsimony problems where the input are binary characters, sometimes with missing values. The remainder of this paper is organized as follows. We begin with some notation and preliminaries in Section 2. In Section 3, we introduce the *clade-constrained character parsimony problem* and present an algorithm that solves this problem in polynomial time for the Dollo criterion score. We then show how our algorithm can easily be adapted to Camin-Sokal parsimony but not Fitch parsimony. Dollo parsimony, in particular, is widely used for tumor phylogenetics [3, 4, 7, 13] as well as species phylogenetics, for example analyses of low-homoplasy retroelement insertions across the vertebrate tree of life. Prior studies have leveraged Dollo parsimony to analyze higher-level clades of birds (e.g., *Palaeognathe* [8]) and mammals (e.g., *Laurasiatheria* [11, 12]), in addition to clades at the family and genus levels (e.g., rorquals [26], mouse-eared bats [23, 25], and primates [33]).

This motivated us to implement our algorithm for the Dollo criterion score in `Dollo-CDP`, an open-source software package available on Github (`https://github.com/molloy-lab/Dollo-CDP`). In Section 4, we describe an experimental study evaluating `Dollo-CDP` on real and synthetic data sets from species phylogenetics in comparison to branch-and-bound and heuristic search. Our results, presented in Section 5, reveal that `Dollo-CDP` can improve upon heuristic search from a single starting tree, often recovering a better scoring tree. Moreover, `Dollo-CDP` scales to data sets with much larger numbers of taxa than branch-and-bound while still having an optimality guarantee, albeit a more restricted one. We conclude in Section 6 by discussing limitations and opportunities for future research.

2 Preliminaries and Background

Before introducing the clade-constrained Dollo parsimony problem, we review some preliminaries on phylogenetic trees, characters, and parsimony approaches.

2.1 Phylogenetic Trees

A *phylogenetic tree* T is an acyclic graph whose leaves (i.e., vertices with degree one) are bijectively labeled by a set S of species (note that in the context of tumor phylogenetics the leaves may be labeled by cells in a tumor). For convenience and simplicity of notation, we treat leaves and species as being interchangeable. We use $L(T)$, $V(T)$, and $E(T)$ to denote the leaf set, vertex set, and edge set of T, respectively.

Phylogenetic trees can be either *unrooted* or *rooted*; for the former the graph is undirected and for the latter the graph is directed, with edges orientated away from the root (a special vertex with in-degree zero). For rooted trees, we say that vertex u is an ancestor of v (or that v is a descendant of u) if u is on a directed path from the root to v. The *lowest common ancestor (LCA)* for a set V of vertices is the vertex that is the ancestor of all vertices in V that is farthest away from the root.

Unless otherwise noted, we will assume that all trees are *binary*. An unrooted tree is binary if all non-leaf vertices (called internal vertices) have degree three, and a rooted tree is binary if all non-leaf vertices have out-degree two (all non-root vertices have in-degree one). For rooted trees, we use $v.parent$ to indicate the parent of vertex v; similarly, we use $v.left$ and $v.right$ to denote the left and right children of vertex v, respectively. Sometimes it will be useful to restrict a tree T to a subset $X \subseteq S$ of leaves, meaning that each leaf in $S \setminus X$ is deleted from T and then all vertices with degree two are suppressed. There are three additional concepts for phylogenetic trees that will also prove useful later.

▶ **Definition 1** (Bipartition). *Each edge e in an unrooted phylogenetic tree T induces a* bipartition*, which splits the leaf set of T into two disjoint subsets whose union is the complete leaf set S. The bipartition $Bip(e) = X|Y$ is formed by deleting edge e but not its endpoints from T and assigning the leaves in one of the resulting subtrees to X and the leaves in the other to Y.*

▶ **Definition 2** (Clade). *Each vertex v in a rooted phylogenetic tree T induces a* clade*, denoted $Clade(v)$, which is simply the set of leaves that are descendants of v. A clade is* trivial *if it contains only a single element (as it must be associated with a leaf vertex) or if it contains all leaves (as it must be associated with the root vertex).*

▶ **Definition 3** (Subtree bipartition [24]). *Each internal vertex v in a rooted binary phylogenetic tree T induces a* subtree bipartition*, which partitions the leaf set of the subtree into two disjoint subsets whose union is $Clade(v)$. A subtree bipartition $SubBip(v) = X|Y$ is formed by setting X to be the leaves that are descendants of $v.left$ and Y to be the set of leaves that are descendants of $v.right$ (or vice versa).*

It is well established that an unrooted phylogenetic tree t is uniquely defined by its bipartition set $Bip(t) = \{Bip(e) : e \in E(t)\}$ and that a rooted phylogenetic tree T is uniquely defined by its clade set $Clade(T) = \{Clade(v) : v \in V(T)\}$ (see Chapters 2 and 3 in [39]). Similarly, we define the subtree bipartition set of T as $SubBip(T) = \{SubBip(v) : v \in V(T)\}$. Note that $X|Y \in SubBip(T)$ if and only if $\{X, Y, X \cup Y\} \subseteq Clade(T)$; thus, we can go back and forth between clades and subtree bipartitions.

2.2 Characters and Parsimony

A *character* c on species set S is a function mapping species in S to a state set, which is $\{0, 1\}$ for binary characters. For biological data, the 0 and 1 might refer to some feature of the genomic data with all species assigned the same state having the same feature. If 0 indicates the ancestral state and 1 indicates the derived (i.e., mutated) state, we say the characters are *ordered*; otherwise, we say they are *unordered*. We describe biological data that are encoded as ordered binary characters in Section 4. These character matrices also include a third state ? to indicate the state assignment is ambiguous or missing (in other words it could not be called as 0 or 1 for whatever reason).

For now, we assume that we are given a set $\mathcal{C}$ of binary characters with no missing values, and our goal is to find a phylogenetic tree T that best explains our data. To explain how character c evolves on a tree T on the same species set as c, we must assign states to the internal vertices of T. The quality of our explanation is determined by the number of substitutions, where a substitution is implied by any edge $e = (u, v) \in E(T)$ such that $c[u] \neq c[v]$ (and $c[u], c[v]$ are not the ambiguous state ?). This brings us to the small and large parsimony problems.

▶ **Definition 4** (Fitch Parsimony Score). *Given an unrooted binary tree T and an unordered binary character c, both on species set S, the* Fitch parsimony score*, denoted $Fitch(T, c)$, is the minimum number of substitutions needed to explain the evolution of c on T. A* Fitch-labeling *for (T, c) is a function $\hat{c}$ mapping vertices in $V(T)$ to states in $\{0, 1\}$ so that $\hat{c}[l] = c[l]$ for all $l \in L(T)$ and the number of substitutions equals $Fitch(T, c)$.*

▶ **Problem 5** (Large Fitch Parsimony Problem). *The* large Fitch parsimony problem *takes as input a set $\mathcal{C}$ of unordered binary characters, each on species set S; the output is an unrooted binary tree T on S that minimizes $\sum_{c \in \mathcal{C}} Fitch(T, c)$.*

Although the Fitch parsimony score can be computed in polynomial-time [18], the large Fitch parsimony problem is NP-hard [19]. We now consider ordered characters and rooted phylogenetic trees, which enables us to distinguish between the $0 \to 1$ substitution (indicating a mutation is gained) and the $1 \to 0$ substitution (indicating a mutation is lost).

▶ **Definition 6** (Dollo and Camin-Sokal Parsimony Score). *Given a rooted binary tree T and an ordered binary character c, both on species set S, the* Dollo parsimony score*, denoted $Dollo(T, c)$, is the minimum number of losses needed to explain the evolution of c on T when at most one gain is allowed (note that sometimes the gains are also counted as part of the score). The* Camin-Sokal parsimony score*, denoted $CamSok(T, c)$, is the minimum number of gains needed to explain the evolution of c on T when losses are prohibited.*

Just as the Fitch parsimony score was used to define the Fitch-labeling and the large Fitch parsimony problem, we can define similar concepts for the Dollo and Camin-Sokal parsimony scores. The following result about Dollo-labelings is the basis of a linear-time algorithm presented by Bouckaert *et al.* [5] for computing $Dollo(T, c)$.

▶ **Theorem 7** (Theorem 1 in [5]). *Let T be a rooted, binary tree and let c be an ordered, binary character with no missing values, both on the same species set. A Dollo-labeling for (T, c) must assign state 1 to every internal vertex on a path from the LCA of all leaves assigned state 1 (including the LCA) to any leaf assigned state 1 and assign state 0 to all remaining vertices.*

It follows that a Dollo-labeling for (T, c) is unique and always exists if we allow the root to be assigned state `1`, in which case the gain must have occurred above the root so no gains are allowed on T (see Figure 1A for an example).

Lastly, the large Dollo and Camin-Sokal parsimony problems are NP-hard [9]. The existing methods for these problems, including those implemented in `PAUP*` [36] and `Phylip` [16], are based on either heuristic searches of tree space (which have no optimality guarantees) or branch-and-bound (which is guaranteed to find an optimal solution). We describe these approaches further in Section 4 and refer the interested reader to [14, 15] for more information about parsimony problems and the related methodologies.

3 The clade-constrained large Dollo parsimony problem and a polynomial-time algorithm

We now introduce the *clade-constrained large Dollo parsimony* (CC-LDP) problem.

▶ **Problem 8** (Clade-constrained large Dollo parsimony problem). *The CC-LDP problem is defined by the following input and output.*
Input: *A set S of species, a set $\mathcal{C}$ of ordered binary characters (each on species set S), and a set Σ of clades (subsets of S)*
Output: *A rooted binary tree on species set S such that $\sum_{c\in\mathcal{C}} Dollo(T, c)$ is minimized and $Clade(T) \subseteq \Sigma$, if such a tree exists*

The CC-LDP problem has a solution provided there exists at least one binary tree T on S with $Clade(T) \subseteq \Sigma$. We present approaches for constructing Σ in Section 4 and evaluate their empirical performance in Section 5. In the remainder of this section, we assume that Σ is constructed from $\mathcal{C}$ in such a way that a solution to CC-LDP always exists. To present a polynomial-time algorithm for CC-LDP, a corollary of Theorem 7 will be useful.

▶ **Corollary 9.** *Let c be an ordered binary character on species set S (with no missing values), let T be a rooted binary tree on S, and let $\hat{c}$ be the unique Dollo-labeling for (T, c). Then for any internal vertex $v \in V(T)$, $\hat{c}[v]$ can be determined just by having knowledge of its subtree bipartition $SubBip(v)$; no other information about T is needed.*

Proof. Consider an arbitrary internal vertex $v \in V(T)$ that induces subtree bipartition $X|Y$. Without loss of generality, let X contain the leaves that are descendants of $v.left$, let Y contain the leaves that are descendants of $v.right$, and let Z contain all other leaves so $X|Y|Z$ is a partition of S. When c has no missing values, by Theorem 7, there exists a unique Dollo-labeling $\hat{c}$ for (T, c), where $\hat{c}[v] = \mathtt{1}$ if at least one of the following two cases holds (otherwise $\hat{c}[v] = \mathtt{0}$).

- **Case A:** Vertex v is the LCA of two leaves assigned state `1`.
- **Case B:** Vertex v is on the path from the LCA of two leaves assigned state `1` to one of those two leaves.

Looking at subtree bipartition $SubBip(v) = X|Y$, case A holds if condition 1 below is true, and case B holds if at least one of conditions 2 and 3 below is true.

- **Condition 1:** There exists a leaf $x \in X$ and a leaf $y \in Y$ such that $c[x] = c[y] = \mathtt{1}$.
- **Condition 2:** There exists a leaf $x \in X$ and a leaf $z \in Z$ such that $c[x] = c[z] = \mathtt{1}$.
- **Condition 3:** There exists a leaf $y \in Y$ and a leaf $z \in Z$ such that $c[y] = c[z] = \mathtt{1}$.

Therefore, $\hat{c}[v] = \mathtt{1}$ if at least one of conditions 1–3 is true; otherwise $\hat{c}[v] = \mathtt{0}$. Conditions 1–3 can be evaluated so long as we know the subtree bipartition induced by v (given the character c and its complete leaf set S). ◀

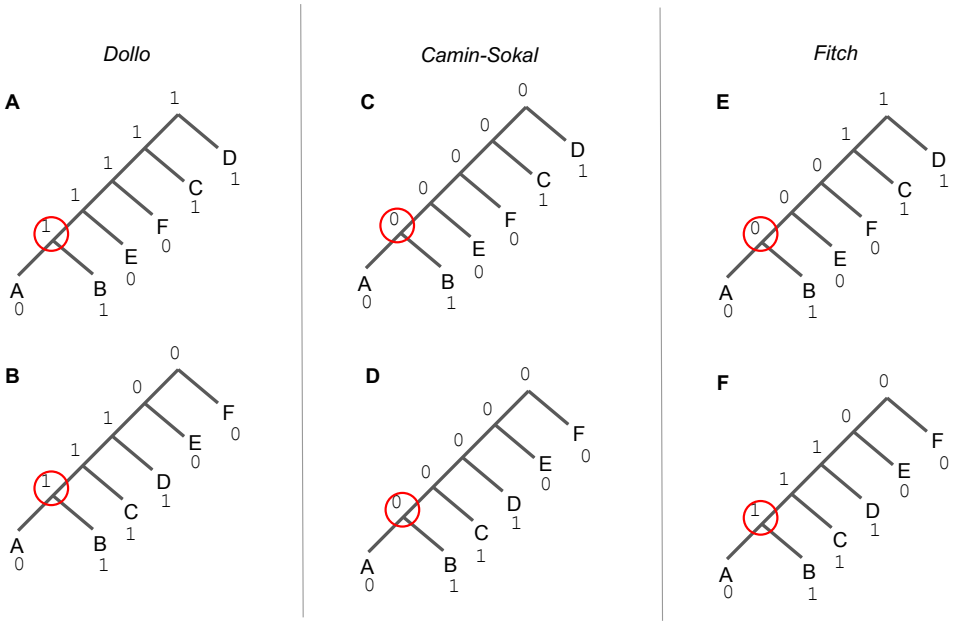


Figure 1 Let v be the internal vertex associated with the subtree bipartition $X|Y$, where $X = \{A\}$ and $Y = \{B\}$ (note that these vertices are circled in the trees above). Subfigures A and B show two different trees on the same species set with Dollo-labelings for the same character. The state assignment at v only requires us to know the subtree bipartition associated with v (Corollary 9). Vertex v is assigned state `1` because there is a leaf in Y assigned state `1` and a leaf in $S \setminus X \cup Y$ assigned state `1`. Subfigures C and D show two different trees with Camin-Sokal-labelings for the same character. The state assignment at v only requires us to know the clade associated with v (Corollary 16). Vertex v is assigned state `0` because there is a leaf in clade $X \cup Y$ assigned state `0`. Lastly, subfigures E and F show two different trees with Fitch-labelings for the same character. In subfigure E, the assignment of state `0` or `1` to v results in a score of two or three, respectively (so `0` is better). In subfigure F, the assignment of state `0` or `1` to v results in a score of three or two, respectively (so `1` is better). Thus, for the Fitch criterion score, the state assignment at v depends on more than the bipartition induced by the edge incident to v.

▶ **Theorem 10.** *Let c be an ordered binary character on species set S, and let T be a rooted binary tree on S. If c has missing values, the Dollo-labeling for (T, c) may not be unique. However, we can define a unique Dollo-labeling $\hat{c}_*$ for (T, c) with the property that $\hat{c}_*[v]$ can be determined for any internal vertex $v \in V(T)$ just by having knowledge of its subtree bipartition $SubBip(v)$; no other knowledge of T is needed.*

Proof. If c is allowed to have missing values, we compute the Dollo parsimony score *after* restricting T and c to the subset R of leaves that are *not* assigned the ambiguous state `?`. Letting $T|_R$ and $c|_R$ denote the restriction of T and c to R, respectively, we say that $\hat{c}$ is a Dollo-labeling for (T, c) if $Dollo(T, c) = Dollo(T|_R, c|_R)$.

We define a unique Dollo-labeling $\hat{c}_*$ for (T, c) with the property that it can be determined for any vertex $v \in V(T)$ with $SubBip(v) = X|Y$ by applying conditions 1–3 (see proof of Corollary 9) plus one additional condition:

- **Condition 0:** For all leaves $l \in X \cup Y$, $c[l] =$ `?`.

If condition 0 is true, we set $\hat{c}_*[v] =$ `?`. Otherwise, we proceed in the usual fashion, setting $\hat{c}_*[v] =$ `1` if at least one of conditions 1–3 is true and setting $\hat{c}_*[v] =$ `0` otherwise.

We need to show that this procedure produces a valid Dollo-labeling for (T, c) for all $v \in V(T)$. We begin by classifying the vertices of T into three groups based on the formation of $T|_R$ described next.

1. First, we identify every edge incident to the root of a maximally-sized subtree of T whose leaves are all assigned state `?`. That is, we identify every edge $a \mapsto b \in E(T)$ such that all leaves in $Clade(b)$ are assigned state `?` *and* at least one leaf in $Clade(a)$ is assigned a non-ambiguous state. Let $\mathcal{E}$ denote the set of edges with this property.
2. Second, we delete each edge in $\mathcal{E}$ but not its endpoints from T. This produces a tree T' with no leaves assigned state `?` plus a collection $\mathcal{P}$ of subtrees of T whose leaves are all assigned state `?`.
3. Lastly, we form $T|_R$ by suppressing all vertices in T' with out-degree one.

The above procedure can be used to classify vertices in $V(T)$ into three groups by which we can build a valid Dollo-labeling $\hat{c}$ for (T, c) as follows.

- **Group 1** contains every vertex $v \in V(T)$ that maps to a vertex $w \in V(T|_R)$. For each vertex v in this group, we assign $\hat{c}[v] = \hat{c}|_R[w]$, where $\hat{c}|_R$ is the Dollo-labeling for $(T|_R, c|_R)$. The idea is to propagate the state assignments for $T|_R$ back to T. Now we need to assign states to the remaining vertices in T without changing the Dollo score.
- **Group 2** contains every vertex $v \in V(T)$ that maps to a vertex $w \in \cup_{t\in\mathcal{P}} V(t)$, meaning that v is in a maximally-sized subtree of T whose leaves are all assigned state `?`. For each vertex v in this group, we set $\hat{c}[v] =$ `?`. This state assignment does not change the Dollo score because substitutions are not counted on edges with at least one of their two endpoints assigned state `?`.
- **Group 3** contains every vertex $v \in V(T)$ that does not map to any vertex in $V(T|_R) \cup \cup_{t\in\mathcal{P}} V(t)$, meaning that the corresponding vertex must have been suppressed in step 3. Thus, v maps to an edge $e \in E(T|_R)$, and exactly one of v's children is the root of a maximally-sized subtree of T whose leaves are all assigned state `?`. A state assignment that does not change the Dollo score can be achieved by assigning every vertex v in this group that maps back to the same edge $e = s \mapsto t \in E(T|_R)$ the same state: either the state assigned to s or the state assigned to t.

Because there can be multiple ways to assign states to vertices in group 3, a unique Dollo labeling for (T, c) does not necessarily exist.

We previously proposed how to define a unique Dollo-labeling $\hat{c}_*$ for (T, c) with the property that it could be determined for any internal vertex $v \in V(T)$ with $SubBip(v) = X|Y$. We now verify that our procedure produces a valid Dollo-labeling by examining the outcomes of applying conditions 0–3 to vertices in groups 1–3.

- **Group 2:** Condition 0 will be true for vertex v if and only if v is in group 2; thus, applying condition 0 assigns state `?` correctly.
- **Group 1:** The outcomes of applying conditions 1–3 to any vertex v in group 1 will be the same as applying conditions 1–3 to the corresponding vertex w in $V(T|_R)$ because $SubBip(v) = SubBip(w)$ after we remove all leaves assigned state `?` from the $SubBip(v)$.
- **Group 3:** Each vertex v in group 3 maps to an edge $e = s \mapsto t \in E(T|_R)$. For each of these vertices v, the subtree bipartition $SubBip(v) = X|Y$ will have *either* all leaves in X assigned state `?` *or* all leaves in Y assigned state `?`. Thus, for any two vertices v_1 and v_2 in group 3 that map to the same edge $e = s \mapsto t$, $SubBip(v_1) = SubBip(v_2)$ after all leaves assigned `?` are removed. It follows that v_1 and v_2 will be assigned the state when applying conditions 1–3. Lastly, we need to show that the state assigned v_1 (call v) will be the same as either the state assigned to s or the state assigned to t. For simplicity, assume v induces $SubBip(v) = X|Y$ and that all leaves in X are assigned state `?`; we can now check the outcomes of applying conditions 1–3 to v.
  - Condition 1 is false because there does not exist a leaf $x \in X$ such that $c[x] =$ `1`.
  - Condition 2 is false for the same reason.
  - Condition 3 may be either true or false.

- If condition 3 is true for v, then v is assigned state `1`. Because condition 3 is true for v, there exists a leaf $y \in Y$ and a leaf $z \in Z$ such that $c[y] = c[z] = 1$. Thus, t is on the path from the $LCA(y, z)$ to leaf y. It follows that t is also assigned state `1`.
- If condition 3 is false for v, then v is assigned state `0`. Because condition 3 is false for v, at least one of the following statements is true.
  - For all $y \in Y$, $c[y] \neq 1$. In this case, t is also assigned state `0`.
  - For all $z \in Z$, $c[z] \neq 1$. In this case, s is also assigned state `0`.

  The above logic can be applied if all leaves in Y are assigned state `?` (we just swap our arguments for conditions 2 and 3, replacing set Y with X).

This proves our claim. Note that our procedure still works when c has no missing values because all vertices in T will be in group 1. ◀

For a set $\mathcal{C}$ of k characters, each on a set S of n species, the proof of Theorem 10 gives an $O(nk)$ algorithm for determining a unique Dollo-labeling $\hat{c}$ for *all* characters in $\mathcal{C}$ at vertex v provided we know $SubBip(v)$. We refer to this procedure as `GetState` (see Algorithm 1 for details). This relationship between the subtree bipartitions of a tree and its Dollo-labeling brings us back to the CC-LDP problem, where the solution space is constrained by a set of clades, which in turn constrains the subtree bipartitions of any solution and thus its Dollo-labeling.

▶ **Corollary 11.** *Consider the set $\mathcal{T}$ of all solutions to CC-LDP given species set S, character set $\mathcal{C}$, and clade set Σ. Let $\mathcal{STB} = \{X|Y : X, Y, X \cup Y \in \Sigma\}$ be the set of all subtree bipartitions that can be formed from Σ, and let $\mathcal{STB}(A) = \{X|Y \in \mathcal{STB} : X \cup Y = A\}$ be the subset of subtree bipartitions in $\mathcal{STB}$ that are associated with clade A. Then, the unique Dollo-labeling at internal vertex v in an arbitrary phylogenetic tree $T \in \mathcal{T}$ that induces clade A must be in the set $Lab(A) = \{\texttt{GetState}(X|Y, S, \mathcal{C}) : X|Y \in \mathcal{STB}(A)\}$. Note that if v is a leaf, $Lab(A)$ is simply given by the input character set $\mathcal{C}$.*

This easily follows from Theorem 10. We refer to $Lab(A)$ as the *allowed* state assignments for clade A and $\mathcal{STB}(A)$ as the *allowed* subtree bipartitions for clade A. These quantities can be precomputed or computed on the fly. Whenever this is done, we also compute the set $\mathcal{STB}(A, st)$ of subtree bipartitions $X|Y$ such that $X \cup Y = A$ and $\texttt{GetState}(X|Y, S, \mathcal{C}) = st$.

Lastly, if we know the state assignments for some vertex v as well as its children $v.left$ and $v.right$, it is possible for us to compute the number of losses that occur on the outgoing edges of v. We simply need to count the number of times v is assigned state `1` and $v.left$ is assigned state `0`, repeating for $v.right$. This can be done if $O(k)$ time. We refer to this procedure as `CountLosses` (see Algorithm 2 for details). Now we are ready to present the dynamic programming algorithm for CC-LDP.

▶ **Dynamic Programming 12.** *Let $Dollo[A, st]$ be the smallest number of losses for any pair $(t_A, \hat{c}_A)$ such that t_A is a rooted binary tree on leaf set A that draws all its clades from Σ and $\hat{c}_A$ is an assignment of states to all vertices of t_A constrained by Corollary 11 and returning st for the root of t_A (note that these requirements imply $A \in \Sigma$ and $st \in Lab(A)$). The quantity $Dollo[A, st]$ can be computed with dynamic programming as follows.*
***Base Case:** Clade A contains a single species, i.e., $|A| = 1$.*

$$Dollo[A, st] := 0$$

***Recurrence:** Clade A contains multiple species, i.e., $|A| > 1$.*

$$Dollo[A, st] := \min_{X|Y \in \mathcal{STB}(A,st), St_X \in Lab(X), St_Y \in Lab(Y)} Dollo[X, St_X] + Dollo[Y, St_Y] + \texttt{CountLosses}(st, St_X, St_Y)$$

The Dollo score of any solution to CC-LDP equals $\min_{st \in Lab(S)} Dollo[S, st]$*, and a solution can be recovered by backtracking.*

▶ **Theorem 13.** *The dynamic programming algorithm above correctly solves CC-LDP.*

Proof.

Base case. The base case for $Dollo[A, st]$ is trivial because when $|A| = 1$ there is only one rooted, binary tree possible: a single leaf assigned the states given by the input character set $\mathcal{C}$. There are no edges and thus no losses, so $Dollo[A, st] = 0$.

Induction step. Now we consider the case where $|A| > 1$. By the induction hypothesis, we assume that we have correctly solved $Dollo[X, St_X]$ and $Dollo[Y, St_Y]$. Let $(t_X, \hat{c}_X)$ be an arbitrary solution to subproblem $Dollo[X, St_X]$, implying that (1) t_X is a rooted binary tree on leaf set X with $Clade(t_X) \subseteq \Sigma$, (2) $\hat{c}_X$ is an assignment of states to all vertices of t_X constrained by Corollary 11 and returning St_X for the root of t_X, and (3) the number of losses for $(t_X, \hat{c}_X)$ equals $Dollo[X, St_X]$, which is the minimum number of losses for any pair (tree and state assignment) that satisfies both (1) and (2). Similarly, let $(t_Y, \hat{c}_Y)$ be an arbitrary solution to subproblem $Dollo[Y, St_Y]$. Note that by (1), $X, Y \in \Sigma$ and that by (2), $St_X \in Lab(X)$ and $St_Y \in Lab(Y)$.

Let $(t, \hat{c})$ be formed by connecting $(t_X, \hat{c}_X)$ and $(t_Y, \hat{c}_Y)$ at their roots and assigning state st to the new root. The pair $(t, \hat{c})$ is a candidate solution for subproblem $Dollo[A, st]$ provided that $X|Y \in \mathcal{STB}(A)$ (so requirement 1 is satisfied) and $st = \texttt{State}(X|Y, S, \mathcal{C}) \in Lab(A)$ (so requirement 2 is satisfied). These two requirements can be summarized as $X|Y \in \mathcal{STB}(A, st)$. For this candidate solution, the number of losses equals $Dollo[X, St_X] + Dollo[Y, St_Y] + \texttt{CountLosses}(st, St_X, St_Y)$, where the last term gives the number of losses for the new edges.

Now we need to consider requirement 3. Specifically, we need to check whether a better score (i.e., lower number of losses) can be obtained from any other candidate solution. Keeping clades X and Y, other candidate solutions can be formed by all other ways of selecting $St_X \in Lab(X)$ and $St_Y \in Lab(Y)$. Moreover, this process can be repeated for the other allowed subtree bipartitions for clade A that produce state assignment st (i.e., all other ways of selecting $X|Y \in \mathcal{STB}(A, st)$). Any other possibilities will violate (1) and/or (2), and thus will not produce valid candidate solutions. Thus, our recurrence is correct.

Recurrence is solvable by dynamic programming. In our recurrence, $Dollo[A, st]$ only depends on subproblems: $Dollo[X, St_X]$ and $Dollo[Y, St_Y]$ for all $X|Y \in STB(A, st)$, $St_X \in Lab(X)$, and $St_Y \in Lab(Y)$. Since $|X|$ and $|Y|$ must be less than $|A|$, solving subproblems in order of clade cardinality will guarantee that all trivial subproblems are solved first (hitting the base cases) and that all subproblem dependencies are satisfied moving forward (because there are no dependencies between subproblems corresponding to the same clade but different state assignments at the root). Thus, we can use dynamic programming to solve this recurrence.

Putting it all together. We compute $Dollo[S, st]$ for all $st \in Lab(S)$, recording a subproblem for which the number of losses is minimized. Backtracking gives an arbitrary solution to this subproblem, which is also a solution to CC-LDP by (1), (2), and (3). ◀

▶ **Theorem 14.** *The runtime of the dynamic programming algorithm is polynomial in the number* n *of species, the number* k *of characters, and the number of clades in* Σ*.*

Proof. We need to show that the number of subproblems is polynomial and that each subproblem can be solved in polynomial time.

The dynamic programming matrix has two dimensions. The first corresponds to clade $A \in \Sigma$, and the second corresponds to an allowed state assignment $st \in Lab(A)$. The former is clearly $O(|\Sigma|)$. The latter is also $O(|\Sigma|)$ because $|Lab(A)|$ has an upper bound of $|\mathcal{STB}(A)|$, which in turn has an upper bound of $|\Sigma|$. This can be seen by noting that all allowed subtree bipartitions for A can be identified by looping over $X \in \Sigma$ and checking whether $X \subset A$ and $A \setminus X \in \Sigma$ (see Algorithm 3 for details). Thus, the number of subproblems is $O(|\Sigma|^2)$. Note that to perform the traceback, we also need to store pointers back to the two child subproblems for every subproblem.

Subproblem $Dollo[A, st]$ can be solved by considering all candidate solutions taking $X|Y \in \mathcal{STB}(A, st)$, $St_X \in Lab(X)$, and $St_Y \in Lab(Y)$. All of these sets are bounded above by $|\Sigma|$, so their product is bounded. For each candidate solution, we must execute `CountLosses` and sum three terms together; this can be done in $O(k)$ time. We also do some related bookkeeping that is polynomial time. As an example, for each subtree bipartition $X|Y \in \mathcal{STB}(A)$, we need to compute its state assignment st' with `GetStates` and then add it to the set $\mathcal{STB}(A, st')$; this can be done in $O(nk)$ time. It follows that the worst-case runtime of our algorithm is polynomial, albeit a high degree one (see Algorithm 4 for details). ◀

3.1 Extension to Camin-Sokal Parsimony

Our results for Dollo can be extended to Camin-Sokal parsimony. Specifically, we can define the *clade-constrained large Camin-Sokal parsimony* (CC-LCSP) problem, in the natural way. We can also extend our algorithm by replacing the `CountLosses` function (Algorithm 2) to count gains $0 \to 1$ instead and by redefining the `GetStates` function (Algorithm 1) based on the following result.

▶ **Theorem 15.** *Let T be a rooted binary tree, and let c be an ordered binary character (with no missing values), both on the same species set. A Camin-Sokal-labeling for (T, c) must assign state `0` to every internal vertex that is an ancestor of a leaf assigned state `0` and assign state `1` to all remaining vertices.*

The assignment of state `0` can be shown to be correct by contradiction. The assignment of state `1` to vertex v only occurs when all leaves below v are assigned state `1`; thus, labeling v with state `1` does not increase the number of gains needed to explain the data. It follows that a Camin-Sokal-labeling for (T, c) always exists and is unique.

▶ **Corollary 16.** *Let c be an ordered binary character on species set S, let T be an arbitrary rooted binary tree on S, and let $\hat{c}$ be the unique Camin-Sokal-labeling for (T, c). Then for any internal vertex v in T, $\hat{c}[v]$ can be determined just by having knowledge of $Clade(v)$; no other knowledge of T is needed. Moreover, if c is allowed to have missing values, a unique Camin-Sokal-labeling can be found in a similar fashion.*

Proof. Consider an internal vertex v in T that induces clade A. By Theorem 15, $\hat{c}[v] =$ `0` if the following case holds (otherwise $\hat{c}[v] =$ `1`)

- **Case C:** Vertex v is an ancestor of at least one leaf assigned state `0`.

Looking at clade $Clade(v) = A$, case C holds if condition 4 below is true.

- **Condition 4**: There exists a leaf $a \in A$ such that $c[a] =$ `0`.

To summarize, $\hat{c}[v] =$ `0` if condition 4 is true; otherwise $\hat{c}[v] =$ `1`. Condition 4 can be evaluated so long as we know the clade induced by v (given the character c and its complete leaf set S). This proves our first claim.

We now allow for missing values as discussed in Theorem 10, except applying condition 4 instead of conditions 1–3 to vertices in groups 1 and 3 (recall that condition 0 will be true if and only if a vertex is in group 2). Applying condition 4 to vertices in group 1 will simply

propagate state assignments from $T|_R$ back to T. For vertices in group 3, we need to show that every vertex v that maps to the same edge $e = s \mapsto t \in E(T|_R)$ will be assigned the same state: either the state assigned to s or the state assigned to t (so there may be multiple state assignments that achieve the optimal score). For each of these vertices v, the subtree bipartition $SubBip(v) = X|Y$ will have *either* all leaves in X assigned state `?` *or* all leaves in Y assigned state `?`. Thus, for any two vertices v_1 and v_2 in group 3 that map to the same edge $e = s \mapsto t$, $Clade(v_1) = Clade(v_2)$ after all leaves assigned `?` are removed. It follows that v_1 and v_2 will be assigned the same state when applying condition 4. Now we just need to check the outcomes of applying condition 4 to v_1 (call v).

- If condition 4 is true for v, it is true for both s and t. Thus, v, s, t are all assigned state `0`.
- If condition 4 is false for v, it is false for t. Thus, v and t are both assigned state `1`.

This proves our second claim. Note that our procedure still works when c has no missing values because all vertices in T will be in group 1. ◀

Although our algorithm for the large Dollo parsimony problem can be extended to Camin-Sokal parsimony, extensions to Fitch parsimony are not so straight-forward. Recall that the large Fitch parsimony problem takes unordered binary characters as input and seeks an unrooted binary tree to minimize the Fitch score. Both trees in Figure 1E–F when unrooted have an edge that induces bipartition $A, B|C, D, E$; however, the Fitch-labeling of the vertex incident to this edge and adjacent to leaves A and B depends on the remainder of the tree (or at least requires more information about the tree than a single bipartition).

4 Experimental Study

We now describe an experimental study evaluating our dynamic programming algorithm for CC-LDP against traditional methods for parsimony: heuristic search and branch-and-bound. All analysis scripts and data sets simulated for this study are available on Github (`https://github.com/molloy-lab/dollo-study`).

4.1 Character Data Sets

We evaluate methods in the context of species tree estimation under the infinite sites plus neutral Wright-Fisher (IS+nWF) model [17, 40]. Under the infinite sites model, characters evolve without homoplasy, meaning parallel mutations and reversals are prohibited. Some types of retroelement insertions, like L1 in mammals [22], are typically assumed to evolve with little homoplasy [34]. The idea is that two insertions are unlikely to occur at exactly the same position in the genome (so no parallel evolution) and that insertions are unlikely to be *precisely excised* (so no reversals) [34]. Note that the absence/presence of an insertion corresponds to ancestral/derived states so these characters are ordered.

Characters that evolve without homoplasy would result in a perfect phylogeny; however, this ignores population-level processes. For sexually reproducing organisms, insertions arising in egg or sperm cells are transmitted from parent to offspring. Thus, the mutation is polymorphic in the population when it arises and its frequency in the population changes randomly assuming neutral evolution (note that the population structure is governed by the species tree). To summarize, an insertion is gained ($0 \to 1$) exactly once but then it can be lost ($1 \to 0$) due to genetic drift. These rules are suitable for Dollo parsimony, and indeed, Dollo parsimony has been used to estimate species trees from low-homoplasy retroelement insertions in prior studies (e.g., [11, 12, 33, 26, 23, 25]). Here, we re-analyze two retroelement presence/absence data sets; we also benchmark methods on a collection of synthetic data sets simulated under the IS+nWF model.

Biological Data Sets

The *Myotis* data set from [25] has 11 taxa and 10,595 characters. Each character represents the presence/absence of a Ves SINE (short interspersed nuclear element) insertion at an orthologous position across the species' genomes. No character states are ambiguous, and all characters are parsimony-informative, specifically there are at least two `0`'s and at least two `1`'s. The original analysis of this data set included maximum parsimony using branch-and-bound with the Dollo criterion score (Figure 2 in [25]).

The *Palaeognathe* data set from [8] has 13 taxa and 4,301 parsimony-informative characters (note that over 18% of the character states in this matrix are ambiguous). Each character represents the presence/absence of a CR1 LINE (long interspersed nuclear element) insertion at an orthologous position across the species' genomes. The original analysis of this data set did not include Dollo parsimony; rather insertions were used to corroborate a species tree estimated from (estimated) gene trees with `ASTRAL` and `MP-EST` [27].

Simulated Data Sets

All synthetic data sets used in our study were simulated under an approximation to the IS+nWF model using `ms` [21]. The simulation requires a model species tree. At the high-level, a gene genealogy is simulated within the model species tree under the coalescent and then a mutation arises on a branch of the genealogy so all taxa that are descendants have the mutation (all other taxa do not). This process is repeated, producing a collection of binary characters that evolved independently within the model species tree. We only utilize the parsimony-informative characters, varying the total number of characters given to methods as input from 500 to 50,000.

The first collection of synthetic data sets are taken from Molloy *et al.* [31]. These data sets were simulated given the *Palaeognathe* species tree estimated by Cloutier *et al.* [8] using `MP-EST`. The `ms` simulation was repeated 25 times to produce 25 replicate data sets. We created a second collection of synthetic data sets by taking the species trees generated in a prior study [30]. Specifically, Mirarab *et al.* [30] simulated species trees with varying numbers of taxa under the Yule model with `SimPhy` [28], setting the species tree height to 2 million generations and the effective population size to 200,000. This process was repeated 50 times for each number of taxa. We ran the `ms` simulation, described above, for the first 25 species trees with 10, 50, 100, and 200 ingroup taxa (and one outgroup taxa). This produced 25 replicate characters matrices for each number of taxa.

4.2 Methods

We evaluated four different methods for the large Dollo parsimony problem. All approaches implemented in `PAUP*` [36] were executed using version 4a168_centos64 (downloaded from `https://paup.phylosolutions.com`).

Branch-and-bound Implementation

Branch-and-bound performs an exhaustive search of tree space in a systematic fashion using the parsimony score of an initial tree to rule out parts of tree space that do not need to be searched. We used the implementation of branch-and-bound in `PAUP*` (see Section B.4 for command) and saved all optimal trees.

SlowH and FastH Implementations

`FastH` is a "fast heuristic search" that operates in two phases. First, a starting tree is constructed via random taxon addition, meaning that taxa are put in a random order and then iteratively added to the tree (note that each taxon is attached to an edge of the current tree so that the criterion score is maximized). This process is repeated ten times and then the best scoring tree is taken as the starting tree. Second, hill-climbing is performed from the starting tree with Tree Bisection and Reconnection (TBR) edit moves. `FastH` was implemented with `PAUP*` (see Section B.1 for the command; the reconnection limit was set to eight branches by default). The 100 best-scoring trees found during the heuristic search were saved for use with our dynamic programming method.

`SlowH` is a "slow heuristic search" that operates by performing 100 independent searches. In each search, a starting tree is built via random taxon addition and then hill climbing is initiated from the starting tree using TBR edit moves. `SlowH` was implemented with `PAUP*` (see Section B.3 for the command; again the reconnection limit was set to eight branches by default). All trees with the best criterion score found were saved.

Dollo-CDP Implementation

`Dollo-CDP` is an implementation of our dynamic programming algorithm for CC-LDP. This method requires not only a character matrix but also a set of clades to use as constraints. We evaluated two different approaches for generating the constraints, both of which rely on `ASTRAL-III` [43]. The idea is to give `ASTRAL-III` a set of trees from which it will generate a set of clusters (note that clusters are subsets of taxa, like clades).

Our two approaches differ in the set of trees given to `ASTRAL-III` as input. Our first approach gives `ASTRAL-III` the 100 best-scoring trees found by `FastH`. Our second approach gives `ASTRAL-III` the input characters reformatted as unrooted trees, as proposed by [35]. The idea is that each parsimony-informative character encodes an unrooted tree with exactly one internal branch, indicating the transition between taxa in state 0 to taxa in state `1` or vice versa. After the clusters are computed with `ASTRAL-III`, `Dollo-CDP` processes them, keeping only the clusters that form clades in a tree rooted at some set O of outgroup taxa (note that outgroups are typically available when using Dollo parsimony, as they are often used when calling variants and coding them as ancestral or derived). A cluster C produced by `ASTRAL-III` is added to the constraint (clade) set if either (1) $C \subseteq O$ or (2) $C \subseteq \{S \setminus O\}$.

When the outgroup is a single taxon, both of our approaches ensure a solution to CC-LDP exists. For the first approach, all trees produced by `FastH`, denoted $\mathcal{P}$, can be rooted at O; therefore, $\Sigma = \{Clade(T) : T \in \mathcal{P}\}$. The second approach guarantees a solution by virtue of how `ASTRAL-III` handles polytomies (i.e., vertices of degree greater than three). However, this produces a large number of clades, which, in turn, makes the second approach more computationally intensive. Consequently, we only apply it to the biological data sets.

The command for running `Dollo-CDP` is given in Section B.2. Users are responsible for providing trees for building constraints if using the first approach. They are also responsible for downloading and extracting `ASTRAL-III` into the `src` directory so that `Dollo-CDP` can find it. We used `ASTRAL-III` version 5.7.8 from Github (`https://github.com/smirarab/ASTRAL`).

4.3 Evaluation Metrics

All computational experiments were run on the compute cluster for the Center for Bioinformatics and Computational Biology at the University of Maryland, College Park. This is a homogenous compute cluster, with all compute nodes having dual socket AMD EPYC 7313 16-Core processors and two terabytes of memory. All methods were given access to 64 GB of

memory, one CPU, and a maximum wallclock time of 24 hours (resources were managed by the SLURM submission system). We recorded the total wallclock time in minutes as well as the best Dollo parsimony score found. We added the runtime of `FastH` to the runtime of `Dollo-CDP`, when the former was used to construct the constraint set for the latter. Our method `Dollo-CDP` only counts losses and does not count gains, unlike `PAUP*`. To ensure scores were comparable, we recomputed the Dollo criterion score for all trees using `PAUP*` (see Section B.5 for the command).

5 Experimental Results

We now present the results of our experimental study on biological and synthetic data sets.

5.1 Results on biological data sets

For the *Myotis*, all methods completed in less than a second, except for `Dollo-CDP` when it used the characters to construct the constraint set Σ (this analysis took 54 seconds). All approaches recovered the same tree as branch-and-bound, which was the unique optimal solution (Figure 2A). For the *Palaeognathe* data set, all methods completed in less than 3 seconds (again `Dollo-CDP` using characters to construct constraints was the slowest). Branch-and-bound recovered 60 optimal trees, and the strict consensus had just three branches (Figure 2B), all of which are in the species tree estimated by Cloutier *et al.* [8] using `MP-EST`. All methods we ran recovered one of the 60 optimal trees. Thus, on the biological data sets,

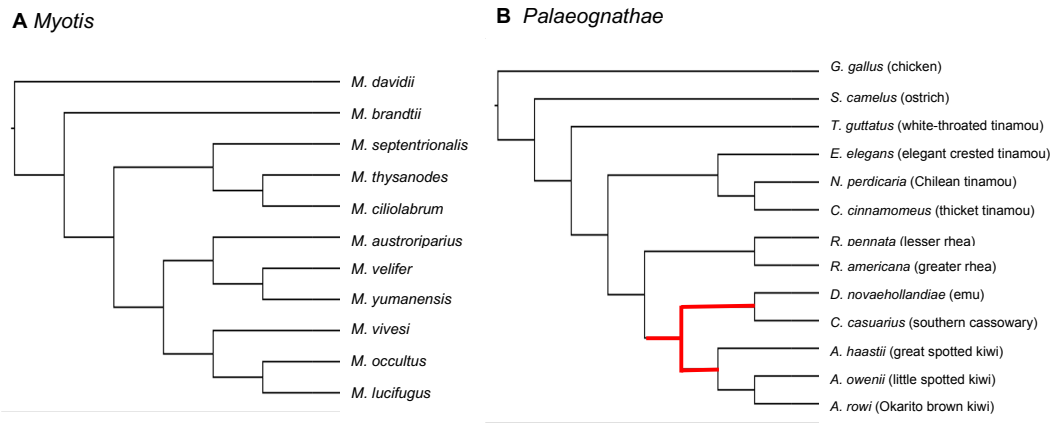


Figure 2 Subfigure A shows the tree returned by `Dollo-CDP` for the *Myotis* data set [25]. This is the same tree recovered by branch-and-bound. Subfigure B shows the tree returned by `Dollo-CDP` for the *Palaeognathae* data set [8]. A branch-and-bound analysis recovered 60 optimal trees (the `Dollo-CDP` tree is one of the 60). The three branches highlighted in red indicate the strict consensus of the 60 equally optimal trees.

5.2 Results on simulated data sets

Similar trends to the biological data sets were observed for the first collection of synthetic data sets, which were simulated from a *Palaeognathe* species tree by Molloy *et al.* [31]. For all but one replicate, all methods recovered trees with the same Dollo parsimony score as branch-and-bound (which typically recovered one or two equally optimal trees). For the

remaining replicate, FastH returned a tree with a slightly lower score than the other methods. Thus, Dollo-CDP slightly improved upon FastH in terms of Dollo parsimony score for one replicate. Similar trends were observed for the second collection of simulated data sets with 10 ingroup taxa. Like the biological data sets, the analyses of synthetic data sets suggest that as long as the number of taxa is sufficiently small, all methods will produce similar results and compare favorably to branch-and-bound.

We were unable to run branch-and-bound on data sets with 50 or more taxa (specifically the jobs were killed due to our maximum wallclock time of 24 hours). For these data sets, we first looked at whether Dollo-CDP found trees with better scores than FastH (note that Dollo-CDP cannot be worse because it is given all trees produced by FastH as constraints – and thus is guaranteed to find a tree with at least as good of a score as the best one found by FastH). For 50 ingroup taxa, Dollo-CDP typically improved upon FastH for about half of the replicates (Table 1). For 100 and 200 ingroup taxa (and at least 5000 characters), Dollo-CDP nearly always improved upon FastH. Moreover, Dollo-CDP was quite fast with an average runtime less than 3 minutes for all data sets.

Next, we compared the performance of Dollo-CDP to SlowH. For 50 and 100 ingroup taxa, Dollo-CDP performed as well as SlowH in terms of parsimony scores and was faster, although SlowH still always completed in less than 16 minutes on average. For 200 ingroup taxa, SlowH took nearly 40 minutes on average for 5000-100,000 characters and over 1.5 hours for 100,000 characters. In contrast, Dollo-CDP always completed in less than 5 minutes. On the other hand, SlowH did recover better scoring trees in about one third of the replicates. To summarize, our results suggest that for larger numbers of taxa and larger numbers of characters, Dollo-CDP improves upon FastH and is faster than SlowH.

6 Discussion and Conclusion

We have introduced the clade-constrained large Dollo parsimony problem and presented a polynomial time algorithm that solves it. Although constraints and dynamic programming (CDP) have been a powerful combination in phylogenetics, to our knowledge this is the first attempt at using CDP for character parsimony. An important distinction between prior problems and character parsimony is the assignment of states at internal vertices required to compute the parsimony score of a tree. We found that Dollo as well as Camin-Sokal criterion scores have nice properties that make CDP possible (they also might make heuristic search quite effective in practice). These nice properties for state assignments did not easily extend to Fitch parsimony, so our algorithmic approach seems less favorable in this setting.

We implemented the CDP algorithm for the Dollo criterion score in a package called Dollo-CDP. In an experimental evaluation, we found that Dollo-CDP had good performance (finding high scoring trees quickly), although all existing methods performed similarly when data sets had relatively few taxa. We found, by way of a simulation study, that Dollo-CDP can provide a benefit for larger numbers of taxa. Most notably, branch-and-bound could not scale to data sets with 50 or more taxa, so Dollo-CDP was the only method run that could provide some guarantee of optimality, albeit a more limited one. In practice, we found Dollo-CDP often found higher scores trees than FastH, even though the trees from FastH were used to form constraints for Dollo-CDP. SlowH sometimes found higher scoring trees than Dollo-CDP but was much slower for large numbers of taxa and characters. A caveat of our study is that all methods were run with one thread. All searches in SlowH are independent so it would be much faster with threading. Dollo-CDP could also take advantage of threading, using techniques from Yin *et al.* [41] and could be better optimized. We leave this to future

Table 1 This table shows a comparison of the tree produced by `Dollo-CDP` and the best trees found by the fast heuristic search (`FastH`) and the slow heuristic search (`SlowH`). We provide the number of replicates for which `Dollo-CDP` is better (b), worse (w), or the same (s) in terms of Dollo criterion score than the alternative method. We also provide the absolute difference in scores for the trees estimated by the two methods averaged across these three cases. Lastly, we provide the runtime (in minutes), averaged across all 25 replicates. The runtime of `Dollo-CDP` includes the time to run `FastH`, the output of which is used to constrain tree space.

	`Dollo-CDP` *vs.* `FastH`		`Dollo-CDP` *vs.* `SlowH`		
# of characters	# of reps b/w/s	Δ score b/w/s	# of reps b/w/s	Δ score b/w/s	Runtime (min) `Dollo-CDP`/`SlowH`
50 taxa					
500	9/0/16	1.56/NA/0	0/0/25	NA/NA/0	0.03/0.14
1000	14/0/11	2.43/NA/0	0/0/25	NA/NA/0	0.04/0.19
5000	13/0/12	6.54/NA/0	0/0/25	NA/NA/0	0.05/0.45
10000	11/0/14	16.09/NA/0	0/1/24	NA/14.00/0	0.06/0.67
50000	12/0/13	53.58/NA/0	0/0/25	NA/NA/0	0.16/1.90
100 taxa					
500	0/0/25	NA/NA/0	1/0/24	1.00/NA/0	0.46/0.46
1000	3/0/22	1.00/NA/0	1/0/24	3.00/NA/0	0.38/1.26
5000	25/0/0	6.40/NA/NA	0/0/25	NA/NA/0	0.16/3.44
10000	23/0/2	12.87/NA/0	0/1/24	NA/7.00/0	0.21/5.56
50000	21/0/4	42.81/NA/0	0/2/23	NA/56.00/0	0.56/15.58
200 taxa					
500	0/0/25	NA/NA/0	5/1/19	1.20/1.00/0	2.19/2.20
1000	0/0/25	NA/NA/0	0/0/25	NA/NA/0	3.04/2.90
5000	19/0/6	1.63/NA/0	0/8/17	NA/3.00/0	1.28/39.29
10000	21/0/4	3.10/NA/0	0/6/19	NA/6.00/0	1.72/39.37
50000	25/0/0	26.68/NA/NA	0/8/17	NA/23.62/0	2.42/93.96

work. Even if the runtime of `SlowH` improves with threading, `Dollo-CDP` could then be run using trees found by `SlowH`, in addition to the ones found by `FastH`, to form constraints. The benefit here is that running `Dollo-CDP` is fast, strictly improves upon prior searches, and gives a guarantee of optimality for the constrained solution space.

An issue that arose when analyzing the *Palaeognathae* biological data set with branch-and-bound is that there can be many optimal trees. `Dollo-CDP` returns a single binary tree, and future work should enable users to get a consensus of the optimal trees in the constrained solution space, similar to `SIESTA` [38]. Lastly, we explored methods for Dollo parsimony in the context of species tree estimation, where data are assumed to follow a population genetics model. Dollo parsimony has also been leveraged in tumor phylogenetics [3, 4, 7, 13]. It would be interesting to explore the utility of `Dollo-CDP` in this application area, especially as the number of leaves (cells instead of species) can be quite large in this setting.

References

1 Md. Shamsuzzoha Bayzid and Tandy Warnow. Gene Tree Parsimony for Incomplete Gene Trees. In Russell Schwartz and Knut Reinert, editors, *17th International Workshop on Algorithms in Bioinformatics (WABI 2017)*, volume 88 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:13, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.WABI.2017.2`.

2 Md. Shamsuzzoha Bayzid and Tandy Warnow. Gene tree parsimony for incomplete gene trees: addressing true biological loss. *Algorithms for Molecular Biology*, 13(1), 2018. doi:10.1186/s13015-017-0120-1.

3 Paola Bonizzoni, Simone Ciccolella, Gianluca Della Vedova, and Mauricio Soto. Beyond perfect phylogeny: Multisample phylogeny reconstruction via ilp. In *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology,and Health Informatics*, ACM-BCB '17, pages 1–10, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3107411.3107441.

4 Paola Bonizzoni, Simone Ciccolella, Gianluca Della Vedova, and Mauricio Soto. Does relaxing the infinite sites assumption give better tumor phylogenies? an ilp-based comparative approach. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 16(5):1410–1423, 2019. doi:10.1109/TCBB.2018.2865729.

5 Remco Bouckaert, Mareike Fischer, and Kristina Wicke. Combinatorial perspectives on dollo-k characters in phylogenetics. *Advances in Applied Mathematics*, 131:102252, 2021. doi:10.1016/j.aam.2021.102252.

6 David Bryant and Mike Steel. Constructing optimal trees from quartets. *Journal of Algorithms*, 38(1):237–259, 2001. doi:10.1006/jagm.2000.1133.

7 Simone Ciccolella, Mauricio Soto Gomez, Murray D. Patterson, Gianluca Della Vedova, Iman Hajirasouliha, and Paola Bonizzoni. gpps: an ILP-based approach for inferring cancer progression with mutation losses from single cell data. *BMC Bioinformatics*, 21(Suppl 1):313, 2020. doi:10.1186/s12859-020-03736-7.

8 Alison Cloutier, Timothy B. Sackton, Phil Grayson, Michele Clamp, Allan J. Baker, and Scott V. Edwards. Whole-genome analyses resolve the phylogeny of flightless birds (Palaeognathae) in the presence of an empirical anomaly zone. *Systematic Biology*, 68(6):937–955, 2019. doi:10.1093/sysbio/syz019.

9 William H.E. Day, David S. Johnson, and David Sankoff. The computational complexity of inferring rooted phylogenies by parsimony. *Mathematical Biosciences*, 81(1):33–42, 1986. doi:10.1016/0025-5564(86)90161-6.

10 Payam Dibaeinia, Shayan Tabe-Bordbar, and Tandy Warnow. FASTRAL: improving scalability of phylogenomic analysis. *Bioinformatics*, 37(16):2317–2324, 2021. doi:10.1093/bioinformatics/btab093.

11 Liliya Doronina, Gennady Churakov, Andrej Kuritzin, Jingjing Shi, Robert Baertsch, Hiram Clawson, and Jürgen Schmitz. Speciation network in laurasiatheria: retrophylogenomic signals. *Genome Research*, 27:997–1003, 2017. doi:10.1101/gr.210948.116.

12 Liliya Doronina, Graham M. Hughes, Diana Moreno-Santillan, Colleen Lawless, Tadhg Lonergan, Louise Ryan, David Jebb, Bogdan M. Kirilenko, Jennifer M. Korstian, Liliana M. Dávalos, Sonja C. Vernes, Eugene W. Myers, Emma C. Teeling, Michael Hiller, Lars S. Jermiin, Jürgen Schmitz, Mark S. Springer, and David A. Ray. Contradictory phylogenetic signals in the laurasiatheria anomaly zone. *Genes*, 13(5), 2022. doi:10.3390/genes13050766.

13 Mohammed El-Kebir. SPhyR: tumor phylogeny estimation from single-cell sequencing data under loss and error. *Bioinformatics*, 34(17):i671–i679, 2018. doi:10.1093/bioinformatics/bty589.

14 Joseph Felsenstein. Parsimony in systematics: Biological and statistical issues. *Annual Review of Ecology and Systematics*, 14:313–333, 1983. URL: http://www.jstor.org/stable/2096976.

15 Joseph Felsenstein. *Inferring Phylogenies*. Sinauer Associates, Inc., Sunderland, Massachusetts, 2 edition, 2004. doi:10.1007/BF01734359.

16 Joseph Felsenstein. Phylip (phylogeny inference package), 2005. Accessed on XX. URL: https://evolution.genetics.washington.edu/phylip.html.

17 Ronald A. Fisher. On the dominance ratio. *Proceedings of the Royal Society of Edinburgh*, 42:321–341, 1923. doi:10.1017/S0370164600023993.

18 Walter M. Fitch. Toward Defining the Course of Evolution: Minimum Change for a Specific Tree Topology. *Systematic Biology*, 20(4):406–416, 1971. doi:10.1093/sysbio/20.4.406.

19 Ronald L. Graham and Les R. Foulds. Unlikelihood that minimal phylogenies for a realistic biological study can be constructed in reasonable computational time. *Mathematical Biosciences*, 60(2):133–142, 1982. doi:10.1016/0025-5564(82)90125-0.

20 Michael T. Hallett and Jens Lagergren. New algorithms for the duplication-loss model. In *Proceedings of the Fourth Annual International Conference on Computational Molecular Biology*, RECOMB '00, pages 138–146, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/332306.332359.

21 Richard R. Hudson. Generating samples under a Wright–Fisher neutral model of genetic variation. *Bioinformatics*, 18(2):337–338, February 2002. doi:10.1093/bioinformatics/18.2.337.

22 Roy N. Platt II, Michael W. Vandewege, and David A. Ray. Mammalian transposable elements and their impacts on genome evolution. *Chromosome Research*, 26:25–43, 2018. doi:10.1007/s10577-017-9570-z.

23 Roy N. Platt II, Yuhua Zhang, David J. Witherspoon, Jinchuan Xing, Alexander Suh, Megan S. Keith, Lynn B. Jorde, Richard D. Stevens, and David A. Ray. Targeted capture of phylogenetically informative ves sine insertions in genus myotis. *Genome Biology and Evolution*, 7(6):1664–1675, 2015. doi:10.1093/gbe/evv099.

24 Mazharul Islam, Kowshika Sarker, Trisha Das, Rezwana Reaz, and Md. Shamsuzzoha Bayzid. STELAR: a statistically consistent coalescent-based species tree estimation method by maximizing triplet consistency. *BMC Genomics*, 21(1):136, 2020. doi:10.1186/s12864-020-6519-y.

25 Jennifer M. Korstian, Nicole S. Paulat, Roy N. Platt II, Richard D. Stevens, and David A. Ray. Sine-based phylogenomics reveal extensive introgression and incomplete lineage sorting in myotis. *Genes*, 13(3):399, 2022. doi:10.3390/genes13030399.

26 Fritjof Lammers, Moritz Blumer, Cornelia Rücklé, and Maria A. Nilsson. Retrophylogenomics in rorquals indicate large ancestral population sizes and a rapid radiation. *Mobile DNA*, 10:5, 2019. doi:10.1186/s13100-018-0143-2.

27 Liang Liu, Lili Yu, and Scott V. Edwards. A maximum pseudo-likelihood approach for estimating species trees under the coalescent model. *BMC Evolutionary Biology*, 10:302, 2010. doi:10.1186/1471-2148-10-302.

28 Diego Mallo, Leonardo De Oliveira Martins, and David Posada. SimPhy : Phylogenomic simulation of gene, locus, and species trees. *Systematic Biology*, 65(2):334–344, November 2015. doi:10.1093/sysbio/syv082.

29 Siavash Mirarab, Rezwana Reaz, Md. Shamsuzzoha Bayzid, Théo Zimmermann, Michelle S. Swenson, and Tandy Warnow. ASTRAL: genome-scale coalescent-based species tree estimation. *Bioinformatics*, 30(17):i541–i548, 2014. doi:10.1093/bioinformatics/btu462.

30 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, 2015. doi:10.1093/bioinformatics/btv234.

31 Erin K. Molloy, John Gatesy, and Mark S Springer. Theoretical and practical considerations when using retroelement insertions to estimate species trees in the anomaly zone. *Systematic Biology*, 71(3):721–740, 2021. doi:10.1093/sysbio/syab086.

32 Erin K. Molloy and Tandy Warnow. FastMulRFS: fast and accurate species tree estimation under generic gene duplication and loss models. *Bioinformatics*, 36(Supplement_1):i57–i65, July 2020. doi:10.1093/bioinformatics/btaa444.

33 Abdel-Halim Salem, David A. Ray amd Jinchuan Xing, Pauline A. Callinan, Jeremy S. Myers, Dale J. Hedges, Randall K. Garber, David J. Witherspoon, Lynn B. Jorde, and Mark A. Batzer. Alu elements and hominid phylogenetics. *Proceedings of the National Academy of Sciences of the United States of America*, 100(22):12787–12791, 2003. doi:10.1073/pnas.2133766100.

34 Andrew M. Shedlock, Michael C. Milinkovitch, and Norihiro Okada. SINE evolution, missing data, and the origin of whales. *Systematic Biology*, 49:808–817, 2000.

35 Mark S Springer, Erin K Molloy, Daniel B Sloan, Mark P Simmons, and John Gatesy. ILS-aware analysis of low-homoplasy retroelement insertions: Inference of species trees and introgression using quartets. *Journal of Heredity*, 111(2):147–168, 2019. doi:10.1093/jhered/esz076.

36 David L. Swofford. *PAUP*. Phylogenetic Analysis Using Parsimony (*and Other Methods). Version 4*. Sinauer Associates, Sunderland, Massachusetts, 2003.

37 Pranjal Vachaspati and Tandy Warnow. FastRFS: fast and accurate robinson-foulds supertrees using constrained exact optimization. *Bioinformatics*, 33(5):631–639, September 2016. doi:10.1093/bioinformatics/btw600.

38 Pranjal Vachaspati and Tandy Warnow. SIESTA: enhancing searches for optimal supertrees and species trees. *BMC Genomics*, 19(Suppl 5):252, 2018. doi:10.1186/s12864-018-4621-1.

39 Tandy Warnow. *Computational Phylogenetics: An Introduction to Designing Methods for Phylogeny Estimation*. Cambridge University Press, Cambridge, United Kingdom, 2017.

40 Sewall Wright. Evolution in mendelian populations. *Genetics*, 16(2):97–159, 1931. doi:10.1093/genetics/16.2.97.

41 John Yin, Chao Zhang, and Siavash Mirarab. ASTRAL-MP: scaling ASTRAL to very large datasets using randomization and parallelization. *Bioinformatics*, 35(20):3961–3969, 2019. doi:10.1093/bioinformatics/btz211.

42 Yun Yu, Tandy Warnow, and Luay Nakhleh. Algorithms for MDC-based multi-locus phylogeny inference: Beyond rooted binary gene trees on single alleles. *Journal of Computational Biology*, 18(11):1543–1559, 2011. doi:10.1089/cmb.2011.0174.

43 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, 2018. doi:10.1186/s12859-018-2129-y.

44 Chao Zhang, Celine Scornavacca, Erin K. Molloy, and Siavash Mirarab. ASTRAL-Pro: Quartet-based species-tree inference despite paralogy. *Molecular Biology and Evolution*, 37(11):3292–3307, 2020. doi:10.1093/molbev/msaa139.

A Algorithms

Algorithm 1 `GetState`.

Input: Subtree bipartition $X|Y$, the set S of n species, and an $n \times k$ character matrix $\mathbf{C}$, with $\mathbf{C}[i,j]$ indicating the state assigned to leaf i for character j

Output: State assignment for subtree bipartition $X|Y$ for each of the k characters in $\mathbf{C}$

function GetState($X|Y, S, \mathbf{C}$)
  $states \leftarrow$ an array of length k containing all 0's ; $A \leftarrow X \cup Y$; $Z \leftarrow S \setminus A$
  for $i \in \{0, 1, 2, \ldots, k-1\}$ **do**
    if for all $x \in A$, $\mathbf{C}[x,i] =$? **then** $states[i] \leftarrow$?
    else
      $flag \leftarrow 0$
      if $\exists x \in X$ s.t. $\mathbf{C}[x,i] = 1$ **then** $flag \leftarrow flag + 1$
      if $\exists x \in Y$ s.t. $\mathbf{C}[x,i] = 1$ **then** $flag \leftarrow flag + 1$
      if $\exists x \in Z$ s.t. $\mathbf{C}[x,i] = 1$ **then** $flag \leftarrow flag + 1$
      if $flag \geq 2$ **then** $states[i] \leftarrow 1$
  return $states$

Algorithm 2 `CountLosses`.

Input: Three k-vectors of state assignments: $statesU$, $statesV$, $statesW$

Output: Number of losses assuming that $statesU$ are associated with a vertex u and $statesV$ and $statesW$ are associated with children of u

```
1: function CountLosses(statesU, statesV, statesW)
2:     nlosses ← 0
3:     for i ∈ {0, 1, 2, ..., k − 1} do
4:         if statesU[i] = 1 and statesV[i] = 0 then nlosses ← nlosses + 1
5:         if statesU[i] = 1 and statesW[i] = 0 then nlosses ← nlosses + 1
6:     return nlosses
```

Algorithm 3 Construct subtree bipartitions from clades.

Input: Set Σ of clades

Output: Subtree bipartitions allowed from Σ stored as a dictionary, where $SubBip[A]$ is a list of the allowed subtree bipartitions for clade A (note that only one side of the subtree bipartition is stored)

```
1: function ConstructSubBipsFromClades(Σ)
2:     Sort Σ by cardinality from least to greatest
3:     for i ∈ {0, 1, ..., |Σ| − 1} do
4:         A ← Σ[i]; SubBip[A] ← [ ]
5:         for j ∈ {0, 1, ..., i − 1} do
6:             X ← Σ[j]
7:             if X ⊂ A then SubBip[A].append(X) (already have ptr to SubBip[A])
8:     return SubBip
```

Algorithm 4 Dynamic Programming for CC-LDP.

Input: Set Σ of clades (sorted by cardinality from least to greatest), a dictionary $SubBip$ of allowed subtree bipartitions previously computed from Σ, an $n \times k$ character matrix $\mathbf{C}$, with each character on species set S

Output: Fills in the dynamic programming matrix $Dollo$ and the traceback matrix $TraceBack$

$Lab \leftarrow dict()$
$SubBipByLab \leftarrow dict()$
for $A \in \Sigma$ **do**
  if $|A| = 1$ **then**
    Do base case for leaves
    $st \leftarrow \mathbf{C}[A, :]$
    $Dollo[A, st] \leftarrow 0$
    $Lab[A] \leftarrow st$
    $SubBipByLab[A][st] \leftarrow \varnothing$
    $Traceback[A][st] \leftarrow NULL$
  else
    Find state assignments for clade A and their associated subtree bipartitions
    $Lab[A] \leftarrow \varnothing$
    $SubBipByLab[A] \leftarrow dict()$
    for $X \in SubBip[A]$ **do**
      $Y \leftarrow A \setminus X$
      $st \leftarrow$ `GetState`$(X|Y, S, \mathbf{C})$
      Add st to set $Lab[A]$
      Add X to set $SubBipByLab[A][st]$ (initialize if it does not exist)
    For each unique state assignment fill in DP matrix
    $bestDollo \leftarrow \infty$
    $bestX, bestY, bestXState, bestYState \leftarrow NULL$
    for $st \in Lab[A]$ **do**
      for $X \in SubBipByLab[A][st]$ **do**
        $Y \leftarrow A \setminus X$
        for $St_X \in Lab[X]$ **do**
          for $St_Y \in Lab[Y]$ **do**
            $cX \leftarrow Dollo[X, St_X]$
            $cY \leftarrow Dollo[Y, St_Y]$
            $cA \leftarrow$ `CountLosses`(st, St_X, St_Y)
            $score \leftarrow cX + cY + cA$
            if $bestDollo > score$ **then**
              $bestDollo \leftarrow score$
              $bestX \leftarrow X$
              $bestXState \leftarrow St_X$
              $bestY \leftarrow Y$
              $bestYState \leftarrow St_Y$
    $Dollo[A, st] \leftarrow bestDollo$
    $TraceBack(A, st) \leftarrow (bestX, bestXState, bestY, bestYState)$ (two children)

B Software Commands

In all PAUP* commands for heuristic search, we specified seeds so that the results would be reproducible.

B.1 PAUP* command for running the fast heuristic search (FastH)

```
#NEXUS
BEGIN PAUP;
set autoclose=yes warntree=no warnreset=no;
execute <character matrix nexus file>;
outgroup <outgroup name>;
ctype dollo:1-<total number of characters>;
hsearch start=stepwise addSeq=random swap=None nreps=10 rseed=55555;
hsearch start=1 swap=TBR nbest=100 rseed=12345;
rootTrees;
savetrees File=<output file> root=yes trees=all format=newick;
END;
```

B.2 Dollo-CDP command

```
./dollo-cdp \
  -i <character matrix nexus file> \
  -g <outgroup> \
  -t <best 100 trees found from fast heuristic search> \
  -o <output file name>
```

B.3 PAUP* command for running the slow heuristic search (SlowH)

```
#NEXUS
BEGIN PAUP;
set autoclose=yes warntree=no warnreset=no;
execute <character matrix nexus file>;
outgroup <outgroup name>;
ctype dollo:1-<total number of characters>;
hsearch start=stepwise addSeq=random swap=TBR nreps=100 rseed=12345;
rootTrees;
savetrees File=<output file> root=yes trees=all format=newick;
END;
```

B.4 PAUP* command for running branch-and-bound

```
#NEXUS
BEGIN PAUP;
set autoclose=yes warntree=no warnreset=no;
execute <character matrix nexus file>;
outgroup <outgroup name>;
ctype dollo:1-<total number of characters>;
bandb;
rootTrees;
savetrees File=<output file> root=yes trees=all format=newick;
END;
```

B.5 PAUP* command for computing the Dollo criterion score

```
#NEXUS
BEGIN PAUP;
set autoclose=yes warntree=no warnreset=no;
execute <character matrix nexus file>;
execute <tree nexus file file>;
set criterion=parsimony;
ctype dollo:1-<total number of characters>;
pscores / single=var;
END;
```

Simultaneous Reconstruction of Duplication Episodes and Gene-Species Mappings

Paweł Górecki
Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland

Natalia Rutecka
Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland

Agnieszka Mykowiecka
Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland

Jarosław Paszek
Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland

Abstract

We present a novel problem, called MetaEC, which aims to infer gene-species assignments in a collection of gene trees with missing labels by minimizing the size of duplication episode clustering (EC). This problem is particularly relevant in metagenomics, where incomplete data often poses a challenge in the accurate reconstruction of gene histories. To solve MetaEC, we propose a polynomial time dynamic programming (DP) formulation that verifies the existence of a set of duplication episodes from a predefined set of episode candidates. We then demonstrate how to use DP to design an algorithm that solves MetaEC. Although the algorithm is exponential in the worst case, we introduce a heuristic modification of the algorithm that provides a solution with the knowledge that it is exact. To evaluate our method, we perform two computational experiments on simulated and empirical data containing whole genome duplication events, showing that our algorithm is able to accurately infer the corresponding events.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Applied computing → Computational genomics

Keywords and phrases Genomic Duplication, Gene-Species Mapping, Duplication Episode, Gene Tree, Species Tree

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.6

Supplementary Material *Software (Source Code and Datasets)*: `https://bitbucket.org/pgor17/metaec/`; archived at `swh:1:dir:c2cc43b86e0954bda7806f5d7007188ca33938b5`

Funding *Paweł Górecki*: National Science Centre grant nr 2017/27/B/ST6/02720.
Natalia Rutecka: National Science Centre grant nr 2017/27/B/ST6/02720.
Agnieszka Mykowiecka: National Science Centre grant nr 2017/27/B/ST6/02720.
Jarosław Paszek: National Science Centre grant nr 2020/39/D/ST6/03321.

1 Introduction

In the field of computational biology, the use of gene families and the reconciliation model has become increasingly popular for studying the evolution of diverse organisms. These tools have facilitated the development of new algorithms and computational methods capable of handling large and complex datasets and exploring various types of evolutionary events. These events range from simple macroevolutionary processes such as gene duplications, gene losses, and horizontal gene transfers to more complex ones such as genomic duplications, speciations, and hybridizations. The reconciliation model has enabled researchers to reconstruct evolutionary histories by reconciling gene trees with species trees, and identifying the evolutionary events that have led to the observed patterns of gene evolution. In the context of metagenomics,

the reconciliation model has also been used to detect missing gene-species assignments using polynomial time algorithms. These developments have led to a better understanding of the evolutionary processes.

A classical reconciliation model [10, 26] defines a mapping from every node from a gene tree into a node in the species tree and determines if such a node is related to a speciation or can be classified as a single gene duplication [19]. In result, an embedding of the set of gene trees into a species tree can be interpreted as a joint evolutionary scenario [11]. The classical least common ancestor (LCA) mapping minimizes the number of single gene duplications and losses for one gene tree and the species tree [11].

The whole genome duplication (WGD) phenomenon incorporates additional copies of a complete genome into the original genetic material, thus creating an opportunity to introduce novel evolutionary traits [16, 25]. From a macro perspective, this phenomenon played a crucial role in the divergence and formation of species and shaped the evolution of almost all major lineages of life. In particular, many WGDs were uncovered in the evolutionary histories of plants, especially crops. WDGs potentially enabled the successful domestication of plants [36] and are important in the fight against famine [40]. Many traces and evidence of whole genome duplications can be found in the genomes of yeast and other fungal species [16, 39]. From the perspective of single cell evolution studies, WGDs are prevalent in cancer progression [18] and can lead us to the prognosis of advanced cancer stages [3] or the creation of strategies for targeted therapy [33].

Guigó et al. [13] proposed the first approach for detecting multiple gene duplication episodes from a collection of rooted gene trees. They designed a heuristic that aggregates single gene duplication events into a large gene duplication, given a collection of rooted gene trees and a rooted species tree. This approach was formalized and improved by Page and Cotton [27], who defined the problem of *episode clustering* (EC) as the task of identifying the minimal number of locations in the species tree where all duplications from the input gene trees can be placed. Fellows [8] applied this model in the context of the supertree problem. Polynomial-time solutions for two types of multiple gene duplication problems episode clustering and a more general variant of clustering called *minimum episodes* (ME) were proposed in [1, 4]. Luo et al. [17] proposed linear time and space algorithms, partially based on [22], for these problems. [29] introduced a unified approach by proposing a concept of interval models with a linear time and space solution to a broad class of clustering problems including EC and ME. Alternative approaches include generalization to unrooted gene trees; however, such approaches are often computationally complex [28, 30]. Other approaches include variants of clustering rules that depend on the maximal number of duplication episodes placed in one path [15, 32]. A comprehensive analysis of various models is available in [29]. Furthermore, [31] proposes an integer linear programming formulation that simplifies the process of testing these models. Relevant computational complexity results on the ME problem are presented in [6].

Metagenomic studies provide valuable information for analyzing entire communities of organisms and revealing a complete picture of their functional and adaptive capacities crucial for ecology [35] or human health [38]. Genetic material isolated in such studies can be used to detect whole genome duplication events.

One of the steps in metagenomic analysis is called binning. The aim of this procedure is to assign sequenced DNA fragments to the appropriate taxonomic groups. The assignment of certain genes to species can be ambiguous due to the limitations of annotation methods. A precise and comprehensive gene tree topology is essential for the accurate identification of potential duplication sites. The absence or misplacement of duplications in gene trees can, in turn, result in incorrect outcomes of methods aimed at determining whole genome duplication events.

To tackle the challenge of missing gene-species assignments in evolutionary studies, previous research has introduced methods based on the reconciliation score using gene duplication and loss events [2, 42]. In a related work, Mykowiecka et al. [24] extended this model by including horizontal gene transfer to better analyze bacterial evolution and proposed polynomial time algorithms for these models. These approaches utilize tree reconciliation according to the classical scheme, in which the gene tree includes symbols representing sequences with unclear species assignment in addition to the known gene labels. The objective is to assign the unknown gene labels to their corresponding species by resolving the missing labels in a gene tree while minimizing the total reconciliation score, which is typically a weighted sum of evolutionary events such as gene duplication, gene loss, and horizontal gene transfer.

Here, we present a novel problem, called MetaEC, which aims to infer gene-species assignments in a collection of gene trees with missing labels by minimizing the size of episode clustering. This problem is particularly relevant in metagenomics, where incomplete data often poses a challenge in the accurate reconstruction of gene histories. To solve MetaEC, we propose a dynamic programming (DP) algorithm that verifies the existence of a set of duplication episodes from a predefined set of episode candidates. We then demonstrate how to use DP to design an algorithm that solves MetaEC. Although the algorithm is exponential in the worst case, we introduce a heuristic modification of the algorithm that provides a solution with the knowledge that it is exact. To evaluate our method, we perform two computational experiments on simulated and empirical data containing WGD events, showing that our algorithm is able to accurately infer the corresponding events.

2 Basic Definitions

2.1 Gene trees, species trees, and the duplication cost

We begin by recalling some basic definitions from graph theory. All trees in this article are rooted and binary, therefore we refer to them as *trees*. For a tree $T = (V(T), E(T))$, by $\mathsf{root}(T)$ we denote the root, and by $L(T)$ we denote the set of all leaves. Every non-leaf node will be called *internal*. A *species tree* is a tree whose leaves are called *species*. A *gene tree* over a species tree S is a tree with leaves labeled by the species from S. The set of all species present in a species tree or a gene tree T is denoted by $\mathcal{L}(T)$. Note that for a species tree S, $L(S) = \mathcal{L}(S)$. Also, for a gene tree G over S, $\mathcal{L}(G) \subseteq L(S)$.

For nodes a and b, $a \preceq b$ means that a and b are on the same path from the root, with b being closer to the root than a. We write $a \prec b$ if $a \preceq b$ and $a \neq b$.

For a gene tree G over a species tree S, *the least common ancestor (lca) mapping* between G and S is a function $\mathsf{M}_G \colon V(G) \to V(S)$ defined as follows. If v is a leaf in G then $\mathsf{M}_G(v)$ is the label of v. When v is an internal node in T having two children a and b, then $\mathsf{M}_G(v)$ is the least common ancestor of $\mathsf{M}_G(a)$ and $\mathsf{M}_G(b)$ in S. An internal node $g \in V(G)$ is called a *duplication* if $\mathsf{M}_G(g) = \mathsf{M}_G(a)$ for a child a of g. *The duplication cost*, denoted by $\mathsf{D}(G, S)$, is the total number of duplications in G. Each non-duplication internal node of G we call a *speciation*. In the latter part of the article, a duplication g is called an s-duplication if $\mathsf{M}_G(g) = s$. Similarly, we use the notation for an s-speciation and an s-leaf. An example of tree reconciliation and the lca-mapping is depicted in the leftmost part of Figure 1.

2.2 Episode Clustering Problems

Below we present a model of duplication episodes proposed in [29]. In short, this model admits all evolutionary scenarios using duplication and loss events with a minimal number of gene duplications.

Formally, the model of gene duplication episodes allows for relocating a gene duplication from its lca-mapping node to one of its ancestors under some additional constraints required to preserve the biological soundness of the scenario. For a gene tree G over S, a mapping $\mathsf{F}_G\colon V(G) \to V(S)$ is called *valid* if the following conditions are satisfied:

- $\mathsf{F}_G(a) \preceq \mathsf{F}_G(b)$ if $a \preceq b$ (time consistency),
- $\mathsf{F}_G(a) = \mathsf{M}_G(a)$ for any speciation node a (fixed speciations),
- $\mathsf{F}_G(a) \succeq \mathsf{M}_G(a)$ for any duplication node a (duplication can be raised),
- $\mathsf{F}_G(a) \prec \mathsf{M}_G(b)$ for any speciation node b such that $a \prec b$ (fixed number of gene duplications).

Note that the model of valid mappings described above is more comprehensive than the model presented in [1].

Figure 1 provides an example of valid mappings that uniquely define an evolutionary scenario that can be represented as a tree with an additional decoration of nodes. For more information on the formal modeling of evolutionary scenarios, refer to [11].

We denote by $\mathsf{Dup}_T \subset V(T)$, the set of all duplication nodes in T. Let $G_1, G_2, \ldots, G_n$ be a collection of rooted gene trees over a species tree S. Assume that, for every $i \in \{1, 2, \ldots, n\}$, F_i is a valid mapping between G_i and S. Every element $s \in \bigcup_i \mathsf{F}_i[\mathsf{Dup}_{G_i}]$ denotes the location of multiple gene duplication events in S. We will refer to such locations as *duplication episodes* or simply *episodes*. Later on, we may also use the term episode to refer to the set of duplications that are mapped into it.

▶ **Problem 1** (Episode Clustering, EC). *Given a collection of rooted gene trees $G_1, G_2, \ldots, G_n$ over a species tree S. Compute the minimal number of duplication episodes, denoted by $\mathsf{EC}(G_1, G_2, \ldots, G_n, S)$, in the set of all valid mappings $\mathsf{F}_1, \mathsf{F}_2, \ldots, \mathsf{F}_n$ such that $\mathsf{F}_i\colon V(G_i) \to V(S)$.*

This problem can be solved in linear time and space [17].

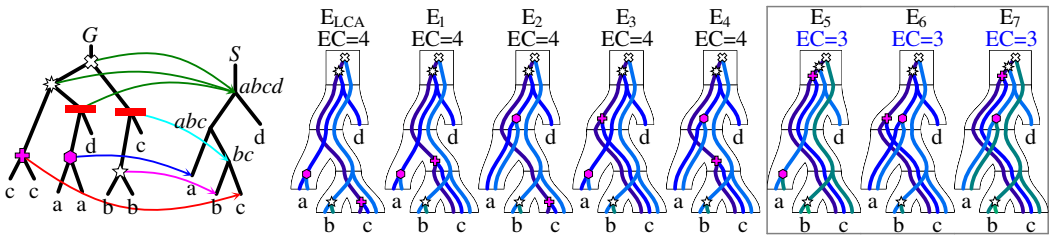


Figure 1 From the left side: a gene tree G and a species tree S with the lca-mapping M shown using arrows from the internal nodes of G to the nodes of S. There are 5 gene duplications, 2 speciation nodes in G (red bars), and 8 valid mappings depicted as embeddings of G into S [11], where the blue lines in these embeddings correspond to the edges of G. E_{LCA} is induced by the lca-mapping. Here, $\mathsf{EC}(G, S) = 3$ as indicated in the three rightmost scenarios with episode sets $\{a, b, abcd\}$ for E_5 and $\{a, abc, abcd\}$ for E_6 and E_7. The example trees are partially adopted from [32].

2.3 Gene-species mappings

We present the main problem for joint reconstruction of gene-species mappings and minimizing the set of episodes.

A *partial gene tree* is a rooted binary tree where each leaf is labeled by a species or has no label. Let G be a partial gene tree over S. By $\Lambda_G\colon L(G) \to L(S)$ we denote the partial *leaf labelling function* such that $\Lambda_G(g)$ is the label (species) of the leaf g in G if defined. Note that any gene tree is a partial gene tree with the leaf labeling being a total function. If a leaf in G has no label we write $\Lambda_G(g) = \perp$. We say that a gene tree G^* over S *extends* a partial gene tree G over S if G and G^* are isomorphic as graphs (i.e., $V(G) = V(G^*)$ and $E(G) = E(G^*)$), and, Λ_{G^*} is a total function that extends Λ_G.

2.4 Inferring labelings by minimizing episodes

Now, we present the problem of the simultaneous reconstruction of leaf labelings and duplication episodes from collections of partial gene trees.

▶ **Problem 2** (MetaEC). *Given a collection of partial gene trees $G_1, G_2, \ldots, G_k$ over a species tree S. Compute the minimum $\mathsf{EC}(G_1^*, G_2^*, \ldots, G_k^*, S)$, denoted $\mathsf{EC}(G_1, G_2, \ldots, G_k, S)$, in the set of all collections of gene trees $G_1^*, G_2^*, \ldots, G_k^*$ such that G_i^* extends G_i, for each i.*

For example, if $(a, (\perp, (\perp, \perp)))$ is a single gene tree with three undefined labels and $(a, (b, c))$ is a species tree, then the problem is to replace all occurrences of $\perp$ by a, b or c such that the total number of duplication episodes is minimized. In this case, the optimal cost is 1, since at least one duplication is needed when the gene tree has four leaves and there are only three species.

3 Methods

We first solve a simpler problem in which we assume that the set of duplication episodes is constrained to a given set of species tree nodes. Then, we show how to solve MetaEC for a single gene tree. Section 3.3 presents the general solution.

3.1 Episode Feasibility Problem

We start with a related constrained problem. Given a partial gene tree, we are interested in the question, of whether there is an extension of the partial gene tree such that the set of corresponding duplication episodes is contained in a given fixed set of episode candidates.

▶ **Problem 3** (Episode Feasibility). *Given a partial gene tree G over a species tree S and $X \subseteq V(S)$. Does there exist a gene tree G^* and a valid mapping F_{G^*} such that G^* extends G and $\mathsf{F}_{G^*}(\mathsf{Dup}_{G^*}) \subseteq X$?*

If a partial gene tree G satisfies the above property, we call G *X-feasible* with respect to a species tree S. If the context is clear, we omit the reference to S.

The solution to Episode Feasibility is a dynamic programming (DP) formulation expressed using Łukasiewicz's Three-Valued Logic Ł$_3$ [43] with three constants True, False, and Unknown (representing uncertainty) and ordered linearly: False $<$ Unknown $<$ True. The logic has binary operators $\vee$ (disjunction, max), $\wedge$ (conjunction, min), and two unary operators L (certainty) and M (possibility). See the interpretation in Figure 2.

$\vee$	$\mathbb{F}$	$\mathbb{U}$	$\mathbb{T}$
$\mathbb{F}$	$\mathbb{F}$	$\mathbb{U}$	$\mathbb{T}$
$\mathbb{U}$	$\mathbb{U}$	$\mathbb{U}$	$\mathbb{T}$
$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$

$\wedge$	$\mathbb{F}$	$\mathbb{U}$	$\mathbb{T}$
$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$
$\mathbb{U}$	$\mathbb{F}$	$\mathbb{U}$	$\mathbb{U}$
$\mathbb{T}$	$\mathbb{F}$	$\mathbb{U}$	$\mathbb{T}$

L	
$\mathbb{F}$	$\mathbb{F}$
$\mathbb{U}$	$\mathbb{F}$
$\mathbb{T}$	$\mathbb{T}$

M	
$\mathbb{F}$	$\mathbb{F}$
$\mathbb{U}$	$\mathbb{T}$
$\mathbb{T}$	$\mathbb{T}$

Figure 2 Boolean operations in Three-Valued-Logic used in DP. Here $\mathbb{F}$= False, $\mathbb{U}$= Unknown, and $\mathbb{T}$= True.

For a node g of a gene tree G, by $G|g$ we denote the subtree of G rooted at g. For any non-leaf node t in a tree, by t' and t'', we denote the children of t. To simplify the notation, we assume that the set $X \subseteq V(S)$ is fixed. Then, we have the following dynamic programming formulas that solve Episode Feasibility. Let $g \in V(G)$ and $s \in V(S)$.

$$\delta(g,s) = \begin{cases} \delta^*(g,s) & g \text{ is internal and } s \in X, \quad (1)\\ \delta^*(g,s) \wedge \mathsf{Unknown} & g \text{ is internal and } s \notin X, \quad (2)\\ \mathsf{False} & \text{otherwise}, \quad (3) \end{cases}$$

$$\delta^{\downarrow}(g,s) = \begin{cases} \epsilon(g,s) & s \text{ is a leaf}, \quad (4)\\ \epsilon(g,s) \vee \mathsf{M}\,\delta^{\downarrow}(g,s') \vee \mathsf{M}\,\delta^{\downarrow}(g,s'') & s \text{ internal, and } s \in X. \quad (5)\\ \epsilon(g,s) \vee \delta^{\downarrow}(g,s') \vee \delta^{\downarrow}(g,s'') & \text{otherwise}, \quad (6) \end{cases}$$

$$\sigma(g,s) = \begin{cases} \mathsf{L}\left(\delta^{\downarrow}(g',s') \wedge \delta^{\downarrow}(g'',s'') \vee \delta^{\downarrow}(g',s'') \wedge \delta^{\downarrow}(g'',s')\right) & g \text{ and } s \text{ are internal}, \quad (7)\\ \mathsf{True} & g \in L(G),\ \Lambda_G(g) \in \{s, \perp\}, \quad (8)\\ \mathsf{False} & \text{otherwise}, \quad (9) \end{cases}$$

where

$$\epsilon(g,s) = \sigma(g,s) \vee \delta(g,s), \quad (10)$$

$$\delta^*(g,s) = \epsilon(g',s) \wedge \delta^{\downarrow}(g'',s) \vee \epsilon(g'',s) \wedge \delta^{\downarrow}(g',s). \quad (11)$$

In the next Lemma, we express properties satisfied by the above formulas.

For a partial gene tree G over S, and nodes $g \in V(G)$ and $s \in V(S)$, we say that a valid mapping $\mathsf{F}_T\colon V(G|g) \to V(S|s)$ is *feasible* for (g,s,X), if and only if, T extends $G|g$ and $\mathsf{F}_T(\mathsf{Dup}_T) \subseteq X$. Feasible mappings represent episode scenarios that correspond to partial solutions to the instance of Episode Feasibility that forces duplications from $G|g$ to be present in episodes from $X \cap V(S|s)$.

We say a duplication g in a gene tree T is *upper* if the path from g to the root of T contains only duplications. The set of all upper duplications in a tree T we denote by UDup_T. We say that a valid mapping $\mathsf{F}_T\colon V(G|g) \to V(S|s)$ is *weakly feasible* for (g,s,X), if and only if, there is no feasible mapping for (g,s,X), but there is T that extends $G|g$, T has at least one upper duplication d such that $F_T(d) \notin X$ and $\mathsf{F}_T(\mathsf{Dup}_T \setminus \mathsf{UDup}_T) \subseteq X$. In contrast to feasible mappings, in weakly feasible mappings we constrain only non-upper duplications present in $G|g$. Here, the upper duplications are elements of episodes $s \notin X$. This situation is modeled by Unknown value returned from $\delta^{\downarrow}(g,s)$ and $\delta(g,s)$ calls, meaning that there is at least one duplication that needs to be assigned later (if possible) to an episode from $X \setminus V(S|s)$, which eventually occurs at levels of recursion shallower than the level of (g,s).

Informally, the meaning of DP formulas can be understood as follows. Below, let T be an extension of the subtree of G rooted at a node g. The value of $\delta(g,s)$ is True if there exists T where g is an s-duplication and all duplications are assigned to the episodes from X (where $s \in X$ as well). Similarly, the value of $\sigma(g,s)$ is True if there exists T, where g represents an s-speciation or an s-leaf node. Next, $\delta(g,s)$ is Unknown if the condition for $\delta(g,s) =$ True is not met. However, there still exists T, where g represents an s-duplication,

and all non-upper duplications from T are assigned to the episodes from X, while the upper duplications are assigned to episodes outside of X. It is important to note that in this case, s is not an element of X. Note that $\sigma(g, s)$ cannot be Unknown since speciation nodes are fixed. Moving on, $\delta^{\downarrow}(g, s)$ is True if there exists T where all duplications are assigned to the episodes from X. Lastly, $\delta^{\downarrow}(g, s)$ is Unknown if the condition for $\delta^{\downarrow}(g, s) =$ True is not met. However, there still exists T, where all non-upper duplications from T are assigned to the episodes from X, while the upper duplications are assigned to episodes outside of X.

While ϵ and δ^* should be treated as "local" in the main formulas (i.e., they should not form separate arrays in implementation), their properties can be formulated as follows. Generally, if $\epsilon(g, s)$ is True, then there is a tree T, where g is mapped into s and all duplications are assigned to the episodes from X. Since $\sigma(g, s)$ cannot be Unknown, $\epsilon(g, s)$ is Unknown only if $\sigma(g, s)$ is False and $\delta(g, s) =$ Unknown. Again, here g is mapped to s. Now, $\delta^*(g, s)$ is True only if there is T where g is an s-duplication, and all duplication nodes below g are assigned to episodes from X. Importantly, at least one of the children of g must be mapped to s, which is captured by ϵ. Furthermore, $\delta^*(g, s)$ resembles $\delta(g, s)$, but the condition that duplications must be assigned to episodes from X only applies to the duplications (or upper-duplications) below g if $\delta^*(g, s)$ is True (or Unknown, respectively).

The following Lemma formalizes the conditions described above.

▶ **Lemma 1.** *Given a partial gene tree G over a species tree S and $X \subseteq V(S)$. Let $g \in V(G)$, $s \in V(S)$. Then,*

P1 *$\delta(g, s) =$ True if and only if there is a gene tree T and a feasible mapping F_T for (g, s, X) such that g is an s-duplication in T.*

P2 *$\delta(g, s) =$ Unknown if and only if there is a gene tree T and a weakly feasible mapping F_T for (g, s, X) such that g is an s-duplication in T.*

P3 *$\sigma(g, s) =$ True if and only if there a gene tree T and a feasible mapping F_T for (g, s, X) such that and g is an s-speciation or an s-leaf in T.*

P4 *For any g and s, $\sigma(g, s) \neq$ Unknown.*

P5 *$\delta^{\downarrow}(g, s)$ is True if and only if there is a feasible mapping for (g, s, X).*

P6 *$\delta^{\downarrow}(g, s)$ is Unknown if and only if there is a weakly feasible mapping for (g, s, X).*

To solve Episode Feasibility, we have to apply $\delta^{\downarrow}$ on the roots of the input trees.

▶ **Theorem 2** (Correctness). *Given a partial gene tree G over a species tree S and $X \subseteq V(S)$. G is X-feasible if and only if $\delta^{\downarrow}_X(\mathsf{root}(G), \mathsf{root}(S))$ is* True.

Proof. The proof follows immediately from P5 of Lemma 1: $\delta^{\downarrow}_X(\mathsf{root}(G), \mathsf{root}(S))$ is True if and only if there is a feasible F_T for $(\mathsf{root}(G), \mathsf{root}(S), X))$ such that T extends G and $\mathsf{F}_T(\mathsf{Dup}_T) \subseteq X$. ◀

▶ **Theorem 3** (Complexity). *Given a partial gene tree G over a species tree S and $X \subseteq V(S)$. The time and space complexity of solving Episode Feasibility by the dynamic programming algorithm is $O(|V(G)||V(S)|)$.*

Proof. We have three arrays δ_X, $\delta^{\downarrow}_X$ and σ_X (note that ϵ and δ^* can be directly inserted in their calls), each of size $O(|V(G)||V(S)|)$ and every cell of an array can be computed in $O(1)$ time. ◀

An example of DP execution with a feasible solution is depicted in Figure 3.

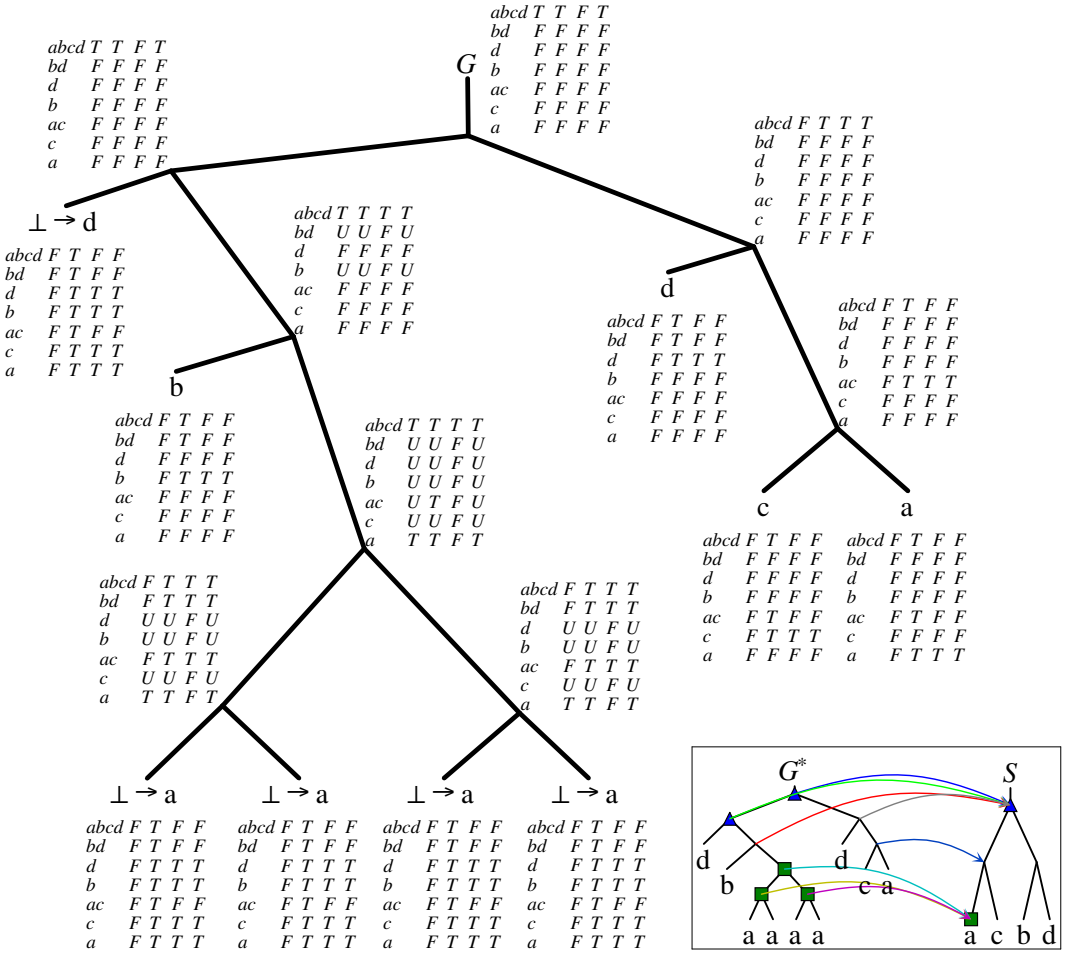


Figure 3 An example of the dynamic programming algorithm (from Section 3.1) execution. The partial gene tree is $G = ((\perp, (b, ((\perp, \perp), (\perp, \perp)))), (d, (c, a)))$, which contains five missing labels. The species tree, denoted as S, is represented as $((a, c), (b, d))$. The marked nodes in S indicate episode candidates from X: the root of S ($abcd$) and the leaf node a. By applying dynamic programming, we obtain a feasible solution, depicted in the bottom-right corner. The resulting extension of the partial gene tree G is G^*, where the valid mapping between G^* and S is the lowest common ancestor (LCA) mapping. In G^*, each duplication node is marked with a triangle or a square denoting their corresponding episode in S. Each node in G is decorated with an array that represents the values of DP formulas, where each row corresponds to a node in S, starting from $abcd$, bd, and so on as indicated in the first column. The next columns have the values of δ, $\delta^{\downarrow}$, σ, and ϵ, respectively, for the gene tree node and the corresponding species tree node. For example, considering the root of G and the root of S, the top row of the array contains the following values: $\delta(\mathsf{root}(G), \mathsf{root}(S)) = \delta^{\downarrow}(\mathsf{root}(G), \mathsf{root}(S)) = \epsilon(\mathsf{root}(G), \mathsf{root}(S)) = \mathsf{True}$, while $\sigma(\mathsf{root}(G), \mathsf{root}(S)) = \mathsf{False}$.

3.2 Solving MetaEC for a single partial gene tree

Here we describe the main algorithm to solve MetaEC for instances with a single gene tree. First, we characterize an important property of episodes.

▶ **Lemma 4** (Fixed Episodes). *Given a partial gene tree G over a species tree S. Assume that there are nodes g in G and s in S such that*

- *if s is the root of S, then at least one proper subtree of G contains species (leaf-labels) from both children of s.*
- *otherwise, let p be the parent of s, then $G|g$ is a gene tree, g is a p-speciation and a child of g is an s-duplication.*

Then, for any G^* that extends G, s is an episode in every valid mapping between G^* and S.

The nodes satisfying the above conditions we call *fixed episodes* (for G and S). For example, for trees from Figure 1, there are two fixed episodes: the root of S and the leaf b, where the duplications with fixed mappings are depicted using white marks in the exemplary gene tree G. The set of all fixed episodes can be computed in linear time and space by bottom-up traversal of the partial gene tree G and by using LCA-queries in the species tree S as follows. For each node g from $V(G)$, the algorithm computes a tuple (u, s, d), where

- $u \in \{\mathsf{True}, \mathsf{False}\}$ is True if and only if $\perp$ is reachable from g,
- $s \in V(S) \cup \{\mathsf{None}\}$ is the least common ancestor of all non-$\perp$ labels reachable from g in S and None if only $\perp$'s are visible from g,
- and $d \in \{\mathsf{True}, \mathsf{False}\}$ is True if and only if $u = \mathsf{False}$ and g is a duplication node in a gene tree $G|g$.

Then, for each g and its tuple (u, s, d), and for each child of g with a tuple (u', s', d'):

- if $s = s' = \mathsf{root}(S)$, then s (the root of S) is a fixed episode,
- if $u = d = \mathsf{False}$, $d' = \mathsf{True}$ and the parent of s' is s, then s' is a fixed episode.

We omit correctness and complexity proofs for brevity. Note that the number of fixed episodes is the lower bound of $\mathsf{EC}(G, S)$.

Algorithm 1 takes as input a partial gene tree G over a species tree S and outputs $\mathsf{EC}(G, S)$. It first computes the set of fixed episodes F (see Lemma 4). The algorithm then starts with an initial maximal episode number b equal to the number of nodes in S. In each iteration of a while loop, the algorithm checks if there is a set C of size $b - |F| - 1$ from the vertices of S that are not in F, such that the partial gene tree G is $C \cup F$-feasible using the dynamic programming algorithm. This step requires $\binom{|V(S)|-|F|}{b-|F|-1}$ calls of DP in the worst case. If such a set C exists, the algorithm computes $\mathsf{EC}(G^*, S)$ by the linear time algorithm from [29], where G^* is the gene tree obtained by backtracking from the corresponding call of DP, and updates b with the result. Note that b is not assigned the value of $|C \cup F|$, since the minimal set of episodes for G^* and S is a subset of $C \cup F$, and it is often significantly smaller than $C \cup F$ in early steps of iteration. Updating b with $\mathsf{EC}(G^*, S)$ guarantees the minimal number of episodes, where some elements of C may be unused. This is an important optimization step. If such a set C does not exist, the algorithm terminates and returns the current value of b.

The correctness of the algorithm follows from the fact that if there is no set X of size $b - 1$ such that G is X-feasible, then there is no set of any size smaller than b that satisfies the property. Since, b represents the number of episodes from some valid mapping, it is also minimal in such a case. Therefore, when the algorithm terminates, $b = \mathsf{EC}(G, S)$, and the algorithm returns the correct value.

Algorithm 1 Solution to MetaEC with a single gene tree.

Data: A partial gene tree G over a species tree S
Result: $\mathsf{EC}(G, S)$
$F \leftarrow$ the set of all fixed episodes for G and S ; `// See Lemma 4`
$b \leftarrow |V(S)|$; `// the initial maximal episode number`
while *there is* $C \subseteq V(S) \setminus F$ *of the size* $b - |F| - 1$ *and* G *is* $(C \cup F)$*-feasible* **do**
  $G^* \leftarrow$ the gene tree obtained from the DP from Section 3.1 by backtracking;
  $b \leftarrow \mathsf{EC}(G^*, S)$; `//` $O(n)$`-time algorithm from [29];` $\mathsf{EC}(G^*, S) \leq |C \cup F|$.
return b;

The algorithm's worst-case time complexity is $\sum_{k=f}^{n-f} \binom{n-f}{k} nm = O(nm2^n)$, where f is the size of the set of fixed episodes ($f = |F|$), n denotes the number of vertices in S, and m denotes the number of vertices in G. Despite the exponential time complexity, in our experiments on both simulated and empirical data, we were able to compute exact solutions after only a few executions of the main loop.

3.2.1 Extensions

To identify the optimal solution within the main loop, enumerating all possible combinations of size $b-f-1$ from the set of episode candidates $V(S) \setminus F$ may be time-consuming for larger instances. To address this issue, we propose a heuristic approach that randomly samples combinations of size $b-f-1$ if $\binom{n-f}{b-f-1}$ is large (e.g., > 1000), and adds a stopping condition based on the number of dynamic programming (DP) calls without improvement (e.g., after 100 calls). This approach not only speeds up the algorithm but also provides additional information on whether the returned value is exact or an upper bound obtained by switching to a heuristic mode. See Figure 6 in Section 4 for more details.

Furthermore, based on our experiments, we have observed that the solution is often close to the set of fixed episodes. To leverage this observation, we propose a bottom-up algorithm that explores candidate sets starting from sizes 0, 1, 2, and so on until a feasible solution is found. In this case, the internal search has a time complexity of $O(\binom{n-f}{i})$, starting from $i = 0$. This algorithm can be combined with the heuristic variant described earlier to improve its effectiveness. However, the experimental evaluation did not show significant improvement compared to the top-down method in Algorithm 1.

3.3 MetaEC in the general case

Here we show that MetaEC in a general case, can be solved using a single partial gene tree under an additional assumption. Given a collection of partial gene trees $G_1, G_2, \ldots, G_k$ over a species tree S, let ω be a new species, called *outgroup*, not present in S. We first add the outgroup to every input tree. Let S^ω be species tree (S, ω), $G_1^\omega = (G_1, \omega)$ and $G_i^\omega = ((G_i, \omega), G_{i-1}^\omega)$, for $i > 1$. Then, by ω-MetaEC we define the problem MetaEC with a single partial gene tree, where the extension of a partial labeling cannot introduce ω, i.e., if $\Lambda_{G_1}(v) = \perp$ then $\Lambda_{G_1^*}(v) \neq \omega$. See Figure 4 for illustration.

We have the following property.

▶ **Lemma 5.** *Given a collection of at least two partial gene trees $G_1, G_2, \ldots, G_k$ over a species tree S such that $\omega \notin L(S)$. $X \subseteq V(S)$ is the set of episodes that yields the solution of MetaEC for $G_1, G_2, \ldots, G_k$ and S if and only if $X \cup \{\mathsf{root}(S^\omega)\}$ is the set of episodes that yields the solution to the instance G_k^ω and S^ω of ω-MetaEC.*

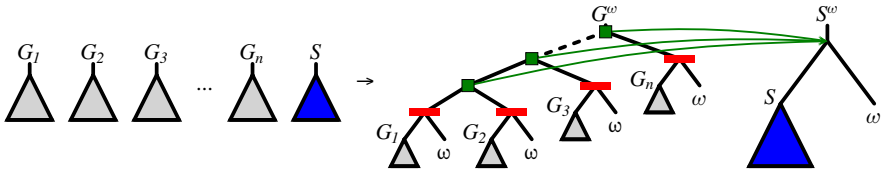


Figure 4 Converting a multiple gene tree instance to a single gene tree instance using an outgroup ω. Red bars in G^ω denote speciation nodes mapped to the root of S^ω. Green squares represent new duplications clustered at a new duplication episode in the root of S^ω.

Note that the algorithms provided in the previous sections can be easily modified to solve ω-MetaEC, by replacing case (8) with:

$$\sigma(g, s) = \mathsf{True} \text{ if } g \in L(G) \text{ and } (\Lambda_G(g) = s \text{ or } (\Lambda_G(g) = \perp \text{ and } s \neq \omega)).$$

Then, DP will exclude extensions of $\perp$ by ω.

4 Experiments

In this Section we present two computational studies based on simulated and empirical data.

4.1 Simulated Trees

Our algorithm was evaluated on five datasets consisting of simulated gene trees and a species tree, each with one or two whole duplication events (as reported in [31]). We generated these datasets and subsequently modified the gene trees to account for the uncertainty commonly associated with metagenomic data.

To begin, we describe how the datasets were generated. Then, we explain the modifications made to the gene trees to account for the uncertainty of metagenomic data. Finally, we present the results provided by our algorithms, specifically in regard to their ability to infer whole genome duplication events.

4.1.1 Dataset preparation

First, we briefly summarize the simulation procedure from [31]. The simulated dataset was generated by SimPhy [20] with parameter settings used in a simulated study [23] that was based on an empirical dataset of 16 Fungi species [34]. The species tree S of 20 taxa was generated by SimPhy with the speciation rate parameter equal to 1.8×10^{-9} and the tree height parameter set to 1.8×10^{9}. To simulate a whole genome duplication (WGD), a node v in the species tree S was chosen as the location of the event. Subsequently, a modified species tree, denoted as S', was constructed by substituting a subtree $S|v$ with a duplicated version of itself. This duplication involved creating a new root connected to the original root of $S|v$ and the root of its copy.

For every S_i one hundred gene trees were then generated from locus trees produced by SimPhy using the modified species tree S', with a duplication and loss rate parameter set to 2^{-10} events per generation per lineage. To minimize the effect of incomplete lineage sorting, the population size parameter was set to 10. In summary, the simulated data from [31] was partitioned into five datasets of gene trees denoted $\mathcal{G}_i = \{G_1^i, G_2^i, \ldots, G_{100}^i\}$, where each dataset comprises 100 gene trees generated using the same species tree S but with a different WGD scenario. The WGD variants used in the simulations are illustrated in Figure 5, where

S_1 represents a single recent WGD Γ, S_2 represents a single ancient WGD Θ, S_3 represents two WGDs Λ and Δ, with Δ occurring after Λ, S_4 represents two close WGDs Π and Π' at the same branch, and S_5 represents two recent independent WGDs Ψ and Φ.

In our evaluation study, for each dataset $\mathcal{G}_i$ from [31], we generated a set of partial gene trees $\tilde{\mathcal{G}}_i^{k,p}$ by randomly removing each leaf label from every gene tree G_j^i in $\mathcal{G}_i$ with probability p. We considered values of p from 0.1 to 0.6 in increments of 0.1, and generated 10 instances of $\tilde{\mathcal{G}}_i^{k,p}$ for each p value and $k = 1, 2, \ldots, 10$. This resulted in a total of 300 instances $(\tilde{\mathcal{G}}_i^{k,p}, S)$ of MetaEC, where S is the species tree used to generate $\mathcal{G}_i$. In addition to these instances, we also included the result for $p = 0$, which corresponds to no removal of leaf labels.

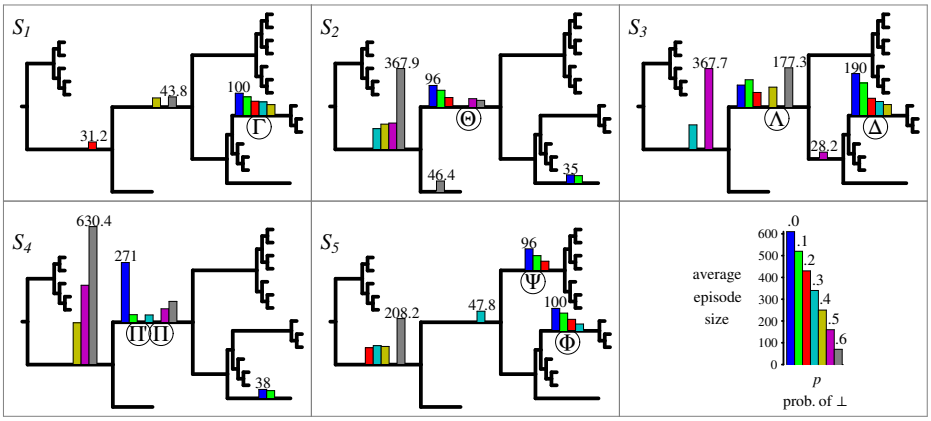


Figure 5 This figure presents a summary of the inferred gene-species mappings and duplication episodes based on the simulated dataset from [31]. Locations of simulated whole-genome duplication (WGD) events are denoted by Greek letters. For clarity, all leaf labels have been removed from the visualization of species trees (see [31] for details). Each bar in the histograms shows the average episode size obtained from the set of partial gene trees $\tilde{\mathcal{G}}_i^{k,p}$, where the average is computed over 10 instances of partial gene trees for fixed p and i (if $p > 0$). For $p = 0$, the bars represent the results for the set of simulated gene trees $\mathcal{G}_i$ (with no $\perp$'s). The key to the diagrams is present at the bottom-right corner. The number above a single bar represents the maximum height of a bar in its histogram. Bars with an average below 25 duplications were omitted as they were deemed insignificant. Additionally, histograms for the root have been moved to Figure 6 (bottom left diagram).

4.1.2 Results

The results of our algorithm on the simulated partial gene trees are depicted in Figure 5, where we summarize the episode sizes in the form of histograms. Also, additional data is provided in Figure 6. The evaluation took about 5 hours of a computing server with 64 cores. In general, we observed that the runtime and the number of DP calls grow linearly with the value of parameter p on average as indicated in the bottom-right diagram of Figure 6. We used the heuristic variant of the algorithm, where random sampling was applied if the number of combinations exceeded 100 trees, and with the stopping criterion equal to 50. Out of 300 instances, 240 were completed with the exact solution (see the top-left diagram of Figure 6). The resulting costs without exact guarantee, were more often obtained for larger values of p. Additionally, we observed that the lower bound given by the number of fixed episodes was a tight approximation of the inferred cost (see the top-right diagram of Figure 6). As p increased, the number of fixed episodes decreased. Note that for $p = 1$

(not included in our analysis), the solution to MetaEC is 1. In such a case the leaf labelings are constant functions and all internal nodes of each gene tree are duplications. Then there is just one episode, which can be placed at the root of the species tree.

Now, we briefly summarize the outcome of WGD detection. First, we present the analysis for the root episodes in a separate diagram in the bottom-left diagram of Figure 6 since the model overestimates the size of the root episode. This is partly due to the property that the last node to which a duplication can be assigned via a valid mapping is the root. In more detail, since the number of fixed episodes is a tight lower bound of the EC cost, the algorithm more likely assigns a duplication at the nearest fixed episode and at the root in the case where such an episode is not present.

The results for S_1 in Figure 5 show that Γ WGD is supported when $p \leq .4$. When $p > .4$, the duplications are relocated to more ancient locations. In S_2, Θ is supported only for $p \leq .2$, while for $p > .2$ large episodes are present at the parent of the WGD location. In S_3, Δ is well supported when $p \leq .4$. The case of Λ is more complicated, since for $p \leq .2$, $p = .4$ and $p = .6$ we see good support, while for the remaining values, the parent location is more supported. The ancient double WGDs in S_4 have perhaps the worst support in our study. Here, we observe generally low supports at the WGDs node. Most of the duplications were shifted to more ancient nodes of S_4, i.e., to the root or to the parent location. In S_5, the support is for smaller values of p.

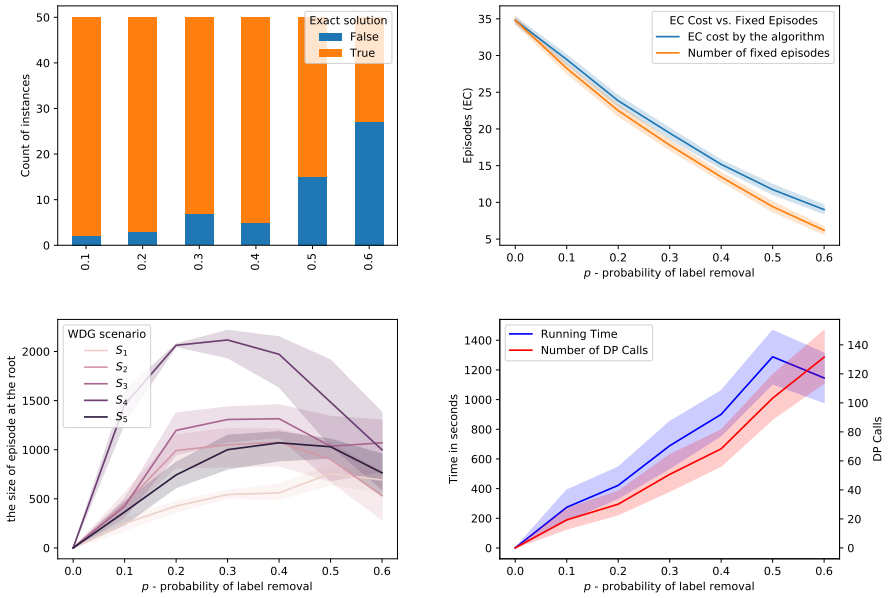


Figure 6 Summary of simulated dataset experiments. All diagrams, except for the top-left, display mean values with 95% confidence intervals. The top-left diagram shows histograms of exact and heuristic solutions returned by Algorithm 1. The top-right diagram presents the EC cost and the number of fixed episodes. The bottom-left diagram displays the sizes of root episodes in all five WGD scenarios. The bottom-right diagram shows the runtime in seconds and the number of executed DP calls.

4.2 Empirical evaluation

To ensure that our algorithm was properly tested, we required a dataset that would capture the characteristics of the metagenomic data as closely as possible, while allowing us to assess the quality and accuracy of the results obtained. For this reason, we decided to prepare

a dataset consisting of gene trees for species identified during metagenomic analysis. To simulate potentially missing gene-species assignments, we artificially removed some of the gene labels from the gene trees and retained information about their taxonomic origin for further analysis of the results. Another important issue was the presence of a previously described whole-genome duplication event that occurred in the evolutionary tree of selected species. Given the above requirements, we decided to use proteomes belonging to yeast species identified during metagenomic analysis of kefir [41].

4.2.1 Data preparation

The eight selected species are: *Kazachstania Africana*, *Kazachstania naganishii*, *Naumovozyma dairenensis*, *Tetrapisispora blatte*, *Tetrapisispora phaffi*, *Torulaspora delbrueckii*, *Zygosaccharomyces rouxii* and *Saccharomyces cerevisiae*. A species tree containing the listed species, consistent with the NCBI taxonomy and many papers on yeast evolution, is shown in Figure 7. It also shows the location of the whole-genome duplication event confirmed by previous studies [9, 21].

The proteomes used to infer gene trees were sourced from the UniProt database [5]. Protein families were created by dividing the proteins into groups using the *mcl* program [37] with parameters $I = 2$ and $I = 5$. However, since the differences between the obtained sets were minimal, we used the set obtained for $I = 2$ in subsequent steps. The protein sequences in each group were aligned with the MUSCLE algorithm [7], and unrooted gene trees were inferred using the *phyml* program [14] with the default parameter setting. Rooting of the gene trees was performed by *URec* program [12] using the minimal duplication-loss cost as the rooting criterion.

We removed trees containing fewer than 3 leaves or 3 species, as well as trees with edges of length 0 from the final set of trees. This resulted in 3430 rooted gene trees. Similar to the first experiment, we created 10 datasets for each $p \in \{0.1, 0.2, \ldots, 0.6\}$ by randomly removing each leaf label with the probability p. This resulted in 60 datasets plus the original dataset representing $p = 0$.

4.2.2 Results

Figure 7 depicts histograms showing the results obtained for the described dataset. The evaluation was performed on the same computing server as before and took approximately one hour. For this evaluation, we set the sampling threshold and stopping criterion to 50, which yielded exact solutions for all cases. The number of fixed episodes was consistent across datasets with $p < .6$, at 12 (note that the number of leaves in a tree was 15). For $p = 0.6$, the number of fixed episodes fell within the range of 9 to 12. The number of DP calls ranged from 2 to 3 for $p < .6$ and between 2 and 11 for $p = .6$.

The results obtained by the algorithm for the yeast dataset are consistent with our knowledge of the whole-genome duplication localization. For the dataset with all leaf labels present and for $p = .1$, we have the highest support for the WGD event. The number of supporting single duplications decreases gradually for successive $p's$. For values of $p \geq .5$, the correct WGD localization is still supported by a significantly large number of single duplications. It is worth noting that even for the $p = .6$, the right location is supported by three times as many duplications as the second most supported location, which is in the root. Additionally, we observed an increase in duplications at the leaves of the species tree as p increased. Since most of the leaves are fixed episodes, the algorithm often assigned labels to create duplications at the leaves, resulting in larger sizes of episodes at leaves.

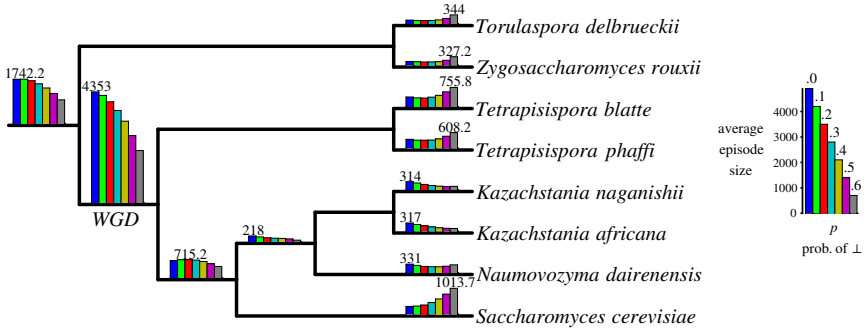


Figure 7 Summary of gene-species mappings and duplication episodes inference for the yeast dataset consisting of 3430 gene trees. WGD denotes the whole genome duplication event postulated in [9, 21]. For the description of symbols refer to Figure 5.

4.3 Software

The software package, which is partially based on the embretnet repository, has been written in Python and is available with all datasets at `https://bitbucket.org/pgor17/metaEC`.

5 Conclusions and Future Outlook

In this article, we presented a novel problem that integrates gene-species mapping inference and genomic duplication detection. We proposed efficient algorithms to solve the problem exactly in the majority of instances, along with a heuristic modification for cases where exact solutions are not feasible. To demonstrate the effectiveness and accuracy of our proposed algorithm, we conducted computational experiments on both simulated and empirical data. The results showed that our algorithm was able to accurately infer the corresponding events when the number of missing labels was relatively small for simulated data. Moreover, the algorithm performed even better on empirical data, demonstrating its robustness and applicability to real-world scenarios.

However, to maximize topological similarities between a gene tree and its species tree, speciation nodes should more frequently appear in the resulting extensions of input partial gene trees. We observe that the optimization model tends to reconstruct leaf labels in a way that prioritizes duplication events assigned to the nearest fixed episodes or the root, in the absence of such episodes. This is confirmed by the property that fixed episodes are tight approximations of the EC cost, leading to a reduction in the number of speciation events in the final gene tree extensions. As a consequence, the model's effectiveness may be limited in some cases when the number of missing labels in partial gene trees is significant.

In the future, we plan to extend the analyzed model to address the importance of speciation nodes in the EC cost formulation. Alternatively, one may limit the distance between the lca-mapping of a gene duplication and its destination mapping in the final scenario similarly to [31]. Additionally, there are models of genomic duplications providing a higher level of detail than EC, such as minimum episodes (ME) [29] and RMP [15], which can be adapted in a similar way to infer gene-species mappings and minimize the number of duplication episodes simultaneously. These models can be further combined with more general models of valid mappings, which allow the introduction of more duplication events than the minimum obtained by the lca-mapping [11]. The combination of these models can provide a more comprehensive approach to inferring gene-species mappings and identifying the minimum number of duplication episodes.

References

1 Mukul S Bansal and Oliver Eulenstein. The multiple gene duplication problem revisited. *Bioinformatics*, 24(13):i132–i138, 2008.

2 Arkadiusz Betkier, Paweł Szczęsny, and Paweł Górecki. Fast algorithms for inferring gene-species associations. In *Bioinformatics Research and Applications: 11th International Symposium, ISBRA 2015 Norfolk, USA, June 7-10, 2015 Proceedings 11*, pages 36–47. Springer, 2015.

3 Craig M. Bielski, Ahmet Zehir, Alexander V. Penson, Mark T. A. Donoghue, Walid Chatila, Joshua Armenia, Matthew T. Chang, Alison M. Schram, Philip Jonsson, Chaitanya Bandlamudi, Pedram Razavi, Gopa Iyer, Mark E. Robson, Zsofia K. Stadler, Nikolaus Schultz, Jose Baselga, David B. Solit, David M. Hyman, Michael F. Berger, and Barry S. Taylor. Genome doubling shapes the evolution and prognosis of advanced cancers. *Nature Genetics*, 50(8):1189–1195, 2018.

4 J Gordon Burleigh, Mukul S Bansal, Andre Wehe, and Oliver Eulenstein. Locating multiple gene duplications through reconciled trees. In *Research in Computational Molecular Biology: 12th Annual International Conference, RECOMB 2008, Singapore, March 30-April 2, 2008. Proceedings 12*, pages 273–284. Springer, 2008.

5 The UniProt Consortium. Uniprot: the universal protein knowledgebase in 2023. *Nucleic Acids Research*, 51(D1):D523–D531, 2023.

6 Riccardo Dondi, Manuel Lafond, and Celine Scornavacca. Reconciling multiple genes trees via segmental duplications and losses. *Algorithms for Molecular Biology*, 14:7, 2019.

7 Robert C Edgar. Muscle: a multiple sequence alignment method with reduced time and space complexity. *BMC Bioinformatics*, 5(1):1–19, 2004.

8 Michael Fellows, Michael Hallet, and Ulrike Stege. On the multiple gene duplication problem. In *9th International Symposium on Algorithms and Computation (ISAAC'98), Lecture Notes in Computer Science 1533*, pages 347–356, Taejon, Korea, 1998.

9 Bing Feng, Yu Lin, Lingxi Zhou, Yan Guo, Robert Friedman, Ruofan Xia, Fei Hu, Chao Liu, and Jijun Tang. Reconstructing yeasts phylogenies and ancestors from whole genome data. *Scientific Reports*, 7(1):1–12, 2017.

10 Morris Goodman, John Czelusniak, G. William Moore, A. E. Romero-Herrera, and Genji Matsuda. Fitting the gene lineage into its species lineage, a parsimony strategy illustrated by cladograms constructed from globin sequences. *Systematic Zoology*, 28(2):132–163, 1979.

11 Paweł Górecki and Jerzy Tiuryn. DLS-trees: A model of evolutionary scenarios. *Theoretical Computer Science*, 359(1-3):378–399, 2006.

12 Paweł Górecki and Jerzy Tiuryn. Urec: a system for unrooted reconciliation. *Bioinformatics*, 23(4):511–512, 2007.

13 Roderic Guigó, Ilya B. Muchnik, and Temple F. Smith. Reconstruction of ancient molecular phylogeny. *Molecular Phylogenetics and Evolution*, 6(2):189–213, 1996.

14 Stéphane Guindon, Jean-François Dufayard, Lefort Vincent, 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.

15 Leo Van Iersel, Remie Janssen, Mark Jones, Yukihiro Murakami, and Norbert Zeh. Polynomial-Time Algorithms for Phylogenetic Inference Problems involving duplication and reticulation. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2019.

16 Elena Kuzmin, Benjamin VanderSluis, Alex N. Nguyen Ba, Wen Wang, Elizabeth N. Koch, Matej Usaj, Anton Khmelinskii, Mojca Mattiazzi Usaj, Jolanda van Leeuwen, Oren Kraus, Amy Tresenrider, Michael Pryszlak, Ming-Che Hu, Brenda Varriano, Michael Costanzo, Michael Knop, Alan Moses, Chad L. Myers, Brenda J. Andrews, and Charles Boone. Exploring whole-genome duplicate gene retention with complex genetic interaction analysis. *Science*, 368(6498):eaaz5667, 2020.

17 Cheng-Wei Luo, Ming-Chiang Chen, Yi-Ching Chen, Roger W. L. Yang, Hsiao-Fei Liu, and Kun-Mao Chao. Linear-time algorithms for the multiple gene duplication problems. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(1):260–265, 2011.

18 Saioa López, Emilia L Lim, Stuart Horswell, Kerstin Haase, Ariana Huebner, Michelle Dietzen, Thanos P Mourikis, Thomas B K Watkins, Andrew Rowan, Sally M Dewhurst, Nicolai J Birkbak, Gareth A Wilson, Peter Van Loo, Mariam Jamal-Hanjani, TRACERx Consortium, Charles Swanton, and Nicholas McGranahan. Interplay between whole-genome doubling and the accumulation of deleterious alterations in cancer evolution. *Nature Genetics*, 52(3):283–293, 2020.

19 Bin Ma, Ming Li, and Louxin Zhang. From gene trees to species trees. *SIAM Journal on Computing*, 30(3):729–752, 2000.

20 Diego Mallo, Leonardo De Oliveira Martins, and David Posada. Simphy: Phylogenomic simulation of gene, locus, and species trees. *Systematic Biology*, 65(2):334–344, 2016.

21 Marina Marcet-Houben and Toni Gabaldón. Beyond the whole-genome duplication: phylogenetic evidence for an ancient interspecies hybridization in the baker's yeast lineage. *PLoS biology*, 13(8):e1002220, 2015.

22 Vacharapat Mettanant and Jittat Fakcharoenphol. A linear-time algorithm for the multiple gene duplication problem. In *The 12th National Computer Science and Engineering Conference (NCSEC)*, pages 198–203, 2008.

23 Erin K Molloy and Tandy Warnow. FastMulRFS: fast and accurate species tree estimation under generic gene duplication and loss models. *Bioinformatics*, 36(Supplement_1):i57–i65, 2020.

24 Agnieszka Mykowiecka, Paweł Szczęsny, and Paweł Górecki. Inferring gene-species assignments in the presence of horizontal gene transfer. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 15(5):1571–1578, 2017.

25 Susumu Ohno. *Evolution by gene duplication*. Springer-Verlag, Berlin, 1970.

26 Roderic D. M. Page. Maps Between Trees and Cladistic Analysis of Historical Associations among Genes, Organisms, and Areas. *Systematic Biology*, 43(1):58–77, 1994.

27 Roderic D.M. Page and James A. Cotton. Vertebrate phylogenomics: Reconciled trees and gene duplications. *Pacific Symposium on Biocomputing*, pages 536–547, 2002.

28 Jarosław Paszek and Paweł Górecki. Genomic duplication problems for unrooted gene trees. *BMC Genomics*, 17(1):165–175, 2016.

29 Jarosław Paszek and Paweł Górecki. Efficient algorithms for genomic duplication models. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 15(5):1515–1524, 2018.

30 Jarosław Paszek and Paweł Górecki. Inferring duplication episodes from unrooted gene trees. *BMC Genomics*, 19(S5), 2018.

31 Jarosław Paszek, Alexey Markin, Paweł Górecki, and Oliver Eulenstein. Taming the duplication-loss-coalescence model with integer linear programming. *Journal of Computational Biology*, 28(8):758–773, 2021.

32 Jarosław Paszek, Jerzy Tiuryn, and Paweł Górecki. Minimizing genomic duplication episodes. *Computational Biology and Chemistry*, 89:107260, 2020.

33 Ryan J Quinton, Amanda DiDomizio, Marc A Vittoria, Kristýna Kotýnková, Carlos J Ticas, Sheena Patel, Yusuke Koga, Jasmine Vakhshoorzadeh, Nicole Hermance, Taruho S Kuroda, Neha Parulekar, Alison M Taylor, Amity L Manning, Joshua D Campbell, and Neil J Ganem. Whole-genome doubling confers unique genetic vulnerabilities on tumour cells. *Nature*, 590(7846):492–497, 2021.

34 Matthew D. Rasmussen and Manolis Kellis. Unified modeling of gene duplication, loss, and coalescence using a locus tree. *Genome Research*, 22(4):755–765, 2012.

35 Marta Royo-Llonch, Pablo Sánchez, Clara Ruiz-González, Guillem Salazar, Carlos Pedrós-Alió, Marta Sebastián, Karine Labadie, Lucas Paoli, Federico M. Ibarbalz, Lucie Zinger, Benjamin Churcheward, Tara Oceans Coordinators, Samuel Chaffron, Damien Eveillard, Eric Karsenti, Shinichi Sunagawa, Patrick Wincker, Lee Karp-Boss, Chris Bowler, and Silvia G. Acinas. Compendium of 530 metagenome-assembled bacterial and archaeal genomes from the polar Arctic Ocean. *Nature Microbiology*, 6(12):1561–1574, 2021.

36 Ayelet Salman-Minkov, Niv Sabath, and Itay Mayrose. Whole-genome duplication as a key factor in crop domestication. *Nature Plants*, 2:16115, 2016.

37 Stijn Van Dongen. Graph clustering via a discrete uncoupling process. *SIAM Journal on Matrix Analysis and Applications*, 30(1):121–141, 2008.

38 Jakob Wirbel, Paul Theodor Pyl, Ece Kartal, Konrad Zych, Alireza Kashani, Alessio Milanese, Jonas S Fleck, Anita Y Voigt, Albert Palleja, Ruby Ponnudurai, Shinichi Sunagawa, Luis Pedro Coelho, Petra Schrotz-King, Emily Vogtmann, Nina Habermann, Emma Niméus, Andrew M Thomas, Paolo Manghi, Sara Gandini, Davide Serrano, Sayaka Mizutani, Hirotsugu Shiroma, Satoshi Shiba, Tatsuhiro Shibata, Shinichi Yachida, Takuji Yamada, Levi Waldron, Alessio Naccarati, Nicola Segata, Rashmi Sinha, Cornelia M Ulrich, Hermann Brenner, Manimozhiyan Arumugam, Peer Bork, and Georg Zeller. Meta-analysis of fecal metagenomes reveals global microbial signatures that are specific for colorectal cancer. *Nature Medicine*, 25(4):679–689, 2019.

39 Kenneth H Wolfe and Denis C Shields. Molecular evidence for an ancient duplication of the entire yeast genome. *Nature*, 387(6634):708–713, 1997.

40 Shan Wu, Kin H Lau, Qinghe Cao, John P Hamilton, Honghe Sun, Chenxi Zhou, Lauren Eserman, Dorcus C Gemenet, Bode A Olukolu, Haiyan Wang, Emily Crisovan, Grant T Godden, Chen Jiao, Xin Wang, Mercy Kitavi, Norma Manrique-Carpintero, Brieanne Vaillancourt, Krystle Wiegert-Rininger, Xinsun Yang, Kan Bao, Jennifer Schaff, Jan Kreuze, Wolfgang Gruneberg, Awais Khan, Marc Ghislain, Daifu Ma, Jiming Jiang, Robert O M Mwanga, Jim Leebens-Mack, Lachlan J M Coin, G Craig Yencho, C Robin Buell, and Zhangjun Fei. Genome sequences of two diploid wild relatives of cultivated sweetpotato reveal targets for genetic improvement. *Nature Communications*, 9(1):4580, 2018.

41 Birsen Yilmaz, Emine Elibol, H Nakibapher Jones Shangpliang, Fatih Ozogul, and Jyoti Prakash Tamang. Microbial communities in home-made and commercial kefir and their hypoglycemic properties. *Fermentation*, 8(11):590, 2022.

42 Louxin Zhang and Yun Cui. An efficient method for dna-based species assignment via gene tree and species tree reconciliation. In *Algorithms in Bioinformatics: 10th International Workshop, WABI 2010, Liverpool, UK, September 6-8, 2010. Proceedings 10*, pages 300–311. Springer, 2010.

43 Jan Łukasiewicz. *Selected Works*, volume 1. North-Holland Publishing Company, Amsterdam, 1970.

Making a Network Orchard by Adding Leaves

Leo van Iersel
Delft Institute of Applied Mathematics, Delft University of Technology, The Netherlands

Mark Jones
Delft Institute of Applied Mathematics, Delft University of Technology, The Netherlands

Esther Julien
Delft Institute of Applied Mathematics, Delft University of Technology, The Netherlands

Yukihiro Murakami[1]
Delft Institute of Applied Mathematics, Delft University of Technology, The Netherlands

Abstract

Phylogenetic networks are used to represent the evolutionary history of species. Recently, the new class of orchard networks was introduced, which were later shown to be interpretable as trees with additional horizontal arcs. This makes the network class ideal for capturing evolutionary histories that involve horizontal gene transfers. Here, we study the minimum number of additional leaves needed to make a network orchard. We demonstrate that computing this proximity measure for a given network is NP-hard and describe a tight upper bound. We also give an equivalent measure based on vertex labellings to construct a mixed integer linear programming formulation. Our experimental results, which include both real-world and synthetic data, illustrate the efficiency of our implementation.

2012 ACM Subject Classification Mathematics of computing → Trees; Mathematics of computing → Graph algorithms; Applied computing → Biological networks; Applied computing → Bioinformatics

Keywords and phrases Phylogenetics, Network, Orchard Networks, Proximity Measures, NP-hardness

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.7

Supplementary Material
Software (Source Code): `https://github.com/estherjulien/OrchardProximity`
archived at `swh:1:dir:d80c03267ac6a1474c350655e337f1ae7de3f945`

Funding *Leo van Iersel*: Research funded in part by Netherlands Organization for Scientific Research (NWO) grants OCENW.KLEIN.125 and OCENW.GROOT.2019.015.
Mark Jones: Research funded by Netherlands Organization for Scientific Research (NWO) grant OCENW.KLEIN.125.
Esther Julien: Research funded by Netherlands Organization for Scientific Research (NWO) grant OCENW.GROOT.2019.015.

1 Introduction

Phylogenetic trees are used to represent the evolutionary history of species. While they are effective for illustrating speciation events through vertical descent, they are insufficient in representing more intricate evolutionary processes. Reticulate (net-like) events such as hybridization and horizontal gene transfer (HGT) can give rise to signals that cannot be represented on a single tree [9, 21]. In light of this, phylogenetic networks have gained increasing attention due to their capability in elucidating reticulate evolutionary processes.

[1] Corresponding author

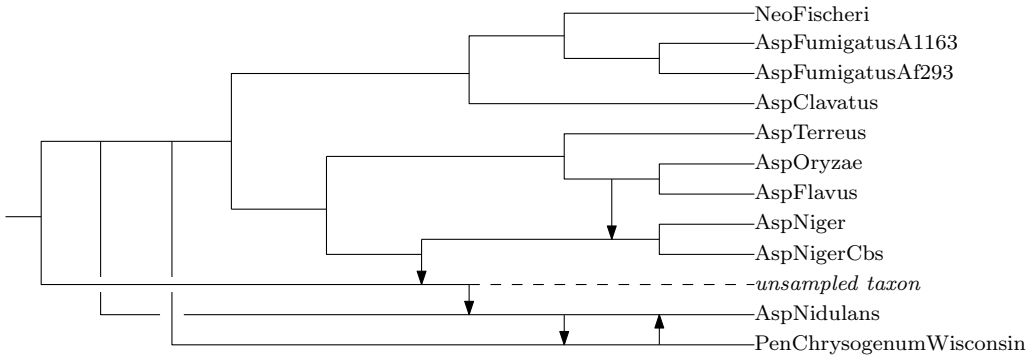


Figure 1 A network on 11 different taxa (excluding *unsampled taxon*) of fungi including 5 reticulations, which is part of a larger network from [18]. The directed arcs in the figure are linking arcs, which represent gene transfer highways. In order to make all linking arcs horizontal, we require an additional leaf (*unsampled taxon*) to represent the evolutionary history. To see that the network needs the leaf *unsampled taxon* in order to have only horizontal linking arcs, we refer the interested reader to Appendix A.

Phylogenetic networks are often categorized into different classes based on their topological features. These are often motivated computationally, but some classes are also defined based on their biological relevance [16]. Classical examples of network classes involve the *tree-child networks* [3] and the *tree-based networks* [8]. Roughly speaking, tree-child networks are those where every vertex has passed on a gene via vertical descent to an extant species, and tree-based networks are those obtainable from a tree by adding so-called *linking arcs* between tree arcs. Recent developments have culminated in the introduction of *orchard networks*, which lie – inclusion-wise – between the two aforementioned network classes [14, 4]. The class has shown to be both algorithmically attractive and biologically relevant; they are defined as networks that can be reduced to a single leaf by a series of so-called *cherry-picking operations*, and they were shown to be networks that can be obtained by adding horizontal arcs to trees (where the tree is drawn with the root at the top and arcs pointing downwards) [19]. Such horizontal arcs can be used to model HGT events, making orchard networks especially apt in representing evolutionary scenarios where every reticulate event is a horizontal transfer. Orchard networks have also been characterized statically based on so-called *cherry covers* [20].

When considering a non-orchard network, a natural question arises: how many additional leaves are required to transform the network into one that is orchard? From a biological standpoint, this question can be interpreted as asking how many extinct species or unsampled taxa need to be introduced into the network to yield a scenario where every reticulation represents an HGT event. Given that HGT is the primary driver of reticulate evolution in bacteria [10], this is an essential inquiry. We provide a network of a few fungi species in Figure 1, which requires one additional leaf to make it orchard. Formally speaking, the problem of computing this leaf addition measure is as follows.

$L_{\mathcal{OR}}$-Distance (Decision)
Input: A network N on a set of taxa X and a natural number k.
Decide: Can N be made orchard with at most k leaf additions?

In related research, the leaf addition measure has been investigated for other network classes. It has been shown that tracking down the minimum leaf additions to make a network tree-based can be done in polynomial time [7]. In the same paper, it was shown that the leaf addition measure was equivalent to two other proximity measures, namely those based on spanning trees and disjoint path partitions. The same question was posed for the unrooted

variant (where the arcs of the network are undirected), for which the problem turned out to be NP-complete [5]. A total of eight proximity measures were introduced in this latter paper, including those based on edge additions and rearrangement moves. Instead of considering leaf additions, some manuscripts have even considered leaf deletions (in general, vertex deletions) as proximity measures for the class of so-called *edge-based networks* [6]. Finally for orchard networks, a recent bachelor's thesis compared how the leaf addition proximity measure differs in general to another proximity measure based on arc deletions [17].

In this paper, we show that the leaf addition proximity measure can be computed in polynomial time for the class of tree-child networks, and we give a more efficient algorithm for computing the measure for tree-based networks. We show that $L_{\mathcal{OR}}$-Distance is NP-complete by a polynomial-time reduction from Degree-3 Vertex Cover. To model the problem as a *mixed integer linear program* (MILP), we consider a reformulation of the leaf addition measure in terms of vertex labellings. Orchard networks are known to be trees with added horizontal arcs; roughly speaking, this means we can label the vertices of an orchard network so that every vertex of indegree-2 has exactly one incoming arc whose end-vertices have the same labels. The reformulated measure, called the *vertical arcs* proximity measure, counts – over all possible vertex labellings (defined formally in Section 6.1) – the minimum number of indegree-2 vertices with only non-horizontal incoming arcs. Our experimental results are promising, as the real world cases are solved in a fraction of a second. Furthermore, the model also scales well to larger synthetic data.

The structure of the paper is as follows. In Section 2, we provide all necessary definitions and characterizations of orchard networks and tree-based networks. In Section 3, we formally introduce the leaf addition measure for the classes of tree-child, orchard, and tree-based networks. In Section 4 we show that $L_{\mathcal{OR}}$-Distance is NP-complete (Theorem 17). In Section 5, we give a sharp upper bound for the leaf addition proximity measure. In Section 6 we give a reformulation of the leaf addition measure to describe the MILP to solve $L_{\mathcal{OR}}$-Distance, and in Section 6.3, experimental results are shown for the MILP, applied to real and simulated networks. In Section 7, we give a brief discussion of our results and discuss potential future research directions. We include proofs for select results in Appendix A.

2 Preliminaries

A *binary directed phylogenetic network* on a non-empty set X is a directed acyclic graph with

- a single *root* of indegree-0 and outdegree-1;
- *tree vertices* of indegree-1 and outdegree-2;
- *reticulations* of indegree-2 and outdegree-1;
- *leaves* of indegree-1 and outdegree-0, that are labelled bijectively by elements of X.

For the sake of brevity, we shall refer to binary directed phylogenetic networks simply as *networks*. Throughout the paper, assume that N is a network on some non-empty set X where $|X| = n$, unless stated otherwise. Networks without reticulations are called *trees*. Tree vertices and reticulations may sometimes collectively be referred to as *internal vertices*.

The arc uv of a network is a *root arc* if u is the root of the network. An arc uv of a network is a *reticulation arc* if v is a reticulation, and a *tree arc* otherwise. We say that a vertex u is a *parent* of another vertex v if uv is an arc of the network; in such instances we call v a *child* of u. Also, we say that u and v are the *tail* and the *head* of the arc uv, respectively. In other words, we may rewrite arcs as $uv = \mathrm{tail}(uv)\,\mathrm{head}(uv)$. The *neighbours* of v refer to the set of vertices that are parents or children of v. We also say that vertices u and v are *siblings* if they share a parent.

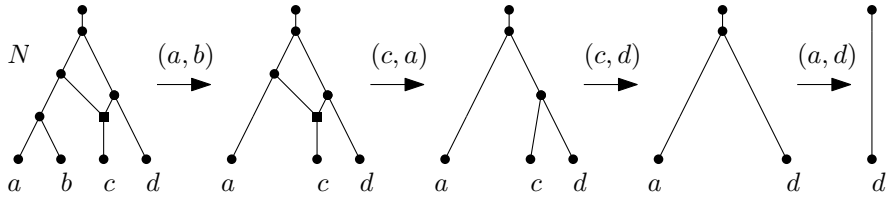


Figure 2 An example of an orchard network N that is reduced by a sequence $(a, b)(c, a)(c, d)(a, d)$. The network N contains a cherry (a, b) and a reticulated cherry (c, d). Subsequent networks are those obtained by a single cherry picking reduction from the previous network. For example, the second network $N(a, b)$ is obtained from N by removing the leaf b and cleaning up. Note that the network is also tree-child.

In what follows, we shall define graph operations based on vertex and arc deletions. To make sure resulting graphs remain networks, we follow-up every graph operation with a *cleaning up* process. Formally, we *clean up* a network by applying the following until none is applicable.

- Suppress an indegree-1 outdegree-1 vertex (e.g., if uv and vw are arcs where v is an indegree-1 outdegree-1 vertex, we suppress v by deleting the vertex v and adding an arc uw).
- Replace parallel arcs by a single arc (e.g., if uv is an arc twice in a network, delete one of the arcs uv).

We observe that deleting a tree arc and cleaning up results in a graph containing two indegree-0 vertices. On the other hand, deleting a reticulation arc and cleaning up results in a network. Therefore, we shall use arc deletions to mean reticulation arc deletions.

2.1 Tree-Child Networks

A network is *tree-child* if every non-leaf vertex has a child that is a tree vertex or a leaf. We call an internal vertex of a network an *omnian* if all of its children are reticulations [15]. It follows from definition that a network is tree-child if and only if it contains no omnians.

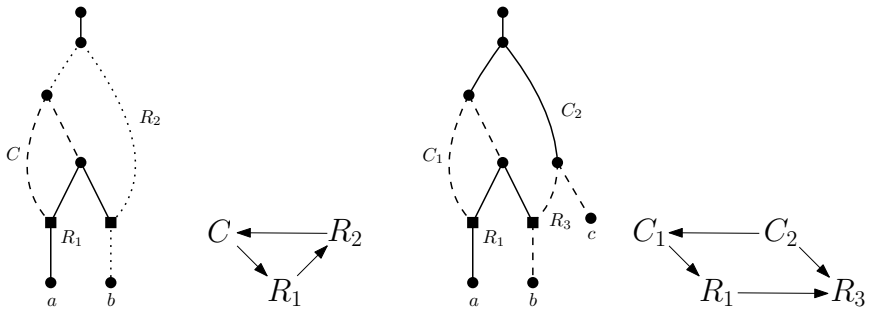
2.2 Orchard Networks

To define orchard networks, we must first define cherries and reticulated cherries, as well as operations to reduce them. See Figure 2 for the illustration of the following definitions. Let N be a network. Two leaves x and y of N form a *cherry* if they are siblings. In such a case, we say that N *contains* a cherry (x, y) or a cherry (y, x). Two leaves x and y of N form a *reticulated cherry* if the parent p_x of x is a reticulation and the parent of y is also a parent of p_x. In such a case, we say that N *contains* a reticulated cherry (x, y). *Reducing the cherry* (x, y) *from* N is the process of deleting the leaf x and cleaning up. *Reducing the reticulated cherry* (x, y) *from* N is the process of deleting the arc from the parent of y to the parent of x and cleaning up. In both cases, we use $N(x, y)$ to denote the resulting network.

A network N is *orchard* if there is a sequence $S = (x_1, y_1)(x_2, y_2) \dots (x_k, y_k)$ such that NS is a network on a single leaf y_k. It has been shown that the order in which (reticulated) cherries are reduced does not matter [4, 14]. Apart from this recursive definition, orchard networks have been characterized based on cherry covers (arc decompositions) [20] and vertex labellings [19]. We include both characterizations here.

Cherry covers (see [20] for more details). A *cherry shape* is a subgraph on three distinct vertices x, y, p with arcs px and py. The *internal vertex* of a cherry shape is p, and the *endpoints* are x and y. A *reticulated cherry shape* is a subgraph on four distinct vertices x, y, p_x, p_y with arcs $p_x x, p_y p_x, p_y y$, such that p_x is a reticulation in the network. The *internal vertices* of a reticulated cherry shape are p_x and p_y, and the endpoints are x and y. The *middle arc* of a reticulated cherry shape is $p_y p_x$. We will often refer to cherry shapes and the reticulated cherry shapes by their arcs (e.g., we would denote the above cherry shape $\{p_x x, p_y y\}$ and the reticulated cherry shape $\{p_x x, p_y p_x, p_y y\}$). We say that an arc uv is covered by a cherry or reticulated cherry shape B if $uv \in B$. A *cherry cover* of a binary network is a set P of cherry shapes and reticulated cherry shapes, such that each arc except for the root arc is covered exactly once by P. In general, a network can have more than one cherry cover.

We define the *cherry cover auxiliary graph* $G = (V, A)$ of a cherry cover as follows. For all shapes $B \in P$, we have $v_B \in V$. A shape $B \in P$ is *directly above* another shape $C \in P$ if B and C contain a same vertex v, such that v is an endpoint of B and an internal vertex of C. Then, $v_B v_C \in A$ (adapted from [20, Definition 2.13]). We say that a cherry cover is *cyclic* if its auxiliary graph has a cycle. We call it *acyclic* otherwise. See Figure 3 for an illustration of a cyclic and acyclic cherry cover.



(a) Network N_1. **(b)** Cherry cover aux. graph of N_1. **(c)** Network N_2. **(d)** Cherry cover aux. graph of N_2.

Figure 3 A cherry cover example. (a) A network N_1 on $\{a, b\}$ with a cherry cover $\{C, R_1, R_2\}$. (b) The (cyclic) auxiliary graph of N_1 based on the cherry cover of (a). (c) The network N_2 obtained from N_1 by adding a leaf c, with a cherry cover $\{C_1, C_2, R_1, R_3\}$ (d) The (acyclic) auxiliary graph of N_2 based on the cherry cover of (d).

▶ **Theorem 1** (Theorem 4.3 of [20]). *A network N is orchard if and only it has an acyclic cherry cover.*

Non-Temporal Labellings. Let N be a network with vertex set $V(N)$. A *non-temporal labelling*[2] of N is a labelling $t : V(N) \to \mathbb{R}$ such that

- for all arcs uv, $t(u) \leq t(v)$ and equality is allowed only if v is a reticulation;
- for each internal vertex u, there is a child v of u such that $t(u) < t(v)$;
- for each reticulation r with parents u and v, at most one of $t(u) = t(r)$ or $t(v) = t(r)$ holds.

[2] This is named in contrast to *temporal representations* of [1]. There, it was required for the endpoints of every reticulation arc to have the same label.

Observe that every network (orchard or not) admits a non-temporal labelling by labelling each vertex by its longest distance from the root (assuming each arc is of weight 1).

Under non-temporal labellings, we call an arc *horizontal* if its endpoints have the same label; we call an arc *vertical* otherwise. By definition, only reticulation arcs can be horizontal. We say that a non-temporal labelling is an *HGT-consistent labelling* if every reticulation is incident to exactly one incoming horizontal arc. We recall the following key result.

▶ **Theorem 2** (Theorem 1 of [19]). *A network is orchard if and only if it admits an HGT-consistent labelling.*

2.3 Tree-Based Networks

A network N is *tree-based* with *base tree* T if it can be obtained from T in the following steps.

1. Replace some arcs of T by paths, whose internal vertices we call *attachment points*; each attachment point is of indegree-1 and outdegree-1.
2. Place arcs between attachment points, called *linking arcs*, so that the graph contains no vertices of total degree greater than 3, and so that it remains acyclic.
3. Clean up.

The relation between the classes of tree-child, orchard, and tree-based networks can be stated as follows.

▶ **Lemma 3** ([14] and Corollary 1 of [19]). *If a network is tree-child, then it is orchard. If a network is orchard, then it is tree-based.*

We include here a static characterization of tree-based networks based on an arc partition, called *maximum zig-zag trails* [11, 23]. Let N be a network. A *zig-zag trail* of length k is a sequence $(a_1, a_2, \ldots, a_k)$ of arcs where $k \geq 1$, where either $\mathrm{tail}(a_i) = \mathrm{tail}(a_{i+1})$ or $\mathrm{head}(a_i) = \mathrm{head}(a_{i+1})$ holds for $i \in [k-1] = \{1, 2, \ldots, k-1\}$. We call a zig-zag trail Z *maximal* if there is no zig-zag trail that contains Z as a subsequence. Depending on the nature of $\mathrm{tail}(a_1)$ and $\mathrm{tail}(a_k)$, we have four possible maximal zig-zag trails.

- *Crowns*: $k \geq 4$ is even and $\mathrm{tail}(a_1) = \mathrm{tail}(a_k)$ or $\mathrm{head}(a_1) = \mathrm{head}(a_k)$.
- *M-fences*: $k \geq 2$ is even, it is not a crown, and $\mathrm{tail}(a_i)$ is a tree vertex for every $i \in [k]$.
- *N-fences*: $k \geq 1$ is odd and $\mathrm{tail}(a_1)$ or $\mathrm{tail}(a_k)$, but not both, is a reticulation. By reordering the arcs, assume henceforth that $\mathrm{tail}(a_1)$ is a reticulation and $\mathrm{tail}(a_k)$ a tree vertex.
- *W-fences*: $k \geq 2$ is even and both $\mathrm{tail}(a_1)$ and $\mathrm{tail}(a_k)$ are reticulations.

We call a set S of maximal zig-zag trails a *zig-zag decomposition* of N if the elements of S partition all arcs, except for the root arc, of N.

▶ **Lemma 4** (adapted from Corollary 4.6 of [11]). *Let N be a network. Then N is tree-based if and only if it has no W-fences.*

▶ **Theorem 5** (adapted from Theorem 4.2 of [11]). *Any network N has a unique zig-zag decomposition.*

▶ **Theorem 6** (adapted from Theorem 3.3 of [20]). *Let N be a network. Then N is tree-based if and only if it has a cherry cover.*

3 Leaf Addition Proximity Measure

Let N be a network on X. *Adding a leaf $x \notin X$ to an arc e of N* is the process of adding a labelled vertex x, subdividing the arc e by a vertex w (if $e = uv$ then we delete the arc uv, add the vertex w, and add arcs uw and wv), and adding an arc wx. We denote the resulting network by $N + (e, x)$. When the arc e in the above is irrelevant or clear, we simply call this process *adding a leaf x to N*, and denote the resulting network by $N + x$.

In this section, $\mathcal{C}$ will be used to denote a network class. In particular, we shall use $\mathcal{TC}, \mathcal{OR}$, and $\mathcal{TB}$ to denote the classes of tree-child networks, orchard networks, and tree-based networks, respectively. Let $L_{\mathcal{C}}(N)$ denote the minimum number of leaf additions required to make the network N a member of $\mathcal{C}$. We first show that computing $L_{\mathcal{TC}}(N)$ and $L_{\mathcal{TB}}(N)$ can be done in polynomial time.

▶ **Lemma 7.** *Let N be a network. Then $L_{\mathcal{TC}}(N)$ is equal to the number of omnians. Moreover, N can be made tree-child by adding a leaf to exactly one outgoing arc of each omnian.*

▶ **Lemma 8.** *Let N be a network. Then $L_{\mathcal{TC}}(N)$ can be computed in $O(|N|)$ time.*

It has been shown already that $L_{\mathcal{TB}}(N)$ can be computed in $O(|N|^{3/2})$ time where $|N|$ is the number of vertices in N [7]. We show that this can in fact be computed in $O(|N|)$ time.

▶ **Lemma 9.** *Let N be a network. Then $L_{\mathcal{TB}}(N)$ is equal to the number of W-fences. Moreover, N is tree-based by adding a leaf to any arc in each W-fence in N.*

▶ **Lemma 10.** *Let N be a network. Then $L_{\mathcal{TB}}(N)$ can be computed in $O(|N|)$ time.*

Interestingly, computing $L_{\mathcal{OR}}(N)$ proves to be a difficult problem, although the leaf addition proximity measure is easy to compute for its neighbouring network classes. We prove the following in Section 4.

▶ **Theorem 17.** Let N be a network. Computing $L_{\mathcal{OR}}(N)$ is NP-hard.

We also include the following theorem which states that when considering leaf addition proximity measures for orchard networks, it suffices to consider leaf additions to reticulation arcs. We shall henceforth assume that all leaf additions are on reticulation arcs.

▶ **Theorem 11** (Theorem 4.1 of [17]). *A network N is orchard if and only if the network obtained by adding a leaf to a tree arc of N is orchard.*

The rest of the paper will now focus on the problem of computing $L_{\mathcal{OR}}(N)$.

4 Hardness Proof

In this section, we show that computing $L_{\mathcal{OR}}(N)$ is NP-hard by reducing from degree-3 vertex cover.

Degree-3 Vertex Cover (Decision)
Input: A 3-regular graph $G = (V, E)$ and a natural number k.
Decide: Does G have a vertex cover with at most k vertices?

$L_{\mathcal{OR}}$-Distance (Decision)
Input: A network N on a set of taxa X and a natural number k.
Decide: Can N be made orchard with at most k leaf additions?

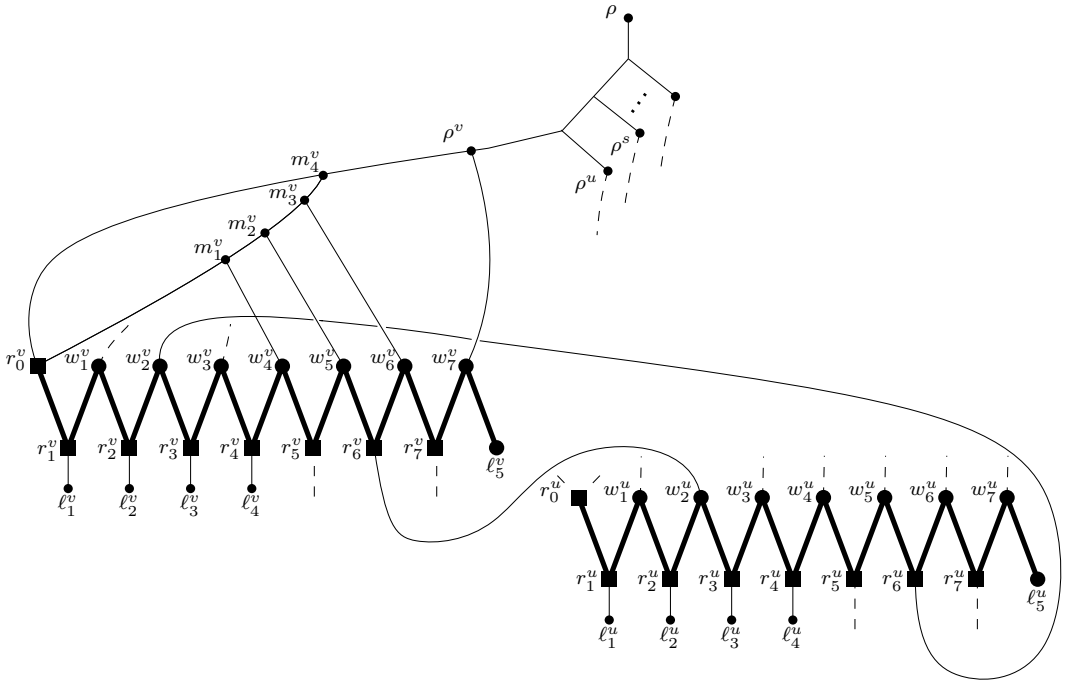


Figure 4 Sketch of the network N_G for the case when G contains an edge uv.

We now describe the reduction from Degree-3 Vertex Cover to $L_{\mathcal{OR}}$-Distance. For a graph G, let $V(G)$ and $E(G)$ be its vertex and edge sets, respectively. Given an instance (G, k) of Degree-3 Vertex Cover, construct an instance (N_G, k) of $L_{\mathcal{OR}}$-Distance as follows (see Figure 4):

1. For each vertex v in $V(G)$, construct a gadget Gad(v) as described below. In what follows, vertices of the form ℓ_i^v are leaves, vertices r_i^v are reticulations, and vertices w_i^v, m_i^v and ρ^v are tree vertices.
   The key structure in Gad(v) is an N-fence with 15 arcs, starting with the arc $\boldsymbol{r_0^v r_1^v}$, then followed by arcs of the form $\boldsymbol{w_i^v r_i^v, w_i^v r_{i+1}^v}$ for each $i \in [6]$, and finally the arcs $\boldsymbol{w_7^v r_7^v, w_7^v \ell_5^v}$. This set of arcs, in bold type, is called the *principal part* of Gad(v). In addition,the reticulations $r_1^v, r_2^v, r_3^v, r_4^v$ have leaf children $\ell_1^v, \ell_2^v, \ell_3^v, \ell_4^v$ respectively.
   Above the principal part of Gad(v), add a set of tree vertices $m_1^v, m_2^v, m_3^v, m_4^v, \rho^v$ with the following children: m_1^v has children r_0^v and w_4^v, m_2^v has children m_1^v and w_5^v, m_3^v has children m_2^v and w_6^v, m_4^v has children m_3^v and r_0^v, and ρ^v has children m_4^v and w_7^v (see Figure 4).
   This completes the construction of Gad(v). Note that so far, the vertices w_1^v, w_2^v, w_3^v have no incoming arcs, and r_5^v, r_6^v, r_7^v have no outgoing arcs. Such arcs will be added later to connect different gadgets together.
2. Connect the vertices ρ^v from each Gad(v) as follows: take some ordering of the vertices $\{v_1, \ldots, v_g\}$ of G. Add a vertex ρ and vertices s_i for $i \in [g-1]$. Add arcs ρs_1 and also arcs from the set $\{s_i s_{i+1} : i \in [g-2]\}$, as well as arcs from the set $\{s_i \rho^{v_i} : i \in [g-1]\}$, and finally an arc $s_{g-1}\rho^{v_g}$.
3. Next add arcs between the gadgets corresponding to adjacent vertices in G, in the following way: for every pair of adjacent vertices u, v in G, add an arc connecting one of the vertices r_5^u, r_6^u, r_7^u in Gad(u) to one of the vertices w_1^v, w_2^v, w_3^v in Gad(v) (and, symmetrically, an arc connecting one of r_5^v, r_6^v, r_7^v to one of w_1^u, w_2^u, w_3^u). The exact choice

of vertices connected by an arc does not matter, except that we should ensure each vertex is used by such an arc exactly once. Formally: for each vertex v in G with neighbours a, b, c, fix two (arbitrary) mappings $\pi_v : \{a, b, c\} \to \{1, 2, 3\}$ and $\tau_v : \{a, b, c\} \to \{5, 6, 7\}$. Then for each pair of adjacent vertices u, v in G, add an arc from $r^u_{\tau_u(v)}$ to $w^v_{\pi_v(u)}$ (and, symmetrically, add an arc from $r^v_{\tau_v(u)}$ to $w^u_{\pi_u(v)}$).

4. Finally, for each vertex v in G, label the vertices $\{\ell^v_i : i \in [5]\}$ in $\mathrm{Gad}(v)$ by ℓ^v_i.

Call the resulting graph N_G; it is easy to see that N_G is directed and acyclic with a single root ρ. Therefore it is a network on the leaf-set $\{\ell^v_i : i \in [5],\ v \in V(G)\}$. As the arcs of N_G are decomposed into M-fences and N-fences, we have the following observation.

▶ **Observation 12.** *Let G be a 3-regular graph and let N_G be the network obtained by the reduction. Then N_G is tree-based.*

By Observation 12 and Theorem 6, we use freely from now on that N_G has a cherry cover. Before proving the main result, we require some notation and helper lemmas. Let N be a network and let $\hat{N}_i$ be an N-fence of N. In what follows, we shall write $\hat{N}_i := (a^i_1, a^i_2, \ldots, a^i_{k_i})$, and we will let c^i_{2j-1} denote the child of $\mathrm{head}(a^i_{2j-1})$ for $j \in \left[\frac{k_i-1}{2}\right]$. The first lemma states that although a tree-based network may have non-unique cherry covers, the reticulated cherry shapes that cover arcs of N-fences are fixed.

▶ **Lemma 13.** *Let N be a tree-based network, and let $\hat{N}_1, \hat{N}_2, \ldots, \hat{N}_n$ denote the N-fences of N of length at least 3. Then every cherry cover of N contains the reticulated cherry shapes $\{(\mathrm{head}(a^i_{2j-1})c^i_{2j-1}), a^i_{2j}, a^i_{2j+1}\}$ for $i \in [n]$ and $j \in \left[\frac{k_i-1}{2}\right]$.*

Note that the principal part of a gadget $\mathrm{Gad}(v)$ for every $v \in V(G)$ is an N-fence. Let us denote the principal part of a gadget $\mathrm{Gad}(v)$ by $(a^v_1, a^v_2, \ldots, a^v_{15})$ for all $v \in V(G)$. By Lemma 13, a^v_i for $i = 2, \ldots, 15$ and $v \in E(G)$ are covered in the same manner across all possible cherry covers of N_G. Let us denote the reticulated cherry shape that contains a^v_i and a^v_{i+1} by $R^v_{i/2}$ for even $i \in [15]$. Figures 5a and 5b show an example of the part of cherry cover auxiliary graph containing R^v_i and R^u_i for $i \in [7]$, for some edge uv in G. Note that the cherry shapes form a cycle. The next lemma implies that in fact, such a cycle exists for any edge uv in G.

▶ **Lemma 14.** *Let N be a tree-based network and suppose that for two N-fences $\hat{N}_u := (a^u_1, a^u_2, \ldots, a^u_{k_u})$ and $\hat{N}_v := (a^v_1, a^v_2, \ldots, a^v_{k_v})$ of length at least 3, there exist directed paths in N from $\mathrm{head}(a^u_h)$ to $\mathrm{tail}(a^v_i)$ and from $\mathrm{head}(a^v_j)$ to $\mathrm{tail}(a^u_k)$, for even h, i, j, k with $k < h$ and $i < j$. Then every cherry cover auxiliary graph of N contains a cycle.*

In order to remove all possible cycles from a possible cherry cover, it is therefore necessary to disrupt the principal part of either $\mathrm{Gad}(u)$ or $\mathrm{Gad}(v)$, for any edge uv in G.

▶ **Lemma 15.** *Let G be a 3-regular graph and let N_G be the network obtained by the reduction above. Suppose that A is a set of arcs of N_G, for which adding leaves to every arc in A results in an orchard network. For every edge $uv \in E(G)$, there exists an arc $a \in A$ that is an arc of the principal part of $\mathrm{Gad}(u)$ or $\mathrm{Gad}(v)$.*

To complete the proof of the validity of the reduction, we show that in order to make N_G orchard by leaf additions, it is sufficient (and necessary) to add a leaf z^v to an appropriate arc of $\mathrm{Gad}(v)$ for every v in a vertex cover V_{sol} of G (see Figure 5c). The key idea is that this splits the principal part of $\mathrm{Gad}(v)$ from an N-fence into an N-fence and an M-fence, and this allows us to avoid the cycle in the cherry cover auxiliary graph (see Figure 5d).

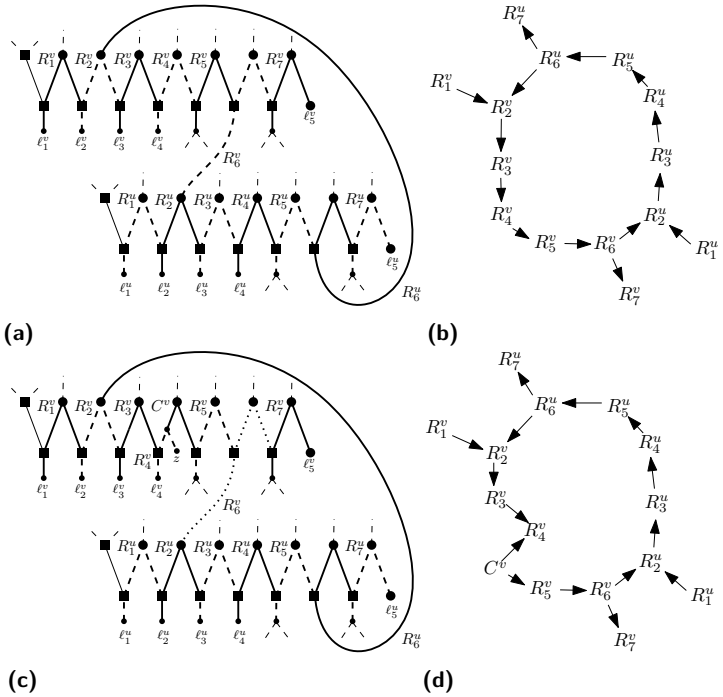


Figure 5 Cherry cover of Gad(v) and Gad(u). In (a), the unique cherry cover of the principal part of Gad(v) and Gad(u) is displayed, in (b), the cherry cover auxiliary graph of (a) is given. In (c), the leaf $z \notin X$ is added to the principal part of Gad(v), and one possible cherry cover of the same part of the network is given. And in (d), the cherry cover auxiliary graph of (c) is given.

▶ **Lemma 16.** *Let G be a 3-regular graph and let N_G be the network obtained by the reduction described above. Then G has a minimum vertex cover of size k if and only if $L_{\mathcal{OR}}(N_G) = k$.*

▶ **Theorem 17.** *Let N be a network. The decision problem $L_{\mathcal{OR}}$-Distance is NP-complete. Computing $L_{\mathcal{OR}}(N)$ is NP-hard.*

Proof. Suppose we are given a set of arcs A_{sol} of N_G of size at most k. Upon adding leaves to every arc in A_{sol}, we may check that the resulting network is orchard in polynomial time (see Section 6 of [14]). This implies that $L_{\mathcal{OR}}$-Distance is in NP. The reduction from Degree-3 Vertex Cover to $L_{\mathcal{OR}}$-Distance outlined at the start of the section takes polynomial time, since we add a constant number of vertices and arcs for every vertex in the Degree-3 Vertex Cover instance. The NP-completeness of $L_{\mathcal{OR}}$-Distance follows from the equivalence of the two problems shown by Lemma 16. The optimization problem of $L_{\mathcal{OR}}$-Distance, i.e., the one of computing $L_{\mathcal{OR}}(N)$ is therefore NP-hard. ◀

5 Upper Bound

In the previous section we showed that computing $L_{\mathcal{OR}}(N)$ is NP-hard. Here, we provide a sharp upper bound for $L_{\mathcal{OR}}(N)$. We call a reticulation *highest* if it has no reticulation ancestors.

▶ **Lemma 18.** *Let N be a network. Suppose there is a highest reticulation r such that all other reticulations have a leaf sibling. Then N is orchard.*

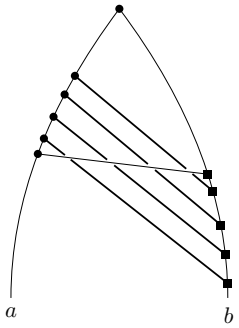


Figure 6 A network N on two leaves $\{a, b\}$ with $r(N) = 5$ reticulations. Observe that $L_{\mathcal{OR}}(N) = r(N) - 1 = 4$, since the highest reticulation cannot be reduced by cherry picking unless the reticulations below it are first reduced. For each non-highest reticulation, we must add a leaf to one of its incoming arcs to reduce it, which leads to $L_{\mathcal{OR}}(N) = 4$. Note that this construction can be extended for any k reticulations.

Proof. We prove the lemma by induction on the number of reticulations k. For the base case, observe that a network with one reticulation is tree-child since it has no omnians. A tree-child network is orchard [14], and so this network must be orchard.

Suppose now that we have proven the lemma for all networks with fewer than k reticulations, where $k > 1$. Let N be a network with reticulation set R where $|R| = k$, and suppose there exists a highest reticulation r in N such that all other reticulations have a leaf sibling. Let r denote the highest reticulation as specified in the statement of the lemma. Choose a lowest reticulation $r' \in R \setminus \{r\}$. By assumption, r' has a leaf sibling c. Every vertex below r' must be tree vertices and leaves. Reduce cherries until the child x of r' is a leaf. Then (x, c) is a reticulated cherry; the network N' obtained by reducing this reticulated cherry has $k - 1$ reticulations and has a highest reticulation r such that all other reticulations have a leaf sibling. By induction hypothesis, N' must be orchard. Since a sequence of cherry reductions can be applied to N to obtain N', the network N must also be orchard. ◀

▶ **Theorem 19.** *Let N be a network, and let $r(N)$ denote the number of reticulations. Then $L_{\mathcal{OR}}(N) = 0$ if N is a tree, and otherwise, $L_{\mathcal{OR}}(N) \leq r(N) - 1$, where the bound is sharp.*

Proof. If N is a tree, then it is orchard, and so $L_{\mathcal{OR}}(N) = 0$. So suppose $r(N) > 0$. Let r be a highest reticulation of N, and for every other reticulation, arbitrarily choose one incoming reticulation arc. Add a leaf to each of these reticulation arcs. By Lemma 18, the resulting network must be orchard. We have added a leaf for all but one reticulation in N. It follows that $L_{\mathcal{OR}}(N) \leq r(N) - 1$. The network in Figure 6 shows that this upper bound is sharp. ◀

6 MILP Formulation

To model the problem of computing the leaf addition proximity measure as a MILP, we reformulate the measure in terms of non-temporal labellings.

6.1 Vertical Arcs into Reticulations

By Theorem 2, every orchard network can be viewed as a network with a base tree where each of the linking arcs are horizontal. Recall that in terms of non-temporal labellings, this means that there exists a labelling wherein every reticulation has exactly one incoming reticulation arc that is horizontal. Following this definition, we introduce a second orchard proximity measure. Given a non-temporal labelling for a network N, let us use *inrets* to refer to reticulations of N with only vertical incoming arcs. Let $V_{\mathcal{OR}}(N)$ denote the minimum number of inrets over all possible non-temporal labellings.

▶ **Observation 20.** *Let N be a network. A network admits an HGT-consistent labelling if and only if $V_{\mathcal{OR}}(N) = 0$. In other words, a network is orchard if and only if $V_{\mathcal{OR}}(N) = 0$.*

In particular, we show a stronger result that equates the two proximity measures.

▶ **Lemma 21.** *Let N be a network. Then $L_{\mathcal{OR}}(N) = V_{\mathcal{OR}}(N)$.*

Proof. Suppose first that we have a network N with some non-temporal labelling $t : V(N) \to \mathbb{R}$ which gives rise to h inrets. For every inret r with parents u and v, we add a leaf x to the arc ur (this addition is done without loss of generality; the argument also follows by adding the leaf to vr). Since r is an inret, we must have $t(u) < t(r)$ and $t(v) < t(r)$. Letting p_x denote the parent of x, we label $t(p_x) := t(r)$ and $t(x) := t(p_x) + 1$. This ensures that the extension of the map t that includes x and p_x is a non-temporal labelling for $N + x$. Observe that r is no longer an inret in $N + x$, since the arc $p_x r$ is horizontal. Therefore, a leaf addition to an incoming arc of an inret can reduce the number of inrets by exactly one. By repeating this procedure for every inret, it follows that $L_{\mathcal{OR}}(N) \leq V_{\mathcal{OR}}(N)$.

To show the other direction, suppose we can add ℓ leaves to N to make it orchard. By Theorem 11, we may assume all such leaves are added to reticulation arcs in the set $\{e_1, \ldots, e_\ell\}$. The resulting network N' has an HGT-consistent labelling $t : V(N') \to \mathbb{R}$ by Theorem 2.

We claim that the labelling $t|_{V(N)}$ restricted to N is a non-temporal labelling, and that under $t|_{V(N)}$, the number of inrets is at most ℓ. Suppose that a leaf x_i was added to the reticulation arc $e_i = u_i r_i$. Let p_i denote the parent of x_i in the network N'. By definition of HGT-consistent labellings, we must have that $t(u_i) < t(r_i)$, since $u_i p_i r_i$ is a path in N'. Therefore, restricting the labelling to the network obtained from N' by removing the leaf x_i is non-temporal. Furthermore, if v_i is the parent of r_i that is not u_i, we have that one of $v_i r_i$ or $p_i r_i$ must be horizontal in N'. If $v_i r_i$ was horizontal, then r_i still has a horizontal incoming arc upon removing x_i, and the number of inrets does not change. On the other hand, if $p_i r_i$ was horizontal, then $v_i r_i$ must have been a vertical arc. Upon deleting x_i, the reticulation r_i becomes an inret as its other incoming arc $u_i r_i$ is also vertical. Since leaf deletions are local operations, deleting a leaf increases the number of inrets by at most one. By repeating this for each reticulation arc e_i for $i \in [\ell]$, it follows that N contains at most ℓ inrets, and therefore $V_{\mathcal{OR}}(N) \leq L_{\mathcal{OR}}(N)$. ◀

6.2 MILP Formulation

By Lemma 21 we have that $L_{\mathcal{OR}}(N) = V_{\mathcal{OR}}(N)$. In this section, we introduce a MILP formulation to obtain $V_{\mathcal{OR}}(N)$, and therefore also $L_{\mathcal{OR}}(N)$. This is done by searching for a non-temporal labelling of networks in which the number of vertical arcs is minimized.

Let N be a given network with vertex set V and arc set A. Let R denote the set of reticulations of N. We define the decision variable l_v to be the non-temporal label of the vertex $v \in V$. A tree arc and a vertical linking arc uv have the property that $l_u < l_v$. We

define x_a to be one if arc $a \in A$ is vertical and zero otherwise. We define h_v to be one if $v \in R$ is a reticulation with only incoming vertical arcs and zero otherwise. Let $v \in V$ be a vertex of N. In what follows, let $P_v \subset V$ be the set of parent nodes of v, $C_v \subset V$ the set of children nodes of v, and X the set of leaves. Let ρ be the root of N. Then, the MILP formulation is as follows:

$$\min_{x,h,l} \quad \sum_{v \in R} h_v$$

$$\text{s.t.} \quad \sum_{u \in P_v} x_{uv} - 1 \leq h_v \qquad \forall v \in R \tag{1}$$

$$\sum_{v \in C_u} x_{uv} \geq 1 \qquad \forall u \in V \setminus X \tag{2}$$

$$\sum_{u \in P_v} x_{uv} \geq 1 \qquad \forall v \in V \setminus \{\rho\} \tag{3}$$

$$l_u \leq l_v \qquad \forall uv \in A \tag{4}$$

$$l_u \leq l_v - 1 \qquad \forall v \in V \setminus R, \forall u \in P_v \tag{5}$$

$$l_u \leq l_v - 1 + |V|(1 - x_{uv}) \qquad \forall v \in R, \forall u \in P_v \tag{6}$$

$$l_u \geq l_v - |V| x_{uv} \qquad \forall v \in R, \forall u \in P_v \tag{7}$$

$$x_a \in \{0, 1\} \qquad \forall a \in A$$

$$h_v \in \{0, 1\} \qquad \forall v \in R$$

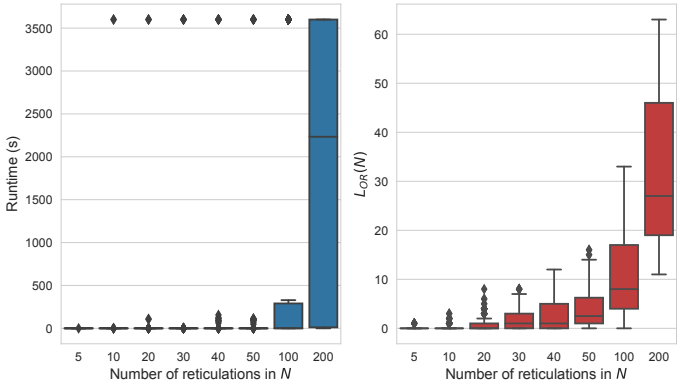
$$l_v \in \mathbb{R}_+ \qquad \forall v \in V$$

With constraint (1), h_v becomes one if all incoming arcs of reticulation v are vertical. With (2) we have that all vertices must have at least one outgoing vertical arc. Then, (3) guarantees that each reticulation has at least one incoming vertical arc. Constraint (4) creates the non-temporal labelling in the network, where with (5) the label of u is strictly smaller than that of v if v is not a reticulation. Then, (6) sets x_{uv} to one if uv is vertical, for all reticulation vertices v. Finally, with (7) the labels of u and v become equal if x_{uv} is zero.

6.3 Experimental Results

In this section, we apply the MILP described in the previous section to a set of real binary networks and to simulated networks, in order to assess the practical running time. The code for these experiments is written in Python and is available at `https://github.com/estherjulien/OrchardProximity`. All experiments ran on an Intel Core i7 CPU @ 1.8 GHz with 16 GB RAM. For solving the MILP problems, we use the open-source solver SCIP [2].

The real data set consists of different binary networks found in a number of papers, collected on `http://phylnet.univ-mlv.fr/recophync/networkDraw.php`. These networks have a leaf set of size up to 39 and a number of reticulations up to 9, with one outlier that has 32 reticulations. All the binary instances completed within one second (at most 0.072 seconds). Based on the results, we observe that only two out of the 22 binary networks have a value of $L_{\mathcal{OR}}(N) > 0$, thus, that only two are non-orchard. The most interesting of these is the network from [18] since its reticulations represent HGT highways. Even though each highway represents many gene transfers, it is still natural to expect these highways to be horizontal. However, our experimental results show that this network is not orchard (see Appendix A for mathematical arguments) and that its $L_{\mathcal{OR}}$ distance is 1. The most interesting part of this network is redrawn in Figure 1, where we also indicate a way to draw



(a)

R	Compl. (%)
5	100.0
10	99.2
20	98.8
30	98.0
40	97.2
50	97.6
100	75.6
200	61.6

(b)

Figure 7 Results of simulated networks. (a) Box plots for the runtime results (blue plot) and the $L_{\mathcal{OR}}(N)$ solutions (red plot) per number of reticulations of simulated networks N. The box plots are drawn with respect to the median of the runtime and the leaf addition score. (b) A table with the percentage of instances that were solved within the one-hour time limit. In the plots of (a), we also included instances that did not complete within the one-hour time limit. For these instances, we set their runtime to one hour.

it as a tree with horizontal linking arcs, after adding a single leaf. This added leaf represents a hypothesised missing taxon. In general, the $L_{\mathcal{OR}}(N)$ value gives a lower bound on the number of missing taxa that needs to be added to a network to make it HGT-consistent.

The simulated data is generated using the birth-hybridization network generator of [22], which can generate all binary network topologies [13]. Hence, even though it uses a model with hybridization to construct networks, it can also generate, for example, networks where reticulations represent HGT. This generator has two user-defined parameters: λ, which regulates the speciation rate, and ν, which regulates the hybridization rate. Following [13] we set $\lambda = 1$ and we sampled $\nu \in [0.0001, 0.4]$ uniformly at random. We generated an instance group of size 50 for each pair of values (L, R), with the number of leaves $L \in \{20, 50, 100, 150, 200\}$ and the number of reticulations $R \in \{5, 10, 20, 30, 40, 50, 100, 200\}$. In our implementation, we only defined variables x_a for incoming reticulation arcs. Therefore, the number of binary variables only depends on the number of reticulations in the network. In Figure 7a, the runtime and $L_{\mathcal{OR}}(N)$ value results for the simulated instances are shown against the reticulation number of the networks. The time limit was set to one hour. We can observe from these results that for networks with up to 50 reticulations, almost all instances are solved to optimality within a second. Then for $R = 100, 200$ the runtime increases, mainly because only 75.6% and 61.6% of the instances could be solved within the time limit, respectively (see Figure 7b). The completed instances are often still solved within reasonable time.

7 Discussion

In this paper we investigated the minimum number of leaf additions needed to make a network orchard, as a way to measure the extent to which a network deviates from being orchard. We showed that computing this measure is NP-hard (Theorem 17), and give a sharp upper bound by the number of reticulations minus one (Theorem 19). The measure was reformulated to

one in terms of minimizing the number of inrets over all possible non-temporal labellings. In Section 6 we use this reformulation to model the problem of computing the leaf addition measure as a MILP. Experimental results show that real-world data instances were solved within a second and the formulation worked well also over synthetic instances, being able to solve almost all instances up to 50 reticulations and 200 leaves within one second. For bigger instances the runtime however increased.

In this paper we have simulated networks using the network generator of [22] in order to analyse the running time of our MILP. Alternatively, one could simulate networks by generating orchard networks and deleting leaves from them. Since the leaf addition score is finite for any network by Theorem 19, it is possible to obtain any network by using this method. The leaf addition score gives a lower bound on the number of leaves that must be added to make the network orchard. The actual number of missing leaves could be larger, but this value cannot be estimated from the leaf-deleted network.

Of interest is how these results can potentially be used in practice. As mentioned in Section 1, one can consider a scenario in which it is suspected *a priori* that species under consideration evolve under a network in which all reticulate events are horizontal. An example of such scenarios can be seen for horizontal gene transfers, for instance when one considers the evolutionary history of species in bacteria [10] and fungi [18]. If a produced network does not admit an HGT-consistent labelling, there can in general be many reasons. For one, the output network may not be accurate. It is also possible that certain species have gone extinct, or that undersampling is present in the taxon set. In these latter two potential causes, our method gives a way of quantifying the minimum number of taxa that may have gone extinct / been undersampled. Moreover, it can be used to find all optimal corresponding orchard networks with added leaves. This could, for example, be used to try to identify the missing taxa.

Our NP-hardness result is interesting when comparing it to the computational complexity of the corresponding problem for different network classes. The problem of finding the minimum number of leaves to add to make a network tree-based can be solved in polynomial time [7] (Lemma 10) and we showed that the same is true for the class of tree-child networks (Lemma 8). Interestingly, the class of tree-child networks is contained in the class of orchard networks [14] which is in turn contained in the class of tree-based networks [12]. The reason for such an NP-hardness sandwich can perhaps be attributed to the lack of forbidden shapes. Leaf additions to obtain tree-child or tree-based networks target certain forbidden shapes in the network. In the case of tree-child networks, we add a leaf to exactly one outgoing arc of each omnian; for tree-based networks, we add a leaf to exactly one arc of each W-fence. The problem of finding a characterization of orchard networks in terms of (local) forbidden shapes has been elusive thus far [14] - perhaps the NP-hardness result for the orchard variant of the problem indicates that finding such a characterization for orchard networks may not be possible.

One can also consider the leaf addition problem for non-binary networks. Non-binary networks generalize the networks considered in this paper by allowing vertices to have varying indegrees and outdegrees. This generalized problem remains NP-complete since the binary version is a specific case. It could be interesting to try to find an MILP formulation for the nonbinary version.

Another natural research direction is to consider different proximity measures. One that may be of particular interest is a proximity measure based on arc deletions. That is, what is the minimum number of reticulate arc deletions needed to make a network orchard? Susanna showed that this measure is incomparable to the leaf addition proximity measure [17], yet it is not known if it is also NP-hard to compute.

References

1 Mihaela Baroni, Charles Semple, and Mike Steel. Hybrids in real time. *Systematic biology*, 55(1):46–56, 2006.

2 Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 8.0. Technical report, Optimization Online, December 2021. URL: `http://www.optimization-online.org/DB_HTML/2021/12/8728.html`.

3 Gabriel Cardona, Francesc Rosselló, and Gabriel Valiente. Comparison of tree-child phylogenetic networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 6(4):552–569, 2008.

4 Péter L Erdős, Charles Semple, and Mike Steel. A class of phylogenetic networks reconstructable from ancestral profiles. *Mathematical biosciences*, 313:33–40, 2019.

5 Mareike Fischer and Andrew Francis. How tree-based is my network? Proximity measures for unrooted phylogenetic networks. *Discrete Applied Mathematics*, 283:98–114, 2020.

6 Mareike Fischer, Tom Niklas Hamann, and Kristina Wicke. How far is my network from being edge-based? Proximity measures for edge-basedness of unrooted phylogenetic networks. *arXiv preprint*, 2022. `arXiv:2207.01370`.

7 Andrew Francis, Charles Semple, and Mike Steel. New characterisations of tree-based networks and proximity measures. *Advances in Applied Mathematics*, 93:93–107, 2018.

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

9 Benjamin E Goulet, Federico Roda, and Robin Hopkins. Hybridization in plants: old ideas, new techniques. *Plant physiology*, 173(1):65–78, 2017.

10 Carlton Gyles and Patrick Boerlin. Horizontally transferred genetic elements and their role in pathogenesis of bacterial disease. *Veterinary pathology*, 51(2):328–340, 2014.

11 Momoko Hayamizu. A structure theorem for rooted binary phylogenetic networks and its implications for tree-based networks. *SIAM Journal on Discrete Mathematics*, 35(4):2490–2516, 2021.

12 Katharina T Huber, Leo van Iersel, Remie Janssen, Mark Jones, Vincent Moulton, Yukihiro Murakami, and Charles Semple. Orienting undirected phylogenetic networks. *arXiv preprint*, 2019. `arXiv:1906.07430`.

13 Remie Janssen and Pengyu Liu. Comparing the topology of phylogenetic network generators. *Journal of Bioinformatics and Computational Biology*, 19(06):2140012, 2021.

14 Remie Janssen and Yukihiro Murakami. On cherry-picking and network containment. *Theoretical Computer Science*, 856:121–150, 2021.

15 Laura Jetten and Leo van Iersel. Nonbinary tree-based phylogenetic networks. *IEEE/ACM transactions on computational biology and bioinformatics*, 15(1):205–217, 2016.

16 Fabio Pardi and Celine Scornavacca. Reconstructible phylogenetic networks: do not distinguish the indistinguishable. *PLoS computational biology*, 11(4):e1004135, 2015.

17 Merel Susanna. Making phylogenetic networks orchard: Algorithms to determine if a phylogenetic network is orchard and to transform non-orchard to orchard networks. Bachelor's thesis, Delft University of Technology, 2022. `http://resolver.tudelft.nl/uuid:724ac2af-e569-4586-b367-288fef890252`.

18 Gergely J Szöllősi, Adrián Arellano Davín, Eric Tannier, Vincent Daubin, and Bastien Boussau. Genome-scale phylogenetic analysis finds extensive gene transfer among fungi. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 370(1678):20140335, 2015.

19 Leo van Iersel, Remie Janssen, Mark Jones, and Yukihiro Murakami. Orchard networks are trees with additional horizontal arcs. *Bulletin of Mathematical Biology*, 84(8):1–21, 2022.

20 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.

21 David A Wickell and Fay-Wei Li. On the evolutionary significance of horizontal gene transfers in plants. *New Phytologist*, 225(1):113–117, 2020.

22 Chi Zhang, Huw A Ogilvie, Alexei J Drummond, and Tanja Stadler. Bayesian inference of species networks from multilocus sequence data. *Molecular biology and evolution*, 35(2):504–517, 2018.

23 Louxin Zhang. On tree-based phylogenetic networks. *Journal of Computational Biology*, 23(7):553–565, 2016.

A Appendix

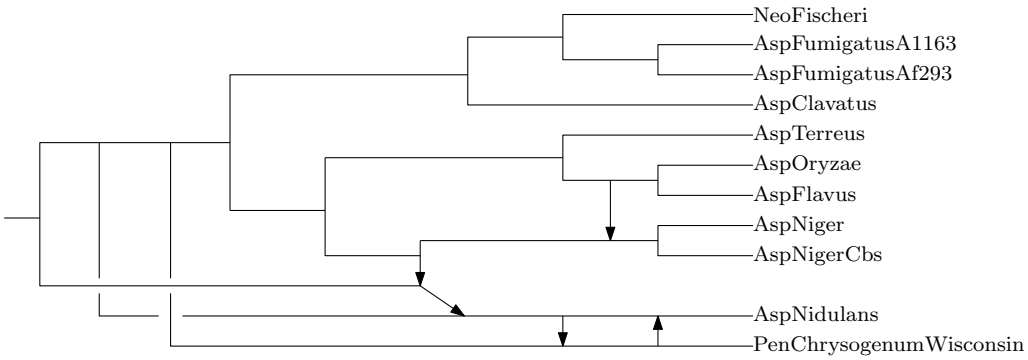


Figure 8 The network of Figure 1 without the added leaf. Observe that there exists no HGT-consistent labelling for the network, by the arguments provided in Remark 22.

▶ Remark 22. We first elaborate on why we need an added leaf (*unsampled taxon*) in the network of Figure 1 to ensure that the network admits an HGT-consistent labelling. We know that a network has an HGT-consistent labelling if and only if it is orchard (Theorem 2). Let N be the network without *unsampled taxon* (see Figure 8). We will show that N is not orchard. To see this, note that the order in which cherries and reticulated cherries are reduced does not matter [14]. This means that if N were orchard, then there would exist a cherry picking sequence starting with

$$(AspNidulans, PenChrysogenumWisconsin)(PenChrysogenumWisconsin, AspNidulans).$$

After reducing these cherries, the distance between the leaf $AspNidulans$ and any other leaf remains of distance at least 4, regardless of other reductions that take place in the network. This shows that the network cannot be orchard, and therefore the network cannot have an HGT-consistent labelling.

▶ **Lemma 7.** *Let N be a network. Then $L_{\mathcal{TC}}(N)$ is equal to the number of omnians. Moreover, N can be made tree-child by adding a leaf to exactly one outgoing arc of each omnian.*

Proof. By definition, a network is tree-child if and only if it contains no omnians. We show that every leaf addition can result in a network with one omnian fewer than that of the original network. Let uv be an arc where u is an omnian. Add a leaf x to uv. In the

resulting network, u has a child (the parent of x) that is a tree vertex, and it is no longer an omnian. The newly added tree vertex has a leaf child x; the parent-child combinations remain unchanged for the rest of the network, so at most one omnian (in this case u) can be removed per leaf addition. It follows that $L_{\mathcal{TC}}(N)$ is at least the number of omnians in N. By targeting arcs with omnian tails, we can remove at least one omnian per every leaf addition, so that $L_{\mathcal{TC}}(N)$ is at most the number of omnians in N. Therefore, $L_{\mathcal{TC}}(N)$ is exactly the number of omnians in N. ◀

▶ **Lemma 8.** *Let N be a network. Then $L_{\mathcal{TC}}(N)$ can be computed in $O(|N|)$ time.*

Proof. We first show that the number of omnians of N can be computed in $O(|N|)$ time, by checking, for each vertex, the indegrees of its children. A vertex is an omnian if and only if all of its children are of indegree-2. Since the degree of every vertex is at most 3, each search within the for loop takes constant time. The for loop iterates over the vertex set which is of size $O(N)$. By Lemma 7, since $L_{\mathcal{TC}}(N)$ is the number of omnians in N, we can compute $L_{\mathcal{TC}}(N)$ in $O(|N|)$ time. ◀

It has been shown already that $L_{\mathcal{TB}}(N)$ can be computed in $O(|N|^{3/2})$ time where $|N|$ is the number of vertices in N [7]. We show that this can in fact be computed in $O(|N|)$ time.

▶ **Lemma 9.** *Let N be a network. Then $L_{\mathcal{TB}}(N)$ is equal to the number of W-fences. Moreover, N can be made tree-based by adding a leaf to any arc in each W-fence in N.*

Proof. By Lemma 4, a network is tree-based if and only if it contains no W-fences. We show that every leaf addition can result in a network with one W-fence fewer than that of the original network. Suppose that N contains at least one W-fence. Otherwise we may conclude that the network is tree-based by Lemma 4. Let $(a_1, a_2, \ldots, a_k)$ be a W-fence in N where $a_i = u_i v_i$ for $i \in [k]$, and add a leaf x to a_1; let p_x be the tree vertex parent of x. In the resulting network, the arcs in $\{u_1p_x, p_xv_1, p_xx, a_2, a_3, a_4, \ldots, a_k\}$ are decomposed into their unique maximal zig-zag trails (Theorem 5) as two N-fences (u_1p_x) and $(a_k, a_{k-1}, \ldots, a_3, a_2, p_xv_1, p_xx)$. All other arcs remain in the same maximal zig-zag trails as that of N. Therefore the number of W-fences has gone down by exactly one. This can be repeated for all W-fences in the network; it follows that $L_{\mathcal{TB}}(N)$ is the number of W-fences in N.

A quick check shows that adding a leaf to any arc in the W-fence decomposes the W-fence into two N-fences. ◀

▶ **Lemma 10.** *Let N be a network. Then $L_{\mathcal{TB}}(N)$ can be computed in $O(|N|)$ time.*

Proof. Finding the maximal zig-zag decomposition takes $O(|N|)$ time (Proposition 5.1 of [11]). Counting the number of W-fences in the decomposition gives $L_{\mathcal{TB}}(N)$ by Lemma 9. ◀

▶ **Observation 12.** *Let G be a 3-regular graph and let N_G be the network obtained by the reduction. Then N_G is tree-based.*

Proof. It is easy to check that the arcs of N_G are decomposed into M-fences and N-fences (the principal part of each gadget $\mathrm{Gad}(v)$ is an N-fence; each arc leaving the principal part of a gadget $\mathrm{Gad}(v)$ is an N-fence of length 1; the remaining arcs decompose into M-fences of length 2). By Lemma 4, N_G must be tree-based. ◀

▶ **Lemma 13.** *Let N be a tree-based network, and let $\hat{N}_1, \hat{N}_2, \ldots, \hat{N}_n$ denote the N-fences of N of length at least 3. Then every cherry cover of N contains the reticulated cherry shapes $\{(\mathrm{head}(a^i_{2j-1})c^i_{2j-1}), a^i_{2j}, a^i_{2j+1}\}$ for $i \in [n]$ and $j \in \left[\frac{k_i-1}{2}\right]$.*

Proof. Let $\hat{N}_i = (a^i_1, a^i_2, \ldots, a^i_{k_i})$ be an N-fence of length $k_i \geq 3$. Observe that in every cherry cover, exactly one incoming arc of every reticulation is covered by a reticulated cherry shape as a middle arc (since the network is binary; for non-binary networks, this is not true in general [20]). Since $\text{head}(a^i_1)$ is a reticulation, one of a^i_1 or a^i_2 must be in a reticulated cherry shape as a middle arc. But $\text{tail}(a^i_1)$ is a reticulation; therefore, a^i_2 must be in a middle arc of a reticulated cherry shape. The other two arcs of the same reticulated cherry shapes are then fixed to be $\text{head}(a^i_1)c^i_1$ and a^i_3. Repeating this argument for the reticulations $\text{head}(a^i_{2j+1})$ for $j \in \left[\frac{k_i-1}{2}\right]$ gives the required claim for the N-fence $\hat{N}_i$; further repeating this argument for every N-fence gives the required claim. ◀

▶ **Lemma 14.** *Let N be a tree-based network and suppose that for two N-fences $\hat{N}_u := (a^u_1, a^u_2, \ldots, a^u_{k_u})$ and $\hat{N}_v := (a^v_1, a^v_2, \ldots, a^v_{k_v})$ of length at least 3, there exist directed paths in N from $\text{head}(a^u_h)$ to $\text{tail}(a^v_i)$ and from $\text{head}(a^v_j)$ to $\text{tail}(a^u_k)$, for even h, i, j, k with $k < h$ and $i < j$. Then every cherry cover auxiliary graph of N contains a cycle.*

Proof. Let us again denote the reticulated cherry shape that contains a^u_h and a^u_{h+1} by $R^u_{h/2}$, and similarly for $R^v_{i/2}$, $R^v_{j/2}$,and $R^u_{k/2}$. By Lemma 13, all of $R^u_{h/2}$, $R^v_{i/2}$, $R^v_{j/2}$, $R^u_{k/2}$ appear in the cherry cover auxiliary graph. Moreover $R^u_{k/2}$ is above $R^u_{h/2}$, and $R^v_{i/2}$ is above $R^v_{j/2}$. Now observe that for any consecutive arcs on the path from $\text{head}(a^u_h)$ to $\text{tail}(a^v_i)$, either they are part of the same reticulated cherry shape in the cherry cover, or they are part of different cherry shapes with one cherry shape directly above the other. This implies that there is a path from $R^u_{h/2}$ to $R^v_{i/2}$ in the cherry cover auxiliary graph. A similar argument shows that there is a path from $R^v_{j/2}$ to $R^u_{k/2}$. But then we have that $R^u_{h/2}$ is above $R^v_{i/2}$, which is above $R^v_{j/2}$, which is above $R^u_{k/2}$, which is above $R^u_{h/2}$ and we have a cycle. ◀

▶ **Lemma 15.** *Let G be a 3-regular graph and let N_G be the network obtained by the reduction above. Suppose that A is a set of arcs of N_G, for which adding leaves to every arc in A results in an orchard network. For every edge $uv \in E(G)$, there exists an arc $a \in A$ that is an arc of the principal part of $\text{Gad}(u)$ or $\text{Gad}(v)$.*

Proof. We prove this lemma by contraposition. Let us assume that there is an edge $uv \in E(G)$, such that no arcs of the principal part of $\text{Gad}(u)$ or $\text{Gad}(v)$ are in A. We shall show that the network obtained by adding leaves to all $a \in A$ in N_G – which we denote $N_G + A$ – is not orchard.

From Theorem 1 we know that N_G is orchard if and only if N_G has an acyclic cherry cover. We show here that $N_G + A$ will not have an acyclic cherry cover, thereby showing that $N_G + A$ is not orchard.

As no arcs were added to the principal part of $\text{Gad}(u)$ or $\text{Gad}(v)$, these principal parts remain N-fences in $N_G + A$. Furthermore by construction N has an arc from some $\text{head}(a^v_h)$ to $\text{tail}(a^u_i)$ for even $h \geq 10$ and even $i \leq 6$, and so $N_G + A$ has a path from $\text{head}(a^v_h)$ to $\text{tail}(a^u_i)$. Similarly $N_G + A$ has a path from $\text{head}(a^u_j)$ to $\text{tail}(a^v_k)$ for some even $j \geq 10$ and $h \leq 6$. Then Lemma 14 implies that the auxiliary graph of any cherry cover of $N_G + A$ contains a cycle. By Theorem 1, we have that $N_G + A$ is not orchard. ◀

▶ **Lemma 16.** *Let G be a 3-regular graph and let N_G be the network obtained by the reduction described above. Then G has a minimum vertex cover of size k if and only if $L_{\mathcal{OR}}(N_G) = k$.*

Proof. Suppose first that V_{sol} is a vertex cover of G with at most k vertices. We shall show that adding a leaf to an arc of the principal part of each $\text{Gad}(v)$ for $v \in V_{sol}$ makes N_G orchard. This will show that the minimum vertex cover of G is at least $L_{\mathcal{OR}}(N_G)$. In the remainder of this proof, we will refer to vertices and arcs of N_G as introduced above in the reduction.

For every $v \in V_{sol}$, we add a leaf z^v to the arc $w_4^v r_4^v$ of Gad(v) (see Figure 5c). Let q^v be the parent of z^v. The key idea is that this splits the principal part of Gad(v) from an N-fence into an N-fence and an M-fence, and this allows us to avoid the cycle in the cherry cover auxiliary graph (see Figure 5d).

Let us call the new network M. To formally show that M is orchard, we give an HGT-consistent labelling $t : V(M) \to \mathbb{R}$.

Begin by setting $t(\rho) = 0$, and for any vertex in $s_1, \ldots, s_{g-1}$ or $\rho^v, m_4^v, \ldots, m_2^v$ for any v in $V(G)$, let this vertex have label equal to the label of its parent plus 1. Let h be the maximum value assigned to a vertex so far, and now adjust t by subtracting $(h+1)$ from each label. Thus, we may now assume that all vertices in $\rho, s_1, \ldots, s_{g-1}$ or $m_4^v, \ldots, m_2^v$ for any v in $V(G)$ have label ≤ -1. Now set $t(m_1^v) = 0$ and $t(r_0^v) = 0$, for each v in $V(G)$.

It is easy to see that so far t satisfies the properties of an HGT-consistent labelling. It remains to label the vertices in the principal part of each gadget Gad(v), and the leaves of each gadget, and the new vertices q^v and z^v for $v \in V_{sol}$. We do this as follows.

For $v \in V_{sol}$, set $t(r_1^v) = t(w_1^v) = 12, t(r_2^v) = t(w_2^v) = 13, t(r_3^v) = t(w_3^v) = 14$, and $t(r_4^v) = t(q^v) = 15$. Set $t(w_4^v) = 1, t(r_5^v) = t(w_5^v) = 2, t(r_6^v) = t(w_6^v) = 3$, and $t(r_7^v) = t(w_7^v) = 4$.

For $v \notin V_{sol}$, set $t(r_1^v) = t(w_1^v) = 5$, and $t(r_i^v) = t(w_i^v) = i + 4$ for every i up to $t(r_7^v) = t(w_7^i) = 11$.

Finally, for each leaf ℓ with parent p set $t(\ell) = t(p) + 1$.

It remains to observe that t is a non-temporal labelling of M and for every reticulation r in M, r has exactly one parent p with $t(p) = t(r)$. Thus t is an HGT-consistent labelling of M, and it follows from Theorem 2 that M is orchard.

Suppose now that we have a set of arcs A_{sol} of N_G of size at most k, such that adding leaves to the arcs in A_{sol} makes N_G orchard. By Lemma 15, for every edge $uv \in E(G)$, there exists an arc $a \in A_{sol}$ that is an arc of the principal part of Gad(u) or Gad(v). It follows immediately that the set $\{v \in V(G) : A_{sol}$ contains an arc of the principal part of Gad(v)$\}$ is a vertex cover of G. Since this is true for any such set of arcs A_{sol}, it follows that if there is such an A_{sol} of size at most k, then there must exist a vertex cover of G of size at most k. ◀

Quartets Enable Statistically Consistent Estimation of Cell Lineage Trees Under an Unbiased Error and Missingness Model

Yunheng Han
Department of Computer Science, University of Maryland, College Park, MD, USA

Erin K. Molloy[1]
Department of Computer Science, University of Maryland, College Park, MD, USA

Abstract

Cancer progression and treatment can be informed by reconstructing its evolutionary history from tumor cells [5]. Although many methods exist to estimate evolutionary trees (called phylogenies) from molecular sequences, traditional approaches assume the input data are error-free and the output tree is fully resolved. These assumptions are challenged in tumor phylogenetics because single-cell sequencing produces sparse, error-ridden data and because tumors evolve clonally [3, 12]. Here, we study the theoretical utility of methods based on quartets (four-leaf, unrooted phylogenetic trees) and triplets (three-leaf, rooted phylogenetic trees), in light of these barriers.

Quartets and triplets have long been used as the building blocks for reconstructing the evolutionary history of species [14]. The reason triplet-based methods (e.g., MP-EST [6]) and quartet-based methods (e.g., ASTRAL [7]) have garnered such success in species phylogenetics is their good statistical properties under the Multi-Species Coalescent (MSC) model [9, 10] (see [1] and [2] for identifiability results under the MSC model for quartets and triplets, respectively).

Inspired by these efforts, we study the utility of quartets and triplets for estimating cell lineage trees under a popular tumor phylogenetics model [3, 11, 15, 4] with two phases. First, mutations arise on a (highly unresolved) cell lineage tree according to the infinite sites model, and second, errors (false positives and false negatives) and missing values are introduced to the resulting mutation data in an unbiased fashion, mimicking data produced by single-cell sequencing protocols. This infinite sites plus unbiased error and missingness (IS+UEM) model generates mutations (rather than gene genealogies like the MSC model). However, a quartet (with leaves bijectively labeled by four cells) is implied by a mutation being present in two cells and absent from two cells [8, 13]; similarly, a triplet (on three cells) is implied by a mutation being present in two cells and absent from one cell.

Our main result is that under the IS+UEM, the most probable quartet identifies the unrooted model cell lineage tree on four cells, with a mild assumption: the probability of false negatives and the probability of false positives must not sum to one. Somewhat surprisingly, our identifiability result for quartets does not extend to triplets, with more restrictive assumptions being required for identifiability. These results motivate seeking an unrooted cell lineage tree such that the number of quartets shared between it and the input mutations is maximized. We prove an optimal solution to this problem is a consistent estimator of the unrooted cell lineage tree under the IS+UEM model; this guarantee includes the case where the model tree is highly unresolved, provided that tree error is defined as the number of false negative branches. We therefore conclude by outlining how quartet-based methods might be employed for tumor phylogenetics given other important challenges like copy number aberrations and doublets.

2012 ACM Subject Classification Applied computing → Molecular evolution

Keywords and phrases Tumor Phylogenetics, Cell Lineage Trees, Quartets, Supertrees, ASTRAL

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.8

Category Abstract

[1] Corresponding author

Related Version *Full Version*: https://doi.org/10.1101/2023.04.04.535437

Funding This work was financially supported by the State of Maryland.

Acknowledgements The authors thank Michael Nute for very helpful feedback on a preliminary version of this paper, especially our notation.

References

1 Elizabeth S. Allman, James H. Degnan, and John A. Rhodes. Identifying the rooted species tree from the distribution of unrooted gene trees under the coalescent. *Journal of Mathematical Biology*, 62(6):833–862, 2011. doi:10.1007/s00285-010-0355-7.

2 James H. Degnan and Noah A. Rosenberg. Discordance of species trees with their most likely gene trees. *PLOS Genetics*, 2(5):1–7, May 2006. doi:10.1371/journal.pgen.0020068.

3 Katharina Jahn, Jack Kuipers, and Niko Beerenwinkel. Tree inference for single-cell data. *Genome Biology*, 17:86, 2016. doi:10.1186/s13059-016-0936-x.

4 Can Kizilkale, Farid Rashidi Mehrabadi, Erfan Sadeqi Azer, Eva Pérez-Guijarro, Kerrie L. Marie, Maxwell P. Lee, Chi-Ping Day, Glenn Merlino, Funda Ergün, Aydın Buluç, S. Cenk Sahinalp, and Salem Malikić. Fast intratumor heterogeneity inference from single-cell sequencing data. *Nature Computational Science*, 2:577–583, 2022. doi:10.1038/s43588-022-00298-x.

5 Bora Lim, Yiyun Lin, and Nicholas Navin. Advancing cancer research and medicine with single-cell genomics. *Cancer Cell*, 37(4):456–470, 2020. doi:10.1016/j.ccell.2020.03.008.

6 Liang Liu, Lili Yu, and Scott V. Edwards. A maximum pseudo-likelihood approach for estimating species trees under the coalescent model. *BMC Evolutionary Biology*, 10:302, 2010. doi:10.1186/1471-2148-10-302.

7 Siavash Mirarab, Rezwana Reaz, Md. Shamsuzzoha Bayzid, Theo Zimmermann, Michelle S. Swenson, and Tandy Warnow. ASTRAL: genome-scale coalescent-based species tree estimation. *Bioinformatics*, 30(17):i541–i548, 2014. doi:10.1093/bioinformatics/btu462.

8 Erin K. Molloy, John Gatesy, and Mark S. Springer. Theoretical and practical considerations when using retroelement insertions to estimate species trees in the anomaly zone. *Systematic Biology*, 71(3):721–740, 2021. doi:10.1093/sysbio/syab086.

9 Pekka Pamilo and Masatoshi Nei. Relationships between gene trees and species trees. *Molecular Biology and Evolution*, 5(5):568–583, 1988. doi:10.1093/oxfordjournals.molbev.a040517.

10 Bruce Rannala and Ziheng Yang. Bayes estimation of species divergence times and ancestral population sizes using DNA sequences from multiple loci. *Genetics*, 164(4):1645–1656, 2003. doi:10.1093/genetics/164.4.1645.

11 Edith M. Ross and Florian Markowetz. OncoNEM: inferring tumor evolution from single-cell sequencing data. *Genome Biology*, 17(1):69, 2016. doi:10.1186/s13059-016-0929-9.

12 Russell Schwartz and Alejandro A Schäffer. The evolution of tumour phylogenetics: principles and practice. *Nature Reviews Genetics*, 18(4):213–229, 2017. doi:10.1038/nrg.2016.170.

13 Mark S. Springer, Erin K. Molloy, Daniel B. Sloan, Mark P. Simmons, and John Gatesy. ILS-aware analysis of low-homoplasy retroelement insertions: Inference of species trees and introgression using quartets. *Journal of Heredity*, 111(2):147–168, 2019. doi:10.1093/jhered/esz076.

14 Mark Wilkinson, James A. Cotton, Chris Creevey, Oliver Eulenstein, Simon R. Harris, Francois-Joseph Lapointe, Claudine Levasseur, James O. Mcinerney, Davide Pisani, and Joseph L. Thorley. The Shape of Supertrees to Come: Tree Shape Related Properties of Fourteen Supertree Methods. *Systematic Biology*, 54(3):419–431, 2005. doi:10.1080/10635150590949832.

15 Yufeng Wu. Accurate and efficient cell lineage tree inference from noisy single cell data: the maximum likelihood perfect phylogeny approach. *Bioinformatics*, 36(3):742–750, August 2019. doi:10.1093/bioinformatics/btz676.

Inferring Temporally Consistent Migration Histories

Mrinmoy Saha Roddur
Department of Computer Science, University of Illinois at Urbana-Champaign, IL, USA

Sagi Snir
Department of Evolutionary Biology, University of Haifa, Israel

Mohammed El-Kebir
Department of Computer Science, University of Illinois at Urbana-Champaign, IL, USA
Cancer Center at Illinois, University of Illinois at Urbana-Champaign, IL, USA

Abstract

Not only do many biological populations undergo evolution, but population members may also migrate from one location to another. For example, tumor cells may migrate from the primary tumor and seed a new metastasis, and pathogens may migrate from one host to another. One may represent a population's migration history by labeling the vertices of a given phylogeny T with locations such that an edge incident to vertices with distinct locations represents a migration. Additionally, in some biological populations, taxa from distinct lineages may comigrate from one location to another in a single event, a phenomenon known as a comigration. Here, we show that a previous problem statement for inferring migration histories that are parsimonious in terms of migrations and comigrations may lead to temporally inconsistent solutions. To remedy this deficiency, we introduce precise definitions of temporal consistency of comigrations in a phylogeny, leading to three successive problems. First, we formulate the Temporally Consistent Comigrations (TCC) problem to check if a set of comigrations is temporally consistent and provide a linear time algorithm for solving this problem. Second, we formulate the Parsimonious Consistent Comigration (PCC) problem, which aims to find comigrations given a location labeling of a phylogeny. We show that PCC is NP-hard. Third, we formulate the Parsimonious Consistent Comigration History (PCCH) problem, which infers the migration history given a phylogeny and locations of its extant vertices only. We show that PCCH is NP-hard as well. On the positive side, we propose integer linear programming models to solve the PCC and PCCH problems. We apply our approach to real and simulated data.

2012 ACM Subject Classification Applied computing → Computational biology

Keywords and phrases Metastasis, Migration, Integer Linear Programming, Maximum parsimony

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.9

Supplementary Material *Software*: `https://github.com/elkebir-group/PCCH`
archived at `swh:1:dir:f563e890bf28a3f64b5d29b7089358469025fc8c`

Funding *Sagi Snir*: Israel Science Foundation (grant no. ISF 1927/21) and the American/Israeli Binational Science Foundation (grant no. BSF 2021139).
Mohammed El-Kebir: National Science Foundation award number CCF 2046488 as well as funding from the Cancer Center at Illinois.

Acknowledgements This project started as a collaboration at the Computational Genomics Summer Institute 2022.

1 Introduction

Throughout history, various biological populations, ranging from cells and microorganisms to large mammals, have migrated from one place to another. The study of these migrations holds significant importance in various areas of biology and medical science. For instance, understanding the migration history of metastatic cancer can provide insights into the

mechanism of metastasis and aid in the development of novel drugs [7, 11, 12, 26, 31, 33]. Similarly, investigating the transmission of pathogens can help in identifying the source of an outbreak and tracing the patterns of disease spread [3, 10, 13, 14, 32]. Analyzing genomic data is a potential approach to tracing the migration history of a biological population since the migrated subpopulations evolve independently of their counterparts, leading to location-specific genomic differences. One way to do this is by first constructing a rooted phylogeny T, where each vertex v corresponds to a subpopulation with similar genetic makeup, and then labeling each vertex v with their location of origin $\ell(v)$. Therefore each edge (u, v) with distinct labels at its endpoints i.e. $\ell(u) \neq \ell(v)$ corresponds to a migration of subpopulation u from location $\ell(u)$ to location $\ell(v)$ and further evolution to subpopulation v. One challenge with this approach is that although it is possible to know the location of extant subpopulations, the location of extinct subpopulations cannot accurately be known, and so labeling internal vertices of phylogeny T is nontrivial. One parsimony based approach proposed by Slatkin and Madisson [29] to infer internal vertex labeling is to select the labeling that minimizes the number of migrations. Later McPherson et al. [22] used this approach to infer the migration history of cancer cells in metastatic ovarian cancer.

In many evolutionary processes, multiple migrations between the same pair of locations may occur as part of a single event. For instance, in cancer cells from distinct clones may co-migrate as part of a single cluster [1, 2, 5, 6, 8, 11, 17, 20, 21, 35, 36]. Similarly, many pathogens are subject to a weak transmission bottleneck, where multiple variants of the same pathogen are co-transmitted in a single event [27, 28, 30]. MACHINA [11] was the first method to incorporate comigrations in the analysis of metastatic cancer. Specifically, MACHINA defined a comigration to be a set of migrations between the same pair of locations but occurring on different lineages of the tree. In other words, two migrations (u, v) and (u', v') from different lineages belong to the same comigration if $\ell(u) = \ell(u')$ and $\ell(v) = \ell(v')$. Based on this definition, MACHINA extended Slatkin and Madisson [29]'s approach by selecting the location labeling that minimizes the number of migrations, and subsequently the number of comigrations. Following this, another method SharpTNI [27] to infer transmission history was published which uses a similar notion of comigration. One key issue is that this definition of comigration does not adequately capture temporal dependencies between migrations. Time flows from root to leaves in a phylogeny, so if a migration (u, v) occurs before (u', v') in the tree, then all migrations in the comigration containing (u, v) should occur before all migrations in the comigration containing (u', v'). However, the above comigration definition does not enforce this criterion, potentially resulting in temporally inconsistent solutions. In species phylogenetics, similar temporal restrictions arise with lateral gene transfers. Specifically, since gene transfer occurs in co-existing entities, if a transfer occurs from some species A to species B in a species tree, there cannot be another transfer from an ancestor of A to a descendant of B. The temporal consistency of lateral gene transfers has been addressed in studies involving gene tree reconciliation [9, 19, 23, 24, 34], species tree ranking [4], and species tree inference [18].

In this work, we introduce a comigration model that accurately captures both spatial and temporal aspects of simultaneous migrations. To that end, we formulate three new problems. Our first problem, the Temporally Consistent Comigration (TCC) problem aims to assign timestamps to migrations such that migrations in the same comigration have the same timestamp and timestamps are monotonically increasing along the edges of any root-to-leaf path of the tree (Figure 1a). We introduce a linear time algorithm to check if a given set of comigrations is temporally consistent. Our second problem is the Parsimonious Consistent Comigrations (PCC) where, given a rooted tree with locations assigned to all vertices, we seek a minimum set of spatially and temporally consistent comigrations

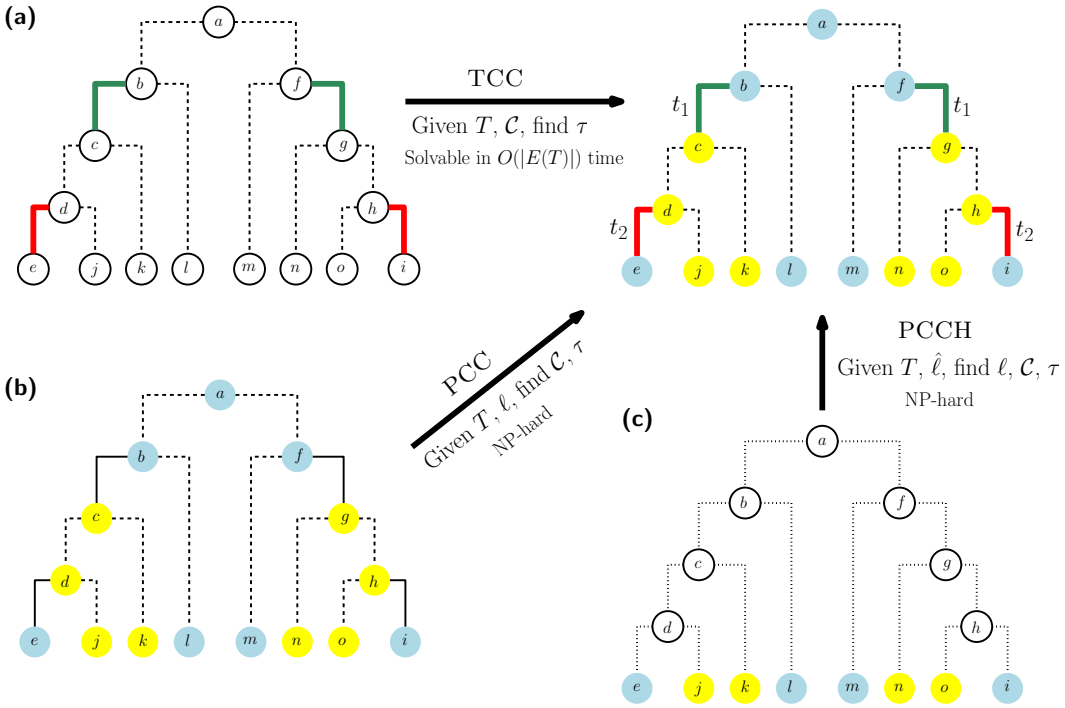


Figure 1 Overview of the three successive problem statements. (a) In the Temporally Consistent Comigrations (TCC) problem, we are given a rooted tree T and a set $\mathcal{C}$ of comigrations (edge colors). We seek a timestamp labeling τ that is temporally consistent with $\mathcal{C}$. Here, timestamp labeling τ for the output tree is indicated by the labels on the edges, where $\tau((b,c)) = \tau((f,g)) = t_1 < t_2 = \tau((d,e)) = \tau((h,i))$, satisfying temporal consistency. (b) In the Parsimonious Consistent Comigrations (PCC) problem, we are no longer given $\mathcal{C}$ only T and a location labeling ℓ (vertex colors). We seek a minimum set $\mathcal{C}$ of comigrations that are spatially and temporally consistent with (T, ℓ). Note that in both TCC and PCC, migrations (solid edges) and non-migrations (dashed edges) are uniquely determined from $\mathcal{C}$ and ℓ, respectively. (c) Finally, in the Parsimonious Consistent Comigration History (PCCH) problem, we are no longer given $\mathcal{C}$ and ℓ, so whether an edge is a migration or not is unknown (dotted edge). Rather we are given T and a leaf location labeling $\hat{\ell}$, seeking a location labeling ℓ inducing a minimum set $|M(T,\ell)|$ of migrations that subsequently admit a smallest set $\mathcal{C}$ of comigrations.

(Figure 1b). We prove this problem to be NP-hard. We then formulate our third problem, Parsimonious Consistent Comigration History (PCCH), where one is given a rooted tree with locations assigned to only the leaves. The goal is to identify a location labeling and comigrations that minimize the number of migrations and subsequently comigrations, while maintaining spatial and temporal consistency (Figure 1c). We show that PCCH is also NP-hard. We formulate integer linear programs for exactly solving PCC and PCCH. We run our methods on real and simulated data, finding that in practice, for small trees with non-complex migration patterns, MACHINA's definition of comigrations is adequate and does not lead to temporal inconsistencies.

2 Problem Statement

Let T be a tree rooted at vertex $r(T)$. As the tree T is rooted, its edges (u, v) are directed such that u is closest to the root $r(T)$ – in this manuscript, we refer to a directed edge or arc as an edge. We denote the vertex set of T by $V(T)$, the edge set by $E(T)$, and the leaf

set by $L(T)$. We refer to root-to-leaf paths as lineages. We write $u \preceq_T v$ to indicate vertex u is an ancestor of vertex v in tree T, i.e. there is a directed path from u to v. Note that $\preceq_T$ is reflexive, i.e. $v \preceq_T v$ for all vertices v. Additionally, we use $\delta(v)$ to denote the set of children of any vertex v. Our goal is to augment T such it allows us to represent a migration history. To that end, following the work of Slatkin and Maddison [29], we let Σ be the set of all locations of origin and define the *location labeling* $\ell : V(T) \to \Sigma$, mapping each vertex with its location of origin as follows.

▶ **Definition 1.** *A* location labeling *is a function $\ell : V(T) \to \Sigma$ that labels the vertices of T with locations from Σ.*

Given a location labeling ℓ of T, we define migrations as edges whose endpoints have different labels.

▶ **Definition 2.** *A* migration *is an edge $(u, v) \in E(T)$ whose endpoints u and v have different locations, i.e. $\ell(u) \neq \ell(v)$. The set of all migrations of T induced by location labeling ℓ is denoted by $M(T, \ell)$.*

In many evolutionary processes, multiple migrations between the pair of locations may occur as part of a single event. To model this, rather than considering migrations in isolation, we wish to partition the set $M(T, \ell)$ of migrations into comigrations $\mathcal{C}$.

▶ **Definition 3.** *A set $\mathcal{C}$ of* comigrations *is a partition of a set $M \subseteq E(T)$ of migrations, i.e. (i) each migration $(u, v) \in M$ occurs in exactly one part and (ii) the union of all parts $C \in \mathcal{C}$ equals M.*

Importantly, not all comigrations $\mathcal{C}$ are valid, as we require all the migrations in each single comigration event to migrate between the same pair of locations at the same time. In other words, we require spatial and temporal consistency defined as follows.

▶ **Definition 4.** *A set $\mathcal{C}$ of comigrations is* spatially consistent *with location labeling ℓ if for all two migrations $(u, v), (u', v')$ in the same part $C \in \mathcal{C}$ it holds that $\ell(u) = \ell(u')$ and $\ell(v) = \ell(v')$.*

To model temporal consistency, we label each migration by a timestamp defined as follows.

▶ **Definition 5.** *A* timestamp labeling *is a function $\tau : M \to \mathbb{N}$ that labels each migration of M with a timestamp.*

We say that comigrations $\mathcal{C}$ are *temporally consistent* if we can assign timestamps to each migration s.t. (i) all edges in the same part occur simultaneously and (ii) time moves forward along the directed edges of the tree.

▶ **Definition 6.** *A set $\mathcal{C}$ of comigrations is* temporally consistent with timestamp labeling τ *provided (i) all pairs $(u, v), (u', v')$ of migrations in the same part $C \in \mathcal{C}$ have the same timestamp, i.e. $\tau((u, v)) = \tau((u, v'))$ and (ii) $\tau((u, v)) < \tau((u', v'))$ for any two migrations $(u, v), (u', v')$ where $v \preceq_T u'$.*

The first problem focuses on determining the chronological order of comigration events. In other words, the goal of the first problem is to identify a timestamp labeling τ that is temporally consistent with a given set $\mathcal{C}$ of comigrations. Formally we define the problem as follows.

▶ **Problem 1** (Temporally Consistent Comigrations (TCC)). *Given a rooted tree T and comigrations $\mathcal{C}$ on migrations $M \subseteq E(T)$, find a timestamp labeling τ s.t. $\mathcal{C}$ is temporally consistent with τ.*

We say that comigrations $\mathcal{C}$ are *temporally consistent* if the above problem has a solution. A variant of the problem is when we are not given the set $\mathcal{C}$ of comigrations but only the location labeling ℓ. The task is to identify the comigration events, i.e. the set of migrations that happened simultaneously. In case there are multiple possible scenarios, we seek the most parsimonious solution, i.e. solution with the fewest comigration events. This leads to the following problem.

▶ **Problem 2** (Parsimonious Consistent Comigrations (PCC)). *Given a rooted tree T with location labeling $\ell : V(T) \to \Sigma$, find comigrations $\mathcal{C}$ of migrations $M(T, \ell)$ s.t. (i) $\mathcal{C}$ is spatially consistent with ℓ, (ii) $\mathcal{C}$ is temporally consistent for some timestamp labeling τ and (iii) the number $|\mathcal{C}|$ of comigrations is minimized.*

In practice, observing the locations of ancestral vertices from data obtained at present is not feasible. So instead of a location labeling on all vertices, we are only given a leaf labeling $\hat{\ell} : L(T) \to \Sigma$ as input, where each leaf $v \in L(T)$ is labeled with a location $\hat{\ell}(v)$ from Σ. Given the leaf labelings, we wish to infer the vertex labeling that corresponds to a most parsimonious solution. Similarly to the problem solved by MACHINA [11], we seek to find the solution that lexicographically minimizes the number of migrations and the number of comigrations.

▶ **Problem 3** (Parsimonious Consistent Comigration History (PCCH)). *Given a rooted tree T with location leaf labeling $\hat{\ell} : L(T) \to \Sigma$, find location labeling ℓ and comigrations $\mathcal{C}$ of $M(T, \ell)$ s.t. (i) $\ell(v) = \hat{\ell}(v)$ for all leaves $v \in L(T)$, (ii) C is spatially consistent with ℓ, (iii) there exist timestamps τ temporally consistent with $\mathcal{C}$ and (iv) the number $|M(T, \ell)|$ of migrations, and subsequently the number $|\mathcal{C}|$ of comigrations is minimized.*

For any tree T with leaf labeling $\hat{\ell} : L(T) \to \Sigma$, it holds that the number $|\mathcal{C}|$ of comigrations is at least $|\Sigma| - 1$ for any set $\mathcal{C}$ of comigrations. To see why, observe that each location is seeded at least once except the location at the root. This lower bound can be achieved trivially by labeling all the internal nodes with the same location. Since minimizing the number of comigrations results in each location being seeded exactly once, we minimize the number of migrations first and then comigrations to allow more complex migration scenarios while retaining simplicity. The key difference between the above problem and the problem solved by MACHINA is that here we include an explicit definition of temporal consistency. We will show in the next section that without this condition migration histories that contain temporal inconsistencies might be inferred.

3 Combinatorial Characterization and Complexity

This section includes the theoretical results on the combinatorial characteristics and complexity of the three discussed problems. Due to space constraints, proofs sketches have been moved to Appendix A.

3.1 Combinatorial Characterization of the TCC Problem

In the TCC problem we are given a set $\mathcal{C}$ of comigrations, partitioning the set M of migrations of a tree T. The task is to identify a timestamp labeling $\tau : M \to \mathbb{N}$ that is temporally consistent with $\mathcal{C}$. To solve this problem, we begin by defining the comigration graph $G_{T,\mathcal{C}}$.

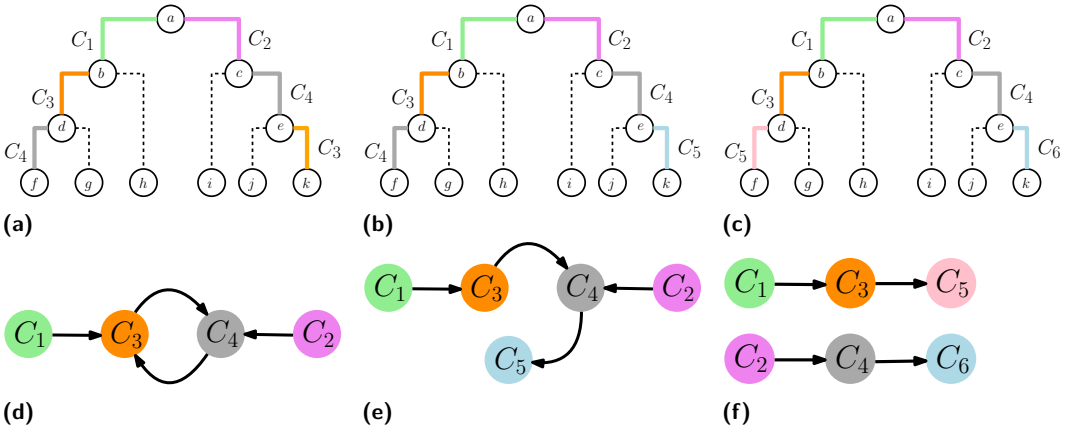


Figure 2 Temporally inconsistent and consistent comigrations with comigration graphs. Illustrated in (a), (b), and (c) are three distinct sets of comigrations within the same tree, where solid edges indicate migrations and dashed edges indicate non-migrations. Edge colors represent the comigrations to which the edges belong, with migrations of the same color belonging to the same comigration. The corresponding comigration graphs for (a), (b), and (c) are shown in (c), (d), and (e). Since the comigration graph illustrated in (d) contains a cycle, the comigrations illustrated in (a) are not temporally consistent. Comigrations corresponding to (b) and (c) are temporally consistent, as the corresponding comigration graphs (e) and (f) are DAG, even though (f) is disconnected.

▶ **Definition 7.** *A* comigration graph $G_{T,\mathcal{C}}$ *for a tree* T *with comigrations* $\mathcal{C} = \{C_1, \ldots, C_{|\mathcal{C}|}\}$ *is a directed graph with vertices* $V(G_{T,\mathcal{C}}) = \mathcal{C}$ *and a directed edge* $(C_a, C_b) \in E(G_{T,\mathcal{C}})$ *if there exist migrations* $(u_a, v_a) \in C_a$ *and* $(u_b, v_b) \in C_b$ *s.t.* $v_a \preceq_T u_b$ *and* $\mathcal{C}$ *does not contain any other migration on the path from* v_a *to* u_b *in* T*.*

Intuitively, a comigration graph $G_{T,\mathcal{C}}$ orders the comigrations $\mathcal{C}$ by the locations of their corresponding migrations in the tree T. That is, there is an edge (C_a, C_b) in $G_{T,\mathcal{C}}$ if and only if T contains two consecutive migrations on the same lineage, first a migration from C_a followed by a migration from C_b. Note that a comigration graph $G_{T,\mathcal{C}}$ can be disconnected, as shown in Figure 2f. Comigration graphs of sets of comigrations for migrations obtained by a location labeling do not contain self-loops.

▶ **Lemma 8.** *There are no self-loops in the comigration graph* $G_{T,\mathcal{C}}$ *of any set* $\mathcal{C}$ *of comigrations for migrations* $M(T, \ell)$ *induced by location labeling* ℓ *of a tree* T*.*

More generally, comigrations $\mathcal{C}$ admit temporally consistent timestamps if and only if the corresponding comigration $G_{T,\mathcal{C}}$ is a directed acyclic graph (DAG), as we prove in the following proposition.

▶ **Theorem 9.** *There exists a timestamp labeling* τ *that is temporally consistent with comigrations* $\mathcal{C}$ *of a tree* T *if and only if the comigration graph* $G_{T,\mathcal{C}}$ *is a DAG.*

In Section 4.1, we provide an algorithm for solving TCC in $O(|E(T)|)$ time.

3.1.1 MACHINA's Definition of Compatible Comigrations

As we have mentioned earlier, MACHINA [11] was the first method to incorporate comigrations in their problem formulation. Our notion of comigrations is similar to the one introduced in MACHINA [11], but there are significant distinctions. In MACHINA, comigrations $\mathcal{C}$ are

considered valid if for each comigration $C \in \mathcal{C}$, all the migrations belonging to C migrate between the same pair of locations, and no two migrations from C are in the same lineage. In other words, given location labeling ℓ, comigrations $\mathcal{C}$ are valid if they maintain compatibility defined as follows.

▶ **Definition 10** (El-Kebir et al. [11]). *Comigrations $\mathcal{C}$ for migrations $M(T, \ell)$ are* compatible *with location labeling ℓ provided for any two migrations $(u, v), (u', v')$ in the same comigration $C \in \mathcal{C}$ it holds that (i) $\ell(u) = \ell(u')$ and $\ell(v) = \ell(v')$, and (ii) neither $v \preceq_T u'$ nor $v' \preceq_T u$.*

Clearly, compatibility implies spatial consistency. As for the other direction, we have the following lemma relating our notions of spatial and temporal consistency (Definitions 4 and 6, respectively) with compatibility as defined above.

▶ **Lemma 11.** *Comigrations $\mathcal{C}$ for migrations $M(T, \ell)$ that are spatially and temporally consistent with location labeling ℓ of a tree T are also compatible with ℓ.*

The MACHINA paper shows that the minimum number $\gamma(T, \ell)$ of comigrations among all comigrations $\mathcal{C}$ that are compatible with a fixed location labeling ℓ is as follows.

▶ **Lemma 12** (El-Kebir et al. [11]). *The minimum number $\gamma(T, \ell)$ of comigrations among all comigrations compatible with ℓ equals*

$$\gamma(T, \ell) = \sum_{s,t \in \Sigma : s \neq t} \gamma(T, \ell, s, t). \tag{1}$$

where $\gamma(T, \ell, s, t)$ is the maximum number of migrations between locations (s, t) on any root-to-leaf path of T.

The above lemma combined with Lemma 11 leads to the following corollary.

▶ **Corollary 13.** *Comigrations $\mathcal{C}$ that are spatially and temporally consistent with location labeling ℓ of a tree T consist of at least $|\mathcal{C}| \geq \gamma(T, \ell)$ parts.*

While MACHINA only computes the number $\gamma(T, \ell)$ of comigrations and does not explicitly infer corresponding comigrations $\mathcal{C}^*$ s.t. $|\mathcal{C}^*| = \gamma(T, \ell)$, we show here that this can be done using a simple greedy algorithm. Briefly, we initialize $\mathcal{C}^* = \{C_1, \ldots, C_{|M(T,\ell)|}\}$ with each comigration C_i containing migration e_i for all $i \in [|M(T, \ell)|]$. Then, iteratively, we merge two distinct parts C and C' in $\mathcal{C}$ if their comprising migrations are between the same pair of locations and do not occur on the same root-to-leaf path in T. We repeat this procedure until no further merging is possible. Correctness follows from the fact that compatibility is maintained as a loop invariant.

Importantly, while comigrations $\mathcal{C}$ compatible with location labeling ℓ are also spatially consistent with ℓ, temporal consistency is not guaranteed. As an example, consider Figure 3a defining a tree T and location labeling ℓ with locations $\Sigma = \{\text{red}, \text{green}, \text{cyan}, \text{orange}\}$. Tree T and ℓ contain four migrations $(u_{\text{red}}, v_{\text{green}})$, $(u'_{\text{red}}, v'_{\text{green}})$, $(u_{\text{cyan}}, v_{\text{orange}})$ and $(u'_{\text{cyan}}, v'_{\text{orange}})$ s.t. one lineage of T contains the migration $(u_{\text{red}}, v_{\text{green}})$ followed by $(u_{\text{cyan}}, v_{\text{orange}})$ and another distinct lineage contains the migration $(u'_{\text{cyan}}, v'_{\text{orange}})$ followed by $(u'_{\text{red}}, v'_{\text{green}})$. Clearly, $\gamma(T, \ell) = 2$ as no lineage of T contains more than one migration between the same pair of locations. There is a unique set $\mathcal{C}^*$ of comigrations that is compatible with T s.t. $|\mathcal{C}^*| = \gamma(T, \ell) = 2$. That is, $\mathcal{C}^* = \{C_{(\text{red},\text{green})}, C_{(\text{cyan},\text{orange})}\}$ where $C_{(\text{red},\text{green})} = \{(u_{\text{red}}, v_{\text{green}}), (u'_{\text{red}}, v'_{\text{green}})\}$ and $C_{(\text{cyan},\text{orange})} = \{(u_{\text{cyan}}, v_{\text{orange}}), (u'_{\text{cyan}}, v'_{\text{orange}})\}$. Although $\mathcal{C}^*$ is spatially consistent, it is not temporally consistent as can be seen from the cycle in the corresponding migration graph $G_{T,\mathcal{C}^*}$. Indeed, if we assign timestamps τ s.t. migrations

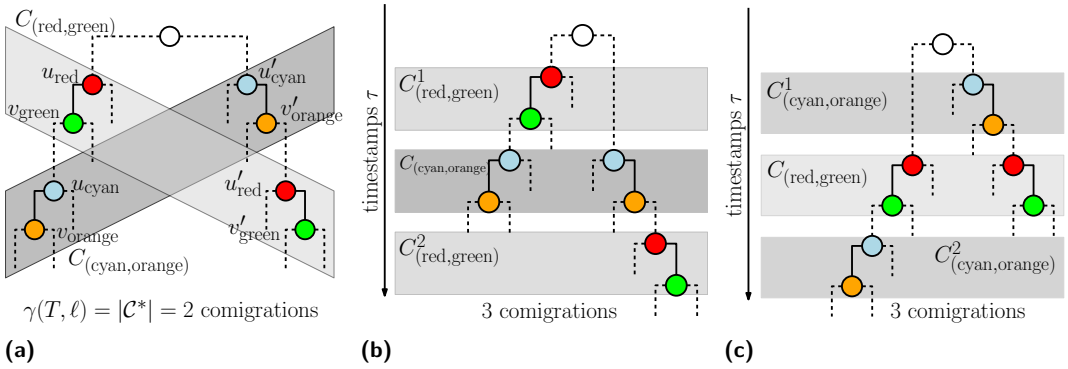


Figure 3 Comigrations inferred by MACHINA [11] might not be temporally consistent. Comigrations $\mathcal{C}^* = \{C_{(\text{red,green})}, C_{(\text{cyan,orange})}\}$ are compatible with the given location labeling ℓ of tree T. Moreover, these comigrations achieve the smallest number $\gamma(T,\ell) = 2$ possible for (T,ℓ). However, the comigration graph $G_{T,\mathcal{C}^*}$ has a cycle between its two vertices $C_{(\text{red,green})}$ and $C_{(\text{cyan,orange})}$. As such, by Theorem 9, $\mathcal{C}^*$ is not temporally consistent. To arrive at temporally consistent comigrations, either (b) $C_{(\text{red,green})}$ or (c) $C_{(\text{cyan,orange})}$ must be split.

in $C_{(\text{red,green})}$ precede $C_{(\text{cyan,orange})}$, we would have a violation of temporal consistency as $(u'_{\text{cyan}}, v'_{\text{orange}}) \preceq_T (u'_{\text{red}}, v'_{\text{green}})$ and yet $\tau((u'_{\text{cyan}}, v'_{\text{orange}})) > \tau((u'_{\text{red}}, v'_{\text{green}}))$. A similar violation would occur when using timestamps s.t. $C_{(\text{cyan,orange})}$ precedes $C_{(\text{red,green})}$. To obtain temporally consistent comigrations, we must break up either $C_{(\text{red,green})}$ (Figure 3b) or $C_{(\text{cyan,orange})}$ (Figure 3c), leading to an additional comigration in either case.

We look deeper into when compatible comigrations $\mathcal{C}$ are temporally consistent. We say a location labeling ℓ results in *reseeding* if there exists a root-to-leaf path in T containing two migrations $(u,v),(u',v')$ labeled as $\ell(u) \neq \ell(v)$, $\ell(u') \neq \ell(v')$ and $\ell(u) = \ell(v')$.

▶ **Proposition 14.** *If a location labeling ℓ of a tree T does not result in reseeding then any set $\mathcal{C}$ of comigrations on $M(T,\ell)$ that is compatible with ℓ is also temporally consistent.*

3.2 NP-Hardness of the PCC Problem

The example shown in Figure 3a and discussed in the previous section shows that not all comigrations $\mathcal{C}$ are temporally consistent, and that to achieve temporal consistency more comigrations than the polynomial-time computable lower bound $\gamma(T,\ell)$ might be needed. In this section, we study the complexity of the PCC problem of finding the smallest set $\mathcal{C}$ of temporally consistent comigrations for migrations $M(T,\ell)$ induced by a location labeling ℓ of a tree T. We have the following hardness result.

▶ **Theorem 15.** *PCC is NP-hard when $|\Sigma| \geq 3$.*

We begin by showing that PCC is NP-hard by reducing it from Shortest Common Supersequence. In Shortest Common Supersequence (SCS) problem, the input is a set $\{S_1,\ldots,S_n\}$ of n sequences, where each sequence S_i is an ordered list $s_{i,1}s_{i,2}\ldots s_{i,|S_i|}$ of symbols from a finite set $\mathcal{S}$. We say sequence Y is a *supersequence* of sequence X if there exists a function $F_{X,Y} : \{1,\ldots,|X|\} \to \{1,\ldots,|Y|\}$ s.t. $F_{X,Y}(i) = j$ if $X_i = Y_j$ and F is a strictly increasing monotone function. In the SCS problem, we seek the shortest sequence S^* s.t. S^* is a supersequence of all input sequences $S_1,\ldots,S_n$. The SCS problem is NP-hard when $|\mathcal{S}| \geq 2$ [25]. We describe a polynomial time reduction from SCS to PCC. To that end, given the input sequences $S_1,\ldots,S_n$, we build a tree T with location set $\Sigma = \mathcal{S} \cup \{\perp\}$ and location labeling $\ell : V(T) \to [\mathcal{S} \cup \{\perp\}]$ in polynomial time. The steps are as follows.

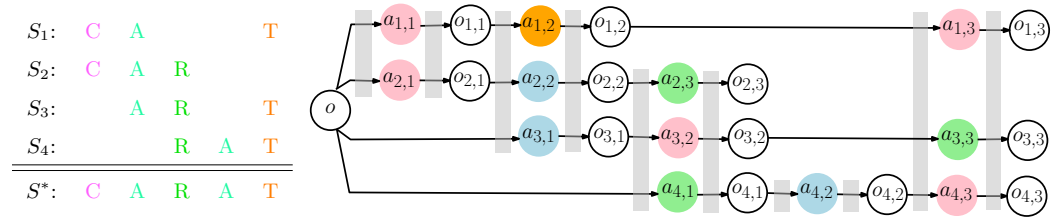


Figure 4 Reduction from Shortest Common Supersequence (SCS) to PCC. (a) An SCS problem instance of $n = 4$ input sequences $\{S_1, \ldots, S_4\}$ with the shortest common supersequence $S^* = s^*_1 \ldots s^*_{m^*}$ of length $m^* = |S^*| = 5$. The solution can be represented as an alignment A, with each column A_p containing pairs (i, j) indicating matched symbols $s_{i,j}$ and s^*_p. (b) The corresponding tree T with location labeling ℓ on $\Sigma = \{\bot, C, A, R, T\}$ is shown. Each vertex $a_{i,j}$ is labeled by location $\ell(v) = s_{i,j}$, with the color matching panel (a). Vertices $o_{i,j}$ are labeled by locations $\ell(v) = \bot$ and are colored white. The corresponding set $\mathcal{C}$ of $2m^* = 2 \cdot 5 = 10$ comigrations is shown using gray boxes, with migrations/edges overlapping a gray box belonging to the same part of $\mathcal{C}$.

1. First we add the root o to tree T. We label the root o with $\ell(o) = \bot$. For convenience, the root o may also be denoted by $o_{i,0}$ for any $1 \leq i \leq n$.
2. For each input sequence S_i, we attach to the root o the path $a_{i,1}, o_{i,1}, \ldots, a_{i,|S_i|}, o_{i,|S_i|}$ of length $2|S_i|$. We refer to vertices $a_{i,j}$ as *a-vertices* and vertices $o_{i,j}$ as *o-vertices*. Note that the edges in the constructed tree are either from an o-vertex to an a-vertex, or from an a-vertex to an o-vertex. As such, we call the former *o-a edges* and the latter *a-o edges*.
3. We set $\ell(a_{i,j}) = s_{i,j}$ and $\ell(o_{i,j}) = \bot$ for each $j \in \{1, \ldots, |S_i|\}$. Since $s_{i,j} \neq \bot$ for all $i \in [n]$ and $j \in \{1, \ldots, |S_i|\}$, all the edges in the tree are migrations.

Since $\Sigma = \mathcal{S} \cup \{\bot\}$ in the PCC instance corresponding to an SCS instance and SCS is NP-hardness when $|\mathcal{S}| \geq 2$, we obtain a lower bound of $|\Sigma| \geq 3$ in Theorem 15. We show an example reduction in Figure 4. Given the constructed tree T with location labeling ℓ, PCC seeks a set $\mathcal{C}^*$ of comigrations that is spatially consistent with ℓ, temporally consistent, and minimizes the number $|\mathcal{C}^*|$ of comigrations. We have the following definition.

▶ **Definition 16.** *A set $\mathcal{C}$ of comigrations for migrations $M(T, \ell) = E(T)$ is* balanced *if $\mathcal{C}$ consists of an even number of parts, half of which comprised of only o-a edges and the other half comprised of only a-o edges.*

▶ **Lemma 17.** *Any optimal set $\mathcal{C}^*$ of comigrations that is spatially and temporally consistent with location labeling ℓ of T is balanced.*

Next, we show there is a mapping between supersequences S of length m and balanced sets $\mathcal{C}$ of $2m$ comigrations that are spatially and temporally consistent with ℓ.

▶ **Lemma 18.** *There exists a common supersequence $S = s_1 \ldots s_m$ of $\{S_1, \ldots, S_n\}$ if and only if there exists a balanced set $\mathcal{C}$ of comigrations with $|\mathcal{C}| = 2m$ parts that is spatially and temporally consistent with location labeling ℓ of T.*

Now we can finally prove the following lemma, from which hardness follows.

▶ **Lemma 19.** *There exists a shortest common supersequence $S^* = s^*_1 \ldots s^*_{m^*}$ of $\{S_1, \ldots, S_n\}$ if and only if there exists a minimum-cardinality set $\mathcal{C}^*$ of comigrations for migrations $M(T, \ell) = E(T)$ that is spatially and temporally consistent with ℓ and has $|\mathcal{C}^*| = 2m^*$ parts.*

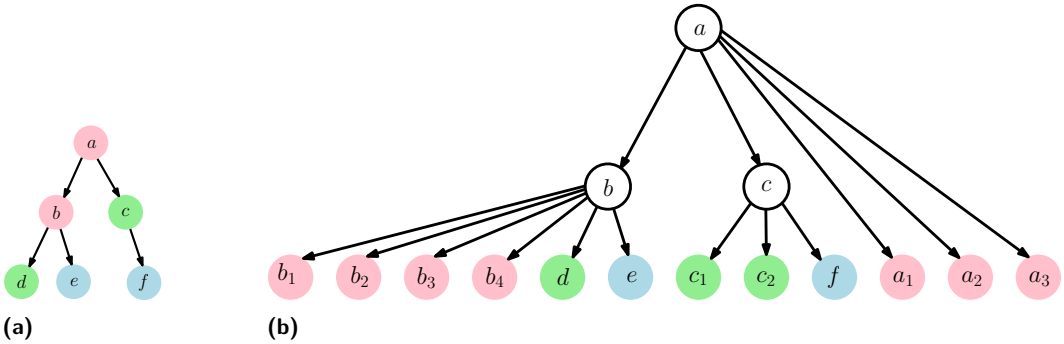


Figure 5 Reduction from PCCH to PCC. (a) An input tree T with vertex labeling ℓ. (b) The corresponding tree T' with leaf labeling $\hat{\ell}'$.

3.3 NP-Hardness of the PCCH Problem

In this subsection, we prove the hardness of PCCH.

▶ **Theorem 20.** *PCCH is NP-hard when* $|\Sigma| \geq 3$.

We prove this by reducing PCC to PCCH in polynomial time. That is, given a tree T with location labeling ℓ, we construct another tree T' with leaf labeling $\hat{\ell}'$. The construction is described below.

1. For each vertex $v \in V(T)$, add vertex v' to $V(T')$.
2. For each edge $(u, v) \in E(T)$, add the edge (u', v') to $E(T')$.
3. For each internal vertex $v \in V(T) \backslash L(T)$ with degree $\deg(v)$ attach $\deg(v) + 1$ leaves $\{v'_1, \ldots, v'_{\deg(v)+1}\}$ to vertex v' of T', labeling each of these leaves with $\ell(v)$, i.e. $\hat{\ell}'(v'_i) = \ell(v)$ for $i \in \{1, \ldots, \deg(v) + 1\}$.
4. For each leaf $v \in L(T)$, retain its label for the corresponding vertex v' in T', i.e. $\hat{\ell}'(v') = \ell(v)$.

Clearly, this reduction takes polynomial time. Since the reduction retains the set Σ of locations of the PCC instance, our hardness result for PCCH has the same bound $|\Sigma| \geq 3$ as in Theorem 15 establishing hardness for PCC. We illustrate the construction with an example in Figure 5. Given the constructed tree T' with leaf labeling $\hat{\ell}'$, PCCH aims to find the location labeling ℓ' as well as spatially and temporally consistent comigrations $\mathcal{C}'$ that result in the minimum number $|M(T', \ell')|$ of migrations and subsequently the minimum number $|\mathcal{C}'|$ of comigrations. As we show in the following lemma, the reduction ensures that an optimal location labeling ℓ' assigns the same locations to internal vertices of T' as location labeling ℓ does to the corresponding internal vertices of T.

▶ **Lemma 21.** *For each vertex* $v \in V(T)$*, an optimal location labeling* ℓ' *of* T' *labels the corresponding vertex* v' *as* $\ell'(v') = \ell(v)$.

The previous lemma means that the number $|M(T', \ell')|$ of migrations is fixed for optimal location labelings ℓ'.

▶ **Corollary 22.** *The number* $|M(T', \ell')|$ *of migrations for an optimal location labeling* ℓ' *of* T' *equals the number* $|M(T, \ell)|$ *of migrations in* T *with location labeling* ℓ.

We now prove the main lemma from which Theorem 20 follows.

▶ **Lemma 23.** *Let (T, ℓ) be a PCC instance with $|M(T, \ell)| = \mu$ and $(T', \hat{\ell}')$ be the corresponding PCCH instance. There exists an optimal solution $\mathcal{C}$ for (T, ℓ) s.t. $|\mathcal{C}| = \gamma$ if and only if there exists an optimal solution $(\ell', \mathcal{C}')$ for $(T', \hat{\ell}')$ s.t. $|M(T', \ell')| = \mu$ and $|\mathcal{C}'| = \gamma$.*

4 Method

In this section, we introduce algorithms to solve the three problems we discussed.

4.1 Linear Time Algorithm for the TCC Problem

Theorem 9 describes a way of solving TCC by computing a topological ordering of the vertices of the given comigration graph $G_{T,\mathcal{C}}$. Using Kahn's algorithm [16], we can obtain the topological ordering in time $O(|V(G_{T,\mathcal{C}})| + |E(G_{T,\mathcal{C}})|)$. As the number $|\mathcal{C}|$ of comigrations is at most the number $|M|$ of migrations, which in turn is at most the number $|E(T)|$ of edges in tree T, we have $|V(G_{T,\mathcal{C}})| = |\mathcal{C}| = O(|E(T)|)$. We bound $|E(G_{T,\mathcal{C}})|$ in the following lemma.

▶ **Lemma 24.** *The number of edges in comigration graph $G_{T,\mathcal{C}}$ is at most the number of edges in T, i.e. $|E(G_{T,\mathcal{C}})| = O(|E(T)|)$.*

Thus, given a comigration graph $G_{T,\mathcal{C}}$, TCC can be solved in $O(|V(G_{T,\mathcal{C}})| + |E(G_{T,\mathcal{C}})|) = O(|E(T)|)$ time. It remains to show how to construct the comigration graph $G_{T,\mathcal{C}}$ itself. Naively, we can check each pair of migrations $(u, v), (u', v') \in M$, and add edge (C_s, C_t) to $G_{T,\mathcal{C}}$ is $(u, v) \in C_s$, $(u', v') \in C_t$, $v \preceq_T u'$ when there is no other migration on the path from v and u'. But this approach is expensive, so we propose a new algorithm that runs in linear time. The recursive algorithm BuildComigrationGraph$(T, M, \mathcal{C}, v)$ takes as input tree T, migration set M, and comigrations $\mathcal{C}$, and a vertex v. It returns two outputs: (i) a comigration graph denoted as $G_{T_v,\mathcal{C}}$, where an edge (C_s, C_t) exists if there are two migrations $(u, v) \in C_s$ and $(u', v') \in C_t$ in the subtree T_v rooted at v, and (ii) a subset $X_v \subseteq \mathcal{C}$ of comigrations s.t. each $C \in X_s$ if C includes a migration (u', v') that is the first migration encountered on a path starting from v. Since $T_{r(T)} = T$, BuildComigrationGraph$(T, M, \mathcal{C}, r(T))$ infers the comigration graph $G_{T,\mathcal{C}}$. The pseudocode is given in Algorithm 1.

▶ **Theorem 25.** *BuildComigrationGraph$(T, M, \mathcal{C}, r(T))$ returns comigration graph $G_{T,\mathcal{C}}$ in $O(|E(T)|)$ time.*

4.2 ILP for the PCC Problem

We solve PCC by formulating an integer linear program that models comigrations $\mathcal{C}$ and timestamp labeling τ for a given tree T and location labeling ℓ, and minimizes over the number of comigrations $|\mathcal{C}|$ while maintaining temporal consistency for some τ. Due to space constraints, we refer the reader to Appendix B.1 for further details.

4.3 ILP for the PCCH Problem

To solve PCCH, we formulate an integer linear program (ILP) that models location labeling ℓ given tree T with leaf labeling $\hat{\ell}$. To do so, we model (i) location labeling, (ii) comigrations characterized by the labels of endpoints and timestamps of the member edges, (iii) assignment of edges to parts, and (iv) additional constraints to break symmetries. We describe each step in detail as follows.

Algorithm 1 BuildComigrationGraph$(T, M, \mathcal{C}, u)$.

Input: Rooted tree T, migrations $M \subseteq E(T)$, comigrations $\mathcal{C}$ and vertex u of T
Output: Comigration graph $G_{T_u,\mathcal{C}}$ for the subtree T_u of T rooted at u and comigrations $\mathcal{C}$ as well as set X_u comprised of parts $C \in \mathcal{C}$ containing a migration (v, w) that is the first migration on the path from u to v

```
if u ∈ L(T) then
    return (G_{T_u,C}, X_u) where V(G_{T_u,C}) = C, E(G_{T_u,C}) = ∅ and X_u = ∅
else
    V(G_{T_u,C}) ← C
    E(G_{T_u,C}) ← ∅
    X_u ← ∅
    foreach child v of u do
        (G_{T_v,C}), X_v) ← BuildComigrationGraph(T, M, C, v)
        E(G_{T_u,C}) ← E(G_{T_u,C}) ∪ E(G_{T_v,C})
        if (u, v) ∈ M then
            Let C_s be the part of C containing (u, v)
            for C_t ∈ X_v do
                E(G_{T_u,C}) ← E(G_{T_u,C}) ∪ {(C_s, C_t)}
            X_u ← X_u ∪ {C_s}
        else
            X_u ← X_u ∪ X_v
    return (G_{T_u,C}, X_u)
```

Location labeling. We introduce binary variables $\Lambda \in \{0,1\}^{|V(T)|\times|\Sigma|}$ to model location labeling ℓ. More specifically, we require $\Lambda_{v,s} = 1$ if $\ell(v) = s$, and $\Lambda(v, s) = 0$ otherwise.

$$\sum_{s\in\Sigma} \Lambda_{v,s} = 1, \qquad \forall v \in V(T).$$

Additionally, for the leaves of T, vertex labeling ℓ should maintain the input leaf labeling $\hat{\ell}$.

$$\Lambda_{v,\hat{\ell}(v)} = 1, \qquad \forall v \in L(T).$$

Timestamp labeling. To efficiently formulate the ILP, we put timestamps on non-migrations too, and include them in comigrations. This does not change the original PCCH algorithm, as we can ignore the timestamps on non-migrations and still get temporal consistency. Similar to our ILP for PCC, we introduce binary variables $\Gamma = \{0,1\}^{|E(T)\times|\Sigma|\times|\Sigma|\times|E(T)|}$ s.t. $\Gamma_{(u,v),s,t,e}$ is 1 if $\ell(u) = s$, $\ell(v) = t$, and $\tau((u, v)) = e$, and $\Gamma_{(u,v),s,t,e} = 0$ otherwise. The maximum number of such unique timestamps is $|E(T)|$, occurring when each edge of the tree is in a distinct comigration. The following three constraints enforce these described conditions.

$$\sum_{t\in\Sigma}\sum_{e\in[|E(T)|]} \Gamma_{(u,v),s,t,e} \le \Lambda_{u,s}, \qquad \forall (u, v) \in E(T), \forall s \in \Sigma,$$

$$\sum_{s\in\Sigma}\sum_{e\in[|E(T)|]} \Gamma_{(u,v),s,t,e} \le \Lambda_{v,t}, \qquad \forall (u, v) \in E(T), \forall t \in \Sigma,$$

$$\sum_{s\in\Sigma}\sum_{t\in\Sigma}\sum_{e\in[|E(T)|]} \Gamma_{(u,v),s,t,e} = 1, \qquad \forall (u, v) \in E(T).$$

To ensure temporal consistency, we require for any two consecutive edges $(u, v), (v, w) \in E(T)$, the timestamp of (u, v) to be smaller than the timestamp of (v, w).

$$\sum_{s\in\Sigma}\sum_{t\in\Sigma}\sum_{e\in[E]}\Gamma_{(u,v),s,t,e} \geq \sum_{s\in\Sigma}\sum_{t\in\Sigma}\sum_{e\in[E]}\Gamma_{(v,w),s,t,e} \quad \forall (u,v),(v,w)\in E(T), \forall E\in[|E(T)|]$$

Comigrations. Just like our ILP model for PCC, we introduce binary variables $\pi \in \{0,1\}^{|E(T)|\times|\Sigma|\times|\Sigma|}$ s.t. $\pi_{e,s,t} = 1$ if for any migration $(u, v) \in C$, it holds that $\ell(u) = s$, $\ell(v) = t$, and $\tau((u, v)) = e$. Again, we require each comigration to have a unique timestamp in this ILP and use the timestamps to identify a specific comigration. We have the following constraint that ensures spatial consistency by enforcing each comigration to be associated with a specific pair of locations.

$$\sum_{s\in\Sigma}\sum_{t\in\Sigma}\pi_{e,s,t},\leq 1 \qquad \forall e\in[|E(T)|].$$

If there is an edge (u, v) with $\ell(u) = s$, $\ell(v) = t$, and $\tau((u, v)) = e$, we force $\pi_{e,s,t}$ to be 1.

$$\Gamma_{(u,v),s,t,e} \leq \pi_{e,s,t}, \qquad \forall (u,v)\in E(T), \forall s,t\in\Sigma, \forall e\in[|E(T)|].$$

Additional constraints. Like the ILP model for PCC, we eliminate some symmetrical solutions by forcing smaller partition numbers to fill up first.

$$\sum_{s\in\Sigma}\sum_{t\in\Sigma}\pi_{e,s,t} \geq \sum_{s\in\Sigma}\sum_{t\in\Sigma}\pi_{e+1,s,t}, \qquad \forall e\in[|E(T)|-1].$$

Optimization function. We can compute the number of migrations from Γ by counting the number of migrations, i.e. edges with different labels at their endpoints. Since we ignore the comigrations with non-migrations, we only count the number of comigrations that contain migrations from π. Thus, given a tree T with location labeling ℓ, we define the objective function as

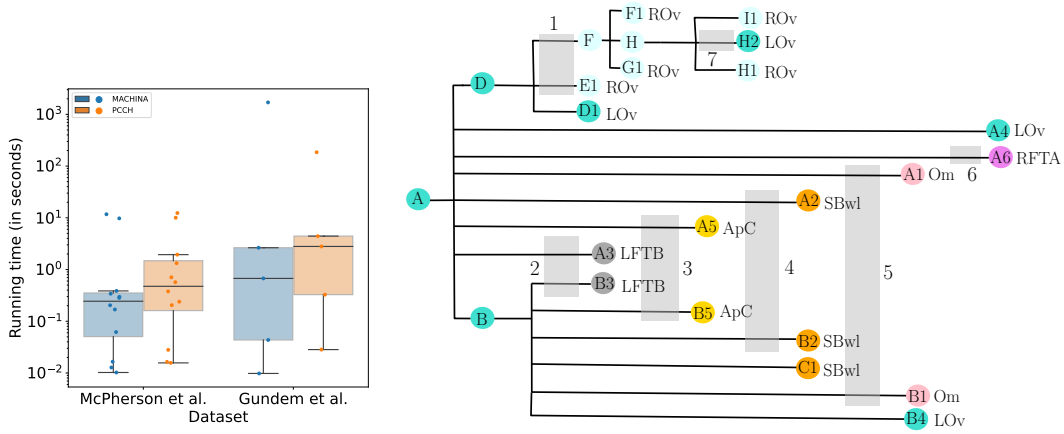
$$\min \sum_{(u,v)\in E(T)}\sum_{s,t\in\Sigma:s\neq t}\sum_{e\in[|E(T)|]}\Gamma_{(u,v),s,t,e} + \frac{1}{|E(T)|}\sum_{e\in[|E(T)|]}\sum_{s,t\in\Sigma:s\neq t}\pi_{e,s,t}.$$

5 Results

In this section, we present a performance comparison between MACHINA and PCCH by running both methods on real (Section 5.1) and simulated data (Section 5.2). All experiments were run on a server with Intel Xeon Gold 5120 dual CPUs with 14 cores each at 2.20 GHz and 512 GB RAM. The code is available at `https://github.com/elkebir-group/PCCH`.

5.1 Real data

Ovarian cancer. We applied PCCH to infer the migration history of a high-grade serous ovarian cancer dataset by McPherson et al. [22]. The available data contains the phylogenies of seven high-grade serous metastatic ovarian cancer patients. By employing whole genome and single nucleus sequencing, McPherson et al. [22] sequenced 68 tumor samples in total from seven patients including samples from the ovary, omentum, fallopian tube, peritoneal sites, and other distant metastatic sites, and inferred the migration history without considering comigrations. The same dataset was re-analyzed by El-Kebir et al. [11], reporting simpler



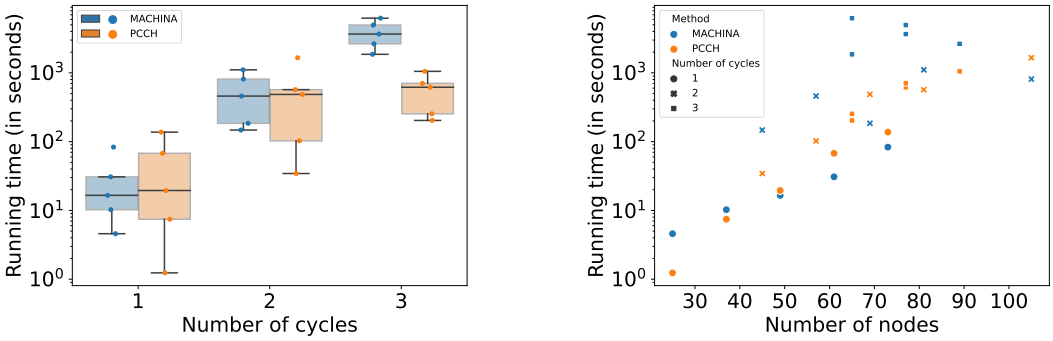
(a) (b)

Figure 6 MACHINA and PCCH results for ovarian [22] and prostate cancer [15] datasets. For all instances, MACHINA and PCCH inferred the same location labeling and the same number of comigrations, meaning that the previously reported MACHINA solutions are indeed temporally consistent. (a) PCCH is slower than MACHINA due to additionally enforcing temporal consistency. (b) PCCH results for ovarian cancer Patient 1. Gray boxes indicate comigrations (additionally labeled by timestamp) and vertex colors indicate location labeling.

migration patterns for some patients using the comigration criterion of MACHINA. For instance, for Patient 7, MACHINA reported that the tumor originated in the left or the right ovary, even though McPherson et al. claimed the right uterosacral ligament to be the primary region. MACHINA found a simpler migration history for Patient 3 without metastasis-to-metastasis migration or multisource seeding, unlike McPherson et al.'s prediction.

For each of the seven patients, we generated the location labeling with timestamps by solving PCCH. We found that PCCH's location labelings perfectly matched those of MACHINA as well as that both methods returned the same number of comigrations, thus showing that MACHINA's solutions were temporally consistent. We show the running time analysis for both PCCH and MACHINA in Figure 6a and Table S1. We found that PCCH generally takes slightly longer to finish (median of 0.474 s vs. 0.244 s for MACHINA). This is expected, as unlike MACHINA, PCCH includes checks for temporal consistency and returns timestamps along with a location labeling. As an example, we show the PCCH output for Patient 1 in Figure 6b with location and timestamp labels. Both MACHINA and PCCH report reseeding in the migration history, which can easily be seen by observing the edges with timestamps 1 and 7. Note that there are other possible timestamp labelings, and PCCH returns only one single solution.

Prostate cancer. We ran PCCH on another dataset by Gundem et al. [15]. The dataset contained matched primary and metastasis samples from five prostate cancer patients. This is another dataset that was previously analyzed by MACHINA [11], where MACHINA could infer alternative migration histories that were simpler and more consistent with the data compared to those reported in the original paper. Though the dataset contained examples of metastasis-to-metastasis spread, MACHINA did not infer reseeding in any of the patients. As such, by Proposition 14, MACHINA's results will be temporally consistent. Indeed, we found that the location labeling from our results from PCCH and the number of comigrations perfectly matched those reported by MACHINA. As shown in Figure 6a and Table S2,



(a) (b)

Figure 7 MACHINA and PCCH results for simulated data. We use colors to differentiate between MACHINA and PCCH. (a) The x-axis corresponds to the number of cycles in the induced comigration graph, where the y-axis stands for running time in seconds. (b) The x-axis represents the number of vertices, and the y-axis stands for running time in seconds. The number of cycles in the induced comigration graph is indicated by marker style.

we observed similar trends for running times (median of 2.795 s for PCCH vs. 0.67 s for MACHINA), although for Patient A22, MACHINA (1702.24 s) took longer than PCCH (185.18 s).

5.2 Simulated data

Results from real data suggest that although it is theoretically possible for MACHINA to return temporally inconsistent solutions, this does not occur in practice. For the same reason, realistic simulation models often fail to generate instances where MACHINA underestimates the number of comigrations. So to assess the performance of PCCH properly, we specifically simulated instances where MACHINA will fail to infer the correct number of comigrations. To be more exact, we sample a comigration graph with $k = \{1, 2, 3\}$ cycles first, and then generate trees with location labeling and comigrations that lead to k cycles in the induced comigration graph (Figure 3a shows an example with $k = 1$ cycle). In our simulated dataset, the cycles in the sampled comigration graph do not share any edges, so the difference between the number of comigrations inferred by PCCH and MACHINA equals the number of cycles. The running time comparison is given in Figure 7 and Table S3. Strikingly, we observed different trends here – MACHINA tends to be slower (median: 459.787 s) than PCCH (median: 203.438 s), especially when the corresponding comigration graph has increasing numbers of cycles (for $k = 3$ cycles, median of 3668.93 and 618.012 seconds for MACHINA and PCCH, respectively).

6 Conclusion

In this paper, we addressed a flaw in the definition of comigration adopted by multiple methods including MACHINA [11]. Specifically, we defined spatial and temporal consistency conditions for comigrations. This led us to formulate three successive problems First, Temporally Consistent Comigration (TCC) asks if a given comigration is temporally consistent, and, if so, returns a certifying timestamp labeling of migrations. We showed that TCC can be solved in linear time. Second, Parsimonious Consistent Comigration (PCC) aims to infer the smallest set of comigrations given a location labeling of both leaf and

internal vertices. We proved the problem is NP-hard, meaning even if we know the location of origin of every vertex and thus every migration, it is still hard to know which migrations occurred simultaneously under a parsimony criterion. Third, we formulated Parsimonious Consistent Comigration History (PCCH), which takes as input a leaf labeling, and infers the location labeling that minimizes the number of migrations, and subsequently the number of comigrations while maintaining spatial and temporal consistency. We showed that PCCH is NP-hard. Additionally, we discussed MACHINA's views on comigrations and its limitations in light of temporal consistency. We also investigated the sufficient conditions for MACHINA to correctly compute the number of comigrations. We compared the performance of PCCH with that of MACHINA on real and simulated data. We observed PCCH to return the same location labeling as MACHINA for all real data instances, which tells us that although it is theoretically possible for MACHINA to fail to compute the correct minimum number of comigrations for some instances, it is unlikely to come across such an instance in practice. Finally, we generated simulated instances where MACHINA fails to determine temporally consistent comigrations.

References

1 Nicola Aceto, Aditya Bardia, David T Miyamoto, Maria C Donaldson, Ben S Wittner, Joel A Spencer, Min Yu, Adam Pely, Amanda Engstrom, Huili Zhu, et al. Circulating tumor cell clusters are oligoclonal precursors of breast cancer metastasis. *Cell*, 158(5):1110–1122, 2014.

2 Nicolai J Birkbak and Nicholas McGranahan. Cancer genome evolutionary trajectories in metastasis. *Cancer cell*, 37(1):8–19, 2020.

3 Finlay Campbell, Anne Cori, Neil Ferguson, and Thibaut Jombart. Bayesian inference of transmission chains using timing of symptoms, pathogen genomes and contact data. *PLoS computational biology*, 15(3):e1006930, 2019.

4 Cédric Chauve, Akbar Rafiey, Adrian A Davin, Celine Scornavacca, Philippe Veber, Bastien Boussau, Gergely J Szöllősi, Vincent Daubin, and Eric Tannier. MaxTiC: Fast ranking of a phylogenetic tree by maximum time consistency with lateral gene transfers. *bioRxiv*, page 127548, 2017.

5 Kevin J Cheung and Andrew J Ewald. A collective route to metastasis: Seeding by tumor cell clusters. *Science*, 352(6282):167–169, 2016.

6 Kevin J Cheung, Veena Padmanaban, Vanesa Silvestri, Koen Schipper, Joshua D Cohen, Amanda N Fairchild, Michael A Gorin, James E Verdone, Kenneth J Pienta, Joel S Bader, et al. Polyclonal breast cancer metastases arise from collective dissemination of keratin 14-expressing tumor cell clusters. *Proceedings of the National Academy of Sciences*, 113(7):E854–E863, 2016.

7 Elizabeth Comen, Larry Norton, and Joan Massague. Clinical implications of cancer self-seeding. *Nature reviews Clinical oncology*, 8(6):369–377, 2011.

8 Maya Dadiani, Vyacheslav Kalchenko, Ady Yosepovich, Raanan Margalit, Yaron Hassid, Hadassa Degani, and Dalia Seger. Real-time imaging of lymphogenic metastasis in orthotopic human breast cancer. *Cancer research*, 66(16):8037–8041, 2006.

9 Lawrence A David and Eric J Alm. Rapid evolutionary innovation during an archaean genetic expansion. *Nature*, 469(7328):93–96, 2011.

10 Simon Dellicour, Guy Baele, Gytis Dudas, Nuno R Faria, Oliver G Pybus, Marc A Suchard, Andrew Rambaut, and Philippe Lemey. Phylodynamic assessment of intervention strategies for the West African Ebola virus outbreak. *Nature communications*, 9(1):2222, 2018.

11 Mohammed El-Kebir, Gryte Satas, and Benjamin J Raphael. Inferring parsimonious migration histories for metastatic cancers. *Nature genetics*, 50(5):718–726, 2018.

12 Mark B Faries, Shawn Steen, Xing Ye, Myung Sim, and Donald L Morton. Late recurrence in melanoma: clinical implications of lost dormancy. *Journal of the American College of Surgeons*, 217(1):27–34, 2013.

13 Ousmane Faye, Pierre-Yves Boëlle, Emmanuel Heleze, Oumar Faye, Cheikh Loucoubar, N'Faly Magassouba, Barré Soropogui, Sakoba Keita, Tata Gakou, Lamine Koivogui, et al. Chains of transmission and control of Ebola virus disease in Conakry, Guinea, in 2014: an observational study. *The Lancet Infectious Diseases*, 15(3):320–326, 2015.

14 Neil M Ferguson, Christl A Donnelly, and Roy M Anderson. Transmission intensity and impact of control policies on the foot and mouth epidemic in Great Britain. *Nature*, 413(6855):542–548, 2001.

15 Gunes Gundem, Peter Van Loo, Barbara Kremeyer, Ludmil B Alexandrov, Jose MC Tubio, Elli Papaemmanuil, Daniel S Brewer, Heini ML Kallio, Gunilla Högnäs, Matti Annala, et al. The evolutionary history of lethal metastatic prostate cancer. *Nature*, 520(7547):353–357, 2015.

16 Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

17 Sau Yee Kok, Hiroko Oshima, Kei Takahashi, Mizuho Nakayama, Kazuhiro Murakami, Hiroki R Ueda, Kohei Miyazono, and Masanobu Oshima. Malignant subclone drives metastasis of genetically and phenotypically heterogenous cell clusters through fibrotic niche generation. *Nature communications*, 12(1):863, 2021.

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

19 Ran Libeskind-Hadas and Michael A Charleston. On the computational complexity of the reticulate cophylogeny reconstruction problem. *Journal of Computational Biology*, 16(1):105–117, 2009.

20 Ravikanth Maddipati and Ben Z Stanger. Pancreatic cancer metastases harbor evidence of polyclonality. *Cancer discovery*, 5(10):1086–1097, 2015.

21 Dena Marrinucci, Kelly Bethel, Anand Kolatkar, Madelyn S Luttgen, Michael Malchiodi, Franziska Baehring, Katharina Voigt, Daniel Lazar, Jorge Nieva, Lyudmila Bazhenova, et al. Fluid biopsy in patients with metastatic prostate, pancreatic and breast cancers. *Physical biology*, 9(1):016003, 2012.

22 Andrew McPherson, Andrew Roth, Emma Laks, Tehmina Masud, Ali Bashashati, Allen W Zhang, Gavin Ha, Justina Biele, Damian Yap, Adrian Wan, et al. Divergent modes of clonal spread and intraperitoneal mixing in high-grade serous ovarian cancer. *Nature genetics*, 48(7):758–767, 2016.

23 Daniel Merkle and Martin Middendorf. Reconstruction of the cophylogenetic history of related phylogenetic trees with divergence timing information. *Theory in Biosciences*, 123:277–299, 2005.

24 Nikolai Nøjgaard, Manuela Geiß, Daniel Merkle, Peter F Stadler, Nicolas Wieseke, and Marc Hellmuth. Time-consistent reconciliation maps and forbidden time travel. *Algorithms for Molecular Biology*, 13(1):1–17, 2018.

25 Kari-Jouko Räihä and Esko Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16(2):187–198, 1981. doi:10.1016/0304-3975(81)90075-X.

26 J Zachary Sanborn, Jongsuk Chung, Elizabeth Purdom, Nicholas J Wang, Hojabr Kakavand, James S Wilmott, Timothy Butler, John F Thompson, Graham J Mann, Lauren E Haydu, et al. Phylogenetic analyses of melanoma reveal complex patterns of metastatic dissemination. *Proceedings of the National Academy of Sciences*, 112(35):10995–11000, 2015.

27 Palash Sashittal and Mohammed El-Kebir. SharpTNI: Counting and sampling parsimonious transmission networks under a weak bottleneck. *bioRxiv*, page 842237, 2019.

28 Palash Sashittal and Mohammed El-Kebir. Sampling and summarizing transmission trees with multi-strain infections. *Bioinformatics*, 36(Supplement_1):i362–i370, 2020.

29 Montgomery Slatkin and Wayne P Maddison. A cladistic measure of gene flow inferred from the phylogenies of alleles. *Genetics*, 123(3):603–613, 1989.

30 Ashley Sobel Leonard, Daniel B Weissman, Benjamin Greenbaum, Elodie Ghedin, and Katia Koelle. Transmission bottleneck size estimation from pathogen deep-sequencing data, with an application to human influenza A virus. *Journal of virology*, 91(14):e00171–17, 2017.

31 Jason A Somarelli, Kathryn E Ware, Rumen Kostadinov, Jeffrey M Robinson, Hakima Amri, Mones Abu-Asab, Nicolaas Fourie, Rui Diogo, David Swofford, and Jeffrey P Townsend. Phylooncology: Understanding cancer through phylogenetic analysis. *Biochimica et Biophysica Acta (BBA)-Reviews on Cancer*, 1867(2):101–108, 2017.

32 Enea Spada, Luciano Sagliocca, John Sourdis, Anna Rosa Garbuglia, Vincenzo Poggi, Carmela De Fusco, and Alfonso Mele. Use of the minimum spanning tree model for molecular epidemiological investigation of a nosocomial outbreak of hepatitis C virus infection. *Journal of clinical microbiology*, 42(9):4230–4236, 2004.

33 Doris P Tabassum and Kornelia Polyak. Tumorigenesis: it takes a village. *Nature Reviews Cancer*, 15(8):473–483, 2015.

34 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.

35 Ami Yamamoto, Andrea E Doak, and Kevin J Cheung. Orchestration of collective migration and metastasis by tumor cell clusters. *Annual Review of Pathology: Mechanisms of Disease*, 18:231–256, 2023.

36 Min Yu, Aditya Bardia, Ben S Wittner, Shannon L Stott, Malgorzata E Smas, David T Ting, Steven J Isakoff, Jordan C Ciciliano, Marissa N Wells, Ajay M Shah, et al. Circulating breast tumor cells exhibit dynamic changes in epithelial and mesenchymal composition. *science*, 339(6119):580–584, 2013.

A Supplementary Proof Sketches

A.1 Combinatorial Characterization of the TCC Problem

▶ **(Main Text) Lemma 8.** *There are no self-loops in the comigration graph $G_{T,\mathcal{C}}$ of any set $\mathcal{C}$ of comigrations for migrations $M(T,\ell)$ induced by location labeling ℓ of a tree T.*

Proof sketch. If there were a self-loop $(C,C) \in E(G_{T,\mathcal{C}})$ then there would be at least one pair $(u,v),(u',v') \in C$ such that there is no migration edge in the path from v to u'. But then $\ell(v) = \ell(u') = \ell(u)$, which means (u,v) is not a migration. ◀

▶ **(Main Text) Theorem 9.** *There exists a timestamp labeling τ that is temporally consistent with comigrations $\mathcal{C}$ of a tree T if and only if the comigration graph $G_{T,\mathcal{C}}$ is a DAG.*

Proof sketch. If $G_{T,\mathcal{C}}$ is a DAG then τ can be constructed by setting $\tau((u,v)) = i$ if (u,v) is in the i-th comigration in the topological ordering of $V(G_{T,\mathcal{C}})$. Also, it can be shown that if there is a cycle $C_i, C_{i+1}, \ldots, C_j, C_i$ in $G_{T,\mathcal{C}}$ then there does not exist a timestamp labeling τ that labels the edges present in $C_i, C_{i+1}, \ldots, C_j$. ◀

A.2 MACHINA's Definition of Compatible Comigrations

▶ **(Main Text) Lemma 11.** *Comigrations $\mathcal{C}$ for migrations $M(T,\ell)$ that are spatially and temporally consistent with location labeling ℓ of a tree T are also compatible with ℓ.*

Proof sketch. The proof follows from the definitions of compatibility, spatial and temporal consistency. ◀

▶ **(Main Text) Proposition 15.** *If a location labeling ℓ of a tree T does not result in reseeding then any set $\mathcal{C}$ of comigrations on $M(T,\ell)$ that is compatible with ℓ is also temporally consistent.*

Proof sketch. It can be shown that there cannot be a cycle in the comigration graph if there is no reseeding. ◀

A.3 NP-Hardness of the PCC Problem

▶ **(Main Text) Lemma 19.** *Any optimal set $\mathcal{C}^*$ of comigrations that is spatially and temporally consistent with location labeling ℓ of T is balanced.*

Proof sketch. It can be shown that if the number of comigrations containing $o - a$ edges is smaller than the number of comigrations containing $a - o$ edges in set $\mathcal{C}^*$, it is possible to construct a set of comigrations $\mathcal{C}'$ with twice the number of comigrations containing $o - a$ edges, indicating that $|\mathcal{C}'| < |\mathcal{C}^*|$, and so $\mathcal{C}^*$ is not optimal. This can be achieved by assigning edges $(a_{i,j}, o_{i,j+1})$ and $(a_{p,q}, o_{p,q+1})$ to the same comigration in $\mathcal{C}'$ if the corresponding $o - a$ edges $(o_{i,j-1}, a_{i,j})$ and $(o_{p,q-1}, a_{p,q})$ belong to the same part in $\mathcal{C}$. A similar argument applies when the number of comigrations containing $o - a$ edges is greater than the number of comigrations containing $a - o$ edges. ◀

▶ **(Main Text) Lemma 20.** *There exists a common supersequence $S = s_1 \dots s_m$ of $\{S_1, \dots, S_n\}$ if and only if there exists a balanced set $\mathcal{C}$ of comigrations with $|\mathcal{C}| = 2m$ parts that is spatially and temporally consistent with location labeling ℓ of T.*

Proof sketch. For each supersequence S of length m, we can construct comigrations $\mathcal{C}$ of size $2m$, and conversely for comigrations $\mathcal{C}$ of size $2m$, there exists a supersequence S of length m. Figure 4 provides an illustrative example of such a relationship between S and $\mathcal{C}$. ◀

▶ **(Main Text) Lemma 21.** *There exists a shortest common supersequence $S^* = s_1^* \dots s_{m^*}^*$ of $\{S_1, \dots, S_n\}$ if and only if there exists a minimum-cardinality set $\mathcal{C}^*$ of comigrations for migrations $M(T, \ell) = E(T)$ that is spatially and temporally consistent with ℓ and has $|\mathcal{C}^*| = 2m^*$ parts.*

Proof sketch. It can easily be proven using Lemma 18. ◀

A.4 NP-Hardness of the PCCH Problem

▶ **(Main Text) Lemma 23.** *For each vertex $v \in V(T)$, an optimal location labeling ℓ' of T' labels the corresponding vertex v' as $\ell'(v') = \ell(v)$.*

Proof sketch. It can be proven by showing that the number of migrations increases if $\ell'(v') = \ell(v)$. ◀

▶ **(Main Text) Lemma 25.** *Let (T, ℓ) be a PCC instance with $|M(T, \ell)| = \mu$ and $(T', \hat{\ell}')$ be the corresponding PCCH instance. There exists an optimal solution $\mathcal{C}$ for (T, ℓ) s.t. $|\mathcal{C}| = \gamma$ if and only if there exists an optimal solution $(\ell', \mathcal{C}')$ for $(T', \hat{\ell}')$ s.t. $|M(T', \ell')| = \mu$ and $|\mathcal{C}'| = \gamma$.*

Proof sketch. From Lemma 21, it is easy to show that the number of migrations $|M(T', \ell')|$ for PCCH instance (T', ℓ') is μ. Next to complete the proof, it suffices to show two things. First, if $\mathcal{C}$ is an optimal set of comigrations for PCC instance (T, ℓ) then the set $\mathcal{C}'$ of comigrations is optimal for PCCH instance $(T', \hat{\ell}')$ where in $\mathcal{C}'$ a pair of migrations (u', v') and (p', q') belong to the same part if the corresponding migrations (u, v) and (p, q) belong to the same part in $\mathcal{C}$. Second, if $\mathcal{C}'$ is an optimal set of comigrations for PCCH instance

$(T', \hat{\ell}')$ then the set $\mathcal{C}$ of comigrations is optimal for PCC instance (T, ℓ) where in $\mathcal{C}$ a pair of migrations (u, v) and (p, q) belong to the same part if the corresponding migrations (u', v') and (p', q') belong to the same part in $\mathcal{C}$. ◀

A.5 Linear Time Algorithm for the TCC Problem

▶ **(Main Text) Lemma 26.** *The number of edges in comigration graph $G_{T,\mathcal{C}}$ is at most the number of edges in T, i.e. $|E(G_{T,\mathcal{C}})| = O(|E(T)|)$.*

Proof sketch. It can be shown that the number of edges in $G_{T,\mathcal{C}}$ is bounded by the number of migrations, which in turn is bounded by the number of edges. ◀

▶ **(Main Text) Theorem 27.** *BuildComigrationGraph$(T, M, \mathcal{C}, r(T))$ returns comigration graph $G_{T,\mathcal{C}}$ in $O(|E(T)|)$ time.*

Proof sketch. This can be shown by structural induction, proving that the values of $G_{T_u,\mathcal{C}}$ and X_u are properly calculated for each node $u \in V(T)$. ◀

B Supplementary Methods

B.1 ILP for the PCC Problem

We solve PCC by formulating an integer linear program that models comigrations $\mathcal{C}$ and timestamp labeling τ for a given tree T and location labeling ℓ, and minimizes over the number of comigrations $|\mathcal{C}|$ while maintaining temporal consistency for some τ.

Timestamp labeling. We introduce binary variables $\Gamma = \{0,1\}^{|M(T,\ell)\times|M(T,\ell)|}$ to model $\tau : M(T,\ell) \to [\mathcal{N}]$ s.t. $\Gamma_{(u,v),e}$ is 1 if $\tau((u,v)) = e$,and 0 otherwise. Since the number of unique timestamps cannot exceed the number $|M(T,\ell)|$ of migrations, we limit the index e corresponding to timestamps to be at most $|M(T,\ell)|$. Since Γ is modeling a function τ, there cannot be more than one image e for each argument $(u,v) \in M(T,\ell)$:

$$\sum_{e=1}^{|E(T)|} \Gamma_{(u,v),e} = 1, \qquad \forall (u,v) \in M(T,\ell).$$

To ensure temporal consistency, we require $\tau((u,v)) < \tau((u',v')))$ for any two migrations $(u,v),(u',v') \in M(T,\ell)$ where $v \preceq_T u'$, and there is no migration in the path from v to u'. Now if $\tau((u,v)) < \tau((u',v')))$ then for any $E \in \{\tau((u',v')),\ldots,\tau((u,v))\}$ we have $\sum_{e=1}^{E} \Gamma_{(u,v),e} < \sum_{e=1}^{E} \Gamma_{(u',v'),e}$. Conversely, if $\tau((u,v)) = \tau((u',v')))$ then $\sum_{e=1}^{E} \Gamma_{(u,v),e} = \sum_{e=1}^{E} \Gamma_{(u',v'),e}$. We combine these two conditions to form the following constraint.

$$\sum_{e=1}^{E} \Gamma_{(u,v),e} \geq \sum_{e=1}^{E} \Gamma_{(u',v'),e}, \qquad \forall (u,v),(u',v') \in X(T,\ell), E \in [|M(T,\ell)|],$$

where $X(T,\ell)$ consists of all ordered pairs $((u,v),(u',v'))$ of migrations s.t. (i) $(u,v),(u',v') \in M(T,\ell)$, (ii) $v \preceq_T u'$ and (iii) there is no migration in the path from v to u'.

Comigrations. For spatiotemporally consistent comigrations $\mathcal{C}$, each part $C \in \mathcal{C}$ consists of migrations between the same pair of locations indicated by ℓ and have the same timestamp given by a timestamp labeling τ. Thus, to model comigrations, we introduce binary variables $\pi \in \{0,1\}^{|M(T,\ell)|\times|\Sigma|\times|\Sigma|}$, where $\pi_{e,s,t} = 1$ if the migrations in the comigration with timestamp $1 \leq e \leq |M(T,\ell)|$ migrate from $s \in \Sigma$ to $t \in \Sigma$. In other words, $\pi_{e,s,t}$ corresponds to comigration $C \in \mathcal{C}$ if for any migration $(u,v) \in C$ it holds that $\ell(u) = s$, $\ell(v) = t$, and $\tau((u,v)) = e$. Without loss of generality, we require each comigration to have a unique timestamp in this formulation, which we use to identify the comigration. This is enforced by the following constraint.

$$\sum_{s\in\Sigma}\sum_{t\in\Sigma}\pi_{e,s,t} \leq 1, \qquad \forall e \in [|M(T,\ell)|].$$

If there is a migration (u,v) and $\tau((u,v)) = e$, i.e. $\Gamma_{(u,v),e} \leq \pi_{e,\ell(u),\ell(v)} = 1$, then we force $\pi_{e,\ell(u),\ell(v)}$ to be 1.

$$\pi_{e,\ell(u),\ell(v)} \geq \Gamma_{(u,v),e}, \qquad \forall (u,v) \in M(T,\ell), \forall e \in [|M(T,\ell)|].$$

Additional constraints. To increase performance, we eliminate some symmetrical solutions by forcing smaller partition numbers to fill up first.

$$\sum_{s\in\Sigma}\sum_{t\in\Sigma}\pi_{e,s,t} \geq \sum_{s\in\Sigma}\sum_{t\in\Sigma}\pi_{e+1,s,t}, \qquad \forall e \in [|M(T,\ell)| - 1].$$

Optimization function. Since each comigration has a unique timestamp, we can get the total number of comigrations by counting nonzero entries in π.

$$\min \sum_{e\in[|E(T)|]}\sum_{s,t\in\Sigma:s\neq t}\pi_{e,s,t}.$$

Table S1 Detailed results for the ovarian cancer dataset from McPherson et al. [22].

Primary	Patient ID	#vertices	#migrations	#comigrations	Running time (in seconds)	
					MACHINA	PCCH
LOv	Patient 1	24	13	7	0.386	1.941
	Patient 3	36	27	7	9.764	10.042
	Patient 4	18	7	3	0.204	0.378
	Patient 7	19	12	6	0.341	0.569
	Patient 9	9	4	2	0.0128	0.015
ROv	Patient 1	24	13	10	0.284	1.326
	Patient 2	10	2	1	0.01	0.027
	Patient 3	36	27	7	11.725	12.364
	Patient 4	18	6	3	0.0618	0.239
	Patient 7	19	13	6	0.297	0.707
	Patient 9	9	5	2	0.0165	0.0163
	Patient 10	17	6	2	0.169	0.204

Table S2 Detailed results for the prostate cancer dataset from Gundem et al. [15].

Patient ID	#vertices	#migrations	#comigrations	Running time (in seconds)	
				MACHINA	PCCH
A10	15	3	3	0.043	0.327
A22	58	32	14	1702.24	185.18
A29	9	1	1	0.009	0.028
A31	29	13	7	2.64	4.428
A32	27	9	4	0.67	2.795

Table S3 Detailed results for the simulated dataset.

#cycles	#vertices	#migrations	#comigrations		Running time (in seconds)	
			MACHINA	PCCH	MACHINA	PCCH
1	25	6	4	5	4.59	1.24
	37	9	6	7	10.24	7.44
	49	12	8	9	16.56	19.48
	61	15	10	11	30.76	67.71
	73	18	12	13	83.34	137.48
2	45	11	6	8	147.46	34.41
	57	14	8	10	459.79	102.84
	69	17	10	12	184.58	485.99
	81	20	12	14	1107.75	570.13
	105	26	16	18	811.93	1658.93
3	65	16	9	12	6241.98	203.44
	65	16	8	11	1861.78	253.82
	77	19	11	14	4959.63	618.01
	77	19	10	13	3668.93	703.97
	89	22	12	15	2645.30	1052.21

Finding Maximal Exact Matches in Graphs

Nicola Rizzo
Department of Computer Science, University of Helsinki, Finland

Manuel Cáceres
Department of Computer Science, University of Helsinki, Finland

Veli Mäkinen
Department of Computer Science, University of Helsinki, Finland

Abstract

We study the problem of finding maximal exact matches (MEMs) between a query string Q and a labeled graph G. MEMs are an important class of seeds, often used in seed-chain-extend type of practical alignment methods because of their strong connections to classical metrics. A principled way to speed up chaining is to limit the number of MEMs by considering only MEMs of length at least κ (κ-MEMs). However, on arbitrary input graphs, the problem of finding MEMs cannot be solved in truly sub-quadratic time under SETH (Equi et al., ICALP 2019) even on acyclic graphs.

In this paper we show an $O(n \cdot L \cdot d^{L-1} + m + M_{\kappa,L})$-time algorithm finding all κ-MEMs between Q and G spanning exactly L nodes in G, where n is the total length of node labels, d is the maximum degree of a node in G, $m = |Q|$, and $M_{\kappa,L}$ is the number of output MEMs. We use this algorithm to develop a κ-MEM finding solution on indexable Elastic Founder Graphs (Equi et al., Algorithmica 2022) running in time $O(nH^2 + m + M_\kappa)$, where H is the maximum number of nodes in a block, and M_κ is the total number of κ-MEMs.

Our results generalize to the analysis of multiple query strings (MEMs between G and any of the strings). Additionally, we provide some preliminary experimental results showing that the number of graph MEMs is an order of magnitude smaller than the number of string MEMs of the corresponding concatenated collection.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Pattern matching; Theory of computation → Sorting and searching; Applied computing → Genomics

Keywords and phrases Sequence to graph alignment, bidirectional BWT, r-index, suffix tree, founder graphs

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.10

Supplementary Material *Software (Source Code)*: `https://github.com/algbio/br-index-mems` archived at `swh:1:dir:62181b3b38fb659c1c266508948c6058a61f3623`

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. 956229, and from the Academy of Finland grants No. 352821 and 328877.

Acknowledgements We thank an anonymous reviewer for pointing out an anomaly that lead us to correct the choice of a parameter value in our experiments.

1 Introduction

Sequence alignment has been studied since the 1970s [34, 46] and is a fundamental problem of computational molecular biology. Solving the classical problems of *longest common subsequence* (LCS) and *edit distance* (ED) between two strings takes quadratic time with simple dynamic programs, and it was recently proven that no strongly subquadratic-time algorithms exist conditioned on the Strong Exponential Time Hypothesis (SETH) [5, 3]. To overcome this hardness, researchers have used heuristics such as *co-linear chaining* [1]: by

taking (short) matches between the input strings, known as *anchors*, we can take an ordered subset of these anchors and *chain* them together into an alignment. Furthermore, when using *maximal exact matches* (MEMs) as anchors, different co-linear chaining formulations capture both LCS [31] and ED [24] in near-linear time. MEMs are also used in popular seed-and-extend read aligners [25, 32]. In fact, practical tools limit the number of MEMs by considering only κ-MEMs (MEMs of length at least κ) [36, 45].

Extending alignment between sequences to sequence-to-graph alignment is an emerging and central challenge of *computational pangenomics* [10], as labeled graphs are a popular representation of pangenomes used in recent bioinformatics tools [27, 8, 37, 26]. We assume that a labeled graph $G = (V, E, \ell)$ ($\ell\colon V \to \Sigma^+$) is the reference pangenome of interest. Unfortunately, even finding exact occurrences of a given pattern in G does not admit strongly subquadratic-time solutions under SETH [15], and furthermore, a graph cannot be indexed in polynomial time to answer strongly subquadratic-time pattern matching queries [14]. To circumvent this difficulty, research efforts have concentrated on finding parameterized solutions to (exact) pattern matching in labeled graphs [7, 12, 11, 41]. Moreover, the use of MEMs and co-linear chaining has also been extended to graphs [27, 8, 37, 26].

In this paper, we study the problem of efficiently finding MEMs between a query string Q and a labeled graph G, where we extend the MEM definition to capture any maximal match between Q and the string spelled by some path of G. More precisely, our contributions are as follows:

- In Section 3.1, we adapt the MEM finding algorithm between two strings of Belazzougui et al. [4] to find all κ-node-MEMs between Q and $G = (V, E, \ell)$ in $O(m + n + M_\kappa)$ time, where $m = |Q|$, $n = \sum_{v \in V} |\ell(v)|$ is the cumulative length of the node labels, and M_κ is the number of κ-node-MEMs (of length at least κ and between the node labels and Q).
- Next, in Section 3.2, we extend the previous algorithm to find all κ-MEMs spanning exactly L nodes of G in time $O(m + n \cdot L \cdot d^{L-1} + M_{\kappa,L})$, where d is the maximum degree of any node $v \in V$ and $M_{\kappa,L}$ are the κ-MEMs of interest. Note that MEMs spanning less than L nodes can occur multiple times in paths spanning exactly L nodes, and our contribution is to introduce an efficient technique to filter out these MEMs.
- In Section 4, we obtain the following results:
  - We study κ-MEMs in indexable Elastic Founder Graphs (EFGs) [16], a subclass of labeled acyclic graphs admitting a poly-time indexing scheme for linear-time pattern matching. Given an indexable EFG G of height H (the maximum number of nodes in a graph block), we develop a suffix-tree-based solution to find all κ-MEMs spanning more than 3 nodes in G in $O(nH^2 + m + M_{\kappa,4+})$ time, where $M_{\kappa,4+}$ are the number of output MEMs.
  - Combined with the above results for $L = 1, 2, 3$, we can find κ-MEMs of an indexable EFG G in $O(nH^2 + m + M_\kappa)$ time.
  - We note that the previous results easily generalize to the batched query setting: by substituting Q with the concatenation of different query strings $Q_1, \ldots, Q_t$ of total length m, we compute all κ-MEMs between any query string and the graph with the same stated running time.
- Finally, in Section 5 we provide some preliminary experimental results on finding MEMs from a collection of strains of covid19. We use the bidirectional r-index [2] as the underlying machinery. On the one hand, we build the r-index of the concatenation of the strains and find all m_κ κ-MEMs. On the other hand, we build an indexable EFG of the strains and find an upper bound on all M_κ κ-MEMs in this case. Our results indicate that M_κ is an order of magnitude smaller than m_κ, thus confirming that graph MEMs compactly represent all MEMs.

2 Preliminaries

Strings

We denote integer intervals by $[x..y]$, x and y inclusive. Let $\Sigma = [1..\sigma]$ be an alphabet. A *string* $T[1..n]$ is a sequence of symbols from Σ, that is, $T \in \Sigma^n$ where Σ^n denotes the set of strings of length n over Σ. The *length* of a string T is denoted $|T|$ and the *empty string* ε is the string of length 0. In this paper, we assume that σ is always smaller or equal to the length of the strings we are working with. The concatenation of strings T_1 and T_2 is denoted as $T_1 \cdot T_2$, or just T_1T_2. We denote by $T[x..y]$ the *substring* of T made of the concatenation of its characters from the x-th to the y-th, both inclusive; if $x = y$ then we also use $T[x]$ and if $y < x$ then $T[x..y] = \varepsilon$. The *reverse* of a string $T[1..n]$, denoted by $\overline{T}$, is the string T read from right to left, that is, $\overline{T} = T[n]T[n-1]..T[1]$. A *suffix* (*prefix*) of string $T[1..n]$ is $T[x..n]$ ($T[1..y]$) for $1 \leq x \leq n$ ($1 \leq y \leq n$) and we say it is *proper* if $x > 1$ ($y < n$). We denote by Σ^* the set of finite strings over Σ, and also $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. String Q *occurs* in T if $Q = T[x..y]$ for some interval $[x..y]$; in this case, we say that $[x..y]$ is a match of Q in T. Moreover, we study matches between substrings of Q and T: a *maximal exact match* (MEM) between Q and T is a triplet (x_1, x_2, ℓ) such that $Q[x_1..x_1 + \ell - 1] = T[x_2..x_2 + \ell - 1]$ and the match cannot be extended to the left nor to the right, that is, $x_1 = 1$ or $x_2 = 1$ or $Q[x_1 - 1] \neq T[x_2 - 1]$ (*left-maximality*) and $x_1 + \ell = |Q|$ or $x_2 + \ell = |T|$ or $Q[x_1 + \ell] \neq T[x_2 + \ell]$ (*right-maximality*). In this case, we say that the substring $Q[x_1..x_1 + \ell - 1]$ is a *MEM string* between Q and T. The *lexicographic order* of two strings T_1 and T_2 is naturally defined by the total order $\leq$ of the alphabet: $T_1 < T_2$ if and only if $T_1 \neq T_2$ and T_1 is a prefix of T_2 or there exists $y \geq 0$ such that $T_1[1..y] = T_2[1..y]$ and $T_1[y + 1] < T_2[y + 1]$. We avoid the prefix-case by adding an *end marker* $\$ \notin \Sigma$ to the strings and considering $\$$ to be the lexicographically smaller than any character in Σ.

Labeled graphs

Let $G = (V, E, \ell)$ be a labeled graph with V being the set of nodes, E being the set of edges, and $\ell : V \to \Sigma^+$ being a function giving a label to each node. A length-k *path* P from v_1 to v_k is a sequence of nodes $v_1, \ldots, v_k$ connected by edges, that is, $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k) \in E$. A node u *reaches* a node v if there is a path from u to v. The label $\ell(P) \coloneqq \ell(v_1) \cdots \ell(v_k)$ of P is the concatenation of the labels of the nodes in the path. For a node v and a path P we use $\|\cdot\|$ to denote its *string length*, that is, $\|v\| = |\ell(v)|$ and $\|P\| = |\ell(P)|$. Let Q be a query string. We say that Q *occurs* in G if Q occurs in $\ell(P)$ for some path P. In this case, the *exact match* of Q in G can be identified by the triple $(i, P = v_1 \ldots v_k, j)$, where $Q = \ell(v_1)[i..] \cdot \ell(v_2) \cdots \ell(v_{k-1}) \cdot \ell(v_k)[..j]$, with $1 \leq i \leq \|v_1\|$ and $1 \leq j \leq \|v_k\|$, and we call such triple a *substring* of G. Given a substring (i, P, j) of G, we define its *left-extension* $\text{lext}(i, P, j)$ as the singleton $\{\ell(v_1)[i - 1]\}$ if $i > 1$ and otherwise as the set of characters $\{\ell(u)[\|u\|] \mid (u, v_1) \in E\}$. Symmetrically, the *right-extension* $\text{rext}(i, P, j)$ is $\{\ell(v_k)[j + 1]\}$ if $j < \|v_k\|$ and otherwise it is $\{\ell(v)[1] \mid (v_k, v) \in E\}$. Note that the left (right) extension can be equal to the empty set $\emptyset$, if the start (end) node of P does not have incoming (outgoing) edges. Figure 1 illustrates these concepts.

Basic tools

A *trie* or *keyword tree* [13] of a set of strings is an ordinal tree where the outgoing edges of each node are labeled by distinct symbols (the order of the children follows the order of the alphabet) and there is a unique root-to-leaf path spelling each string in the set; the shared

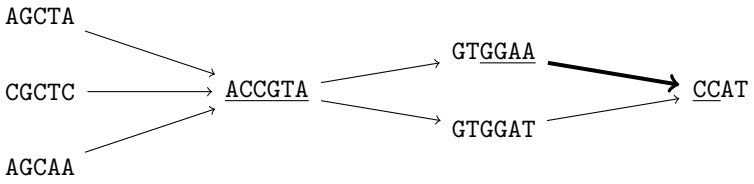


Figure 1 Substring ACCGTA (underlined) with left-extension {A, C} and right-extension {G}, and substring GGAACC (underlined, bold edge) with left-extension {T} and right-extension {A}.

part of two root-to-leaf paths spells the longest common prefix of the corresponding strings. In a *compact trie* [22], the maximal non-branching paths of a trie become edges labeled with the concatenation of labels on the path. The *suffix tree* of $T \in \Sigma^*$ is the compact trie of all suffixes of the string $T' = T\$$. In this case, the edge labels are substrings of T and can be represented in constant space as an interval of T. Such tree uses linear space and can be constructed in linear time, assuming that $\sigma \leq |T|$, so that when reading the root-to-leaf paths from left to right, the suffixes are listed in their lexicographic order [44, 17]. As such, the order spelled by the leaves of the suffix tree form the *suffix array* $\mathsf{SA}_T[1..|T'|]$, where $\mathsf{SA}_T[i] = j$ iff $T'[j..|T'|]$ is the i-th smallest suffix in lexicographic order. When applied to a string T, the *Burrows–Wheeler transform* (BWT) [6] yields another string BWT_T such that $\mathsf{BWT}_T[i] = T'[\mathsf{SA}[i] - 1]$ (we assume T' to be a circular string, i.e. $T'[-1] = T'[|T| + 1] = \$$).

Let $Q[1..m]$ be a query string. If Q occurs in T, then the *locus* or *implicit node* of Q in the suffix tree of T is (v, k) such that $Q = XY$, where X is the path spelled from the root to the parent of v and Y is the prefix of length k of the edge from the parent of v to v. The leaves in the subtree rooted at v, also known as *the leaves covered by* v, correspond to all the suffixes sharing the common prefix Q. Such leaves form an interval in the SA and equivalently in the BWT. Let aX and X be the strings spelled from the root of the suffix tree to nodes v and w, respectively. Then one can store a *suffix link* from v to w. Suffix links from implicit nodes are called implicit suffix links.

The *bidirectional BWT* [43] is a compact BWT-based index capable of solving the MEM finding problem in linear time [4]. The algorithm simulates the traversal of the corresponding suffix trees to find maximal occurrences in both strings: in the first step, it locates the suffix tree nodes (intervals in the BWTs) corresponding to the maximal matches, that is, the MEM strings, and then it uses a cross-product algorithm to extract each MEM from the BWT intervals.

Let $B[1..n]$ be a bitvector, that is, a string over the alphabet $\{0, 1\}$. There is a data structure that can be constructed in time $O(n)$ which answers $r = \texttt{rank}(B, i)$ and $j = \texttt{select}(B, r)$ in constant time, where the former operation returns the number of 1s in $B[1..i]$ and the latter returns the position $j \leq i$ of the r-th 1 in B [9, 23].

Let $D[1..n]$ be an array of integers. There is a *range minimum query* data structure that can be constructed in $O(n)$ time which answers $\mathsf{RMQ}_D(i, j)$ in constant time [19], where $\mathsf{RMQ}_D(i, j)$ returns an index k such that $D[k]$ is the minimum value in the subarray $D[i..j]$. We will use the following lemma that exploits range queries recursively.

▶ **Lemma 1.** *Let $D[1..n]$ be an array of integers. One can preprocess D in $O(n)$ time so that given a threshold Δ, one can list all elements of D such that $D[i] \leq \Delta$ in linear time in the size of the output.*

Proof. We can build the range minimum query data structure on D. Consider a recursive algorithm analogous to the one in [33] which starts with $k = \mathsf{RMQ}_D(1, n)$. If $D[k] > \Delta$, the algorithm stops as no element in the range is at least Δ. Otherwise, the algorithm reports k and recursively continues with $\mathsf{RMQ}_D(1, k-1)$ and $\mathsf{RMQ}_D(k+1, n)$. Note that each recursive call performs exactly one RMQ operation: if an element is reported the RMQ is charged to this element, otherwise it is charged to its parent (in the recursion tree), and thus the number of RMQ operations is linear in the output size. ◀

3 Finding MEMs in Labeled Graphs

Let us consider the problem of finding all *maximal exact matches* (MEMs) between a labeled graph $G = (V, E, \ell)$, with $\ell : V \to \Sigma^+$, and a query string $Q \in \Sigma^+$.

▶ **Definition 2** (MEM between a pattern and a graph). *Let $G = (V, E, \ell)$ be a labeled graph, with $\ell\colon V \to \Sigma^+$, let $Q \in \Sigma^+$ be a query string, and let $\kappa > 0$ be a threshold. Given a match (i, P, j) of $Q[x..y]$ in G, we say that the pair $([x..y], (i, P, j))$ is* left-maximal *(*right-maximal*) if it* cannot *be extended to the left (right, respectively) in both Q and G, that is,*

$$\text{(LeftMax)} \quad x = 1 \ \vee\ \mathrm{lext}(i, P, j) = \emptyset \ \vee\ Q[x-1] \notin \mathrm{lext}(i, P, j) \qquad \textit{and}$$
$$\text{(RightMax)} \quad y = |Q| \ \vee\ \mathrm{rext}(i, P, j) = \emptyset \ \vee\ Q[y+1] \notin \mathrm{rext}(i, P, j).$$

We call $([x..y], (i, P, j))$ a κ-MEM iff LeftMax$\vee|\mathrm{lext}(i, P, j)| \geq 2$, RightMax$\vee|\mathrm{rext}(i, P, j)| \geq 2$, *and $y - x + 1 \geq \kappa$, meaning that it is of length at least κ, it is left-maximal or its left (graph) extension is not a singleton, and it is right-maximal or its right (graph) extension is not a singleton.*

For example, with $Q = \texttt{CACCGTAT}$, $\kappa = 0$, v being the first underlined node of Figure 1, and u being the second in-neighbor of v, then $([1..7], (5, uv, 6))$ is a MEM since it is left and right maximal. Note that pair $([2..7], (1, v, 6))$ is also a MEM since it is right-maximal, and the left extension of $(1, v, 6)$ is not a singleton ($\mathrm{lext}(v) = \{\texttt{A}, \texttt{C}\}$): this match is not left-maximal but our definition includes it as there are at least two different characters to the left.

We use this particular extension of MEMs to graphs – with the additional conditions on non-singletons lext and rext – as it captures all MEMs between Q and $\ell(P)$, where P is a source-to-sink path in G. Moreover, this MEM formulation (with $\kappa = 1$) captures LCS through co-linear chaining, whereas avoiding the additional conditions would fail [38].

The rest of this section is structured as follows. In Section 3.1, we show how to adapt the MEM finding algorithm of Belazzougui et al. [4] for the case of node MEMs, which ignore the singleton conditions of Definition 2. Then, in Section 3.2, we show how to further generalize this approach to report all κ-MEMs spanning exactly L nodes.

3.1 MEMs in Node Labels

We say that a match $([x..y], (i, v, j))$ is a *node MEM* if it is left and right maximal w.r.t. $\ell(P)$ only in the string sense. In other words, a node MEM is a (string) MEM between Q and $\ell(v)$ (especially in the case $x = 1$ or $y = \ell(v)$). For this, we consider the text

$$T_{\text{nodes}} = \prod_{v \in V} \mathbf{0} \cdot \ell(v),$$

where $\mathbf{0} \notin \Sigma$ is used as a delimiter to prevent MEMs spanning more than a node label.

Running the MEM finding algorithm of Belazzougui et al. [4, Theorem 7.10] on Q and T_{nodes} will retrieve exactly the node MEMs we are looking for. Given such a MEM (x_1, x_2, ℓ), to transform the coordinates of $T_{\text{nodes}}[x_2..x_2 + \ell - 1]$ into the corresponding graph substring (i, P, j) we augment the index with a bitvector B marking the locations of **0**s of T_{nodes}, so that $r = \texttt{rank}(B, x_2)$ identifies the corresponding node of G, $i = x_2 - \texttt{select}(B, r)$ and $j = i + \ell - 1$.

▶ **Lemma 3.** *Given a labeled graph $G = (V, E, \ell)$, with $\ell : V \to \Sigma^+$, a query string Q, and a threshold $\kappa > 0$, we can compute all node MEMs of length at least κ between Q and G in time $O(n + m + N_\kappa)$, where n is the total length of node labels, $m = |Q|$, and N_κ is the number of output MEMs.*

Proof. The claim follows by applying the MEM finding algorithm of Belazzougui et al. [4, Theorem 7.10] on T_{nodes} and Q, and using B to extract the corresponding graph matches. We include a brief explanation of the original algorithm (following the later adaptation of [28, Algorithm 11.3]) as we will modify this algorithm later. In our context, the algorithm takes as inputs the bidirectional BWT index on $T_{\text{nodes}}\$$ and the bidirectional BWT index on $Q\$$, and produces MEM strings Q', with $|Q'| \geq \kappa$, and four BWT intervals $[i_G..j_G]$, $[i'_G..j'_G]$, $[i_Q..j_Q]$, and $[i'_Q..j'_Q]$, for each such Q'. The first two BWT intervals are such that Q' occurs in T_{nodes} $j_G - i_G + 1 = j'_G - i'_G + 1$ times: the suffixes of T_{nodes} having Q' as a prefix have lexicographic ranks between i_G and j_G, and the prefixes of T_{nodes} having $\overline{Q'}$ (the reverse of Q') as a suffix have co-lexicographic (lexicographic of the reverse) ranks between i'_G and j'_G. Analogously, the last two BWT intervals are such that Q' occurs in Q $j_Q - i_Q + 1 = j'_Q - i'_Q + 1$ times: the suffixes of Q having Q' as a prefix have lexicographic ranks between i_Q and j_Q, and the prefixes of Q having $\overline{Q'}$ as a suffix have co-lexicographic ranks between i'_Q and j'_Q. For each of these four BWT intervals reported, the algorithm runs a *cross-product* routine outputting all MEMs whose MEM string is Q'. Globally, the algorithm is linear in both the input and output size, since the exploration of MEM strings Q' takes linear time in the size of the input strings, and since in total the cross-product routine runs in linear time in the number of BWT intervals considered as well as in the number of output MEMs [4, Theorem 7.10]. To see the linearity with respect to the input strings, let us study how the MEM strings Q' are explored. The algorithm starts with intervals covering all suffixes/prefixes corresponding to a match of the empty string. It then executes recursively, so we can consider a generic step with intervals $[i_G..j_G]$, $[i'_G..j'_G]$, $[i_Q..j_Q]$, and $[i'_Q..j'_Q]$ corresponding to a match of the string Q'. Moreover, the algorithm maintains the invariant that the current match Q' is right-maximal. In the recursive step, the algorithm first checks whether the match is left-maximal, in which case it reports the corresponding MEMs using the aforementioned cross-product algorithm. It then extends the match to the left with every possible character extension. For each such extension aQ', it checks whether the extension contains a right-maximal match: if this is not the case the suffix tree of $T_{\text{nodes}}\#Q$ ($\# \notin \Sigma$) does not have an internal node corresponding to aQ', as all suffixes starting with aQ' continue with the same symbol, otherwise (the extension contains a right-maximal match) the suffix tree has an internal node corresponding to aQ'. This exploration is bounded by the number of *implicit Weiner links* in the suffix tree of $T_{\text{nodes}}\#Q$, which is linear in the input length [4, Observation 1]. ◀

3.2 MEMs Spanning Exactly L Nodes

Given a threshold κ, we want to find all κ-MEMs $([x..y], (i, P, j))$ spanning exactly L nodes in G, that is, $|P| = L$. Note that the MEMs obtained for $L = 1$ are a subset of the ones obtained in Lemma 3: for a node MEM $([x..y], (i, v, j))$ it might hold that $i = 1$ and $\{Q[x-1]\} = \text{lext}(1, v, j)$, or that $j = \|v\|$ and $\{Q[y+1]\} = \text{rext}(i, v, j)$.

As per Definition 2, MEMs cannot be recognized without looking at the context of the paths in G (sets lext and rext). With this in mind, we consider the text

$$T_L \coloneqq \mathbf{0} \cdot \prod_{(u_1,\ldots,u_L)\in\mathcal{P}_G^L} \Big(\text{left}(u_1)\cdot\ell(u_1)\cdots\ell(u_L)\cdot\text{right}(u_L)\cdot\mathbf{0}\Big), \tag{1}$$

where $\text{left}(u) = c$ when $\text{lext}(u) = \{c\}$ and otherwise $\text{left}(u) = \#$, $\text{right}(u) = d$ when $\text{rext}(u) = \{d\}$ and otherwise $\text{right}(u) = \#$, $\mathbf{0} \neq \#$, $\mathbf{0}, \# \notin \Sigma$, and

$$\mathcal{P}_G^L \coloneqq \left\{ P \;\middle|\; \begin{array}{c} P \text{ path of } G, \\ |P| = L \end{array} \right\}.$$

We have added the unique left- and right-extension symbols c and d to avoid reporting exact matches that can potentially be extended to longer paths. When these extensions are not unique (or empty), one can safely report a MEM, since there is a path diverting with a symbol different from that of the pattern (or the path cannot be extended further). With these left- and right-extension symbols, it suffices to modify the MEM finding algorithm of Section 3.1 to use some extra information regarding the starting position of each suffix inside string the $\ell(P)$, as explained next.

To avoid reporting MEMs spanning less than L nodes (only if $L > 1$), we use an array $D[1..|T_L|]$ such that $D[k] = \infty$ if the k-th suffix $T_L[s..|T_L|]$ of T_L in the lexicographic order is such that $T_L[s+1..|T_L|]$ is not starting inside node u_1 of a path $P = u_1 \cdots u_L$, otherwise $D[k] = |\ell(P)| - \ell(u_L) - i + 2$, where suffix $T_L[s+1..|T_L|]$ starts at position i inside u_1. That is, when $D[k] \neq \infty$, it tells the distance of the k-th suffix of T_L in the lexicographic order to the start of the last node of the corresponding path. With the help of Lemma 1 on D, we can then adapt the MEM finding algorithm to output suffixes corresponding to MEMs spanning exactly L nodes as follows.

▶ **Lemma 4.** *Let alphabet Σ be of constant size. Given a labeled graph $G = (V, E, \ell)$, a pattern $Q \in \Sigma^m$, a threshold $\kappa \geq 1$, and an integer $L \geq 1$, we can compute an encoding of all MEMs of length at least κ and spanning exactly L nodes of G in time $O(m + |T_L| + M_{\kappa,L})$. Here, T_L is defined as in Equation (1) and $M_{\kappa,L}$ is the number of output MEMs.*

Proof. We build the bidirectional BWT indexes for $T_L\$$ and $Q\$$, the suffix array of $T_L\$$, and preprocess $D[1..|T_L|]$ as in Lemma 1 in time $O(|T_L| + |Q|)$. We also preprocess, in linear time, a bitvector B marking the locations of $\mathbf{0}$s of T_L so that we can map in constant time a position i in T_L to the r-th path appended to T_L for $r = \texttt{rank}(B, i)$.

The only modification of [4, Theorem 7.10] required to only output MEMs spanning exactly L nodes (and only if $L > 1$) is to change its very last step when considering a MEM candidate Q' with $|Q'| \geq \kappa$. Namely, the cross-product routine loops over all characters $a, b, c, d \in \Sigma \cup \{\#\}$ with $a \neq c$ and $b \neq d$, such that $aQ'b$ is a substring of Q and $cQ'd$ is a substring of T_L. It then computes (in constant time) the intervals $[i_{aQ'b}..j_{aQ'b}]$, $[i'_{aQ'b}..j'_{aQ'b}]$, $[i_{cQ'd}..j_{cQ'd}]$, and $[i'_{cQ'd}..j'_{cQ'd}]$, where the first two are the intervals in the bidirectional BWT on T_L corresponding to $aQ'b$ and the latter two are the intervals in the bidirectional BWT on Q corresponding to $cQ'd$. After that, it outputs a triple $(k, k', |Q'|)$ representing each MEM, where $k \in [i_{aQ'b}..j_{aQ'b}]$ and $k' \in [i_{cQ'd}..j_{cQ'd}]$. It suffices to modify the first iteration using Lemma 1 to loop only over $k \in [i_{aQ'b}..j_{aQ'b}]$ such that $D[k] \leq |Q'| + 1$. Our claims are that the running time stays linear in the input and output size on constant-size alphabet, and that only MEMs spanning exactly L nodes are output. The latter claim follows directly on how array D is defined and used with Lemma 1. For the former claim, the cross-product part of the original algorithm is linear in the output size (also on non-constant-size alphabet)

since for each combination of left- and right-extension considered, the work can be charged to the output. In our case, due to the use of Lemma 1, some combinations may lead to empty outputs introducing an alphabet-factor (constant) multiplier on the input length. ◀

▶ Remark 5. Note that the algorithm in Lemma 4 works in time $O(m + n \cdot L \cdot d^{L-1} + M_{\kappa,L})$, where $n = \sum_{v \in V} \ell(v)$ is the total label length of G and d is the maximum in- or out-degree of a node. Indeed, T_L corresponds to the concatenation of length-L paths of G: the number of paths containing label $\ell(v)$ (for a node v) is at most $L \cdot d^{L-1}$.

4 MEMs in Elastic Founder Graphs

The approach of Section 3.2 is exponential on L, so we can only use it for constant L if aiming for a poly-time MEM finding routine. For general labeled graphs, this may be the best achievable, as we cannot find MEMs with a threshold κ between a pattern Q and G in truly sub-quadratic time unless the *Orthogonal Vectors Hypothesis* (OVH) is false [15], and exponential-time indexing is required for truly sub-quadratic MEM finding with threshold κ unless OVH is false [14]: finding MEMs between Q and G finds matches of Q in G, and if $\kappa = |Q|$ MEM finding is exactly equivalent to matching a pattern in a labeled graph. To have a poly-time indexing of G that can solve MEM finding in truly sub-quadratic time, it is necessary to constrain the family of graphs in question. Therefore, we now focus on indexable Elastic Founder Graphs (indexable EFGs), that are a subclass of labeled directed acyclic graphs (labeled DAGs) having the feature that they support poly-time indexing for linear-time queries [16]. We will show that the same techniques used to query if Q appears in indexable EFG G can be extended to solve MEM finding on G with arbitrary length threshold κ.

▶ **Definition 6** (Elastic Founder Graph [16])**.** *Consider a* block graph $G = (V, E, \ell)$*, where* $\ell\colon V \to \Sigma^+$*,* V *is partitioned into* k blocks $V_1, \ldots, V_k$*, and edges* $(u, v) \in E$ *are such that* $u \in V_i$*,* $v \in V_{i+1}$ *for some* $i \in [1..k-1]$*. We say that a block graph is an* indexable Elastic Founder Graph *(indexable EFG) if the* semi-repeat-free property *holds: for each* $v \in V_i$*,* $\ell(v)$ *occurs in* G *only as prefix of paths starting with some* $w \in V_i$*.*

Note that the semi-repeat-free property allows a node label to be prefix of other node labels in the same block, whereas it forbids them to appear as a proper suffix of other node labels nor anywhere else in the graph. Indexable EFGs can be obtained from a set of aligned sequences, in a way such that the resulting indexable EFG spells the sequences but also their *recombination*: for a gapless alignment, we can build in time linear to the size of the alignment an optimal indexable EFG with minimum height H of a block, where the height of block V_i is defined as $|V_i|$, solution generalized to the case with gaps by using an alternative height definition [40].

Let us now consider MEM finding with threshold κ on an indexable EFG $G = (V, E, \ell)$. We can use the general κ-MEM finding algorithm of Lemma 4 between Q and G spanning exactly L nodes, with $L = 1, 2, 3$; then, we find κ-MEMs that span longer paths with a solution specific to indexable EFGs. To find all MEMs between Q and G spanning more than three nodes, we index

$$T'_3 \coloneqq \mathbf{0} \cdot \prod_{(u,v),(v,w) \in E} \Big(\ell(u)\ell(v)\ell(w)\mathbf{0}\Big) \qquad \text{where } \mathbf{0} \notin \Sigma.$$

Equi et al. [16] showed that the suffix tree of T'_3 can be used to query string Q in G, taking time $O(|Q|)$. We now extend this algorithm to find MEMs between Q and indexable EFG G with threshold κ and spanning more than 3 nodes. For simplicity, we describe a solution for case $\kappa = 1$ and later argue case $\kappa > 1$.

First, we augment the suffix tree of T'_3:

- we mark all implicit or explicit nodes $\overline{p}$ such that the corresponding root-to-$\overline{p}$ path spells $\ell(u)\ell(v)$ for some $(u,v) \in E$, so that we can query in constant time if $\overline{p}$ is such a node;
- we compute pointers from each node $\overline{p}$ to an arbitrarily chosen leaf in the subtree rooted at $\overline{p}$;
- for each node $v \in V$ of the indexable EFG we build trie T_v for the set of strings $\{\overline{\ell(u)} : (u,v) \in E\}$;
- for each leaf, we store the corresponding path uvw and the starting position of the suffix inside $\ell(u)\ell(v)\ell(w)$.

▶ **Observation 7** ([16, Lemma 9]). *Given an indexable EFG $G = (V, E, \ell)$, for each $(v, w) \in E$ string $\ell(v)\ell(w)$ occurs only as prefix of paths starting with v. Thus, all occurrences of some string S in G spanning at least four nodes can be decomposed as $\alpha\ell(u_2)\cdots\ell(u_{L-1})\beta$ such that: (i) $u_2\cdots u_{L-1}$ is a path in G and $u_2, \ldots, u_{L-1}$ are unequivocally identified; (ii) $\alpha = \ell(u_1)[i..\|u_1\|]$ with $1 \le i \le \|u_1\|$ for some $(u_1, u_2) \in E$; and (iii) $\beta = \ell(u_L)$ for some $(u_{L-1}, u_L) \in E$ or $\beta = \ell(u_L)(\ell(u_{L+1})[1..j])$ with $1 \le j < \|u_{L+1}\|$ for some $(u_{L-1}, u_L), (u_L, u_{L+1}) \in E$. Note that $\alpha, \beta \neq \varepsilon$ and β has as prefix a full node label, whereas α might spell any suffix of a node label.*

The strategy to compute long MEMs between Q and G is to first consider, with a left-to-right scan of Q, all MEMs $([x..y], (i, P, j))$ such that:

(i) $|P| > 3$;
(ii) they satisfy conditions LeftMax and RightMax of Definition 2; and
(iii) are maximal with respect to substring $Q[x..y]$, that is, there is no other MEM $([x'..y'], (i', P', j'))$ with $x \le x' \le y' \le y$.

Next, we will describe how to modify our solution to compute also all the other MEMs spanning more than 3 nodes. Due to Observation 7, if $\alpha\ell(u_2)\cdots\ell(u_{L-1})\beta$ is a decomposition of $Q[x..y]$, all MEMs $([x'..y'], (i', P', j'))$ with $x \le x' < y' \le y$ spanning more than 3 nodes are constrained to involve some u_i with $i \in [2..L-1]$.

Consider the following modification of [16, Theorem 8] that matches $Q[1..y]$ in G. Let $\overline{p}$ be the suffix tree node of T'_3 reached from the root by spelling $Q[1..y]$ in the suffix tree until we cannot continue with $Q[y+1]$:

1. If we cannot continue with **0**, $Q[1..y]$ is part of some MEM between Q and G spanning at most 3 nodes, so we ignore it, take the suffix link of $\overline{p}$ and consider matching $Q[2..y]$ in G.
2. If we can continue with **0** and the occurrences of $Q[1..y]$ span at most two nodes in G, then we also take the suffix link of $\overline{p}$ and consider matching $Q[2..y]$. Thanks to the semi-repeat-free property, we can check this condition by retrieving any leaf in the subtree rooted at node $\overline{p}_{\mathbf{0}}$, reached by reading **0** from $\overline{p}$.
3. In the remaining case, $Q[1..y] = \alpha\ell(u_2)\ell(u_3)$ for exactly one $u_2 \in V$, with $(u_2, u_3) \in E$, due to Observation 7, and we follow the suffix link walk from $\overline{p}$ until we find the marked node $\overline{q}$ corresponding to $\ell(u_2)\ell(u_3)$: from $\overline{q}$ we try to match $Q[y+1..]$ until failure, matching $Q[y+1..y']$ and reaching node $\overline{r}$.

By repeating the suffix walk and tentative match of case **3.** until we cannot read **0** from the failing node, we find the maximal prefix $Q[1..y]$ occurring in G and its decomposition $\alpha\ell(u_2)\cdots\ell(u_{L-1})\beta$ as per Observation 7. Indeed, we can find unique nodes $u_2, \ldots, u_{L-1}$ by analyzing the (arbitrarily chosen) leaf of the subtree rooted at $\overline{q}$ in every iteration of case **3.** Moreover, we can retrieve:

- set U_1 of pairs (i, u) such that $(u, u_2) \in E$ and $\alpha = \ell(u)[i..\|u\|]$, by iterating over the leaves of $\overline{p}$;

- unique node u_L such that $(u_{L-1}, u_L) \in E$ and $\ell(u_L) = \beta$, if such u_L exists; and
- set E_L of triplets (u, u', j) such that $(u_{L-1}, u), (u, u') \in E$ and $\ell(u)\ell(u')[1..j] = \beta$.

Then $([1..y], (i, u_1 \cdot u_2 \cdots u_{L-1} \cdot u \cdot u_{L+1}, j))$ is a MEM between Q and G for all $(i, u_1) \in U_1$ and $(u, u_{L+1}, j) \in E_L$, and also $([1..y], (i, u_1 \cdot u_2 \cdots u_{L-1} \cdot u_L, \|u_L\|))$ is a MEM for all $(i, u_1) \in U_1$, if u_L exists: these MEMs satisfy conditions **(i)**, **(ii)**, and **(iii)**, and U_1, $u_2 \cdots u_{L-1}$, u_L, and E_L form a compact representation of all MEMs spelling $Q[1..y]$.

So far the procedure computes all MEMs spanning more than 3 nodes, satisfying LeftMax and RightMax, and spelling maximal $Q[1..y]$. We can extend it to find all MEMs satisfying the first two constraints and spelling any substring $Q[x..y]$, with $Q[x..y]$ maximal. Let $\hat{x}$ be the index for which we have computed MEMs spelling $Q[\hat{x}..y]$ ($\hat{x} = 1$ in the first iteration). If cases **1.** or **2.** hold, we can start to search MEMs spelling $Q[\hat{x}+1..]$ in amortized linear time, since we follow the suffix link of $\overline{p}$. If case **3.** holds, we can restart the algorithm looking for MEMs spelling $Q[\hat{x}'..y]$, where $\hat{x}' = \hat{x} + |\alpha\ell(u_2) \cdots \ell(u_{L-2})|$. We are not missing any MEM satisfying conditions **(i)**, **(ii)**, and **(iii)**: due to the semi-repeat-free property, any MEM $([x..y], (i', P', j'))$ with $\hat{x} < x < \hat{x}'$ spanning more than 3 nodes shares substring $\ell(u_k)\ell(u_{k+1})$ with the previously computed MEM, for some $k \in [2..L-3]$, and is such that $\hat{x}' < y$ since we assume **(iii)** to hold; the algorithm would have matched $Q[\hat{x}..y]$ with case **3.** in the previous iteration, leading to a contradiction. The time globally spent reading Q is still $O(|Q|)$, because each character of Q is considered at most twice.

Finally, we are ready to describe how to compute all remaining MEMs $([x..y], (i, P, j))$ between Q and indexable EFG G spanning at least 4 nodes, that is, MEMs such that condition **(i)** holds and at least one of **(ii)** and **(iii)** do not: it is easy to see that $Q[x..y]$ must be contained in the MEMs that we have already computed; also, since they span at least 4 nodes their matches must involve some of nodes $u_2, \ldots, u_{L-1}$ of MEMs satisfying **(i)**, **(ii)**, and **(iii)**. Indeed, whenever case **3.** holds and we decompose $Q[\hat{x}..y]$ as $\alpha\ell(u_2) \cdots \ell(u_{L-1})\beta$, we can find set U_{RT} of pairs (v, j), with $v \in V$ and $1 \leq j \leq \|v\|$, such that $(v, j) \in U_{\mathrm{RT}}$ iff $(i, P = u \cdot u_2 \cdots u_{b-1} \cdot v, j)$ is a match of $Q[\hat{x}..y']$ in G, with $(i, u) \in U_1$, $|P| < L$, $y' < y$, and $Q[y'+1] \notin \mathrm{rext}(1, u, j)$ – verifying RightMax and describing a MEM where **(iii)** fails – or $|\mathrm{rext}(1, u, j)| \geq 2$ – verifying the non-singleton condition of Definition 2 and describing a MEM where **(ii)** fails. We can gather all the elements of U_{RT} during each descending walk in the suffix tree of T_3', since they correspond to the leaves of subtrees of branching nodes in the tentative match of $Q[\hat{x}..y]$. Analogously, we can find set U_{LT} of pairs (i, v), with $v \in V$ and $1 \leq i \leq \|v\|$, such that $(i, v) \in U_{\mathrm{LT}}$ iff $(i, v) = (1, u_i)$ for $2 \leq i \leq L-1$ and $\mathrm{lext}(u_i) \geq 2$, or $(i, P = v \cdot u_b \cdots u_{L-1}, \|u_{L-1}\|)$ is a match of $Q[x..y - |\beta|]$ in G, with $x > \hat{x}$ and $Q[x-1] \notin \mathrm{lext}(i, v, \|v\|)$. We can compute U_{LT} by analyzing the leaves of subtrees of branching nodes in the walk in T_{u_i} spelling $\overline{\ell(u_i)}$, with $2 \leq i \leq L-1$. Sets U_1, $u_2 \cdots u_{L-1}$, u_L, E_L, U_{LT} and U_{RT} are a compact representation of all MEMs spanning at least 4 nodes and involving (any substring of) $Q[\hat{x}..y]$: a cross-product-like algorithm that matches elements of U_1 or L with elements of u_L, E_L, or U_{RT}, joined by the relevant part of $u_2 \cdots u_{L-1}$, can explicitly output the MEMs spanning more than 3 nodes in linear time with respect to the size of the output, by exploiting the fact that U_{LT} and U_{RT} are computed and ordered block by block.

▶ **Theorem 8.** *Let alphabet Σ be of constant size, and let $G = (V, E, \ell)$ be an indexable Elastic Founder Graph of height H, that is, the maximum number of nodes in a block of G is H. An encoding of MEMs between query string $Q \in \Sigma^m$ and G with arbitrary length threshold κ can be reported in time $O(nH^2 + m + M_\kappa)$, where $n = \sum_{v \in V} \|v\|$ and M_κ is the number of MEMs of interest.*

Proof. We can apply the algorithm of Lemma 4 to find κ-MEMs spanning L nodes, with $L = 1, 2, 3$, taking time $O(|Q| + |T_1| + |T_2| + |T_3| + M_{\kappa,1} + M_{\kappa,2} + M_{\kappa,3})$.

Let $M_{\kappa,4+}$ be the number of MEMs satisfying threshold κ and spanning at least 4 nodes in G. The suffix tree of T'_3 can be constructed in time $O(|T'_3|)$ and the described modification of a descending suffix walk on Q takes constant amortized time per step, assuming constant-size alphabet. The time spent gathering U_1, $u_2 \cdots u_{L-1}$, E_L, U_{LT}, and U_{RT}, forming an encoding of the MEMs involving $Q[\hat{x}..y]$, can be charged to $M_{\kappa,4+}$ because each element of U_1, E_L, U_{LT}, and U_{RT} corresponds to one or more MEMs, that could be retrieved in an explicit form with a cross-product-like procedure. Indeed: for U_1 we can retrieve all leaves of the subtree rooted at $\overline{p}$ of the suffix tree of T'_3; for E_L and U_{RT}, we can do the same for node $\overline{r}$ reached by the last tentative match of $Q[y+1..]$, and for branching nodes reached during every tentative match; for U_{LT}, using a compact trie and *blind search* [18] in the representation of each T_u allows to compare only the branching symbols. Finally, it is easy to see that in case **3.**, after we decomposed $|Q[\hat{x}..y]|$ as $\alpha\ell(u_2)\cdots\ell(u_{L-1})\beta$ as in Observation 7, we know the length of strings α, $\ell(u_2)$, …, $\ell(u_{L-1})$, and β, so we can postpone the computation of sets U_1, E_L, U_{LT}, and U_{RT} and avoid computing MEMs of length smaller than κ. Thus, finding an encoding of all MEMs between Q and G with threshold κ and spanning more than 3 nodes takes $O(|Q| + |T'_3| + M_{\kappa,4+})$ time.

The stated time complexity is reached due to the fact that $|T_3|$ dominates $|T'_3|$, $|T_2|$, and $|T_1|$, and for indexable EFGs $|T_3| \in O(nH^2)$, since every character of every node label $\ell(u)$ gets repeated at most H^2 times, which is an upper bound on the number of paths of length 3 containing u. ◀

▶ **Corollary 9.** *The results of Lemmas 3 and 4 and Theorem 8 hold when query $Q[1..m]$ is replaced by a set of queries of total length m. The respective algorithms can be modified to report MEMs between the graph and each query separately.*

Proof. Consider a concatenation $Q = Q^1\$Q^2\$\cdots Q^d$ of d query sequences, where \$ is a unique symbol not occurring in the queries nor in the graph. No MEM can span over such unique separator and hence the MEMs between graph G and concatenation Q are the same as those between G and each Q^i. It is thus sufficient to feed concatenation Q as input to the algorithms and project each resulting MEM to the corresponding query sequence. ◀

▶ **Corollary 10.** *The algorithms of Lemmas 3 and 4, Theorem 8, and Corollary 9 can be modified to report only MEMs that occur in text T formed by concatenating the rows (ignoring gaps and adding separator symbols) of the input MSA of the indexable EFG. This can be done in additional $O(|T| + r \log r)$ time and $O(r \log n)$ bits of space, and with multiplicative factor $O(\log \log n)$ added to the running times of the respective algorithms, where r is the number of equal-letter runs in the BWT of T.*

Proof. Lemma 3 does not need any modification as the node labels are automatically substrings of T. Same applies for the edge labels, but for longer paths we need to make sure we do not create combinations not supported by T. This can be accomplished with the help of the r-index: with the claimed time and space one can build the run-length encoded BWT of T [35] and the associated data structures to form the counting version of the r-index that supports backward step in $O(\log \log n)$ time [20]. As we concatenate paths consisting of L nodes for MEM finding in Lemma 4, we can first search them using the r-index, and only include them if they occur in T. MEMs spanning more than 3 nodes in Theorem 8 and Corollary 9 can be searched afterwards with the r-index to filter out those MEMs not occurring in T; these MEMs cannot mutually overlap each other in Q by more than one full node label, so the running time of the verification can be charged on the size of the elastic founder graph. ◀

5 Experiments

The benefit of Corollary 10 over the mere use of r-index for MEM finding [42] is that a MEM can occur many times in a repetitive collection while the occurrences starting at the same column of a MSA of the collection can be represented by a small number of paths in the indexable Elastic Founder Graph.

To test this hypothesis, we implemented the MEM finding algorithm using the bidirectional r-index [2]. The code is available at `https://github.com/algbio/br-index-mems`. The implementation works both for MEM finding between two sequences and between a sequence and a graph. In the case of a graph, the implementation covers the algorithms of Sections 3.1 and 3.2 up to paths of length 3 nodes. If the input graph is an indexable EFG and κ is at most the length of the shortest string spelled by an edge, the implementation outputs all κ-MEMs, yet some longer MEMs are not fully extended and/or may be reported in several pieces.[1] With these considerations, the implementation gives an upper bound on the number of κ-MEMs in the case of graphs, but it provides the exact number of MEMs in the case of two sequences as input.

We performed experiments with the same multiple sequence alignment (MSA) of covid19 strains as in [29]. We first filtered out strains whose alignments had a run of gaps of length more than 100 bases.[2] Then we extracted a sub-MSA of 100 random strains from the remaining and extracted MSAs of the first 20, 40, 60, and 80 strains from this MSA of 100 strains. For each such dataset, we built the bidirectional r-index of the sequences (without gaps) and the indexable EFG of the MSA. The latter were constructed using the tool `https://github.com/algbio/founderblockgraphs` with parameters `--elastic --gfa`.

We used $\kappa = 12$ in all experiments, as this was the length of the shortest string spelled by an edge over all indexable EFGs. For the queries, we extracted 1000 substrings of length 100 from the first 20 strains. For each query, we selected two random positions and mutated them with equal probability for A, C, G, or T. The queries were then concatenated into a long sequence and the bidirectional r-index was built on it as described by Corollary 9. The MEMs were computed between the queries and the respective text/graph index.

The number of MEMs for each index is reported in Figure 2 and the number of runs in the two Burrows–Wheeler transforms of each index is reported in Figure 3. As can be seen from the results, the number of MEMs is greatly reduced when indexing the graph compared to indexing the collection of strains. In this setting, almost all graph MEMs have a counterpart in the collection: we implemented also a filtering mode analogous to Corollary 10 so that only graph MEMs occurring in the original strains are reported. The number of path MEMs (and thus total number of MEMs) reported dropped by 92, 97, 90, 91, and 91 for the case of indexable EFG on 20, 40, 60, 80, and 100 strains, respectively. That is, less than 2% of the reported graph MEMs were recombinants of the input strains. However, since the implementation outputs long MEMs in pieces, this percentage may be higher for full-length graphs MEMs.

The number of runs (the major factor affecting the space used by the indexes) for the bidirectional r-index of the collection and that of the concatenation of node labels are comparable. For edges and especially for paths of length 3 nodes the number of runs is

[1] In an earlier version of this paper, we mistakenly set κ as the minimum length string spelled by paths spanning 3 nodes, but we later observed that this may lead the implementation to miss some κ-MEMs.

[2] We used this filter to address a current limitation of the MSA segmentation algorithms for indexable EFG, where a long gap at the beginning or end of an MSA row hurts the effectiveness of the method.

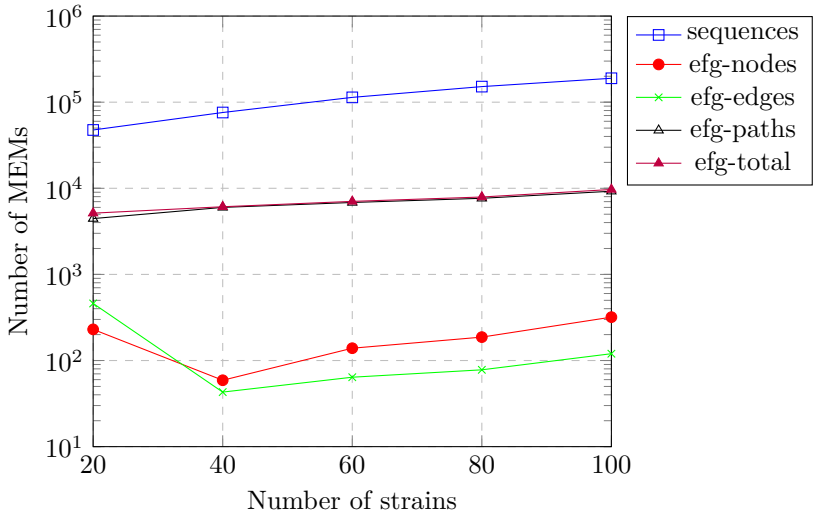


Figure 2 Number of MEMs with different indexes and varying number of covid19 strains. Here sequences refers to bidirectional r-index. For indexable EFG the results are shown for nodes, edges, and paths of length 3 nodes (these also include longer MEMs counted multiple times). Line efg-total is the total number of EFG MEMs. Note the logarithmic scale on the y-axis.

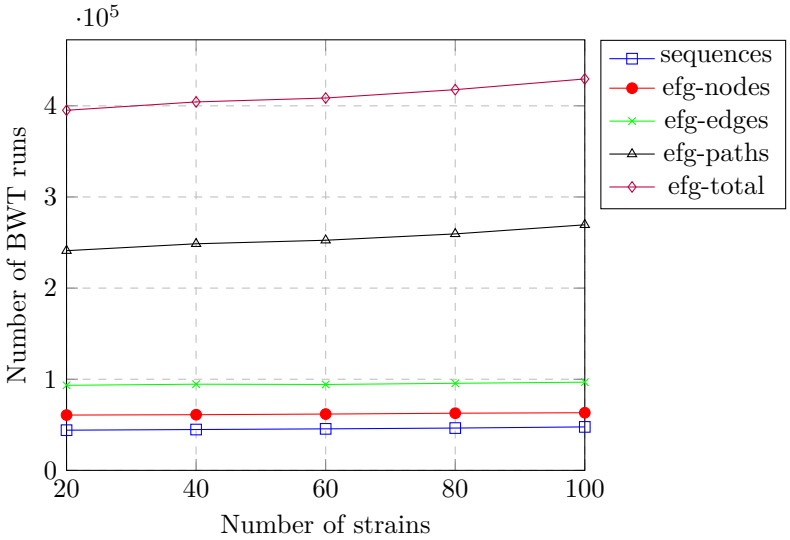


Figure 3 Number of BWT runs with different indexes and varying number of covid19 strains. Here sequences refers to the bidirectional r-index, labels efg-nodes, efg-edges, efg-paths refer to the concatenation of node labels, paths of length 2 and paths of length 3, respectively. Label efg-total is the sum of the previous three numbers of runs.

significantly higher. Fortunately, the growth of these metrics when more strains are added is limited. This is not surprising, as the stains are highly similar and thus the added information content is limited and known to be correlated with the number of BWT runs [30].

Running times correlate with the index size comparison: with 100 strains, MEM finding using the the bidirectional r-index took 20 seconds, node MEM finding on the indexable EFG took 31 seconds, edge MEM finding 97 seconds, and path MEM finding 217 seconds. The running times were measured on a server with Intel Xeon 2.9 Ghz processor and exclude index construction, which took 1 second for the patterns, 11 seconds for the bidirectional r-index of 100 strains, and 13 seconds for the additional indexes required by the corresponding indexable EFG. The construction of the indexable EFG on 100 strains took 10 seconds. To speed-up MEM finding, we also tested switching the bidirectional r-index to a wavelet tree implementation of bidirectional BWT (`https://https://github.com/jnalanko/BD_BWT_index`). On the same 100 strains indexable EFG, the total time for MEM finding was 96 seconds, including index construction, which means 3.7 speed-up over the bidirectional r-index-based implementation.

6 Discussion

An alternative strategy to achieve the same goal as in our experiments is to encode the graph as an aggregate over the collection, apply MEM finding on the r-index, and report the distinct aggregate values on lexicographic MEM ranges to identify MEM locations in the graph [21]. This approach is not comparable to ours directly, as the compressibility of the aggregates depends on the graph properties, and the indexable Elastic Founder Graph's size has not been analyzed with respect to r. Also, the two approaches use different MEM definitions. Our Definition 2 is symmetric and local, while the version used in earlier work with the r-index [42, 21] is asymmetric and semi-global: they define a MEM as a substring of a query that occurs in the text, but its query extensions do not appear in the text. For the purpose of chaining, only the symmetric definition yields connections to the Longest Common Subsequence problem [38]. For completeness, our implementation (`https://github.com/algbio/br-index-mems`) also supports this asymmetric MEM definition; our algorithms can be simplified for this case.

We did not implement the general suffix tree-based approach to handle arbitrary long MEMs. From the experiments, we can see that already the case of paths of length 3 nodes handled by the generic MEM finding routine causes some scalability issues, and the suffix tree-based approach uses even more space. In our recent work [39], we have solved pattern search in indexable EFGs using only edges, and our aim is to extend that approach to work with MEMs, so that the whole mechanism could work on top of a plain bidirectional r-index. Some of our results assume a constant-size alphabet. This assumption can be relaxed with additional data structures, but also a more careful amortized analysis may lead to better bounds.

References

1 Mohamed Ibrahim Abouelhoda. A chaining algorithm for mapping cdna sequences to multiple genomic sequences. In Nivio Ziviani and Ricardo A. Baeza-Yates, editors, *String Processing and Information Retrieval, 14th International Symposium, SPIRE 2007, Santiago, Chile, October 29-31, 2007, Proceedings*, volume 4726 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2007. `doi:10.1007/978-3-540-75530-2_1`.

2 Yuma Arakawa, Gonzalo Navarro, and Kunihiko Sadakane. Bi-directional r-indexes. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPIcs*, pages 11:1–11:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CPM.2022.11.

3 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.*, 47(3):1087–1097, 2018. doi:10.1137/15M1053128.

4 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *ACM Trans. Algorithms*, 16(2):17:1–17:54, 2020. doi:10.1145/3381417.

5 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 79–97. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.15.

6 M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

7 Manuel Caceres. Parameterized algorithms for string matching to DAGs: Funnels and beyond. *arXiv preprint*, 2022. To appear in the proceedings of CPM 2023. arXiv:2212.07870.

8 Ghanshyam Chandra and Chirag Jain. Sequence to graph alignment using gap-sensitive co-linear chaining. In Haixu Tang, editor, *Research in Computational Molecular Biology*, pages 58–73, Cham, 2023. Springer Nature Switzerland.

9 David Clark. *Compact pat trees*. PhD thesis, University of Waterloo, 1997.

10 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, October 2016. doi:10.1093/bib/bbw089.

11 Nicola Cotumaccio. Graphs can be succinctly indexed for pattern matching in $O(|E|^2 + |V|^{5/2})$ time. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Data Compression Conference, DCC 2022, Snowbird, UT, USA, March 22-25, 2022*, pages 272–281. IEEE, 2022. doi:10.1109/DCC52660.2022.00035.

12 Nicola Cotumaccio and Nicola Prezza. On indexing and compressing finite automata. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10–13, 2021*, pages 2585–2599. SIAM, 2021. doi:10.1137/1.9781611976465.153.

13 Rene De La Briandais. File searching using variable length keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), pages 295–298, New York, NY, USA, 1959. Association for Computing Machinery. doi:10.1145/1457838.1457895.

14 Massimo Equi, Veli Mäkinen, and Alexandru I. Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. In Tomás Bures, Riccardo Dondi, Johann Gamper, Giovanna Guerrini, Tomasz Jurdzinski, Claus Pahl, Florian Sikora, and Prudence W. H. Wong, editors, *SOFSEM 2021: Theory and Practice of Computer Science – 47th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2021, Bolzano-Bozen, Italy, January 25-29, 2021, Proceedings*, volume 12607 of *Lecture Notes in Computer Science*, pages 608–622. Springer, 2021. doi:10.1007/978-3-030-67731-2_44.

15 Massimo Equi, Veli Mäkinen, Alexandru I Tomescu, and Roberto Grossi. On the complexity of string matching for graphs. *ACM Transactions on Algorithms*, 19(3):1–25, 2023.

16 Massimo Equi, Tuukka Norri, Jarno Alanko, Bastien Cazaux, Alexandru I. Tomescu, and Veli Mäkinen. Algorithms and complexity on indexing founder graphs. *Algorithmica*, 85(6):1586–1623, 2023. doi:10.1007/s00453-022-01007-w.

17 Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997.

18 Paolo Ferragina and Roberto Grossi. The String B-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999. doi:10.1145/301970.301973.

19 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.

20 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. doi:10.1145/3375890.

21 Adrián Goga, Andrej Baláz, Alessia Petescia, and Travis Gagie. MARIA: Multiple-alignment r-index with aggregation. *CoRR*, abs/2209.09218, 2022. doi:10.48550/arXiv.2209.09218.

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

23 Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October – 1 November 1989*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.

24 Chirag Jain, Daniel Gibney, and Sharma V. Thankachan. Co-linear chaining with overlaps and gap costs. In Itsik Pe'er, editor, *Research in Computational Molecular Biology – 26th Annual International Conference, RECOMB 2022, San Diego, CA, USA, May 22-25, 2022, Proceedings*, volume 13278 of *Lecture Notes in Computer Science*, pages 246–262. Springer, 2022. doi:10.1007/978-3-031-04749-7_15.

25 Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint*, 2013. arXiv:1303.3997.

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

27 Jun Ma, Manuel Cáceres, Leena Salmela, Veli Mäkinen, and Alexandru I. Tomescu. Chaining for accurate alignment of erroneous long reads to acyclic variation graphs. *bioRxiv*, 2022. doi:10.1101/2022.01.07.475257.

28 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.

29 Veli Mäkinen, Bastien Cazaux, Massimo Equi, Tuukka Norri, and Alexandru I. Tomescu. Linear time construction of indexable founder block graphs. 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 7:1–7:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.WABI.2020.7.

30 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010. doi:10.1089/cmb.2009.0169.

31 Veli Mäkinen and Kristoffer Sahlin. Chaining with overlaps revisited. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, volume 161 of *LIPIcs*, pages 25:1–25:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CPM.2020.25.

32 Guillaume Marçais, Arthur L. Delcher, Adam M. Phillippy, Rachel Coston, Steven L. Salzberg, and Aleksey V. Zimin. Mummer4: A fast and versatile genome alignment system. *PLoS Comput. Biol.*, 14(1), 2018. doi:10.1371/journal.pcbi.1005944.

33 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*, pages 657–666. ACM/SIAM, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545469.

34 Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

35 Takaaki Nishimoto, Shunsuke Kanda, and Yasuo Tabei. An Optimal-Time RLBWT Construction in BWT-Runs Bounded Space. In Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, volume 229 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 99:1–99:20, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2022.99.

36 Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In Edgar Chávez and Stefano Lonardi, editors, *String Processing and Information Retrieval – 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11-13, 2010. Proceedings*, volume 6393 of *Lecture Notes in Computer Science*, pages 347–358. Springer, 2010. doi:10.1007/978-3-642-16321-0_36.

37 Mikko Rautiainen and Tobias Marschall. Graphaligner: rapid and versatile sequence-to-graph alignment. *Genome biology*, 21(1):1–28, 2020.

38 Nicola Rizzo, Manuel Cáceres, and Veli Mäkinen. Chaining of maximal exact matches in graphs. Manuscript in submission. doi:10.48550/arXiv.2302.01748.

39 Nicola Rizzo, Massimo Equi, Tuukka Norri, and Veli Mäkinen. Elastic founder graphs improved and enhanced. *CoRR*, abs/2303.05336, 2023. Submitted manuscript. doi:10.48550/arXiv.2303.05336.

40 Nicola Rizzo and Veli Mäkinen. Indexable elastic founder graphs of minimum height. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPIcs*, pages 19:1–19:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CPM.2022.19.

41 Nicola Rizzo, Alexandru I. Tomescu, and Alberto Policriti. Solving string problems on graphs using the labeled direct product. *Algorithmica*, 84(10):3008–3033, 2022. doi:10.1007/s00453-022-00989-x.

42 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. doi:10.1089/cmb.2021.0445.

43 Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Inf. Comput.*, 213:13–22, 2012. doi:10.1016/j.ic.2011.03.007.

44 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

45 Michaël Vyverman, Bernard De Baets, Veerle Fack, and Peter Dawyndt. essamem: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinform.*, 29(6):802–804, 2013. doi:10.1093/bioinformatics/btt042.

46 Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

Revisiting the Complexity of and Algorithms for the Graph Traversal Edit Distance and Its Variants

Yutong Qiu[1]
Computational Biology Department, Carnegie Mellon University, Pittsburgh, PA, USA

Yihang Shen[1]
Computational Biology Department, Carnegie Mellon University, Pittsburgh, PA, USA

Carl Kingsford[2]
Computational Biology Department, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

The graph traversal edit distance (GTED), introduced by Ebrahimpour Boroojeny et al. (2018), is an elegant distance measure defined as the minimum edit distance between strings reconstructed from Eulerian trails in two edge-labeled graphs. GTED can be used to infer evolutionary relationships between species by comparing de Bruijn graphs directly without the computationally costly and error-prone process of genome assembly. Ebrahimpour Boroojeny et al. (2018) propose two ILP formulations for GTED and claim that GTED is polynomially solvable because the linear programming relaxation of one of the ILPs will always yield optimal integer solutions. The claim that GTED is polynomially solvable is contradictory to the complexity of existing string-to-graph matching problems.

We resolve this conflict in complexity results by proving that GTED is NP-complete and showing that the ILPs proposed by Ebrahimpour Boroojeny et al. do not solve GTED but instead solve for a lower bound of GTED and are not solvable in polynomial time. In addition, we provide the first two, correct ILP formulations of GTED and evaluate their empirical efficiency. These results provide solid algorithmic foundations for comparing genome graphs and point to the direction of heuristics that estimate GTED efficiently.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Integer Linear Programming, Genome Graphs, Flow Network, Graph Comparison

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.11

Related Version *Extended Version*: https://arxiv.org/abs/2305.10577

Supplementary Material *Software*: https://github.com/Kingsford-Group/gtednewilp/
archived at swh:1:dir:8e7019bfc1f7a9e732ad616024e4e36b142c35f8

Funding This work was supported in part by the US National Science Foundation [DBI-1937540, III-2232121], the US National Institutes of Health [R01HG012470] and by the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program.

Acknowledgements Conflict of Interest: C.K. is a co-founder of Ocean Genomics, Inc.

1 Introduction

Graph traversal edit distance (GTED) [6] is an elegant measure of the similarity between the strings represented by edge-labeled Eulerian graphs. For example, given two de Bruijn assembly graphs [19], computing GTED between them measures the similarity between two genomes without the computationally intensive and possibly error-prone process of

[1] These authors contributed equally to this work.
[2] Corresponding Author

assembling the genomes. Using an estimation of GTED between assembly graphs of Hepatitis B viruses, Ebrahimpour Boroojeny et al. [6] group the viruses into clusters consistent with their taxonomy. This can be extended to inferring phylogeny relationships in metagenomic communities or comparing heterogeneous disease samples such as cancer. There are several other methods to compute a similarity measure between strings encoded by two assembly graphs [20, 15, 13, 8]. GTED has the advantage that it does not require prior knowledge on the type of the genome graph or the complete sequence of the input genomes. The input to the GTED problem is two unidirectional, edge-labeled Eulerian graphs, which are defined as:

▶ **Definition 1** (Unidirectional, edge-labeled Eulerian Graph). *A unidirectional, edge-labeled Eulerian graph is a connected directed graph $G = (V, E, \ell, \Sigma)$, with node set V, edge multi-set E, constant-size alphabet Σ, and single-character edge labels $\ell : E \to \Sigma$, such that G contains an Eulerian trail that traverses every edge $e \in E$ exactly once. The unidirectional condition means that all edges between the same pair of nodes are in the same direction.*

Such graphs arise in genome assembly problems (e.g. the de Bruijn subgraphs). Computing GTED is the problem of computing the minimum edit distance between the two most similar strings represented by Eulerian trails in each input graph. A trail in a graph is a walk that contains distinct edges and may contain repeated nodes.

▶ **Problem 1** (Graph Traversal Edit Distance (GTED) [6]). *Given two unidirectional, edge-labeled Eulerian graphs G_1 and G_2, compute*

$$GTED(G_1, G_2) \triangleq \min_{\substack{t_1 \in trails(G_1) \\ t_2 \in trails(G_2)}} edit(str(t_1), str(t_2)). \tag{1}$$

Here, $trails(G)$ is the collection of all Eulerian trails in graph G, $str(t)$ is a string constructed by concatenating labels on the Eulerian trail $t = (e_0, e_1, \ldots, e_n)$, and $edit(s_1, s_2)$ is the edit distance between strings s_1 and s_2.

Ebrahimpour Boroojeny et al. [6] claim that GTED is polynomially solvable by proposing an integer linear programming (ILP) formulation of GTED and arguing that the constraints of the ILP make it polynomially solvable. This result, however, conflicts with several complexity results on string-to-graph matching problems. Kupferman and Vardi [10] show that it is NP-complete to determine if a string exactly matches an Eulerian tour in an edge-labeled Eulerian graph. Additionally, Jain et al. [9] show that it is NP-complete to compute an edit distance between a string and strings represented by a labeled graph if edit operations are allowed on the graph. On the other hand, polynomial-time algorithms exist to solve string-to-string alignment [17] and string-to-graph alignment [9] when edit operations on graphs are not allowed.

We resolve the conflict among the results on complexity of graph comparisons by revisiting the complexity of and the proposed solutions to GTED. We prove that computing GTED is NP-complete by reducing from the Hamiltonian Path problem, reaching an agreement with other related results on complexity. Further, we point out with a counter-example that the optimal solution of the ILP formulation proposed by Ebrahimpour Boroojeny et al. [6] does not solve GTED.

We give two ILP formulations for GTED. The first ILP has an exponential number of constraints and can be solved by subtour elimination iteratively [3, 5]. The second ILP has a polynomial number of constraints and shares a similar high-level idea of the global ordering approach [5] in solving the Traveling Salesman problem [14].

In Qiu and Kingsford [21], Flow-GTED (FGTED), a variant of GTED is proposed to compare two sets of strings (instead of two strings) encoded by graphs. FGTED is equal to the edit distance between the most similar sets of strings spelled by the decomposition of flows between a pair of predetermined source and sink nodes. The similarity between the sets of strings reconstructed from the flow decomposition is measured by the Earth Mover's Edit Distance [23, 21]. FGTED is used to compare pan-genomes, where both the frequency and content of strings are essential to represent the population of organisms. Qiu and Kingsford [21] reduce FGTED to GTED, and via the claimed polynomial-time algorithm of GTED, argued that FGTED is also polynomially solvable. We show that this claim is false by proving that FGTED is also NP-complete.

While the optimal solution to ILP proposed in Ebrahimpour Boroojeny et al. [6] does not solve GTED, it does compute a lower bound to GTED that we call Closed-trail Cover Traversal Edit Distance (CCTED). We characterize the cases when GTED is equal to CCTED. In addition, we point out that solving this ILP formulation finds a minimum-cost matching between closed-trail decompositions in the input graphs, which may be used to compute the similarity between repeats in the genomes. Ebrahimpour Boroojeny et al. [6] claim their proposed ILP formulation is solvable in polynomial time by arguing that the constraint matrix of the linear relaxation of the ILP is always totally unimodular. We show that this claim is false by proving that the constraint matrix is not always totally unimodular and showing that there exists optimal fractional solutions to its linear relaxation.

We evaluate the efficiency of solving ILP formulations for GTED and CCTED on simulated genomic strings and show that it is impractical to compute GTED on larger genomes.

In summary, we revisit two important problems in genome graph comparisons: Graph Traversal Edit Distance (GTED) and its variant FGTED. We show that both GTED and FGTED are NP-complete, and provide the first correct ILP formulations for GTED. We also show that the ILP formulation proposed by Ebrahimpour Boroojeny et al. [6], i.e. CCTED, is a lower bound to GTED. We evaluate the efficiency of the ILPs for GTED and CCTED on genomic sequences. These results provide solid algorithmic foundations for continued algorithmic innovation on the task of comparing genome graphs and point to the direction of heuristics that estimate GTED efficiently.

2 GTED and FGTED are NP-complete

2.1 Conflicting results on computational complexity of GTED and string-to-graph matching

The natural decision versions of all of the computational problems described above and below are clearly in NP. Under the assumption that P $\neq$ NP, the results on the computational complexity of GTED and string-to-graph matching claimed in Ebrahimpour Boroojeny et al. [6] and Kupferman and Vardi [10], respectively, cannot be both true.

Kupferman and Vardi [10] show that the problem of determining whether a given string can be spelled by concatenating edge labels in an Eulerian trail in an input graph is NP-complete. We call this problem Eulerian Trail Equaling Word. We show in Theorem 2 that we can reduce ETEW to GTED, and therefore if GTED is polynomially solvable, then ETEW is polynomially solvable. The complete proof is in Appendix A.1.

▶ **Problem 2** (Eulerian Trail Equaling Word [10]). *Given a string $s \in \Sigma^*$, an edge-labaled Eulerian graph G, find an Eulerian trail t of G such that $str(t) = s$.*

▶ **Theorem 2.** *If GTED ∈ P then ETEW ∈ P.*

Proof sketch. We first convert an input instance $\langle s, G\rangle$ of ETEW into an input instance $\langle G_1, G_2\rangle$ of GTED by (a) creating graph G_1 that only contains edges that reconstruct string s and (b) modifying G into G_2 by extending the anti-parallel edges so that G_2 is unidirectional. We show that if $\text{GTED}(G_1, G_2) = 0$, there must be an Eulerian trail in G that spells s, and if $\text{GTED}(G_1, G_2) > 0$, G must not contain an Eulerian trail that spells s. ◀

Hence, an (assumed) polynomial-time algorithm for GTED solves ETEW in polynomial time. This contradicts Theorem 6 of Kupferman and Vardi [10] of the NP-completeness of ETEW (under P ≠ NP).

2.2 Reduction from Hamiltonian Path to GTED and FGTED

We resolve the contradiction by showing that GTED is NP-complete. The details of the proof are in Appendix A.2.

▶ **Theorem 3.** *GTED is NP-complete.*

Proof sketch. We reduce from the Hamiltonian Path problem, which asks whether a directed, simple graph G contains a path that visits every vertex exactly once. Here, simple means no self-loops or parallel edges.

Let $\langle G = (V, E)\rangle$ be an instance of Hamiltonian Path, with $n = |V|$ vertices. We first create the Eulerian closure of G, which is defined as $G' = (V', E')$ where

$$V' = \{v^{in}, v^{out} : v \in V\} \cup \{w\}. \tag{2}$$

Here, each vertex in V is split into v^{in} and v^{out}, and w is a newly added vertex. E' is the union of the following sets of edges and their labels:

- $E_1 = \{(v^{in}, v^{out}) : v \in V\}$, labeled `a`,
- $E_2 = \{(u^{out}, v^{in}) : (u, v) \in E\}$, labeled `b`,
- $E_3 = \{(v^{out}, v^{in}) : v \in V\}$, labeled `c`,
- $E_4 = \{(v^{in}, u^{out}) : (u, v) \in E\}$, labeled `c`,
- $E_5 = \{(u^{in}, w) : u \in V\}$, labeled `c`,
- $E_6 = \{(w, u^{in}) : u \in V\}$, labeled `b`.

G' is an Eulerian graph by construction but contains anti-parallel edges. We further create G'' from G' by adding dummy nodes so that each pair of antiparallel edges is split into two parallel, length-2 paths with labels `x#`, where `x` is the original label.

We also create a graph C that has the same number of edges as G'' and spells out a string

$$q = \texttt{a\#}(\texttt{b\#a\#})^{n-1}(\texttt{c\#})^{2n-1}(\texttt{c\#b\#})^{|E|+1}. \tag{3}$$

We then argue that G has a Hamiltonian path if and only if G'' spells out the string q, which uses the same line of arguments and graph traversals as in Kupferman and Vardi [10]. We then show that $\text{GTED}(G'', C) = 0$ if and only if G'' spells q. ◀

Following a similar argument, we show that FGTED is also NP-complete, and its proof is in Appendix A.3.

▶ **Theorem 4.** *FGTED is NP-complete.*

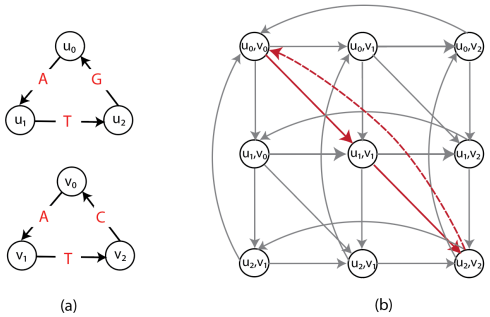


Figure 1 (a) Examples of two edge-labeled Eulerian graphs G_1 (top) and G_2 (bottom). (b) The alignment graph $\mathcal{A}(G_1, G_2)$. The cycle with red edges is the path corresponding to GTED(G_1, G_2). Red solid edges are matches with cost 0 and red dashed-line edge is mismatch with cost 1.

3 Revisiting the correctness of the proposed ILP solutions to GTED

We revisit two proposed ILP solutions to GTED by Ebrahimpour Boroojeny et al. [6] and show that the optimal solution to these ILP is not always equal to GTED.

3.1 Alignment graph

The previously proposed ILP formulations for GTED are based on the alignment graph constructed from input graphs. The high-level concept of an alignment graph is similar to the dynamic programming matrix for the string-to-string alignment problem [17].

▶ **Definition 5** (Alignment graph). *Let G_1, G_2 be two unidirectional, edge-labeled Eulerian graphs. The* alignment graph $\mathcal{A}(G_1, G_2) = (V, E, \delta)$ *is a directed graph that has vertex set $V = V_1 \times V_2$ and edge multi-set E that equals the union of the following:*
Vertical edges $[(u_1, u_2), (v_1, u_2)]$ *for* $(u_1, v_1) \in E_1$ *and* $u_2 \in V_2$,
Horizontal edges $[(u_1, u_2), (u_1, v_2)]$ *for* $u_1 \in V_1$ *and* $(u_2, v_2) \in E_2$,
Diagonal edges $[(u_1, u_2), (v_1, v_2)]$ *for* $(u_1, v_1) \in E_1$ *and* $(u_2, v_2) \in E_2$.
Each edge is associated with a cost by the cost function $\delta : E \to \mathbb{R}$.

Each diagonal edge $e = [(u_1, u_2), (v_1, v_2)]$ in an alignment graph can be projected to (u_1, v_1) and (u_2, v_2) in G_1 and G_2, respectively. Similarly, each vertical edge can be projected to one edge in G_1, and each horizontal edge can be projected to one edge in G_2.

We define the edge projection function π_i that projects an edge from the alignment graph to an edge in the input graph G_i. If the alignment edge is a vertical or horizontal edge, it is projected to one edge in only one input graph. We also define the path projection function Π_i that projects a trail in the alignment graph to a trail in the input graph G_i. For example, let a trail in the alignment graph be $p = (e_1, e_2, \ldots, e_m)$, and $\Pi_i(p) = (\pi_i(e_1), \pi_i(e_2), \ldots, \pi_i(e_m))$ is a trail in G_i.

An example of an alignment graph is shown in Figure 1(b). The horizontal edges correspond to gaps in strings represented by G_1, vertical edges correspond to gaps in strings represented by G_2, and diagonal edges correspond to the matching between edge labels from the two graphs. In the rest of this paper, we assume that the costs for horizontal and vertical edges are 1, and the costs for the diagonal edges are 1 if the diagonal edge represents a mismatch and 0 if it is a match. The cost function δ can be defined to capture the cost of matching between edge labels or inserting gaps. This definition of alignment graph is also a generalization of the alignment graph used in string-to-graph alignment [9].

3.2 The first previously proposed ILP for GTED

Lemma 1 in Ebrahimpour Boroojeny et al. [6] provides a model for computing GTED by finding the minimum-cost trail in the alignment graph. We reiterate it here for completeness.

▶ **Lemma 6** ([6]). *For any two edge-labeled Eulerian graphs G_1 and G_2,*

$$\begin{aligned} GTED(G_1, G_2) = \text{minimize}_c \quad & \delta(c) \\ \text{subject to} \quad & c \text{ is a trail in } \mathcal{A}(G_1, G_2), \\ & \Pi_i(c) \text{ is an Eulerian trail in } G_i \text{ for } i = 1, 2, \end{aligned} \tag{4}$$

where $\delta(c)$ is the total edge cost of c, and $\Pi_i(c)$ is the projection from c to G_i.

An example of such a minimum-cost trail is shown in Figure 1(b). Ebrahimpour Boroojeny et al. [6] provide the following ILP formulation and claim that it is a direct translation of Lemma 6:

$$\underset{x \in \mathbb{N}^{|E|}}{\text{minimize}} \quad \sum_{e \in E} x_e \delta(e) \tag{5}$$

$$\text{subject to} \quad Ax = 0 \tag{6}$$

$$\sum_{e \in E} x_e I_i(e, f) = 1 \quad \text{for } i = 1, 2 \text{ and for all } f \in E_i, \tag{7}$$

where

$$A_{ue} = \begin{cases} -1 & \text{if } e = (u, v) \in E \text{ for some vertex } v \in V \\ 1 & \text{if } e = (v, u) \in E \text{ for some } u \in V \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

Here, E is the edge set of $\mathcal{A}(G_1, G_2)$. A is the negative incidence matrix of size $|V| \times |E|$, and $I_i(e, f)$ is an indicator function that is 1 if edge e in E projects to edge f in the input graph G_i (and 0 otherwise). We define the domain of each x_e to include all non-negative integers. However, due to constraints (7), the values of x_e are limited to either 0 or 1. We describe this ILP formulation with the assumption that both input graphs have closed Eulerian trails, which means that each node has equal numbers of incoming and outgoing edges. We discuss the cases when input graphs contain open Eulerian trails in Section 4.

The ILP in (5)–(8) allows the solutions to select disjoint cycles in the alignment graph, and the projection of edges in these disjoint cycles need not correspond to a single string represented by either of the input graphs. We show that the ILP in (5)–(8) does not solve GTED by giving an example where the objective value of the optimal solution to the ILP in (5)–(8) is not equal to GTED.

Construct two input graphs as shown in Figure 2(a). Specifically, G_1 spells circular permutations of `TTTGAA` and G_2 spells circular permutations of `TTTAGA`. It is clear that $\text{GTED}(G_1, G_2) = 2$ under Levenshtein edit distance. On the other hand, as shown in Figure 2(a), an optimal solution in $\mathcal{A}(G_1, G_2)$ contains two disjoint cycles with nonzero x_e values that have a total edge cost equal to 0. This solution is a feasible solution to the ILP in (5)–(8). It is also an optimal solution because the objective value is zero, which is the lower bound on the ILP in (5)–(8). This optimal objective value, however, is smaller than $\text{GTED}(G_1, G_2)$. Therefore, the ILP in (5)–(8) does not solve GTED since it allows the solution to be a set of disjoint components.

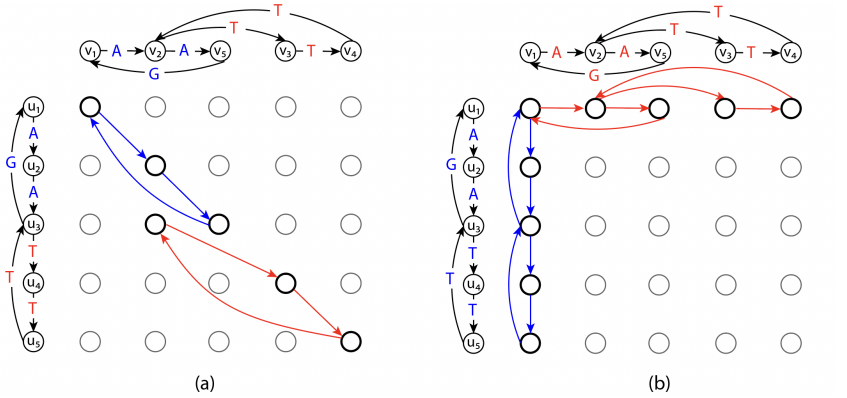


Figure 2 (a) The subgraph in the alignment graph induced by an optimal solution to the ILP in (5)–(8) and the ILP in (11)–(12) with input graphs on the left and top. The red and blue edges in the alignment graph are edges matching labels in red and blue font, respectively, and are part of the optimal solution to the ILP in (5)–(8). The cost of the red and blue edges are zero. (b) The subgraph induced by x^{init} with $s_1 = u_1$ and $s_2 = v_1$ according to the ILP in (11)–(12). The rest of the edges in the alignment graph are omitted for simplicity.

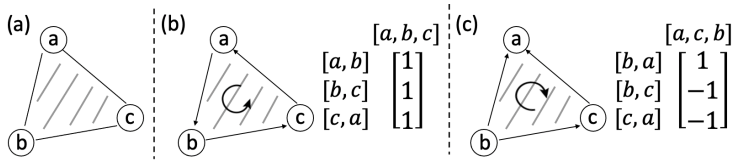


Figure 3 (a) A graph that contains an unoriented 2-simplex with three unoriented 1-simplices. (b), (c) The same graph with two different ways of orienting the simplices and the corresponding boundary matrices.

3.3 The second previously proposed ILP formulation of GTED

We describe the second proposed ILP formulation of GTED by Ebrahimpour Boroojeny et al. [6]. Following Ebrahimpour Boroojeny et al. [6], we use simplices, a notion from geometry, to generalize the notion of an edge to higher dimensions. A k-simplex is a k-dimensional polytope which is the convex hull of its $k+1$ vertices. For example, a 1-simplex is an undirected edge, and a 2-simplex is a triangle. We use the orientation of a simplex, which is given by the ordering of the vertex set of a simplex up to an even permutation, to generalize the notion of the edge direction [16, p. 26]. We use square brackets $[\cdot]$ to denote an oriented simplex. For example, $[v_0, v_1]$ denotes a 1-simplex with orientation $v_0 \to v_1$, which is a directed edge from v_0 to v_1, and $[v_0, v_1, v_2]$ denotes a 2-simplex with orientation corresponding to the vertex ordering $v_0 \to v_1 \to v_2 \to v_0$. Each k-simplex has two possible unique orientations, and we use the signed coefficient to connect their forms together, e.g. $[v_0, v_1] = -[v_1, v_0]$.

For each pair of graphs G_1 and G_2 and their alignment graph $\mathcal{A}(G_1, G_2)$, we define an oriented 2-simplex set $T(G_1, G_2)$ which is the union of:

- $[(u_1, u_2), (v_1, u_2), (v_1, v_2)]$ for all $(u_1, v_1) \in E_1$ and $(u_2, v_2) \in E_2$, or
- $[(u_1, u_2), (u_1, v_2), (v_1, v_2)]$ for all $(u_1, v_1) \in E_1$ and $(u_2, v_2) \in E_2$,

We use the boundary operator [16, p. 28], denoted by ∂, to map an oriented k-simplex to a sum of oriented $(k-1)$-simplices with signed coefficients.

$$\partial[v_0, v_1, \ldots, v_k] = \sum_{i=0}^{p} (-1)^i [v_0, \ldots, \hat{v_i}, \ldots, v_k], \tag{9}$$

where $\hat{v_i}$ denotes the vertex v_i is to be deleted. Intuitively, the boundary operator maps the oriented k-simplex to a sum of oriented $(k-1)$-simplices such that their vertices are in the k-simplex and their orientations are consistent with the orientation of the k-simplex. For example, when $k = 2$, we have:

$$\partial[v_0, v_1, v_2] = [v_1, v_2] - [v_0, v_2] + [v_0, v_1] = [v_1, v_2] + [v_2, v_0] + [v_0, v_1]. \tag{10}$$

We reiterate the second ILP formulation proposed in Ebrahimpour Boroojeny et al. [6]. Given an alignment graph $\mathcal{A}(G_1, G_2) = (V, E, \delta)$ and the oriented 2-simplex set $T(G_1, G_2)$,

$$\begin{aligned} \underset{x \in \mathbb{N}^{|E|}, y \in \mathbb{Z}^{|T(G_1,G_2)|}}{\text{minimize}} \quad & \sum_{e \in E} x_e \delta(e) \\ \text{subject to} \quad & x = x^{init} + [\partial] y \end{aligned} \tag{11}$$

Entries in x and y correspond to 1-simplices and 2-simplices in E and $T(G_1, G_2)$, respectively. $[\partial]$ is a $|E| \times |T(G_1, G_2)|$ boundary matrix where each entry $[\partial]_{i,j}$ is the signed coefficient of the oriented 1-simplex (the directed edge) in E corresponding to x_i in the boundary of the oriented 2-simplex in $T(G_1, G_2)$ corresponding to y_j. The index i, j for each 1-simplex or 2-simplex is assigned based on an arbitrary ordering of the 1-simplices in E or the 2-simplices in $T(G_1, G_2)$. An example of the boundary matrix is shown in Figure 3. $\delta(e)$ is the cost of each edge. $x^{init} \in \mathbb{R}^{|E|}$ is a vector where each entry corresponds to a 1-simplex in E with $|E_1| + |E_2|$ nonzero entries that represent one Eulerian trail in each input graph. x^{init} is a feasible solution to the ILP. Let s_1 be the source of the Eulerian trail in G_1, and s_2 be the sink of the Eulerian trail in G_2. Each entry in x^{init} is defined by

$$x_e^{init} = \begin{cases} 1 & \text{if } e = [(u_1, s_2), (v_1, s_2)] \text{ or } e = [(s_1, u_2), (s_1, v_2)], \\ 0 & \text{otherwise.} \end{cases} \tag{12}$$

If the Eulerian trail is closed in G_i, s_i can be any vertex in V_i. An example of x^{init} is shown in Figure 2(b).

We provide a complete proof in the extended version of this paper [22] that the ILP in (5)–(8) is equivalent to the ILP in (11)–(12). Therefore, the example we provided in Section 3.2 is also an optimal solution to the ILP in (11)–(12) but not a solution to GTED. Thus, the ILP in (11)–(12) does not always solve GTED.

Ebrahimpour Boroojeny et al. [6] argue that the linear programming relaxation of the ILP in (11)–(12) always yields integer optimal solutions by claiming that the constraint matrix of the LP relaxation of the ILP in (11)–(12) is always totally unimodular, and therefore the ILP in (11)–(12) can be solved in polynomial time. We provide a counterexample where the ILP in (11)–(12) yields fractional optimal solutions with fractional variable values. Additionally, we show that the constraint matrix of the LP relaxation of the ILP in (11)–(12) is not totally unimodular given most non-trivial input graphs. The details of the proofs and the counterexample are in the extended version of this paper [22] (Appendix F).

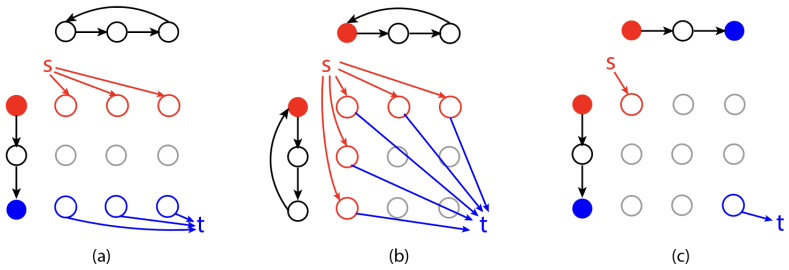


Figure 4 Modified alignment graphs based on input types. (a) G_1 has open Eulerian trails while G_2 has closed Eulerian trails. (b) Both G_1 and G_2 have closed Eulerian trails. (c) Both G_1 and G_2 have open Eulerian trails. Solid red and blue nodes are the source and sink nodes of the graphs with open Eulerian trails. "s" and "t" are the added source and sink nodes. Colored edges are added alignment edges directing from and to source and sink nodes, respectively.

4 New ILP solutions to GTED

To ensure that our new ILP formulations are applicable to input graphs regardless of whether they contain an open or closed Eulerian trail, we add a source node s and a sink node t to the alignment graph. Figure 4 illustrates three possible cases of input graphs.

1. If only one of the input graphs has closed Eulerian trails, wlog, let G_1 be the input graph with open Eulerian trails. Let a_1 and b_1 be the start and end of the Eulerian trail that have odd degrees. Add edges $[s, (a_1, v_2)]$ and $[(b_1, v_2), t]$ to E for all nodes $v_2 \in V_2$ (Figure 4(a)).
2. If both input graphs have closed Eulerian trails, let a_1 and a_2 be two arbitrary nodes in G_1 and G_2, respectively. Add edges $[s, (a_1, v_2)]$, $[s, (v_1, a_2)]$, $[(a_1, v_2), t]$ and $[(v_1, a_2), t]$ for all nodes $v_1 \in V_1$ and $v_2 \in V_2$ to E (Figure 4(b)).
3. If both input graphs have open Eulerian trails, add edges $[s, (a_1, a_2)]$ and $[t, (b_1, b_2)]$, where a_i and b_i are start and end nodes of the Eulerian trails in G_i, respectively (Figure 4(c)).

According to Lemma 6, we can solve GTED(G_1, G_2) by finding a trail in $\mathcal{A}(G_1, G_2)$ that satisfies the projection requirements. This is equivalent to finding a s-t trail in $\mathcal{A}(G_1, G_2)$ that satisfies constraints:

$$\sum_{(u,v)\in E} x_{uv} I_i((u, v), f) = 1 \quad \text{for all } (u, v) \in E, f \in G_i,\ u \neq s,\ v \neq t, \tag{13}$$

where $I_i(e, f) = 1$ if the alignment edge e projects to f in G_i, and x_{uv} is the ILP variable for edge $(u, v) \in E$. An optimal solution to GTED in the alignment graph must start and end with the source and sink node because they are connected to all possible starts and ends of Eulerian trails in the input graphs.

Since a trail in $\mathcal{A}(G_1, G_2)$ is a flow network, we use the following flow constraints to enforce the equality between the number of in- and out-edges for each node in the alignment graph except the source and sink nodes.

$$\sum_{(s,u)\in E} x_{su} = 1 \tag{14}$$

$$\sum_{(v,t)\in E} x_{vt} = 1 \tag{15}$$

$$\sum_{(u,v)\in E} x_{uv} = \sum_{(v,w)\in E} x_{vw} \quad \text{for all } v \in V \tag{16}$$

Constraints (13) and (16) are equivalent to constraints (7) and (6), respectively. Therefore, we rewrite the ILP in (5)–(8) in terms of the modified alignment graph.

$$\begin{aligned} &\underset{x\in\mathbb{N}^{|E|}}{\text{minimize}} && \sum_{e\in E} x_e\delta(e) \\ &\text{subject to} && \text{constraints (13)–(16).} \end{aligned} \tag{lower bound ILP}$$

As we show in Section 3.2, constraints (13)-(16) do not guarantee that the ILP solution is one trail in $\mathcal{A}(G_1, G_2)$, thus allowing several disjoint covering trails to be selected in the solution and fail to model GTED correctly. We show in Section 5 that the solution to this ILP is a lower bound to GTED.

According to Lemma 1 in Dias et al. [5], a subgraph of a directed graph G with source node s and sink node t is a s-t trail if and only if it is a flow network and every strongly connected component (SCC) of the subgraph has at least one edge outgoing from it. Thus, in order to formulate an ILP for the GTED problem, it is necessary to devise constraints that prevent disjoint SCCs from being selected in the alignment graph. In the following, we describe two approaches for achieving this.

4.1 Enforcing one trail in the alignment graph via constraint generation

Section 3.2 of Dias et al. [5] proposes a method to design linear constraints for eliminating disjoint SCCs, which can be directly adapted to our problem. Let $\mathcal{C}$ be the collection of all strongly connected subgraphs of the alignment graph $\mathcal{A}(G_1, G_2)$. We use the following constraint to enforce that the selected edges form one s-t trail in the alignment graph:

$$\text{If } \sum_{(u,v)\in E(C)} x_{uv} = |E(C)|, \text{ then } \sum_{(u,v)\in \varepsilon^+(C)} x_{uv} \geq 1 \quad \text{for all } C \in \mathcal{C}, \tag{17}$$

where $E(C)$ is the set of edges in the strongly connected subgraph C and $\varepsilon^+(C)$ is the set of edges (u, v) such that u belongs to C and v does not belong to C. $\sum_{(u,v)\in E(C)} x_{uv} = |E(C)|$ indicates that C is in the subgraph of $\mathcal{A}(G_1, G_2)$ constructed by all edges (u, v) with positive x_{uv}, and $\sum_{(u,v)\in \varepsilon^+(C)} x_{uv} \geq 1$ guarantees that there exists an out-going edge of C that is in the subgraph.

We use the same technique as Dias et al. [5] to linearize the "if-then" condition in (17) by introducing a new variable β for each strongly connected component:

$$\sum_{(u,v)\in E(C)} x_{uv} \geq |E(C)|\beta_C \quad \text{for all } C \in \mathcal{C} \tag{18}$$

$$\sum_{(u,v)\in E(C)} x_{uv} - |E(C)| + 1 - |E(C)|\beta_C \leq 0 \quad \text{for all } C \in \mathcal{C} \tag{19}$$

$$\sum_{(u,v)\in \varepsilon^+(C)} x_{uv} \geq \beta_C \quad \text{for all } C \in \mathcal{C} \tag{20}$$

$$\beta_C \in \{0, 1\} \quad \text{for all } C \in \mathcal{C} \tag{21}$$

To summarize, given any pair of unidirectional, edge-labeled Eulerian graphs G_1 and G_2 and their alignment graph $\mathcal{A}(G_1, G_2) = (V, E, \delta)$, $\mathrm{GTED}(G_1, G_2)$ is equal to the optimal solution of the following ILP formulation:

$$\begin{aligned} &\underset{x\in\{0,1\}^{|E|}}{\text{minimize}} && \sum_{e\in E} x_e\delta(e) \\ &\text{subject to} && \text{constraints (13)–(16) and} \\ & && \text{constraints (18)–(21).} \end{aligned} \qquad \text{(exponential ILP)}$$

This ILP has an exponential number of constraints as there is a set of constraints for every strongly connected subgraph in the alignment graph. To solve this ILP more efficiently, we can use the procedure similar to the iterative constraint generation procedure in Dias et al. [5]. Initially, solve the ILP with only constraints (13)-(16). Create a subgraph, G', induced by edges with positive x_{uv}. For each disjoint SCC in G' that does not contain the sink node, add constraints (18)-(21) for edges in the SCC and solve the new ILP. Iterate until no disjoint SCCs are found in the solution. Algorithm 1 is the pseudo-code of this procedure, which is similar to Algorithm 1 of Dias et al. [5].

Algorithm 1 Iterative constraint generation algorithm to solve (exponential ILP).

```
Input Two unidirectional, edge-labeled Eulerian graphs and their alignment graph
C ← ∅
while true do
    Solve the ILP (exponential ILP) with C
    if the ILP variables x_uv induce a strongly connected component C not satisfying (17) then
        C = C ∪ {C}
    else
        return the optimal ILP value and the corresponding optimal solution x
    end if
end while
```

4.2 A compact ILP for GTED with polynomial number of constraints

In the worst cases, the number of iterations to solve (exponential ILP) via constraint generation is exponential. As an alternative, we introduce a compact ILP with only a polynomial number of constraints. The intuition behind this ILP is that we can impose a partially increasing ordering on all the edges so that the selected edges forms a s-t trail in the alignment graph. This idea is similar to the Miller-Tucker-Zemlin ILP formulation of the Travelling Salesman problem (TSP) [14].

We add variables d_{uv} that are constrained to provide a partial ordering of the edges in the s-t trail and set the variables d_{uv} to zero for edges that are not selected in the s-t trail. Intuitively, there must exist an ordering of edges in a s-t trail such that for each pair of consecutive edges (u, v) and (v, w), the difference in their order variable d_{uv} and d_{vw} is 1. Therefore, for each node v that is not the source or the sink, if we sum up the order variables for the incoming edges and outgoing edges respectively, the difference between the two sums is equal to the number of selected incoming/outgoing edges. Lastly, the order variable for the edge starting at source is 1, and the order variable for the edge ending at sink is the number of selected edges. This gives the ordering constraints as follows:

$$\text{If } x_{uv} = 0, \text{ then } d_{uv} = 0 \quad \text{for all } (u,v) \in E \tag{22}$$

$$\sum_{(v,w)\in E} d_{vw} - \sum_{(u,v)\in E} d_{uv} = \sum_{(v,w)\in E} x_{vw} \quad \text{for all } v \in V \setminus \{s,t\} \tag{23}$$

$$\sum_{(s,u)\in E} d_{su} = 1 \tag{24}$$

$$\sum_{(v,t)\in E} d_{vt} = \sum_{(u,v)\in E} x_{uv} \tag{25}$$

We enforce that all variables $x_e \in \{0,1\}$ and $d_e \in \mathbb{N}$ for all $e \in E$.

The "if-then" statement in Equation (22) can be linearized by introducing an additional binary variable y_{uv} for each edge [2, 5]:

$$-x_{uv} - |E| y_{uv} \leq -1 \tag{26}$$

$$d_{uv} - |E|(1 - y_{uv}) \leq 0 \tag{27}$$

$$y_{uv} \in \{0,1\}. \tag{28}$$

Here, y_{uv} is an indicator of whether $x_{uv} \geq 0$. The coefficient $|E|$ is the number of edges in the alignment graph and also an upper bound on the ordering variables. When $y_{uv} = 1$, $d_{uv} \leq 0$, and y_{uv} does not impose constraints on x_{uv}. When $y_{uv} = 0$, $x_{uv} \geq 1$, and y_{uv} does not impose constraints on d_{uv}. As we show in Lemma 7 , these constraints prevent finding disjoint components, thus guaranteeing the correctness of the ILP.

▶ **Lemma 7.** *Let x_e and d_e be ILP variables. Let G' be a subgraph of $\mathcal{A}(G_1, G_2)$ that is induced by edges with $x_e = 1$. If x_e and d_e satisfy constraints* (13)-(25) *for all $e \in E$, G' is connected with one trail from s to t that traverses each edge in G' exactly once.*

Proof. We prove the lemma in 2 parts: (1) all nodes except s and t in G' have an equal number of in- and out-edges, (2) G' contains only one connected component.

The first statement holds because the edges of G' form a flow from s to t, and is enforced by constraints (16).

We then show that G' does not contain isolated subgraphs that are not reachable from s or t. Due to constraint (16), the only possible scenario is that the isolated subgraph is strongly connected. Suppose for contradiction that there is a strongly connected component, C, in G' that is not reachable from s or t.

The sum of the left hand side of constraint (23) over all vertices in C is

$$\sum_{v\in C}\Big(\sum_{(u,v)\in C} d_{uv} - \sum_{(v,w)\in C} d_{vw}\Big) = \sum_{v\in C}\sum_{(u,v)\in C} d_{uv} - \sum_{v\in C}\sum_{(v,w)\in C} d_{vw} \tag{29}$$

$$= \sum_{(u,v)\in E(C)} d_{uv} - \sum_{(v,w)\in E(C)} d_{vw} = 0 \tag{30}$$

However, the right-hand side of the same constraints is always positive. Hence, we have a contradiction. Therefore, G' has only one connected component. ◀

Due to Lemma 6 and Lemma 7, given input graphs G_1 and G_2 and the alignment graph $\mathcal{A}(G_1, G_2)$, $\text{GTED}(G_1, G_2)$ is equal to the optimal objective of

$$\begin{aligned} &\underset{x\in\{0,1\}^{|E|}}{\text{minimize}} && \sum_{e\in E} x_e\delta(e) \\ &\text{subject to} && \text{constraints (13)–(16),} \\ & && \text{constraints (23)–(25)} \\ & && \text{and constraints (26)–(28).} \end{aligned} \qquad \text{(compact ILP)}$$

5 Closed-trail Cover Traversal Edit Distance

While the (lower bound ILP) and the ILP in (11)–(12) do not solve GTED, the optimal solution to these ILPs is a lower bound of GTED. These ILP formulations also solve an interesting variant of GTED, which is a local similarity measure between two genome graphs. We call this variant as Closed-trail Cover Traversal Edit Distance (CCTED). In the following, we provide the formal definition of CCTED problem, and then show that the (lower bound ILP) is the correct ILP formulation for solving CCTED.

We first introduce the min-cost item matching problem between two multi-sets. Let two multi-sets of items be S_1 and S_2, and, wlog, let $|S_1| \leq |S_2|$. Let $c : (S_1 \cup \{\epsilon\}) \times S_2 \to \mathbb{N}$ be the cost of matching either an empty item ϵ or an item in S_1 with an item in S_2. Given S_1, S_2 and the cost function c, min-cost matching problem finds a matching, $\mathcal{M}_c(S_1, S_2)$, such that each item in $S_1 \cup \{\epsilon\}^{|S_2|-|S_1|}$ is matched with exactly one distinct item in S_2 and the total cost of the matching, $\sum_{(s_1,s_2)\in\mathcal{M}_c(S_1,S_2)} c(s_1, s_2)$, is minimized.

The min-cost item matching problem is similar to the Earth Mover's Distance defined in [18], except that only integral units of items can be matched and the cost of matching an empty item with another item is not constant. Similar to the Earth Mover's Distance, the min-cost item matching problem can be computed using the ILP formulation of the min-cost max-flow problem [23, 21]. When the cost is the edit distance, the cost to match ϵ with a string is equal to the length of the string.

Define traversal edit distance, $edit_t(t_1, t_2)$ as the edit distance between the strings constructed from a pair of trails t_1 and t_2. In other words, $edit_t(t_1, t_2) = edit(\text{str}(t_1), \text{str}(t_2))$. CCTED is defined as:

▶ **Problem 3** (Closed-Trail Cover Traversal Edit Distance (CCTED)). *Given two unidirectional, edge-labeled Eulerian graphs G_1 and G_2 with closed Eulerian trails, compute*

$$CCTED(G_1, G_2) \triangleq \min_{\substack{C_1\in CC(G_1),\\ C_2\in CC(G_2)}} \sum_{(t_1,t_2)\in\mathcal{M}_{edit_t}(C_1,C_2)} edit(str(t_1), str(t_2)), \tag{31}$$

Here, $CC(G)$ denotes the collection of all possible sets of edge-disjoint, closed trails in G, such that every edge in G belongs to exactly one of these trails. Each element of $CC(G)$ can be interpreted as a cover of G using such trails. $\mathcal{M}_{edit_t}(C_1, C_2)$ is a min-cost matching between two covers using the traversal edit distance as the cost.

CCTED is likely a more suitable metric comparisons between genomes that undergo large-scale rearrangements. This analogy is to the relationship between the synteny block comparison [20] and the string edit distance computation, where the former is more often used in interspecies comparisons and in detecting segmental duplications [1, 24] and the latter is more often seen in intraspecies comparisons.

Following similar ideas as Lemma 6, we can compute CCTED by finding a set of closed trails in the alignment graph such that the total cost of alignment edges are minimized, and the projection of all edges in the collection of selected trails is equal to the multi-set of input graph edges.

▶ **Lemma 8.** *For any two edge-labeled Eulerian graphs G_1 and G_2,*

$$CCTED(G_1, G_2) = \underset{C}{\text{minimize}} \quad \sum_{c \in C} \delta(c) \tag{32}$$

$$\text{subject to} \quad C \text{ is a set of closed trails in } \mathcal{A}(G_1, G_2),$$

$$\bigcup_{e \in C} \Pi_i(e) = E_i \quad \text{for } i = 1, 2, \tag{33}$$

where C is a collection of trails and $\delta(c)$ is the total cost of edges in trail c.

Proof. Given any pair of covers $C_1 \in \mathrm{CC}(G_1)$ and $C_2 \in \mathrm{CC}(G_2)$ and their min-cost matching based on the edit distance $\mathcal{M}_{\mathrm{edit}_t}(C_1, C_2)$, we can project each pair of matched closed trails to a closed trail in the alignment graph. For a matching between a trail and the empty item ϵ, we can project it to a closed trail in the alignment graph with all vertical edges if the trail is from G_1 or horizontal edges if the trail is from G_2. The total cost of the projected edges must be greater than or equal to the objective (32). On the other hand, every collection of trails C that satisfy constraint (33) can be projected to a cover in each of the input graphs, and $\sum_{c \in C} \delta(c) \geq CCTED(G_1, G_2)$. Hence equality holds. ◀

We show that (lower bound ILP) solves CCTED (the proof is in Appendix B).

▶ **Theorem 9.** *Given two input graphs G_1 and G_2, the optimal objective value of* (lower bound ILP) *based on $\mathcal{A}(G_1, G_2)$ is equal to $CCTED(G_1, G_2)$.*

5.1 CCTED is a lower bound of GTED

Since the constraints for (lower bound ILP) are a subset of (exponential ILP), a feasible solution to (exponential ILP) is always a feasible solution to (lower bound ILP). Since two ILPs have the same objective function, $\mathrm{CCTED}(G_1, G_2) \leq \mathrm{GTED}(G_1, G_2)$ for any pair of graphs. Moreover, when the solution to (lower bound ILP) forms only one connected component, the optimal value of (lower bound ILP) is equal to GTED.

▶ **Theorem 10.** *Let $\mathcal{A}'(G_1, G_2)$ be the subgraph of $\mathcal{A}(G_1, G_2)$ induced by edges $(u, v) \in E$ with $x^{opt}_{uv} = 1$ in the optimal solution to* (lower bound ILP)*, where x_{uv} is the value of the variable corresponding to edge (u, v). There exists $\mathcal{A}'(G_1, G_2)$ that has exactly one connected component if and only if $CCTED(G_1, G_2) = GTED(G_1, G_2)$.*

Proof. We first show that if $\mathrm{CCTED}(G_1, G_2) = \mathrm{GTED}(G_1, G_2)$, then there exists $\mathcal{A}'(G_1, G_2)$ that has exactly one connected component. Since $\mathrm{CCTED}(G_1, G_2) = \mathrm{GTED}(G_1, G_2)$, an optimal solution to (exponential ILP) is also an optimal solution to (lower bound ILP), which can induce a subgraph in the alignment graph that only contains one connected component.

Conversely, if x^{opt} induces a subgraph in the alignment graph with only one connected component, it satisfies constraints (18)-(21) and therefore is feasible to the ILP for GTED (exponential ILP). Since $\mathrm{CCTED}(G_1, G_2) \leq \mathrm{GTED}(G_1, G_2)$, this solution must also be optimal for $\mathrm{GTED}(G_1, G_2)$. ◀

In practice, we may estimate GTED by the solution to (lower bound ILP). As we show in Section 6, the time needed to solve (lower bound ILP) is much less than the time needed to solve GTED. When the subgraph induced by the solution to (lower bound ILP) has only one connected component, CCTED is exactly equal to GTED. However, in adversarial cases, CCTED could be zero but GTED could be arbitrarily large.

6 Empirical evaluation of the ILP formulations for GTED and its lower bound

6.1 Implementation of the ILP formulations

We implement the algorithms and ILP formulations for (exponential ILP), (compact ILP) and (lower bound ILP). In practice, the multi-set of edges of each input graph may contain many duplicates of edges that have the same start and end vertices due to repeats in the strings. We reduce the number of variables and constraints in the implemented ILPs by merging the edges that share the same start and end nodes and recording the multiplicity of each edge. Each x variable is no longer binary but a non-negative integer that satisfies the modified projection constraint (13):

$$\sum_{(u,v)\in E} x_{uv} I_i((u,v),f) = M_i(f) \quad \text{for all } (u,v)\in E,\ f\in G_i, u\neq s, v\neq t, \tag{34}$$

where $M_i(f)$ is the multiplicity of edge f in G_i. Let C be the strongly connected component in the subgraph induced by positive x_{uv}, now $\sum_{(u,v)\in E(C)} x_{uv}$ is no longer upper bounded by $|E(C)|$. Therefore, the constraint (19) is changed to

$$\sum_{(u,v)\in E(C)} x_{uv} - |E(C)| + 1 - W(C)\beta_C \leq 0 \quad \text{for all } C\in\mathcal{C}, \tag{35}$$

$$W(C) = \sum_{(u,v)\in E(C)} \max\left(\sum_{f\in G_1} M_1(f) I_1((u,v),f), \sum_{f\in G_2} M_2(f) I_2((u,v),f)\right),$$

where $W(C)$ is the maximum total multiplicities of edges in the strongly connected subgraph in each input graph that is projected from C.

Likewise, constraints (27) that set the upper bounds on the ordering variables also need to be modified as the upper bound of the ordering variable d_{uv} for each edge no longer represents the order of one edge but the sum of orders of copies of (u,v) that are selected, which is at most $|E|^2$. Therefore, constraint (27) is changed to

$$d_{uv} - |E|^2(1-y_{uv}) \leq 0. \tag{36}$$

The rest of the constraints remain unchanged.

We ran all our experiments on a server with 48 cores (96 threads) of Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz and 378 GB of memory. The system was running Ubuntu 18.04 with Linux kernel 4.15.0. We solve all the ILP formulations and their linear relaxations using the Gurobi solver [7] using 32 threads.

6.2 GTED on simulated TCR sequences

We construct 20 de Bruijn graphs with $k = 4$ using 150-character sequences extracted from the V genes from the IMGT database [11]. We solve (exponential ILP), (lower bound ILP) and its linear relaxation, and the linear relaxation of (compact ILP) on all 190 pairs of graphs. We do not show results for solving (compact ILP) for GTED on this set of graphs as the running time exceeds 30 minutes on most pairs of graphs.

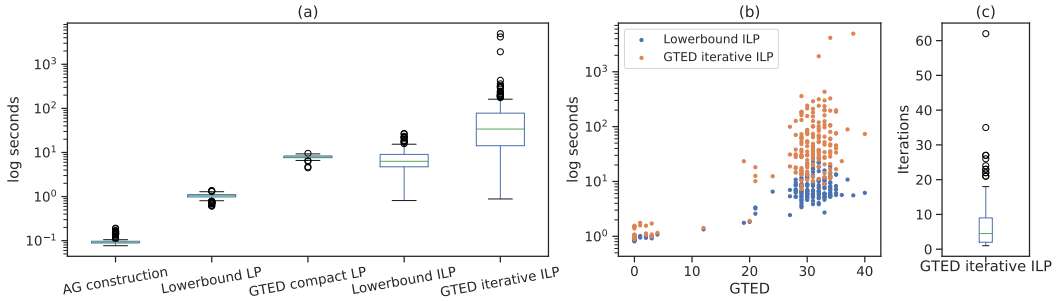


Figure 5 (a) The distribution of wall-clock running time for constructing alignment graphs, solving the ILP formulations for GTED and its lower bound, and their linear relaxations on the log scale. (b) The relationship between the time to solve (lower bound ILP), (exponential ILP) iteratively and GTED. (c) The distribution of the number of iterations to solve exponential ILP. The box plots in each plot show the median (middle line), the first and third quantiles (upper and lower boundaries of the box), the range of data within 1.5 inter-quantile range between Q1 and Q3 (whiskers), and the outlier data points.

To compare the time to solve the ILP formulations when GTED is equal to the optimal objective of (lower bound ILP), we only include 168 out of 190 graphs where GTED is equal to CCTED. On average, it takes 26 seconds wall-clock time to solve (lower bound ILP), and 71 seconds to solve (exponential ILP) using the iterative algorithm. On average, it takes 9 seconds to solve the LP relaxation of (compact ILP) and 1 second to solve the LP relaxation of (lower bound ILP). The time to construct the alignment graph for all pairs is less than 0.2 seconds. The distribution of wall-clock running time is shown in Figure 5(a). The time to solve (exponential ILP) and (lower bound ILP) is generally positively correlated with the GTED values (Figure 5(b)). On average, it takes 7 iterations for the iterative algorithm to find the optimal solution that induces one strongly connected subgraph (Figure 5(c)).

In summary, it is fastest to compute the lower bound of GTED. Computing GTED exactly by solving the proposed ILPs on genome graphs of size 150 is already time consuming. When the sizes of the genome graphs are fixed, the time to solve for GTED and its lower bound increases as the GTED between the two genome graphs increases. In the case where GTED is equal to its lower bound, the subgraph induced by some optimal solutions of (lower bound ILP) contains more than one strongly connected component. Therefore, in order to reconstruct the strings from each input graph that have the smallest edit distance, we generally need to obtain the optimal solution to the ILP for GTED. In all cases, the time to solve the (exponential ILP) is less than the time to solve the (compact ILP).

6.3 GTED on difficult cases

Repeats, such as segmental duplications and translocations [12, 4] in the genomes increase the complexity of genome comparisons. We simulate such structures with a class of graphs that contain n simple cycles of which $n-1$ peripheral cycles are attached to the n-th central cycle at either a node or a set of edges (Figure 6(a)). The input graphs in Figure 2 belong to this class of graphs that contain 2 cycles. This class of graphs simulates the complex structural variants in disease genomes or the differences between genomes of different species.

We generate pairs of 3-cycle graphs with varying sizes and randomly assign letters from `{A,T,C,G}` to edges. We compute the CCTED and GTED using (lower bound ILP) and (compact ILP), respectively. We group the generated 3-cycle graph pairs based on the

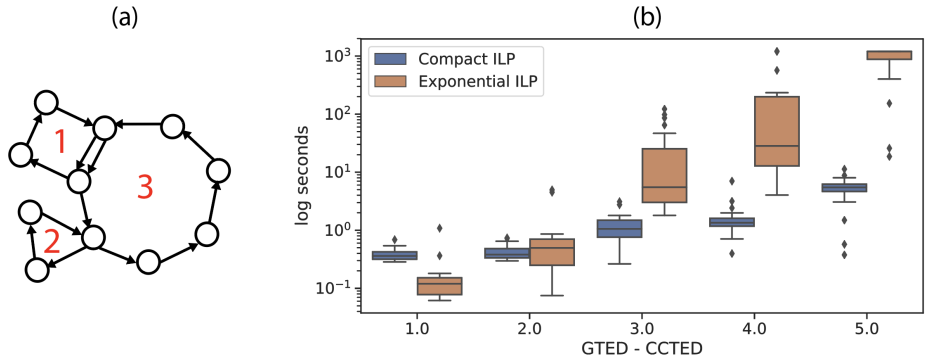


Figure 6 (a) An example of a 3-cycle graph. Cycle 1 and 2 are attached to cycle 3. (b) The distribution of wall-clock time to solve the compact ILP and the iterative exponential ILP on 100 pairs of 3-cycle graphs.

value of (GTED − CCTED) and select 20 pairs of graphs randomly for each (GTED − CCTED) value ranging from 1 to 5. The maximum number of edges in all selected graphs is 32.

We show the difficulty of computing GTED using the iterative algorithm on the 100 selected pairs of 3-cycle graphs. We terminate the ILP solver after 20 minutes. As shown in Figure 6, as the difference between GTED and CCTED increases, the wall-clock time to solve (exponential ILP) for GTED increases faster than the time to solve (compact ILP) for GTED. For pairs of graphs with (GTED − CCTED) = 5, on average it takes more than 15 minutes to solve (exponential ILP) with more than 500 iterations. On the other hand, it takes an average of 5 seconds to solve (compact ILP) for GTED and no more than 1 second to solve for the lower bound. The average time to solve each ILP is shown in Table S1.

In summary, on the class of 3-cycle graphs introduced above, the difficulty to solve GTED via the iterative algorithm increases rapidly as the gap between GTED and CCTED increases. Although (exponential ILP) is solved more quickly than (compact ILP) for GTED when the sequences are long and the GTED is equal to CCTED (Section 6.2), (compact ILP) may be more efficient when the graphs contain overlapping cycles such that the gap between GTED and CCTED is larger.

7 Conclusion

We point out the contradictions in the result on the complexity of labeled graph comparison problems and resolve the contradictions by showing that GTED, as opposed to the results in Ebrahimpour Boroojeny et al. [6], is NP-complete. On one hand, this makes GTED a less attractive measure for comparing graphs since it is unlikely that there is an efficient algorithm to compute the measure. On the other hand, this result better explains the difficulty of finding a truly efficient algorithm for computing GTED exactly. In addition, we show that the previously proposed ILP of GTED [6] does not solve GTED and give two new ILP formulations of GTED.

We further show that the previously proposed ILP solves for a lower bound of GTED. We characterize the LP relaxation of the ILP in (11)–(12) and show that, contrary to the results in Ebrahimpour Boroojeny et al. [6], the LP in (11)–(12) does not always yield optimal integer solutions.

As shown previously [6, 21], it takes more than 4 hours to solve (lower bound ILP) for graphs that represent viral genomes that contain ≈ 3000 bases with a multi-threaded LP solver. Likewise, we show that computing GTED using either (exponential ILP) or (compact ILP) is already slow on small genomes, especially on pairs of genomes that are different due to segmental duplications and translations. The empirical results show that it is currently impossible to solve GTED or its lower bound directly using this approach for bacterial- or eukaryotic-sized genomes on modern hardware. The results here should increase the theoretical interest in GTED along the directions of heuristics or approximation algorithms as justified by the NP-hardness of finding GTED.

References

1 Guillaume Bourque, Pavel A Pevzner, and Glenn Tesler. Reconstructing the genomic architecture of ancestral mammals: lessons from human, mouse, and rat genomes. *Genome Research*, 14(4):507–516, 2004.

2 Stephen P Bradley, Arnoldo C Hax, and Thomas L Magnanti. *Applied mathematical programming*. Addison-Wesley, 1977.

3 George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954.

4 Eva Darai-Ramqvist, Agneta Sandlund, Stefan Müller, George Klein, Stefan Imreh, and Maria Kost-Alimova. Segmental duplications and evolutionary plasticity at tumor chromosome break-prone regions. *Genome Research*, 18(3):370–379, 2008.

5 Fernando HC Dias, Lucia Williams, Brendan Mumey, and Alexandru I Tomescu. Minimum flow decomposition in graphs with cycles using integer linear programming. *arXiv preprint*, 2022. `arXiv:2209.00042`.

6 Ali Ebrahimpour Boroojeny, Akash Shrestha, Ali Sharifi-Zarchi, Suzanne Renick Gallagher, S. Cenk Sahinalp, and Hamidreza Chitsaz. Graph traversal edit distance and extensions. *Journal of Computational Biology*, 27(3):317–329, 2020.

7 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL: `https://www.gurobi.com`.

8 Steve Huntsman and Arman Rezaee. De Bruijn entropy and string similarity. *arXiv preprint*, 2015. `arXiv:1509.02975`.

9 Chirag Jain, Haowen Zhang, Yu Gao, and Srinivas Aluru. On the complexity of sequence-to-graph alignment. *Journal of Computational Biology*, 27(4):640–654, 2020.

10 Orna Kupferman and Gal Vardi. Eulerian paths with regular constraints. In Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier, editors, *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, volume 58 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 62:1–62:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

11 Marie-Paule Lefranc. IMGT, the international ImMunoGeneTics information system. *Cold Spring Harbor Protocols*, 2011(6):595–603, 2011.

12 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.

13 Serghei Mangul and David Koslicki. Reference-free comparison of microbial communities via de Bruijn graphs. In *Proceedings of the 7th ACM international conference on bioinformatics, computational biology, and health informatics*, pages 68–77, 2016.

14 Clair E Miller, Albert W Tucker, and Richard A Zemlin. Integer programming formulation of traveling salesman problems. *Journal of the ACM (JACM)*, 7(4):326–329, 1960.

15 Ilia Minkin and Paul Medvedev. Scalable pairwise whole-genome homology mapping of long genomes with Bubbz. *IScience*, 23(6):101224, 2020.

16 James R Munkres. *Elements of algebraic topology*. CRC Press, 2018.

17 Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

18 Ofir Pele and Michael Werman. A linear time histogram metric for improved sift matching. In *Computer Vision–ECCV 2008: 10th European Conference on Computer Vision, Marseille, France, October 12-18, 2008, Proceedings, Part III 10*, pages 495–508. Springer, 2008.

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

20 Evgeny Polevikov and Mikhail Kolmogorov. Synteny Paths for Assembly Graphs Comparison. In Katharina T. Huber and Dan Gusfield, editors, *19th International Workshop on Algorithms in Bioinformatics (WABI 2019)*, volume 143 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:14, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.WABI.2019.24.

21 Yutong Qiu and Carl Kingsford. The effect of genome graph expressiveness on the discrepancy between genome graph distance and string set distance. *Bioinformatics*, 38:i404–i412, 2022.

22 Yutong Qiu, Yihang Shen, and Carl Kingsford. Revisiting the complexity of and algorithms for the Graph Traversal Edit Distance and Its variants. *arXiv preprint*, 2023. arXiv:2305.10577.

23 Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. The Earth Mover's distance as a metric for image retrieval. *International Journal of Computer Vision*, 40(2):99–121, 2000.

24 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, et al. Segmental duplications and their variation in a complete human genome. *Science*, 376(6588):eabj6965, 2022.

A Proofs for the NP-completeness of GTED

A.1 Reduction from ETEW to GTED

We provide below the complete proof for Theorem 2.

▶ **Theorem 2.** *If GTED $\in P$ then ETEW $\in P$.*

Proof. Let $\langle s, G\rangle$ be an instance of ETEW. Construct a directed, acyclic graph (DAG), C, that has only one path. Let the path in C be $P = (e_1, \ldots, e_{|s|})$ and the edge label of e_i be $s[i]$. Clearly, C is a unidirectional, edge-labeled Eulerian graph, P is the only Eulerian trail in C, and $str(P) = s$.

For the graph $G = (V_G, E_G, \ell_G, \Sigma)$ from the ETEW instance, which may not be unidirectional, create another graph G' that contains all of the nodes and edges in G except the anti-parallel edges. Let $\Sigma_{G'} = \Sigma \cup \{\epsilon\}$, where ϵ is a character that is not in Σ. For each pair of anti-parallel edges (u, v) and (v, u) in G, add four edges $(u, w_1), (w_1, v), (v, w_2), (w_2, u)$ by introducing new vertices w_1, w_2 to G'. Let $\ell_{G'}(u, w_1) = \ell_G(u, v)$ and $\ell_{G'}(w_2, u) = \ell_G(v, u)$. Let $\ell_{G'}(w_1, v) = \ell_{G'}(v, w_2) = \epsilon$ for every newly introduced vertex. G' has at most twice the number of edges as G and is Eulerian and unidirectional.

Define the cost of changing a character from a to b $\text{cost}(a, b)$ for $a, b \in \Sigma \cup \{-\}$ to be 0 if $a = b$ and 1 otherwise. "$-$" is the gap character indicating an insertion or a deletion. Define $\text{cost}(a, \epsilon)$ with $a \in \Sigma$ to be 1. Define $\text{cost}(-, \epsilon)$ to be 0.

Use the (assumed) polynomial-time algorithm for GTED to ask whether $\text{GTED}(C, G') \leq 0$ under edit distance Σ. If yes, then let (s_1, s_2) be the 0-cost alignment of the strings spelled out by the trails in C and G', respectively. The non-gap characters of s_1 must spell out s since there is only one Eulerian trail in C. Because the alignment cost is 0, any $-$ (gap)

characters in s_1 must be aligned with ϵ characters in s_2 and any non-gap characters in s_1 must be aligned to the same character in s_2. The trail in G' that spells s_2 can be transformed to a trail that spells s_3 by collapsing the edges with ϵ character labels, and $s_3 = s_1$.

If $\text{GTED}(C, G') > 0$, G must not contain an Eulerian trail that spells s. Otherwise, such a trail could be extended to a trail introducing some ϵ characters that could be aligned to s with zero cost by aligning gaps with ϵ characters.

Hence, an (assumed) polynomial-time algorithm for GTED solves ETEW in polynomial time. ◀

A.2 Reduction from Hamiltonian Path to GTED

We provide below the complete proof for Theorem 3.

▶ **Theorem 3.** *GTED is NP-complete.*

Proof. We reduce from the Hamiltonian Path problem, which asks whether a directed, simple graph G contains a path that visits every vertex exactly once. Here simple means no self-loops or parallel edges. Let $\langle G = (V, E)\rangle$ be an instance of Hamiltonian Path, with $n = |V|$ vertices. The reduction is almost identical to that presented in Kupferman and Vardi [10], and from here until noted later in the proof the argument is identical except for the technicalities introduced to force unidirectionality (and another minor change described later). The first step is to construct the Eulerian closure of G, which is defined as $G' = (V', E')$ where

$$V' = \{v^{in}, v^{out} : v \in V\} \cup \{w\}, \tag{37}$$

and E' is the union of the following sets of edges and their labels:

- $E_1 = \{(v^{in}, v^{out}) : v \in V\}$, labeled `a`,
- $E_2 = \{(u^{out}, v^{in}) : (u, v) \in E\}$, labeled `b`,
- $E_3 = \{(v^{out}, v^{in}) : v \in V\}$, labeled `c`,
- $E_4 = \{(v^{in}, u^{out}) : (u, v) \in E\}$, labeled `c`,
- $E_5 = \{(u^{in}, w) : u \in V\}$, labeled `c`,
- $E_6 = \{(w, u^{in}) : u \in V\}$, labeled `b`.

Since G' is connected and every outgoing edge in G' has a corresponding antiparallel incoming edge, G' is Eulerian. It is not unidirectional, so we further create G'' from G' by adding dummy nodes to each pair of antiparallel edges and labelling the length-2 paths so created with `x#`, where `x` is the original label of the split edge (`a`, `b`, or `c`) and `#` is some new symbol (shared between all the new edges). We call these length-2 paths introduced to achieve unidirectionality "split edges".

We now argue that G has a Hamiltonian path iff G'' has an Eulerian trail that spells out

$$q = \texttt{a\#}(\texttt{b\#a\#})^{n-1}(\texttt{c\#})^{2n-1}(\texttt{c\#b\#})^{|E|+1}. \tag{38}$$

If such an Eulerian trail exists, then the trail starts with spelling the string $\texttt{a\#}(\texttt{b\#a\#})^{n-1}$, which corresponds to a Hamiltonian trail in G since it visits exactly n "vertex split edges" (type E_1, labeled `a#`) and each vertex split edge can be used only once (since it is an Eulerian trail). Further, successively visited vertices must be connected by an edge in G since those are the only `b#` split edges in G'' (except those leaving w, but w must not be involved in spelling out $\texttt{a\#}(\texttt{b\#a\#})^{n-1}$, since entering w requires using a split edge labeled `c#`).

For the other direction, if a G has a Hamiltonian path $v_1, \ldots, v_n$, then walking that sequence of vertices in G'' will spell out $\texttt{a\#}(\texttt{b\#a\#})^{n-1}$. This path will cover all E_1 edges and the E_2 edges that are on the Hamiltonian path. Retracing the path so far in reverse will

use $2n-1$ split edges labeled `c#`, consuming the $(\texttt{c\#})^{2n-1}$ term in q and covering all nodes' reverse vertex edges E_3 (since the path is Hamiltonian). The reverse path also covers the E_4 edges corresponding to reverse Hamiltonian path edges. Our Eulerian trail is now "at" node v_1^{in}.

What remains is to complete the Eulerian walk covering (a) edges and their antiparallel counterparts corresponding to edges in G that were not used in the Hamiltonian path, and (b) the edges adjacent to node w. To do this, define $\text{pred}(v)$ be the vertices u in G for which edge (u, v) exists and u is not the predecessor of v along the Hamiltonian path. For each $u \in \text{pred}(v_1)$, traverse the split edge labeled `c#` to u^{out} then traverse the forward split edge labeled `b#` back to v_1^{in}. This results in a string $(\texttt{c\#b\#})^{|\text{pred}(v_1)|}$. Once the predecessors of v_1 are exhausted, traverse the split edge labeled `c#` from v_1^{in} into node w and then traverse the split edge labeled `b#` to v_2^{in}. This again generates a `c#b#` string. Repeat the process, covering the edges of v_2's predecessors and returning to w to move to the next node along the Hamiltonian path for each node $v_3, \ldots, v_n$. After covering the predecessors of v_n^{in}, go to v_1^{in} through the remaining edges in E_5 and E_6, (v_n^{in}, w) and (w, v_1^{in}), which completes the Eulerian tour. This covers all the edges of G''. The word spelled out in this last section of the Eulerian trail is a sequence of repetitions of `c#b#`, with one repetition for each edge that is not in the Hamiltonian path $(|E| - n + 1)$ and all of the edges in E_5 and E_6 for entering and leaving each node $(2n)$, with a total of $|E| + 1$ repetitions, which is the final $(\texttt{c\#b\#})^{|E|+1}$ term in q.

This ends the slight modification of the proof in Kupferman and Vardi [10], where the differences are (a) the introduction of the `#` characters and (b) using the exponent $|E| + 1$ of the final part of q instead of $|E| + n + 1$ as in Kupferman and Vardi [10] since we create w-edges only to v^{in} vertices. (This second change has no material effect on the proof, but reduces the length of the string that must be matched.)

Now, given an instance $\langle G = (V, E)\rangle$ of Hamiltonian Path, with $n = |V|$ vertices, we construct G'' as above (obtaining a unidirectional Eulerian graph) and create graph C that only represents string q. Note that $|\Sigma| = 4$ and G'' and C can be constructed in polynomial time. $\text{GTED}(G'', C) = 0$ if and only if an Eulerian path in G'' spells out q, since there can be no indels or mismatches. By the above argument, an An eulerian tour that spells out q exists if and only if G has a Hamiltonian path. ◀

A.3 FGTED is NP-complete

▶ **Problem 4** (Flow Graph Traversal Edit Distance (FGTED) [21]). *Given unidirectional, edge-labeled Eulerian graphs G_1 and G_2, each of which has distinguished s_1, s_2 source and t_1, t_2 sink vertices, compute*

$$FGTED(G_1, G_2) \triangleq \min_{\substack{D_1 \in flow(G_1, s_1, t_1) \\ D_1 \in flow(G_2, s_2, t_2)}} emedit(strset(D_1), strset(D_2)), \tag{39}$$

where $flow(G_i, s_i, t_i)$ is the collection of all possible sets of s_1-t_1 trail decomposition of saturating flow from s_i to t_i, $strset(D)$ is the multi-set of strings constructed from trails in D.

▶ **Theorem 4.** *FGTED is NP-complete.*

Proof. Let $G = (v, E)$ be an instance of the Hamiltonian Cycle problem. Let $n = |V|$ be the number of vertices in G. Construct the Eulerian closure of G and split the anti-parallel edges. Let the new graph be $G' = (V', E')$. Attach a source s and a sink node t to an arbitrary node v_1^{in} by adding edge (s, v_1^{in}) and (v_1^{in}, t) with labels `s` and `t`, respectively.

Construct a string q, such that

$$q = \texttt{sa\#(b\#a\#)}^{n-1}\texttt{(c\#)}^{2n-1}\texttt{(c\#b\#)}^{|E|+1}\texttt{t}. \tag{40}$$

Create a graph Q that only contains one path with labels on the edges of the path that spell the string q. The union of the set of trails in any flow decomposition of G' is equal to a set of Eulerian trails, $\mathcal{E}$, that starts at s and ends at t. All Eulerian trails in $\mathcal{E}$ are also closed Eulerian trails of $G' \setminus \{s, t\}$ that starts and ends at v_1^{in}.

Using the same line of argument in the proof of Theorem 3, an Eulerian trail in G' that spells q is equivalent to a Hamiltonian cycle in G. In addition, $\text{FGTED}(Q, G') = 0$ if and only if all Eulerian trails in $\mathcal{E}$ spell out q. Therefore, if $\text{FGTED}(Q, G') = 0$, then there is a Hamiltonian cycle in G. Otherwise, then there must not exist a Hamiltonian cycle in G. ◀

B Correctness of the ILP formulation for CCTED

We show that the ILP in (5)–(8) proposed by Ebrahimpour Boroojeny et al. [6] solves CCTED.

▶ **Theorem 9.** *Given two input graphs G_1 and G_2, the optimal objective value of the ILP in (5)–(8) based on $\mathcal{A}(G_1, G_2)$ is equal to CCTED(G_1, G_2).*

Proof. As shown in the proof of Lemma 8, any pair of edge-disjoint, closed-trail covers in the input graph can be projected to a set of closed trails in $\mathcal{A}(G_1, G_2)$, which satisfied constraints (6)-(8). The objective of this feasible solution, which is the total cost of the projected closed trails, equals CCTED. Therefore, $\text{CCTED}(G_1, G_2)$ is greater than or equal to the objective of the ILP in (5)–(8).

Conversely, we can transform any feasible solution of the ILP in (5)–(8) to a pair of covers of G_1 and G_2. We can do this by transforming one closed trail at a time from the subgraph of the alignment graph, $\mathcal{A}'$ induced by edges with ILP variable $x_{uv} = 1$. Let c be a closed trail in $\mathcal{A}'$. Let $c_1 = \Pi_1(c)$ and $c_2 = \Pi_2(c)$ be two closed trails in G_1 and G_2 that are projected from c. We can construct an alignment between $\text{str}(c_1)$ and $\text{str}(c_2)$ from c by adding match or insertion/deletion columns for each match or insertion/deletion edges in c accordingly. The cost of the alignment is equal to the total cost of edges in c by the construction of the alignment graph. We can then remove edges in c from the alignment graph and edges in c_1 and c_2 from the input graphs, respectively. The remaining edges in $\mathcal{A}'$ and G_1 and G_2 still satisfy the constraints (6)-(8). Repeat this process and we get a total cost of $\sum_{e \in E} x_e \delta(e)$ that aligns pairs of closed trails that form covers of G_1 and G_2. This total cost is greater than or equal to $\text{CCTED}(G_1, G_2)$. ◀

C The average wall-clock time to solve ILPs on 3-cycle graphs

Table S1 The average wall-clock time to solve lower bound ILP, exponential ILP, compact ILP and the number of iterations for pairs of 3-cycle graphs for each GTED − CCTED.

GTED - CCTED	lower bound ILP runtime (s)	GTED iterative runtime (s)	Iterations	GTED compact runtime (s)
1.0	0.06	0.17	3.55	0.39
2.0	0.05	0.87	13.00	0.43
3.0	0.08	25.41	67.60	1.24
4.0	0.07	205.59	179.10	1.70
5.0	0.08	943.68	502.85	5.37

Co-Linear Chaining on Pangenome Graphs

Jyotshna Rajput
Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India

Ghanshyam Chandra
Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India

Chirag Jain
Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India

Abstract

Pangenome reference graphs are useful in genomics because they compactly represent the genetic diversity within a species, a capability that linear references lack. However, efficiently aligning sequences to these graphs with complex topology and cycles can be challenging. The seed-chain-extend based alignment algorithms use co-linear chaining as a standard technique to identify a good cluster of exact seed matches that can be combined to form an alignment. Recent works show how the co-linear chaining problem can be efficiently solved for acyclic pangenome graphs by exploiting their small width [Makinen et al., TALG'19] and how incorporating gap cost in the scoring function improves alignment accuracy [Chandra and Jain, RECOMB'23]. However, it remains open on how to effectively generalize these techniques for general pangenome graphs which contain cycles. Here we present the first practical formulation and an exact algorithm for co-linear chaining on cyclic pangenome graphs. We rigorously prove the correctness and computational complexity of the proposed algorithm. We evaluate the empirical performance of our algorithm by aligning simulated long reads from the human genome to a cyclic pangenome graph constructed from 95 publicly available haplotype-resolved human genome assemblies. While the existing heuristic-based algorithms are faster, the proposed algorithm provides a significant advantage in terms of accuracy.

2012 ACM Subject Classification Applied computing → Computational genomics

Keywords and phrases Sequence alignment, variation graph, genome sequencing, path cover

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.12

Supplementary Material *Software (Source Code)*: `https://github.com/at-cg/PanAligner`

Funding This research is supported in part by the funding from National Supercomputing Mission, India under DST/NSM/ R&D_HPC_Applications and the Science and Engineering Research Board (SERB) under SRG/2021/000044. We used computing resources provided by the National Energy Research Scientific Computing Center (NERSC), USA.

Acknowledgements The authors thank Manuel Cáceres, Shravan Mehra and Sunil Chandran for providing useful feedback.

1 Introduction

Graph-based representation of genome sequences has emerged as a prominent data structure in genomics, offering a powerful means to represent the genetic variation within a species across multiple individuals [11, 17, 26, 49, 51, 53]. A pangenome graph can be represented as a directed graph $G(V, E)$ such that vertices are labeled by characters (or strings) from the alphabet {A,C,G,T}. The topology of the graph is determined by the count and the type of variants included in the graph. For example, inversions, duplications, or copy number variation are best represented as cycles in a pangenome graph [8, 26, 27, 41, 49]. As a result, the draft pangenome graphs published by the Human Pangenome Reference Consortium [26] and the Chinese Pangenome Consortium [14] are also cyclic. Aligning reads or assembly

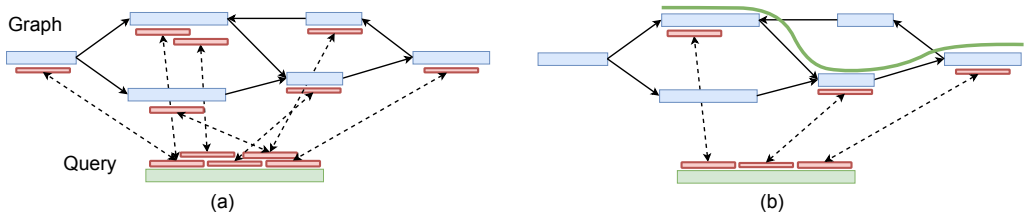


Figure 1 An illustration of co-linear chaining for sequence-to-graph alignment. Assume that the vertices of the graph are labeled with nucleotide sequences. Short exact matches, i.e., anchors, are illustrated using red blocks joined by dotted lines. In (b), the anchors corresponding to the best-scoring chain are retained, and the rest are removed. The retained anchors are combined to produce an alignment of the query sequence to the graph.

contigs to a directed labeled graph is a fundamental problem in computational pangenomics [2, 7]. Aligning reads to graphs is also useful for other bioinformatics tasks such as long-read *de novo* assembly [6, 15, 43] and long-read error correction [28, 47].

Formally, the sequence-to-graph alignment problem seeks a walk in the graph that spells a sequence with minimum edit distance from the input sequence. $O(|Q||E|)$ time alignment algorithms for this problem are already known, where Q is the query sequence [22, 36]. The known conditional lower bound [3] implies that an exact algorithm significantly faster than $O(|Q||E|)$ is unlikely. This lower bound also holds for de Bruijn graphs [18]. Therefore, fast heuristics are used to process high-throughput sequencing data.

Seed-chain-extend is a common heuristic used by modern alignment tools [21, 23, 46]. This is a three-step process. First, the seeding stage involves computing exact seed matches, such as k-mer matches, between a query sequence and a reference. These matches are referred to as *anchors*. The presence of repetitive sequences in genomes often leads to a large number of false positive anchors. Subsequently, the *chaining* stage is employed to link the subsets of anchors in a coherent manner while optimizing specific criteria (Figure 1). This procedure also eliminates the false positive anchors. Finally, the extend stage returns a base-to-base alignment along the selected anchors. Efficient generalization of the three stages to pangenome graphs is an active research topic [7]. Many sequence-to-graph aligners already exist that differ in terms of implementing these stages [5, 9, 24, 30, 42, 49]. This paper addresses the generalization of the chaining stage to cyclic pangenome graphs.

1.1 Related Work

Co-linear chaining is a mathematically rigorous method to filter anchors after the seeding stage. It has been well-studied for the sequence-to-sequence alignment case [1, 12, 13, 20, 32, 35, 40]. The input to the chaining problem is a set of N weighted anchors. An anchor can be denoted as a pair of intervals in the two sequences corresponding to the exact seed match. A chain is an ordered subset of anchors whose intervals must appear in increasing order in both sequences. The co-linear chaining problem seeks the chain with the highest score, where the score of a chain is calculated by summing the weights of the anchors in the chain and subtracting the penalty for gaps between adjacent anchors. The problem is solvable in $O(N \log N)$ time [1].

The first effort to generalize the co-linear chaining problem to graphs was made by Makinen et al. [33]. They addressed the co-linear chaining problem on directed acyclic graphs (DAGs). The authors introduced a sparse dynamic programming algorithm whose runtime complexity is parameterized in terms of the *width* of the DAG. The width of a DAG

is defined as the minimum number of paths in the DAG such that each vertex is included in at least one path. Parameterizing the complexity in terms of the width is helpful because pangenome graphs typically have small width in practice [5, 30, 33]. An optimized version of their algorithm requires $O(KN \log KN)$ time for chaining, where K is the width of the DAG [30]. This formulation has been further extended to incorporate gap cost in the scoring function [5], and for solving the longest common subsequence problem between a DAG and a sequence [44]. However, there is limited work on formulating and solving the co-linear chaining problem for general pangenome graphs which might contain cycles. One way to address this was discussed in [30, Appendix section], but the proposed formulation is oblivious to the coordinates of anchors that lie in a strongly connected component of the graph. Their algorithm works by shrinking every strongly connected component into a single vertex and applying the same algorithm developed for DAGs. With this approach, the high-scoring anchor chains in cyclic regions of the graph may result in low-quality alignments.

1.2 Contributions

In this paper, we build on top of the algorithmic techniques developed for DAGs [5, 30, 33] and propose novel formulations for cyclic pangenome graphs. Our proposed algorithm exploits the small width of pangenome graphs similar to [33]. Our approach for defining the gap cost between a pair of anchors is inspired by the corresponding function defined on DAGs [5].

We address the following three challenges that arise on cyclic pangenome graphs. First, the dynamic programming-based chaining algorithms developed for DAGs exploit the topological ordering of vertices [5, 30, 33], but such an ordering is not available in cyclic graphs. Second, computing the width and a minimum path cover can be solved in polynomial time for DAGs but is NP-hard for general instances [4]. Third, the walk corresponding to the optimal sequence-to-graph alignment can traverse a vertex multiple times if there are cycles. Accordingly, a chain of anchors should be allowed to loop through vertices. Our proposed problem formulation and the proposed algorithm address the above challenges. Our approach involves computing a path cover $\mathcal{P}$ of the input graph followed by using iterative algorithms. Let $\Gamma_c, \Gamma_l, \Gamma_d$ be the parameters that specify the count of iterations used in our algorithms (formally defined later). Our chaining algorithm solves the stated objective in $O(\Gamma_c|\mathcal{P}|N \log N + |\mathcal{P}|N \log |\mathcal{P}|N)$ time after a one-time preprocessing of the graph in $O((\Gamma_l + \Gamma_d + \log |V|)|\mathcal{P}||E|)$ time. We will show that parameters $|\mathcal{P}|, \Gamma_c, \Gamma_l, \Gamma_d$ are small in practice to justify the practicality of this approach.

We implemented the proposed chaining algorithm as an open-source software PanAligner. We designed PanAligner as an end-to-end sequence-to-graph aligner using seeding and alignment code from Minigraph [24]. We evaluated the scalability and alignment accuracy of PanAligner by using a cyclic human pangenome graph constructed from 94 high-quality haplotype-resolved assemblies [26] and CHM13 human genome assembly [38]. We achieve the highest long-read mapping accuracy 98.7% using PanAligner when compared to existing methods Minigraph [24] (98.1%) and GraphAligner [42] (97.0%).

2 Notations and Problem Formulations

Pangenome graph $G(V, E, \sigma)$ is a string labeled graph such that function $\sigma : V \to \Sigma^+$ labels each vertex v with string $\sigma(v)$ over alphabet $\Sigma = \{A, C, G, T\}$. Let Q be a query sequence over Σ. Let $M[1..N]$ be an array of anchor tuples $(v, [x..y], [c..d])$ with the interpretation that substring $\sigma(v)[x..y]$ from the graph matches substring $Q[c..d]$ in the query sequence. Throughout this paper, all indices start at 1. We will assume that $|E| \geq |V| - 1$. Function *weight* assigns a user-specified weight to each anchor. For example, the weight of an anchor could be proportional to the length of the matching substring.

A path cover is a set $\mathcal{P} = \{P_1, P_2, \ldots, P_{|\mathcal{P}|}\}$ of paths in graph G such that every vertex in V is included in at least one of the $|\mathcal{P}|$ paths. We define $paths(v)$ as $\{i : P_i \text{ includes } v\}$. If $i \in paths(v)$, then let $index(v, i)$ specify the position of vertex v on path P_i. Suppose $\mathcal{R}^-(v)$ is the set of vertices in V that can reach vertex v through any walk in graph G. We will assume that the set $\mathcal{R}^-(v)$ always includes the vertex v. The value $last2reach(v, i)$ for $v \in V, i \in [1, |\mathcal{P}|]$ represents the last vertex on path P_i that belongs to set $\mathcal{R}^-(v)$. Note that $last2reach(v, i)$ does not exist if there is no vertex on path P_i that belongs to $\mathcal{R}^-(v)$. Let $N^+(v)$ and $N^-(v)$ be the set of outgoing and incoming neighbor vertices of vertex v, respectively.

We need to calculate character distances between pairs of anchors in the graph while solving the co-linear chaining problem. Assume that edge $(v, u) \in E$ has length $|\sigma(v)| > 0$. Let $D(v_1, v_2)$ denote the length of the shortest path from vertex v_1 to v_2 in G. We set $D(v_1, v_2) = \infty$ if there is no path from v_1 to v_2, whereas $D(v_1, v_2) = 0$ if $v_1 = v_2$. We use $D^\circ(v)$ to specify the length of the shortest proper cycle containing v. $D^\circ(v) = \infty$ if v is not part of any proper cycle. If P_i includes v, let $dist2begin(v, i)$ denote the length of the sub-path of path P_i from the start of P_i to v.

Our algorithm will use a balanced binary search tree data structure for executing range queries efficiently. It has the following properties.

▶ **Lemma 1** (ref. [31]). *Let n be the number of leaves in a tree, each storing a $(key, value)$ pair. The following operations can be supported in $O(\log n)$ time:*

- *update(k, val): For the leaf w with $key = k$, $value(w) \leftarrow \max(value(w), val)$.*
- *RMQ(l, r): Return $\max\{value(w) \mid l < key(w) < r\}$ such that w is a leaf. This is the range maximum query.*

Given n $(key, value)$ pairs, the tree can be constructed in $O(n \log n)$ time and $O(n)$ space.

Next, we define a precedence relation between a pair of anchors, which is a partial order among the input anchors [30].

▶ **Definition 2** (Precedence). *Given two anchors $M[i]$ and $M[j]$, we define $M[i]$ precedes $(\prec)$ $M[j]$ as follows. If $M[i].v \neq M[j].v$, then $M[i] \prec M[j]$ if and only if $M[i].d < M[j].c$ and $M[i].v$ reaches $M[j].v$. If $M[i].v = M[j].v$, then $M[i] \prec M[j]$ if and only if $M[i].d < M[j].c$, and $M[i].y < M[j].x$ or the graph has a proper cycle containing $M[i].v$.*

▶ **Definition 3** (Chain). *Given the set of anchors $\{M[1], M[2], \ldots, M[N]\}$, a chain is an ordered subset of anchors $S = s_1 s_2 \cdots s_q$ of M, such that s_j precedes s_{j+1} for all $1 \leq j < q$.*

Our co-linear chaining problem formulation seeks a chain $S = s_1 s_2 \cdots s_q$ that maximizes the chain score defined as $\sum_{j=1}^{q} weight(s_j) - \big(\sum_{j=1}^{q-1} gap_Q(s_j, s_{j+1}) + \sum_{j=1}^{q-1} gap_G(s_j, s_{j+1})\big)$. Functions gap_Q and gap_G specify the gap cost incurred in the query sequence and the graph, respectively. Although we specifically focus on problem formulations where the gap cost is calculated as the sum of gap_G and gap_Q, our approach can be extended to other gap definitions such as $|gap_G - gap_Q|$, $\min(gap_G, gap_Q)$, or $\max(gap_G, gap_Q)$, similar to [5]. We define $gap_Q(s_j, s_{j+1})$ as $s_{j+1}.c - s_j.d - 1$, which can be interpreted as the count of characters in the query sequence between the endpoints of the two anchors. Next, we will define two versions of the co-linear chaining problem that differ in their definition of gap_G. In both versions, $gap_G(s_j, s_{j+1})$ is calculated by looking at the count of characters spelled along a walk in the graph from s_j to s_{j+1}. In the first version of the problem formulation, we use the shortest path from vertex $s_j.v$ to $s_{j+1}.v$ to calculate $gap_G(s_j, s_{j+1})$.

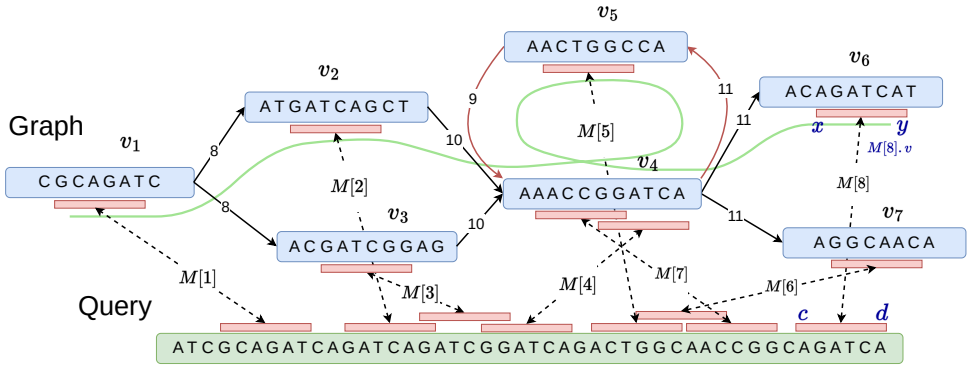


Figure 2 An example illustrating a graph, a query sequence, and multiple anchors as input for co-linear chaining. The sequence of anchors $(M[1], M[2], M[4], M[5], M[7], M[8])$ forms a valid chain that visits vertex v_4 twice due to a cycle in the graph. The coordinates associated with anchor $M[8]$ are also highlighted as an example.

▶ **Problem 4.** *Given a query sequence Q, graph $G(V, E, \sigma)$ and anchors $M[1..N]$, determine the optimal chaining score by using the following definition of gap_G:*

$$gap_G(s_j, s_{j+1}) = \begin{cases} s_{j+1}.x - s_j.y - 1 + D(s_j.v, s_{j+1}.v) & s_{j+1}.v \neq s_j.v \\ s_{j+1}.x - s_j.y - 1 & s_j.v = s_{j+1}.v \text{ and } s_j.y < s_{j+1}.x \\ s_{j+1}.x - s_j.y - 1 + D^{\circ}(s_j.v) & s_j.v = s_{j+1}.v \text{ and } s_j.y \geq s_{j+1}.x, \end{cases}$$

where (s_j, s_{j+1}) is a pair of anchors from M such that s_j precedes s_{j+1}.

▶ **Lemma 5.** *Problem 4 can be solved in $\Theta(|V||E| + |V|^2 \log |V| + N^2)$ time.*

Proof. Compute the shortest distance $D(v_i, v_j)$ between all pairs of vertices $v_i, v_j \in V$ in $O(|V||E| + |V|^2 \log |V|)$ time by using Dijkstra's algorithm from every vertex. Next, compute $D^{\circ}(v)$ as $\min_{u \in N^+(v)} |\sigma(v)| + D(u, v)$ in $\Theta(|E|)$ time for all $v \in V$. These computations need to be done only once for a graph. To solve the chaining problem for a given query sequence, sort the input anchor array $M[1..N]$ in non-decreasing order by the component $M[\cdot].c$. Let $C[1..N]$ be a one-dimensional table in which $C[j]$ will be the optimal score of a chain ending at anchor $M[j]$. Initialize $C[j]$ as $weight(M[j])$ for all $j \in [1, N]$. Subsequently, compute C in the left-to-right order by using the recursion $C[j] = \max_{M[i] \prec M[j]} \{C[j], weight(M[j]) - gap_Q(M[i], M[j]) - gap_G(M[i], M[j])\}$. Computing $C[j]$ takes $\Theta(N)$ time because precedence condition can be checked in constant time. Report $\max_j C[j]$ as the optimal chaining score. ◀

The above algorithm is unlikely to scale to large whole-genome sequencing datasets because it requires $\Theta(N^2)$ time for the dynamic programming recursion. Motivated by [5], we will define an alternative definition of gap_G. We will approximate the distance between a pair of vertices by using a path cover of the graph. We will later propose an efficient algorithm for the revised problem formulation.

Suppose $\mathcal{P} = \{P_1, P_2, \ldots, P_{|\mathcal{P}|}\}$ is a path cover of graph G. Consider a pair of vertices $v_1, v_2 \in V$ such that v_1 reaches v_2. For each path $i \in paths(v_1)$, consider the walk starting from v_1 along the edges of path P_i till vertex α_i, where vertex $\alpha_i = v_2$ if v_2 also lies on path P_i anywhere after v_1, i.e., $index(v_2, i) \geq index(v_1, i)$, and $\alpha_i = last2reach(v_2, i)$ otherwise. If $\alpha_i \neq v_2$, the rest of the walk till v_2 is completed by using the shortest path from vertex α_i to v_2. Denote $D_{\mathcal{P}}(v_1, v_2)$ as the length of the shortest walk among such $|paths(v_1)|$ possible walks from v_1 to v_2. Formally, we can write $D_{\mathcal{P}}(v_1, v_2)$ as

$$\min_{i \in paths(v_1)} dist2begin(\alpha_i, i) - dist2begin(v_1, i) + D(\alpha_i, v_2). \quad (1)$$

$D_{\mathcal{P}}(v_1, v_2)$ is well defined if v_2 is reachable from v_1. We set $D_{\mathcal{P}}(v_1, v_2) = \infty$ if v_2 is not reachable from v_1. Finally, if vertex v is part of a proper cycle in G, we define $D^{\circ}_{\mathcal{P}}(v)$ as the length of a specific walk that starts and ends at v, i.e., $D^{\circ}_{\mathcal{P}}(v)$ as $\min_{u \in N^+(v)} |\sigma(v)| + D_{\mathcal{P}}(u, v)$ for all $v \in V$. $D^{\circ}_{\mathcal{P}}(v) = \infty$ if v is not part of any proper cycle.

▶ **Problem 6.** *Given a query sequence Q, graph $G(V, E, \sigma)$ and anchors $M[1..N]$, determine a path cover $\mathcal{P}$ of the graph, and the optimal chaining score by using the following definition of gap_G:*

$$gap_G(s_j, s_{j+1}) = \begin{cases} s_{j+1}.x - s_j.y - 1 + D_{\mathcal{P}}(s_j.v, s_{j+1}.v) & s_{j+1}.v \neq s_j.v \\ s_{j+1}.x - s_j.y - 1 & s_j.v = s_{j+1}.v \text{ and } s_j.y < s_{j+1}.x \\ s_{j+1}.x - s_j.y - 1 + D^{\circ}_{\mathcal{P}}(s_j.v) & s_j.v = s_{j+1}.v \text{ and } s_j.y \geq s_{j+1}.x, \end{cases}$$

where (s_j, s_{j+1}) is a pair of anchors from M such that s_j precedes s_{j+1}.

3 Proposed Algorithms

A single experiment typically requires aligning millions of reads to a graph. Therefore, we will do a one-time preprocessing of the graph that will help reduce the runtime of our chaining algorithm for solving Problem 6.

3.1 Algorithms for Preprocessing the Graph

We compute the following quantities during the preprocessing stage:

- A path cover $\mathcal{P}$ of $G(V, E, \sigma)$. We require the path cover to be small (in the number of paths). However, determining the minimum path cover in a graph with cycles is an NP-hard problem. We will discuss an efficient heuristic for determining a small path cover.
- A bijective function $rank : V \to [1, |V|]$ that specifies a linear ordering of vertices. The ordering should satisfy the following property: If vertex v_2 occurs anywhere after v_1 in a path in $\mathcal{P}$, then $rank(v_2) > rank(v_1)$ for all $v_1, v_2 \in V$. Such an ordering may not exist for an arbitrary path cover but it will exist for the path cover chosen by us.
- $last2reach(v, i)$, $D(last2reach(v, i), v)$, $dist2begin(v, i)$ and $D^{\circ}_{\mathcal{P}}(v)$ for all $v \in V$ and $i \in [1, |\mathcal{P}|]$. These values will be frequently used by our chaining algorithm to compute gap costs.

We propose the following heuristic for computing a small path cover of graph $G(V, E, \sigma)$. We derive a DAG $G'(V, E', \sigma)$ from G by removing a small number of edges. Next, we determine the minimum path cover $\mathcal{P}$ of G' in $O(|\mathcal{P}||E| \log |V|)$ time by using a known algorithm [33]. Our intuition is that removing as few edges as possible will provide a close to optimal path cover of G. One way to compute G' is to use standard heuristic-based solvers for minimum feedback arc set (FAS) problem, e.g., [10], but we empirically observed that this approach could sometimes disconnect a weak component of a graph, leading to a large path cover. Therefore, instead of using FAS heuristics, we use a simple idea where we identify all strongly connected components [50] and perform a depth-first search within each strong component to remove back edges. This approach provides a DAG that has the same number of weak

components as G while removing a small number of edges in practice, thus resulting in a small path cover. Next, we compute a function $rank$ for all vertices $\in V$ by topological sorting of vertices in DAG G'.

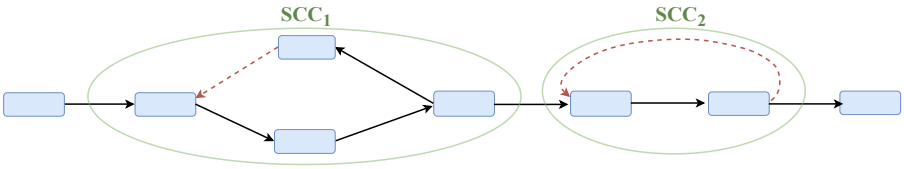


Figure 3 An illustration of the proposed heuristic used to convert a cyclic graph into a DAG. Red-dotted edges represent the removed back edges in each strongly connected component (SCC).

If there is no cycle in G, then $last2reach(v,i)$ and $D(last2reach(v,i),v)$ can be computed in $O(|\mathcal{P}||E|)$ time by using dynamic programming algorithms that process vertices in topological order [5, 33]. We extend these ideas to cyclic graphs by designing iterative algorithms. We will formally prove that as the iterations proceed, the output gets closer to the desired solution. Our approach to computing $last2reach(v,i)$ is outlined in Algorithm 1. If $last2reach(v,i)$ exists, the algorithm determines it in terms of its $rank$. We maintain an array $L2R$ to save intermediate results. $L2R(v,i)$ is initialised to $rank(v)$ if v lies on path P_i. In each iteration, we revise $L2R(v,i)$ by probing $L2R(u,i)$ for all $u \in N^-(v)$. In Lemma 7, we prove the correctness of this algorithm by arguing that all $|\mathcal{P}||V|$ values in array $L2R$ converge to their optimal values through label propagation in $\leq |V|$ iterations. Let Γ_l denote the count of iterations used by the algorithm. $L2R(v,i)$ remains 0 if $last2reach(v,i)$ does not exist.

Algorithm 1 $O(\Gamma_l|\mathcal{P}||E|)$ time algorithm to compute $last2reach(v,i)$ for all $v \in V$ and $i \in [1,|\mathcal{P}|]$.

```
1  Initialize L2R(v,i) to rank(v) if i ∈ paths(v) and 0 otherwise for all v ∈ V and i ∈ [1,|P|]
2  Initialize L2R_prev(v,i) to 0 for all v ∈ V and i ∈ [1,|P|]
3  /*L2R and L2R_prev will hold the values of current and previous iteration respectively*/
4  while ∃v ∈ V, ∃i ∈ [1,|P|], L2R(v,i) ≠ L2R_prev(v,i) do
5      for i ∈ [1,|P|] do
6          for v ∈ V in the increasing order of rank(v) do
7              L2R_prev(v,i) ← L2R(v,i)
8              L2R(v,i) ← max_{u∈N⁻(v)∪v} L2R(u,i)
9          end
10     end
11 end
```

▶ **Lemma 7.** *In Algorithm 1, $L2R(v,i)$ converges to the rank of $last2reach(v,i)$ in at most $|V|$ iterations for all $v \in V$ and $i \in [1,|\mathcal{P}|]$.*

Proof. A vertex $v_2 \in V$ is said to be reachable within k hops from vertex $v_1 \in V$ if there exists a path with $\leq k$ edges from v_1 to v_2. We will prove by induction that Algorithm 1 satisfies the following invariant: After j iterations, $L2R(v,i)$ has converged to $rank(last2reach(v,i))$ if $last2reach(v,i)$ exists and vertex v is reachable within j hops from $last2reach(v,i)$ in G. This argument will prove the lemma because vertex $v_2 \in V$ must be reachable within $|V|-1$ hops from $v_1 \in V$ if v_2 is reachable from v_1. Base case ($j = 0$) holds due to initialisation of $L2R(v,i)$ in Line 1. If v lies 0-hop from $last2reach(v,i)$, i.e., $v = last2reach(v,i)$, then v must lie on path P_i and $rank(last2reach(v,i)) = rank(v)$. Next, assume that the

invariant is true for $j = n$. Now consider the situation after $n + 1$ iterations. Suppose $v \in V$ is reachable within $n + 1$ hops from $last2reach(v, i)$. Then, at least one neighbor $u \in N^-(v)$ of vertex v exists which is reachable within n hops from $last2reach(v, i)$ and $last2reach(u, i) = last2reach(v, i)$. Based on our assumption, $L2R(u, i)$ must have already converged to $rank(last2reach(u, i))$ before $(n+1)^{th}$ iteration. Therefore, Line 8 in Algorithm 1 ensures that $L2R(v, i) \leftarrow rank(last2reach(v, i))$ after $(n + 1)^{th}$ iteration. ◀

It is possible to design an adversarial example where the algorithm uses $\Omega(|V|)$ iterations. However, in practice, we expect the algorithm to converge quickly. Each iteration of Algorithm 1 requires $O(|\mathcal{P}||E|)$ time. Therefore, the total worst-case time of Algorithm 1 is bounded by $O(\Gamma_l|\mathcal{P}||E|)$. A similar approach is applicable to compute $D(last2reach(v, i), v)$ for all $v \in V$ and $i \in [1, |\mathcal{P}|]$ (Algorithm 2). We use Γ_d to denote the count of iterations needed in Algorithm 2. Similar to parameter Γ_l in Algorithm 1, Γ_d is also upper bounded by $|V|$. We will later show empirically that $\Gamma_l \ll |V|$ and $\Gamma_d \ll |V|$ in practice.

Algorithm 2 $O(\Gamma_d|\mathcal{P}||E|)$ time algorithm to compute $D(last2reach(v, i), v)$ for all $v \in V$ and $i \in [1, |\mathcal{P}|]$.

```
1  Initialize D(last2reach(v,i),v) to 0 if last2reach(v,i) = v and ∞ otherwise
2  Initialize D_prev(last2reach(v,i),v) to ∞
3  /*Arrays D and D_prev will hold the values of current and previous iteration respectively*/
4  while ∃v ∈ V, ∃i ∈ [1,|P|], D(last2reach(v,i),v) ≠ D_prev(last2reach(v,i),v) do
5      for i ∈ [1,|P|] do
6          for v ∈ V in the increasing order of rank(v) do
7              D_prev(last2reach(v,i),v) ← D(last2reach(v,i),v)
8              if last2reach(v,i) exists and last2reach(v,i) ≠ v then
9                  D(last2reach(v,i),v) ←
                     min_{u:u∈N⁻(v),last2reach(u,i)=last2reach(v,i)} D(last2reach(u,i),u) + |σ(u)|
10             end
11         end
12     end
13 end
```

Array $dist2begin$ is trivially precomputed in $O(|\mathcal{P}||V|)$ time. $D^\circ_\mathcal{P}(v)$ is computed as $\min_{u \in N^+(v)} |\sigma(v)| + D_\mathcal{P}(u, v)$ based on its definition. $D_\mathcal{P}(u, v)$ can be calculated by using Equation 1 for any $u, v \in V$ in $O(|\mathcal{P}|)$ time. Accordingly, computation of $D^\circ_\mathcal{P}(v)$ for all $v \in V$ is done in $O(|\mathcal{P}||E|)$ time. The following lemma summarises the worst-case time complexity of all the preprocessing steps.

▶ **Lemma 8.** *Preprocessing of graph $G(V, E, \sigma)$ requires $O((\Gamma_l + \Gamma_d + \log |V|)|\mathcal{P}||E|)$ time.*

3.2 Co-linear Chaining Algorithm

We propose an iterative chaining algorithm to address Problem 6. The proposed algorithm builds on top of the known algorithms for DAGs [5, 33]. Similar to [33], we maintain one search tree $\mathcal{T}_i$ for each path $P_i \in \mathcal{P}$. Given anchors $M[1..N]$, our algorithm will return array $C[1..N]$ such that $C[j]$ corresponds to the optimal score of a chain that ends at anchor $M[j]$.

If there are no cycles in G, then one iteration of Algorithm 3 suffices to compute the optimal chaining scores. For a DAG, a single iteration of Algorithm 3 works equivalently to the known algorithm for DAGs in [5]. In this case, Algorithm 3 would essentially visit the vertices of graph G in topological order while ensuring that $C[j]$ is optimally solved after $M[j].v$ is visited. To solve the chaining problem on cyclic graphs, we design an iterative solution where chaining scores $C[1..N]$ get closer to optimal values in each iteration. We will use Γ_c to specify the total count of iterations.

Algorithm 3 $O(\Gamma_c N|\mathcal{P}|\log N + N|\mathcal{P}|\log N|\mathcal{P}|)$ time chaining algorithm.

Input: Array of weighted anchors $M[1..N]$, preprocessed $G(V, E, \sigma)$
Output: Array $C[1..N]$ such that $C[j]$ equals score of an optimal chain that ends at anchor $M[j]$
Initialize search tree $\mathcal{T}_i$, for all $i \in [1, |\mathcal{P}|]$ using keys $\{M[j].d \mid 1 \leq j \leq N\}$ and values $-\infty$
Initialize $C[j]$ as $weight(M[j])$ and $C_{prev}[j] \leftarrow 0$, for all $j \in [1, N]$
/* Create array Z that stores tuples of the form $(v, pos, task, anchor, path)$ where $v \in V$, $pos \in \mathbb{N}$, $task \in \{1, 2, 3\}$, $anchor \in [1, N]$ and $path \in [1, |\mathcal{P}|]$*/
for $j \leftarrow 1$ *to* N **do**
  for $i \leftarrow 1$ *to* $|\mathcal{P}|$ **do**
    if $i \in paths(M[j].v)$ **then**
      $Z.push(M[j].v, M[j].x, 1, j, i)$
      $Z.push(M[j].v, M[j].y, 2, j, i)$
    end
    if $last2reach(M[j].v, i)$ *exists and* $last2reach(M[j].v, i) \neq M[j].v$ **then**
      $v \leftarrow last2reach(M[j].v, i)$
      $Z.push(v, |\sigma(v)| + 1, 1, j, i)$
    end
    if $M[j].v$ *is contained in a proper cycle in* G *and* $i \in paths(M[j].v)$ **then**
      $Z.push(v, |\sigma(M[j].v)| + 1, 3, j, i)$
    end
  end
end
while $\exists j \in [1, N]$, $C_{prev}[j] \neq C[j]$ **do**
  $C_{prev}[j] \leftarrow C[j]$, for all $j \in [1, N]$
  for $z \in Z$ *in lexicographically ascending order based on the key* $(rank(v), pos, task)$ **do**
    $j \leftarrow z.anchor, i \leftarrow z.path, v \leftarrow z.v, wt \leftarrow weight(M[j])$
    if $z.task = 1$ **then**
      $gap \leftarrow (M[j].x + Dist2begin(v, i) + D(v, M[j].v) + M[j].c - 2)$
      $C[j] \leftarrow \max(C[j],\ wt + \mathcal{T}_i.\mathsf{RMQ}(0, M[j].c) - gap)$
    end
    else if $z.task = 2$ **then**
      $\mathcal{T}_i.\mathsf{update}(M[j].d, C[j] + M[j].y + Dist2begin(v, i) + M[j].d)$
    end
    else if $z.task = 3$ **then**
      $gap^\circ \leftarrow (M[j].x + Dist2begin(v, i) + D^\circ_{\mathcal{P}}(v) + M[j].c - 2)$
      $C[j] \leftarrow \max(C[j],\ wt + \mathcal{T}_i.\mathsf{RMQ}(0, M[j].c) - gap^\circ)$
      $\mathcal{T}_i.\mathsf{update}(M[j].d,\ C[j] + M[j].y + Dist2begin(v, i) + M[j].d)$
    end
  end
  Reset all values in search tree $\mathcal{T}_i$ to $-\infty$, for all $i \in [1, |\mathcal{P}|]$
end

An overview of Algorithm 3 is as follows. At the beginning of each iteration, all search trees $\mathcal{T}_i$s are filled with keys $\{M[j].d \mid 1 \leq j \leq N\}$ and values $-\infty$. The values will be used to specify the priorities of anchors based on their scores $C[\,]$ and coordinates. Each iteration of our algorithm processes $v \in V$ in the increasing order of $rank(v)$. While processing v, Algorithm 3 performs three types of tasks:

1. The first type of task is to revise chaining scores $\{C[j] : M[j].v = v\}$ corresponding to the anchors that lie on vertex v. We also revise scores corresponding to those anchors that are located on vertex $u \neq v$ such that v is the last vertex on a path $\in \mathcal{P}$ to reach u. This is achieved by querying search trees $\mathcal{T}_i$ for all $i \in paths(v)$. In all these tasks, we use $D_{\mathcal{P}}(v_1, v_2)$ to calculate distance from vertex $v_1 \in V$ to vertex $v_2 \in V$.
2. Suppose score $C[j]$ is revised by using the first category tasks. The second type of task is to update the value of key $M[j].d$ in search trees $\mathcal{T}_i$ for all $i \in paths(v)$. The value gets updated if the new value is greater than the previously stored value (Lemma 1).
3. The third type of task is to again update scores $\{C[j] : M[j].v = v\}$ and search trees if v is part of a proper cycle in G. Here we use $D^\circ_{\mathcal{P}}(v)$ to calculate the distance of vertex v to itself while determining gap costs.

Lines 4–18 in Algorithm 3 build array Z that contains up to $4N|\mathcal{P}|$ tuples corresponding to all the above type of tasks. Array Z is sorted in $O(N|\mathcal{P}| \log N|\mathcal{P}|)$ time to ensure that all tasks are executed in the proper order (Line 21). Next, we start the iterative procedure. Lines 19–33 form a single iteration of the algorithm. These tasks lead to updates on score array C and the search trees. The arithmetic operations in Lines 24, 25, 31, 32 enable calculation of gap cost based on our definitions of gap_G and gap_Q in Section 2. Each iteration requires $O(N|\mathcal{P}| \log N)$ time because each task corresponds to either update or RMQ operation on a search tree of size $\leq N$. In Lemma 9, we prove that array $C[1..N]$ converges to optimality in at most N iterations. We will also prove that $\Omega(N)$ iterations are required for convergence in the worst case.

▶ **Lemma 9.** *In Algorithm 3, co-linear chaining scores $C[1..N]$ converge to optimality in $\leq N$ iterations.*

Proof. $C[j]$ always specifies the score of a chain of size ≥ 1 that ends at anchor $M[j]$ throughout the execution of the algorithm. Let $f_i(j)$ denote the optimal chaining score ending at anchor $M[j]$ over all chains of size $\leq i$. We will prove by induction that before i^{th} iteration begins, $C[j] \geq f_i(j)$ for all $j \in [1, N]$. It suffices to prove this statement because the size of a chain must be $\leq N$. Base case ($i = 1$) holds due to the initialization step in Line 2. Next, assume that before x^{th} iteration begins, $C[j] \geq f_x(j)$ holds for all $j \in [1, N]$. We will prove that the invariant holds for iteration $x + 1$.

Let $C_x[j]$ and $C_{x+1}[j]$ denote the intermediate values of $C[j]$ at the start of x^{th} and $(x+1)^{th}$ iteration, respectively. From Lines 25 and 32, we know $C_x[j] \leq C_{x+1}[j]$. If $f_{x+1}(j) = f_x(j)$, then $C_{x+1}[j] \geq C_x[j] \geq f_x(j) = f_{x+1}(j)$. Next consider the other case when $f_{x+1}(j) > f_x(j)$. Suppose the optimal chain corresponding to $f_{x+1}(j)$ is $M[\beta_1], M[\beta_2], \ldots, M[\beta_x], M[j]$ where $\beta_i \in [1, N]$ for all $i \in [1, x]$. Accordingly, $f_{x+1}(j) = weight(M[j]) + f_x(\beta_x) - gap_Q(M[\beta_x], M[j]) - gap_G(M[\beta_x], M[j])$. Based on our induction hypothesis, $C[\beta_x] \geq f_x(\beta_x)$ at the start of the x^{th} iteration. Each iteration of Algorithm 3 processes $v \in V$ by increasing the order of $rank(v)$. To prove that $C_{x+1}[j] \geq f_{x+1}(j)$, we have the following four cases to consider:

- Case 1: $rank(M[\beta_x].v) < rank(M[j].v)$. The algorithm processes vertex $M[\beta_x].v$ before vertex $M[j].v$. When $M[\beta_x].v$ is processed during the x^{th} iteration, the value of key $M[\beta_x].d$ gets updated in search trees (Line 28). $C[j]$ gets updated later. At the end of the x^{th} iteration, $C[j] \geq weight(M[j]) + f_x(\beta_x) - gap_Q(M[\beta_x], M[j]) - gap_G(M[\beta_x], M[j])$. Therefore, $C_{x+1}[j] \geq f_{x+1}(j)$.
- Case 2: $rank(M[\beta_x].v) > rank(M[j].v)$. In this case, $C[j]$ may not meet the desired threshold after $M[j].v$ is processed because $M[\beta_x].v$ is processed later than $M[j].v$. However, $M[j].v$ must be reachable from $M[\beta_x].v$ using walks through $\{last2reach(M[j].v, i) : i \in paths(M[\beta_x].v)\}$. Therefore, $C[j]$ gets updated again due to tuples created in Line 12. This will ensure that $C_{x+1}[j] \geq f_{x+1}(j)$.
- Case 3: $rank(M[\beta_x].v) = rank(M[j].v)$ and $M[\beta_x].y < M[j].x$. $rank(M[\beta_x].v) = rank(M[j].v)$ implies $M[\beta_x].v = M[j].v$. The ordering of tuples based on *pos* in Line 21 ensures that the value of key $M[\beta_x].d$ gets updated in search trees, and $C[j]$ gets updated afterward.
- Case 4: $rank(M[\beta_x].v) = rank(M[j].v)$ and $M[\beta_x].y \geq M[j].x$. The tuples created in Line 15 ensure that $C[j]$ is updated again after finishing the processing of vertex $M[j].v$. In this case, the gap between anchors $M[\beta_x]$ and $M[j]$ is computed by considering the distance of vertex $M[j].v$ to itself, i.e., $D^{\circ}_{\mathcal{P}}(M[j].v)$. ◀

Accordingly, the time complexity of Algorithm 3 is $O(\Gamma_c N|\mathcal{P}|\log N + N|\mathcal{P}|\log N|\mathcal{P}|)$. In our experiments, we will highlight that parameters Γ_c and $|\mathcal{P}|$ are small in practice. The space complexity of the algorithm is $O(N|\mathcal{P}| + |V||\mathcal{P}|)$ due to construction of array Z, the sorting operation on array Z, $|\mathcal{P}|$ search trees and the precomputed data structures. Next, we show that $O(N)$ upper bound on the number of iterations is tight.

▶ **Lemma 10.** *The count of iterations required by Algorithm 3 is $\Omega(N)$ in the worst-case.*

Proof. An example where Algorithm 3 requires $\Omega(N)$ iterations is shown in Figure 4. The graph has two vertices forming a cycle. Assume that *weight* of all N input anchors is equal and sufficiently high to outweigh the gap cost between any pair of anchors. As $M[1] \prec M[2] \prec M[3] \ldots \prec M[N]$, the sequence of anchors in the optimal chain is $(M[1], M[2], M[3], \ldots, M[N-1], M[N])$. After the first iteration of the algorithm, the size of the highest scoring chain computed until then will be $\frac{N}{2} + 1$. The size will grow slowly by one in each subsequent iteration. A step-by-step dry run of the algorithm is left for the journal version of this paper. ◀

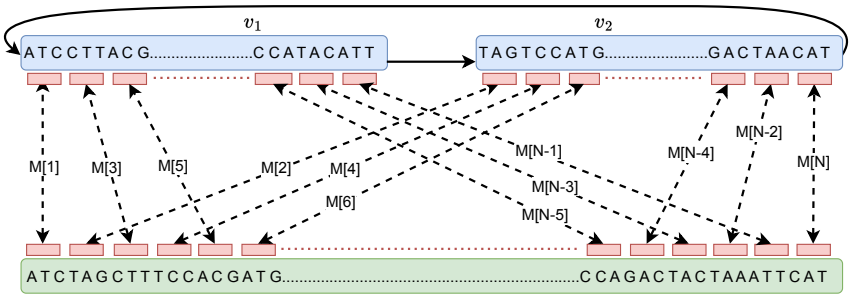


Figure 4 A worst-case example for Algorithm 3 where it requires $\Omega(N)$ iterations to converge.

4 Implementation

We have implemented the proposed algorithm in `C++` (https://github.com/at-cg/PanAligner). We call our software as PanAligner. PanAligner is developed as an end-to-end long-read aligner for cyclic pangenome graphs. We borrow open-source code from Minichain [5], Minigraph [24], and GraphChainer [30] for other necessary components besides co-linear chaining. While using PanAligner, a user needs to provide a graph (`GFA` format) and a set of reads or contigs (`fasta` or `fastq` format) as input. We use the standard data structure to store the pangenome graph while accounting for double stranded nature of DNA sequences. For each vertex $v \in V$, we also add another vertex $\bar{v}$ whose string label is the reverse complement of string $\sigma(v)$. For each edge $u \to v \in E$, we add the complementary edge $\bar{v} \to \bar{u}$. This enables read alignment irrespective of which strand the read was sequenced from.

For the benchmark, we built pangenome graphs by using Minigraph v0.20 [24]. Minigraph augments large insertion, deletion, and inversion variants into the graph while incrementally aligning each input assembly. Inversion variants can introduce cycles in the graph because Minigraph augments them by linking the vertices from opposite strands. The graph contains multiple weakly connected components because the components corresponding to different chromosomes are never linked during graph construction. Similar to [5, 30], we consider each weak component independently during both the preprocessing and co-linear chaining stages to enable efficient multithreading and memory optimization.

We defined our problem formulation to produce an optimal chain, but we actually compute multiple best chains, similar to [5, 23, 24]. This is because there can be multiple high-scoring alignments of a read on the graph. PanAligner also outputs a mapping quality score between 0 to 60 to indicate the confidence score for each alignment [25]. We used seeding and extension code from Minigraph [24]. Seeding is done by identifying minimizer matches [45] between vertex labels of the graph and the read. The extension code produces the final base-to-base alignment by joining the chained anchors [52]. We used code from GraphChainer [30] to compute the minimum path cover of a DAG and range queries.

5 Experiments

Benchmark Datasets

We constructed four cyclic pangenome graphs by using subsets of publicly available 95 haplotype-resolved human genome assemblies [26, 37]. These graphs were generated using Minigraph v0.20 [24]. We used CHM13 human genome assembly [37] as the starting sequence during graph construction in all four graphs. We refer to these graphs as 10H, 40H, 80H, and 95H, where the prefix integer represents the count of haplotypes in each graph. The properties of these graphs are provided in Table 1.

Table 1 Properties of four cyclic pangenome graphs that were used for evaluation.

Graph	|V|	|E|	No. of weak components	No. of structural variants	N50 length of vertex labels (kb)
10H	283,296	406,292	30	61,523	225
40H	679,846	978,122	28	149,163	127
80H	1,106,286	1,594,980	26	244,372	85
95H	1,224,853	1,765,222	26	270,888	79

Evaluation Methodology

We simulated long reads using PBSIM2 v2.0.1 [39] from CHM13 assembly with N50 length 10 kb, 0.5× sequencing coverage and 5% error-rate to approximately mimic the properties of long-reads. We labeled the IDs of the simulated reads with their true interval coordinates in the CHM13 assembly for correctness evaluation. To confirm the correctness of a read alignment, we used similar criteria from prior studies [5, 23, 24]. We require that the reported walk corresponding to a correct alignment should only use the vertices corresponding to the CHM13 assembly in the graph, and it should overlap with the true walk. We used `paftools` [23] to automate this evaluation. By default, it requires the overlapping portion to be at least 10% of the union of the true and the reported walk length. We executed all experiments on a computer with AMD EPYC 7763 64-core processor and 512 GB RAM. We ran each aligner using 32 threads to leverage the multi-threading capabilities of the tested aligners. All aligners process reads in parallel. We used the `/usr/bin/time -v` command to measure wall clock time and peak memory usage.

Size of Path Cover and Count of Iterations

Finding a suitable path cover $\mathcal{P}$ of the input graph such that $|\mathcal{P}| \ll |V|$ is a crucial step in our proposed framework because the scalability of our algorithms depends on this parameter. We discussed a heuristic to compute path cover in Section 3.1 because determining minimum

path cover in general graphs is NP-hard. Table 2 shows the sizes of path covers computed by our heuristic in all four graphs. Recall that our algorithms process the weakly connected components of a graph independently. In each graph, we indicate the size of the path cover as a range because path covers vary per component. The results show that our heuristic is effective in finding a small path cover (in the number of paths).

Table 2 All four graphs have multiple weakly connected components. Therefore, the size of the identified path cover of each graph is presented as a range. The other columns show the statistics for the number of anchors and the number of iterations used by our iterative algorithms (Algorithms 1, 2, 3). The statistics were gathered while aligning simulated long reads to cyclic pangenome graphs.

Graph	Size of path cover (min-max)	Number of anchors		Number of iterations					
				Array *last2reach*		Array D		Chaining	
		Mean	Max	Mean	Max	Mean	Max	Mean	Max
10H	1-20	10,900	309,591	2.0	4	2.0	5	2.3	77
40H	1-36	10,850	309,603	2.0	4	2.0	5	2.4	72
80H	1-49	10,804	309,364	3.0	4	3.0	5	2.4	61
95H	1-59	10,798	309,367	3.0	4	3.0	5	2.4	64

The number of anchors N that were provided as input to the co-linear chaining algorithm varies per read. We report the mean and maximum value in Table 2. Observe that N does not change much with increasing haplotype count. Next, we evaluate the count of iterations Γ_l, Γ_d used by our graph preprocessing algorithms (Algorithms 1–2) and also report them as a range for each graph. These algorithms compute *last*2*reach* and D arrays. Observe that the iteration count is significantly smaller in practice than the proven upper limit of $|V|$ (Lemma 7). This is because the worst-case situation is not observed in practice. Accordingly, there is minimal time overhead during the preprocessing phase.

The count of iterations Γ_c required by our chaining algorithm (Algorithm 3) varies per component as well as per read. We collect the iteration count statistics as follows. For a single read, we define the iteration count as the maximum number of iterations used over all components. Based on this definition, we report the average and the maximum count over all reads in Table 2. Observe that the average count is < 2.5 using all four graphs. The maximum count is < 100. These numbers are again significantly better compared to the upper bound from Lemma 9.

Alignment of Simulated Reads to Cyclic Graphs

We assessed the performance of PanAligner against two sequence-to-graph aligners, Minigraph v2.20 [24] and GraphAligner v1.0.17b [42], that can handle cycles. Unlike PanAligner, Minigraph and GraphAligner use heuristics to join anchors. Minichain [5] and GraphChainer [30] were excluded from this comparison because they do not support cyclic graphs.

We highlight the accuracy, runtime, and memory usage of different aligners using graphs 10H and 95H in Tables 3 and 4, respectively. Observe that PanAligner outperformed Minigraph and GraphAligner in terms of accuracy, i.e., the fraction of correctly aligned reads. This advantage is even more apparent if low-confidence alignments with mapping quality < 10 are ignored. We show the comparison plots in Figure 5. Both PanAligner and Minigraph left a small fraction of reads unaligned. This may be because (i) both methods drop high-frequency minimizer matches, and (ii) they do not consider low-scoring chains for the extension stage. In contrast, GraphAligner achieved higher recall by aligning all reads, but this came at the expense of lower accuracy.

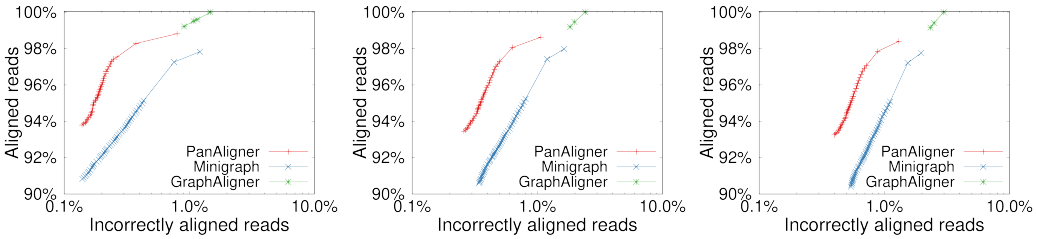
Table 3 A comparison of the performance of long-read aligners using the 10H graph. MQ stands for mapping quality.

	PanAligner	Minigraph	GraphAligner
Indexing time (sec)	96	**57**	238
Alignment time (sec)	2924	**46**	4928
Memory usage (GB)	**23.14**	23.19	24.68
Unaligned reads	1.18%	2.17%	**0%**
Incorrectly Aligned reads	**0.79%**	1.19%	1.47%
Unaligned reads (MQ≥10)	3.51%	5.85%	**0.78%**
Incorrectly Aligned reads (MQ≥10)	**0.20%**	0.32%	0.91%

Table 4 A comparison of the performance of long-read aligners using the 95H graph. MQ stands for mapping quality.

	PanAligner	Minigraph	GraphAligner
Indexing time (sec)	83	**61**	272
Alignment time (sec)	9276	**58**	5170
Memory usage (GB)	43.6	**24.68**	26.1
Unaligned reads	1.60%	2.24%	**0%**
Incorrectly Aligned reads	**1.28%**	1.93%	2.98%
Unaligned reads (MQ≥10)	4.20%	6.21%	**0.84%**
Incorrectly Aligned reads (MQ≥10)	**0.57%**	0.85%	2.33%

Table 2 shows that the size of the path cover computed by our heuristic increases by roughly a factor of three from 10H to 95H. We can see how this parameter proportionally affects PanAligner's runtime in Tables 3 and 4. PanAligner's runtime is significantly higher than Minigraph for both 10H and 95H graphs because it uses an iterative algorithm. Runtimes of PanAligner and GraphAligner are in the same order of magnitude. PanAligner's computational needs are within practical limits, thus making it an effective method for accurately aligning long reads or contigs to cyclic pangenome graphs. We observe a consistent drop in alignment accuracy of all three aligners with increasing haplotype count (Figure 5). This is likely because the number of combinatorial paths to which a read can align grows exponentially with respect to the haplotype count.



(a) 10H. **(b)** 40H. **(c)** 95H.

Figure 5 The plots show the fraction of aligned reads and the accuracy obtained by using all the aligners on graphs 10H, 40H, and 95H. These plots were generated by varying mapping quality cutoffs from 0 to 60. X-axis in these plots uses a logarithmic scale to indicate the percentage of incorrectly aligned reads.

Alignment of Simulated Reads to Acyclic Graphs

We also tested PanAligner for acyclic pangenome graphs. We followed the same procedure as [5] to generate a DAG from 95 haplotype-phased assemblies and refer to this graph as 95H-DAG. This graph was generated by disabling inversion variants during graph construction in Minigraph [24]. 95H-DAG has 1.2M vertices and 1.8M edges. We also include Minichain v1.0 [5] and GraphChainer v1.0.2 [30] in this comparison. GraphChainer uses a co-linear chaining algorithm for DAGs without penalizing gaps. For DAG inputs, the problem formulation in PanAligner becomes equivalent to the one used in Minichain [5]. A single iteration of our algorithms suffices for DAGs. Therefore, we simply check if the input graph is a DAG at the preprocessing stage, and run a single iteration of Algorithms 1–3. PanAligner achieves similar performance as Minichain in terms of speed and accuracy for DAGs (Table 5). It compares favorably to other methods in terms of accuracy.

Table 5 A comparison of the performance of long-read aligners using the 95H-DAG graph. MQ stands for mapping quality.

	Pan-Aligner	**Minichain**	**Mini-graph**	**Graph-Aligner**	**Graph-Chainer**
Indexing time(sec)	78	77	**62**	276	575
Alignment time(sec)	2406	2515	**50**	5136	23710
Memory usage (GB)	30.04	25.61	**24.79**	26.12	185.83
Unaligned reads	1.62%	1.62%	2.23%	**0%**	**0%**
Incorrectly Aligned reads	**1.28%**	1.29%	1.92%	3.06%	4.93%
Unaligned reads (MQ≥10)	4.75%	4.75%	6.26%	0.85%	**0%**
Incorrectly Aligned reads (MQ≥10)	**0.53%**	0.54%	0.84%	2.41%	4.93%

6 Discussion

Co-linear chaining is a fundamental technique for scalable sequence alignment. Several classes of structural variants, such as duplications, tandem repeat polymorphism, and inversions, are best represented as cycles in pangenome graphs [41, 26]. Existing alignment software designed for cyclic graphs are based on heuristics to join anchors [24, 42]. We proposed the first practical problem formulation and an efficient algorithm for co-linear chaining on pangenome graphs with cycles. We gave a rigorous analysis of the algorithm's time complexity. The proposed algorithm serves as a useful generalization of the previously known ideas for DAGs [5, 29, 30, 33]. We implemented the proposed algorithm as an open-source software PanAligner. We demonstrated that PanAligner scales to large pangenome graphs built by using haplotype-phased human genome assemblies. It offers superior alignment accuracy compared to existing methods.

In our formulation, we did not allow anchors to span two or more vertices in a graph for simplicity of notations, but the proposed ideas can be generalized. PanAligner software borrows seeding logic from Minigraph [24], which also restricts anchors within a single vertex. This simplification is appropriate if the graph only includes structural variants (> 50 bp). The current version of PanAligner software may not be suitable for graphs which include substitution and indel variants.

Future work will be directed in the following directions. First, we will test the performance of PanAligner on pangenome graphs that are constructed by using alternative methods, e.g., [16, 19, 26]. Second, we will explore formulations for haplotype-constrained co-linear

chaining to control the exponential growth of combinatorial search space with the increasing number of haplotypes [34, 48]. Third, we will generalize the proposed techniques for aligning reads to long-read genome assembly graphs which also contain cycles. It will be interesting to understand whether the small width assumption is appropriate for assembly graphs.

References

1 Mohamed Abouelhoda and Enno Ohlebusch. Chaining algorithms for multiple genome comparison. *Journal of Discrete Algorithms*, 3(2-4):321–341, 2005.

2 Jasmijn A Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangenomics: a tutorial on data structures and their applications. *Natural Computing*, pages 1–28, 2022.

3 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 51–58, 2015.

4 Manuel Cáceres, Massimo Cairo, Brendan Mumey, Romeo Rizzi, and Alexandru I Tomescu. Sparsifying, shrinking and splicing for minimum path cover in parameterized linear time. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 359–376. SIAM, 2022.

5 Ghanshyam Chandra and Chirag Jain. Sequence to graph alignment using gap-sensitive co-linear chaining. In *Research in Computational Molecular Biology: 27th Annual International Conference, RECOMB 2023, Istanbul, Turkey, April 16–19, 2023, Proceedings*, pages 58–73. Springer, 2023.

6 Haoyu Cheng, Mobin Asri, Julian Lucas, Sergey Koren, and Heng Li. Scalable telomere-to-telomere assembly for diploid and polyploid genomes with double graph. *arXiv preprint*, 2023. `arXiv:2306.03399`.

7 Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in bioinformatics*, 19(1):118–135, 2018.

8 Egor Dolzhenko, Viraj Deshpande, Felix Schlesinger, Peter Krusche, Roman Petrovski, Sai Chen, Dorothea Emig-Agius, Andrew Gross, Giuseppe Narzisi, Brett Bowman, et al. Expansionhunter: a sequence-graph-based tool to analyze variation in short tandem repeat regions. *Bioinformatics*, 35(22):4754–4756, 2019.

9 Tatiana Dvorkina, Dmitry Antipov, Anton Korobeynikov, and Sergey Nurk. Spaligner: alignment of long diverged molecular sequences to assembly graphs. *BMC bioinformatics*, 21(12):1–14, 2020.

10 Peter Eades, Xuemin Lin, and W.F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, October 1993.

11 Hannes P Eggertsson, Hakon Jonsson, Snaedis Kristmundsdottir, et al. Graphtyper enables population-scale genotyping using pangenome graphs. *Nature genetics*, 49(11):1654–1660, 2017.

12 David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F Italiano. Sparse dynamic programming i: linear cost functions. *Journal of the ACM*, 39(3):519–545, 1992.

13 David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F Italiano. Sparse dynamic programming ii: convex and concave cost functions. *Journal of the ACM*, 39(3):546–567, 1992.

14 Yang Gao, Xiaofei Yang, Hao Chen, Xinjiang Tan, Zhaoqing Yang, Lian Deng, Baonan Wang, Shuang Kong, Songyang Li, Yuhang Cui, et al. A pangenome reference of 36 chinese populations. *Nature*, pages 1–10, 2023.

15 Shilpa Garg, Mikko Rautiainen, Adam M Novak, et al. A graph-based approach to diploid genome assembly. *Bioinformatics*, 34(13):i105–i114, 2018.

16 Erik Garrison, Andrea Guarracino, Simon Heumos, Flavia Villani, Zhigui Bao, Lorenzo Tattini, Jörg Hagmann, Sebastian Vorbrugg, Santiago Marco-Sola, Christian Kubica, et al. Building pangenome graphs. *bioRxiv*, pages 2023–04, 2023.

17 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, October 2018. doi:10.1038/nbt.4227.

18 Daniel Gibney, Sharma V Thankachan, and Srinivas Aluru. The complexity of approximate pattern matching on de Bruijn graphs. In *Research in Computational Molecular Biology: 26th Annual International Conference, RECOMB 2022, San Diego, CA, USA, May 22–25, 2022, Proceedings*, pages 263–278. Springer, 2022.

19 Glenn Hickey, Jean Monlong, Jana Ebler, Adam M Novak, Jordan M Eizenga, Yan Gao, Tobias Marschall, Heng Li, and Benedict Paten. Pangenome graph construction from genome alignments with minigraph-cactus. *Nature Biotechnology*, pages 1–11, 2023.

20 Chirag Jain, Daniel Gibney, and Sharma V Thankachan. Algorithms for colinear chaining with overlaps and gap costs. *Journal of Computational Biology*, 29(11):1237–1251, 2022.

21 Chirag Jain, Arang Rhie, Nancy F Hansen, Sergey Koren, and Adam M Phillippy. Long-read mapping to repetitive reference sequences using winnowmap2. *Nature Methods*, pages 1–6, 2022.

22 Chirag Jain, Haowen Zhang, Yu Gao, and Srinivas Aluru. On the complexity of sequence-to-graph alignment. *Journal of Computational Biology*, 27(4):640–654, April 2020. doi:10.1089/cmb.2019.0066.

23 Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, May 2018. doi:10.1093/bioinformatics/bty191.

24 Heng Li, Xiaowen Feng, and Chong Chu. The design and construction of reference pangenome graphs with minigraph. *Genome Biology*, 21(1), October 2020.

25 Heng Li, Jue Ruan, and Richard Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851–1858, 2008.

26 Wen-Wei Liao, Mobin Asri, Jana Ebler, Daniel Doerr, Marina Haukness, Glenn Hickey, Shuangjia Lu, Julian K Lucas, Jean Monlong, Haley J Abel, et al. A draft human pangenome reference. *Nature*, 617(7960):312–324, 2023.

27 Tsung-Yu Lu, Human Genome Structural Variation Consortium, et al. Profiling variable-number tandem repeat variation across populations using repeat-pangenome graphs. *Nature Communications*, 12(1):4250, 2021.

28 Xiao Luo, Xiongbin Kang, and Alexander Schönhuth. Vechat: correcting errors in long reads using variation graphs. *Nature Communications*, 13(1):6657, 2022.

29 Jun Ma. Co-linear chaining on graphs with cycles. Master's thesis, University of Helsinki, Faculty of Science, 2021. URL: http://hdl.handle.net/10138/330781.

30 Jun Ma, Manuel Cáceres, Leena Salmela, Veli Mäkinen, and Alexandru I Tomescu. Chaining for accurate alignment of erroneous long reads to acyclic variation graphs. *bioRxiv*, 2022. doi:10.1101/2022.01.07.475257.

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

32 Veli Mäkinen and Kristoffer Sahlin. Chaining with overlaps revisited. In *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

33 Veli Mäkinen, Alexandru I Tomescu, Anna Kuosmanen, Topi Paavilainen, Travis Gagie, and Rayan Chikhi. Sparse dynamic programming on DAGs with small width. *ACM Transactions on Algorithms (TALG)*, 15(2):1–21, 2019.

34 Tom Mokveld, Jasper Linthorst, Zaid Al-Ars, Henne Holstege, and Marcel Reinders. Chop: haplotype-aware path indexing in population graphs. *Genome biology*, 21:1–16, 2020.

35 Gene Myers and Webb Miller. Chaining multiple-alignment fragments in sub-quadratic time. In *SODA*, volume 95, pages 38–47, 1995.

36 Gonzalo Navarro. Improved approximate pattern matching on hypertext. *Theoretical Computer Science*, 237(1-2):455–463, 2000.

37 Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V Bzikadze, Alla Mikheenko, Mitchell R Vollger, Nicolas Altemose, Lev Uralsky, Ariel Gershman, et al. The complete sequence of a human genome. *Science*, 376(6588):44–53, 2022.

38 Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, et al. The complete sequence of a human genome. *Science*, 376(6588):44–53, April 2022. doi:10.1126/science.abj6987.

39 Yukiteru Ono, Kiyoshi Asai, and Michiaki Hamada. Pbsim2: a simulator for long-read sequencers with a novel generative model of quality scores. *Bioinformatics*, 37(5):589–595, 2021.

40 Christian Otto, Steve Hoffmann, Jan Gorodkin, and Peter F Stadler. Fast local fragment chaining using sum-of-pair gap costs. *Algorithms for Molecular Biology*, 6(1):4, 2011.

41 Benedict Paten, Adam M Novak, Jordan M Eizenga, and Erik Garrison. Genome graphs and the evolution of genome inference. *Genome research*, 27(5):665–676, 2017.

42 Mikko Rautiainen and Tobias Marschall. Graphaligner: rapid and versatile sequence-to-graph alignment. *Genome biology*, 21(1):253, 2020.

43 Mikko Rautiainen, Sergey Nurk, Brian P Walenz, Glennis A Logsdon, David Porubsky, Arang Rhie, Evan E Eichler, Adam M Phillippy, and Sergey Koren. Telomere-to-telomere assembly of diploid chromosomes with verkko. *Nature Biotechnology*, pages 1–9, 2023.

44 Nicola Rizzo, Manuel Cáceres, and Veli Mäkinen. Chaining of maximal exact matches in graphs. *arXiv preprint*, 2023. arXiv:2302.01748.

45 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.

46 Kristoffer Sahlin, Thomas Baudeau, Bastien Cazaux, and Camille Marchet. A survey of mapping algorithms in the long-reads era. *bioRxiv*, 2022.

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

48 Jouni Sirén, Erik Garrison, Adam M Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinformatics*, 36(2):400–407, 2020.

49 Jouni Sirén, Jean Monlong, Xian Chang, et al. Pangenomics enables genotyping of known structural variants in 5202 diverse genomes. *Science*, 374(6574):abg8871, 2021.

50 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

51 Ting Wang, Lucinda Antonacci-Fulton, Kerstin Howe, et al. The human pangenome project: a global resource to map genomic diversity. *Nature*, 604(7906):437–446, April 2022.

52 Haowen Zhang, Shiqi Wu, Srinivas Aluru, and Heng Li. Fast sequence to graph alignment using the graph wavefront algorithm. *arXiv preprint*, 2022. arXiv:2206.13574.

53 Yao Zhou, Zhiyang Zhang, Zhigui Bao, Hongbo Li, Yaqing Lyu, Yanjun Zan, Yaoyao Wu, Lin Cheng, Yuhan Fang, Kun Wu, et al. Graph pangenome captures missing heritability and empowers tomato breeding. *Nature*, 606(7914):527–534, 2022.

Acceleration of FM-Index Queries Through Prefix-Free Parsing

Aaron Hong[1]
Department of Computer and Information Science and Engineering,
Herbert Wertheim College of Engineering, University of Florida, Gainesville, FL, USA

Marco Oliva
Department of Computer and Information Science and Engineering,
Herbert Wertheim College of Engineering, University of Florida, Gainesville, FL, USA

Dominik Köppl
Institut für Informatik, Univeristät Münster, Germany

Hideo Bannai
M&D Data Science Center, Tokyo Medical and Dental University, Japan

Christina Boucher
Department of Computer and Information Science and Engineering,
Herbert Wertheim College of Engineering, University of Florida, Gainesville, FL, USA

Travis Gagie
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Abstract

FM-indexes are a crucial data structure in DNA alignment, but searching with them usually takes at least one random access per character in the query pattern. Ferragina and Fischer [6] observed in 2007 that word-based indexes often use fewer random accesses than character-based indexes, and thus support faster searches. Since DNA lacks natural word-boundaries, however, it is necessary to parse it somehow before applying word-based FM-indexing. Last year, Deng et al. [4] proposed parsing genomic data by induced suffix sorting, and showed the resulting word-based FM-indexes support faster counting queries than standard FM-indexes when patterns are a few thousand characters or longer. In this paper we show that using prefix-free parsing – which takes parameters that let us tune the average length of the phrases – instead of induced suffix sorting, gives a significant speedup for patterns of only a few hundred characters. We implement our method and demonstrate it is between 3 and 18 times faster than competing methods on queries to GRCh38. And was consistently faster on queries made to 25,000, 50,000 and 100,000 SARS-CoV-2 genomes. Hence, it is very clear that our method accelerates the performance of count over all state-of-the-art methods with a minor increase in the memory.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases FM-index, pangenomics, scalability, word-based indexing, random access

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.13

Supplementary Material *Software (Source Code of PFP-FM)*: `https://github.com/marco-oliva/afm`

Funding *Aaron Hong*: NIH/NHGRI grant R01HG011392 to Ben Langmead, NSF/BIO grant DBI-2029552 to Christina Boucher
Marco Oliva: NIH/NHGRI grant R01HG011392 to Ben Langmead, NSF/BIO grant DBI-2029552 to Christina Boucher
Dominik Köppl: JSPS KAKENHI Grant Number JP21K17701, JP22H03551, and JP23H04378
Hideo Bannai: JSPS KAKENHI Grant Number JP20H04141

[1] Corresponding author

Christina Boucher: NIH/NHGRI grant R01HG011392 to Ben Langmead, NSF/BIO grant DBI-2029552 to Christina Boucher, NSF/SCH grant INT-2013998 to Christina Boucher, and NIH/NIAID grant R01AI14180 to Christina Boucher
Travis Gagie: NIH/NHGRI grant R01HG011392 to Ben Langmead, NSERC grant RGPIN-07185-2020 to Travis Gagie, NSF/BIO grant DBI-2029552 to Christina Boucher

1 Introduction

The FM-index [5] is one of the most famous data structures in bioinformatics as it has been applied to countless applications in the analysis of biological data. Due to the long-term impact of this data structure, Burrows, Ferragina, and Manzini earned the 2022 ACM Paris Kanellakis Theory and Practice Award[2]. It is the data structure behind important read aligners – e.g., Bowtie [10] and BWA [11] – which take one or more reference genomes and build the FM-index for these genomes and use the resulting index to find short exact alignments between a set of reads and the reference(s) which then can be extended to approximate matches [10, 11]. Briefly, the FM-index consists of a sample of the suffix array (denoted as SA) and the Burrows–Wheeler transform (BWT) array. Given an input string S and a query pattern Q, `count` queries that answer the number of times the longest match of Q appears in S, can be efficiently supported using the BWT. To locate all of these occurrences the SA sample is needed. Hence, together the FM-index efficiently supports both `count` and `locate` queries. We mathematically define the SA and BWT in the next section.

There has been a plethora of research papers on reducing the size of the FM-index (see, e.g., [13, 9, 7]) and on speeding up queries. The basic query, `count`, returns the number of times a pattern Q appears in the indexed text S, but usually requires at least $|Q|$ random accesses to the BWT of S, which are usually much slower than the subsequent computations we perform on the information those accesses return. More specifically, a `count` query for Q use rank queries at $|Q|$ positions in the BWT; if we answer these using a single wavelet tree for the whole BWT, then we may use a random access for every level we descend in the wavelet tree, or $\Omega(|Q| \log \sigma)$ random access in all, where σ is the size of the alphabet; if we break the BWT into blocks and use a separate wavelet tree for each block [9], we may need only one or a few random accesses per rank query, but the total number of random accesses is still likely to be $\Omega(|Q|)$. As far back as 2007, Ferragina and Fischer [6] addressed compressed indexes' reliance on random access and demonstrated that word-based indexes perform fewer random accesses than character-based indexes: *"The space reduction of the final word-based suffix array impacts also in their query time (i.e. less random access binary-search steps!), being faster by a factor of up to 3."*

Thus, one possibility of accelerating the random access to genomic data – where it is widely used – is to break up the sequences into words or phrases. In light of this insight, Deng et al. [4] in 2022 applied a grammar [18] that factorizes S into phrases based on the leftmost S-type suffixes (LMS) [17]. Unfortunately, one round of that LMS parsing leads to phrases that are generally too short, so they obtained speedup only when Q was thousands of characters. The open problem was how to control the length of phrases with respect to the input to get longer phrases that would enable larger advances in the acceleration of the random access.

Here, we apply the concept of prefix-free parsing to the problem of accelerating `count` in the FM-index. Prefix-free parsing uses a rolling hash to first select a set of strings (referred to as *trigger strings*) that are used to define a parse of the input string S; i.e., the prefix-free

[2] `https://awards.acm.org/kanellakis`

parse is a parsing of S into phrases that begin and end at a trigger string and contain no other trigger string. All unique phrases are lexicographically sorted and stored in the dictionary of the prefix-free parse, which we denote as D. The prefix-free parse can be stored as an ordered list of the phrases' ranks in D. Hence, prefix-free parsing breaks up the input sequence into phrases, whose size is more controllable by the selection of the trigger strings. This leads to a more flexible acceleration than Deng et al. [4] obtained.

We assume that we have an input string S of length n. Now suppose we build an FM-index S, an FM-index for the parse P, and a bitvector B of length n with `1`'s marking characters in the BWT of S that immediately precede phrase boundaries in S, i.e., that immediately precede a trigger string. We note that all the 1s are bunched into at most as many runs as there are distinct trigger strings in S. Also, as long as the ranks of the phrases are in the same lexicographic order as the phrases themselves, we can use the bitvector to map from the interval in the BWT of S for any pattern starting with a trigger string to the corresponding interval in the BWT of P, and vice versa. This means that, given a query pattern Q, we can backward search for Q character by character in the FM-index for S until we hit the left end of the rightmost trigger string in Q, then map into the BWT of P and backward search for Q phrase by phrase until we hit the left end of the leftmost trigger string in Q, then map back into the BWT of S and finish backward searching character by characters again.

We implement this method, which we refer to as `PFP-FM`, and extensively compare against the FM-index implementation in `sdsl` [8], `RLCSA` [19], `RLFM` [13, 12], and `FIGISS` [4] using sets of SARS-CoV-2 genomes taken from the NCBI website, and the Genome Reference Consortium Human Build 38 with varying query string lengths. When we compare `PFP-FM` to FM-index in `sdsl` using 100,000 SARS-CoV-2 genomes, we witnessed that `PFP-FM` was able to perform between 2.1 and 2.8 more queries. In addition, `PFP-FM` was between 64.38% and 74.12%, 59.22% and 78.23%, and 49.10% and 90.70% faster than `FIGISS`, `RLCSA`, and `RLFM`, respectively on 100,000 SARS-CoV-2 genomes. We evaluated the performance of `PFP-FM` on the Genome Reference Consortium Human Build 38, and witnessed that it was between 3.86 and 7.07, 2.92 and 18.07, and 10.14 and 25.46 times faster than `RLCSA`, `RLFM`, and `FIGISS`, respectively. With respect to construction time, `PFP-FM` had the most efficient construction time for all SARS-CoV-2 datasets and was the second fastest for Genome Reference Consortium Human Build 38. All methods used less than 60 GB for memory for construction on the SARS-CoV-2 datasets, making the construction feasible on any entry level commodity server – even the build for the 100,000 SARS-CoV-2 dataset. Construction for the Genome Reference Consortium Human Build 38 required between 26 GB and 71 GB for all methods, with our method using the most memory. In summary, we develop and implement a method for accelerating the FM-index, and achieve an acceleration between 2 and 25 times, with the greatest acceleration witnessed with longer patterns. Thus, accelerated FM-index methods – such as the one developed in this paper – are highly applicable to finding very long matches (125 to 1,000 in length) between query sequences and reference databases. As reads get longer and more accurate (i.e., Nanopore data), we will soon be prepared align long reads to reference databases with efficiency that surpasses traditional FM-index based alignment methods. The source code is publicly available at `https://github.com/marco-oliva/afm`.

2 Preliminaries

2.1 Basic Definitions

A string S of length n is a finite sequence of symbols $S = S[0..n-1] = S[0] \cdots S[n-1]$ over an alphabet $\Sigma = \{c_1, \ldots, c_\sigma\}$. We assume that the symbols can be unambiguously ordered. We denote by ε the empty string, and the length of S as $|S|$. Given a string S, we denote the reverse of S as $rev(S)$, i.e., $rev(S) = S[n-1] \cdots S[0]$.

We denote by $S[i..j]$ the substring $S[i] \cdots S[j]$ of S starting in position i and ending in position j, with $S[i..j] = \varepsilon$ if $i > j$. For a string S and $0 \leq i < n$, $S[0..i]$ is called the i-th prefix of S, and $S[i..n-1]$ is called the i-th suffix of S. We call a prefix $S[0..i]$ of S a *proper prefix* if $0 \leq i < n-1$. Similarly, we call a suffix $S[i..n-1]$ of S a *proper suffix* if $0 < i < n$.

Given a string S, a symbol $c \in \Sigma$, and an integer i, we define $S.\mathsf{rank}_c(i)$ (or simply `rank` if the context is clear) as the number of occurrences of c in $S[0..i-1]$. We also define $S.\mathsf{select}_c(i)$ as $\min(\{j-1 \mid S.\mathsf{rank}_c(j) = i\} \cup \{n\})$, i.e., the position in S of the i-th occurrence of c in S if it exists, and n otherwise. For a bitvector $B[0..n-1]$, that is a string over $\Sigma = \{0,1\}$, to ease the notation we will refer to $B.\mathsf{rank}_1(i)$ and $B.\mathsf{select}_1(i)$ as $B.\mathsf{rank}(i)$ and $B.\mathsf{select}(i)$, respectively.

2.2 SA, BWT, and Backward Search

We denote the *suffix array* [14] of a given a string $S[0..n-1]$ as SA_S, and define it to be the permutation of $\{0,\ldots,n-1\}$ such that $S[\mathsf{SA}_S[i]..n-1]$ is the i-th lexicographical smallest suffix of S. We refer to SA_S as SA when it is clear from the context. For technical reasons, we assume that the last symbol of the input string is $S[n-1] = \$$, which does not occur anywhere else in the string and is smaller than any other symbol.

We consider the matrix W containing all sorted rotations of S, called the BWT matrix of S, and let F and L be the first and the last column of the matrix. The last column defines the BWT array, i.e., $\mathsf{BWT} = \mathsf{L}$. Now let $\mathsf{C}[c]$ be the number of suffixes starting with a character smaller than c. We define the LF-mapping as $\mathsf{LF}(i,c) = \mathsf{C}[c] + \mathsf{BWT}.\mathsf{rank}_c(i)$ and $\mathsf{LF}(i) = \mathsf{LF}(i, \mathsf{BWT}[i])$. With the LF-mapping, it is possible to reconstruct the string S from its BWT. It is in fact sufficient to set an iterator $s = 0$ and $S[n-1] = \$$ and for each $i = n-2,\ldots,0$ do $S[i] = \mathsf{BWT}[s]$ and $s = \mathsf{LF}(s)$. The LF-mapping can also be used to support `count` by performing the backward search, which we now describe.

Given a query pattern Q of length m, the *backward search* algorithm consists of m steps that preserve the following invariant: at the i-th step, p stores the position of the first row of W prefixed by $Q[i,m]$ while q stores the position of the last row of W prefixed by $Q[i,m]$. To advance from i to $i-1$, we use the LF-mapping on p and q, $p = \mathsf{C}[c] + \mathsf{BWT}.\mathsf{rank}_c(p)$ and $q = \mathsf{C}[c] + \mathsf{BWT}.\mathsf{rank}_c(q+1) - 1$.

2.3 FM-index and count Queries

Given a query string $Q[0..m-1]$ and an input string $S[0..n-1]$, two fundamental queries are: (1) `count` which counts the number of of occurrences of Q in S; (2) `locate` which finds the location of each of these matches in S. Ferragina and Manzini [5] showed that, by combining SA with the BWT, both `count` and `locate` can be efficiently supported. Briefly, backward search on the BWT is used to find the lexicographical range of the occurrences of Q in S; the size of this range is equal to `count`. The SA positions within this range are the positions where these occurrences are in S.

2.4 Prefix-Free Parsing

As we previously mentioned, the *Prefix-Free Parsing* (PFP) takes as input a string $S[0..n-1]$, and positive integers w and p, and produces a parse of S (denoted as P) and a dictionary (denoted as D) of all the unique substrings (or phrases) of the parse. We note that w defines the length of the trigger strings and p is used in the rolling-hash. We briefly go over the algorithm for producing this dictionary and parse. First, we assume there exists two symbols,

say # and \$, which are not contained in Σ and are lexicographically smaller than any symbol in Σ. Next, we let T be an arbitrary set of w-length strings over Σ and call it the set of *trigger strings*. As mentioned before, we assume that $S[n-1] = \$$ and consider S to be cyclic, i.e., for all i, $S[i] = S[i \bmod n]$. Furthermore, we assume that $\$S[0..w-2] = S[n-1..n+w-2] \in T$, i.e., the substring of length w that begins with \$ is a trigger string.

We let the dictionary $\mathsf{D} = \{d_1, .., d_{|\mathsf{D}|}\}$ be a (lexicographically sorted) maximum set of substrings of S such that the following holds for each d_i: i) exactly one proper prefix of d_i is contained in T, ii) exactly one proper suffix of d_i is contained in T, iii) and no other substring of d_i is in T. These properties allow for the SA and BWT to be constructed since the lexicographical placement of each rotation of the input string can be indentified unambiguously from D and P [2, 3, 16]. An important consequence of the definition is that D is prefix-free, i.e., for any $i \neq j$, d_i cannot be a prefix of d_j.

Since we assumed $S[n-1..n+w-2] \in T$, we can construct D by scanning $S' = \$S[0..n-2]S[n-1..n+w-2]$ to find all occurrences of T and adding to D each substring of S' that starts and ends at a trigger string being inclusive of the starting and ending trigger string. We can also construct the list of occurrences of D in S', which defines the parse P.

We choose T by a Karp-Rabin fingerprint f of strings of length w. We slide a window of length w over S', and for each length w substring r of S', include r in T if and only if $f(r) \equiv 0 \pmod p$ or $r = S[n-1..n+w-2]$. Let $0 = s_0 < \cdots < s_{k-1}$ be the positions in S' such that for any $0 \leq i < k$, $S'[s_i..s_i+w-1] \in T$. The dictionary is $\mathsf{D} = \{S'[s_i..s_{i+1}+w-1] \mid i = 0, \ldots, k-1\}$, and the parse is defined to be the sequence of lexicographic ranks in D of the substrings $S'[s_0..s_1+w-1], \ldots, S'[s_{k-2}..s_{k-1}+w-1]$.

As an example, suppose we have $S' =$ \$AGACGACT#AGATACT#AGATTCGAGACGAC\$A, where the trigger strings are highlighted in red, blue, or green. It follows that we have $\mathsf{D} =$ {\$AGAC, AC\$A, ACGAC, ACT#AGATAC, ACT#AGATTC, TCGAGAC} and $\mathsf{P} = 0, 2, 3, 4, 5, 2, 1$.

3 Methods

As we previously mentioned, we will use prefix-free parsing to build a word-based FM-index in a manner in which the length of the phrases can be controlled via the parameters w and p. To explain our data structure, we first describe the various components of our data structure, and then follow with describing how to support `count` queries in a manner that is more efficient than the standard FM-index.

3.1 Data Structure Design

It is easiest to explain our two-level design with an example, so consider a text

$$S[0..n-1] = \texttt{TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT\$}$$

of length $n = 41$ that is terminated by a special end-of-string character `$` lexicographically less than the rest of the alphabet. Suppose we parse S using $w = 2$ and a Karp-Rabin hash function such that the normal trigger strings occurring in S are `AA`, `CG` and `TA`. We consider S as cyclic, and we have $\$S[0..w-2] =$ \$T as a special trigger string, so the the dictionary D is

$$\mathsf{D}[0..5] = \{\texttt{\$TCCAGAA}, \texttt{AAGACATA}, \texttt{AAGAGTA}, \texttt{CGACATGTTGAA}, \texttt{TATCTCCTCG}, \texttt{TATGAT\$T}\},$$

with the phrases sorted in lexicographic order. (Recall that phrases consecutive in S overlap by $w = 2$ characters.) If we start parsing at the `$`, then the prefix-free parse for S is

$$\mathsf{P}[0..5] = (0, 2, 4, 3, 1, 5),$$

where each element (or phrase ID) in P is the lexicographic rank of the phrase in D.

0	2	4	3	1	5	$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT
1	5	0	2	4	3	AAGACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTG
2	4	3	1	5	0	AAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAG
3	1	5	0	2	4	CGACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCT
4	3	1	5	0	2	TATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAG
5	0	2	4	3	1	TATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACA

Figure 1 The BWT matrix for our prefix-free parse P **(left)** and the cyclic shifts of S that start with a trigger string **(right)**, in lexicographic order.

Next, we consider the BWT matrix for P. Figure 1 illustrates the BWT matrix of P for our example. We note that since there is only one $ in S, it follows that there is only one 0 in P; we can regard this 0 as the end-of-string character for (a suitable rotation of) P corresponding to $ in S. If we take the i-th row of this matrix and replace the phrase IDs by the phrases themselves, collapsing overlaps, then we get the lexicographically i-th cyclic shift of S that start with a trigger string, as shown on the right of the figure. This is one of the key insights that we will use later on.

▶ **Lemma 1.** *The lexicographic order of rotations of* P *correspond to the lexicographic order of their corresponding rotations of* S*.*

Proof. The characters of P are the phrase IDs that act as meta-characters. Since the meta-characters inherit the lexicographic rank of their underlying characters, and due to the prefix-freeness of the phrases, the suffix array of P permutes the meta-characters of P in the same way as the suffix array of S permutes the phrases of S. This means that the order of the phrases in the BWT of S is the same as the order of the phrase IDs in P. ◀

Next, we let $B[0..n-1]$ be a bitvector marking these cyclic shifts' lexicographic rank among all cyclic shifts of S, i.e., where they are among the rows of the BWT matrix of S. Figure 2 shows the SA, BWT matrix and BWT of S, together with B; we highlight the BWT – the last column of the matrix – and the cyclic shifts from Figure 1 in red. We note that B contains at most one run of 1's for each distinct trigger string in S so it is usually highly run-length compressible in practice.

In addition to the bitvector, we store a hash function h on phrases and a map M from the hashes of the phrases in D to those phrases' lexicographic ranks, which are their phrase IDs; M returns NULL when given any other key. Therefore, in total, we build the FM-index for S, the FM-index for P, the bitvector B marking the cyclic rotations, the hash function h on the phrases and the map M. For our example, suppose

$$\begin{aligned} h(\texttt{\$TCCAGAA}) &= 91785 & M(91785) &= 0 \\ h(\texttt{AAGACATA}) &= 34865 & M(34865) &= 1 \\ h(\texttt{AAGAGTA}) &= 49428 & M(49428) &= 2 \\ h(\texttt{CGACATGTTGAA}) &= 98759 & M(98759) &= 3 \\ h(\texttt{TATCTCCTCG}) &= 37298 & M(37298) &= 4 \\ h(\texttt{TATGAT\$T}) &= 68764 & M(68764) &= 5 \end{aligned}$$

and $M(x) = \text{NULL}$ for any other value of x.

If we choose the range of h to be reasonably large then we can still store M in space proportional to the number of phrases in D with a reasonably constant coefficient and evaluate $M(h(\cdot))$ in constant time with high probability, but the probability is negligible that $M(h(\gamma)) \neq \text{NULL}$ for any particular string γ not in D. This means that in practice we can use $M(h(\cdot))$ as a membership dictionary for D, and not store D itself.

i	SA$[i]$	$B[i]$	T[SA$[i]$..(SA$[i]-1$) mod n]	BWT$[i]$
0	40	1	$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT	T
1	28	1	AAGACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTG	G
2	5	1	AAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAG	G
3	31	0	ACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAG	G
4	20	0	ACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCG	G
5	3	0	AGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCC	C
6	29	0	AGACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGA	A
7	6	0	AGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAGA	A
8	8	0	AGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAG	G
9	38	0	AT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATG	G
10	33	0	ATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGAC	C
11	11	0	ATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGT	T
12	35	0	ATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACAT	T
13	22	0	ATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGAC	C
14	2	0	CAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TC	C
15	32	0	CATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGA	A
16	21	0	CATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGA	A
17	1	0	CCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$T	T
18	15	0	CCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTATCT	T
19	18	1	CGACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCT	T
20	13	0	CTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTAT	T
21	16	0	CTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTC	C
22	27	0	GAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTT	T
23	4	0	GAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCA	A
24	30	0	GACATATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAA	A
25	19	0	GACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTC	C
26	7	0	GAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAA	A
27	37	0	GAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATAT	T
28	9	0	GTATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGA	A
29	24	0	GTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACAT	T
30	39	0	T$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGA	A
31	10	1	TATCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAG	G
32	34	1	TATGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACA	A
33	0	0	TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT$	$
34	14	0	TCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTATC	C
35	17	0	TCGACATGTTGAAGACATATGAT$TCCAGAAGAGTATCTCC	C
36	12	0	TCTCCTCGACATGTTGAAGACATATGAT$TCCAGAAGAGTA	A
37	26	0	TGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACATGT	T
38	36	0	TGAT$TCCAGAAGAGTATCTCCTCGACATGTTGAAGACATA	A
39	23	0	TGTTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACA	A
40	25	0	TTGAAGACATATGAT$TCCAGAAGAGTATCTCCTCGACATG	G

Figure 2 The SA, BWT matrix and BWT of T, together with the bitvector B in which 1s indicate rows of the matrix starting with trigger strings. The BWT is highlighted in red, as are the columns marked by 1s.

3.2 Query Support

Next, given the data structure that we define above, we describe how to support `count` queries for a given pattern Q. We begin by parsing Q using the same Karp-Rabin hash we used to parse S, implying that we will have all the same trigger strings as we did before and possibly additional ones that did not occur in S. However, we will not consider Q to be cyclic nor assume an end-of-string symbol that would assure that Q starts and ends with a trigger string.

If Q is a substring of S, then, since Q contains the same trigger strings as its corresponding occurrence in S, the sequence of phrases induced by the trigger strings in Q must be a substring of the sequence of phrases of S. Together with the prefix and suffix of Q that are a suffix and prefix of the phrases in S to the left and right of the shared phrases, we call this the partial encoding of Q, defined formally as follows.

▶ **Definition 2** (partial encoding). *Given a substring $S[i..j]$ of S, the* partial encoding *of $S[i..j]$ is defined as follows: If no trigger string occurs in $S[i..j]$, then the partial encoding of $S[i..j]$ is simply $S[i..j]$ itself. Otherwise, the partial encoding of $S[i..j]$ is the concatenation of: (1) the shortest prefix α of $S[i..j]$ that does not start with a trigger string and ends with a trigger string, followed by (2) the sequence of phrase IDs of phrases completely contained in $S[i..j]$, followed by (3) the shortest suffix β of $S[i..j]$ that begins with a trigger string and does not end with a trigger string.*

So the partial encoding partitions $S[i..j]$ into a prefix α, a list of phrase IDs, and a suffix β. If $S[i..j]$ begins (respectively ends) with a trigger string, then α (respectively β) is the empty string.

Parsing Q can be done in time linear in the length of Q.

▶ **Lemma 3.** *We can represent M with a data structure taking space (in words) proportional to the number of distinct phrases in D. Given a query pattern Q, this data structure returns NULL with high probability if Q contains a complete phrase that does not occur in S. Otherwise (complete phrases of Q occur in S), it returns the partial encoding of Q. In either case, this query takes $O(|Q|)$ time.*

Proof. We keep the Karp-Rabin (KR) hashes of the phrases in D, with the range of the KR hash function mapping to $[1..n^3]$ so the hashes each fit in $O(\log n)$ bits. We also keep a constant-time map (implemented as a hash table with a hash function that's perfect for the phrases in D) from the KR hashes of the phrases in D to their IDs, that returns NULL given any value that is not a KR hash of a phrase in D. We set M to be the map composed with the KR hash function.

Given Q, we scan it to find the trigger strings in it, and convert it into a sequence of substrings consisting of: (a) the prefix α of Q ending at the right end of the first trigger string in Q; (b) a sequence of PFP phrases, each starting and ending with a trigger string with no trigger string in between; and (c) the suffix β of Q starting at the left end of the last trigger string in Q.

We apply M to every complete phrase in (b). If M returns NULL for any complete phrase in (b), then that phrase does not occur in S, so we return NULL; otherwise, we return α, the sequence of phrase IDs M returned for the phrases in (b), and β.

Notice that, if a phrase in Q is in S, then M will map it to its lexicographic rank in D; otherwise, the probability the KR hash of any particular phrase in Q but not in D collides with the KR hash of a phrase in D, is at most $n/n^3 = 1/n^2$. It follows that, if Q contains a complete phrase that does not occur in S, then we return NULL with high probability; otherwise, we return Q's partial encoding. ◀

▶ **Corollary 4.** *If we allow $O(|Q|)$ query time with high probability, then we can modify M to always report NULL when Q contains a complete phrase not in S.*

Proof. We augment each Karp-Rabin (KR) hash stored in the hash table with the actual characters of its phrase such that we can check, character by character, whether a matched phrase of Q is indeed in D. In case of a collision we recompute the KR hashes of D and rebuild the hash table. That is possible since we are free to choose different Karp-Rabin fingerprints for the phrases in D. ◀

Continuing from our example above where the trigger strings are `AA`, `CG` and `TA`, suppose we have a query pattern Q,

$Q[0..34] =$ `CAGAAGAGTATCTCCTCGACATGTTGAAGACATAT`

we can compute the parse of Q to obtain the following

`CAGAA`, `AAGAGTA`, `TATCTCCTCG`, `CGACATGTTGAA`, `AAGACATA`, `TAT`.

Next, we use $M(h(\cdot))$ to map the complete phrases of this parse of Q to their phrase IDs – which is their rank in D. If any complete phrase maps to NULL then we know Q does not occur in T. Using our example, we have the partial encoding

`CAGAA`, 2, 4, 3, 1, `TAT`.

Next, we consider all possible cases. First, we consider the case that the last substring β in our parse of Q ends with a trigger string, which implies that it is a complete phrase. Here, we can immediately start backward searching for the parse of Q in the FM-index for P. Next, if β is not a complete phrase then we backward search for β in the FM-index for S. If this backward search for β returns nothing then we know Q does not occur in S. If the backward search for β returns an interval in the BWT of P that is not contained in the BWT interval for a trigger string then β does not start with a trigger string so $Q = \beta$ and we are done backward searching for Q.

Finally, we consider the case when β is a proper prefix of a phrase and the backward search for β returns a BWT_S interval contained in the BWT_S interval for a trigger string. In our example, $\beta =$ `TAT` and our backward search for β in the FM-index for S returns the interval $\mathsf{BWT}_S[31..32]$, which is the interval for the trigger string `TA`. Next, we use B to map the interval for β in the BWT_S to the interval in the BWT_P that corresponds to the cyclic shifts of S starting with β.

▶ **Lemma 5.** *We can store in space (in words) proportional to the number of distinct trigger strings in S a data structure B with which,*

- *given the lexicographic range of suffixes of S starting with a string β such that β starts with a trigger string and contains no other trigger string, in $O(\log\log n)$ time we can find the lexicographic range of suffixes of P starting with phrases that start with β;*
- *given a lexicographic range of suffixes of P such that the corresponding suffixes of S all start with the same trigger string, in $O(\log\log n)$ time we can find the lexicographic range of those corresponding suffixes of S.*

Proof. Let $B[0..n-1]$ be a bitvector with 1s marking the lexicographic ranks of suffixes of S starting with trigger strings. There are at most as many runs of 1s in B as there are distinct trigger strings in S, so we can store it in space proportional to that number and support rank and select operations on it in $O(\log\log n)$ time.

If $\mathsf{BWT}_S[i..j]$ contains the characters immediately preceding, in S, occurrences of a string β that starts with a trigger string and contains no other trigger strings, then $\mathsf{BWT}_\mathsf{P}[B.\mathsf{rank}_1(i)..B.\mathsf{rank}_1(j)]$ contains the phrase IDs immediately preceding, in P, the IDs of phrases starting with β.

If $\mathsf{BWT}_\mathsf{P}[i..j]$ contains the phrase IDs immediately preceding, in P, suffixes of P such that the corresponding suffixes of S all start with the same trigger string, then $\mathsf{BWT}_S[B.\mathsf{select}_1(i+1)..B.\mathsf{select}_1(j+1)]$ contains the characters immediately preceding the corresponding suffixes of S.

The correctness follows from Lemma 1. ◀

Continuing with our example mapping $\mathsf{BWT}_S[31..32]$ yield the following interval:

$$\mathsf{BWT}_\mathsf{P}[B.\mathsf{rank}_1(31), B.\mathsf{rank}_1(32)] = \mathsf{BWT}_\mathsf{P}[4..5]$$

as shown in Figure 1. Starting from this interval in BWT_P, we now backward search in the FM-index for P for the sequence of complete phrase IDs in the parse of Q. In our example, we have the interval $\mathsf{BWT}_\mathsf{P}[4..5]$ which yields the following phrase IDs: 2 4 3 1.

If this backward search in the FM-index for P returns nothing, then we know Q does not occur in S. Otherwise, it returns the interval in BWT_P corresponding to cyclic shifts of S starting with the suffix of Q that starts with Q's first complete phrase. In our example, if we start with $\mathsf{BWT}_\mathsf{P}[4..5]$ and backward search for 2 4 3 1 then we obtain $\mathsf{BWT}_\mathsf{P}[2]$, which corresponds to the cyclic shift

`AAGAGTATCTCCTCGACATGTTGAAGACATATGAT$TCCAG`

of S that starts with the suffix

`AAGAGTATCTCCTCGACATGTTGAAGACATAT`

of Q that is parsed into $2, 4, 3, 1,$ `TAT`.

To finish our search for Q, we use B to map the interval in BWT_P to the corresponding interval in the BWT_S, which is the interval of rows in the BWT matrix for S which start with the suffix of Q we have sought so far. In our example, we have that $\mathsf{BWT}_\mathsf{P}[2]$ maps to

$$\mathsf{BWT}_S[B.\mathsf{select}_1(2+1)] = \mathsf{BWT}_S[2].$$

We note that our examples contain BWT intervals with only one entry because our example is so small, but in general they are longer. If the first substring α in our parse of Q is a complete phrase then we are done backward searching for Q. Otherwise, we start with this interval in BWT_S and backward search for α in the FM-index for S, except that we ignore the last w last characters of α (which we have already sought, as they are also contained in the next phrase in the parse of Q).

In our example, $\alpha =$ `CAGAA` so, starting with $\mathsf{BWT}_S[2]$ we backward search for `CAG`, which returns the interval $\mathsf{BWT}_S[14]$. As shown in Figure 2,

$$S[\mathsf{SA}[4]..n] = S[2..n] = \texttt{CAGAAGAGTATCTCCTCGACATGTTGAAGACATATGAT\$}$$

does indeed start with

$$Q = \texttt{CAGAAGAGTATCTCCTCGACATGTTGAAGACATAT}\,.$$

This concludes our explanation of `count`.

To conclude, we give some intuition as to why we expect our two-level FM-index to be faster in practice than standard backward search. First, we note that standard backward search takes linear time in the length of Q and usually uses at least one random access per character in Q. Whereas, prefix-free parsing Q takes linear time but does not use random access; backward search in the FM-index of S is the same as standard backward search but we use it only for the first and last substrings in the parse of Q. Backward search in the FM-index for P is likely to use about $\lg |\mathsf{D}|$ random access for each complete phrase in the parse of Q: the BWT of P is over an effective alphabet whose size is the number of phrases in D. Therefore, a balanced wavelet tree to support rank on that BWT should have depth about $\lg |\mathsf{D}|$ and we should use at most about one random access for each level in the tree.

In summary, if we can find settings of the prefix-free parsing parameters w and p such that

- most query patterns will span several phrases,
- most phrases in those patterns are fairly long,
- $\lg |\mathsf{D}|$ is significantly smaller than those phrases' average length,

then the extra cost of parsing Q should be more than offset by using fewer random accesses.

4 Results

We implemented our algorithm and measured its performance against all known competing methods. We ran all experiments on a server with AMD EPYC 75F3 CPU with the Red Hat Enterprise Linux 7.7 (64bit, kernel 3.10.0). The compiler was g++ version 12.2.0. The running time and memory usage was recorded by SnakeMake benchmark facility [15]. We set a memory limitation of 128 GB of memory and a time limitation of 24 hours.

Datasets. We used the following datasets. First, we considered sets of SARS-CoV-2 genomes taken from the NCBI website. We used three collections of $25,000$, $50,000$, and $100,000$ SARS-CoV-2 genomes from EMBL-EBI's COVID-19 data portal. Each collection is a superset of the previous. We denote these as `SARS-25k`, and `SARS-50k`, `SARS-100k`. Next, we considered a single human reference genome, which we denote as `GRCh38`, downloaded from NCBI. We report the size of the datasets as the number of characters in each in Table 1. We denote n as the number of characters.

Implementation. We implemented our method in C++ 11 using the `sdsl-lite` library [8] and extended the prefix-free parsing method of Oliva, whose source code is publicly available here `https://github.com/marco-oliva/pfp`. The source code for `PFP-FM` is available at `https://github.com/marco-oliva/afm`.

Competing methods. We compared `PFP-FM` against the following methods the standard FM-index found in `sdsl-lite` library [8], `RLCSA` [19], `RLFM` [13, 12], and `FIGISS` [4]. We note that `RLCSA` and `FIGISS` have publicly-available source codes, while `RLFM` is provided only as an executable. We performed the comparison by selecting 1,000 strings from the genome file at random of the specified length, performing the `count` operation on each query pattern, and measuring the time usage for all the methods under consideration. It is worth noting that `FIGISS` and `RLCSA` only support `count` queries where the string is provided in an input text file. More specifically, the original `FIGISS` implementation supports counting with the entire content of a file treated as a single pattern. To overcome this limitation, we modified the source code to enable the processing of multiple query patterns within a single file. In addition to the time consideration for `count`, we measured the time and memory required to construct the data structure.

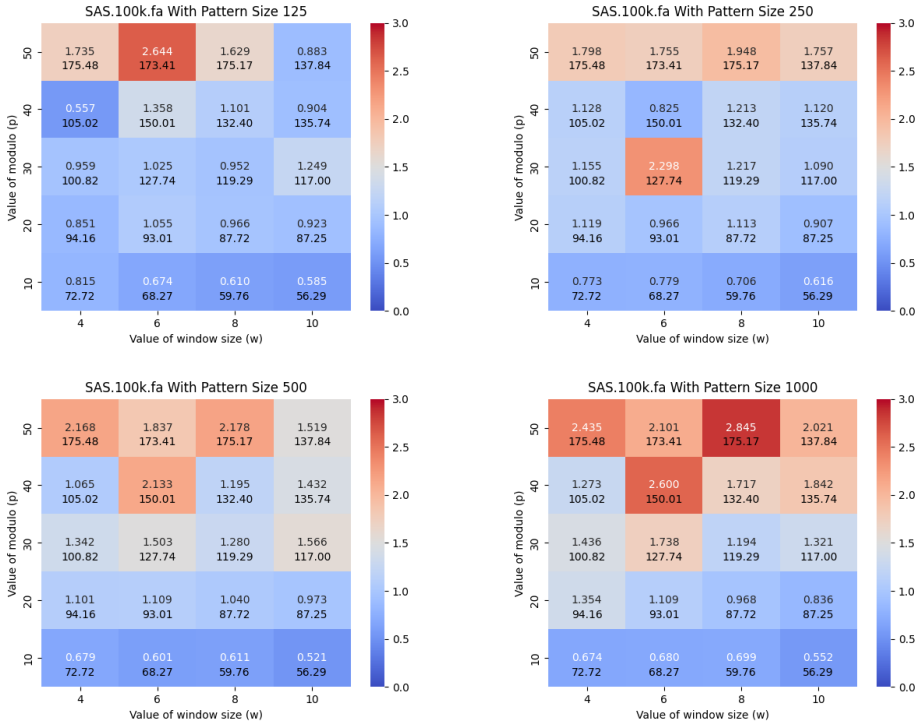


Figure 3 Illustration of the impact of w, p and the length of the query pattern on the acceleration of the FM-index. Here, we used `SARS-100K` dataset and varied the length of the query pattern to be equal to 125, 250, 500, and 1000. The y-axis corresponds to p and the x-axis corresponds to w. The heatmap illustrates the number of queries that can be performed in a CPU second with the acceleration versus the standard FM-index from `sdsl`, i.e., `PFP-FM` / `sdsl`. The second value in each block represents the average length of phrases.

4.1 Acceleration versus Baseline

In this subsection, we compare `PFP-FM` versus the standard FM-index in `sdsl` with varying values of window size (w) and modulo value (p), and varying the length of the query pattern. We calculated the number of `count` queries that were able to be performed in CPU second with `PFP-FM` versus the standard FM-index. We generated heatmaps that illustrate the number of `count` queries of `PFP-FM` verses `sdsl` for various lengths of query patterns, namely, 125, 250, 500, and 1,000. We performed this for each SARS-CoV-2 set of genomes. Figure 3 shows the resulting heatmaps for `SARS-100K`. As shown in this figure, `PFP-FM` was between 2.178 and 2.845 times faster than the standard FM-index with the optimal values of w and p. In particular, an optimal performance gain of 2.6, 2.3, 2.2, and 2.9 was witnessed for pattern lengths of 125, 250, 500, and 1,000, respectively. The (w, p) pairs that correspond to these results are $(6, 50)$, $(6, 30)$, $(8, 50)$, and $(8, 50)$.

4.2 Results on SARS-CoV-2 Genomes

We used the optimal parameters that were obtained from the previous experiment for this section. We constructed the index using these parameters for each SARS-CoV-2 dataset and assessed the time consumption for performing 1,000 `count` queries using all competing methods and `PFP-FM`. We illustrate the result of this experiment in Figure 4. It is clear

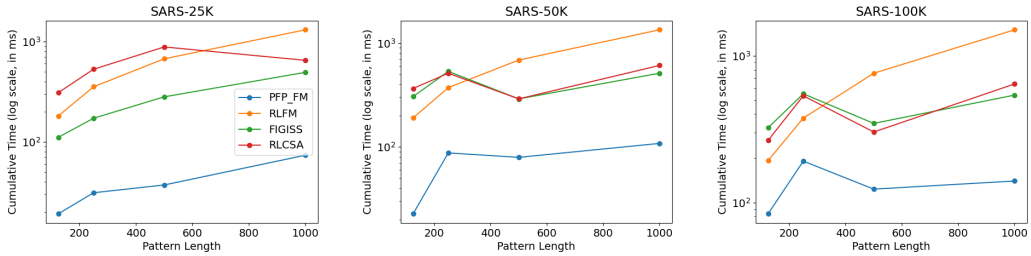


Figure 4 Illustration of the impact of the dataset size, and the length of the query pattern on the query time for answering `count`. We vary the length of the query pattern to be equal to 125, 250, 500, and 1000, and report the times for `SARS-25K`, `SARS-50K`, and `SARS-100K`. We illustrate the cumulative time required to perform 1,000 `count` queries. The y-axis is in log scale.

from this `PFP-FM` consistently exhibits the lowest time consumption and a gradual, stable trend. For the `SARS-25K` dataset, the time consumption of `FIGISS` was between 451% and 568% higher than our method. And the time consumption of `RLCSA` and `RLFM` was between 780% and 1598%, and 842% and 1705% more than `PFP-FM`, respectively. The performance of `FIGISS` surpasses that of `RLFM` and `RLCSA` when using the SARS-25k dataset; however for the larger datasets `FIGISS` and `RLCSA` converge in their performance .Neither method was substantially better than the other. In addition, on the larger datasets, when the query pattern length was 125 and 250, `RLFM` performed better than `RLCSA` and `FIGISS` but was slower for the other query lengths. Hence, it is very clear that `PFP-FM` accelerates the performance of `count` over all state-of-the-art methods.

The gap in performance between `PFP-FM` and the competing methods increased with the dataset size. For `SARS-50K`, `FIGISS`, `RLCSA` and `RLFM` were between 3.65 and 13.44, 3.65 and 16.08, and 4.25 and 12.39 times slower, respectively. For `SARS-100K`, `FIGISS`, `RLCSA` and `RLFM` were between 2.81 and 3.86, 2.45 and 4.59, and 1.96 and 10.75 times slower, respectively.

Next, we consider the time and memory required for construction; which is given in Table 1. Our experiments revealed that all methods used less than 60 GB of memory on all SARS-CoV-2 datasets; `PFP-FM` used the most memory with the peak being 54 GB on the `SARS-100K` dataset. Yet, `PFP-FM` exhibited the most efficient construction time across all datasets for generating the FM-index, and this gap in the time grew with the size of the dataset. More specifically, for the `SARS-100K` dataset, `PFP-FM` used 71.04%, 65.81%, and 73.41% less time compared to other methods. In summary, `PFP-FM` significantly accelerated the `count` time, and had the fastest construction time. All methods used less than 60 GB, which is available on most commodity servers.

4.3 Results on Human Reference Genome

After measuring the time and memory usage required to construct the data structure across all methods using the `GRCh38` dataset, we observed that `PFP-FM` exhibited the second most efficient construction time but used the most construction space (71 GB vs. 26 GB to 45 GB). More specifically, `PFP-FM` was able to construct the index between 1.25 and 1.6 times faster than the `FIGISS` and `RLFM`.

Next, we compare the performance of `PFP-FM` against other methods by performing 1,000 `count` queries on, and illustrate the results in Figure 5. Our findings demonstrate that `PFP-FM` consistently outperforms all other methods. Although `RLCSA` shows better performance than `RLFM` and `FIGISS` when the pattern length is over 125 but is still 3.9, 6.2, 3.4, and 7.1 times

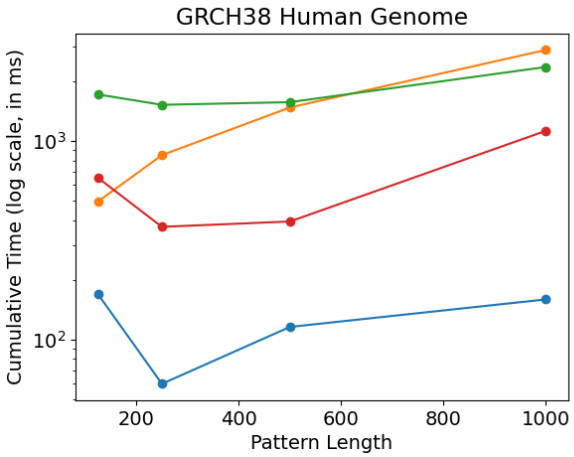


Figure 5 Comparison of query times for `count` between the described solutions when varying the length of the query pattern. For each pattern length equal to 125, 250, 500, and 1000, we report the times for the `GRCH38` dataset. We plot the cumulative time required to perform 1,000 `count` queries. The y-axis is in log scale. `PFP-FM` is shown in blue, `RLFM` is shown in orange, `RLFM` is shown in red, and `FIGISS` is shown in green.

slower than `PFP-FM`. Meanwhile, the `RLFM` method exhibits a steady increase in time usage, and it is 2.9, 14.2, 12.8, and 18.07 times slower than `PFP-FM`. It is worth noting that the `FIGISS` grammar is less efficient for non-repetitive datasets, as demonstrated in the research by Akagi et al. [1], which explains its (worse) performance on `GRCh38` versus the `SARS-100K` dataset. Hence, `FIGISS` is 10.1, 25.5, 13.6, and 14.8 times slower than `PFP-FM`. These results are inline with the performance of our previous results, and demonstrate that `PFP-FM` has both competitive construction memory and time, and achieves a significant acceleration.

5 Conclusion

In this work, we presented `PFP-FM` that shows significant acceleration over existing state-of-the-art methods. Hence, this work begins to resolve a relatively long-standing issue in data structures as to how we can parse input that has no natural word boundaries in a manner that enables acceleration of the FM-index. We note that it is possible to similarly augment `locate` queries since for that we need the suffix array samples only in the final step when matching α (or β in case that $Q = \beta$), which can be done by the usually suffix array samplings for the FM-index. If α is empty, then we can instead match the first block of the pattern with the FM-index on S and not on P. We leave this for future work. With respect to practical applications, as reads are getting longer and more accurate, we will soon see an opportunity to apply accelerations of finding patterns that have length between 125 and 1,000. Hence, a larger area that warrants future consideration is accelerating the backward search with approaches such as `PFP-FM` for aligning Nanopore reads to a database. Our last experiment shows significant acceleration with query patterns of length 1,000 to a full human reference genome, giving proof that the research community is in the position to begin such an endeavour.

Table 1 Comparison of the construction performance with the construction time and memory for all datasets. The number of characters in each dataset (denoted as n) is given in the second column. The time is reported in seconds (s), and the memory is reported in gigabytes (GB).

Dataset	n	Method	Construction Memory (GB)	Construction Time (s)
SARS-25k	751,526,774	RLCSA	9.90	322.85
		RLFM	3.47	363.74
		FIGISS	4.89	378.49
		PFP-FM	12.99	117.29
SARS-50k	1,503,252,577	RLCSA	19.88	679.89
		RLFM	6.94	701.36
		FIGISS	12.44	795.70
		PFP-FM	26.12	233.04
SARS-100k	3,004,588,730	RLCSA	39.47	1690.22
		RLFM	25.01	1432.16
		FIGISS	25.57	1840.80
		PFP-FM	53.90	489.45
GRCh38	3,189,750,467	RLCSA	45.45	924.60
		RLFM	26.31	1839.25
		FIGISS	34.65	1440.19
		PFP-FM	71.13	1154.12

References

1 Tooru Akagi, Dominik Köppl, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Grammar index by induced suffix sorting. In *Proceedings of the 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 85–99, 2021.

2 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms Molecular Biology*, 14(1):13:1–13:15, 2019.

3 Christina Boucher, Travis Gagie, Alan Kuhnle, and Giovanni Manzini. Prefix-free parsing for building big BWTs. In *Proceedings of the Workshop of Algorithms in Biology (WABI)*, pages 2:1–2:16, 2018.

4 Jin-Jie Deng, Wing-Kai Hon, Dominik Köppl, and Kunihiko Sadakane. FM-indexing grammars induced by suffix sorting for long patterns. In *Proceedings of the IEEE Data Compression Conference (DCC)*, pages 63–72, 2022.

5 Paola Ferragina and Giovanni Manzini. Indexing Compressed Text. *Journal of the ACM*, 52:552–581, 2005.

6 Paolo Ferragina and Johannes Fischer. Suffix arrays on words. In *Proceedings of the 18th Annual Symposium Combinatorial Pattern Matching (CPM)*, pages 328–339, 2007.

7 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. *Journal of the ACM*, 67(1):1–54, 2020.

8 S Gog, T Beller, A Moffat, and M Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Proceedings of the 13th Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.

9 Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J Puglisi. Fixed block compression boosting in fm-indexes: Theory and practice. *Algorithmica*, 81:1370–1391, 2019.

10 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25–R25, 2009.

11 Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *CoRR*, 2013. `arXiv:1303.3997`.

12 Veli Mäkinen and Gonzalo Navarro. Run-length FM-index. In *Proceedings of the DIMACS Workshop: "The Burrows-Wheeler Transform: Ten Years Later"*, pages 17–19, 2004.

13 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching*, pages 45–56, 2005.

14 Udi Manber and Gene W. Myers. Suffix Arrays: A New Method for On-line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

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 Taher Mun, Alan Kuhnle, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Matching Reads to Many Genomes with the r-Index. *Journal of Computational Biology*, 27(4):514–518, 2020. `doi:10.1089/cmb.2019.0316`.

17 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. `doi:10.1109/TC.2010.188`.

18 Daniel Saad Nogueira Nunes, Felipe Alves da Louza, Simon Gog, Mauricio Ayala-Rincón, and Gonzalo Navarro. A grammar compression algorithm based on induced suffix sorting. In *Proc. DCC*, pages 42–51, 2018. `doi:10.1109/DCC.2018.00012`.

19 Jouni Siren. Compressed suffix arrays for massive data. In *Proceedings of the 16th International Symposium String Processing and Information Retrieval (SPIRE)*, pages 63–74, 2009.

Exact Sketch-Based Read Mapping

Tizian Schulz
Faculty of Technology and Center for Biotechnology (CeBiTec), Bielefeld University, Germany
Bielefeld Institute for Bioinformatics Infrastructure (BIBI), Bielefeld University, Germany
Graduate School "Digital Infrastructure for the Life Sciences" (DILS), Bielefeld University, Germany

Paul Medvedev
Department of Computer Science and Engineering,
The Pennsylvania State University, University Park, PA, USA
Department of Biochemistry and Molecular Biology,
The Pennsylvania State University, University Park, PA, USA
Huck Institutes of the Life Sciences,
The Pennsylvania State University, University Park, PA, USA

Abstract

Given a sequencing read, the broad goal of read mapping is to find the location(s) in the reference genome that have a "similar sequence". Traditionally, "similar sequence" was defined as having a high alignment score and read mappers were viewed as heuristic solutions to this well-defined problem. For sketch-based mappers, however, there has not been a problem formulation to capture what problem an exact sketch-based mapping algorithm should solve. Moreover, there is no sketch-based method that can find all possible mapping positions for a read above a certain score threshold.

In this paper, we formulate the problem of read mapping at the level of sequence sketches. We give an exact dynamic programming algorithm that finds all hits above a given similarity threshold. It runs in $\mathcal{O}(|t| + |p| + \ell^2)$ time and $\Theta(\ell^2)$ space, where $|t|$ is the number of k-mers inside the sketch of the reference, $|p|$ is the number of k-mers inside the read's sketch and ℓ is the number of times that k-mers from the pattern sketch occur in the sketch of the text. We evaluate our algorithm's performance in mapping long reads to the T2T assembly of human chromosome Y, where ampliconic regions make it desirable to find all good mapping positions. For an equivalent level of precision as minimap2, the recall of our algorithm is 0.88, compared to only 0.76 of minimap2.

2012 ACM Subject Classification Applied computing → Computational biology

Keywords and phrases Sequence Sketching, Long-read Mapping, Exact Algorithm, Dynamic Programming

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.14

Supplementary Material *Software (Source Code)*: `https://github.com/medvedevgroup/eskemap` archived at `swh:1:dir:a8eaacfde8a32bab844b58efd605f7f7196c00eb`

Funding This work was supported 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).
Tizian Schulz: This research is funded in part by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie agreement [872539].
Paul Medvedev: This material is based upon work supported by the National Science Foundation under grant nos. 2138585. Research reported in this publication was also supported by the National Institutes of Health under Grant NIH R01GM146462 (to P.M.). The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

Acknowledgements We would like to thank K. Sahlin for helpful early feedback.

1 Introduction

Read mapping continues to be one of the most fundamental problems in bioinformatics. Given a read, the broad goal is to find the location(s) in the reference genome that have a "similar sequence". Traditionally, "similar sequence" was defined as having a high alignment score and read mappers were viewed as heuristic solutions to this well-defined problem. However, the last few years has seen the community embrace sketch-based mapping methods, best exemplified by minimap2 [11] (see [16] for a survey). These read mappers work not on the original sequences themselves but on their sketches, e.g. the minimizer sketch. As a result, it is no longer clear which exact problem they are trying to solve, as the definition using an alignment score is no longer directly relevant. To the best of our knowledge, there has not been a problem formulation to capture what problem an exact sketch-based mapping algorithm should solve.

In this work, we provide a problem formulation (Section 3) and an exact algorithm to find all hits above a given score (Section 6). More formally, we consider the problem of taking a sketch t of a text T and a sketch p of a query P and identifying all sub-sequences of t that match p with a score above some threshold. A score function could for example be the weighted Jaccard index, though we explore several others in this paper (Section 4). We provide both a simulation-based and an analytical-based method for setting the score threshold (Section 5). Our algorithm runs in time $\mathcal{O}(|t| + |p| + \ell^2)$ and space $\Theta(\ell^2)$, where ℓ is the number of times that k-mers from p occur in t.

Other sketch-based mappers are heuristic: they typically find matching elements between the reference and the read sketches (i.e. anchors) and extend these into maps using chaining [16]. Our algorithm is more resource intensive than these heuristics, as is typical for exact algorithms. However, a problem formulation and an exact algorithm gives several long-term benefits. First, the exact algorithm could be used in place of a greedy heuristic when the input size is not too large. Second, the formulation can spur development of exact algorithms that are optimized for speed and could thus become competitive with heuristics. Third, the formulation could be used to find the most effective score functions, which can then guide the design of better heuristics. Finally, our exact algorithm can return all hits with a score above a threshold, rather than just the best mapping(s). This is important for tasks such as the detection of copy number variation [12] or detecting variation in multi-copy gene families [1].

We evaluate our algorithm (called ESKEMAP), using simulated long reads from the T2T human Y chromosome (Section 7). For the same level of precision, the recall of ESKEMAP is 0.88, compared to 0.76 of minimap2. This illustrates the power of ESKEMAP as a method to recover more of the correct hits than a heuristic method. We also compare against Winnowmap2 [10] and edlib [18], which give lower recall but higher precision than ESKEMAP.

2 Preliminaries

Sequences. Let t be a sequence of elements (e.g. k-mers) that may contain duplicates. We let $|t|$ denote the length of the sequence, and we let $t[i]$ refer to the i-th element in t, with $t[0]$ being the first element. For $0 \leq i \leq j < |t|$, let $t[i, j]$ represent the subsequence $(t[i], t[i+1], \ldots, t[j])$. The set of elements in t is denoted by $\bar{t}$, e.g. if $t = (\texttt{ACG}, \texttt{TTT}, \texttt{ACG})$ then $\bar{t} = \{\texttt{ACG}, \texttt{TTT}\}$. We let $\text{occ}(x, t)$ represent the number of occurrences of an element x in t, e.g. $\text{occ}(\texttt{ACG}, t) = 2$.

Sketch. Let T be a string and let t be the sequence of k-mers appearing in T. Note that t is a sequence of DNA sequences. For example, if $T = \texttt{ACGAC}$ and $k = 2$, then $t = (\texttt{AC}, \texttt{CG}, \texttt{GA}, \texttt{AC})$. For the purposes of this paper, a *sketch* of T is simply a subsequence of t, e.g. $(\texttt{AC}, \texttt{GA})$. This type of sketch could for example be a minimizer sketch [15, 17], a syncmer sketch [6], or a FracMinHash sketch [9, 7].

Scoring Scheme. A *scoring scheme* $(\mathrm{sc}, \mathrm{thr})$ is a pair of functions: the score function and the threshold function. The *score function* sc is a function that takes as input a pair of non-empty sketches and outputs a real number, intuitively representing the degree of similarity. We assume it is symmetric, i.e. $\mathrm{sc}(p, s) = \mathrm{sc}(s, p)$ for all sketches p and s. If the score function has a parameter, then we write $\mathrm{sc}(s, p; \theta)$, where θ is a vector of parameter values. The *threshold function* thr takes the length of a sketch and returns a score cutoff threshold, i.e. scores below this threshold are not considered similar. Note that the scoring scheme is not allowed to depend on the underlying nucleotide sequences besides what is captured in the sketch.

Miscellenous. We use U_k to denote the universe of all k-mers. Given two sequences p and s, the *weighted Jaccard* is defined as $\frac{\sum_{x \in U_k} \min(\mathrm{occ}(x,p), \mathrm{occ}(x,s))}{\sum_{x \in U_k} \max(\mathrm{occ}(x,p), \mathrm{occ}(x,s))}$. It is 0 when s and p do not share any elements, 1 when s is a permutation of p, and strictly between 0 and 1 otherwise. The weighted Jaccard is a natural extension of Jaccard similarity that accounts for multi-occurring elements.

3 Problem Definition

In this section, we first motivate and then define the Sketch Read Mapping Problem. Fix a scoring scheme $(\mathrm{sc}, \mathrm{thr})$. Let p and t be two sketches, which we refer to as the pattern and the text, respectively. Define a *candidate mapping* as a subinterval $t[a, b]$ of t. A naive problem definition would ask to return all candidate mappings with $\mathrm{sc}(p, t[a, b]) \geq \mathrm{thr}(|p|)$.[1] However, a lower-scoring candidate mapping could contain a higher-scoring candidate mapping as a subinterval, with both scores above the threshold. This may arise due to a large candidate mapping containing a more conserved small candidate mapping, in which case both candidate mappings are of interest. But it may also arise spuriously, as a candidate mapping with a score sufficiently higher than $\mathrm{thr}(|p|)$ can be extended with non-shared k-mers that decrease the score but not below the threshold.

To eliminate most of these spurious cases, we say that a candidate mapping $t[a, b]$ is *reasonable* if and only if for $x \in \{t[a], t[b]\}$, $\mathrm{occ}(x, t[a, b]) \leq \mathrm{occ}(x, p)$. In other words, a reasonable candidate mapping must start and end with a k-mer that has a match in the pattern. We also naturally do not wish to report a candidate mapping that is a subinterval of a longer candidate mapping with a larger score. Formally, we call a candidate mapping $t[a, b]$ *maximal* if there does not exist a candidate mapping $t[a', b']$, with $a' \leq a \leq b \leq b'$ and $\mathrm{sc}(t[a', b'], p) > \mathrm{sc}(t[a, b], p)$. We can now formally define $t[a, b]$ to be a *final mapping* if it is both maximal and reasonable and $\mathrm{sc}(t[a, b], p) \geq \mathrm{thr}(|p|)$. The *Sketch Read Mapping Problem* is then to report all final mappings. We now restate the problem in a succinct manner:

[1] Notice that in this framing, the threshold is not a single parameter but can vary depending on the read length. This gives flexibility to the scoring function, since the scores of candidate mappings of reads of different lengths do not need to be comparable to each other. Moreover, computing the threshold value is not a challenge since it needs to be computed just once for each read.

$p = (\texttt{AC}, \texttt{CG}, \texttt{GA}, \texttt{AC})$ $\quad t = (\texttt{AC}, \texttt{AC}, \texttt{CA}, \texttt{TA}, \texttt{CA}, \texttt{GA}, \texttt{CG}, \texttt{AC}, \texttt{GG})$ $\quad thr(|p|) = 1$

$a\backslash b$	0	1	2	3	4	5	6	7	8
0	−2	0	−1	−2	−3	−1	**1**	0	−1
1		−2	−3	−4	−5	−3	−1	**1**	−5
2			−5	−6	−7	−5	−3	−1	−2
3				−5	−6	−4	−2	0	−1
4					−5	−3	−1	1	0
5						−2	0	**2**	1
6							−2	0	−1
7								−2	−3
8									−5

Figure 1 An example of the Sketch Read Mapping Problem. We show all candidate mappings $t[a, b]$ for a given pattern p and a text t. Each candidate mapping is represented by its score calculated using $sc_\ell(p, t[a, b]; 1)$ (see Section 4). Reasonable candidate mappings are shown in black (rather than gray) and final mappings are further bolded.

▶ **Definition 1** (Sketch Read Mapping Problem). *Given a pattern sketch p, a text sketch t, a score function sc, and a threshold function thr, the Sketch Read Mapping Problem is to find all $0 \le a \le b < |t|$ such that*

- $sc(p, t[a, b]) \ge thr(|p|)$,
- $occ(t[a], t[a, b]) \le occ(t[a], p)$,
- $occ(t[b], t[a, b]) \le occ(t[b], p)$,
- *there does not exist $a' \le a \le b \le b'$ such that $sc(t[a', b'], p) > sc(t[a, b], p)$, i.e. $t[a, b]$ is maximal.*

4 Score Function

In this section, we explore the design space of score functions and fix two score functions for the rest of the paper. Let p be the sketch of the pattern and let s be a continuous subsequence of the sketch of the text t, i.e. $s = t[a, b]$ for some $a \le b$. For example if $p = (\texttt{ACT}, \texttt{GTA}, \texttt{TAC})$ and $t = (\texttt{AAC}, \texttt{ACT}, \texttt{CCT}, \texttt{GTA})$, we could have $s = t[1, 3] = (\texttt{ACT}, \texttt{CCT}, \texttt{GTA})$. In the context of the Sketch Read Mapping Problem, p is fixed and s varies. Therefore, while the score function is symmetric, the threshold function sets the score threshold as a function of $|p|$. Since p is fixed, the threshold is a single number in the context of a single problem instance.

In the following, we exclusively consider score functions that calculate the similarity of s and p by ignoring the order of k-mers inside the sketches. Taking k-mer order into account would likely make it more complex to compute scores, while not necessarily giving better results on real data. However, score functions that do take order into account are possible and could provide better accuracy in some cases.

A good score function should reflect the number of k-mers shared between s and p. For a given k-mer x, we define

$$x_{\min} := \min(\mathrm{occ}(x, p), \mathrm{occ}(x, s))$$
$$x_{\max} := \max(\mathrm{occ}(x, p), \mathrm{occ}(x, s))$$
$$x_{\mathrm{diff}} := x_{\max} - x_{\min}$$

Intuitively, x occurs a certain number of times in p and a certain number of times in s; we let $x_{\min}$ be the smaller of these two numbers and $x_{\max}$ be the larger of these two numbers. Similarly, x_{diff} is the absolute difference between how often x occurs in p and s. We say that the number of *shared* occurrences is $2x_{\min}$ and the number of *non-shared* occurrences is x_{diff}. These quantities are governed by the relationships

$$|s| + |p| = \sum_{x \in U_k} \text{occ}(x, p) + \text{occ}(x, s) = \sum_{x \in U_k} x_{\min} + x_{\max} = \sum_{x \in U_k} 2x_{\min} + x_{\text{diff}}. \tag{1}$$

A good score function should be (1) increasing with respect to the number of shared occurences and (2) decreasing with respect to the number of non-shared occurences. There are many candidate score functions within this space. The first score function we consider is the weighted Jaccard. Formally,

$$\text{sc}_{\text{j}}(s, p) := \frac{\sum_{x \in U_k} x_{\min}}{\sum_{x \in U_k} x_{\max}} = \frac{\sum_x x_{\min}}{|s| + |p| - \sum_x x_{\min}} = \frac{\sum_x x_{\min}}{\sum_x (x_{\min} + x_{\text{diff}})} \tag{2}$$

The above formula includes first the definition but then two algebraically equivalent versions of it, derived using Eq. 1. The weighted Jaccard has the two desired properties of a score function and is a well-known similarity score. However, it has two limitations. First, the use of a ratio makes it challenging to analyze probabilistically, as is the case with the non-weighted Jaccard [3]. Second, it does not offer a tuning parameter which would control the relative benefit of a shared occurence to the cost of a non-shared occurence. We therefore consider another score function, parameterized by a real-valued tuning parameter $w > 0$:

$$\text{sc}_{\ell}(s, p; w) := \sum_{x \in U_k} x_{\min} - w x_{\text{diff}}.$$

It is sometimes more useful to use an equivalent formulation, obtained using Eq. 1:

$$\text{sc}_{\ell}(s, s'; w) = \sum_{x \in U_k} (1 + 2w) x_{\min} - w(|s| + |s'|). \tag{3}$$

As with the weighted Jaccard, sc_{ℓ} has the two desired properties of a score function. But, unlike the weighted Jaccard, it is linear and contains a tuning parameter w.

To understand how score functions relate to each other, we introduce the notion of domination and equivalence. Informally, a score function sc_1 dominates another score function sc_2 when sc_1 can always recover the separation between good and bad scores that sc_2 can. In this case, the solution obtained using sc_2 can always be obtained by using sc_1 instead. Formally, let sc_1 and sc_2 be two score functions, parameterized by θ_1 and θ_2, respectively. We say that sc_1 *dominates* sc_2 if and only if for any parameterization θ_2, threshold function thr_2, and pattern sketch p there exist a θ_1 and thr_1 such that, for all sequences s, we have that $\text{sc}_2(s, p; \theta_2) \geq \text{thr}_2(|p|)$ if and only if $\text{sc}_1(s, p; \theta_1) \geq \text{thr}_1(|p|)$. Furthermore, sc_1 dominates sc_2 *strictly* if and only if the opposite does not hold, i.e. sc_2 does not dominate sc_1. Otherwise, sc_1 and sc_2 are said to be *equivalent*, i.e. if and only if each one dominates the other.

We can now precisely state the relationship between sc_{ℓ} and sc_{j}, i.e. that sc_{ℓ} strictly dominates sc_{j}. In other words, any solution to the Sketch Read Mapping Problem that is obtained by sc_{j} can also be obtained by sc_{ℓ}, but not vice-versa. Formally,

▶ **Theorem 2.** *sc_{ℓ} stricly dominates the weighted Jaccard score function sc_j.*

Proof. We start by proving that sc_ℓ dominates sc_j. Let p be a pattern sketch and let thr_j be the threshold function associated with sc_j. We will use the shorthand $t = \mathrm{thr}_j(|p|)$. First, consider the case that $t < 1$. Let $w = \frac{t}{1-t}$ and let thr_ℓ evaluate to zero for all inputs. Let s be any sketch. The following is a series of equivalent transformations that proves domination.

$$\begin{aligned}
\mathrm{sc}_\mathrm{j}(s,p) &\geq t \\
\frac{\sum_x x_{\min}}{\sum_x x_{\min} + x_{\mathrm{diff}}} &\geq t \\
\sum_x x_{\min} &\geq \sum_x t x_{\min} + t x_{\mathrm{diff}} \\
\sum_x (1-t) x_{\min} - t x_{\mathrm{diff}} &\geq 0 \\
\sum_x x_{\min} - \frac{t}{1-t} x_{\mathrm{diff}} &\geq 0 \\
\mathrm{sc}_\ell(s,p;w) &\geq \mathrm{thr}_\ell(|p|)
\end{aligned}$$

Next, consider the case $t > 1$. In this case, for all s, $\mathrm{sc}_\mathrm{j}(s,p) < t$, since the weighted Jaccard can never exceed one. Observe that $\mathrm{sc}_\ell(s,p;w) \leq |p|$ for any non-negative w. Therefore, we can set $\mathrm{thr}_\ell(|p|) = |p| + 1$ and let w be any non-negative number, guaranteeing that for all s, $\mathrm{sc}_\ell(s,p;w) < \mathrm{thr}_\ell(|p|)$.

Finally consider the case that $t = 1$. Then, $\mathrm{sc}_\mathrm{j}(s,p) \geq t$ if and only if s and p are permutations of each other, i.e. $x_{\mathrm{diff}} = 0$ for all x. Setting $\mathrm{thr}_\ell(|p|) = |p|$ and letting w be any strictly positive number guarantees that $\mathrm{sc}_\ell(s,p;w) \geq \mathrm{thr}_\ell(|p|)$ if and only if s and p are permutations of each other.

To prove that sc_ℓ is not dominated by sc_j, we fix $w = 1$ (though any value could be used) and give a counterexample family to show that sc_j cannot recover the separation that sc_ℓ can. Pick an integer $i \geq 1$ to control the size of the counterexample. Let p be a pattern sketch of length $4i$ consisting of arbitrary k-mers. We construct two sketches, s_1 and s_2. The sequence s_1 is an arbitrary subsequence of p of length i. Observe that $\sum_{x \in \overline{p} \cup \overline{s_1}} x_{\min} = \sum_x \mathrm{occ}(x, s_1) = i$. The sequence s_2 is p appended with arbitrary k-mers to get a length $12i$. Observe that $\sum_{x \in \overline{p} \cup \overline{s_2}} x_{\min} = \sum_x \mathrm{occ}(x,p) = 4i$. Using Eq. 3 for sc_ℓ and Eq. 2 for sc_j,

$$\begin{aligned}
\mathrm{sc}_\ell(s_1,p) &= -2i & \mathrm{sc}_\mathrm{j}(s_1,p) &= 1/4 \\
\mathrm{sc}_\ell(s_2,p) &= -4i & \mathrm{sc}_\mathrm{j}(s_2,p) &= 1/3
\end{aligned}$$

Under sc_ℓ, s_1 has a higher score, while under sc_j, s_2 has a higher score. If thr_ℓ is set to accept s_1 but not s_2 (e.g. $\mathrm{thr}_\ell = -3i$), then it is impossible to set thr_j to achieve the same effect. In other words, since $\mathrm{sc}_\mathrm{j}(s_2) > \mathrm{sc}_\mathrm{j}(s_1)$, any threshold that accepts s_1 must also accept s_2. ◀

Next, we show that many other natural score functions are equivalent to sc_ℓ. Consider the following score functions:

$$\mathrm{sc_A}(s,p;a_1) := \sum_{x\in U_k} (a_1 x_{\min} - x_{\mathrm{diff}}) \qquad \text{with } a_1 > 0$$

$$\mathrm{sc_B}(s,p;b_1,b_2) := \sum_{x\in U_k} (b_1 x_{\min} - b_2 x_{\mathrm{diff}}) \qquad \text{with } b_1 > 0 \text{ and } b_2 > 0$$

$$\mathrm{sc_C}(s,p;c_1,c_2) := \sum_{x\in U_k} (c_1 x_{\min} - c_2 x_{\max}) \qquad \text{with } c_1 > c_2 > 0$$

$$\mathrm{sc_D}(s,p;d_1,d_2) := \sum_{x\in U_k} (d_1 x_{\min}) - d_2|s| \qquad \text{with } d_1 > 2d_2 \text{ and } d_2 > 0$$

The conditions on the parameters are there to enforce the two desired properties of a score function. Each of these score functions is natural in its own way, e.g. $\mathrm{sc_A}$ is similar to sc_ℓ but places the weight on $x_{\min}$ rather than on x_{diff}. One could also have two separate weights, as in the score $\mathrm{sc_B}$. One could then replace x_{diff} with $x_{\max}$, as in $\mathrm{sc_C}$, which is the straightforward reformulation of the weighted Jaccard score as a difference instead of a ratio. Or one could replace x_{diff} with the length of s, as in $\mathrm{sc_D}$. The following theorem shows that the versions turn out to be equivalent to sc_ℓ and to each other. The proof is a straightforward algebraic manipulation and is left for the appendix.

▶ **Theorem 3.** *The score functions sc_ℓ, sc_A, sc_B, sc_C, and sc_D are pairwise equivalent.*

5 Choosing a Threshold

In this section, we propose two ways to set the score threshold. The first is analytical (Section 5.1) and the second is with simulations (Section 5.2). The analytical approach gives a closed form formula for the expected value of the score under a mutation model. However, it only applies to the FracMinHash sketch, assumes a read has little internal homology, and does not give a confidence interval. The simulation approach can apply to any sketch but does not offer any analytical insight into the behavior of the score. The choice of approach ultimately depends on the use case.

We first need to define a generative mutation model to capture both the sequencing and evolutionary divergence process:

▶ **Definition 4** (Mutation model). *Let S be a circular string[2] with n characters. The mutation model produces a new string S' by first setting $S' = S$ and then taking the following steps:*

1. *For every $0 \le i < n$, draw an action $a_i \in \{sub, del, unchanged\}$ with probability of p_{sub} for sub, p_{del} for del, and $1 - p_{sub} - p_{del}$ for unchanged. Also, draw an insertion length b_i from a geometric distribution with mean p_{ins}[3].*
2. *Let track be a function mapping from a position in S to its corresponding position in S'. Initially, $track(i) = i$, but as we delete and add characters to S', we assume that track is updated to keep track of the position of $S[i]$ in S'.*
3. *For every i such that $a_i = sub$, replace $S'[i]$ with one of the three nucleotides not equal to $S[i]$, chosen uniformly at random.*
4. *For every $0 \le i < n$, insert b_i nucleotides (chosen uniformaly at random) before $S'[track(i)]$.*
5. *For every i such that $a_i = del$, remove $S'[track(i)]$ from S'.*

[2] We assume the string is circular to avoid edge cases in the analysis but, for long enough strings, this assumption is unlikely to effect the accuracy of the results.

[3] Here, a geometric distribution is the number of failures before the first success of a Bernoulli trial. This geometric distribution has parameter $\frac{1}{p_{\mathrm{ins}}+1}$.

5.1 Analytical Analysis

To derive an expected score under the mutation model, we need to specify a sketch. We will use the FracMinHash sketch [9], due its simplitiy of analysis [7].

▶ **Definition 5** (FracMinHash). *Let h be a hash function that maps a k-mer to a real number between 0 and 1, inclusive. Let $0 < q \leq 1$ be a real-valued number called the sampling rate. Let S be a string. Then the* FracMinHash *sketch of S, denoted by s, is the sequence of all k-mers x of S, ordered as they appear in S, such that $h(x) \leq q$.*

Consider an example with $k = 2$, $S = \texttt{CGGACGGT}$, and the only k-mers hashing to a value $\leq q$ being `CG` and `GG`. Then, $s = (\texttt{CG}, \texttt{GG}, \texttt{CG}, \texttt{GG})$.

We make an assumption, which we refer to as the *mutation-distinctness assumption*, that the mutations on S never create an k-mer that is originally in S. Based on previous work [4], we find this necessary to make the analysis mathematically tractable (for us). The results under this assumption become increasingly inaccurate as the read sequence contains increasingly more internal similarity. For example, reads coming from centromeres might violate this assumption. In such cases, it may be better to choose a threshold using the technique in Section 5.2.

We can now derive the expected value of the score under the mutation model and FracMinHash.

▶ **Theorem 6.** *Let S be a circular string and let S' be generated from S under the mutation model with the mutation-distinctness assumption and with parameters p_{sub}, p_{del}, and p_{ins}. Let s and s' be the FracMinHash sketches of S and S', respectively, with sampling rate q. Then, for all real-valued tuning parameters $w > 0$,*

$$E[sc_\ell(s, s'; w)] = |s|q\left(\alpha + w(2\alpha - 2 + p_{del} - p_{ins})\right),$$

where $\alpha = \frac{(1-p_{del}-p_{sub})^k}{(p_{ins}+1)^{k-1}}$.

Proof. Observe that under mutation-distinctness assumption, the number of occurrences of a k-mer that is in s can only decrease after mutation, and a k-mer that is newly created after mutation has an $x_{\min}$ of 0. Therefore, applying Eq. 3,

$$\mathrm{sc}_\ell(s, s'; w) = \sum_{x \in \overline{s}} (1 + 2w)(\mathrm{occ}(x, s') - w(|s| + |s'|)$$

(Recall that $\overline{s}$ is the set of all k-mers in s.) We will first compute the score conditioned on the hash function of the sketch being fixed. Note that when h is fixed, then the sketch s becomes fixed and s' becomes only a function of S'. By linearity of expectation,

$$E[\mathrm{sc}_\ell(s, s'; w) \mid h] = \sum_{x \in \overline{s}} (1 + 2w) E[\mathrm{occ}(x, s') \mid h] - w(|s| + E[|s'| \mid h]) \tag{4}$$

It remains to compute $E[|s'| \mid h]$ and $E[\mathrm{occ}(x, s') \mid h]$. Observe that the number of elements in s' is the number of elements in s minus the number of deletions plus the sum of all the insertion lengths. By linearity of expectation,

$$E[|s'| \mid h] = |s| - p_{\mathrm{del}}|s| + p_{\mathrm{ins}}|s| = |s|(1 - p_{\mathrm{del}} + p_{\mathrm{ins}})$$

Next, consider a k-mer $x \in \overline{s}$ and $E[\mathrm{occ}(x, s')]$. Recall by our mutation model that no new occurrenes of x are introduced during the mutation process. So $\mathrm{occ}(x, s')$ is equal to the number of occurrences of x in S that remain unaffected by mutations. Consider an

occurrence of x in s. The probability that it remains is the probability that all actions on the k nucleotides of x were "unchanged" and the length of all insertions in-between the nucleotides was 0. Therefore,

$$E[\mathrm{occ}(x, s') \mid h] = \mathrm{occ}(x, s)(1 - p_{\mathrm{del}} - p_{\mathrm{sub}})^k \left(\frac{1}{p_{\mathrm{ins}} + 1}\right)^{k-1} = \alpha \mathrm{occ}(x, s)$$

Putting it all together,

$$\begin{aligned}
E[\mathrm{sc}_\ell(s, s'; w) \mid h] &= \sum_{x \in \overline{s}} (1 + 2w) E[\mathrm{occ}(x, s') \mid h] - w(|s| + E[|s'| \mid h]) \\
&= \alpha(1 + 2w) \sum_{x \in \overline{s}} \mathrm{occ}(x, s) - w(|s| + |s|(1 - p_{\mathrm{del}} + p_{\mathrm{ins}})) \\
&= \alpha(1 + 2w)|s| - w(|s| + |s|(1 - p_{\mathrm{del}} + p_{\mathrm{ins}})) \\
&= |s|(\alpha(1 + 2w) - w(2 - p_{\mathrm{del}} + p_{\mathrm{ins}}))) \\
&= |s|(\alpha + w(2\alpha - 2 + p_{\mathrm{del}} - p_{\mathrm{ins}}))
\end{aligned}$$

To add the sketching step, we know from [7] that the expected size of a sketch is the size of the original text times q. Then,

$$\begin{aligned}
E[\mathrm{sc}_\ell(s, s'; w)] &= E[E[\mathrm{sc}_\ell(s, s'; w) \mid h]] \\
&= E[|s|(\alpha + w(2\alpha - 2 + p_{\mathrm{del}} - p_{\mathrm{ins}}))] \\
&= E[|s|](\alpha + w(2\alpha - 2 + p_{\mathrm{del}} - p_{\mathrm{ins}})) \\
&= |s|q(\alpha + w(2\alpha - 2 + p_{\mathrm{del}} - p_{\mathrm{ins}}))
\end{aligned}$$

◀

5.2 Simulation-Based Analysis

First, we choose the parameters of the mutation model according to the target sequence divergence between the reads and the reference caused by sequencing errors, but also due to the evolutionary distance between the reference and the organism sequenced. If one is also interested in mapping reads to homologous regions within the reference that are related more distantly, e.g. if there exist multiple copies of a gene, the mutation parameters can be increased further.

To generate a threshold for a given read length, we generate sequence pairs (S, S'), where S is a uniformly random DNA sequence of the given length and S' is mutated from S under the above model. We then calculate the sketch of S and S', which we call s and s', respectively. The sketch can for example be a minimizer sketch, a syncmer sketch, or a FracMinHash sketch. We can then use the desired score function to calculate a score for each pair (s, s'). For a sufficiently large number of pairs, their scores will form an estimate of the underlying score distribution for sequences that evolved according to the used model. It is then possible to choose a threshold such that the desired percentage of mappings would be reported by our algorithm. For example, one could choose a threshold to cover a one sided 95% confidence interval of the score.

In order to be able to adjust thresholds according to the variable length of reads produced from a sequencing run, the whole process may be repeated several times for different lengths of S. Thresholds can then be interpolated dynamically for dataset reads whose lengths were not part of the simulation.

6 Algorithm for the Sketch Read Mapping Problem

In this section, we describe a dynamic programming algorithm for the Sketch Read Mapping Problem under both the weighted Jaccard and the linear scores ($\mathrm{sc_j}$ and sc_ℓ, respectively). Let t be the sketch of the text, let p be the sketch of the pattern, let L be the sequence of positions in t that have a k-mer that is in $\overline{p}$, in increasing order, and let $\ell = |L|$. Our algorithm takes advantage of the fact that p is typically much shorter than t and hence the number of elements of t that are shared with p is much smaller than $|t|$ (i.e. $\ell \ll |t|$). In particular, it suffices to consider only candidate mappings that begin and end in positions listed in L, since by definition, if $t[a, b]$ is a reasonable candidate mapping, then $t[a] \in \overline{p}$ and $t[b] \in \overline{p}$.

We present our algorithm as two parts. In the first part (Section 6.1), we compute a matrix S with ℓ rows and ℓ columns so that $S(i, j) = \sum_x \min(\mathrm{occ}(x, p), \mathrm{occ}(x, t[L[i], L[j]])$. S is only defined for $j \geq i$. We also mark each cell of S as being reasonable or not. In the second part (Section 6.2), we scan through S and output the candidate mapping $t[i, j]$ if and only if it is maximal and has a score above the threshold.

The reason that $S(i, j)$ is not defined to store the score of the candidate mapping $t[L[i], L[j]]$ is that the score can be computed from $S(i, j)$ in constant time, for both $\mathrm{sc_j}$ and sc_ℓ. To see this, let $x_{\min} := \min(\mathrm{occ}(x, p), \mathrm{occ}(x, t[L[i], L[j]])$. Recall that Equation (2) allows us to express $\mathrm{sc_j}(t[i, j], p)$ as a function of $\sum x_{\min}$, $|p|$, and the length of the candidate mapping, i.e. $j - i + 1$. Similarly, we can apply Equation (1) to express sc_ℓ as

$$\begin{aligned}\mathrm{sc}_\ell(t[i, j], p; w) &:= \sum_x (x_{\min} - w x_{\mathrm{diff}}) = \left(\sum_x x_{\min}\right) - w(|s| + |p| - \sum_x 2x_{\min}) \\ &= (1 + 2w) \sum_x x_{\min} - w(j - i + 1 + |p|)\end{aligned}$$

Thus, once $\sum_x x_{\min}$ is computed, either of the scores can be computed trivially.

6.1 Computing S

We compute S using dynamic programming. For the base case of the diagonal, i.e. for $0 \leq i < \ell$, we can set $S(i, i) = 1$. Here, since we know that $L[i] \in \overline{p}$, we get that the k-mer $t[L[i]]$ occurs at least once in p and exactly once in $t[L[i], L[i]]$. For the general case, i.e. for $0 \leq i < j < \ell$, we can define S using a recursive formula:

$$S(i, j) = S(i, j - 1) + \begin{cases} 1 & \text{if } \mathrm{occ}(t[L[j]], t[L[i], L[j - 1]]) < \mathrm{occ}(t[L[j]], p) \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

To see the correctness of this formula, observe that all the elements of $t[L[j-1]+1, L[j]-1]$ are, by definition, not in $\overline{p}$ and hence their minimum occurrence value is 0. If the element $x = t[L[j]]$ helps increase $\min(\mathrm{occ}(x, t[L[i], L[j-1]]), \mathrm{occ}(x, p))$, then we increase the minimum count by one, otherwise the minimum occurrence does not increase. Furthermore, we can mark $S(i, j)$ as being right-reasonable anytime that the top case is used and as not being right-reasonable otherwise.

To design an efficient algorithm based on Equation (5), we need two auxiliary data structures. The first is a hash table H_{cnt} that stores, for every k-mer in $\overline{p}$, how often it occurs in p. A second hash table H_{loc} stores, for every k-mer $x \in \overline{p}$, the number of locations i such that $t[L[i]] = x$.

The algorithm for computing S and the hash tables is given in Algorithm 1. As a first step, the H_{cnt} hash table is constructed via a scan through p. Then, the S matrix is filled in column-by-column using Equation (5). However, doing the check to determine which case of Equation (5) to use (i.e. to compute $\text{occ}(t[L[j]], t[L[i], L[j-1]])$) would take non-constant time using a naive approach. In order to compute this in constant time, let $c_1 = \text{occ}(x, t[0, L[i-1]])$ and let $c_2 = \text{occ}(x, t[0, L[j-1]])$ and observe that $\text{occ}(x, t[L[i], L[j-1]]) = c_2 - c_1$. We will now describe how to maintain c_2 and c_1 as we process a column of S, with only constant time per cell.

To compute c_2, we avoid building H_{loc} outright and instead build H_{loc} at the same time as we are processing S, column-by-column. When processing column j with $x = L[j]$, we start by incrementing the count of $H_{\text{loc}}[x]$ (Line 7). Let H^j_{loc} refer to H_{loc} right after making this increment. Observe that H^j_{loc} is H_{loc} but only containing the counts of locations up to $L[j]$, and $H^{\ell}_{\text{loc}} = H_{\text{loc}}$. Computing c_2 is trivial from H^j_{loc}– it is simply $H^j_{\text{loc}}[x] - 1$ (Line 9).

To compute c_1, we use the fact that when computing a column of S, we are processing all the rows starting from 0 up to $\ell - 1$. We initially set $c_1 = 0$ (Line 8) and then, for each new row i, we increment c_1 if $t[L[i]] = x$ (Line 17).

After S has been filled, we can identify which of the candidate mappings are reasonable. Observe that a candidate mapping $t[L[i], L[j]]$ is reasonable if and only if $S(i, j) > S(i+1, j)$ and $S(i, j) > S(i, j-1)$. This can be verified by a simple pass through the matrix (Lines 22–31).

6.2 Computing Maximality

In the second step, we identify which of the candidate mappings in S are maximal. Our algorithm is shown in Algorithm 2. We traverse S column-by-column starting with the last column and then row-by-row, starting from the first row. While traversing S, we maintain a list M of all maximal reasonable candidate mappings above the threshold found so far and their scores. M has the invariant that the candidate mappings are increasingly ordered by their start positions.

To maintain the invariant that M is sorted by start position, we maintain a pointer *cur* to a location in M (Lines 7–11). At the start of a new column traversal, when the row $i = 0$, *cur* points to the start of M. As the row is increased, we move *cur* forward until it hits the first value in M with a start larger than i. When a new final mapping is added to M, we do so at *cur*, which guarantees the order invariant of M (Lines 16–20).

Due to the order cells in S are processed during our traversal, a candidate mapping $t[L[i], L[j]]$ is maximal if and only if its score is larger than the score of all other final mappings in M with position $i' \leq i$. For a given column, since we are processing the candidate mappings in increasing order of i, we can simultenously maintain a running variable $maxSoFar$ that holds the maximum value in M up to *cur* (Line 8). We can then determine if a candidate mapping is maximal by simply checking its score against $maxSoFar$ (Line 14).

Note that as long as we have not yet seen any final mapping up to position $i' \leq i$, a candidate mapping is already maximal if its score equals $\text{thr}(|p|)$. This is ensured via a flag $supMpFnd$ and an additional satisfiable subclause (Line 14). As soon as $maxSoFar$ is updated, $supMpFnd$ is set (Line 10).

Algorithm 1 Part 1 of ESKEMAP Algorithm.

Input: two sketches t and p
Output: the matrix S and annotation of each upper diagonal cell as being reasonable or not

Construct H_{cnt}
Initialize H_{loc} to be an empty hash table initialized with zeros
Construct L array
$S(0,0) = 1$
for $j = 1$ to $\ell - 1$ **do**
    $x \leftarrow t[L[j]]$
    $H_{\text{loc}}[x] = H_{\text{loc}}[x] + 1$
    $c_1 \leftarrow 0$ ▷ This will hold $\text{occ}(x, t[0, L[i-1]])$
    $c_2 \leftarrow H_{\text{loc}}[x] - 1$ ▷ This holds $\text{occ}(x, t[0, L[j-1]])$
    for $i = 0$ to $j - 1$ **do**
        if $c_2 - c_1 < H_{\text{cnt}}[x]$ **then**
            $S(i,j) \leftarrow S(i, j-1) + 1$
        else
            $S(i,j) \leftarrow S(i, j-1)$
        end if
        if $t[L[i]] = x$ **then**
            $c_1 \leftarrow c_1 + 1$
        end if
    end for
    $S(j,j) \leftarrow 1$
end for
for $i = 0$ to $\ell - 1$ **do** ▷ Mark each cell as reasonable or not
    Mark $S(i,i)$ as reasonable
    for $j = i + 1$ to $\ell - 1$ **do**
        if $S(i,j) > S(i+1,j)$ and $S(i,j) > S(i,j-1)$ **then**
            Mark $S(i,j)$ as reasonable
        else
            Mark $S(i,j)$ as not reasonable
        end if
    end for
end for

Algorithm 2 Part 2 of ESKEMAP algorithm.

Input: two sketches t and p, the matrix S computed by Algorithm 1, a score function, and a threshold function thr
Output: all final mappings that are reasonable, maximal, and have a score of at least thr($|p|$)

```
Let M be an empty linked list with (i, j, s) tuples.
for j = ℓ − 1 down to 0 do
    maxSoFar ← thr(|p|)
    cur ← M.start
    supMpFnd ← false
    for i = 0 to j do
        while cur ≠ M.end and cur.i ≤ i do
            maxSoFar = max(maxSoFar, cur.s)
            cur++
            supMpFnd ← true
        end while
        if S(i, j) is reasonable then
            s ← score of S(i, j)
            if s > maxSoFar or (¬supMpFnd ∧ s = thr(|p|) then
                Output t[L[i], L[j]]
                if cur ≠ M.start then
                    M.insertBefore(cur, (i, j, s))
                else
                    M.insertAt(cur, (i, j, s))
                end if
            end if
        end if
    end for
end for
```

6.3 Runtime and Memory Analysis

The runtime for Algorithm 1 is $\Theta(\ell^2) + |t| + |p|$. Note that the H_{cnt} table can be constructed in a straightforward manner in $\mathcal{O}(|p|)$ time, assuming a hash table with constant insertion and lookup time; the L array is constructed in $\mathcal{O}(|t|)$. Algorithm 2 runs two for loops with constant time internal operations, with the exception of the while loop to fast forward the *cur* pointer. The total time for the loop is amortized to $O(\ell)$ for each column. Therefore, the total time for Algorithm 2 is $\Theta(\ell^2)$. This gives the total running time for our algorithm as $\mathcal{O}(|t| + |p| + \ell^2)$.

The total space used by the algorithm is the sum of the space used by S (i.e. $\Theta(\ell^2)$) and the space used by H_{cnt}, H^j_{loc}, and L. The H_{cnt} table stores $|p|$ integers with values up to $|p|$. However, notice that when $|p| > \ell$, we can limit the table to only store k-mers that are in $\bar{t}$, i.e. only ℓ k-mers. We can also replace integer values greater than ℓ with ℓ, as it would not affect the algorithm. Therefore, the H_{cnt} table uses $\mathcal{O}(\ell \log \ell)$ space. The H^j_{loc} table stores at most ℓ entries with values at most ℓ and therefore takes $\Theta(\ell \log \ell)$ space. Thus our algorithm uses a total of $\Theta(\ell^2)$ space.

7 Results

We implemented the ESKEMAP algorithm described in Section 6 using sc_ℓ as score function and compared it to other methods in a read mapping scenario. For better comparability, we implemented it with the exact same minimizer sketching approach as used by minimap2. Source code of our implementation as well as a detailed documentation of our comparison described below including exact program calls is available from `https://github.com/medvedevgroup/eskemap`.

7.1 Datasets

For our evaluation, we used the T2T reference assembly of human chromosome Y (T2T-CHM13v2.0) [13]. The chromosome contains many ampliconic regions with duplicated genes from several gene families. Identifying a single best hit for reads from such regions is not helpful and instead it is necessary to find all good mappings [5]. Such a reference poses a challenge to heuristic algorithms and presents an opportunity for an all-hits mapper like ESKEMAP to be worth the added compute.

We simulated a read dataset on this assembly imitating characteristics of a PacBio Hifi sequencing run [8]. For each read, we randomly determined its length r according to a gamma distribution with a 9000bp mean and a standard deviation of 7000bp. Afterwards, a random integer $i \in [1, n - r + 1]$ was drawn as the read's start position, where n refers to the length of the chromosome. Sequencing errors were simulated by introducing mutations into each read's sequence using the mutation model described in Definition 4 and a total mutation rate of 0.2% distributed with a ratio of 6:50:54 between substitution/insertion/deletion, as suggested in [14]. Aiming for a sequencing depth of 10x, we simulated 69401 reads.

The T2T assembly of the human chromosome Y contains long centromeric and telomeric regions which consist of short tandem and higher order repeats. Mapping reads in such regions results in thousands of hits that are meaningless for many downstream analyses and significantly increases the runtime of mapping. Therefore, we excluded all reads from the initially simulated set which could be aligned to more than 20 different, non-overlapping positions using edlib (see below). After filtering, a set of 32295 reads remained.

7.2 Tools

We compared ESKEMAP to two other sketch-based approaches and an exact alignment approach. The sketch-based approaches were minimap2 (version 2.24-r1122) and Winnowmap2 (version 2.03), run using default parameters. In order to be able to compare our results also to an exact, alignment-based mapping approach, we used the C/C++ library of Edlib [18] (version 1.2.7) to implement a small script that finds all non-overlapping substrings of the reference sequence a read could be aligned to with an edit distance of at most T. We tried values $T \in \{0.01r, 0.02r, 0.03r\}$, where recall that r is the read length. We refer to this script as simply *edlib*.

For ESKEMAP, we aimed to make the results as comparable as possible to minimap2. We therefore used a minimizer sketch with the same k-mer and window size as minimap2 ($k = 15$, $w = 10$). However, we excluded minimizers that occurred > 100 times inside the reference sketch, to limit the $\mathcal{O}(\ell^2)$ memory use of ESKEMAP, even as this exclusion may potentially hurt ESKEMAP's accuracy. We used the default $w = 1$ as the tuning parameter in the linear score. To set the score threshold, we used the dynamic procedure described in Section 5.2. In particular, we used five different sequence lengths for simulations and used a

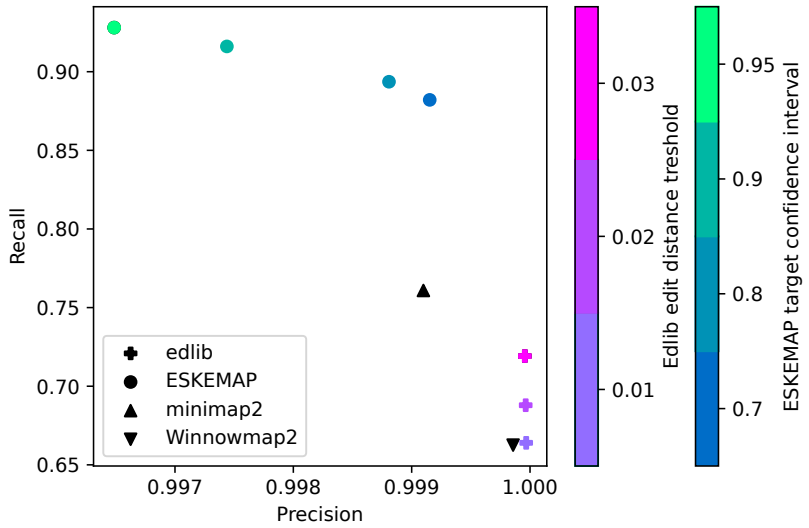


Figure 2 Mapping accuracies of all tools. For edlib, the color of the cross encodes the various edit distance thresholds (0.01, 0.02, 0.03). For ESKEMAP, the color of the circles indicate the score threshold used, in terms of the target confidence interval used (0.7, 0.8, 0.9, 0.95). The ground truth is determined by combining the mappings from all tools and filtering out those with bad BLAST scores. The most lenient thresholds for edlib and ESKEMAP were used.

divergence of 1%. We used the same sequencing error profile as for read simulation. Four thresholds were then chosen so at to cover the one-sided confidence interval of 70%, 80%, 90%, and 95%, respectively.

7.3 Accuracy Measure

We compared the reference substrings corresponding to each reported mapping location of any tool to the mapped read's sequence using BLAST [2]. If a pairwise comparison of both sequences resulted either in a single BLAST hit with an E-value not exceeding 0.01[4] and covering at least 90% of the substring or the read sequence or if a set of non-overlapping BLAST hits was found of which none had an E-value above 0.01 and their lengths summed up to at least 90% of either the reference substring's or the read sequence's length, we considered the reference mapping location as homologous.

For each read, we combine all the homologous reference substrings found across all tools into a ground truth set for that read. We then measure the accuracy of a mapping as follows. We determined for each k-mer of the reference sequence's sketch whether it is either a *true positive* (TP), *false positive* (FP), *true negative* (TN) or *false negative* (FN). A k-mer was considered a TP if it was covered by both a mapping and a ground truth substring. It was considered a FP if it was covered by a mapping, but not by any ground truth substring. Conversely, it was considered a TN if it was covered by neither a mapping nor a ground truth substring and considered a FN if it was covered by a substring of the ground truth exclusively. The determination was performed for each read independently and results were accumulated per tool to calculate precision and recall measures.

[4] In order to ensure robustness of results, BLAST runs were also repeated for E-value thresholds of 0.005 and 0.001 causing only neglectable differences for subsequent analyses.

Table 1 Runtime and memory usage comparison of all sketch-based methods. The tools were called to map 32295 simulated PacBio Hifi sequencing reads on the T2T assembly of human chromosome Y. Runtimes are shown both as total values and normalized by the number of reported mapping positions.

Tool	User Time [s]		Memory [GB]
	total	per mapping	
ESKEMAP	100,770	0.01	69
minimap2	26,232	0.55	4.5
Winnowmap2	9,207	0.19	7

7.4 Accuracy Results

The precision and recall of the various tools is shown in Figure 2. The most controlled comparison can be made with respect to minimap2, since the sketch used by ESKEMAP is a subset of the one used by minimap2. At a score threshold corresponding to 70% recovery, ESKEMAP achieves the same precision (0.999) as minimap2. However, the recall of ESKEMAP is 0.88, compared to 0.76 of minimap2. This illustrates the potential of ESKEMAP as a method to recover more of the correct hits than a heuristic method. More generally, ESKEMAP achieves a recall around 90%, while all other tools have a recall of at most 76%. However, both edlib and Winnowmap2 achieve a slightly higher precision (by 0.001).

7.5 Time and Memory Results

We compared the runtimes and memory usage of all sketch-based methods (Table 1). Calculations were performed on a virtual machine with 28 cores and 256 GB of RAM. We did not include edlib in this alignment since, as an exact alignment-based method, it took much longer to complete (i.e. running highly parallelized on many days on a system with many cores). We see that both heuristics are significantly faster than our exact algorithm. However, they also find many fewer mapping positions per read. E.g., only one mapping position is reported for 67% and 75% of all reads by minimap2 and Winnowmap2, respectively. In comparison, ESKEMAP finds more than one mapping position for almost every second read (49%). When the runtime is normalized per output mapping, ESKEMAP is actually more than an order of magnitude faster than the other tools.

The memory usage of ESKEMAP is dominated by the size of S. In particular, the highest value of ℓ was $185,702$ and a matrix with dimensions $\ell \times \ell$ that stores a 4-byte value in the upper diagonal takes 69GB, which corresponds to the peak reported in Table 1. As expected, the memory use depends on the repetitiveness of the text and on the sketching scheme used.

8 Conclusion

In this work, we formally defined the Sketch Read Mapping Problem, i.e. to find all positions inside a reference sketch with a certain minimum similarity to a read sketch under a given similarity score function. We also proposed an exact dynamic programming algorithm called ESKEMAP to solve the problem, running in $\mathcal{O}(|t| + |p| + \ell^2)$ time and $\Theta(\ell^2)$ space. We evaluated ESKEMAP's performance by mapping a simulated long read dataset to the T2T assembly of human chromosome Y and found it to have a superior recall for a similar level of precision compared to minimap2, while offering precision/recall tradeoffs compared with edlib or Winnowmap2.

A clear drawback of ESKEMAP remains its high memory demand for storing the dynamic programming matrix. If many k-mers from a read's sketch occur frequently inside the sketch of the reference sequence, its quadratic dependence on the number of shared k-mers becomes a bottleneck. It may be possible to modify the algorithm to store only the recently calculated column, but that would require a novel way to perform the maximality check of Algorithm 2.

In order to further improve on ESKEMAP's runtime, a strategy could be to develop filters that prune the result's search space. This could be established, e.g., by terminating score calculations for a column once it is clear an optimal solution would not make use of the rest of that column. Our prototype implementation of ESKEMAP would also benefit from additional engineering of the code base, potentially leading to substantial improvements of runtime and memory in practice.

Having an exact sketch-based mapping algorithm at hand also opens the door for the exploration of novel score functions to determine sequence similarity on the level of sketches. Using our algorithm, combinations of different sketching approaches and score functions may be easily tested. Eventually, this may lead to a better understanding of which sketching methods and similarity measures are most efficient considering sequences with certain properties like high repetitiveness or evolutionary distance.

References

1 Can Alkan, Jeffrey M Kidd, Tomas Marques-Bonet, Gozde Aksay, Francesca Antonacci, Fereydoun Hormozdiari, Jacob O Kitzman, Carl Baker, Maika Malig, Onur Mutlu, et al. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature genetics*, 41(10):1061–1067, 2009.

2 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. doi:10.1016/S0022-2836(05)80360-2.

3 Mahdi Belbasi, Antonio Blanca, Robert S Harris, David Koslicki, and Paul Medvedev. The minimizer jaccard estimator is biased and inconsistent. *Bioinformatics*, 38(Supplement_1):i169–i176, June 2022. doi:10.1093/bioinformatics/btac244.

4 Antonio Blanca, Robert S Harris, David Koslicki, and Paul Medvedev. The statistics of k-mers from a sequence undergoing a simple mutation process without spurious matches. *Journal of Computational Biology*, 29(2):155–168, 2022.

5 Monika Cechova, Rahulsimham Vegesna, Marta Tomaszkiewicz, Robert S Harris, Di Chen, Samarth Rangavittal, Paul Medvedev, and Kateryna D Makova. Dynamic evolution of great ape y chromosomes. *Proceedings of the National Academy of Sciences*, 117(42):26273–26280, 2020.

6 Robert Edgar. Syncmers are more sensitive than minimizers for selecting conserved k-mers in biological sequences. *PeerJ*, 9:e10805, February 2021.

7 Mahmudur Rahman Hera, N. Tessa Pierce-Ward, and David Koslicki. Debiasing FracMinHash and deriving confidence intervals for mutation rates across a wide range of evolutionary distances. *bioRxiv*, January 2022.

8 Ting Hon, Kristin Mars, Greg Young, Yu-Chih Tsai, Joseph W Karalius, Jane M Landolin, Nicholas Maurer, David Kudrna, Michael A Hardigan, Cynthia C Steiner, et al. Highly accurate long-read hifi sequencing data for five complex genomes. *Scientific data*, 7(1):399, 2020.

9 Luiz Irber, Phillip T. Brooks, Taylor Reiter, N. Tessa Pierce-Ward, Mahmudur Rahman Hera, David Koslicki, and C. Titus Brown. Lightweight compositional analysis of metagenomes with FracMinHash and minimum metagenome covers. *bioRxiv*, January 2022. doi:10.1101/2022.01.11.475838.

10 Chirag Jain, Arang Rhie, Nancy F Hansen, Sergey Koren, and Adam M Phillippy. Long-read mapping to repetitive reference sequences using Winnowmap2. *Nature Methods*, 19:705–710, 2022.

11 Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

12 Paul Medvedev, Monica Stanciu, and Michael Brudno. Computational methods for discovering structural variation with next-generation sequencing. *Nature Methods*, 6:S13, 2009.

13 Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V Bzikadze, Alla Mikheenko, Mitchell R Vollger, Nicolas Altemose, Lev Uralsky, Ariel Gershman, et al. The complete sequence of a human genome. *Science*, 376(6588):44–53, 2022.

14 Yukiteru Ono, Kiyoshi Asai, and Michiaki Hamada. Pbsim2: a simulator for long-read sequencers with a novel generative model of quality scores. *Bioinformatics*, 37(5):589–595, 2021.

15 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.

16 Kristoffer Sahlin, Thomas Baudeau, Bastien Cazaux, and Camille Marchet. A survey of mapping algorithms in the long-reads era. *Genome Biology*, 24(1):1–23, 2023.

17 Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 22nd International conference on Management of Data (SIGMOD 2003)*, pages 76–85, 2003.

18 Martin Šošić and Mile Šikić. Edlib: A C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 2017.

A Proofs

Proof of Theorem 3. Observe that domination is a transitive property, i.e. if sc_1 dominates sc_2 and sc_2 dominates sc_3, then sc_1 dominates sc_3. To prove equivalence, we will prove the following circular chain of domination: $\mathrm{sc}_\ell \leftarrow \mathrm{sc}_\mathrm{B} \leftarrow \mathrm{sc}_\mathrm{C} \leftarrow \mathrm{sc}_\mathrm{D} \leftarrow \mathrm{sc}_\mathrm{A} \leftarrow \mathrm{sc}_\ell$.

First, observe that sc_B trivially dominates sc_ℓ by keeping the threshold function the same and setting $b_1 = 1$ and $b_2 = w$.

Next, we prove that sc_C dominates sc_B. Let p be a pattern and let $t = \mathrm{thr}_\mathrm{B}(|p|)$. Set $\mathrm{thr}_\mathrm{C} = \mathrm{thr}_\mathrm{B}$ and $c_1 = b_1 + b_2$ and $c_2 = b_2$. Then, for all s, the following series of equivalent transformations proves that sc_C dominates sc_B.

$$\begin{aligned}
\mathrm{sc}_\mathrm{B}(s, p; b_1, b_2) &\geq t \\
\sum_x b_1 x_{\min} - b_2 x_{\mathrm{diff}} &\geq t \\
\sum_x b_1 x_{\min} - b_2(x_{\max} - x_{\min}) &\geq t \\
\sum_x (b_1 + b_2) x_{\min} - b_2 x_{\max} &\geq t \\
\mathrm{sc}_\mathrm{C}(s, p; c_1, c_2) &\geq \mathrm{thr}_\mathrm{C}(|p|)
\end{aligned}$$

Next, we prove that sc_D dominates sc_C. Let p be a pattern and let $t = \mathrm{thr}_\mathrm{C}(|p|)$. Set $d_1 = c_1 + c_2$, $d_2 = c_2$, and $\mathrm{thr}_\mathrm{D}(i) = \mathrm{thr}_\mathrm{C}(i) + ic_2$. Then, for all s, the following series of equivalent transformations proves that sc_D dominates sc_C.

$$\begin{aligned}
\mathrm{sc}_{\mathrm{C}}(s,p;c_1,c_2) &\geq \mathrm{thr}_{\mathrm{C}}(|p|)\\
\sum_x c_1 x_{\min} - c_2 x_{\max} &\geq t\\
\sum_x c_1 x_{\min} - c_2\left(|s|+|p|-\sum_x x_{\min}\right) &\geq t\\
\sum_x (c_1+c_2)x_{\min} - c_2|s| - c_2|p| &\geq t\\
\sum_x (c_1+c_2)x_{\min} - c_2|s| &\geq t + c_2|p|\\
\mathrm{sc}_{\mathrm{D}}(s,p;d_1,d_2) &\geq \mathrm{thr}_{\mathrm{D}}(|p|)
\end{aligned}$$

Next, we prove that sc_{A} dominates sc_{D}. Let p be a pattern and let $t = \mathrm{thr}_{\mathrm{D}}(|p|)$. Set $a_1 = \frac{d_1}{d_2} - 2$ and $\mathrm{thr}_{\mathrm{A}}(i) = \frac{\mathrm{thr}_{\mathrm{D}}(i)}{d_2} - i$. Then, for all s, the following series of equivalent transformations proves that sc_{D} dominates sc_{C}.

$$\begin{aligned}
\mathrm{sc}_{\mathrm{D}}(s,p;d_1,d_2) &\geq \mathrm{thr}_{\mathrm{D}}(|p|)\\
\left(\sum_x d_1 x_{\min}\right) - d_2|s| &\geq t\\
\left(\sum_x d_1 x_{\min}\right) - d_2\left(\sum_x 2x_{\min} + \sum_x x_{\mathrm{diff}} - |p|\right) &\geq t\\
\sum_x ((d_1-2d_2)x_{\min} - d_2 x_{\mathrm{diff}}) + d_2|p| &\geq t\\
\sum_x \left(\frac{d_1-2d_2}{d_2}x_{\min} - x_{\mathrm{diff}}\right) + |p| &\geq \frac{t}{d_2}\\
\sum_x \left((\frac{d_1}{d_2}-2)x_{\min} - x_{\mathrm{diff}}\right) &\geq \frac{t}{d_2} - |p|\\
\mathrm{sc}_{\mathrm{A}}(s,p;a_1) &\geq \mathrm{thr}_{\mathrm{A}}(|p|)
\end{aligned}$$

Finally, we prove that sc_{ℓ} dominates sc_{A}. Let p be a pattern and let $t = \mathrm{thr}_{\mathrm{A}}(|p|)$. Set $w = \frac{1}{a_1}$ and $\mathrm{thr}_{\ell}(i) = \frac{\mathrm{thr}_{\mathrm{A}}(i)}{a_1}$. Then, for all s, the following series of equivalent transformations proves that sc_{ℓ} dominates sc_{A}.

$$\begin{aligned}
\mathrm{sc}_{\mathrm{A}}(s,p;a_1) &\geq \mathrm{thr}_{\mathrm{A}}(|p|)\\
\sum_x (a_1 x_{\min} - x_{\mathrm{diff}}) &\geq t\\
\sum_x (x_{\min} - \frac{1}{a_1}x_{\mathrm{diff}}) &\geq \frac{t}{a_1}\\
\mathrm{sc}_{\ell}(s,p;w) &\geq \mathrm{thr}_{\ell}(|p|)
\end{aligned}$$

◀

Fractional Hitting Sets for Efficient and Lightweight Genomic Data Sketching

Timothé Rouzé
Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Igor Martayan
ENS Rennes, Univ. Rennes, France

Camille Marchet
Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Antoine Limasset[1]
Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Abstract

The exponential increase in publicly available sequencing data and genomic resources necessitates the development of highly efficient methods for data processing and analysis. Locality-sensitive hashing techniques have successfully transformed large datasets into smaller, more manageable sketches while maintaining comparability using metrics such as Jaccard and containment indices. However, fixed-size sketches encounter difficulties when applied to divergent datasets.

Scalable sketching methods, such as Sourmash, provide valuable solutions but still lack resource-efficient, tailored indexing. Our objective is to create lighter sketches with comparable results while enhancing efficiency. We introduce the concept of Fractional Hitting Sets, a generalization of Universal Hitting Sets, which uniformly cover a specified fraction of the k-mer space. In theory and practice, we demonstrate the feasibility of achieving such coverage with simple but highly efficient schemes.

By encoding the covered k-mers as super-k-mers, we provide a space-efficient exact representation that also enables optimized comparisons. Our novel tool, SuperSampler, implements this scheme, and experimental results with real bacterial collections closely match our theoretical findings.

In comparison to Sourmash, SuperSampler achieves similar outcomes while utilizing an order of magnitude less space and memory and operating several times faster. This highlights the potential of our approach in addressing the challenges presented by the ever-expanding landscape of genomic data.

2012 ACM Subject Classification Applied computing → Bioinformatics

Keywords and phrases k-mer, subsampling, sketching, Jaccard, containment, metagenomics

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.15

Supplementary Material
Software (Source Code): `https://github.com/TimRouze/supersampler`
archived at `swh:1:dir:fc533a18a688ebd041d716dfe50b5fa9fbbc9be7`
Dataset: `https://github.com/TimRouze/Expe_SPSP`

1 Introduction

The field of genomics has exploded in recent years, driven by the availability of cheap and easy sequencing data generation. The Sequence Read Archive (SRA) is a vast and to some extent under-exploited goldmine of genomic data, containing an enormous amount of genetic information. However, one of the biggest challenges in utilizing this data is the lack of

[1] corresponding author

efficient indexing and querying tools. The GenBank database, for example, already contains 1.2 million bacterial genomes, totaling over 5 Terabases of data. In the face of such vast genetic information, a crucial need is to promptly and precisely determine the most similar (or contained) known entry for a given query document (assembled or unassembled reads). Specifically, in this work we focus on the metagenomic assessment problem, which entails characterizing microbial communities in a specific environment using DNA sequencing data and potentially large amounts of reference entries. The complexity and diversity of the data, which contains sequences from multiple genomes, presents significant challenges.

In the metagenomic context, traditional alignment-based methods such as BLAST are increasingly computationally prohibitive due to the sheer number of potential targets for metagenome mapping. A spectrum of alignment-free techniques based on k-mer content have emerged as a viable alternative, with different tradeoffs. On one side of the spectrum, exact k-mer indexing offers linear query time [9, 14, 21], but may be too memory-intensive for large-scale applications. Probabilistic structures, when applied to large queries (on the order of kilobases), enable more scalable indexing at the expense of a random false positive rate [15, 5].
Adequate query length allows for handling false positive noise since it remains significantly lower than the required matching signal (e.g., 70% of queried k-mers present in the document). For extensive queries at the Megabase level, the signal strength is sufficiently robust, eliminating the need to consider all k-mers and enabling sublinear query time. On the other side of the spectrum, fixed-size sketches like Minhash [6], Hyperloglog [16], and Hyperminhash [29] have been effectively used for large-scale collection comparison [18, 2, 3, 30, 1]. However, they are ill-suited for divergent documents in terms of content or size, a critical limitation considering metagenomic samples typically comprise many organisms with amount of distinct k-mers varying by orders of magnitude.
Scaled sketches, whose size scales linearly with input size, have demonstrated better resilience to such issues. Sourmash [23], which implements scaledminhash [23] and Fracminhash [10], efficiently approximates containment and Jaccard indexes even for documents with size disparities spanning several orders of magnitude. Sourmash's simplicity is one of its key strengths: it stores each uniformly selected k-mer as a 32-bit integer and compares them using a dictionary. An observation is that this technique is generic and can be applied to any type of data. Therefore, computational and memory requirements could benefit from customized selection techniques and index structures.

To this end, we propose capitalizing on the ability to represent overlapping k-mers with a low number of bits per k-mer using a *Spectrum Preserving String Set* [25]. The challenge we face is optimizing the overlap of chosen k-mers to achieve maximum space efficiency. To address this, we build upon the concept of super-k-mers [12], which are sequences of k-mers sharing a common selected m-mer called minimizer [26]. Universal Hitting Sets (UHS [19]) methods aim to design optimized m-mer selection schemes that cover all k-mers while minimizing the density of selected positions. However, our application does not require complete coverage of the k-mer space. Therefore, we introduce *Fractional Hitting Sets* that encompass a uniformly selected fraction of the k-mer space. We conduct a study on the achievable density in relation to the selected fraction and present a straightforward minimizer selection scheme that closely approaches optimal bounds. We implement this scheme in a tool called SuperSampler. The storage of enhanced super-k-mer sequences, partitioned by minimizers, facilitates space and time-efficient k-mer set comparisons. Our evaluation reveals that SuperSampler significantly reduces resource usage compared to Sourmash while maintaining similar results. Overall, this work presents a promising approach to making large-scale genomic data more accessible and manageable.

2 Preliminaries

This paper presents results on finite strings on the DNA alphabet $\Sigma = \{A, C, G, T\}$, we use σ to denote the size of the alphabet. We consider two input multisets of strings longer or equal to k, S_A and S_B. These multisets can in practice be read sets from sequencing experiments or genome sequences. We call k-mers strings of size k over strings of the input sets. $A = \{x_0, x_1, \ldots, x_{n-1}\}$ is the set of distinct k-mers from S_A and $B = \{y_0, y_1, \ldots, y_{n-1}\}$ is defined similarly for S_B. Our goal is to estimate the similarity between those two sets. Two metrics are mainly used: Jaccard index and the containment index.

Jaccard Index, containment index and estimators

For two finite, non empty sets of k-mers A and B, the Jaccard index [6] gives a measure of the similarity A and B by comparing the relative size of the $A \cap B$ intersection over the $A \cup B$ union,

$$0 \leq J_{A,B} = \frac{|A \cap B|}{|A \cup B|} \leq 1.$$

The containment index C of the previously defined sets A and B measures the relative size of $A \cap B$ intersection over the size of A, i.e., the proportion of distinct k-mers of A that are present in B,

$$0 \leq C_{A,B} = \frac{|A \cap B|}{|A|} \leq 1.$$

In practice, comparing extremely large sets can be resource-intensive, whereas estimating their similarity can be accomplished much more efficiently. MinHash is a locality-sensitive hashing technique that represents each set with a list of hash values of fixed size. The fraction of shared hash values between two lists provides an estimation of the Jaccard index for the corresponding sets, allowing for efficient similarity measurement.

▶ **Property 1** (error in MinHash's Jaccard estimator). *The error bound E of the MinHash Jaccard estimate $\tilde{J}_{A,B}$ only relies on the sketch size s [24]*

$$E = \mathcal{O}\left(\frac{1}{\sqrt{s}}\right).$$

However, the relative error, $E/J_{A,B}$, increases significantly for dissimilar documents, as maintaining the error for a decreasing Jaccard index requires a quadratic compensation in the sketch size (refer to additional Figure S1 of [18]).

Such a property makes fixed-size sketches ill-suited for comparing dissimilar contents, in particular in a resource-frugal context since the sketch size to reach an acceptable precision would be too high to handle.

Happily, scaled sketches that grow linearly with the input size do not present such drawback. Sourmash is the leading tool that implements such scheme and provides sketches composed of k-mers represented as 32 bits hashes. To do so, k-mers are selected if their hash is below a specific threshold calculated in regard to the desired subsampling rate. Our goal is to improve the space efficiency of such sketch using efficient k-mer encoding dubbed SPSS [25] exploiting the fact that k-mers share overlaps. The most convenient yet relatively space efficient SPSS are super-k-mers and our work relies on such efficient k-mer representation using the notion of minimizers.

Minimizers and super-k-mers

▶ **Definition 2** (minimizer). *Given a total order $\mathcal{O}$ and a k-mer u, the minimizer of u is the smallest m-mer of u according to $\mathcal{O}$. The method to select minimizers is referred to as a minimizer scheme.*

In practice, $\mathcal{O}$ is defined on integers by hashing m-mers using a random hash function h and selecting the smallest hash value. We assume that the hash function is chosen such that the hashes are independent and uniformly distributed. From now on, we use $w = k - m + 1$ to denote the number of m-mers inside a k-mer.

▶ **Definition 3** (super-k-mer). *A super-k-mer is a maximal substring of a string s ($|s| \geq k$) in which each consecutive k-mers have the same minimizer.*

Super-k-mers are an interesting SPSS because a super-k-mer containing x k-mers is composed of $k + x - 1$ bases which incur a cost of $\frac{2(k+x-1)}{x}$ bits per k-mer. Therefore longer super-k-mers provide lighter k-mer representation.

By omitting repeated minimizers inside k-mers, the first k-mer of the longest possible super-k-mers have their minimizers as a suffix. Equally, the last k-mers of these super-k-mers have their minimizers as a prefix.

▶ **Definition 4** (maximal super-k-mer). *Let s ($|s| \geq k$) be a string and v a super-k-mer of s. Let i_m be the first position of the minimizer on s. v is a maximal super-k-mer iff v starts at position $i_m + m - k$ in s and v ends at position $i_m + k - 1$ in s. It follows that v has a length of $2k - m$ and contains $w = k - m + 1$ k-mers.*

Examples of regular and maximal super-k-mers are shown in Figure 1. Since maximal super-k-mer are the most space-efficient, our approach aims to rely on long super-k-mers (ideally maximal) in order to have a compact encoding of the k-mer sketch.

Figure 1 Super-k-mers extracted from a sequence for $k = 6, m = 3$. Minimizers are shown in purple, here we use the lexicographic order instead of hashing minimizers for the sake of the simplicity. Super-k-mers 1 and 4 are maximal (they contain respectively k-mers $\{CTGAAA, TGAAAT, GAAATG, AAATGC\}$ and $\{TGCACA, GCACAT, CACATT, ACATTT\}$, while 2 and 3 are not (and contain respectively k-mers $\{AATGCA\}$ and $\{ATGCAC\}$.

▶ **Property 5** (coverage of maximal super-k-mers). *In [22], authors show that maximal super-k-mers cover approximately half of the k-mers of an input string (Proof of Theorem 3) for a sufficiently large m.*

Density and universal hitting sets

Since every minimizer corresponds to one super-k-mer, the proportion of m-mers chosen as minimizers is exactly the inverse of the average length of super-k-mers. This proportion, which quantifies the sparsity of a minimizer scheme, is referred to as the *density*.

▶ **Definition 6** (density of a minimizer scheme). *The density of a minimizer scheme is the expected number of selected minimizers divided by the total number of m-mers. The density factor is equal to the density multiplied by $w + 1$.*

The density of a minimizer scheme is lower bounded by $1/w$ since each k-mer contains w m-mers and one of them must be selected as a minimizer. It is known that the expected density of a minimizer scheme based on a random ordering is $2/(w + 1)$, and that minimizer schemes cannot have a density below $1.5/(w + 1)$ [28]. More generally, m-mers selection scheme able to cover every k-mer are called *universal hitting sets*.

▶ **Definition 7** (universal hitting set or UHS). *A set $\mathcal{U} \subseteq \Sigma^m$ is defined as a Universal Hitting Set (UHS) if it includes a minimum of one element from every sequence of w consecutive m-mers.*

Note in particular that the set of all minimizers of Σ^k forms a UHS. Recent publications introduced different methods based on UHS to bring the density below $2/(w + 1)$ and closer to the $1.5/(w + 1)$ barrier [31, 20]. Thus, a question that naturally arises is: can we cross this barrier by relaxing some constraints on the selection scheme?

3 Fractional Hitting Sets

In this section, we introduce the concept of *fractional hitting sets*, which are a generalization of universal hitting sets. These sets are designed to cover a fraction f of the k-mer space.

▶ **Definition 8** (fractional hitting set or FHS). *A set $\mathcal{F} \subseteq \Sigma^m$ is a Fractional Hitting Set (FHS) if it contains at least one element from a fraction at least f of the sequences of w consecutive m-mers.*

To avoid selection bias in practice, we aim to ensure that k-mers are selected randomly, using a random seed as a basis, and that every k-mer has an equal chance of being chosen.

We introduce a simple way to build such fractional hitting sets by selecting minimizers with a hash smaller than a certain threshold. We call such selected minimizers *small minimizers*. Note that any method selecting a fraction of the minimizers hashes would be suitable here.

▶ **Definition 9** (small m-mer). *Given a fixed threshold $t \in [\![0, \sigma^m]\!]$, we say that a m-mer is small if its hash is below t. We denote by $\mathcal{S}$ the set of small m-mers, and $p = \frac{t}{\sigma^m}$ the probability that a m-mer is small.*

From p we can derive the proportion of covered k-mers.

▶ **Property 10.** *The fraction of k-mers with distinct m-mers containing a small m-mer is*

$$f = 1 - (1 - p)^w$$

where $w = k - m + 1$ and p is the probability that a m-mer is small.

Proof. Given a k-mer with w distinct m-mers $x_1, \ldots, x_w$,

$$\mathbb{P}(\forall i \in [\![1, w]\!], h(x_i) \geq t) = \prod_{i=1}^{w} \mathbb{P}(h(x_i) \geq t) = (1 - p)^w$$

because the hashes of distinct m-mers are independent, so

$$f = \mathbb{P}\left(\min_{i \in [\![1,w]\!]} h(x_i) < t\right) = 1 - (1 - p)^w. \qquad \blacktriangleleft$$

Conversely, if we want a given fraction f of the k-mers to be covered, the threshold should be chosen as

$$t = \left[1 - (1-f)^{1/w}\right] \cdot \sigma^m. \tag{1}$$

Density of small minimizers

We showed that selecting k-mers with a small minimizer induces an FHS. By considering the usual definition of density (from Definition 6) for this scheme (which cover a fraction f of all k-mers), we obtain the following bound (proven in supplementary materials):

▶ **Theorem 11.** *Assuming $m > (3+\varepsilon)\log_\sigma w$, the expected density of small minimizers in a random sequence is upper bounded by*

$$\frac{2f}{w+1} + o(1/w).$$

At first glance, the results may be surprising, as the density is smaller than the lower bound of $1/w$ for $f < 1/2$ and can approach zero. This is because some k-mers may not contain any small minimizers and are therefore not covered, and the proportion of such k-mers increases as f approaches 0. However, it's worth noting that this bound does match the $2/(w+1)$ density when $f = 1$ (i.e., when every k-mer is covered).

To obtain a more meaningful metric, we can compute the density on the fraction of the sequence that is covered, instead of the entire sequence. With this approach, we obtain the following theorem, which has been proven in the supplementary materials:

▶ **Theorem 12** (restricted density)**.** *If we restrict to sequences in which every k-mer contains a small minimizer, assuming $m > (3+\varepsilon)\log_\sigma w$, the expected density of small minimizers is upper bounded by*

$$2 \cdot \frac{f + (1-f)\ln(1-f)}{f^2(w+1)} + o(1/w).$$

Although less intuitive than the previous one, this result provides valuable insights into the density within the covered portion of the sequence. As shown in Figure 5 (see supplementary materials), the associated density factor ranges from 2 when $f = 1$ (consistent with existing results) to 1 when $f = 0$. Therefore, as f approaches 0, we can approach the optimal density.

Maximal super-k-mers

Although measuring the density provides an overview of the average length of super-k-mers, it doesn't indicate how many of them are maximal (i.e., of length $2k - m$). The following result (proven in the supplementary materials) answers this question:

▶ **Theorem 13.** *In a random sequence, the average proportion of maximal super-k-mers (with respect to all super-k-mers) built from small minimizers is given by*

$$\left[\left(1 - \frac{1}{w}\right)\frac{f}{1+f}\right]^2 + \frac{1 - f(1-2/w)}{1+f}.$$

Note that this result generalizes theorem 4 from [22], which corresponds to $f = 1$. As shown in figure 12b (see supplementary materials), the proportion increases towards 100% as f approaches 0.

Improving the density of fractional hitting sets using UHS

This effect is more pronounced for smaller values of f. This observation raises a natural question: what is the lowest achievable density for a given f? Since universal hitting sets (UHS) with a density lower than 2 have been proposed for $f = 1$, it is possible that they may also improve the density for smaller f values by considering only the m-mers selected by the UHS as potential minimizers.

▶ **Theorem 14.** *Given a UHS $\mathcal{U}$ with density $d_{\mathcal{U}}$, assuming $m > (3+\varepsilon)\log_\sigma w$, the expected density of small minimizers selected from $\mathcal{U}$ (that is, $\mathcal{S} \cap \mathcal{U}$) in a random sequence is upper bounded by*

$$f \cdot d_{\mathcal{U}} + o(1/w).$$

The proof is given in supplementary materials. Note that this result generalizes theorem 11 since the UHS of minimizers selected using a random ordering has a density of $2/(w+1)$ [28].

4 SuperSampler

4.1 Related work

Sketching

Sketching strategies involve building a summary, or "sketch", of an input dataset that is smaller than the original set. The purpose of this sketch is to approximate the similarity between two documents based on the similarity between their respective sketches. In our context, sketches are lists or sets of fingerprints that represent uniformly selected k-mers from the set.

Fixed-size sketches

MinHash is a seminal method for fixed-size sketch-based set comparison. In our context, a MinHash sketch is a list of k-mer hashes that are minimal in some sense. As these k-mers are selected uniformly, the proportion of shared hashes between two sketches approximates the Jaccard index of their original documents. Tools such as `MASH` [18] employ this principle to efficiently compare large collections. However, two main limitations arise from this strategy. First, hash collisions need to be addressed, as they introduce more resemblance between two compared sets than what actually exists. Therefore, the fingerprint size must be chosen appropriately to control such collisions. Recent approaches like [30], `Dashing` [2], `Dashing2` [3] and `NIQKI` [1] used enhanced fingerprint scheme related to Hyperloglog [7] or to HyperMinHash [29] to get the best possible precision memory trade-off. Second, as stated in Property 1, this approach is best suited for comparing sets of similar size and content. When comparing sets with significant size or content differences, the accuracy of the estimate decreases substantially.

Scaled sketches

As the problem of inaccuracy of fixed-size sketches for dissimilar datasets was pointed out but only partially solved using MinHash schemes [17, 13], novel approaches were proposed. Scaled sketches methods build their sketches with adapted sizes in regard to the input. The main contribution in this direction for compositional analysis is `sourmash` [23], and our paper also falls in this category. `sourmash` incorporates a scaled factor indicating the fraction of

the k-mer's input that should be included in the sketch as fingerprints. To do so they add a fingerprint in the sketch if their integer value is a multiple of S (ScaledMinhash) or lower than a fixed threshold (FracMinHash). The latter technique is recently described and theoretically studied in [10, 8]. In contrast to previously presented fixed-size sketching approaches, those approaches have the advantage of providing a more accurate comparison of uneven sized sets. This flexibility comes at the cost of possibly large sketches. Schemes based on minimizers or other seeds are also used for read mapping [11, 27] but are out of the scope of this paper.

4.2 SuperSampler's sketch construction

▶ **Definition 15** (SuperSampler's sketch)**.** *Given a sequence S, each super-k-mer whose minimizer's hash is lower or equal to a threshold T is selected, and all its surrounding k-mers are kept in the sketch as a super-k-mer. A supersampler sketch can therefore be represented as a super-k-mer set.*

In Sourmash, k-mers are represented as integer fingerprints through hashing (using a default of 32-bit). This approach reduces space requirements compared to employing $2k$ bits per k-mer, but it also introduces the potential for false positives due to hash collisions. However, the false positive rate is exponentially low, depending on the fingerprint size, which allows for efficient control.

In contrast, SuperSampler explicitly stores k-mers as super-k-mers, offering two significant advantages over the conventional method: First, the selected k-mers are represented exactly, eliminating false matches and enabling the output of shared k-mers when they are of interest to users. Second, this technique enables a more space-efficient representation of k-mers, typically requiring less than the 32-bit per k-mer space cost of Sourmash. Figure 2 illustrates an example of sketch construction in SuperSampler. In the following sections, we propose a model to evaluate the space efficiency of a SuperSampler sketch.

In the regular case with hashed minimizer with a density factor of 2 we can expect $(k-m+1)/2$ k-mers per super-k-mers [22]. This results in a mean super-k-mer length of $(3k-m-1)/2$ bases. We can give a lower bound of the cost in bit per k-mer to encode such super-k-mers:

$$\frac{2(3k-m-1)}{k-m+1}.$$

However we need to encode the length of each super-k-mer to avoid considering artefactual k-mers created by two successive super-k-mers so we can add $\log_2(k-m+1)$ bits per super-k-mer (encoding the number of k-mers). This leads to a bits per k-mer ratio of

$$2 \cdot \frac{3k-m-1+\log_2(k-m+1)}{k-m+1}.$$

In practice we use the formula (1) to select a k-mer fraction chosen by the user. As a side effect, since we used an FHS, selected super-k-mers are longer than those selected by regular hashed minimizer scheme as our hitting set provides a lower density. Importantly for low selected fraction a very large proportion of super-k-mers are maximal. This property is of prime interest because maximal super-k-mers can be efficiently encoded for two reasons. First they are all of the same size so we do not need to encode their respective length or any kind of separator. Second, they represent $2k-m$ bases encoding for $k-m+1$ k-mers, they provide a lower bits per k-mers ratio

$$\frac{2(2k-m)}{k-m+1}.$$

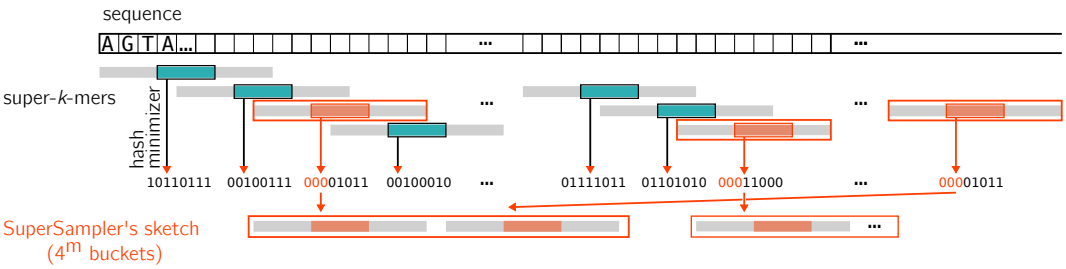


Figure 2 SuperSampler's sketching strategy. In order to build sketches, SuperSampler computes super-k-mers over the input sequence. Fingerprints are associated with each super-k-mer by hashing their minimizers to an integer, hence an integer per super-k-mer. Super-k-mers associated to sufficiently low integers are kept in the sketch. Super-k-mers are put into buckets according to their minimizer.

4.3 Partitioned sketches

Minimizers naturally partition the super-k-mer space into $\mathcal{O}(4^m)$ buckets. Since only a subset of the minimizer space is selected, a smaller number of buckets are actually considered. SuperSampler relies on the fact that super-k-mers are centered around a shared minimizer to build a partitioned sketch. In practice, we examine all non-empty selected buckets, each storing a minimizer with their corresponding super-k-mers independently. This strategy offers several crucial advantages. First, when encoding maximal super-k-mers, we know the position of the minimizer within each super-k-mer. By storing the minimizer sequence once in the bucket, we can omit it in all maximal super-k-mers. This results in an even lower bit per k-mer ratio:

$$\frac{4(k-m)}{k-m+1}.$$

Figure 3 shows the space cost of the different encoding: super-k-mer, maximal super-k-mer and partitioned super-k-mer according the the minimizer size along with the actual performances of Supersampler. While this mechanism enhances space efficiency, it also significantly improves actual memory usage. When comparing document sketches, matching k-mers between documents are necessarily found in the same partition, so only a given partition is needed in memory at a time. For sufficiently large m, such partitions should be orders of magnitude smaller than the total amount of selected k-mers, as the expected partition size decreases exponentially with m. This partitioning technique also allows for substantial speed-ups, notably in sketch comparison time, which are discussed later on.

4.4 Sketches comparisons

Sketches comparisons

Unlike `sourmash` that treats sketches in their entirety, SuperSampler focuses on small related partitions. This allows for two distinct computational improvements. First, a partition that is specific to a file can be skipped as we know that no k-mer present in such partition will be found in another file so no matching k-mer can be found. Second, the size of the buckets stored in memory being small, we expect few cache-misses when comparing partitions unlike `sourmash` for which a cache-miss for each queried fingerprint can be expected. In other words, as illustrated in Figure 4, SuperSampler concentrates exclusively on small, relevant partitions and processes each of them only once. The efficiency benefits of this

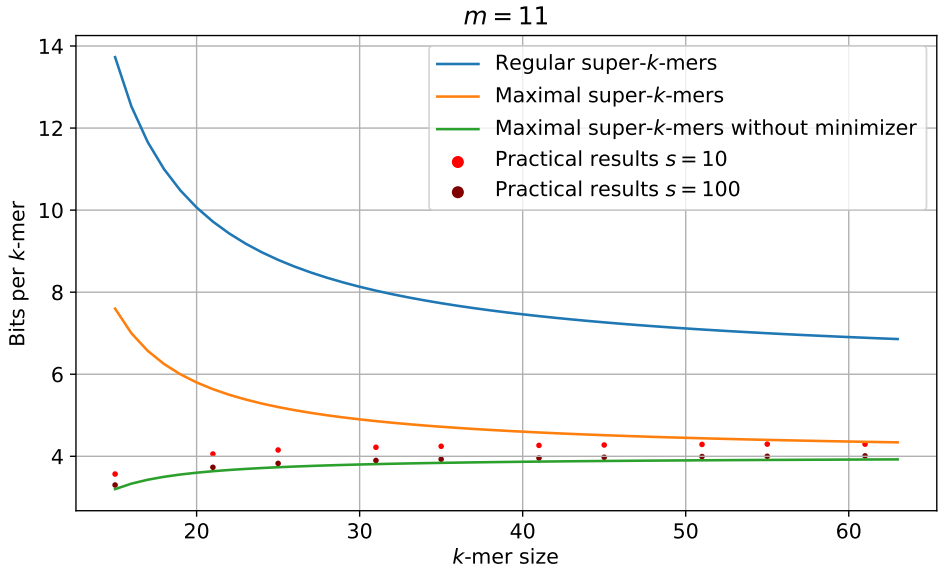


Figure 3 Theoretical space cost of different encodings in bits per k-mer according the k-mer size size along with practical space usage of super-sampler sketches on random sequences.

approach are magnified when comparing one or multiple documents against a large collection, as SuperSampler processes only a specific partition of the relevant documents at a given time. This targeted processing reduces the computational load and enhances the overall performance of the comparison.

5 Results

All experiments were performed on a single cluster node running with Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz and 2 64GiB DIMM DDR4 Synchronous 2666 MHz ram.

In the first section, we evaluate supersampler sketches space usage. In the second section, we evaluate the precision and performance of SuperSampler in comparison to Sourmash, the current state-of-the-art solution. Finally, in the last section, we demonstrate SuperSampler's scalability when indexing extensive collections. Every file used to perform these experiments can be found GitHub : `github.com/TimRouze/Expe_SPSP`.

5.1 Space efficiency of supersampler

As previously discussed, the lower bound of memory cost for storing k-mers as super-k-mers is given by

$$2 \cdot \frac{3k - m - 1 + \log_2(k - m + 1)}{k - m + 1}$$

bits per k-mer , assuming we store the size of each super-k-mer in the index. For $m = 15$, this bound equates to 9.2 bits/k-mer with $k = 31$ and 7.1 bits/k-mer with $k = 63$. However, in practice, SuperSampler exhibits lower space usage. Figure 13 in the supplementary materials reveals that approximately 6.5 bits/k-mer and 5 bits/k-mer are achieved for $k = 31$ and 63, respectively.

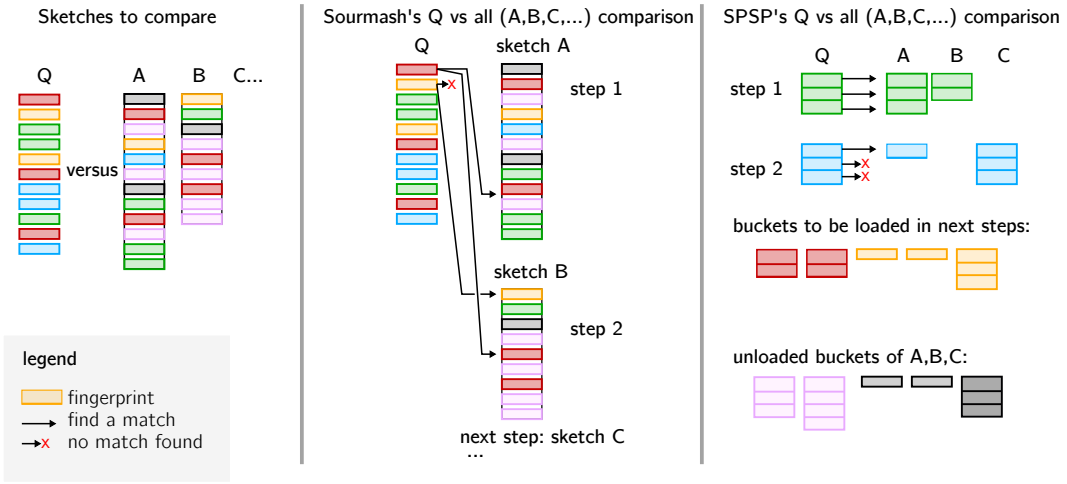


Figure 4 How SuperSampler and Sourmash perform their respective sketch comparison. In this example, we discuss the comparison of one document against a collection, although other use cases can be inferred. SuperSampler is capable of skipping certain buckets that are not relevant to the query. By focusing on smaller sub-parts of the collection one at a time, SuperSampler effectively improves practical performance and reduces memory usage.

These results can be attributed to the low density permitted by SuperSampler's minimizer selection scheme. As illustrated in Figure 5, the density factor quickly diminishes as the subsampling rate increases, respectivey as the fraction f diminishes. When the subsampling rate is 2, the density factor falls below 1.5, the lower bound of the minimizer scheme, and continues to decline toward 1. In general, subsampling tools seldom apply rates below 100, with Sourmash defaulting to a rate of 1000 i.e. $f = 1/1000$. Consequently, SuperSampler consistently remains close to the lower bound for the density factor, since the density factor for a subsampling rate of 100 is already below 1.04. This facilitates the indexing of longer super-k-mers, which are stored more efficiently as their length increases.

To further reduce memory costs, SuperSampler offers an option to use its selection scheme in conjunction with existing UHS-based minimizer schemes, specifically the modified double decycling sets introduced in [20]. As depicted in Figure 6, this approach marginally improves the bit/k-mer cost, particularly for smaller values of k.

A high proportion of maximal super-k-mers in a sketch is advantageous for SuperSampler, as it lowers the bit/k-mer cost. Figure 7 demonstrates that the percentage of maximal super-k-mers increases rapidly with the subsampling rate, reaching 90%, 99%, and 99.9% of indexed super-k-mers when the subsampling rate is around 10, 100, and 1000, respectively. This feature is particularly significant because it enables a rapid and considerable reduction in the bit/k-mer cost by efficiently encoding maximal super-k-mers. Therefore, with the subsampling rates commonly used in practice, which involve a very high proportion of maximal super-k-mers, the actual bound is determined by the following formula

$$\frac{2(2k-m)}{k-m+1}.$$

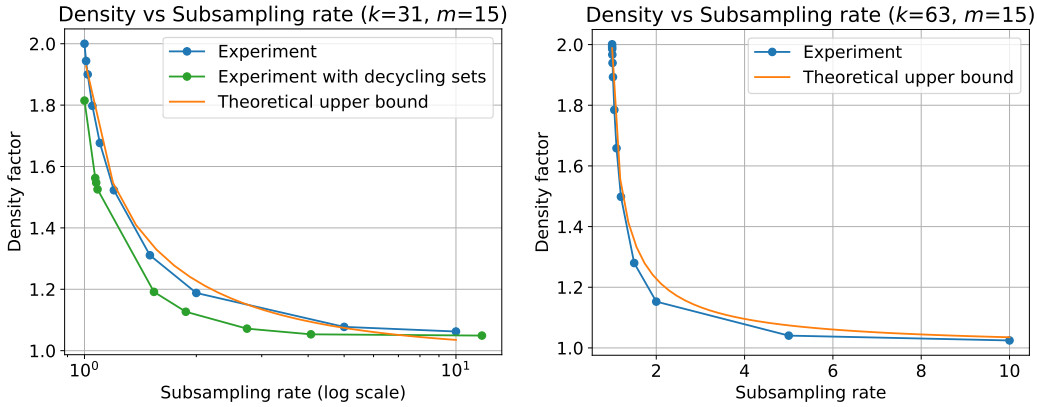


Figure 5 Measured density factor compared to the model.

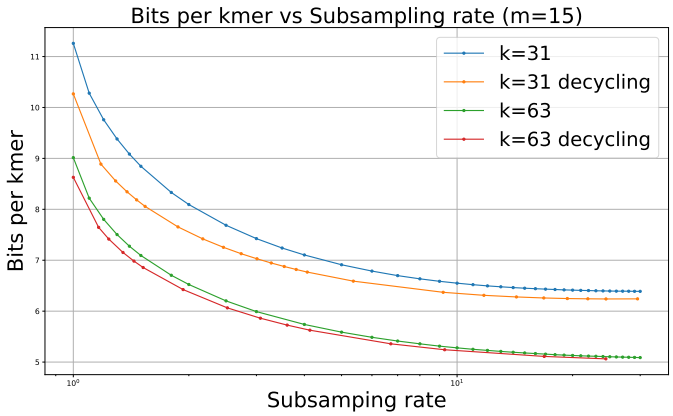


Figure 6 Space cost in bits per k-mer according to the subsampling rate with and without using the improved UHS.

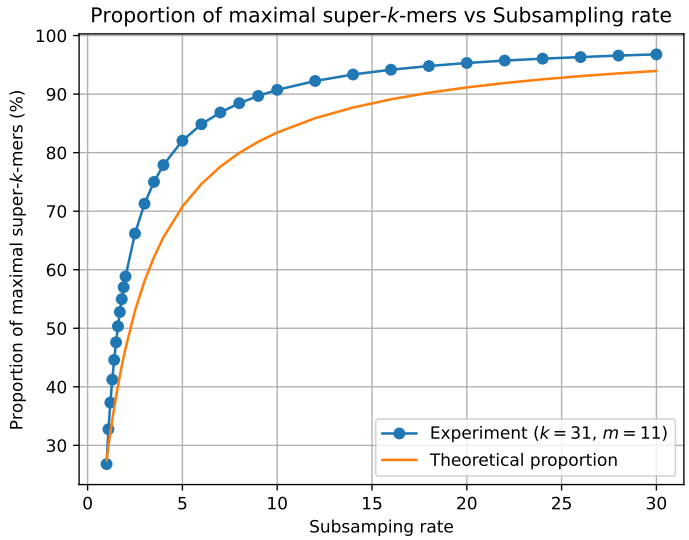


Figure 7 Fraction of maximal super-k-mers according to the subsampling rate.

5.2 Performance Comparison

In our qualitative experiment, we compared the performance of SuperSampler with Sourmash, which implements FracMinHash. We evaluated both tools on two distinct datasets: 1024 Salmonella genomes from GenBank and 1024 bacterial genomes from RefSeq. These collections were chosen due to their differing containment indexes; Salmonella genomes are highly similar to each other, while the bacterial genomes in RefSeq exhibit much greater dissimilarity (Jaccard index close to 0).

We carried out an all-versus-all comparison of these collections using both tools and monitored RAM and disk usage, as well as computation time during the sketch comparisons. To assess the precision of the approximated scores, we calculated the exact Jaccard and containment index values using Simka [4], which performs efficient k-mer counting operations on large collections. With these scores as a reference, the precision of the approximation can be evaluated.

RAM and computation time were measured using the benchmark flag from Snakemake, with one run per command. Disk usage was determined by comparing the sketch sizes of Sourmash (using the *zip* option for *sourmash sketch*) and SuperSampler, with the latter's sketch sizes examined through a Python script. SuperSampler sketches were stored in a tar archive and compressed using *gzip -9*.

5.2.1 RAM, time, and sketch size

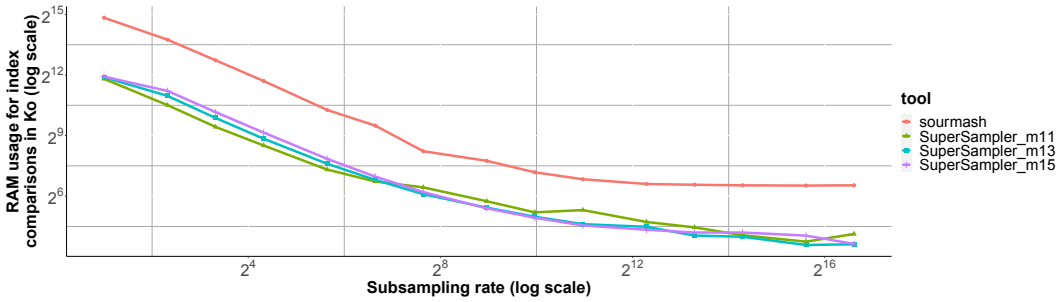
Figures 14 and 8 demonstrate that, for $k = 31$, SuperSampler generally consumes 5 times less RAM and requires generally 16 times less space than Sourmash. Additionally, SuperSampler performs computations 50 times faster than Sourmash when comparing highly dissimilar genomes. However, when genomes are very similar, such as with Salmonella, comparison times are comparable since SuperSampler's time optimization does not apply on very similar documents. We also note that minimizer size has little to no impact on these metrics.

Figures 15 and 16 reveal that the improvement in sketch disk size is even more significant with larger values of k. SuperSampler uses in general 50 times less disk space than Sourmash with $k = 63$, while maintaining similar differences in RAM and computation time.

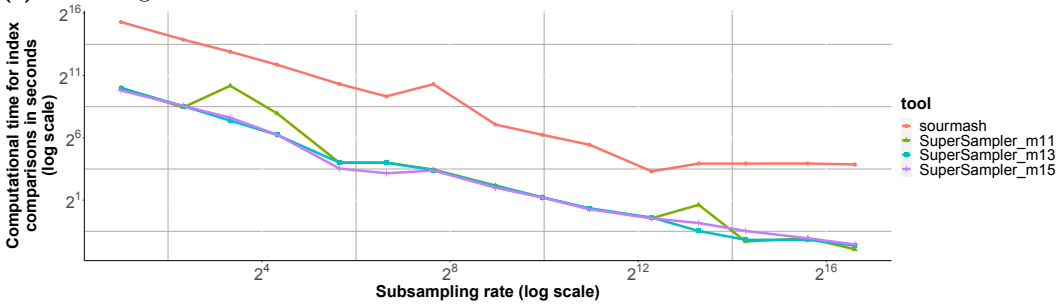
5.2.2 Error

As demonstrated in Figure 9, SuperSampler's performance is similar to Sourmash in terms of error, although it exhibits slightly lower accuracy. This reduced accuracy results from a clustering effect caused by SuperSampler selecting overlapping k-mers around small minimizers. This effect is counterbalanced by SuperSampler's ability to index and compare more k-mers using the same amount of memory and generally less computation time. Figure 9 shows strong variability in error for both Sourmash and SuperSampler, it is to be noted that the error is constantly small and that small variability on already small number leads to such fuzzy figures.

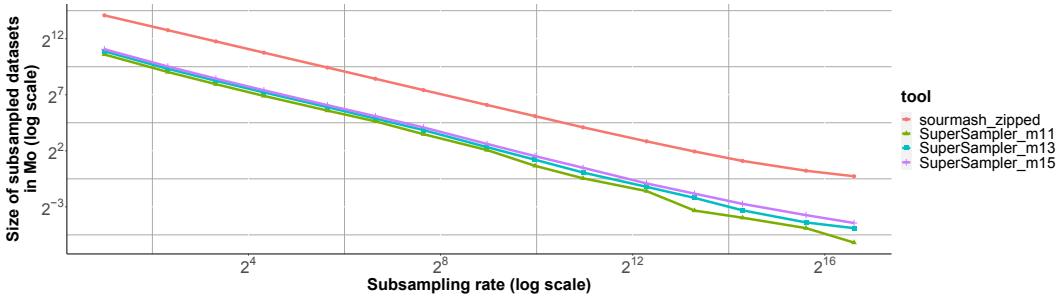
Additionally, SuperSampler stores k-mers in plain text without any loss of information, which means that k-mers of interest can actually be retrieved. While Sourmash could rely on invertible hash functions and larger hashes to match this ability, doing so would effectively increase their space usage.



(a) RAM usage.



(b) Computational time.



(c) Disk usage.

Figure 8 Resource consumption results for 1024 bacterial genomes from RefSeq. For these results, k = 31. For results on Salmonella genomes and k = 63, see figures 14, 16 and 15 on the appendix.

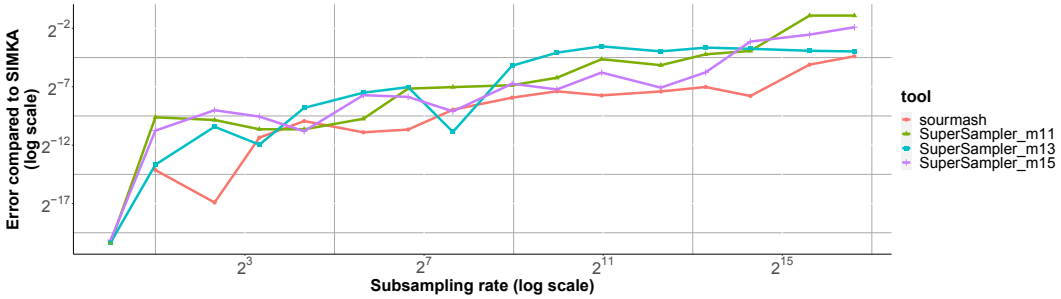


Figure 9 Error against Simka on Containment index approximation for Sourmash (red line) and SuperSampler with different minimizer sizes. This plot is for 1024 Salmonellas genoms with $k = 63$. Other results for RefSeq and Salmonellas are available at figure 17. Jaccard index error is available at figure 18 on the appendix.

5.2.3 Massive collection indexing

As a scalability experiment, SuperSampler and Sourmash were monitored on their performances while analysing growing collection of RefSeq bacterial genomes. Result are displayed in Figure 10. We can see that our tool is able to handle all versus all comparison very large scale collection comparison effectively. For the biggest amount of files, SuperSampler took 25 CPU hours. We observe that the gap between Sourmash and Supersampler is diminishing for larger collection as the output matrix itself become large and generate cache-miss for every update. The sketch creation step is essentially cheap as both tools only took a couple CPU hours the actual bottleneck being IO.

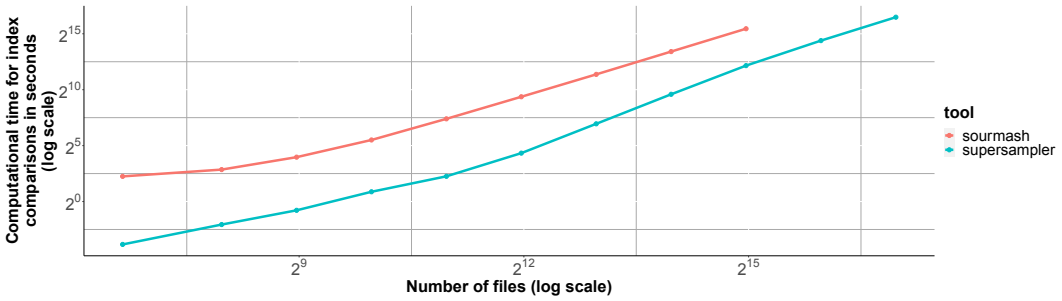


Figure 10 Computational time for comparisons on different amounts of bacterial genomes from RefSeq. From 100 to 128,000 genomes with $k = 63$, $s = 1000$ and $m = 15$ Sourmash was run up to 32,000 genomes as it was taking too much time for the 2 last experiments.

6 Conclusion

In this paper, we present both theoretical and practical results of an innovative subsampling scheme based on super-k-mers. We introduce the fractional hitting sets framework and propose a straightforward sketching method to highlight its benefits. This approach offers improved density compared to other schemes and tends to select k-mers that contribute to better space usage. Capitalizing on this scheme, we propose SuperSampler, an open-source sketching method for metagenomic assessment.

Through comprehensive experimental evaluation, we demonstrate that SuperSampler enables efficient and lightweight analysis of extensive genomic data sets with fewer resource requirements compared to the state-of-the-art tool Sourmash. More generally our results confirm the validity of our methodology from both theoretical and experimental standpoints.

We recognize several potential enhancements for our study. First, concerning SuperSampler's implementation, we aim to refine the tool for increased user-friendliness and adaptability for routine analysis while augmenting its capabilities, such as abundance tracking. Implementation of such improvements will lead to more thorough experiments with existing sampling methods as well as new comparisons with Sourmash using the same amount of disk memory in order to better show SuperSampler's capacity with regard to both fixed-size and scalable sketches.

Second, we plan to investigate alternative methods for sketch comparison, like sorted fingerprints, which could potentially reduce the complexity of the comparison process. From a theoretical perspective, delving deeper into the properties of Fractional Hitting Sets and gaining a better understanding of density and restricted density bounds for various values of f may lead to even more efficient and robust sketching techniques.

References

1 Clément Agret, Bastien Cazaux, and Antoine Limasset. Toward optimal fingerprint indexing for large scale genomics. In *22nd International Workshop on Algorithms in Bioinformatics*, 2022.

2 Daniel N Baker and Ben Langmead. Dashing: fast and accurate genomic distances with hyperloglog. *Genome biology*, 20(1):1–12, 2019.

3 Daniel N Baker and Ben Langmead. Dashing 2: genomic sketching with multiplicities and locality-sensitive hashing. In *RECOMB*, 2023.

4 Gaëtan Benoit, Pierre Peterlongo, Mahendra Mariadassou, Erwan Drezen, Sophie Schbath, Dominique Lavenier, and Claire Lemaitre. Multiple comparative metagenomics using multiset k-mer counting. *PeerJ Computer Science*, 2:e94, 2016.

5 Grace A Blackwell, Martin Hunt, Kerri M Malone, Leandro Lima, Gal Horesh, Blaise TF Alako, Nicholas R Thomson, and Zamin Iqbal. Exploring bacterial diversity via a curated and searchable snapshot of archived dna sequences. *PLoS biology*, 19(11):e3001421, 2021.

6 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.

7 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.

8 Mahmudur Rahman Hera, N Tessa Pierce-Ward, and David Koslicki. Debiasing fracminhash and deriving confidence intervals for mutation rates across a wide range of evolutionary distances. *bioRxiv*, 2022.

9 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.

10 Luiz Carlos Irber, Phillip T Brooks, Taylor E Reiter, N Tessa Pierce-Ward, Mahmudur Rahman Hera, David Koslicki, and C Titus Brown. Lightweight compositional analysis of metagenomes with fracminhash and minimum metagenome covers. *bioRxiv*, 2022.

11 Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

12 Yang Li et al. Mspkmercounter: a fast and memory efficient approach for k-mer counting. *arXiv preprint*, 2015. `arXiv:1505.06550`.

13 Shaopeng Liu and David Koslicki. Cmash: fast, multi-resolution estimation of k-mer-based jaccard and containment indices. *Bioinformatics*, 38(Supplement_1):i28–i35, 2022.

14 Camille Marchet, Mael Kerbiriou, and Antoine Limasset. Blight: efficient exact associative structure for k-mers. *Bioinformatics*, 37(18):2858–2865, 2021.

15 Camille Marchet and Antoine Limasset. Scalable sequence database search using partitioned aggregated bloom comb-trees. In *ISMB*, 2023. `doi:10.1101/2022.02.11.480089`.

16 Frédéric Meunier, Olivier Gandouet, Éric Fusy, and Philippe Flajolet. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science*, 2007.

17 Brian D Ondov, Gabriel J Starrett, Anna Sappington, Aleksandra Kostic, Sergey Koren, Christopher B Buck, and Adam M Phillippy. Mash screen: high-throughput sequence containment estimation for genome discovery. *Genome biology*, 20(1):1–13, 2019.

18 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.

19 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.

20 David Pellow, Lianrong Pu, Baris Ekim, Lior Kotlar, Bonnie Berger, Ron Shamir, and Yaron Orenstein. Efficient minimizer orders for large values of k using minimum decycling sets. *bioRxiv*, pages 2022–10, 2022.

21 Giulio Ermanno Pibiri. Sparse and skew hashing of K-mers. *Bioinformatics*, 38(Supplement_1):i185–i194, June 2022. doi:10.1093/bioinformatics/btac245.

22 Giulio Ermanno Pibiri, Yoshihiro Shibuya, and Antoine Limasset. Locality-preserving minimal perfect hashing of k-mers. *arXiv preprint*, 2022. arXiv:2210.13097.

23 N Tessa Pierce, Luiz Irber, Taylor Reiter, Phillip Brooks, and C Titus Brown. Large-scale sequence comparisons with sourmash. *F1000Research*, 8, 2019.

24 Mihai Pătraşcu and Mikkel Thorup. The power of simple tabulation hashing. *Journal of the ACM (JACM)*, 59(3):1–50, 2012.

25 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.

26 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.

27 Kristoffer Sahlin. Faster short-read mapping with strobemer seeds in syncmer space. *bioRxiv*, 2021.

28 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.

29 Yun William Yu and Griffin M Weber. Hyperminhash: Minhash in loglog space. *IEEE Transactions on Knowledge and Data Engineering*, 34(1):328–339, 2020.

30 XiaoFei Zhao. Bindash, software for fast genome distance estimation on a typical personal laptop. *Bioinformatics*, 35(4):671–673, 2019.

31 Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. Improved design and analysis of practical minimizers. *Bioinformatics*, 36(Supplement_1):i119–i127, 2020.

A Useful Lemmas

▶ **Lemma 16.** *Assuming p is non-increasing with respect to w, $1-(1-p)^{w+1} \underset{w\to\infty}{=} f+o(1)$.*

Proof. $(1-p)^{w+1} = (1-p)^w - p(1-p)^w$

- if $p \underset{w\to\infty}{\longrightarrow} 0$, $p(1-p)^w \underset{w\to\infty}{\longrightarrow} 0$
- otherwise $p \geq c$ for some $c > 0$ since it is non-increasing, so $p(1-p)^w \leq p(1-c)^w \underset{w\to\infty}{\longrightarrow} 0$

Therefore, $(1-p)^{w+1} = (1-p)^w + o(1)$. ◀

We use $\widehat{\mathcal{S}}$ to denote the set of k-mers containing a small m-mer.

▶ **Lemma 17.** *Given two consecutive k-mers W_0 and W_1, $\mathbb{P}(W_0, W_1 \in \widehat{\mathcal{S}}) \underset{w\to\infty}{=} f+o(1)$.*

Proof.

$$\begin{aligned}
\mathbb{P}(W_0, W_1 \in \widehat{\mathcal{S}}) &= 1 - \mathbb{P}(W_0 \notin \widehat{\mathcal{S}} \vee W_1 \notin \widehat{\mathcal{S}}) \\
&= 1 - \left[\mathbb{P}(W_0 \notin \widehat{\mathcal{S}}) + \mathbb{P}(W_1 \notin \widehat{\mathcal{S}}) - \mathbb{P}(W_0 \notin \widehat{\mathcal{S}} \wedge W_1 \notin \widehat{\mathcal{S}})\right] \\
&= 1 - 2(1-p)^w + (1-p)^{w+1} = 1 - (1-p)^w + o(1) && \text{(Lemma 16)} \\
&= f + o(1).
\end{aligned}$$

◀

▶ **Lemma 18.** $p \underset{w\to\infty}{=} -\frac{1}{w}\ln(1-f) + o(1/w)$

Proof. Because of property 10, we have $p = 1 - (1-f)^{1/w}$ and

$$(1-f)^{1/w} = \exp\left[\frac{1}{w}\ln(1-f)\right] = 1 + \frac{1}{w}\ln(1-f) + o(1/w).$$ ◀

B Proof of Theorem 11

In order to upper bound the density, we follow the same approach as the one presented in [31] (for the proof of theorem 7). As stated in [31], the density is equivalent to the probability that a context c (that is, the string formed by two consecutive k-mers) is *charged*, i.e. the two k-mers of c have different minimizers.

$$\begin{aligned} d &= \mathbb{P}_{c,h}(c \text{ is charged}) \\ &\leq \mathbb{P}_{c,h}(c \text{ has duplicate } m\text{-mers}) + \mathbb{P}_{c,h}(c \text{ is charged} \mid \text{no duplicate } m\text{-mers}) \end{aligned}$$

▶ **Lemma 19** (lemma 9 from [31]). *Assuming* $m > (3+\varepsilon)\log_\sigma w$,

$$\mathbb{P}_{c,h}(c \textit{ has duplicate } m\textit{-mers}) = o(1/w).$$

If c has no duplicate m-mers, the small m-mers are all distinct and each of them has the same probability to be minimal since h is random. Therefore,

$$\mathbb{P}_{c,h}(c \text{ is charged} \mid \text{no duplicate } m\text{-mers}) = \mathbb{E}_{c,h}\left[\frac{M_{boundary}}{M_{total}}\right]$$

where $M_{boundary}$ denotes the number of boundary m-mers that are small and M_{total} denotes the total number of small m-mers in c.

Let x_0 denote the first m-mer of c and x_w denote the last one,

$$\begin{aligned} \mathbb{E}_{c,h}\left[\frac{M_{boundary}}{M_{total}}\right] = \mathbb{E}_{c,h}\left[\frac{\mathbf{1}_{x_0\in\mathcal{S}} + \mathbf{1}_{x_w\in\mathcal{S}}}{M_{total}}\right] &= 2\cdot\mathbb{E}_{c,h}\left[\frac{\mathbf{1}_{x_0\in\mathcal{S}}}{M_{total}}\right] && \text{(symmetry)} \\ &= 2\cdot\mathbb{E}_{c,h}[1/M_{total} \mid x_0\in\mathcal{S}]\cdot\mathbb{P}(x_0\in\mathcal{S}). \end{aligned}$$

Assuming x_0 is small, we have $M_{total} = 1 + X$ with $X \sim B(w,p)$, since each other m-mer of c has a probability p to be small.

$$\mathbb{E}_{c,h}[1/M_{total} \mid x_0\in\mathcal{S}] = \mathbb{E}_{c,h}\left[\frac{1}{1+X}\right] = \sum_{i=0}^{w}\frac{1}{1+i}\binom{w}{i}p^i(1-p)^{w-i}$$

▶ **Lemma 20.** $\sum_{i=0}^{w}\frac{1}{1+i}\binom{w}{i}p^i(1-p)^{w-i} = \frac{1-(1-p)^{w+1}}{(w+1)p}$

Proof.

$$\begin{aligned} (w+1)p\sum_{i=0}^{w}\frac{1}{1+i}\binom{w}{i}p^i(1-p)^{w-i} &= \sum_{i=0}^{w}\frac{w+1}{1+i}\frac{w!}{i!(w-i)!}p^{i+1}(1-p)^{w-i} \\ &= \sum_{i=0}^{w}\frac{(w+1)!}{(i+1)!(w+1-(i+1))!}p^{i+1}(1-p)^{w+1-(i+1)} = \sum_{j=1}^{w+1}\binom{w+1}{j}p^j(1-p)^{w+1-j} \\ &= 1-(1-p)^{w+1} \end{aligned}$$ ◀

Finally, since $\mathbb{P}(x_0\in\mathcal{S}) = p$, $d \leq 2\cdot\frac{1-(1-p)^{w+1}}{w+1} + o(1/w) = \frac{2f}{w+1} + o(1/w)$ (Lemma 16)

C Proof of Theorem 12

In this section, **we assume that every k-mer we work with contains a small m-mer**.

Just as for the proof of theorem 11, we still have
$d \leq \mathbb{P}_{c,h}(c \text{ has duplicate } m\text{-mers}) + \mathbb{P}_{c,h}(c \text{ is charged} \mid \text{no duplicate } m\text{-mers})$ and

$$\begin{aligned}\mathbb{P}_{c,h}(c \text{ is charged} \mid \text{no duplicate } m\text{-mers}) &= \mathbb{E}_{c,h}\left[\frac{M_{boundary}}{M_{total}}\right] \\ &= 2 \cdot \mathbb{E}_{c,h}[1/M_{total} \mid x_0 \in \mathcal{S}] \cdot \mathbb{P}(x_0 \in \mathcal{S} \mid W_1 \in \widehat{\mathcal{S}}).\end{aligned}$$

▶ **Lemma 21.** *Assuming $m > (3+\varepsilon)\log_\sigma w$, $\mathbb{P}_{c,h}(c \text{ has duplicate } m\text{-mers}) = o(1/w)$.*

Proof. This proof is similar to the proof of lemma 9 from [31]. Let $i, j \in [\![0, w]\!]$ with $i < j$, $\delta = j - i$.

If $\delta < m$, $\mathbb{P}(x_i = x_j) = \frac{\sigma^\delta}{\sigma^{m+\delta}} = \frac{1}{\sigma^m} = o(1/w^3)$.

If $\delta \geq m$,

$$\begin{aligned}\mathbb{P}(x_i = x_j) &= \mathbb{P}(x_i = x_j \mid x_i, x_j \in \mathcal{S})\mathbb{P}(x_i, x_j \in \mathcal{S}) + \mathbb{P}(x_i = x_j \mid x_i, x_j \notin \mathcal{S})\mathbb{P}(x_i, x_j \notin \mathcal{S}) \\ &= \frac{\mathbb{P}(x_i, x_j \in \mathcal{S})}{p \cdot \sigma^m} + \frac{\mathbb{P}(x_i, x_j \notin \mathcal{S})}{(1-p)\sigma^m}.\end{aligned}$$

Because of lemma 17, $\mathbb{P}(x_i, x_j \in \mathcal{S}) = \frac{p^2}{\mathbb{P}(W_0, W_1 \in \widehat{\mathcal{S}})} = \frac{p^2}{f + o(1)} \leq p$ and

$$\mathbb{P}(x_i, x_j \notin \mathcal{S}) \leq \frac{(1-p)^2\left[1-(1-p)^{w-1}\right]}{\mathbb{P}(W_0, W_1 \in \widehat{\mathcal{S}})} = \frac{(1-p)^2\left[1-(1-p)^{w-1}\right]}{f + o(1)} \leq (1-p)^2.$$

Therefore, $\mathbb{P}(x_i = x_j) \leq \frac{p}{p \cdot \sigma^m} + \frac{(1-p)^2}{(1-p)\sigma^m} \leq \frac{2}{\sigma^m} = o(1/w^3)$.
Thus $\mathbb{P}_{c,h}(c \text{ has duplicate } m\text{-mers}) = \binom{w}{2} \times o(1/w^3) = o(1/w)$. ◀

Assuming x_0 is small, the w next m-mers of c form a k-mer, so we know that at least one of them is also small. Therefore,

$$\begin{aligned}\mathbb{E}_{c,h}[1/M_{total} \mid x_0 \in \mathcal{S}] &= \mathbb{E}_{c,h}\left[\frac{1}{1+X} \mid X \geq 1\right] = \frac{1}{\mathbb{P}(X \geq 1)} \sum_{i=1}^{w} \frac{1}{1+i}\binom{w}{i} p^i (1-p)^{w-i} \\ &= \frac{1}{f}\left[\frac{1-(1-p)^{w+1}}{(w+1)p} - (1-p)^w\right] = \frac{1}{f}\left[\frac{f+o(1)}{(w+1)p} - (1-f)\right]. \qquad \text{(Lemma 20 and 16)}\end{aligned}$$

What's more, $\mathbb{P}(x_0 \in \mathcal{S} \mid W_1 \in \widehat{\mathcal{S}}) = \frac{\mathbb{P}(x_0 \in \mathcal{S})}{\mathbb{P}(W_1 \in \widehat{\mathcal{S}})} = \frac{p}{f}$. Hence,

$$\begin{aligned}d &\leq \frac{2p}{f^2}\left[\frac{f+o(1)}{(w+1)p} - (1-f)\right] + o(1/w) = \frac{2}{f(w+1)} - \frac{2(1-f)p}{f^2} + o(1/w) \\ &= \frac{2}{f(w+1)} + \frac{2(1-f)\ln(1-f)}{f^2 w} + o(1/w) \qquad \text{(Lemma 18)} \\ &= 2 \cdot \frac{f + (1-f)\ln(1-f)}{f^2(w+1)} + o(1/w).\end{aligned}$$

D Proof of Theorem 13

In order to compute the proportion of maximal super-k-mers, we adapt the proof of theorem 4 from [22].

First, we introduce a similar Markov chain representing the position X of the small minimizer in the k-mer, with an extra state $\varnothing$ when there is no small minimizer.

We reuse the following notations introduced in [22]:

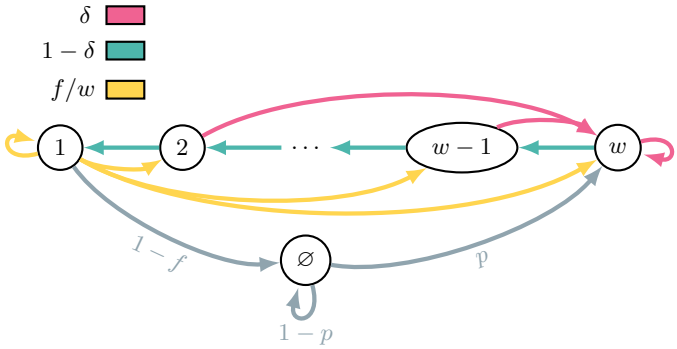


Figure 11 The chain is in state $i \in [\![1, w]\!]$ if the small minimizer starts at position i in the k-mer, and $\varnothing$ if there is no small minimizer. Different edge colors represent different probabilities.

- P_{lr} is the proportion of left-right-max (i.e. maximal) super-k-mers
- P_l is the proportion of left-max super-k-mers
- P_r is the proportion of right-max super-k-mers
- P_n is the proportion of non-max super-k-mers

$$\forall i \in [\![1, w-1]\!], \mathbb{P}(\text{first } X = i) = \mathbb{P}(X = 1) \cdot \tfrac{f}{w}$$

$$P_{lr} + P_r = \mathbb{P}(X = w) = \mathbb{P}(\text{first } X = w) = 1 - \sum_{i=1}^{w-1} \mathbb{P}(\text{first } X = i) = 1 - \mathbb{P}(X = 1) \cdot f \cdot (1 - 1/w)$$

$$P_{lr} + P_l = \mathbb{P}(\text{last } X = 1) = \mathbb{P}(X = 1) + \mathbb{P}(\text{first } X = 1) = \mathbb{P}(X = 1) \cdot (1 + f/w)$$

By symmetry, $P_l = P_r$, so $1 - \mathbb{P}(X = 1) \cdot f \cdot (1 - 1/w) = \mathbb{P}(X = 1) \cdot (1 + f/w)$.
Therefore, $\mathbb{P}(X = 1) = \frac{1}{1+f}$ and $\mathbb{P}(X = w) = 1 - \frac{f}{1+f}\left(1 - \frac{1}{w}\right) = \frac{1+f/w}{1+f}$.
What's more, $1 + P_{lr} = P_{lr} + P_l + P_{lr} + P_r + P_n = 2 \cdot \mathbb{P}(X = w) + P_n$ so

$$P_{lr} = P_n + 2 \cdot \mathbb{P}(X = w) - 1 = P_n + \frac{1 - f(1 - 2/w)}{1 + f} \text{ and } P_l = P_r = \mathbb{P}(X = w) - P_{lr} = \frac{f(1 - 1/w)}{1 + f} - P_n$$

$$\begin{aligned} P_n &= \mathbb{P}(\text{first } X \neq w) \cdot \mathbb{P}(\text{last } X \neq 1) = \mathbb{P}(X = 1) \cdot f \cdot (1 - 1/w) \cdot [1 - \mathbb{P}(X = 1) \cdot (1 + f/w)] \\ &= \left(1 - \frac{1}{w}\right) \cdot \frac{f}{1+f} \cdot \left[1 - \frac{1 + f/w}{1+f}\right] = \left(1 - \frac{1}{w}\right) \cdot \frac{f}{1+f} \cdot \frac{f(1 - 1/w)}{1+f} = \left[\left(1 - \frac{1}{w}\right)\frac{f}{1+f}\right]^2 \end{aligned}$$

Thus $P_l = P_r = \left[\left(1 - \frac{1}{w}\right)\frac{f}{1+f}\right]\left[1 - \left(1 - \frac{1}{w}\right)\frac{f}{1+f}\right]$ and $P_{lr} = \left[\left(1 - \frac{1}{w}\right)\frac{f}{1+f}\right]^2 + \frac{1-f(1-2/w)}{1+f}$.

E Proof of Theorem 14

This proof generalizes the proof of theorem 11 when the minimizers are selected from a UHS $\mathcal{U}$ with density $d_{\mathcal{U}}$.

First, because of independence, we have $\mathbb{P}(x_0 \in \mathcal{S} \cap \mathcal{U}) = \mathbb{P}(x_0 \in \mathcal{S}) \cdot \mathbb{P}(x_0 \in \mathcal{U})$ and $\mathbb{P}(|\mathcal{S} \cap \mathcal{U} \cap c| = i) = \sum_{n \geq i} \mathbb{P}(|\mathcal{U} \cap c| = n)\mathbb{P}(|\mathcal{S} \cap \mathcal{U} \cap c| = i \mid |\mathcal{U} \cap c| = n)$.

The main change of the proof lies in the bound on the expectation:

$$\mathbb{E}\left[\frac{1}{|\mathcal{S}\cap\mathcal{U}\cap c|} \mid x_0\in\mathcal{S}\cap\mathcal{U}\right] = \sum_{i=0}^{w}\frac{1}{i+1}\mathbb{P}(|\mathcal{S}\cap\mathcal{U}\cap c| = i+1 \mid x_0\in\mathcal{S}\cap\mathcal{U})$$

$$= \sum_{i=0}^{w}\frac{1}{i+1}\sum_{n=i}^{w}\mathbb{P}(|\mathcal{U}\cap c| = n+1 \mid x_0\in\mathcal{U})\mathbb{P}(|\mathcal{S}\cap\mathcal{U}\cap c| = i+1 \mid |\mathcal{U}\cap c| = n+1, x_0\in\mathcal{S}\cap\mathcal{U})$$

$$= \sum_{n=0}^{w}\mathbb{P}(|\mathcal{U}\cap c| = n+1 \mid x_0\in\mathcal{U})\sum_{i=0}^{n}\frac{1}{i+1}\mathbb{P}(|\mathcal{S}\cap\mathcal{U}\cap c| = i+1 \mid |\mathcal{U}\cap c| = n+1, x_0\in\mathcal{S}\cap\mathcal{U})$$

$$= \sum_{n=0}^{w}\mathbb{P}(|\mathcal{U}\cap c| = n+1 \mid x_0\in\mathcal{U})\sum_{i=0}^{n}\frac{1}{i+1}\mathbb{P}(|\mathcal{S}| = i+1 \mid x_0\in\mathcal{S}, |W| = n)$$

$$= \sum_{n=0}^{w}\mathbb{P}(|\mathcal{U}\cap c| = n+1 \mid x_0\in\mathcal{U})\cdot\frac{1-(1-p)^{n+1}}{(n+1)p} \qquad \text{(Lemma 20)}$$

$$\leq \sum_{n=0}^{w}\mathbb{P}(|\mathcal{U}\cap c| = n+1 \mid x_0\in\mathcal{U})\cdot\frac{f+o(1)}{(n+1)p}. \qquad \text{(Lemma 16)}$$

Therefore, using the same arguments as in the proof of theorem 11, we obtain

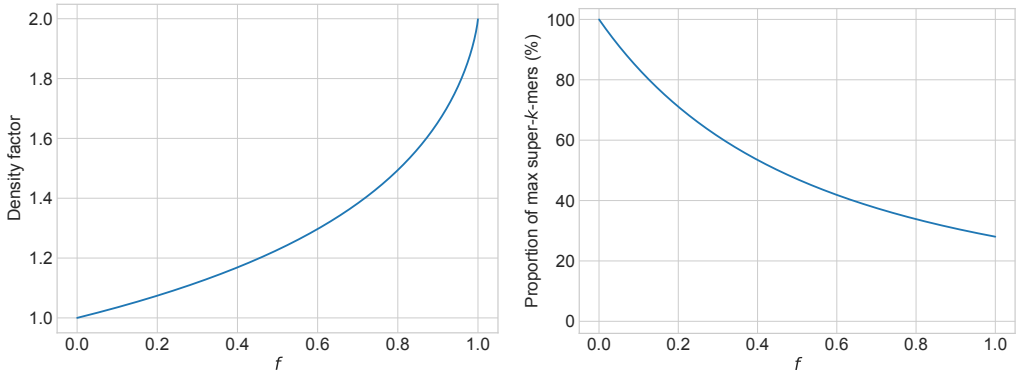
$$d_{\mathcal{S}\cap\mathcal{U}} \leq 2\cdot\mathbb{P}(x_0\in\mathcal{S}\cap\mathcal{U})\cdot\mathbb{E}\left[\frac{1}{|\mathcal{S}\cap\mathcal{U}\cap c|} \mid x_0\in\mathcal{S}\cap\mathcal{U}\right] + o(1/w)$$

$$\leq 2\cdot\mathbb{P}(x_0\in\mathcal{U})\cdot\mathbb{P}(x_0\in\mathcal{S})\sum_{n=0}^{w}\mathbb{P}(|\mathcal{U}\cap c| = n+1 \mid x_0\in\mathcal{U})\cdot\frac{f}{(n+1)p} + o(1/w)$$

$$= f\cdot 2\cdot\mathbb{P}(x_0\in\mathcal{U})\sum_{n=0}^{w}\frac{1}{n+1}\mathbb{P}(|\mathcal{U}\cap c| = n+1 \mid x_0\in\mathcal{U}) + o(1/w)$$

$$= f\cdot d_{\mathcal{U}} + o(1/w).$$

F Additional figures and algorithms



(a) Theoretical upper bound on the density factor from theorem 12, lower is better.

(b) Average proportion of maximal super-k-mers built from small minimizers (for $k = 31, m = 15$), higher is better.

Figure 12 Theoretical bounds on the density factor and the proportion of maximal super-k-mers depending on the fraction f of covered k-mers.

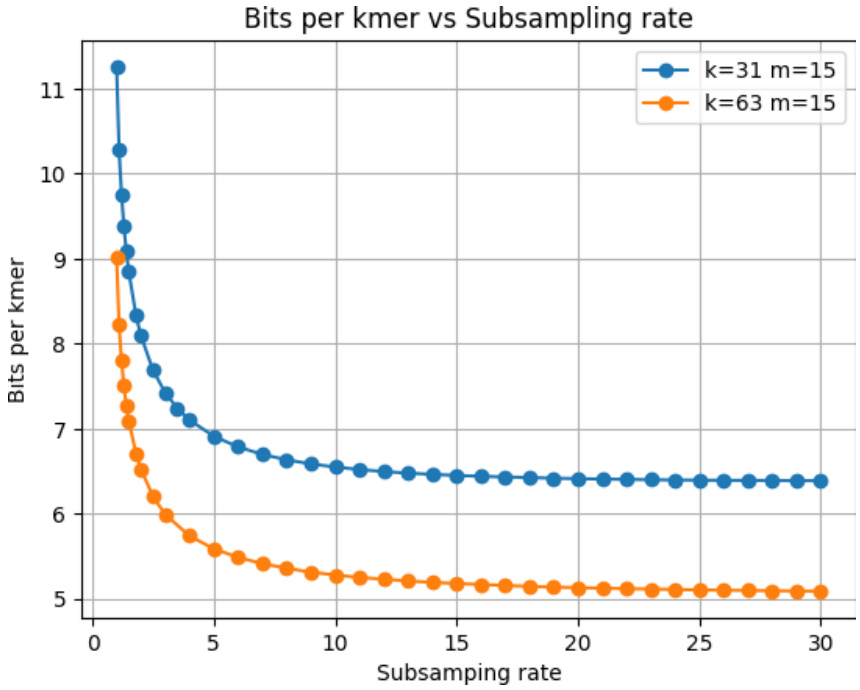


Figure 13 Space cost in bits per k-mer according to the subsampling rate.

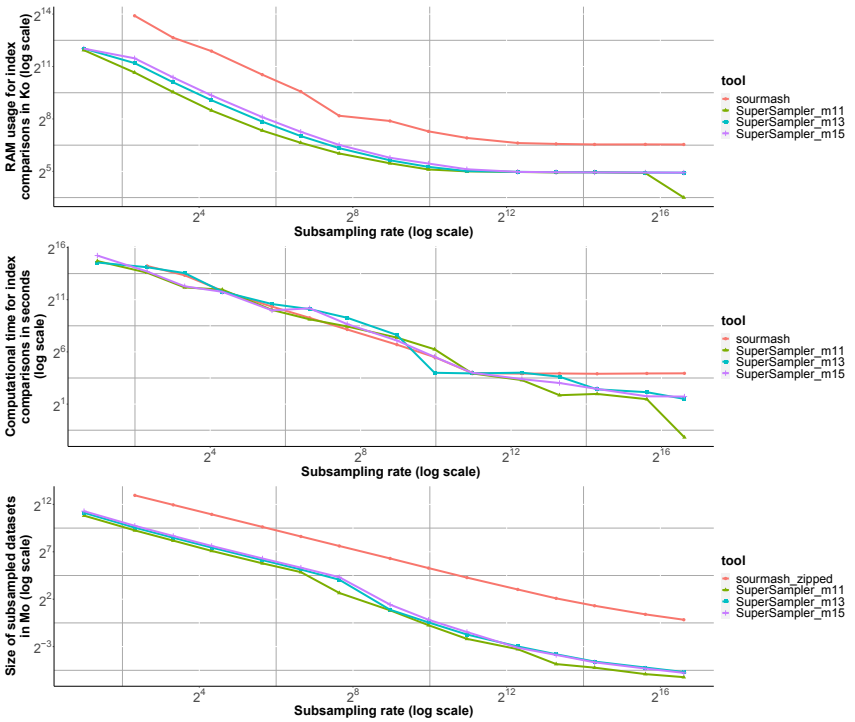


Figure 14 Salmonellas 1K k=31.

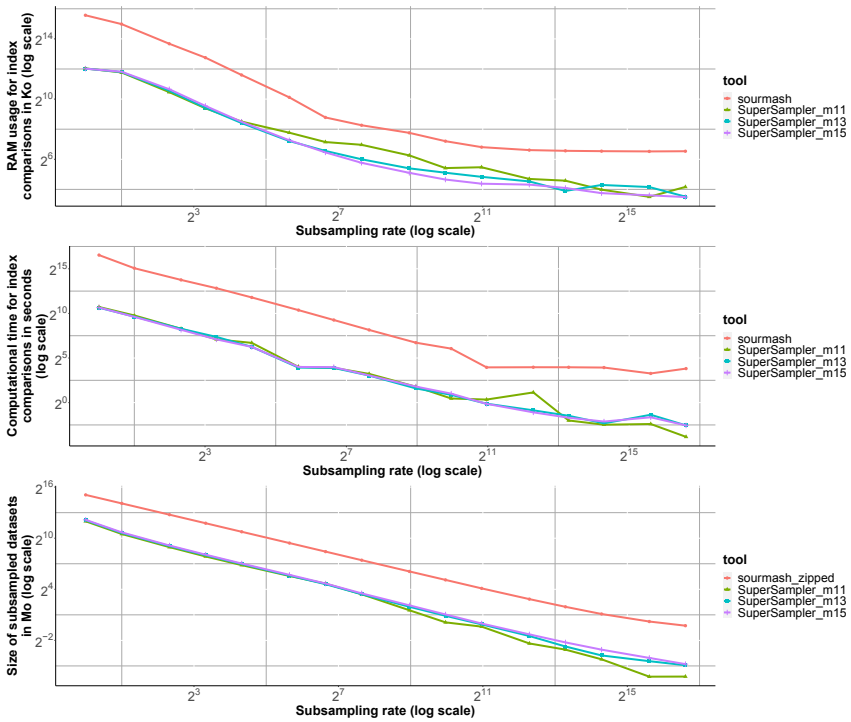


Figure 15 Refseq 1K k=63.

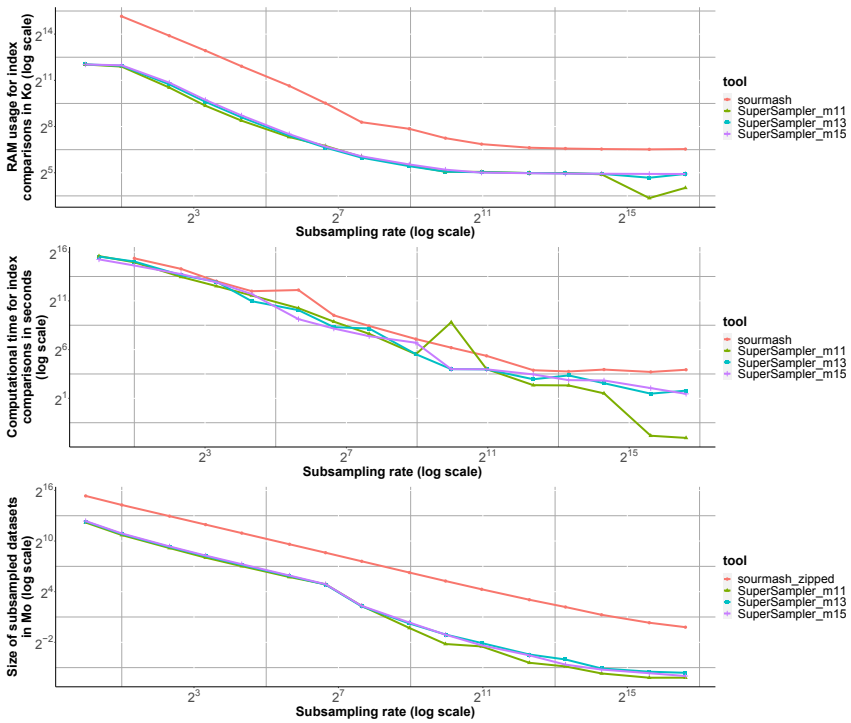
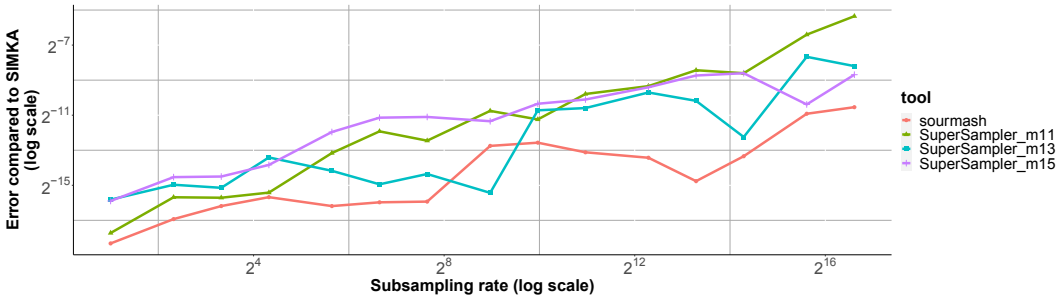
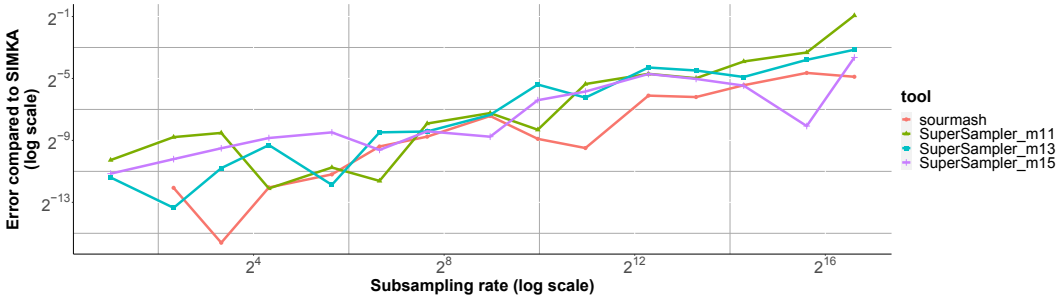


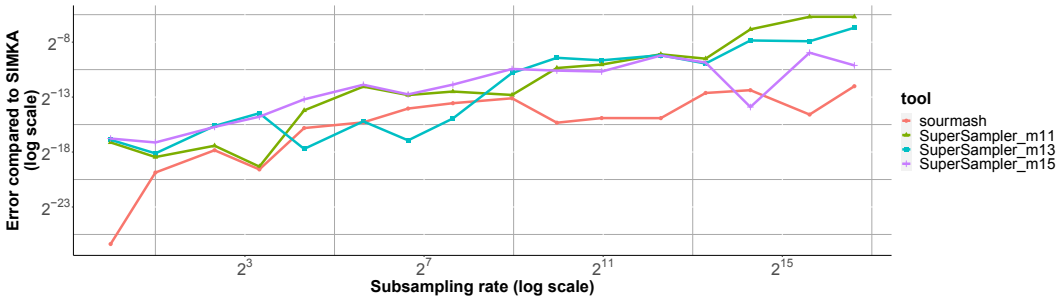
Figure 16 Salmonellas 1K k=63.



(a) RefSeq $k = 31$.

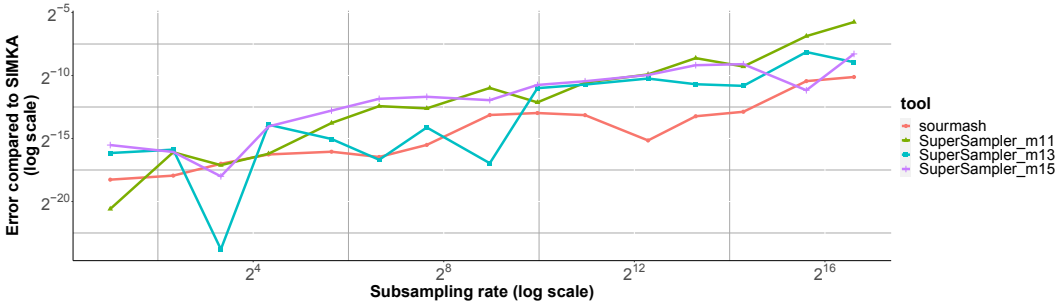


(b) Salmonellas $k = 31$.

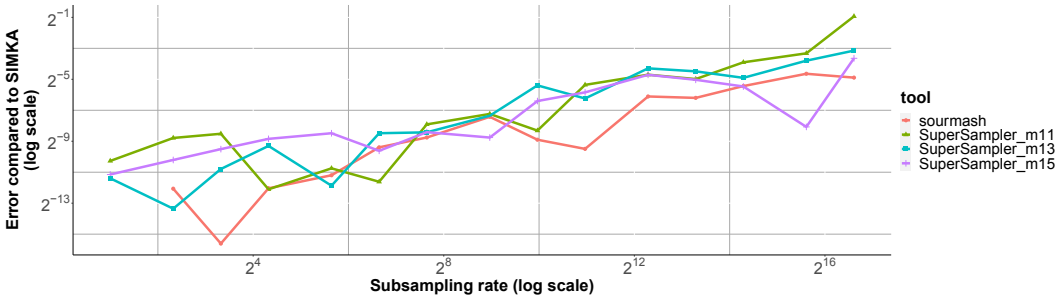


(c) RefSeq $k = 63$.

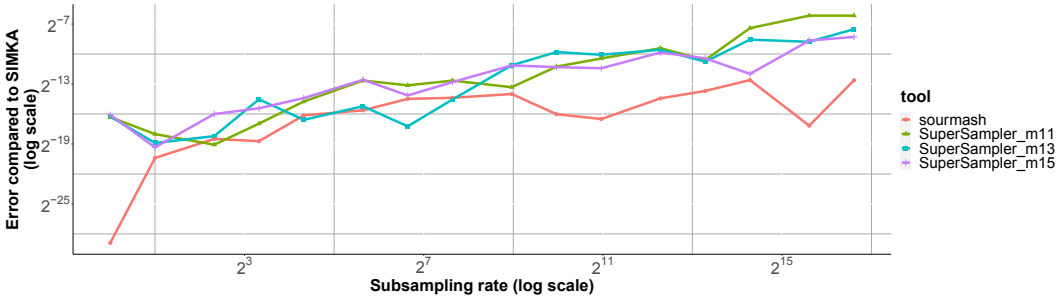
Figure 17 Error for containment index approximation for Sourmash (red line) and SuperSampler on different values for minimizer sizes.



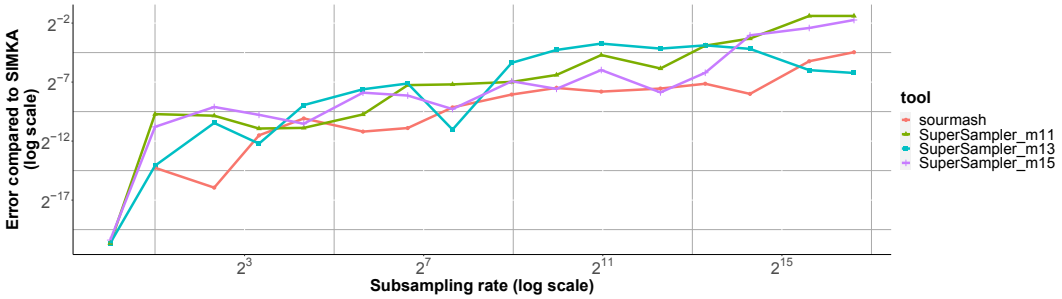
(a) RefSeq $k = 31$.



(b) Salmonellas $k = 31$.



(c) RefSeq $k = 63$.



(d) Salmonellas $k = 63$.

Figure 18 Error for Jaccard index approximation for Sourmash (red line) and SuperSampler on different values for minimizer sizes.

Algorithm 1 Comparison of two sketches.

```
procedure SketchCompare(sketch1, sketch2)
    for all minimizer ∈ sketch1 do
        minimizer1, minimizer2 ← 0
        minimizer1 ← updateMinimizer(minimizer1, sketch1)
        minimizer2 ← updateMinimizer(minimizer2, sketch2)
        if minimizer1 > minimizer2 then
                    ▷ If query minimizer is smaller than ref, skip current query minimizer
            updateMinimizer(minimizer2, sketch2)
        else if minimizer1 < minimizer2 then
                        ▷ If ref minimizer is smaller than query, skip current ref minimizer
            updateMinimizer(minimizer1, sketch1)
        else
                ▷ If both minimizers are equal, load buckets and compare every k-mer for
presence/absence.
            bucket1 ← getBucket(sketch1, minimizer1)
            bucket2 ← getBucket(sketch2, minimizer2)
            while buckets are not empty do
                kmer1 ← nextKmer(bucket1)
                kmer2 ← nextKmer(bucket2)
                if kmer1 == kmer2 then
                            ▷ When equal k-mers are found, similarity score is increased.
                    intersection + +
                end if
            end while
        end if
    end for
end procedure
```

Algorithm 2 All versus all comparison.

```
procedure AllVsAll(files)
    currMinimizer ← smallestMinimizer(files)
    Indices ← findInFiles(files, currMinimizer)
    if Indices > 1 then
        compareBuckets(Indices, files)
        incrementFiles(Indices, files)
    else
        incrementFiles(Indices, files)
    end if
    while filesarenotallempty do
        currMinimizer ← smallestMinimizer(files)
        Indices ← findInFiles(files, currMinimizer)
        if Indices > 1 then
            compareBuckets(Indices, files)
            incrementFiles(Indices, files)
        else
            incrementFiles(Indices, files)
        end if
    end while
end procedure
```

Fast, Parallel, and Cache-Friendly Suffix Array Construction

Jamshed Khan
University of Maryland, College Park, MD, USA

Tobias Rubel
University of Maryland, College Park, MD, USA

Laxman Dhulipala
University of Maryland, College Park, MD, USA

Erin Molloy
University of Maryland, College Park, MD, USA

Rob Patro
University of Maryland, College Park, MD, USA

Abstract

String indexes such as the suffix array (SA) and the closely related longest common prefix (LCP) array are fundamental objects in bioinformatics and have a wide variety of applications. Despite their importance in practice, few scalable parallel algorithms for constructing these are known, and the existing algorithms can be highly non-trivial to implement and parallelize. In this paper we present CaPS-SA, a simple and scalable parallel algorithm for constructing these string indexes inspired by samplesort. Due to its design, CaPS-SA has excellent memory-locality and thus incurs fewer cache misses and achieves strong performance on modern multicore systems with deep cache hierarchies. We show that despite its simple design, CaPS-SA outperforms existing state-of-the-art parallel SA and LCP-array construction algorithms on modern hardware. Finally, motivated by applications in modern aligners where the query strings have bounded lengths, we introduce the notion of a bounded-context SA and show that CaPS-SA can easily be extended to exploit this structure to obtain further speedups.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases Suffix Array, Longest Common Prefix, Data Structures, Indexing, Parallel Algorithms

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.16

Supplementary Material *Software (Source Code)*: `https://github.com/jamshed/CaPS-SA`
archived at `swh:1:dir:a503464134894c9067090bc3f65c04308842c5bb`

Funding *Rob Patro*: supported by the NIH under grant award numbers R01HG009937 and by NSF awards CCF-1750472 and CNS-176368.

1 Introduction

Methods for aligning sequencing reads to reference genomes underlie some of the most well-developed and widely-used tools in bioinformatics [2]. Modern read-to-reference aligners typically employ an *index* over the reference text. A classic index for strings is the suffix array (SA) [35], which is an array of indices of the lexicographically sorted suffixes of a string. In alignment, the SA index is used by the popular STAR aligner [12] as well as in other tools [48, 50]. The SA has also been used in short-read error correction [20] and sequence clustering [19]. A related object frequently used in conjunction with the SA is the Longest Common Prefix (LCP) array, which contains the lengths of the longest shared prefixes between pairs of successive indices in the SA. For instance, the SA can be used

in concert with the LCP-array (and other auxiliary tables derived from these) in a data structure called an enhanced suffix array [1] to mimic the functionality of a suffix tree [49], but often more efficiently and using less space. An account of the pervasiveness of the SA and the LCP-array in computational genomics is best left to a dedicated review (see e.g. [46]).

Because of the utility of the SA (and the LCP-array) in string indexing, significant work has been dedicated to developing practical algorithms for its construction. It is well-established that SA and LCP-array construction can be performed sequentially in time linear to the size of strings. However, as modern genomics pipelines produce ever more data – including more complete reference genomes and pangenomes – there has been a concerted effort to improve the practical efficiency and reduce the runtime of SA and LCP-array construction. A host of efficient serial algorithms have been developed [30, 26, 29, 37, 14, 39, 33]. Likewise, in an effort to take advantage of the increased parallelism of modern computer hardware, a number of parallel algorithms have also been proposed; e.g. parallel DivSufSort [32], parallel DC3 [31], and parallel divide-and-conquer based SA-construction [24]. External memory algorithms [22, 25, 23] have also been a focus of recent research because of the memory bottlenecks that arise when building the SA and the LCP-array on genomic datasets. Besides, algorithms for GPU-settings [34] and distributed-memory [15, 16] have been developed. We refer the interested reader to [43, 5, 6] for a comprehensive review. For our purposes, we note that these increasingly advanced methods introduce new algorithmic techniques to enable parallelism or improve the worst-case time complexity (so that it is sublinear). The trade-off, often, is that these more complex algorithms may potentially be more difficult to implement, optimize for modern hardware and cache layouts, and to maintain.

In this work, we address these issues by introducing CaPS-SA, a highly parallel method for constructing the SA and the LCP-array. A core principle behind CaPS-SA is simplicity. Our approach draws on several existing algorithms and techniques, and focuses on their efficient combination for the problem of highly parallel SA construction. The algorithm builds upon the *parallel samplesort* algorithm [17], and is easy to implement and optimize for modern hardware.

A potential downside of our approach is that it is *output-sensitive* and as a result its worst-case time complexity on adversarial inputs is quadratic. However, in practice we find that the shared-memory implementation of CaPS-SA outperforms state-of-the-art methods (specifically Parallel-divsufsort [32] and Parallel-DC3 [31, 3]) in terms of runtime and scalability (although not in memory for Parallel-divsufsort). For example, CaPS-SA can build the SA and the LCP-array for the telomere-to-telomere human genome assembly (CHM13 v2) [40] in 141 seconds using 32 threads on a typical shared-memory machine, whereas the leading method Parallel-divsufsort requires 199 seconds. Our experimental study demonstrates that this superior performance of CaPS-SA can largely be attributed to two causes. First, CaPS-SA achieves better memory-locality (fewer cache misses) than the other methods (likely thanks to its straightforward approach), and second, the worst-case analysis of CaPS-SA does not represent a typical or real world use case. Overall, our work demonstrates that as parallel resources increase combining domain-specific optimizations (i.e. LCP-informed merging) with highly-efficient general sorting strategies (i.e. samplesort [17]) can outperform more sophisticated but complex algorithms. CaPS-SA is implemented in C++17 and is available under an open source license at `https://github.com/jamshed/CaPS-SA`.

The remainder of this manuscript is organized as follows. We discuss the preliminary concepts required for a formal treatment of the algorithm as well as the most relevant prior work and the methods against which we compare CaPS-SA in Sec. 2. Then we discuss CaPS-SA in Sec. 3, and provide an analysis of its asymptotic behavior. Sec. 4 describes the

experimental study for the proposed algorithm, and reports the results. We conclude with discussion on the potential of the method and prospective future directions for building on top of it.

2 Preliminaries

A *string* (or *text*) $T = a_0 a_1 \dots a_{n-1}$ is a finite ordered sequence of n symbols drawn from a finite ordered alphabet Σ. $|T|$ denotes the length n of T. The half-open interval $[i, j)$ is a shorthand for the closed interval $[i \dots j-1]$. T_i denotes the i'th symbol in T. The *substring* $T_{[i,j)}$ of T is the sequence of characters of T on the half-open interval $[i, j)$. We call a substring $T_{[i,j)}$ with $i = 0$ a *prefix* of T. Likewise, A substring $T_{[i,j)}$ with $j = |T|$ is a *suffix* of T, denoted by $T_{[i:]}$.

The ordering of Σ induces a lexicographical ordering of all possible strings over Σ. The *Suffix Array* (SA) of a string T is an array of the starting indices of all suffixes of T ordered by the suffixes' lexicographical order. The *Longest Common Prefix* $\mathrm{LCP}(T_1, T_2)$ of two strings T_1 and T_2 is the largest-sized prefix P of both T_1 and T_2, such that if $|P| = k$ then for all $0 < i < k$, $T_{1i} = T_{2i}$, and $T_{1k} \neq T_{2k}$. Given the suffix array SA of a string T, its LCP-array is the array L such that $L_i = \mathrm{LCP}(T_{[SA_i:]}, T_{[SA_{i-1}:]})$. [1] For instance, given the string $T = \mathrm{AACTGCGGAT\$}$ the SA and LCP array are given by following data structure:

Index	0	1	2	3	4	5	6	7	8	9	10
T	A	A	C	T	G	C	G	G	A	T	**$**
SA	10	0	1	8	5	2	7	4	6	9	3
LCP array	0	0	1	1	0	1	0	1	1	0	1

The *work* of an algorithm is the total number of operations it performs to compute the result. The *depth* (or span) of an algorithm is the longest sequence of dependent computations in its execution. In pseudo-code, we will use $(\|)$ as an infix operator to specify the parallel execution of statements – so $f(x) \parallel g(y)$ denotes the parallel execution of $f(x)$ and $g(y)$. We use $\mathcal{O}\big(f(n)\big)$ *with high probability (whp)* in n to mean $\mathcal{O}\big(cf(n)\big)$ with probability at least $1 - n^{-c}$ for $c \geq 1$.

Prior Work. The SA can be constructed naively in $\mathcal{O}(n^2 \log n)$ work for an n-length text. Efficient algorithms can operate in $\mathcal{O}(n)$ work and $\mathcal{O}(n)$ space, which is the theoretical optimum, as it is the time and space required to record the SA and LCP array themselves. A comprehensive discussion of work on SA construction is well beyond the scope of this manuscript. As such, we here focus on several sequential and parallel algorithms which are of particular interest due to their speed and wide use.

The state-of-the-art sequential program for SA construction is `divsufsort` [37, 14]. Subsequent work has elucidated the algorithm to be an efficient implementation of some two-stage algorithms [21, 14].

`Divsufsort` has been parallelized by Labeit et. al. in 2017 [32], and is considered a state-of-the-art tool in parallel SA construction. It has recently been used in several computational genomics tools, including the `CAMMiQ` method for microbial abundance quantification [51] and the `macle` tool for computing match complexity [42].

Another well-known strategy for SA construction is the Difference Cover modulo 3 (DC3) method [27], which has also been effectively parallelized [31].

[1] With the special case of $L_0 = 0$.

Relevant String Sorting Methods. String sorting is a well-studied algorithmic problem. The main difficulty in string sorting is that comparing two strings T_1 and T_2 requires $\mathcal{O}(\min(|T_1|, |T_2|))$ comparisons, which renders many traditional sorting algorithms for atomic objects costly. Of particular relevance to our work is the problem of *merging* two sorted lists of strings. Farach-Colton used an efficient merge-routine for building suffix trees in linear time [13] (though the space overhead of suffix trees renders them impractical for most modern applications). Ng and Kakehi analyzed an efficient merging strategy of sorted lists of strings with associated LCP-information [38]. They show that given random strings with uniform distribution of symbols, an LCP-informed merge-sort algorithm has an expected running time of $\mathcal{O}(n \log n)$ to sort n strings. The same merge procedure was used by Bingmann and Sanders in several samplesorting algorithms for sorting collections of strings [8].

Bingmann and Sanders propose two merge-based string sorting algorithms of particular interest to us here: Parallel Super Scalar String Sample Sort (pS^5) and Parallel Multiway LCP-Mergesort. pS^5 makes use of the merge routine in a samplesort framework, much like CAPS-SA. The algorithms differ in their inputs (a set of strings vs a single string) as well as their approach to partitioning the data. pS^5 uses machine-word-sized pivot keys to create a binary search tree which can fit into the cache of each core, then divides up the input set of strings evenly across the cores and bins them accordingly. As described in Section 3.1, CAPS-SA divides up the input into evenly sized partitions, then samples pivots using a two-step process and places them into each partition. Parallel Multiway LCP-Mergesort generalizes the merge-algorithm to k-way merges [7].

3 Methods

The proposed algorithm, CAPS-SA, is based on the *samplesort* [17] algorithm. Samplesort is a popular generalization of quicksort that achieves excellent performance on both shared-memory and distributed-memory architectures [44, 4]. Instead of partitioning the input array into two parts around a single pivot as in quicksort, it chooses a number of pivots $z_1, z_2, \ldots, z_{p-1}$ along with two sentinel pivots $z_0 = -\infty$ and $z_p = +\infty$, and partitions the data into p partitions such that an input element x_i is assigned to partition j iff $z_{j-1} < x_i \leq z_j$. It then sorts each partition using another (usually sequential) sorting algorithm (e.g., quicksort).

For constructing a suffix array, simply applying samplesort is costly since string comparisons in general require super-constant time. In more detail, first each suffix needs to be assigned to its partition by binary searching over the pivots. Secondly, sorting the suffixes in each partition may cost substantially more than linearithmic time due to string comparisons.

CAPS-SA addresses these issues using the following key idea of *jointly leveraging merge sort and LCP-arrays*. Whenever two suffixes are compared, the comparison is always done inside the operation of merging two sorted arrays of suffixes. Each sorted array is augmented with its LCP-array, and the merge operations avoid repeated comparisons of common prefixes among suffixes by exploiting these LCP-arrays. This approach has previously been used in general string sorting algorithms [38, 8, 7], but to our knowledge has not been leveraged for SA construction. The partitioning strategy for the suffixes is modified to make better use of the merge operation and achieve good parallelism. In particular, instead of randomly sampling pivots at the beginning of the algorithm, CAPS-SA partitions the suffixes uniformly into p subarrays, sorts the subarrays locally, and only then selects the pivots using oversampling. Once pivots are placed within each partition, the p partitions are further subdivided into $p - 1$ subarrays each, for a total of $p(p - 1)$ sub-subarrays. Since

each sub-subarray is flanked by two pivots, the partition that it should go to is known. Each partition is thus a collection of sorted sub-subarrays, which can be merged efficiently. The initial sorting of the uniform-sized subarrays is done using merge-sort to exploit the merge operation. Thus CaPS-SA ends up exploiting an efficient merging procedure with associated LCP-information to reduce expensive comparisons of suffixes, while not having to merge large sub-arrays due to its pivoting strategy. We discuss the algorithm in more detail in the following sections.

3.1 Parallel SA and LCP-array construction: CaPS-SA

Next, we provide a high-level overview of the CaPS-SA(T, p) algorithm. The input to the algorithm is a string T and a partition count (or, *subproblem count*) p, and as output it produces the SA and the LCP-array of T. Conceptually, the algorithm executes in four high-level steps which we illustrate in Figure 1.

```
CaPS-SA(T, p).
 1  SA, SA′, L, L′ = arrays of size |T|

    // populate the suffix array with some permutation of [0, n)
 2  for i ∈ [0, |T|) in parallel
 3      SA_i = SA′_i = i

    // sort p subarrays of uniform size
 4  m = |T|/p
 5  for i ∈ [0, p) in parallel
 6      b = im, e = (i + 1)m // range of the i'th subarray
 7      Merge-Sort(SA′_[b:e), SA_[b:e), L_[b:e), L′_[b:e), T)

    // sample (p − 1) pivots from SA
 8  V = Sample-Pivots(SA, L, T, p)

    // locate each v_j ∈ V in each sorted subarray; for two successive pivots v_{j−1} and v_j,
    // place all suffixes k in each subarray s.t. T_[v_{j−1}:] < T_[k:] ≤ T_[v_j:] in partition j in SA′
 9  (S, R) = Collate-Partitions(SA, SA′, L, L′, V, T, p)
    // S_j: index of partition j in SA′, R_{j,i}: position of pivot v_i in partition j

    // merge the p sorted subarrays in each partition
10  Par-Copy(SA′, SA), Par-Copy(L′, L) // Par-Copy(X, Y) copies X to Y in parallel
11  for j ∈ [0, p) in parallel
12      b = S_j, e = S_{j+1}
13      Merge-Partition(SA′_[b,e), SA_[b,e), p, R_j, L′_[b:e), L_[b:e), T)

    // compute LCP-values at the partition boundaries
14  Compute-Boundary-LCPs(SA, p, S, L, T)

15  return (SA, L)
```

First, it populates an unsorted SA. Then this initial SA is broken into p subarrays of uniform size $|T|/p$, and each subarray is sorted with Merge-Sort, in parallel. Next, $p-1$ global pivots are sampled from the sorted subarrays together. Then in each sorted subarray, in parallel, each pivot is located with a binary search. The locations of the $p-1$ pivots thus found in each sorted subarray break the subarray into p sorted sub-subarrays. Besides, the position of each pivot in the final SA is now defined by its location in each of the p subarrays. The local ordering of the suffixes in each sorted subarray and the global position of the pivots thus define p partitions for the final SA, each of which is a collection of p sub-subarrays: one from each of the p sorted subarrays. Then for each partition, in parallel, its p sorted sub-subarrays are merged recursively into a fully sorted partition. Together, these sorted partitions, in order, produce the final SA and the final LCP-array. The LCP-values for pairs that cross partition boundaries are computed at the end.

```
MERGE(X, Y, L_X, L_Y, Z, L_Z, T).
    // merges the sorted arrays of suffixes (of T) X
    // and Y with LCP arrays L_X and L_Y resp.,
    // into Z; also constructs Z's LCP array in L_Z
 1  i = j = k = 0
 2  m = 0 // LCP of the last compared pair
 3  l_x = 0 // LCP(X_i, X_{i-1})
 4  while i < |X| and j < |Y|
 5      l_x = L_{X i}
 6      if l_x > m
 7          Z_k = X_i, L_{Z k} = l_x, m = m
 8      elseif l_x < m
 9          Z_k = Y_j, L_{Z k} = m, m = l_x
10      else
11          n = m + LCP(T_{[X_i + m :]}, T_{[Y_j + m :]})
12          Z_k = { X_i   if T_{X_i + n} < T_{Y_j + n}
                  { Y_j   otherwise
13          L_{Z k} = { l_x   if Z_k == X_i
                      { m   otherwise
14          m = n

15      if Z_k == X_i
16          i = i + 1
17      else
18          j = j + 1
19          SWAP(X, Y), SWAP(L_X, L_Y), SWAP(i, j)

20      k = k + 1

21  COPY(X_{[i:]}, Z_{[k:]}), COPY(L_{X [i:]}, L_{Z [k:]})

22  if j < |Y|
23      Z_k = Y_j, L_{Z k} = m
24      COPY(Y_{[j+1:]}, Z_{[k+1:]}),
        COPY(L_{Y [j+1:]}, L_{Z [k+1:]})
```

```
MERGE-SORT(X, Y, L, L', T).
    // sorts the array X of suffixes (of T) into Y;
    // constructs the LCP array of sorted X in L
    // using L' as working space;
    // a necessary precondition is X == Y
1   n = |X|
2   if n == 1
3       L_0 = 0
4       return

5   m = n/2
6   MERGE-SORT(Y_{[0:m)}, X_{[0:m)}, L'_{[0:m)}, L_{[0:m)}, T) ||
    MERGE-SORT(Y_{[m:n)}, X_{[m:n)}, L'_{[m:n)}, L_{[m:n)}, T)
7   MERGE(X_{[0:m)}, X_{[m:n)}, L'_{[0:m)}, L'_{[m:n)}, Y, L, T)
```

```
MERGE-PARTITION(X, Y, n, R, L_X, L_Y, T).
    // merges the collection X of n sorted subarrays
    // of suffixes into Y; R contains the ranges of the
    // subarrays in X; L_X is the collection of the
    // LCP arrays of the subarrays, and the LCP array
    // of Y is constructed in L_Y;
    // necessary preconditions are X == Y and L_X == L_Y
1   if n == 1
2       return

3   m = n/2
4   l = S_m - R_0 // #suffixes in the first m subarrays
5   MERGE-PARTITION(Y_{[0:l)}, X_{[0:l)}, m, R_{[0:m)},
                    L_{Y [0:l)}, L_{X [0:l)}, T) ||
    MERGE-PARTITION(Y_{[l:R_n)}, X_{[l:R_n)}, n - m, R_{[m:n)},
                    L_{Y [l:R_n)}, L_{X [l:R_n)})
6   MERGE(X_{[0:l)}, X_{[l:R_n)}, L_{X [0:l)}, L_{X [l:R_n)}, Y, L_Y, T)
```

The algorithm is presented as following, and its major steps are detailed in the following subsections. Then we analyze the asymptotic characteristics of the algorithm.

The MERGE operation. For efficient suffix comparisons, CAPS-SA utilizes the MERGE operation. A pair of suffixes is compared only when merging two sorted lists of suffixes, with the only exception being the case when the algorithm performs a binary search using a pivot suffix. When merging sorted suffixes, merging without any extra information about the suffixes in its input lists can be costly due to super-constant time string comparisons. To avoid comparing repeated prefixes of suffixes, the MERGE procedure in CAPS-SA utilizes the LCP-arrays of the input suffix lists, generated recursively in the MERGE-SORT procedure.

The MERGE$(X, Y, L_X, L_Y, Z, L_Z, T)$ procedure takes two sorted arrays X and Y of suffixes, their respective LCP-arrays L_X and L_Y, and populates the array Z as the merged output for X and Y. Also, the LCP-array of Z is produced in L_Z. The procedure works exactly like the classic merge routine, with the following modifications.

At a given moment, let X_i and Y_j be the two suffixes being compared, and Z_k be the output of the comparison. Without loss of generality, say that $X_i < Y_j$ is found, i.e. $Z_k = X_i$. Let m denote the LCP-length of the the last compared pair in each step of the merge. Then after the current step finishes comparing X_i, Y_j, we have that $m = \text{LCP}(X_i, Y_j)$. $X_i < Y_j$ implies that $T_{X_i+m} < T_{Y_j+m}$. The next suffixes to compare are X_{i+1} and Y_j. Let $l_x = L_{Xi+1} = \text{LCP}(X_{i+1}, X_i)$. $X_{i+1} > X_i$ implies that $T_{X_{i+1}+l_x} > T_{X_i+l_x}$. There are three possible outcomes when comparing l_x and m (illustrated in Figure 2):

1. $l_x > m$: It implies that $T_{X_{i+1}+m} = T_{X_i+m}$. Combining with $T_{X_i+m} < T_{Y_j+m}$, we get $T_{X_{i+1}+m} < T_{Y_j+m}$. It follows that

$$Z_{k+1} = X_{i+1}, L_{Zk+1} = \text{LCP}(Z_{k+1}, Z_k) = \text{LCP}(X_{i+1}, X_i) = l_x, m = \text{LCP}(X_{i+1}, Y_j) = m$$

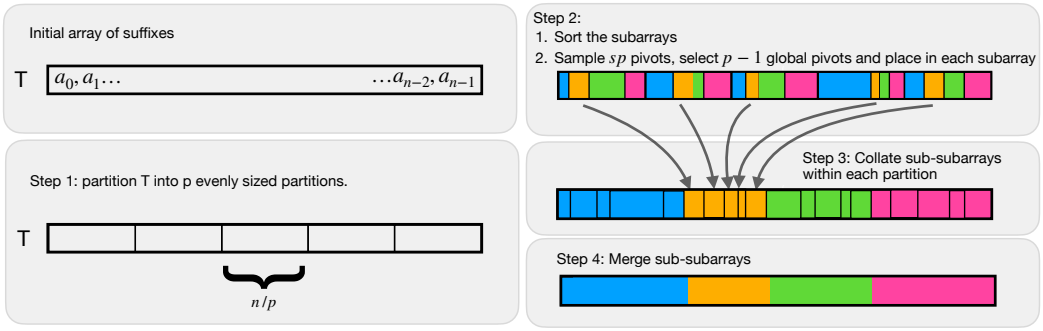


Figure 1 Overview of CAPS-SA. In the first step of the algorithm the input text T is partition evenly across n partitions. Then each partition is sorted, pivots are sampled using the sampling routine, and located within each partition to create sub-partitions. Subsequently each sub-partition is collated. Finally the MERGE routine is used to complete the suffix and the LCP-array construction.

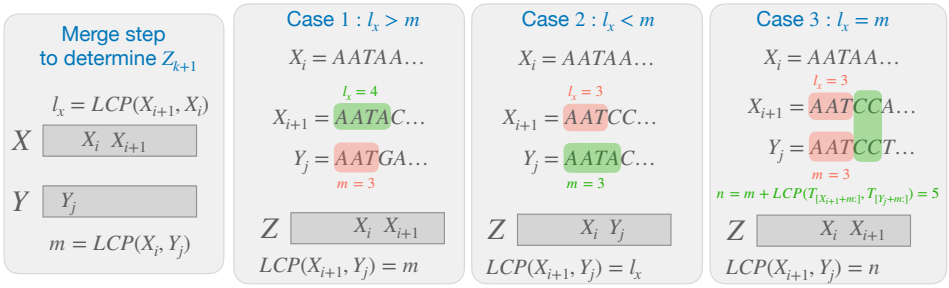


Figure 2 Figure illustrating the cases that can occur on the $(k+1)$'th step of the MERGE routine, which determines Z_{k+1}. Cases 1 and 2 require $\mathcal{O}(1)$ work and simply compare the LCP-lengths of the previous step and $LCP(X_{i+1}, X_i)$, which are already available. Step 3 requires work proportional to $\mathcal{O}(LCP(X_{i+1}, Y_j) - m)$, since X_{i+1}, Y_j already share a prefix of size m.

2. $l_x < m$: It implies that $T_{X_i+l_x} = T_{Y_j+l_x}$. Combining with $T_{X_{i+1}+l_x} > T_{X_i+l_x}$, we get $T_{X_{i+1}+l_x} > T_{Y_j+l_x}$. It follows that

$$Z_{k+1} = Y_j, L_{Z\,k+1} = LCP(Z_{k+1}, Z_k) = LCP(Y_j, X_i) = m, m = LCP(X_{i+1}, Y_j) = l_x$$

3. $l_x == m$: We compute $n = m + LCP(T_{[X_{i+1}+m:]}, T_{[Y_j+m:]})$, and set the following:

$$Z_{k+1} = \begin{cases} X_{i+1} & \text{if } T_{X_{i+1}+n} < T_{Y_j+n} \\ Y_j & \text{otherwise} \end{cases}$$

$$L_{Z\,k+1} = LCP(Z_{k+1}, Z_k) = \begin{cases} LCP(X_{i+1}, X_i) = l_x & \text{if } Z_{k+1} == X_{i+1} \\ LCP(Y_j, X_i) = m & \text{otherwise} \end{cases}$$

$$m = LCP(X_{i+1}, Y_j) = n$$

The MERGE procedure continues this way through X and Y. Finally, when either of X and Y has been depleted, the rest of the entries at the other one are copied to the end of Z and L_Z.

Local sorting. CAPS-SA starts out with some permutation of $[0, |T|)$, and sorts its p disjoint subarrays, each of size $|T|/p$, in parallel using MERGE-SORT. The MERGE-SORT(X, Y, L, L', T) procedure takes as input an array X of suffixes, and sorts it into Y. Besides, the LCP-array

of sorted suffixes is produced in L, using L' as working space. As typical Merge-Sort implementation requires linear extra space in each invocation, CaPS-SA uses the arrays X and Y in a back-and-forth manner to reuse the extra space in the invocations. For such, Y needs to be equal to X before an invocation. The merge step in the sort uses the Merge-procedure described earlier.

Pivot selection. CaPS-SA deviates from samplesort in its pivot selection strategy. In a typical samplesort, pivots are to be sampled from the initial array and then partitioning would be based on their intervals. Instead, in parallel, the Sample-Pivots(SA, T, p) procedure (see Suppl.) in CaPS-SA samples s suffixes from each of the p subarrays, where s is the *sampling factor*. Then these $s \times p$ sample suffixes are sorted using a sequential Merge-Sort. Subsequently, $p - 1$ evenly-spaced pivots are selected from the sorted output to form the pivot set V.

These pivots define the ranges of the samplesort partitions, and are used to split each of the subarrays in the next collation step. We show in Theorem 1 that with a sufficient sampling factor s, the size of each partition is within a constant factor of $|T|/p$ with high probability, which ensures a balanced load for processing each partition in the last step of the algorithm.

Collating partitions. Having finalized the pivot set V, the algorithm locates each pivot suffix $v \in V$ in each sorted subarray. Each subarray is searched for the $p-1$ pivots in parallel.

Consider a pivot $v \in V$ and some sorted subarray A. The position of v in A is the last index where v can be inserted without breaking the sorted order of A. This index is computed using a binary search for the suffix v in A. During a binary search suffixes are compared without any associated LCP array, contrary to the Merge procedure. As a practical speedup, we skip some repeated character comparisons between v and the suffixes in A using the *simple accelerant* idea [18].

After placing each pivot v into A, the index i of v in A implies that all the suffixes in $A_{[0:i)}$ are $\leq v$. Hence the sum C_v of these indices of v across all the p sorted subarrays provides the total suffix count in the SA that are not lexicographically larger than v – the index of v in the final SA is $C_v - 1$. Along with the sentinel pivot positions $C_0 = 0$ and $C_p = |T|$, these $p - 1$ pivots divide the final SA into p partitions. Consider two successive pivots v_{j-1} and v_j. In each sorted subarray A, all the suffixes k such that $T_{[v_{j-1}:]} < T_{[k:]} \leq T_{[v_j:]}$ will be present in the index-range $[C_{j-1}, C_j)$ of the final SA. That is, all the suffixes between the locations for v_{j-1} and v_j belong to the $(j-1)$'th partition.

Thus the pivot locations in a sorted subarray A break A into p sub-subarrays, where the j'th sub-subarray is known to be present in the j'th partition of the final SA. After the binary searches, CaPS-SA moves these sub-subarrays in parallel to collate all sub-subarrays for the same partition. The LCP-arrays of these sub-subarrays are also collated together. The Collate-Partitions(SA, SA', L, L', V, T, p) procedure (see Suppl.) describes it in more detail.

Merging partitions. Having grouped together the corresponding sub-subarrays for every partition, CaPS-SA merges together the sorted sub-subarrays in each partition, in parallel. A partition consists of p sorted collections of suffixes, with all of the collections stored contiguously. The Merge-Partition(X, Y, n, R, L_X, L_Y, T) procedure takes this collection X of n sorted sub-subarrays, and produces the merged output in the same contiguous region of memory Y recursively. L_X is the collection of the LCP-arrays of the sorted groups in X, and the merged LCP-array is produced in L_Y. The sorted groups in X (and L_X) are delineated by R.

The Merge-Partition procedure is same as the Merge-Sort procedure, except for that it is more general – the sorted units where Merge-Sort bottoms out are single suffixes, whereas Merge-Partition bottoms out earlier at sorted groups of suffixes. As noted earlier, Merge-Partition also uses the space in X and Y back-and-forth to reuse the extra spaces required.

3.2 Asymptotics

In this section, we analyze the computational complexity of the CaPS-SA(T, p) algorithm executed on a text T with length $n = |T|$, given a subproblem-count p.

3.2.1 Work Analysis

We start by analyzing the overall work of the algorithm and providing self-contained proofs on the total work due to symbol comparisons made by our algorithm.

Local sorting. This step executes the classic Merge-Sort on each subarray. For a subarray A with m suffixes, this amounts to a total work of $T(m) = 2T(m/2) + \mathcal{O}(m) + C(A)$, where $C(A)$ denotes the number of symbol comparisons made in the execution in the third case of the Merge procedure. We analyze the total amortized cost of these $C(A)$ values across all the recursion-trees of all the subarrays in Theorem 3. Omitting $C(A)$ from $T(m)$, each local sort has $n/p \log n/p$ work.

Pivot selection. With a sampling factor s, there are $s \times p$ pivots sampled in total across all the subarrays. CaPS-SA sorts these pivots with Merge-Sort and picks the $p - 1$ equidistant pivots from these as the global pivots. The Merge-Sort amounts to a total work of $\mathcal{O}(sp \log sp + \sum L_{p_i})$, where L_p is the output LCP-array of the sort. This holds from Theorem 3.

Collating partitions. The collation step first locates each of the $p - 1$ pivots in each of the sorted subarrays using binary search. The length of a pivot suffix is $\mathcal{O}(n)$, and the sorted subarrays are of size n/p. The work of each binary search is $\mathcal{O}(n + \log n/p)$ in practice ($n = |T|$) with the simple-accelerant [18] strategy. For adversarial inputs however, the work can still be $\mathcal{O}(n \log n/p)$ in the worst-case. Then the suffix indices are moved into their appropriate final partitions. This step simply reorders the elements across the sorted subarrays, and thus requires $\mathcal{O}(n)$ total work.

Merging partitions. The Merge-Partition procedure works similar to the Merge-Sort procedure, except for that the recursion bottoms out at a sorted group of suffixes, instead of at a single suffix. Unlike the Merge-Sort instances however, each of which operate on n/p-sized subarrays, the Merge-Partition instances may work on various sizes of partitions. Theorem 1 provides a bound on the partition sizes.

▶ **Theorem 1.** *With a sampling factor s, every partition has size at most cn/p for some constant c with high probability.*

Proof. The algorithm samples s pivots from each subarray, for a total of sp samples. It then picks $p - 1$ evenly spaced pivots (every s'th sample) from the sorted samples to use as global pivots.

Consider the final location of these sp samples in the final suffix array. Every s'th of them marks the boundary of a partition. Thus, a partition has size $d \geq cn/p$ only if fewer than s of the samples fall into these d suffixes. Otherwise, at least one sample would be picked as a final pivot and would thus break this partition.

Let SA be the final suffix array, and X_i be a random variable indicating whether SA_i is one of the sp samples. Then $\Pr[X_i = 1] = sp/n$. Thus the random variable denoting the number of samples picked from a region of size cn/p is $X = \sum_{i=1}^{cn/p} X_i$. By linearity of expectation, we get $E[X] = \sum_{i=1}^{cn/p} E[X_i] = cn/p \Pr[X_i = 1] = cn/p \cdot sp/n = cs$. Applying the Chernoff bound we have:

$$\begin{aligned}\Pr[X < s] \leq \Pr[X \leq s] &= \Pr\big[X \leq \frac{1}{c}E[X]\big] = \Pr\big[X \leq \big(1 - (1 - \frac{1}{c})\big)E[X]\big] \\ &\leq \exp\big(-1/2(1 - 1/c)^2 E[X]\big) = \exp(-1/2(1 - 1/c)^2 cs) \qquad (1)\end{aligned}$$

With $s = 32 \ln n$ and letting $c' = c(1 - 1/c)^2$,

$$\Pr[X < s] \leq \exp\big(-1/2(1 - 1/c)^2 c \cdot 32 \ln n\big) = \exp\big(\ln(n^{-16c(1-1/c)^2})\big) = 1/n^{16c'}$$

Since the event of a partition having size $\geq cn/p$ implies the event $X < s$, we get:
$\Pr[\text{a given partition has size} \geq cn/p] \leq \Pr[X < s] \leq 1/n^{16c'}$.
$\Rightarrow \Pr[\text{at least one partition has size } \geq cn/p] \leq \sum_{i=1}^{p} 1/n^{16c'} = p/n^{16c'}$ (by union-bound).
$\Rightarrow \Pr[\text{no partition has size} \geq cn/p] \geq 1 - p/n^{16c'}$.
Since p is at most $\mathcal{O}(n)$, no partition has size $\geq cn/p$ *whp*. ◀

Thus each partition has size at most cn/p for some constant c *whp*. Merging a partition A with m sorted subgroups has work $T(m) = 2T(m/2) + \mathcal{O}(m) + C(A)$, where $C(A)$ is the number of symbol comparisons made in MERGE. Omitting $C(A)$ from $T(m)$, this solves to $\mathcal{O}(cn/p \log p) = \mathcal{O}(n/p \log p)$ *whp*.

Total symbol comparisons. A subproblem in the algorithm is some subarray of the SA that can be processed independently of the other subarrays in a given step. Let X be some subproblem in either of the two steps: local-sorting and partition-merging. For the local-sorting case, sorting X with MERGE-SORT consists of $\log n/p$ recursion-levels. For the partition-merging case, the MERGE-PARTITION procedure for X executes in $\log p$ recursion-levels. We label the bottom-most level as level 0, and count the levels upwards in the recursion-tree.

Let $x \in X$ be a suffix. At any given level i, x is present in exactly one MERGE-SORT (or MERGE-PARTITION) instance executing on X. Let x'_i be the suffix that immediately precedes x in the output of that MERGE-SORT (or MERGE-PARTITION) instance, and let $L_i(x) = \text{LCP}(x, x'_i)$. If x is the first suffix in the output, then x'_i is the empty suffix. We prove the following.

▶ **Theorem 2.** *In* MERGE-SORT *and* MERGE-PARTITION, $L_i(x) \leq L_{i+1}(x)$ *for a suffix* x *at each recursion level* $i \in [0, d-1)$*, where* d *is the depth of the recursion-tree.*

Proof. Let x be present in the MERGE-SORT (or MERGE-PARTITION) instance M at level $i+1$, and say M spawns the two instances M_l and M_r. M_l and M_r are at level i, and x is present in exactly one of them. Let it be M_l.

Now, x'_{i+1} is either x'_i i.e. the same suffix preceding x in M_l, or some other suffix y from M_r. If $x'_{i+1} = x_i$, then $L_{i+1}(x) = L_i(x)$, and the claim holds.

In the other case, the output array of M has the following form: $[\ldots, x'_i, \ldots, y, x, \ldots]$. Suppose that the claim is false, i.e. $L_i(x) > L_{i+1}(x)$. Which is, $\mathrm{LCP}(x, x'_i) > \mathrm{LCP}(x, y)$. $\mathrm{LCP}(x, y) < \mathrm{LCP}(x, x'_i)$ implies that x'_i and y share the same prefix of length $l = \mathrm{LCP}(x, y)$, and mismatch first at index l. Let $c_x, c_{x'_i}$, and c_y be the l'th symbol in x, x'_i, and y resp. As $y > x'_i$ in the output, $c_y > c_{x'_i}$. Besides, since x and y first mismatch at the l'th symbol and $x > y$, $c_x > c_y$. Thus $c_x > c_{x'_i}$. $\mathrm{LCP}(x, x'_i) > l$ implies that the l'th symbols in x and x'_i are the same, i.e. $c_x = c_{x'_i}$. Thus we get $c_x > c_{x'_i}$ and $c_x = c_{x'_i}$, resulting in a contradiction. Hence, $\mathrm{LCP}(x, y) \leq \mathrm{LCP}(x, x'_i)$. ◀

Theorem 3 provides a bound on the number of total comparisons made across all the Merge-Sort and Merge-Partition instances in the algorithm execution.

▶ **Theorem 3.** *The total number of symbol comparisons made across all the* Merge-Sort *and* Merge-Partition *instances in* CaPS-SA *for an n-length text is $\mathcal{O}(n \log n + \sum_{i=1}^{n-1} L_i)$ w.h.p, where L is the output* LCP*-array.*

Proof. Symbol comparisons occur only as part of the Merge procedure in both Merge-Sort and Merge-Partition. Given two sorted lists of suffixes X and Y along with their LCP-arrays, the $\mathrm{Merge}(X, Y, L_X, L_Y, Z, L_Z, T)$ procedure iterates through the X_i's and Y_j's and fills in the Z_k's in the sorted order, along with their LCP-values in L_{Zk}. Without loss of generality, suppose that $X_i < Y_j$ is found at some iteration. Then the next iteration compares X_{i+1} and Y. Let $l_x = \mathrm{LCP}(X_{i+1}, X_i)$ and $m = \mathrm{LCP}(X_i, Y_j)$, LCP of the last compared pair. Symbols from the suffixes X_{i+1} and Y_j will only be compared iff $l_x = m$ holds. In this case, we compute $n = \mathrm{LCP}(X_{i+1}, Y_j)$ with exactly $n - m + 1$ symbol comparisons. The $+1$ term is due to the first mismatching symbol pair. m is set to n for the next iteration. We argue that before the new $m = n$ value is assigned as the LCP-value in the output LCP-array L_Z in some future iteration, it remains unchanged.

In the next iteration, if case (1), i.e. $l_x > m$ holds, then m remains unchanged. If case (2), i.e. $l_x < m$ holds, then m is assigned at output L_{Zk+1}. In the event of case (3), either l_x or m is assigned to L_{Zk+1}, and these are equal in this case. If X has been depleted during the merge while Y still has remaining elements, the current m is assigned as the LCP-value for the first of the remaining elements from Y.

Thus, whenever symbol comparisons are done in case (3) of merge, it results in a new value $m' \geq m$ for the variable m. $m = m'$ persists until m' has been assigned as the LCP-value for some merged output. Thus the number of matching symbol comparisons made in case (3) accumulates in the LCP-values at the output.

All the LCP-values start out with 0 at Merge-Sort. Theorem 2 states that the LCP-value associated to a given suffix can never decrease while winding up the recursion-trees of Merge-Sort and Merge-Partition. Thus the sum $\sum_{i=1}^{n-1} L_i$ of the final LCP-values in the SA is the total number of matching symbol comparisons made across all the Merge-Sort and Merge-Partition executions.

The extra mismatching comparison in case (3) of Merge costs $\mathcal{O}(1)$. In the worst case, this case occurs in each iteration of Merge. Omitting the matching symbol comparisons, a Merge-Sort or a Merge-Partition instance working on m elements incurs $T(m) = 2T(m/2) + \mathcal{O}(m)$ mismatches in the worst case. This solves to $p \times \mathcal{O}(n/p \log n/p)$ and $p \times \mathcal{O}(n/p \log p)$ *whp* for the p Merge-Sorts and Merge-Partitions, resp. Thus $\mathcal{O}(n \log n)$ mismatching symbol comparisons are made *whp*. ◀

Total work. Locally sorting the p subarrays cost $p \times \mathcal{O}(n/p \log n/p) = \mathcal{O}(n \log n/p) =$ work without the symbol comparisons. Omitting the symbol comparisons in sorting the sampled pivots, the pivot selection step has $\mathcal{O}(sp \log sp)$ work. In the collation step, there are $p(p-1)$

binary searches, costing $\mathcal{O}(p^2 n \log n/p)$ work in the worst-case, and $\mathcal{O}(p^2(n + \log n/p))$ in practice. Merging the p partitions separately cost $p \times \mathcal{O}(n/p \log p) = \mathcal{O}(n \log p)$ whp without the symbol comparisons.

The total number of symbol comparisons in the local-sort and the partitions-merge steps is $\mathcal{O}(n \log n + \sum_{i=1}^{n-1} L_i)$ *whp* as per Theorem 3, where L is the output LCP-array. In sorting the sampled suffixes, the number of symbol comparisons done is also bounded by this [2].

We note that the algorithm requires on the order of $4w|T|$ bytes of working space, where $w \in \{4, 8\}$ is the numerical size used to store SA and LCP values.

3.2.2 Parallelization

Our implementation fully parallelizes the work across the different partitions. Within a partition, we perform recursive calls to MERGE-SORT in parallel, but perform the MERGE procedure serially. We show the following theorem about the depth of our algorithm:

▶ **Theorem 4.** *The overall depth of the algorithm is* $\mathcal{O}((n/p) \log n)$ whp.

Proof. The dominant factor in the merge algorithm is the depth of the MERGE routine, which simply performs a linear number of comparisons in the input size. The depth of a comparison is $\mathcal{O}(1)$ in cases 1 and 2 of Figure 2, and requires a string comparison in the final case.

The string comparison can be parallelized work-efficiently (i.e., in the same work as a serial character-by-character comparison) by using a simple prefix-doubling strategy. In more detail, the comparison algorithm works in rounds comparing 2^i characters in the i'th round until a mismatch occurs. Clearly for strings of length $\mathcal{O}(n)$ only $\mathcal{O}(\log n)$ rounds are required, and thus the overall work is asymptotically the same as the serial algorithm, and the depth is $\mathcal{O}(\log n)$. Thus, for merging two sorted arrays in the algorithm, each of size k, we require a depth of $\mathcal{O}(k \log n)$.

Putting these facts together, for a single call to the MERGE-SORT routine, we have a recurrence of the form $D(k) = D(k/2) + \mathcal{O}(k \log n)$, which is root dominated and solves to $\mathcal{O}(k \log n)$. Since our algorithm is parallelized across different partitions, and by Theorem 1 each partition has size at most $\mathcal{O}(n/p)$, the overall depth of the algorithm is $\mathcal{O}((n/p) \log n)$ *whp*. ◀

We note that the depth is not poly-logarithmic, as in the classic parallel MERGE-SORT. However, the amount of parallelism generated by our algorithm is more than enough to keep the processors all busy in practice. Indeed, we note that many samplesort implementations use a similar strategy in practice and use a serial sort within each partition, and thus also do not have poly-logarithmic depth in practice. In our implementation, we exploit parallelism using the parallel primitives and the work-stealing scheduler from `ParlayLib` [9].

3.2.3 Optimizations

We applied a number of optimizations into the implementation of the algorithm that provide practical speedups. We make use of vectorization support using `AVX` instructions from modern processors to speed up the computation of the LCP(X, Y) routine used in the MERGE-procedure and in the binary searches in locating pivots.

[2] $\mathcal{O}(sp \log sp + \sum_{i=1}^{sp} L_{p\,i})$ comparisons are performed, where L_p is the LCP-array of the sorted samples.

In the proposed Merge-Sort and the Merge-Partition procedures in the algorithm, we have nested parallelism for their recursive invocations. This is applied in the implementation up-to some fixed granularity, due to the associated overhead of scheduling small tasks.

In the binary searches for the sampled pivots in each sorted subarray, instead of searching for the appropriate position of an entire pivot suffix, we look for a fixed-length prefix of the pivot. This helps reduce the total work associated to locating the pivots, with an associated trade-off with the final partition sizes. With sufficiently large prefix lengths, the partition sizes do not get significantly affected in our observation.

4 Results

We performed a number of experiments to characterize the performance of the CaPS-SA algorithm and its implementation. We evaluated its performance compared to the available implementations of two leading methods for SA construction: Parallel-divsufsort [32] and Parallel-DC3 [3]. We assessed its ability to construct SA and LCP-arrays on a number of genomic datasets.

Next, we evaluated the parallel scaling of the algorithm. Then we explore the idea of *Bounded-context suffix arrays*, and the performance of CaPS-SA for various prefix-context lengths.

A varied collection of datasets has been used in the experiments. Table 1 delineates the pertinent characteristics of the datasets. We follow [46] by removing `N`-repeats, which occur when the sequence underlying a region of the assembly cannot be resolved. We verified the correctness of the implementation by cross-checking its output against from that of Parallel-divsufsort.

Computation system. The experiments have been performed on a server having two Intel Xeon E5-2699 2.20 GHz CPUs with 44 cores in total and 512 GB of 2.40 GHz DDR4 RAM. The system is run with Ubuntu 20.04.4 (GNU/Linux 5.4.0-132-generic x86_64). The SA and LCP-array construction times and the maximum memory usages of the tools were measured with the GNU `time` command. For the large datasets Axolotl genome dataset, we used machines with two AMD EPYC 7313 CPUs with 32 cores in total and 2 TB of DDR4 RAM, running on Red Hat Enterprise Linux 8.7 (GNU/Linux 4.18.0-425.19.2.el8_7.x86_64).

4.1 Dataset characteristics

Table 1 provides some pertinent characteristics of the datasets used. The GRCh38 dataset is the Human Build 38 patch release 13 version of the human genome reference from the Genome Reference Consortium[3], which is a chromosome-level assembly of the full genome. The T2T dataset is the latest T2T CHM13v2.0 Telomere-to-Telomere assembly of the CHM13 cell line with chromosome Y from NA24385, from the T2T consortium, which is a complete genome-level assembly of the genome [40]. Together, these two human datasets represent what we imagine may be a *typical* use-case for genome construction in the context of a tool like STAR [12]. Though largely similar, the CHM13 assembly has resolved telomeric and centromeric regions, and more complete coverage, specifically in highly-repetitive regions. Thus, we expect it represents a more challenging problem instance for suffix array construction.

[3] `https://www.ncbi.nlm.nih.gov/grc`

The CdBG (Compacted de Bruijn Graph) dataset is the collection of the maximal unitigs extracted from the de Bruijn graph (with k-mer size 27) of the human sequencing read set NIST HG004 (SRA3440461 – 95) [52] by Cuttlefish 2 [28]. This dataset represents a potential use-case where one may wish to build an index for the sequence stored in the CdBG data structure. The ability to index the CdBG has proven useful in many contexts [10], and the SA can provide one possible index for providing efficient lookup over the sequence contained in the CdBG.

The great white shark dataset is the genome reference of *Carcharodon carcharias* [36] and the axolotl dataset is the genome reference sequence of *Ambystoma mexicanum* [47]. These represent large problem instances, where one may wish to build the SA on large reference genomes.

Table 1 Dataset statistics: number of bases, mean LCP, and standard deviation (rounded to nearest whole number) of the final LCP-array.

Dataset	Size	Mean LCP	Std. Dev. of LCP
Human (GRCh38)	2,945,849,068	3,807	72,678
Human (T2T)	3,117,292,071	2,518	61,987
CdBG (Human reads)	3,993,272,308	18	6
Great white shark	4,267,160,925	109	1,192
Axolotl	28,203,219,824	49	160

4.2 SA and LCP-array construction

We evaluate the performance of CaPS-SA in constructing the SA and the LCP-array of a number of genomic datasets, compared to the SA construction performance of: 1. Parallel-divsufsort [32] and 2. Parallel-DC3 [3]. Table 2 contains the results of the benchmarking. As the state-of-the-art sequential benchmark, we note the performance of the `divsufsort` implementation from PBBS [3].

Table 2 Time- and memory-performance results for constructing SAs (and LCP-array in case of CaPS-SA) with 32 threads. `divsufsort` is shown as a serial benchmark. Time is reported in seconds, and the memory usages are reported in GBs in parentheses. Best performances among the parallel algorithms in each instance are highlighted. Parallel-DC3 and `divsufsort` could not be run on Axolotl because we could not modify the PBBS code-base to accommodate the large numerical size.

Dataset	CaPS-SA	Parallel-divsufsort	Parallel-DC3	`divsufsort`
Human (GRCh38)	**131** (47)	173 (**32**)	250 (110)	273 (17)
Human (T2T)	**141** (50)	199 (**34**)	261 (115)	319 (18)
CdBG (Human reads)	**138** (64)	291 (**44**)	323 (126)	409 (23)
Great white shark	**173** (71)	278 (**48**)	381 (135)	438 (24)
Axolotl	**900** (910)	1068 (**326**)	-	-

We note that CaPS-SA executes significantly faster than the other parallel algorithms in all the instances, whereas Parallel-divsufsort uses the least amount of memory. Interestingly for the smaller datasets CaPS-SA does not require much more memory than Parallel-divsufsort despite constructing both the SA and LCP. Memory usage could be improved by bit-packing the indexes, or through the extension to an external memory algorithm.

4.3 Parallel Scaling

In order to assess how sensitive runtime is to parallelism we evaluated CaPS-SA against Parallel-DC3 and Parallel-divsufsort as the number of threads increased. We report the results in Figure 3, which illustrated that CaPS-SA exploits parallelism better – becoming the fastest method as the thread count becomes high despite doing more work asymptotically.

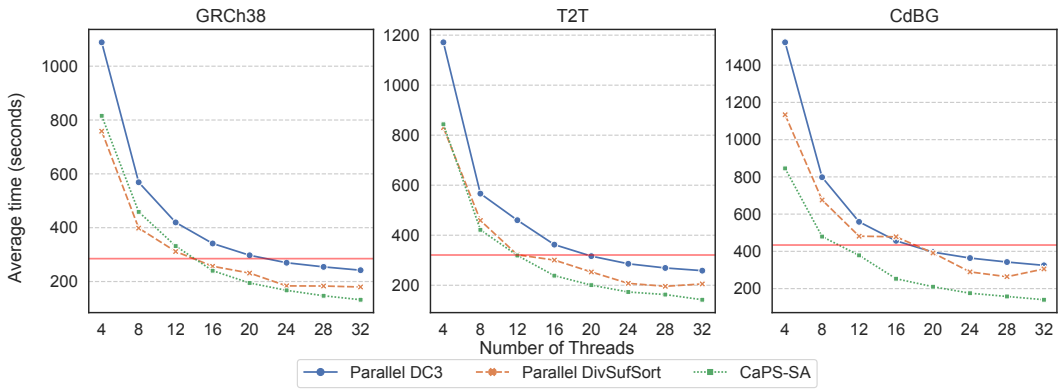


Figure 3 Runtimes of CaPS-SA, Parallel-divsufsort, and Parallel-DC3 as thread count increases for GRCh38, T2T, and CdBG. `divsufsort` runtime is in red.

On the GRCh38 and T2T datasets, CaPS-SA and Parallel-divsufsort become faster than `divsufsort` at around the same number of threads, after which CaPS-SA becomes faster. On CdBG, CaPS-SA sees gains from parallelism over the best serial program with approximately half the number of threads. This may be because CdBG has fewer long common-prefixes, and so CaPS-SA achieves closer to its best-case work.

4.4 Cache Performance

The samplesort-based design of CaPS-SA optimizes cache-performance. In order to evaluate the empirical cache behavior of CaPS-SA as compared to other algorithms for SA construction, we profiled the programs on the GRCh38 and the T2T reference genomes. Because cache-behavior can degenerate as parallelism increases, we evaluate it across 1, 16, and 32 threads. The results in Table 3 show that CaPS-SA outperforms other parallel SA indexing programs by an order of magnitude. All measurements were taken with the Linux `perf` command.

Table 3 Cache-miss rates (in %) for compared methods on GRCh38 and T2T datasets with respect to number of threads. Reported numbers are averaged over 5 runs to obviate operating system jitter. Lower is better, and the best result is highlighted.

	Human (GRCh38)			Human (T2T)		
Method	**1 thr**	**16 thr**	**32 thr**	**1 thr**	**16 thr**	**32 thr**
CaPS-SA	4.83	**4.92**	**4.99**	7.34	**7.61**	**7.81**
Parallel-divsufsort	21.98	25.03	26.31	37.38	37.98	38.68
Parallel-DC3	34.75	35.91	37.43	35.04	36.23	37.66
`divsufsort`	**0.93**	–	–	**0.99**	–	–

`divsufsort` is somewhat more cache-efficient than CaPS-SA, but interestingly its parallelization in Parallel-divsufsort has significantly worse caching performance than both `divsufsort` and CaPS-SA. It is possible that incorporating some of the implementation details of `divsufsort` could provide further improvements to the cache-performance of CaPS-SA.

4.5 Bounded-context SA Construction

By virtue of organizing all suffixes of the underlying text T, the suffix array provides the powerful ability to efficiently search for query patterns of *any* length in the text. While this capability arises naturally from the definition of the SA, such flexibility is rarely needed in the SA's most common applications in genomics. Specifically, when used to efficiently find *seed* sequences from a genomic read, the maximum length of the query is often very short. Many modern aligners use seed lengths in the range of 15–31, and even with the maximum mappable prefix concept used by STAR [12], the query length is bounded above by the error-free prefix length of the remainder of the read (rarely more than ~ 100 nucleotides).

As such, indices that can provide efficient lookup and locate queries for patterns less than some maximum length, say k, are often very useful in this context. For example, the k-BWT data structure [45, 41, 11] builds a transform of the text that organizes character occurrences by their *bounded context* (in this case, their right context of length k). This allows the index to be built efficiently, since rotations of the text need not have their relative orders resolved beyond their initial length k contexts, while simultaneously allowing efficient and correct query for any pattern length $\leq \mathsf{k}$.

Here, we experiment with an analogous version of the SA– the *bounded-context SA*. Specifically, the bounded-context SA of order k resolves the lexicographic order of all suffixes of the text up to (and including) their prefixes of length k. If a pair of suffixes share a prefix of length $\geq \mathsf{k}$, then they may appear in an arbitrary relative order within the bounded-context SA of order k. Without any meaningful modifications to the query algorithms, this variant of the SA allows locating all occurrences of queries of any length $\leq \mathsf{k}$ in the text. Such a variant of the SA is very straightforward to construct using CaPS-SA, as we simply declare equal any suffixes that are equal up to (or beyond) their length k prefixes. At the same time, this variant can be more efficient to construct using our algorithm, as the context length k places a strict upper bound on the number of comparisons we must perform when attempting to determine the relative order of a pair of suffixes. Specifically, it follows directly from Theorem 3 that CaPS-SA performs at most $\mathcal{O}(\mathsf{n} \log \mathsf{n} + \mathsf{nk})$ character comparisons, in the worst case, when constructing the context-bounded SA of order k. In Table 4, we report the time required to construct the context-bounded SA of orders 64 and 256 of the GRCh38 and CHM13 human genome assemblies compared to the time required to construct the standard (full-context) SA. As expected, the bounded-context SA can be constructed substantially faster than the full-context SA.

Table 4 Timings (in seconds) in constructing the bounded-context SA with various orders.

Dataset	**full-context**	64	256
Human (GRCh38)	132	104	100
Human (T2T)	144	105	103

5 Conclusion

In this manuscript, we introduced a new method, CaPS-SA, for parallel SA and LCP-array construction. CaPS-SA displays very good cache performance (i.e. very low cache miss rate), and scales well to many threads. As a result, CaPS-SA is able to outperform existing state-of-the-art parallel SA construction algorithms like Parallel-divsufsort and Parallel-DC3 on genomic datasets. At the same time, CaPS-SA is substantially simpler than existing state-of-the-art algorithms. This simplicity eases implementation, and leads to many opportunities for further future improvements. Likewise, CaPS-SA provides the LCP-array directly as a byproduct of SA construction, and does not require a separate algorithm to produce this useful auxiliary data structure. We hope that will prove to be useful in utilities where parallel SA construction is a core subproblem, and also hope that the relatively straightforward algorithm will benefit from further optimizations, enhancements, and alternative implementations within the community.

As CaPS-SA scales well with the level of available parallelism, and performs well for large references, we expect that it will provide a useful option for tools that seek to build the SA in parallel environments. In addition to the *time* taken to construct the SA or the LCP array, another consideration is the memory (specifically the RAM) required for construction. One approach to improve the *memory-scalability* of SA construction algorithms is to develop external-memory construction algorithms. For example, `pSAscan` [24] is a state-of-the-art external-memory algorithm for SA construction. Such approaches make use of external-memory (i.e. disk) and algorithms that access and construct the SA in a structured way are likely amenable to external-memory variants.

We note that, though we have not explored it in this manuscript, CaPS-SA is highly-amenable to external memory implementation. This is because the initial partitioning generates many small subproblems that can be solved independently – i.e. some subproblem can be paged into RAM while others remain on disk. Pivot sampling from the subproblems can be done through a similar paging process. Likewise, after pivot selection, many approximately equal-sized partitions will be created, and these sub-problems, which target specific output intervals of the final suffix array, can be solved independently and in parallel with the relevant data for only a working subset of partitions paged into RAM with the remaining partitions residing on disk. Further, given a sufficiently fine-grained partitioning, the algorithm can likely provide tight controls on the required working memory. As more RAM use is allowed, a larger number of partitions will be allowed in RAM at once, and our algorithm will be able to better make use of available parallelism. On the other hand, as the maximum allowed RAM usage is restricted, fewer partitions will be present in memory at once, potentially limiting parallelism, but adhering to the requested RAM constraints. In practice, we believe that, so long as a sufficiently fine-grained partitioning is used, external-memory variants of our algorithm will still be able to efficiently make use of many threads while still substantially reducing the required working memory. We leave the efficient implementation of an external-memory variant of CaPS-SA to future work.

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86, 2004.

2 Mohammed Alser, Jeremy Rotman, Dhrithi Deshpande, Kodi Taraszka, Huwenbo Shi, Pelin Icer Baykal, Harry Taegyun Yang, Victor Xue, Sergey Knyazev, Benjamin D. Singer, Brunilda Balliu, David Koslicki, Pavel Skums, Alex Zelikovsky, Can Alkan, Onur Mutlu, and Serghei Mangul. Technology dictates algorithms: recent developments in read alignment. *Genome Biology*, 22(1):249, August 2021. doi:10.1186/s13059-021-02443-7.

3 Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. The problem-based benchmark suite (PBBS), v2. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 445–447, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503221.3508422.

4 Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-Place Parallel Super Scalar Samplesort (IPSSSSo). In *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ESA.2017.9.

5 Timo Bingmann. Scalable string and suffix sorting: Algorithms, techniques, and tools. *arXiv preprint arXiv:1808.00963*, 2018.

6 Timo Bingmann, Patrick Dinklage, Johannes Fischer, Florian Kurpicz, Enno Ohlebusch, and Peter Sanders. *Scalable Text Index Construction*, pages 252–284. Springer Nature Switzerland, Cham, 2022. doi:10.1007/978-3-031-21534-6_14.

7 Timo Bingmann, Andreas Eberle, and Peter Sanders. Engineering parallel string sorting. *Algorithmica*, 77:235–286, 2017.

8 Timo Bingmann and Peter Sanders. Parallel string sample sort. In *Algorithms–ESA 2013: 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings 21*, pages 169–180. Springer, 2013.

9 Guy E Blelloch, Daniel Anderson, and Laxman Dhulipala. Parlaylib-a toolkit for parallel algorithms on shared-memory multicore machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 507–509, 2020.

10 Rayan Chikhi, Jan Holub, and Paul Medvedev. Data structures to represent a set of k-long DNA sequences. *ACM Comput. Surv.*, 54(1), March 2021. doi:10.1145/3445967.

11 J Shane Culpepper, Matthias Petri, and Simon J Puglisi. Revisiting bounded context block-sorting transformations. *Software: Practice and Experience*, 42(8):1037–1054, 2012.

12 Alexander Dobin, Carrie A Davis, Felix Schlesinger, Jorg Drenkow, Chris Zaleski, Sonali Jha, Philippe Batut, Mark Chaisson, and Thomas R Gingeras. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

13 M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 137–143, 1997. doi:10.1109/SFCS.1997.646102.

14 Johannes Fischer and Florian Kurpicz. Dismantling divsufsort. In *Prague Stringology Conference 2017*, page 62, 2017.

15 Johannes Fischer and Florian Kurpicz. *Lightweight Distributed Suffix Array Construction*, pages 27–38. Society for Industrial and Applied Mathematics, 2019. doi:10.1137/1.9781611975499.3.

16 Patrick Flick and Srinivas Aluru. Parallel distributed memory construction of suffix and longest common prefix arrays. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2807591.2807609.

17 W Donald Frazer and Archie C McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM (JACM)*, 17(3):496–507, 1970.

18 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

19 Scott Hazelhurst and Zsuzsanna Lipták. KABOOM! a new suffix array based algorithm for clustering expression data. *Bioinformatics*, 27(24):3348–3355, December 2011.

20 Lucian Ilie, Farideh Fazayeli, and Silvana Ilie. HiTEC: accurate error correction in high-throughput sequencing data. *Bioinformatics*, 27(3):295–302, February 2011.

21 Hideo Itoh and Hozumi Tanaka. An efficient method for in memory construction of suffix arrays. In *6th International Symposium on String Processing and Information Retrieval. 5th International Workshop on Groupware (Cat. No. PR00268)*, pages 81–88. IEEE, 1999.

22 Juha Kärkkäinen and Dominik Kempa. Engineering a lightweight external memory suffix array construction algorithm. *Mathematics in Computer Science*, 11:137–149, 2017.

23 Juha Kärkkäinen and Dominik Kempa. Engineering external memory LCP array construction: Parallel, in-place and large alphabet. In *16th International Symposium on Experimental Algorithms (SEA 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

24 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Parallel external memory suffix sorting. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Combinatorial Pattern Matching*, pages 329–342, Cham, 2015. Springer International Publishing.

25 Juha Kärkkäinen, Dominik Kempa, Simon J Puglisi, and Bella Zhukova. Engineering external memory induced suffix sorting. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 98–108. SIAM, 2017.

26 Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming: 30th International Colloquium, ICALP 2003 Eindhoven, The Netherlands, June 30–July 4, 2003 Proceedings 30*, pages 943–955. Springer, 2003.

27 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006.

28 Jamshed Khan, Marek Kokot, Sebastian Deorowicz, and Rob Patro. Scalable, ultra-fast, and low-memory construction of compacted de bruijn graphs with Cuttlefish 2. *Genome Biology*, 23(1):190, September 2022. doi:10.1186/s13059-022-02743-6.

29 Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003 Morelia, Michoacán, Mexico, June 25–27, 2003 Proceedings 14*, pages 186–199. Springer, 2003.

30 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003 Morelia, Michoacán, Mexico, June 25–27, 2003 Proceedings*, pages 200–210. Springer, 2003.

31 Fabian Kulla and Peter Sanders. Scalable parallel suffix array construction. *Parallel Computing*, 33(9):605–612, 2007.

32 Julian Labeit, Julian Shun, and Guy E Blelloch. Parallel lightweight wavelet tree, suffix array and fm-index construction. *Journal of Discrete Algorithms*, 43:2–17, 2017.

33 Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting. In *String Processing and Information Retrieval: 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings*, pages 268–284. Springer, 2018.

34 Gang Liao, Longfei Ma, Guangming Zang, and Lin Tang. Parallel DC3 algorithm for suffix array construction on many-core accelerators. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1155–1158, 2015. doi:10.1109/CCGrid.2015.56.

35 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

36 Nicholas J. Marra, Michael J. Stanhope, Nathaniel K. Jue, Minghui Wang, Qi Sun, Paulina Pavinski Bitar, Vincent P. Richards, Aleksey Komissarov, Mike Rayko, Sergey Kliver, Bryce J. Stanhope, Chuck Winkler, Stephen J. O'Brien, Agostinho Antunes, Salvador Jorgensen, and Mahmood S. Shivji. White shark genome reveals ancient elasmobranch adaptations associated with wound healing and the maintenance of genome stability. *Proceedings of the National Academy of Sciences*, 116(10):4446–4455, 2019. doi:10.1073/pnas.1819778116.

37 Yuta Mori. divsufsort. https://github.com/y-256/libdivsufsort, 2015. Accessed on 1 May 2023.

38 Waihong Ng and Katsuhiko Kakehi. Merging string sequences by longest common prefixes. *IPSJ Digital Courier*, 4:69–78, 2008.

39 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE transactions on computers*, 60(10):1471–1484, 2010.

40 Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V Bzikadze, Alla Mikheenko, Mitchell R Vollger, Nicolas Altemose, Lev Uralsky, Ariel Gershman, et al. The complete sequence of a human genome. *Science*, 376(6588):44–53, 2022.

41 Matthias Petri, Gonzalo Navarro, J Shane Culpepper, and Simon J Puglisi. Backwards search in context bound text transformations. In *2011 First International Conference on Data Compression, Communications and Processing*, pages 82–91. IEEE, 2011.

42 Anton Pirogov, Peter Pfaffelhuber, Angelika Börsch-Haubold, and Bernhard Haubold. High-complexity regions in mammalian genomes are enriched for developmental genes. *Bioinformatics*, 35(11):1813–1819, 2019.

43 Simon J Puglisi, William F Smyth, and Andrew H Turpin. A taxonomy of suffix array construction algorithms. *acm Computing Surveys (CSUR)*, 39(2):4–es, 2007.

44 Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *Algorithms–ESA 2004: 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004. Proceedings 12*, pages 784–796. Springer, 2004.

45 M. Schindler. A fast block-sorting algorithm for lossless data compression. In *Proceedings DCC '97. Data Compression Conference*, pages 469–, 1997. doi:10.1109/DCC.1997.582137.

46 Anish Man Singh Shrestha, Martin C Frith, and Paul Horton. A bioinformatician's guide to the forefront of suffix array construction algorithms. *Briefings in bioinformatics*, 15(2):138–154, 2014.

47 Jeramiah J Smith, Nataliya Timoshevskaya, Vladimir A Timoshevskiy, Melissa C Keinath, Drew Hardy, and S Randal Voss. A chromosome-scale assembly of the axolotl genome. *Genome Res.*, 29(2):317–324, February 2019.

48 Michaël Vyverman, Bernard De Baets, Veerle Fack, and Peter Dawyndt. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29(6):802–804, March 2013.

49 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.

50 Yuzhen Ye, Jeong-Hyeon Choi, and Haixu Tang. RAPSearch: a fast protein similarity search tool for short reads. *BMC Bioinformatics*, 12(1):159, May 2011.

51 Kaiyuan Zhu, Alejandro A Schäffer, Welles Robinson, Junyan Xu, Eytan Ruppin, A Funda Ergun, Yuzhen Ye, and S Cenk Sahinalp. Strain level microbial detection and quantification with applications to single cell metagenomics. *Nature Communications*, 13(1):6430, 2022.

52 Justin M. Zook, David Catoe, Jennifer McDaniel, Lindsay Vang, Noah Spies, Arend Sidow, Ziming Weng, Yuling Liu, Christopher E. Mason, Noah Alexander, Elizabeth Henaff, Alexa B.R. McIntyre, Dhruva Chandramohan, Feng Chen, Erich Jaeger, Ali Moshrefi, Khoa Pham, William Stedman, Tiffany Liang, Michael Saghbini, Zeljko Dzakula, Alex Hastie, Han Cao, Gintaras Deikus, Eric Schadt, Robert Sebra, Ali Bashir, Rebecca M. Truty, Christopher C. Chang, Natali Gulbahce, Keyan Zhao, Srinka Ghosh, Fiona Hyland, Yutao Fu, Mark Chaisson, Chunlin Xiao, Jonathan Trow, Stephen T. Sherry, Alexander W. Zaranek, Madeleine Ball, Jason Bobe, Preston Estep, George M. Church, Patrick Marks, Sofia Kyriazopoulou-Panagiotopoulou, Grace X.Y. Zheng, Michael Schnall-Levin, Heather S. Ordonez, Patrice A. Mudivarti, Kristina Giorda, Ying Sheng, Karoline Bjarnesdatter Rypdal, and Marc Salit. Extensive sequencing of seven human genomes to characterize benchmark reference materials. *Scientific Data*, 3(1):160025, June 2016. doi:10.1038/sdata.2016.25.

A Methods

Pseudo-codes for the procedures absent in the main text, Sample-Pivots, Collate-Partitions, and Compute-Boundary-LCPs are provided in the following.

Sample-Pivots(SA, T, p).

```
1  s = 32 ln |T| // sampling factor
2  V, V', w1, w2 = arrays of size s × p

   // sample pivots from each sorted subarray
3  for i ∈ [0, p) in parallel
4      b = im, e = (i + 1)m // range of the i'th subarray
5      sample s pivots from SA[b:e) into V[b:e) w/o replacement

6  V' = V
7  Merge-Sort(V, V', w1, w2, T)
8  sample (p − 1) pivots from V' into V
9  return V
```

COLLATE-PARTITIONS(SA, SA', L, L', V, T, p).

```
1  P = 2-D array of size p × (p + 1) // P_{i,j}: location of j'th pivot in i'th subarray

2  m = |T|/p // size of the subarrays
3  for i ∈ [0, p) in parallel // for each sorted subarray
4      b = im, e = (i + 1)m // range of the i'th subarr
5      P_{i,0} = 0, P_{i,p} = m // two sentinel pivots
6      for j ∈ [0, p − 1) // for each pivot
7          P_{i,j+1} = BINARY-SEARCH(SA_{[b:e)}, V_j)

8  S = array of size p
9  for j ∈ [0, p) in parallel
10     S_j = Σ_{i=0}^{p−1} (P_{i,j+1} − P_{i,j}) // sum size of the j'th sub-subarrays
11 S = PREFIX-SUM-SCAN(S) // S_j: index of the j'th partition in the final SA

12 R = 2-D array of size p × (p + 1) // R_{j,i}: index of the i'th sub-subarray in partition j
13 for j ∈ [0, p) in parallel // for each partition
14     let Y : SA'_{[S_j:S_{j+1})} // location for partition j
15     R_{j,0} = 0
16     for i ∈ [0, p) // for each sorted subarray
17         let A : SA_{[im,(i+1)m)} // i'th subarray
18         let X : A_{[P_{i,j}:P_{i,j+1})} // j'th sub-subarray in A
19         COPY(X, Y_{[R_{j,0}:)})
20         R_{j,i+1} = R_{j,i} + |X|
21 return (S, R)
```

COMPUTE-BOUNDARY-LCPS(SA, p, S, L, T).

```
1  for j ∈ [1, p) in parallel
2      s = SA_{S_j}, t = SA_{S_j−1}
3      L_{S_j} = LCP(T_{[s:]}, T_{[t:]})
```

Compression Algorithm for Colored de Bruijn Graphs

Amatur Rahman
Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA, USA

Yoann Dufresne
Institut Pasteur, Université Paris Cité, G5 Sequence Bioinformatics, Paris, France
Institut Pasteur, Université Paris Cité, Bioinformatics and Biostatistics Hub, F-75015 Paris, France

Paul Medvedev
Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA, USA
Department of Biochemistry and Molecular Biology, The Pennsylvania State University, University Park, PA, USA
Huck Institutes of the Life Sciences, The Pennsylvania State University, University Park, PA, USA

Abstract

A colored de Bruijn graph (also called a set of k-mer sets), is a set of k-mers with every k-mer assigned a set of colors. Colored de Bruijn graphs are used in a variety of applications, including variant calling, genome assembly, and database search. However, their size has posed a scalability challenge to algorithm developers and users. There have been numerous indexing data structures proposed that allow to store the graph compactly while supporting fast query operations. However, disk compression algorithms, which do not need to support queries on the compressed data and can thus be more space-efficient, have received little attention. The dearth of specialized compression tools has been a detriment to tool developers, tool users, and reproducibility efforts. In this paper, we develop a new tool that compresses colored de Bruijn graphs to disk, building on previous ideas for compression of k-mer sets and indexing colored de Bruijn graphs. We test our tool, called ESS-color, on various datasets, including both sequencing data and whole genomes. ESS-color achieves better compression than all evaluated tools and all datasets, with no other tool able to consistently achieve less than 44% space overhead.

2012 ACM Subject Classification Applied computing → Computational biology

Keywords and phrases colored de Bruijn graphs, disk compression, k-mer sets, simplitigs, spectrum-preserving string sets

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.17

Supplementary Material *Software (Source Code)*: `https://github.com/medvedevgroup/ESSColor`

Funding This material is based upon work supported by the National Science Foundation under grant nos. 2138585 and 1931531. Research reported in this publication was also supported by the National Institutes of Health under Grant NIH R01GM146462 (to P.M.). A.R. was supported by the National Institutes of Health Computation, Bioinformatics, and Statistics (CBIOS) training program. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

Acknowledgements We would like to thank R. Chikhi for helpful discussions.

1 Introduction

Modern methods for analyzing biological sequences often reduce the input dataset to a set of short, fixed length strings called k-mers. When working with a collection of such datasets $E = (E_0, \ldots, E_{|E|-1})$, it is fruitful to represent them as one union set of k-mers and, for each k-mer, the indices of the datasets to which it belongs. The set of indices of each k-mer is referred to as its color *class*, and E is referred to as a *colored de Bruijn graph* [13]. A colored de Bruijn graph (cdBG) is commonly used to represent a sequence database, such as a collection of sequencing experiments or a collection of assembled genomes. For example, it is used by tools for inferring phylogenies [29], quantification of RNA expression [23], and studying the evolution of antimicrobial resistance [4].

As sequence database sizes grow to petabytes [22], the cost of storing or transferring the data (e.g. on Amazon Web Services or in-house compute infrastructure) has underscored the need for efficient disk compression algorithms. Such costs are often prohibitive for smaller labs and, even for larger labs, limit the scale of data that can be analyzed. Large file sizes also hamper tool development, which relies on iterative loading/copying/modifying data, and reproducibility efforts, which require downloading and storing the data. For example, storing the 31-mers from 450,000 microbial genomes in compressed form takes about 12 Tb [4]. Unfortunately, there has not been a lot of work to develop methods for disk compression of colored de Bruijn graphs.

In contrast to disk compression, indexing cdBGs have received much attention [19]. A slew of data structures have been developed, optimizing various metrics such as index size, construction time, or query time (see the survey [19] and its follow up [18]). Indexing data structures exploit the structure of cdBGs and use clever tricks to compress the color classes of similar k-mers. But they also carry a space overhead to efficiently support queries; since this is not needed for disk compression, indexing data structures are usually not competitive with custom made cdBG compression methods.

The simplest option for compressing a cdBG is to compress each color (i.e. dataset) independently, using a compression tool designed for a single set of k-mers (e.g. [25]). This approach can work well when k-mers tend to not be shared among colors. However, most cdBGs have a large overlap between the k-mers of various colors. In such cases, independently compressing each color does not exploit the properties of cdBGs and, as we show in this paper, results in subpar compression ratios. There exist two tools designed specifically for disk compression of cdBGs. The first tool is unfortunately limited to only three k-mer sizes [16]. The second tool, called GGCAT [7] is a space efficient indexing method that, while not originally evaluated in this regard, turns out to also be a good disk compression method when combined with a generic post-compression step.

In this paper, we design, implement, and evaluate an algorithm ESS-color for the disk compression of cdBGs. We build upon the idea of spectrum-preserving string sets [26, 6, 5] and the followup compression format for a k-mer set [25], called *ESS*. By constructing an ESS of the union of k-mers in E, we represent the k-mer sequences themselves compactly. We exploit the fact that consecutive k-mers in an ESS have similar color classes in order to efficiently compress the color vectors of each k-mer.

We evaluate ESS-color on a variety of datasets, including bacteria, fungi, human, and including whole genome sequencing data, metagenome sequencing data, and whole assembled genomes. ESS-color achieves better compression than all evaluated tools and on all datasets, with all other tools using $\geq 44\%$ more space on at least one of our datasets. On some datasets the improvement over all other tools is quite large, e.g. for a gut metagenome, all the other tools use at $\geq 27\%$ more space than ESS-color. Compressing each color independently uses between 1.2x and 6.9x more space than ESS-color. The absolute compression ratio is more

than 26x on datasets of assembled genomes and between 1.4x and 8.7x on datasets from sequencing experiments. The software is available at `http://github.com/medvedevgroup/ESSColor`.

2 Preliminaries

In this section we give some important definitions. Please refer to Figure 1 for examples of the introduced concepts.

Strings

A string of length k is called a *k-mer*. We assume k-mers are over the DNA alphabet. A string x is canonical if it is the lexicographically smaller of x and its reverse complement. Let K be a set of k-mers. A *spectrum-preserving string set (SPSS)* of K is a set of strings S such that each string $s \in S$ is at least k characters long, every k-mer that appears in S appears exactly once, and the set of k-mers that appear in S is K [26, 6, 5]. For example, if $K = \{ACG, CGT, CGA\}$, then $\{ACGT, CGA\}$ would an SPSS of K. Note that K can have multiple spectrum-preserving string sets. There are several efficient tools for computing an SPSS so as to minimize the total number of characters [15, 7]. In this paper, we rely on the implementation in [25]. Each string in the resulting SPSS is referred to as a *simplitig*.

Compression of a k-mer set

ESS is a disk-compression format to store a set of k-mers K. It was introduced in [25] as the output of a compression tool, which, in this paper, we will refer to as ESS-basic. The technical details of the format and of the tool are irrelevant for this paper and can be viewed as black boxes. An ESS representation cannot be queried efficiently but can be decompressed into an SPSS of K. This output gives an ordering of the k-mers of K, and therefore the ESS compression of K induces an ordering on K. Note that because the decompression algorithm is deterministic, by storing an ESS representation, we are implicitly storing an SPSS representation as well.

Colored k-mer sets

Let $C > 0$ be an integer indicating the number of colors. Let $E = \{E_0, \ldots, E_{C-1}\}$ be a set of C k-mer sets, also referred to as a *colored de Bruijn graph*. Let $\overline{E}$ be the set of all k-mers in E, i.e. $\overline{E} = \{x \mid \exists i \text{ s. t. } x \in E_i\}$. The *(color) class* of a k-mer $x \in \overline{E}$ is the set of indices i such that $x \in E_i$. The *color vector* of x is a binary vector of length C where position i is 1 iff $x \in E_i$.

Non-compressed representation of cdBGs

Assume you have an ordering of $\overline{E}$, e.g. the one given by an ESS of $\overline{E}$. A *color matrix* of E is a file with row i containing the color vector of the i^{th} k-mer. Storing an ESS of $\overline{E}$ together with a color matrix of E is a lossless representation of E.

3 Methods

In this section, we describe our algorithm ESS-color for the compression of cdBGs. Let $E = \{E_0, \ldots, E_{C-1}\}$ be a colored dBG over C colors. Recall that $\overline{E}$ is the set of all k-mers in E. Let M denote the number of distinct color classes in E.

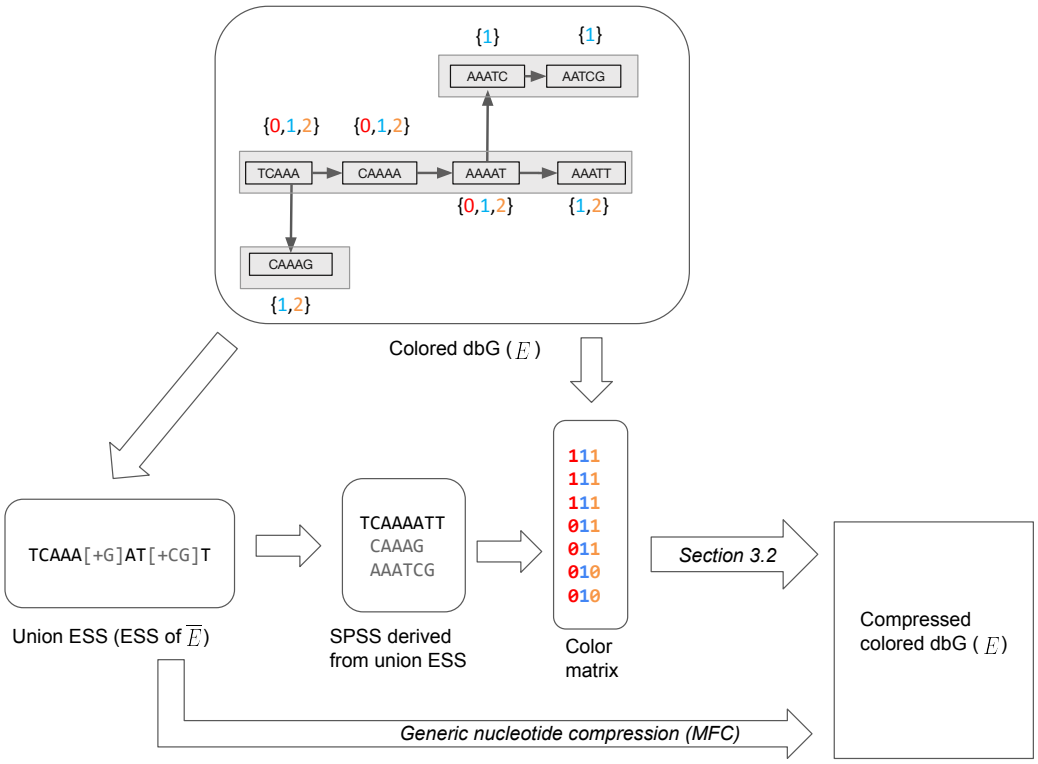


Figure 1 An example illustrating the various definitions in Section 2 and the first step of our compression method (Section 3.1). The input to the compression algorithm is a colored de Bruin Graph. The top panel shows an example with three colors (i.e. $C = 3$), $k = 5$, and a colored cdBG of $E = \{E_0, E_1, E_2\}$. Here, $E_0 = \{TCAAA, CAAAA, AAAAT\}$, $E_1 = \{TCAAA, CAAAA, AAAAT, AAATT, CAAAG, AAATC, AATCG\}$, and $E_2 = \{TCAAA, CAAAA, AAAAT, AAATT, CAAAG\}$. The color class is shown next to each k-mer, e.g. the color class of $TCAAA$ is $\{0, 1, 2\}$. There are three distinct color classes, i.e. $M = 3$. These are $\{0, 1, 2\}, \{1, 2\}$, and $\{1\}$. The lower left panel shows the union ESS, i.e. the ESS representation of the set $\overline{E} = \{TCAAA, CAAAA, AAAAT, AAATT, CAAAG, AAATC, AATCG\}$. This union ESS can be decomposed into an SPSS of $\overline{E}$, shown in the figure. The third column in bottom panel shows the color matrix, with k-mers in the order of the SPSS. To obtain the final compressed representation the color matrix is compressed using the algorithm we describe in Section 3.2.

ESS-color can accept input in one of two formats. First, it can accept each E_i stored in the KFF file format [10]. Alternatively, we can take as input a collection of FASTA files, each one assigned one of C colors, and an abundance parameter a. E_i is then implicitly defined as the set of all canonical k-mers that appear at least a times in the FASTA files of color i. We obtain E_i by running KMC [17] on the FASTA files of color i.

3.1 Building a color matrix of E and compressing $\overline{E}$

In this step, we first compress $\overline{E}$ using ESS-basic [25] and then build the color matrix of E, ordered by the ESS order. Specifically, we first compress the nucleotide sequences themselves, i.e. we run ESS-basic [25] on all the input files jointly. We refer to this as the *union ESS*.

We then decompress this file to obtain an SPSS of $\overline{E}$, denoted by S. From S, we build an SSHash dictionary [24] that allows us to map each k-mer in $\overline{E}$ to its rank in S. We then build on top of the KMC API to read in the binary files representing $E_0, \ldots, E_{C-1}$ and output a color matrix ordered by the SSHash dictionary. At the end of this stage, we have the union ESS, which is retained in the final compression output, and we have S and the color matrix, which are used in later stages but not retained in the final compression output.

3.2 Compression of the color matrix

Given an SPSS S of $\overline{E}$ and a color matrix of E over the order induced by S, we now generate a compressed representation of the matrix. Our representation consists of a *global class table* and, for every simplitig of S, a few bits of metadata, a *local class table* and one bitvector m. The local class table is optional, as we describe below. Figure 2 gives a schematic representation. We now explain each of these in detail.

Global class table: For most applications, the number of distinct color vectors M is significantly smaller than 2^C. Hence, the color matrix representation, which uses C bits per k-mer, is very inefficient. Instead, we use Huffman coding to assign a *global ID* to each class, so as to minimize the number of bits that will be used to store these IDs later (this is similar to what was done in [3]). To do this, we scan the color matrix to determine all the distinct classes and the number of k-mers that have each class. We then use Huffman coding to assign a global ID to each distinct class, so that more frequent global IDs tend to use less bits. This table is then stored in two forms: one that is compressed to disk, and the other that is stored in memory to be used during the compression algorithm.

We store the table on disk using three files: a color encoding Δ, a boundary bitvector b, and a text file. First, we sort the color classes in increasing numerical order, interpreting each color vector as a C-bit integer. For Δ, we write a concatenation of the M color vectors to disk, with the first color vector being written using C bits and the following colors being encoded as a difference with their predecessor. Specifically, if h_i is the Hamming distance between the i^{th} and the $(i-1)^{\text{st}}$ color vectors, then we use $h_i \lceil \log C \rceil$ bits to encode the indices where the i^{th} color vector is different from the $(i-1)^{\text{st}}$ color vector. We also store a boundary bitvector b which is the same length as Δ and contains a "1" whenever Δ starts a new color class. Finally, we store the frequencies of the color classes in a text file. These three files are then sufficient to reconstruct the global IDs during decompression. [1]

Simultaneously, we need to be able to map a color vector to an ID during the compression process. To do this, we create a minimal perfect hash function h (CHM [8]) that maps from each distinct color vector to an integer between 0 and $M-1$. We then maintain an array A of size M, where for each color vector c, $A(h(c))$ holds the global ID of color vector c.

After the global class table is created, we process the simplitigs of S one at time. For each simplitig, we dynamically set two parameters: a boolean variable $UseLocalID$, and an integer $0 \leq maxDif \leq 2$. We postpone the discussion on how these are set until the end of the section. The values of $maxDif$ and $UseLocalID$ are stored using 3 bits of metadata per simplitig. If $UseLocalID$ is set, we create a local class table:

[1] For readers familiar with Mantis-MST [2], we also tried their approach for storing our global table. Surprisingly, we found that our approach outperformed their more sophisticated approach, at least in our datasets. Though the Mantis-MST approach resulted in a smaller Δ vector, the overhead of storing the tree parent vector outweighed this gain.

Local class table: In the case that the frequencies of color classes are evenly distributed, we need approximately $\log M$ bits to represent the global class ID of each k-mer. We observe that sometimes a class is used at multiple locations of a simplitig, in which case using $\log M$ bits for each occurrence can be wasteful. Let ℓ be the number of distinct classes appearing in a simplitig. To save space on class IDs, we create a separate *local class table*, which maps from ℓ integers, called *local class IDs*, to their respective global IDs. Then, the encoding of k-mer classes for this simplitig can use local class IDs, which take only $\log \ell$ space. The local class table is written to disk, with $\log M$ bits encoding ℓ followed by ℓ consecutive global class IDs together.

The bitvector m is constructed by scanning the simplitig from left to right and, for each k-mer x, deciding how to encode it, and appending that encoding to m. Intuitively, the encodings follow three basic possibilities. The first possibility is to just append m with the k-mer's class ID. Second, we observed in practice that simplitigs often contain runs of k-mers with identical classes, in which case we can append m with the length of the run, rather than writing out each class IDs (such runs are similar to the monotigs of [20]). Finally, we often observe that a k-mer has a color vector with a small Hamming distance (i.e. 1 or 2) to that of the previous k-mer. In this case, we append m with the indices in the color vector that are different. Since there are three types of encoding, we will also need to prepend each encoding with two bits denoting the type of encoding. Formally, for each k-mer in a simplitig, we choose one of four options:

Skip: This option is invoked if x is not the first or last k-mer in its simplitig and has the same class as the preceding and succeeding k-mer. In this option, nothing is appended to m.

Small class difference: Let h be the Hamming distance between the color vector of x and the color vector of the preceding k-mer in the simplitig. This option is invoked when $0 < h \leq maxDif$. First, we append m with "10" to indicate that the following encoding will encode a class difference. If $maxDif = 2$, then we append m with a "1" to indicate that $h = 2$ or a "0" to indicate that $h = 1$. If $maxDif = 1$, then we do not append this extra bit, since it is implicit. Then, we append m with $h \log C$ bits which list the colors that are different. Note that setting $maxDif = 0$ effectively disables this type of encoding.

End of run: This option is invoked if x has the same class as the preceding k-mer and either has a different class than the succeeding k-mer or is the last k-mer in the simplitig. First, we indicate that the following encoding will encode a run length by appending m with "11" if $maxDif > 0$ and "1" if $maxDif = 0$. This difference is due to the fact that if $maxDif = 0$, then there are only two types of encodings and so we can just use one bit for the type.

Let $runLen$ be the number of consecutive k-mers that preceded x (not including x) and have the same class. We encode $runLen$ by separating it into a quotient q and remainder r (with respect to a global parameter $runDivisor$), and then encoding the quotient q in unary and the remainder r in binary. Formally, let $q = \lfloor \frac{runLen}{runDivisor} \rfloor$ and $r = runLen \bmod runDivisor$. We append to m q "1"s followed by a "0". Then, we append to m the binary encoding of r, using $\log runDivisor$ bits. For example, if $runDivisor = 16$ and $runLen = 21$, then $q = 1$ and $r = 5$, and m is appended with 100101. Observe that a smaller value of $runDivisor$ results in more bits used to encode long runs (i.e. q is larger) while a larger value of $runDivisor$ uses more bits to encode short runs (i.e. $\log runDivisor$ is larger). We found that a default value of 16 works best in our experiments.

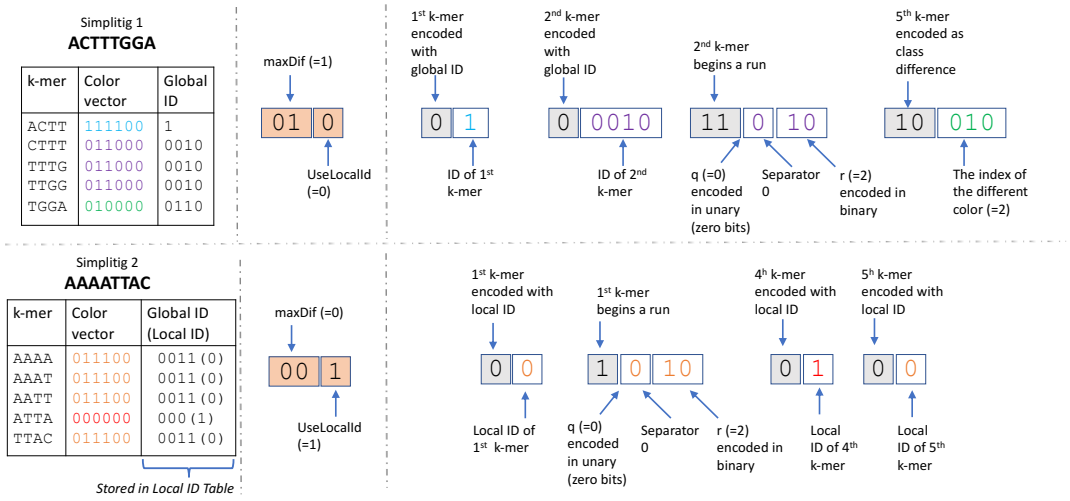


Figure 2 Example of how we compress the color matrix (Section 3.2). The top panel shows the compression of a simplitig ACTTTGGA and the bottom panel shows the compression of a simplitig AAAATTAC. Other simplitigs exist but are not shown. In this example, $C = 6$, $k = 4$, and $runDivisor = 4$. The figure is divided into three columns. The first column shows the information that the algorithm has about each k-mer (i.e. its color vector and corresponding global ID). The second column shows the metadata which holds the values of $maxDif$ and $useLocalId$ for the corresponding simplitig. The third column shows the m vector. Within the m vector, the type of encoding is shown in gray colored boxes and other values are shown in white boxes. The colors of the values inside the white boxes correspond to the color used for the corresponding k-mers' color vector in the first column of the figure. Note that in the bottom simplitig, a local table is used. In particular, there are two distinct color vectors in this simplitig, with global ID 0011 assigned to local ID 0 and global ID 000 assigned to local ID 1.

Store the class ID: This option is invoked when none of the criteria for the other options are satisfied. In this case, we append m with "0" to indicate the type of encoding, followed by the class ID of x. If $UseLocalID$ is set, we use the local class ID, otherwise we use the global class ID.

After finishing with all simplitigs, we compress the global class table, local class tables, and m using RRR [27] and write them to disk.

Setting the parameters $UseLocalID$ and $maxDif$ involves trade-offs that are difficult to quantify in advance. For example, the cost of having to store the local class table may exceed the benefits of using less bits to encode class IDs for a simplitig where every present class ID is contained within a single run. Similarly, when d is too large, then writing the positions of the color differences to m can take more space then just writing the class ID. Moreover, there is a benefit of setting $d = 0$, since it enables to save one bit per run by using "1" instead of "11" for the "end of run" encoding. All bitvectors are additionally compressed with RRR, making it difficult to determine in advance which parameters result in the least space. We therefore try all possibilities of $maxDif \in \{0, 1, 2\}$ and $UseLocalID \in \{True, False\}$, and, for each combination, compute the encoding. We then use the encoding that takes less space and disregard the rest. Though this step can likely be optimized, we found that the time taken to try all possibilities was not a large factor in the overall compression time.

The decompression algorithm for the m vector is straightforward since our color matrix compression scheme is designed to be unambiguously decompressed. Simultaneously, we decompress $\overline{E}$ with ESS-decompress. The result is an SPSS S of $\overline{E}$ and a color matrix of

Table 1 Dataset characteristics. C is the number of colors and M is the number color classes.

Source	type	C	M	n. k-mers $\times 10^6$ (% single color)		n. simplitigs $\times 10^6$	
				$k = 23$	$k = 31$	$k = 23$	$k = 31$
E. coli	assemblies	100	542,545	27 (30%)	31 (31%)	0.5	0.5
E. coli	assemblies	10	826	13 (38%)	14 (41%)	0.2	0.2
Fungi	assemblies	20	13,227	394 (93%)	409 (93%)	1.8	1.7
Gut	metagenome reads	9	511	2,236 (67%)	2,477 (70%)	76	95
Human	RNA-seq reads	19	9,654	120 (71%)	103 (75%)	7.2	10

E in the order of S. If the output is to be processed downstream in a streaming manner, our decompression algorithm can trivially stream out k-mer sequence and color vectors, one k-mer at a time.

4 Results

4.1 Evaluated tools and datasets

As far as we are aware, there are two other tools designed for compressing colored de Bruijn graphs: KS [16] and GGCAT [7]. We refer to the first tool as KS after the authors' last names [16]. KS is limited to support only three k values (15, 19, and 23), so we compare against it for $k = 23$ but also evaluate ESS-color on a more practical k value of $k = 31$. For GGCAT, we additionally compressed its Fasta output file with MFC and its binary color table with gzip to maximize its compression ratio. We also compare ESS-color against the naive approach of compressing each color independently using the algorithm of [25], which we refer to as *ESS-basic*.

Table 1 shows the datasets we use for evaluation and their properties. We chose five datasets so as to cover a broad range of input types. Three of the datasets are from assembled genomes, one is from RNA-seq reads, and one is from metagenome reads. We used all k-mers from the three assemblies datasets and all k-mers that appear at least twice from the two read datasets. The datasets cover various species, from Bacteria to Fungi to Human. Concretely, we have 1) 100 arbitrarily selected *E.coli* strains from GenBank, 2) an arbitrary subset of 10 of those, 3) 20 arbitrarily selected fungi sequences from RefSeq, 4) gut microbiome read sets from nine individuals sequenced in [21], and 5) 19 paired-end, human, bulk RNA-seq short-read experiments previously used in [2]. All accession numbers are listed in `https://github.com/medvedevgroup/ESSColor/wiki/Experiments`.

Finally, all experiments were run on a server with an Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz processor with 64 cores and 512 GB of memory. We ran all tools in unrestricted memory mode. We used 8 threads for all tools and their components, whenever they supported multi-threading.

4.2 Comparison against other disk compression tools

Table 2 shows the bits per k-mer achieved by ESS-color compared with KS, GGCAT ESS-basic, and a p7zip compression of the original fasta file. ESS-color achieves better compression than all evaluated tools and on all datasets. No other tool was able to consistently achieve less than 44% space overhead compared to ESS-color. On some datasets the improvement over all other tools is quite large, e.g. for Gut ($k = 23$), all the other tools use at least 27% more space than ESS-color.

Table 2 Compression results, in bits per k-mer. ESS-color is our new tool. ESS-basic is the non-integrative approach of compressing every color separately. KS is tool from [16], and a hyphen indicates that it does not support $k > 23$. fa.p7zip is the p7zip compression of the original data. We show bits per k-mer, which is the total compressed size divided by the number of distinct k-mer in the input (i.e. $|\overline{E}|$). Compression ratios can be inferred by comparing to the fa.p7zip column.

Dataset	$k = 23$					$k = 31$				
	fa.p7zip	ESS-color	KS	GGCAT	ESS-basic	fa.p7zip	ESS-color	KS	GGCAT	ESS-basic
Ecoli100	37	5.2	17.3	6.4	35.0	33	4.4	–	5.5	30.3
Ecoli10	8	2.9	5.3	3.3	7.3	7	2.6	–	3.0	6.6
Fungi20	2.4	2.2	2.5	2.3	2.3	2.3	2.2	–	2.3	2.5
Gut9	60	3.7	5.7	4.9	4.7	54	3.4	–	4.9	4.4
HumRNA19	112	5.6	8.8	7.1	10.4	131	8.9	–	10.5	12.1

The compression ratio of ESS-color relative to the original Fa.p7zip files varies (Table 2). For the read datasets, it is more than 15x, since high-coverage FASTA files are by their nature very redundant. For the assemblies datasets, the ratio is between 1x and 7x. We found that a good predictor of compressibility of assemblies is the percentage of k-mers that have exactly one color (Table 1). At one extreme, 93% of Fungi20 k-mers have exactly one color, and ESS-color achieves little improvement over fa.p7zip. At the other extreme, only 30% of Ecoli100 k-mers have exactly one color, and the compression ratio is relatively high at 7.1x (for $k = 23$). This trend makes intuitive sense since single-color k-mers do not benefit from ESS-color's multi-color compression algorithm.

KS is not as effective as ESS-color on our datasets, using between 1.4x and 3.3x more space than ESS-color (Table 2). We note that even though KS is also designed to exploit the fact that k-mers are shared across colors, it makes a different design trade-off compared to ESS-color. Specifically, it does not allow simplitigs to extend beyond a single color class (resulting in more space needed to store k-mers), but, in exchange, it is more efficient in storing color information.

GGCAT is generally the closest competitor against ESS-color, using between 14 and 44% more space on the non-Fungi datasets (note that for Fungi all tools did well). Like ESS-color, it builds a kind of global class table, constructs an SPSS of the k-mers, and annotates each run of single-class k-mers with their class ID. Unlike ESS-color, however, it does not use ESS, does not encode small class differences, and does not use local class tables.

As expected, ESS-basic is not as effective as ESS-color, using up to 6.9x more space than ESS-color (Table 2). These results are not surprising since ESS-basic does not exploit the redundancy created by shared k-mers across samples. For the assemblies datasets, the compression improvement of ESS-color over ESS-basic closely tracks that of ESS-color over the original fa.p7zip. For the sequencing datasets, ESS-basic uses between 1.3 and 1.9 more space than ESS-color.

Tables 3 and 4 show the run time and memory usage of compression, respectively. Here, ESS-color is outperformed by other tools. In particular, if optimal compression space is not needed, then GGCAT becomes a good alternative to ESS-color. Note that the decompression time (Table 5) is an order of magnitude smaller compared to the compression times.

Table 3 Time (min) used by the various compression algorithms. For the Gut9 ESS-color run with $k = 23$ (marked with *), we used an unoptimized implementation of the color matrix generation step, since SSHash was not working as expected. For GGCAT on Gut9 with $k = 23$ (denoted by **), the original run crashed because of exceeding the number of open files allowed by the operating system. We therefore re-ran GGCAT using our simplitigs as a starting point, which allowed the run to complete. However, the time shown here does not include the time we used to construct the simplitigs.

	$k = 23$				$k = 31$		
Dataset	ESS-color	KS	GGCAT	ESS-basic	ESS-color	GGCAT	ESS-basic
Ecoli100	14.8	11.6	0.7	6.3	20.7	0.7	27
Ecoli10	2.6	7.0	0.3	0.9	2.6	0.3	2
Fungi20	59.1	7.0	3.3	7.3	76	3.3	25
Gut9	1101*	148.5	37.8**	234	611.6	92.2	341
HumRNA19	31.9	10.5	21.2	31	60.1	15.0	39

Table 4 Maximum memory (Gb) used by the various compression algorithms. The (*) and (**) annotations are the same as in Table 3.

	$k = 23$				$k = 31$		
Dataset	ESS-color	KS	GGCAT	ESS-basic	ESS-color	GGCAT	ESS-basic
Ecoli100	1.2	0.9	1.5	1.2	1.1	1.4	1
Ecoli10	0.6	3.2	0.8	1.1	0.6	0.8	1
Fungi20	5.4	3.2	4.8	5.9	4.3	3.9	6
Gut9	174.6*	50.6	87.2**	33.2	121	78.2	119
HumRNA19	26.8	6.0	8.8	9.1	12.1	10.2	7

Table 5 Time and memory for decompression of ESS-color, for $k = 31$.

Dataset	Memory (GB)	Time (min)
Ecoli100	0.5	2
Ecoli10	0.5	1
Fungi20	0.5	13
Gut9	8.5	90
HumRNA19	1.0	5

Table 6 Breakdown of the space usage (in percentage of total space) of the components of ESS-color, for $k = 31$. Note that all components except union ESS are shown after compression with RRR.

Dataset	union ESS	m	global table (Δ and b)	global table (frequencies)	local tables
Ecoli100	51	23	23	2	1
Ecoli10	81	18	<0.1	<0.1	0.5
Fungi20	95	5	<0.1	<0.1	0.1
Gut9	77	22	<0.1	<0.1	1
HumRNA19	73	26	<0.1	<0.1	0.8

Table 7 The percentage of simplitigs ($k = 31$) that fall into the six compression modes, i.e. combinations of $UseLocalID$ and $maxDif$.

Dataset	$UseLocalID$ = False			$UseLocalID$ = True		
	$maxDif = 0$	$maxDif = 1$	$maxDif = 2$	$maxDif = 0$	$maxDif = 1$	$maxDif = 2$
Ecoli100	75	20	5	<0.1	<0.1	<0.1
Ecoli10	82	16	2	<0.1	<0.1	<0.1
Fungi20	99	1	0.2	<0.1	<0.1	<0.1
Gut9	80	19	2	0.2	<0.1	<0.1
HumRNA19	91	9	0.1	0.1	<0.1	<0.1

Table 8 Compression results in bits per k-mer ($k = 31$) of indexing approaches, compared to ESS-color.

Dataset	ESS-color	RowDiff-ESS	RowDiff+	Rainbow-MST
Ecoli100	4.4	14.6	8.0	34.2
Ecoli10	2.6	7.8	6.3	19.3
Fungi20	2.2	3.6	4.3	9.5
Gut9	3.4	11.2	40.6	15.6
HumRNA19	8.9	37.9	112.6	19.1

4.3 Inside the space usage of ESS-color

ESS-color's compressed representation includes several components, with the major ones being the union ESS, the m vector, the global table, and the local tables. Table 6 shows that the majority of space used by ESS-color is taken by the union ESS. Except for Ecoli100, the rest of the space is taken up almost exclusively by m. For Ecoli100, which has the largest number of colors, the global table takes 23% of the total space.

Recall that ESS-color chooses one of six different compression modes for each simplitig, i.e. $UseLocalId \in \{0, 1\}$ and $maxDif \in \{0, 1, 2\}$. In order to access the relative contribution of the various compression techniques, we count the frequency with which each mode occurs (Table 7). First, we observe that the idea of a local table was rarely helpful on our data. Local tables are only beneficial when a single color class appears in more than one run in a simplitig, which apparently was rare. Second, the majority of simplitigs use $maxDif = 0$. This mode is optimal when the simplitig has just one color class. There is also a more complicated trade-off since setting $maxDif > 0$ adds one extra bit for each run encoding, which may outweigh the benefits of encoding some k-mers with a class difference. Third, the Gut dataset demonstrates the benefit of encoding class differences, especially compared to GGCAT. It is the dataset with the highest percentage of simplitigs using $maxDif > 0$ (21%) and, simultaneously, it is also the dataset where GGCAT uses the most space relative to ESS-color (44%).

4.4 Comparison to indexing data structures

There exist numerous indexing data structures for cdBGs [19]. Indexing data structures are similar to disk compression but additionally support efficient membership and color queries. We expect this overhead to make them non-competitive with respect to disk compression schemes. To verify this, we compared the space taken by ESS-color against three indexing approaches. We note that since these approaches are designed for indexing, they do not implement decompression and are thus not viable for disk compression in their current state. We also note that GGCAT also supports indexing, but, since it is trivial to decompress, we included it in the main analysis of Section 4.2.

The first two approaches are ones that are shown in [9, 14] to be the most space efficient. These are RowDiff+, which is the latest version [14] of RowDiff [9], and Rainbow-MST [9], which is a space-improved version of Mantis-MST [2]. As a trivial improvement, we further compress these indices using gzip. The third approach we compare to is a natural hybrid of ESS-color and the RowDiff indexing algorithm for cdBGs [9]. We refer to this as *RowDiff-ESS* and describe it in detail in the Appendix. We do not compare against other indices such as REINDEER [20], Bifrost [12], Themisto [1], or Mantis-MST [2], because they are less space efficient than RowDiff+, and we do not compare against Sequence Bloom Tree approaches (e.g. [11, 28]) because they are lossy.

Table 8 shows the results. As expected, the compression ratios of these indexing tools are not competitive against ESS-color. Even the most space efficient indexing approach for each dataset takes 60% more space than ESS-color. We do note that GGCAT, which was shown in Table 2, is an exception, since it implements both efficient indexing and disk compression.

5 Conclusion

Colored de Bruijn graphs are a popular way to represent sequence databases. In spite of their ever-growing sizes, there have not been many specialized tools for compressing them to disk. In this paper, we present a novel disk compression algorithm tailormade for cdBGs that achieves superior space compression compared to all other tools on the evaluated datasets.

Our algorithm is a novel combination of ideas borrowed from previous work on disk compression of k-mer sets and indexing of cdBGs. We use a spectrum-preserving string set (SPSS) as a basis for both compressing the nucleotide sequences and for ordering the rows in the color matrix. By using the SPSS ordering, we can avoid the costly storage of an indexed de Bruijn graph (e.g. BOSS in [9, 14] or a counting quotient filter in [2]). We also exploit the fact that consecutive k-mers in an SPSS likely have the same or similar color class. A major component of our approach is that we select a different compression scheme for each simplitig, depending on what gives the best compression on that simplitig.

The most important practical direction for future work is to improve the running time of our algorithm. The generation of the union ESS is done by ESS-basic. ESS-basic can be sped up by extending the latest SPSS generation tools [15, 7] to also compute an ESS. We could even build on top of GGCAT, taking advantage of their efficient implementation (unfortunately, GGCAT was only released once our project was near completion). Another bottleneck is the color matrix generation step, which could be parallelized or even avoided by using color lists.

A theoretical future direction is to derive bounds on the bits used by the compression scheme. Unfortunately, we do not see an easy way to do this, since the choice of encoding depends on the order of the k-mers in the SPSS and on the decomposition of the k-mers into simplitigs. It is unclear to us how to capture these properties as a function of the input data.

Finally, we could further improve the compression algorithm by modifying the SPSS generated by the ESS-basic algorithm. Currently, the choice of how to select from multiple simplitig extensions is made arbitrarily. Instead, the choice could be made to use the extension that has the most similar color class. Such a modification to the SPSS construction algorithm would likely be computationally non-trivial, since it would require accessing the color information.

References

1 Jarno N. Alanko, Jaakko Vuohtoniemi, Tommi Mäklin, and Simon J. Puglisi. Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. *Bioinformatics*, 2023.

2 Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. An Efficient, Scalable, and Exact Representation of High-Dimensional Color Information Enabled Using de Bruijn Graph Search. *Journal of Computational Biology*, 27(4):485–499, 2020.

3 Fatemeh Almodaresi, Prashant Pandey, and Rob Patro. Rainbowfish: A succinct colored de Bruijn graph representation. In *17th International Workshop on Algorithms in Bioinformatics (WABI 2017)*, volume 88 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:15, 2017.

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

6 Karel Břinda. *Novel computational techniques for mapping and classifying Next-Generation Sequencing data*. PhD thesis, Université Paris-Est, 2016.

7 Andrea Cracco and Alexandru I. Tomescu. Extremely-fast construction and querying of compacted and colored de Bruijn graphs with GGCAT. *BioRxiv*, 2022.

8 Zbigniew J Czech, George Havas, and Bohdan S Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information processing letters*, 43(5):257–264, 1992.

9 Daniel Danciu, Mikhail Karasikov, Harun Mustafa, André Kahles, and Gunnar Rätsch. Topology-based sparsification of graph annotations. *Bioinformatics*, 37(Supplement_1):i169–i176, 2021.

10 Yoann Dufresne, Teo Lemane, Pierre Marijon, Pierre Peterlongo, Amatur Rahman, Marek Kokot, Paul Medvedev, Sebastian Deorowicz, and Rayan Chikhi. The K-mer File Format: a standardized and compact disk representation of sets of k-mers. *Bioinformatics*, 38(18):4423–4425, 2022.

11 Robert S Harris and Paul Medvedev. Improved representation of Sequence Bloom Trees. *Bioinformatics*, 36(3):721–727, 2020.

12 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.

13 Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature genetics*, 44(2):226–232, 2012.

14 Mikhail Karasikov, Harun Mustafa, Gunnar Rätsch, and André Kahles. Lossless indexing with counting de Bruijn graphs. In *Research in Computational Molecular Biology*, pages 374–376, 2022.

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. *Genome biology*, 23(1):190, 2022.

16 Kazushi Kitaya and Tetsuo Shibuya. Compression of multiple k-mer sets by iterative SPSS decomposition. In *21st International Workshop on Algorithms in Bioinformatics (WABI 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

17 Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, 2017.

18 Camille Marchet. Data-structures for sets of k-mer sets: what's new since 2020. Blog post, 2022. URL: `https://kamimrcht.github.io/webpage/sets_kmer_sets.html`.

19 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, 2020.

20 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.

21 Joan Mas-Lloret, Mireia Obón-Santacana, Gemma Ibáñez-Sanz, Elisabet Guinó, Miguel L Pato, Francisco Rodriguez-Moranta, Alfredo Mata, Ana García-Rodríguez, Victor Moreno, and Ville Nikolai Pimenoff. Gut microbiome diversity detected by high-coverage 16S and shotgun sequencing of paired stool and colon sample. *Scientific data*, 7(1):92, 2020.

22 Louis Papageorgiou, Picasi Eleni, Sofia Raftopoulou, Meropi Mantaiou, Vasileios Megalooikonomou, and Dimitrios Vlachakis. Genomic big data hitting the storage bottleneck. *EMBnet. journal*, 24, 2018.

23 Rob Patro, Geet Duggal, Michael I Love, Rafael A Irizarry, and Carl Kingsford. Salmon provides fast and bias-aware quantification of transcript expression. *Nature methods*, 14(4):417–419, 2017.

24 Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *Bioinformatics*, 38(Supplement_1):i185–i194, 2022.

25 Amatur Rahman, Rayan Chikhi, and Paul Medvedev. Disk compression of k-mer sets. *Algorithms for Molecular Biology*, 16(1):1–14, 2021.

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

27 Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct dynamic data structures. In *Algorithms and Data Structures: 7th International Workshop, WADS 2001 Providence, RI, USA, August 8–10, 2001 Proceedings 7*, pages 426–437. Springer, 2001.

28 Brad Solomon and Carl Kingsford. Fast search of thousands of short-read sequencing experiments. *Nature biotechnology*, 34(3):300–302, 2016.

29 Roland Wittler. Alignment-and reference-free phylogenomics with colored de Bruijn graphs. *Algorithms for Molecular Biology*, 15:1–12, 2020.

A Appendix

Here we describe *RowDiff-ESS*, the hybrid of ESS-color and the RowDiff indexing algorithm for cdBGs [9]. Though the approach turned out to not be comptetive against ESS-color, we describe it here for completeness. The RowDiff index is composed of two parts: BOSS, which is an index of $\overline{E}$, and a compressed color matrix whose labels are implicitly given by BOSS. Because of its structural similarity to our approach, we can swap out BOSS (which supports queries and is therefore not space efficient for disk compression) with an ESS of $\overline{E}$. We then feed the k-mer ordering implied by the ESS to the RowDiff color matrix compression algorithm. The space used by this scheme is the ESS space plus the space of the RowDiff's color matrix, compressed with gzip.

Fulgor: A Fast and Compact k-mer Index for Large-Scale Matching and Color Queries

Jason Fan
Department of Computer Science, University of Maryland, College Park, MD, USA

Noor Pratap Singh
Department of Computer Science, University of Maryland, College Park, MD, USA

Jamshed Khan
Department of Computer Science, University of Maryland, College Park, MD, USA

Giulio Ermanno Pibiri
DAIS, Ca' Foscari University of Venice, Italy
STI-CNR, Pisa, Italy

Rob Patro
Department of Computer Science, University of Maryland, College Park, MD, USA

Abstract

The problem of sequence identification or matching – determining the subset of reference sequences from a given collection that are likely to contain a short, queried nucleotide sequence – is relevant for many important tasks in Computational Biology, such as metagenomics and pan-genome analysis. Due to the complex nature of such analyses and the large scale of the reference collections a resource-efficient solution to this problem is of utmost importance. This poses the threefold challenge of representing the reference collection with a data structure that is efficient to query, has light memory usage, and scales well to large collections.

To solve this problem, we describe how recent advancements in associative, order-preserving, k-mer dictionaries can be combined with a compressed inverted index to implement a fast and compact *colored de Bruijn* graph data structure. This index takes full advantage of the fact that unitigs in the colored de Bruijn graph are *monochromatic* (all k-mers in a unitig have the same set of references of origin, or "color"), leveraging the *order-preserving* property of its dictionary. In fact, k-mers are kept in unitig order by the dictionary, thereby allowing for the encoding of the map from k-mers to their inverted lists in as little as $1 + o(1)$ bits per unitig. Hence, one inverted list per unitig is stored in the index with almost no space/time overhead. By combining this property with simple but effective compression methods for inverted lists, the index achieves very small space.

We implement these methods in a tool called Fulgor. Compared to Themisto, the prior state of the art, Fulgor indexes a heterogeneous collection of 30,691 bacterial genomes in 3.8× less space, a collection of 150,000 *Salmonella enterica* genomes in approximately 2× less space, is at least twice as fast for color queries, and is $2 - 6\times$ faster to construct.

2012 ACM Subject Classification Applied computing → Bioinformatics

Keywords and phrases k-mers, Colored de Bruijn Graph, Compression, Read-mapping

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.18

Related Version
bioRxiv Version: https://www.biorxiv.org/content/10.1101/2023.05.09.539895v2

Supplementary Material *Software (Source Code)*: https://github.com/jermp/fulgor
Software (Source Code): https://github.com/jermp/fulgor-benchmarks

Funding This work is supported by the NIH under grant award numbers R01HG009937 to R.P.; the NSF awards CCF-1750472 and CNS-1763680 to R.P., and DGE-1840340 to J.F. Funding for this research has also been provided by the European Union's Horizon Europe research and innovation programme (EFRA project, Grant Agreement Number 101093026).

Conflicts of Interest R.P. is a co-founder of Ocean Genomics Inc.

1 Introduction

At the core of many metagenomic and pan-genomic analyses is *read-mapping*, the atomic operation that assigns observed sequence reads to putative genome(s) of origin. A wide range of methods have been developed for mapping reads to large collections of reference genomes. Of note, alignment-based methods, though accurate [20, 23], are relatively computationally intensive as they must provide the ability to *locate* the read on each genome. A queried read must, with low edit-distance, be matched with a sub-string of some reference genome in the collection. For alignment, the index is also required to report the position of this match. As a matter of fact, alignment against hundreds or even tens of thousands of reference genomes can be impractically slow and simply require too much space in practice.

Fortunately, *alignment-free* techniques have become popular and widespread for metagenomic analyses [43, 27, 42, 35, 40, 36]. These methods generally work by avoiding alignment altogether, and replacing it with strategies for matching (exactly or approximately) sub-strings, signatures, or sketches between the queries and the referenced sequences. Ideally, good matching heuristics can assign or match a query against the correct reference with high precision while also retaining high recall (i.e., being sensitive to sequencing error or small divergence between the query and the reference). One particular type of alignment-free method for assigning reads to compatible references that has recently gained substantial traction is *pseudoalignment* [6, 39, 37, 21]. While tremendous progress has been made in supporting alignment-free methods for metagenomic analyses, continued development of ever more efficient indexing methods is required for such analyses to scale to tens, even hundreds, of thousands of bacterial reference genomes.

A practical data structure that is suitable for alignment-free matching methods is the *colored de Bruijn graph*, a graph where each node corresponds to a k-mer in a reference collection and is annotated with a *color*, the set of references in which it occurs. Bifrost [15] and Metagraph [17] are two efficient approaches that index the colored de Bruijn graph and support the k-mer-to-color query. Recently, Alanko et al. [2] developed Themisto, an index for alignment-free matching (and specifically pseudoalignment) that substantially outperforms these prior methods in the context of indexing and mapping against large collections of genomes. Compared to Bifrost, Themisto uses practically the same space, but is faster to build and query. Compared to the fastest variant of Metagraph, Themisto offers similar query performance, but is much more space-efficient; on the other hand, Themisto is much faster to query than Metagraph-BRWT, the most-space efficient variant of Metagraph.

1.1 Contributions

We describe how recent advancements in associative, order-preserving, k-mer dictionaries [30, 29] can be combined with a compressed inverted index to implement a fast index over the *colored compacted* de Bruijn graph (ccdBG). Leveraging the *order-preserving* property of its dictionary, our index takes full advantage of the fact that unitigs in this variant of the ccdBG are *monochromatic* – i.e., all k-mers in a unitig have the same set of references of origin, or "colors". In fact, k-mers are kept in unitig order, and our index takes advantage of the ability of our associative dictionary to store the unitigs in any order. Reordering the unitigs so that all unitigs with the same color are adjacent in the index allows the construction of a map from k-mers to their corresponding colors that uses only $1 + o(1)$ bits per unitig. Our index combines this property with a simple but effective hybrid compression scheme for inverted lists (colors) to require little space. By storing unitigs and keeping k-mers in unitig order, our index also supports very fast streaming queries for consecutive k-mers in a read, and

additionally allows efficient implementation of skipping heuristics that have previously been suggested to speed up pseudoalignment [6]. We implemented our index in a C++17 tool called Fulgor, which is available at `https://github.com/jermp/fulgor`.

Compared to Themisto [2], the prior state of the art, Fulgor indexes a heterogeneous collection of 30,691 bacterial genomes in 3.8× less space, a collection of 150,000 *Salmonella enterica* genomes in approximately 2× less space, is at least twice as fast at query time, and even $2-6\times$ faster to construct.

Perhaps unsurprisingly, the rapid development of novel indexing data structures has been accompanied by novel and custom strategies for matching and assigning reads to colors (i.e., reference sets) and algorithms that each make different design choices and trade-offs. Many of these strategies can be considered as a form of pseudoalignment. Having been iterated on since its introduction [6], the term "pseudoalignment" has come to describe a family of efficient heuristics for read-to-color assignment, rather than a single concept or algorithm. Prior methods have taken either *exhaustive* approaches that queries every k-mer on a read (previously termed *exact* pseudoalignment [21, 2]) or have implemented *skipping* based approaches that skip the query of "redundant" consecutive k-mers that likely map to the same set of reference genomes [6, 13]. To our knowledge, the precise details of the types of skipping heuristics used in the latter methods – including those adopted by the initial pseudoalignment method – have been discussed only in passing. Complete details, instead exist only in the source code of the corresponding tools. To shed light on these algorithms, we provide a more structured discussion of how these algorithms are designed. Using Fulgor, we implement two previously proposed variants and benchmark them.

2 Preliminaries

In this section, we first formalize the problem under study here. We then describe a modular indexing layout that solves the problem using the interplay between two well-defined data structures. Lastly we describe the properties induced by the problem and how these are elegantly captured by the notion of *colored compacted de Bruijn graph*.

2.1 Problem definition

▶ **Problem 1** (Colored k-mer indexing problem)**.** *Let $\mathcal{R} = \{R_1, \ldots, R_N\}$ be a collection of references. Each reference R_i is a string over the DNA alphabet $\Sigma = \{A, C, G, T\}$. We want to build a data structure (referred to as the* index*) that allows us to retrieve the set* $\mathsf{Color}(x) = \{i | x \in R_i\}$ *as efficiently as possible for any k-mer $x \in \Sigma^k$. Note that* $\mathsf{Color}(x) = \varnothing$ *if x does not occur in any reference.*

Hence, we call the set $\mathsf{Color}(x)$ the *color* of the k-mer x.

2.2 Modular indexing layout

In principle, Problem 1 could be solved using an old but elegant data structure: the *inverted index* [45, 34]. The inverted index, say $\mathcal{L}$, stores explicitly the ordered set $\mathsf{Color}(x)$ for each k-mer $x \in \mathcal{R}$. What we want is to implement the map $x \to \mathsf{Color}(x)$ as efficiently as possible in terms of both memory usage and query time. To this end, all the distinct k-mers of $\mathcal{R}$ are stored in an *associative* dictionary data structure, $\mathcal{D}$. Suppose the dictionary $\mathcal{D}$ stores n k-mers. To implement the map $x \to \mathsf{Color}(x)$, the operation that $\mathcal{D}$ is required to support is $\mathsf{Lookup}(x)$ which returns $\perp$ if k-mer x is not found in the dictionary or a unique integer identifier in $[n] = \{1, \ldots, n\}$ if x is found. Problem 1 can then be solved using these two data structures – $\mathcal{D}$ and $\mathcal{L}$ – thanks to the interplay between $\mathsf{Lookup}(x)$ and $\mathsf{Color}(x)$: logically, the index stores the sets $\{\mathsf{Color}(x)\}_{x \in \mathcal{R}}$ in compressed format in the order given by $\mathsf{Lookup}(x)$.

To our knowledge, all prior solutions proposed in the literature that fall under the "color-aggregative" classification [22], are incarnations of this *modular indexing framework* and, as such, require an efficient k-mer dictionary joint with a compressed inverted index. For example, Themisto [2] makes use of the *spectral* BWT (or SBWT) data structure [1] for its k-mer dictionary, whereas Metagraph [17] implements a general scheme to compress metadata associated to k-mers which is, in essence, an inverted index.

2.3 The colored compacted de Bruijn graph and its properties

Problem 1 has some specific properties that one would like to exploit to implement as efficiently as possible the modular indexing framework described in Section 2.2. First, consecutive k-mers share $(k-1)$-length overlaps; second, co-occurring k-mers have the same color. A useful, standard, formalism that describes these properties is the *colored compacted de Bruijn graph* (abbreviated "ccdBG").

Given the collection of references $\mathcal{R}$, the (node-centric) de Bruijn graph (dBG) of $\mathcal{R}$ is a directed graph whose nodes are all the distinct k-mers of $\mathcal{R}$ and there is an edge connecting node u to node v if the $(k-1)$-length suffix of u is equal to the $(k-1)$-length prefix of v. We refer to k-mers and nodes in a (node-centric) dBG interchangeably; likewise, a path in a dBG spells the string obtained by "glueing" together all the k-mers along the path. Thus, unary (i.e., non-branching) paths in the graph can be collapsed into single nodes spelling strings that are referred to as *unitigs*. The dBG arising from this compaction step is called the compacted dBG (cdBG). Lastly, the *colored* compacted dBG is obtained by logically annotating each k-mer x with its color, $\mathsf{Color}(x)$, and only collapsing non-branching paths with nodes having the same color.

Below, we notate n to be the number of distinct k-mers of $\mathcal{R}$ and m to be the number of unitigs $\{u_1, \ldots, u_m\}$ of the ccdBG induced by the k-mers of $\mathcal{R}$. The unitigs of the ccdBG that we consider have the following key properties.

1. *Unitigs are contiguous subsequences that spell references in $\mathcal{R}$.* Each distinct k-mer of $\mathcal{R}$ appears once, as sub-string of some unitig of the cdBG. By construction, each reference $R_i \in \mathcal{R}$ can be a *tiling* of the unitigs – a sequence of unitig occurrences that spell out R_i [11]. Joining together k-mers into unitigs reduces their storage requirements. In Sections 3.1 and 3.2, we show how this property can be exploited to make indexes compact. In Section 4, we show how this property can be exploited to make queries fast.

2. *Unitigs are monochromatic.* The k-mers belonging to the same unitig u_i all have the same color. Thus, we shall use $\mathsf{Color}(u_i)$ to denote the color of each k-mer $x \in u_i$. We note that this property holds only if one considers k-mers appearing at the start or end of reference sequences to be *sentinel* k-mers that must terminate their containing unitig [24, 19, 18], and that such conventions are not always adopted [15, 8].

3. *Unitigs co-occur and share colors.* Unitigs often have the same color (i.e., occur in the same set of references) because they derive from conserved sequences in indexed references that are longer than the unitigs themselves. We indicate with M the number of distinct color sets $\mathcal{C} = \{C_1, \ldots, C_M\}$. Note that $M \leq m$ and that in practice there are dramatically more unitigs than there are distinct colors. We use $\mathsf{ColorID}(u_i) = j$ to indicate that unitig u_i has color C_j. As a consequence, each k-mer $x \in u_i$ has color C_j.

In this work our goal is to design an index that takes full advantage of these key properties.

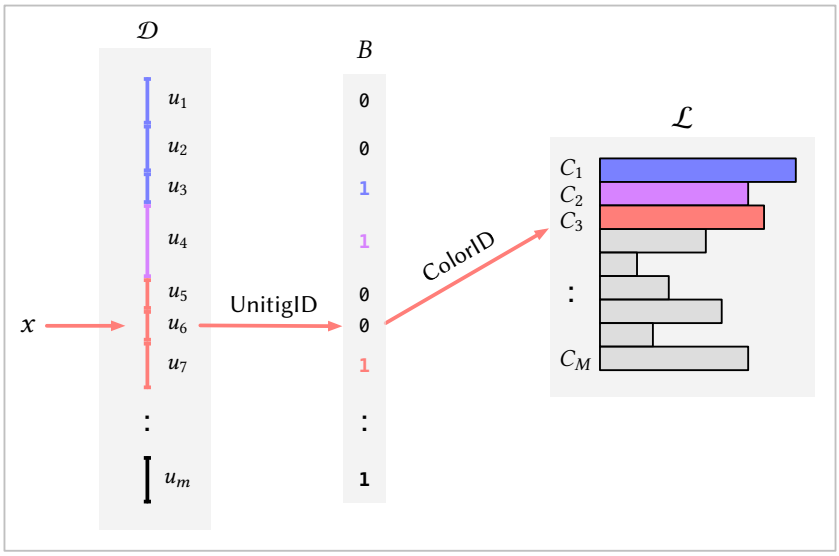


Figure 1 A schematic picture of the index described in Section 3, highlighting the interplay between the k-mer dictionary $\mathcal{D}$, the bit-vector B, and the inverted index $\mathcal{L}$. The red arrows show how the index is queried for a k-mer x, assuming that x occurs in unitig u_6 and has color C_3. The k-mer x is first mapped by $\mathcal{D}$ to its unitig u_6 via the query $\mathsf{UnitigID}(x) = 6$. Then we compute $\mathsf{ColorID}(u_6) = \mathsf{Rank}_1(6, B) + 1 = 2 + 1 = 3$ and lastly retrieve C_3 from $\mathcal{L}$.

3 Index description

In this section we describe a modular index that implements a colored compacted de Bruijn graph (ccdBG) and fully exploits its properties described in Section 2.3. We adopt the modular indexing framework from Section 2.2 – comprising a k-mer dictionary $\mathcal{D}$ and an inverted index $\mathcal{L}$ – to work seamlessly over the *unitigs* of the ccdBG. We extend the ideas from Fan et al. [11] for the modular indexing of k-mer positions to k-mer colors.

Our strategy is to first map k-mers to unitigs using a dictionary $\mathcal{D}$, and then map unitigs to their colors $\mathcal{C} = \{C_1, \ldots, C_M\}$. By *composing* these mappings, we obtain an efficient map directly from k-mers to their associated colors. The colors themselves in $\mathcal{C}$ are stored in compressed form in a inverted index $\mathcal{L}$. Figure 1 offers a pictorial overview of how we orchestrate these different components in the index. The goal of this section is to describe how these mapping steps can be performed efficiently and in small space.

3.1 The k-mer dictionary: mapping k-mers to unitigs with SSHash

For a k-mer dictionary, we use the SSHash data structure [30, 29], which fulfills the requirement described in Section 2.2, in that it implements the query $\mathsf{Lookup}(x)$ for any k-mer x efficiently and in compact space. This is achieved by storing the unitigs explicitly (i.e., as contiguous, 2-bit encoded strings) in some prescribed order so that a k-mer x occurring in some unitig u_i can be quickly located using a minimal perfect hash function [32] built for the set of the *minimizers* [38] of the k-mers. Laying out unitigs in this principled manner also enables very efficient streaming query. That is, when querying consecutive k-mers from input reads, the query for a given k-mer can often be answered very efficiently given the query result from its predecessor, since it often shares the same minimizer and frequently even occupies the very next position on the same unitig as its predecessor. We refer the interested reader to [30, 29] for a complete overview of SSHash.

Even more importantly for our purposes, a query into the SSHash dictionary returns, among other quantities, $\mathsf{UnitigID}(x) = i$, the ID of the unitig containing the k-mer x, as a byproduct of $\mathsf{Lookup}(x)$. For any k-mer occurring in $\mathcal{R}$, $\mathsf{UnitigID}(x) = i$ is an integer in $[1..m]$. This map from k-mers to unitigs will be exploited in the subsequent sections.

3.2 Mapping unitigs to colors

Now that we have an efficient map from k-mers to unitigs, i.e., the operation $\mathsf{UnitigID}(x)$, we must subsequently map unitigs to distinct colors. That is, we have to describe how to implement the operation $\mathsf{ColorID}(u_i)$ for each unitig u_i. Since each $\mathsf{ColorID}(u_i)$ is an integer in $[1..M]$, we could implement $\mathsf{ColorID}(u_i)$ just by storing $\mathsf{ColorID}(u_1), \ldots, \mathsf{ColorID}(u_m)$ explicitly in an array of $\lceil \log_2(M) \rceil$-bit integers. We show how to do this in just $1 + o(1)$ bits per unitig rather than $\lceil \log_2(M) \rceil$ bits per unitig.

We do so by exploiting another key property of SSHash: the unitigs it stores internally can be permuted in any desired order without impacting the correctness or efficiency of the dictionary. This was already noted and exploited in [29] to compress k-mer abundances. Similarly, here we sort the unitigs by $\mathsf{ColorID}(u_i)$, so that all the unitigs having the same color are stored consecutively in SSHash. To compute $\mathsf{ColorID}(u_i)$, all that is now required is a Rank_1 query over a bit-vector $B[1..m]$ where:

- $B[i] = 1$ if $\mathsf{ColorID}(u_i) \neq \mathsf{ColorID}(u_{i+1})$ and $B[i] = 0$ otherwise, for $1 \leq i < m$;
- $B[m] = 1$.

It follows that B has exactly M bits set. The operation $\mathsf{Rank}_1(i, B)$ returns the number of ones in $B[1, i)$ and can be implemented in $O(1)$ time, requiring only $o(m)$ additional bits as overhead on top of the bit-vector [41, 31]. This means that $\mathsf{ColorID}(u_i)$ can be computed in $O(1)$ as $\mathsf{Rank}_1(i, B) + 1$.

We illustrate this unitig to color ID mapping in Figure 1. In this toy example, $\mathsf{ColorID}(u_6) = 3$ can be computed with $\mathsf{Rank}_1(6, B) + 1 = 2 + 1$ because there are two bits set in $B[1, 6)$ – each marking where previous groups of unitigs with the same color end. Therefore, according to B, unitigs $\{u_1, u_2, u_3\}$ all have the same color as also $\{u_5, u_6, u_7\}$; u_4's color is not shared by any other unitig instead.

3.3 Compressing the colors

The inverted index $\mathcal{L}$ is a collection of sorted integer sequences $\{C_1, \ldots, C_M\}$, whose integers are drawn from a universe of size N (the total number of references in the collection $\mathcal{R}$). There is a plethora of different methods that may be used to compress integer sequences (see, e.g., the survey [34]). Testing the many different techniques available on genomic data is surely an interesting benchmark study to carry out. Here, however, we choose to adopt a simple strategy based on the widespread observation that effective compression appears to require using different strategies based on the density of the sequence C_i to be compressed (ratio between $|C_i|$ and N) [34]. For example, for the colored k-mer indexing problem, Alanko et al. also observe and report highly skewed distributions of color densities [2].

We therefore implement the following *hybrid* compression scheme:

1. For a sparse color set C_i where $|C_i|/N < 0.25$, we adopt a delta-gap encoding: the differences between consecutive integers are computed and represented via the universal Elias' δ code [10].
2. For a dense color set C_i where $|C_i|/N > 0.75$, we first take the complementary set of C_i, that is, the set $\overline{C_i} = \{j \in [1..N] | j \notin C_i\}$, and then compress $\overline{C_i}$ as explained in 1. above.

3. Finally, for a color set C_i, that does not fall into either above density categories, we store a characteristic bit-vector encoding of C_i – a bit-vector $b[1..N]$ such that $b[j] = 1$ if $j \in C_i$ and $b[j] = 0$ otherwise.

The compressed representations of all sequences are then concatenated into a single bit-vector, say *sequences*. An additional sorted sequence, *offsets*$[1..M]$, is used to record where each sequence begins in the bit-vector *sequences*, so that the compressed representation of the i-th sequence begins at the bit-position *offsets*$[i]$ in *sequences*, $1 \leq i \leq M$. The *offsets* sequence is compressed using the Elias-Fano encoding [9, 12] and takes only a (very) small part of the whole space of $\mathcal{L}$ unless the sequences are very short.

This hybrid encoding scheme is similar in spirit to the one also used in Themisto which, in turn, draws inspiration from Roaring bitmaps [7]. However, our choice of switching to the complementary set when $|C_i|$ approaches N turns out to be a very effective strategy, especially for pan-genome data, where a striking fraction of integers in $\mathcal{L}$ are indeed covered by these extremely dense sets (see also Table 4 from Section 5).

3.4 Construction

Fulgor is constructed by directly processing the output of GGCAT [8], an efficient algorithm to build ccdBGs using external memory and multiple threads. Importantly, GGCAT provides the ability to iterate over unitigs grouped by color. Therefore, Fulgor construction just requires a single scan of the unitigs in the order given by GGCAT. SSHash is built on the set of unitigs, each distinct color is compressed as described in Section 3.3, and the bit-vector B is also built during the scan.

4 Pseudoalignment algorithms

The term *pseudoalignment*, originally coined by Bray et al. [6] and developed in the context of RNA-seq quantification, has been used to describe many different algorithms and approaches, several of which do not actually comport with the original definition. Specifically, Bray et al. [6] define a "pseudoalignment of a read to a set of transcripts, T" as "a subset, $S \subseteq T$, without specific coordinates mapping each base in the read to specific positions in each of the transcripts in S". The goal of such an approach then becomes to determine, for a given read, the *set* of indexed reference sequences with which the read is *compatible*, where, in the most basic scenario, the compatibility relation can be determined entirely by the presence/absence of k-mers in the read in specific references.

Given any index of k-mer colors, a variety of different pseudoalignment algorithms can be implemented that rapidly map given reads to compatible reference sequences according to a set of heuristics. Below, we review four pseudoalignment algorithms and describe their properties. Various existing tools implement a subset of these pseudoalignment strategies and Fulgor implements all four. These algorithms fall into two categories: (1) *exhaustive* methods that retrieves the color of every k-mer on a given read (as described in [2]), and (2) *skipping* heuristics that skip or jump over k-mers during pseudoalignment that are likely to be *uninformative* (i.e., to have the same color as the k-mer that was just queried).

4.1 Exhaustive methods

For a given query sequence Q, exhaustive approaches return colors with respect to a set of k-mers of Q, $K(Q)$, that map to a non-empty color (i.e., each k-mer $x \in K(Q)$ if found in the dictionary $\mathcal{D}$).

Full-intersection. The first of the two exhaustive approaches, the *full-intersection* method, simply returns the intersection between all the colors of the k-mers in $K(Q)$. Algorithm 1 in the Appendix (page 20) shows how this query mode is implemented in Fulgor. In the current implementation, Fulgor has a generic intersection algorithm that can work over *any* compressed color sets, provided that an iterator over each color supports two primitives – Next and NextGEQ(x), respectively returning the integer immediately after the one currently pointed to by the iterator and the smallest integer which larger-than or equal-to x. (We point the reader to [26] and [34] for details.)

Threshold-union. The second algorithm, which we term the *threshold-union* approach, relaxes the full-intersection method to trade off precision for increased recall. Instead of requiring a reference to be compatible with *all* mapped k-mers, the threshold-union method requires a reference to be compatible with a user defined proportion of k-mers. Given a parameter $\tau \in (0, 1]$, this method returns the set of references that occur in *at least* $s \cdot \tau$ returned (i.e., non-empty) k-mer colors, where s can be either chosen to be $s = |K(Q)|$ (the number of positive k-mers only) or $s = |Q| - k + 1$ (the total number of k-mers in Q). Themisto [2] implements the variant with $s = |K(Q)|$ (called the "hybrid" method), whereas both Bifrost [15] and Metagraph [17] use $s = |Q| - k + 1$. In fact, the latter approach of simply looking up all of the k-mers in a query, and requiring a specified fraction of them to match, is a long-standing strategy that predates the notion of pseudoalignment [44, 43]. In the following, we assume $s = |K(Q)|$ is used by the threshold-union algorithm, unless otherwise specified. The pseudocode for this query mode is given in Algorithm 3 in the Appendix (page 21).

In practice, both the aforementioned exhaustive methods are efficient to compute for two reasons. First, intersections, thresholding, and unions are easy to compute because colors are encoded as monotonically increasing lists of reference IDs. Second, for Fulgor in particular, querying *every* k-mer for its color can be performed in a highly-optimized way via *streaming* queries to SSHash. In the streaming setting, SSHash may skip comparatively slow hashing and minimizer lookup operations because it stores *unitig* sequences contiguously in memory. When sequentially querying adjacent k-mers on a read that are also likely adjacent on indexed unitigs, it can rapidly lookup and check k-mers that are cached and adjacent in memory (we refer the reader to [30] for more details).

4.2 *Skipping* heuristics

For even faster read mapping, pseudoalignment algorithms can implement heuristic *skipping* approaches that avoid exhaustively querying all k-mers on a given read. These skipping heuristics make the assumption that whenever a k-mer on a read is found to belong to a unitig, subsequent k-mers will likely map to the same unitig and can therefore be skipped, since they will be uninformative with respect to the final color assigned to the query (i.e., the intersection of the colors of the mapped k-mers).

Bray et al. [6] first described such an approach, where a successful search that returns a unitig u triggers a skip that moves the search position forward to either the end of the query or the implied distance to the end of u (whichever is less). Subsequent searches follow the same approach as new unitigs are discovered and traversed in the query. Later, other tools extended or modified the proposed skipping heuristics, and introduced "structural constraints", which take into account the co-linearity and spacing between matched seeds on the query and on the references to which they map [13]. In contrast to Themisto, Fulgor has

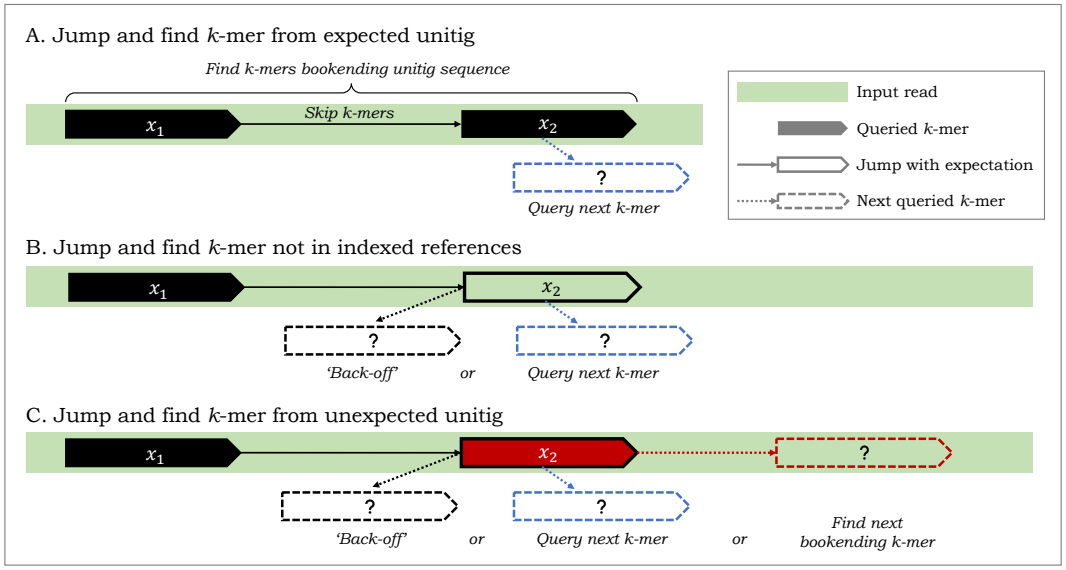


Figure 2 Some relevant design choices for pseudoalignment with skipping heuristics that *jump* and skip k-mers on a given read. After k-mer x_1 is queried and found to map to a "black" unitig, an algorithm can jump to query the k-mer x_2 on input read, where the number of k-mers skipped is given by the length of the black unitig. (A) In the ideal scenario, x_2 maps to the black unitig sequence and k-mers x_1 and x_2 are found to bookend this unitig sequence as it appears on the read. (B) If x_2 misses the index, an algorithm can *back-off* to an earlier k-mer on the read to find a k-mer bookending a shorter subsequence of the black unitig; or it may just query the next k-mer. (C) If x_2 maps to a different "red" unitig, an algorithm has an alternative, aggressive, heuristic option to jump and find the next k-mer bookending the red unitig sequence.

rapid access to the topology of the ccdBG because its k-mer dictionary, SSHash, explicitly maps k-mers to unitig sequences that are stored contiguously in memory. Fulgor thus permits efficient implementation of pseudoalignment algorithms with skipping heuristics since, due to the underlying capabilities provided by SSHash, it can rapidly find k-mers bookending unitig substrings because SSHash can explicitly map k-mers to their offsets (positions) in indexed unitig sequences.

In general, pseudoalignment methods that implement skipping heuristics must specify what steps the algorithm will take in *all* scenarios, not just what should happen when search proceeds as expected. In practice, implementations for resolution strategies are complicated and difficult to describe succinctly in prose, and prior work has only discussed these important details in passing. Here, using the depicted scenarios in Figure 2, we provide a more structured (though certainly not exhaustive) discussion of possible design choices that can be made. These design choices impact the performance of the pseudoalignment algorithm, both in terms of how many k-mers it queries (and, hence, its speed), and in how many distinct color sets it collects (and, hence, the actual compatibility assignment it makes).

Jump and find k-mer in expected unitig. Before the first matching k-mer of a read is found, there is relatively little difference between exhaustive and heuristic pseudoalignment approaches; subsequent k-mers are queried until the read is exhausted or some k-mer is found in the index. At this point, however, heuristic skipping methods diverge from the exhaustive approaches. At a high level, when a k-mer on a read is found to map to a unitig, skipping heuristics make an assumption that said unitig appears wholly on the read. A

pseudoalignment algorithm then jumps, on the read, to what would be the last k-mer on the unitig sequence occurring on the given read (i.e., a bookending k-mer). Scenario A in Figure 2 depicts when this assumption is correctly made. Moving left-to-right on a given read, if a k-mer on the *left* is found to occur on the unitig depicted in black color in the figure (referred to as the "black" unitig henceforth), an algorithm can then skip a distance given by the length of the black unitig and jump to a k-mer to the right that also maps to the black unitig and bookends it. Doing so, an algorithm can assume that all k-mers bookended by these two queried k-mers map to the black unitig, avoid querying k-mers in-between, and instead continue to query the next k-mer on the read (indicated in dashed lines in blue).

Jump and miss k-mer. In practice however, the implemented skipping heuristics are not so simple. This is because, when skipping k-mers according to unitig lengths, the resulting k-mer that an algorithm jumps to may not necessarily map to the unitig it *expects*. In scenario B, an algorithm jumps to a k-mer on a read, expecting it to map to a black unitig, but finds that it does not correspond to any indexed k-mer. Here, an algorithm can make several choices, and in fact, current skipping heuristics make two distinct choices in this scenario. It can ignore this missed k-mer and simply query the next k-mer after the position that was jumped to (in blue). Or, it can take a more conservative approach and implement a *back-off* scheme to look for another k-mer that maps to the black unitig. An algorithm can back-off and jump a lesser distance, and such a back-off approach can happen once or can be recursive or iterative until some termination condition is satisfied.

Jump and find k-mer in *un*-expected unitig. In scenario C, an algorithm that jumps to a k-mer but finds that it maps to a *different* (red) unitig than expected. Here, we suggest three choices an algorithm can make. Like in scenario B, an algorithm can back-off to find another k-mer mapping to the black unitig or it can query the next k-mer after the jumped position. Alternatively, it can take a new more aggressive approach and jump to a k-mer on the read where it expects to find the end of an occurrence of the red unitig.

In this work, we have retrofit the pseudoalignment with skipping algorithms from Kallisto [6][1] and Alevin-fry [13][2] to make use of Fulgor, rather than the distinct indexes atop which they were implemented in their original work. Using Fulgor, we compare their resulting pseudoalignments, along with those from the full-intersection and threshold-union approaches, in a simple simulated scenario in Section 5.4.

5 Results

In this section, we report experimental results to assess Fulgor's construction time/space, index size, and query speed. Throughout the section, we compare Fulgor to Themisto [2], which has been shown to outperform other methods that build similarly capable indexes (namely Bifrost [15] and Metagraph [17]) in terms of speed and space. We build Themisto indexes using the fastest configuration, i.e., without sampling of k-mer colors in the SBWT (build option `-d1`), as done by the authors in [2]. Not sampling k-mer colors yields slightly larger indexes but makes Themisto faster to query. For our largest benchmarked reference collection (150,000 genomes), potential space savings from sampling is not significant anyway

[1] `https://github.com/jermp/fulgor/blob/main/kallisto_psa/psa.cpp`
[2] `https://github.com/jermp/fulgor/blob/main/piscem_psa/hit_searcher.cpp`

Table 1 Summary statistics for the tested collections. The row "Integers in colors" reports the total number of reference IDs that are required to encode all colors – i.e., the sum set sizes for all colors, $\sum_i |C_i|$.

	Salmonella					Gut Bacteria
Genomes	5,000	10,000	50,000	100,000	150,000	30,691
Distinct colors ($\times 10^6$)	2.69	4.24	13.92	19.36	23.61	227.80
Integers in colors ($\times 10^9$)	5.77	15.68	133.49	303.53	490.04	10.04
k-mers in dBG ($\times 10^6$)	104.69	239.88	806.23	1,018.69	1,194.44	13,936.86
Unitigs in dBG ($\times 10^6$)	4.95	8.24	30.64	41.16	49.60	566.39

because the space required to store distinct colors dominate the overall space. We also use Themisto's default color set representation (i.e., without Roaring bitmaps). For both Fulgor and Themisto, we set the k-mer size to $k = 31$.

Datasets. We follow the experimental methodology of [2] and build Fulgor over subsets of *Salmonella enterica* genomes (up to 150,000 genomes) from [5], to demonstrate Fulgor's effectiveness when indexing collections of similar reference sequences. We also consider a heterogeneous collection of 30,691 genomes of bacterial species representative of the human gut [14] (as also benchmarked in our previous work [11]). We report some summary statistics for the indexed ccdBGs in Table 1.

Hardware and software. All experiments were run on a machine equipped with Intel Xeon Platinum 8276L CPUs (clocked at 2.20GHz), 500 GB of RAM running Ubuntu 18.04.6 LTS (GNU/Linux 4.15.0). Fulgor is available at `https://github.com/jermp/fulgor`. For the experiments reported here we use v1.0.0 of the software, compiled with gcc 11.1.0. For Themisto, we use the shipped compiled binaries (v3.1.1).

5.1 Construction time and space

Construction time and peak RAM usage is reported in Table 2 for the different datasets evaluated. Both tools use GGCAT to build the ccdBG. However, Fulgor is $2-6\times$ faster, and typically consumes much less memory during construction. This is because Themisto spends most of its time and memory building the color mapping. However, the analogous component of Fulgor is just a bit vector, demarcating groups of unitigs with the same color, that is built via a linear scan of the unitigs produced by GGCAT.

Figure 3 shows, instead, Fulgor's construction time breakdown for some illustrative datasets. We distinguish between three phases in the construction: (1) running GGCAT, (2) compressing the colors and, (3) building SSHash. While GGCAT and color compression take most of the construction time on the Salmonella pangenomes, building SSHash is the most expensive step on the Gut Bacteria collection. This is consistent with the statistics reported in Table 1. Here, there are far more integers to compress in the Salmonella collections whereas the Gut Bacteria collection contains one order of magnitude more k-mers. This suggests that one could achieve even faster construction for Fulgor if the colors are compressed in parallel with the SSHash construction (currently, these two phases are sequential).

Table 2 Total index construction time and GB of memory (max. RSS), as reported by `/usr/bin/time` with option `-v`. The reported time includes the time taken by GGCAT to build the ccdBG (using 48 processing threads) and the time to serialize the index on disk.

	Genomes	Fulgor		Themisto	
		hh:mm	GB	hh:mm	GB
Salmonella	5,000	00:04	12.91	00:11	12.97
	10,000	00:09	23.60	00:25	23.58
	50,000	01:13	43.76	02:32	96.00
	100,000	02:56	73.54	06:25	202.42
	150,000	04:36	136.94	10:00	323.10
Gut Bacteria	30,691	02:27	115.05	15:35	327.72

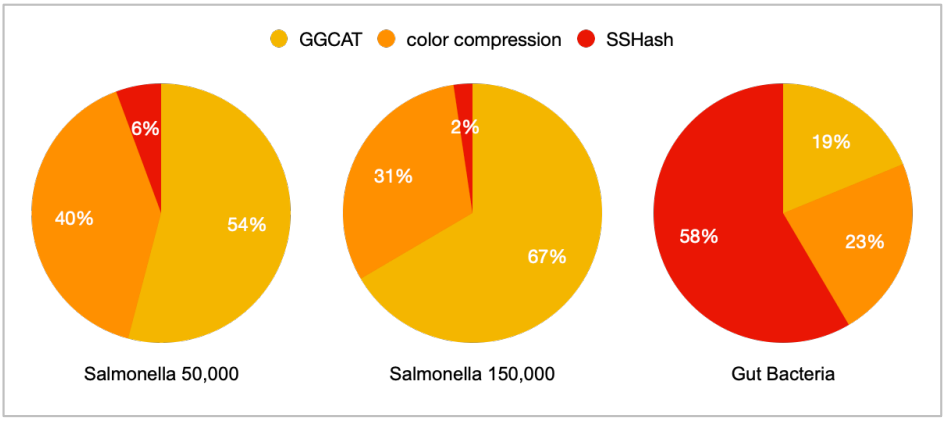


Figure 3 Construction time breakdown for Fulgor.

5.2 Index size

When indexing collections of Salmonella genomes, Fulgor is consistently $\approx 2\times$ smaller than Themisto as apparent from Table 3. For example, on the largest collection comprising 150,000 genomes, Fulgor takes 70.66GB whereas Themisto takes 133.63GB. This remarkable space improvement is primarily due to the more effective color compression scheme adopted by Fulgor. This leads to, for example, 48% less space to encode colors for the 150,000 collection of Salmonella genomes. Looking at Table 4, we highlight that for all indexed Salmonella reference collections, approximately 50% of all encoded integers in the distinct colors belong to colors that are *at least* 90% dense. For such extremely dense colors, the complementary encoding strategy described in Section 3.3 is very effective: only ≈ 0.2 bits/int (bpi) are required to encode them in all benchmarked indexes. In fact, even for our largest collection of 150,000 Salmonella genomes, encoding *all* integers in *all* colors requires only 1.120 bpi.

Unsurprisingly, Fulgor also uses less space than Themisto to support the ColorID operation. We recall from Section 3.2 that Fulgor requires only $1 + o(1)$ bits per unitig by design. This amounts to a negligible space usage compared to the overall index size. For example, while Themisto requires 7.26GB to map k-mers to color IDs for 150,000 Salmonella genomes, our strategy just takes 7.75 MB.

Table 3 Index space in GB, broken down by space required for indexing the k-mers in a dBG (SSHash for Fulgor, and the SBWT for Themisto); and data structures required to encode colors and map k-mers to colors.

	Genomes	Fulgor			Themisto		
		dBG	Colors	Total	dBG	Colors	Total
Salmonella	5,000	0.16	0.59	0.75	0.14	1.82	1.96
	10,000	0.35	1.66	2.01	0.32	4.78	5.09
	50,000	1.26	17.03	18.30	1.07	36.89	37.96
	100,000	1.72	40.70	42.44	1.35	81.82	83.17
	150,000	2.03	68.60	70.66	1.58	132.05	133.63
Gut Bacteria	30,691	21.23	15.45	36.77	18.33	121.08	139.41

When indexing a *heterogeneous* collection, e.g., the 30,691 bacterial genomes [14], with many more unique k-mers, the space advantage Fulgor has over Themisto is even more apparent. First, the overall size of Fulgor is 3.8× smaller (36.77GB versus 139.41GB). Second, Fulgor's near optimal approach of mapping *unitigs* to colors instead of k-mers to colors is dramatically more efficient, requiring only 88MB compared to Themisto's 91GB. Themisto, by using the SBWT, organizes k-mers based on their *colexicographical* order and requires $\lceil \log_2(M) \rceil$ bits per sampled k-mer to record the color IDs. Here, the SBWT must record colors for each of the 13.9 billion distinct k-mers *and* their reverse complement. In contrast, Fulgor uses SSHash that maintains k-mers in unitig order and requires only $1 + o(1)$ bits per unitig to map *all* k-mers from the same unitig to a single color. Although not the default behavior, Themisto can optionally sample k-mer colors to avoid storing one color ID per each k-mer. Clearly, the sampling scheme reduces space usage at the expense of some overhead at query time by requiring an implicit walk in the dBG. While this sampling strategy can be quite effective when the underlying k-mer set induce long unitigs, allowing the sampling of the terminal k-mers of a non-branching path [2], it is unlikely to be similarly effective in a highly-branching and fragmented graph like the one underlying this heterogeneous dataset where unitigs are short. On the contrary, our index does not have this issue *by design* and can thus scale to more heterogenous collections using small space.

5.3 Query speed

To compare query speed, we benchmark Fulgor and Themisto using both low- and high-hit rate read-sets, i.e., read-sets for which we have a low and high number of positive k-mers respectively. Precisely, we use the files containing the first read of the following paired-end libraries: SRR896663[3] with 5.7×10^6 reads, SRR801268[4] with 6.6×10^6 reads, and ERR321482[5] with 6.8×10^6 reads.

In Table 5 we report the result of the comparison using the full-intersection method (Algorithm 1). We repeated the same experiment using the threshold-union method (Algorithm 3) with parameter $\tau = 0.8$ as this is the preferred query mode in Themisto. However, we did not observe any appreciable difference compared to the full-intersection method in terms of query speed.

[3] https://www.ebi.ac.uk/ena/browser/view/SRR896663
[4] https://www.ebi.ac.uk/ena/browser/view/SRR801268
[5] https://www.ebi.ac.uk/ena/browser/view/ERR321482

Table 4 Average bits/int (bpi) spent for representing colors whose density is $(i \times 10)\%$ of N, for $i = 1, \ldots, 10$. The first two columns for each collection, "lists" and "ints", report the percentage of lists (i.e., colors) and integers (stored reference identifiers) that belong to all colors within a given density. The last row, "Total bpi", is comprehensive of the space spent for the integer lists themselves and the space spent for the offsets delimiting the lists' boundaries.

Density	Salmonella															Gut Bacteria		
	$N = 5{,}000$			$N = 10{,}000$			$N = 50{,}000$			$N = 100{,}000$			$N = 150{,}000$			$N = 30{,}691$		
	lists	ints	bpi	lists	ints	bpi	lists	ints	bpi	lists	ints	bpi	lists	ints	bpi	lists	ints	bpi
0–10%	38.88	2.00	4.66	46.15	1.81	4.64	70.96	2.62	6.00	76.52	3.14	6.16	79.23	3.27	6.32	99.99	99.99	12.05
10–20%	6.04	2.11	2.66	4.83	1.93	2.66	3.74	2.84	3.05	2.82	2.61	3.77	2.54	2.68	3.92	0.00	0.00	0.00
20–30%	4.70	2.69	2.86	4.44	2.93	2.81	2.69	3.50	3.24	2.32	3.66	3.41	2.09	3.76	3.46	0.00	0.00	0.00
30–40%	3.13	2.55	2.88	4.27	4.02	2.87	1.90	3.43	2.89	1.57	3.49	2.88	1.40	3.51	2.88	0.00	0.00	0.00
40–50%	4.05	4.25	2.23	3.32	4.04	2.22	1.81	4.25	2.22	1.44	4.14	2.23	1.29	4.19	2.23	0.00	0.00	0.00
50–60%	4.13	5.30	1.83	3.54	5.29	1.81	1.82	5.24	1.82	1.42	4.99	1.82	1.24	4.94	1.82	0.00	0.00	0.00
60–70%	3.98	6.07	1.54	4.24	7.44	1.54	2.04	6.94	1.53	1.59	6.61	1.54	1.40	6.59	1.54	0.00	0.00	0.00
70–80%	5.53	9.72	0.94	4.86	9.91	0.93	2.33	9.13	1.08	1.87	8.96	1.14	1.64	8.91	1.15	0.00	0.00	0.00
80–90%	5.80	11.52	0.47	3.71	8.57	0.47	3.03	13.43	0.56	2.49	13.65	0.63	2.09	12.95	0.66	0.00	0.00	0.00
90–100%	23.77	53.80	0.15	20.65	54.07	0.14	9.67	48.63	0.19	7.94	48.76	0.21	7.07	49.21	0.21	0.00	0.00	0.00
Total bpi	0.817			0.848			1.020			1.072			1.120			12.32		

In a low-hit rate workload where a small proportion of reads map to the indexed references, Fulgor is much faster than Themisto. In this scenario, we expect many queried k-mers to not occur in the indexed references. When k-mers are absent from the index, no color needs to be retrieved and only the k-mer dictionary is queried. Here, Fulgor is faster than Themisto because its reliance on the fast streaming query capabilities of SSHash. It is worth noting here that in any *streaming* setting where consecutive k-mers are queried, Fulgor can fully exploit the monochromatic property of unitigs in ways which Themisto cannot. Queries to SSHash have very good locality compared to the SBWT because adjacent k-mers in unitigs are stored contiguously in memory. Further, streaming queries to SSHash can be very efficiently cached and optimized. When looking up consecutive k-mers, SSHash can entirely avoid computing its minimal perfect hash (a slow operation) and instead perform fast comparisons of k-mers stored in cached positions pointing to adjacent addresses in memory.

In a high-hit rate workload, Fulgor also outperforms Themisto, but by a smaller margin, since most of the time is now spent in performing the intersection between colors. It is interesting to note that the workloads can be processed significantly faster (by both tools) on the Gut Bacteria collection: this is a direct consequence of the fact that the lists being intersected are much shorter on average for the Gut Bacteria compared to the Salmonella collections. This is evident from Table 4: essentially all lists are just 10% dense, i.e., have length at most $\lceil 30,691/10 \rceil < 3{,}070$.

We also note that part of the slowdown seen for Themisto is due to the time spent in loading the index from disk to RAM, which takes at least twice as Fulgor's because of its larger index size.

5.4 Comparison of pseudoalignment algorithms on simulated data

To analyze the accuracy of the underlying pseudoalignment algorithms, we perform additional testing with read sets simulated using the Mason [16] simulator. To analyze how mapping and hit rates affect query speed, we simulate a varying proportion of "positive" reads from

Table 5 Total query time as elapsed time reported by `/usr/bin/time`, using 16 processing threads for both indexes. The read-mapping output is written to `/dev/null` for this experiment. We also report the mapping rate in percentage (fraction of mapped read over the total number of queried reads). Results are relative to the full-intersection query mode. All reported timings are relative to a second run of the experiment, when the index is loaded faster from the disk cache. For each workload, we indicate the run accession number.

(a) low-hit, Salmonella, SRR896663

Genomes	Mapping rate	Fulgor	Themisto
		mm:ss	mm:ss
5,000	1.27	00:09	00:32
10,000	13.86	00:10	00:36
50,000	32.61	00:25	01:05
100,000	34.09	00:45	01:39
150,000	34.01	01:06	05:02

(b) high-hit, Salmonella, SRR801268

Genomes	Mapping rate	Fulgor	Themisto
		mm:ss	mm:ss
5,000	89.53	01:16	03:50
10,000	89.76	02:26	07:35
50,000	91.31	19:15	41:25
100,000	91.52	35:50	82:14
150,000	91.61	42:30	120:08

(c) low-hit, Gut Bacteria, SRR896663

Genomes	Mapping rate	Fulgor	Themisto
		mm:ss	mm:ss
30,691	11.90	0:57	2:58

(d) high-hit, Gut Bacteria, ERR321482

Genomes	Mapping rate	Fulgor	Themisto
		mm:ss	mm:ss
30,691	92.98	01:16	02:45

indexed reference sequences and generate "negative" reads from the human chromosome 19 from the CHM13 v2.0 human genome assembly [25]. We use Fulgor to compare the four mapping algorithms described in Section 4.

From Table 6, we see that at various proportions of ground truth positive reads (simulated reads deriving from indexed references), all mapping methods have a true positive rate (TPR), i.e., total reads correctly mapped over the total ground truth positives, greater than 95%. This high sensitivity for all four methods is to be expected since all methods simply check for k-mer's membership to references of origin and do not consider k-mer positions in references. One main drawback of eliding positions, heuristically avoiding "locate" queries, and entirely ignoring k-mers that are not present in the index, is also clear. All methods incur approximately a 30% false positive rate (FPR), i.e., total reads spuriously mapped over the total ground truth negatives. As is expected, the threshold-union method incurs a slightly higher FPR compared to other methods (30% compared to 27% for other methods) because of its less strict criteria only requiring references to be compatible with τ fraction of mapped k-mers instead of *all* k-mers.

In these benchmarks, we find very little difference in terms of TPR and FPR between the exhaustive methods and skipping heuristics. These results also gesture at one desirable and one undesirable quality of these methods. First, skipping heuristics correctly assume and successfully skip k-mers that likely occur on the same unitig and have the same color. Likewise, they have the *potential* to be even more sensitive than the full-intersection method, as they do not, in general, search for every k-mer in a query, and can thus avoid scenarios where variation or sequencing errors in a query cause spurious matches to the index, shrinking or eliminating the set of references appearing in the final color assigned to the query. In fact, in a small-scale test, Alanko et al. [2] report that Kallisto's skipping heuristic results in a small but persistent increase of approximately 0.03% in the mapping rate. However, all four

Table 6 Quality of pseudoalignment algorithms querying 100,000 simulated reads against 50,000 Salmonella genomes indexed with Fulgor. We vary the percentage of *positive* reads simulated from indexed Salmonella genomes by diluting queried read sets with *negative* reads simulated from a reference human transcriptome. We consider a mapped positive read (deriving from indexed references) to be a *true positive* if the reference of origin is in the returned set of compatible references; and a mapped negative read (deriving from human chromosome 19) to be a *false positive*. We denote true and false positive rates (%) to be TPR and FPR, respectively. For the threshold-union method, we use $\tau = 0.8$.

% Positive	Full-intersection		Threshold-union		Kallisto		Alevin-fry	
	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR
90%	95.0	27.0	97.7	30.0	95.0	27.0	95.1	27.0
70%	95.1	27.0	97.7	30.0	95.1	27.0	95.1	27.0
25%	95.1	27.0	97.7	30.0	95.2	27.0	95.2	27.0
10%	95.5	27.0	97.8	30.0	95.5	27.0	95.5	27.0

of the pseudoalignment methods evaluated here suffer from a high FPR and low precision. Better algorithms to lower FPR and improve precision without lowering sensitivity too much should be investigated in future work. Such improvements may be possible by adding back information about the *reference positions* where k-mers from the query match, incorporating structural constraints [13] or other such restrictions atop the color intersection rule. Yet, those approaches are more computationally involved, require the index to support locate queries, and also substantially diverge from "pseudoalignment" as traditionally understood. Regardless, we highlight here that Fulgor more easily enables implementing skipping and unitig-based heuristics compared to other methods that do not explicitly store unitig sequences and keep k-mers in unitig order. In fact, Fulgor implicitly maintains additional information regarding the *local structural consistency* of k-mers. For example, with Fulgor, one can easily check if consecutive k-mers are valid on an indexed unitig or check if consecutive unitigs on a read have valid overlaps, in an attempt to reduce the FPR.

6 Conclusions and future work

We introduce Fulgor, a fast and compact index for the k-mers of a colored compacted de Bruijn graph (ccdBG). Using, SSHash, an order-preserving k-mer dictionary, Fulgor fully exploits the monochromatic property of unitigs in ccdBGs. Fulgor implements a very succinct map from unitigs to colors, taking only $1 + o(1)$ bits per unitig. Further, Fulgor applies an effective hybrid compression scheme to represent the set of distinct colors. Across all benchmarked scenarios, Fulgor outperforms Themisto, the prior state-of-the-art in terms of space *and* speed. There is still room for improvement in future work. We discuss some promising directions below.

In terms of speed, we remark that when processing a high-hit workload, the overall runtime is dominated by the time required to *intersect* the colors. As explained in Section 4.1, Fulgor currently implements a generic intersection algorithm that only requires two primitive operations, namely Next and NextGEQ (see also Appendix A). But this is not the only paradigm available for efficient intersection. We could, for example, try approaches that exploit different indexing paradigms, such as Roaring [7] and Slicing [28], that are explicitly designed for fast intersections. These alternative approaches may be significantly faster especially on the high-hit workloads.

Another possible optimization is to implement a caching scheme for frequently occurring and/or recently intersected colors. Caching the uncompressed or intersection-optimized versions of frequently occurring color sets, or previously computed intersections, could speed up query processing substantially when many reads map to the same set of colors.

In terms of space, one property that Fulgor does not yet exploit is the fact that *many* unitigs in the ccdBG share *similar colors* – i.e., co-occur in many reference sequences. This is so because unitigs arising from conserved genomic sequences will share similar occurrence patterns. In a related line of research, [3] developed a method that efficiently compresses distinct, but highly-correlated colors, through a variant of referential encoding. Specifically, they compute a minimum spanning tree (MST) on a subgraph of the color graph induced by the ccdBG, and encode a color by recording its differences with respect to its parent in the MST. This vastly reduces the space required to encode the color set when many similar colors exist, as we would expect in a pangenome, and fast query speed can be retained through color caching. Another related approach would be to resort to clustering similar colors and encoding all colors within a cluster with respect to a cluster representative color [33]. Likewise, although not specifically designed to compress colors, Metagraph and its variants can exploit similarity between colors using a general compression scheme that records differences in stored metadata (in this case, the colors) between adjacent k-mers [17]. We note that, since the colored k-mer indexing problem is *modular* (Section 2.2), novel relational compression techniques for the set of distinct colors can be developed and optimized independently of the other components of the index.

Finally, in our experiments with simulated data analyzing the quality of pseudoalignment algorithms from Section 5.4, we find higher than desirable false positive rates. This suggests that, at least for the metagenomic and pangenomic reference collections where many references share similar k-mer content, better read-mapping heuristics and algorithms that improve specificity (i.e., reduce the spurious mapping of reads not arising from indexed references) without trading-off too much recall are still sorely needed. Here, it will be desirable to search for methods that can improve specificity without the need to retain reference positions or issue locate queries for all k-mers. We suggest that there may be several promising directions. For example, one may consider enforcing local structural consistency among matched k-mers to potentially reduce spurious mapping. Likewise, one may consider filtering repetitive and low-complexity k-mers from contributing to the final pseudoalignment result. Finally, by analogy to BLAST [4], one may consider evaluating the likelihood that a pseudoalignment result is spurious by comparing the matching rate against against some null or background expectation to account for the fact that, in very large reference databases, a very small number of (potentially correlated) k-mers may be insufficient evidence to consider a query as compatible with a subset of references.

References

1 Jarno N. Alanko, Simon J. Puglisi, and Jaakko Vuohtoniemi. Small searchable k-spectra via subset rank queries on the spectral burrows-wheeler transform. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA23)*, pages 225–236, 2023.

2 Jarno N Alanko, Jaakko Vuohtoniemi, Tommi Mäklin, and Simon J Puglisi. Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. In *Proceedings of the 31st Conference on Intelligent Systems for Molecular Biology (ISMB)*, 2023.

3 Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. An Efficient, Scalable, and Exact Representation of High-Dimensional Color Information Enabled Using de Bruijn Graph Search. *Journal of Computational Biology*, 27(4):485–499, 2020. PMID: 32176522.

4 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.

5 Grace A. Blackwell, Martin Hunt, Kerri M. Malone, Leandro Lima, Gal Horesh, Blaise T. F. Alako, Nicholas R. Thomson, and Zamin Iqbal. Exploring bacterial diversity via a curated and searchable snapshot of archived DNA sequences. *PLOS Biology*, 19(11):1–16, November 2021. doi:10.1371/journal.pbio.3001421.

6 Nicolas L Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic rna-seq quantification. *Nature biotechnology*, 34(5):525–527, 2016.

7 Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 46(5):709–719, 2016.

8 Andrea Cracco and Alexandru I Tomescu. Extremely-fast Construction and Querying of Compacted and Colored de Bruijn Graphs with GGCAT. In *Proceedings of the 27th Annual International Conference on Research in Computational Molecular Biology*, pages 208–210. Springer, 2023.

9 Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.

10 Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

11 Jason Fan, Jamshed Khan, Giulio Ermanno Pibiri, and Rob Patro. Spectrum preserving tilings enable sparse and modular reference indexing. In *Research in Computational Molecular Biology*, pages 21–40, 2023.

12 Robert Mario Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT*, 1971.

13 Dongze He, Mohsen Zakeri, Hirak Sarkar, Charlotte Soneson, Avi Srivastava, and Rob Patro. Alevin-fry unlocks rapid, accurate and memory-frugal quantification of single-cell RNA-seq data. *Nature Methods*, 19(3):316–322, 2022.

14 Pranvera Hiseni, Knut Rudi, Robert C Wilson, Finn Terje Hegge, and Lars Snipen. HumGut: a comprehensive human gut prokaryotic genomes collection filtered by metagenome data. *Microbiome*, 9(1):1–12, 2021.

15 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.

16 M. Holtgrewe. Mason – a read simulator for second generation sequencing data. *Technical Report FU Berlin*, October 2010.

17 Mikhail Karasikov, Harun Mustafa, Amir Joudaki, Sara Javadzadeh-No, Gunnar Rätsch, and André Kahles. Sparse Binary Relation Representations for Genome Graph Annotation. *J Comput Biol*, 27(4):626–639, December 2019.

18 Jamshed Khan, Marek Kokot, Sebastian Deorowicz, and Rob Patro. Scalable, ultra-fast, and low-memory construction of compacted de Bruijn graphs with Cuttlefish 2. *Genome Biology*, 23(1):190, 2022.

19 Jamshed Khan and Rob Patro. Cuttlefish: fast, parallel and low-memory compaction of de Bruijn graphs from large-scale genome collections. *Bioinformatics*, 37(Supplement_1):i177–i186, 2021.

20 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.

21 Tommi Mäklin, Teemu Kallonen, Sophia David, Christine J Boinett, Ben Pascoe, Guillaume Méric, David M Aanensen, Edward J Feil, Stephen Baker, Julian Parkhill, et al. High-resolution sweep metagenomics using fast probabilistic inference [version 1; peer review: 1 approved, 1 approved with reservations]. *Wellcome open research*, 5(14), 2021.

22 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, 2021.

23 Alexa B. R. McIntyre, Rachid Ounit, Ebrahim Afshinnekoo, Robert J. Prill, Elizabeth Hénaff, Noah Alexander, Samuel S. Minot, David Danko, Jonathan Foox, Sofia Ahsanuddin, Scott Tighe, Nur A. Hasan, Poorani Subramanian, Kelly Moffat, Shawn Levy, Stefano Lonardi, Nick Greenfield, Rita R. Colwell, Gail L. Rosen, and Christopher E. Mason. Comprehensive benchmarking and ensemble approaches for metagenomic classifiers. *Genome Biology*, 18(1):182, September 2017.

24 Ilia Minkin, Son Pham, and Paul Medvedev. TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinformatics*, 33(24):4024–4032, 2017.

25 Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V. Bzikadze, Alla Mikheenko, et al. The complete sequence of a human genome. *Science*, 376(6588):44–53, 2022.

26 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.

27 Rachid Ounit, Steve Wanamaker, Timothy J Close, and Stefano Lonardi. Clark: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers. *BMC Genomics*, 16(1):1–13, 2015.

28 Giulio Ermanno Pibiri. Fast and compact set intersection through recursive universe partitioning. In *2021 Data Compression Conference (DCC)*, pages 293–302. IEEE, 2021.

29 Giulio Ermanno Pibiri. On weighted k-mer dictionaries. In *International Workshop on Algorithms in Bioinformatics (WABI)*, pages 9:1–9:20, 2022.

30 Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *Bioinformatics*, 38(Supplement_1):i185–i194, June 2022.

31 Giulio Ermanno Pibiri and Shunsuke Kanda. Rank/select queries over mutable bitmaps. *Information Systems*, 99(101756), 2021.

32 Giulio Ermanno Pibiri and Roberto Trani. PTHash: Revisiting FCH minimal perfect hashing. In *Proceedings of the 44th international ACM SIGIR conference on Research & development in information retrieval*, pages 1339–1348, 2021.

33 Giulio Ermanno Pibiri and Rossano Venturini. Clustered elias-fano indexes. *ACM Transactions on Information Systems*, 36(1):1–33, 2017.

34 Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression. *ACM Comput. Surv.*, 53(6):125:1–125:36, 2021.

35 N Tessa Pierce, Luiz Irber, Taylor Reiter, Phillip Brooks, and C Titus Brown. Large-scale sequence comparisons with sourmash. *F1000Research*, 8, 2019.

36 NT Pierce, L Irber, T Reiter, P Brooks, and CT Brown. Large-scale sequence comparisons with sourmash [version 1; peer review: 2 approved]. *F1000Research*, 8(1006), 2019. doi:10.12688/f1000research.19675.1.

37 Mark Reppell and John Novembre. Using pseudoalignment and base quality to accurately quantify microbial community composition. *PLOS Computational Biology*, 14(4):1–23, April 2018.

38 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.

39 L Schaeffer, H Pimentel, N Bray, P Melsted, and L Pachter. Pseudoalignment for metagenomic read assignment. *Bioinformatics*, 33(14):2082–2088, February 2017.

40 Wei Shen, Hongyan Xiang, Tianquan Huang, Hui Tang, Mingli Peng, Dachuan Cai, Peng Hu, and Hong Ren. KMCP: accurate metagenomic profiling of both prokaryotic and viral populations by pseudo-mapping. *Bioinformatics*, 39(1), December 2022. btac845.

41 Sebastiano Vigna. Broadword implementation of rank/select queries. In *International Workshop on Experimental and Efficient Algorithms*, pages 154–168, 2008.

42 Derrick E. Wood, Jennifer Lu, and Ben Langmead. Improved metagenomic analysis with Kraken 2. *Genome Biology*, 20(1):257, November 2019.

43 Derrick E Wood and Steven L Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology*, 15(3):1–12, 2014.

44 Ilya Y Zhbannikov, Samuel S Hunter, Matthew L Settles, and James A Foster. SlopMap: a software application tool for quick and flexible identification of similar sequences using exact k-mer matching. *Journal of data mining in genomics & proteomics*, 4(3), 2013.

45 Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys (CSUR)*, 38(2):6–es, 2006.

A Pseudocode for the algorithms from Section 4.1

Algorithm 1 The FULL-INTERSECTION algorithm for a query sequence Q. The algorithm uses the three index components: $\mathcal{D}$ (the dictionary, mapping k-mers to unitigs), B (the bit-vector mapping from unitigs to colors), and $\mathcal{L}$ (the inverted index storing the compressed colors). As discussed in Section 3.1, the dictionary $\mathcal{D}$ can stream through the query sequence Q and collect unitig ids. The inverted index $\mathcal{L}$, instead, returns an iterator over a color set given the color id c as ITERATOR(c).

```
1:  function FULL-INTERSECTION(Q)
2:      if |Q| < k then return ∅
3:      U = D.STREAM-THROUGH(Q)                ▷ U is the set of unitig ids.
4:      DEDUPLICATE(U)
5:      C = ∅                                  ▷ C is the set of color ids.
6:      for u ∈ U do
7:          c = B.COLOR-ID(u)
8:          C.ADD(c)
9:      DEDUPLICATE(C)
10:     I = ∅                                  ▷ I is the set of iterators over colors.
11:     for c ∈ C do
12:         i = L.ITERATOR(c)
13:         I.ADD(i)
14:     R = INTERSECT(I)                       ▷ R is the result set of reference ids.
15:     return R
```

Algorithm 2 The INTERSECT algorithm for a set of iterators $I = \{i_1, \ldots, i_p\}$. An iterator object supports three primitive operations: VALUE(), returning the value currently pointed to by the iterator; NEXT(), returning the value immediately after the one currently pointed to by the iterator; NEXT-GEQ(x), returning the smallest value that is larger-than or equal-to x. We assume that if i is an iterator over color C_j then calling i.NEXT() for more than $|C_j|$ times will return the (invalid) reference id $N + 1$.

```
1:  function INTERSECT(I)
2:      if I = ∅ then return ∅
3:      R = ∅
4:      candidate = i_1.VALUE()
5:      j = 2
6:      while candidate ≤ N do
7:          for ; j ≤ p; j = j + 1 do
8:              i_j.NEXTGEQ(candidate)
9:              v = i_j.VALUE()
10:             if v ≠ candidate then
11:                 candidate = v
12:                 j = 1
13:                 break
14:         if j = p + 1 then
15:             R.ADD(candidate)
16:             i_1.NEXT()
17:             candidate = i_1.VALUE()
18:             j = 2
19:     return R
```

Algorithm 3 The THRESHOLD-UNION algorithm for a query sequence Q. Differently from the FULL-INTERSECTION method (Algorithm 1), here U, C, and I, are sets of pairs. The first component of a pair is a unitig id, a color id, or an iterator, respectively if the pair is in U, C, or U. The second component, read by calling the method SCORE() in the pseudocode, is the number of positive k-mers that have a given unitig id or have a given color. The score of iterator i is the score of the color id c if $i = \mathcal{L}$.ITERATOR(c). Clearly, when deduplicating the sets U and C, the scores of equal unitig or color ids must be summed.

```
1:  function THRESHOLD-UNION(Q, τ)
2:      if |Q| < k then return ∅
3:      U = D.STREAM-THROUGH(Q)                          ▷ U is the set of unitig ids.
4:      |K(Q)| = Σ_{u∈U} u.SCORE()                       ▷ |K(Q)| is the number of positive hits.
5:      DEDUPLICATE-AND-SUM-SCORES(U)
6:      C = ∅                                            ▷ C is the set of color class ids.
7:      for u ∈ U do
8:          c = B.COLOR-ID(u)
9:          C.ADD(c)
10:     DEDUPLICATE-AND-SUM-SCORES(C)
11:     I = ∅                                            ▷ I is the set of iterators over color sets.
12:     for c ∈ C do
13:         i = L.ITERATOR(c)
14:         I.ADD(i)
15:     t = |K(Q)| × τ                                   ▷ A reference is returned iff it contains at least t k-mers.
16:     R = UNION(I, t)                                  ▷ R is the result set of reference ids.
17:     return R
```

Algorithm 4 The UNION algorithm for a set of iterators $I = \{i_1, \ldots, i_p\}$ and minimum score t.

```
1:  function UNION(I, t)
2:      if I = ∅ then return ∅
3:      R = ∅
4:      candidate = min{i_1.VALUE(), ..., i_p.VALUE()}
5:      while candidate ≤ N do
6:          min = N + 1
7:          score = 0
8:          for j = 1; j ≤ p; j = j + 1 do
9:              if i_j.VALUE() = candidate then
10:                 score = score + i_j.SCORE()
11:                 i_j.NEXT()
12:             if i_j.VALUE() < min then min = i_j.VALUE()
13:         if score ≥ t then R.ADD(candidate)
14:         candidate = min
15:     return R
```

SparseRNAFolD: Sparse RNA Pseudoknot-Free Folding Including Dangles

Mateo Gray
Department of Biomedical Engineering, University of Alberta, Canada

Sebastian Will
CNRS/LIX (UMR 7161), Institut Polytechnique de Paris, France

Hosna Jabbari
Department of Biomedical Engineering, University of Alberta, Canada

Abstract

Motivation. Computational RNA secondary structure prediction by free energy minimization is indispensable for analyzing structural RNAs and their interactions. These methods find the structure with the minimum free energy (MFE) among exponentially many possible structures and have a restrictive time and space complexity ($O(n^3)$ time and $O(n^2)$ space for pseudoknot-free structures) for longer RNA sequences. Furthermore, accurate free energy calculations, including dangles contributions can be difficult and costly to implement, particularly when optimizing for time and space requirements.

Results. Here we introduce a fast and efficient sparsified MFE pseudoknot-free structure prediction algorithm, SparseRNAFolD, that utilizes an accurate energy model that accounts for dangle contributions. While the sparsification technique was previously employed to improve the time and space complexity of a pseudoknot-free structure prediction method with a realistic energy model, SparseMFEFold, it was not extended to include dangle contributions due to the complexity of computation. This may come at the cost of prediction accuracy. In this work, we compare three different sparsified implementations for dangles contributions and provide pros and cons of each method. As well, we compare our algorithm to LinearFold, a linear time and space algorithm, where we find that in practice, SparseRNAFolD has lower memory consumption across all lengths of sequence and a faster time for lengths up to 1000 bases.

Conclusion. Our SparseRNAFolD algorithm is an MFE-based algorithm that guarantees optimality of result and employs the most general energy model, including dangle contributions. We provide a basis for applying dangles to sparsified recursion in a pseudoknot-free model that has the ability to be extended to pseudoknots.

2012 ACM Subject Classification Applied computing → Molecular structural biology

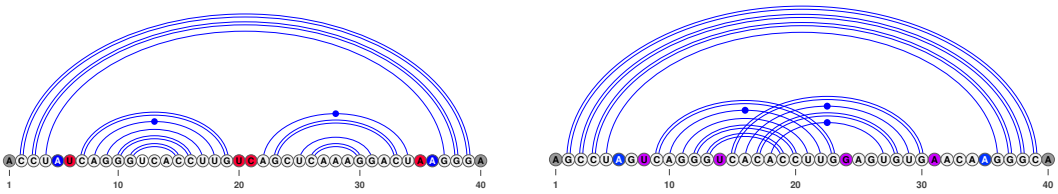
Keywords and phrases RNA, MFE, Secondary Structure Prediction, Dangle, Sparsification, Space Complexity, Time Complexity

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.19

Related Version *Previous Version*: `https://www.biorxiv.org/content/10.1101/2023.06.05.543808v1`

Supplementary Material *Software (SparseRNAFolD's Algorithm and Detailed Results)*:
`https://github.com/mateog4712/SparseRNAFolD`
archived at `swh:1:dir:0eca16668f1ba547f3f24ec55cb10e053df4492e`

Author Contributions Statement M.G. and H.J. conceived the experiment(s), M.G. conducted the experiment(s), M.G. analysed the results. M.G. and H.J. and S.W. wrote and reviewed the manuscript.



(a) Pseudoknot-free. **(b)** Pseudoknotted.

Figure 1 An RNA structure is shown with dangles highlighted. **(a)** In red, we have the dangles on the bands in the multi-loop. In blue, we have the dangle on the closing bases of the multi-loop. In gray, we have dangles on the outer end of the RNA. **(b)** We include purple to show dangles occurring in a pseudoknot. Dangles in pseudoknots can be handled differently depending on the program.

1 Introduction

Non-coding RNAs play crucial roles in the cell, such as in transcription [3], translation [3, 16], splicing [23, 32], catalysis [3, 36] and regulating gene expression [3, 12, 18, 23]. Since RNA's function heavily relies on its molecular structure, facilitated by hydrogen bonding both within and between molecules, predicting and comprehending the structure of RNA is a dynamic area of research. It is reasonable to assume (without further knowledge) that RNA forms the structure with the lowest free energy [19, 24]. This is the motivation for algorithms that aim to predict the RNA minimum free energy (MFE) structure from the pool of exponentially many structures it can form. Such methods employ a set of energy parameters for various loop types, called an energy model; to find the free energy of a structure, they add up the energy of its loops. While prediction accuracy of these methods depends on the quality of their energy models, these methods are applicable to novel RNAs with unknown families or functions and for the prediction of the structure of interacting molecules. The large time and space complexity of MFE-based methods ($O(n^3)$ time and $O(n^2)$ space where n is the length of the RNA), however, restricted their applications to small RNAs. The sparsification technique was recently utilized in existing MFE-based algorithms to reduce their time and/or space complexity [34, 29, 22, 2, 4, 35, 15, 14] by removing redundant cases in the complexity-limiting steps of the dynamic programming algorithms. While the majority of these methods focused on simple energy models, some expanded sparsification techniques to more realistic energy models [35, 15, 14]. To the best of our knowledge, no existing method has yet incorporated dangles energy contributions into a sparsified prediction algorithm. Dangle energies refer to the free energy contributions of unpaired nucleotides that occur at the end of a stem-loop structure.

We show in Figure 1 the location of dangles on a pseudoknot-free structure (see Figure 1a) and a pseudoknotted structure (see Figure 1b). The complexity of dangles in a pseudoknot further increases as dangles have to be tracked for both bands within the pseudoknot.

Neglecting dangle energies in the prediction of RNA structure stability can lead to inaccuracies. For instance, a stem-loop structure that includes an unpaired nucleotide at the end may appear less stable than its actual stability if the dangle energy contribution is ignored. Conversely, a stem-loop structure with an unpaired nucleotide that interacts positively with another one may appear more stable than its actual stability if the dangle energy contribution is not taken into account.

Dangles, in some form, are implemented in the majority of MFE pseudoknot-free secondary structure prediction algorithms [9, 8]. RNAFold [9, 11, 21, 6, 17, 41, 7] is an $O(n^3)$ time and $O(n^2)$ space algorithm which implements the dangle 0 ("no dangle"), dangle 2 ("always

dangle"), and dangle 1 ("exclusive dangle") model (defined in Section 2.1). It also utilizes a dangle model that implements coaxial stacking – a type of stacking that gives a bonus to stacks in the vicinity of each other. LinearFold [8], a sparsified $O(n)$ space heuristic algorithm has implemented the "no dangle" and "always dangle" model but has not implemented an "exclusive dangle" model. Fold from the RNAstructure library [27] is an $O(n^3)$ time and $O(n^2)$ space algorithm which implements an "exclusive dangle" model with coaxial stacking. MFold [40, 33, 39] is an $O(n^3)$ time and $O(n^2)$ space algorithm which has implemented an "exclusive dangle" model with coaxial stacking.

Handling dangles in pseudoknot prediction algorithms are less developed. Pknots [28], an $O(n^6)$ time and $O(n^4)$ space pseudoknot prediction algorithm has implemented an "exclusive dangle" model that also includes coaxial stacking. Within Pknots, a set of parameters is defined for non-pseudoknot and pseudoknot dangles. The pseudoknot parameters are estimated and rely on an estimated weighting parameter. Hotknots [26], a heuristic algorithm, uses the DP09 parameters, which include pseudoknotted parameters from Dirks and Pierce [5] and tuned by Andronescu et al. [26]; however, the energies for the pseudoknotted dangles are the same as those for pseudoknot-free dangles, and there is no weighting parameter.

Contributions. In [35], we already discussed the sparsification of RNA secondary structure prediction by minimizing the energy in the Turner energy model. However, in this former work, we did not yet consider the energy contributions due to the interactions of base pairs at helix ends with dangling bases (i.e., "dangling ends"). Here, we identify the correct handling of dangling end energies in the context of sparsification as a non-trivial problem, characterize the issues, and present solutions.

For this purpose, we first state precisely how dangle energies are handled by energy minimization algorithms; to the best of our knowledge, this is elaborated here for the first time. Consequently, we devise novel MFE prediction algorithms that include dangling energy contributions *and* use sparsification techniques to significantly improve the time and space complexity of MFE prediction.

Like the algorithm in [35], our efficient SparseRNAFolD algorithm keeps the additional information to a minimum using garbage collection. In total, we study three different possible implementations and compare their properties, which make them suitable for different application scenarios. Finally, while we study the case of non-crossing structure prediction, we discuss extensions to the more complex cases of pseudoknot and RNA–RNA interaction prediction (such extensions being the main motivation for this work in the first place).

2 Preliminaries: sparsification without dangling ends

We restate the preliminaries and main results from our former work on sparsification of free energy minimization without dangling ends [35].

We represent an ***RNA sequence*** of length n as a sequence $S = S_1, \ldots, S_n$ over the alphabet $\{A, C, G, U\}$; $S_{i,j}$ denotes the ***subsequence*** $S_i, \ldots, S_j$. A ***base pair*** of S is an ordered pair $i.j$ with $1 \leq i < j \leq n$, such that ith and jth bases of S are complementary (i.e. $\{S_i, S_j\}$ is one of $\{A, U\}, \{C, G\}$, or $\{G, U\}$). A ***secondary structure*** R for S is a set of base pairs with at most one pair per base (i.e. for all $i.j$, $i'.j' \in R$: $\{i, j\} \cap \{i', j'\} = \emptyset$). Base pairs of secondary structure R partition the unpaired bases of sequence S into ***loops*** [25] (i.e., hairpin loop, interior loop and multiloop). Hairpin loops have a minimum length of m; consequently, $j - i > m$ for all base pairs $i.j$ of R. Two base pairs $i.j$ and $i'.j'$ cross each other iff $i < i' < j < j'$ or $i' < i < j' < j$. A secondary structure R is ***pseudoknot-free*** if it does not contain ***crossing base pairs***.

The unsparsified, original algorithm for energy minimization over pseudoknot-free secondary structures was stated by Zuker and Stiegler [41]. It is a dynamic programming algorithm that, given an RNA sequence S of length n, recursively calculates the minimum free energies (MFEs) for subsequences $S_{i,j}$ as $W(i,j)$ (stored in a dynamic programming matrix). Finally, $W(1,n)$ is the optimal free energy. We state this algorithm in a sparsification-friendly form following [35]. As usual, the algorithm is described by a set of recursion equations (for a minimum hairpin loop size of m and a maximum interior loop size of M). For $1 \leq i < j \leq n$, $i < j - m$:

$$W(i,j) = \min\{\, W^p(i,j), V(i,j) \,\} \tag{1}$$

$$W^p(i,j) = \min\{\, W(i,j-1), \min_{i<k<j} W(i,k-1) + W(k,j) \,\} \tag{2}$$

$$V(i,j) = \min\{\mathcal{H}(i,j); \min_{\substack{i<p<q<j \\ p-i+j-q-2\leq M}} \mathcal{I}(i,j;p,q) + V(p,q); WM^2(i+1,j-1) + a\} \tag{3}$$

$$WM(i,j) = \min\{\, WM^p(i,j), V(i,j) + b \,\} \tag{4}$$

$$WM^p(i,j) = \min\{\, WM(i+1,j) + c, WM(i,j-1) + c, WM^2(i,j) \,\} \tag{5}$$

$$WM^2(i,j) = \min_{i<k<j} WM(i,k-1) + WM(k,j). \tag{6}$$

Here, a, b, c are multi-loop initialization penalty, branch penalty, and unpaired penalty in a multi-loop, respectively. $\mathcal{I}(i,j;p,q)$ refers to an interior loop between base pairs i.j and p.q. The initialization cases are $W(i,i) = 0$; $V(i,j) = WM(i,j) = \infty$ for all $j - i \leq m$ and $WM^2 = \infty$ for all $j - i \leq 2m + 3$.

In these recursions, all function values (like $W(i,j)$ or $W^p(i,j)$) denote minimum free energies over certain classes of structures of subsequences $S_{i,j}$. The classical Zuker/Stiegler matrices W, V and WM are defined as: W yields the MFEs over general structures; V, over closed structures, which contain the base pair $i.j$; WM, over structures that are part of a multi-loop and contain at least one base pair.

Since sparsification is based on the idea that certain optimal structures can be decomposed into two optimal parts, while others (namely closed structures) are non-decomposable, we single out the partitioning cases and introduce additional function symbols W^p, WM^p, and WM^2.

Sparsification without dangling ends. This allows us to cleanly explain the *key idea of sparsification* and consequently formalize it: to minimize over the energies of general structures in $W(i,j)$ – note that there is another minimization inside of multi-loops that is handled analogously – the algorithm considers all closed structures $V(i,j)$ and all others $W^p(i,j)$. Optimal structures in the latter class can be decomposed into two optimal structures of some prefix $S_{i,k-1}$ and suffix $S_{k,j}$ of the subsequence. Classically, the minimum is therefore obtained by minimizing over all ways to split the subsequence. Sparsification saves time and space since it is sufficient to consider only the splits where the optimum of the suffix $S_{k,j}$ is not further decomposable (formally, where $W(k,j) < W^p(k,j)$). Briefly (for more detail, see [35] or [34]), this is sufficient since otherwise there is a k' to optimally split the suffix further into $S_{k,k'-1}$ and $S_{k',j}$. The split of $S_{i,j}$ at k cannot be better than the split at k' and therefore does not have to be considered in the minimization; thus, it can be restricted to a set of *candidates*. This is argued by the ***triangle inequality for*** W (which directly follows from the definition of W as minimum):

$$W(i,j) \leq W(i,k-1) + W(k,j) \qquad \text{for all } 1 \leq i < k \leq j \leq n.$$

Consequently, sparsification improves the computation of W^p, WM^p and WM^2. The corresponding sparsified version are

$$\widehat{W}^{\mathrm{p}}(i,j) = \min\{\, W(i,j-1); \min_{[k,j] \text{ is candidate}, k>i} W(i,k-1) + V(k,j)\,\}$$

$$\widehat{WM}^{\mathrm{p}}(i,j) = \min\{\, WM(i,j-1) + c; \min_{[k,j] \text{ is candidate}, k>i} c\cdot(k-i) + V(k,j)\,; WM^2(i,j)\}$$

$$\widehat{WM}^{2}(i,j) = \min\{\, WM^2(i,j-1) + c; \min_{[k,j] \text{ is candidate}, k>i} WM(i,k-1) + V(k,j)\,\}$$

where candidates $[k,j]$ correspond to the not optimally decomposable subsequences $S_{k,j}$ (in either situation: general structures or structures inside of multi-loops), i.e. $[i,j]$ is a ***candidate*** iff $V(i,j) < \widehat{W}^{\mathrm{p}}(i,j)$ or $V(i,j) + b < \widehat{WM}^{\mathrm{p}}(i,j)$.

Time and space complexity of sparsified energy minimization. Will and Jabbari showed that following the above algorithm, $W(1,n)$ can be calculated in $O(n^2 + nZ)$ time, where Z is the total number of *candidates*. While the MFE structure in the Zuker and Stiegler algorithm can be trivially reconstructed following a traceback procedure, this is not the case if sparsification is used for improving time *and* space as in the SparseMFEFold algorithm (and our novel algorithms). To improve the space complexity, sparsification avoids storing all entries of the energy matrix. The idea is to store the candidates and as few additional matrix entries as possible. A specific challenge is posed by the decomposition of interior loops (the single most significant major complication over base pair maximization, see [2]). For this reason, Will and Jabbari introduced *trace arrows* for cases, where the trace cannot be recomputed efficiently during the traceback procedure; they discussed several space optimization techniques, such as avoiding trace arrows by rewriting the MFE recursions, and removing trace arrows as soon as they become obsolete. Due to such techniques, SparseMFEFold requires only linear space in addition to the space for candidates and trace arrows; its space complexity is best described as $O(n + T + Z)$, where T is the maximum number of trace arrows.

2.1 Dangles

Recall that sparsification was discussed before (e.g., in SparseMFEFold) only for the simplest and least accurate variant of the Turner model, namely the one without dangling end contributions. Before we improve this situation, let's look in more detail at dangling ends and different common ways to handle them. Specifically, we discuss different *dangle models* "no dangle" (model 0), "exclusive dangle" (model 1), and "always dangle" (model 2) as implemented by RNAfold of the Vienna RNA package (and available via respective command line options `-d0`, `-d1`, and `-d2`).

Dangling end contributions occur only at the ends of stems (either in multiloops or externally) due to stacking interaction between the closing base pair of the stem and one or both immediately adjacent unpaired bases. In contrast, dangling end terms are not considered within (interior loops of) stems by the energy model.

We present modified DP recursions in order to reflect precisely where and how dangling ends are taken into account. Therefore, in preparation, let's replace V in the Equations (1) and (4) of the free energy minimization recursions of Section 2 by a new function V^{d}. The dangle models differ in the exact definition of V^{d}.

$$W(i,j) = \min\{\, W^p(i,j), V^{\mathrm{d}}(i,j)\,\} \tag{1'}$$

$$WM(i,j) = \min\{\, WM^p(i,j), V^{\mathrm{d}}(i,j) + b\,\} \tag{4'}$$

Note that in the energy model, dangling ends can also occur at the inner ends of helices that close a multi-loop. These dangles can be handled directly in the recurrence of $V(i,j)$; specifically, in the subcase where $i.j$ closes a multi-loop.

No dangles. In the simplest model "no dangle", dangling ends are ignored. We achieve this by defining

$$V^{\mathrm{d}}(i,j) := V(i,j) \qquad \text{(no dangle)}$$

While easy to implement, it is clearly wrong to ignore dangling end contributions, and this has a significant negative effect on the prediction accuracy compared to the other dangle models [30, 38, 37].

Always dangle. A second relatively simple way is to apply a 53' dangle energy at both ends of a stem (both 5' and 3' ends), assuming that stem ends always dangle with their adjacent bases. As a strong simplification, in this model, one disregards whether the bases are paired and/or dangle with a different stem (either case would actually make them unavailable for dangling).

This dangle model allows the dangling ends to have a thermodynamic influence while keeping the model easy to implement as neither the conflicting adjacent nucleotides nor the energies of single dangle have to be tracked; it only requires knowledge of the bases on the 3' and 5' sides of a base pair. Formally, we implement V^{d} as

$$V^{\mathrm{d}}(i,j) := V(i,j) + \mathrm{dangle}_{53}(i,j) \qquad \text{(always dangle)}$$

Moreover, we add the appropriate dangle contribution when closing a multi-loop in Eq. (3) in the last case of the V-recurrence of Eq. (3). The term $WM^2(i+1,j-1)+a$ is rewritten to

$$WM^2(i+1,j-1) + a + \mathrm{dangle}_{53}(i+1,j-1) \qquad \text{(always dangle, ML closing)}$$

Exclusive dangling. The most complex but general secondary structure dangle model, "exclusive dangle" considers both single and double unpaired nucleotides adjacent to a stem. Furthermore, the model does not allow shared dangling ends i.e. no base can be used simultaneously in two dangles (in other words, adjacent unpaired bases dangle *exclusively* with a single stem end). As the restriction requires tracking of unpaired bases, $V(i,j)$ places the possible unpaired bases at i and j and looks at the adjacent V energies. As this requires knowledge of energies adjacent to the current bases being looked at, this inherently causes difficulty in sparsification.

$$V^{\mathrm{d}}(i,j) := \min \begin{cases} V(i,j) \\ V(i+1,j) + \mathrm{dangle}_5(i) \\ V(i,j-1) + \mathrm{dangle}_3(j) \\ V(i+1,j-1) + \mathrm{dangle}_{53}(i,j) \end{cases} \qquad \text{(exclusive dangle)}$$

Moreover, we consider dangles at the closing of a multi-loop. In this model, the case $WM^2(i+1,j-1)+a$ in the minimization of Eq. (3) is replaced by (the minimum of) four different cases:

$$\min \begin{cases} WM^2(i+1,j-1) + a \\ WM^2(i+2,j-1) + a + \mathrm{dangle}_3(i) \\ WM^2(i+1,j-2) + a + \mathrm{dangle}_5(j) \\ WM^2(i+2,j-2) + a + \mathrm{dangle}_{53}(i,j) \end{cases} \qquad \text{(exclusive dangle, ML closing)}$$

2.2 Space-efficient sparsification with exclusive dangles is non-trivial

We approach our main motivation for this work, which is to study and solve the issues of sparsification in the exclusive dangle model (dangle model 1). Let's thus start by applying the idea of sparsification (Section 2) straightforwardly to the Recursion (2) (where W and V^{d} are defined for exclusive dangles).
We quickly come up with the equation:

$$\widehat{W}^{\mathrm{p}}(i,j) = \min\{\, W(i,j-1); \min_{[k,j] \text{ is ed-candidate}, k>i} W(i,k-1) + V^{\mathrm{d}}(k,j)\,\},$$

but we would still have to define *ed-candidate* (exclusive dangle candidate) to make this work. We could define: [i,j] is an ***ed-candidate*** iff $V^{\mathrm{d}}(i,j) < \widehat{W}^{\mathrm{p}}(i,j)$, where the correctness of sparsification holds to a sparsification-typical triangle inequality argument (Section 2).

Expanding V^{d} shows that this is not the only possible path to sparsifying the recursion. We could consider

$$\widehat{W}^{\mathrm{p}}(i,j) = \min \begin{cases} W(i,j-1) \\ \min_{[k,j] \text{ is ed0-candidate}, k>i} W(i,k-1) + V(i,j) \\ \min_{[k,j] \text{ is ed5-candidate}, k>i} W(i,k-1) + V(i+1,j) + \mathrm{dangle}_5(i) \\ \min_{[k,j] \text{ is ed3-candidate}, k>i} W(i,k-1) + V(i,j-1) + \mathrm{dangle}_3(j) \\ \min_{[k,j] \text{ is ed53-candidate}, k>i} W(i,k-1) + V(i+1,j-1) + \mathrm{dangle}_{53}(i,j) \end{cases}$$

with different sets of candidates for all four cases. However, storing all these candidate sets (recall that there is even a second recursion that needs to be sparsified) is easily prone to compromising any space benefits due to sparsification in practice.

The transfer of the techniques from [35] brings even more problems, since due to such definitions, candidates [i,j] do not necessarily correspond to subsequences that have closed optimal structures. Will and Jabbari strongly exploited this fact for their strong space savings.

Even considering our definition of an ed-candidate, we still run into the challenge of to tracing back to the corresponding base pair. With just the dangle energy, this poses issues as an ed-candidate can be one of four cases.

▶ **Lemma 1.** *In the exclusive dangle model, storing only the energy of each ed-candidate is not sufficient to correctly trace back from the candidate.*

Proof. Concretely, for the loop-based Turner 2004 energy model [20]) with exclusive dangles, consider the following RNA sequence S of length 12 with its MFE structure:

```
UGGGAAAACCCC
.(((....))).
```

In the calculation of $W(1,12)$, the recurrences unfold to $W(1,12) = W(1,1) + V^{\mathrm{d}}(2,12) = W(1,1) + V(2,11) + \mathrm{dangle}_3(12) = \cdots = -2.9$ kcal/mol, i.e. it is optimal to assume dangling of base pair $(2,11)$ to the right.

In a non time- and space-sparsified algorithm, recomputing V^{d} from V adjacent energies would be trivial. However, due to space sparsification, the values of V are generally unavailable in the trace-back phase. In the constructed example, recomputation would require us to know $V(2,12)$, $V(2,11)$, $V(3,12)$, and $V(3,11)$. Thus, under the assumption of the lemma, the optimal dangling cannot be efficiently recomputed for a candidate like [2,12]. ◀

In our preceding work SparseMFEFold [35], trace arrows were introduced to trace back to non-candidate values necessary to the structure within the interior loop case: $V^{\text{il-cand}}(i,j)$. Trace arrows that point to candidates are not stored as they can be avoided by minimizing over candidates as seen in Equation 7.

$$V^{\text{il-cand}}(i,j) = \min_{\substack{i<p<q<j \\ p-i+j-q-2\leq M \\ [p,q] \text{ is candidate}}} \mathcal{I}(i,j;p,q) + V(p,q). \tag{7}$$

Consequently, finding the inner base pair of a loop through a candidate relies on the energy saved being $V(p,q)$. However, as shown in Eq. (exclusive dangle, ML closing), the dangle energy could be $V(p,q)$, $V(p+1,q)$, $V(p,q-1)$, or $V(p+1,q-1)$. Replacing the stored energy within a candidate with V^{d} may conflict with the interior loop calculation. Recomputation of the V values required for V_d would negate the sparsification benefit. In summary, there is no easy or direct way to save the V energy required for the interior loop as well as the V^{d} energy required for a multi-loop or external loop within the current candidate structure.

▶ **Lemma 2.** *The minimization over inner base pairs in the recursion of V cannot be restricted to candidates in the same way as in SparseMFEFold.*

Proof. Again consider the loop-based Turner 2004 energy model. There is a sequence S and $1 \leq i < j \leq n$, such that $V^{\text{d}}(p,q) < V(p,q)$, but there is no way to trace back to p and q from i and j, namely, consider the RNA sequence S of length 19 with its MFE structure:

```
GGGAGGGAAAACCCCACCC
(((.(((.....))).)))
```

The optimal recursion case of $V(3,17)$ forms the interior loop closed by 3.17 with inner base pair 5.15, because $V(5,15) = -2.4$ kcal/mol and V(3,17) $= \mathcal{I}(3,17;5,15) + V(5,15) = -1.5$ kcal/mol.

The space optimization of SparseMFEFold removes trace arrows to candidates since the trace-back to candidates can be reconstructed based on candidate energies (compare Eq. (7).

In the way of SparseMFEFold, we would not store a trace arrow pointing to 5.15 from $[3,17]$, since $[5,15]$ is a candidate. However, without a trace arrow, we would not reconstruct the correct trace. This happens, since the optimal structure in the subsequence 5..15 `GGGAAAACCCC` would be `(((....)))`. due to the 3' dangle ($V^{\text{d}}(5,15) = -2.9$ kcal/mol). Consequently, tracing back the optimal path from $V^{\text{d}}(5,15)$ wrongly introduces a base pair at 5.14. ◀

3 SparseRNAFolD

SparseRNAFolD combines the power of sparsification and a general energy model including dangle energies to achieve a fast and highly accurate RNA pseudoknot-free secondary structure prediction. To this end, we started with the sparsified dynamic programming recurrences of SparseMFEFold (which implements the "no dangles" model), rewriting and revising them to accommodate various dangle energies.

3.1 "always dangle" model

Recall that "always dangle" model considers both the 5' and 3' ends of a branch of a multi-loop or external loop for dangle contributions. The addition of this model is trivial, with no change necessary to the recurrences of the SparseMFEFold. Note that, as mentioned earlier, this model ignores overlapping cases and may overcount the contributions of dangles.

3.2 "exclusive dangle" model

As mentioned in Section 2.2, accounting for the "exclusive dangle" model is non-trivial when dealing with candidates, as ed-candidates do not hold enough information to identify the direction of dangles. To alleviate this problem, we provide three different strategies, as described below. Each strategy has its pros and cons and should be selected based on the application.

In order to handle the changes for exclusive dangles, we extend the candidate data structure. A candidate base pair, $[i, j]$ as implemented in SparseMFEFold, holds i, the start position, and the energy $V(i, j)$ as a tuple $(i, V(i, j))$ and is stored at the jth index of the candidate list. Our extensions to candidate structures involves including the energy values for W and WM in the candidate tuples as $(i, V(i, j), W(i, j), WM(i, j))$. The modification reflects the need to store more information about the dangles positions and directions.

Strategy 1: Trace Arrow implementation. As the first strategy to trace an ed-candidate to its position, we used modified trace arrows. We refer to this strategy as **SparseRNAFolD-Trace**.

Recall that in SparseMFEFold, a trace arrow structures were introduced to identify energy matrix entries that are necessary for calculating the energy of internal loops but are not kept as candidates. Here, we define ***ed-trace-arrows*** to hold information about dangle positions to aid with the traceback procedure from ed-candidates. In particular, in the sparse fold reconstruction procedure of SparseRNAFolD, an ed-trace-arrow is checked for a chosen ed-candidate within W, WM, and $WM2$ to adjust the energy and position of the base pair as required. The drawback of this strategy comes from the innate inefficiencies of the trace arrows, meaning an increase in space usage. Recall that within SparseMFEFold, we used strategies such as garbage collection and trace arrow avoidance to save space. These strategies are not, however, possible for SparseRNAFolD-Trace, as an ed-candidate cannot be excluded from the optimal MFE path, and an ed-trace-arrow is therefore required for every ed-candidate.

Bit encoding. Within the second and third strategies, as explained next, we employed bit encoding and bit decoding to store the information about the dangle within the energy values to reduce space usage. Currently, energy values are stored as 32-bits *int* data type. We note that the maximum expected bit usage for the energy value of an RNA sequence of up to 20000 bases is about 13 bits. We employed a bit shift to store the dangle type in the first two bits of the V entries, referred to as V_{enc}, and represented in eq. 8.

$$V_{enc} = (V \ll 2) \mid dangle \tag{8}$$

Bit decoding technique was used to retrieve the energy value and type/direction of dangle contributions. Bit decoding was done in two steps. Shifting the encoded energy, V_{enc}, two bits forward gave back the energy, V (see eq. 9).

$$V = V_{enc} \gg 2 \tag{9}$$

The dangle type is found in the first two bits; no dangle is represented with a "00" in bits; a 5' dangle with a "01"; a 3' dangle with a "10"; and a 53' dangle with a "11". The dangle type is decoded using a bit-wise AND with "11" to only keep the first two bits of the encoded energy, as represented in Eq. 10.

$$dangle = V_{enc} \ \&\& \ 11 \tag{10}$$

Strategy 2: Bit encoding with candidate extension. As the second strategy, we used bit encoding within the W and WM entries of the ed-candidate data structure. We refer to this strategy as **SparseRNAFolD-standard**. This implementation of bit encoding was utilized in W and WM entries, as other loop types do not deal with dangles.

Strategy 3: Bit encoding with altered candidate. As the third strategy, we further optimize for space by reducing the candidate size. To reduce candidate size, we stored energy values in ed-candidates in W and WM as V^{d} minus the dangle energy. We refer to this strategy as **SparseRNAFolD-Triplet**. This strategy allows for the correct identification of dangle types regardless of energy parameters used. Note that currently, in the Turner 2004 energy model, the parameter values for 53' dangle for an external loop and multi-loop are the same. These values may be further estimated and revised in future energy models. The extra calculations to retrieve the V^{d} value ensure the accuracy of the result in the event of such a change.

3.3 Compared methods

To evaluate the performance of our SparseRNAFolD, we compared it to two of the best-performing methods for prediction of pseudoknot-free RNA secondary structure, namely RNAFold [9] and LinearFold [8].

3.3.1 RNAFold

RNAFold is part of the Vienna RNA package [9]. As discussed in Section 2.1, RNAfold is an $O(n^3)$ time and $O(n^2)$ space algorithm. It takes an RNA sequence as input and provides the MFE structure as output. RNAFold is well-maintained and highly optimized and is used here as a benchmark for a fast implementation of the Zuker and Steigler-type MFE algorithm.

3.3.2 LinearFold

LinearFold [8] is a pseudoknot-free RNA secondary structure prediction algorithm that uses heuristic techniques to run in linear time and space. As the main goal of sparsification is to speed up the time and space complexity of MFE prediction, we set out to investigate how our SparseRNAFolD compares in practice to LinearFold with better asymptotic complexities.

Linearfold employs two techniques to reduce its time and space complexity to $O(n)$, namely ***beam pruning*** and ***k-best parsing***. Both methods aim to prune the structure path to optimal cases only. Beam pruning works by only keeping a predetermined number (specified by the beam width, b) of the optimal states. Within LinearFold, best sets are kept for each possible loop type as defined in the Zuker algorithm: hairpin, multi-loop fragments, and internal loop. Through beam pruning, time complexity is reduced to $O(nb^2)$ and the space to $O(nb)$ where b is the beam width. K-best parsing further reduces the time to $O(nb\log(b))$. We note that due to the heuristic nature of the LinearFold algorithm, it does not guarantee finding the MFE structure for a given RNA sequence.

4 Experimental Design

We implemented SparseRNAFolD in C++. All experiments were performed using an Azure virtual machine. The virtual machine contained 8 vCPUs with 128 GiB of memory.

4.1 Dataset

We used the original dataset from SparseMFEFold [35]. This dataset is comprised of 3704 sequences in 6 different families selected from the RNAstrand V2.0 database [1]. The smallest sequence is 8 nucleotides long, while the largest is 4381 nucleotides long.

4.2 Energy Model

We used the energy parameters of the Turner 2004 energy model [20, 31], as implemented in the ViennaRNA package [9].

4.3 Accuracy Measures

The number of *true positives* (TP) is defined as the number of correctly predicted base pairings within the structure. The number of *false positives* (FP), similarly, is the number of predicted base pairs that do not exist in the reference structure. Any base missed in the prediction that corresponds to a pairing in the reference structure is a *false negative* (FN).

We evaluate the performance of algorithms based on three measures: sensitivity, positive predictive value (PPV), and their harmonic mean (F-measure).

$$Sensitivity = \frac{TP}{TP + FN} \tag{11}$$

$$PPV = \frac{TP}{TP + FP} \tag{12}$$

$$F_{measure} = \frac{2 \cdot PPV \cdot Sensitivity}{PPV + Sensitivity} \tag{13}$$

4.4 Proof of concept with RNAFold

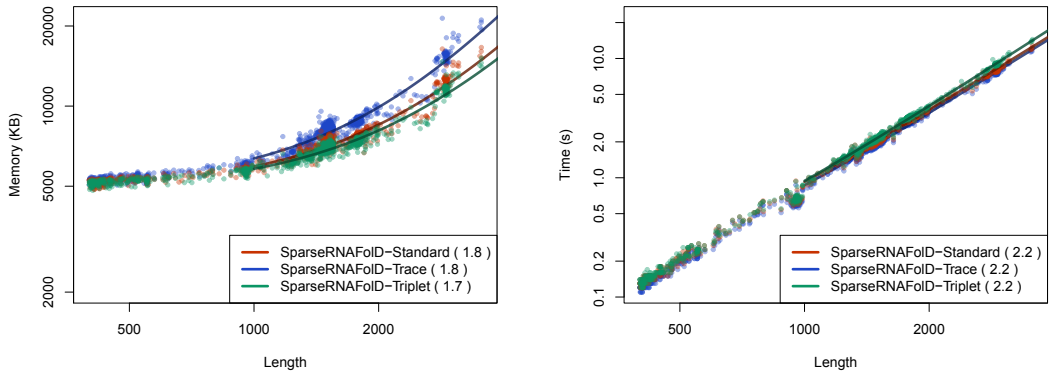
As a proof of concept for the correct implementation of dangle energy models (i.e., "always dangle" and "exclusive dangle"), we assessed SparseRNAFolD against RNAFold. As the MFE structure may not be unique, we restricted our assessment to the MFE value obtained by each method. We found that the MFE predicted by SparseRNAFolD and RNAFold was the same. Details of the results can be found in our repository.

5 Results

We measured runtime using user time and memory using the maximum resident set size.

5.1 Alternative Models

We start by comparing the three different implementations of SparseRNAFolD. SparseRNAFolD-standard was found to be in the middle in terms of memory and time. The effect of additional trace arrows in SparseRNAFolD-Trace had a 27% increase in memory usage on the largest sequence compared to SparseRNAFolD-Standard. However, the increase in computation from the bit encoding only resulted in a 5% increase in time on the largest sequence. We find a similar effect when comparing SparseRNAFolD-standard and SparseRNAFolD-Triplet. The altered triplet structure reduced the memory by 9% but increased the time by 10% due to extra computation. These are highlighted in Figure 2.



(a) Memory vs Length.

(b) Time vs Length.

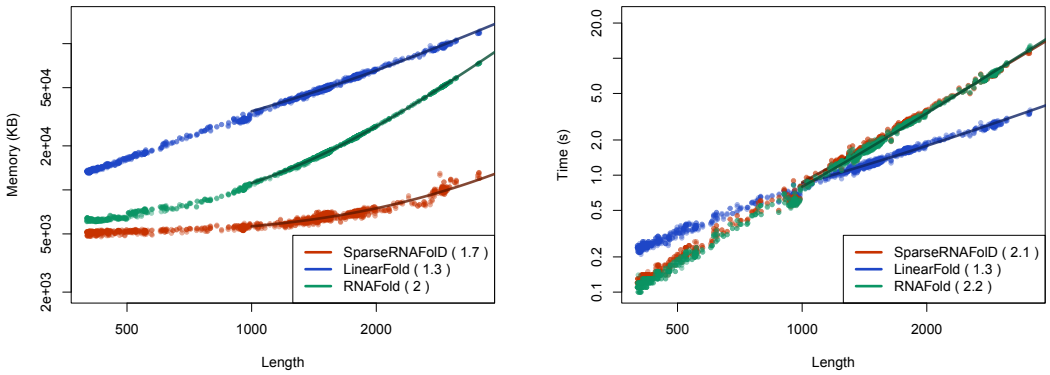
Figure 2 We plot the results of the three versions of SparseRNAFolD when given RNA sequence only as input against each other and an "exclusive dangle" model based on the dataset. (a) Memory Usage (maximum resident set size in KB) versus length (log-log plot) over all benchmark instances. The solid line shows an asymptotic fit ($c1 + c2n^x$) for sequence length n, constants c1, c2, and exponent x for the fit. We ignored all values < 1000. (b) Run-time (s) versus length (log-log plot) over all benchmark instances. For each tool in both plots, we report (in parenthesis) the exponent x that we estimated from the benchmark results; it describes the observed complexity as $\Theta(n^x)$.

5.2 Comparison with Linearfold and RNAFold

When comparing SparseRNAFolD-Standard with LinearFold and RNAFold, we look at the "always dangle" model, as LinearFold does not implement the "exclusive dangle" model.

We first compared the three algorithms by their predictive accuracy (F-measure). For comparison, we selected all sequences from our dataset whose structure was available on RNAstrand. We further constrained it to sequences that contained hairpins greater than 3 and no pseudoknots. This resulted in 986 sequences. We found that SparseRNAFolD-Standard had a marginally better, but not significant, average F-measure of 0.6394 compared to 0.6391 of LinearFold. As described in section 4.4, RNAFold and SparseRNAFold-standard are identitcal in predictive accuracy.

We then assessed their time and space usage. To increase the size of our dataset for this testing, we included a dinucleotide shifted version of our dataset in our test data. We then constrained the size of sequences to those > 400. The maximum time and memory used by Linearfold on this dataset were 3.34 seconds and $118,848$ KB. The maximum time and memory used by RNAFold were 22.26 seconds and 109136 KB. In contrast, the maximum time and memory spent by SparseMFEFold were 10.86 seconds and $13,000$ KB, respectively. This is illustrated in Figures 3b and 3a. The results show that SparseRNAFolD-Standard uses far less memory on even the largest pseudoknot-free sequences in our dataset. Note that the maximum resident set size is nine times lower than that of LinearFold and eight times lower than that of RNAFold. RNAFold's time remains consistent with SparseRNAFolD-Standard until longer sequences where it fell behind. LinearFold, whose time complexity is $O(nb\log(b))$, where n is the length of the sequence and b is the beam width, did perform faster than SparseRNAFolD-Standard as the length of the sequence increased. However, we did find that SparseRNAFolD-Standard outperformed LinearFold in practice for sequences of up to about 1000 nucleotides.



(a) Memory vs Length. **(b)** Time vs Length.

Figure 3 We plot the results of SparseRNAFoldD-Standard against two state of the art algorithms: RNAFold and LinearFold when given RNA sequence only as input against each other and an "always dangle" model on our dataset and the dinucleotide shuffled version of our dataset. (a) Memory Usage (maximum resident set size in KB) versus length (log-log plot) over all benchmark instances. The solid line shows an asymptotic fit ($c1 + c2n^x$) for sequence length n, constants c1,c2, and exponent x for the fit. We ignored all values < 1000. (b) Run-time (s) versus length (log-log plot) over all benchmark instances. For each tool in both plots, we report (in parenthesis) the exponent x that we estimated from the benchmark results; it describes the observed complexity as $\Theta(n^x)$.

Table 1 We tabulate the results of the comparison between RNAFold and SparseRNAFoldD-Standard when given only sequences with length > 2500 from our dataset as input and using the "exclusive dangle" model. We looked at time (s) and memory (maximum resident set size in KB) for the minimum, median and maximum length sequence within the constrained dataset.

	Run-time (s)		Memory: resident set size (KB)	
	RNAfold	SparseRNAFoldD	RNAfold	SparseRNAFoldD
Minimum	5.04	5.36	40148	8832
Median	7.28	7.86	51284	12592
Maximum	22.08	19.94	109040	16836

5.2.1 Highlighting RNAFold

To highlight the difference in space between RNAFold and SparseRNAFoldD-Standard, we selected 81 sequences from our dataset with size greater than or equal to 2500. The sequence with the maximum length in the set was 4381 nucleotides long. As seen in Table 1, while SparseRNAFoldD-Standard's runtime is comparable to RNAfold's, its memory consumption is about five times lower.

5.3 Candidate Comparison

In order to illustrate the effectiveness of candidates in terms of memory consumption, we plotted the relationship between the number of candidates and trace arrows, against the quadratic space, using the dataset that includes dinucleotide shifted elements. To emphasize the upper limit of candidate usage when executing SparseRNAFoldD-Standard, we employ the "exclusive dangle" model.

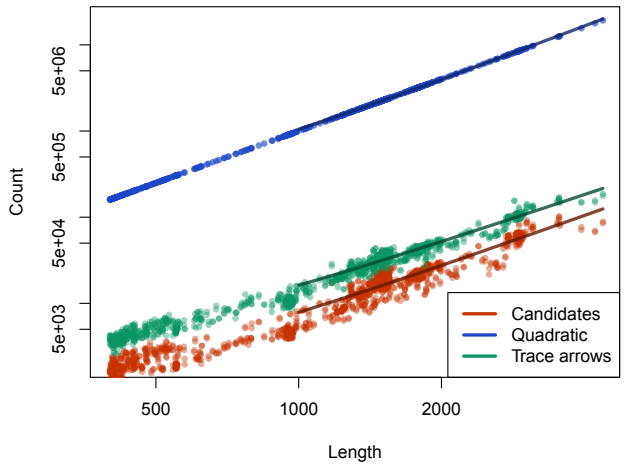


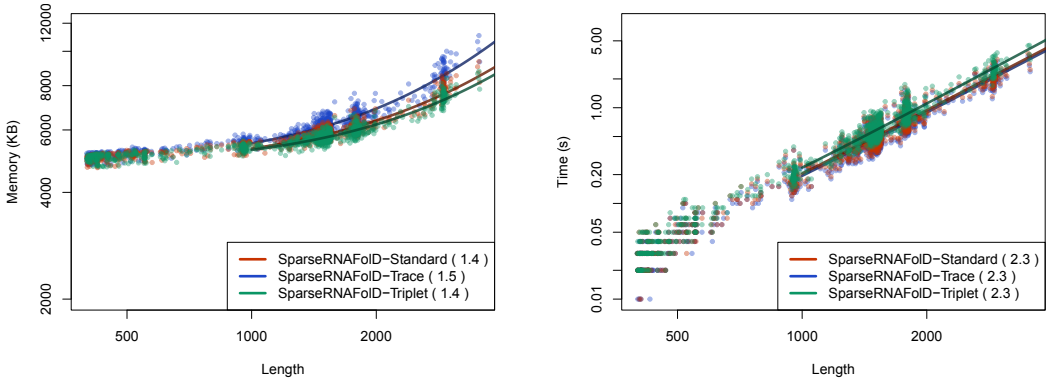
Figure 4 We plot the results of the number of candidates and trace arrows compared to quadratic space. "Quadratic" shows the count within an nxn matrix as it would be given quadratic space. In contrast, "Candidates" and "Trace arrows" show the contrasting number for the same length.

For a more meaningful comparison, we juxtapose the counts of candidates and trace arrows with the count obtained from a single quadratic matrix. It is important to note that the majority of algorithms employing quadratic space make use of multiple quadratic matrices. Considering this aspect, we discovered that, on average, the disparity in count between the number of candidates and trace arrows with quadratic space was approximately a factor of 100. Figure 4 highlights that the increase in candidates is consistent with the increase in length.

5.4 Folding with Hard Constraints

As partial information on structure has become more available and is extensively used for better prediction of possibly pseudoknotted structures [13, 10], we further extend our evaluation of the SparseRNAFolD versions to cases where we are folding with a hard constraint [17] in addition to the RNA sequence.

To do so, for each sequence, a pseudoknot-free input structure was generated. The structure was generated by taking two random indices at a time from the sequence. If the two bases could pair, were at least 3 bases apart, and did not form a pseudoknot with the other base pairs, the base pair was added to the input structure. In order to avoid overpopulating the input structure, the number of base pairs in an input structure was capped at $0.5 \times \log_2(length)$. This resulted in an average of 3-7 base pairs per sequence. There was a noticeable decrease in time and space when an input structure was provided in addition to an RNA sequence. Between RNA sequence only as input and sequence as well as an input structure, SparseRNAFolD saw a 67% decrease in time and a 40% decrease in memory. As the input structure reduced the number of candidates for a sequence, the difference in memory was less apparent between the models. SparseRNAFolD-standard had a 6% increase in time from SparseRNAFolD-Trace but a 15% decrease in memory on the largest sequence. From SparseRNAFolD-standard to SparseRNAFolD-Triplet, there was an 8% decrease in memory but a 13% increase in time. Note that even when reducing the number of candidates, the increase in time from Standard to Triplet was greater by 3%. This can be seen in Figure 5.



(a) Memory vs Length with a hard constraint.

(b) Time vs Length with a hard constraint.

Figure 5 We plot the results of the three versions of SparseRNAFolD when given an RNA sequence, an "exclusive dangle" model, and a random pseudoknot-free structure as input against each other based on our dataset. (a) Memory usage (maximum resident set size in KB) versus length (log-log plot) over all benchmark instances. The solid line shows an asymptotic fit ($c1 + c2n^x$) for sequence length n, constants c1,c2, and exponent x for the fit. We ignored all values < 1000. (b) Run-time (s) versus length (log-log plot) over all benchmark instances. For each tool in both plots, we report (in parenthesis) the exponent x that we estimated from the benchmark results; it describes the observed complexity as $\Theta(n^x)$.

6 Conclusions

In this work, we introduced SparseRNAFolD, a sparsified MFE RNA secondary prediction algorithm that incorporates dangles contribution to the energy calculation of a sparsified method. We showed that while "no dangle" and "always dangle" models were easy to incorporate into the existing algorithms, "exclusive dangle" introduces non-trivial challenges that need calculated changes to the sparsified recursions to alleviate. We identified three strategies to implement dangle contributions: SparseRNAFolD-Trace which utilizes additional trace arrows; SparseRNAFolD-standard, which incorporates bit encoding as well as extension to the definition of candidate structures; and SparseRNAFolD-Triplet, which, similar to the SparseRNAFolD-standard, utilizes bit encoding but modifies candidate energy calculation in anticipation of possible change in parameters in the future. Comparing these three versions on a large dataset, we concluded that the SparseRNAFolD-Triplet implementation is the most efficient in terms of memory, and SparseRNAFolD-Trace is the most efficient in terms of time. These two versions showcase how space and time trade-offs can improve performance for a specific application. The SparseRNAFolD-standard version provides a middle ground for improvement in both time and space and has been chosen as the standard implementation of our algorithm. While guaranteeing the MFE structure and matching the energy of RNAFold, our SparseRNAFolD is on par with LinearFold on memory usage and run time for sequences up to about 1000 bases. This provides a promising starting point to bring dangles contributions to pseudoknotted MFE structure prediction methods in which memory usage is the prohibitive factor [14].

Our results showcase the substantial difference in the number of candidates when compared to quadratic space. This provides an illuminating perspective on the space improvement achieved through sparsification.

We further assessed the effect of hard structural constraints on the performance of SparseRNAFolD, presenting significant improvements both in terms of time and space. We believe the significant improvement in time and space due to the limitation of search space by hard structural constraints can have a more pronounced impact on sparsified pseudoknotted MFE prediction, which is our ultimate goal.

Finally, memory consumption becomes a bottleneck for the prediction of MFE structure for long RNA sequences or MFE pseudoknotted structure prediction. Utilizing the power of computational servers, such restrictions have been somewhat alleviated. Sparsification provides improvements in both time and space requirements and can be used to bring computations back to personal computers, providing equal access to the existing technology. In addition, improvements in memory usage can improve use cases for computing clusters, as the amount of memory assigned to a computing node is also limited.

References

1 M Andronescu, V Bereg, H H. Hoos, and A Condon. RNA STRAND: The RNA secondary structure and statistical analysis database. *BMC Bioinformatics*, 9(1):340+, August 2008. doi:10.1186/1471-2105-9-340.

2 R Backofen, D Tsur, S Zakov, and M Ziv-Ukelson. Sparse RNA folding: Time and space efficient algorithms. *Journal of Discrete Algorithms*, 9:12–31, March 2011. doi:10.1016/j.jda.2010.09.001.

3 J A. Cruz and E Westhof. The dynamic landscapes of RNA architecture. *Cell*, 136:604–609, February 2009. doi:10.1016/j.cell.2009.02.003.

4 S Dimitrieva and P Bucher. Practicality and time complexity of a sparsified RNA folding algorithm. *Journal of Bioinformatics and Computational Biology*, 10, April 2012. doi:10.1142/S0219720012410077.

5 R M. Dirks and N A. Pierce. A partition function algorithm for nucleic acid secondary structure including pseudoknots. *Journal of Computational Chemistry*, 24:1664–1677, August 2003. doi:10.1002/jcc.10296.

6 A F. Bompfünewerer et al. Variations on RNA folding and alignment: lessons from Benasque. *Journal of Mathematical Biology*, 56:129–144, January 2008. doi:10.1007/s00285-007-0107-5.

7 I L. Hofacker et al. Fast folding and comparison of RNA secondary structures. *Monatshefte für Chemie / Chemical Monthly*, 125:167–188, February 1994. doi:10.1007/BF00818163.

8 L Huang et al. Linearfold: linear-time approximate RNA folding by 5'-to-3' dynamic programming and beam search. *Bioinformatics*, 35:i295–i304, July 2019. doi:10.1093/bioinformatics/btz375.

9 R Lorenz et al. ViennaRNA package 2.0. *Algorithms for Molecular Biology*, 6, November 2011. doi:10.1186/1748-7188-6-26.

10 M Gray, S Chester, and H Jabbari. KnotAli: informed energy minimization through the use of evolutionary information. *BMC Bioinformatics*, 23, May 2022. doi:10.1186/s12859-022-04673-3.

11 I L. Hofacker and P F. Stadler. Memory efficient folding algorithms for circular RNA secondary structures. *Bioinformatics*, 22:1172–1176, May 2006. doi:10.1093/bioinformatics/btl023.

12 C E. Holt and S L. Bullock. Subcellular mRNA localization in animal cells and why it matters. *Science*, 326:1212–1216, September 2013. doi:10.1126/science.1176488.

13 H Jabbari and A Condon. A fast and robust iterative algorithm for prediction of RNA pseudoknotted secondary structures. *BMC Bioinformatics*, 15, May 2014. doi:10.1186/1471-2105-15-147.

14 H Jabbari, I Wark, C Montemagno, and S Will. Knotty: efficient and accurate prediction of complex RNA pseudoknot structures. *Bioinformatics*, 34:3849–3856, November 2018. doi:10.1093/bioinformatics/bty420.

15 H Jabbari, I Wark, C Mothentemagno, and S Will. Sparsification enables predicting kissing hairpin pseudoknot structures of long RNAs in practice. In *17th International Workshop on Algorithms in Bioinformatics (WABI 2017)*, volume 88 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.WABI.2017.12.

16 M Kozak. Regulation of translation via mRNA structure in prokaryotes and eukaryotes. *Gene*, 361:13–37, November 2005. doi:10.1016/j.gene.2005.06.037.

17 R Lorenz, I L. Hofacker, and P F. Stadler. RNA folding with hard and soft constraints. *Algorithms for Molecular Biology*, 11, April 2016. doi:10.1186/s13015-016-0070-z.

18 K C. Martin and A Ephrussi. mRNA localization: Gene expression in the spatial dimension. *Cell*, 136:719–730, February 2009. doi:10.1016/j.cell.2009.01.044.

19 D H. Mathews and D H. Turner. Prediction of RNA secondary structure by free energy minimization. *Current Opinion in Structural Biology*, 16(3):270–278, June 2006. doi:10.1016/j.sbi.2006.05.010.

20 D H. Matthews, M D. Disney, J L. Childs, S J. Schroeder, M Zuker, and D H. Turner. Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure. *Proceeding of the National Academy of Science of the USA*, 101:7287–7292, May 2004. doi:10.1073/pnas.0401799101.

21 J S. McCaskill. The equilibrium partition function and base pair binding probabilities for RNA secondary structure. *Biopolymers*, 29:1105–1119, June 1990. doi:10.1002/bip.360290621.

22 M Mohl, R Salari, S Will, R Backofen, and S Sahinalp. Sparsification of RNA structure prediction including pseudoknots. *Algorithms for Molecular Biology*, 5, December 2010. doi:10.1186/1748-7188-5-39.

23 S A. Mortimer, M A. Kidwell, and J A. Doudna. Insights into RNA structure and function from genome-wide studies. *Nature Reviews Genetics*, 15:469–479, May 2014. doi:10.1038/nrg3681.

24 J Nowakowski and I Tinoco. RNA structure and stability. *Seminars in Virology*, 8(3):153–165, 1997. doi:10.1006/smvy.1997.0118.

25 B Rastegari and A Condon. Parsing nucleic acid pseudoknotted secondary structure: Algorithm and applications. *Journal of Computational Biology*, 14, March 2007. doi:10.1089/cmb.2006.0108.

26 J Ren, B Rastegari, A Condon, and H H. Hoos. HotKnots: Heuristic prediction of RNA secondary structures including pseudoknots. *RNA*, 11:1494–1504, October 2005. doi:10.1261/rna.7284905.

27 J S. Reuter and D H. Matthews. RNAstructure: software for RNA secondary structure prediction and analysis. *Proceeding of the National Academy of Science of the USA*, 11, March 2010. doi:10.1186/1471-2105-11-129.

28 E Rivas and S R. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *Journal of Molecular Biology*, 285:2053–2068, February 1999. doi:10.1006/jmbi.1998.2436.

29 R Salari, M Möhl, S Will, S Sahinalp, and R Backofen. Time and space efficient RNA-RNA interaction prediction via sparse folding. In *Lecture Notes in Computer Science*, volume 6044, pages 473–490. Research in Computational Molecular Biology, 2010. doi:10.1007/978-3-642-12683-3_31.

30 N Sugimoto, R kierzek, and D H. Turner. Sequence dependence for the energetics of dangling ends and terminal base pairs in ribonucleic acid. *Biochemisty*, 19:4554–4558, July 1987. doi:10.1021/bi00388a058.

31 D H. Turner and D H. Matthews. NNDB: the nearest neighbor parameter database for predicting stability of nucleic acid secondary structure. *Nucleic Acids Research*, 38:D280–D282, October 2009. doi:10.1093/nar/gkp892.

32 M B. Warf and J A. Berglund. Role of RNA structure in regulating pre-mRNA splicing. *Trends Biochem Sci.*, 35:169–178, March 2010. doi:10.1016/j.tibs.2009.10.004.

33 A Waugh, P Gendron, R Altman, J W. Brown, D Case, D Gautheret, S C. Harvey, N Leontis, J Westbrook, E Westhof, M Zuker, and F Major. RNAML: A standard syntax for exchanging RNA information. *RNA*, 8:707–717, June 2002. doi:10.1017/s1355838202028017.

34 Y Wexler, C Zilberstein, and M Ziv-Ukelson. A study of accessible motifs and RNA folding complexity. *Journal of Computational Biology*, 14:856–872, August 2007. doi:10.1089/cmb.2007.R020.

35 S Will and H Jabbari. Sparse RNA folding revisited: space-efficient minimum free energy structure prediction. *Algorithms for Molecular Biology*, 11, April 2016. doi:10.1186/s13015-016-0071-y.

36 T J. Wilson and D M. J. Lilley. RNA catalysis—is that it? *RNA*, 21:534–537, April 2015. doi:10.1261/rna.049874.115.

37 J Zuber, B J. Cabral, I McFayden, D M. Mauger, and D H. Matthews. Analysis of RNA nearest neighbor parameters reveals interdependencies and quantifies the uncertainty in RNA secondary structure prediction. *RNA*, 24:1568–1582, November 2018. doi:10.1261/rna.065102.117.

38 J Zuber, H Sun, X Zhang, I McFayden, and D H. Matthews. A sensitivity analysis of RNA folding nearest neighbor parameters identifies a subset of free energy parameters with the greatest impact on RNA secondary structure prediction. *Nucleic Acids Research*, 45:6168–6176, June 2017. doi:10.1093/nar/gkx170.

39 M Zuker. Mfold web server for nucleic acid folding and hybridization prediction. *Nucleic Acids Research*, 31:3406–3415, July 2003. doi:10.1093/nar/gkg595.

40 M Zuker and A B. Jacobson. Using reliability information to annotate RNA secondary structures. *RNA*, 4:669–679, June 1998. doi:10.1017/s1355838298980116.

41 M Zuker and P Stiegler. Optimal computer folding of large RNA sequences using thermodynamic and auxiliary information. *Nucleic Acids Research*, 9:133–148, January 1981. doi:10.1093/nar/9.1.133.

Automatic Exploration of the Natural Variability of RNA Non-Canonical Geometric Patterns with a Parameterized Sampling Technique

Théo Boury
Computer Science Department, Ecole Normale Supérieure de Lyon, France

Yann Ponty
Laboratoire d'Informatique de l'Ecole Polytechnique (CNRS/LIX, UMR 7161),
Institut Polytechnique de Paris, France

Vladimir Reinharz
Department of Computer Science, Université du Québec à Montréal, Canada

Abstract

Motivation. Recurrent substructures in RNA, known as 3D motifs, consist of networks of base pair interactions and are critical to understanding the relationship between structure and function. Their structure is naturally expressed as a graph which has led to many graph-based algorithms to automatically catalog identical motifs found in 3D structures. Yet, due to the complexity of the problem, state-of-the-art methods are often optimized to find exact matches, limiting the search to a subset of potential solutions, or do not allow explicit control over the desired variability.

Results. We developed FuzzTree, a method able to efficiently sample approximate instances of an RNA motif, abstracted as a subgraph within a target RNA structure. It is the first method that allows explicit control over (1) the admissible geometric variability in the interactions; (2) the number of missing edges; and (3) the introduction of discontinuities in the backbone given close distances in the 3D structure. Our tool relies on a multidimensional Boltzmann sampling, having complexity parameterized by the treewidth of the requested motif. We applied our method to the well-known internal loop Kink-Turn motif, which can be divided into 12 subgroups. Given only the graph representing the main Kink-Turn subgroup, FuzzTree retrieved over 3/4 of all kink-turns. We also highlighted two occurrences of new sampled patterns. Our tool is available as free software and can be customized for different parameters and types of graphs.

2012 ACM Subject Classification Applied computing → Molecular structural biology

Keywords and phrases Subgraph Isomorphism, 3D RNA, Parameterized Complexity, Tree Decomposition, Boltzmann sampling, Neighborhood metrics, Kink-Turn family

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.20

Related Version *Full Version*: `https://hal.science/hal-04094288/`

Supplementary Material *Dataset*: `https://doi.org/10.5683/SP3/ZR29QE`
Software: `https://github.com/theoboury/FuzzTree`
archived at `swh:1:dir:3a4c9bfc1b1ee5432433713486308ce401a97192`

Funding *Yann Ponty*: PaRNAssum project from the French Agence Nationale de la Recherche (ANR-19-CE45-0023)
Vladimir Reinharz: NSERC RGPIN-2020-05795, FRQS CBJ1

1 Introduction

The essential regulatory and catalytic roles played by RNAs in cellular processes can largely be attributed to the intriguing and highly versatile nature of their structures [8, 5]. The structure of ncRNAs is inherently modular, with distinct structural domains (loops) divided by stems of rigid canonical bonds, often responsible for their unique functions [20]. This modular architecture has been used for advancements in structure prediction [10] and rational design [11]. Consequently, the characterization of ncRNA structure and identification of structural modules have become critical in the pursuit of understanding their diverse functions and exploiting them for future applications.

Many approaches have been developed to detect and classify conserved modules. These classifications differ in the scale adopted to detect and define a motif: RNA3DMotifsAtlas [26] computes similarity and finds motifs at the atomic level. It can capture local similarities omitting bulged nucleotides. A drawback of such a method is the computation time, which restrains comparisons between loops. RNA Bricks [6] and RAG3D [34] abstract loops and hairpins as unitary elements. At an intermediate layer, CaRNAval [27, 30] models RNA as graphs where vertices are nucleotides, and edges are the sequence backbone phosphodiester bonds or non-covalent interactions. These interactions can be classified following the Leontis-Westhof (LW) annotations in 12 different geometric families [21, 31]. Such an approach allows specific graph algorithms to discover much larger and more complex modules than by doing atomic computations while retrieving the known structural modules. However, this approach is not able to identify natural variations since it relies on detecting exact matches.

From the algorithmic point of view, the treewidth tw is a natural parameter to find a match of a pattern graph G_P inside a target graph G_T. In 1995, Alon *et al* [1] proposed an XP [9] algorithm in $O\big(2^{|V_P|}n^{tw(G_P)+1}\big)$ using the color-coding technique. It was shown more generally that only very specific constraints on the input allow having algorithms tractable for bounded treewidths [23]. The problem is not fixed-parameter tractable when parameterized only by the treewidth, and it requires other parameters to become tractable. For instance, some approaches are parameterized both by $tw(G_P)$ and $|G_P|$, and conversely, others are parameterized by $tw(G_T)$ and the maximum degree of G_T [23].

However, there can be an exponential number of variants of a specific pattern so different specialized algorithms allowing missing nodes and edges [25, 12], or requiring only labels to be in a neighborhood [18], have been developed. Such simplifications forget about the precise locations of interactions, which is information that we would like to preserve with RNA structures. A recent approach specific to RNA graph fuzziness uses Relational Graph Convolutional Network to embed the graphs in a vector space, allowing fast computation [24]. Their embedding is based on the nature of base pairs or their isostericity without taking into account gaps or missing edges. By its nature, such a method gives no explicit control over the sampled neighborhoods, and thresholds need to be calibrated depending on the context.

In this paper, we introduce FuzzTree, a multidimensional Boltzmann graph sampling procedure able to sample variants of a motif in a known RNA structure. We allow weighting and control of three key geometric features in the variants: (1) the geometric disruption of mismatched edges, (2) missing edges still constrained by their distance in the 3D structure, and (3) breaks in the backbone also constrained by their distance in the 3D structure. We propose a parameterized bound on the complexity of the algorithm based on the treewidth of the searched motif. We evaluate our method on the well-known interior loop Kink-Turn motif [19] characterized by its sharp bend and clustered into 12 different groups in the RNA3DMotifsAtlas [26]. We show that, from the signature representation of the main subgroup, we sample all their known Kink-Turns in 88% of RNAs. We also retrieve two previously un-annotated loops with a characteristic sharp bend.

2 Method

2.1 RNA as a graph and fundamental problems

We define an RNA structure as a graph G such that its nucleotides are encoded as vertices V, and nucleotide interactions (canonical/non-canonical base pairs, stacking...) are encoded as directed edges $E \subset V \times V$, with labels $L(e)$. Interactions may represent backbone connectivity (phosphodiester bonds), or any of the 12 base-pair types defined by the Leontis-Westhof (LW) nomenclature [31]. Each type specifies an interacting face (Watson-Crick $\circ$, Hoogsteen $\square$, Sugar $\triangleright$) for both nucleotides, along with an orientation cis (filled) or trans (empty). Note that the geometry of the RNA structure is encoded in the edge labels, and our representation does not depend on the sequence. In this work, we are interested in RNA 3D **motifs**, which we abstract as RNA **pattern graphs** as depicted above. We show in Fig. 1 a Kink-Turn motif, represented as a graph with labeled edges.

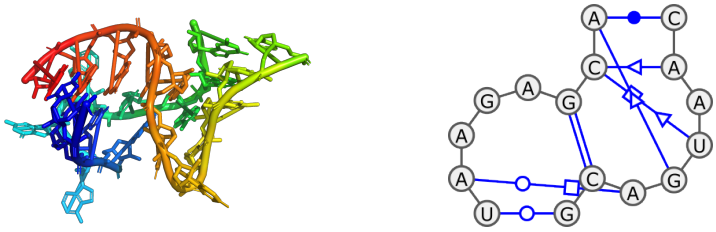


Figure 1 Kink-turn structure. On the left, the 3D structure of a Kink-Turn motif in PDB `3RW6`. On the right, its representation as a pattern graph of its base pair interactions. The backbone connections are represented as black arrowed edges.

We rewrite E, the set of edges as $E = B \sqcup \overline{B}$, composed of two distinct sets: B, the set of edges that are backbone interactions and $\overline{B}$, the edges involved in LW interactions.

Moreover, since vertices in both pattern and target graphs are indexed by their sequence position, we introduce a precedence relation $\prec$, inducing a strict total order within the pattern and target graphs. A valid occurrence of a pattern within a target must be monotonous, *i.e.* remain consistent with the strict precedence relation $\prec$.

The Monotonous Subgraph Isomorphism (MSI) problem identifies an occurrence of a pattern $G_P = (V_P, E_P)$ inside a target graph $G_T = (V_T, E_T)$. In the context of RNA, G_P is a (closed) motif and $\prec$ −Hamiltonian, *i.e.* the total order over V_P induced by the relation $\prec$ represents a (Hamiltonian) path in G_P, while G_T represents an entire RNA structure. Formally, the problem of searching for G_P within G_T can be defined as:

▶ **Problem 1.** *Monotonous Subgraph Isomorphism Problem (MSI)*
Input: *Pattern graph ($\prec$ −Hamiltonian) $G_P = (V_P, E_P)$; Target graph $G_T = (V_T, E_T)$*
Output: *Mapping $M : V_P \to V_T$ such that*

- $\forall (u, v) \in {V_P}^2, u \prec v \Rightarrow M(u) \prec M(v)$ ***(monotonicity)***
- $\forall (u, v) \in E_P, (M(u), M(v)) \in E_T \Rightarrow L((u, v)) = L((M(u), M(v)))$ ***(label comp.)***
- $\forall (u, v) \in E_P, (M(u), M(v)) \in E_T$ ***(no missing edge)***

or $\varnothing$ if no such mapping exists.

The MSI problem represents a constrained version of Subgraph Isomorphism, a well-studied NP-complete problem [13, 23] with mildly-depressing prospects regarding parameterized complexity. Indeed, Subgraph Isomorphism does not admit Fixed-Parameter Tractable (FPT) or slicewise polynomial (XP) solutions for various graph parameters, including the treewidths

$tw(G_P)$ and $tw(G_T)$ of the pattern and target graphs. Namely, the problem was shown [23] to be NP-hard even when $\max(tw(G_P), tw(G_T)) \leq 2$ (Para NP-hardness), ruling out the existence of FPT or XP algorithms under standard hypotheses.

The MSI problem retains the classical NP-hardness of Subgraph Isomorphism since it can be shown to generalize the NP-hard structure-sequence alignment in RNA [28]. However, MSI can be solved in time $\mathcal{O}\left(|E_P|.|V_T|^{tw(V_P)+1}\right)$ (XP algorithm) using classic dynamic programming based on a tree decomposition of G_P (see Section 2.5 and Supp. Mat. A.2.2). Such an algorithm has polynomial complexity for any fixed value of the treewidth $tw(G_P)$, a parameter that remains bounded in practice (typically 2 or 3) for RNA motifs.

2.2 Capturing geometric and chemical similarities

We now extend our problem to embrace the natural diversity of RNA motifs in structures. More precisely, we are interested in sampling graph occurrences that are in the geometric neighborhood of a core motif. To do so, we allow the motif to be deformed by three different biologically relevant edit operations detailed below. Each contributes additively and has its own **neighborhood threshold**, and corresponding difference function, as depicted in Table 1:

- T^L represents how much we allow the edge label, the type of the canonical or non-canonical bond, to be modified. It measures the geometric difference between two interactions (see Sec. 2.4.1).
- T^E corresponds to the maximum number of edges/base pairs within the pattern structure that can be omitted (see Sec. 2.4.2).
- T^G is the maximum allowed distance when introducing a backbone discontinuity, a new gap. As insertions alter the distance between bonds, T^G regulates here the maximum sum over these shifts (see Sec. 2.4.3).

We denote by GEO the **geometric distance** between two nucleotides u_1 and u_2 as

$$\mathsf{GEO}(u_1, u_2) = \min_{a_i \in \text{atoms}(u_i) | i \in \{1,2\}} ||a_1 - a_2||_2,$$

and use it to define two additional criteria to constrain admissible solutions:

- First, nucleotides mapped to the nodes of a missing edge must be closer than D_{edge} Å;
- Second, we enforce a maximal distance D_{gap} between the nucleotide on both sides of an introduced gap. These values correspond to the phosphodiester atoms' distances between the nucleotides. Capping these distances beyond a fixed value not only yields more realistic outputs but also greatly improves the runtime of our algorithm.

We use the **isodiscrepancy index** [31] to quantify geometrically the difference between base pair families and provide values measuring three terms: (1) the difference of intra-base pair C1'–C1' distances; (2) after aligning one base, the inter-base pair C1'–C1' distance between the C1' atoms of the second bases of the base pairs; (3) The angle on an axis perpendicular to the base pair plane required to superpose the second bases. This isostericity measure is defined for pairs of base pairing families (BPF), each representing one of the 12 canonical and non-canonical conformations and named as $\mathsf{BPF}_i, \forall i \in [\![1, 12]\!]$. Inter-family variations are frequent and therefore the average isodiscrepancy of a family to itself is not 0. To correct for this phenomenon, we define the ISO difference between two families as:

$$\mathsf{ISO}(\mathsf{BPF}_i, \mathsf{BPF}_j) = \mathsf{isodiscrepancy}(\mathsf{BPF}_i, \mathsf{BPF}_j) - \mathsf{isodiscrepancy}(\mathsf{BPF}_i, \mathsf{BPF}_i).$$

Moreover, we set the value of ISO to 0 involving undefined labels, backbones or phantom interactions.

We define a **backbone path** as a sequence of at least 2 nucleotides connected through backbone edges.

The set P of paths associated with a target graph $G_T = \left(V_T, E_T = B_T \sqcup \overline{B}_T\right)$ is defined as:

$$P = \bigcup_{k\in\mathbb{N}, k\geqslant 2} \{(p_0, ..., p_k) \mid \forall i \in [\![0, k-1]\!], (p_i, p_{i+1}) \in B_T\}.$$

With this definition, gaps are just paths in P with specific restrictions on length and composition.

A mapping M lying in a relevant neighborhood of a pattern graph is a solution to a problem that we call the **Fuzzy Monotonous Subgraph Isomorphism problem (FMSI)**, which can be defined as:

▶ **Problem 2.** *Fuzzy Monotonous Subgraph Isomorphism problem (FMSI)*
Input: *Pattern graph* $G_P = \left(V_P, E_P = B_P \sqcup \overline{B}_P\right)$ *($\prec$ −Hamiltonian) , target graph* $G_T = \left(V_T, E_T = B_T \sqcup \overline{B}_T\right)$ *and neighborhood thresholds* (T^L, T^E, T^G, D_{edge}, D_{gap})
Output: *Mapping* $M : V_P \to V_T$ *such that:*

1. $\forall (u, v) \in {V_P}^2, u \prec v \Rightarrow M(u) \prec M(v)$ ***(monotonicity)***
2. $\sum_{(u,v)\in\overline{B}_P} \mathit{ISO}(L(u, v), L(M(u), M(v))) \leqslant T^L$ ***(label compatibility)***
3. $\sum_{(u,v)\in\overline{B}_P} 1 - \mathbb{1}_{(M(u),M(v))\in\overline{B}_T} \leqslant T^E$ ***(few missing edges)***
4. $\forall (u, v) \in \overline{B}_P, (M(u), M(v)) \notin \overline{B}_T, \mathit{GEO}(M(u), M(v)) \leqslant D_{edge}$ ***(edge distance limit)***
5. $\sum_{(p_0,...,p_k)\in P, k\geqslant 3} \mathit{GEO}(p_0, p_k) \leqslant T^G$ ***(path size limitation)***
6. $\forall (u, v) \in B_P, \exists (p_0, p_1, p_2, ..., p_k) \in P$ *such that* ***(no missing backbone path)***
   - $p_0 = M(u), p_k = M(v)$ ***(*)***
   - $\mathit{GEO}(p_0, p_k) \leqslant D_{gap}$ ***(**)***

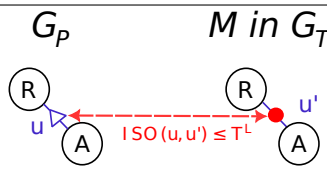
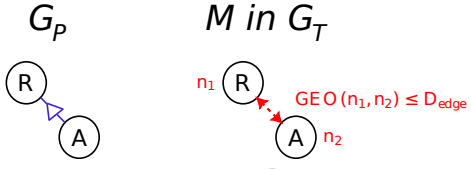
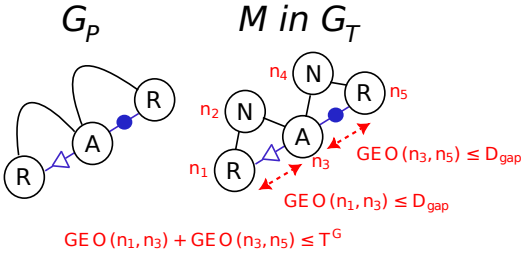
or $\varnothing$ *if no such mapping exists.*

Intuitively, a valid mapping M has to respect the six following conditions: The **monotonicity** condition enforces pattern nodes to map successive nodes in the target. The **label compatibility** controls how much the geometric differences cumulative is allowed between pattern and matched edges (see Sec. 2.4.1). The **few missing edges** constraint ensures that pattern edges that are not mapped to an edge in the target are not numerous. (see Sec. 2.4.2) The **edge missing limit** forces each couple of mapped nodes with no edges to have a bounded geometric distance between each other. (see Sec. 2.4.2) The **path size limitation** controls how large the cumulative of gaps geometric lengths can be. (see Sec. 2.4.3) The **no missing backbone path** condition (as unfolded in Prob. 2) ensures that the start and end points of a path are mapped nodes **(*)**. It also restrained allowed geometric length of individual path **(**)**. (see Sec. 2.4.3) We note that due to the monotonicity condition, it implies that no target node in $p_1, \ldots, p_{k-1}$ can belong to the mapping.

Subsequently, we will denote by $\mathbf{neighborhood}_{G_P}(G_T)$ all the occurrences of the desired pattern graph G_P (in its geometric neighborhood) in our RNA graph target G_T as defined by the previous FMSI mapping.

In practice, RNA graphs are fully ordered but do not necessarily contain a Hamiltonian path due to backbone disconnections, leading to a graph composed of multiple strands. We can reconstitute a Hamiltonian path (with no complexity overhead) in the pattern graph by adding some "phantom edges" (with a specific label) when the backbone is missing which correspond to the set of edges $\{(i, i+1) \mid i \in G_P, (i, i+1) \notin E_P \cup L(i, j) \neq \text{“B53”}\}$. Additionally, to ensure that such edge can be mapped in the target G_T in a way that will conserve the monotonicity of the mapping, we add in G_T the set of edges $\{(i, j) \mid (i, j) \in G_T, i \prec j \cap L(i, j) \neq \text{“B53”}\}$.

Table 1 Neighborhood thresholds and differences. Each measure has a threshold over the sum of differences over all edges in the graph pattern.

Threshold T^F	Difference d^F	Fuzzy mapping M of G_P found in G_T
T^L	Isostericity ISO	
T^E	Missing edges number	
T^G	Geometric GEO from 3D structure	

2.3 Locating alternative occurrences through sampling

Focusing on neighborhood$_{G_P}(G_T)$ is not an easy task as naive methods would describe both this set and its complementary. In the clique worst case, it consists to explore $\binom{|G_T|}{|G_P|}$ graphs. Even the simple exploration of neighborhood$_{G_P}(G_T)$ can be tedious, in particular, when neighborhood thresholds are quite large, which is often the case for label and gap thresholds. Furthermore, due to the nature of the neighborhoods, numerous instances of a few nucleotides apart will often be found. It is relevant in terms of neighborhoods, but, from the biological standpoint, they represent all the same RNA portion and the same underlying geometry and should not be distinguished: a single representative will be enough. It oriented us toward sampling, to identify sets of candidate – ideally diverse – subgraphs inside the target graph G_T that are at a reasonable " distance" from the interesting motif G_P.

This shift in paradigm builds on recent advances in Multidimensional Boltzmann distributions and sampling [2, 15].

Generally, a **Boltzmann distribution** is such that the probability of any possible outcome G depends on its (pseudo-)**energy** E:

$$\mathbb{P}(G) = \frac{e^{-\beta E(G)}}{\mathcal{Z}} \text{ where } \mathcal{Z} = \sum_{G'} e^{-\beta E(G')} \tag{1}$$

where β is a real number, akin to an inverse temperature. A **Multidimensional Boltzmann distribution** (MBD) is a special type of Boltzmann distribution, where the energy is a weighted combination over a collection of features $\{F_i\}$ of interest, such that

$$E(G) = w_1 \times F_1(G) + w_2 \times F_2(G) + \dots$$

where $w_1, w_2 \dots$ are real-valued weights. Weights can be used to steer the sampling towards regions of interest. They can also be learned, through convex optimization, to match the expectations of $F_1, F_2 \dots$ to user-specified values. Moreover, sampling with a pseudo-temperature $\beta \to \infty$ gracefully specializes in a uniform random generation of outcomes achieving optimal (*i.e.* minimal) value for E.

In our case, an outcome is a graph $G \subset G_T$, such as G is the image of mapping M and we have 3 features, one for each neighborhood. Given a specific neighborhood threshold T^F, its relative feature F measures how much the weight of edits D^F relative to neighborhoods, further introduced as a difference in 2.4, deviate from a given center T^{F*}. For instance, T^{F*} can be chosen as equal to 0 if we want to sample mostly G with no fuzziness or as equal to $T^F/2$ if we want to sample them with average fuzziness. More details on this choice and about Boltzmann sampling are available at Supp. Mat. A.1. MBD is well-suited to the sampling that we want to make: the exponential decrease of the probability with the features gives low probabilities to the graphs that are far in terms of neighborhoods from G_P, which allows us to characterize well neighborhood$_{G_P}(G_T)$. In particular, we can define F such that it takes a value equal to $+\infty$ when the corresponding neighborhood threshold T^F, for a mapping M, is not respected, forbidding simply M to be sampled. Additionally, the Multidimensional character of the distribution allows us to take into account the 3 neighborhoods on labels, edges and gaps at the same time.

A general framework called `InfraRed` [33], initially introduced in the context of RNA design [15], can be used to generate efficiently, in a parameterized manner, the MBD. It automatically processes constraints and elements of the scoring into a graph, decomposes it into a Tree Decomposition, and generates automatically the bottom-up dynamic programming sampling procedures. More details on the Tree Decomposition and the dynamic programming used in `InfraRed` can be found in Supp. Mat. A.2.

2.4 Neighborhood difference description

Our goal is to be able to retrieve from a general motif all natural occurrences and their variability. We can observe in well-known motif families that some bases change, some can be added or removed. For instance, the graph pattern G_P on Fig. 2 is a Kink-Turn whose occurrences in the same sub-family can have up to four missing edges. Other sub-families of Kink-Turn motifs can have differences in bond types, additional interactions, or even gaps induced by additional nucleotides. We will define difference functions that will be the features in the MBD and will restrain the samples to a "reasonable" neighborhood of the pattern G_P that can be explicitly defined.

For any feature F (here $F \in \{L, E, G\}$, where L are label changes, E missing edges, and G new gaps) the **Neighborhood cumulative difference** D^F quantifies how distant a mapping is, relatively to a given neighborhood threshold T^F that cannot be exceeded.

Formally, we define a neighborhood cumulative difference D^F relatively to a neighborhood threshold T^F as:

▶ **Definition 1** (Neighborhood cumulative difference / neighborhood difference). *Given a pattern graph $G_P = \left(V_P, E_P = B_P \sqcup \overline{B}_P\right)$, a target graph $G_T = \left(V_T, E_T = B_T \sqcup \overline{B}_T\right)$ and a mapping M, a neighborhood cumulative difference is a function D^F relatively to a neighborhood threshold T^F that act as a wrapper around $d^F_{G_T}$:*

$$D^F(G_P, G_T, M) = \sum_{(u,v)\in E_P} d^F_{G_T}(u, v, M)$$

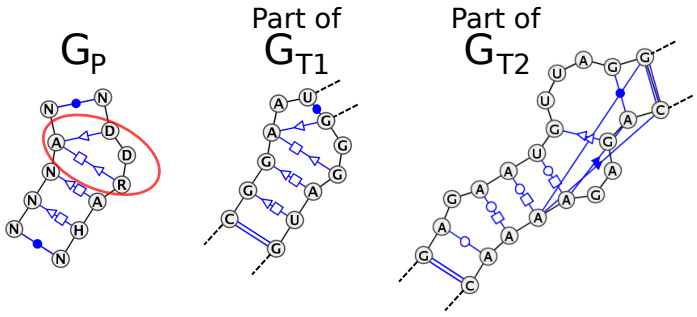


Figure 2 Kink-turn signature and targets. On the left, signature graph of the Kink-Turn IL_29549.9 family and our search pattern. In the middle and on the left, mappings that were missed during the search for the pattern. G_{T1} due to the same nucleotide merging the end of a cSS and a cWW. G_{T2} due to its too large difference.

where $d^F_{G_T}(u, v, M)$ *is the* ***neighborhood difference*** *relative to* G_T*, a function that measures, relatively to* F*, how "different" are the edges in the pattern* $((u, v) \in G_P)$ *from the edges in the mapping* $((M(u), M(v)) \in G_T)$*.*

How the difference is measured depends on the feature as described below.

Neighborhood cumulative differences serve in the Boltzmann distribution to quantify each type of edit. Due to the additivity of these deformations, the neighborhood cumulative differences are computed over all edges in the pattern and their equivalent in the mapping. While our neighborhood cumulative differences are defined relative to the edges of G_P here, they can be easily defined on nodes should novel sequence-dependant features be included. We will now discuss in detail the 3 sources of operations and their neighborhood cumulative difference. A summary is shown in Table 1.

2.4.1 The label difference

The label difference, as represented in Fig. 3, accounts for the difference between base pairs families and we use for that the isodiscrepancy [31] as introduced in part 2.2. We now compute the label difference D^L relative to the neighborhood threshold T^L as a neighborhood cumulative difference entirely defined by the sum over each pattern edge of its mapping neighborhood difference $d^L_{G_T}$ equals to:

$$d^L_{G_T}(u, v, M) = \mathsf{ISO}(L(u, v), L(M(u), M(v))).$$

2.4.2 The edge difference

While the previous section deals with how to incorporate edges changing their type, *i.e.* their interaction geometry, we must also consider that some of these base pair interactions might simply be missing due to the noisiness of the experiments, the accuracy of the annotation, or the flexibility of the module. A natural way to account for missing edges is to count them and enforce an upper bound on the amount. Doing so would omit important geometric information that we have available in the 3D structure. An interaction is missing, but we still want to constrain the physical distance between the mapped nodes of the missing edge. Indeed, with no limitation on that distance, the partner node of a missing edge could be virtually anywhere in the target structure. This is undesirable since we are interested in patterns matching the local conformations. It is also highly inefficient in terms of computation.

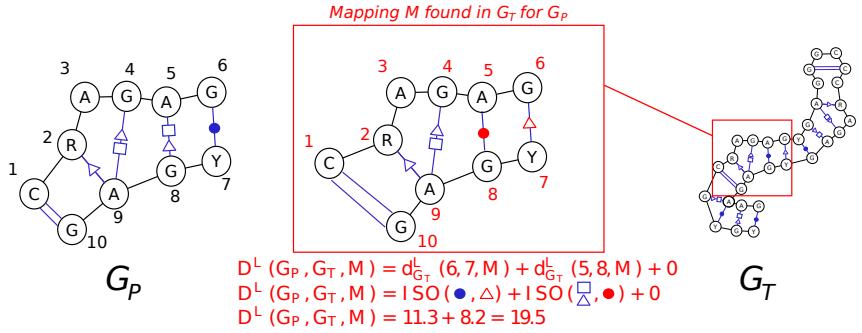


Figure 3 Label difference. Computation of the label difference on a mapping between a motif G_P and an RNA target graph G_T. Label difference is computed using the isostericity ISO to account for the geometric difference between bounds as described in Stombaugh et al [31].

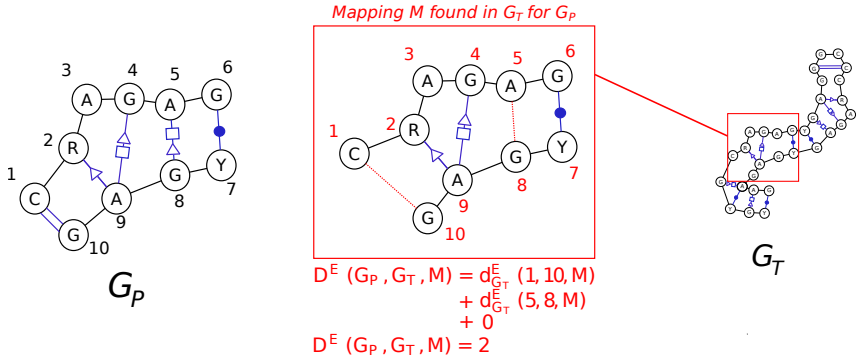


Figure 4 Edge difference. Computation of the edge difference on a mapping between a motif G_P and an RNA target graph G_T. We assume here that $D_{\text{edge}} >> max(\mathsf{GEO}(1,10), \mathsf{GEO}(5,8))$.

Therefore, we will accept mappings of the extremities of an edge in the pattern to nodes u, v that are at most at a set threshold distance D_{edge} computed from the 3D structure (*i.e.* $\mathsf{GEO}(u,v) < D_{edge}$). Setting a weight of ∞ to mappings outside the threshold allows the sampling to simply reject such instances. We additionally use the edge difference to reject cases where backbones are mapped to couples of nodes that are not backbones by putting a weight ∞ in that case. The total edge difference D^E relative to neighborhood threshold T^E, is a neighborhood cumulative difference entirely defined by the sum over $d^E_{G_T}$ with values defined as followed and shown in Fig. 4:

$$d^E_{G_T}(u,v,M) = \begin{cases} 0 & \text{if } (u,v) \in B_P \cap (M(u),M(v)) \in B_T \\ & \text{or } (u,v) \in \overline{B}_P \cap (M(u),M(v)) \in \overline{B}_T \\ 1 & \text{if } (u,v) \in \overline{B}_P \cap (M(u),M(v)) \notin \overline{B}_T \\ & \text{and } \mathsf{GEO}(M(u),M(v)) \leqslant D_{\text{edge}} \\ \infty & \text{otherwise.} \end{cases}$$

2.4.3 The gap difference

A frequent type of natural variability in a motif family is the occurrence of bulging out nucleotides in what would be a continuous sequence in the pattern. These insertions can be of different sizes, but we require that they do not modify (too much) the local structure. To

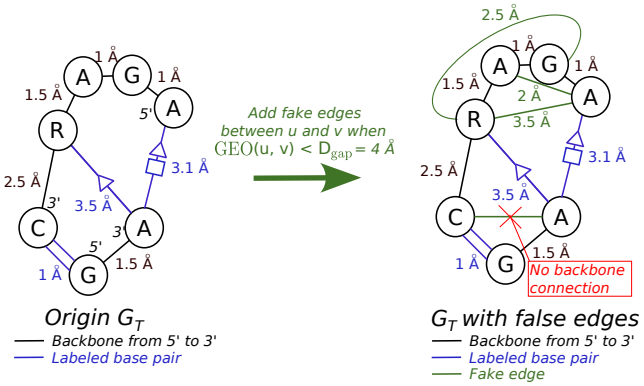


Figure 5 Fake edges. Addition of fake edges to account for gaps. Fake edges are added only when distance is below D_{gap} and when both nucleotides are fully connected by backbone edges. For instance here, we add no fake edge between C and A at the bottom of G_T as these two nucleotides are not connected by a full path of backbones.

take arbitrary insertions into account, we introduce **fake edges** between any two nucleotides present on the same backbone that are at a distance below D_{gap}. An illustration of this process is shown in Fig. 5. For convenience, these edges are added in B_T to keep valid the cases of the edge difference where backbones are wrongly mapped.

An additional difference compared to the missing interaction edges of the previous section is how we sum the total neighborhood difference D^G. We accumulate the total physical distance (*i.e.* GEO) between the nodes connected through the fake edges. This allows an arbitrarily large structure to bulge out without the need to verify or specify admissible lengths, as long as the nucleotides around this inserted gap are close geometrically as illustrated in Fig. 6.

Formally, the gap difference D^G relative to neighborhood threshold T^G is a neighborhood cumulative difference over all edges in the matching entirely defined by the sum of the neighborhood differences $d^G_{G_T}$:

$$d^G_{G_T}(u, v, M) = \begin{cases} \mathsf{GEO}(M(u), M(v)) & \text{if } (M(u), M(v)) \text{ is} \\ & \text{a “Fake Edge” in } E_T \\ 0 & \text{otherwise.} \end{cases}$$

A limitation of this approach is that we cannot detect the deletion of nodes from the pattern. A workaround is to remove all the nodes in the pattern graph that do not directly participate in a base pair interaction, and reconnect the disconnected backbones. Using the new pattern with a large gap threshold T^G would allow us to retrieve the original motif neighborhood efficiently, but introduce more spurious matches.

2.5 Algorithm and complexity

Our method is based on `Infrared` [15, 33], a declarative framework that automatically generates a dynamic programming procedure for MBD sampling, based on a nice tree decomposition (TD). The dynamic programming procedure used in Infrared is described in Supp. Mat. A.2. It precomputes the partition function of the MBD through a bottom-up recursion and uses local contributions to perform an exact sampling within the MBD distribution. Within this framework, a combinatorial problem is abstracted as a set of

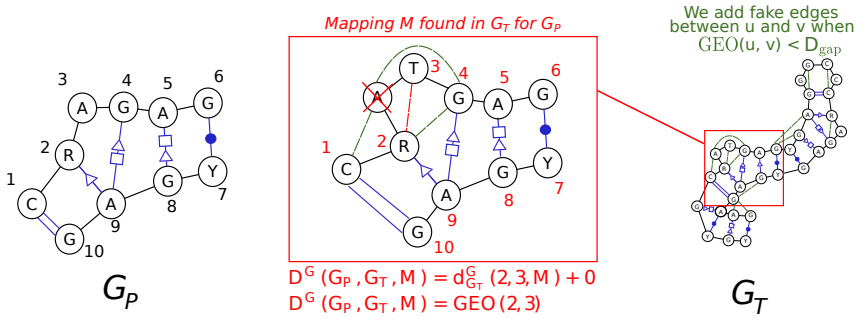


Figure 6 Gap difference. Computation of gap difference on a mapping between a motif G_P and an RNA target graph G_T. We recall that nucleotide labels are not taken into account.

variables $\{X_i\}_i$, each assigned an integer value within a bounded domain. Assignments must respect various constraints expressed as functions $\{C_i\}_i$, each defined over a subset of variables. Similarly, feature functions $\{F_j\}_j$ associate real-valued contributions to subsets of variables, and are summed to represent the pseudo-energy of an assignment.

In this setting, we abstract each node i of the graph pattern G_P as a variable X_i, taking value in $[\![1, n]\!]$. The value of X_i represents the mapping of node i in the graph $G_T = (V_T, E_T)$ with $|V_P| = k$ and $|V_T| = n$. Within RNA motifs, the number of partners of a position is bounded, so we have $|E_P| \in \mathcal{O}(k)$. Remark that all deviations from the pattern defined in Sections 2.4.1 through 2.4.3, can be expressed *locally* as sums on the edges of the pattern graph. It follows that the dependencies *dep* implied by our cumulative differences are only binary, and restricted to pairs sharing an edge in G_P: $dep = \{(X_i, X_j) \mid (i, j) \in E_P\}$. The graph of constraints is thus reducible to the input pattern graph G_P, as shown in Fig. 8.

Due to the neighborhood threshold T^F being a global property over the mapping, the sampling is followed by a rejection step for samples that exceed a neighborhood threshold. An example of such rejection is depicted in Fig. 7. Asymptotically, such rejection will at worst induce a constant overhead with T^F chosen independently from $|G_P|$ and $|G_T|$.

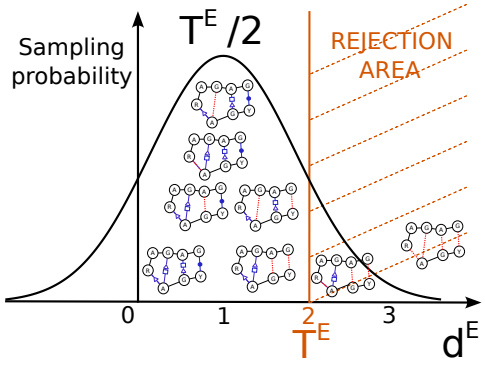


Figure 7 Rejection step. In the above example, rejection is depicted only for the edge neighborhood for the sake of simplicity. Found motifs above T^E thresholds are rejected afterward. Found motifs with an edge difference close to $\frac{T^E}{2} = 1$ here have more chance to be sampled.

▶ **Proposition 2.** *A generation of t Boltzmann-distributed* (1) *putative solutions to FMSI can be performed in time $\mathcal{O}\big(n\,k\,t + k\,n^{(\phi+1)}\big)$ where ϕ is the treewidth of the pattern G_P.*

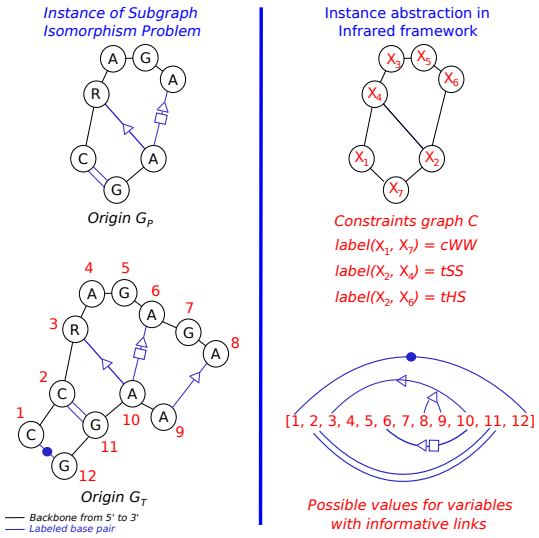


Figure 8 Framework abstraction. Interfacing `Infrared` by considering G_P as the `Infrared` graph of constraints C and all nodes of G_T as values that can be taken by the variables in C.

This complexity directly follows from the complexity of the algorithm [15] underlying `Infrared` for a graph $G_P = (V_P, E_P)$ (with $|V_P| = k$). Restricted to binary constraints/features associated with (a subset of) E, the computation of the partition function can be performed in time $\mathcal{O}((|E_P| + |V_P|) \times \Delta^{\phi+1})$, where Δ is the size of the assignment domain for individual variables, and ϕ is the treewidth of G_P. A stochastic backtrack follows, leading to the generation of t Boltzmann-distributed assignments in time $\mathcal{O}(|V_P|\,\Delta\,t)$. The complexity stated above is obtained by observing that $|E_P| \in \Theta(k)$, and that $\Delta \in \Theta(n)$.

We conclude by noting that preprocessing, including computations of geometrical distances and augmentation of G_T graph, can be performed once, in $O(n^2)$ time and space, leading to a negligible overhead in comparison to the computation of the partition function. Meanwhile, an optimal tree decomposition can be theoretically obtained in time only super polynomial in ϕ [3].

A summary of the complexity and capacity of our FuzzTree method is depicted in Table 2. Regarding the parameterized complexity [9], the FuzzTree method is XP in the treewidth of the pattern graph, both in time and in space. It represents progress compared to VF2 [7], which is indeed implemented and efficient in practice due to the profusion of lookahead rules but has a worst-case time complexity similar to $O(n^n)$. In practice, VF2 becomes costly with dense graphs, even in its most modern versions [4, 17]. Furthermore, we compete with the bound from the Color-Coding [1] technique by improving it in time and space. $2^{O(k)}$ is replaced by $k \leqslant n$ in our bounds, which allows us to get rid of k as a parameter to restrict it simply to the treewidth in our RNA case.

In addition, our method handles at the same time multiple labels on edges, directed graphs and can integrate node labels. The latter has not been implemented but can be added, as with labels on edges, without complexity overhead.

Table 2 Complexities for RNA motif search. Comparison of state-of-the-art methods for RNA motif search. With $\phi = tw(G_P)$, $n = |V_T|$, $k = |V_P|$ and t the number of samples.

Method Name	Color-Coding [1]	VF2 [7]	VeRNAl [24]	FuzzTree
Year	1995	2004 (updated up to 2018)	2021	2022
Method	Tree coloring	DFS with search space reduction	Relational Graph Convolution Network	Sampling technique
Time complexity	$2^{O(k)}n^{\phi+1}log(n)$	$O(\deg(G_T)^n)$	Exponential	$O\left(knt + kn^{\phi+1}\right)$
Space complexity	$2^{O(k)}n^{\phi+1}$	$O(n)$	Exponential	$O\left(n^{\phi+1}\right)$
Supported graph	Directed and undirected	Undirected	Directed and undirected	Directed and undirected
Supported labels	One label by edge	One label by node	Any number of labels on edges and nodes	Any number of labels on edges and nodes
Type of found neighborhoods	None	None	Isostericity related	Exact bound on isostericity, missing edge and missing gap.
Implementation?	No	Yes	Yes	Yes

3 Results

3.1 Computations

The larger target graphs (of more than 500 nucleotides) were split into overlapping voxels to increase computational efficiency. We extracted $|G_T|$ graphs centered in each nucleotide c at a given radius R from c. For an extracted graph G, centered on c, we have:

$$\forall j \in G, R(G) = \mathsf{GEO}(j, c) \leqslant R.$$

Choices of technical parameters, such as the value for R, hardware and computation times are discussed in Supp. Mat. A.3. For the sake of efficiency, we refrained from adding "phantom edges" described in Section 2.2. Doing so enables possible violations of the monotonicity, leading to the detection of motif occurrences in the context of a more remote homology, but necessitated a further round of rejection (whose impact on performances remained negligible).

3.2 Data: the Kink-Turn motifs family

All interactions in the RNA structures are provided by FR3D [29]. We also use interactions annotated as "near". The Kink-Turn is an important RNA structural motif common in duplex RNA that creates a sharp axial bend, enabling crucial tertiary interactions and binding [19]. The Kink-Turn has been shown to appear in multitudes of contexts through computational and experimental methods [16, 22]. As of January 2023, there were 72 instances of the Kink-Turn RNA annotated in the RNA3DMotifAtlas [26]. One was omitted because it was not annotated on the main structure but one of its symmetric alternatives. The others span 46 different RNAs and are divided into 12 different families with different lengths, between 9 and 23 nucleotides and base pair signature. Members of the same family also differ in terms of number of nucleotides and pairing.

The Kink-Turn family IL_29549.9 in RNA3DMotifsAtlas has the most occurrences (32) and its signature graph shown in Fig. 2 is used as the pattern graph G_P for the subsequent sampling.

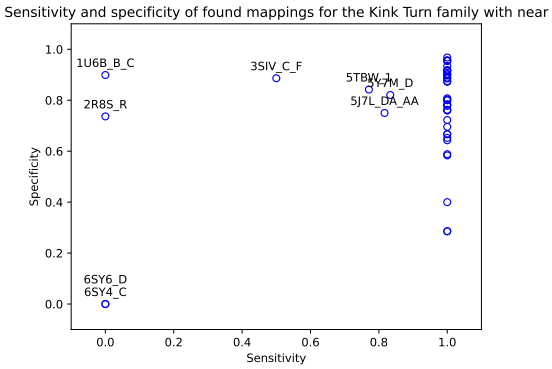


Figure 9 Sensitivity and Specificity of regions corresponding to sampled graphs in the 46 RNA structures containing Kink-Turns. Each dot represents an RNA chain, where one or multiple Kink-Turns can be found. To keep track of them, nodes whose sensitivity is not equal to one, are named of the graph "RNAname"_"chain".

Empirically, RNA 3D motifs are small motifs that, despite not being tree-like, have relatively small treewidth. It is especially the case for the Kink-Turn family, where 50 Kink-Turns pattern graphs have treewidth equal to 2 and 21 have treewidth equal to 3, which makes our parameterization in treewidth practically quite relevant.

3.2.1 Results

We use the parameters shown in Table 3 with G_P in Fig. 2 to sample at least 1000 graphs in each of the 46 RNA structures. We also introduce a bias in the Boltzmann distribution to favor values of neighborhood thresholds equal to $\frac{T^F}{2}$ (instead of 0) to favor slightly fuzzy mappings more often than exact mappings or extremely fuzzy ones. This choice is motivated by the focus on the neighborhood more than on the exact mappings for which lots of techniques already exist.

Table 3 Parameters. Used parameters and relevant range for FuzzTree computation on the Kink-Turn group.

Parameter	T^L	T^E	T^G	D_{edge}	D_{gap}	R	nb_samples
Used value	20.0	4	20.0	5.0	10.0	$R(G_P) + \frac{D_{\text{gap}}}{4}$	1000
Relevant range	$[0, 50]$	$[\![0, 6]\!]$	$[0, 50]$	$[5, 10]$	$[5, 20]$	$R(G_P) + [\frac{D_{\text{gap}}}{4}, D_{\text{gap}}]$	

Our sampling returns sub-graphs of the target graphs G_T. Using a python implementation of VF2 [14, 7], we annotate in the 46 RNAs graphs all nucleotides in any of the mappings. Each of the connected components in the 46 RNAs becomes a hit. The True Positives (TP) are these covering a known Kink-Turn found by our method. The True Negative (TN) are those that do not cover a Kink-Turn, rightly not found by our method. P designs the set of all Kink-Turn motifs and N the set of all other motifs. We show the sensitivity (TP/P) and specificity (TN/N) per RNA structure in Fig. 9.

In 38 out of the 46 RNAs a sensitivity of 1 is achieved, all Kink-Turns are covered in graphs sampled by our method. The missing Kink-Turns fall into two categories. First, too many missing edges: with only 6 Leontis-Westhof interactions in G_T, allowing more missing edges would match any interaction in the targets. Second, backbone connections replaced by Leontis-Westhof interactions, as seen in the middle of Fig. 2, is not an allowable transformation in our model.

We also obtain in 33 RNAs a specificity over 75%. It indicates that even with relatively lax parameters, not that many other instances in comparison to the amount of known Kink-Turns are close to G_T.

3.2.2 Other identified regions

An additional 198 locations in the 46 RNAs were identified. The Kink-Turn is essentially an internal loop motif. We investigate if other internal loops sharing the same main 3D feature, a sharp bend in an interior loop, are found. Using the python library forgi [32] we decomposed these regions in their secondary structure elements. The majority, 125, mapped to regions forming multiloops. A total of 33 were covering continuous double-stranded regions. The angles of surrounding stems for each interior loop in the 46 RNAs (in blue) the identified Kink-Turns in these RNAs (red) and the other 33 elements (in green) are shown in Fig. 10.

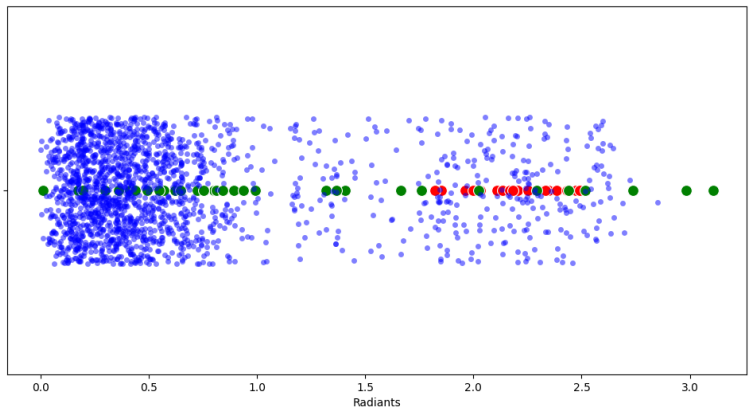


Figure 10 Angles in radiants. In blue for stems around every interior loop in the 46 RNAs. In red for the Kink-Turns identified in these RNAs. In green for the additional 33 continuous double-stranded regions.

There are 10 additional regions with angles above 1.4rad, and two of these had a sharp turn in their structure in un-annotated region as seen in Fig. 11. We show below their graph of interactions, with the cross-strand stackings in orange.

The first is in 5J7L chain DA and positions 78–86, 96–108. It overlaps an un-annotated motif (IL_85931.1) that covers positions 81–85, 97–101, and 103–105. The second is located in 7RQB, chain 1A, positions 2129–2138, 2153–2160, and is not covered or surrounded by any annotated motif.

4 Conclusion

In this paper, we introduce FuzzTree, a multidimensional Boltzmann method for sampling a graph pattern neighborhood in a target graph. FuzzTree defines three types of neighborhoods based on RNA geometric diversity, LW interaction modifications, missing edges, and breaks in the backbone. Each can be explicitly controlled. We show that our sampling method complexity is parameterized by the treewidth of the pattern graph.

Two main limitations are inherent to our approach. Due to the intrinsic nature of sampling, we cannot be assured that all neighboring graphs will be reported. In itself, for large patterns, this is a feature since sampling allows uniform exploration of the exponentially

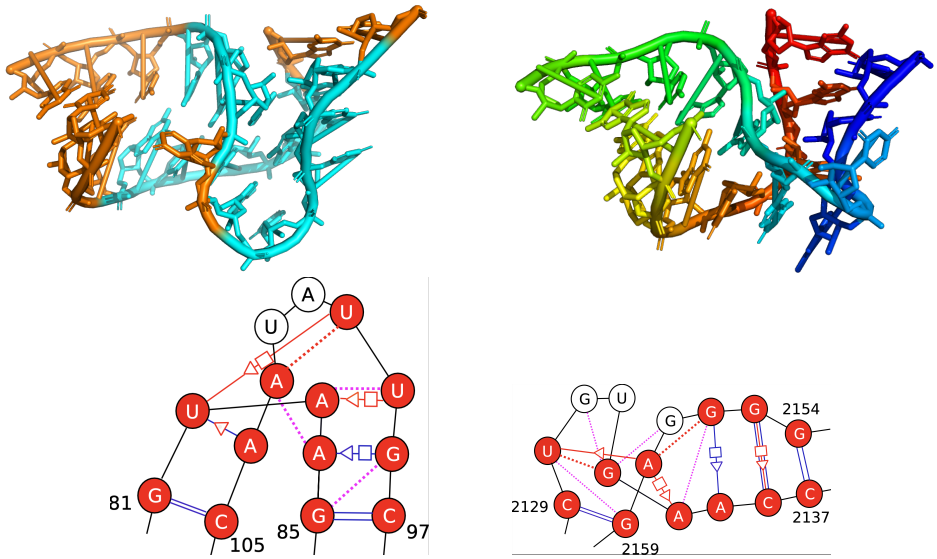


Figure 11 Other matches. 5J7L on the left and 7RQB on the right. The 3D structure on the left has IL_85931.1 highlighted in cyan, on the right each nucleotide is colored independently. In the graphs, red nodes are matched with the pattern. Blue edges are in the RNA structure and red ones are in the pattern, indicating modifications and removal. Red dashed lines are introduced "Fake edges". Magenta dashed lines indicate stackings.

growing neighborhood. By enabling per-feature biases, FuzzTree can also be calibrated to favor the sampling of graphs at a desired location in the neighborhood to favor specific types of variants (e.g., isostericity of modified edges). Letting the sampling run for longer will also mitigate the problem. More importantly, some patterns cannot be identified, particularly if an LW interaction is replaced by a backbone connection. While such cases are rare, they do exist, and additional improvement will be needed to capture them.

We evaluate our method on the Kink-Turn group, a well-known interior loop motif that induces a sharp bend in the structure and is annotated in 46 different RNA structures. The Kink-Turns are grouped in the RNA3DMotifAtlas into 12 different subgroups with varying lengths and interactions. Using only the signature graph of one subgroup, FuzzTree samples conformations of over 2/3 of all Kink-Turns and identifies all of them in 88% of RNA structures. A closer examination of the other sampled patterns reveals two previously unannotated sub-structures, each with a characteristic G-A trans-Hoogsteen-sugar interaction and a sharp local bend.

Future work to complement this should broaden the evaluation framework by testing FuzzTree on diverse RNA modules. There is also a need for new techniques to overcome pattern identification limitations and explore adaptive sampling strategies to dynamically steer the sampled neighborhood.

While FuzzTree was developed and adapted for RNA structure modules, it highlights the flexibility of multidimensional Boltzmann sampling and could be applied to other biological networks such as protein-protein interaction networks or metabolic pathways. Addressing these questions and areas for future work could lead to more comprehensive insights into complex RNA structures and other biological networks.

References

1 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, July 1995. doi:10.1145/210332.210337.

2 Olivier Bodini and Yann Ponty. Multi-dimensional Boltzmann Sampling of Languages. *Discrete Mathematics & Theoretical Computer Science*, DMTCS Proceedings vol. AM, 21st International Meeting on Probabilistic, Combinatorial, and Asymptotic Methods in the Analysis of Algorithms (AofA'10), January 2010. doi:10.46298/dmtcs.2793.

3 Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008. doi:10.1093/comjnl/bxm037.

4 Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. Introducing vf3: A new algorithm for subgraph isomorphism. In Pasquale Foggia, Cheng-Lin Liu, and Mario Vento, editors, *Graph-Based Representations in Pattern Recognition*, pages 128–139, Cham, 2017. Springer International Publishing.

5 Thomas R Cech and Joan A Steitz. The noncoding RNA revolution – Trashing old rules to forge new ones. *Cell*, 157(1):77–94, 2014.

6 G Chojnowski, T Waleń, and JM Bujnicki. RNA Bricks – A database of RNA 3D motifs and their interactions. *Nucleic Acids Research*, 42, 2013. doi:10.1093/nar/gkt1084.

7 Luigi Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26:1367–1372, November 2004. doi:10.1109/TPAMI.2004.75.

8 José Almeida Cruz and Eric Westhof. The dynamic landscapes of RNA architecture. *Cell*, 136(4):604–609, 2009.

9 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2016.

10 Rhiju Das, Rachael C Kretsch, Adam J Simpkin, Thomas Mulvaney, Phillip Pham, Ramya Rangan, Fan Bu, Ronan Keegan, Maya Topf, Daniel Rigden, et al. Assessment of three-dimensional RNA structure prediction in CASP15. *bioRxiv*, pages 2023–04, 2023.

11 Sven Findeiß, Christoph Flamm, and Yann Ponty. Rational Design of RiboNucleic Acids (Dagstuhl Seminar 22381). *Dagstuhl Reports*, 12(9):121–149, 2023. doi:10.4230/DagRep.12.9.121.

12 Nagoor Gani. 63. isomorphism on fuzzy graphs. *International Journal of Computational and Mathematical Sciences*, Vol. 2:200–206, January 2008. doi:10.13140/2.1.1873.9847.

13 M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

14 Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using NetworkX. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

15 S. Hammer, W. Wang, S. Will, and Y. Ponty. Fixed-parameter tractable sampling for rna design with multiple target structures. *BMC Bioinformatics*, 2019.

16 Lin Huang and David MJ Lilley. The kink turn, a key architectural element in RNA structure. *Journal of molecular biology*, 428(5):790–801, 2016.

17 Alpár Jüttner and Péter Madarasi. Vf2++ – An improved subgraph isomorphism algorithm. *Discrete Applied Mathematics*, 242:69–81, 2018. Computational Advances in Combinatorial Optimization. doi:10.1016/j.dam.2018.02.018.

18 Arijit Khan, Nan Li, Xifeng Yan, Ziyu Guan, Supriyo Chakraborty, and Shu Tao. Neighborhood based fast graph search in large networks. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 901–912, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1989323.1989418.

19 Daniel J Klein, T Martin Schmeing, Peter B Moore, and Thomas A Steitz. The kink-turn: a new RNA secondary structure motif. *The EMBO journal*, 20(15):4214–4221, 2001.

20 Neocles B Leontis, Aurelie Lescoute, and Eric Westhof. The building blocks and motifs of RNA architecture. *Current Opinion in Structural Biology*, 16(3):279–287, 2006. Nucleic acids/Sequences and topology. doi:10.1016/j.sbi.2006.05.009.

21 Neocles B Leontis and Eric Westhof. Geometric nomenclature and classification of RNA base pairs. *Rna*, 7(4):499–512, 2001.

22 Bin Li, Shurong Liu, Wujian Zheng, Anrui Liu, Peng Yu, Di Wu, Jie Zhou, Ping Zhang, Chang Liu, Qiao Lin, et al. RIP-PEN-seq identifies a class of kink-turn RNAs as splicing regulators. *Nature Biotechnology*, pages 1–13, 2023.

23 Dániel Marx and Michal Pilipczuk. Everything you always wanted to know about the parameterized complexity of Subgraph Isomorphism (but were afraid to ask). In Ernst W. Mayr and Natacha Portier, editors, *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 542–553, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.STACS.2014.542.

24 Carlos Oliver, Vincent Mallet, Pericles Philippopoulos, William L Hamilton, and Jérôme Waldispühl. Vernal: a tool for mining fuzzy network motifs in RNA. *Bioinformatics*, 38(4):970–976, November 2021. doi:10.1093/bioinformatics/btab768.

25 Aymeric Perchant and Isabelle Bloch. Fuzzy morphisms between graphs. *Fuzzy Sets and Systems*, 128(2):149–168, 2002. doi:10.1016/S0165-0114(01)00131-2.

26 Anton I. Petrov, Craig L. Zirbel, and Neocles B. Leontis. Automated classification of rna 3d motifs and the rna 3d motif atlas. *RNA*, 2013.

27 Vladimir Reinharz, Antoine Soulé, Eric Westhof, Jérôme Waldispühl, and Alain Denise. Mining for recurrent long-range interactions in RNA structures reveals embedded hierarchies in network families. *Nucleic Acids Research*, 46(8):3841–3851, March 2018. doi:10.1093/nar/gky197.

28 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 Ben Raphael and Jijun Tang, editors, *Algorithms in Bioinformatics*, pages 149–164, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

29 Michael Sarver, Craig L Zirbel, Jesse Stombaugh, Ali Mokdad, and Neocles B Leontis. FR3D: finding local and composite recurrent structural motifs in RNA 3D structures. *Journal of mathematical biology*, 56:215–252, 2008.

30 Antoine Soulé, Vladimir Reinharz, Roman Sarrazin-Gendron, Alain Denise, and Jérôme Waldispühl. Finding recurrent RNA structural networks with fast maximal common subgraphs of edge-colored graphs. *PLoS computational biology*, 17(5):e1008990, 2021.

31 Jesse Stombaugh, Craig L. Zirbel, Eric Westhof, and Neocles B. Leontis. Frequency and isostericity of RNA base pairs. *Nucleic Acids Research*, 37(7):2294–2312, February 2009. doi:10.1093/nar/gkp011.

32 Bernhard C Thiel, Irene K Beckmann, Peter Kerpedjiev, and Ivo L Hofacker. 3D based on 2D: Calculating helix angles and stacking patterns using forgi 2.0, an RNA Python library centered on secondary structure elements. *F1000Research*, 8, 2019.

33 Hua-Ting Yao, Yann Ponty, and Sebastian Will. Developing complex rna design applications in the infrared framework. *RNA Folding – Methods and Protocols*, 2022.

34 M. Zahran, C. Sevim Bayrak, S. Elmetwaly, and T. Schlick. RAG-3D: a search tool for RNA 3D substructures. Nucleic acids research. *Nucleic Acids Research*, 43(19):9474–9488, 2015. doi:10.1093/nar/gkv823.

A Supplementary material

A.1 About the sampling process

Sampling from a Multidimensional distribution in our case can be written formally as below:

▶ **Definition 3** (Boltzmann distribution/Partition function). *In the* ***Multidimensional Boltzmann Distribution****, the probability to sample graph G, subgraph of G_T with features F_1, ... F_m (that embody neighborhoods differences of G_P for mapped graph G in G_T) of respective weights w_1, ... w_m (that we can write more simply $w = (w_1, ...w_m)$) is proportional to its* ***energy****:*

$$\mathbb{P}_{G_P,G_T}(G \mid w) = \frac{\prod_{i=1}^{m} e^{-\beta w_i . F_i(G)}}{\mathcal{Z}_w}$$

where $\beta := (RT)^{-1}$, R is the Gas constant, T the temperature in Kelvin, and $\mathcal{Z}_w$ denotes the ***partition function***

$$\mathcal{Z}_w = \sum_{G \subseteq G_T} \prod_{i=1}^{m} e^{-\beta w_i . F_i(G)}$$

We can forget about the β contribution as we can rewrite the weight $w_i' = \beta w_i$. The weights w_i are values chosen or tuned by us.

Tuning the weights is done by fixing a mean T^{F*} (and T^F threshold) for each type of neighborhood. We can then tune the weight $w(F_i)$ to give more "importance" that will favor value around T^{F*}. In practice, when a feature for a neighborhood varies greatly between instances, it means that this neighborhood is strongly relevant to distinguish the different matches. It gives us an incentive to modify its weight accordingly. To do so, instead of choosing weights manually, we solve the following problem:

$$min_w \sum_{i=1}^{m} |\mathbb{E}[F_i|w] - F_i^*|$$

This problem is known to be convex. We used so convex optimization method. Further details about this problem, including the proof of convexity, are addressed in [15].

A.2 Computation of the partition function using dynamic programming

A.2.1 Definitions

First, we introduce the formal definition of the treewidth, we also depict what is a nice tree decomposition (NTD) as it allows a simpler search during the dynamic programming procedure. NTD implies no additional cost because an NTD has at most a size $n = |G_T|$.

▶ **Definition 4** (Tree Decomposition (TD)). *Given a graph $G = (V, E)$, a tree decomposition of G is a tree T, whose nodes are bags $Y_1...Y_t$ such that: (definition from Bodlander et al [3])*

1. $V \subset \bigcup_{i=1}^{t} Y_i$
2. $\forall (u, v) \in E, \exists i \in [\![1, t]\!], (u \in Y_i) \cap (v \in Y_i)$
3. $\forall u \in V, \{u | u \in Y_i\}$ *is a subtree of* T.

▶ **Definition 5** (Nice Tree Decomposition). *A tree decomposition T of $G = (V, E)$ is said "nice" if each bags Y_i has one of the three following forms:*

- *Introduce: Node Y_i has exactly one child of index c in T and $Y_i = Y_c \cup \{v\}$*
- *Forget: Node Y_i has exactly one child of index c in T and $Y_c = Y_i \cup \{v\}$*
- *Join: Node Y_i has exactly two children of indices c_1 and c_2 in T and $Y_i = Y_{c_1} = Y_{c_2}$*

▶ **Definition 6** (Treewidth). *The treewidth ϕ of a graph G is defined as the biggest bag of the "best" tree decomposition of G:*

$$\phi = \min_{tree\ dec.\ T\ of\ G} max_{Y_i \in T} |Y_i| - 1.$$

A.2.2 Dynamic programming solution

We now address the computation of the partition function [15] from 3 through a dynamic programming procedure on the nice tree decomposition of G_T.

It is a bottom-up dynamic procedure (from leaves to the root) that relies on the following different equations depending on the type of the node Y_i in the nice tree decomposition T. We denote:

- The set of neighborhood thresholds: $F = \left(T^L, T^E, T^G\right)$.
- M_i, **partial mapping** at node Y_i of T.
- The **separator node** of Y_i, $\text{sep}(Y_i)$ chosen as the first element of the set S:

  $$S = \{x \in Y_i \mid x \notin Y' \text{ with } Y' \text{ a children of } Y_i\}.$$

  We can point out that, with a nice tree decomposition, there exists only a unique choice for this node and the set S is reduced to a singleton.
- Given a partial mapping M_i, we introduce the following Boolean condition to map each contribution to a single bag and avoid multiple computations of it:

  $$C(u_1, u_2, Y_i, M_i) = (u_1 = \text{sep}(Y_i) \cap M_i(u_2) \neq \emptyset) \cup (u_2 = \text{sep}(Y_i) \cap M_i(u_1) \neq \emptyset).$$

  From this we introduce $\Delta(\cdot)$ to denote the global contribution

  $$\Delta(M_i', G_T, Y_i, T^F) = \left\{d^F_{G_T}(u_1, u_2, M_i') \mid C(u_1, u_2, Y_i, M_i') \text{ is True}\right\}.$$

We fill the dynamic programming table P that stores the partial computation of the partition function with equations:

- Forget Node Y_i with child Y':

  $$P[Y_i; M_i] = P[Y'; M_i]$$

- Introduction Node, creating vertex $s := \text{sep}(Y_i) \in V_P$ having child Y':

  $$P[Y_i; M_i] = \sum_{v \in D(s|M_i)} P[Y'; M_i \cup (s \leftarrow v)] \times \prod_{\substack{T^F \in F \\ \delta \in \Delta(M_i \cup (s \leftarrow v), G_T, Y_i, T^F)}} e^{-\mu . w(T^F) . \delta}$$

  where $D(v \mid M)$ denotes the set of admissible mappings for $v \in V_P$, consistent with prior assignment M, such that:

  $$D(v \mid M) := \begin{cases} V_T & \text{if } M = \varnothing \\ \bigcap\limits_{\substack{u \in M \\ \text{s.t. } u \prec v}} \{x \in V_T \mid M(u) \prec x\} \bigcap\limits_{\substack{u \in M \\ \text{s.t. } v \prec u}} \{x \in V_T \mid x \prec M(u)\} & \text{otherwise.} \end{cases}$$

- Join Node:

$$P[Y_i; M_i] = \prod_{Y' \in children(Y_i)} P[Y'; M_i]$$

The backtracking step to retrieve the value of probability for each graph (and so the whole Boltzmann distribution as introduced in 3) uses the same type of equations but going from top to bottom: a number is drawn at each node to know if we have to add a value for current mapping, given the partial partition function computed at each step of the forward procedure. Both the forward and backward steps are currently known procedures that have been studied and automatized in a framework named `Infrared`. [33], which has the advantage to be quite permissive about the definition of the neighborhood cumulative differences.

A.3 Choice on technical parameters

For the choice of the radius R for creating slices of target graph G_T, given an extracted graph G from G_T centered in nucleotide c, we first defined $R(G) = \min_{j \in G} \mathsf{GEO}(j, c)$. To be exhaustive with our search, we must ensure that every G from G_T is extracted with a radius at least equals to $R(G_P) + D_{\text{gap}}$ as it ensures that we have enough "space" to make G_P fit in G even if some gaps occur. It is due to these gaps that we need to add D_{gap} in R. It embodies the specific case where the gap would have increased the length of the motif to search in G_T in a single direction by putting gaps one after the other. Due to the rarity of this case, we choose, in the tests, to use a smaller radius equal to $R(G_P) + \frac{D_{\text{gap}}}{4}$. The only taken risk here is to miss some patterns, but it is more convenient to favor time convergence as the pathological case on gaps evoked above is not one that we would like to target.

We also choose to use a timeout equal to 2000 seconds for the convergence of our algorithm on each extracted graph. Here again, the only risk is to miss some additional patterns. Nonetheless, all these limitations only mean that our current results can probably be slightly better regarding expressiveness, which means that somebody with more computational resources could use this tool and wait for even better performances.

A.4 Time results on Narval and Beluga clusters for FuzzTree

For this paper, computations were done on the Narval cluster and the Beluga cluster of the Digital Research Alliance of Canada. Each used node on Narval is made of 64 cores with 2 CPUs AMD Rome 7532 @ 2.40 GHz. Each used node on Beluga is made of 40 cores with 2 CPUs Intel Gold 6148 Skylake @ 2.4 GHz. Multiprocessing was used simply by separating the computations by chains of the same RNA and next, when relevant, by slices identified in these RNA chains.

Some time results for computation of the FuzzTree method, by requesting one motif on each RNA chain where Kink-Turns are known, are available in Fig. 12. The time of computation is large but it is something expected with the XP theoretical complexity. However, one can notice that in practice the treewidth of the selected pattern is equal to 2 which allows a complexity in $O(n^3)$. No true time discrepancy appears between the computation without near edges and the one with. On large graphs, due to the slicing, the time of computation is reduced, but such reduction is not perfect as slicing computation is still quite redundant: multiple graphs cover sometimes the same portion of the Kink-Turn.

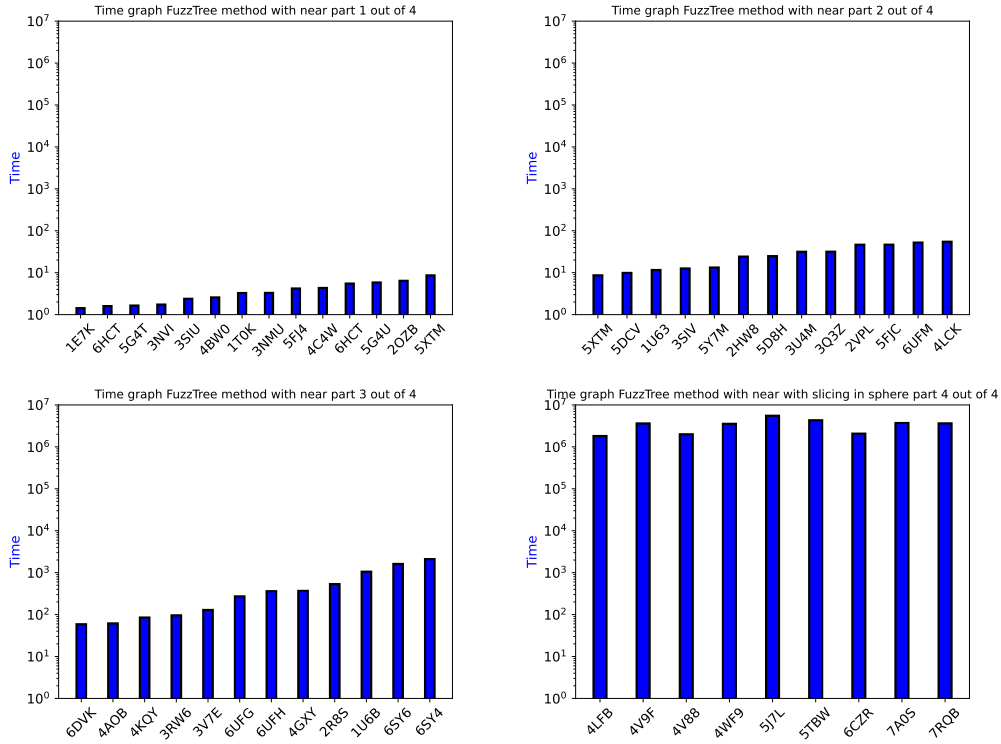


Figure 12 Time graph of the FuzzTree method on each group of studied RNA chains. On the Beluga cluster, computations were done on 1 processor for small RNAs (less than 500 nucleotides, which corresponds to the three first graphs) and on 40 processors for large RNAs (more than 500 nucleotides, which corresponds to the fourth graph). In that case, the depicted time is the sum of each time consumed for each processor.

Balancing Minimum Free Energy and Codon Adaptation Index for Pareto Optimal RNA Design

Xinyu Gu
Department of Computer Science, University of Illinois Urbana-Champaign, Urbana, IL, USA

Yuanyuan Qi
Department of Computer Science, University of Illinois Urbana-Champaign, Urbana, IL, USA

Mohammed El-Kebir
Department of Computer Science, University of Illinois Urbana-Champaign, Urbana, IL, USA

Abstract

The problem of designing an RNA sequence $\mathbf{v}$ that encodes for a given target protein $\mathbf{w}$ plays an important role in messenger RNA (mRNA) vaccine design. Due to codon degeneracy, there exist exponentially many RNA sequences for a single target protein. These candidate RNA sequences may adopt different secondary structure conformations with varying minimum free energy (MFE), affecting their thermodynamic stability and consequently mRNA half-life. In addition, species-specific codon usage bias, as measured by the codon adaptation index (CAI), also plays an essential role in translation efficiency. While previous works have focused on optimizing either MFE or CAI, more recent works have shown the merits of optimizing both objectives. Importantly, there is a trade-off between MFE and CAI, i.e. optimizing one objective is at the expense of the other. Here, we formulate the Pareto Optimal RNA Design problem, seeking the set of Pareto optimal solutions for which no other solution exists that is better in terms of both MFE and CAI. We introduce DERNA (DEsign RNA), which uses the weighted sum method to enumerate the Pareto front by optimizing convex combinations of both objectives. DERNA uses dynamic programming to solve each convex combination in $\mathcal{O}(|\mathbf{w}|^3)$ time and $\mathcal{O}(|\mathbf{w}|^2)$ space. Compared to a previous approach that only optimizes MFE, we show on a benchmark dataset that DERNA obtains solutions with identical MFE but superior CAI. Additionally, we show that DERNA matches the performance in terms of solution quality of LinearDesign, a recent approach that similarly seeks to balance MFE and CAI. Finally, we demonstrate our method's potential for mRNA vaccine design using SARS-CoV-2 spike as the target protein.

2012 ACM Subject Classification Applied computing → Computational biology

Keywords and phrases Multi-objective optimization, dynamic programming, RNA sequence design, reverse translation, mRNA vaccine design

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.21

Supplementary Material *Software (Source code)*: `https://github.com/elkebir-group/derna` archived at `swh:1:dir:d180327fb14fede1d3fcf822df273b4dc12c6069`

Funding *Mohammed El-Kebir*: National Science Foundation award number CCF 2046488

1 Introduction

With the emergence of the COVID-19 pandemic, messenger RNA (mRNA) vaccines have garnered significant attention due to their effectiveness in combating the disease [13, 17]. However, there remain significant challenges in the delivery [4] as well as the *in vitro* and *in vivo* stability of mRNA-based vaccines and therapeutics [28]. Importantly, due to codon degeneracy with $4^3 = 64$ codons encoding for 20 distinct amino acids as well as translation termination signals, there are exponentially many RNA sequences $\mathbf{v}$ for a single target protein $\mathbf{w}$. Synonymous codon choice impacts translational efficiency and mRNA stability in two interrelated ways. First, a subset of "optimal" codons occur at a higher frequency in

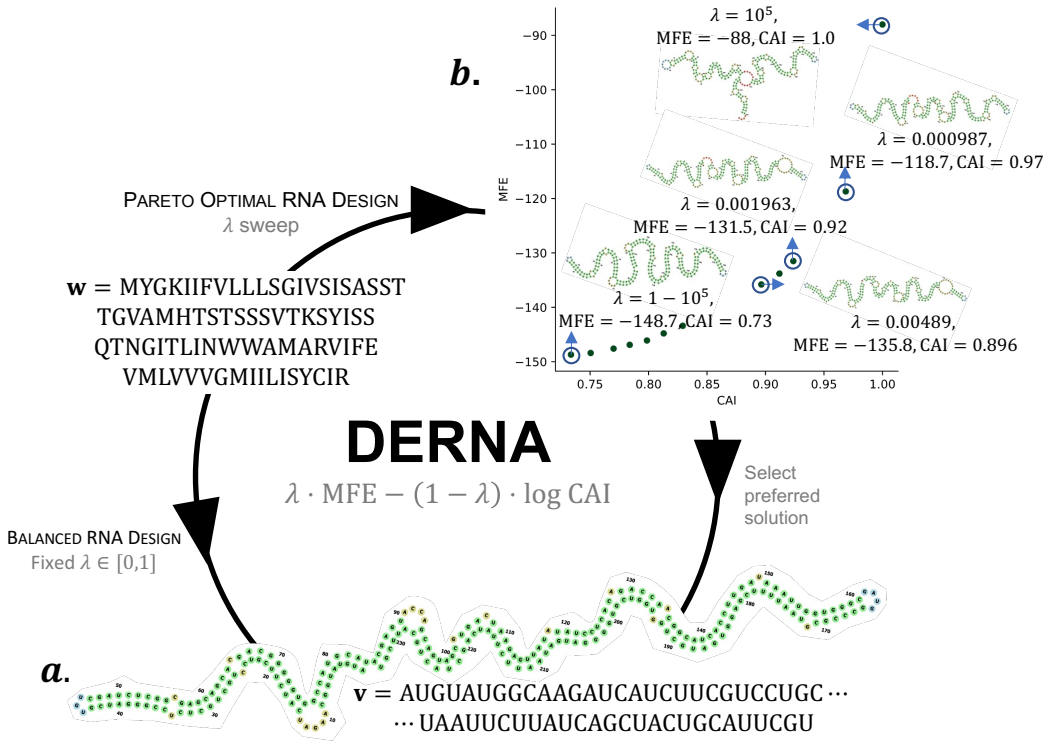


Figure 1 DERNA seeks Pareto optimal RNA sequences v for a target protein w, balancing the minimum free energy (MFE) and codon adaptation index (CAI). (a) For the BALANCED RNA DESIGN (BRD) problem, DERNA takes as input the parameter $\lambda \in [0, 1]$ and returns the RNA sequence $\mathbf{v}$ whose corresponding secondary structure P minimizes $\lambda \cdot \mathrm{MFE}(\mathbf{v}, P) - (1 - \lambda) \cdot \overline{\mathrm{CAI}}(\mathbf{v}, \mathbf{w})$. (b) For the PARETO OPTIMAL RNA DESIGN problem, DERNA performs a systematic sweep on λ, solving multiple BRD instances and returning a set of Pareto optimal solutions $(\mathbf{v}, P)$.

highly-expressed genes [8] and "non-optimal" codons lead to increased ribosomal pausing and decreased mRNA half-life [19, 29]. Second, depending on codon choice, each candidate RNA sequence folds into a distinct *secondary structure* or conformation, affecting its thermodynamic stability and consequently mRNA half-life. Recent studies have demonstrated the importance of both factors, showing that increased secondary structure as well as optimal codon usage lead to increased protein expression [16, 24]. This leads to the following key question of this paper. How does one identify RNA sequences that optimize both criteria?

Different organisms and even different genes within the same organism can have distinct codon usage patterns. The *codon adaptation index* (CAI) is a measure that quantifies the degree of codon usage bias in a protein coding sequence relative to a reference set of highly-expressed genes [22]. The reference set is often chosen based on the assumption that these genes have evolved to use codons that are mostly efficiently translated by the ribosome. Thus, an RNA sequence with high CAI is expected to have higher rates of translation [8, 19, 29]. Specifically, for a reference gene set, we are given the relative frequencies $g(\mathbf{x})$ of each codon $\mathbf{x}$ in the gene set. Then, the CAI of an RNA sequence $\mathbf{v}$ is the geometric mean of the ratios $g(\mathbf{x}) / \max_{\mathbf{y} \in S(\mathbf{x})} g(\mathbf{y})$ of each codon $\mathbf{x}$ vs. the maximum relative frequency of a synonymous codon $\mathbf{y} \in S(\mathbf{x})$ (see Equation (1)). RNA sequences that are composed of only optimal

codons with maximum relative frequencies have by definition a CAI of 1. In our setting, it is trivial to identify such an RNA sequence with CAI equal to 1 by simply choosing the codon with maximum relative frequency for each amino acid of the target protein. However, such an RNA sequence with optimal CAI may exhibit suboptimal amounts of secondary structure (Figure 1).

RNA molecules adopt secondary structure and three-dimensional conformations as the nucleotides within the RNA molecule and the surrounding solvent interact with each other. When an RNA molecule folds into its conformation, it forms base-pairing interactions between nucleotides that result in the lowest possible free energy [7]. This conformation is said to have the *minimum free energy* (MFE). In general, an RNA molecule with a lower MFE is more likely to be stable and maintain its integrity over time, whereas an RNA molecule with a higher MFE is more likely to be degraded. Thermodynamic stability is an important factor in identifying the most stable RNA sequences that are likely to be functional and efficient in producing a target protein [16, 24]. Zuker and Stiegler [32] introduced a dynamic programming algorithm to identify the conformation of RNA molecules with minimum free energy from a given RNA sequence $\mathbf{v}$. This approach was later extended independently by Terai et al. [23] and Cohen and Skiena [1] to identify a RNA sequence $\mathbf{v}$ and corresponding secondary structure with overall minimum MFE for a given target protein sequence $\mathbf{w}$. However, an RNA sequence with optimal MFE may have suboptimal CAI (Figure 1).

Recognizing the importance of examining both CAI and MFE, Zhang et al. [31] introduced LinearDesign, which uses stochastic context-free grammars and deterministic finite automata and applies a beam search heuristic to optimize $\text{MFE} + \lambda_{\text{LD}} \log \text{CAI}$ where λ_{LD} is a user-specified parameter.

In this work, we model the trade-off between CAI and MFE as a multi-objective optimization problem. That is, we introduce the Pareto Optimal RNA Design problem, seeking the set of *Pareto optimal solutions* for which no other solution exists that is better in terms of both MFE and CAI (Figure 1). We use the weighted sum method [30] to enumerate the Pareto front by optimizing convex combinations of both objectives – leading to the Balanced RNA Design problem (Figure 1). Our resulting algorithm, DERNA (DEsign RNA), extends the Zuker and Stiegler dynamic programming approach [32] to solve each convex combination in $\mathcal{O}(|\mathbf{w}|^3)$ time and $\mathcal{O}(|\mathbf{w}|^2)$ space. Unlike LinearDesign, where key functions are closed source, DERNA is fully open source with all code and functionality available to the user under a permissive license. We show on a benchmark dataset that DERNA obtains solutions with identical MFE but superior CAI compared to CDSfold [23]. Additionally, we show that DERNA matches LinearDesign's performance in terms of solution quality. Finally, we run our method on the SARS-CoV-2 spike protein and demonstrate its potential for mRNA vaccine design.

2 Problem Statement

A secondary structure for an RNA sequence with length n is a set of ordered base pairings $(i, j) \in [n] \times [n]$ such that each base is paired with at most one other base and there are no crossings base pairings (also known as pseudoknots). More formally, we define a secondary structure as follows.

▶ **Definition 1.** *A set $P \subseteq [n] \times [n]$ of base pairings is a* secondary structure *provided (i) for each base pairing $(i, j) \in P$ it holds that $i < j$, and for any two base pairings $(i, j), (i', j') \in P$ it holds that (ii) $i = i'$ if and only if $j = j'$ and (iii) if $i < i' < j$ then $i < i' < j' < j$.*

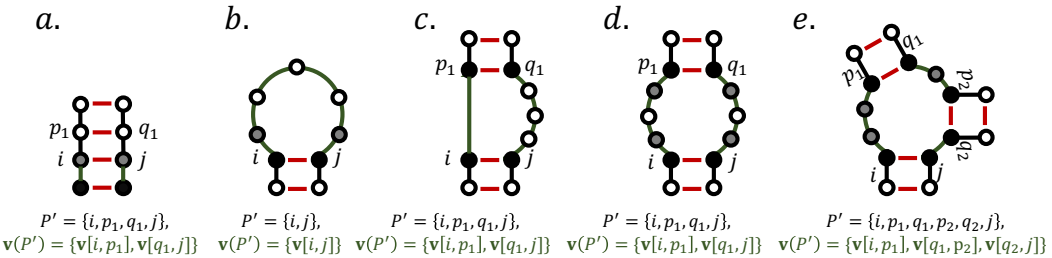


Figure 2 There are five secondary structure elements. (a) Stacking. (b) Hairpin loop. (c) Bulge loop. (d) Internal loop. (e) Multi-branch loop. Each structural element is defined by a unique set P' of nucleotide indices involved in base pairings (indicated in red). In addition, each structural element corresponds to a unique face of a planar embedding comprised of subsequences $\mathbf{v}(P')$ (indicated in green). Nucleotides next to the base pairings (indicated in gray) are involved in providing a free energy contribution to some structural components.

Following Zuker and Stiegler [32], a secondary structure P can be decomposed into several secondary structure elements, such that the free energy of the secondary structure P is the sum of the free energies contributed by each secondary structure element, defined as follows.

▶ **Definition 2.** *A subset* $P' = \{i, p_1, q_1, \ldots, p_k, q_k, j\} \subseteq [n]$ *of bases is a* secondary structure element *of* P *provided (i)* $i < p_1 < q_1 < \ldots < p_k < q_k < j$*, (ii)* $(i, j) \in P$*, (iii)* $(p_l, q_l) \in P$ *for each* $l \in [k]$ *and (iv) there exists no base pairing* $(i', j') \in P$ *such that* $i < i', j' < p_1 < j$*;* $i < q_k < i', j' < j$*; or* $q_l < i', j' < p_{l+1}$ *for all* $l \in \{1, \ldots, k-1\}$*.*

Alternatively, each secondary structure element corresponds to a unique face of a planar embedding of the secondary structure. Denoting the subsequence $v_i, \ldots, v_j$ by $\mathbf{v}[i, j]$, the subsequences that make up each face or secondary structure element are defined as follows.

▶ **Definition 3.** *A secondary structure element* $P' = \{i, p_1, q_1, \ldots, p_k, q_k, j\} \subseteq [n]$ *is comprised of* RNA subsequences $\mathbf{v}(P') = \{\mathbf{v}[i, p_1], \mathbf{v}[q_1, p_2], \ldots, \mathbf{v}[q_{k-1}, p_k], \mathbf{v}[q_k, j]\}$*. For* $k = 0$*, i.e.* $P' = \{i, j\}$*, the corresponding RNA subsequence* $\mathbf{v}(P')$ *equals* $\{\mathbf{v}[i, j]\}$*.*

Depending on the topology, we distinguish five types of secondary structure elements. In the simplest case, base pairing (i, j) is immediately followed by the pairing $(i+1, j-1)$, which is called a stacking element.

▶ **Definition 4.** *A secondary structure element* P' *of the form* $\{i, i+1, j-1, j\}$ *is a* stacking element*.*

Base pairing (i, j) forms a hairpin loop if there are no other base pairings involving bases $i+1, \ldots, j-1$.

▶ **Definition 5.** *A secondary structure element* P' *of the form* $\{i, j\}$ *is a* hairpin loop*.*

If base pairing (i, j) does not form a stacking element and there are other base pairings (i', j') occurring between i and j then (i, j) forms an interior loop element. We distinguish three types of interior loop elements: (i) a bulge loop, (ii) an internal loop and (iii) a multi-branch loop. The first two types correspond to a interior loop element enclosing a single base pairing (p_1, q_1), i.e. $k = 1$.

In a *bulge loop*, the enclosing base pairing is contiguous to either i or j, i.e. $p_1 = i + 1$ or $q_1 = j - 1$.

▶ **Definition 6.** *A secondary structure element P' of the form $\{i, p_1, q_1, j\}$ is a* bulge loop *provided (i) $(p_1, q_1) \neq (i+1, j-1)$ and (ii) $p_1 = i+1$ or $q_1 = j-1$.*

On the other hand, in an *internal loop* the enclosing base pairing is not contiguous, i.e. $p_1 > i+1$ and $q_1 < j-1$.

▶ **Definition 7.** *A secondary structure element P' of the form $\{i, p_1, q_1, j\}$ is an* internal loop *provided $i+1 < p_1 < q_1 < j-1$.*

If the interior loop element encloses more than one base pairing, i.e. $k > 1$, then we call the secondary structure element a *multi-branch loop*.

▶ **Definition 8.** *A secondary structure element P' of the form $\{i, p_1, q_1, \ldots, p_k, q_k, j\}$ is a* multi-branch loop *provided $k > 1$.*

As mentioned, we define the minimum free energy $\text{MFE}(\mathbf{v}, P) = \sum_{(i,j)\in P} \text{MFE}(\mathbf{v}, P, (i,j))$ of an RNA sequence $\mathbf{v}$ as the sum of the minimum free energies $\text{MFE}(\mathbf{v}, P, (i,j))$ of the secondary structure elements induced by each base pairing $(i,j) \in P$.

▶ **Definition 9.** *The* minimum free energy $\text{MFE}(\mathbf{v}, P)$ *of secondary structure P of RNA sequence $\mathbf{v}$ equals $\sum_{(i,j)\in P} \text{MFE}(\mathbf{v}, P, (i,j))$ where $\text{MFE}(\mathbf{v}, P, (i,j))$ is the contribution of the secondary structure element induced by a base pairing $(i,j) \in P$ defined as*

$$\text{MFE}(\mathbf{v}, P, (i,j)) = \begin{cases} f_{\text{s}}(\mathbf{v}(P')), & \text{if } P' = \{i, i+1, j-1, j\} \text{ is a stacking element,} \\ f_{\text{h}}(\mathbf{v}(P')), & \text{if } P' = \{i, j\} \text{ is a hairpin,} \\ f_{\text{b}}(\mathbf{v}(P')), & \text{if } P' = \{i, p_1, q_1, j\} \text{ is a bulge loop,} \\ f_{\text{i}}(\mathbf{v}(P')), & \text{if } P' = \{i, p_1, q_1, j\} \text{ is an internal loop,} \\ f_{\text{m}}(\mathbf{v}(P')), & \text{if } P' = \{i, p_1, q_1, \ldots, p_k, q_k, j\} \text{ is a multi-branch loop.} \end{cases}$$

The actual definitions of f_s, f_h, f_b, f_i and f_m depend on the used energy model. Briefly, in the widely used Turner energy model [26], the stacking energy value f_s is computed using a lookup table indexed by the four nucleotides comprising the base pairings $(i,j), (i+1, j-1)$. Similarly, the hairpin energy value f_h is a function of the four nucleotides $v_i, v_{i+1}, v_{j-1}, v_j$ and the length $j-i+1$ of the hairpin loop. For a bulge loop, the energy value f_b is a function of the four nucleotides in the base pairings $(i,j), (p_1, q_1)$ and the number of unpaired nucleotides in the loop $\mathbf{v}(\{i, p_1, q_1, j\})$. For an internal loop, the energy value f_h is a function of the eight nucleotides $v_i, v_{i+1}, v_{p_1-1}, v_{p_1}, v_{q_1}, v_{q_1+1}, v_{j-1}, v_j$ surrounding the base pairings $(i,j), (p_1, q_1)$ as well as the number of unpaired nucleotides in the loop $\mathbf{v}(\{i, p_1, q_1, j\})$. Finally, the energy value f_m is a function of the number k of base pairings enclosed in the multi-loop, the four nucleotides surrounding each base pairing and the number of unpaired nucleotides in the loop $\mathbf{v}(\{i, p_1, q_1, \ldots, p_k, q_k, j\})$. We refer to Appendix A.1 for more details.

The classical RNA Secondary Structure Prediction problem is defined as follows.

▶ **Problem 1** (RNA Secondary Structure Prediction (RSSP))**.** *Given an RNA sequence $\mathbf{v} \in \Sigma_{\text{rna}}^n$, find a secondary structure P such that $\text{MFE}(\mathbf{v}, P)$ is minimized.*

This problem can be solved in $O(n^3)$ time using the Zuker algorithm [32]. In this work we are interested in a reverse translation variant of the problem. That is, given a protein sequence $\mathbf{w} \in \Sigma_{\text{prot}}^m$ where Σ_{prot} is the set of 20 amino acids, we seek a corresponding RNA sequence $\mathbf{v} \in \Sigma_{\text{rna}}^{3m}$ that translates into $\mathbf{w}$. To that end, we use the function $S : \Sigma_{\text{prot}} \to \mathcal{P}(\Sigma_{\text{rna}}^3)$ such that $S(\alpha)$ is the set of codons that encode amino acid $\alpha \in \Sigma_{\text{prot}}$. We define $\sigma(a, s) = 3(a-1)+s$ to indicate the RNA sequence index corresponding to protein sequence index $a \in [m]$ and codon index $s \in \{1, 2, 3\}$.

▶ **Definition 10.** *RNA sequence* $\mathbf{v} \in \Sigma_{\text{rna}}^n$ encodes *for protein sequence* $\mathbf{w} \in \Sigma_{\text{prot}}^m$ *provided (i)* $|\mathbf{v}| = n = 3m = 3|\mathbf{w}|$ *and (ii)* $\mathbf{v}[\sigma(a,1), \sigma(a,3)] \in S(w_a)$ *for all protein indices* $a \in [m]$.

Rather than only considering the minimum free energy $\text{MFE}(\mathbf{v}, P)$, we also take species-specific codon usage bias into account. In other words, given species-specific relative codon frequencies $g : \Sigma_{\text{rna}}^3 \to [0,1]$, we compute the codon adaptation index $\text{CAI}(\mathbf{v}, \mathbf{w})$ defined as follows.

▶ **Definition 11.** *The* codon adaptation index $\text{CAI}(\mathbf{v}, \mathbf{w})$ *of RNA sequence* $\mathbf{v}$ *that translates into protein sequence* $\mathbf{w}$ *is defined as*

$$\text{CAI}(\mathbf{v}, \mathbf{w}) = \sqrt[m]{\prod_{a=1}^{m} \frac{g(\mathbf{v}[\sigma(a,1), \sigma(a,3)])}{\max_{\mathbf{x} \in S(w_a)} g(\mathbf{x})}} \tag{1}$$

where $g(\mathbf{x})$ *is the* species-specific relative frequency *of codon* $\mathbf{x} \in \Sigma_{\text{rna}}^3$ *such that* $g(\mathbf{x}) \geq 0$ *for all codons* $\mathbf{x}$ *and* $\sum_{\mathbf{x} \in \Sigma_{\text{rna}}^3} g(\mathbf{x}) = 1$.

The CAI ranges from 0 to 1, where a value of 1 indicates that for each amino acid w_a the maximum frequency codon $\arg\max_{\mathbf{x} \in S(w_a)} g(\mathbf{x})$ is used [22]. Thus, given a target protein sequence $\mathbf{w}$, there are two competing objective functions; we seek a corresponding RNA sequence $\mathbf{v}$ and secondary structure P that simultaneously minimizes $\text{MFE}(\mathbf{v}, P)$ and maximizes $\text{CAI}(\mathbf{v}, \mathbf{w})$. Equivalently, rather than maximizing $\text{CAI}(\mathbf{v}, \mathbf{w})$, we maximize $\overline{\text{CAI}}(\mathbf{v}, \mathbf{w})$ defined as

$$\text{CAI}(\mathbf{v}, \mathbf{w}) = \sqrt[m]{\prod_{a=1}^{m} \frac{g(\mathbf{v}[\sigma(a,1), \sigma(a,3)])}{\max_{\mathbf{x} \in S(w_a)} g(\mathbf{x})}} \propto \sum_{a=1}^{m} \log \frac{g(\mathbf{v}[\sigma(a,1), \sigma(a,3)])}{\max_{\mathbf{x} \in S(w_a)} g(\mathbf{x})} = \overline{\text{CAI}}(\mathbf{v}, \mathbf{w}). \tag{2}$$

We model the trade-off between MFE and CAI by introducing a parameter $\lambda \in [0,1]$ and minimizing a convex combination of $\text{MFE}(\mathbf{v}, P)$ and $-\overline{\text{CAI}}(\mathbf{v}, \mathbf{w})$.

▶ **Problem 2** (Balanced RNA Design (BRD)). *Given a protein sequence* $\mathbf{w} \in \Sigma_{\text{prot}}^m$ *and parameter* $\lambda \in [0,1]$*, find an RNA sequence* $\mathbf{v} \in \Sigma_{\text{rna}}^{3m}$ *with secondary structure* P *such that (i)* $\mathbf{v}$ *encodes for* $\mathbf{w}$ *and (ii) solution* $(\mathbf{v}, P)$ *minimizes* $\lambda \cdot \text{MFE}(\mathbf{v}, P) - (1-\lambda) \cdot \overline{\text{CAI}}(\mathbf{v}, \mathbf{w})$.

We say that a solution $(\mathbf{v}, P)$ is *Pareto optimal* if $(\mathbf{v}, P)$ is better than all other feasible solutions in at least one of the two objectives. In other words, there does not exist another solution $(\mathbf{v}', P')$ that is better in both objectives, or equal in one objective and better in the other. In our final problem, we seek all Pareto optimal RNA sequences $\mathbf{v}$.

▶ **Problem 3** (Pareto Optimal RNA Design (PORD)). *Given a protein sequence* $\mathbf{w} \in \Sigma_{\text{prot}}^m$*, enumerate all RNA sequences* $\mathbf{v} \in \Sigma_{\text{rna}}^{3m}$ *each with a secondary structure* P *such that (i)* $\mathbf{v}$ *encodes for* $\mathbf{w}$ *and (ii)* $(\mathbf{v}, P)$ *is Pareto optimal w.r.t. to* $\text{MFE}(\mathbf{v}, P)$ *and* $\text{CAI}(\mathbf{v}, \mathbf{w})$.

3 Methods

3.1 RNA Design with Fixed λ

In the Balanced RNA Design problem (Problem 2), we are given a protein sequence $\mathbf{w} \in \Sigma_{\text{prot}}^m$ and parameter $\lambda \in [0,1]$ that models the trade-off between MFE and CAI. In this section, we show how to solve this problem using dynamic programming. Specifically, for protein sequence indices $a, b \in [m]$, codon indices $s, t \in \{1,2,3\}$, codons $\mathbf{x} \in S(w_a)$ and

$\mathbf{y} \in S(w_b)$, $O[a][b][s][t][\mathbf{x}][\mathbf{y}]$ is the minimum objective value when solving a problem instance restricted to RNA sequence $\mathbf{v}[\sigma(a,s), \sigma(b,t)]$ such that codons $\mathbf{x}$ and $\mathbf{y}$ are used to encode amino acids w_a and w_b, respectively. Using $O[a][b][s][t][\mathbf{x}][\mathbf{y}]$, we express the objective value of an optimal solution as

$$\min_{\mathbf{x} \in S(w_1), \mathbf{y} \in S(w_m)} O[1][m][1][3][\mathbf{x}][\mathbf{y}]. \tag{3}$$

To see why this is the case, observe that $O[1][m][1][3][\mathbf{x}][\mathbf{y}]$ equals the minimum objective value for the complete RNA sequence $\mathbf{v}[\sigma(1,1), \sigma(m,3)] = \mathbf{v}[1, 3m] = \mathbf{v}$ restricted to using codons $\mathbf{x}$ and $\mathbf{y}$ for amino acid w_1 and w_m, respectively. Thus, the overall minimum objective value is obtained for the codon pair $(\mathbf{x}, \mathbf{y}) \in S(w_1) \times S(w_m)$ that minimizes $O[1][m][1][3][\mathbf{x}][\mathbf{y}]$.

Let $\Gamma = \{(A,U), (U,A), (G,C), (C,G), (G,U), (U,G)\}$ be the set of allowed base pairings in the Turner energy model [14]. To express the contribution of the CAI, we introduce the shorthand $\bar{g}(w, \mathbf{x}) = \log(g(\mathbf{x}) / \max_{\mathbf{y} \in S(w)} g(\mathbf{y}))$ such that $\overline{\text{CAI}}(\mathbf{v}, \mathbf{w}) = \sum_{a=1}^{m} \bar{g}(w_a, \mathbf{v}[\sigma(a,1), \sigma(a,3)])$. We define $O[a][b][s][t][\mathbf{x}][\mathbf{y}]$ recursively as

$$\min \begin{cases} -(1-\lambda)\bar{g}(w_a, \mathbf{x}), & \text{if } a = b, \mathbf{x} = \mathbf{y}, \\ \infty, & \text{if } a = b, \mathbf{x} \neq \mathbf{y}, \\ O[a][b][s+1][t][\mathbf{x}][\mathbf{y}], & \text{if } a < b, s \in \{1,2\}, \\ O[a][b][s][t-1][\mathbf{x}][\mathbf{y}], & \text{if } a < b, t \in \{2,3\}, \\ \min\limits_{\mathbf{x}' \in S(w_{a+1})} \{O[a+1][b][1][t][\mathbf{x}'][\mathbf{y}]\} - (1-\lambda)\bar{g}(w_a, \mathbf{x}), & \text{if } a \leq b-1, s = 3, \\ \min\limits_{\mathbf{y}' \in S(w_{b-1})} \{O[a][b-1][s][3][\mathbf{x}][\mathbf{y}']\} - (1-\lambda)\bar{g}(w_b, \mathbf{y}), & \text{if } a \leq b-1, t = 1, \\ \min\limits_{a \leq c < b, t' \in \{1,2\}, \mathbf{x}' \in S(w_c)} \left\{ \begin{array}{l} O[a][c][s][t'][\mathbf{x}][\mathbf{x}'] \\ \quad + E[c][b][t'+1][t][\mathbf{x}'][\mathbf{y}] \\ \quad + (1-\lambda)\bar{g}(w_c, \mathbf{x}') \end{array} \right\}, & \text{if } a < b, \\ \min\limits_{a \leq c < b-1, \mathbf{y}' \in S(w_c), \mathbf{x}' \in S(w_{c+1})} \left\{ \begin{array}{l} O[a][c][s][3][\mathbf{x}][\mathbf{y}'] \\ \quad + E[c+1][b][1][t][\mathbf{x}'][\mathbf{y}] \end{array} \right\}, & \text{if } a < b-1, \\ E[a][b][s][t][\mathbf{x}][\mathbf{y}], & \text{if } a < b, (x_s, y_t) \in \Gamma. \end{cases}$$

There are two components in the objective function, the CAI and the MFE. We account for MFE upon identifying structural elements at base pairing $(\sigma(a,s), \sigma(b,t))$ using the energy functions in Definition 9. To avoid double counting, we must ensure that CAI is only accounted for once for each codon. As such, we include a CAI contribution when crossing codon boundaries or reaching a valid base case.

The first case in the above recurrence corresponds to the base case where $a = b$ and $\mathbf{x} = \mathbf{y}$. In that case, base pairing between $\sigma(a,s)$ and $\sigma(b,t)$ is not possible as the Turner energy model [14] requires at least two nucleotides in between a pairing. In this base case, we must account for the CAI contribution of codon $\mathbf{x}$. The other base case occurs when $a = b$ and $\mathbf{x} \neq \mathbf{y}$, which is not allowed as any one amino acid must be encoded by a single codon – this case thus receives a value of ∞.

The next two cases correspond to, respectively, incrementing either the left index $\sigma(a,s)$ or decrementing the right index $\sigma(b,t)$ without crossing any codon boundary and leaving the corresponding nucleotide unpaired. As such, we do not have to account for CAI. However, in the following two cases, we additionally cross the codon boundary and thus must account for the CAI contribution of respectively codons $\mathbf{x}$ and $\mathbf{y}$. Next, we include two cases corresponding to bifurcating into two parts, one part is between nucleotides $\sigma(a,s)$ and

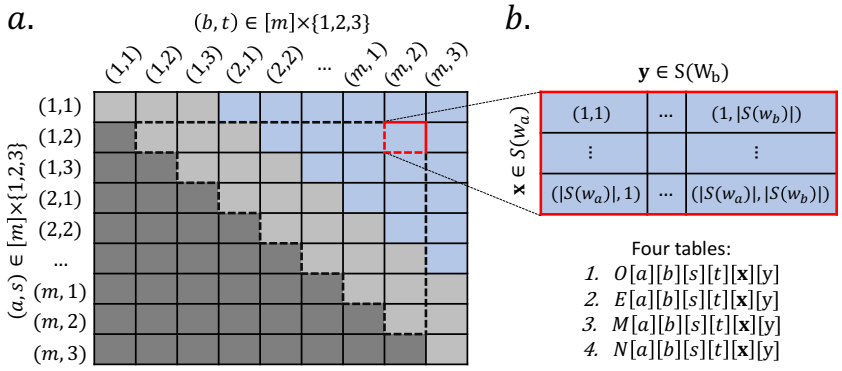


Figure 3 Dynamic programming for solving the Balanced RNA Design problem. (a) To solve this problem, we store four dynamic programming tables O, E, M and N with identical dimensions indexed as $[a][b][s][t][\mathbf{x}][\mathbf{y}]$. Rows and columns correspond to pairs $(a, s), (b, t) \in [m] \times \{1, 2, 3\}$, respectively, both ordered lexicographically in increasing order. With the exception of the base cases for table O where $a = b$ (indicated in light gray), the recurrences require $a < b$ (indicated in blue). The dashed lines outline the entries of the table on which the red entry depends. (b) Each entry $[(a, s)][(s, t)]$ expands into another codon-by-codon table, whose rows are codons $\mathbf{x} \in S(w_a)$ and columns are codons $\mathbf{y} \in S(w_b)$.

$\sigma(c, t')$ and the other part is between nucleotides $\sigma(c, t') + 1$ and $\sigma(b, t)$. In the first case, the split happens inside a codon, i.e. $t' \in \{1, 2\}$. We must include a correction of $+(1-\lambda)\bar{g}(w_c, \mathbf{x}')$ as both parts will include a CAI contribution of the same codon $\mathbf{x}'$. On other hand, when the split happens outside a codon, i.e. $t' = 3$ then no such correction is needed.

The last case corresponds to base pairing between $\sigma(a, s)$ and $\sigma(b, t)$. Specifically, $E[a][b][s][t][\mathbf{x}][\mathbf{y}]$ denotes the optimal objective value when nucleotides $v_{\sigma(a,s)}$ and $v_{\sigma(b,t)}$ correspond to codons $\mathbf{x}$ and $\mathbf{y}$, respectively, and form a base pairing. When calculating $E[a][b][s][t][\mathbf{x}][\mathbf{y}]$, we consider the minimum among the five cases corresponding the five secondary structures elements defined in Section 2. That is, $E[a][b][s][t][\mathbf{x}][\mathbf{y}]$ equals $\min\{E_s[a][b][s][t][\mathbf{x}][\mathbf{y}], E_h[a][b][s][t][\mathbf{x}][\mathbf{y}], E_b[a][b][s][t][\mathbf{x}][\mathbf{y}], E_i[a][b][s][t][\mathbf{x}][\mathbf{y}], E_m[a][b][s][t][\mathbf{x}][\mathbf{y}]\}$. The precise definitions are given in Appendix A.2. In particular, we require two additional recurrences $M[a][b][s][t][\mathbf{x}][\mathbf{y}]$ and $N[a][b][s][t][\mathbf{x}][\mathbf{y}]$ for solving the multi-branch loop case.

3.1.1 Dynamic Programming, Time and Space Complexity

We store the following four tables: (i) $O[a][b][s][t][\mathbf{x}][\mathbf{y}]$, (ii) $E[a][b][s][t][\mathbf{x}][\mathbf{y}]$, (iii) $M[a][b][s][t][\mathbf{x}][\mathbf{y}]$ and (iv) $N[a][b][s][t][\mathbf{x}][\mathbf{y}]$, each with the same dimensions. In particular, as each potential base pairing $(\sigma(a, s), \sigma(b, t))$ corresponds to exactly one of five structural elements, we do not store the corresponding values $E_s[a][b][s][t][\mathbf{x}][\mathbf{y}]$, $E_h[a][b][s][t][\mathbf{x}][\mathbf{y}]$, $E_b[a][b][s][t][\mathbf{x}][\mathbf{y}]$, $E_i[a][b][s][t][\mathbf{x}][\mathbf{y}]$ and $E_m[a][b][s][t][\mathbf{x}][\mathbf{y}]$ separately, but only their minimum value in $E[a][b][s][t][\mathbf{x}][\mathbf{y}]$. Note that the four stored tables have the same dimensions comprised of protein sequence indices $a, b \in [m]$, codon indices $s, t \in \{1, 2, 3\}$, and codons $\mathbf{x} \in S(w_a)$ and $\mathbf{y} \in S(w_b)$. Letting K denote the maximum number of codons associated with a single amino acid – the amino acids leucine (L), serine (S) and arginine (R) each have $K = 6$ of codons – we conclude that the space complexity is $\mathcal{O}(m^2K^2)$.

Inspection of the recurrences reveals that the computation of each entry $[a][b][s][t][\mathbf{x}][\mathbf{y}]$ in the four tables does not require access to entries $[a][b][s][t][\mathbf{x}'][\mathbf{y}']$ using other codons $\mathbf{x}' \neq \mathbf{x}$ and $\mathbf{y}' \neq \mathbf{y}$. On the other hand, we do require access to entries $[a'][b'][s'][t'][\mathbf{x}'][\mathbf{y}']$ where

$\sigma(a', s') \geq \sigma(a, s)$, $\sigma(b', t') \leq \sigma(b, t)$ (indicated with dashed lines in Figure 3). Moreover, with the exception of the base cases for table O, where $a = b$, the recurrences require $a < b$. This means we can organize the four tables as two-dimensional tables where the rows correspond to entries (a, s) and the columns correspond to entries (b, t), both sorted in increasing lexicographical order. Each entry $[(a, s)][(b, t)]$ corresponds to another two-dimensional table whose rows correspond to codons $\mathbf{x} \in S(w_a)$ and columns to codons $\mathbf{y} \in S(w_b)$ – see Figure 3b. We fill out the tables diagonally. More precisely, filling out the four entries indexed by $[a][b][s][t][\mathbf{x}][\mathbf{y}]$, we check if base pairing between $\sigma(a, s)$ and $\sigma(b, t)$ is possible, i.e. if $(x_s, y_t) \in \Gamma$. If so, we will first fill out the entry in E followed by N and then finally M. On the other hand, we will first fill out the entry in N, then M and finally E. After completely filling out tables E, M and N, we fill out table O. This ordering follows from the recurrences. We use back pointers to identify the optimal solution $(\mathbf{v}, P)$ when backtracing.

For each entry $O[a][b][s][t][\mathbf{x}][\mathbf{y}]$, the running time is dominated by the case to determine $E[a][b][s][t][\mathbf{x}][\mathbf{y}]$. That is, for each entry $E[a][b][s][t][\mathbf{x}][\mathbf{y}]$, it takes $\mathcal{O}(K^2)$ time to compute a stacking element or a hairpin loop element, $\mathcal{O}(mK^2)$ time to compute a bulge loop element, and worst case $\mathcal{O}(m^2K^6)$ time to determine the contribution of an internal loop element. To remedy the worst case $\mathcal{O}(m^2K^6)$ time, we follow other secondary structure prediction methods and employ a parameter L to bound the maximum interior loop size, including bulge loop and internal loop [9,12]. Then, the time to determine the contribution of an internal loop element can be reduced to $\mathcal{O}(mLK^6)$. Since there are $\mathcal{O}(m^2K^2)$ entries to compute, the overall time complexity of solving the dynamic program is $\mathcal{O}(m^2K^2) \cdot \mathcal{O}(mLK^6) = \mathcal{O}(m^3LK^8)$.

When disregarding CAI, i.e. $\lambda = 1$, we can adapt the recurrences such that for each entry $E[a][b][s][t][\mathbf{x}][\mathbf{y}]$, it would take $\mathcal{O}(1)$ time to compute a a stacking or a hairpin loop element, $\mathcal{O}(m)$ time to compute a bulge loop element, and worst case $\mathcal{O}(m^2)$ time to determine the contribution of an internal loop element. With a similar implementation of a maximum interior loop size L, the time to compute an internal loop element can be reduced to $\mathcal{O}(mL)$. Thus, the overall time complexity drops to $\mathcal{O}(m^2K^2) \cdot \mathcal{O}(mL) = \mathcal{O}(m^3LK^2)$ when $\lambda = 1$.

3.2 Pareto Optimal RNA Design

In the Pareto Optimal RNA Design problem (Problem 3), we are given a protein sequence $\mathbf{w} \in \Sigma_{prot}^m$ and seek a set of Pareto optimal solutions $(\mathbf{v}, P)$. We use the weighted sum method [30]. In this method, distinct convex combinations of the multiple objective functions is optimized. In our case this corresponds to solving distinct convex combinations of the two objectives MFE (Definition 9) and $\overline{\text{CAI}}$ (Equation (2)), which correspond to solving distinct instances of the Balanced RNA Design problem with varying values of the parameter $\lambda \in [0, 1]$. The weighted sum method has several limitations: (i) multiple λs may generate the same solution, (ii) the non-convex part of the Pareto front cannot be recovered, and (iii) there are non-uniform sampling issues [2, 5].

We mitigate the first limitation by recursively examining λ values. More specifically, we maintain a queue Q of intervals $[\lambda^-, \lambda^+]$ as well as a hash table X such that $X[\lambda]$ yields the solution $(\mathbf{v}, P)$ of the Balanced RNA Design (BRD) problem instance $(\mathbf{w}, \lambda)$. Initially, Q contains a single interval $[\epsilon, 1 - \epsilon]$ where ϵ is a small constant (the default value in our implementation is $\epsilon = 10^{-5}$). Additionally, we initialize $X[\epsilon]$ and $X[1 - \epsilon]$ with the solutions of BRD problem instances $(\mathbf{w}, \epsilon)$ and $(\mathbf{w}, 1 - \epsilon)$, respectively. As long as the queue Q is not empty, we obtain an interval $[\lambda^-, \lambda^+]$ from Q, and solve a new BRD instance $(\mathbf{w}, \lambda)$ where $\lambda = \lambda^- + (\lambda^+ - \lambda^-)/2$, yielding solution $(\mathbf{v}, P)$. If this solution differs from $X[\lambda^-]$ and $X[\lambda^+]$, we set $X[\lambda] = (\mathbf{v}, P)$ and add (λ^-, λ) and (λ, λ^+) to the queue Q if $\lambda - \lambda^- > \tau$. We use a default value of 10^{-3} for the threshold parameter τ.

3.3 Implementation Details of DERNA

We implemented our algorithms for solving the BRD and PORD problems in C++11. The resulting method, DERNA (short for DEsign RNA), is available at `https://github.com/elkebir-group/derna.git` under the BSD 3-clause license. Usage instructions and examples are also available on the GitHub site.

DERNA uses the same energy model [15] as CDSfold [23]. For codon usage data, DERNA use the *Homo sapiens* codon usage table published in the codon usage database [18]. In addition, DERNA accepts alternative energy models and codon usage data in CSV format.

To validate the correctness of our algorithm and its implementation, we split our recurrences into two separate components and utilized two separate tables to store the MFE and the CAI separately. Using the real data instances examined in Section 4, for each solution $(\mathbf{v}, P)$ identified by DERNA, we confirmed that the MFE predicted by DERNA matched the MFE calculated using the Zuker algorithm [32] when given DERNA's inferred RNA sequence $\mathbf{v}$. Additionally, we recomputed the CAI of DERNA's inferred RNA sequence $\mathbf{v}$ and confirmed that the resulting value matched the CAI inferred by DERNA.

4 Results

We compare DERNA to CDSfold [23] and LinearDesign [31] on 100 protein sequences from the UniProt database [3] (Section 4.1) as well as on a case study involving the SARS-CoV-2 spike protein (Section 4.2). While the LinearDesign paper [31] describes both an exact and heuristic algorithm, only the heuristic algorithm was publicly available. As such, we were only able to include the heuristic algorithm in our benchmarking. All experiments were performed on a laptop with an Apple M1 Max 10-core CPU and 64 GB of RAM.

4.1 Benchmarking on 100 UniProt Protein Sequences

We begin by performing experiments that prioritize MFE over CAI in Section 4.1.1. In Section 4.1.2, we focus on the Pareto Optimal RNA Design problem, seeking solutions that collectively capture the trade-off between MFE and CAI.

4.1.1 Prioritizing MFE

The goal of this section is to assess the ability of RNA design methods to prioritize MFE over CAI. We seek solutions that achieve the minimum MFE and, as a secondary criterion, achieve largest CAI – i.e. among the space of solutions that achieve minimum MFE, we prefer those solutions that have the largest CAI value. We benchmarked using the same 100 protein sequences used in the CDSfold paper [23], which come from the UniProt database [3] and have lengths ranging from 78 to 2828 amino acids (Figure 4a). By design, CDSfold does not take CAI into account. Our method DERNA as well as LinearDesign support balancing MFE and CAI. For DERNA, we set $\lambda = 1 - \epsilon = 1 - 10^{-5}$. We note that LinearDesign's objective function is slightly different than DERNA's, seeking an RNA sequence $\mathbf{v}$ and secondary structure P that minimize $\text{MFE}(\mathbf{v}, P) - \lambda_{\text{LD}} \cdot \log \overline{\text{CAI}}(\mathbf{v}, \mathbf{w})$ for a target protein sequence $\mathbf{w}$. To similarly prioritize MFE, we set $\lambda_{\text{LD}} = \epsilon = 10^{-5}$ for LinearDesign and ran it with default parameters.

With the exception of the longest sequence (Q9NR99) with 2828 amino acids, which LinearDesign failed to complete within 24 hours (after which we killed the process), all methods ran successfully on all sequences. Moreover, with the exception of protein sequence Q9HAE3 (with 211 amino acids), all methods achieved the same minimum MFE (Figure 4b.

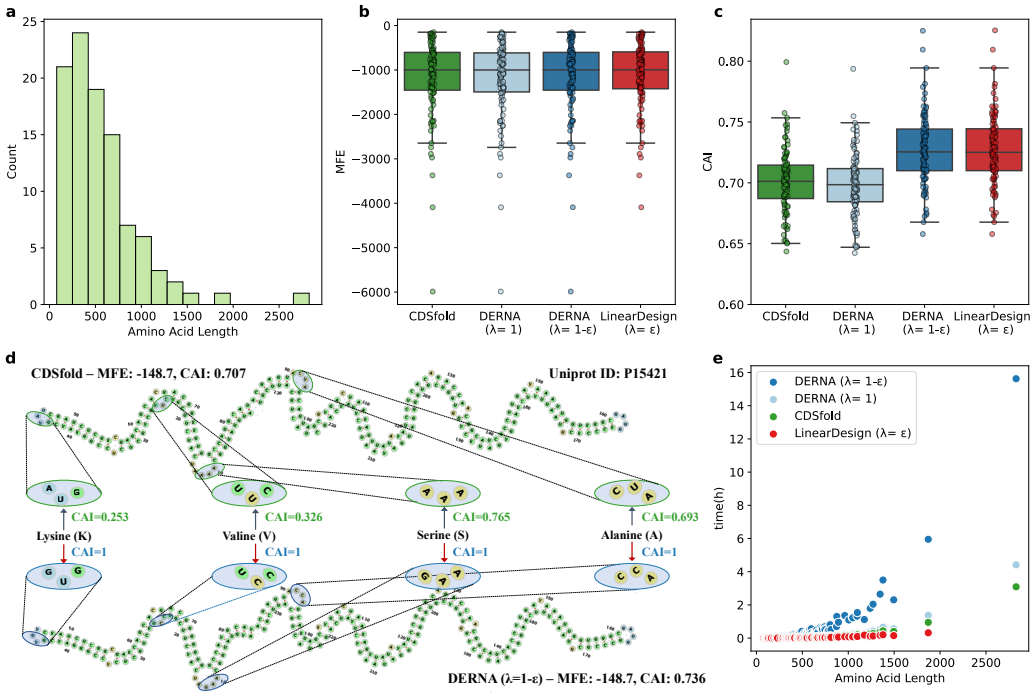


Figure 4 DERNA with $\lambda = 1 - \epsilon$ identifies solutions that achieve optimal MFE and largest CAI as a secondary objective. (a) We used 100 UniProt sequences, with varying lengths as shown. (b) With one exception (discussed in the text), all methods returned solutions with the same MFE. (c) However, the CAI values differed drastically between methods, with DERNA ($\lambda = 1 - \epsilon$) and LinearDesign outperforming CDSfold. (d) As an example, we show protein sequence P15421 for which CDSfold (top) and DERNA (bottom) inferred the same MFE and identical secondary structures. However, the solutions contain different codons resulting in different CAI values. (e) Wall-clock running times.

However, the CAI values varied between methods. In particular, DERNA with $\lambda = \epsilon$ and LinearDesign λ_{LD} achieved larger CAI values than CDSfold for all instances (Figure 4c and Figure S1). This makes sense because CDSfold only optimizes MFE but not CAI. The improved CAI values suggest that the sequences generated using our approach may exhibit higher *in vivo* translational efficiency without sacrificing mRNA half-life [16].

To further illustrate this point, we highlight the results for protein sequence P15421 with 78 amino acids. Both CDSfold and DERNA achieved the same MFE value of -148.7, yielding identical secondary structures (in terms of complementary base pairings) consisting of mostly stacking elements that achieve the lowest MFE. CDSfold, however, identified a different RNA sequence than DERNA resulting in a CAI of 0.707 whereas DERNA achieved a CAI of 0.736. The two RNA sequences differ at four codons encoding four distinct amino acids. For each such amino acid, DERNA used the codon that achieved the largest CAI value. For example, for the first codon encoding for the amino acid lysine (K), DERNA used the codon GUG with a relative usage frequency of 1 whereas CDSfold used the codon GUA with a smaller relative frequency of 0.253. The other three codons differed in a similar fashion. We note that LinearDesign identified the same RNA sequence as DERNA for this instance.

As for the CAI values inferred by LinearDesign, these largely match those inferred by DERNA (Figure 4c and Figure S1). The only exception is protein sequence Q9HAE3 where LinearDesign performed better in terms of CAI with a value of 0.754 vs. 0.748 for

DERNA. The solution inferred by DERNA, however, has a better MFE of -369.9 vs -369.4 for LinearDesign. Using a smaller $\lambda = 0.062509 < 1 - \epsilon$, DERNA was able to recover LinearDesign's solution. On the other hand, we were not able to identify a λ_{LD} value for which LinearDesign would identify DERNA's solution that achieved better MFE. A potential reason for this is that publicly-available version of LinearDesign is not an exact algorithm.

Finally, we consider the running times of CDSfold, LinearDesign and DERNA. Leaving out the largest instance (for which LinearDesign failed), we found that LinearDesign was the fastest algorithm with running times ranging from 1.80 to 1149.02 seconds, followed by CDSfold ranging from 1.86 to 3411.91 seconds and then DERNA with running times ranging from 16 to 21434 seconds. It is important to note that DERNA is an exact algorithm, while the publicly-available version of LinearDesign is a heuristic utilizing beam search. Indeed, as discussed above there was one instance were LinearDesign returned a suboptimal solution (in terms of the lexicographical objective of prioritizing MFE first followed by CAI).

We note that the difference in running times between DERNA and CDSfold because DERNA takes into consideration both MFE and CAI whereas CDSfold only considers MFE. As discussed in Section 3.1.1, leaving out CAI from the objective value reduces the asymptotic running time from $O(m3LK^8)$ to $O(m^3LK^2)$ where m is the protein sequence length and K and L are constants corresponding to the maximum number of codons per amino acid and the maximum interior loop length, respectively. Indeed, this is also reflected in wall-clock times when running an altered version of DERNA that only considers MFE, reducing the running times to between 2 and 4951 seconds, closely matching those of LinearDesign (Figure 4e). As expected, however, this comes at the expense of decreased CAI values for the inferred RNA sequences (Figure 4c and Figure S1).

4.1.2 Balancing MFE and CAI

We now assess DERNA's ability to identify Pareto optimal solutions. To that end, we ran DERNA in λ-sweep mode with a termination threshold value of $\tau = 0.001$. Note that the number of λ values explored by DERNA depends on both the value of τ as well as the input instance itself. Unlike our method, LinearDesign does not include an automated way of altering their λ_{LD} parameter. As such, we manually varied $\lambda_{\text{LD}} \in \{10^{-10}, 10^{-3}, 0.1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100\}$ – we did not set λ_{LD} to the same, instance-specific λ values examined by DERNA as the two parameters play different roles in the corresponding objective functions of both methods. Due to an increased number of runs per instance, we restricted our analysis to the 50 smallest instances with lengths ranging from 78 to 494 amino acids.

We begin by discussing the results for protein sequence P15421, which has 78 amino acids. DERNA examined 27 distinct λ values, leading to 12 distinct solutions (Figure 5a). On the other hand, the list of 14 λ_{LD} values resulted in 9 distinct solutions identified by LinearDesign. Recall that $\lambda = 1$ prioritizes MFE for DERNA whereas $\lambda = 0$ prioritizes CAI. Moreover, recall that each value of $\lambda \in [0, 1]$ leads to a Pareto optimal solution. A natural question is what is the smallest value λ^{MFE} that resulted in the optimal MFE? For protein sequence P15421 this was $\lambda^{\text{MFE}} = 0.0371186$. Given that $\tau = 0.001$, this means that DERNA does not explore the part of the Pareto front that contains solutions with higher CAI values. Indeed, for this protein sequence, the largest non-optimal CAI value identified by DERNA equals 0.968613, obtained using $\lambda = 0.000987$, followed by a CAI of 0.923779 using $\lambda = 0.001963$. On the other hand, the largest non-optimal CAI value identified by LinearDesign equals 0.991, which was obtained using $\lambda_{\text{LD}} = 10$, with a total of 7 solutions that have a CAI of at least 0.923779. A downside of LinearDesign's objective function, which

is of the form $\text{MFE}(\mathbf{v}, P) - \lambda_{\text{LD}} \cdot \log \overline{\text{CAI}}(\mathbf{v}, \mathbf{w})$, is that a non-bounded $\lambda_{\text{LD}} = \infty$ is required to exclusively prioritize CAI as opposed to a bounded value of $\lambda = 1$ for DERNA. Here, LinearDesign obtained this CAI-optimal solution only using $\lambda_{\text{LD}} = 100$.

We now extend these analyses to all 50 protein sequences. First, we observed that the median value of λ_{MFE} – the smallest λ that produces an MFE optimal solution – equals 0.0546964. Second, for $\tau = 0.001$, the median number of λs examined by DERNA is 36 (Figure S2a), yielding a median number of 17 solutions (Figure S2b). On the other hand, the 14 λ_{LD} examined by LinearDesign yielded a median number of 13 solutions (Figure S2c). To compare MFEs across instances, we define the MFE percentage as $(\text{MFE}(\mathbf{w}, \lambda) - \text{MFE}(\mathbf{w}, 0)/(\text{MFE}(\mathbf{w}, 1) - \text{MFE}(\mathbf{w}, 0))$ for each protein sequence $\mathbf{w}$ where $\text{MFE}(\mathbf{w}, \lambda)$ equals the MFE value of the solution obtained using λ. In other words, an MFE percentage of 100% means that the identified solution achieved the best possible MFE whereas an MFE percentage of 0% means that the worst MFE that favors CAI was obtained. We define CAI percentage similarly. Matching the previous analysis, we indeed see that DERNA favored the part of the Pareto front that prioritizes MFE (Figure 5b). Conversely, for our choices of λ_{LD}, LinearDesign more heavily favored the part of the Pareto front that prioritizes CAI (Figure 5c).

Finally, we delve more into the trade-off between CAI and MFE. To that end, we explored the following two questions. First, if one is willing to accept a certain CAI percentage, what is the best MFE that one can obtain? Second, for a specified minimum MFE percentage, what is the best CAI that one can obtain? Among the 50 considered instances, we found that if we accept solutions with a CAI percentage of at least 50% the corresponding best MFE percentages for these solutions identified by DERNA range from 81.298% to 92.65% with a median of 88.066% (Figure 5e). However, increasing the minimum CAI percentage to at least 80%, resulted in a decrease in best MFE of solutions identified by DERNA, with MFE percentages ranging from 55.624% to 72.965% with a median of 61.263%. Conversely, for an MFE percentage of at least 50%, DERNA obtained solutions that have CAI percentages ranging from 86.403% to 91.386% with a median of 87.874% (Figure 5f). Increasing the minimum MFE percentage to at least 80%, resulted in a decrease in best CAI of solutions identified by DERNA, with CAI percentages ranging from 54.443% to 71.362% with a median of 61.075%. When designing an RNA sequence for a target protein it is important to understand the trade-off between MFE and CAI, especially when trying to identify a single solution on the Pareto front.

4.2 Case Study: SARS-CoV-2 Spike Protein

The spike (S) protein on the surface of the SARS-CoV-2 virus is responsible for recognizing and binding to the host cell's receptors, as well as merging itself with the host cell membrane, without which the virus would be unable to interact with the host cells and initiate infection [10]. The SARS-CoV-2 S protein, with its 1273 amino acids, is therefore the primary target of the Moderna and Pfizer-BioNTech mRNA vaccines [20].

We applied LinearDesign to the S protein using a list of manually set values for λ_{LD}, specifically $\lambda_{\text{LD}} \in \{10^{-10}, 10^{-3}, 0.1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100\}$. LinearDesign generated 14 distinct solutions corresponding to the 14 chosen λ values. Similarly, we ran DERNA on the S protein with termination threshold $\tau = 0.0001$, ten times smaller than the previous analysis in Section 4.1.2. DERNA evaluated 76 distinct λ values and generated 56 distinct solutions. The set of solutions obtained through LinearDesign overlaps with those generated by DERNA (Figure 6a), with 3 identical solutions identified by both LinearDesign and DERNA.

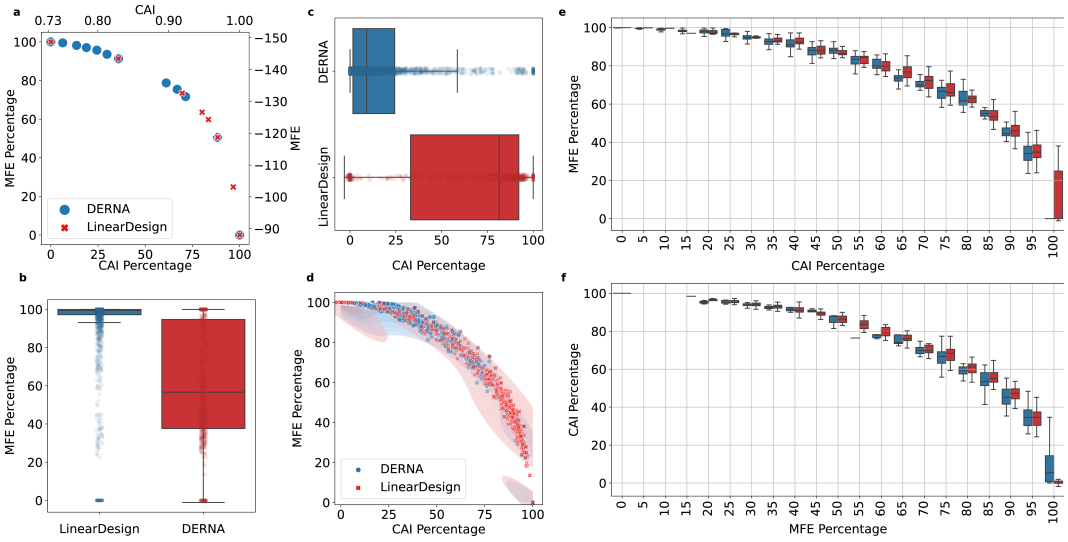


Figure 5 DERNA models the trade-off between MFE and CAI. These analyses are restricted to the 50 smallest UniProt sequences. (a) Solutions identified by DERNA (blue) and LinearDesign (red) for proteins sequence P15421. The right y-axis shows the MFE whereas the left y-axis shows the range-normalized MFE percentage. Similarly, the top x-axis shows the CAI wheres the bottom x-axis shows the range-normalized CAI percentage. (b-d) MFE and CAI percentages inferred by both methods across all 50 instances. (e) For each instance, we show the best MFE percentage on the y-axis when only considering solutions that achieve the CAI percentage specified on the x-axis. (f) For each instance, we show the best CAI percentage on the y-axis when only considering solutions that achieve the MFE percentage specified on the x-axis.

Finally, we compared DERNA's solutions to the Pfizer-BioNTech and Moderna mRNA sequences. The Pfizer-BioNTech mRNA sequence has an MFE of -1217 and a CAI of 0.95 (Figure 6b). For the same CAI value, DERNA identified a solution with a better MFE of -1955.2 (Figure 6c). On the other hand, the Moderna mRNA sequence has an MFE of -1369.2 and a CAI of 0.98. Similarly, for the same CAI value, DERNA identified a solution with a better MFE of -1724.8 These two alternative solutions might lead to increased mRNA half-life without sacrificing translational efficacy [16, 24]. We note that the overall minimum MFE equals -2486.7 with a corresponding CAI of 0.737 (Figure S3a), whereas solutions with overall maximum CAI of 1 lead to a decreased best MFE of -1384.3 (Figure S3b).

5 Discussion

Given a target protein sequence $\mathbf{w}$, we introduced the Pareto Optimal RNA Design (PORD) problem of identifying a set of Pareto optimal solutions $(\mathbf{v}, P)$ composed of an RNA sequence $\mathbf{v}$ that encodes for $\mathbf{w}$ and its corresponding secondary structure P that together balance the minimum free energy (MFE) and codon adaptation index (CAI). In addition, we introduced the Balanced RNA Design (BRD) problem, where we additionally take as input the parameter $\lambda \in [0, 1]$ and return an RNA sequence $\mathbf{v}$ whose corresponding secondary structure P minimizes $\lambda \cdot \text{MFE}(\mathbf{v}, P) - (1 - \lambda) \cdot \overline{\text{CAI}}(\mathbf{v}, \mathbf{w})$. To solve both problems, we introduced DERNA (DEsign RNA). Building on the work of Zuker and Stiegler [32], DERNA solves the BRD problem via dynamic programming in $\mathcal{O}(|\mathbf{w}|^3)$ time and $\mathcal{O}(|\mathbf{w}|^2)$ space. In addition, DERNA solves the PORD problem via the weighted sum method [30], enumerating the Pareto front by solving multiple distinct instances of the BRD problem via a systematic

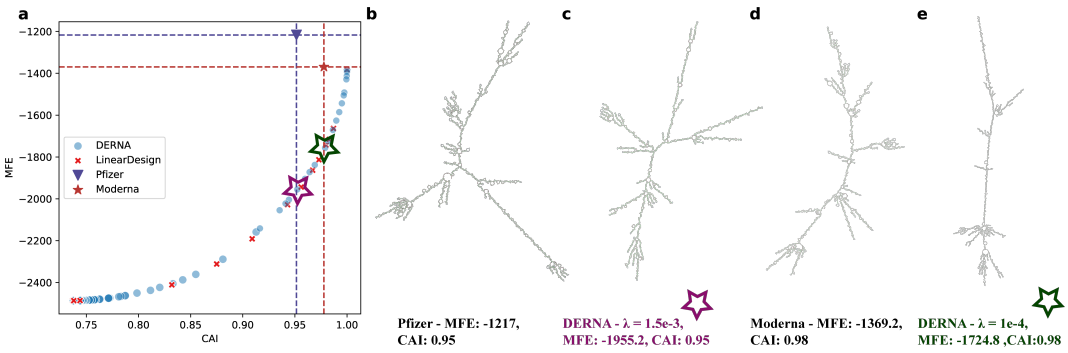


Figure 6 DERNA identifies alternative sequences for the SARS-CoV-2 spike (S) protein. (a) Solution identified by DERNA (blue) and LinearDesign (red). (b) Secondary structures of the Pfizer-BioNTech and Moderna mRNA vaccine sequences and alternative solutions provided by DERNA, from left to right are Pfizer-BioNTech, DERNA with $\lambda = 1.5 \cdot 10^{-3}$, Moderna, and DERNA with $\lambda = 10^{-4}$ respectively.

sweep on λ. On a benchmark dataset of 100 protein sequences, we demonstrated that DERNA obtained solutions with identical MFE but superior CAI compared to CDSfold [23], a previous approach that only optimizes MFE. Additionally, we showed that DERNA matched LinearDesign's performance in terms of solution quality, a recent approach that similarly seeks to balance MFE and CAI. While LinearDesign demonstrated better performance in terms of runtime, it is important to note that it employs a parameter-dependent algorithm that produces heuristic outcomes, whereas DERNA is guaranteed to solve the problem to optimality. In addition, key functionality of LinearDesign is closed source, whereas DERNA is fully open source. Finally, we demonstrated our method's potential for mRNA vaccine design using SARS-CoV-2 spike as the target protein.

For future development, it would be beneficial to integrate additional secondary structures beyond the five already considered in the algorithm, such as dangling ends. In particular, dangling ends allow one to capture the importance of 5' end in mRNA stability. That is, several studies have shown that secondary structure near the 5' untranslated region leads to decreased translation initiation and therefore decreased translational efficiency [6, 25, 27]. It will be particularly interesting to identify RNA sequences whose best MFE secondary structure lacks secondary structure at the 5' – this will probably require similar techniques as employed in traditional RNA design where one seeks an RNA sequence that folds into a desired RNA secondary structure [9, 11]. Finally, it will be valuable to investigate computing the Pareto front through algebraic dynamic programming [21].

References

1 Barry Cohen and Steven Skiena. Natural selection and algorithmic design of mRNA. *Journal of Computational Biology*, 10(3-4):419–432, 2003.

2 Jared L Cohon. *Multiobjective programming and planning*, volume 140. Courier Corporation, 2004.

3 The UniProt Consortium. UniProt: the Universal Protein Knowledgebase in 2023. *Nucleic Acids Research*, 51(D1):D523–D531, November 2022. doi:10.1093/nar/gkac1052.

4 Daan JA Crommelin, Thomas J Anchordoquy, David B Volkin, Wim Jiskoot, and Enrico Mastrobattista. Addressing the cold reality of mRNA vaccine stability. *Journal of Pharmaceutical Sciences*, 110(3):997–1001, 2021. doi:10.1016/j.xphs.2020.12.006.

5 I. Das and J. E. Dennis. A closer look at drawbacks of minimizing weighted sums of objectives for Pareto set generation in multicriteria optimization problems. *Structural Optimization*, 14(1):63–69, August 1997. doi:10.1007/BF01197559.

6 Yiliang Ding, Yin Tang, Chun Kit Kwok, Yu Zhang, Philip C Bevilacqua, and Sarah M Assmann. In vivo genome-wide profiling of RNA secondary structure reveals novel regulatory features. *Nature*, 505(7485):696–700, 2014.

7 Susan M Freier, Ryszard Kierzek, John A Jaeger, Naoki Sugimoto, Marvin H Caruthers, Thomas Neilson, and Douglas H Turner. Improved free-energy parameters for predictions of RNA duplex stability. *Proceedings of the National Academy of Sciences*, 83(24):9373–9377, 1986. doi:10.1073/pnas.83.24.9373.

8 Claes Gustafsson, Sridhar Govindarajan, and Jeremy Minshull. Codon bias and heterologous protein expression. *Trends in Biotechnology*, 22(7):346–353, 2004. doi:10.1016/j.tibtech.2004.04.006.

9 Ivo L Hofacker, Walter Fontana, Peter F Stadler, L Sebastian Bonhoeffer, Manfred Tacker, Peter Schuster, et al. Fast folding and comparison of RNA secondary structures. *Monatshefte fur chemie*, 125:167–167, 1994.

10 Yuan Huang, Chan Yang, Xin-feng Xu, Wei Xu, and Shu-wen Liu. Structural and functional properties of sars-cov-2 spike protein: potential antivirus drug development for covid-19. *Acta Pharmacologica Sinica*, 41(9):1141–1149, 2020.

11 Robert Kleinkauf, Martin Mann, and Rolf Backofen. antaRNA: ant colony-based RNA sequence design. *Bioinformatics*, 31(19):3114–3121, 2015.

12 Rune B Lyngso, Michael Zuker, and CN Pedersen. Fast evaluation of internal loops in RNA secondary structure prediction. *Bioinformatics (Oxford, England)*, 15(6):440–445, 1999. doi:10.1093/bioinformatics/15.6.440.

13 Elisabeth Mahase. Covid-19: Moderna vaccine is nearly 95% effective, trial involving high risk and elderly people shows. *BMJ: British Medical Journal (Online)*, 371, 2020.

14 David H Mathews, Matthew D Disney, Jessica L Childs, Susan J Schroeder, Michael Zuker, and Douglas H Turner. Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure. *Proceedings of the National Academy of Sciences*, 101(19):7287–7292, 2004. doi:10.1073/pnas.0401799101.

15 David H Mathews, Jeffrey Sabina, Michael Zuker, and Douglas H Turner. Expanded sequence dependence of thermodynamic parameters improves prediction of rna secondary structure. *Journal of molecular biology*, 288(5):911–940, 1999. doi:10.1006/jmbi.1999.2700.

16 David M Mauger, B Joseph Cabral, Vladimir Presnyak, Stephen V Su, David W Reid, Brooke Goodman, Kristian Link, Nikhil Khatwani, John Reynders, Melissa J Moore, et al. mRNA structure regulates protein expression through changes in functional half-life. *Proceedings of the National Academy of Sciences*, 116(48):24075–24083, 2019. doi:10.1073/pnas.1908052116.

17 SA Meo, IA Bukhari, J Akram, AS Meo, and D COVID Klonoff. COVID-19 vaccines: comparison of biological, pharmacological characteristics and adverse effects of pfizer/biontech and moderna vaccines. *Eur Rev Med Pharmacol Sci*, pages 1663–1669, 2021.

18 Yasukazu Nakamura, Takashi Gojobori, and Toshimichi Ikemura. Codon usage tabulated from international dna sequence databases: status for the year 2000. *Nucleic acids research*, 28(1):292–292, 2000. doi:10.1093/nar/28.1.292.

19 Vladimir Presnyak, Najwa Alhusaini, Ying-Hsin Chen, Sophie Martin, Nathan Morris, Nicholas Kline, Sara Olson, David Weinberg, Kristian E. Baker, Brenton R. Graveley, and Jeff Coller. Codon optimality is a major determinant of mRNA stability. *Cell*, 160(6):1111–1124, 2015. doi:10.1016/j.cell.2015.02.029.

20 Giovanni Salvatori, Laura Luberto, Mariano Maffei, Luigi Aurisicchio, Giuseppe Roscilli, Fabio Palombo, and Emanuele Marra. Sars-cov-2 spike protein: an optimal immunological target for vaccines. *Journal of translational medicine*, 18(1):222, 2020. doi:10.1186/s12967-020-02392-y.

21 Cédric Saule and Robert Giegerich. Pareto optimization in algebraic dynamic programming. *Algorithms for Molecular Biology*, 10(1):1–20, 2015.

22 Paul M Sharp and Wen-Hsiung Li. The codon adaptation index-a measure of directional synonymous codon usage bias, and its potential applications. *Nucleic acids research*, 15(3):1281–1295, 1987. doi:10.1093/nar/15.3.1281.

23 Goro Terai, Satoshi Kamegai, and Kiyoshi Asai. CDSfold: an algorithm for designing a protein-coding sequence with the most stable secondary structure. *Bioinformatics*, 32(6):828–834, 2016. doi:10.1093/bioinformatics/btv678.

24 Tamir Tuller, Yedael Y. Waldman, Martin Kupiec, and Eytan Ruppin. Translation efficiency is determined by both codon bias and folding energy. *Proceedings of the National Academy of Sciences*, 107(8):3645–3650, 2010. doi:10.1073/pnas.0909910107.

25 Tamir Tuller and Hadas Zur. Multiple roles of the coding sequence 5' end in gene expression regulation. *Nucleic acids research*, 43(1):13–28, 2015.

26 Douglas H Turner, Naoki Sugimoto, and Susan M Freier. RNA structure prediction. *Annual review of biophysics and biophysical chemistry*, 17(1):167–192, 1988.

27 Yue Wan, Kun Qu, Qiangfeng Cliff Zhang, Ryan A Flynn, Ohad Manor, Zhengqing Ouyang, Jiajing Zhang, Robert C Spitale, Michael P Snyder, Eran Segal, et al. Landscape and variation of RNA secondary structure across the human transcriptome. *Nature*, 505(7485):706–709, 2014.

28 Hannah K Wayment-Steele, Do Soon Kim, Christian A Choe, John J Nicol, Roger Wellington-Oguri, Andrew M Watkins, R Andres Parra Sperberg, Po-Ssu Huang, Eterna Participants, and Rhiju Das. Theoretical basis for stabilizing messenger RNA through secondary structure design. *Nucleic Acids Research*, 49(18):10604–10617, September 2021. doi:10.1093/nar/gkab764.

29 David E. Weinberg, Premal Shah, Stephen W. Eichhorn, Jeffrey A. Hussmann, Joshua B. Plotkin, and David P. Bartel. Improved ribosome-footprint and mrna measurements provide insights into dynamics and regulation of yeast translation. *Cell Reports*, 14(7):1787–1799, 2016. doi:10.1016/j.celrep.2016.01.043.

30 L. Zadeh. Optimality and non-scalar-valued performance criteria. *IEEE Transactions on Automatic Control*, 8(1):59–60, 1963. doi:10.1109/TAC.1963.1105511.

31 He Zhang, Liang Zhang, Ang Lin, Congcong Xu, Ziyu Li, Kaibo Liu, Boxiang Liu, Xiaopin Ma, Fanfan Zhao, Huiling Jiang, et al. Algorithm for optimized mRNA design improves stability and immunogenicity. *Nature*, pages 1–3, 2023. doi:10.1038/s41586-023-06127-z.

32 Michael Zuker and Patrick Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic acids research*, 9(1):133–148, 1981. doi:10.1093/nar/9.1.133.

A Supplementary Methods

A.1 Turner Energy Function

In this section, we give detailed definitions for $f_{\mathrm{s}}, f_{\mathrm{h}}, f_{\mathrm{b}}, f_{\mathrm{i}}$ and f_{m} based on the Turner energy model [14]. Let $\mathbf{v} \in \Sigma_{\mathrm{rna}}^n$ be an RNA sequence. Recall that $\Gamma = \{(A,U),(U,A),(G,C),(C,G),(G,U),(U,G)\}$ is the set of allowed base pairings.

We begin with f_{s}, which takes in two base pairings $(v_i, v_j), (v_{i+1}, v_{j-1}) \in \Gamma$. Then, f_{s} computes the free energy contributed by the stacking element as

$$f_{\mathrm{s}}(v_i v_j, v_{i+1}, v_{j-1}) = \mathrm{stacking}[(v_i, v_j)][(v_{i+1}, v_{j-1})]$$

where $\mathrm{stacking} : \Gamma \times \Gamma \to \mathbb{R}$ is a lookup table with experimentally measured element energies.

For the hairpin element, f_{h} takes in the base pairing $(v_i, v_j) \in \Gamma$, the unpaired nucleotides v_{i+1}, v_{j-1}, and the length of the hairpin loop $l = j - i$. Then, f_{h} yields the free energy contributed by the hairpin loop as

$$f_{\mathrm{h}}(v_i, v_j, v_{i+1}, v_{j-1}, l) = \mathrm{hairpin}[l] + \mathrm{mismatchH}[(v_i, v_j])][v_{i+1}][v_{j-1}] \\ + \mathbb{1}\{l = 3 \wedge (v_i, v_j) \in \{(A,U),(U,A)\}\} \cdot D$$

where $\mathrm{hairpin} : \mathbb{N} \to \mathbb{N}$ and $\mathrm{mismatchH} : \Gamma \times \Sigma_{\mathrm{rna}} \times \Sigma_{\mathrm{rna}} \to \mathbb{R}$ are lookup tables with free energies for the length of the hairpin and paired and their directly adjacent nucleotides, respectively. Finally, D is an additional penalty term applied to AU base pairings.

For the bulge loop element, f_{b} takes in two base pairings $(v_i, v_j), (v_{p_1}, v_{q_1}) \in \Gamma$ and the length of the bulge loop $l = \max(j - i, q_1 - p_1)$. The free energy f_{b} contributed by the bulge loop equals

$$f_{\mathrm{b}}(v_i, v_j, v_{p_1}, v_{q_1}, l) = \mathrm{bulge}[l] + \mathbb{1}\{(v_i, v_j) \in \{(A,U),(U,A)\}\} \cdot D \\ + \mathbb{1}\{(v_{p_1}, v_{q_1}) \in \{(A,U),(U,A)\}\} \cdot D$$

where $\mathrm{bulge} : \mathbb{N} \to \mathbb{R}$ is a lookup table with free energies for the length of the hairpin, and D is an additional penalty term applied to AU base pairings.

For the internal loop element, f_{i} takes in takes in two base pairings $(v_i, v_j), (v_{p_1}, v_{q_1}) \in \Gamma$, the unpaired nucleotides $v_{i+1}, v_{j-1}, v_{p_1-1}, v_{q_1+1}$, and the length of the left loop $ll = j - i$ as well as the length of the right loop $lr = q_1 - p_1$. The free energy $f_{\mathrm{i}}(v_i, v_j, v_{p_1}, v_{q_1}, v_{i+1}, v_{j-1}, v_{p_1-1}, v_{q_1+1}, ll, lr)$ contributed by the internal loop equals

$$\mathrm{mismatchI}[(v_i, v_j)][(v_{p_1}, v_{q_1})][v_{i+1}][v_{j-1}][v_{p_1-1}][v_{q_1+1}] + \mathrm{internal}[ll + lr] + |ll - lr| \cdot E$$

where internal and mismatchI are lookup tables with experimentally measured energies, and E is a penalty applied to imbalanced loops. Note that the above equation is a simplification – in the actual implementation the used lookup table mismatchI may vary based on ll and lr.

A.2 Recurrences for Structural Elements

Due to space constraints we omit the precise definitions of the recurrences of the various structural elements.

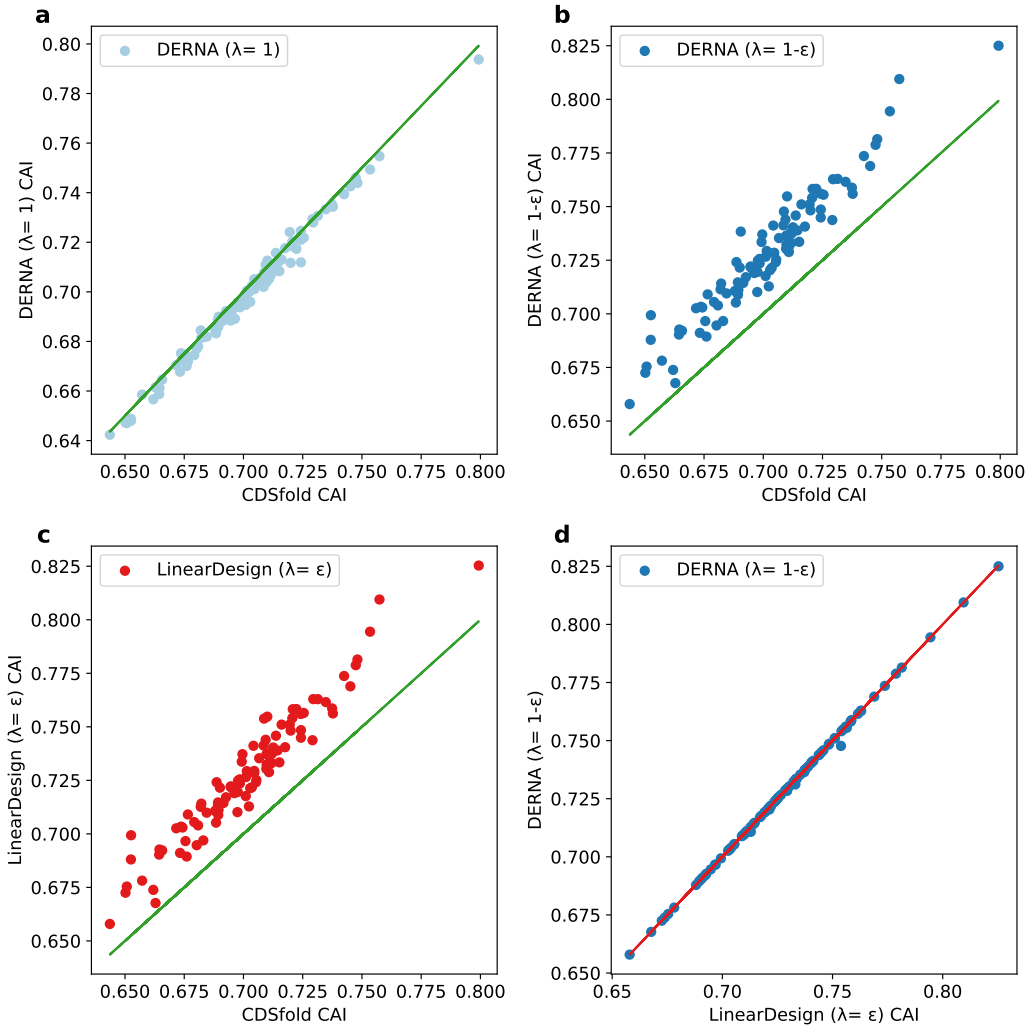


Figure S1 Comparison of CAI values for the benchmark dataset of 100 protein sequences. We compare CDSfold, DERNA with $\lambda = 1$, DERNA with $\lambda = 1 - \epsilon$ and LinearDesign with $\lambda_{\text{LD}} = \epsilon$. (a) DERNA with $\lambda = 1$ performs slightly worse than CDSfold in terms of CAI. However, neither method optimizes for CAI. (b-c) DERNA with $\lambda = 1 - \epsilon$ and LinearDesign achieve better CAI values than CDSfold. (d) With the exception of one protein sequence (Q9HAE3), discussed in the main text, LinearDesign and DERNA achieve the same CAI value. For protein sequence Q9HAE3, LinearDesign achieves a better CAI of 0.754 vs 0.748 for DERNA, but this comes at the expense of MFE (LinearDesign: -369.4 vs. DERNA: -369.9).

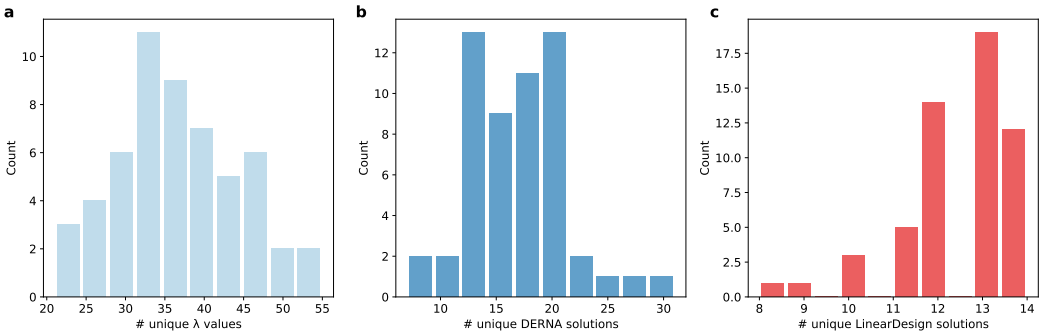


Figure S2 Distribution of (a) the number of unique λ values, (b) the number of unique solutions by DERNA, and (c) the number of unique solutions by LinearDesign for the dataset of 50 protein sequences.

Figure S3 DERNA identifies distinct mRNA sequences for SARS-CoV-2 S protein for (a) $\lambda = \epsilon$ (b) $\lambda = 1 - \epsilon$.

Bridging Disparate Views on the DCJ-Indel Model for a Capping-Free Solution to the Natural Distance Problem

Leonard Bohnenkämper
Faculty of Technology and Center for Biotechnology (CeBiTec), Bielefeld University, Germany

Abstract

One of the most fundamental problems in genome rearrangement is the (genomic) distance problem. It is typically formulated as finding the minimum number of rearrangements under a model that are needed to transform one genome into the other. A powerful multi-chromosomal model is the Double Cut and Join (DCJ) model.

While the DCJ model is not able to deal with some situations that occur in practice, like duplicated or lost regions, it was extended over time to handle these cases. First, it was extended to the DCJ-indel model, solving the issue of lost markers. Later ILP-solutions for so called *natural genomes*, in which each genomic region may occur an arbitrary number of times, were developed, enabling in theory to solve the distance problem for any pair of genomes. However, some theoretical and practical issues remained unsolved.

On the theoretical side of things, there exist two disparate views of the DCJ-indel model, motivated in the same way, but with different conceptualizations that could not be reconciled so far.

On the practical side, while the solutions for natural genomes typically perform well on telomere to telomere resolved genomes, they have been shown in recent years to quickly loose performance on genomes with a large number of contigs or linear chromosomes. This has been linked to a particular technique increasing the solution space superexponentially named *capping*.

Recently, we introduced a new conceptualization of the DCJ-indel model within the context of another rearrangement problem. In this manuscript, we will apply this new conceptualization to the distance problem. In doing this, we uncover the relation between the disparate conceptualizations of the DCJ-indel model. We are also able to derive an ILP solution to the distance problem that does not rely on capping and therefore significantly improves upon the performance of previous solutions for genomes with high numbers of contigs while still solving the problem exactly. To the best of our knowledge, our approach is the first allowing for an exact computation of the DCJ-indel distance for natural genomes with large numbers of linear chromosomes.

We demonstrate the performance advantage as well as limitations in comparison to an existing solution on simulated genomes as well as showing its practical usefulness in an analysis of 11 *Drosophila* genomes.

2012 ACM Subject Classification Applied computing → Bioinformatics

Keywords and phrases Comparative Genomics, Genome Rearrangement, Double-Cut-And-Join, Indels, Integer Linear Programming, Capping

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.22

Supplementary Material *Text*: `https://doi.org/10.6084/m9.figshare.23552247.v1`
Software: `https://gitlab.ub.uni-bielefeld.de/gi/ding-cf`
archived at `swh:1:dir:d01607c38f4a3c75816eaca253b5a4f7131143ac`
Software: `https://gitlab.ub.uni-bielefeld.de/gi/ffs-dcj`
archived at `swh:1:dir:e727f21413734ee5cde8d042ae941fc537099eef`

Acknowledgements I thank my supervisor Jens Stoye and Marília D. V. Braga for their helpful comments in discussions regarding notation, terms and overall structure of the paper. I furthermore thank Daniel Doerr for making me switch from water to lava and all members of the genome informatics group of Bielefeld University for their advice on how to present my research in this manuscript.

1 Introduction

In genome rearrangement studies, genomes are analyzed on a high level. Most often, the basic unit used is therefore not nucleotides, but oriented genetic *markers*, such as genes. The most fundamental problem in theoretical studies of genome rearrangements is the *distance problem*, which asks to provide the minimum number of rearrangements needed to transform one genome into the other under a restricted set of operations, also called a *model*.

In early approaches, such as the inversion model [12], solutions to the distance problem focused primarily on unichromosomal data, in which each marker appeared exactly once in each genome. These assumptions limited the applications of the models to real biological data, which often contained multiple chromosomes and a wide variety of marker distributions. Since then, researchers have sought to enable models to handle more realistic data. A major breakthrough was the DCJ-model introduced by Yancopoulos et al. in 2005 [18], a simple model that was nonetheless capable of handling multiple chromosomes. In 2010, Braga, Willing and Stoye extended the DCJ-model to the DCJ-indel model, enabling it to handle markers unique to one genome [5]. An independent, equivalent conceptualization of the same DCJ and indel operations was developed by Compeau in 2012 [7], although the precise relationship of the two conceptualizations remained unclear [8]. We refer to these views as the BWS- and Compeau-conceptualization respectively.

In 2020, the BWS-conceptualization was combined with previous results by Shao et al. [17] in [4] to yield the performant ILP solution `ding` for genome pairs with arbitrary distributions of markers, the so called *natural genomes*. In theory, `ding` enables the computation of the rearrangement distance between any pair of genomes available today.

However, `ding` uses a technique known as *capping*, which transforms linear chromosomes into circular ones during solving time. As described in [15], capping increases the solution space of ILPs like `ding` super-exponentially in the number of linear chromosomes. Since many assemblies available today are not resolved on a chromosome level and instead fragment into sometimes thousands of contigs, this renders distance computation infeasible yet again for many available genomes today. In [15], Rubert and Braga develop a heuristic solution to reduce the search space spanned by capping. Nonetheless, no exact solutions for the DCJ-indel distance problem of natural genomes avoiding capping exist as of yet.

In this work, we apply a new view on the DCJ-indel model developed in [3] to the distance problem. Using this, we are able to bridge the gap between the BWS- and Compeau conceptualizations in Section 3.1. Furthermore, this new conceptualization lends itself to a distance formula which is simple enough to be developed into a capping-free ILP (Section 4), which we then evaluate in Section 5 to show its performance advantage over `ding`.

2 Problem Definition

For this work, we use the same notation as in our previous work. Therefore large parts of this section are adapted from [3]. We conceptualize a genome $\mathbb{G}$ as a graph $(X_\mathbb{G}, M_\mathbb{G} \cup A_\mathbb{G})$. Its vertices $X_\mathbb{G}$ are the beginnings m^t and ends m^h of markers $m := \{m^t, m^h\} \in M_\mathbb{G}$, which we refer to as *extremities*. The genome's *adjacencies* $A_\mathbb{G}$ are undirected edges $\{m^x, n^y\} \in A_\mathbb{G}$, which signify that the extremities m^x and n^y are neighboring on the same chromosome. As a shorthand notation, we write ab for an adjacency $\{a, b\}$. Both $A_\mathbb{G}$ and $M_\mathbb{G}$ are required to form a matching on $X_\mathbb{G}$.

Because of that requirement, each path in $\mathbb{G}$ is simple and alternates between markers and adjacencies. A component of a genome is thus either a linear or circular simple path. We refer to them as linear and circular *chromosomes* respectively. The extremities in which

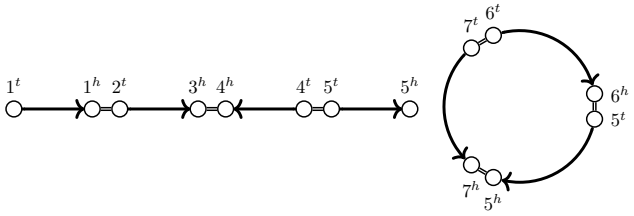


Figure 1 Genome of 7 markers with one linear and one circular chromosome. Markers drawn as arrows, adjacencies drawn as double lines.

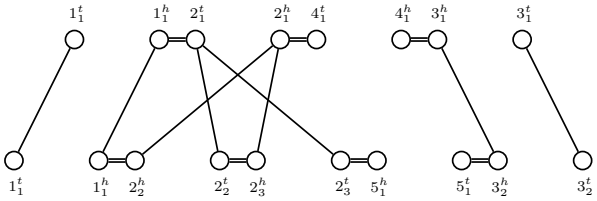


Figure 2 MRD for two genomes on an unresolved homology ($\equiv_1$) with families $\{1_1, 1_2\}$, $\{2_1, 2_2, 2_3\}$, $\{3_1, 3_2\}$, $\{4_1\}$, $\{5_1\}$.

a linear chromosome ends are called *telomeres*. Additionally, we refer to a subpath of a chromosome of which the first and last edge are markers as a *chromosome segment* (called a *marker path* in [3]). An example of a genome is given in Figure 1.

In our model, each marker is unique, thus there are no markers shared between genomes. Therefore, in order to calculate a meaningful distance between genomes, we borrow a concept from biology, namely *homology*. Homology can be modeled as an equivalence relation on the markers, i.e. $m \equiv n$ for some $m, n \in M_{\mathbb{G}}$. We call the equivalence class $[m]$ of a marker m its *family*. We also extend the equivalence relation to the extremities with $m^t \equiv n^t$ and $m^h \equiv n^h$ if and only if $m \equiv n$, but require no head being equivalent to any tail, i.e. $m^t \not\equiv n^h \forall m, n \in M_{\mathbb{G}}$. We can then extend the equivalence relation to adjacencies as $ab \equiv cd$ if and only if both of the extremities are equivalent, i.e. $a \equiv c \wedge b \equiv d$ or $a \equiv d \wedge b \equiv c$.

To illustrate our concept of homology, we introduce the Multi-Relational Diagram (MRD), a graph data structure here that is also useful for the distance computation. We deviate from the definition in [4] by omitting indel edges from our definition.

▶ **Definition 1.** *The MRD of two genomes $\mathbb{A}, \mathbb{B}$ and a homology relation ($\equiv$) is a graph $\mathcal{MRD}(\mathbb{A}, \mathbb{B}, \equiv) = (V, E)$ with $V = X_{\mathbb{A}} \cup X_{\mathbb{B}}$ and two types of edges $E = E_\gamma \cup E_\xi$, namely adjacency edges $E_\gamma = A_{\mathbb{A}} \cup A_{\mathbb{B}}$ and extremity edges $E_\xi = \{\{x, y\} \in X_{\mathbb{A}} \times X_{\mathbb{B}} \mid x \equiv y\}$.*

We give an example of a MRD in Figure 2. We see that in that example, 4_1 and 5_1 have no homologues in the other genome respectively. We refer to such markers as *singular*. Additionally, We call a circular or linear chromosome consisting only of singular markers a circular or linear *singleton*.

Note also that the family $\{2_1, 2_2, 2_3\}$ in this example has more than just one marker per genome. We call markers of such families *ambiguous*. We refer to a homology, in which no markers are ambiguous as *resolved*. In order to determine the precise nature of rearrangements occurring between two genomes, it is helpful to find a *matching* between the markers of two genomes.

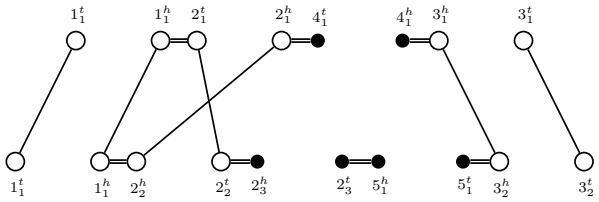


Figure 3 MRD for two genomes on a resolved homology ($\overset{\star}{\equiv}_1$) with families $\{1_1, 1_2\}$, $\{2_1, 2_2\}, \{2_3\}, \{3_1, 3_2\}, \{4_1\}, \{5_1\}$. Extremities of singular markers (called lava vertices from Section 3 onward) are filled black. ($\overset{\star}{\equiv}_1$) is a (maximal) matching on ($\equiv_1$) of Figure 2.

▶ **Definition 2.** *A* matching *($\overset{\star}{\equiv}$) on a given homology ($\equiv$) is a resolved homology for which holds $m \overset{\star}{\equiv} n \implies m \equiv n$ for any pair of markers m, n.*

We call two genomes $\mathbb{A}, \mathbb{B}$ *equal* under a homology ($\equiv$), if there is a matching ($\overset{\star}{\equiv}$) on ($\equiv$), such that each marker and adjacency of $\mathbb{A}$ has one equivalent in $\mathbb{B}$ under $\overset{\star}{\equiv}$ and vice versa.

We note that when the homology is resolved, in the MRD at most one extremity edge connects to each vertex. Because the adjacencies form a matching on the extremities, the resulting MRD consists of only simple cycles and paths. We therefore call such MRDs *simple*. We note that a simple MRD fits the definition of a simple rearrangement graph as studied in Section 3 of [3]. An example of a simple MRD is given in Figure 3.

Rearrangements in our transformation distance are modeled by the *Double-Cut-And-Join (DCJ)* operation. A DCJ operation applies up to two cuts in the genome and reconnects the incident extremities or telomeres. More formally, we can write as in [2]:

▶ **Definition 3.** *A* DCJ operation *transforms up to two the adjacencies $ab, cd \in A_{\mathbb{A}}$ or telomeres s, t of genome $\mathbb{A}$ in one of the following ways:*

- $ab, cd \to ac, bd$ *or* $ab, cd \to ad, bc$
- $ab \to a, b$
- $ab, s \to as, b$ *or* $ab, s \to bs, a$
- $s, t \to st$

To model markers being gained or lost, we introduce segmental insertions and deletions.

▶ **Definition 4.** *An* insertion *of length k transforms a genome $\mathbb{A}$ into $\mathbb{A}'$ by adding a chromosome segment $p = p_1, p_2, ..., p_{2k-1}p_{2k}$ to the genome. Note that this adds the markers $(p_1, p_2), ..., (p_{2k-1}, p_{2k}) \in M_{\mathbb{A}'}$. An insertion may additionally either add the adjacency $p_{2k}p_1 \in A_{\mathbb{A}'}$, apply the transformation $ab \to ap_1, p_{2k}b$ for an adjacency ab or the transformation $s \to p_1 s$ for a telomere s. A* deletion *of length k removes the chromosome segment $p = p_1, ..., p_{2k}$ and creates the adjacency ab if previously $ap_1, p_{2k}b \in A_{\mathbb{A}}$.*

We are now in a position to formulate the distance problem as finding a shortest transformation of DCJ and indel operations of one genome into the other.

▶ **Problem 5.** *Given two genomes, $\mathbb{A}, \mathbb{B}$ and a homology ($\equiv$), find a shortest sequence $s_1, ..., s_k$ of DCJ and indel-operations transforming $\mathbb{A}$ into a genome equal to $\mathbb{B}$. We call the length of k the DCJ-indel distance of $\mathbb{A}, \mathbb{B}$ under ($\equiv$) and write $d^{id}_{DCJ}(\mathbb{A}, \mathbb{B}, \equiv) = k$.*

The original DCJ-indel model by Braga et al. [6] only allowed indels on chromosome segments of singular markers to avoid scenarios that deleted and reinserted whole chromosomes. For a resolved homology $\overset{\star}{\equiv}$, we call $\overline{d^{id}_{DCJ}}(\mathbb{A}, \mathbb{B}, \overset{\star}{\equiv})$ the *restricted* DCJ-indel distance if we allow only indels of segments comprised solely of singular markers in scenarios in Problem 5.

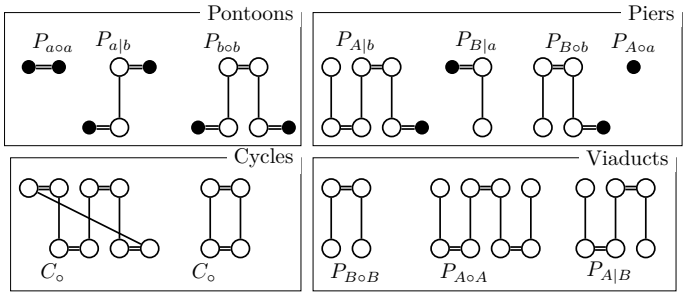


Figure 4 All different types of components in a simple MRD. Vertices of genome $\mathbb{A}$ are on the top, vertices of genome $\mathbb{B}$ are on the bottom of each component. Lava vertices are filled black.

For unresolved homologies, we can apply the same model by just finding a matching on the original homology. However, in order to not create a similar "free lunch" issue, we restrict ourselves to an established model, the *Maximum Matching model* [11]. We call a matching ($\overset{+}{\equiv}$) on a homology ($\equiv$) *maximal* if it has at most one singular marker for every family in ($\equiv$).

▶ **Problem 6.** *Given two genomes, $\mathbb{A}, \mathbb{B}$ and a homology ($\equiv$), find a maximal matching ($\overset{+}{\equiv}$) on ($\equiv$), such that $\overline{d^{id}_{DCJ}}(\mathbb{A}, \mathbb{B}, \overset{+}{\equiv})$ is minimized.*

3 A New DCJ-Indel Distance Formula

We note that the only maximal matching on a resolved homology ($\overset{\star}{\equiv}$) is ($\overset{\star}{\equiv}$) itself. Thus, for resolved homologies, in any scenario for Problem 6, we know deletions can only affect singular markers. Let us now regard the MRD of a pair of genomes $\mathbb{A}, \mathbb{B}$ for a resolved homology ($\overset{\star}{\equiv}$). Since each marker has at most one homologue, each vertex is connected to at most one extremity edge. Since adjacency edges form a matching on the vertices, again, the graph consists only of simple cycles and paths. All cycles are even and we write the set of cycles as $\mathbb{C}_\circ$. Paths can end either in a vertex without an extremity edge or adjacency edge. We name the vertices, in which a path ends in its *endpoints*. Vertices without extremity edges are special, because, as we established earlier, they are the extremities of the markers that will be part of indels during the sorting. We therefore name them *lava vertices*. The other type of vertex is a vertex without an adjacent adjacency edge, i.e. a *telomere*. Note that there is a special case wherein a lava vertex can also be a telomere. We can then identify different types of paths by their endpoints. We write a or b for a lava vertex and A or B for a telomere, depending on whether its part of genome $\mathbb{A}$ or $\mathbb{B}$. We then obtain a partition of paths into 10 different subsets, namely $\mathbb{P}_{A\circ A}, \mathbb{P}_{A|B}, \mathbb{P}_{B\circ B}, \mathbb{P}_{A\circ a}, \mathbb{P}_{A|b}, \mathbb{P}_{B|a}, \mathbb{P}_{B\circ b}, \mathbb{P}_{a\circ a}, \mathbb{P}_{a|b}, \mathbb{P}_{b\circ b}$. In order to be consistent with [3], we use $\circ$ and $|$ to distinguish even and odd paths respectively. Furthermore, we write $p_{x(*)y}$ as a shorthand for the cardinality of $\mathbb{P}_{x(*)y}$ and $P_{x(*)y}$ for a generic example of an element of $\mathbb{P}_{x(*)y}$.

Usually it is not necessary to think of all 10 different sets as separate entities, because they behave very similarly with respect to applied DCJ or indel operations. In textual form we therefore often use a coarser distinction, naming paths with two lava vertices as *pontoons*, paths with a telomere and a lava vertex as *piers* as well as paths with two telomeres as *viaducts*. An overview of this notation is given in Figure 4.

Another notation, we adopt from [3] is for a DCJ $ab, cd \to ac, bd$ affecting the adjacencies ab and cd in components K_{ab}, K_{cd} of the MRD respectively, we can instead view the DCJ as $K_{ab}, K_{cd} \to K_{ac}, K_{bd}$ transforming the components K_{ab}, K_{cd} into K_{ac}, K_{bd}. In combination

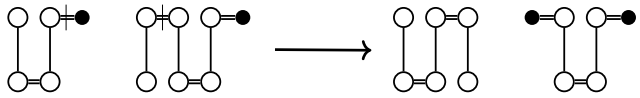


Figure 5 An example of a DCJ operation that can be written as $P_{A\circ a}, P_{B|a} \to P_{A|B}, P_{a\circ a}$.

with the generic member notation from above, we can write operations abstractly like so: $P_{A\circ a}, P_{B|a} \to P_{A|B}, P_{a\circ a}$. For reference, we have also shown this DCJ operation in Figure 5. Based on this notation and with the help of observations from [3], it is possible to derive a distance formula. We do so in detail in Supplement S.1. However, this formula is equivalent to that of Compeau and BWS as we will see in the following subsection. We thus only state it here.

▶ **Theorem 7.** *For two genomes $\mathbb{A}, \mathbb{B}$ and a resolved homology ($\overset{\star}{\equiv}$) for which both genomes contain no circular singletons, we have the distance formula*

$$\overline{d^{id}_{DCJ}}(\mathbb{A}, \mathbb{B}, \overset{\star}{\equiv}) = n - c_\circ + \left\lceil \frac{p_{a|b} + \max(p_{A\circ a}, p_{B|a}) + \max(p_{A|b}, p_{B\circ b}) - p_{A|B}}{2} \right\rceil$$

with n the number of matched markers*, $n = |\{(m, n) \in M_\mathbb{A} \times M_\mathbb{B} \mid m \overset{\star}{\equiv} n\}|$.*

Note that the constraint to genomes without circular singletons constitutes no serious restriction, as Compeau showed that circular singletons each require one indel operation and can thus be dealt with in pre-processing [8].

To more easily address individual terms in the formula, we use the followig shorthands,

$$\begin{aligned} F := n - c_\circ + \tilde{P} &:= n - c_\circ + \left\lceil \frac{\tilde{p}}{2} \right\rceil \\ &:= n - c_\circ + \left\lceil \frac{p_{a|b} + \max(p_{A\circ a}, p_{B|a}) + \max(p_{A|b}, p_{B\circ b}) - p_{A|B}}{2} \right\rceil. \end{aligned}$$

3.1 Relation of the BWS- and Compeau-Conceptualization

We now examine how the terms in our distance formula relate to both the Compeau- and BWS-conceptualizations of the DCJ-indel model. In doing that, we uncover the nature of the relation between these two views that have been perceived as entirely separate since their conception [8].

Braga et al. [6] and Compeau [8] use the adjacency and breakpoint graphs respectively. Both graphs are strongly related to the MRD. In fact, one obtains the adjacency graph by collapsing all adjacency edges of a simple MRD and the breakpoint graph by collapsing all its extremity edges. In order to avoid confusion, we will present their results here as if they had been formulated on a simple MRD. When consulting the original works in [6, 8], the reader should keep this in mind. Particularly in [8] the length of a path is determined by its adjacency edges instead of by its extremity edges as defined here. Therefore, parities of viaducts and pontoons are exactly opposite in [8] to the ones stated here.

We will compare the models by examining the chromosome segments that are deleted or inserted (see Definition 4), which we refer to as *indel groups*. We say an adjacency ab or its extremities a, b are *part of an indel group* p if there is $a'b' \equiv ab$ with $a'b' \in p$ or p starts and ends in a' and b'. In terms of indel groups, our view is closely related to the BWS-conceptualization, because both create the indel groups implicitly during sorting (see Supplement S.1). In terms of the graph, our conceptualization is more closely related to

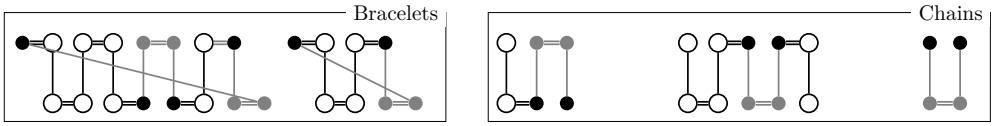


Figure 6 Components resulting from a completion as in [8]. Vertices and Edges added during completion are colored in grey.

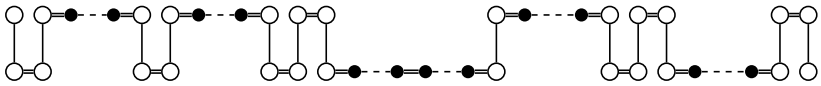


Figure 7 Path as found in [4] as another way of writing the paths in [6] by adding indel edges between lava vertices of the same gene. Indel edges here drawn in dashed. In this work, indel edges are omitted and the collection of components arising is called a *bridge*.

Compeau's because it essentially operates on the same type of components (lava vertices are called *open* in [8], piers are π- and γ-paths and pontoons are $\{\pi,\pi\}$-, $\{\pi,\gamma\}$- and $\{\gamma,\gamma\}$-paths). However, in [8], indels are not modeled as an explicit operation, but instead emulated by integrating or excising artificial circular chromosomes during sorting. Adding the correct chromosomes, the *completion*, is therefore the main problem solved in [8]. These additional chromosomes are then the explicitly constructed indel groups in the sorting. Because the homology of the markers needed for the completion is already known beforehand on a resolved homology, the task is to find the correct new adjacencies to add to the graph. Then, if an adjacency $a'b'$ is found in the completion, the extremities $a \equiv a', b \equiv b'$ of the originally singular markers will be part of the same indel group. Once the completion is constructed, there are no more lava vertices in the graph. Instead, former piers and pontoons are joined into new components, either *bracelets*, which are circular and consist of pontoons only, or *chains*, which consist of two piers and possibly pontoons. An example of a completion can be found in Figure 6.

In [6], lava vertices are avoided by viewing singular markers as part of adjacencies of matched markers, called $\mathcal{G}$*-adjacencies*. This is equivalent to connecting the head and tail vertex of a singular marker with a special type of edge, called *indel edge* as is done in [4]. We call a collection of components that can be traversed as one once indel edges are introduced a *crossing*. We distinguish between circular crossings called *ferries* and linear crossings called *bridges*.

▶ **Definition 8.** *A* pontoon bridge $b_1, .., b_k$ *for* $k \geq 2$ *is a string of components* b_i*, such that* b_1, b_k *are piers,* $(b_i)_{i=2}^{k-1}$ *are pontoons and there are singular markers* $(m_i)_{i=1}^{k-1}$ *with* $m_i \neq m_j$ *for* $i \neq j$ *whose extremities are contained as lava vertex in* b_i, b_{i+1} *for all* m_i*. A string of components is called a* bridge *if it is a pontoon bridge or consists of a single viaduct.*

▶ **Definition 9.** *A* pontoon ferry $f_1, ..., f_l$ *for* $l \geq 1$ *is a string of pontoons* f_i*, such that here are singular markers* $(m_i)_{i=1}^{l}$ *with* $m_i \neq m_j$ *for* $i \neq j$ *whose extremities are contained as lava vertices in* f_i, f_{i+1} *for all* m_i *for* $i < l$ *and the extremities of* m_l *are contained in* f_1 *and* f_l*. A string of components is called a* ferry *if it is a pontoon ferry or consists of a single cycle.*

Ferries and bridges are cycles and paths in [6] respectively. An example of a bridge can be found in Figure 7. Crossings are first sorted separately in [6], so we start our comparison by doing the same. We thus aim to find *internal* operations that only involve components of the same crossing. During sorting, we want to make sure that the operations we apply are not only optimal in the context of the crossing, but in the graph as a whole. There are

Figure 8 Safe DCJ operations accumulating markers separated by even pontoons (in [6] called a *run*) remain optimal in the safe bracelet joining the extremities of these markers.

certain operations that are guaranteed to be optimal because they reduce F in any MRD by 1, no matter which other components are found in the graph. We call such an operation *safe*. For example, extracting a cycle from any component is safe (as $\Delta c_\circ = 1$), whereas recombining two even piers, such as $P_{A\circ a}, P_{A\circ a} \to P_{A\circ A}, P_{a\circ a}$ is not safe, because it is only optimal under the premise that $p_{A\circ a} > p_{B|a}$. There are only 7 distinct types of safe DCJ operations. We list them in Table 1. We also note that as in [6], instead of sorting $\mathbb{A}$ to $\mathbb{B}$, we can sort both $\mathbb{A}$ and $\mathbb{B}$ to a common genome. By thinking this way, we can better exploit the symmetry of the situation.

Table 1 All safe types of DCJ operations. Each reduces the F by 1, no matter the number of other components in the graph. Above are all safe operations in a pure DCJ scenario. The operations below can also function as safe deletions if one of the resultants in brackets is removed. For reference: $F = n - c_\circ + \left\lceil (p_{a|b} + \max(p_{A\circ a}, p_{B|a}) + \max(p_{A|b}, p_{B\circ b}) - p_{A|B})/2 \right\rceil$.

Safe operation	$-\Delta c_\circ$	$\Delta p_{a\|b}$	$\Delta \max(p_{A\circ a}, p_{B\|a})$	$\Delta \max(p_{A\|b}, p_{B\circ b})$	$-\Delta p_{A\|B}$
$K \to K' + C_\circ$	-1	0	0	0	0
$P_{A\circ A} \to P_{A\|B}, P_{A\|B}$	0	0	0	0	-2
$P_{B\circ B} \to P_{A\|B}, P_{A\|B}$	0	0	0	0	-2
$P_{A\circ A}, P_{B\circ B} \to P_{A\|B}, P_{A\|B}$	0	0	0	0	-2
$P_{a\|b}, P_{a\|b} \to (P_{a\circ a})^*, (P_{b\circ b})^*$	0	-2	0	0	0
$P_{A\circ a}, P_{B\|a} \to P_{A\|B}, (P_{a\circ a})^*$	0	0	-1	0	-1
$P_{A\|b}, P_{B\circ b} \to P_{A\|B}, (P_{b\circ b})^*$	0	0	0	-1	-1

The most obvious safe operation is the extraction of an even cycle from another component. If one continues to extract even cycles from an even pontoon $p = x_1 ... x_k$, one arrives at the pontoon $p' = x_1 x_k$, which consists of a single adjacency. The corresponding markers can then be dealt with with the same indel operation, meaning x_1, x_k are part of the same indel group. Braga et al. notice the same thing in [6]; they refer to markers that are only separated by even pontoons as a *run*, which they notice can be "accumulated" in this fashion. For an extensive example, see Figure S.2.1 in Supplement S.2, Steps (a), (b). In [8], genomes are not explicitly sorted, so there is no true equivalent to safe operations, but Compeau systematically finds chains and bracelets he can be sure are optimal in any breakpoint graph (Algorithm 9, Steps 1 to 3). We therefore call these chains and bracelets safe, too. In fact, the very first safe bracelet Compeau identifies, is a 1-bracelet consisting of a single even pontoon (Lemma 5 in [8]). If one creates this bracelet from the even pontoon $p = x_1 ... x_k$ the adjacency added for the completion is $x'_1 x'_k$ with $x_1 \equiv x'_1$ and $x_k \equiv x'_k$. Thus, here too, $x_1 x_k$ are part of the same indel group. This way of constructing the indel groups is shown in Supplement S.2, Figure S.2.2 with Bracelets (a), (b). Notice also that the safe operations sorting the two adjacent lava vertices of an even pontoon together remain optimal in a bracelet like this (see Figure 8).

The next safe bracelet Compeau finds, is joining two odd pontoons together. He shows that it is safe by ruling out all other uses of two pontoons as at best co-optimal (Lemma 6, Proof of Thm 8 and Step 2 of Algorithm 9 in [8]). An example can be found as Bracelet (c) of Figure S.2.2. This again, corresponds to a safe operation, namely $P_{a|b}, P_{a|b} \to P_{a\circ a}, P_{b\circ b}$. In fact, all safe chains and bracelets of two components correspond directly to safe operations. We

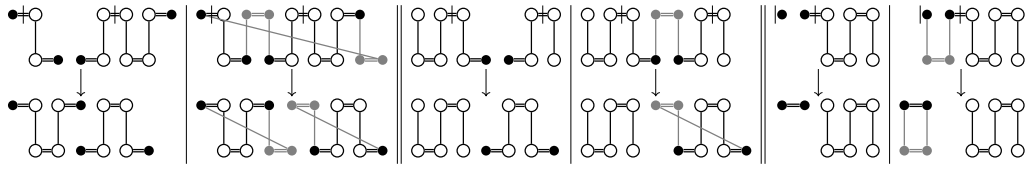


Figure 9 For all safe DCJ operations with two piers or pontoons as sources, there is a safe bracelet or chain in which the same operation is optimal and vice versa.

have visualized this fact in Figure 9. Note that the corresponding safe operation again remains optimal in the safe chain or bracelet. Because of this more direct correspondence between the Compeau-conceptualization and our formula, we focus more on the correspondence between the BWS-conceptualization and our formula in the following. Braga et al. identify the same operation by noticing that the number of runs can be reduced by 2 if one applies cuts in between between runs of $\mathbb{A}$ and $\mathbb{B}$ (see Proposition 3 in [6]). This is of course precisely a DCJ with two odd pontoons as sources in our model. Because the resultants of this operation are the two even pontoons $P_{a\circ a}, P_{b\circ b}$, these can in turn be reduced to single adjacencies by excising even cycles. Again, the implication for indel groups in all models is that for two odd pontoons $p_1 = a_1x_1, ..., x_kb_1, p_2 = a_2x_{k+1}, ..., x_lb_2$, the adjacency a_1a_2 can be part of the same indel group if b_1b_2 is part of the same indel group and vice versa. This equivalence is further illustrated by comparing the effects of Steps (c) and (d) of Figure S.2.1 to Bracelet (c) of Figure S.2.2 of Supplement S.2.

Dealing in this fashion with all pontoons of a crossing, we reduce all but possibly one odd pontoon to single adjacency edges, which can then be dealt with in a single indel operation. Because ferries must contain an even number of odd pontoons, they can be sorted entirely by safe operations in this way. To quantify the number of operations needed, Braga et al. define the *indel potential* $\lambda(X)$ of a crossing X as the number of indel operations obtained in a DCJ-optimal sorting [6]. Since it is possible to trade off indel and DCJ operations, this definition is not easily reflected in the other conceptualizations. However, as they show that sorting a crossing X separately needs $d^{id}_{DCJ}(X) = \mathrm{d}_{DCJ}(X) + \lambda(X)$ steps, we can also think of the indel potential as the overhead introduced by the singular markers if we sort the crossing separately. In [6], it is shown that $\lambda(X) = \left\lceil \frac{\Lambda(X)+1}{2} \right\rceil$ with $\Lambda(X)$ the number of runs for a crossing X. If a ferry contains at least two runs, we can find a bijection between runs and odd pontoons. Denoting $q(X)$ as the contribution to quantity q by crossing X. We can thus write $\Lambda(X) = p_{a|b}(X)$ for a ferry with at least two runs. Therefore, we find for a ferry X with at least two runs, their formula translates to ours,

$$\begin{aligned} n(X) - c(X) + \lambda(X) &= n(X) - c(X) + \left\lceil \frac{\Lambda(X)+1}{2} \right\rceil \\ &= n(X) - 1 + \frac{\Lambda(X)+2}{2} = n(X) + \frac{\Lambda(X)}{2} = n(X) + \left\lceil \frac{p_{a|b}(X)}{2} \right\rceil. \end{aligned}$$

Similarly, this equivalence can be shown if there is only 1 run in X. By the Compeau method, if there are d singular markers, d markers are added as part of completion chromosomes, so the number of markers after completion is $N = n + d$. Meanwhile, each $P_{a\circ a}$ and $P_{b\circ b}$ creates a bracelet. Each pair $P_{a|b}, P_{a|b}$ also forms a bracelet. Since $d = p_{a|b} + p_{a\circ a} + p_{b\circ b}$, we have

$$n(X) + \left\lceil \frac{p_{a|b}(X)}{2} \right\rceil = n(X) + \frac{p_{a|b}(X)}{2} = n(X) - \frac{p_{a|b}(X)}{2} + p_{a|b}(X)$$
$$= n(X) + d(X) - \frac{p_{a|b}(X)}{2} - p_{a\circ a}(X) - p_{b\circ b}(X)$$
$$= N(X) - (p^{\pi,\pi}(X) + p^{\gamma,\gamma}(X) + \left\lfloor \frac{p^{\pi,\gamma}(X)}{2} \right\rfloor),$$

which is precisely the Compeau formula if no piers or viaducts are involved. We see that our formula acts as a sort of missing link between the two other formulas here. Since ferries can be dealt with entirely with internal safe operations, this formula can even be generalized to the whole graph for circular genomes. In fact, this has been done in [4], yielding our formula for this specific case.

Using this way of examining the contribution of individual crossings, we were also able to re-calculate the indel potential with our formula for all 10 types of bridges in [6]. The results can be found in Table S.2.1 of Supplement S.2. Notably, when sorting a bridge independently, one can also first exhaust all safe operations. After this, only the piers and possibly a single odd pontoon might be "left over" (see also Figure S.2.1 after Step (d)). We call these components *unsaturated*. Since each safe operation also has a corresponding safe chain or bracelet, these are also the only components, which end up in unsafe chains if one restricts the completion to a single crossing (compare to Figure S.2.2). Since every other component can be dealt with safe operations, unsaturated components are the only ones that might have to be involved in what is called in [6] a *(path) recombination*, that is, a DCJ operation going beyond a single crossing. When studying recombinations, we can therefore abstract from any concrete bridge $p = p_1, ..., p_k$ with piers p_1, p_k and only write it as its unsaturated components, that is $p_1 p_k$ if p contains an even number of odd pontoons or as $p_1 P_{a|b} p_k$ otherwise. We call this the *reduced bridge*. Interestingly, Braga et al. make the same abstraction and identify the bridges by the genome of their telomeres and the genome of the first and last run. This direct correspondence is illustrated by comparing Columns 1 and 4 of Table S.2.1. In [6], another observation is that (reduced) bridges of the type $P_{A|b}, P_{B\circ b}$ or $P_{A\circ a}, P_{B|a}$ never need to appear as sources for any recombination. Using our conceptualization, we can confirm that because $P_{A|b}, P_{B\circ b} \to P_{A|B}, P_{b\circ b}$ and $P_{A\circ a}, P_{B|a} \to P_{A|B}, P_{a\circ a}$ are safe operations, these types of bridges can be sorted entirely by internal safe operations. It therefore makes sense to group them as in [6] with viaducts, the other type of bridge that can be sorted in this way.

All other bridges might need recombinations to be sorted optimally. If there is a safe operation between the components of two bridges, we know that this recombination must be optimal. In fact, if we only regard unsaturated components, we see that the only remaining safe operations are (i) $P_{A\circ a}, P_{B|a} \to P_{A|B}, P_{a\circ a}$, (ii) $P_{A|b}, P_{B\circ b} \to P_{A|B}, P_{b\circ b}$ and (iii) $P_{a|b}, P_{a|b} \to P_{a\circ a}, P_{b\circ b}$. We know (either by combinatorics or Table S.2.1) that each source of (i) and (ii) appears in 3 types of (reduced) bridges and thus there are $3 \times 3 = 9$ path recombinations facilitated by each of these two safe operations. For (iii), we have 4 types of (reduced) bridges containing $P_{a|b}$ and thus $\binom{4}{1} + \binom{4}{2} = 10$ path recombinations using this operation. Of course, these are not mutually exclusive, but since Operations (i) and (ii) involve the end of a bridge and one of its resultants is a viaduct, we can always choose to do one of these operations first, upon which all other possible safe operations on piers and pontoons will be in the same component and will not require any further recombinations. In [6] all of these recombinations are catalogued. We were able to confirm this by recreating their tables of recombinations with $\Delta d \leq 0$ as Table S.2.2 of Supplement S.2. It is easily

checked that (i) and (ii) each occur 9 and (iii) occurs 10 times. The unsaturated components after the operation in these cases form precisely those bridges listed in [6] as the resultant(s). The precise difference for the distance as opposed to sorting the crossing separately can then be derived by comparing the term $\tilde{P}$ in our formula on the graphs containing each bridge separately and on a graph containing the union of the two bridges (see Table S.2.2 Columns 3, 6, 9, 10). In summary, we can see that in all but two cases, the DCJ chosen to recombine the bridges in [6] is safe and the resultants are exactly comprised of the unsaturated components after the operation.

The two exceptions are the recombinations of $P_{A\circ a}, P_{A\circ a}$ with $P_{B\circ b}, P_{B\circ b}$ and $P_{A|a}, P_{A|b}$ with $P_{B|a}, P_{B|a}$ (marked with $\star$ in the table). In these cases, there is no safe operation and therefore all piers remain unsaturated. The reason this recombination can still be done in some cases is that an unsafe operation like $P_{A\circ a}, P_{B\circ b} \to P_{A|B}, P_{a|b}$ in this specific case reduces F by one, but since there are equally optimal internal operations (i.e. $P_{A\circ a}, P_{A\circ a} \to P_{A\circ A}, P_{a\circ a}$) in this case, this recombination actually never has to be used. The only task remaining is then to find a sequence of recombinations that improve upon the distance. Braga et al. give this as their *recombination groups*. We have listed these groups in Table S.2.3 of Supplement S.2. The first observation is that by exhausting all safe DCJ operations in a recombination group, we are able to create the unsaturated components of what are called in [6] *reusable resultants*. In combination with our observations about pairwise recombinations, we thus know that all recombinations in the groups can be facilitated purely by safe DCJs. We also see that in many cases, after sorting a group, no further unsaturated components are present. In the other cases, Braga et al. make sure that all partners of the unsaturated components are "used up" in earlier recombinations of the table (see last column) such that the unsafe operations sorting the unsaturated components are still optimal.

4 Capping-free Generalization to Natural Genomes

In this section, we describe briefly how to generalize the distance formula presented as Theorem 7 into an ILP for which no capping of the MRD is required. For reference, the full ILP is given in Appendix A as Algorithm 1. The ILP works by determining a matching on the markers as described in Problem 6. The basic framework (Constraints `C.01` to `C.06`) is the same as for `ding` [4] and the ILP by Shao et al. [17]: Variable x is used to indicate whether or not an edge is part of the matching and z_v marks the vertex v with the lowest index $\texttt{ix}(v)$ in its component. We also adopt the way circular singletons are dealt with in [4] as Constraint `C.18`. The only major change we make w.r.t. [4] in Constraints `C.01` to `C.06` is the addition of Constraint `C.02`, where we allow for different matching models than the maximum matching model by specifying an upper (U_f) and lower bound (L_f) for the number of markers to be matched per family f. Specifications for how to set these bounds to achieve the maximum matching model and other popular models can be found in Table 2 of Appendix A. Another minor change is that we have a variable $d_{g(u)}$ indicating whether a gene $g(u)$ of an extremity u is to be singular instead of indel edges as in [4].

To distinguish different types of paths, we use binary variables to track endpoints, namely $n_v^a, n_v^b, n_v^A, n_v^B$ as well as $m_v^a, m_v^b, m_v^A, m_v^B$ for each vertex v. A variable n_v^i is used to represent the sub-path starting with the adjacency edge at v ending in $i \in \{A, B, a, b\}$ while the m_v^i does the same, just for the sub-path starting with the extremity edge at v. We set these variables accordingly at the end of a path (see Constraints `C.07`, `C.08`). We also require only one of the variables be set per vertex (`C.09`). The variables are then required to be the same if their respective vertices are connected by the respective edge (see Constraint `C.10`), "passing" the label through the edge. Lastly, we require the m and n variables of a vertex

to be equal unless that vertex is labeled by z_v, i.e. it has the smallest identifier in the component (`C.11`). If z_v is set, we then report the component type based on the adjacent sub-paths in r_v^{ij} if the sub-paths have the correct labels m_v^i and n_v^j (or m_v^j and n_v^i) set (Constraints `C.13`, `C.14`). We reserve the special case $m_v^i = 0 \forall i \wedge n_v^j = 0 \forall j$ for cycles and report cycles via r_v^c in these constraints. The remaining constraints deal with applying the maximization function (`C.15`, `C.16`) for $\tilde{p}$ and calculating $\tilde{P}$ as variable q (`C.17`).

As an additional optimization, we set m variables to 0 in those components of the MRD that have either no telomere or indels of a given type in Constraint `C.19`.

5 Evaluation of the ILP

We implemented the ILP described in the previous section and made it publicly available here: `https://gitlab.ub.uni-bielefeld.de/gi/ding-cf`. We refer to this implementation as `ding-cf` for the rest of this work.

In this section, we show results of applying the ILP to both simulated and real data and comparing its performance to the python3 version of `ding` [4], namely `dingII`, a similar ILP solution to the DCJ-indel distance problem for natural genomes. In contrast to `ding-cf`, `dingII` uses the capping technique.

We first test the ILPs on simulated data in Subsection 5.1 before demonstrating the practical usefulness of rearrangement analyses even on contig level resolved genomes by analysing 11 Drosophila genomes in Subsection 5.2.

5.1 Performance Evaluation on Simulated Data

We initially planned to use the simulation script that comes with `dingII`, but due to the script regularly encountering stack overflows on large genomes owing to its reliance on recursion, we instead re-implemented it in `C++`. This implementation is available at `https://gitlab.ub.uni-bielefeld.de/gi/ffs-dcj`.

The re-implementation has the same features as the original script with only two minor changes. Firstly, instead of the number of DCJ-operations to be performed being passed to the script with additional numbers of other operations (insertions, deletions and duplications) being randomly performed according to rate parameters, our simulation takes a fixed number of total operations and distributes them according to rates relative to a rate of 1 for DCJ operations. Secondly, our simulation is not yet able to simulate arbitrary trees, but instead only simulates the topology `(A,B);`, which was used in [4]. For more detail on the simulation, the interested reader is referred to the description of the original simulation script in [4].

In our experiments, we simulated two genomes from a common root for each sample. In all experiments, we set the length of the root genomes to 20,000 markers and performed 10,000 operations in total, with an insertion rate of 0.1 and an deletion rate of 0.2 unless specified otherwise. For reference, this amounts to 5882 DCJ operations in expectation for a duplication rate of 0.4 to compare to experiments run with the python script of `dingII`. The shape parameter for the Zipf distribution was set to 4 for indel lengths and to 6 for duplication lengths. In all experiments, we used `gurobi10.0` on a single thread on an AMD EPYC 7452 Processor to solve the ILPs, limiting its runtime to $1h$ ($3600s$). Both experiments were designed to test parameters to which ILPs like `ding` have been shown to be sensitive.

In our first experiment, we increased the duplication rate in steps of 0.1 from 0.1 to 1.1, generating 10 genome pairs from a root genome with 1 linear chromosome per step. We then created the ILPs for `dingII` and `ding-cf`. The number of ambiguous families ranged from 615 to 2760 (median 2708) in this experiment with the maximum family size per sample reaching up to 7 markers.

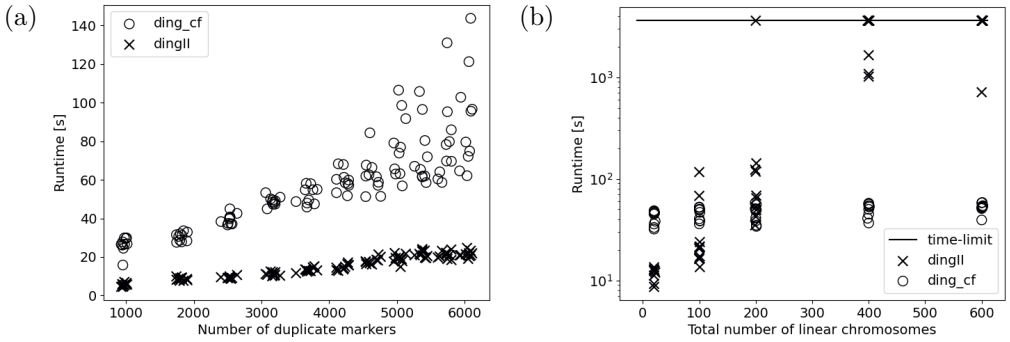


Figure 10 Runtimes for `dingII` and `ding-cf` for genomes simulated in 10,000 steps from a common root, in (a) increasing the duplication rate in steps of 0.1 from 0.1 to 1.1, in (b) increasing the number of linear chromosomes in the root genome progressively from 10 to 50 to 100, 200 and 300.

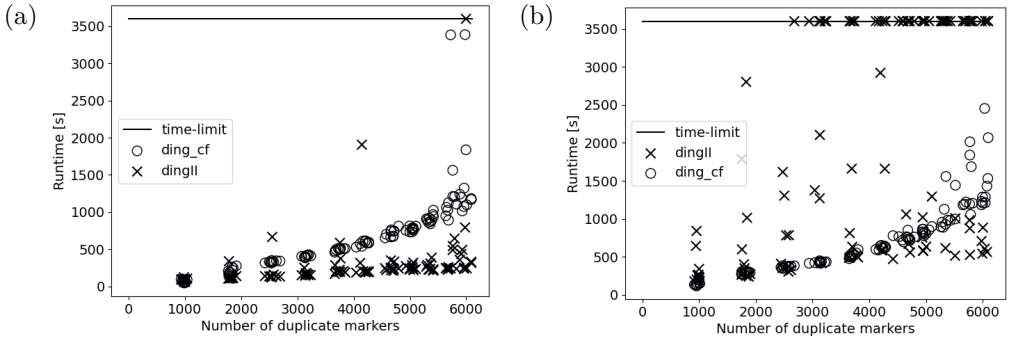


Figure 11 Runtimes for `dingII` and `ding-cf` for genomes simulated in 10,000 steps from a common root increasing the duplication rate in steps of 0.1 from 0.1 to 1.1 with (a) 100 total linear chromosomes and (b) 200 total linear chromosomes on average per sample pair.

We show the results in runtimes of `gurobi10.0` in Figure 10 (a). We see clearly that `dingII` has a performance advantage over `ding-cf` as long as the number of linear chromosomes is low. This is not surprising as `dingII` works in a very similar manner to `ding-cf`, but with fewer variables because it only needs to identify odd pontoons due to capping transforming other types of paths to pontoons and cycles. However, we see that the performance loss is not dramatic, staying well within a few minutes of solving time for this experiment. Nonetheless, the expected exponential increase in runtimes of `ding-cf` happens earlier than the one demonstrated for `ding` in [4]. We were able to further verify that on genomes with few linear chromosomes, `ding-cf` behaves similarly to `ding` for varying different parameters but having worse performance overall in Supplement S.3.

To test the actual use case for `ding-cf`, that is, high numbers of linear chromosomes, we increased the number of linear chromosomes in the root genome progressively from 10 to 50 to 100, 200 and 300 chromosomes with a fixed duplication rate of 0.4 and 10 samples per step. The runtimes are shown in Figure 10. We see that up to 100 linear chromosomes in the simulated pair of genomes, `dingII` on average outperforms `ding-cf`, but its runtime rises exponentially until the majority of the `dingII` ILPs are not solved within an hour of solving time. Meanwhile, the runtimes of `ding-cf` are stable throughout the experiments, staying below 100 seconds in each case.

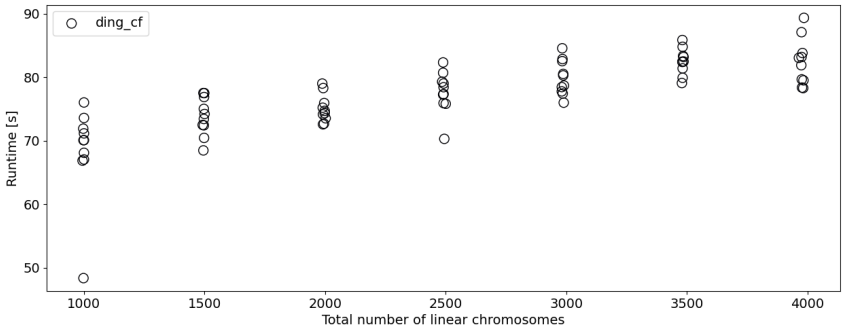


Figure 12 Runtimes for `ding-cf` for genomes simulated from a root with 500 to 2000 chromosomes in steps of 250.

In order to test the composite effect of the number of duplicates and the number of linear chromosomes on solving times, we repeated the first experiment (Figure 10 (a)) with 50 and 100 linear chromosomes at the root genome, resulting in total numbers of about 100 and 200 linear chromosomes for each pair. The results (shown in Figure 11) indicate that both ILPs react strongly to the combined effect for the first increase to 100 linear chromosomes with `dingII` still outperforming `ding-cf` on these samples whereas the second increase to 200 only has a minor effect on `ding-cf` while for `dingII` many of the pairs with high duplicate numbers become unsolvable within an hour.

To confirm that the number of linear chromosomes alone only plays a minor part in the runtime of `ding-cf`, we ran another experiment, this time keeping the duplication rate fixed at 0.4 and increasing the number of linear chromosomes in the root genome from 500 to 2000 in steps of 250 with 10 samples per step. The runtimes are given in Figure 12 and exhibit only a minor, linear increase. In fact, the increase is so slow that even for 2000 linear chromosomes at the root (c.a. 4000 linear chromosomes of the pair in total), the runtime is still below 100 seconds for all 10 samples.

5.2 Analysis of Drosophila Genomes

We obtained 11 assemblies of species in the *Drosophila* genus previously analyzed by Rubert and Braga [15]. We used `FFGC` to extract the longest transcript of each locus and ran `OrthoFinder` version 2.3.7 [10] to obtain orthologous groups. We then translated the genomes into unimog files using the orthogroups as families and translating linear contigs into linear chromosomes. We then filtered out any empty chromosomes. The genomes obtained in this fashion comprised 13,143 markers spread on 97 linear chromosomes on average. More detailed statistics about the genomes after this preprocessing step are listed in Table S.4.1 of Supplement S.4.

We then used `ding-cf` to calculate pairwise distances, running `gurobi10.0` on a single thread on an AMD EPYC 7452 Processor for 24 hours. Of the 55 resulting ILPs, we obtained an exact result for 9 and approximate results for 45, all of which deviated at less than 2% from the exact solution. Only one run, namely *D. melanogaster* vs *D. willistoni* did not yield any result within 24 hours. We therefore re-ran the solver on this ILP, this time using 15 threads and a time limit of 20 hours. In this run, an approximate solution with 0.48% gap was found. We give the distance data obtained in this manner in Table S.4.3 and detailed performance results in Table S.4.2 of Supplement S.4. Additionally we performed an experiment with the same parameters with the `dingII` ILP. Table S.4.2 shows that even though this dataset is not extremely fragmented, `ding-cf` outperforms `dingII` on the majority of samples.

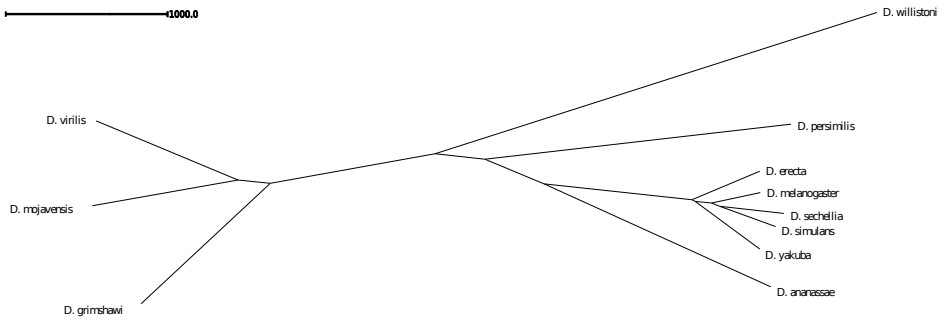


Figure 13 Neighbor joining tree inferred from the distances in Table S.4.3 using `SplitsTree4`. Edge lengths are drawn proportional to their weight. The absolute edge lengths can be found in Supplement S.4.

Phylogenetic Analysis. We proceeded to construct a phylogenetic tree via Neighbor Joining using `SplitsTree4` [13]. The tree, shown in Figure 13, is entirely consistent with the current state of knowledge about the Drosophila phylogeny. Additionally, the phylogenetic signal in the distance data is remarkably strong. To demonstrate this fact, we calculated the distance matrix for the path metric of the tree and compared it to the distances calculated by `ding-cf`. On average, the tree path metric deviates only by 0.53% per entry from the distances calculated by `ding-cf` with the largest relative difference being 2.2% for the distance of *D. melanogaster* and *D. simulans*. For reference, we give the full distance matrix of the path metric in Table S.4.4 of Supplement S.4. We were able to further confirm this strong correspondence between the tree and the distance data via a split decomposition with `SplitsTree4` in Supplement S.4.1 [1, 13]. Overall, judging from these experiments, `ding-cf` looks promising as a distance measure for phylogenetic analyses.

However, we want to draw the reader's attention to one possible pitfall of our method as a phylogenetic tool, namely that the fragmentation of the genome itself appears as a signal in the distance data. To emphasize this, let us pose a hypothetical extreme example: Consider a comparison between two assemblies $\mathbb{A}, \mathbb{B}$ with n markers each, with a matching between all markers of $\mathbb{A}$ and $\mathbb{B}$. Suppose $\mathbb{A}$ is fully assembled into one chromosome and $\mathbb{B}$ fragments into n contigs of one marker. No matter the actual structure of the underlying (true) genome of $\mathbb{B}$, the DCJ distance between the assemblies $\mathbb{A}$ and $\mathbb{B}$ is always $n - 1$. The size of this effect for practical levels of fragmentation needs to be investigated, particularly whether these problems could be exacerbated by biases in the assembly method used to arrive at the studied pair of genomes, such as might be the case for comparative assembly strategies.

Detecting Synteny. We extracted the matchings from the ILP solutions calculated by `gurobi` and plotted them with `Circos` [14]. We show the matching between *D. virilis* and *D. mojavensis* in Figure 14 as compared to just the marker matches identified by `OrthoFinder`. The plots for all other pairs can be found in Supplement S.4.2. We see that even though there are some big rearrangements, such as inversions and transpositions as indicated by the arcs as well as an abundance of duplicates, the calculated matching identifies large syntenic blocks, sometimes even matching the majority of markers of whole contigs to each other.

Moreover, for many of the smaller contigs all markers are matched to markers of exactly one large contig of the other species. Matchings like this could therefore possibly be used to aid in improving very fragmented assemblies, given a sufficiently closely related and resolved reference genome.

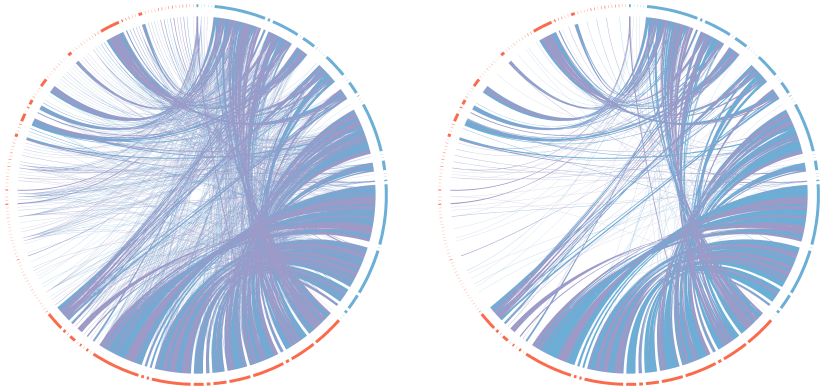


Figure 14 Circos plots for Contigs of *D. virilis* (red segments) and *D. mojavensis* (blue segments). Blue arcs show common markers with the same direction, purple arcs show common markers with different directions. On the left: before matching. On the right: after matching with `ding-cf`.

6 Conclusion

We presented a new, simpler distance formula for the DCJ-indel model. Using this distance formula, we were able to explain the previously unclear relationship between the BWS- and Compeau-conceptualizations of the DCJ-indel model. Furthermore, our formula is easily generalizeable to a performant ILP solution that enables the distance computation even for genomes fragmented into thousands of contigs. We have shown that a DCJ-indel analysis can be meaningful even with relatively fragmented genomes by applying the ILP to 11 *Drosophila* assemblies. From this we obtained a well resolved phylogeny with little noise in the distance data, indicating that our method could be well suited for distance based phylogenetic analyses provided the effect size of genome fragmentation in the particular use case can be bounded. We also showed that the ILP can be used to disambiguate orthologous and paralogous regions, which has potential use cases in orthology assignment and the finalization of fragmented assemblies.

Furthermore, we are confident that using this new formula, capping-free versions of other existing algorithms, such as for the family-free distance problem as in [16, 15] and parsimony problems as in [9] can be devised.

References

1 Hans-Jürgen Bandelt and Andreas 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.

2 Anne Bergeron, Julia Mixtacki, and Jens Stoye. A unifying view of genome rearrangements. In Philipp Bücher and Bernard M. E. Moret, editors, *Algorithms in Bioinformatics*, pages 163–173, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

3 Leonard Bohnenkämper. The floor is lava - halving genomes with viaducts, piers and pontoons. In *Comparative Genomics*. Springer International Publishing, to appear.

4 Leonard Bohnenkämper, Marília D.V. Braga, Daniel Doerr, and Jens Stoye. Computing the rearrangement distance of natural genomes. *Journal of Computational Biology*, 28(4):410–431, 2021. PMID: 33393848. doi:10.1089/cmb.2020.0434.

5 Marília D. V. Braga, Eyla Willing, and Jens Stoye. Genomic distance with DCJ and indels. In Vincent Moulton and Mona Singh, editors, *Algorithms in Bioinformatics*, pages 90–101, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

6 Marília D.V. Braga, Eyla Willing, and Jens Stoye. Double cut and join with insertions and deletions. *Journal of Computational Biology*, 18(9):1167–1184, 2011. PMID: 21899423. doi:10.1089/cmb.2011.0118.

7 Phillip E. C. Compeau. A simplified view of DCJ-indel distance. In Ben Raphael and Jijun Tang, editors, *Algorithms in Bioinformatics*, pages 365–377, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

8 Phillip Ec Compeau. DCJ-indel sorting revisited. *Algorithms for molecular biology : AMB*, 8(1):6–6, March 2013. doi:10.1186/1748-7188-8-6.

9 Daniel Doerr and Cedric Chauve. Small parsimony for natural genomes in the DCJ-indel model. *Journal of Bioinformatics and Computational Biology*, 19(06):2140009, 2021. PMID: 34806948. doi:10.1142/S0219720021400096.

10 David M. Emms and Steven Kelly. Orthofinder: solving fundamental biases in whole genome comparisons dramatically improves orthogroup inference accuracy. *Genome Biology*, 16(1):157, August 2015. doi:10.1186/s13059-015-0721-2.

11 Guillaume Fertin, Anthony Labarre, Irena Rusu, Eric Tannier, and Stéphane Vialette. *Combinatorics of Genome Rearrangements*. Computational Molecular Biology. The MIT Press, 2009.

12 Sridhar Hannenhalli and Pavel A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *J. ACM*, 46(1):1–27, January 1999. doi:10.1145/300515.300516.

13 Daniel H. Huson and David Bryant. Application of Phylogenetic Networks in Evolutionary Studies. *Molecular Biology and Evolution*, 23(2):254–267, October 2005. doi:10.1093/molbev/msj030.

14 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.

15 Diego P. Rubert and Marília D. V. Braga. Gene Orthology Inference via Large-Scale Rearrangements for Partially Assembled Genomes. In Christina Boucher and Sven Rahmann, editors, *22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*, volume 242 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:22, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.WABI.2022.24.

16 Diego P. Rubert, Daniel Doerr, and Marília D. V. Braga. The potential of family-free rearrangements towards gene orthology inference. *Journal of Bioinformatics and Computational Biology*, 19(06):2140014, 2021. PMID: 34775922. doi:10.1142/S021972002140014X.

17 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. PMID: 25517208. doi:10.1089/cmb.2014.0096.

18 Sophia Yancopoulos, Oliver Attie, and Richard Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16):3340–3346, June 2005. doi:10.1093/bioinformatics/bti535.

A Full capping-free ILP

Table 2 Settings for U_f, L_f in Algorithm 1 to enforce different matching models described in [11].

	Maximum Matching (Full)	Intermediate Matching	Exemplar Matching
L_f	$\lfloor \lvert f \rvert /2 \rfloor$	$\lfloor \lvert f \rvert /2 \rfloor$	1
U_f	$\lfloor \lvert f \rvert /2 \rfloor$	1	1

Algorithm 1 ILP for the capping-free computation of the DCJ-indel distance of natural genomes.

Objective:

$$\texttt{Minimize}\ \frac{1}{2}\sum_{e\in E_\xi} x_e - \sum_{u\in V} r_u^c + q + \sum_{k\in K} s_k$$

Constraints:

(C.01)	$x_e = 1$	$\forall\ e \in E_\gamma^A \cup E_\gamma^B$
(C.02)	$\sum_{\{x^h,y^h\},x,y \text{ of family } f} x_e \le U_f$	Family f
	$\sum_{\{x^h,y^h\},x,y \text{ of family } f} x_e \ge L_f$	Family f
(C.03)	$\sum_{\{u,v\}\in E_\xi} x_{\{u,v\}} + d_{g(u)} = 1$	$\forall\ u \in V$
(C.04)	$x_e = x_d$	$\forall\ e,d \in E_\xi$ such that e and d are siblings
(C.05)	$y_v \le y_u + \texttt{ix}(v)(1 - x_{\{v,u\}})$	$\forall\ \{v,u\} \in E$,
(C.06)	$\texttt{ix}(v)\cdot z_v \le y_v$	$\forall\ v \in V$
(C.07)	$n_u^A = 1$	$\forall u$ telomere of A
	$n_{u'}^B = 1$	$\forall u'$ telomere of B
(C.08)	$m_u^i \ge d_{g(u)}$	$\forall u \in V$ with genome $i \in \{a,b\}$
(C.09)	$\sum_{j\in Y} m_u^j \le 1$	$\forall u \in V$
	$\sum_{j\in Y} n_u^j \le 1$	$\forall u \in V$
(C.10)	$m_u^j \le m_v^j + (1 - x_e)$	$\forall e = \{u,v\} \in E_\xi \forall j \in Y$
	$n_u^j \le n_v^j + (1 - x_e)$	$\forall e = \{u,v\} \in E_\gamma \forall j \in Y$
(C.11)	$m_u^j \le n_u^j + z_u$	
	$n_u^j \le m_u^j + z_u$	$\forall u \in V \forall j \in Y$
(C.12)	$r_u^c + \sum_{ij\in W} r_u^{ij} = z_u$	$\forall u \in V$
(C.13)	$r_u^{ij} - m_u^k - n_u^k \le 0$	$\forall u \in V, \forall ij \in W \forall k \in \{i,j\}$
	$8r_u^c \le 8 - (\sum_i m_u^{i\in Y} + \sum_{j\in Y} n_u^j)$	$\forall u \in V$
(C.14)	$r_u^{ij} \ge m_u^i + n_u^j - 1$	$\forall u \in V \forall ij \in W$
	$r_u^{ij} \ge m_u^j + n_u^i - 1$	$\forall u \in V \forall ij \in W$
(C.15)	$r_{ABa} \ge \sum_{u\in V} r_u^{Aa}$	
	$r_{ABa} \ge \sum_{u\in V} r_u^{Ba}$	
(C.16)	$r_{ABb} \ge \sum_{u\in V} r_u^{Ab}$	
	$r_{ABb} \ge \sum_{u\in V} r_u^{Bb}$	
(C.17)	$2q \ge \sum_{v\in V} r_v^{ab} + r_{ABa} + r_{ABb} - \sum_{v\in V} r_v^{AB}$	
(C.18)	$\sum_{g\in k} d_g - \lvert k\rvert + 1 \le s_k$	$\forall k \in K$
(C.19)	$m_u^i = 0$	$\forall u \in V$ if u is in a component without type i.

Domains:

(D.01)	$d_g \in \{0,1\}$	gene g	(D.06)	$m_u^j \in \{0,1\}$	$\forall u \in V \forall j \in Y$	
(D.02)	$x_e \in \{0,1\}$	$\forall\ e \in E$	(D.07)	$r_u^{ij} \in \{0,1\}$	$\forall u \in V, \forall ij \in W \cup \{c\}$	with $Y = \{A,a,B,b\}$
(D.03)	$0 \le y_v \le \texttt{ix}(v)$	$\forall\ v \in V$	(D.08)	$r_{ABa}, r_{ABb}, q \in \mathbb{N}_0$		and $W = \{AB, Aa, Ab, Ba, Bb, ab\}$.
(D.04)	$z_v \in \{0,1\}$	$\forall\ v \in V$	(D.09)	$s_k \in \{0,1\}$	$\forall\ k \in K$	with K the circular chromosomes of $\mathbb{A}$ and $\mathbb{B}$.
(D.05)	$n_u^j \in \{0,1\}$	$\forall u \in V \forall j \in Y$	(D.10)	$y_v \in \{0,...,\texttt{ix}(v)\}$	$\forall v \in V$	with $\texttt{ix}(v)$ the index of vertex $v \in V$.

Reinforcement Learning for Robotic Liquid Handler Planning

Mohsen Ferdosi
School of Computer Science, Computational Biology Department,
Carnegie Mellon University, Pittsburgh, PA, USA

Yuejun Ge
School of Computer Science, Computational Biology Department,
Carnegie Mellon University, Pittsburgh, PA, USA

Carl Kingsford
School of Computer Science, Computational Biology Department,
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Robotic liquid handlers play a crucial role in automating laboratory tasks such as sample preparation, high-throughput screening, and assay development. Manually designing protocols takes significant effort, and can result in inefficient protocols and involve human error. We investigates the application of reinforcement learning to automate the protocol design process resulting in reduced human labor and further automation in liquid handling. We develop a reinforcement learning agent that can automatically output the step-by-step protocol based on the initial state of the deck, reagent types and volumes, and the desired state of the reagents after the protocol is finished. We show that finding the optimal protocol for solving a liquid handler instance is NP-complete, and we present a reinforcement learning algorithm that can solve the planning problem practically for cases with a deck of up to 20×20 wells and four different types of reagents. We design and implement an actor-critic approach, and we train our agent using the Impala algorithm. Our findings demonstrate that reinforcement learning can be used to automatically program liquid handler robotic arms, enabling more precise and efficient planning for the liquid handler and laboratory automation.

2012 ACM Subject Classification Computing methodologies → Sequential decision making

Keywords and phrases Liquid Handler, Reinforcement Learning, Planning

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.23

Supplementary Material *Software (Source Code)*: `https://github.com/Kingsford-Group/rlforlqh`
archived at `swh:1:dir:d8aa207d3ff7e53d35ada7c8ffcaab4918d851ef`

Funding This work was supported in part by the US National Science Foundation [DBI-1937540, III-2232121], the US National Institutes of Health [R01HG012470], the Center for Machine Learning and Health at Carnegie Mellon University through a fellowship to M.F. and by the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program.

Acknowledgements We thank Guillaume Marçais for helpful comments on the manuscript and Haotian Teng and Sam Powers for valuable discussions.

Conflicts of Interest C.K. is a co-founder of Ocean Genomics, Inc.

1 Introduction

A robotic liquid handler is a laboratory instrument that is used to dispense precise amounts of liquids into a variety of containers, such as micro-plates, test tubes, and vials. It automates liquid handling tasks, enabling scientists to process large numbers of samples quickly and accurately. Robotic liquid handlers consist of several components, including a robotic arm that moves the liquid handling tool (e.g. pipette) to the appropriate location, a liquid

dispensing system (e.g. syringe or positive displacement pump) that accurately dispenses the liquid, and software that controls the entire process. These instruments are commonly used in a range of applications, including drug discovery, genomics, proteomics, and clinical diagnostics, where accuracy, precision, and throughput are essential for obtaining reliable and reproducible results [7]. Even though there is a wide range of laboratory equipment and chemicals, diverse preferences in laboratory configurations, as well as variations in the selection of target organisms and research inquiries, there is a fundamental problem common to all protocols in a way that can be generalized to the planning task for liquid handling [2].

In laboratory practices that involve large-scale liquid handling, establishing reliable and consistent protocols is crucial for generating credible data. Robotic liquid handling techniques have played a significant role in accomplishing this objective in medical laboratories. The success of high-throughput PCR screening for SARS-CoV-2, capable of processing millions of samples weekly, is a testament to this [14]. Robotic liquid handling is used for many applications such as protein folding analyses [1], high-throughput processing for selected reaction monitoring assays [17], and in many clinical diagnostics [9], including microbiome screening [10].

There has been to date no analysis of the computational complexity of the planning problem for the liquid handlers. We therefore propose a formal view of the problem with theoretical complexity analysis that shows the problem is NP-complete, as one would expect. This motivates and justifies the development of heuristics for the problem. In order to obtain reliable and reproducible results from experiments while minimizing the cost of human labor in process, there is a clear need for a robust and scalable planning algorithm for liquid handling robots. Roboliq [15] proposed a software pipeline to transform high-level protocols designed by researchers to low-level robot instructions. SynBiopython [16] proposed an open-source Python package aiming to standardize some of the tools used in automated laboratories. ESCALATE [11] proposed a framework that enables writing machine-readable protocols with hybrid human-robot operations that enable the recording of data and optimization of experiments. Aquarium [13] offers visual programming in a way that protocols are represented graphically as blocks that can be assembled to build executable protocols.

All these methods still rely on researchers to design and conduct protocols. Approaches such as ESCALATE and Roboliq require human-intervention for protocol optimization. Automatically designing and finding optimized protocols for liquid handlers will play a significant role in lab automation. We provide the first practical way of automatically finding low-cost liquid handler protocols, significantly reducing the need for human intervention.

Reinforcement learning is a machine learning paradigm that involves learning a policy to make decisions through interaction with an environment. Reinforcement learning has been used for solving combinatorial optimization problems in order to replace using hand-crafted heuristics [8]. By integrating reinforcement learning algorithms into the planning of liquid handler robots, we aim to automate the protocol design of the liquid handler to address common challenges and limitations associated with traditional robotic systems in liquid handling, such as inefficient protocols, human error in protocol design or the high computational cost of planning of the robot. Our results show that it is possible to fully automate the protocol design for the liquid handler in many practical cases. We train reinforcement learning models based on the actor-critic paradigm and using the Impala (Importance Weighted Actor-Learner Architecture) framework [4]. Our trained model solves the vast majority of inputs for cases with a deck of up to 20×20 wells and four different types of reagents. Traditional planning algorithms are computationally intractable for many of these instances due to the super-exponential nature of the problem in terms of the number of possible actions in each step. We also show that the trained model is robust to different problem settings.

2 Formal Problem Statement

A liquid handler consists of a set of small containers (wells) and a robotic arm, called the head, that can transfer liquids between the wells using a set of specified actions. The liquid handler starts with an *initial state* which gives the configuration of the containers at the beginning of the experiment. The aim is to convert the initial state to the goal state using the set of specified actions. Here, we add some idealizing assumptions to make the formulation simpler without losing the generality of the problem:

- The state is defined by a 2D field of wells containing mixtures of reagents. Our idealized handler is designed with a single large "deck", whereas actual handlers may have multiple decks. Any complexities arising from multiple plates in real-world situations can be incorporated into the customized cost function that calculates the protocol cost.
- The head is a 2D array of tips. It can be positioned anywhere within the 2D well field, and its location is represented as an Δ offset from the top-left corner of the field. For the most part of this paper, we assume we are working with head size of 1×1.

▶ **Definition 1** (Reagents). *There is a finite set of* reagents *that can be mixed in each well. We denote them by* $R = \{r_1, r_2, ..., r_K\}$. *The mixture in each well can be represented by a* $1 \times K$ *vector that shows how much of each reagent is in that well.*

▶ **Definition 2** (Deck). *The deck is* 2D *field of wells containing mixtures of reagents. The current state of the reagents in each of the wells, along with the position and contents of the head, gives the current* state *of the liquid handler.*

▶ **Definition 3** (Head). *The head is a smaller* 2D *field of wells positioned anywhere in the deck. The position of the head allows for the two actions of aspiration or dispensation on the wells that are covered by the head. The head can change location within the deck and this movement has a cost.*

▶ **Definition 4** (Aspiration). *Aspiration is the act of transferring some amount of the mixture from the wells covered by the the head to the tips on the head. This action comes with two costs. The cost of aspiration is a constant* c_a *plus a movement cost that is the cost to move the head to the location for the aspiration from its previous location.*

▶ **Definition 5** (Dispensation). *Dispensation is the act of transferring some amount of the mixture from the tips in the head to the wells covered by the the head. This action comes with two costs. The cost of dispensation is a constant* c_d *plus a movement cost that is the cost to move the head to the location for the dispensation from its previous location.*

▶ **Definition 6** (Protocol). *A protocol is the sequence of actions of aspiration, dispensation, and moving the head's location in order to achieve a goal state starting from an initial state. The cost of a protocol is the sum of the cost for each action in the protocol.*

▶ **Problem 7** (Optimal Liquid Handling). *Given an initial state* $\mathcal{I}$ *of the deck (with an empty head), and the goal state* $\mathcal{G}$ *and the set of parameters* c_a *and* c_d *that determines the cost of each action as above, and a choice of polynomial-time computable distance metric that determines the cost to move the head between locations, give a protocol with the minimum cost that will convert* $\mathcal{I}$ *into the goal state* $\mathcal{G}$.

We show that this is an NP-complete problem even in the most restricted case of the unit-sized head and only one type of reagent. We then develop a practical approach to find low-cost protocols using reinforcement learning.

2.1 The Optimal Liquid Handling problem is NP-complete

We show that the Optimal Liquid Handling problem is NP-complete.

▶ **Theorem 8.** *The Optimal Liquid Handling decision problem that asks whether there is a protocol of cost less than or equal to C is NP-complete.*

Proof. Verification of a protocol's cost can be computed easily in polynomial time. We can follow the sequence of actions and calculate the total cost to check if it is less than or equal to C. Hence, the problem is in NP.

We reduce the Traveling Salesman Problem (TSP) to the special case of Optimal Liquid Handling with the head size of 1×1. For this, we need to use the metric TSP where all the cities are points in $\mathbb{R}^2$. The NP-completeness proof in [5] for the Euclidean TSP immediately implies the NP-completeness of a TSP where all the cities are points in $\mathbb{R}^2$ and the distances are defined in a way that the triangle inequality holds [3]. Figure 1 shows the sequence of reductions to show that the Optimal Liquid Handling is NP-Complete.

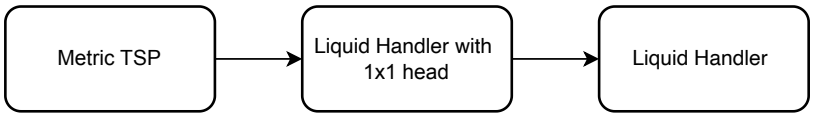


Figure 1 The order of the reductions from the Euclidean TSP to the Optimal Liquid Handling.

TSP takes a list of n cities $\mathcal{C} = \{c_1, c_2, ..., c_n\}$ in $\mathbb{R}^2$ and a budget C as input, and asks whether there is a tour denoted G_{tsp} that visits every city exactly once with total cost at most C. The reduction from TSP to Optimal Liquid Handling is as follows:

Construct an Optimal Liquid Handling instance with a budget C, and the initial state of n units of reagent on the location of c_1 on the deck and the goal configuration of one unit of reagent in each of the city locations on the deck. The cost of aspiration c_a and dispensing c_d are defined to be zero, and the cost of the head movement between the two wells is defined with the same metric as the distance in the TSP. This way the head needs to visit every well that represents the city location on the deck at least once to have the reagent in that location.

▶ **Lemma 9.** *For any sequence of moves of the head of the robotic arm that results in the tour G_0 with cost at most C, there is a tour of G_{tsp} with cost $C^* \leq C$ in TSP where it visits the same cities in the same order.*

Proof. Not every move in the robotic arm's tour G_0 must end up in the cities. The arm can move freely in the grid before visiting the next city. Let's say the moves are segmented by the cities and $C = \{c_1, c_2, ..., c_n\}$ are the cities that have been visited in G_0 in the order of their visit. We can replace any sequence of moves that happened between cities c_i and c_{i+1} with just a direct move from c_i to c_{i+1} and the cost would not be increased due to triangle inequality (which holds for any norm). Hence the new G_{tsp} would have a cost of $C^* \leq C$. ◀

Suppose there exists a protocol that visits the wells in an order that results in a tour G_0 with cost at most C in Optimal Liquid Handling, then there exists a tour of G_{tsp} with a cost of C or less in TSP using Lemma 9. If there is a tour of G_{tsp} with at most cost of C in TSP the robotic arm can follow the same path therefore there is a tour of G_0 with cost at most C in the Optimal Liquid Handling problem. This means the Optimal Liquid Handling can be used to solve the metric TSP problem which proves that Optimal Liquid Handling is NP-complete. ◀

The NP completeness of the Optimal Liquid Handling problem motivates us to use machine learning for a practical solution instead of looking for a polynomial-time algorithm.

3 Methods

3.1 Using Reinforcement Learning to Solve Optimal Liquid Handling

Reinforcement learning (RL) has emerged as a powerful technique for solving complex tasks in various domains. Actor-critic [6] is a type of reinforcement learning algorithm that uses two estimators: an actor and a critic. The actor is responsible for learning a policy, which is a function that maps states to actions. The critic is responsible for learning a value function, which estimates the expected reward from a given state and action. The interaction between the actor and the critic enables the agent to continually refine its policy based on the feedback provided by the critic, leading to improved decision-making over time.

Impala is a distributed reinforcement learning framework that integrates the actor-critic framework with a distributed architecture [4]. Impala is designed to improve the scalability and efficiency of traditional actor-critic algorithms by training multiple agents. It separates the learning process into multiple actors and a central learner [4]. Actors interact with their own instances of the environment, collecting trajectories (sequences of state, action, and reward). Multiple agents simultaneously interact with the environment and collect experience, which is used to update a shared value function and policy. Actors asynchronously send their collected trajectories to the central learner, which processes the data and updates the policy and value function. The central learner periodically sends the updated policy and value function back to the actors, ensuring that they stay relatively up-to-date with the latest learning progress. This enables efficient use of computational resources and allows the algorithm to scale to a large number of actors. This approach allows for faster and more efficient learning compared to traditional reinforcement learning algorithms that rely on a single agent [4].

We tackle the Optimal Liquid Handler problem with the head size of 1×1 using reinforcement learning. We use Impala [4] for the training and CORA [12] for the benchmarking and evaluation. CORA (short for Continual Reinforcement Learning Agents) is a package that provides benchmarking, baselines, and metrics for continual reinforcement learning tasks.

Environment. The environment is the setting in which the RL agent operates. It can be modeled as a state space, where each state corresponds to a particular configuration of the environment, and the agent can take actions to transition from one state to another. The deck, the head, and the configuration of reagents in the deck and the head form the environment in the liquid handler application. Let $N \times M$ be the size of the deck, and K be the number of different types of possible reagents.

Agent. The agent is the entity that interacts with the environment. It learns a policy which maps states to actions. The agent receives information about the environment and takes actions according to its policy and based on the input. Here, the agent outputs steps in a protocol for the liquid handling problem.

State space. The set of all possible states of the environment is described by $\{\mathcal{D}, \mathcal{G}, \mathcal{H}\}$. We define the state space with a pair of $N \times M \times K$ tensors $\mathcal{D}, \mathcal{G}$ where both $\mathcal{D}, \mathcal{G} \in \mathbb{N}^{N \times M \times K}$ and vector $\mathcal{H} \in \mathbb{N}^{K}$. $\mathcal{D}$ describes the current state of the deck where $\mathcal{D}[i][j]$ is a vector of size K that specifies how many units of each reagent are currently present in the well located

at (i, j). Values in $\mathcal{D}$ are initialized by the initial state $\mathcal{I}$ (as the input to the agent) and they can change according to each action defined below. Similarly, $\mathcal{G}$ describes the goal state which shows how many units of each reagent is supposed to be in every well after the arm performs all the actions in the protocol. Finally, $\mathcal{H}$ describes the amount of each reagents currently in the head. The protocol is considered successfully completed if $\mathcal{D} = \mathcal{G}$ at the end. We define the state space by tensors with integer values. This choice limits us in terms of the mixtures we can produce but it is necessary to keep the problem computationally tractable.

Action space. Actions that the agent can take in a given state. We design our agent with an internal state machine that defines the possible actions. There are two states in our setting. The agent is either in aspiration mode or dispensation mode depending on whether the head is empty. These two states alternate as described in Figure 2. For simplicity, a movement of the head is combined with the aspiration or dispensation actions as multiple actions in the same location can be combined into one single action.

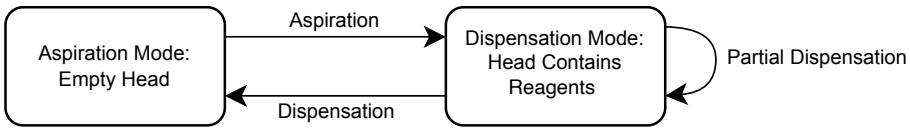


Figure 2 Two sets of actions and their corresponding internal states. Depending on whether the agent's head is empty or not, there are two sets of possible actions: aspiration and dispensation. To avoid potential cross-contamination, aspiration is only allowed when the head is empty because there might be residual liquid in the tip caused by previous aspiration that can mix with the reagent in well.

There are two actions that the agent can take based on the internal states it is currently in. When the agent is in aspiration mode, the head is empty and the agent can perform an "aspiration" action.

Aspiration is defined with the tuple (i, j, P) where P is a vector of size K, $P \in \mathbb{N}^K$, and

$$\exists b \quad 0 < b \leq 1 \quad \text{where} \quad \frac{P[k]}{\mathcal{D}[i][j][k]} = b \quad \text{for every } k \in [1, K]. \tag{1}$$

This action transfers the amount of reagents defined by P from $\mathcal{D}[i][j]$ to the head $\mathcal{H}$. The condition (1) guarantees that the ratios are kept the same for the amount that is being aspirated and the mix in the well. Aspiration is formally defined as:

$$\begin{aligned} \mathcal{D}[i][j] &= \mathcal{D}[i][j] - P \\ \mathcal{H} &= P. \end{aligned} \tag{2}$$

The cost of aspiration is c_a. This cost is independent of P, the volume aspirated, and the location. While this is not completely physically accurate, it is a reasonable model when we want to minimize the number of actions. After each aspiration the internal state of the agent changes to dispensation mode since the head is not empty anymore.

Similarly, when the agent is in dispensation mode, the agent can perform a "dispense" action since there is some liquid in the head. Dispense is defined with the tuple (i, j, P) where P is a vector, $P \in \mathbb{N}^K$, and

$$\exists b \quad 0 < b \leq 1 \quad \text{where} \quad \frac{P[k]}{\mathcal{H}[k]} = b \quad \text{for every } k \in [1, K]. \tag{3}$$

This action transfers the amount of the reagents defined by P from the head $\mathcal{H}$ to $\mathcal{D}[i][j]$. The condition (3) guarantees that the ratios are kept the same for the amount that is being dispensed into the well and the mix in the head. Dispense is formally defined as:

$$\begin{aligned}\mathcal{D}[i][j] &= \mathcal{D}[i][j] + P \\ \mathcal{H} &= \mathcal{H} - P.\end{aligned} \tag{4}$$

We call this partial dispense in the case where $\mathcal{D}[i][j] \neq P$. Partial dispense causes the internal state to stay on dispensation mode since the head is not empty yet. The cost of a dispense is c_d. This is again independent of the volume dispersed.

Each action also comes with a distance cost that models the movement of the head from the well of the previous action to the well of the next action. The cost of the n-th action $(i, j, P)_n$ depends on the previous $(i, j, P)_{n-1}$ action and is defined as

$$\begin{aligned}c_n &= \text{Manhattan Distance}((i, j)_n, (i, j)_{n-1}) \\ &= |i_n - i_{n-1}| + |j_n - j_{n-1}|,\end{aligned} \tag{5}$$

and cost of a protocol is defined as the sum of the cost for all actions and movements. We use Manhattan distance here as our choice of distance, although other metrics are also reasonable.

Reward. The reward is a scalar feedback signal that the agent receives from the environment after taking an action. The goal of the agent is to learn a policy that maximizes the cumulative reward over time. The agent's aim is to reach the goal state while minimizing the distance cost of the protocol. In this task, the agent receives $+1$ reward for dispersing one unit of reagent in a well that is missing at least one unit of that reagent to reach its goal defined by the goal state $\mathcal{G}$. In order to facilitate the exploration and motivate the agent to perform moves, we also add $+1$ reward for aspirating one unit of a reagent that is currently in a well that does not have that reagent in its goal (or the amount currently in the well exceeds the amount described in the goal), meaning that this unit has to be moved in order to achieve the goal state. This way moving one unit of reagent from a wrong well to a correct well results in total of $+2$ reward. There is also -1 reward for aspirating one unit of reagent that is currently in the correct well and another -1 reward for dispersing one unit of reagent in the wrong well. This means moving one unit from a correct well to another correct well or from a wrong well to another wrong well results in 0 reward, and moving a unit from a correct well into a wrong well results in -2 reward. We also weight the distance cost by a parameter α that is the relative weighting of the distance cost to the reward. More formally, we calculate the reward for each action as described below.

In aspiration mode, the reward for the move (i, j, P) is calculated by

$$\begin{aligned}C &= \min(\mathcal{G}[i][j], \mathcal{D}[i][j]) \\ C' &= \min(\mathcal{G}[i][j], \mathcal{D}[i][j] - P) \\ W &= C - C' \\ R &= P - W \\ r &= |R| - |W| - \alpha c_n\end{aligned} \tag{6}$$

where the function "min" between two vectors is defined as a vector of the element-wise minimum. $C \in \mathbb{N}^K$ shows the amount of reagents that are currently in position (i, j) and are part of the goal state (completed units) before the move, and $C' \in \mathbb{N}^K$ shows the amount of reagents that will be in position (i, j) and are part of the goal state after the move. $W \in \mathbb{N}^K$

is the vector of reagents that are being aspirated from a goal well (wrong moves), and $R \in \mathbb{N}^K$ is the vector of reagents that are being aspirated from a well that does not need that type in its goal (right moves). Hence, the reward r is calculated by $+1 \times |R| - 1 \times |W|$ and subtracted by a factor of the distance cost of that move $-\alpha c_n$. We can always choose a small enough α based on the size of the deck so that finding a protocol that completes the task by reaching the goal state has a higher priority than optimizing the distance cost.

Similarly in the dispensation mode, the reward for the move (i, j, P) is calculated by

$$\begin{aligned} L &= \max(\mathcal{G}[i][j] - \mathcal{D}[i][j], 0) \\ R &= \min(P, L) \\ W &= P - R \\ r &= |R| - |W| - \alpha c_n \end{aligned} \tag{7}$$

where the function "max" between a vector and a number is defined as the element-wise maximum of the values in the vector and the number. $L \in \mathbb{N}^K$ shows the amount of reagents that are needed in position (i, j) to complete the goal for well in (i, j) (units left) before the move. Similarly to the aspiration reward, R is the vector of reagents that are being dispersed to a goal well (right moves), and W is the vector reagents that are being dispersed to a non-goal well (wrong moves). The reward r is again calculated by $+1 \times |R| - 1 \times |W| - \alpha c_n$.

Policy. The policy is the mapping between states and actions that the agent uses to make decisions. The goal of the agent is to learn an optimal policy that maximizes the cumulative reward over time.

3.2 The Network Architecture

We use Impala for our training framework. Impala is a distributed reinforcement learning algorithm designed to tackle large-scale RL problems efficiently. It achieves this by employing a distributed architecture, which enables parallel data collection and learning. The actors are responsible for exploring the environment and generating trajectories of state, action, reward, and next-state tuples by following their respective behavior policies. The central learner processes the collected data to update both the policy (actor) and the value function (critic). Each actor in the Impala architecture has an actor network, which is responsible for selecting actions based on the current state of the environment. We design the actor network as a three-layered convolutional network described below.

The designed network needs to take an encoding of the current state $\{\mathcal{D}, \mathcal{G}, \mathcal{H}\}$ as the input and must output a probability distribution over all the possible actions to take at this step. We define observation $\mathcal{O} = \{\mathcal{D}, \mathcal{G}, \mathcal{H}^*\}$ as the encoding, where $\mathcal{D}, \mathcal{G}$ are previously defined as the current state and the goal state. $\mathcal{H}^*$ is a tiling of the vector $\mathcal{H}$ where $\mathcal{H}$ is repeated $N \times M$ times (to count for each well). This way, all three of $\{\mathcal{D}, \mathcal{G}, \mathcal{H}^*\}$ are $N \times M \times K$ tensors. We do this tiling because we want the associations between the current reagents in the head and the goal definition for each well to be highlighted in the convolutional layer (e.g. if the amount in the head matches or mismatches with the goal for each well). We use convolutional layers with kernel size of 1×1 to perform a channel-wise feature mixing. The 1×1 convolution is used to combine features across channels with information on $\mathcal{H}^*$ and the channels with information on $\mathcal{G}$. This is aimed to help the network learn which wells still need some reagents to reach the goal state.

The output of each state is the policy distribution for every possible action. We define the capacity of the tip on the 1×1 head as D. A possible action a is defined as $a = (x, y, p)$ which states "p amount of the the mix and the position (x, y) on the grid". The type of the

action (i.e. aspiration or dispensation) is determined by the internal state machine. If the head is not empty, the action $a = (x, y, p)$ is a dispense, and if it is empty, the action is an aspiration. Since we are using $\mathcal{O} = \{\mathcal{D}, \mathcal{G}, \mathcal{H}^*\}$ as the encoding, we have $3 \times K \times N \times M$ input values, which is equivalent of $(3 \times K)$ channels for images of size $N \times M$, where each channel describes the amount of one reagent in either the current state, the goal state, or the tiled head. Similarly, we have $D \times N \times M$ output values that describe the policy distribution; these are equivalent to D channels for images of size $N \times M$. We use the three layered convolutional network described in Figure 3 for the actor network, and we use a flattening layer at the end to get a one-dimensional vector that describes the the policy distribution.

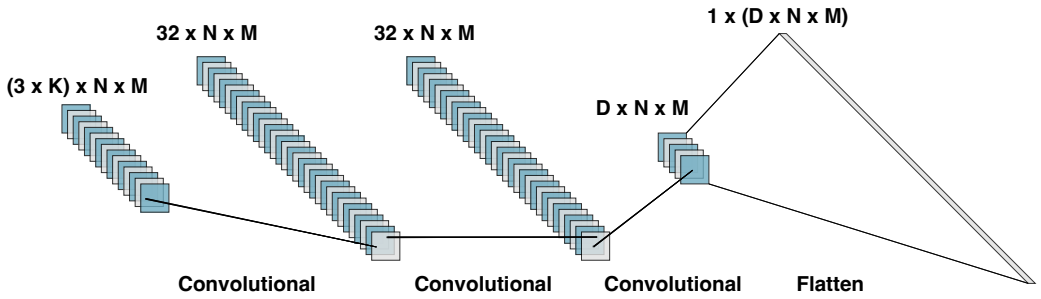


Figure 3 The three-layered convolutional network used for the actor network. The actor learns the policy, which is described by the probability distribution over the possible actions, based on the current state $\{\mathcal{D}, \mathcal{G}, \mathcal{H}\}$ as its input.

The central learner in Impala also has a critic network that estimates the value function, which represents the expected cumulative reward from a given state, following the current policy. For our task, we design the critic network as a fully connected layer that gets the observation $\{\mathcal{D}, \mathcal{G}, \mathcal{H}\}$ as the input and produces a number as its output that describes the expected reward for the given state.

4 Results

We first show some examples to showcase some of the challenges that the agent needs to overcome to find an optimal protocol. We then present experiments benchmarking the proposed reinforcement learning approach compared with several baseline approaches.

4.1 Challenges

The first priority for the RL agent is to reach the goal state. The agent also needs to minimize the distance cost of the protocol. Figure 4 shows two possible protocols for a given state. The agent needs to choose the one with lower distance cost.

One thing that the agent needs to learn is the volume of each action. Always using the whole volume available for aspiration and dispensation would not reach the goal in some cases. Figure 5 shows why it is important for the agent learn to use moves with $b < 1$.

Another challenge for the agent is to learn to mix or not mix reagents when necessary. This is a unique aspect of the planning for liquid handler robotic arm since putting liquid on top of each other results in mixing them and producing a new reagent. This move is irreversible by the liquid handler. Figure 6 shows how mixing when not necessary can result in a state where reaching the goal is not accessible anymore. The agent needs to learn these cases in order to be able to find the optimal protocol.

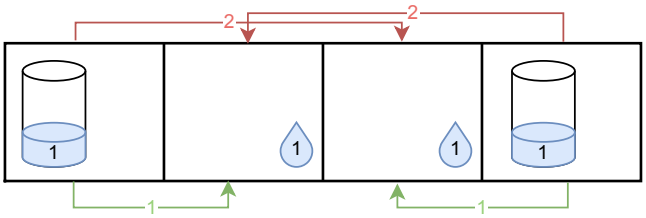


Figure 4 Here we have 1×4 deck. The two side wells each have one unit of the blue reagent in the initial state $\mathcal{D}$ as shown by the two cylinders. The two middle wells each have one unit of the blue reagent in the goal state $\mathcal{G}$ as shown by the two drops on the bottom right of each well. If we do the moves based on the red arrows on the top the distance cost will be equal to 4, but if we do the moves based on the green arrows on the bottom, the distance cost will be equal to 2. We expect the agent to choose the green arrows for the moves.

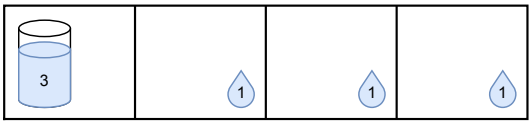


Figure 5 In this example the left wells has three units of the blue reagent in the current state $\mathcal{D}$ and need to distribute it into three well with one unit in each. It would be impossible for the agent to reach the goal state $\mathcal{G}$ without using moves with $b < 1$.

The agent cannot simply collect rewards greedily by moving each reagent to a goal well. Figure 7 shows how a greedy algorithm would fail by falling into a state that makes the goal unreachable because of the irreversibility of mixing.

4.2 Experiments

We show experimental results from the reinforcement learning agent to show the potential of this approach on solving the automated protocol design for robotic liquid handlers. In the first experiment, we show that by training on various settings we are able to receive almost perfect results in terms of receiving the highest possible reward for each instance. In the second experiment, we show that the trained agent is robust to other settings, and in the third experiment we show the comparison of reinforcement learning method and traditional planning and a greedy algorithm in terms of finding the optimal protocols. We used the following hyperparameters for the reinforcement learning: number of actors = 16, batch size = 16, discount factor = 0.9, and learning rate = 10^{-4}. All the experiment are performed on a machine with Apple M2 Pro chip with a 10-core CPU and a 16-core GPU and 32GB of unified memory.

4.2.1 Model Performance in Terms of Completing the Task

In this experiment, we train the model on randomly generated initial and goal states. Every unit of reagent is placed on the locations of the grid uniformly and independently. We make sure that the input is synthesized in a way that the maximum possible reward is fully reachable, meaning if there are S units of reagents in the goal, there is a protocol that reaches $2S$ reward. To create these instances, we separate the wells that initially have the reagents from the wells that have the reagents in the goal state. This guarantees that no reagent is already where it should be in the goal state making the reward for it unreachable. The other condition is that the reagents are placed in a way that a series of aspiration and dispense steps will reach the goal state from the initial state. To guarantee this, we start by creating

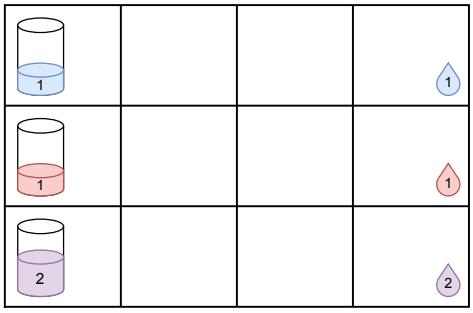


Figure 6 In this example the left column shows the reagents in the initial state. We have one unit of the blue reagent in the top, one unit of the red reagent in middle, and a mix of one unit of blue and one unit of red reagents at the bottom (purple reagent). We have the same formation for the goal state in the column at the right. The agent should never mix the blue reagent and the red reagent to make the purple reagent for the bottom right well since that would make it impossible to reach the goal state. Instead, the agent should move every reagent to its corresponding well on the last column.

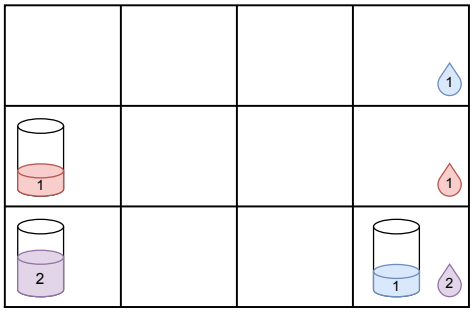


Figure 7 In this example, we have a similar initial state and the same exact goal state as the example in Figure 6. The only difference is that the blue reagent has moved to the bottom right well. A greedy agent could move the red reagent to mix with the blue reagent to receive $+2$ rewards but as we have seen that would make the goal unreachable. Instead the agent should first aspirate the blue reagent from the bottom right corner resulting in -1 reward since one unit of blue reagent is part of the goal state in that well. Only then should it move all three reagents to their corresponding wells.

a random goal state and then randomly move portions of each mixture in the goal state to new wells to create the initial state. This guarantees that there is at least one path to reach the goal from the initial state by doing the reverse of each action that initially produced the test case. We also choose $\alpha = 0$ for this experiment so the distance cost does not affect the rewards.

We train the model on seven different settings with various deck sizes and reagent numbers. Table 1 shows the average performance of each setting for 1000 inputs confirming that the trained model can complete the task for the vast majority of the inputs. For the first four settings, the agent reaches the goal in every single instance and for the three larger settings the model receives around 95% of the rewards on average. That means the model outputs a protocol that gets very close to the goal state, but it is not able to reach the goal in some cases. There is a trade-off between the deck size and the training time in a way that the agent needs more training to work with bigger decks as the state space grows exponentially. In addition, having more reagents increases the complexity of the task, and requires longer protocols to reach the goal. That is the reason we see a drop on the average reward for the bigger instances. Figure 8 shows the average cumulative reward of each run for all the settings over the steps of the training.

Table 1 This table shows the results of training the model on six different settings. The reward is the average reward over 1000 sample and the training time is the amount of time before the model converges. Steps shows the number of training iterations the model was trained on. The training stops if the model converges. For a trained model, it takes less than a second to output the protocol for every setting.

$M \times N$	K	Reagents	Units	Max Reward	Reward	Steps	Training time
2×2	2	$1, 1$	2	4	4 ± 0	1M	3m 59s
5×5	2	$2, 2$	4	8	8 ± 0	10M	21m 46s
10×10	2	$3, 3$	6	12	12 ± 0	16M	1h 31m
12×12	2	$5, 5$	10	20	20 ± 0	16M	1h 53m
15×15	2	$10, 10$	20	40	38.2 ± 0.45	18M	2h 37m
10×10	4	$5, 5, 5, 5$	20	40	39 ± 0.05	18M	1h 41m
20×20	3	$15, 15, 15$	45	90	83.16 ± 5.58	50M	15h

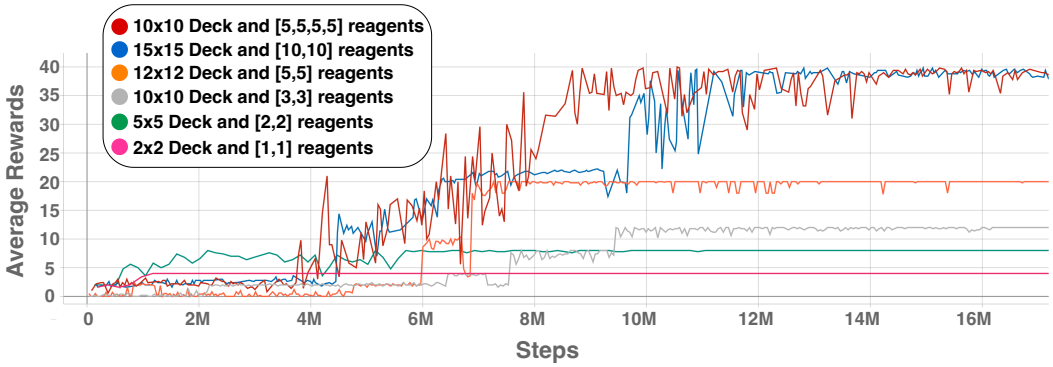


Figure 8 This figure shows the reward over time during the training for different settings. For smaller cases, the model converges to completing all the input tests and receiving full rewards. Every point on the figure is the average of 10 instances and points are generated every 10,000 steps.

4.2.2 Robustness of the Model in Unseen Settings

We show that the trained model can solve settings that are different from what the model has been trained on. To show this, we train a model on the deck size of 10×10, with two types of reagents with total units of $[4, 4]$ (meaning 4 units of reagent r_1 and 4 units of reagent r_2). For evaluation, however, we test the model on the same deck size but with four different settings with total units of $\{[5, 5], [3, 3], [3, 5], [2, 6]\}$ with respective total available rewards of 20, 12, 16, and 16 (taking $\alpha = 0$ to remove the distance costs). Since N, M (deck size) and K (number of different reagents) are the same as the training, the state and action spaces have the same dimensions as before and the network is able to adapt to these new instances. We use the same approach as the previous section to create 1000 random instances. As we can see in Figure 9, the model is able to fully generalize to all of these cases receiving the average reward equal to the max reward possible. Our model is able to do this is because the actor network is designed to be sensitive to every unit of reagent that is not placed correctly and continue working on them until it reaches the goal state making the model generalizable to unseen settings.

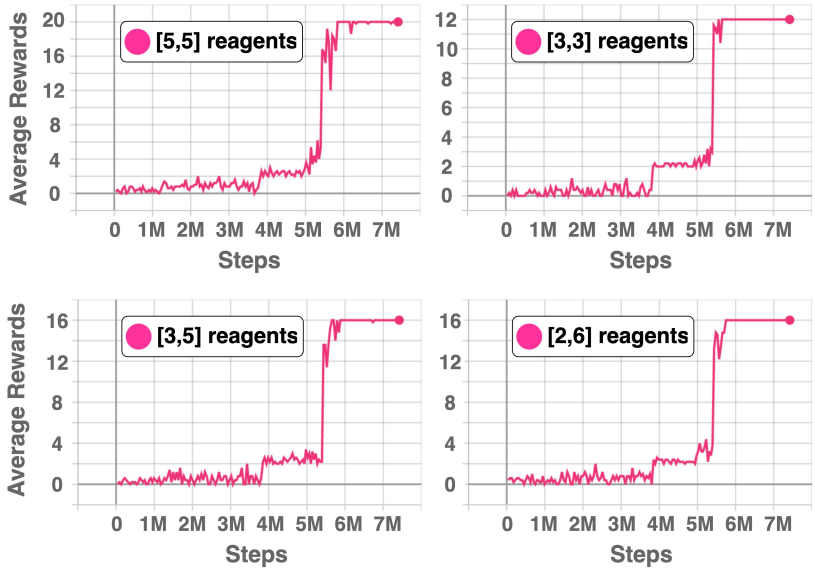


Figure 9 The reward over time for unseen settings during the training. It shows that for all the cases, the model trained on $[4, 4]$ is able to complete all the input tests and receiving full rewards after around 7 millions steps. After each step of the training on $[4, 4]$, we evaluate the model on four new settings that are $[5, 5], [3, 3], [3, 5],$ and $[2, 6]$ with respective total available rewards of 20, 12, 16, and 16 (taking $\alpha = 0$) from left to right.

4.2.3 Optimality of the Protocol in Different Models

Due to the super-exponential nature of the problem, brute force and classic planning algorithm are not tractable for solving this problem. In this section, we compare the proposed reinforcement learning model with a greedy heuristic algorithm and a heuristic-based beam search algorithm as baselines. We compare all the models in terms of success rate which is defined as how often a model outputs a protocol that reaches the goal state, the average distance cost, and the runtime. Table 2 shows the details of each setting.

Greedy Heuristic Algorithm. For this algorithm, we iteratively perform a greedy actions to reach the goal state. We use the same state machine described in Figure 2 but instead of letting the RL agent decide on the tuple $a = (x, y, p)$, we choose them in a greedy fashion. For aspiration, we limit our choices only to the wells that currently have extra reagents compared to what they should have according to the goal state. Similarly, for dispense we only consider the wells that need additional reagents to reach the goal state. We choose the closest x and y to the current position of the head among those choices, and pick p randomly. With this definition of greedy actions, we get closer to the goal state after every step. We stop when we reach the goal state or when there is no more greedy action possible to take.

Heuristic-Based Beam Search. We also use the beam search algorithm, a heuristic search method, as a baseline for this problem. The beam search algorithm uses a breadth-first search (BFS) approach to explore the solution tree. In contrast to the standard BFS that expands all nodes at every level, the beam search uses a heuristic to order the nodes and only keeps a specific set of promising nodes at each stage. The size of this set is called the beam width and is a parameter of the algorithm. At each step, beam search generates all possible successors to the current state by applying all applicable actions. These successors

are then evaluated based on a heuristic function that assigns a score to each state based on its desirability. The states with the highest scores are selected to be the new set of candidate states for the next iteration. This technique significantly cuts down the search space, making it tractable. However, it can also prune the path to the optimal solution, resulting in potentially sub-optimal protocols. We designed a heuristic functions tailored to the optimal liquid handling problem for the beam search. The heuristic function for each action is the sum of the cost and the absolute differences between the volumes of reagents in each well at the current state and the corresponding target volume in the goal state.

As we can see from the Table 2, the proposed reinforcement learning approach outperforms both baselines in terms of the success rate, resulting in a protocol that reaches the goal state in the vast majority of cases. The average distance cost for the proposed reinforcement learning method is also lower than the greedy heuristics method in all the cases. The greedy heuristics method is able to find a valid protocol more often for the cases where the amount of reagents is small compared to the grid size. For those cases, the random positioning of reagents results in a sparse grid which is a simpler task and makes it easier for the greedy heuristic to perform greedy actions one at a time. The beam search algorithm is able to find protocols with smaller distance cost but it has a much smaller success rate. In addition, the beam search has an exponential runtime which makes it not scalable for the larger instances.

Table 2 This table shows the results of comparison between the reinforcement learning (RL), greedy heuristic (GH), and beam search (BS). Each experiment is repeated 1000 times for RL and GH and 100 times for BS. Success rate is the the number each model reached the goal state divided by the total number of experiments. Distance cost is the average of distance cost for those runs that reached the goal. Training time is the amount of time it took for the RL model to train, and query time is the average time it takes each model to output the protocol. We used $\alpha = 0.2$ for the reinforcement learning model and used the beam size of 25 for the beam search on the two larger decks and beam size of 40 for the other instances.

$M \times N$	Reagents	Model	Success Rate(%)	Distance Cost	Training time	Query(s)
4×4	$4, 4$	RL	100	38.496	22m	0.484
		GH	78.7	39.40	–	0.004
		BS	95.0	23.87	–	3.80
6×6	$3, 3$	RL	100	44.471	50m	0.484
		GH	93.2	46.254		0.002
		BS	92	30.74	–	5.37
6×6	$8, 8$	RL	98.9	102.84	1h 21m	0.490
		GH	59.8	113.148	–	0.009
		BS	72	68.44	–	30.10
8×8	$4, 4$	RL	100	76.53	1h 51m	0.486
		GH	93.0	81.455	–	0.003
		BS	91	59.59	–	24.67
8×8	$15, 15$	RL	95.3	264.97	4h 31m	0.494
		GH	40.0	266.652	–	0.014
		BS	43	190.08	–	143.47
10×10	$5, 5, 5, 5$	RL	87.9	232.83	6h 36m	0.474
		GH	70.5	249.99	–	0.008
		BS	34	150.41	–	306.45

5 Conclusion

We demonstrated the potential of reinforcement learning for automating protocol design for robotic liquid handlers. We developed a reinforcement learning agent that automatically generates step-by-step protocols based on the initial state of the deck, reagent types, and volumes. This will result in reduced human labor in the process and further improve the automation of liquid handling tasks. Our proposed reinforcement learning algorithm can effectively solve the planning problem for practical cases involving decks of up to 20×20 wells and four different types of reagents.

Future research could focus on expanding the capabilities of the agent to handle more complex laboratory tasks, larger decks, other distance metrics, and a greater variety of reagent types. Additionally, integrating our approach with existing liquid handling software could streamline the protocol design process and further advance the adoption of this technology in laboratories. Overall, this work represents a promising step towards harnessing the power of reinforcement learning to to further automate the liquid handling process. The implementation of all the models is publicly released at `https://github.com/Kingsford-Group/rlforlqh`.

References

1 Philip An, Dwight Winters, and Kenneth W Walker. Automated high-throughput dense matrix protein folding screen using a liquid handling robot combined with microfluidic capillary electrophoresis. *Protein Expression and Purification*, 120:138–147, 2016.

2 Dominik Buchner, Till-Hendrik Macher, Arne J Beermann, Marie-Thérése Werner, and Florian Leese. Standardized high-throughput biomonitoring using DNA metabarcoding: Strategies for the adoption of automated liquid handlers. *Environmental Science and Ecotechnology*, 8:100122, 2021.

3 Rainer E Burkard, Vladimir G Deineko, René van Dal, Jack AA van der Veen, and Gerhard J Woeginger. Well-solvable special cases of the traveling salesman problem: a survey. *SIAM Review*, 40(3):496–546, 1998.

4 Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-RL with importance weighted actor-learner architectures. In *International Conference on Machine Learning*, pages 1407–1416. PMLR, 2018.

5 Alon Itai, Christos H Papadimitriou, and Jayme Luiz Szwarcfiter. Hamilton paths in grid graphs. *SIAM Journal on Computing*, 11(4):676–686, 1982.

6 Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in Neural Information Processing Systems*, 12, 1999.

7 Fanwei Kong, Liang Yuan, Yuan F Zheng, and Weidong Chen. Automatic liquid handling for life science: a critical review of the current state of the art. *Journal of Laboratory Automation*, 17(3):169–185, 2012.

8 Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, 134:105400, 2021.

9 Christoph B Messner, Vadim Demichev, Daniel Wendisch, Laura Michalick, Matthew White, Anja Freiwald, Kathrin Textoris-Taube, Spyros I Vernardis, Anna-Sophia Egger, Marco Kreidl, et al. Ultra-high-throughput clinical proteomics reveals classifiers of COVID-19 infection. *Cell Systems*, 11(1):11–24, 2020.

10 Jeremiah J Minich, Greg Humphrey, Rodolfo AS Benitez, Jon Sanders, Austin Swafford, Eric E Allen, and Rob Knight. High-throughput miniaturized 16s rRNA amplicon library preparation reduces costs while preserving microbiome integrity. *mSystems*, 3(6):e00166–18, 2018.

11 Ian M Pendleton, Gary Cattabriga, Zhi Li, Mansoor Ani Najeeb, Sorelle A Friedler, Alexander J Norquist, Emory M Chan, and Joshua Schrier. Experiment specification, capture and laboratory automation technology (ESCALATE): a software pipeline for automated chemical experimentation and data management. *MRS Communications*, 9(3):846–859, 2019.

12 Sam Powers, Eliot Xing, Eric Kolve, Roozbeh Mottaghi, and Abhinav Gupta. CORA: Benchmarks, baselines, and metrics as a platform for continual reinforcement learning agents. In *Conference on Lifelong Learning Agents*, pages 705–743. PMLR, 2022.

13 Justin Vrana, Orlando de Lange, Yaoyu Yang, Garrett Newman, Ayesha Saleem, Abraham Miller, Cameron Cordray, Samer Halabiya, Michelle Parks, Eriberto Lopez, et al. Aquarium: open-source laboratory software for design, execution and data management. *Synthetic Biology*, 6(1):ysab006, 2021.

14 Yishan Wang, Hanyujie Kang, Xuefeng Liu, and Zhaohui Tong. Combination of RT-qPCR testing and clinical features for diagnosis of COVID-19 facilitates management of SARS-CoV-2 outbreak. *Journal of Medical Virology*, 92(6):538, 2020.

15 Ellis Whitehead, Fabian Rudolf, Hans-Michael Kaltenbach, and Jörg Stelling. Automated planning enables complex protocols on liquid-handling robots. *ACS Synthetic Biology*, 7(3):922–932, 2018.

16 Jing Wui Yeoh, Neil Swainston, Peter Vegh, Valentin Zulkower, Pablo Carbonell, Maciej B Holowko, Gopal Peddinti, and Chueh Loo Poh. SynBiopython: an open-source software library for Synthetic Biology. *Synthetic Biology*, 6(1), 2021.

17 Min Zhu, Pingbo Zhang, Minghui Geng-Spyropoulos, Ruin Moaddel, Richard D Semba, and Luigi Ferrucci. A robotic protocol for high-throughput processing of samples for selected reaction monitoring assays. *Proteomics*, 17(6):1600339, 2017.