Algorithms and Complexity on Indexing Elastic Founder Graphs

Massimo Equi
Department of Computer Science, University of Helsinki, Finland

Tuukka Norri
Department of Computer Science, University of Helsinki, Finland

Jarno Alanko
Department of Computer Science, University of Helsinki, Finland
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Bastien Cazaux
LIRMM, Univ. Montpellier, CNRS, France

Alexandru I. Tomescu
Department of Computer Science, University of Helsinki, Finland

Veli Mäkinen
Department of Computer Science, University of Helsinki, Finland

Abstract

We study the problem of matching a string in a labeled graph. Previous research has shown that unless the *Orthogonal Vectors Hypothesis* (OVH) is false, one cannot solve this problem in strongly sub-quadratic time, nor index the graph in polynomial time to answer queries efficiently (Equi et al. ICALP 2019, SOFSEM 2021). These conditional lower-bounds cover even deterministic graphs with binary alphabet, but there naturally exist also graph classes that are easy to index: E.g. *Wheeler graphs* (Gagie et al. *Theor. Comp. Sci.* 2017) cover graphs admitting a Burrows-Wheeler transform -based indexing scheme. However, it is NP-complete to recognize if a graph is a Wheeler graph (Gibney, Thankachan, ESA 2019).

We propose an approach to alleviate the construction bottleneck of Wheeler graphs. Rather than starting from an arbitrary graph, we study graphs induced from *multiple sequence alignments*. *Elastic degenerate strings* (Bernadini et al. SPIRE 2017, ICALP 2019) can be seen as such graphs, and we introduce here their generalization: *elastic founder graphs*. We first prove that even such induced graphs are hard to index under OVH. Then we introduce two subclasses that are easy to index. Moreover, we give a near-linear time algorithm to construct indexable elastic founder graphs. This algorithm is based on an earlier segmentation algorithm for gapless multiple sequence alignments inducing non-elastic founder graphs (Mäkinen et al., WABI 2020), but uses more involved techniques to cope with repetitive string collections synchronized with gaps. Finally, we show that one of the subclasses admits a reduction to Wheeler graphs in polynomial time.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness; Theory of computation → Graph algorithms analysis; Theory of computation → Pattern matching; Theory of computation → Sorting and searching; Theory of computation → Dynamic programming; Applied computing → Genomics

Keywords and phrases orthogonal vectors hypothesis, multiple sequence alignment, segmentation

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2021.20

Related Version *Full Version*: `https://arxiv.org/abs/2102.12822`

Funding This work was partly funded by the Academy of Finland (grants 309048 and 322595), by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851093, SAFEBIO), and by the Helsinki Institute for Information Technology.

1 Introduction

In string research, many different problems relate to the common question of how to handle a collection of strings. When such a collection contains very similar strings, it can be represented as some “high scoring” *Multiple Sequence Alignment* (MSA), i.e., as a matrix MSA$[1..m, 1..n]$ whose m rows are the individual strings each of length n, which may include special “gap” symbols such that the columns represent the aligned positions. While it is NP-hard to find an optimal MSA even under the simplest score of maximizing the number of identity columns (i.e., longest common subsequence length) [21], the central role of MSA as a model of biological evolution has resulted into numerous heuristics to solve this problem in practice [9]. In this paper, we assume an MSA as an input.

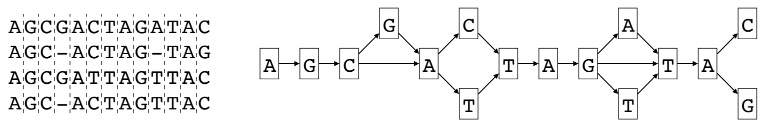
A simple way to define the problem of finding a match for a given string in the MSA is to ask whether the string matches a substring of some row (ignoring gap symbols). This leads to the widely studied problem of indexing repetitive text collections, see, e.g., references [24, 29, 30, 28, 27, 14, 15]. These approaches reducing an MSA to plain text reach algorithms with linear time complexities.

One feature worth considering is the possibility to allow a match to jump from any row to any other row of the MSA between consecutive columns. This property is usually referred to as *recombination* due to its connection to evolution, and leads to the graph representation of the MSA [26]. Figure 1a shows a simple solution, which consists in turning distinct characters of each column into nodes, and then adding the edges supported by row-wise connections. In this graph, a path whose concatenation of node labels matches a given string represents a match in the original MSA (ignoring gaps).

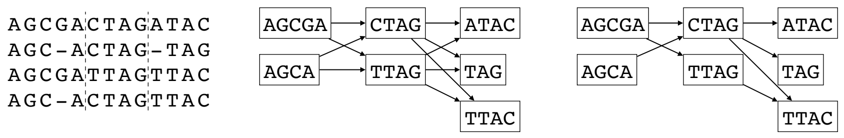
Aligning a sequence against a graph is not a trivial task. Only quadratic solutions are known [3, 25, 32], and this was recently proved to be a conditional lower bound for the problem [10]. Moreover, even attempting to index the graph to query the string faster presents significant difficulties. On one hand, indexes constructed in polynomial time still require quadratic-time queries in the worst case [35]. On the other hand, worst-case linear-time queries are possible, but this has the potential to make the index grow exponentially [34]. These might be the best results possible for general graphs and DAGs without any specific structural property, as the need for exponential indexing time to achieve sub-quadratic time queries constitutes another conditional lower bound for the problem [11].

Thus, if we want to achieve better performances, we have to make more assumptions on the structure of the input, so that the problem might become tractable. Following this line, a possible solution consists in identifying special classes of graphs that, while still able to represent any MSA, have a more limited amount of recombination, thus allowing for fast matching or fast indexing. This is the case for *Elastic Degenerate Strings* (EDS) [4, 5, 7, 6, 18], which can represent an MSA as a sequence of sets of strings, in which a match can span consecutive sets, using any one string in each of these (see Figure 1b, graph in the center). The advantage of this structure is that it is possible to perform expected-case subquadratic time queries [5]. However, EDS are still hard to index [16], and there is a lack of results on how to derive a “suitable” EDS from an MSA.

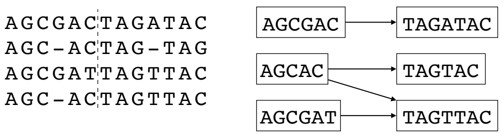
In this context, we propose a generalization of an EDS to what we call an *Elastic Founder Graph* (EFG). An EFG is a DAG that, as an EDS, represents an MSA as a sequence of sets of strings; each set is called a *block*, and each string inside a block is represented as a labeled node. The difference with EDSes is that the nodes of two consecutive blocks are not forced to be fully connected. This means that, while in an EDS a match can always pair any string of a set with any string of the next set, in an EFG it might be the case that only some of



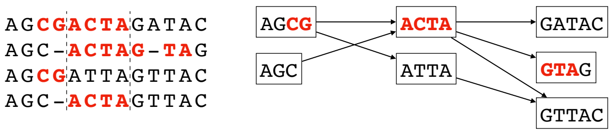
(a) A column-by-column segmentation of an MSA on the left, leading to the variation graph on the right.



(b) A different segmentation of the MSA, leading to the EDS in the center, and the EFG on the right. Notice that in an EDS every node is connected with all nodes to the right, while in an EFG edges are added only if their endpoints are consecutive in some row of the MSA (as in the case of variation graphs).



(c) A segmentation of the MSA that leads to a repeat-free EFG (i.e. no node label has another occurrence on some path of the EFG).



(d) A segmentation of the MSA that leads to a semi-repeat-free EFG (i.e. no node label has another occurrence on some path of the EFG, except as a prefix of another node in the same segment). An occurrence of query $Q =$ `CGACTAGTA` in EFG is depicted in red. As can be seen, such query does not have an occurrence in a single row of the MSA.

Figure 1 An MSA on the left, and various graph-based representations of it on the right. Notice that in all graphs (except the EDS) edges are added only between nodes that are observed as consecutive in some row of the MSA.

these pairings are allowed. Figure 1b illustrates these differences. Allowing for more selective connectivity between consecutive blocks also means that finding a match for a string in an EFG is harder than in an EDS. This is because EDSes are a special case of EFGs, hence the hardness results for the former carry to the latter. Specifically, a previous work [17] showed that, under the *Orthogonal Vectors Hypothesis* (OVH), no index for EDSes constructed in polynomial time can provide queries in time $O(|Q| + |\widetilde{T}|^\delta |Q|^\beta)$, where $|\widetilde{T}|$ is the number of sets of strings, $|Q|$ is the length of the pattern and $\beta < 1$ or $\delta < 1$. Nevertheless, in this work we present an even tighter quadratic lower bound for EFGs, proving that, under OVH, an index built in time $O(|E|^\alpha)$ cannot provide queries in time $O(|Q| + |E|^\delta |Q|^\beta)$, where $|E|$ is the number of edges and $\beta < 1$ or $\delta < 1$. Notice that $|\widetilde{T}|$ could even be $o(|E|)$ (e.g. an EFG of two fully connected blocks), hence our lower bound more closely relates to the total size of an EFG. Additionally, the earlier lower bound [17] naturally applies only to indexing EDSes, and is obtained by performing many hypothetical fast queries; ours is derived by first proving a quadratic OVH-based lower bound for the *online* string matching problem in EFGs, and then using a general result [11] to simply translate this into an indexing lower bound.

Then, in order to break through these lower bounds, we identify two natural classes of EFGs, which respect what we call *repeat-free* and *semi-repeat-free* properties. The repeat-free property (Figure 1c) forces each string in each block to occur only once in the entire graph, and the semi-repeat-free property (Figure 1d) is a weaker form of this requirement. Thanks to these properties, we can more easily locate substrings of a query string in repeat-free EFGs and semi-repeat-free EFGs. In particular, (semi-)repeat-free EFGs and EDSes can be indexed in polynomial time for linear time string matching.

One might think that these time speedups come with a significant cost in terms of flexibility. Instead, the special structure of these EFGs do not hinder their expressive power. Indeed, we show that an MSA can be "optimally" segmented into blocks inducing a repeat-free or semi-repeat-free EFG. Clearly, this depends on how one chooses to define optimality. We consider three optimality notions: maximum number of blocks, minimum maximum block width, and minimum maximum block length. In Figure 1d, the first score is 3, second is 3, and the third is 5. The two latter notions stem from the earlier work on segmentations [31, 8], now combined with the (semi)-repeat-free constraint. The first is the simplest optimality notion, now making sense combined with the (semi)-repeat-free constraint.

For each of these optimality notions, we give a polynomial-time dynamic programming algorithm that converts an MSA into an optimal (semi-)repeat-free EFG if such exists. For the first and the third notion combined with semi-repeat-free constraint, we derive more involved solutions with almost optimal $O(mn \log m)$ and $O(mn \log m + n \log \log n)$ running time, respectively. In previous work [23], an (optimal) $O(mn)$ time solution was given for the special case of MSA without gap symbols. Our new solution does not much build on the previous approach, which was based on a monotonicity property not anymore holding with gaps. Instead, we delve into the combinatorial properties of repetitive string collections synchronized with gaps and show how to use string data structures in this setting. The techniques can be easily adapted for other notions of optimality.

Another class of graphs that admits efficient indexing are Wheeler graphs [13], which offer an alternative way to model an EFG and thus a MSA. However, it is NP-complete to recognize if a given graph is a Wheeler graph [17], and thus, to use the efficient algorithmic machinery around Wheeler graphs [1] one needs to limit the focus on indexable graphs that admit efficient construction. Indeed, we show that any EFG that respects the repeat-free property can be reduced to a Wheeler graph in polynomial time. Interestingly, we were not able to modify this reduction to cover the semi-repeat-free case, leaving it open if these two notions of graph indexability have indeed different expressive power, and whether there are more graph classes with distinctive properties in this context.

2 Definitions

Strings. We denote integer intervals by $[i..j]$. Let $\Sigma = \{1, \dots, \sigma\}$ be an alphabet of size $|\Sigma| = \sigma$. A *string* $T[1..n]$ is a sequence of symbols from Σ, i.e. $T \in \Sigma^n$, where Σ^n denotes the set of strings of length n under the alphabet Σ. A *suffix* of string $T[1..n]$ is $T[i..n]$ for $1 \leq i \leq n$. A *prefix* of string $T[1..n]$ is $T[1..i]$ for $1 \leq i \leq n$. A *substring* of string $T[1..n]$ is $T[i..j]$ for $1 \leq i \leq j \leq n$. The *length* of a string T is denoted $|T|$. The *empty string* is the string of length 0. In particular, substring $T[i..j]$ where $j < i$ is the empty string. The *lexicographic order* of two strings A and B is naturally defined by the order of the alphabet: $A < B$ iff $A[1..i] = B[1..i]$ and $A[i+1] < B[i+1]$ for some $i \geq 0$. If $i+1 > \min(|A|, |B|)$, then the shorter one is regarded as smaller. However, we usually avoid this implicit comparison by adding *end marker* **0** to the strings. Concatenation of strings A and B is denoted AB.

Elastic founder graphs. As mentioned in the introduction, our goal is to compactly represent an MSA using an elastic founder graph. In this section we formalize these concepts.

A *multiple sequence alignment* $\mathsf{MSA}[1..m, 1..n]$ is a matrix with m strings drawn from $\Sigma \cup \{\text{-}\}$, each of length n, as its rows. Here $\text{-} \notin \Sigma$ is the *gap* symbol. For a string $X \in (\Sigma \cup \{\text{-}\})^*$, we denote $\mathsf{spell}(X)$ the string resulting from removing the gap symbols from X.

Let $\mathcal{P}$ be a *partitioning* of $[1..n]$, that is, a sequence of subintervals $\mathcal{P} = [x_1..y_1], [x_2..y_2], \ldots, [x_b..y_b]$, where $x_1 = 1$, $y_b = n$, and for all $j > 2$, $x_j = y_{j-1} + 1$. A *segmentation* S of $\mathsf{MSA}[1..m, 1..n]$ based on partitioning $\mathcal{P}$ is a sequence of b sets $S^k = \{\mathsf{spell}(\mathsf{MSA}[i, x_k..y_k]) \mid 1 \leq i \leq m\}$ for $1 \leq k \leq b$; in addition, we require for a (proper) segmentation that $\mathsf{spell}(\mathsf{MSA}[i, x_k..y_k])$ is not an empty string for any i and k. We call set S^k a *block*, while $\mathsf{MSA}[1..m, x_k..y_k]$ or just $[x_k..y_k]$ is called a *segment*. The *length* of block S^k is $L(S^k) = y_k - x_k + 1$ and the *width* of block S^k is $W(S^k) = |S^k|$. Segmentation naturally leads to the definition of a founder graph through the block graph concept:

▶ **Definition 1** (Block Graph). *A* block graph *is a graph $G = (V, E, \ell)$ where $\ell : V \to \Sigma^+$ is a function that assigns a string label to every node and for which the following properties hold.*

1. *Set V can be partitioned into a sequence of* b blocks *$V^1, V^2, \ldots, V^b$, that is, $V = V^1 \cup V^2 \cup \cdots \cup V^b$ and $V^i \cap V^j = \emptyset$ for all $i \neq j$;*
2. *If $(v, w) \in E$ then $v \in V^i$ and $w \in V^{i+1}$ for some $1 \leq i \leq b-1$; and*
3. *if $v, w \in V^i$ then $|\ell(v)| = |\ell(w)|$ for each $1 \leq i \leq b$ and if $v \neq w$, $\ell(v) \neq \ell(w)$.*

With *gapless* MSAs, block S^k equals segment $\mathsf{MSA}[1..m, x_k..y_k]$, and in that case the *founder graph* is a block graph induced by segmentation S [23]. The idea is to have a graph in which the nodes represent the strings in S while the edges retain the information of how such strings can be recombined to spell any sequence in the original MSA. With general MSAs with gaps, we consider the following extension, with an analogy to EDSes [5]:

▶ **Definition 2** (Elastic block and founder graphs). *We call a block graph* elastic *if its third condition is relaxed in the sense that each V^i can contain non-empty variable-length strings. An* elastic founder graph *(EFG) is an elastic block graph $G(S) = (V, E, \ell)$ induced by a segmentation S as follows: For each $1 \leq k \leq b$ we have $S^k = \{\mathsf{spell}(\mathsf{MSA}[i, x_k..y_k]) \mid 1 \leq i \leq m\} = \{\ell(v) : v \in V^k\}$. It holds $(v, w) \in E$ if and only if there exists $k \in [1..b-1]$ and $t \in [1..m]$ such that $v \in V^k$, $w \in V^{k+1}$ and $\mathsf{spell}(\mathsf{MSA}[t, x_k..y_{k+1}]) = \ell(v)\ell(w)$.*

By definition, (elastic) founder and block graphs are acyclic. For convention, we interpret the direction of the edges as going from left to right. Consider a path P in $G(S)$ between any two nodes. The label $\ell(P)$ of P is the concatenation of labels of the nodes in the path. Let Q be a query string. We say that Q *occurs* in $G(S)$ if Q is a substring of $\ell(P)$ for any path P of $G(S)$. Figure 1 illustrates such a query.

We use the same repeat-free definition as in the non-elastic case [23]:

▶ **Definition 3.** *EFG $G(S)$ is* repeat-free *if each $\ell(v)$ for $v \in V$ occurs in $G(S)$ only as prefix of paths starting with v.*

We also consider a variant that is relevant due to variable-length strings in the blocks:

▶ **Definition 4.** *EFG $G(S)$ is* semi-repeat-free *if each $\ell(v)$ for $v \in V$ occurs in $G(S)$ only as prefix of paths starting with $w \in V$, where w is from the same block as v.*

These definitions also apply to general elastic block graphs and to elastic degenerate strings as their special case.

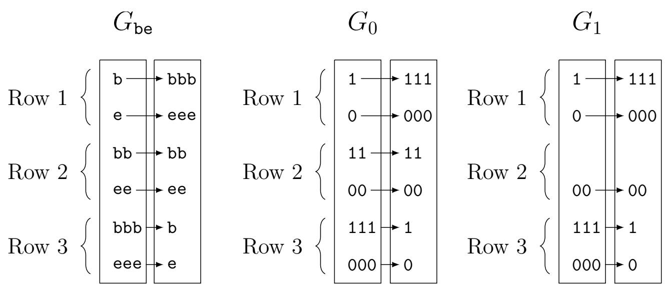


Figure 2 Gadgets $G_{\mathtt{be}}$, G_0 and G_1. Each gadget is organized into three rows, each row encoding a different partitioning of the strings bbbb, eeee, 0000, 1111. This ensures that, when combining these gadgets in Figure 3, edges can be controlled to go within the same row, or to the row below.

We note that not all MSAs admit a segmentation leading to a (semi-)repeat-free EFG, e.g. an alignment with rows -A and AA. However, our algorithms detect such cases, thus one can build an EFG consisting of just one block with the rows of the MSA (with gaps removed). Such EFGs can be indexed using standard string data structures to support efficient queries.

3 Conditional Hardness of Indexing EFGs

We show a reduction from *Orthogonal Vectors* (OV) to the problem of matching a query string in an EFG, continuing the line of research conducted on many related (degenerate) string problems [19, 10, 2, 16]. The OV problem is to find out if there exist $x \in X$ and $y \in Y$ such that $x \cdot y = 0$, given two sets X and Y of n binary vectors each. We construct string Q using X and graph G using Y. Then, we show that Q has a match in G if and only if X and Y form a "yes"-instance of OV. We condition our results on the following OV hypothesis, which is implied by the *Strong Exponential Time Hypothesis* [20].

▶ **Definition 5** (Orthogonal Vectors Hypothesis (OVH) [36]). *Let X, Y be the two sets of an OV instance, each containing n binary vectors of length d.*[1] *For any constant $\epsilon > 0$, no algorithm can solve OV in time $O(poly(d)n^{2-\epsilon})$.*

Query String. We build string Q by combining string gadgets $Q_1, \ldots, Q_n$, one for each vector in X, plus some additional characters. To build string Q_i, first we place four b characters, then we scan vector $x_i \in X$ from left to right. For each entry of x_i, we place substring $Q_{i,h}$ consisting of four 0 characters if $x_i[h] = 0$, or four 1 characters if $x_i[h] = 1$. Finally, we place four e characters. For example, vector $x_i = 101$ results into string

$$Q_i = \mathtt{bbbb}\, Q_{i,1}\, Q_{i,2}\, Q_{i,3}\, \mathtt{eeee}, \text{ where } \quad Q_{i,1} = \mathtt{1111}, \quad Q_{i,2} = \mathtt{0000}, \quad Q_{i,3} = \mathtt{1111}.$$

Full string Q is then the concatenation $Q = \mathtt{bbbb}Q_1Q_2 \ldots Q_n\mathtt{eeee}$. The reason behind these specific quantities will be clear when discussing the structure of the graph.

Elastic Founder Graph. We build graph G combining together three different sub-graphs: G_L, G_M, G_R (for *left*, *middle* and *right*). Our final goal is to build a graph structured in three logical "rows". We denote the three rows of G_M as G_{M1}, G_{M2}, G_{M3}, respectively. The

[1] In this section, keeping in line with the usual notation in the OV problem, we use n to denote the size of X and Y, instead of the number of columns of the MSA.

first and the third rows of G, along with subgraphs G_L and G_R (introduced to allow slack), can match any vector. The second row matches only sub-patterns encoding vectors that are orthogonal to the vectors of set Y. The key is to structure the graph such that the pattern is forced to utilize the second row to obtain a full match. We present the full structure of the graph in Figure 3, which shows the graph built on top of vector set {100, 011, 010}. In particular, G_M consists of n gadgets G_M^j, one for each vector $y_j \in Y$. The key elements of these sub-graphs are gadgets $G_{\mathtt{be}}$, $G_{\mathtt{0}}$ and $G_{\mathtt{1}}$ (see Figure 2), which allow to stack together multiple instances of strings $\mathtt{b}^4$, $\mathtt{e}^4$, $\mathtt{1}^4$, $\mathtt{0}^4$. The overall structure mimics the one by Equi et al. [10], except for the new idea from Figure 2.

Detailed structure of the graph. **Sub-graph G_L** (Figure 3a) consists of a starting segment with a single node labeled $\mathtt{b}^4$, followed by $n-1$ sub-graphs $G_L^1, \dots, G_L^{n-1}$, in this order. Each G_L^i has $d+2$ segments, and is obtained as follows. First, we place a segment containing only one node with label $\mathtt{b}^4$, then we place d other segments, each one containing two nodes with labels $\mathtt{1}^4$ and $\mathtt{0}^4$. Finally, we place a segment containing two nodes with labels $\mathtt{b}^4$ and $\mathtt{e}^4$.

The nodes in each segment are connected to all nodes in the next segment, with the exception of the last segment of each G_L^i: in this case, the node with label $\mathtt{1}^4$ and the one with label $\mathtt{0}^4$ are connected only to the $\mathtt{e}^4$-node of the next (and last) segment of such G_L^i.

Sub-graph G_R (Figure 3c) is similar to sub-graph G_L, and it consists in $n-1$ parts $G_R^1, \dots, G_R^{n-1}$, followed by a segment with a single node labeled $\mathtt{e}^4$. Part G_R^i has $d+2$ segments, and is constructed almost identically to G_L^i. The differences are that, in the first segment of G_R^i, we place two nodes labeled $\mathtt{b}^4$ and $\mathtt{e}^4$, while in the last segment we place only one node, which we label $\mathtt{e}^4$.

As in G_L, the nodes in each segment are connected to all nodes in the next segment, with the exception of the first segment of each G_R^i: in this case, the node labeled $\mathtt{e}^4$ has no outgoing edge.

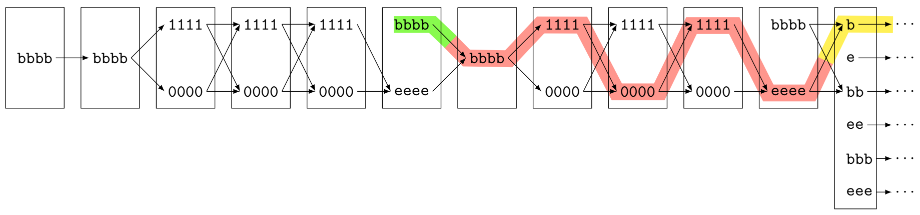
Sub-graph G_M (Figure 3b) implements the main logic of the reduction, and it uses three building blocks, $G_{\mathtt{be}}$, $G_{\mathtt{0}}$ and $G_{\mathtt{1}}$, which are organized in three rows, as shown in Figure 2.

Sub-graph G_M has n parts, $G_M^1, \dots, G_M^n$, one for each of the vectors $y_1, \dots, y_n$ in set Y. Each G_M^j is constructed, from left to right, as follows. First, we place a $G_{\mathtt{be}}$ gadget. Then, we scan vector y_j from left to right and, for each position $h \in \{1, \dots, d\}$, we place a $G_{\mathtt{0}}$ gadget if the h-th entry is $y_j[h] = \mathtt{0}$, or a $G_{\mathtt{1}}$ if $y_j[h] = \mathtt{1}$. Finally, we place another $G_{\mathtt{be}}$ gadget.

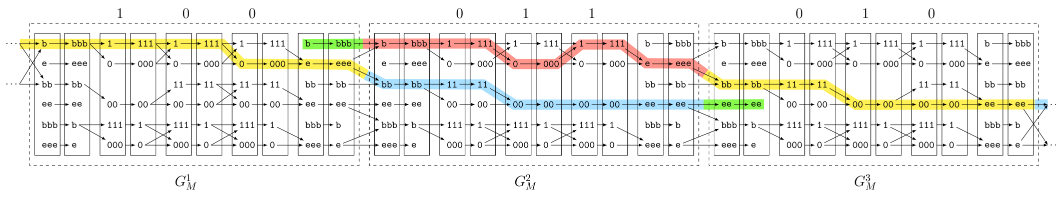
For the edges, we first consider each gadget G_M^j separately. Let G_h and G_{h+1}, be the gadgets encoding $y_j[h]$ and $y_j[h+1]$, respectively. We fully connect the nodes of G_h to the nodes of G_{h+1} row by row, respecting the structure of the segments. Then we connect, row by row, the $\mathtt{b}$-nodes of the left $G_{\mathtt{be}}$ to the leftmost G_h, which encodes $y_j[1]$, and the nodes of the rightmost G_h, which encodes $y_j[d]$, to the $\mathtt{e}$-nodes of the right $G_{\mathtt{be}}$, again row by row. We repeat the same placement of the edges for every vector G_h, G_{h+1}, $1 \le h \le d-1$; this construction is shown in Figure 3b.

To conclude the construction of G_M, we need to connect all the G_M^j gadgets together. Consider the right $G_{\mathtt{be}}$ of gadget G_M^j, and the left $G_{\mathtt{be}}$ of gadget G_M^{j+1}. The edges connecting these two gadgets are depicted in Figure 3b, which shows that following a path we can either remain in the same row or move to the row below, but we cannot move to the row above. Moreover, sub-pattern $\mathtt{b}^8$ can be matched only in the first and second row, while sub-pattern $\mathtt{e}^8$ only in the second and third rows.

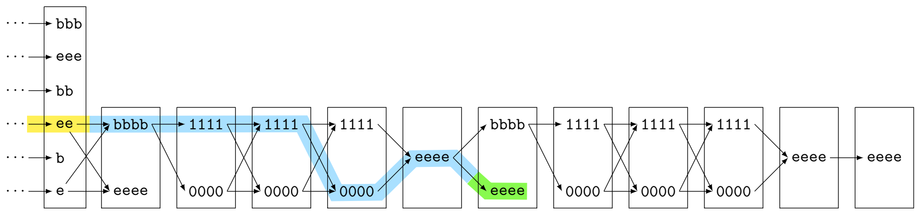
In proving the correctness of the reduction, we will refer to graphs G_{M1}, G_{M2} and G_{M3} as the sub-graphs of G_M consisting of only the nodes and edges of the first, second and third row, respectively. Formally, for $t \in \{1, 2, 3\}$, $V_{Mt} \subset V$ $V_{Mt} \subset V$ is the set of



(a) Sub-graph G_L. The last segment belongs to sub-graph G_M and shows the connection.



(b) Sub-graph G_M for vectors $y_1 = 100$, $y_2 = 011$ and $y_3 = 010$. The dashed rectangles highlight the single G_M^j gadgets.



(c) Sub-graph G_R. The first segment belongs to sub-graph G_M and shows the connection.

Figure 3 An example of graph G. To visualize the entire graph, watch the three sub-figures from top to bottom and from left to right. We also show two example occurrences of a query string Q constructed from $x_1 = 101$, $x_2 = 110$, $x_3 = 100$ (left-most), and from $x_1 = 101$, $x_2 = 100$, $x_3 = 110$ (right-most), respectively. We highlight each Q_i with a different color. Any such occurrence must pass through the middle row of G_M.

nodes placed in the t-th row of each $G_{\texttt{be}}$, G_0 or G_1 gadget belonging to sub-graph G_M, and $E_{Mt} = \{(v, w) \in E \mid v, w \in V_{Mt}\}$. Thus, $G_{Mt} = (V_{Mt}, E_{Mt})$. We will use the notation G^j_{M2} to refer to the nodes belonging to both G^j_M and G_{M2}, excluding the ones in G_{M1} and G_{M3}, and the edges connecting them.

Final graph *G* is obtained by combining sub-graphs G_L, G_M and G_R. To this end, we connect the nodes in the last segment of G_L with the b-nodes in the first and second row of the left $G_{\texttt{be}}$ gadget of G^1_M. Finally, we connect the e-nodes in the second and third row of the right $G_{\texttt{be}}$ gadget of G^n_M with both the $\texttt{b}^4$-node and $\texttt{e}^4$-node in the first segment of G_R. Figures 3a, 3b and 3c can be visualized together, in this order, as one big picture of final graph G. In Figures 3a and 3c we placed the adjacent segment of G_M to show the connection.

OVH Conditional Hardness. The proof of correctness is similar to the one by Equi et al. [10], but with adaptations to the elastic founder graph. The following technical lemma (whose proof is deferred to the full version of this paper) summarizes the structure of the possible matches of Q inside G. In it, we use the notation G^j_{M2} to indicate the nodes and edges that belong to the second row of G^j_M.

▶ **Lemma 6.**

- *If string Q_i has a match in G_{M2}, then the path matching Q_i is fully contained in G^j_{M2}, for some $1 \le j \le n$. Moreover, each $Q_{i,h}$ substring matches a path of two nodes which belong to the G_0 or G_1 gadget encoding $y_j[h]$.*
- *String Q_i has a match in G_{M2} if and only if there exist $y_j \in Y$ such that $x_i \cdot y_j = 0$.*
- *String Q has a match in G if and only if a substring Q_i of Q has a match in the underlying sub-graph G_{M2} of G_M.*

Our first lower bound is on matching a query string in an EFG without indexing.

▶ **Theorem 7.** *For any constant $\epsilon > 0$, it is not possible to find a match for a query string Q into an EFG $G = (V, E, \ell)$ in either $O(|E|^{1-\epsilon}\,|Q|)$ or $O(|E|\,|Q|^{1-\epsilon})$ time, unless OVH fails. This holds even if restricted to an alphabet of size 4.*

Proof sketch. First, Lemma 6 guarantees that string Q has a match in G if and only if there exist orthogonal vectors $x_i \in X$ and $y_j \in Y$. Second, it is easy to check that the reduction requires linear time and space in the size $O(nd)$ of the OV problem. Third, if we find a match for Q in G in $O(|E|^{1-\epsilon}|Q|)$ or $O(|E|\,|Q|^{1-\epsilon})$ time, then we can decide if there is a pair of orthogonal vectors in $O(nd \cdot (nd)^{1-\epsilon}) = O(n^{2-\epsilon}\text{poly}(d))$ time, contradicting OVH. ◀

We obtain the indexing lower bound by proving that the above reduction is a *linear independent-components* (*lic*) reduction, as defined by [11, Definition 3].

▶ **Theorem 8.** *For any $\alpha, \beta, \delta > 0$ such that $\beta + \delta < 2$, there is no algorithm preprocessing an EFG $G = (V, E, \ell)$ in time $O(|E|^\alpha)$ such that for any query string Q we can find a match for Q in G in time $O(|Q| + |E|^\delta |Q|^\beta)$, unless OVH is false. This holds even if restricted to an alphabet of size 4.*

Proof. It is enough to notice that the reduction from OV that we presented is a *lic* reduction. Namely, (1) the reduction is correct and can be performed in linear time and space $O(nd)$ (recall the proof of Theorem 7), and (2) query string $|Q|$ is defined using only vector set X and it is independent from vector set Y, while elastic founder graph G is built using only vector set Y and it is independent from vector set X. Hence, Corollary 1 in [11] can be applied, proving our thesis. ◀

4 Indexing (Semi-)Repeat-Free EFGs

Since indexing a general EFG is hard, we turn our attention to repeat-free and semi-repeat-free EFGs. We show in this section that such EFGs are easy to index.

Let $G = (V, E)$ be a (semi-)repeat-free founder/block graph. We show that it is sufficient to build an index on a string formed by concatenating labels of neighboring nodes. Namely, consider string $D = \prod_{i\in\{1,2,\ldots,b\}} \prod_{v\in V^i,(v,w)\in E} \ell(w)^{-1}\ell(v)^{-1}\mathbf{0}$, where X^{-1} is the reverse $x_m x_{m-1} \cdots x_1$ of string $X = x_1 x_2 \cdots x_m$. The key feature requiring this reversed direction is that the lexicographic ranges of suffixes of D starting with $\ell(v)^{-1}$ for each $v \in V$ are distinct; this would not (on all inputs) hold on suffixes starting with $\ell(v)$ in a forward concatenation if $\ell(v)$ is a prefix of some $\ell(u)$, as it can be in the semi-repeat-free case.

As the reader can easily verify, the *expanded backward search* [23] developed for the case of gapless MSAs applied on D (in place of the forward concatenation therein) works also for repeat-free founder/block graphs; the feature of having variable-length of strings in a block is not used in the correctness analysis. In the following, we give an alternative solution for the semi-repeat-free case using suffix trees.

Let us consider a solution based on the *descending suffix walk*. This search can be supported e.g. by any (compressed) suffix tree [12, 33]. In the following, we assume the reader is familiar with the basic notions on suffix tree [22, Chapter 8]. Consider searching query $Q[1..q]$ from right to left from the suffix tree of D. Consider finding a match for $Q[j..q]$, but there is no branch with $Q[j-1]$. If there is a branch with **0**, there may be j' such that $Q[j..j']$ is a label of some node v of the semi-repeat-free EFG, for some $j' \leq q$. To find such j' (or to find out it does not exist), we can then take suffix links to reach the locus corresponding to $Q[j..j']$, and continue the search with $Q[1..j'-1]$. This process is repeated until Q is found, or, if one cannot proceed, Q does not occur in G. For this to work, we need to have marked all loci of the suffix tree reached by $\ell(v)^{-1}$ for all nodes v of the semi-repeat-free EFG. We stop taking suffix links when reaching a marked loci.

▶ **Theorem 9.** *A (semi-)repeat-free founder/block graph $G = (V, E)$ or a (semi-)repeat-free elastic degenerate string can be indexed in polynomial time into a data structure occupying $O(|D| \log |D|)$ bits of space, where $|D| = O(L|E|)$ and L is the maximum length of a node label. Later, one can find out in $O(|Q|)$ time if a given query string Q occurs in G.*

Proof. Consider the approach above. If and only if Q is reported to be found, there is a path in G that spells Q: If Q is found without taking any suffix links, it is a substring of D and thus a substring of a concatenation of labels of two neighboring nodes of G. Otherwise, suffix $Q[j..q]$ is a prefix of a path starting with $\ell(w) = Q[j..j']$ for some $w \in V$. Since $\ell(w)^{-1}$ occurs in D only when followed by **0** or $\ell(v)^{-1}$ for $(v, w) \in E$, the search continues only on the in-neighbors of w. If Q is found before taking suffix links again, $Q[1..j-1]$ is a suffix of $\ell(v)$ for some v such that $(v, w) \in E$, and thus there is a path in G spelling Q. Otherwise, suffix $Q[k..q]$ is a prefix of a path starting with $\ell(v)\ell(w)$, for some $k \leq j-1$. Continuing this shows that the claim is correct.

Clearly, the length of D is bounded by $O(L|E|)$. Construction of suffix tree on D can be done in linear time [12]. In polynomial time, the nodes of the suffix tree can be preprocessed with perfect hash functions, such that following a downward path takes constant time per step. Following a suffix link takes amortized constant time. ◀

Observe that $|D| \leq 2mn$, where m and n are the number of rows and number of columns, respectively, in the MSA from where the elastic founder graph is induced. That is, the index size is linear in the (original) input size. We also note that the index can be modified to report only matches that are (gap-oblivious) substrings of the MSA rows: Short patterns spanning only one edge are already such. Longer patterns can have only one occurrence in G, and it suffices to verify them with a regular string index on the MSA. Such modified scheme makes the approach functionality equivalent with wide range of indexes designed for repetitive collections [24, 29, 30, 28, 27, 14, 15] and shares the benefit of alignment-based indexes of Na et al. [29, 30, 28, 27] in reporting the aligned matches only once, where e.g. r-index [15] needs to report all occurrences.

Using compressed suffix trees, different space-time tradeoffs can be achieved. We develop an alternative compressed indexing scheme in Section 6 using Wheeler graphs.

5 Construction of (Semi-)Repeat-Free EFGs

Now that we have seen that (semi-)repeat-free EFGs are easy to index, it remains to consider their construction.

Consider a segmentation $S = S^1, S^2, \ldots, S^b$ that induces an EFG $G(S) = (V, E)$. We call such a segmentation (semi-)repeat-free or simply *valid*, if the resulting EFG is (semi-)repeat-free. A segment $\mathsf{MSA}[1..m, x_k..y_k]$ corresponding to block $S^k = \mathsf{spell}(\mathsf{MSA}[1..m, x_k..y_k])$ of such (semi-)repeat-free S is then analogously called a (semi-)repeat-free segment. In an earlier work on gapless MSAs [23], it was observed that a sufficient and necessary condition to check if a segment $\mathsf{MSA}[1..m, x_k..y_k]$ is repeat-free is to check that no string $\mathsf{MSA}[i, x_k..y_k]$, $1 \leq i \leq m$, occurs elsewhere in the MSA except at $\mathsf{MSA}[i', x_k..y_k]$, for some $1 \leq i' \leq m$. Let us refine this condition with gaps. We say that segment $\mathsf{MSA}[1..m, x_k..y_k]$ is repeat-free (semi-repeat-free) if no string $\mathsf{spell}(\mathsf{MSA}[i, x_k..y_k])$, $1 \leq i \leq m$, occurs elsewhere in the MSA except at (as a prefix of, respectively) $\mathsf{spell}(\mathsf{MSA}[i', 1..n])[g(i', x_k)..g(i', y_k)]$, for some $1 \leq i' \leq m$, where $g(i', j)$ is j subtracted with the number of gaps at row i' of MSA up to column j. The earlier arguments [23, Sect. 5.1] carry over to showing that these are sufficient and necessary conditions for inducing a repeat-free and semi-repeat-free EFGs, respectively.

We consider three score functions for the valid segmentations, one maximizing the number of blocks, one minimizing the maximum width of a block, and one minimizing the maximum length of a block. The latter two have been studied earlier without the (semi-)repeat-free constraint, and non-trivial linear time solutions have been found [31, 8], while the first score function makes sense only with this new constraint.

Let $s(j')$ be the optimal score of a semi-repeat-free segmentation $S^1, S^2, \ldots, S^b$ of prefix $\mathsf{MSA}[1..m, 1..j']$ for a selected scoring scheme. Then

$$s(j) = \bigoplus_{\substack{j' : 0 \leq j' < j, \\ \mathsf{MSA}[1..m, j'+1..j] \text{ is semi-repeat-free segment}}} w(s(j'), j', j), \tag{1}$$

gives the optimal score of a semi-repeat-free segmentation $S^1, S^2, \ldots, S^b, S^{b+1}$ of $\mathsf{MSA}[1..m, 1..j]$, where $\bigoplus$ is an operator depending on the scoring scheme and $w(x, j', j)$ is a function on the score x of the segmentation of $S^1, S^2, \ldots, S^b$ and on the last block S^{b+1} corresponding to $\mathsf{MSA}[1..m, j'+1..j]$. To fix this recurrence so that $s(n)$ equals the maximum number of blocks over valid segmentations of $\mathsf{MSA}[1..m, 1..n]$, set $\bigoplus = \max$ and $w(x, j', j) = x + 1$. For initialization, set $s(j) = 0$. Moreover, when there is no valid segmentation for some j, set $s(j) = -\infty$. To fix this recurrence so that $s(n)$ equals the minimum of maximum widths of blocks over valid segmentations of $\mathsf{MSA}[1..m, 1..n]$, set $\bigoplus = \min$ and $w(x, j', j) = \max(x, |\{\mathsf{spell}(\mathsf{MSA}[i, j'+1..j]) \mid 1 \leq i \leq m\}|)$. For initialization, set $s(j) = 0$. Moreover, when there is no valid segmentation for some j, $s(j) = \infty$. Finally, to fix this recurrence so that $s(n)$ equals the minimum of maximum length of blocks over valid segmentations of $\mathsf{MSA}[1..m, 1..n]$, set $\bigoplus = \min$ and $w(x, j', j) = \max(x, j - j')$. For initialization, set $s(j) = 0$. Moreover, when there is no valid segmentation for some j, set $s(j) = \infty$.

The recurrence fixed for the latter case can be solved in $O(mn)$ time when the input is a gapless MSA [23]. However, gaps affect most of the more involved techniques [23, 31, 8], so that we only know of a rather straightforward solution working in $O(mn^2 \log m)$ time for this general case with gaps, for all three score functions and also for the repeat-free case (explicit proof omitted here, but the techniques developed later in this paper are sufficient for deriving such result). In what follows, we develop a different approach that works in $O(mn \log m)$ time for the first and the last score function in the semi-repeat-free case. We leave it as an open question to obtain a faster algorithm for the second score function, and for the repeat-free case. We start with a simple observation:

▶ **Observation 10.** *If segment $\mathsf{MSA}[1..m, j+1..f(j)]$ is semi-repeat-free, then segment $\mathsf{MSA}[1..m, j+1..j']$ is semi-repeat-free for all j' such that $f(j) < j' \leq n$.*

Our goal is to compute for each j the smallest integer $f(j)$ such that $\mathsf{MSA}[1..m, j+1..f(j)]$ is a semi-repeat-free segment. These values can then be used for efficient evaluation of Eq. (1).

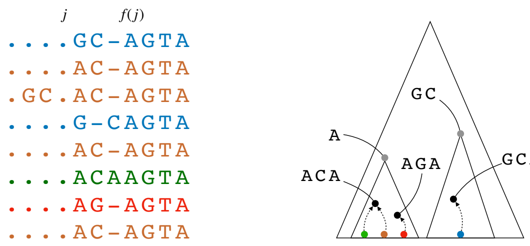


Figure 4 Illustrating the $O(m \log m)$ time algorithm to compute value $f(j)$ for a given j. Node labels correspond to the string spelled from the root to the node. We assume `ACA`, `AGA`, and `GCA` only appear in the region of the MSA visualized, while `GC` and `A` appear also elsewhere.

▶ **Lemma 11.** *Let $f(j)$ be the smallest integer such that* $\mathsf{MSA}[1..m, j+1..f(j)]$ *is a semi-repeat-free segment. We can compute all values $f(j)$ in $O(mn \log m)$ time.*

Proof sketch. Full proof is deferred to the full version of this paper. Here we explain the algorithm through the example of Fig. 4. We build a compact trie on the suffixes of the concatenation of MSA rows with gaps removed and special markers added between rows, that is, a *generalized suffix tree* [22, Chapter 8] on set $\{\mathsf{spell}(\mathsf{MSA}[i, 1..n]) \mid 1 \leq i \leq m\}$. For each column j, locate the subset W of (implicit) suffix tree nodes corresponding to $\{\mathsf{spell}(\mathsf{MSA}[i, j+1..n]) \mid 1 \leq i \leq m\}$; these are the colored nodes in Fig. 4. If the number of leaves covered by the subtrees rooted at W is greater than m, $f(j)$ remains undefined. Otherwise, we know that $f(j) \leq n$, and our aim is to decrease the right boundary, starting with n, until we have reached column $f(j)$. The algorithm picks an arbitrary node in W and tries to replace it with its parent. This replacement is safe, if the new subtree covers only leaves already covered by W. Safe replacements are continued as long as possible; these are the black nodes in Fig. 4, while the gray nodes are unsafe replacements. These replacements place the rows into equivalence classes, each sharing the identified common prefix. For row i in an equivalence class with common prefix length k, one can then locate the smallest column $f^i(j)$ with $|\mathsf{spell}(\mathsf{MSA}[i, j+1..f^i(j)])| = k$. E.g. for row $i = 2$ in Fig. 4, we have $f^2(j) = j + 3$, as $|\mathsf{spell}(\mathtt{AC-A})| = 3 = |\mathtt{ACA}|$. Finally, $f(j) = \max_{i:1 \leq i \leq m} f^i(j)$. At each column, at most m replacements are required and each replacement can be done in $O(\log m)$ time, by maintaining the non-overlapping lexicographic intervals corresponding to the suffix tree nodes in a balanced search tree. ◀

Using these precomputed $f(j)$ values, Algorithms 1 and 2 compute the scores of an optimal semi-repeat-free segmentation under the maximum number of blocks score and minimum of maximum block length score, respectively.

▶ **Theorem 12.** *After an $O(mn \log m)$ time preprocessing, Algorithms 1 and 2 compute the scores* $\mathtt{maxblocks}(n) = b$ *and* $\mathtt{minmaxlength}(n) = \max\limits_{i:1 \leq i \leq b} L(S^i)$ *of optimal semi-repeat-free segmentations $S^1, S^2, \ldots, S^b$ of* $\mathsf{MSA}[1..m, 1..n]$ *in $O(n)$ and $O(n \log \log n)$ time, respectively.*

Proof sketch. Regarding running time, Algorithm 1 and Algorithm 2 clearly take $O(n)$ and $O(n \log n)$ time, respectively, when implemented as described in their pseudo-codes. The correctness for Algorithm 1 follows from the fact that when computing the score at column j, all earlier segmentations that are safe to be extended with a new segment ending at j are considered. This argument can be formalized analogously for Algorithm 2, whose detailed proof of correctness, as well as running time improvement to $O(n \log \log n)$ (by using a more efficient data structure for semi-infinite range queries) are given in the full version of this paper. ◀

Algorithm 1 An $O(n)$ time algorithm for finding an optimal semi-repeat-free segmentation maximizing the number of blocks.

```
Input: Right-extensions $(j, f(j))$ sorted from smallest to largest order by second
       component: $(j_1, f(j_1)), (j_2, f(j_2)), \dots, (j_{n-J}, f(j_{n-J}))$, where $J$ is such that
       $f(j_{n-J+1}), f(j_{n-J+2}), \dots, f(j_n)$ are not defined.
Output: Score of an optimal semi-repeat-free segmentation maximizing the number
        of blocks.
1 $x \leftarrow 1$; $\texttt{maxblocks}(0) \leftarrow 0$; $\texttt{maxblocks}(j) = \texttt{maxscore} = -\infty$ for all $0 < j \leq n$;
2 for $j \leftarrow 1$ to $n$ do
3     while $j = f(j_x)$ do
4         $\texttt{maxscore} \leftarrow \max(\texttt{maxscore}, \texttt{maxblocks}(j_x))$;
5         $x \leftarrow x + 1$;
6     $\texttt{maxblocks}(j) \leftarrow \texttt{maxscore} + 1$;
7 return $\texttt{maxblocks}(n)$;
```

Algorithm 2 An $O(n \log n)$ time algorithm for finding an optimal semi-repeat-free segmentation minimizing the maximum segment length. Minimization over an empty set is assumed to return ∞. Operation $\texttt{Upgrade}(k, v)$ sets key k to value v if the previous value is larger. Operation $\texttt{RangeMin}(a, b)$ returns the smallest value associated with keys in range $[a..b]$. Both operations can be supported in $O(\log n)$ time with standard balanced search trees.

```
Input: Right-extensions $(j, f(j))$ sorted from smallest to largest order by second
       component: $(j_1, f(j_1)), (j_2, f(j_2)), \dots, (j_{n-J}, f(j_{n-J}))$, where $J$ is such that
       $f(j_{n-J+1}), f(j_{n-J+2}), \dots, f(j_n)$ are not defined.
Output: Score of an optimal semi-repeat-free segmentation minimizing the
        maximum segment length.
1 Initialize one-dimensional search trees $\mathcal{T}$ and $\mathcal{I}$ with keys $0, 1, 2, \dots, 2n$, with all keys
  associated with values $\infty$;
2 $x \leftarrow 1$;
3 $\texttt{minmaxlength}(0) \leftarrow 0$;
4 for $j \leftarrow 1$ to $n$ do
5     while $j = f(j_x)$ do
6         $\mathcal{T}.\texttt{Upgrade}(j_x + \texttt{minmaxlength}(j_x), \texttt{minmaxlength}(j_x))$;
7         $\mathcal{I}.\texttt{Upgrade}(j_x + \texttt{minmaxlength}(j_x), -j_x)$;
8         $x \leftarrow x + 1$;
9     $\texttt{minmaxlength}(j) \leftarrow \min(\mathcal{T}.\texttt{RangeMin}(j + 1, \infty), \mathcal{I}.\texttt{RangeMin}(-\infty, j) + j)$;
10 return $\texttt{minmaxlength}(n)$;
```

6 Connection to Wheeler Graphs

Wheeler graphs, also known as Wheeler automata, are a class of labeled graphs that admit an efficient index for path queries [13]. We now give an alternative way to index repeat-free elastic block graphs by transforming the graph into an equivalent Wheeler automaton.

We view a block graph as a nondeterministic finite automaton (NFA) by adding a new initial state and edges from the source node to the starts of the first block, and expanding each string of each block to a path of states. To conform with automata notions, we define that the label of an edge is the label of the destination node.

We denote the repeat-free NFA with F. First we determinize it with the standard subset construction for the reachable subsets of states. The states of the DFA are subsets of states of the NFA such that there is an edge from subset S_1 to subset S_2 with label c iff S_2 is the set of states at the destinations of edges labeled with c from S_1. We only represent the subsets of states reachable from the subset containing only the initial state. We call the deterministic graph G. See Figures 5 and 6 for an example.

A DFA is indexable as a Wheeler graph if there exists an order $<$ on the nodes such that if $u < v$, then every incoming path label to u is colexicographically smaller than every incoming path label to v (recall that the colexicographic order of strings is the lexicographic order of the reverses of the strings). The repeat-free property guarantees that the nodes at the ends of the blocks can be ordered among themselves by picking an arbitrary incoming path as the sorting key.

To make sure that the rest of the nodes are sortable, we modify the graph so that if a node is not at the end of a block, we make it so that the incoming paths to the node do not branch backward before the backward path reaches the end of a previous block. This is done by turning each block into a set of disjoint trees, where the roots of the trees are the ending nodes of the previous block, in a way that preserves the language of the automaton. The roots may have multiple incoming edges from the leaves of the previous tree. See Figure 7 for an example. The formal definition of the transformation and the proof of sortability are deferred to the full version of this paper. We denote the transformed graph with G' and obtain the following result:

▶ **Lemma 13.** *The number of nodes in G' is at most $O(NW)$, where W is the maximum number of strings in a block of F and N is the total number of nodes in F.*

The Wheeler order $<$ of the transformed graph can be found by running the XBWT sorting algorithm on a spanning tree of the graph, as shown by Alanko et al. [1]. Finally, we can find the minimum equivalent Wheeler graph by running the general Wheeler graph minimization algorithm of Alanko et al. [1].

With the input graph now converted into a Wheeler graph, one can deploy succinct data structures supporting fast pattern matching [13, Lemma 4], leading to the following result:

▶ **Corollary 14.** *A repeat-free founder/block graph G or a repeat-free elastic degenerate string can be indexed in $O(NW)$ time into a Wheeler-graph-based data structure occupying $O(NW \log |\Sigma|)$ bits of space, where N is the total number of characters in the node labels of G, W is the width of G (maximum number of strings in a block of G), and Σ is the alphabet. Later, using the data structure, one can find out in $O(|Q| \log |\Sigma|)$ time if a given query string Q occurs in G.*

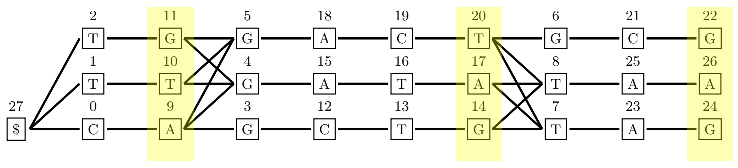

2 T 11 G 5 G 18 A 19 C 20 T 6 G 21 C 22 G
1 T 10 T 4 G 15 A 16 T 17 A 8 T 25 A 26 A
27 $ 0 C 9 A 3 G 12 C 13 T 14 G 7 T 23 A 24 G

Figure 5 Repeat-free block NFA. The last columns of each block are highlighted.

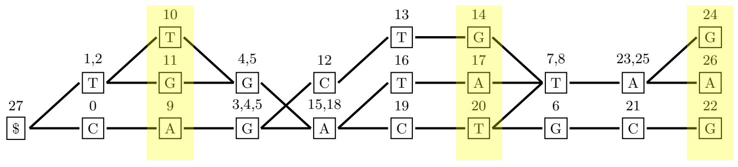

10 T 13 T 14 G 24 G
1,2 T 11 G 4,5 G 12 C 16 T 17 A 7,8 T 23,25 A 26 A
27 $ 0 C 9 A 3,4,5 G 15,18 A 19 C 20 T 6 G 21 C 22 G

Figure 6 The DFA resulting from the subset construction for the NFA in Figure 5. The numbers above the nodes specify the subset of NFA states corresponding to the DFA state.

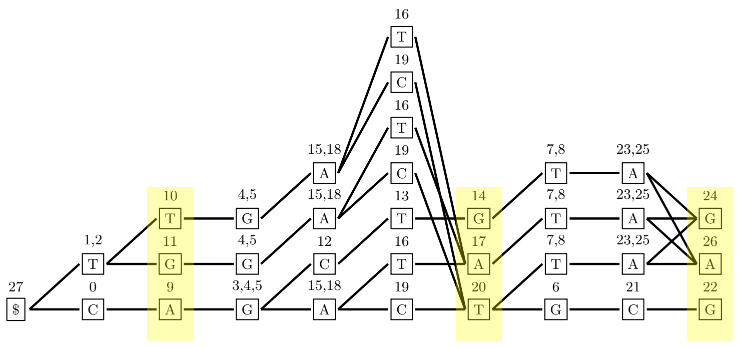

16 T
19 C
16 T
15,18 A 19 C 7,8 T 23,25 A
10 T 4,5 G 15,18 A 13 T 14 G 7,8 T 23,25 A 24 G
1,2 T 11 G 4,5 G 12 C 16 T 17 A 7,8 T 23,25 A 26 A
27 $ 0 C 9 A 3,4,5 G 15,18 A 19 C 20 T 6 G 21 C 22 G

Figure 7 The Wheeler DFA resulting from running our Wheeler expansion algorithm on the DFA in Figure 6.

7 Discussion

There are many options how to optimize among the valid segmentations [31, 8]. We studied some of these here under the (semi-)repeat-free indexability constraint, but left open how to e.g. minimize the maximum number of distinct strings in a segment (i.e. *width* of the graph) [31], or how to control the over-expressiveness of the graph, in this context.

Other open problems include strengthening the conditional indexing lower bound to cover non-elastic founder graphs, and improving the running time for constructing (semi-)repeat-free elastic founder graphs.

We focused on the theoretical aspects of indexable founder graphs. Our preliminary experiments [23] show that the approach works well in practice on multiple sequence alignments without gaps. In our future work, we will focus on making the approach practical also in the general case.

References

1 Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Regular languages meet prefix sorting. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 911–930. SIAM, 2020.

2 Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Comparing degenerate strings. *Fundam. Informaticae*, 175(1-4):41–58, 2020.

3 Amihood Amir, Moshe Lewenstein, and Noa Lewenstein. Pattern matching in hypertext. *J. Algorithms*, 35(1):82–99, 2000.

4 Kotaro Aoyama, Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Online Elastic Degenerate String Matching. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*, volume 105 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:10, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.CPM.2018.9`.

5 Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Even Faster Elastic-Degenerate String Matching via Fast Matrix Multiplication. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ICALP.2019.21`.

6 Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Pattern matching on elastic-degenerate text with errors. In Gabriele Fici, Marinella Sciortino, and Rossano Venturini, editors, *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings*, volume 10508 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 2017. `doi:10.1007/978-3-319-67428-5_7`.

7 Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Approximate pattern matching on elastic-degenerate text. *Theor. Comput. Sci.*, 812:109–122, 2020. `doi:10.1016/j.tcs.2019.08.012`.

8 Bastien Cazaux, Dmitry Kosolobov, Veli Mäkinen, and Tuukka Norri. Linear time maximum segmentation problems in column stream model. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings*, volume 11811 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2019.

9 Maria Chatzou, Cedrik Magis, Jia-Ming Chang, Carsten Kemena, Giovanni Bussotti, Ionas Erb, and Cedric Notredame. Multiple sequence alignment modeling: methods and applications. *Briefings in Bioinformatics*, 17(6):1009–1023, November 2015.

10 Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. On the complexity of string matching for graphs. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 55:1–55:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

11 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`.

12 Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997.

13 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for bwt-based data structures. *Theor. Comput. Sci.*, 698:67–78, 2017.

14 Travis Gagie and Gonzalo Navarro. Compressed indexes for repetitive textual datasets. In Sherif Sakr and Albert Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer, 2019.

15 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.

16 Daniel Gibney. An efficient elastic-degenerate text index? not likely. In *International Symposium on String Processing and Information Retrieval*, pages 76–88. Springer, 2020.

17 Daniel Gibney and Sharma V. Thankachan. On the hardness and inapproximability of recognizing wheeler graphs. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 51:1–51:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

18 Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis. Efficient pattern matching in elastic-degenerate texts. In Frank Drewes, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings*, volume 10168 of *Lecture Notes in Computer Science*, pages 131–142, 2017. `doi:10.1007/978-3-319-53733-7_9`.

19 Costas S. Iliopoulos and Jakub Radoszewski. Truly subquadratic-time extension queries and periodicity detection in strings with uncertainties. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 8:1–8:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

20 Russell Impagliazzo and Ramamohan Paturi. On the Complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.

21 David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, April 1978. `doi:10.1145/322063.322075`.

22 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`.

23 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`.

24 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

25 U. Manber and S. Wu. Approximate string matching with arbitrary costs for text and hypertext. In *IAPR Workshop on Structural and Syntactic Pattern Recognition, Bern, Switzerland*, pages 22–33, 1992.

26 Tobias Marschall, Manja Marz, Thomas Abeel, Louis Dijkstra, Bas E Dutilh, Ali Ghaffaari, Paul Kersey, Wigard Kloosterman, Veli Mäkinen, Adam Novak, et al. Computational pan-genomics: status, promises and challenges. *BioRxiv*, page 043430, 2016.

27 Joong Na, Hyunjoon Kim, Seunghwan Min, Heejin Park, Thierry Lecroq, Martine Leonard, Laurent Mouchard, and Kunsoo Park. FM-index of alignment with gaps. *Theoretical Computer Science*, 710, June 2016. `doi:10.1016/j.tcs.2017.02.020`.

28 Joong Chae Na, Hyunjoon Kim, Heejin Park, Thierry Lecroq, Martine Léonard, Laurent Mouchard, and Kunsoo Park. FM-index of alignment: A compressed index for similar strings. *Theoretical Computer Science*, 638:159–170, 2016. Pattern Matching, Text Data Structures and Compression. `doi:10.1016/j.tcs.2015.08.008`.

29 Joong Chae Na, Heejin Park, Maxime Crochemore, Jan Holub, Costas S. Iliopoulos, Laurent Mouchard, and Kunsoo Park. Suffix tree of alignment: An efficient index for similar data. In Thierry Lecroq and Laurent Mouchard, editors, *Combinatorial Algorithms - 24th International Workshop, IWOCA 2013, Rouen, France, July 10-12, 2013, Revised Selected Papers*, volume 8288 of *Lecture Notes in Computer Science*, pages 337–348. Springer, 2013.

30 Joong Chae Na, Heejin Park, Sunho Lee, Minsung Hong, Thierry Lecroq, Laurent Mouchard, and Kunsoo Park. Suffix array of alignment: A practical index for similar data. In Oren Kurland, Moshe Lewenstein, and Ely Porat, editors, *String Processing and Information Retrieval - 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7-9, 2013, Proceedings*, volume 8214 of *Lecture Notes in Computer Science*, pages 243–254. Springer, 2013.

31 Tuukka Norri, Bastien Cazaux, Dmitry Kosolobov, and Veli Mäkinen. Linear time minimum segmentation enables scalable founder reconstruction. *Algorithms Mol. Biol.*, 14(1):12:1–12:15, 2019.

32 Mikko Rautiainen and Tobias Marschall. Aligning sequences to general graphs in $O(V + mE)$ time. *bioRxiv*, pages 216–127, 2017.

33 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/s00224-006-1198-x.

34 Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):375–388, 2014.

35 Chris Thachuk. Indexing hypertext. *Journal of Discrete Algorithms*, 18:113–122, 2013. Selected papers from the 18th International Symposium on String Processing and Information Retrieval (SPIRE 2011).

36 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005. doi:10.1016/j.tcs.2005.09.023.