Linear Time Construction of Indexable Founder Block Graphs

Veli Mäkinen
Department of Computer Science, University of Helsinki, Finland
veli.makinen@helsinki.fi

Bastien Cazaux
Department of Computer Science, University of Helsinki, Finland

Massimo Equi
Department of Computer Science, University of Helsinki, Finland
massimo.equi@helsinki.fi

Tuukka Norri
Department of Computer Science, University of Helsinki, Finland
tuukka.norri@helsinki.fi

Alexandru I. Tomescu
Department of Computer Science, University of Helsinki, Finland
alexandru.tomescu@helsinki.fi

Abstract

We introduce a compact pangenome representation based on an optimal segmentation concept that aims to reconstruct *founder sequences* from a multiple sequence alignment (MSA). Such founder sequences have the feature that each row of the MSA is a recombination of the founders. Several linear time dynamic programming algorithms have been previously devised to optimize segmentations that induce *founder blocks* that then can be concatenated into a set of founder sequences. All possible concatenation orders can be expressed as a *founder block graph*. We observe a key property of such graphs: if the node labels (founder segments) do not repeat in the paths of the graph, such graphs can be indexed for efficient string matching. We call such graphs *segment repeat-free founder block graphs*.

We give a linear time algorithm to construct a segment repeat-free founder block graph given an MSA. The algorithm combines techniques from the founder segmentation algorithms (Cazaux et al. SPIRE 2019) and *fully-functional bidirectional Burrows-Wheeler index* (Belazzougui and Cunial, CPM 2019). We derive a succinct index structure to support queries of arbitrary length in the paths of the graph.

Experiments on an MSA of SARS-CoV-2 strains are reported. An MSA of size 410×29811 is compacted in one minute into a segment repeat-free founder block graph of 3900 nodes and 4440 edges. The maximum length and total length of node labels is 12 and 34968, respectively. The index on the graph takes only 3% of the size of the MSA.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Sorting and searching; Theory of computation → Dynamic programming; Applied computing → Genomics

Keywords and phrases Pangenome indexing, founder reconstruction, multiple sequence alignment, compressed data structures, string matching

Digital Object Identifier 10.4230/LIPIcs.WABI.2020.7

Related Version `https://arxiv.org/abs/2005.09342`

Supplementary Material The implementation of several methods proposed in this paper is available at `https://github.com/algbio/founderblockgraphs`.

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

Acknowledgements We wish to thank the anonymous reviewers for useful suggestions to improve the readability.

1 Introduction

Computational pangenomics [13] ponders around the problem of expressing a reference genome of a species in a more meaningful way than as a string of symbols. The basic problem in such generalized representations is that one should still be able to support string matching type of operations on the content. Another problem is that any representation generalizing set of sequences also expresses sequences that may not be part of the real pangenome. That is, a good representation should have a feature to control over-expressiveness and simultaneously support efficient queries.

In this paper, we develop the theory around one promising pangenome representation candidate, the *founder block graph*. This graph is a natural derivative of segmentation algorithms [28, 12] related to *founder sequences* [34].

Consider a set of individuals represented as lists of variations from a common reference genome. Such a set can be expressed as a *variation graph* or as a *multiple sequence alignment*. The former expresses reference as a backbone of an automaton, and adds a subpath for each variant. The latter inputs all variations of an individual to the reference, creating a row for each individual into a multiple alignment. Figure 1 shows an example of both structures with 6 very short genomes.

A multiple alignment of much fewer *founder* sequences can be used to approximate the input represented as a multiple alignment as well as possible, meaning that each original row can be mapped to the founder multiple alignment with a minimum amount of row changes (discontinuities). Finding an optimal set of founders is NP-hard [29], but one can solve relaxed problem statements in linear time [28, 12], which are sufficient for our purposes. As an example on the usefulness of founders, Norri et al. [28] showed that, on a large public dataset of haplotypes of human genome, the solution was able to replace 5009 haplotypes with only 130 founders so that the average distance between row jumps was over 9000 base pairs [28]. This means that alignments of short reads (e.g. 100 bp) very rarely hit a discontinuity, and the space requirement drops from terabytes to just tens of gigabytes. Figure 1 shows such a solution on our toy example.

A *block graph* is a labelled directed acyclic graph consisting of consecutive *blocks*, where a block represents a set of sequences of the same length as parallel (unconnected) nodes. There are edges only from nodes of one block to the nodes of the next block. A *founder block graph* is a block graph with blocks representing the segments of founder sequences corresponding to the optimal segmentation [28]. Fig. 1 visualises such a founder block graph: There the founder set is divided into 3 blocks with the first, the second, and the third containing sequences of length 4, 4, and 3, respectively. The coloured connections between sequences in consecutive blocks define the edges. Such graphs interpreted as automata recognise the input sequences just like variation graphs, but otherwise recognise a much smaller subset of the language. With different optimisation criteria to compute the founder blocks, one can control the expressiveness of this pangenome representation.

In this paper, we show that there is a natural subclass of founder block graphs that admit efficient index structures to be built that support exact string matching on the paths of such graphs. Moreover, we give a linear time algorithm to construct such founder block graphs from a given multiple alignment. The construction algorithm can also be adjusted to produce a subclass of *elastic-degenerate strings* [8], which also support efficient indexing.

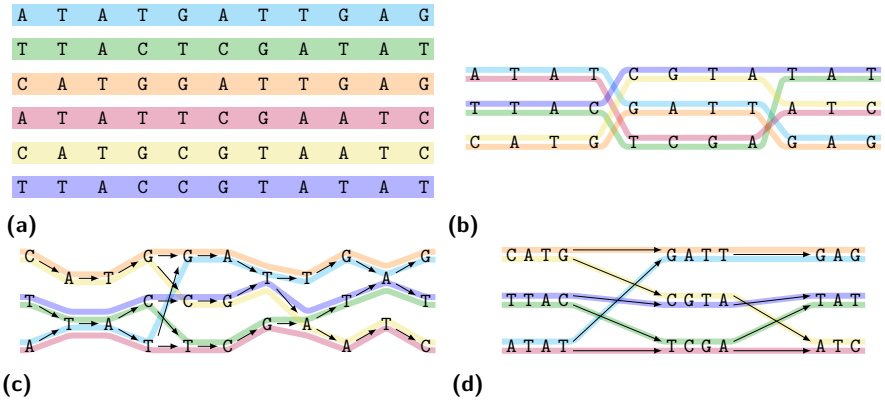

Figure 1 (a) Input multiple alignment, (b) a set of founders with common recombination positions as a solution to a relaxed version of founder reconstruction, (c) a variation graph encoding the input and (d) a founder block graph. Here the input alignment and the resulting variation graph are unrealistically bad; the example is made to illustrate the founders.

The founder block graph definition given above only makes sense if we assume that our input multiple alignment is *gapless*, meaning that the alignment is simply produced by putting strings of equal length under each other, like in Figure 1. We develop the theory around founder block graphs under gapless multiple alignments. However, most of the results can be extended to handle gaps properly.

We start in Sect. 2 by putting the work into the context of related work. In Sect. 3 we introduce the basic notions and tools. In Sect. 4 we study the property of founder block graphs that enable indexing. In Sect. 5 we give the linear time construction algorithm. In Sect. 6 we develop a succinct index structure that supports exact string matching in linear time. In Sect. 7 we consider the general case of having gap symbols in multiple alignment. We report some preliminary experiments in Sect. 8 on the construction and indexing of founder block graphs for a collection of SARS-CoV-2 strains. We consider future directions in Sect. 9.

2 Related work

Indexing directed acyclic graphs (DAGs) for exact string matching on its paths was first studied by Sirén et al. in WABI 2011 [31]. A generalization of *Burrows-Wheeler transform* [11] was proposed that supported near-linear time queries. However, the proposed transformation can grow exponentially in size in the worst case. Many practical solutions have been proposed since then, that either limit the search to short queries or use more time on queries [33, 22, 25, 19, 24, 32]. More recently, such approaches have been captured by the theory on *Wheeler graphs* [18, 20, 2].

Since it is NP-hard to recognize if a given graph is Wheeler [20], it is of interest to look for other graph classes that could provide some indexability functionality. Unfortunately, quite simple graphs turn out to be hard to index [15, 16] (under the Strong Exponential Time Hypothesis). In fact, the reductions by Equi et al. [15, 16] can be adjusted to show that block graphs cannot be indexed in polynomial time to support fast string matching. But as we will see later, further restrictions on block graphs change the situation: We show that there exists a family of founder block graphs that can be indexed in linear time to support linear time queries.

Block graphs have also tight connection to *generalized degenerate* (GD) strings and their *elastic* version. These can also be seen as DAGs with a very specific structure. Matching a GD string is computationally easier and even linear time online algorithms can be achieved to compare two such strings, as analyzed by Alzamel et al. [3]. The elastic counterpart requires more care, as studied by Bernardini et al. [8]. Our results on founder block graphs can be casted on GD strings and elastic strings, as we will show later.

Finally, our indexing solution has connections to succinct representations of *de Bruijn graphs* [10, 9, 7]. Compared to de Bruijn graphs that are cyclic and have limited memory (k-mer length), our solution retains the linear structure of the block graph.

3 Definitions and basic tools

3.1 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 $1 \leq i \leq j \leq n$. Substring $T[i..j]$ where $j < i$ is defined as 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* $\mathbf{0}$ to the strings.

Concatenation of strings A and B is denoted AB.

3.2 Founder block graphs

As mentioned in the introduction, our goal is to compactly represent a *gapless* multiple sequence alignment (MSA) using a founder block graph. In this section we formalize these concepts.

A *gapless multiple sequence alignment* $\texttt{MSA}[1..m, 1..n]$ is a set of m strings drawn from Σ, each of length n. Intuitively, it can be thought of as a matrix in which each row is one of the m strings. Such a structure can be partitioned into what we call a *segmentation*, that is, a collection of sets of shorter strings that can represent the original alignment.

▶ **Definition 1** (Segmentation). *Let* $\texttt{MSA}[1..m, 1..n]$ *be a gapless multiple alignment and let* $R_1, R_2, \ldots, R_m$ *be the strings in MSA. A* segmentation S *of MSA is a set of* b *sets of strings* $S^1, S^2, \ldots, S^b$ *such that for each* $1 \leq i \leq b$ *there exist interval* $[x^{(i)}..y^{(i)}]$ *such that* $S^i = \{R_t[x^{(i)}..y^{(i)}] \mid 1 \leq t \leq m\}$. *Furthermore, it holds* $x^{(1)} = 1$, $y^{(b)} = n$, *and* $y^{(i)} = x^{(i+1)} - 1$ *for all* $1 \leq i < b$, *so that* $S^1, S^2, \ldots, S^b$ covers *the MSA. We call* $W(S^i) = y^{(i)} - x^{(i)} + 1$ *the width of* S^i.

A segmentation of a MSA can naturally lead to the construction of a founder block graph. Let us first introduce the definition of a block graph.

▶ **Definition 2** (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. $\{V^1, V^2, \ldots, V^b\}$ *is a partition of* V, *that is,* $V = V^1 \cup V^2 \cup \ldots \cup V^b$ *and* $V^i \cap V^j = \emptyset$ *for all* $i \neq j$;
2. *if* $(v, w) \in E$ *then* $v \in V^i$ *and* $w \in V^{i+1}$ *for some* $1 \leq i \leq b - 1$;
3. *if* $v, w \in V^i$ *then* $|\ell(v)| = |\ell(w)|$ *for each* $1 \leq i \leq b$ *and if* $v \neq w$, $\ell(v) \neq \ell(w)$.

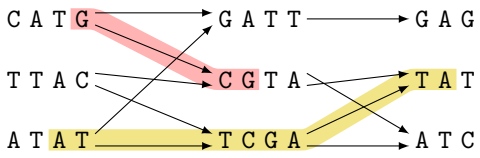

Figure 2 An example of two strings, GCG and ATTCGATA, occurring in $G(S)$.

As a convention we call every V^i a *block* and every $\ell(v)$ a *segment*.

Given a segmentation S of a MSA, we can define the *founder block graph* as a block graph *induced* by S. The idea is to have a graph in which the nodes represents 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.

▶ **Definition 3** (Founder Block Graph). *A* founder block graph *is a block graph* $G(S) = (V, E, \ell)$ induced *by* S *as follows: For each* $1 \leq i \leq b$ *we have* $S^i = \{\ell(v) : v \in V^i\}$ *and* $(v, w) \in E$ *if and only if there exists* $i \in [1..b-1]$ *and* $t \in [1..m]$ *such that* $v \in V^i$, $w \in V^{i+1}$ *and* $R_t[j..j + |\ell(v)| + |\ell(w)| - 1] = \ell(v)\ell(w)$ *with* $j = 1 + \sum_{h=1}^{i-1} W(S^h)$.

We regard the edges of (founder) block graphs to be directed 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 2 illustrates such queries.

In our example in Figure 1, the intervals corresponding to the segmentation would be $[1..4], [5..8], [9..11]$, and the induced founder block graph has thus 3 blocks with 9 nodes and 11 edges in total.

3.3 Basic tools

A *trie* [14] 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; 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.

An *Aho-Corasick automaton* [1] 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 $O(q \log \sigma + \mathtt{occ})$ time, where $\mathtt{occ}$ is the number of matches.

A *suffix array* [26] of string T is an array $\mathtt{SA}[1..n+1]$ such that $\mathtt{SA}[i] = j$ if $T'[j..n+1]$ is the j-th smallest suffix of string $T' = T\mathbf{0}$, where $T \in \{1, 2, \ldots, \sigma\}^n$, and $\mathbf{0}$ is the end marker. Thus, $\mathtt{SA}[1] = n + 1$.

Burrows-Wheeler transform $\mathrm{BWT}[1..n+1]$ [11] of string T is such that $\mathrm{BWT}[i] = T'[\mathtt{SA}[i] - 1]$, where $T' = T\mathbf{0}$ and $T'[-1]$ is regarded as $T'[n+1] = \mathbf{0}$.

A *bidirectional BWT index* [30, 6] 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 $O(n)$ time and it supports finding in $O(q)$ time if a query string $Q[1..q]$ appears as substring of T [6]. Moreover, such query returns an interval pair $([i..j],[i'..j'])$ such that suffixes of T starting at positions $\mathtt{SA}[i], \mathtt{SA}[i+1], \ldots, \mathtt{SA}[j]$ 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 $Q[l-1..r]$ or to the right into $Q[l..r+1]$. This takes constant time [6]. A *fully-functional bidirectional BWT index* [4] expands the steps to allow contracting symbols from the left or from the right. That is, substring $Q[l..r]$ can be modified into $Q[l+1..r]$ or to $Q[l..r-1]$ 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 $\texttt{rank}(B,i)$ returns the number of 1s in $B[1..i]$. Operation $\texttt{select}(B,j)$ 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 [23].

4 Subclass of founder block graphs admitting indexing

We now show that there exists a family of founder block graphs that admit a polynomial time constructable index structure supporting fast string matching. First, a trivial observation: the input multiple alignment is a founder block graph for the segmentation consisting of only one segment. Such founder block graph (set of sequences) can be indexed in linear time to support linear time string matching [6]. Now, the question is, are there other segmentations that allow the resulting founder block graph to be indexed in polynomial time? We show that this is the case.

▶ **Definition 4.** *Founder block graph $G(S)$ is* segment repeat-free *if each $\ell(v)$ for $v \in V$ occurs exactly once in $G(S)$.*

Our example graph (Fig. 1) is not quite segment repeat-free, as `TAT` occurs also as substring of paths starting with `ATAT`.

▶ **Proposition 5.** *Segment repeat-free founder block 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.

Let $P(v)$ be the set of all paths starting from node v and ending in a sink node. 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 segment repeat-free, we know that the lexicographic order of $\ell(L)[i..]$, $L \in P(v)$, 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. 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 on 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 [1] 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 potential match, we need several tries [14]. For each $v \in V$, we build tries $\mathcal{R}(v)$ and $\mathcal{F}(v)$ on the sets $\{\ell(u)^{-1} \mid (u,v) \in E\}$ and $\{\ell(w) \mid (v,w) \in E\}$, respectively, where X^{-1} denotes the reverse $x_{|X|}x_{|X|-1}\cdots x_1$ of string $X = x_1x_2\cdots x_{|X|}$.

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. Note however, that there could be several labels $\ell(v)$ occurring as substrings of Q, so we need to do the verification process for each one of them separately. There can be at most $|Q|$ such candidate occurrences, due to the distinctness of node labels in $G(S)$. In total, this search can take at most $O(|Q|^2\log\sigma)$ time.

We are now ready to specify a theorem that reformulates Proposition 5 in detailed form.

▶ **Theorem 6.** *Let $G = (V,E)$ be a segment repeat-free founder block 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 $O((N+W|E|)\log\sigma)$ time, where $\{1,\ldots,\sigma\}$ is the alphabet for node labels, $W = \max_{v\in V}|\ell(v)|$, and $N = \sum_{v\in V}|\ell(v)|$. 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 $O(|Q|^2\log\sigma)$ time.*

Proof. With preprocessing time $O(N\log\sigma)$ we can build the Aho-Corasick automaton [1]. The tries can be built in $O(\log\sigma)(\sum_{v\in V}(\sum_{(u,v)\in E}|\ell(u)|+\sum_{(v,w)\in E}|\ell(w)|)) = O(|E|W\log\sigma)$ time. The required queries on these structures take $O(|Q|^2\log\sigma)$ time.

To avoid the costly binary search in sorted $\mathcal{P}'$, we instead construct the bidirectional BWT index [6] 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 $O(|E|W)$ from alphabet $\{\mathbf{0},1,2,\ldots,\sigma\}$. The bidirectional 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 [6]. This query equals that of binary search in $\mathcal{P}'$. ◀

We remark that founder block graphs have a connection with *generalized degenerate strings* (GD strings) [3]. In a GD string, sets of strings of equal length are placed one after the other to represent in a compact way a bigger set of strings. Such set contains all possible concatenations of those strings, which are obtained by scanning the individual sets from left to right and selecting one string from each set. The length of the strings in a specific set is called *width*, and the sum of all the widths of all sets in a GD string is the *total width*. Given two GD strings of the same total width it is possible to determine if the intersection of the sets of strings that they represent is non empty in linear time in the size of the GD strings [3]. Thus, the special case in which one of the two GD string is just a standard string can be seen also as a special case of a founder block graph in which every segment is fully connected with the next one and the length of the query string is equal to the maximal length of a path in the graph.

We consider the question of indexing GD strings (fully connected block graphs) to search for queries Q shorter than the total width. We can exploit the segment repeat-free property to yield such an index.

▶ **Theorem 7.** *Let $G = (V,E)$ be a* segment repeat-free GD string *a.k.a. a fully connected segment repeat-free founder block graph with blocks $V^1, V^2, \ldots, V^b$ such that $V = V^1 \cup V^2 \cup \cdots \cup V^b$ and $(v,w) \in E$ for all $v \in V^i$ and $w \in V^{i+1}$, $1 \le i < b$. We can preprocess an index*

structure for G in $O((N+W|E|)\log\sigma)$ time, where $\{1,\ldots,\sigma\}$ is the alphabet for node labels, $W=\max_{v\in V}\ell(v)$, and $N=\sum_{v\in V}\ell(v)$. 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. Recall the index structure of Theorem 6. for the case of GD strings, we can simplify it as follow.

We keep the same BWT index structure and the Aho-Corasick automaton, but we do not need any tries. After finding at most $|Q|$ occurrences of substrings of Q in the graph using the Aho-Corasick automaton on node labels, we mark the matching blocks accordingly. If 2 marked blocks have exactly one marked neighboring block and $|Q|-2$ blocks have 2 marked neighboring blocks, then we have found an occurrence, otherwise not. ◀

Observe that $\max(N, W|E|)\leq mn$, where m and n are the number of rows and number of columns, respectively, in the multiple sequence alignment from where the founder block graph is induced. That is, the index construction algorithms of the above theorems can be seen to be almost linear time in the (original) input size. We study succinct variants of these indexes in Sect. 6, and also improve the construction and query times to linear as side product.

5 Construction of segment repeat-free founder block graphs

Now that we know how to index segment repeat-free founder block 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 [28, 12].

The idea is as follows. Let S be a segmentation of $\texttt{MSA}[1..m, 1..n]$. We say S is *valid* if it induces a segment repeat-free founder block graph $G(S)=(V,E)$. We build such valid S for prefixes of MSA from left to right, minimizing the maximum block length needed.

5.1 Characterization lemma

Given a segmentation S and founder block 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 8** (Characterization). *Let $x^{(i)}=1+\sum_{h=1}^{i-1}W(S^h)$. A segmentation S is valid if and only if, for all blocks $V^i\subseteq V$, $1\leq t\leq m$ and $j\neq x^{(i)}$, if $v\in V^i$ then $R_t[j..j+|\ell(v)|-1]\neq\ell(v)$.*

Proof. To see that this is a necessary condition for the validity of S, notice that each row of 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. ◀

5.2 From characterization to a segmentation

Among the valid segmentations, we wish to select an *optimal* segmentation under some goodness criteria. Earlier work [28, 12] has considered various goodness criteria, solving the associated segmentation problems in linear time. In this paper, we focus on minimizing the maximum width of the segments. For example, the (non-valid) segmentation in Figure 1 has score 4, as the intervals of the segments are $[1..4], [5..8]$, and $[9..11]$. That is, as $W(S^1) = 4 - 1 + 1 = 4$, $W(S^2) = 8 - 5 + 1 = 4$, $W(S^3) = 11 - 9 + 1 = 3$, the maximum of these is 4. This criteria is analogous to one studied by Cazaux et al. [12], and appears to be the most natural one to be studied together with validity constraint; both deal directly with the segment intervals.

Let us now develop a dynamic programming recurrence for finding the minimum scoring valid segmentation with the score being the maximum width. Let $s(j')$ be the score of a minimum scoring valid segmentation $S^1, S^2, \ldots, S^b$ of prefix $\texttt{MSA}[1..m, 1..j']$, where the score is defined as $\max_{i:1 \leq i \leq b} W(S^i)$. We can compute

$$s(j) = \min_{j':0 \leq j' \leq v(j)} \max(j - j', s(j')), \tag{1}$$

where $v(j)$ is the largest integer such that segment $\texttt{MSA}[1..m, v(j) + 1..j]$ is valid. The segment is valid iff each substring $\texttt{MSA}[i, v(j) + 1..j]$, for $1 \leq i \leq m$, occurs as many times in $\texttt{MSA}[1..m, v(j) + 1..j]$ as in the whole MSA. If such $v(j)$ does not exist for some j, we set $v(j) = 0$. The intuition is that $[j' + 1..j]$ forms the last valid segment of the segmentation, and since $s(j')$ is the score of optimal valid segmentation of $\texttt{MSA}[1..m, 1..j']$, the maximum of $s(j')$ and the length of the last segment decides the score $s(j)$. To initialize the recurrence, one can set $s(0) = 0$ and $s(j) = \infty$ for $1 \leq j \leq J$ for which $v(j)$ is undefined. The recurrence can then be applied for $j \in [J + 1..n]$.

We can compute the score $s(n)$ of the optimal segmentation of MSA in $O(ns_{\texttt{max}})$ time after preprocessing values $v(j)$ in $O(mns_{\texttt{max}} \log \sigma)$ time, where $s_{\texttt{max}} = \max_{j:v(j)>0} s(j)$. For the former, one can start comparing $\max(j - j', s(j'))$ from $j' = v(j)$ decreasing j' by one each step, and then the value $j - j'$ grows bigger than $s(j')$ at latest after $s(j) - (j - v(j)) \leq s(j)$ steps. For preprocessing, we build the bidirectional BWT index of the MSA in $O(mn)$ time [6]. At column j, consider the trie containing the reverse of the rows of $M[1..m, 1..j]$. Search the trie paths from the bidirectional BWT index until the number of leaves in each trie subtree equals the length of the corresponding BWT interval. Let j' be the column closest to j where this holds for all trie paths. Then one can set $v(j) = j'$. The $O(m(j - v(j)) \log \sigma)$ time construction of the trie has to be repeated for each column. As $j - v(j) \leq s(j)$, the claimed preprocessing time follows.

5.3 Faster preprocessing

We can do the preprocessing in $O(mn)$ time.

▶ **Theorem 9.** *Given a multiple sequence alignment* $\texttt{MSA}[1 \ldots m, 1 \ldots n]$*, values* $v(j)$ *for each* $1 \leq j \leq n$ *can be computed in* $O(mn)$ *time, where* $v(j)$ *is the largest integer such that segment* $\texttt{MSA}[1..m, v(j) + 1..j]$ *is valid.*

Proof. Let us build the bidirectional BWT index [6] of MSA rows concatenated into one long string. We will run several algorithms in synchronization over this BWT index, but we explain them first as if they would be run independently.

Algorithm 1 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 k-th step stores $M[i] = 1$ iff $BWT[i]$ is the k-th last symbol of some row.

Algorithm 2 uses the *variable length sliding window* approach of Belazzougui and Cunial [4] 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 [4]. 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 $\texttt{MSA}[a, j'+1..n]$ 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 [4]. 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 the Algorithm 2 run on the rest of the rows as Algorithms $3, 4, \ldots, m+1$.

Let the current BWT interval in Algorithms 2 to $m+1$ be $[j'+1..j]$. The problematic part in them is checking if the corresponding *active* BWT intervals $[i'_a..i_a]$ for Algorithms $a \in \{2, 3, \ldots, m+1\}$ contain only suffixes originating from column $j'+1$. To solve this, we run Algorithm 1 as well as Algorithms 2 to $m+1$ in synchronization so that we are at the k-th step in Algorithm 1 when we are processing interval $[j'+1..j]$ in rest of the algorithms, for $k = n - j'$. In addition, we maintain bitvectors B and E such that $B[i'_a] = 1$ and $E[i_a] = 1$ for $a \in \{2, 3, \ldots, m+1\}$. 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 $I = \{[i'_a..i_a]\}_{a \in \{2,3 \ldots m+1\}}$. This is because I cannot contain nested interval pairs as all strings in segment $[j'+1..j]$ of MSA are of equal length, and thus their BWT intervals cannot overlap except if the intervals are exactly the same.

Finally, the running time is $O(mn)$, since each extend-left and contract-right operations take constant time [4], and since the bitvectors are manipulated locally only on indexes that are maintained as variables during the execution. ◀

5.4 Faster main algorithm

Recall Eq. (1). Before proceeding to the involved optimal solution, we give some insights by first improving the running time to logarithmic per entry.

As it holds $v(1) \leq v(2) \leq \cdots \leq v(n)$, the range where the minimum is taken grows as j grows. Now, $[j'..j'+s(j')]$ can be seen as the *effect range* of $s(j')$: for columns $j > j'+s(j')$ the maximum from the options is $j - j'$. Consider maintaining (key, value) pairs $(s(j')+j', s(j'))$ in a binary search tree (BST). When computing $s(j)$ we should have pairs $(s(j')+j', s(j'))$ for $1 \leq j' \leq v(j)$ in BST. Value $s(j)$ can be computed by taking range minimum on BST values with keys in range $[j..\infty]$. Such query is easy to solve in $O(\log n)$ time. If there is nothing in the interval, $s(j) = j - v(j)$. Since this is semi-open interval on keys in range $[1 \ldots 2n]$, BST can be replaced by van Emde Boas tree to obtain $O(n \log\log n)$ time computation of all values [17]. Alternatively, we can remove elements from the BST once they no longer can be answers to queries, and we can get $O(n \log s_{\max})$ solution. To obtain better running time, we need to exploit more structural properties of the recurrence.

Cazaux et al. [12] 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

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

▶ **Lemma 10.** *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 $\max(A+1, B) < \max(C+1, D)$.

◀

▶ **Lemma 11.** *By initialising $s(1)$ to a threshold K, for any $j \in [1..n]$, we have $s(j) \leq \max(j, K)$.*

Proof. We are going to show by induction. The base case is obvious because $s(1) = K \leq \max(1, K)$. As $s(j) = \min_{j':1 \leq j' \leq v(j)} \max(j - j', s(j'))$, by using induction, $s(j) \leq \min_{j':1 \leq j' \leq v(j)} \max(j, K) \leq \max(j, K)$ ◀

Thanks to Lemma 11, by taking the threshold $K = n + 1$, the values $s(j)$ are in $O(n)$ for all j in $[1..n-1]$.

▶ **Lemma 12.** *Given $j^\star \in [x(j-1)+1..v(j)]$, we can compute in constant time if*

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

Proof. We need just to compare $k = \max(j - j^\star, s(j^\star))$ and $s(j^\diamond)$ where $j^\diamond$ is in $Argmin_{j' \in [j^\star+1..v(j)]} s(j')$. 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 for all $\max(j - j', s(j'))$. Hence we have $j^\star = \max Argmin_{j' \in [j^\star..v(j)]} \max(j - j', s(j'))$.

Otherwise, $s(j^\diamond) \geq k$ and as $k \geq j - j^\star$, $\max(j - j^\diamond, s(j^\diamond)) \geq k$. In this case $j^\star \neq \max Argmin_{j' \in [j^\star..v(j)]} \max(j - j', s(j'))$. By using the constant time semi-dynamic range maximum query by Cazaux et al. [12] on the array $s(.)$, we can obtain in constant time $j^\diamond$ and thus check the equality in constant time. ◀

▶ **Theorem 13.** *The values $s(j)$, for all $j \in [1..n]$, can be computed in $O(n)$ time after an $O(nm)$ time preprocessing.*

Proof. We begin by preprocessing all the values of $v(j)$ in $O(mn)$ (Theorem 9). 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 12 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 $j + 1$. ◀

6 Succinct index for segment-free founder block graphs

Recall the indexing solutions of Sect. 4 and the definitions from Sect. 3.

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 $C = \prod_{i \in \{1,2,\ldots,b\}} \prod_{v \in V^i, (v,w) \in E} \ell(v)\ell(w)\mathbf{0}$.

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 segment repeat-free property it follows that this interval can be split into two subintervals, $\mathtt{SA}[i..j]$ and $\mathtt{SA}[j+1..k]$, such that suffixes in $\mathtt{SA}[i..j]$ start with $\ell(v)\mathbf{0}$ and suffixes in $\mathtt{SA}[j+1..k]$ start with $\ell(v)\ell(w)$, where $(v, w) \in E$. Moreover, $\mathrm{BWT}[i..j]$ equals multiset $\{\ell(u)[|\ell(u)| - 1] \mid (u, v) \in E\}$ *sorted in lexicographic order*. This follows by considering the lexicographic order of suffixes $\ell(u)[|\ell(u)| - 1]\ell(v)\mathbf{0}\ldots$ for $(u, v) \in E$. That is, $\mathrm{BWT}[i..j]$ (interpreted as a set) represents the children of the root of the trie $\mathcal{R}(v)$ considered in Sect. 4.

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 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 $\mathrm{BWT}[j'] = \mathbf{0}$. 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 Theorems 6 and 7 as follows.

▶ **Theorem 14.** *Let $G = (V, E)$ be a segment repeat-free founder block graph (or a segment repeat-free GD 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 $O(W|E| \log \sigma)$ bits in $O(W|E|)$ time, where $\{1, \ldots, \sigma\}$ is the alphabet for node labels and $W = \max_{v \in V} \ell(v)$. 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. (sketch) 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 6. We need to show that a) the expansions do not yield spurious occurrences for such short

Table 1 Segment of MSA with and without gaps.

segment of MSA	gaps removed
-AC-CGATC-	ACCGATC
-A-CCGATCC	ACCGATCC
AAC-CGATC-	AACCGATC
AAC-CGA-C-	AACCGAC

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 $\texttt{SA}[i..k]$ where all suffixes match $\ell(v)$ and thus corresponds to a node $v \in V$. The suffix of the query processed before reaching interval $\texttt{SA}[i..k]$ 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 backward step from $\text{BWT}[i..k]$, 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 6. 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. ◀

7 Gaps in multiple alignment

We have so far assumed that our input is a gapless multiple alignment. Let us now consider how to extend the results to the general case. 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.

It turns out that allowing gaps in MSA indeed makes the computation of valid ranges more difficult. To see this, consider the example in Table 1. The second column in Table 1 shows the sequences after gaps are removed. Without even seeing the rest of the MSA, one can see that this is not a valid block, as the first string is a prefix of the second. With gapless MSAs this was not possible and the algorithm in Sect. 5.3 exploited this fact.

Modifying the construction algorithm to handle gaps properly is possible, but non-trivial. We leave this extension to journal version; see however Sections 8 and 9 for some further insights.

Despite the computation of the segmentation is affected by gaps in MSA, once such valid segmentation is found, the rest of the results stay unaffected. All the proposed definitions extend to this interpretation by just omitting gap symbols when reading the strings. The consequence for founder block graph is that strings inside a block can be of variable length. Interestingly, with this interpretation Theorem 7 can be expressed with GD strings replaced by *elastic strings* [8].

8 Implementation and experiments

We implemented several methods proposed in this paper. The implementation is available at `https://github.com/algbio/founderblockgraphs`. Some preliminary experiments are reported below.

8.1 Construction

We implemented the founder block graph construction algorithm of Sect. 5.2 with the faster preprocessing routine of Sect. 5.3. In place of fully-functional bidirectional BWT index, we used similar routines implemented in compressed suffix trees of SDSL library [21]; this affects the theoretical running time by a logarithm factor.

To test the implementation we downloaded 1484 strains of SARS-CoV-2 strains stored in NCBI database.[1] We created a multiple sequence alignment of the strains using *ViralMSA*[27]. We then filtered out rows that contained gaps or N's. We were left with a multiple sequence alignment of 410 rows and 29811 columns. Our algorithm took 58 seconds to produce the optimal segmentation on Intel(R) Xeon(R) CPU, E5-2690, v4, 2.60GHz. There were 3352 segments in the segmentation, the maximum segment length was 12, and the maximum number of founder segments in a block was 12. The founder block graph had 3900 nodes and 4440 edges. The total length of node labels was 34968. The graph size was thus less than 1% of the MSA size.

We also implemented support to construct founder block graphs for general MSA's that contain gaps. The modification to the gapless case was that nested BWT intervals needed to be detected. We stopped left-extension as soon as the BWT intervals contained no other repeats than those caused by nestedness. This left valid range undefined on such columns, but for the rest the valid range can still be computed correctly (undefined values could be postprocessed using matching statistics [5] on all pairs of prefixes preceding suffixes causing nested intervals). Initial experiments show very similar behaviour to the gapless case, but we defer further experiments until the implementation is mature enough.

8.2 Indexing

We implemented the succinct indexing approach of Sect. 6. On the founder block graph of the previous experiment, the index occupied 87 KB. This is 3% of the original input size, as the encoding of the input MSA with 2 bits per nucleotide takes 2984 KB.

Figure 3 shows an experiment with indexes built on different size samples of the MSA rows, and with querying patterns of varying length sampled from the same rows. As can be seen, the query times are not affected by the size of the MSA samples (showing independence of the input MSA), but only on their length (showing linear dependency on the query length).

9 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.

[1] `https://www.ncbi.nlm.nih.gov/`, accessed 24.04.2020.

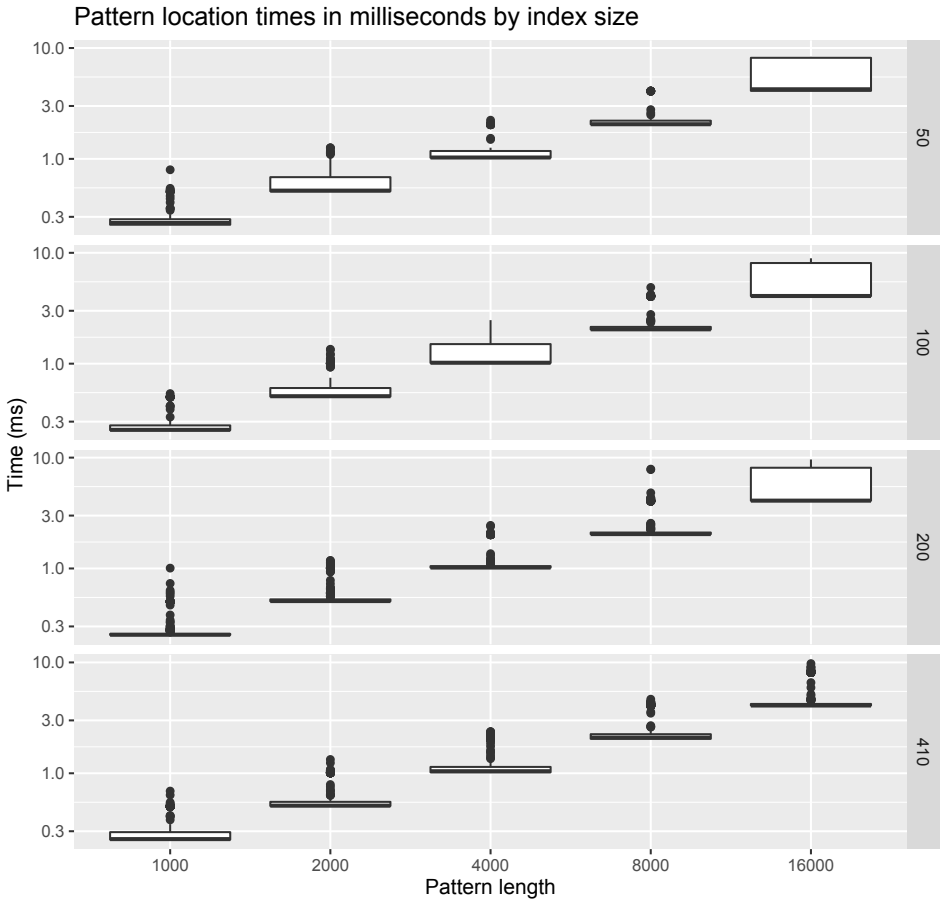

Figure 3 Running time of querying patterns from the founder block graph. The sample sizes (for MSA row subsets) are shown on the right-hand side of each plot. The plots show averages and distribution over 10 repeats of each search, where one search consist of a set of query patterns of given length randomly sampled from the respective MSA row subset. The pattern set sample size (10,20,30,40, respectively) grows by the MSA sample size, but the reported numbers are normalized so that the query time (milliseconds) is per pattern. This experiment was run on Intel(R) Core(TM) i5-4308U CPU, 2.80GHz.

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

For journal version we plan to include a proper extension of the approach to general MSA's containing gaps. Our implementation already contains support for such MSA's, but the theory framework still requires some more work to show that such extension can be done without any effect on the running time.

On the experimental side, there is still much more work to be done. So far, we have not optimized any of the algorithms for multithreading nor for space usage. For example, one could use BWT indexes engineered for highly repetitive text collections to build the founder block graphs in space proportional of the compressed MSA. Such optimizations are essential

for applying the approach on e.g. human genome data. Past experience on similar solutions [28] indicate that our approach should easily be applicable to much larger datasets than those we covered in our preliminary experiments.

Finally, 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 [28, 12], e.g. by optimizing the number of founder segments [28] or controlling the over-expressiveness of the graph, rather than minimizing the maximum block size as studied here.

References

1 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

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

3 Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Degenerate string comparison and applications. In Laxmi Parida and Esko Ukkonen, editors, *18th International Workshop on Algorithms in Bioinformatics, WABI 2018, August 20-22, 2018, Helsinki, Finland*, volume 113 of *LIPIcs*, pages 21:1–21:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

4 Djamal Belazzougui and Fabio Cunial. Fully-functional bidirectional burrows-wheeler indexes and infinite-order de bruijn graphs. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 10:1–10:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

5 Djamal Belazzougui, Fabio Cunial, and Olgert Denas. Fast matching statistics in small space. In Gianlorenzo D'Angelo, editor, *17th International Symposium on Experimental Algorithms, SEA 2018, June 27-29, 2018, L'Aquila, Italy*, volume 103 of *LIPIcs*, pages 17:1–17:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.SEA.2018.17.

6 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *ACM Trans. Algorithms*, 16(2), March 2020. doi:10.1145/3381417.

7 Djamal Belazzougui, Travis Gagie, Veli Mäkinen, Marco Previtali, and Simon J. Puglisi. Bidirectional variable-order de bruijn graphs. *Int. J. Found. Comput. Sci.*, 29(8):1279–1295, 2018. doi:10.1142/S0129054118430037.

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

9 Christina Boucher, Alexander Bowe, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane. Variable-order de bruijn graphs. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*, pages 383–392. IEEE, 2015. doi:10.1109/DCC.2015.70.

10 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de bruijn graphs. In Benjamin J. Raphael and Jijun Tang, editors, *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, volume 7534 of *Lecture Notes in Computer Science*, pages 225–235. Springer, 2012. doi:10.1007/978-3-642-33122-0_18.

11 M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

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

13 Computational Pan-Genomics Consortium et al. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, page bbw089, 2016.

14 Rene De La Briandais. File searching using variable length keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), page 295–298, New York, NY, USA, 1959. Association for Computing Machinery. doi:10.1145/1457838.1457895.

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

16 Massimo Equi, Veli Mäkinen, and Alexandru I. Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless seth fails, 2020. arXiv:2002.00629.

17 Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and related techniques for geometry problems. In Richard A. DeMillo, editor, *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA*, pages 135–143. ACM, 1984. doi:10.1145/800057.808675.

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

19 Erik Garrison, Jouni Sirén, Adam Novak, Glenn Hickey, Jordan Eizenga, Eric Dawson, William Jones, Shilpa Garg, Charles Markello, Michael Lin, and Benedict Paten. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 36, August 2018. doi:10.1038/nbt.4227.

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

21 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014. doi:10.1007/978-3-319-07959-2_28.

22 Lin Huang, Victoria Popic, and Serafim Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):361–370, 2013.

23 G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554, 1989.

24 Daehwan Kim, Joseph Paggi, Chanhee Park, Christopher Bennett, and Steven Salzberg. Graph-based genome alignment and genotyping with hisat2 and hisat-genotype. *Nature Biotechnology*, 37:1, August 2019. doi:10.1038/s41587-019-0201-4.

25 Sorina Maciuca, Carlos del Ojo Elias, Gil McVean, and Zamin Iqbal. A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference. In *Algorithms in Bioinformatics - 16th International Workshop, WABI 2016, Aarhus, Denmark, August 22-24, 2016. Proceedings*, volume 9838 of *Lecture Notes in Computer Science*, pages 222–233. Springer, 2016.

26 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

27 Niema Moshiri. ViralMSA: Massively scalable reference-guided multiple sequence alignment of viral genomes. *bioRxiv*, 2020. doi:10.1101/2020.04.20.052068.

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

29 Pasi Rastas and Esko Ukkonen. Haplotype inference via hierarchical genotype parsing. In *Algorithms in Bioinformatics, 7th International Workshop, WABI 2007, Philadelphia, PA, USA, September 8-9, 2007, Proceedings*, pages 85–97, 2007.

30 Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Inf. Comput.*, 213:13–22, 2012.

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

32 Jouni Sirén, Erik Garrison, Adam M Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinformatics*, 36(2):400–407, July 2019. doi:10.1093/bioinformatics/btz575.

33 Chris Thachuk. Indexing hypertext. *Journal of Discrete Algorithms*, 18:113–122, 2012.

34 Esko Ukkonen. Finding founder sequences from a set of recombinants. In *Algorithms in Bioinformatics, Second International Workshop, WABI 2002, Rome, Italy, September 17-21, 2002, Proceedings*, pages 277–286, 2002.