Algorithms and Complexity on Indexing Founder Graphs

Massimo Equi[1] · Tuukka Norri[1] · Jarno Alanko[1,2] · Bastien Cazaux[3] · Alexandru I. Tomescu[1] · Veli Mäkinen[1]

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: For example, *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* (MSAs). *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, repeat-free and semi-repeat-free graphs, that are easy to index. We give a linear time algorithm to construct a repeat-free (non-elastic) founder graph from a gapless MSA, and (parameterized) near-linear time algorithms to construct a semi-repeat-free (repeat-free, respectively) elastic founder graph from general MSA. Finally, we show that repeat-free founder graphs admit a reduction to Wheeler graphs in polynomial time.

This is the extended full version of conference papers in WABI 2020 [1] and ISAAC 2021 [2]. An earlier preprint has also been reviewed and presented at RECOMB-seq 2021 highlights track.

✉ Veli Mäkinen
veli.makinen@helsinki.fi

Extended author information available on the last page of the article

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) [3], the central role of MSA as a model of biological evolution has resulted into numerous heuristics to solve this problem in practice [4]. 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 [5–11]. These approaches reducing an MSA to plain text reach algorithms with linear time complexities. However, the performance of these algorithms rely on the fact that a match is always found within an individual row of the MSA.

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. To solve this version of the problem, different approaches have to be used, and a possible alternative is a graph representation of the MSA. 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). Refinements of this approach are mostly used in bioinformatics [12], where recombination is a desired feature, and it is realized by the fact that the resulting graph encodes a super-set of the strings of the original MSA.

Aligning a sequence against a graph is not a trivial task. Only quadratic solutions are known [13–15], and this was recently proved to be a conditional lower bound for the problem [16]. 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 [17]. On the other hand, worst-case linear-time queries are possible, but this has the potential to make the index grow exponentially [18]. 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 [19].

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* (EDSs) [20–24], 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

Fig. 1b, graph in the center). The advantage of this structure is that it is possible to perform expected-case subquadratic time queries [21]. However, EDSs are still hard to index [25], 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 EDSs 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 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 EDSs are a special case of EFGs, hence the hardness results for the former carry to the latter. Specifically, a previous work [26] showed that, under the *Orthogonal Vectors Hypothesis* (OVH), no index for EDSs 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 [26] naturally applies only to indexing EDSs, 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 [19] 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 (Fig. 1c) forces each string in each block to occur only once in the entire graph, and the semi-repeat-free property (Fig. 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 EDSs 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 height, and minimum maximum block length. In Fig. 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 [27, 28], 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.
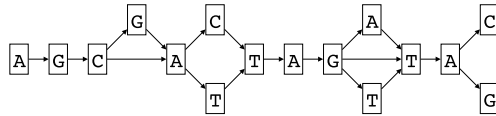
(a) A column-by-column segmentation of an MSA on the left, leading to the variation graph on the right.

(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).
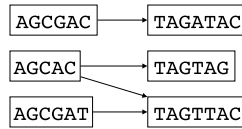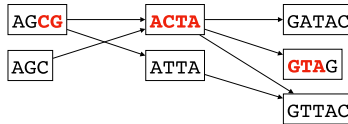

(c) A segmentation of the MSA that leads to a repeat-free EFG (i.e. no node label has another occurrence on some path of the EFG).

(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 $Q = \texttt{CGACTAGTA}$ in EFG is depicted in red. As can be seen, such query does not have an occurrence in a single row of the MSA.

Fig. 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

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 the 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. Furthermore, we give an (optimal) $O(mn)$ time solution for the special case of MSA without gap symbols. The algorithm for the special case uses a monotonicity property not holding with gaps. With general MSAs 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 [29], 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 [26], and thus, to use the efficient algorithmic machinery around Wheeler graphs [30] 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.

The paper is structured as follows. At the high level, we first focus on gapless MSAs, and then we extend the results to the general case. In more detail, Section 2 defines the founder graph concepts and explores some basic techniques. Section 3 gives the first indexing results using just classical data structures as a warm up. Section 4 covers linear time construction of repeat-free (non-elastic) founder graphs from gapless MSAs. Section 5 improves over the basic indexing results using succinct data structures. Section 6 gives the proof of conditional indexing hardness when moving from the gapless case to the general case of EFGs. Indexing results are generalized to (semi-)repeat-free EFGs in Sect. 7. Construction results are generalized to (semi-)repeat-free EFGs in Sect. 8. Connection to Wheeler graphs is considered in Sect. 9. Implementation is discussed in Sect. 10. Finally, future directions are discussed in Sect. 11.

2 Definitions and Basic Tools

2.1 Strings

We denote integer intervals by $[i..j]$. Let $\Sigma = \{1, \ldots, \sigma\}$ 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.

2.2 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* MSA$[1..m, 1..n]$ is a matrix with m strings drawn from $\Sigma \cup \{\text{-}\}$, each of length n, as its rows. Here - $\notin \Sigma$ is the *gap* symbol. For a

string $X \in (\Sigma \cup \{-\})^*$, 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 $S^k = \{\mathsf{spell}(\mathsf{MSA}[i, x_k..y_k]) \mid 1 \le i \le m\}$ for $1 \le k \le b$; 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 *height* of block S^k is $H(S^k) = |S^k|$. 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 $1 \le i \le b - 1$; and
3. if $v, w \in V^i$ then $|\ell(v)| = |\ell(w)|$ for each $1 \le i \le b$ 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. 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 EDSs [21]:

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 $1 \le k \le b$ we have $S^k = \{\mathsf{spell}(\mathsf{MSA}[i, x_k..y_k]) \mid 1 \le i \le m\} = \{\ell(v) : v \in V^k\}$. 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.

As we later learn, some further properties on founder graphs are needed for supporting fast queries:

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.

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.

2.3 Basic Tools

A *trie* [31] of a set of strings is a rooted directed tree with outgoing edges of each node labeled by distinct characters such that there is a root to leaf path spelling each string in the set; the shared part of the root to leaf paths to two different leaves spell the common prefix of the corresponding strings. Such a trie can be computed in $O(N \log \sigma)$ time, where N is the total length of the strings, and it supports string queries that require $O(q \log \sigma)$ time, where q is the length of the queried string.

In a *compact trie* the maximal non-branching paths of a trie become edges labeled with the concatenation of labels on the path. *Suffix tree* is the compact trie of all suffixes of string $T\mathbf{0}$. In this case, the edge labels are substrings of T and can be represented in constant space as an interval. Such tree takes linear space and can be constructed in linear time [32] so that when reading the leaves from left to right, the suffixes are listed in their lexicographic order. The leaves hence form the *suffix array* [33] of string T, which is an array $\mathtt{SA}[1..n + 1]$ such that $\mathtt{SA}[i] = j$ if $T'[j..n + 1]$ is the i-th smallest suffix of string $T' = T\mathbf{0}$, where $T \in \{1, 2, \ldots, \sigma\}^n$. A *generalized suffix tree or array* is one built on a set of strings. In this case, string T above is the concatenation of the strings with symbol $\mathbf{0}$ between each.

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 of the subtree rooted at v are then all the suffixes sharing the common prefix Q. Let the left- and right-most leaves in that subtree be the c-th and d-th smallest suffixes of $T\mathbf{0}$. Then $Q = T[i..i + |Q| - 1]$ for $i \in \mathtt{SA}[c..d]$ and does not occur elsewhere. We use heavily this connection of suffix tree node v and suffix array interval $\mathtt{SA}[c..d]$. Moreover, there are succinct data structures to do this mapping in both directions in constant time [34].

Let aX and X be paths spelled from the root of a suffix tree to nodes v and w, respectively. Then one can store a *suffix link* from v to w. *Implicit suffix links* for implicit nodes are defined analogously, but they are not stored explicitly. In many algorithms, one can simulate implicit suffix links through explicit suffix links, as the work amortizes to a constant per step.

The *Aho-Corasick automaton* [35] is a trie of a set of strings with additional pointers (fail-links). While scanning a query string, these pointers (and some shortcut links on them) allow to identify all the positions in the query at which a match for any of the

strings occurs. Construction of the automaton takes the same time as that of the trie. Queries take $O(q \log \sigma + \text{occ})$ time, where occ is the number of matches.

The *Burrows-Wheeler transform* BWT[$1..n + 1$] [36] of string T is such that BWT[i] = T'[SA[i] − 1], where $T' = T\mathbf{0}$ and T'[−1] is regarded as T'[$n + 1$] = $\mathbf{0}$.

A *bidirectional BWT index* [37, 38] is a succinct index structure based on some auxiliary data structures on BWT. Given a string $T \in \Sigma^n$, with $\sigma \leq n$, such index occupying $O(n \log \sigma)$ bits of space can be built in randomized $O(n)$ time and it supports finding in $O(q)$ time if a query string $Q[1..q]$ appears as substring of T [38]. Moreover, the query returns an interval pair ([$i..j$],[$i'..j'$]) such that suffixes of T starting at positions SA[i], SA[$i + 1$], ..., SA[j] share a common prefix matching the query. Interval [$i'..j'$] is the corresponding interval in the suffix array of the reverse of T. Let ([$i..j$],[$i'..j'$]) be the interval pair corresponding to query substring $Q[l..r]$. A *bidirectional backward step* updates the interval pair ([$i..j$],[$i'..j'$]) to the corresponding interval pair when the query substring $Q[l..r]$ is extended to the left into $Q[l − 1..r]$ (*left extension*) or to the right into $Q[l..r + 1]$ (*right extension*). Such step takes constant time [38].

A *fully-functional bidirectional BWT index* [39] expands the steps to allow contracting symbols from the left or from the right. That is, substring $Q[l..r]$ can be modified into $Q[l + 1..r]$ (*left contraction*) or to $Q[l..r − 1]$ (*right contraction*) and the the corresponding interval pair can be updated in constant time.

Among the auxiliary structures used in BWT-based indexes, we explicitly use the *rank* and *select* structures: String $B[1..n]$ from binary alphabet is called a *bitvector*. Operation rank(B, i) returns the number of 1s in $B[1..i]$. Operation select(B, j) returns the index i containing the j-th 1 in B. Both queries can be answered in constant time using an index requiring $o(n)$ bits in addition to the bitvector itself [40].

We summarize a result that we use later.

Lemma 1 ([39]) *Given a text T of length n from an alphabet $\{1, 2, \ldots, \sigma\}$, it is possible to construct in $O(n)$ randomized time and $O(n \log \sigma)$ bits of space a bidirectional BWT index that supports left extensions and right contractions in $O(1)$ time.*

2.3.1 Deterministic Index Construction for Integer Alphabets

We now describe how to replace the randomized linear time construction of the bidirectional BWT index with a deterministic one, so that left extensions and right contractions are supported in $O(\log \sigma)$ time. We need only a single-directional subset of the index of Belazzougui and Cunial [39], consisting of an index on the BWT, augmented with balanced parentheses representations of the topologies of the suffix tree of T and of the suffix link tree of the reverse of T, such that the nodes corresponding to maximal repeats in both topologies are marked [39].

Belazzougui et al. showed how to construct the BWT of a string $O(n)$ deterministic time and $O(n \log \sigma)$ bits of space [38]. Their algorithm can be used to construct both the BWT of T and the BWT of the reverse of T in $O(n)$ time. We can build the bidirectional BWT index [37, 41] by indexing both BWTs as *wavelet trees* [42]. The bidirectional index can then be used to construct both of the required tree topologies in $O(n \log \sigma)$ time using bidirectional extension operations and the counter-based

topology construction method of Belazzougui et al. [38]. See the supplement of a paper on variable order Markov models by Cunial et al. [43] for more details on the construction of the tree topologies. The topologies are then indexed for various navigational operations required by the contraction operation described in [39].

This index enables left extensions in $O(\log \sigma)$ time using the BWT of T, and right contractions in constant time using the succinct tree topologies as shown by Belazzougui and Cunial [39]. We summarize the result in the lemma below.

Lemma 2 *Given a text T of length n from an alphabet $\{1, 2, \ldots, \sigma\}$, it is possible to construct in $O(n \log \sigma)$ deterministic time and $O(n \log \sigma)$ bits of space a bidirectional BWT index that supports left extensions in $O(\log \sigma)$ time and and right contractions in $O(1)$ time.*

We note that it may be possible to improve the time of left extensions to $O(\log \log \sigma)$ by replacing monotone minimum perfect hash functions by slower yet deterministic linear time constructable data structures in all constructions leading to Theorem 6.7 of Belazzougui et al. [38].

3 Indexable Repeat-free Founder Graphs

We now consider non-elastic founder graphs induced from gapless MSAs, and later turn back to the general case. We show that there exists a family of founder graphs that admit a polynomial time constructable index structure supporting fast string matching. First, a trivial observation: the input multiple alignment is a founder graph for the segmentation consisting of only one segment. Such founder graph (set of sequences) can be indexed in linear time to support linear time string matching [38]. Now, the question is, are there other segmentations that allow the resulting founder graph to be indexed in polynomial time? We show that this is the case.

Proposition 1 *Repeat-free founder graphs can be indexed in polynomial time to support polynomial time string queries.*

To prove the proposition, we construct such an index and show how queries can be answered efficiently. Our first solution uses just classical data structures, and works as a warm up: Later we improve this solution using succinct data structures, and while doing so we exploit the connections to the derivations in this section.

Let $P(v)$ be the set of all paths starting from node v and ending in a sink node, where a sink is a node with no out-going edges. Let $P(v, i)$ be the set of *suffix path labels* $\{\ell(L)[i..] \mid L \in P(v)\}$ for $1 \leq i \leq |\ell(v)|$. Consider sorting $\mathcal{P} = \cup_{v \in V, 1 \leq i \leq |\ell(v)|} P(v, i)$ in lexicographic order. Then one can binary search any query string Q in $\mathcal{P}$ to find out if it occurs in $G(S)$ or not. The problem with this approach is that $\mathcal{P}$ is of exponential size.

However, if we know that $G(S)$ is repeat-free, we know that the lexicographic order of $\ell(L)[i..]$, $L \in P(v)$ in $\mathcal{P}$, is fully determined by the prefix $\ell(v)[i..|\ell(v)|]\ell(w)$ of $\ell(L)[i..]$, where w is the node following v on the path L, except against other suffix path labels starting with $\ell(v)[i..|\ell(v)|]\ell(w)$. Let $P'(v, i)$ denote the set of suffix path

labels cut in this manner. Now the corresponding set $\mathcal{P}' = \cup_{v \in V, 1 \leq i \leq |\ell(v)|} P'(v, i)$ is no longer of exponential size. Consider again binary searching a string Q in sorted $\mathcal{P}'$. If Q occurs in $\mathcal{P}'$ then it occurs in $G(S)$. If not, Q has to have some $\ell(v)$ for $v \in V$ as its substring in order to occur in $G(S)$.

To figure out if Q contains $\ell(v)$ for some $v \in V$ as its substring, we build an Aho-Corasick automaton [35] for $\{\ell(v) \mid v \in V\}$. Scanning this automaton takes $O(|Q| \log \sigma)$ time and returns such $v \in V$ if it exists.

To verify such a potential match, we need several tries [31]. For each $v \in V$, we build tries $\mathcal{R}(v)$ and $\mathcal{F}(v)$ on the sets $\{\ell(u)^R \mid (u, v) \in E\}$ and $\{\ell(w) \mid (v, w) \in E\}$, respectively, where X^R denotes the reverse $x_{|X|} x_{|X|-1} \cdots x_1$ of string $X = x_1 x_2 \cdots x_{|X|}$.

Assume now we have located (using the Aho-Corasick automaton) $v \in V$ with $\ell(v)$ such that $\ell(v) = Q[i..j]$, where v is at the k-th block of $G(S)$. We continue searching $Q[1..i-1]$ from right to left in trie $\mathcal{R}(v)$. If we reach a leaf after scanning $Q[i'..i-1]$, we continue the search with $Q[1..i'-1]$ on trie $\mathcal{R}(v')$, where $v' \in V$ is the node at block $k-1$ of $G(S)$ corresponding to the leaf we reached in the trie. If the search succeeds after reading $Q[1]$ we have found a path in $G(S)$ spelling $Q[1..j]$. We repeat the analogous procedure with $Q[j..m]$ starting from trie $\mathcal{F}(v)$. That is, we can verify a candidate occurrence of Q in $G(S)$ in $O(|Q| \log \sigma)$ time, as the search in the tries takes $O(\log \sigma)$ time per step.

We are now ready to specify a theorem that reformulates Proposition 1 in detailed form.

Theorem 1 *Let $G = (V, E)$ be a repeat-free founder graph with blocks $V^1, V^2, \ldots, V^b$ such that $V = V^1 \cup V^2 \cup \cdots \cup V^b$. We can preprocess an index structure for G in $O((N + L|E|) \log \sigma)$ time, where $\{1, \ldots, \sigma\}$ is the alphabet for node labels, $L = \max_{v \in V} |\ell(v)|$, $N = \sum_{v \in V} |\ell(v)|$, and $\sigma \leq N$. Given a query string $Q[1..q] \in \{1, \ldots, \sigma\}^q$, we can use the index structure to find out if Q occurs in G. This query takes $O(|Q| \log \sigma)$ time.*

Proof To see that the approach stated above works correctly, we need to show that it suffices to verify exactly one arbitrary candidate occurrence identified by the Aho-Corasick automaton. Indeed, for contradiction, assume our verification fails in finding Q starting from a candidate match $Q[i..j] = \ell(v)$, but there is another candidate match $Q[i'..j'] = \ell(w)$, $v \neq w$, resulting in an occurrence of Q in G. First, we can assume it holds that $[i..j]$ is not included in $[i'..j']$ and $[i'..j']$ is not included in $[i..j]$, since such nested cases would contradict the repeat-free property of G. Now, starting from the candidate match $Q[i'..j']$, we will find an occurrence of $Q[i..j]$ in G when extending to the left or to the right. This occurrence cannot be $\ell(v)$, as we assumed the verification starting from $Q[i..j] = \ell(v)$ fails. That is, we found another occurrence of $\ell(v)$ in G, which is a contradiction with the repeat-free property. Hence, the verification starting from an arbitrary candidate match is sufficient.

With preprocessing time $O(N \log \sigma)$ we can build the Aho-Corasick automaton [35]. The tries can be built in $O(\log \sigma)(\sum_{v \in V}(\sum_{(u,v) \in E} |\ell(u)| + \sum_{(v,w) \in E} |\ell(w)|)) = O(|E|L \log \sigma)$ time. The search for a candidate match and the following verification take $O(|Q| \log \sigma)$ time.

We are left with the case of short queries not spanning a complete node label. To avoid the costly binary search in sorted $\mathcal{P}'$, we instead construct the unidirectional

BWT index [38] for the concatenation $C = \prod_{i \in \{1,2,\ldots,b\}} \prod_{v \in V^i, (v,w) \in E} \ell(v)\ell(w)0$. Concatenation C is thus a string of length $O(|E|L)$ from alphabet $\{0, 1, 2, \ldots, \sigma\}$. The unidirectional BWT index for C can be constructed in $O(|C|)$ time, so that in $O(|Q|)$ time, one can find out if Q occurs in C [38]. This query equals that of binary search in $\mathcal{P}'$. □

The above result can also be applied to *degenerate strings* [44]. These are special case of elastic degenerate strings with equal length strings inside each block, and can thus be seen as fully connected block graphs.

Corollary 1 *The results of Theorem 1 hold for a* repeat-free degenerate string *a.k.a. a fully connected repeat-free founder graph.*

Observe that $N < |C| \le 2mn$, where C is the concatenation in the proof above (whose length was bounded by $O(L|E|)$), and m and n are the number of rows and number of columns, respectively, in the multiple sequence alignment from where the founder graph is induced. That is, the index construction algorithms of the above theorems can be seen to be take time almost linear in the (original) input size, namely, $O(mn \log \sigma)$ time. We study succinct variants of these indexes in Sect. 5, and also improve the construction and query times to linear as side product.

4 Construction of Repeat-free Founder Graphs

Now that we know how to index repeat-free founder graphs, we turn our attention to the construction of such graphs from a given MSA. For this purpose, we will adapt the dynamic programming segmentation algorithms for founders [27, 28].

The idea is as follows. Let S be a segmentation of MSA$[1..m, 1..n]$. We say S is *valid* (or repeat-free) if it induces a repeat-free founder graph $G(S) = (V, E)$. A segment MSA$[1..m, j'..j]$ is valid (or repeat-free) if for each $1 \le i \le m$ it holds MSA$[i, j'..j] \ne$ MSA$[i', k'..k' + (j - j')]$ for all $k' \ne j$ and $1 \le i' \le m$. As we will see, a necessary and sufficient condition for a segmentation S to be valid is that it contains only valid segments. We build such valid S considering valid segmentations of prefixes of MSA from left to right, looking at shorter valid segmentations appended with a valid new segment.

4.1 Characterization Lemma

Given a segmentation S and founder graph $G(S) = (V, E)$ induced by S, we can ensure that it is valid by checking if, for all $v \in V$, $\ell(v)$ occurs in the rows of the MSA only in the interval of the block V^i, where V^i is the block of V such that $v \in V^i$.

Lemma 3 (Characterization) *Let $\mathcal{P} = [x_1..y_1], [x_2..y_2], \ldots, [x_b..y_b]$ be the partitioning corresponding to a segmentation S inducing a block graph $G = (V, E)$. The segmentation S is valid if and only if, for all blocks $V^i \subseteq V$, $1 \le t \le m$ and $j \ne x_i$, if $v \in V^i$ then* MSA$[t, j..j + |\ell(v)| - 1] \ne \ell(v)$.

Proof To see that this is a necessary condition for the validity of S, notice that each row of the MSA can be read through G, so if $\ell(v)$ occurs elsewhere than inside the block, then these extra occurrences make S invalid. To see that this is a sufficient condition for the validity of S, we observe the following:

a) For all $(v, w) \in E$, $\ell(v)\ell(w)$ is a substring of some row of the input MSA.
b) Let $(x, u), (u, y) \in E$ be two edges such that $U = \ell(x)\ell(u)\ell(y)$ is not a substring of any row of input MSA. Then any substring of U either occurs in some row of the input MSA or it includes $\ell(u)$ as its substring.
c) Thus, any substring of a path in G either is a substring of some row of the input MSA, or it includes $\ell(u)$ of case b) as its substring.
d) Let α be a substring of a path of G that includes $\ell(u)$ as its substring. If $\ell(z) = \alpha$ for some $z \in V$, then $\ell(u)$ appears at least twice in the MSA. Substring α makes S invalid only if $\ell(u)$ does.

□

4.2 From Characterization to a Segmentation

Among the valid segmentations, we wish to select an *optimal* segmentation under some goodness criteria.

We consider three score functions for the valid segmentations, one maximizing the number of blocks, one minimizing the maximum height of a block, and one minimizing the maximum length of a block. The latter two have been studied earlier without the repeat-free constraint, and non-trivial linear time solutions have been found [27, 28], while the first score function makes sense only with this new constraint.

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

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

gives the score of an optimal scoring repeat-free segmentation $S^1, S^2, \ldots, S^b, S^{b+1}$ of MSA$[1..m, 1..j]$, where $\bigoplus$ is an operator depending on the scoring scheme and $w(x, j', j)$ is a function of the score x of the segmentation of $S^1, S^2, \ldots, S^b$ and of the last block S^{b+1} corresponding to MSA$[1..m, j'+1..j]$. To fix this recurrence so that $s(n)$ equals the maximum number of blocks over valid segmentations of MSA$[1..m, 1..n]$, set $\bigoplus = \max$ and $w(x, j', j) = x + 1$. For initialization, set $s(0) = 0$. 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 heights of blocks over valid segmentations of MSA$[1..m, 1..n]$, set $\bigoplus = \min$ and $w(x, j', j) = \max(x, |\{\text{MSA}[i, j'+1..j] \mid 1 \leq i \leq m\}|)$. For initialization, set $s(0) = 0$. 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 maxi-

mum length of blocks over valid segmentations of MSA[$1..m, 1..n$], set $\bigoplus$ = min and $w(x, j', j) = \max(x, j - j')$. For initialization, set $s(0) = 0$. Moreover, when there is no valid segmentation for some j, set $s(j) = \infty$.

To derive efficient dynamic programming recurrences for these scoring functions, we separate the computation into the preprocessing phase and into the main computation. In the preprocessing phase, we compute values $v(j)$ and $f(j)$, $1 \leq j \leq n$, defined as follows. Value $v(j)$ is the largest integer such that segment MSA[$1..m, v(j)+1..j$] is valid. Value $f(j)$ is the smallest integer such that segment MSA[$1..m, j+1..f(j)$] is valid. Note that $v(j)$ may not be defined for small j and $f(j)$ may not be defined for large j (short blocks may not be repeat-free).

Assuming values $v(j)$ have been preprocessed, we can simplify recurrence (1) into

$$s(j) = \bigoplus_{j':0 \leq j' \leq v(j)} w(s(j'), j', j), \tag{2}$$

by observing that left-extensions of valid segments are also valid. We use this equation later for deriving a linear time solution for minimizing the maximum length of a block score.

With values $f(j)$ we can use an analogous observation that right-extensions of valid segments are also valid. This observation directly yields forward-propagation dynamic programming solutions for maximizing the number of blocks score and for minimizing the maximum length of a block score. These are given in Algorithms 1 and 2. We leave it for future work to derive similar result for minimizing the maximum height of a block score.

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

Require: Right-extensions $(j, f(j))$ sorted from smallest to largest order by second component: $(j_1, f(j_1)), (j_2, f(j_2)), \ldots, (j_{n-J}, f(j_{n-J}))$, where $f(j_1) \leq f(j_2) \leq \cdots \leq f(j_{n-J})$ and J is such that $f(j_{n-J+1}), f(j_{n-J+2}), \ldots, f(j_n)$ are not defined.
Ensure: Score of an optimal repeat-free segmentation maximizing the number of blocks.
1: $x \leftarrow 1$
2: maxblocks$(0) \leftarrow 0$
3: maxblocks$(j) \leftarrow -\infty$ for all $0 < j \leq n$
4: maxscore $\leftarrow -\infty$
5: **for** $j \leftarrow 1$ to n **do**
6:    **while** $j = f(j_x)$ **do**
7:       maxscore $\leftarrow \max($maxscore, maxblocks$(j_x))$
8:       $x \leftarrow x + 1$
9:    **end while**
10:   maxblocks$(j) \leftarrow$ maxscore $+ 1$
11: **end for**
12: **return** maxblocks(n)

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

Algorithm 2 An $O(n \log \log n)$ 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 $\text{Upgrade}(k, v)$ sets key k to value v if the previous value is larger. Operation $\text{RangeMin}(a, b)$ 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, but here the operations required are such that $O(\log \log n)$ time can be achieved with more advanced data structures.

Require: Right-extensions $(j, f(j))$ sorted from smallest to largest order by second component: $(j_1, f(j_1)), (j_2, f(j_2)), \ldots, (j_{n-J}, f(j_{n-J}))$, where $f(j_1) \leq f(j_2) \leq \cdots \leq f(j_{n-J})$ and J is such that $f(j_{n-J+1}), f(j_{n-J+2}), \ldots, f(j_n)$ are not defined.
Ensure: Score of an optimal semi-repeat-free segmentation minimizing the maximum segment length.
1: Initialize data structures $\mathcal{T}$ and $\mathcal{I}$ with keys $0, 1, 2, \ldots, 2n$, with all keys associated with values ∞
2: $x \leftarrow 1$
3: $\text{minmaxlength}(0) \leftarrow 0$
4: **for** $j \leftarrow 1$ to n **do**
5:     **while** $j = f(j_x)$ **do**
6:         $\mathcal{T}.\text{Upgrade}(j_x + \text{minmaxlength}(j_x), \text{minmaxlength}(j_x))$
7:         $\mathcal{I}.\text{Upgrade}(j_x + \text{minmaxlength}(j_x), -j_x)$
8:         $x \leftarrow x + 1$
9:     **end while**
10:    $\text{minmaxlength}(j) \leftarrow \min(\mathcal{T}.\text{RangeMin}(j + 1, \infty), \mathcal{I}.\text{RangeMin}(-\infty, j) + j)$
11: **end for**
12: **return** $\text{minmaxlength}(n)$

---

and $O(n \log \log n)$ time, respectively. The produced segmentations induce repeat-free founder graphs from a gapless MSA.

Proof The $O(mn)$ preprocessing algorithm is provided in Theorem 3. Let us then consider the running time of the main algorithms. In both algorithms, the sorted input can be produced in $O(n)$ time by counting sort from the output of the preprocessing algorithm. The first algorithm takes clearly linear time. The second algorithm takes clearly $O(n \log n)$ time when using standard balanced search trees, but this can be improved: Since the queries are semi-open intervals with keys in range $[1 \ldots 2n]$, these balanced search trees can be replaced by van Emde Boas trees to obtain $O(n \log \log n)$ time computation of all values [45].

Correctness of Algorithms 1 and 2 follow 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. We formalize this argument for Algorithm 2, as the proof for the other is analogous and easier. Assume by induction that $\text{minmaxlength}(j')$ is the score $\max_{i:1 \leq i \leq b} L(S^i)$ of an optimal semi-repeat-free segmentation $S^1, S^2, \ldots, S^b$ of MSA$[1..m, 1..j']$, for $j' < j$. Each $\text{minmaxlength}(j')$ is added to the data structures when the corresponding segmentation can be considered to be appended with segment S^{b+1} corresponding to MSA$[1..m, j'+1..j]$, for $j \geq f(j')$, so that the result is a semi-repeat-free segmentation. The minimum values from the data structures equal the definition of segmentation score $\max_{i:1 \leq i \leq b+1} L(S^i)$:

To see this, we have two cases to consider: a) for j' such that $\text{minmaxlength}(j') > j - j'$ the score of the segmentation ending at j' extended with $[j'+1..j]$ is

minmaxlength(j'), and b) for j' such that minmaxlength(j') $\leq j - j'$ the score of the segmentation ending at j' extended with $[j'+1..j]$ is $j - j'$. The query intervals guarantee that the minima is returned, with the latter adjusted by $+j$ so that it gives $j - j'$ for minimum $-j$ in tree tree $\mathcal{I}$, corresponding to the cases a) and b). Initialization guarantees that score of the first segment is correctly computed. Traceback from minmaxlength(n) gives an optimal semi-repeat-free segmentation.   □

4.3 Preprocessing

We can do the preprocessing for values $v(j)$ and $f(j)$ in $O(mn)$ time. The idea is to build a BWT index on the MSA rows, and then search all rows backward from right to left in parallel (with everything reversed for the latter values). Once we reach a column j' where all suffixes have altogether exactly m occurrences (their union of BWT intervals is of size m), then MSA$[1..m, j'..n]$ is a valid segment for largest j' with this property. Then we can drop the last column (do right-contract on all rows) and continue left-extensions until finding the largest j' such that MSA$[1..m, j'..n-1]$ is a valid segment. Continuing this way, we can find for each column j the value $v(j) = j' - 1$. The bottleneck of the approach is the computation of the size of the union of intervals, but we can avoid a trivial computation by exploiting the repeat-free property and the order in which these intervals are computed.

Theorem 3 *Given a multiple sequence alignment* MSA$[1..m, 1..n]$ *with each* MSA$[i, j]$ $\in [1..\sigma]$*, values* $v(j)$ *and* $f(j)$ *for each* $1 \leq j \leq n$ *can be computed in randomized* $O(mn)$ *time or deterministic* $O(mn \log \sigma)$ *time. Here value* $v(j)$ *is the largest integer such that segment* MSA$[1..m, v(j) + 1..j]$ *is valid, and value* $f(j)$ *is the smallest integer such that segment* MSA$[1..m, j + 1..f(j)]$ *is valid.*

Proof We consider values $v(j)$ as the other case is symmetric. Let us build the bidirectional BWT index [38] of MSA rows concatenated into one long string with some separator symbols added between rows. We will run several phases in synchronization over this BWT index, but we explain them first as if they would be run independently.

Phase 0 searches in parallel all rows from right to left advancing each by one position at a time. Let k be the number of parallel of steps done so far. We can maintain a bitvector M that at the k-th step stores $M[i] = 1$ iff $BWT[i]$ is the k-th last symbol of some row.

Phase 1 uses the *variable length sliding window* approach of Belazzougui and Cunial [39] to compute values $v(j)$. Let the first row of MSA be $T[1..n]$. Search $T[1..n]$ backwards in the fully-functional bidirectional BWT index [39]. Stop the search at $T[j'+1..n]$ such that the corresponding BWT interval $[i'..i]$ contains only suffixes originating from column $j'+1$ of the MSA, that is, spelling MSA$[a, j'+1..n]$ in the concatenation, for some rows a. Set $v^b(n) = j'$ for row $b = 1$. Contract $T[n]$ from the search string and modify BWT interval accordingly [39]. Continue the search (decreasing j' by one each step) to find $T[j'+1..n-1]$ s.t. again the corresponding BWT interval $[i'..i]$ contains only suffixes originating from column $j'+1$. Update $v^b(n-1) = j'$ for row $b = 1$. Continue like this throughout T. Repeat the process

for all remaining rows $b \in [2..m]$, to compute $v^2(j), v^3(j), \ldots, v^m(j)$ for all j. Set $v(j) = \min_i v^i(j)$ for all j.

Let us call the instances of Phase 1 run on the rest of the rows as Phases 2, 3, ..., m.

Let the current MSA interval in Phases 1 to m be $[j' + 1..j]$. The problematic part is checking if the corresponding *active* BWT intervals $[i'_a..i_a]$ for Phases $a \in \{1, 2, \ldots, m\}$ contain only suffixes originating from column $j' + 1$. To solve this, we run Phase 0 as well as Phases 1 to m in synchronization so that we are at the k-th step in Phase 0 when we are processing interval $[j' + 1..j]$ in the rest of the Phases, for $k = n - j'$. In addition, we maintain bitvectors B and E such that $B[i'_a] = 1$ and $E[i_a] = 1$ for $a \in \{1, 2, \ldots, m\}$. For each $M[i]$ that we set to 1 at step k with $B[i] = 0$ and $E[i] = 0$, we check if $M[i - 1] = 1$ and $M[i + 1] = 1$. If and only if this check fails on any i, there is a suffix starting outside column $j' + 1$. This follows from the fact that each suffix starting at column $j' + 1$ must be contained in exactly one of the distinct intervals of the set $I = \{[i'_a..i_a]\}_{a \in \{1, 2 \ldots m\}}$. This is because I cannot contain nested interval pairs as all strings in segment $[j' + 1..j]$ of the MSA are of equal length, and thus their BWT intervals cannot overlap except if the intervals are exactly the same.

Finally, the running time of the algorithm is $O(mn)$ or $O(mn \log \sigma)$, using Lemma 1 or Lemma 2, respectively, and using the fact that the bitvectors are manipulated locally only on indexes that are maintained as variables during the execution. □

4.4 Faster Algorithm for Minimizing the Maximum Block Length

Recall Eq. (2). Let us consider the score $w(x, j', j) = \max(s(j'), j - j')$ with $\bigoplus = \min$, that is, minimizing the maximum block length over valid segmentations. Algorithm 2 solved this problem in near-linear time, but now we improve this to linear using values $v(j)$ instead of $f(j)$. The basic observation is that $v(J) \leq v(J + 1) \leq \cdots \leq v(n)$, for some $J > 0$, and hence the range where the minimum is taken grows as j grows.

Cazaux et al. [28] considered a similar recurrence and gave a linear time solution for it. In what follows we modify that technique to work with valid ranges.

For j between 1 and n, we define

$$x(j) = \max \underset{j' \in [1..v(j)]}{\arg\min} \max(j - j', s(j'))$$

Lemma 4 *For any $j \in [1..n - 1]$, we have $x(j) \leq x(j + 1)$.*

Proof By the definition of $x(.)$, for any $j \in [1..n]$, we have for $j' \in [1..x(j) - 1]$, $\max(j - j', s(j')) \geq \max(j - x(j), s(x(j)))$ and for $j' \in [x(j) + 1..v(j)]$, $\max(j - j', s(j')) > \max(j - x(j), s(x(j)))$.

We assume that there exists $j \in [1..n - 1]$, such that $x(j + 1) < x(j)$. In this case, $x(j + 1) \in [1..x(j) - 1]$ and we have

$$\max(j - x(j + 1), s(x(j + 1))) \geq \max(j - x(j), s(x(j))).$$

As $v(j + 1) \geq v(j)$, $x(j) \in [x(j + 1) + 1..v(j + 1)]$ and thus

$$\max(j + 1 - x(j + 1), s(x(j + 1))) < \max(j + 1 - x(j), s(x(j))).$$

As $x(j + 1) < x(j)$, we have $j - x(j + 1) > j - x(j)$. To simplify the proof, we take $A = j - x(j + 1)$, $B = s(x(j + 1))$, $C = j - x(j)$ and $D = s(x(j))$. Hence, we have $\max(A, B) \geq \max(C, D)$, $\max(A + 1, B) < \max(C + 1, D)$ and $A > C$. Now we are going to prove that this system admits no solution.

- Case where $A = \max(A, B)$ and $C = \max(C, D)$. As $A > C$, we have $A + 1 > C + 1$ and thus $\max(A + 1, B) > \max(C + 1, D)$ which is impossible because $\max(A + 1, B) < \max(C + 1, D)$.
- Case where $B = \max(A, B)$ and $C = \max(C, D)$. We can assume that $B > A$ (in the other case, we take $A = \max(A, B)$) and as $A > C$, we have $B > C + 1$ and thus $\max(A + 1, B) > \max(C + 1, D)$ which is impossible because $\max(A + 1, B) < \max(C + 1, D)$.
- Case where $A = \max(A, B)$ and $D = \max(C, D)$. We have $A > D$ and $A > C$, thus $\max(A + 1, B) > \max(C + 1, D)$ which is impossible because $\max(A + 1, B) < \max(C + 1, D)$.
- Case where $B = \max(A, B)$ and $D = \max(C, D)$. We have $B \geq D$ and $A > C$, thus $\max(A + 1, B) \geq \max(C + 1, D)$ which is impossible because $\max(A + 1, B) < \max(C + 1, D)$. $\qquad\square$

Lemma 5 *Given* $j^\star \in [x(j - 1) + 1..v(j)]$, *we can compute in constant time if*

$$j^\star = \max \arg\min_{j' \in [j^\star..v(j)]} \max(j - j', s(j')).$$

Proof We need just to compare $k = \max(j - j^\star, s(j^\star))$ and $s(j^\diamond)$ where $j^\diamond$ is in $\arg\min_{j' \in [j^\star + 1..v(j)]} s(j')$. If k is smaller than $s(j^\diamond)$, k is smaller than all the $s(j')$ with $j' \in [j^\star + 1..v(j)]$ and thus smaller than all $\max(j - j', s(j'))$. Hence we have $j^\star = \max \arg\min_{j' \in [j^\star..v(j)]} \max(j - j', s(j'))$.

Otherwise, $k \geq s(j^\diamond)$ and as $k \geq j - j^\star > j - j^\diamond$, $k \geq \max(j - j^\diamond, s(j^\diamond))$. In this case $j^\star \neq \max \arg\min_{j' \in [j^\star..v(j)]} \max(j - j', s(j'))$. By using the constant time semi-dynamic range maximum query by Cazaux et al. [28] on the array $s(.)$, we can obtain in constant time $j^\diamond$ and thus check the equality in constant time. $\qquad\square$

Theorem 4 *The values* $s(j)$ *of Eq. (2) with* $w((s(j'), j', j) = \max(s(j'), j - j')$ *and* $\bigoplus = \min$ *for all* $j \in [1..n]$, *can be computed in* $O(n)$ *time after a randomized* $O(mn)$ *time or deterministic* $O(mn \log \sigma)$ *time preprocessing on a gapless multiple sequence alignment. The optimal segmentation defined by Eq. (2) yields a repeat-free founder graph.*

Proof We begin by preprocessing all the values of $v(j)$ in $O(mn)$ randomized or $O(mn \log \sigma)$ deterministic time (Theorem 3). The idea is to compute all the values $s(j)$ by increasing order of j and by using the values $x(j)$. For each $j \in [1..n]$, we check all the j' from $x(j - 1)$ to $v(j)$ with the equality of Lemma 5 until one is true and thus corresponds to $x(j)$. Finally, we add $s(j) = \max(j - x(j), s(x(j)))$ to the constant time semi-dynamic range maximum query and continue with $j + 1$. $\qquad\square$

5 Compact Index for Repeat-free Founder Graphs

Recall the indexing solutions of Sect. 3 and the definitions from Sect. 2.

We now show that explicit tries and Aho-Corasick automaton can be replaced by some auxiliary data structures associated with the Burrows-Wheeler transformation of the concatenation $C = \prod_{i \in \{1,2,\ldots,b-1\}} \prod_{v \in V^i, (v,w) \in E} \ell(v)\ell(w)\mathbf{0}$.

Consider interval $\mathtt{SA}[i..k]$ in the suffix array of C corresponding to suffixes having $\ell(v)$ as prefix for some $v \in V$. From the repeat-free property it follows that this interval can be split into two sub-intervals, $\mathtt{SA}[i..j]$ and $\mathtt{SA}[j+1..k]$, such that suffixes in $\mathtt{SA}[i..j]$ start with $\ell(v)\mathbf{0}$ and each suffix in $\mathtt{SA}[j+1..k]$ start with some $\ell(v)\ell(w)$ for $(v, w) \in E$. Moreover, $\mathtt{BWT}[i..j]$ equals multiset $\{\ell(u)[|\ell(u)| - 1] \mid (u, v) \in E\}$. That is, $\mathtt{BWT}[i..j]$ (interpreted as a set) represents the children of the root of the trie $\mathcal{R}(v)$ considered in Sect. 3.

We are now ready to present the search algorithm that uses only the BWT of C and some small auxiliary data structures. We associate two bitvectors B and E to the BWT of C as follows. We set $B[i] = 1$ and $E[k] = 1$ iff $\mathtt{SA}[i..k]$ is a maximal interval with all suffixes starting with $\ell(v)$ for some $v \in V$.

Consider the backward search with query $Q[1..q]$. Let $\mathtt{SA}[j'..k']$ be the interval after matching the shortest suffix $Q[q'..q]$ such that $\mathtt{BWT}[j'] = \mathbf{0}$. Let $i = \mathtt{select}(B, \mathtt{rank}(B, j'))$ and $k = \mathtt{select}(E, \mathtt{rank}(B, j'))$. If $i \leq j'$ and $k' \leq k$, index j' lies inside an interval $\mathtt{SA}[i..k]$ where all suffixes start with $\ell(v)$ for some v. We modify the range into $\mathtt{SA}[i..k]$, and continue with the backward step on $Q[q'-1]$. We check the same condition in each step and expand the interval if the condition is met. Let us call this procedure *expanded backward search*.

We can now strictly improve Theorem 1 and Corollary 1 as follows.

Theorem 5 *Let $G = (V, E)$ be a repeat-free founder graph (or a repeat-free degenerate string) with blocks $V^1, V^2, \ldots, V^b$ such that $V = V^1 \cup V^2 \cup \cdots \cup V^b$. We can preprocess an index structure for G occupying $O(L|E|\log\sigma)$ bits in $O(L|E|)$ time, where $\{1,\ldots,\sigma\}$ is the alphabet for node labels and $L = \max_{v \in V} \ell(v)$. Given a query string $Q[1..q] \in \{1,\ldots,\sigma\}^q$, we can use expanded backward search with the index structure to find out if Q occurs in G. This query takes $O(|Q|)$ time.*

Proof As we expand the search interval in BWT, it is evident that we still find all occurrences for short patterns that span at most two nodes, like in the proof of Theorem 1. We need to show that a) the expansions do not yield spurious occurrences for such short patterns and b) the expansions yield exactly the occurrences for long patterns that we earlier found with the Aho-Corasick and tries approach. In case b), notice that after an expansion step we are indeed in an interval $\mathtt{SA}[i..k]$ where all suffixes match $\ell(v)$ and thus corresponds to a node $v \in V$. The suffix of the query processed before reaching interval $\mathtt{SA}[i..k]$ must be at least of length $|\ell(v)|$. That is, to mimic Aho-Corasick approach, we should continue with the trie $\mathcal{R}(v)$. This is identical to taking a backward step from $\mathtt{BWT}[i..k]$, and continuing therein to follow the rest of this implicit trie.

To conclude case b), we still need to show that we reach all the same nodes as when using Aho-Corasick, and that the search to other direction with $\mathcal{L}(v)$ can be avoided. These follow from case a), as we see.

In case a), before doing the first expansion, the search is identical to the original algorithm in the proof of Theorem 1. After the expansion, all matches to be found are those of case b). That is, no spurious matches are reported. Finally, no search interval can include two distinct node labels, so the search reaches the only relevant node label, where the Aho-Corasick and trie search simulation takes place. We reach all such nodes that can yield a full match for the query, as the proof of Theorem 1 shows that it is sufficient to follow an arbitrary candidate match.

As we only need a standard backward step, we can use a unidirectional BWT index constructable in deterministic $O(L|E|)$ time supporting a backward step in constant time [38]. □

6 Conditional Hardness of Indexing EFGs

We now turn our attention to general MSAs and elastic founder graphs induced from their segmentation. With non-elastic founder graphs, we have seen that the repeat-free property makes them indexable, but for now we have no proof that such property is necessary. For elastic founder graphs, we are able to derive such a conditional lower bound.

Namely, 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 [16, 25, 44, 46]. 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 set 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* [47].

Definition 5 (Orthogonal Vectors Hypothesis (OVH) [48]) 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 $O(\text{poly}(d)n^{2-\epsilon})$.

6.1 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 sub-string $Q_{i,h}$ 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 to have $Q_i = \text{b}^4 x_i[1]^4 x_i[2]^4 \ldots x_i[d]^4 \text{e}^4$. For example, vector $x_i = 101$ results into string

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

---

[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.
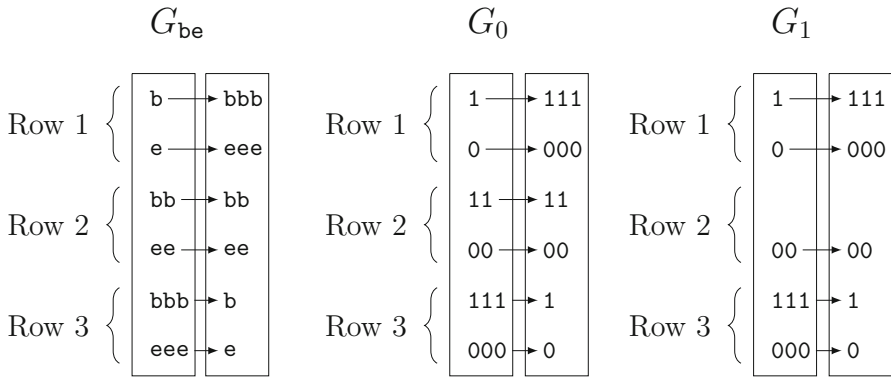
$$G_{\text{be}} \qquad\qquad G_0 \qquad\qquad G_1$$

Row 1 $\Big\{$
| b → bbb |
| e → eee |

Row 2 $\Big\{$
| bb → bb |
| ee → ee |

Row 3 $\Big\{$
| bbb → b |
| eee → e |

Row 1 $\Big\{$
| 1 → 111 |
| 0 → 000 |

Row 2 $\Big\{$
| 11 → 11 |
| 00 → 00 |

Row 3 $\Big\{$
| 111 → 1 |
| 000 → 0 |

Row 1 $\Big\{$
| 1 → 111 |
| 0 → 000 |

Row 2 $\Big\{$
| 00 → 00 |

Row 3 $\Big\{$
| 111 → 1 |
| 000 → 0 |

Fig. 2 Gadgets G_{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 Fig. 3, edges can be controlled to go within the same row, or to the row below

Full string Q is then the concatenation $Q = \text{bbbb} Q_1 Q_2 \ldots Q_n \text{eeee}$. The reason behind these specific quantities will be clear when discussing the structure of the graph.

6.2 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 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 Fig. 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_{be}, G_0 and G_1 (see Fig. 2), which allow to stack together multiple instances of strings b^4, e^4, 1^4, 0^4. The overall structure mimics the one in [16], except for the new idea from Fig. 2.

6.2.1 Detailed Structure of the Graph.

Sub-graph G_L (Fig. 3a) consists of a starting segment with a single node labeled b^4, followed by $n-1$ sub-graphs $G_L^1, \ldots, G_L^{n-1}$, 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 b^4, then we place d other segments, each one containing two nodes with labels 1^4 and 0^4. Finally, we place a segment containing two nodes with labels b^4 and e^4.
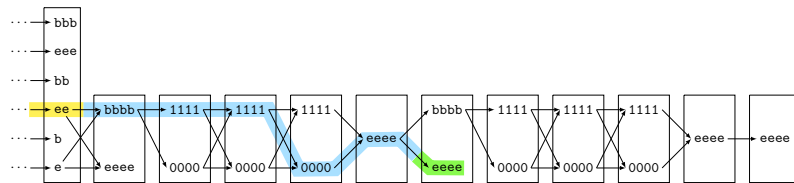
The nodes in each segment are connected to all nodes in the next segment, with the exception of the last segment of each G_L^i: in this case, the node with label 1^4 and the
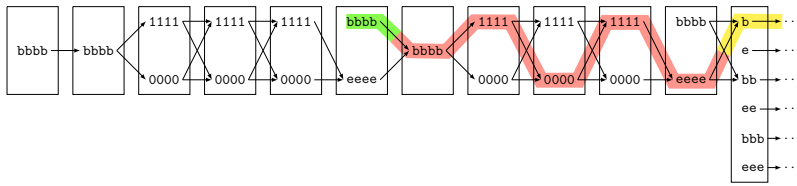
(a) Sub-graph G_L. The last segment belongs to sub-graph G_M and shows the connection.
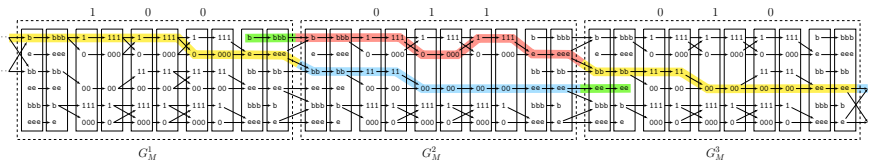
(b) Sub-graph G_M for vectors $y_1 = 100$, $y_2 = 011$ and $y_3 = 010$. The dashed rectangles highlight the single G_M^j gadgets.

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

Fig. 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

one with label 0^4 are connected only to the e^4-node of the next (and last) segment of such G_L^i.

Sub-graph G_R (Fig. 3c) is similar to sub-graph G_L, and it consists in $n-1$ parts $G_R^1, \ldots, G_R^{n-1}$, followed by a segment with a single node labeled 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 b^4 and e^4, while in the last segment we place only one node, which we label 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 e^4 has no outgoing edge.

Sub-graph G_M (Fig. 3b) implements the main logic of the reduction, and it uses three building blocks, G_{be}, G_0 and G_1, which are organized in three rows, as shown in Fig. 2.

Sub-graph G_M has n parts, $G_M^1, \ldots, G_M^n$, one for each of the vectors $y_1, \ldots, y_n$ in set Y. Each G_M^j is constructed, from left to right, as follows. First, we place a G_{be} gadget. Then, we scan vector y_j from left to right and, for each position $h \in \{1, \ldots, d\}$, we place a G_0 gadget if the h-th entry is $y_j[h] = 0$, or a G_1 if $y_j[h] = 1$. Finally, we place another G_{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 b-nodes of the left G_{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 e-nodes of the right G_{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 Fig. 3b.

To conclude the construction of G_M, we need to connect all the G_M^j gadgets together. Consider the right G_{be} of gadget G_M^j, and the left G_{be} of gadget G_M^{j+1}. The edges connecting these two gadgets are depicted in Fig. 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 b^8 can be matched only in the first and second row, while sub-pattern e^8 only in the second and third rows.

In proving the correctness of the reduction, we will use G_{M1}, G_{M2} and G_{M3} to refer to 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 nodes placed in the t-th row of each G_{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_{M2}^j to refer to the nodes belonging to both G_M^j 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_{be} gadget of G_M^1. Finally, we connect the e-nodes in the second and third row of the right G_{be} gadget of G_M^n with both the b^4-node and 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 Figs. 3a and 3c we also included the adjacent segment of G_M to show the connection.

6.3 OVH Conditional Hardness

The proof of correctness is similar to the one in [16], but with adaptations to the elastic founder graph. We prove three lemmas concerning G_{M2}, which are key for the correctness. The first lemma is a straightforward consequence of the structure of G_{M2} and the fact that it is directed.

Lemma 6 *If string Q_i has a match in G_{M2}, then the path matching Q_i is fully contained in G_{M2}^j, for some $1 \le j \le n$. Moreover, each $Q_{i,h}$ sub-string matches a path of two nodes which belong to the G_0 or G_1 gadget encoding $y_j[h]$.*

Proof The claim follows by construction, since Q_i starts with $\mathtt{b}^4\mathtt{0}^4$ or $\mathtt{b}^4\mathtt{1}^4$ (which are found only at the beginning of a G^j_{M2} gadget), and ends with $\mathtt{0}^4\mathtt{e}^4$ or $\mathtt{1}^4\mathtt{e}^4$ (which are found only at the end of a G^j_{M2}). □

Lemma 7 *String Q_i has a match in G_{M2} if and only if there exists $y_j \in Y$ such that $x_i \cdot y_j = 0$.*

Proof Recall that, by construction, the h-th G_*, $* \in \{0, 1\}$ gadget in G^j_{M2} is a G_0 gadget if and only if $y_j[h] = 0$, while it is a G_1 gadget if and only if $y_j[h] = 1$. We handle the two implications of the statement individually.

($\Rightarrow$) By Lemma 6, we can focus on the d distinct and consecutive nodes of G^j_{M2} that match Q_i. In particular we know that each sub-string $Q_{i,h}$ matches in the second row of either the h-th gadget G_0 or the h-th gadget G_1. Consider vectors $x_i \in X$ and $y_j \in Y$. If $Q_{i,h} = 1^4$ has a match in G^j_{M2} it means that its h-th gadget is a G_0, and hence $y_j[h] = 0$, implying $x_i[h] \cdot y_j[h] = 0$. If $Q_{i,h} = 0$, by construction we know that $x_i[h] = 0$, and $Q_{i,h}$ can have a match in G^j_{M2} no matter whether the h-th gadget is a G_0 or a G_1. Thus, it clearly holds that $x_i[h] \cdot y_j[h] = 0$. At this point, we can conclude that $x_i[h] \cdot y_j[h] = 0$ for every $1 \le h \le d$, thus $x_i \cdot y_j = 0$.

($\Leftarrow$) Consider vectors $x_i \in X$ and $y_j \in Y$ that are such that $x_i \cdot y_j = 0$. For $h = 1, 2, \ldots, d$, if $y_j[h] = 0$ then the h-th gadget of G^j_{M2} is a G_0 gadget, and $Q_{i,h}$ can surely match it. If $y_j[h] = 1$ it must hold that $x_i[h] = 0$, since $x_i \cdot y_j = 0$. Thus $Q_{i,h} = 0^4$, and it can have a match in the h-th gadget of G^j_{M2}, no matter if it is a G_0 or G_1 gadget. Finally, sub-strings $\mathtt{b}^4$ and $\mathtt{e}^4$ can have a match in the G_{be} gadgets at the beginning and end of G^j_{M2}, respectively. All characters of Q_i have now a matching node and the definition of the edges allows to visit all such nodes via a matching path starting in the left G_{be} gadget of G^j_{M2} and ending in the right G_{be} gadget of G^j_{M2}. □

Lemma 8 *String Q has a match in G if and only if a sub-string Q_i of Q has a match in the underlying sub-graph G_{M2} of G_M.*

Proof For the ($\Rightarrow$) implication, because of the directed $\mathtt{e}^4\mathtt{b}^4$-edges, each distinct sub-string Q_i matches a path from a distinct portion of either G_L, G_M and G_R. Moreover, each occurrence of P must begin with $\mathtt{b}^8$ and end with $\mathtt{e}^8$. String $\mathtt{b}^8$ can be matched only in G_L, in G_{M1} or in G_{M2}, hence the match must start here. On the other hand, string $\mathtt{e}^8$ is found either in G_{M2}, G_{M3} or in G_R. Observe that, by construction, once a match for pattern Q is started in G_L, in G_{M1} or in G_{M2}, the only way to successfully conclude it is either by matching $\mathtt{e}^8$ within G_{M2}, or by matching also a portion of G_{M3} and/or G_R and then $\mathtt{e}^8$. Because of the structure of the graph, in both cases a sub-string Q_i of Q must match one of the gadget G^j_{M2} that are present in G_{M2}.

The ($\Leftarrow$) implication is trivial. In fact, if Q_i has a match in one gadget G^j_{M2}, then by construction we can match $\mathtt{b}^4 Q_1 \ldots Q_{i-1}$ possibly in G_L, then possibly in G_{M1}. We can then match $Q_{i+1} \ldots Q_n \mathtt{e}^4$ possibly in G_{M3}, then possibly in G_R, and thus have a full match for Q in G. □

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

Theorem 6 *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 First, notice that the reduction that we presented for query string Q and EFG G is correct. Indeed, Lemma 8 guarantees that Q has a match in G if and only if a sub-string Q_i has a match in G_M, and this holds, by Lemma 7, if and only if $x_i \cdot y_j = 0$. Thus, string Q has a match in G if and only if there exist vectors $x_i \in X$ and $y_j \in Y$ which are orthogonal.

The reduction requires linear time and space in the size $O(nd)$ of the OV problem, and this is because of the construction of string Q and graph G. On one hand, when we define string Q, we place a constant number of characters for each entry of each vector, thus $|Q| = O(nd)$. On the other hand, sub-graphs G_L, G_{M1}, G_{M2}, G_{M3} and G_R all consist of $O(n)$ structures, each one containing $O(d)$ nodes, and a constant number of edges for each node, for an overall size of $O(nd)$.

Hence, given two sets of vectors X and Y, we can perform our reduction obtaining string Q and EFG $G = (V, E, \ell)$ in $O(nd)$ time, while observing that $|E| = O(nd)$ and $|Q| = O(nd)$. If we can 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 exists a pair of orthogonal vectors between X and Y in $O(nd \cdot (nd)^{1-\epsilon}) = O(n^{2-\epsilon}\text{poly}(d))$ time, which contradicts OVH.   □

We obtain the indexing lower bound by proving that the above reduction is a *linear independent-components* (*lic*) reduction, as defined by [19, Definition 3].

Theorem 7 *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 6), 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 [19] can be applied, proving our thesis.   □

7 Indexing EFGs

Let us now consider how to extend the indexing results to the general case of MSAs with gaps. The idea is that gaps are only used in the segmentation algorithm to define the valid ranges, and that is the only place where special attention needs to be taken; elsewhere, whenever a substring from MSA rows is read, gaps are treated as empty strings. That is, A-GC-TA- becomes AGCTA.

As we later see, segmentation becomes more difficult with gaps, and we need to consider a relaxed variant of repeat-free property for obtaining efficient algorithms. Recall that in a semi-repeat-free EFG a node label can appear as a prefix of another node label inside the same block.

7.1 Repeat-Free Case

As the reader can check, the indexing solutions in Sects. 3 and 5 work verbatim with the repeat-free elastic founder graphs; the property of having strings of equal length inside the blocks is not exploited in the algorithms.

7.2 Semi-repeat-free case

The case of semi-repeat-free elastic founder graphs is slightly more complex, and we need to combine and extend the previous solutions. The following lemma is the key property needed for the solution.

Lemma 9 *Consider a semi-repeat free* EFG *$G = (V, E)$. String $\ell(v)\ell(w)$, where $(v, e) \in E$, can only appear in G as a prefix of paths starting with v.*

Proof Assume for contradiction that $\ell(v)\ell(w)$ is a prefix of a path starting inside the label of some $v' \in V$, $v' \neq v$. Then $\ell(v)$ is a prefix of such path, and this is only possible if v' is in the same block as v and $\ell(v)$ is a proper prefix of $\ell(v')$: otherwise G would not be semi-repeat free. Then $|\ell(v)| < |\ell(v')|$ and $\ell(w)$ has an occurrence in a path starting inside the label $\ell(v')$. This is a contradiction on the fact that G is semi-repeat free. □

Consider the suffix tree of the concatenation $D = \prod_{v,w,u:(v,w)\in E,(w,u)\in E}(\ell(v)\ell(w) \ell(u))^{-1}\mathbf{0}$. For suffixes of type $\alpha\ell(w)^{-1}\ell(v)^{-1}\mathbf{0}$, where α is a (possibly empty) string, we store node identifier v in the corresponding leaf of the tree. Clearly, queries spanning less than three nodes can be located from this suffix tree. Consider a longer query $Q[1..q]$ whose suffix spans at least three nodes in the graph. We search it backwards in the suffix tree until reaching a locus after which we cannot proceed with $Q[i]$, but could continue with $\mathbf{0}$. Then we know that $Q[i + 1..q]$ matches a path starting with $\ell(v)\ell(w)$ in G. Due to Lemma 9, any leaf in the subtree rooted at the current locus in the suffix tree (which is spelling $Q[i + 1..q]^{-1}$) stores v. Since we cannot know in advance if Q is a longer query, we have stored identifiers v only when this case applies. Once we have identified v, we only need to check if we can read $Q[1..i]$ following a path to the left from v, which is exactly what we did in Sect. 3 using tries $\mathcal{R}(v)$ storing $\{\ell(u)^{-1} \mid (u, v) \in E\}$: The semi-repeat-free property guarantees that no node label can be a suffix of another node label (even inside the same block), and hence the leaves of the tries $\mathcal{R}(v)$ correspond to exactly one row each. The left-extensions are hence not branching, as the search always narrows down to one row (leaf), before continuing on the next trie (see Sect. 3).

Theorem 8 *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 $|D| = O(NH^2)$, N is the total length of the node labels, and H is the height of G. Later, one can find out in $O(|Q|)$ time if a given query string Q occurs in G. Here we assume that the alphabet of the node labels and query string is $[1..\sigma]$, where $\sigma \leq N$.*

Proof Each node label $\ell(v)$ is added to D at most $3H^2$ times, as H is an upper bound for the number of edges from and to v. The length of D is then bounded by $O(NH^2)$. Construction of suffix tree on D can be done in linear time [32]. In polynomial time, the nodes of the suffix tree and the tries can be preprocessed with perfect hash functions, such that following a downward path takes constant time per step. □

We 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, capable for existence or counting queries. Such modified scheme makes the approach functionality equivalent with wide range of indexes designed for repetitive collections [5–11] and shares the benefit of alignment-based indexes of Na et al. [6–9] in reporting the aligned matches only once, where e.g. r-index [11] needs to report all occurrences.

Using compressed suffix trees, different space-time tradeoffs can be achieved. In Sect. 9, we develop an alternative compressed indexing scheme for the repeat-free case using Wheeler graphs.

8 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. First, we observe that the algorithms in Sect. 4 do not work verbatim: Theorem 4 is based on Eq. (2), but now this recurrence is no longer valid, as left-extension of a valid segment may not be a valid segment. To see this, we need to revisit the definition of a valid segment now that MSA can contain gaps. Here we follow the notions developed by Rizzo and Mäkinen [49].

We say that segment $[x..y]$ of a general $\mathsf{MSA}[1..m, 1..n]$ is semi-repeat-free if for any $i, i' \in [1..m]$ string $\mathsf{spell}(\mathsf{MSA}[i, x..y])$ occurs in gaps-removed row $\mathsf{spell}(\mathsf{MSA}[i', 1..n])$ only at position $g(i', x)$, where $g(i', x)$ is equal to x subtracted the number of gaps in $\mathsf{MSA}[i', 1..x]$. Similarly, $[x, y]$ is repeat-free if the eventual occurrence of $\mathsf{spell}(\mathsf{MSA}[i, 1..n])$ at position $g(i', x)$ in row i' also ends at position $g(i', y)$.

It is straightforward to adapt the the proof of Lemma 3 to yield the following characterization:

Lemma 10 ([49]) *Elastic founder graph $G(S)$ is (semi-)repeat-free if and only if all segments of S are (semi-)repeat-free.*

A counterexample for the validity of left-extensions is shown in Table 1. On the other hand, Algorithms 1 and 2 use the right-extension property of a valid segment, and this holds even with general MSAs.

Observation 1 *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 $f(j) < j' \le n$.*

However, these algorithms assume we have precomputed for each j the smallest integer $f(j)$ such that $\mathsf{MSA}[1..m, j+1..f(j)]$ is a (semi-)repeat-free segment. The

Table 1 Semi-repeat-free segment and its extension (to the left) into a non-valid segment	segment of MSA	longer segment of MSA
	X	$\mathrm{A}X$
	X	$-X$
	Y	$\mathrm{A}Y$
	$Z-^{|Y|-|Z|}$	$\mathrm{A}Z-^{|Y|-|Z|}$

Here the distinct strings X, Y and Z do not appear elsewhere in MSA, except Z is a prefix of Y.

Here $|X| = |Y|$. The longer segment is non-valid, because X is not a prefix but a suffix of $\mathrm{A}X$. Reversing the definition does not help, as the same segment contains $\mathrm{A}Z$ as prefix of $\mathrm{A}Y$

earlier sliding windows preprocessing algorithm for values $f(j)$ inherently assumes the values are monotonic (as is the case with gapless MSAs), but this does not hold in the general case: Consider again Table 1. Let j be the column just before the longer segment of MSA. Then $f(j) > j + |X| + 1$ and $f(j+1) = j + 1 + |X|$.

In order to be able to use Algorithms 1 and 2 , we derive a new preprocessing algorithm for values $f(j)$ that does not assume monotonicity.

8.1 Preprocessing for the Non-Monotonic Case

As an internal part of the algorithm we need an efficient data structure to maintain a dynamic set of non-overlapping intervals. Let I be a set of integer intervals, i.e., $I = \{[a_1..b_1], [a_2..b_2], \ldots, [a_m..b_m]\}$, where $a_i, b_i \in [1..n]$ for all i. We say I is *non-overlapping* if for all pairs $[a_i..b_i], [a_j..b_j] \in I$ holds $[a_i..b_i] \cap [a_j..b_j] = \emptyset$.

Lemma 11 *There is a data structure to maintain a non-overlapping set of intervals I supporting insertions and deletions of intervals in $O(\log |I|)$ time. The data structure also supports in $O(\log |I|)$ time a query* $\mathrm{span}([a..b])$ *that returns* $|\cup \{[a_i..b_i] \in I \mid a \le a_i \le b_i \le b\}|$.

Proof Consider a balanced binary search tree with leaves corresponding to intervals of I, sorted by values a_i for $[a_i..b_i] \in I$. Leaf i stores a_i as *key* value and $b_i - a_i + 1$ as *span* value. Each internal node v stores the maximum key and sum of span values of the leaves in its subtree. Assuming the data structure has been maintained with these values, answering query $\mathrm{span}([a..b])$ can be done as follows: Locate the $O(\log |I|)$ internal nodes that form a non-overlapping cover on the keys in the range $[a..b]$ (by searching keys a and b and picking the subtrees bypassed and within the interval). Return the sum of span values stored in those nodes.

It remains to consider how the values can be maintained during insertions and deletions, and during the resulting rebalancing operations. On each such operation, there are $O(\log |I|)$ internal nodes affected on the upward path from the the leaf to the root. It is sufficient to consider one such affected node v assuming its left child ℓ and right child r have already been updated accordingly. We set $\mathrm{key}(v) = \mathrm{key}(r)$ and $\mathrm{span}(v) = \mathrm{span}(\ell) + \mathrm{span}(r)$. $\qquad\square$
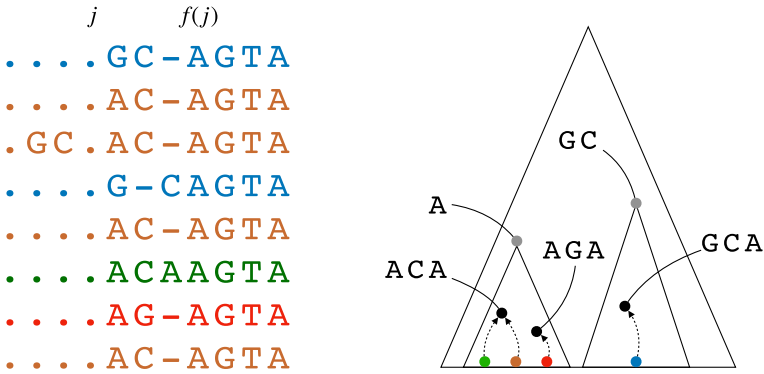
Fig. 4 Illustrating the $O(m \log m)$ time algorithm to compute value $f(j)$ for a given j. Node labels correspond to the string spelled from the root to the node. We assume ACA, AGA, and GCA only appear in the region of the MSA visualized, while GC and A appear also elsewhere

Lemma 12 *Let $f(j)$ be the smallest integer such that 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 Figure 4 illustrates the algorithm to be described in the following. Consider the generalized suffix tree of $\{$spell(MSA$[i, 1..n]) \mid 1 \leq i \leq m\}$. For each j, locate the subset W of (implicit) suffix tree nodes corresponding to $\{$spell(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 $f(j)$. We do this one row at a time, recording values $f^i(j)$ such that in the end $f(j) = \max_i f^i(j)$. We initialize $f^i(j) = n$ for all i. Suffix tree nodes corresponding to rows whose values $f^i(j)$ are final are stored in set F, initially empty. Suffix tree nodes corresponding to rows whose values $f^i(j)$ are redundant (to be detailed later) are stored in set R, initially empty.

To start this process, a) pick an (implicit) suffix tree node $v \in W$ corresponding to spell(MSA$[i, j + 1..f^i(j)]$) for some i. Let w be the parent of v. It corresponds to string spell(MSA$[i, j + 1..j']$) for some j'. Then b) consider replacing v with w in W. If the number of leaves covered by the subtrees rooted at $W \cup F \cup R$ is still m, then this replacement is safe, and we can set $f^i(j) = j'$. Safe replacements are shown as black nodes in Fig. 4, while the gray nodes are unsafe replacements. We also move from W to R all $w' \in W$ that are located in the subtree rooted at w (not including w), as these nodes are now redundant; we will consider later how to compute their values $f^i(j)$. Otherwise, instead of replacement, move v from W to F, as we have found the minimum valid range with regards to row i: Consider string spell(MSA$[i, j + 1..j' + k]$), where MSA$[i, j' + k]$ is the first non-gap symbol at row i after MSA$[i, j']$. This string is spelled by reading the path from the root to w and then reading one symbol on the edge (w, v). This string is thus the shortest string having the same occurrences as spell(MSA$[i, j + 1..f^i(j)]$), and we can safely assign as final value $f^i(j) = j' + k$. Repeat these steps a) and b) until W is empty. At that point, decreasing of the right boundary is no longer possible on any row.

However, we only have computed $f^i(j)$ for i such that there is $v \in F$ corresponding to spell(MSA[i, $j + 1..f^i(j)$]). We also need to compute values $f^{i'}(j)$ for i' such that there is $v' \in R$ corresponding to spell(MSA[i', $j + 1..f^{i'}(j)$]). Note that each $v' \in R$ was made redundant by another node, which in turn, may have been made redundant on its turn. We can store these relationships as a forest of trees. Root of each tree corresponds to some $v \in F$ and rest of the nodes are from R. Now, consider a root $v \in F$ of some of the trees corresponding to spell(MSA[i, $j + 1..f^i(j)$]) and a node $v' \in R$ of the same tree. We can assign $f^{i'}(j) = j'$ for smallest j' such that $|$spell(MSA[i', $j + 1..j'$])$| = |$spell(MSA[i, $j + 1..f^i(j)$])$|$. E.g. for row $i = 2$ in Fig. 4, we have $f^2(j) = j + 3$, as $|$spell(AC − A)$| = 3 = |$ACA$|$. Then we can set set $f(j) = \max_i f^i(j)$.

To achieve the claimed running time, we use backward searching on the unidirectional BWT index [38] on the concatenation of strings $\{$spell(MSA[i, $1..n$]) $\mid 1 \le i \le m\}$ (with special markers added between) to find all the suffix array intervals corresponding to sets $\{$spell(MSA[i, $j + 1..n$]) $\mid 1 \le i \le m\}$ for all j. This takes $O(mn)$ time. To find the largest j for which the union of suffix array intervals is of size m, we can sort the intervals at each column and compute the size of the union by a simple scanning. This takes $O(mn \log m)$ time overall.

Now we need to show that the process of reducing the right boundary for a fixed column can be done in $O(m \log m)$ time. Mapping from a suffix array interval to the (compressed) suffix tree node takes constant time [34]. Steps a) and b) are repeated at most m times at any column j: Either some row i gets completed, or at least one row becomes redundant. In both cases, size of W decreases at each step. The most time consuming part in this process is to compute the number of leaves in the union of subtrees. We can do this in $O(m \log m)$ time, by mapping the nodes back to suffix array intervals, and then computing the size of the union of intervals as above. However, we can only afford to do this at the first step of the process. For the rest of the steps we use Lemma 11: To be able to use the lemma, we need to ensure only non-overlapping intervals are stored in the data structure. Thus, at the first step we remove duplicates and intervals that are nested in another one in $O(m \log m)$ time, and store the remaining intervals to the structure of Lemma 11. While doing so, we move the suffix tree nodes corresponding to these removed intervals to the set R of redundant nodes. This is safe, as initially the union of the intervals is of size m (no extra occurrences in the intervals), and hence the steps a) and b) would anyway move the suffix tree nodes corresponding to those intervals to R at some point. Consider now step a) with w being parent of suffix tree node v. Let $[a..b]$ be the suffix array interval corresponding to w. We can query span($[a..b]$) from the data structure, and if the answer is m, we remove the intervals in the query range, and insert $[a..b]$ in their place.

It remains to consider how to find the first non-gap symbol MSA[i, $j' + k$] at row i after MSA[i, j'], and how to find the smallest j' such that $|$spell(MSA[i', $j + 1..j'$])$| = x$ given x. These can be done in constant time after $O(mn)$ time preprocessing for rank and select queries on bitvectors marking locations of the the gap symbols. □

Corollary 2 *After an $O(mn \log m)$ time preprocessing, Algorithms 1 and 2 compute the scores* maxblocks$(n) = b$ *and* minmaxlength$(n) = \max_{i:1 \le i \le b} L(S^i)$ *of optimal semi-repeat-free segmentations $S^1, S^2, \ldots, S^b$ of* MSA[$1..m$, $1..n$] *in $O(n)$ and*

$O(n \log \log n)$ *time, respectively. The produced segmentations induce semi-repeat-free* EFG*s from a general* MSA.

8.2 Repeat-Free EFGs

While Algorithms 1 and 2 would work also for the repeat-free case, it appears difficult to modify the preprocessing for the same.

Instead of separate preprocessing and dynamic programming to compute the score of an optimal segmentation, we proceed directly with the main recurrence. We consider only minimizing the maximum block length score, as for this score we can derive a non-trivial parameterized solution.

Let $e(j')$ be the score of a minimum scoring repeat-free segmentation $S^1, S^2, \ldots, S^b$ of prefix MSA$[1..m, 1..j']$, where the score is defined as $\max_{i:1 \leq i \leq b} L(S^i)$. Then

$$e(j) = \min_{\substack{j' : 0 \leq j' < j, \\ \text{MSA}[1..m, j'+1..j] \\ \text{is repeat-free segment,} \\ e(j') < j'+1}} \max(j - j', e(j')), \tag{3}$$

with $e(0)$ initialized to 0. When there is no valid segmentation for some $j, e(j) = j+1$.

To test for a valid range $[j' + 1..j]$, we adjust the sliding window preprocessing algorithm of Sect. 4.3 in order to integrate it with the computation of the recurrence as follows:

1. A unidirectional BWT index [38] is built on the MSA rows concatenated into one long string, after the gap characters are removed and some separator symbols added between the rows.
2. The search on each row of MSA is initiated for each j, decreasing j' from $j - 1$ to 0. This means only left-extensions are required.
3. When MSA$[i, j]$ is accessed to alter the BWT interval, the old interval is retained if MSA$[i, j] = $ -.
4. Modification 3) can cause intervals to become nested (exactly when substring spell(MSA$[i', j'..j]$) becomes a prefix of spell(MSA$[i, j'..j]$)), and this needs to be checked for the proper detection of valid ranges.

Recall that at any step of the preprocessing algorithm of Sect. 4.3, we had a non-nested set $I = \{[i'_a..i_a]\}_{a \in \{1,2...m\}}$ of intervals. We exploited the non-nestedness in the use of a bitvectors M (marking suffixes of current column), B (BWT interval beginning), and E (BWT interval ending) to detect if I contains only BWT intervals of suffixes of the current column of the MSA. This gave us the linear time algorithm. Now that the intervals can become nested, these bitvectors no longer work as intended. Instead we resort to a generic method to check nestedness, and to compute the size of the union of distinct intervals in I, when no nestedness is detected. If no nestedness is detected, and the size of the union is m, we know that the range in consideration is valid. This can be done in $m \log m$ time e.g. by sorting the interval endpoints and

simple scanning to maintain how many active intervals there are at the endpoints. If there is more than one active interval at any point, the range is not valid. Otherwise the range is valid, and the size of the union of intervals is just the sum of their lengths. This nestedness check and the computation of the union of intervals is repeated at each column.

Theorem 9 *The values $e(j)$ of Eq. (3), for all $j \in [1..n]$, can be computed in $O(mne_{\max} \log m)$ time, where $e_{\max} = \max_j e(j)$. The optimal segmentation defined by Eq. (3) yields a repeat-free elastic founder graph.*

Proof The unidirectional BWT index can be constructed in $O(mn)$ time and each left-extension takes constant time [38]. We can start comparing $\max(j - j', e(j'))$ from $j' = j - 1$ decreasing j' by one each step and maintaining $e(j)$ as the minimum value so far. Once value $j - j'$ grows bigger than current $e(j)$, we know that the value of $e(j)$ can no longer decrease. This means we can decrease j' exactly $e(j)$ times. At each decrease of j', we do m left-extensions (one for each row), and then check for the validity by computing the union of search intervals in $O(m \log m)$ time. This gives the claimed running time. Traceback from $e(n)$ gives an optimal repeat-free segmentation. □

9 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 [29]. 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 Figs. 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 ensure 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
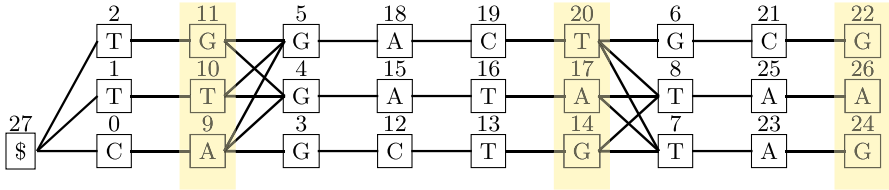
Fig. 5 Repeat-free block NFA. The last columns of each block are highlighted
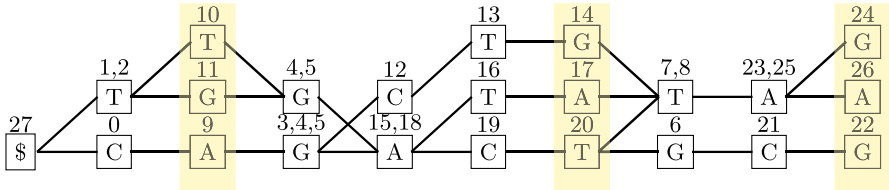

Fig. 6 The DFA resulting from the subset construction for the NFA in Fig. 5. The numbers above the nodes specify the subset of NFA states corresponding to the DFA state
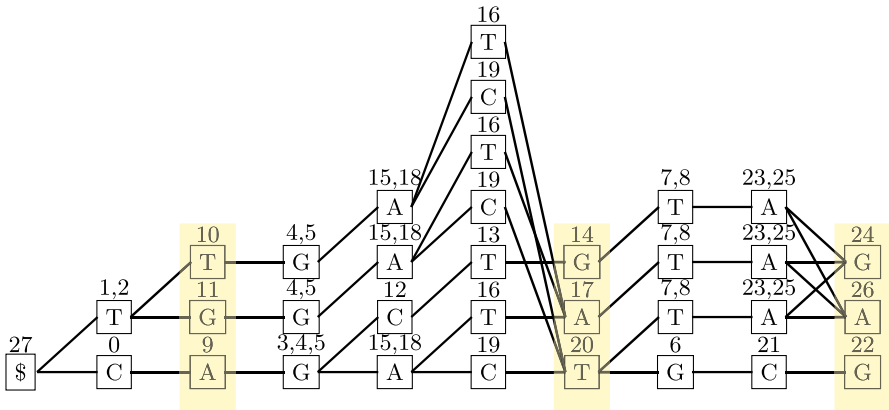

Fig. 7 The Wheeler DFA resulting from running our Wheeler expansion algorithm on the DFA in Fig. 6

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 Fig. 7 for an example. The formal definition of the transformation and the proof of sortability are in Sect. 9.1. We denote the transformed graph with G' and obtain the following result:

Lemma 13 *The number of nodes in G' is at most $O(NH)$, where H 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. [30]. Finally, we can find the minimum equivalent Wheeler graph by running the general Wheeler graph minimization algorithm of Alanko et al. [30].

With the input graph now converted into a Wheeler graph, one can deploy succinct data structures supporting fast pattern matching [29, Lemma 4], leading to the following result:

Corollary 3 *A repeat-free founder/block graph G or a repeat-free elastic degenerate string can be indexed in $O(NH)$ time into a Wheeler-graph-based data structure occupying $O(NH \log |\Sigma|)$ bits of space, where N is the total number of characters in the node labels of G, H is the height 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.*

9.1 Wheeler Graph Details and Proofs

Suppose we have the NFA F corresponding to a repeat-free EFG, and let G be the determinization of F defined above. Denote with $P(v)$ the set of path labels from the initial state to v. Since the graph is a DFA, all sets $P(v)$ are disjoint. Denote with $P_{min}(v)$ and $P_{max}(v)$ the colexicographic minimum and maximum of $P(v)$ respectively. We denote the colexicographic order relation with $\prec$.

We say that a node v is *atomic* if for all path labels α in the graph, we have $\alpha \in P(v)$ iff $P_{min}(v) \preceq \alpha \preceq P_{max}(v)$. A DFA is Wheeler if and only if all its nodes are atomic [30]. In this case, the Wheeler order $<$ of nodes is defined so that $v < u$ if $\alpha \prec \beta$ for some strings $\alpha \in P(v)$ and $\beta \in P(u)$ (this is well-defined when all nodes are atomic).

We will expand the graph so that all its nodes become atomic. To achieve this, we process the nodes in a topological order. If the current node v is at the end of a block, we do nothing. Otherwise, we apply the following transformation. Suppose the in-neighbors of v are $u_1, \ldots, u_k$ and the out-neighbors of v are $w_1, \ldots, w_l$. We delete v, add k new nodes $v_1, \ldots v_k$ and add the sets of edges $\{(u_i, v_i) \mid 1 \leq i \leq k\}$ and $\{(v_i, w_j) \mid 1 \leq i \leq k \text{ and } 1 \leq j \leq l\}$. In other words, we *distribute* the in-edges of v and *duplicate* the out-edges. The automaton remains deterministic after this. Figure 7 shows an example of the expansion.

We denote the resulting graph after all transformations with G'. First we note that by construction, the language of G' is the same as the language of G: any path from the initial state in G can be mapped to a corresponding path with the same label in G' and vice versa. Next, we show that the graph is Wheeler-sortable:

Lemma 14 *Every node v in G' is atomic.*

Proof If v is in the first block G', then $|P(v)| = 1$, so v is trivially atomic. Suppose v is not in the first block. Then all strings in $P(v)$ are of the form $\alpha \beta \gamma$, with $|\alpha| \geq 0$, $|\beta| > 0$ and $|\gamma| \geq 0$, such that β occurs in the NFA F only once (repeat-free property) and γ is a prefix of some string of a block. By the construction of G', strings β and γ are the same for all strings in $P(v)$. Consider a string δ in G' such that $P_{min}(v) \preceq \delta \preceq P_{max}(v)$. Then δ must be suffixed by $\beta \gamma$, so it follows that $\delta \in P(v)$, since (1) all occurrences of β lead to the same node u at the end of the previous block in G' and (2) all paths from u with the label γ must lead to v, or otherwise G' is not deterministic, a contradiction. ☐

Next, we show that transformation from F to G' increases the number of nodes by at most a polynomial amount.

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

Proof Each non-source node v of G' can be associated to a pair (u, α), where u is the node of G' reached by walking from v back to the end of the previous block, and α is the label of the path from u to v. If v is at the first block, we set u to the added source node. If v is at the end of a block, we set $u = v$ and set α to be the empty string. These pairs are all distinct because G' is deterministic.

We bound the number of nodes in G' by bounding the number of possible distinct pairs (u, α). Each end-of-block node v of G' is paired only with prefixes of strings from the next block. Let $end(b)$ be the set of nodes of F that are at the end of block b and let $f(b)$ be the set of nodes at block b in F. Then the total number of possible pairs with nonempty α is at most $\sum_{b=0}^{B-1} |end(b)| \cdot |f(b+1)|$, where B is the number of blocks, with block zero corresponding to the initial state. The number of pairs with empty α is $\sum_{b=0}^{B} |end(b)| \leq N$. Bounding $end(b) \leq H$ in the former sum, we have a total of at most $H \sum_{b=0}^{B-1} |f(b+1)| + N = O(HN)$ possible distinct pairs. $\square$

10 Implementation

We implemented construction and indexing of (semi-)repeat-free (elastic) founder graphs. The implementation is available at https://github.com/algbio/founderblockgraphs. Some proof-of-concept experiments can be found in the conference version of the paper [1].

11 Discussion

One characterization of our solution is that we compact those vertical repeats in MSA that are not horizontal repeats. This can be seen as positional extension of variable order de Bruijn graphs. Also, our solution is parameter-free unlike de Bruijn approaches that always need some threshold k, even in the variable order case.

The founder graph concept could also be generalized so that it is not directly induced from a segmentation. One could consider cyclic graphs having the same repeat-free property. This could be an interesting direction in defining parameter-free de Bruijn graphs.

This paper only scratches the surface of a new family of pangenome representations. There are myriad of options how to optimize among the valid segmentations [27, 28]. We studied some of these here, but left open how to e.g. minimize the maximum number of distinct strings in a segment (i.e. *height* of the graph) [27], or how to control the over-expressiveness of the graph. A special case of the former has recently been solved [50].

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. For the latter, a recent work improves the construction time to linear [49]. In our preliminary version in ISAAC 2021, we claimed an indexing solution for the semi-repeat-free case with better space complexity than what we report here in Section 7.2. The reason for this update is a flaw in the indexing solution in ISAAC 2021 such that it can return false positives. In our subsequent work (under preparation), we have found a way to verify such false positives without changing the bounds claimed in ISAAC 2021.

We focused here on the theoretical aspects of indexable founder graphs. Our preliminary experiments [1] 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. Mäkinen, V., Cazaux, B., Equi, M., Norri, T., Tomescu, A.I.: Linear time construction of indexable founder block graphs. In: Kingsford, C., Pisanti, N. (eds.) 20th International Workshop on Algorithms in Bioinformatics, WABI 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference). LIPIcs, vol. 172. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/LIPIcs.WABI.2020.7. pp. 7:1–7:18
2. Equi, M., Norri, T., Alanko, J., Cazaux, B., Tomescu, A.I., Mäkinen, V.: Algorithms and complexity on indexing elastic founder graphs. In: Ahn, H., Sadakane, K. (eds.) 32nd International Symposium on Algorithms and Computation, ISAAC 2021, December 6-8, 2021, Fukuoka, Japan. LIPIcs, vol. 212. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ISAAC.2021.20. pp. 20:1–20:18
3. Maier, D.: The complexity of some problems on subsequences and supersequences. J. ACM **25**(2), 322–336 (1978). https://doi.org/10.1145/322063.322075
4. Chatzou, M., Magis, C., Chang, J.-M., Kemena, C., Bussotti, G., Erb, I., Notredame, C.: Multiple sequence alignment modeling: methods and applications. Briefings in Bioinformatics **17**(6), 1009–1023 (2015)
5. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. Journal of Computational Biology **17**(3), 281–308 (2010)
6. Na, J.C., Park, H., Crochemore, M., Holub, J., Iliopoulos, C.S., Mouchard, L., Park, K.: Suffix tree of alignment: An efficient index for similar data. In: Lecroq, T., Mouchard, L. (eds.) Combinatorial

Algorithms - 24th International Workshop, IWOCA 2013, Rouen, France, July 10-12, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8288, pp. 337–348. Springer, Germany (2013)

7. Na, J.C., Park, H., Lee, S., Hong, M., Lecroq, T., Mouchard, L., Park, K.: Suffix array of alignment: A practical index for similar data. In: Kurland, O., Lewenstein, M., Porat, E. (eds.) String Processing and Information Retrieval - 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7-9, 2013, Proceedings. Lecture Notes in Computer Science, vol. 8214, pp. 243–254. Springer, Germany (2013)

8. Na, J.C., Kim, H., Park, H., Lecroq, T., Léonard, M., Mouchard, L., Park, K.: FM-index of alignment: A compressed index for similar strings. Theoretical Computer Science **638**, 159–170 (2016). https://doi.org/10.1016/j.tcs.2015.08.008. (**Pattern Matching, Text Data Structures and Compression**)

9. Na, J., Kim, H., Min, S., Park, H., Lecroq, T., Leonard, M., Mouchard, L., Park, K.: FM-index of alignment with gaps. Theoretical Computer Science **710**, 148–157 (2016). https://doi.org/10.1016/j.tcs.2017.02.020

10. Gagie, T., Navarro, G.: Compressed indexes for repetitive textual datasets. In: Sakr, S., Zomaya, A.Y. (eds.) Encyclopedia of Big Data Technologies. Springer, Germany (2019)

11. Gagie, T., Navarro, G., Prezza, N.: Fully functional suffix trees and optimal text searching in bwt-runs bounded space. J. ACM **67**(1), 2–1254 (2020)

12. Marschall, T., Marz, M., Abeel, T., Dijkstra, L., Dutilh, B.E., Ghaffaari, A., Kersey, P., Kloosterman, W., Mäkinen, V., Novak, A., et al.: Computational pan-genomics: status, promises and challenges. BioRxiv, 043430 (2016)

13. Amir, A., Lewenstein, M., Lewenstein, N.: Pattern matching in hypertext. J. Algorithms **35**(1), 82–99 (2000)

14. Manber, U., Wu, S.: Approximate string matching with arbitrary costs for text and hypertext. In: IAPR Workshop on Structural and Syntactic Pattern Recognition, Bern, Switzerland, pp. 22–33 (1992)

15. Rautiainen, M., Marschall, T.: Aligning sequences to general graphs in $O(V + mE)$ time. bioRxiv, 216–127 (2017)

16. Equi, M., Grossi, R., Mäkinen, V., Tomescu, A.I.: On the complexity of string matching for graphs. In: Baier, C., Chatzigiannakis, I., Flocchini, P., Leonardi, S. (eds.) 46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece. LIPIcs, vol. 132. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2019). pp. 55:1–55:15

17. Thachuk, C.: 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)

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

19. Equi, M., Mäkinen, V., Tomescu, A.I.: Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. In: Bures, T., Dondi, R., Gamper, J., Guerrini, G., Jurdzinski, T., Pahl, C., Sikora, F., Wong, P.W.H. (eds.) 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. Lecture Notes in Computer Science, vol. 12607, pp. 608–622. Springer, Germany (2021). https://doi.org/10.1007/978-3-030-67731-2_44

20. Aoyama, K., Nakashima, Y., I, T., Inenaga, S., Bannai, H., Takeda, M.: Faster Online Elastic Degenerate String Matching. In: Navarro, G., Sankoff, D., Zhu, B. (eds.) Annual Symposium on Combinatorial Pattern Matching (CPM 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 105. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.CPM.2018.9. pp. 9:1–9:10. https://drops.dagstuhl.de/opus/volltexte/2018/8701

21. Bernardini, G., Gawrychowski, P., Pisanti, N., Pissis, S.P., Rosone, G.: Even Faster Elastic-Degenerate String Matching via Fast Matrix Multiplication. In: Baier, C., Chatzigiannakis, I., Flocchini, P., Leonardi, S. (eds.) 46th International Colloquium on Automata, Languages, and Programming (ICALP 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 132. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). https://doi.org/10.4230/LIPIcs.ICALP.2019.21. pp. 21:1–21:15. http://drops.dagstuhl.de/opus/volltexte/2019/10597

22. Bernardini, G., Pisanti, N., Pissis, S.P., Rosone, G.: Approximate pattern matching on elastic-degenerate text. Theor. Comput. Sci. **812**, 109–122 (2020). https://doi.org/10.1016/j.tcs.2019.08.012

23. Bernardini, G., Pisanti, N., Pissis, S.P., Rosone, G.: Pattern matching on elastic-degenerate text with errors. In: Fici, G., Sciortino, M., Venturini, R. (eds.) String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10508, pp. 74–90. Springer, Germany (2017). https://doi.org/10.1007/978-3-319-67428-5_7

24. Iliopoulos, C.S., Kundu, R., Pissis, S.P.: Efficient pattern matching in elastic-degenerate texts. In: Drewes, F., Martín-Vide, C., Truthe, B. (eds.) Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10168, pp. 131–142 (2017). https://doi.org/10.1007/978-3-319-53733-7_9

25. Gibney, D.: An efficient elastic-degenerate text index? not likely. In: International Symposium on String Processing and Information Retrieval, pp. 76–88 (2020). Springer

26. Gibney, D., Thankachan, S.V.: On the hardness and inapproximability of recognizing wheeler graphs. In: Bender, M.A., Svensson, O., Herman, G. (eds.) 27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany. LIPIcs, vol. 144. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2019). pp. 51:1–51:16

27. Norri, T., Cazaux, B., Kosolobov, D., Mäkinen, V.: Linear time minimum segmentation enables scalable founder reconstruction. Algorithms Mol. Biol. **14**(1), 12:1-12:15 (2019)

28. Cazaux, B., Kosolobov, D., Mäkinen, V., Norri, T.: Linear time maximum segmentation problems in column stream model. In: Brisaboa, N.R., Puglisi, S.J. (eds.) String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11811, pp. 322–336. Springer, Germany (2019)

29. Gagie, T., Manzini, G., Sirén, J.: Wheeler graphs: A framework for bwt-based data structures. Theor. Comput. Sci. **698**, 67–78 (2017)

30. Alanko, J., D'Agostino, G., Policriti, A., Prezza, N.: Regular languages meet prefix sorting. In: Chawla, S. (ed.) Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020, pp. 911–930. SIAM, USA (2020)

31. De La Briandais, R.: 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), pp. 295–298. Association for Computing Machinery, New York, NY, USA (1959). https://doi.org/10.1145/1457838.1457895

32. Farach, M.: Optimal suffix tree construction with large alphabets. In: Proceedings 38th Annual Symposium on Foundations of Computer Science, pp. 137–143 (1997). IEEE

33. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993). https://doi.org/10.1137/0222058

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

35. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)

36. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)

37. Schnattinger, T., Ohlebusch, E., Gog, S.: Bidirectional search in a string with wavelet trees and bidirectional matching statistics. Inf. Comput. **213**, 13–22 (2012)

38. Belazzougui, D., Cunial, F., Kärkkäinen, J., Mäkinen, V.: Linear-time string indexing and analysis in small space. ACM Trans. Algorithms **16**(2), 1–54 (2020). https://doi.org/10.1145/3381417. (**Article 17**)

39. Belazzougui, D., Cunial, F.: Fully-functional bidirectional burrows-wheeler indexes and infinite-order de bruijn graphs. In: Pisanti, N., Pissis, S.P. (eds.) 30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy. LIPIcs, vol. 128. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2019). pp. 10:1–10:15

40. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. FOCS, pp. 549–554 (1989)

41. Belazzougui, D., Cunial, F., Kärkkäinen, J., Mäkinen, V.: Versatile succinct representations of the bidirectional burrows-wheeler transform. In: European Symposium on Algorithms, pp. 133–144 (2013). Springer

42. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA, pp. 841–850. ACM/SIAM, USA (2003). http://dl.acm.org/citation.cfm?id=644108.644250

43. Cunial, F., Alanko, J., Belazzougui, D.: A framework for space-efficient variable-order markov models. Bioinformatics **35**(22), 4607–4616 (2019)

44. Alzamel, M., Ayad, L.A.K., Bernardini, G., Grossi, R., Iliopoulos, C.S., Pisanti, N., Pissis, S.P., Rosone, G.: Comparing degenerate strings. Fundam. Informaticae **175**(1–4), 41–58 (2020)

45. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: DeMillo, R.A. (ed.) Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA, pp. 135–143. ACM, USA (1984). https://doi.org/10.1145/800057.808675

46. Iliopoulos, C.S., Radoszewski, J.: Truly subquadratic-time extension queries and periodicity detection in strings with uncertainties. In: Grossi, R., Lewenstein, M. (eds.) 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel. LIPIcs, vol. 54. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2016). pp. 8:1–8:12

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

48. Williams, R.: A new algorithm for optimal 2-constraint satisfaction and its implications. Theor. Comput. Sci. **348**(2–3), 357–365 (2005)

49. Rizzo, N., Mäkinen, V.: Linear time construction of indexable elastic founder graphs. In: Proc. 33rd International Workshop on Combinatorial Algorithms (IWOCA 2022), Springer, LNCS, vol. 13270 (2022). pp. 480–493

50. Rizzo, N., Mäkinen, V.: Indexable elastic founder graphs of minimum height. In: Proc. 33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022), Schloss Dagstuhl - Leibniz-Zentrum für Informatik, LIPIcs, vol. 223 (2022). pp. 19:1–19:19

Authors and Affiliations

Massimo Equi[1] · Tuukka Norri[1] · Jarno Alanko[1,2] · Bastien Cazaux[3] · Alexandru I. Tomescu[1] · Veli Mäkinen[1] ⓘD

Massimo Equi
massimo.equi@helsinki.fi

Tuukka Norri
tuukka.norri@helsinki.fi

Jarno Alanko
jarno.alanko@helsinki.fi

Bastien Cazaux
bastien.cazaux@lirmm.fr

Alexandru I. Tomescu
alexandru.tomescu@helsinki.fi

[1] Department of Computer Science, University of Helsinki, P. O. Box 68, Pietari Kalmin katu 5, Helsinki 00014, Finland

[2] Faculty of Computer Science, Dalhousie University, P. O. Box 15000, 6050 University Avenue, Halifax 6050, Nova Scotia, Canada

[3] LIRMM, Univ. Montpellier, CNRS, LIRMM UMR 5506, CC477, 161 rue Ada, Montpellier 34095, Cedex 5, France