

Graph Indexing Beyond Wheeler Graphs

Jarno N. Alanko ✉ 

Department of Computer Science, University of Helsinki, Finland

Massimo Equi ✉ 

Department of Computer Science, Aalto University, Finland

Simon J. Puglisi ✉ 


Department of Computer Science, University of Helsinki, Finland

Kunihiko Sadakane ✉ 

Department of Mathematical Informatics, University of Tokyo, Japan

Elena Biagi ✉ 

Department of Computer Science, University of Helsinki, Finland

Veli Mäkinen ✉ 

Department of Computer Science, University of Helsinki, Finland

Nicola Rizzo ✉ 

Department of Computer Science, University of Helsinki, Finland

Jouni Sirén ✉ 

Genomics Institute, University of California, Santa Cruz, CA, USA

Abstract

After the discovery of the FM index, which linked the Burrows-Wheeler transform (BWT) to pattern matching on strings, several contemporaneous strands of research began on indexing more complex structures with the BWT, such as tries, finite languages, de Bruijn graphs, and aligned sequences. These directions can now be viewed as culminating in the theory of Wheeler Graphs, but sometimes they go beyond. This chapter reviews the significant body of “proto Wheeler Graph” indexes, many of which exploit characteristics of their specific case to outperform Wheeler graphs, especially in practice.

2012 ACM Subject Classification Theory of computation → Sorting and searching; Theory of computation → Graph algorithms analysis; Theory of computation → Pattern matching; Theory of computation → Formal languages and automata theory

Keywords and phrases indexing, compression, compressed data structures, string algorithms, pattern matching

Digital Object Identifier 10.4230/OASICS.Manzini.2025.13

Funding *Jarno N. Alanko*: Funded by the Helsinki Institute for Information Technology (HIIT).

Elena Biagi: Funded by the Academy of Finland via grant 339070.

Massimo Equi: Partially funded by the Research Council of Finland, Grant 359104.

Veli Mäkinen: Partially funded by the European Union’s Horizon Europe research and innovation programme under grant agreement No 101060011 (TeamPerMed).

Simon J. Puglisi: Funded by the Academy of Finland via grant 339070.

Nicola Rizzo: Funded by the European Union’s Horizon Europe research and innovation programme under grant agreement No 101060011 (TeamPerMed).

Jouni Sirén: Funded by the National Human Genome Research Institute (NHGRI) under award numbers U01HG013748 and U41HG010972.

Acknowledgements We wish to thank Hideo Bannai for spotting a mistake in the original version of Definition 22.

1 Introduction

After the discovery of the FM index [22], designed for pattern matching on strings using the Burrows-Wheeler transform (BWT)[11], many groups explored BWT-inspired indexing of more complex structures, such as tries [21], finite languages [48, 49], de Bruijn graphs [10, 5], and aligned sequences [20, 43]. These directions can now be viewed as culminating in



© Jarno N. Alanko, Elena Biagi, Massimo Equi, Veli Mäkinen, Simon J. Puglisi, Nicola Rizzo, Kunihiko Sadakane, and Jouni Sirén;
licensed under Creative Commons License CC-BY 4.0

The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini’s 60th Birthday.

Editors: Paolo Ferragina, Travis Gagie, and Gonzalo Navarro; Article No. 13; pp. 13:1–13:29



OpenAccess Series in Informatics

OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the theory of Wheeler Graphs, which is covered in the next chapter. Before that, however, in this chapter we review the significant body of “proto Wheeler Graph” indexes, many of which exploit characteristics of their specific case to outperform Wheeler graphs, especially in practice. While indeed many of the approaches reviewed (but not all) can be encapsulated in the general Wheeler Graph framework, the connection usually covers only the last indexing step. Construction of the object to be indexed is typically the most complex step and this is tailored for each approach. Especially these construction considerations go beyond the Wheeler Graph framework.

We start with key notions and index structures for strings and string collections. Before proceeding to general graphs, we extend the techniques to tries. Then we continue with de Bruijn graphs that represent overlaps of length k substrings of a string collection. The studied extensions of BWT are then shown to work for general graphs, albeit with the caveat of worst case exponential index size. This worst case behavior is then avoided by resorting back to de Bruijn graphs, but this time indexing paths of a graph rather than string collection. Finally, we explore an approach to directly build graphs from aligned sequences that have indexing properties similar to de Bruijn graphs, but without fixed parameter k for the substring length.

2 Notation

2.1 Strings

Let $S = S[1] \cdots S[n]$ be a string of length $|S|$ over an alphabet Σ of size $|\Sigma|$. If the string and the alphabet are clear from the context, we often denote their sizes as $n = |S|$ and $\sigma = |\Sigma|$. We write substrings as $S[i : j] = S[i] \cdots S[j]$ and call substrings of the form $S[: j] = S[1 : j]$ and $S[i :] = S[i : n]$ prefixes and suffixes, respectively. Let $\$ \in \Sigma$ be the smallest symbol in the alphabet. We often use it as a sentinel $S[n] = \$$ that occurs only at the end of the string.

Let S be a string of length n , let $i \in \mathbb{Z}$ be an offset in it, and let $c \in \Sigma$ be a symbol. We define $S.\text{rank}(i, c)$ as the number of occurrences of symbol c in the prefix $S[1 : i]$. For convenience, we define $S.\text{rank}(i, c) = 0$ when $i < 1$ and $S.\text{rank}(i, c) = S.\text{rank}(n, c)$ when $i > n$.

We define the inverse operation $S.\text{select}(i, c)$ as the position of the occurrence of c of rank i . Let $n_c = S.\text{rank}(n, c)$ be the total number of occurrences of symbol c in the string. For $1 \leq i \leq n_c$, we have $j = S.\text{select}(i, c)$ such that $S.\text{rank}(j, c) = i$ and $S.\text{rank}(j - 1, c) = i - 1$. We also define $S.\text{select}(0, c) = 0$ and $S.\text{select}(n_c + 1, c) = n + 1$ for convenience and leave the function undefined for the remaining values of i .

A *bitvector* is a data structure that stores a binary string and supports efficient rank/select queries. If we store the string as such, we can support both queries in $O(1)$ time and $n + o(n)$ bits of space [14, 37]. When the bitvector is sparse, we can achieve better space usage by using Elias–Fano encoding to store the positions with ones [31, 42]. If there are n_1 ones in a bitvector of length n , we can store it in $n_1 \log(n/n_1) + O(n_1)$ bits of space and support select queries in $O(1)$ time and rank queries in $O(\log(n/n_1))$ time.

2.2 String collections

Let $\mathcal{S} = (S_1, \dots, S_m)$ be an ordered collection of $|\mathcal{S}|$ strings of total length $\|\mathcal{S}\|$. We assume that each string S_i ends with a sentinel $\$_i$ and that $\$_i < \$_j$ iff $i < j$ to make the suffixes of the strings all distinct in lexicographic comparisons. However, we represent the sentinels using the same symbol $\$$ in the actual strings. When the collection is clear from the context,

we use $m = |\mathcal{S}|$ and $N = \|\mathcal{S}\|$ to denote the number of strings and the total length of the strings. When the collection consists of a single string S , we may consider the collection and the string to be the same.

► **Definition 1** (Multiple sequence alignment). *Let $\mathcal{S} = (S_1, \dots, S_m)$ be a collection of strings over alphabet Σ , and let $- \notin \Sigma$ be a gap symbol. Let $\text{spell} : (\Sigma \cup \{-\})^* \rightarrow \Sigma^*$ be a function such that $\text{spell}(S)$ is string S with all occurrences of the gap symbol removed. A multiple sequence alignment of collection \mathcal{S} is a matrix with m rows and n columns $\text{MSA}[1 : m, 1 : n] \in (\Sigma \cup \{-\})^{m \times n}$ such that for each row i , we have $S_i = \text{spell}(\text{MSA}[i, 1 : n])$.*

Let (i, j) and (i', j) be two positions in the same column. If we have $\text{MSA}[i, j] \neq -$ (if $\text{MSA}[i', j] \neq -$), let $S_i[x]$ ($S_{i'}[y]$) be the corresponding symbol. We interpret the alignment in the following way:

- If $\text{MSA}[i, j] = \text{MSA}[i', j] \neq -$, positions $S_i[x]$ and $S_{i'}[y]$ are a *match*.
- If $- \neq \text{MSA}[i, j] \neq \text{MSA}[i', j] \neq -$, positions $S_i[x]$ and $S_{i'}[y]$ are a *mismatch*.
- If $\text{MSA}[i, j] \neq -$ and $\text{MSA}[i', j] = -$, string S_i has an *insertion* relative to string $S_{i'}$ and string $S_{i'}$ has a *deletion* relative to string S_i at the position.

Recombinations let us build new strings by breaking existing strings in two at aligned positions and concatenating the prefix of one string with the suffix of another. Using the above notation, if positions $S_i[x]$ and $S_{i'}[y]$ are a match, we can recombine strings S_i and $S_{i'}$ at those positions as $S_i[:x-1]S_{i'}[y:]$ and $S_{i'}[:y-1]S_i[x:]$. This corresponds to switching between rows i and i' in column j of the multiple sequence alignment.

In later sections, we discuss data structures for indexing all possible recombinations of a string collection compatible with a given alignment. We formalize this as follows.

► **Definition 2** (Closure under recombinations). *Let \mathcal{S} be a collection of strings and let \sim be an equivalence relation for its suffixes such that if $S[j:] \sim S'[j']:$, we have $S[j] = S'[j']$. The closure of collection \mathcal{S} under recombinations is a collection \mathcal{S}^\sim such that:*

1. $\mathcal{S} \subseteq \mathcal{S}^\sim$
2. For any two strings $S, S' \in \mathcal{S}^\sim$ and positions j, j' such that $S[j:] \sim S'[j']:$, it holds that $S'' = S[:j-1]S'[j']:] \in \mathcal{S}^\sim$.
3. Given a string $S'' \in \mathcal{S}^\sim$ as above, $S''[j'']:] \sim S[j'']:]$ if $j'' < j$ and $S''[j'']:] \sim S'[j''-j+j']:]$ if $j'' \geq j$.

According to rule 3, each suffix $S''[j'']:]$ of the recombined string S'' inherits the alignment of the suffix starting from the corresponding position in the source strings S and S' .

2.3 Arrays over sorted suffixes

Let \mathcal{S} be an ordered collection of strings. We define the (generalized) *suffix array* of collection \mathcal{S} as an array of pointers $\text{SA}[1 : N]$ to the suffixes of the strings in lexicographic order. Each pointer $\text{SA}[x] = (i, j)$ refers to the suffix $S_i[j:]$ of string S_i . Given two pointers $\text{SA}[x] = (i, j)$ and $\text{SA}[x'] = (i', j')$, we have $x < x'$ iff $S_i[j:] < S_{i'}[j']:$ in lexicographic order. We therefore call x the *lexicographic rank* of suffix $S_i[j:]$ among the suffixes of collection \mathcal{S} . Note that for $1 \leq x \leq m$, we have $\text{SA}[x] = (x, |S_x|)$.

The multi-string *Burrows–Wheeler transform* (BWT) [11] of collection \mathcal{S} is a string $\text{BWT}[1 : N]$ representing a permutation of the symbols in strings $S \in \mathcal{S}$. We define it using the suffix array. If $\text{SA}[x] = (i, j)$ and $j > 1$, the BWT stores the preceding symbol $\text{BWT}[x] = S_i[j-1]$. When $j = 1$, we set $\text{BWT}[x] = \$$.

In addition to the suffix array and the Burrows–Wheeler transform, we often define other arrays over the lexicographically sorted suffixes. The *longest common prefix (LCP) array* $\text{LCP}[1 : N]$ stores the length of the longest common prefix of two lexicographically adjacent suffixes. If $\text{SA}[x] = (i, j)$ and $\text{SA}[x - 1] = (i', j')$, we store the LCP of suffixes $S_i[j :]$ and $S_{i'}[j' :]$ in $\text{LCP}[x]$, with $\text{LCP}[1] = 0$.

2.4 FM-index

The Burrows–Wheeler transform can be inverted using a function called *LF-mapping* that maps the lexicographic rank of a suffix to the lexicographic rank of its cyclic predecessor. We define it in two parts. Let $\text{SA}[x] = (i, j)$. If $j > 1$, we define $\text{LF}(x) = y$ such that $\text{SA}[y] = (i, j - 1)$. We sometimes write this as $\text{SA}[\text{LF}(x)] = \text{SA}[x] - 1$. For $j = 1$, we define $\text{LF}(x) = y$ such that $\text{SA}[y] = (i, |S_i|)$.

We can compute the LF-mapping for most values of x using the BWT. Let $\text{C}[1 : \sigma]$ be an array such that $\text{C}[c]$ is the total number of occurrences of all symbols $c' < c$ in collection \mathcal{S} . Let $\text{SA}[x] = (i, j)$. If $j > 1$, which is equivalent to $\text{BWT}[x] \neq \$$, we have $\text{LF}(x) = \text{C}[\text{BWT}[x]] + \text{BWT.rank}(x, \text{BWT}[x])$. Here $\text{C}[\text{BWT}[x]]$ is the number of suffixes starting with a symbol smaller than $\text{BWT}[x] = S_i[j - 1]$ and $\text{BWT.rank}(x, \text{BWT}[x])$ is the lexicographic rank of suffix $S_i[j - 1 :]$ among the suffixes starting with symbol $\text{BWT}[x]$.

Inverse LF-mapping, denoted as Ψ , can be computed as $\Psi(x) = \text{LF}^{-1}(x)$ by identifying a symbol c such that $\text{C}[c] < x \leq \text{C}[c + 1]$ and then applying $\Psi(x) = \text{BWT.select}(x - \text{C}[c], c)$.

We can generalize LF-mapping for hypothetical suffixes. Let S be the lexicographic rank of string X among the suffixes of collection \mathcal{S} . That is, x is the number of suffixes Y of the collection such that $Y \leq X$. Then, given a symbol $c \in \Sigma$ with $c \neq \$$, we define the generalized LF-mapping as the lexicographic rank of string cX . We can compute it as $\text{LF}(x, c) = \text{C}[c] + \text{BWT.rank}(x, c)$.

Let $\text{find}(X) = [sp : ep]$ be the *lexicographic range* of suffixes starting with string X . That is, for each $x \in [sp : ep]$, we have $\text{SA}[x] = (i, j)$ such that $S_i[j : j + |X| - 1] = X$. Then for any symbol c , the range of suffixes starting with cX is $\text{find}(cX) = [\text{LF}(sp - 1, c) + 1 : \text{LF}(ep, c)]$. This backward search step forms the core of the *FM-index* [22]. Given a pattern P , we can determine the range of suffixes $\text{find}(P)$ starting with P with $O(|P|)$ rank operations on the BWT.

Another core operation is $\text{locate}(x) = \text{SA}[x] = (i, j)$, which converts the lexicographic rank of a suffix to its starting position. To compute it, the FM-index needs *suffix array samples* in addition to the BWT. Let $B[1 : N]$ be a bitvector that marks the sampled suffix array positions with 1, and let S be the subsequence of the suffix array corresponding to those positions. If $B[x] = 1$, we can compute $\text{locate}(x)$ as $S[B.\text{rank}(x, 1)]$. Otherwise we rely on iterated LF-mapping and the fact that $\text{SA}[\text{LF}(x)] = \text{SA}[x] - 1$ in the general case. If we need k steps to find the nearest sampled position, we have $\text{locate}(x) = \text{locate}(\text{LF}^k(x)) + k$. For this scheme to function, we need to sample $\text{SA}[x] = (i, j)$ whenever $j = 1$. If we sample $\text{SA}[x]$ every time $j \equiv 1 \pmod{s}$ for a parameter $s > 0$, we can compute $\text{locate}(x)$ with $O(s)$ rank operations.

2.5 Graphs

Let $G = (V, E)$ be a graph with a finite set of vertices V and a set of edges $E \subseteq V \times V$. We assume that the graph is directed. For any two vertices $u, v \in V$, with $u \neq v$, edges (u, v) and (v, u) are distinct. A path is a string $P[1 : n]$ of vertices such that $(P[i], P[i + 1]) \in E$ for $1 \leq i < n$. We call the graph acyclic if there is no path that visits a vertex $v \in V$ more than once.

A *vertex-labeled graph* is a tuple $G = (V, E, \Sigma, \ell)$, where (V, E) is a graph, Σ is an alphabet, and $\ell : V \rightarrow \Sigma$ is a function that assigns a label $\ell(v) \in \Sigma$ to each vertex $v \in V$. If P is a path, we write $\ell(P) = \ell(P[1]) \cdots \ell(P[n])$ to refer to the string formed by concatenating the vertex labels. We can similarly define *edge-labeled graphs* as tuples $G = (V, E, \Sigma, \ell)$, where $\ell : E \rightarrow \Sigma$ is a function that assigns labels to edges. Later, we extend the notions to vertices and edges labeled with non-empty strings.

► **Definition 3** (Alignment graph). *Let \mathcal{S} be a collection of strings over alphabet Σ , and let \sim be an equivalence relation for its suffixes such that if $S[j:] \sim S'[j']:$, we have $S[j] = S'[j']$. The alignment graph of collection \mathcal{S} with respect to \sim is a vertex-labeled graph $G^\sim = (V, E, \Sigma, \ell)$ such that:*

1. *The set of vertices V is the set of equivalence classes for relation \sim .*
2. *For any two vertices $u, v \in V$, we have $(u, v) \in E$ if and only if there is a string $S \in \mathcal{S}$ and a position j such that $S[j:] \in u$ and $S[j+1:] \in v$.*
3. *The label of vertex $v \in V$ is the shared first symbol $\ell(v) = S[j]$, where $S[j:] \in v$.*

Note that the set of labels $\ell(P)$ for paths P in the alignment graph is the set of substrings of strings $S \in \mathcal{S}^\sim$. If \sim is an equivalence relation based on a multiple sequence alignment such that $S[j:] \sim S'[j']:$ when the corresponding positions match, the resulting alignment graph is always acyclic.

3 XBWT: Indexing Labeled Trees

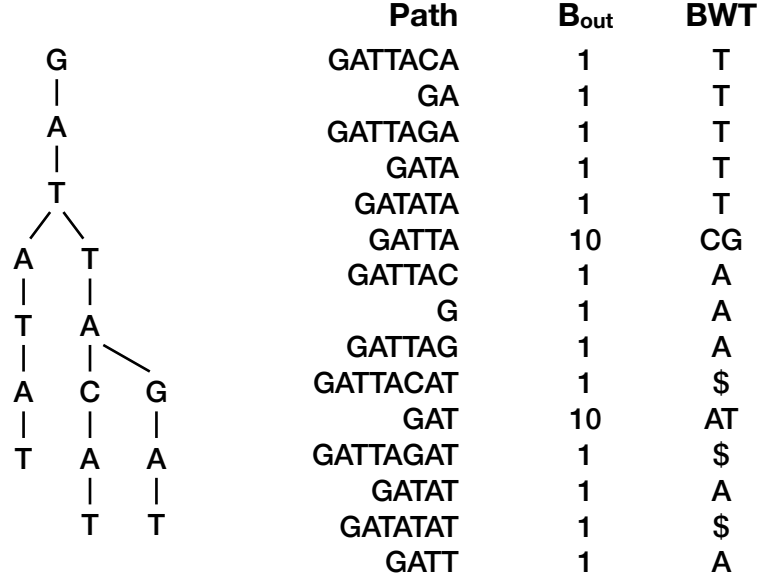
The Burrows–Wheeler transform was originally defined for a single string. Extending the definition to an ordered collection of strings is straightforward enough. But a collection of strings is still a linear structure, while paths in a graph can both diverge and converge. The first step towards using the BWT for graph indexing was the *extended Burrows–Wheeler transform (XBWT)* [21], which can index labeled trees. The key technique was using select queries on a bitvector for expanding a (co-)lexicographic range $[sp : ep]$ of vertices to a (co-)lexicographic range $[sp_{out} : ep_{out}]$ of outgoing edges.

In this section, we present a variant of the XBWT that is similar to the indexes for vertex-labeled graphs discussed in Sections 4 and 5. There are other fundamentally equivalent representations. The representations differ in their handling of leaf vertices and outdegrees, on whether they consider vertex-labeled or edge-labeled trees, and on whether the search proceeds from the root towards the leaves or the other way around. Some of them achieve better space bounds than what is presented here.

3.1 Data structure

A *vertex-labeled tree* with root $v_r \in V$ is a vertex-labeled graph $T = (V, E, \Sigma, \ell)$ such that the indegree of the root is 0 and the indegree of every other vertex $v \in V$ is 1, with a unique path P_v from the root to every vertex $v \in V$. Given an edge $(u, v) \in E$, we call vertex u the parent of vertex v and v a child of u . We call vertices with outdegree 0 the leaves of the tree.

XBWT uses vertices in place of suffixes. The vertices are sorted by the label $\ell(P_v)$ in colexicographic order (in lexicographic order by reverse path labels). For a given vertex $u \in V$, the BWT stores the concatenation of the labels $\ell(v)$ for each outgoing edge $(u, v) \in E$. We encode the outdegree d in unary as a one followed by $d - 1$ zeros and concatenate the outdegrees in the same order to form bitvector B_{out} . In order to handle the leaves, we add a technical vertex $v_\$ \notin V$ such that $\ell(v_\$) = \$$. For each leaf v , we encode an edge $(v, v_\$) \notin E$. As with strings, we assume that $\ell(v) \neq \$$ for every vertex $v \in V$. If



■ **Figure 1** XBWT for the trie of strings GATATAT, GATTACAT, and GATTAGAT. The XBWT consists of the bitvector B_{out} and the string BWT.

the colexicographic rank of vertex $v \in V$ is x , the labels of its children are stored in $BWT[B_{out}.select(x, 1) : B_{out}.select(x + 1, 1) - 1]$. See Figure 1 for an example. With an uncompressed wavelet tree [29] for the BWT and an uncompressed bitvector for B_{out} , the XBWT data structure requires $O(|V| \log \sigma)$ bits of space, not including the C array.

Let $find(X) = [sp : ep]$ be the colexicographic range of vertices $v \in V$ with path labels $\ell(P_v)$ ending with string X . Given a symbol $c \in \Sigma$, we want to find the range $find(Xc)$ of vertices with path labels ending with Xc . We first expand the range into a colexicographic range of outgoing edges $[sp_{out} : ep_{out}] = [B_{out}.select(sp, 1) : B_{out}.select(ep + 1, 1) - 1]$. Every edge $(u, v) \in E$ in that range is encoded with the label $\ell(v)$ of the child vertex, and the colexicographic rank of that vertex is based on path label $\ell(P_v) = \ell(P_u)\ell(v)$. Hence we can use a backward search step (Section 2.4) to get the range of children with label c : $find(Xc) = [LF(sp_{out} - 1, c) + 1 : LF(ep_{out}, c)]$. Note that the backward search step becomes a forward search step, as the XBWT is based on colexicographic order. Given a string X , we can therefore compute $find(X)$ using $O(|X|)$ rank operations on BWT and $O(|X|)$ select operations on B_{out} .

If we know the colexicographic rank x_r of the root v_r , we can match entire path labels instead of suffixes by starting the above search from range $[x_r : x_r]$. We can also use the XBWT for tree navigation. Given a vertex $v \in V$ with colexicographic rank x , the range of its children is $[B_{out}.select(x, 1) : B_{out}.select(x + 1, 1) - 1]$. The colexicographic rank of its i -th child is $LF(B_{out}.select(x, 1) + i - 1)$ and the rank of its parent is $B_{out}.rank(\Psi(x), 1)$.

3.2 Applications

The initial XBWT paper proposed using the structure as a fast and space-efficient representation for hierarchical documents, such as XML [21]. In such applications, it is important that a vertex can have multiple children with the same label (such as list items). When storing string collections (often called string dictionaries in this context), we can collapse the tree into

a trie, improving space usage. Compared to other representations for string collections, the XBWT is smaller but slower [36]. The slowness comes from symbol-to-symbol navigation and the poor memory locality of iterated LF-mapping. However, the same FM-index functionality enables substring searches not supported by faster representations based on hashing or front coding.

The Aho–Corasick algorithm can find the occurrences of multiple patterns \mathcal{P} in a string S in linear time $O(|S| + \|\mathcal{P}\| + occ)$, where occ is the number of occurrences. It is based on a trie $T = (V, E, \Sigma, \ell)$ of the patterns augmented with failure links (suffix links). A failure link from vertex $v \in V$ points to a vertex $u \in V$ such that $\ell(P_u)$ is the longest proper suffix of $\ell(P_v)$ among all vertices $u \in V$. The XBWT can be augmented with a $2|V|(1 + o(1))$ -bit balanced parentheses representation to support failure links, making it a space-efficient Aho–Corasick automaton [35]. Here the relative slowness matters less, as the algorithm is based on symbol-to-symbol navigation anyway.

Because the XBWT represents a collection of strings, it can be built from other representations, such as the suffix array and the Burrows–Wheeler transform [35]. The link also works in the other direction: we can go from the XBWT directly to a run-length encoded multi-string BWT [12]. As a byproduct, we can efficiently find the order of the strings that minimizes the number of runs in the BWT [12]. Because the order of the strings determines how well the suffixes close to the end of the string can be run-length encoded, this can be valuable with highly repetitive collections of short strings.

4 SBWT: Indexing k -Spectra

The k -spectrum of a string S is the set of distinct k -mers occurring in S . In this section we give an overview of how k -spectra can be indexed for k -mer lookup queries via the Spectral Burrows–Wheeler transform (SBWT) [5]. Given a query k -mer, a k -mer lookup query returns the colexicographic rank of the k -mer in the k -spectrum. It is not difficult to see that a structure for k -mer lookup queries gives a convenient representation of a de Bruijn graph.

The SBWT is a distillation of the ideas found in the BOSS [10] and Wheeler graph data structures. It is indeed an evolution of the BOSS data structure, which is an indexed representation of the edge-centric de Bruijn graph. The SBWT differs from the BOSS being vertex-centric, and more general – the SBWT is a mathematical transformation which can be indexed in various ways, one of which closely resembles the BOSS data structure. In addition, the SBWT graph is also a Wheeler graph [25] covered in the next chapter of this book. This implies that the generic Wheeler graph index could be used to index the graph. However, that index requires storage of the sequence of indegrees and outdegrees of the vertices in the graph, whereas in the SBWT this is not required because every vertex apart from the vertex of the empty string has indegree of exactly 1, and the sequence of outdegrees is already included in the sizes of the sets of the outgoing edge label sets. Thus, the SBWT can be seen as a specialization of the Wheeler graph framework to k -spectra, taking advantage of the properties of the special case.

4.1 The Spectral Burrows–Wheeler Transform

In this section we define the Spectral Burrows–Wheeler transform, SBWT. We begin with few basic definitions:

► **Definition 4** (k -spectrum). *A string's k -spectrum, denoted with $S_k(S)$, consists of all unique k -mers present in the string: $\{S[i : i + k - 1] \mid i = 1, \dots, |S| - k + 1\}$. The k -spectrum $S_k(S_1, \dots, S_m)$ of a set of strings S_1, \dots, S_m is the union $\bigcup_{i=1}^m S_k(S_i)$.*

padded k -spectrum adds a minimal set of $\$$ -padded *dummy* k -mers to ensure that in the de Bruijn graph (to be defined shortly), every non-dummy k -mer has an incoming path of length at least k :

► **Definition 5** (Source vertex set). *Let $R = S_k(S_1, \dots, S_m)$ be a k -spectrum, the source vertex set R' is a subset of R containing k -mers Y such that $Y[1 : k - 1]$ is not a suffix of any k -mer in R .*

► **Definition 6** (Padded k -spectrum). *Let R be a k -spectrum of $\mathcal{S} = (S_1, \dots, S_m)$, and let $R' \subseteq R$ be the source vertex set of \mathcal{S} for a given k . The padded k -spectrum is the set $S_k^+(S_1, \dots, S_m) = R \cup \{\$^k\} \cup \bigcup_{Y \in R'} \{\$^{k-i} Y[1 : i] \mid i = 1, \dots, k - 1\}$, where $\$$ is a special character of Σ , which is smaller than all other characters of the alphabet.*

The number of dummy vertices can be calculated with a simple formula. It is indeed equal to the number of internal vertices of $\mathcal{T}(R')$, where $\mathcal{T}(R')$ is the trie of the source vertex set of a set of strings.

For example, if we consider two strings $S_1 = \text{ACAGTG}$ and $S_2 = \text{ATCAGA}$, and $k = 3$, then $R = S_3(S_1, S_2) = \{\text{ACA}, \text{CAG}, \text{AGT}, \text{GTG}, \text{ATC}, \text{TCA}, \text{AGA}\}$, $R' = \{\text{ACA}, \text{ATC}\}$, and the padded k -spectrum $S_3^+(S_1, S_2) = S_3(S_1, S_2) \cup \{\$\$\$, \$\$A, \$AC, \$AT\}$.

► **Definition 7** (de Bruijn graph). *The de Bruijn graph of a set of strings $\mathcal{S} = (S_1, \dots, S_m)$ is an edge-labeled graph $G = (V, E, \Sigma, \ell, k)$, where:*

1. *The set of vertices V is the set of k -mers from the padded k -spectrum of \mathcal{S} , in symbols $V = S_k^+(S_1, \dots, S_m)$.*
2. *For any two vertices, k -mers, $u, v \in V$, we have $(u, v) \in E$ if and only if $u[2 : k] = v[1 : k - 1]$.*
3. *The label of an edge $e = (u, v) \in E$, denoted with $\ell(e)$ or $\ell(u, v)$, is the character $v[k]$. $\$$ never appears as an edge label.*

See figure 2 for an illustration of a de Bruijn graph.

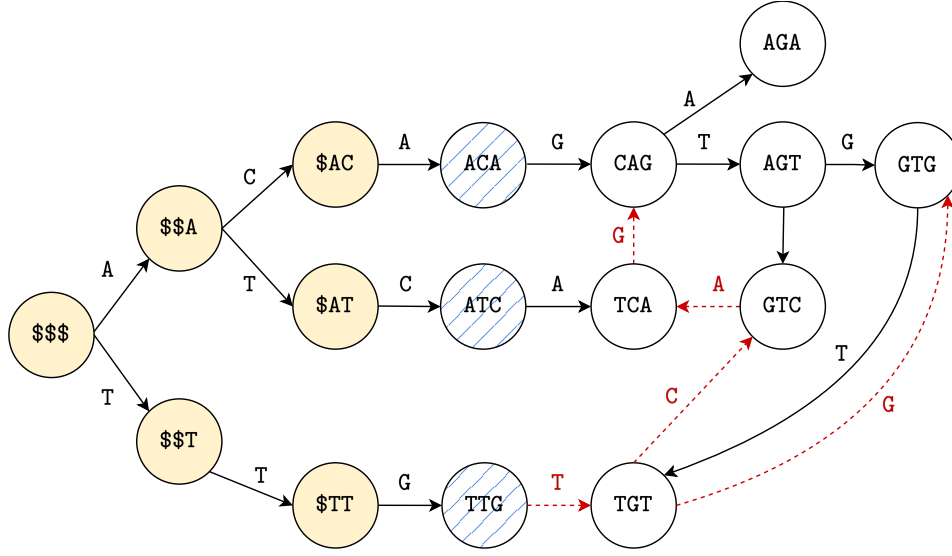
We can now define the Spectral BWT.

► **Definition 8** (Spectral Burrows-Wheeler transform (SBWT) [5]). *Let R^+ be a padded k -spectrum and let $X_1, \dots, X_{|R^+|}$ be the elements of R^+ in colexicographic order. The SBWT is the sequence of sets $A_1, \dots, A_{|R^+|}$ with $A_i \subseteq \Sigma$ such that $A_i = \emptyset$ if $i > 1$ and $X_i[2 : k] = X_{i-1}[2 : k]$, otherwise $A_i = \{c \in \Sigma \mid X_i[2k]c \in R^+\}$.*

Continuing the example above and adding a new sequence:

$\mathcal{S} = \{\text{ACAGTG}, \text{ATCAGA}, \text{TTGTCAGTGT}\}$. The colexicographically ordered list of $S_3^+(\mathcal{S})$ is: $\$\$\$, \$\$A, \text{ACA}, \text{TCA}, \text{AGA}, \$AC, \text{ATC}, \text{GTC}, \text{CAG}, \text{GTG}, \text{TTG}, \$\$T, \$AT, \text{AGT}, \text{TGT}, \TT , and the SBWT of \mathcal{S} is the sequence of sets: $\{\text{A}, \text{T}\}, \{\text{C}, \text{T}\}, \{\text{G}\}, \emptyset, \emptyset, \{\text{A}\}, \{\text{A}\}, \emptyset, \{\text{A}, \text{T}\}, \{\text{T}\}, \emptyset, \{\text{T}\}, \{\text{C}\}, \{\text{C}, \text{G}\}, \emptyset, \{\text{G}\}$.

The sets in the SBWT represent the labels of outgoing edges in the vertex-centric de Bruijn graph, such that we only include outgoing edges from k -mers that have a different suffix of length $k - 1$ than the preceeding k -mer in the colexicographically sorted list. The padding of dollar-symbols is a technical detail required to make the SBWT work. It ensures that every k -mer of the k -spectrum has an incoming path of length at least k in the de Bruijn graph. Alternatively, the sequences of the set \mathcal{S} on which the SBWT is built could be made cyclic, thus removing the need for the padded k -spectrum $S_k^+(\mathcal{S})$.



■ **Figure 2** The de Bruijn graph of $S = \{ACAGTG, ATCAGA, TTGTCAGTGT\}$, $k = 3$. The *source vertex set*, k -mers with no predecessor in S , is striped, and the solid-colored vertices represent the dummy vertex set. Dashed edges are pruned from the graph as the vertices they point to can be reached from other (solid) edges.

We now describe how to implement efficient k -mer membership queries on the spectrum of the input strings, using only the information encoded in the SBWT. It is helpful to think of the SBWT as encoding the set R^+ of k -mers as a list arranged in colexicographical order (as in the example above). In the context of k -mer lookup, the SBWT allows navigation of this ordered list via a search routine based on *subset rank queries*.

► **Definition 9** (Subset rank query). Let X_1, \dots, X_n be a sequence of subsets of an alphabet $\Sigma = \{1, \dots, \sigma\}$. A *subset rank query* takes as input an index i and a character $c \in \Sigma$, and returns the number of subsets X_j with $j \leq i$ such that $c \in X_j$.

The search routine, introduced below, works by searching the k -mer character by character from left to right. It does so maintaining the interval of k -mers in the colexicographically-sorted list that are suffixed by the prefix which has been processed so far.

Since the k -mers in R^+ are colexicographically sorted in the SBWT, the subset of k -mers in R^+ that share a string α as a suffix are next to each other, and they are called the *colexicographic interval* of α . The colexicographic interval of a string α longer than k is defined as the range of k -mers in the sorted list of R^+ suffixed by the last k characters of α . Now let $[s : e]_\alpha$ denote the *colexicographic interval* of the string α , where s and e are respectively the colexicographic ranks of the smallest and largest k -mer in the SBWT having substring α as a suffix. Finding the interval $[s' : e']$ of αc , where c is a character from $\Sigma - \{\$ \}$, is equivalent to following all edges labeled with c from the vertices in $[s : e]$. Due to the way the edges are defined, the end points of these edges are a contiguous range $[s' : e']$ such that s' is the destination of the first outgoing edge labeled c from $[s : e]$, and e' is the destination of the last one. Let $C[c]$ be the number of edge labels with label smaller than c in the graph. We now have the following formulas:

$$\begin{aligned} s' &= 1 + C[c] + \text{subsetrank}_c(s - 1) + 1 \\ e' &= 1 + C[c] + \text{subsetrank}_c(e) \end{aligned} \tag{1}$$

\$\$\$	\$\$A	ACA	TCA	AGA	\$AC	ATC	GTC	CAG	GTG	TTG	\$\$T	\$AT	AGT	TGT	\$TT
1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1
1	1	0	0	0	0	0	0	1	1	0	1	0	0	0	0

■ **Figure 3** MatrixSBWT of $\mathcal{S} = \{\text{ACAGTG}, \text{ATCAGA}, \text{TTGTCAGTGT}\}$ with $k = 3$. The dashed lines indicate borders of suffix groups. Two adjacent columns are in the same group if they share the same suffix of length $k - 1$. Bits may be moved horizontally inside a suffix group without affecting the k -mer set encoded in the matrix.

where subsetrank_c is a subset rank query on the SBWT sequence. The “+1” at the start of the formulas is to skip over the vertex of $\k . The values $C[c]$ can be precomputed for all characters $c \in \Sigma$. By iterating these formulas k times, we have the k -mer search routine.

The simplest data structures for subset rank queries is the plain matrix. See Figure 3 for an example.

► **Definition 10** (Plain Matrix representation). *The plain matrix representation of the SBWT sequence is a $|\Sigma - \{\$\}| \times n$ binary matrix M , such that $M[i][j] = 1$ iff subset X_j includes the i^{th} character in the alphabet.*

The rows of the matrix M are indexed for constant time rank queries [37] (implementation from [28]). A single rank query on row i at index j in M would answer in constant time $\text{subsetrank}_c(j)$, where c is the i^{th} character of the alphabet. There exist alternative succinct data structures for subset rank, such as the *subset wavelet tree*, details of which can be found in [4, 5]. A concatenation of the sets indexed as a wavelet tree leads to something resembling the BOSS data structure. All k -mers of a longer string can be searched efficiently by adding the LCS array of the SBWT [3]. At the end of each single k -mer search, the LCS array allows to extend the colexicographic interval to the last $k - 1$ characters observed in the previous k -mer.

5 GCSA: Indexing Finite Languages

The *generalized compressed suffix array (GCSA)* [48, 49] is a generalization of the FM-index to graphs. It was originally designed for indexing acyclic vertex-labeled graphs (or equivalently finite languages) that arise from aligned DNA sequences, using techniques similar to the XBWT (Section 3).

GCSA can be built for any such graph, and it supports efficient queries in it, but the index can be exponentially larger than the graph. Some other BWT-based graph indexes, such as BWBBLE [32] and vBWT [34], take the opposite approach. By encoding the graph as a string, they make index construction fast and the index itself small. Queries can be slower than in GCSA, as they often require branching in the graph.

BOSS [10] represents de Bruijn graphs using a data structure equivalent to the GCSA. This equivalence led to a realization that the GCSA approach works with any graph that is sufficiently similar to a de Bruijn graph. Such graphs can recognize a strict subclass of regular languages [49]. GCSA2 [46] uses this realization in a practical way to index any vertex-labeled graph by approximating it with a high-order (pruned) de Bruijn graph.

GCSA requires that the vertices of the graph can be sorted unambiguously by the labels of maximal paths starting from them. This is a global property that is sometimes difficult to reason about. Wheeler graphs [25] replaced this requirement with an essentially equivalent local property. Vertices are sorted by their labels, and ties are broken by inheriting the order of the successor vertices. (The original formulation used edge-labeled graphs and inherited the order from the predecessors.) This alternate characterization made reasoning about Wheeler graphs easier in many situations.

5.1 Structural runs in the BWT

Consider a collection that consists of m identical copies of the same string S . Then for any position j , the suffixes starting at that position are consecutive in lexicographic order. Because the preceding symbol $S[j-1]$ (if $j > 1$; otherwise $S[n] = \$$) is also the same for every string, the suffixes are in the same run of symbol $S[j-1]$ in the BWT of the collection. Hence the BWT of a collection of m identical strings always has the same number of runs, regardless of the number of copies. We now consider what happens when we apply edits to a collection of identical strings.

► **Definition 11** (Aligned suffixes). *Let $\mathcal{S} = (S_1, \dots, S_m)$ be a collection of strings, and let MSA be a multiple sequence alignment of the collection. For any two strings $S_i, S_{i'} \in \mathcal{S}$ and positions j, j' in them, suffixes $S_i[j:]$ and $S_{i'}[j':]$ are aligned if and only if positions $S_i[j]$ and $S_{i'}[j']$ are a match in MSA. We represent this with equivalence relation \sim_0*

► **Definition 12** (Structural BWT runs). *Let $\mathcal{S} = (S_1, \dots, S_m)$ be a collection of strings, let \sim_0 be an alignment of the suffixes, and let BWT and SA be the Burrows–Wheeler transform and the suffix array of the collection. A structural BWT run is a maximal interval $[sp : ep]$ such that for any $x, x' \in [sp : ep]$:*

1. $S_i[j:] \sim_0 S_{i'}[j':]$, where $SA[x] = (i, j)$ and $SA[x'] = (i', j')$; and
2. $BWT[x] = BWT[x']$.

We use equivalence relation \sim_0^b to denote that suffixes are in the same structural BWT run.

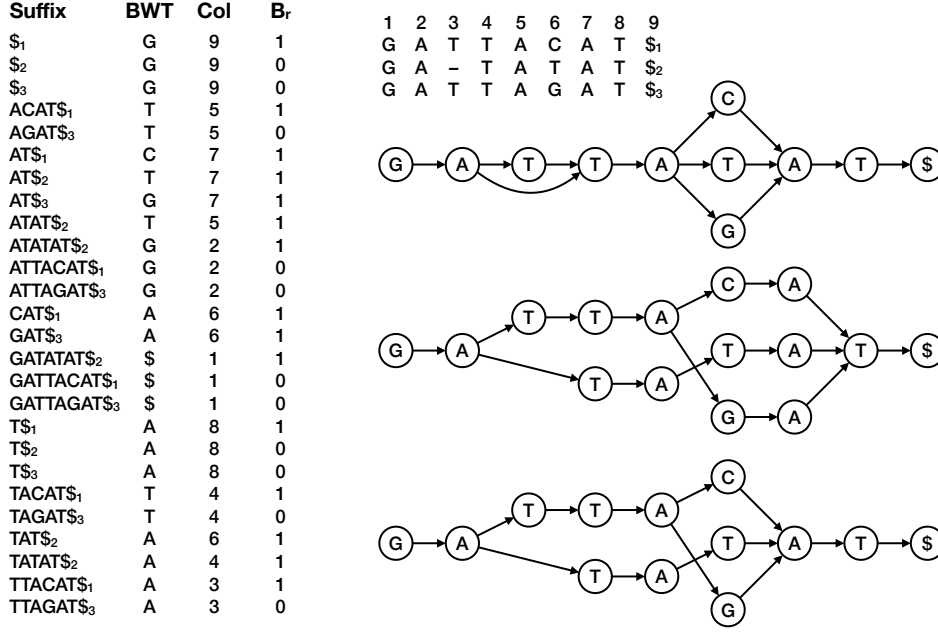
Let $\mathcal{S} = (S_1, \dots, S_m)$ be a *highly repetitive* collection of similar strings. We assume that there is a sequence of s edit operations that transforms a collection of m copies of string S into \mathcal{S} , with $s \ll mn$. We further assume that there is a multiple sequence alignment corresponding to the edit operations, where suffixes that have been derived from the same suffix of the original string S start with a match or a mismatch in the same column. Note that different sequences of edit operations leading to the same collection may correspond to different alignments.

Initially, there are n structural runs, one for each position in string S . Each edit operation that changes a substring $S_i[j : j']$ can affect the suffixes of S_i in three ways:

1. The symbol in the BWT can change for the suffixes starting from $S_i[j+1]$ to $S_i[j'+1]$.
2. Suffixes starting from $S_i[j-k]$ to $S_i[j']$, for some $k \geq 0$, may leave their current structural runs due to changes in lexicographic ranks.
3. New unaligned suffixes can be created and existing suffixes may be removed if substring $S_i[j : j']$ is replaced with a string of different length.

Every affected suffix can increase the number of structural runs by at most two: by becoming a new run and by splitting an existing run in two.

Mäkinen et al. [38] showed that if the original string S is random and we apply random edits, k is at most $O(\log_\sigma N)$ in the expected case. If we assume single-symbol edits, s edit operations increase the number of structural runs by at most $O(s \log_\sigma N)$ in the expected



■ **Figure 4** Structural runs in the BWT of strings $S_1 = \text{GATTACAT}\$, S_2 = \text{GATATAT}\$, and $S_3 = \text{GATTAGAT}\$$. The table lists the suffixes in lexicographic order, the BWT symbol for each suffix, and the starting position of the suffix in the MSA (also shown). Bitvector B_r marks positions at the start of each structural BWT run. The graphs from top to bottom are: $G^{\sim 0}$ representing the MSA; $G^{\sim b}$ representing structural BWT runs; and $G^{\sim s}$ representing structural SA runs (see Section 5.4).$

case. Because structural runs are (possibly non-maximal) BWT runs, there are at most $r + O(s \log_\sigma N)$ runs in the BWT of collection \mathcal{S} in the expected case. Here r is the number of runs in the BWT of string S .

5.2 Indexing the graph of structural runs

Let \mathcal{S} be a string collection and \sim_0^b an equivalence relation that denotes that two suffixes are in the same structural BWT run. Let $G^{\sim b} = (V, E, \Sigma, \ell)$ be the alignment graph of the collection with respect to \sim_0^b .

Graphs like this have been used as space-efficient representations of the collection. Both SimDNA [33] and FM-index of alignment [40, 39] index the collection using a graph based on a specific type of alignment. The alignment alternates between common regions, where all strings are aligned, and non-common regions, where they diverge. BWT tunneling [7] also uses a similar graph to improve BWT-based data compression by not storing run lengths within unary paths. Our goal is instead to extend the FM-index of the collection to index all strings in the closure $\mathcal{S}^{\sim b}$ using the graph.

We could augment the FM-index with a bitvector $B_r[1 : N]$ that marks the start (or equivalently the end) of each structural run with ones. After each backward search step, we use some rank/select operations on B_r to extend the interval $[sp : ep]$ to the smallest interval $[sp' : ep']$ that contains it and consists of entire structural runs. If we continue the search from interval $[sp' : ep']$, we are effectively accepting recombinations within each vertex in the range. No matter which suffixes we used to reach a vertex, we can continue the search to the predecessor of any suffix in the vertex. See Figure 4 for an example.

5.3 GCSA structure

Because we discard the information on how we arrived to the current vertex, we no longer need to keep track of the exact BWT interval. Hence we choose the GCSA representation [48] (Figure 5) that corresponds more closely to the structure of the graph. We keep the vertices in the same order as the structural runs. Let $\text{rank}_V : V \rightarrow [1 : |V|]$ be a function that maps a vertex $v \in V$ to its lexicographic rank $\text{rank}_V(v)$. Lexicographic range $[sp : ep]$ now corresponds to vertices $v \in V$ such that $\text{rank}_V(v) \in [sp : ep]$.

The data structure consists of three components B_{in} , BWT, and B_{out} . We use them all for computing a backward search step from $\text{find}(X) = [sp : ep]$ to $\text{find}(cX) = [sp' : ep']$ for a string X and a symbol c .

1. Bitvector B_{in} encodes the indegree of each vertex in unary. Similar to using B_{out} in XBWT (Section 3.1), we map the range of vertices to a range of incoming edges as $[sp_{in} : ep_{in}] = [B_{in}.\text{select}(sp, 1) : B_{in}.\text{select}(ep + 1, 1) - 1]$.
2. The BWT range for vertex $v \in V$ is the concatenation of $\ell(u)$ for each vertex $u \in V$ with $(u, v) \in E$ in an arbitrary order. A backward search step with this BWT maps the range $[sp_{in} : ep_{in}]$ of incoming edges to a range $[sp_{out} : ep_{out}] = [\text{LF}(sp_{in} - 1, c) + 1 : \text{LF}(ep_{in}, c)]$ of outgoing edges, ordered by the predecessor vertex.
3. Bitvector B_{out} encodes the outdegree of each vertex in unary. With rank queries, we get the interval of the predecessors of the vertices in $[sp : ep]$ with label c as $[sp' : ep'] = [B_{out}.\text{rank}(sp_{out}, 1) : B_{out}.\text{rank}(ep_{out}, 1)]$.

Together these take $|E|(\log \sigma + 2)(1 + o(1))$ bits, assuming an uncompressed wavelet tree for the BWT and uncompressed bitvectors and excluding the C array. To handle the edge cases with source/sink vertices, we add technical edges connecting vertices with outdegree 0 to vertices with indegree 0.

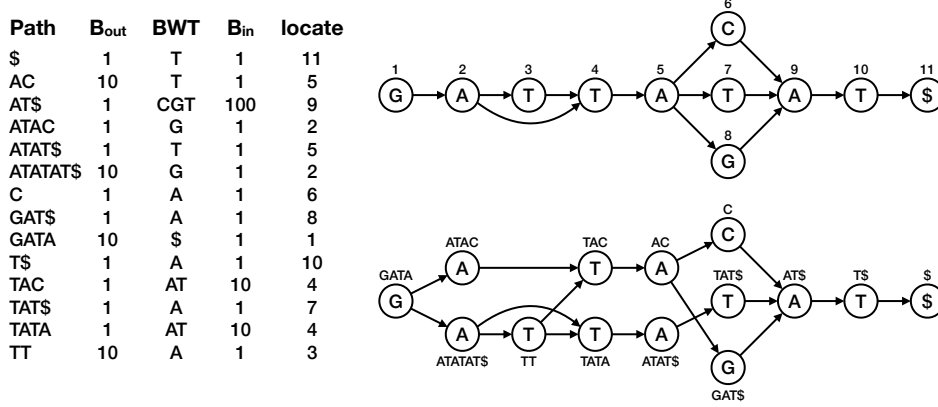
This data structure also facilitates graph navigation. For instance, if $v \in V$ is a vertex and $x = \text{rank}_V(v)$, we can navigate backward along a unary path from v . This is effectively the same as using $\text{LF}(x)$ to navigate from a suffix to its predecessor. The rank of the first incoming edge to v is $x_{in} = B_{in}.\text{select}(x, 1)$. If $B_{in}[x_{in} + 1] = 0$, vertex v has multiple predecessors and the path is no longer unary. With $x_{out} = \text{LF}(x_{in})$, we get the rank of the corresponding outgoing edge, and $B_{out}.\text{rank}(x_{out}, 1)$ is the rank of the predecessor vertex. Navigation like this is used in locate queries (Section 5.5).

The alignment graph based on structural runs allowed us to index the original strings as well as their recombinations within the runs. Our ultimate goal is allowing recombinations at any aligned position. In principle, we already have the necessary tools for it. Because \sim_0^b is a refinement of \sim_0 , we have $\mathcal{S}^{\sim_0} = (\mathcal{S}^{\sim_0})^{\sim_0^b}$. In other words, recombinations at aligned positions include recombinations at aligned positions within the same BWT run. We could therefore build the GCSA structure for the closure \mathcal{S}^{\sim_0} . The challenge is doing that without representing the closure explicitly.

5.4 GCSA construction

Before proceeding, we relax the definition of structural runs. The GCSA structure only requires that the equivalence classes of suffixes are lexicographic ranges that start with the same symbol. By not requiring that they also share the preceding symbol, we simplify the discussion and make the alignment graph $G^{\sim_0^s}$ smaller.

► **Definition 13** (Structural SA runs). *Let $\mathcal{S} = (S_1, \dots, S_m)$ be a collection of strings, let \sim_0 be an alignment of the suffixes, and let SA be the suffix array of the collection. A structural SA run is a maximal interval $[sp : ep]$ such that for any $x, x' \in [sp : ep]$ with $\text{SA}[x] = (i, j)$ and $\text{SA}[x'] = (i', j')$, we have $S_i[j :] \sim_0 S_{i'}[j' :]$. We use equivalence relation \sim_0^s to denote that suffixes are in the same structural SA run.*



■ **Figure 5** GCSA example for the string collection in Figure 4. We show the original graph (top) again with each vertex v annotated with $\text{id}(v)$ (see Section 5.5). The bottom graph is the final graph $G^{\sim s} = (V', E', \Sigma, \ell')$ for the closure, with each vertex $v' \in V'$ annotated with $\ell'(P_{v'})$. Paths $P_{v'}$ are chosen to be prefixes of the lexicographically smallest maximal path starting from the vertex. The table lists each vertex $v' \in V'$ identified by the label $\ell'(P_{v'})$, the corresponding parts of the GCSA structure, and the identifier $\text{id}(\text{locate}(v'))$ of the corresponding vertex in the original graph.

The algorithm we use for building GCSA [48, 49] is similar to the prefix-doubling approach to suffix array construction. We start from graph $G_1 = G^{\sim 0} = (V, E, \Sigma, \ell)$ and build a series of graphs $G_l = (V_l, E_l, \Sigma, \ell_l)$ for increasing path lengths l . If we consider the graphs finite automata, they all recognize the same language $\mathcal{S}^{\sim 0}$. There are two types of vertices $v \in V_l$:

1. The vertex corresponds to a path P_v of length $|P_v| = l$ starting from vertex $\text{locate}(v) = P_v[1] \in V$ in graph $G^{\sim 0}$. For every maximal path P starting from vertex v in graph G_l , we require that $\ell(P_v)$ is a prefix of $\ell_l(P)$.
2. The vertex corresponds to a half-open lexicographic range $[X_v : Y_v)$ of path labels. There is an arbitrary path P_v of length $|P_v| \leq l$ starting from vertex $\text{locate}(v) = P_v[1] \in V$ in graph $G^{\sim 0}$ such that $\ell(P_v) \in [X_v : Y_v)$. For every maximal path P starting from vertex v in graph G_l , we require that $\ell_l(P) \in [X_v : Y_v)$. For every other vertex $u \in V_l$ with $u \neq v$, we require that $\ell(P_u) \notin [X_v : Y_v)$ (if u is of type 1) or $[X_u : Y_u) \cap [X_v : Y_v) = \emptyset$ (if u is of type 2).

In both cases, we set $\ell_l(v) = \ell(\text{locate}(v))$.

If all vertices $v \in V_l$ are of type 2, we have a graph G_l that is equivalent to $G^{\sim 0}$, with vertices that can be sorted unambiguously by the labels of maximal paths starting from them. For each suffix $S_i[j :]$ of the closure $\mathcal{S}^{\sim 0}$, there is a vertex $v \in V_l$ such that $S_i[j :] \in [X_v : Y_v)$ and a path P starting from v with label $\ell_l(P) = S_i[j :]$. Therefore the vertices of G_l correspond to (possibly non-maximal) structural runs. If we merge them to form maximal runs, we get the graph $G^{\sim s}$. See Figure 5 for an example.

We derive graph G_{2l} from graph G_l with a *doubling* step followed by a *pruning* step. Let $u \in V_l$ be a type 1 vertex, and let $\text{rank}_l(u)$ be the lexicographic rank of $\ell(P_u)$ in the set of distinct path labels $\{\ell(P_v) \mid v \in V_l\}$. In the doubling step, we combine it with all vertices $v \in V_l$ such that $P_u P_v$ is a valid path. For every combination, we create a new vertex w with $P_w = P_u P_v$ and $\text{rank}_{2l}(w) = (\text{rank}_l(u), \text{rank}_l(v))$. The new vertex will be of the same type as vertex v . This requires that graph $G^{\sim 0}$ is reverse deterministic; see below for a discussion. If $u \in V_l$ is a type 2 vertex, we copy it as vertex $w \in V_{2l}$ with $\text{rank}_{2l}(w) = (\text{rank}_l(u), 0)$. The doubling step finishes by sorting the new vertices by their ranks and replacing the pairs of ranks with integer values.

In the pruning step, we merge lexicographic ranges of type 1 vertices into type 2 vertices. Let $V_{2l}^x = \{v \in V_{2l} \mid \text{rank}_{2l}(v) = x\}$ be the set of vertices with rank x . If all vertices $v \in V_{2l}^x$ share the same $\text{locate}(v)$ vertex, the shared path label uniquely defines the starting vertex. We can therefore merge them into a type 2 vertex. Similarly, if sets V_{2l}^x and V_{2l}^{x+1} also share the starting vertex, we can merge them into a single vertex.

If S_{\max} is the longest string in \mathcal{S}° , the prefix-doubling algorithm finishes after at most $\lceil \log |S_{\max}| \rceil$ doubling steps. In the worst case, the final graph $G^{\sim^{\circ}}$ can be exponentially larger than the graph G° we started from. However, if the distances between the positions of the original string S with edits in any string $S' \in \mathcal{S}$ are sufficiently high, graph $G^{\sim^{\circ}}$ will have $O(|V|)$ vertices and $O(|E|)$ edges in the expected case [49]. If we assume a random string and random edits, the phase transition to exponential growth occurs at average distance $O(\log_{\sigma} |S|)$.

We do not need to maintain the edges of the intermediate graphs. Determining them for the final graph $G^{\sim^{\circ}} = (V', E', \Sigma, \ell')$ is simple enough. Given a vertex $v' \in V'$ of the final graph and an edge $(u, \text{locate}(v')) \in E$ of the original graph, we know that there must be a corresponding edge $(u', v') \in E'$ such that $\text{locate}(u') = u$ in the final graph. We can therefore generate the edges as (u, v') and sort them by $(\ell(u), \ell'(P_{v'}))$. This also sorts them by $\ell'(P_{u'})$. By scanning the lists of vertices and edges in sorted order, we can then match edges (u, v') to source vertices u' by $\text{locate}(u') = u$.

This construction algorithm only works if the original graph G° is *reverse deterministic*. For every vertex $v \in V$, we require that each predecessor $u \in V$ with $(u, v) \in E$ has a distinct label $\ell(u)$. If two predecessors $u, u' \in V$ share a label, $\ell(uP) = \ell(u'P)$ for every path P starting from vertex v . The pruning step cannot merge them, because no extension of the path label can determine the starting vertex uniquely. On the other hand, if the graph is reverse deterministic and $v \in V$ is a type 2 vertex, $\ell(uP_v)$ always determines the unique predecessor u .

Determinizing a finite automaton takes exponential time in the worst case, because the states of the determinized automaton correspond to sets of states of the non-deterministic automaton. However, when the textbook algorithm was used to determinize the graph before GCSA construction, it was observed to be fast [49]. This was later explained using Wheeler graphs [16]. The exponent in the time complexity is not the total number of states but the maximal number of states that are not comparable in lexicographic order. Intuitively, the closer the graph is to a Wheeler graph, the faster the determinization is. If the graph is simple enough that the prefix-doubling algorithm only increases its size by a constant factor, it must already be close to being a Wheeler graph.

5.5 Locating the hits

Let $G^{\circ} = (V, E, \Sigma, \ell)$ be the original alignment graph and $G^{\sim^{\circ}} = (V', E', \Sigma, \ell')$ the final graph in GCSA construction. During the construction, we annotated each vertex $v \in V_l$ in each intermediate graph $G_l = (V_l, E_l, \Sigma, \ell_l)$ (including G° and $G^{\sim^{\circ}}$) with the corresponding vertex $\text{locate}(v) \in V$ of the original graph. As the notation implies, we want GCSA locate queries return positions in the original graph. While it would be easier to return positions in $G^{\sim^{\circ}}$ (we built GCSA for that graph, after all), it is only a technical construct. Positions in the original graph (or columns in the alignment) are more likely to be meaningful to the user.

A locate query in an FM-index relies on the fact that $\text{SA}[x] = \text{SA}[\text{LF}(x)] + 1$. If a position has not been sampled, we can derive the value from the predecessor. Let $\text{id} : V \rightarrow [1 : |V|]$ be a function that gives each vertex $v \in V$ a unique integer identifier. To apply the same idea with GCSA, we require that $\text{id}(v) = \text{id}(u) + 1$ for each edge $(u, v) \in E$ such that u is

the only predecessor of v and v is the only successor of u . If we sample $\text{id}(\text{locate}(v))$ for each vertex $v \in V'$ that has multiple predecessors or $\text{id}(\text{locate}(v)) \neq \text{id}(\text{locate}(u)) + 1$ for the only predecessor u , the locate algorithm works as before [48, 49]. (We also want to sample $\text{id}(\text{locate}(v))$ at regular intervals on unary paths to ensure query performance.)

This approach was originally designed for sampling the LCP array [45]. If $\text{BWT}[x]$ is not at the start of a run, $\text{LCP}[x] = \text{LCP}[\text{LF}(x)] - 1$. We therefore need to sample the irreducible values, while the rest can be derived from the predecessor. More generally, the approach can be understood as a sampled tag array [8]. We annotate each position in the FM-index with a tag, which can store arbitrary information related to the position. If most tags can be derived easily from the only predecessor, we can store the tag array space-efficiently and still support efficient queries.

5.6 Simplifying the graph

As mentioned earlier, there is a phase transition in GCSA construction. As the distances between positions with edits in the alignment get below a critical threshold, the number of vertices and edges in the final graph G^{\sim^s} starts growing rapidly. This makes the GCSA impossible to build in practice for many graphs. One way to handle this is adding a context length requirement to the definition of alignment [48, 49].

► **Definition 14** (*k*-aligned suffixes). *Let $\mathcal{S} = (S_1, \dots, S_m)$ be a collection of strings, let MSA be a multiple sequence alignment for the collection, and let $k \geq 0$ be a context length. For any two strings $S_i, S_{i'} \in \mathcal{S}$ and positions j, j' in them, suffixes $S_i[j:]$ and $S_{i'}[j'::]$ are *k*-aligned if and only if positions $S_i[j]$ and $S_{i'}[j']$ are a match in MSA and $S_i[j:j+k] = S_{i'}[j':j'+k]$. We represent this with equivalence relation \sim_k .*

We similarly define structural SA runs based on *k*-aligned suffixes and denote them with equivalence relation \sim_k^s . Then we start GCSA construction from graph G^{\sim^k} and use prefix-doubling to build graph $G^{\sim_k^s}$. In practice, context length *k* imposes a minimum distance between recombinations. Regions that are heavily branching in G^{\sim^0} gradually become simpler in G^{\sim^k} as *k* increases. That makes it more likely that as path length *l* grows, path labels become unique at a faster rate than the number of type 1 vertices grows.

5.7 Simplified data structure

The final graph $G^{\sim_k^s} = (V, E, \Sigma, \ell)$ is always reverse deterministic. To see that, consider any two edges $(u, v) \in E$ and $(u', v) \in E$ such that $\ell(u) = \ell(u')$. Because *u* and *u'* are type 2 vertices, it must be that $\ell(uP) = \ell(u)\ell(P) \in [X_u : Y_u]$ and $\ell(u'P) = \ell(u')\ell(P) \in [X_{u'} : Y_{u'}]$, where *P* is a maximal path starting from *v*. As the lexicographic ranges for two type 2 vertices cannot overlap, we have $u = u'$.

For each vertex $v \in V$ and symbol $c \in \Sigma$, there is at most one vertex $u \in V$ such that $(u, v) \in E$ and $\ell(u) = c$. We can therefore replace the indegree bitvector B_{in} and the BWT with a binary matrix [49]. For each symbol $c \in \Sigma$, let B_c be a bitvector of length $|V|$, with $B_c[\text{rank}_V(v)] = 1$ whenever vertex $v \in V$ has a predecessor $u \in V$ with $\ell(u) = c$. The first two parts of a backward search step become $[sp_{out}, ep_{out}] = [C[c] + B_c.\text{rank}(sp - 1, 1) + 1 : C[c] + B_c.\text{rank}(ep, 1)]$.

We replaced a select query on a bitvector and a rank query on a string with a single rank query on a bitvector. A backward search step now consists of four rank queries on bitvectors. Depending on the data structures used, this can be considerably faster than with the full structure. With uncompressed bitvectors, the size bound increases from $|E|(\log \sigma + 2)(1 + o(1))$

bits to $(\sigma|V| + |E|)(1 + o(1))$ bits, which can be significant with large alphabets. With Elias–Fano encoded bitvectors for B_c , we get $|E|(\log(\sigma|V|/|E|) + O(1))$ bits at the expense of somewhat slower queries. A practical hybrid approach [46] uses uncompressed bitvectors for common symbols and compressed bitvectors for rare ones.

SBWT [5] (Section 4.1) goes even further with a representation, where the rank queries on B_c and B_{out} can be done with a single rank query on a bitvector. This comes at the expense of some functionality. In particular, SBWT does not tell directly whether vertex $v \in V$ has a predecessor with symbol $c \in \Sigma$.

5.8 Graphs with cycles

BOSS [10] represents de Bruijn graphs with a data structure that is equivalent to the full GCSA structure (Section 5.3). While a de Bruijn graph often contains cycles, its vertices v are type 2 vertices, with $[X_v : Y_v) = \{S \in \Sigma^* \mid S_v \text{ is a prefix of } S\}$ as the interval. Here S_v is a string of length $|S_v| = k$ and k is the order of the de Bruijn graph. We could therefore use GCSA, including the more efficient simplified structure (Section 5.7), with de Bruijn graphs. The main difference is that while LF-mapping and backward search follow the edges in BOSS, they go against the direction of the edges in GCSA.

A natural question is which classes of graphs can be represented using GCSA, and what is the corresponding class of formal languages. Because vertex-labeled graphs can be understood as finite automata, the class of languages is a subclass of regular languages. Sirén et al. [49] showed that the class is a proper subclass: GCSA can represent the obvious automaton recognizing language $\{a, b\}^*$ but no automaton recognizing $\{a, b\}^* \cup \{a, c\}^*$. The characterization of type 2 vertices with lexicographic ranges suggests that the class of graphs admitting the GCSA representation is a generalization of de Bruijn graphs. The properties of this class were later investigated based on the Wheeler graph formalization [1, 2].

5.9 GCSA2

GCSA was designed for indexing pangenome graphs based on aligned DNA sequences. The construction algorithm assumes that the alignments can be represented as a multiple sequence alignment or an acyclic graph. Such models are appropriate for simple variants, such as substitutions, insertions, and deletions. Other forms of variation, such as rearrangements (same substrings appear in different order in different strings) and inversions (a substring is replaced with its reverse complement) can be represented better with cycles in the graph. Such graphs often do not have an equivalent Wheeler graph that can be indexed with GCSA.

GCSA2 [46] is a reimplement of GCSA for general pangenome graphs. Because the graph cannot be fully indexed, GCSA2 approximates it with an order- k de Bruijn graph. It matches patterns of length at most k correctly, while longer matches may be false positives combining partial matches in different parts of the original graph. The initial implementation used $k = 128$, which was chosen to make excessive growth of the number of vertices less likely. This was later increased to $k = 256$ to allow mapping perfectly matching 150 bp and 250 bp reads as a single hit [47].

Let $G = (V, E, \Sigma, \ell)$ be the original graph and $G_k = (V', E', \Sigma, \ell')$ be the graph after $\log k$ rounds of prefix-doubling (Section 5.4). We replace the final pruning step with a merging step that also merges all type 1 vertices $v \in V'$ sharing $\text{rank}_k(v)$, regardless of whether the $\text{locate}(v)$ vertices match. The resulting *pruned de Bruijn graph* is equivalent to an order- k de Bruijn graph but potentially much smaller. To handle some edge cases, we must ensure

that all type 2 vertices $v \in V'$ have intervals $[X_v : Y_v)$ defined by a shared prefix. We build GCSA2 for graph G_k . As a consequence of the final merge, $\text{locate}(v) \subseteq V$ is now a set of vertices in the original graph.

Pangenome graphs contain regions that are dense in variation. While GCSA2 construction stops prefix-doubling early to prevent the worst exponential growth, such regions must often be simplified to keep the size of the index manageable. Because the original GCSA considered the alignments and the graph computational artifacts, it could handle complex regions by changing the definition of alignment (Section 5.6). GCSA2 cannot do that, because the provided alignments are assumed to be biologically meaningful. Instead, a preprocessing step removes everything except the reference genome in regions where there are too many branching vertices in a short window [26]. There is also an option to avoid sequence loss by unfolding the original strings in the complex region while maintaining a mapping to the corresponding positions in the original graph. Variation is then replaced with a set of disjoint paths representing unaligned strings [47].

GCSA2 supports some additional functionality beyond that provided by FM-indexes. It builds a trie of path labels $\ell(P_v)$ for vertices $v \in V'$ and represents it using next/previous smaller value queries and range minimum queries over the LCP array [23]. The resulting suffix tree-like functionality is used for finding maximal exact matches between the pattern and the graph [41]. A similar approach was first used with variable-order de Bruijn graphs [9] and later generalized to Wheeler graphs [15]. There is also a data structure for reporting the number of distinct $\text{locate}(v)$ values for vertices $v \in V'$ in a range $[sp : ep]$. It is based on compressed versions [24] of Sadakane’s document counting structure [44].

6 Founder Graphs

In this section, we deal with graphs built from the segmentation of multiple sequence alignments (MSAs), which is a line of research originating from the study of *founder sequences* [50]. Similarly to the GCSA approach from Section 5, respecting the MSA positions (i.e. the columns) results in an acyclic graph spelling the input strings and their recombination. However, differently from GCSA, in these segmentation-induced graphs that we call founder graphs, the recombinations depend exclusively on the chosen segments.

Many approaches reviewed so far fix a k -mer length to avoid indexing exponentially many paths of a graph. While such fixed parameter yields practically efficient solutions, it is of great theoretical value to study more general approaches. Wheeler Graphs [25] have been proposed as such, but they appear too general; Wheeler Graph recognition is an NP-complete problem [27] and thus to use Wheeler graph machinery one first needs to limit to graph classes that somehow avoid this complexity. Founder graphs also have similar hardness issues: in Section 6.2 we present the negative results from [17] stating that matching in founder graphs is not easier than matching in general labeled graphs. However, this hardness can be avoided, as we present in Section 6.3 that suitable segmentations of MSAs respecting a specific uniqueness property induce graph classes that are easy to index for linear-time pattern matching. These graphs are not limited to a fixed k -mer length, but can handle node labels of arbitrary length (provided that they are unique) and arbitrary queries. After delving into more details, we continue discussing the connection to Wheeler Graphs in Sect. 6.3. We conclude this section with a simple extension of the uniqueness properties of de Bruijn graphs and founder graphs to overlap graphs.

6.1 Inducing graphs from multiple sequence alignments

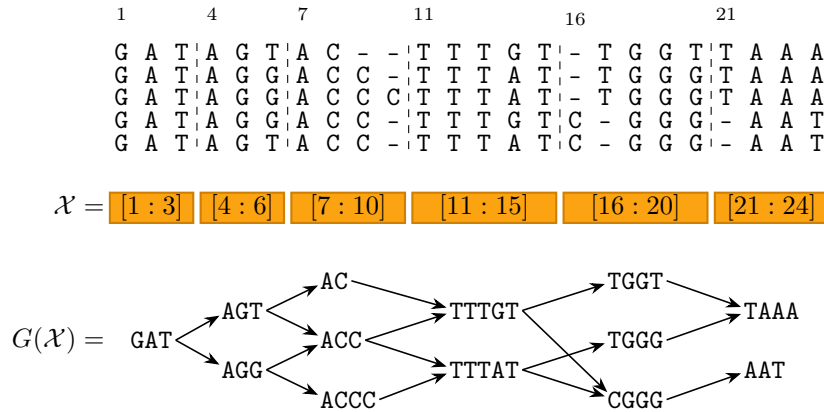
We define a *segmentation* \mathcal{X} of $\text{MSA}[1 : m, 1 : n]$ as a partition of columns $[1 : n]$ in b segments $[1 = x_1 : y_1], [x_2 : y_2], \dots, [x_b : y_b = n]$ such that $y_j = x_{j+1} - 1$ for $j \in [1 : b - 1]$, and additionally we forbid any segmentation to spell the empty string in one of its segments, in symbols $\text{spell}(\text{MSA}[i, x_j : y_j]) \neq \varepsilon$ for all $i \in [1 : m], j \in [1 : b]$.

► **Definition 15** (elastic founder graph [20]). *Let $\text{MSA}[1 : m, 1 : n] \in (\Sigma \cup \{-\})^{m \times n}$ be a multiple sequence alignment. Given segmentation $\mathcal{X} = [x_1 : y_1], \dots, [x_b : y_b]$, the elastic founder graph (EFG) induced by \mathcal{X} is a vertex-labeled graph $G = (V, E, \ell)$, where $\ell : V \rightarrow \Sigma^+$ assigns a non-empty label to every vertex, and the set of vertices V is partitioned into blocks V_1, \dots, V_b such that $|\{\ell(v) : v \in V_k\}| = |V_k|$ (unique strings in a block) and for all $k \in [1 : b]$ we have that*

$$\{\ell(v) : v \in V_k\} = \{\text{spell}(\text{MSA}[i, x_k : y_k]) \mid 1 \leq i \leq m\}.$$

Moreover, it holds that $(v, w) \in E$ if and only if there exists $k \in [1 : b - 1]$ and $i \in [1 : m]$ such that $v \in V_k, w \in V_{k+1}$, and $\text{spell}(\text{MSA}[i, x_k : y_{k+1}]) = \ell(v) \cdot \ell(w)$.

If $E = \bigcup_{k=1}^{b-1} (V_k \times V_{k+1})$, that is all vertices in V_k are connected to all vertices in V_{k+1} for each $b \in [1..b]$, then the EFG is an elastic degenerate string (EDS) [30] with no empty string, and we can omit edge set E . Moreover, given an EFG, we define its EDS relaxation as the corresponding EDS over V_1, \dots, V_b .



■ **Figure 6** Example of $\text{MSA}[1 : 5, 1 : 24]$, segmentation \mathcal{X} , and elastic founder graph $G = (V, E, \ell)$ induced by \mathcal{X} .

See Figure 6. Given an elastic founder graph $G = (V, E, \ell)$, for each vertex $v \in V$ we denote as $\|v\|$ the vertex label length $|\ell(v)|$. We define height $H(G) = \max_{k=1}^b |V_k|$ as the maximum number of vertices in a block, and $L(G) = \max_{v \in V} \|v\|$ as its maximum vertex label length. We can extend ℓ to label paths in the graph: given a $u_1 u_k$ -path $P = u_1 \cdots u_k$, with $(u_i, u_{i+1}) \in E$ for $i \in [1..k - 1]$, we define $\ell(P) = \ell(u_1) \cdots \ell(u_k)$; for an edge $(u, v) \in E$, we call $\ell(uv)$ its *edge label*. Then, we say that some string $Q \in \Sigma^+$ occurs in G if there exists path P in G such that Q is a substring of $\ell(P)$. More specifically, we indicate with triple $(i, u_1 \cdots u_k, j)$ the *subpath* of P spelling Q starting from position $i \in [1 : \|u_1\|]$ in u_1 and ending in position $j \in [1 : \|u_k\|]$ in u_k , that is, $Q = \ell(u_1)[i : j] \ell(u_2 \cdots u_{k-1}) \ell(u_k)[1 : j]$. The *length* of subpath $(i, u_1 \cdots u_k, j)$ is then $|Q|$. We say that Q occurs *starting from the beginning* of u_1 if $i = 1$; Q occurs *starting inside* u_1 if $i \in [2 : \|u_1\|]$.

6.2 Hardness of matching EFGs

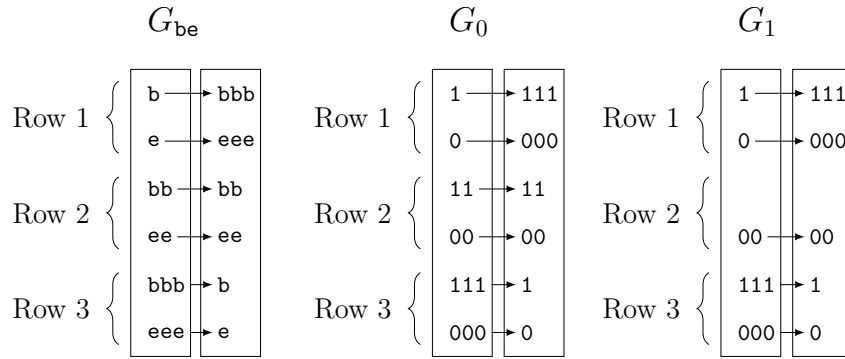
Starting from an MSA, we can construct EFG G . After this step, we would like to answer string matching queries for G , namely determining whether a given pattern string Q occurs in G . We will now show that, conditioned on some complexity hypotheses, this problem requires quadratic time complexity. This justifies the approach we will adopt in later sections, where we assume an additional property to hold in EFGs in order to break through this quadratic barrier.

The quadratic lower bound is obtained via fine-grained complexity techniques and is conditioned on the *Strong Exponential Time Hypothesis* (SETH) and the *Orthogonal Vectors Conjecture* (OVC). Since Williams [51] showed that SETH implies OVC, it suffices to show a subquadratic reduction from the *Orthogonal Vectors* (OV) problem to obtain this result. Let us first give a formal definition of OV.

► **Definition 16** (OV). *Let $X, Y \subseteq \{0, 1\}^d$ be two sets of n binary vectors of length d . Determine whether there exist $x \in X, y \in Y$ such that $x \cdot y = \sum_{i=1}^d x[i] \cdot y[i] = 0$.*

The notation $x[i] \cdot y[i]$ indicates the scalar product when used for two single entries of vectors x and y , while it refers to the dot product $x \cdot y$ when applied on the vectors themselves. Currently, it is conjectured that no algorithm can solve OV in truly subquadratic time.

► **Definition 17** (Orthogonal Vectors Conjecture (OVC)). *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 $O(\text{poly}(d)n^{2-\epsilon})$, where poly is a polynomial function on d .*



■ **Figure 7** 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 into the final graph, edges can be controlled to go within the same row, or to the row below. Figure from [20].

Then, we can show that determining whether string Q occurs in graph G is at least as hard as solving OV, obtaining a quadratic conditional lower bound.

► **Theorem 18** ([20]). *For any constant $\epsilon > 0$, it is not possible to find a match for a query string Q in an EFG $G = (V, E, \ell)$ in either $O(|E|^{1-\epsilon} |Q|)$ or $O(|E| |Q|^{1-\epsilon})$ time, unless OVC fails. This holds even if restricted to an alphabet of size 4.*

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.

We will provide an intuition on how to prove Theorem 18 by giving an idea of how to perform a subquadratic reduction from OV.

Starting from sets $X, Y \subseteq \{0, 1\}^d$ of n binary vectors of length d , the goal of the reduction is to build string Q and EFG G such that Q occurs in G if and only if there exist $x \in X, y \in Y$ such that $x \cdot y = 0$. Moreover, this reduction should be performed in not more than $O(n^{2-\epsilon} \text{poly}(d))$ time, for some constant $\epsilon > 0$.

Pattern. We start by constructing string Q from set of vectors X . We build Q by combining string gadgets Q_1, \dots, Q_n , one for each vector in X , plus some additional characters. For example, vector $x_i = 101$ results in string

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

$$Q_{i,2} = 0000, \quad Q_{i,3} = 1111.$$

Full string Q is then the concatenation $Q = \text{bbbb}Q_1Q_2 \dots Q_n\text{eeee}$.

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. Sub-graphs G_L is able to match any prefix of pattern Q and is connected only to row G_{M1} , which can also match any prefix of Q . Symmetrically, sub-graphs G_R is able to match any suffix of pattern Q and it is connected only to row G_{M3} , which can also match any suffix of Q . Thus, 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 Figure 8, 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 Figure 7), which allow to stack together multiple instances of strings $b^4, e^4, 1^4, 0^4$. The overall structure mimics the one in [17, 19], except for the new idea from Figure 7 ([20]).

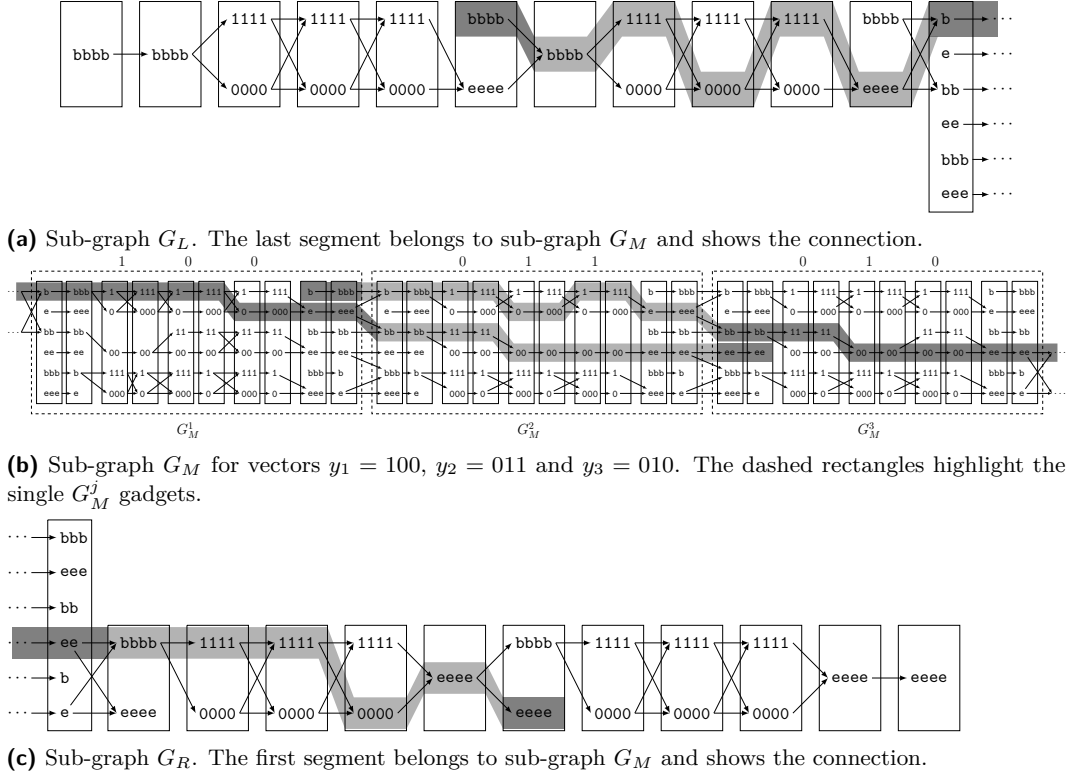
6.3 Indexable founder graphs

As we have seen, EFGs are not likely to support fast pattern matching without additional conditions. To alleviate this hardness, a uniqueness property has been introduced:

► **Definition 19** (semi-repeat-free indexability property [20]). *An EFG $G = (V, E, \ell)$ is semi-repeat-free or indexable (denoted as iEFG) if each subpath $(i, u_1 \dots u_p, j)$ spelling $\ell(v)$ for $v \in V_k$ is such that $i = 1$ and $u_1 \in V_k$, that is, $\ell(v)$ only occurs from the beginning of vertices in block V_k . The semi-repeat-free property is a generalization of the stronger repeat-free property, imposing that each $\ell(v)$ only occurs in G from the beginning of v .*

The simpler repeat-freeness condition requiring $\ell(v)$ to start only from the beginning of vertex v already enables fast pattern matching, but it is open if there is a fast algorithm to construct such EFG from the MSA [20].

Before proceeding to the construction of iEFGs, let us first gather some intuition why such graphs are useful. To fix ideas, assume the input MSA has no gaps —. Then semi-repeat-freeness matches with repeat-freeness, as the strings in each block are of the same length, and cannot be prefixes of one another. Consider searching for query string $Q[1 : m]$ in iEFG.



■ **Figure 8** 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. The shade of gray changes from one Q_i to the next. Any such occurrence must pass through the middle row of G_M . Figure from [20].

An FM-index for the collection of edge labels (concatenation of adjacent vertex labels) can be used for checking if Q occurs inside a vertex or an edge. Longer occurrences must span at least one complete vertex label. One can build an Aho-Corasick automaton on the vertex labels to find all candidate occurrences. Consider vertex v with $\ell(v)$ occurring closest to the end of Q , say, at $Q[i : j]$. Consider a trie built on the set $F(v) = \{\ell(w) \mid (v, w) \in E\}$. We can check if $Q[j + 1..m]$ can be traced from the root of $F(v)$ to verify if $Q[i : j]$ occurs starting from the beginning of v . To verify if $Q[1 : i - 1]$ occurs ending at the end of vertex u s.t. $(u, v) \in E$, we can use tries $R(v) = \{\ell(u)^{-1} \mid (u, v) \in E\}$: We can scan $Q[1 : i - 1]$ from right to left in $R(v)$. If we reach a leaf, it corresponds to a unique vertex u due to the (semi-)repeat-free property. The scanning can continue from $R(u)$, until we have verified an occurrence for Q or detected that there was no occurrence starting from our candidate vertex v .

▷ **Claim 20.** If Q is found neither inside an edge label nor with the verification algorithm above, it has no occurrence in iEFG.

Proof. For contradiction, assume Q has an occurrence $(i', u_1 \dots u_k, j')$ in G . Since such occurrence is not found within an edge, it must span at least one vertex of G . Therefore the verification algorithm has started with some vertex v such that $Q[i : j] = \ell(v)$, but did not report this occurrence. That is, the substring $Q[i : j] = \ell(v)$ occurs starting somewhere else than in the beginning of v and thus our iEFG is not repeat-free. ◁

This tailored Aho-Corasick approach can be made more space-efficient by an *expanded backward search* mechanism [20] similar to one used for de Bruijn graphs in Section 4.1. Alternatively, one can reduce this case of repeat-free graphs induced from MSAs with no gaps to Wheeler Graphs [20].

It is important to note why none of the above approaches work if the MSA has gaps and our iEFG is semi-repeat-free: $\ell(v)$ is allowed to occur starting from another vertex v' from the same block. Due to this, straightforward extensions lead to false positive matches, and it is open if there is a reduction to Wheeler Graphs. However, there is a way to fix the Aho-Corasick approach: consider instead edges of the graph, as they have a uniqueness property analogous to the one above for vertices in the repeat-free case. One can basically repeat similar scanning of suffix and prefix of Q on indexes built on the set of edges, but there is a non-trivial detail in connecting the searches we omit here [43]:

► **Theorem 21** ([43]). *A semi-repeat-free EFG $G = (V, E, \ell)$ can be indexed in polynomial time into a data structure occupying $O(|D| \log |D|)$ bits of space, where $|D| = O(N \cdot H(G))$, $N = \sum_{v \in V} \|v\|$ is the total length of the vertex labels, and $H(G)$ is the height of G . Later, one can find out in $O(|Q|)$ time if a given query string Q occurs in G .*

Next, we will consider how to construct iEFGs. To fix ideas, we consider only the problem of maximizing the number of blocks in iEFG; there exist similar solutions for other objective functions like minimizing the maximum segment length and minimizing the maximum height of a block.

Recall from Definition 15 that an EFG is defined by a graph induced by a segmentation $\mathcal{X} = [x_1 : y_1], \dots, [x_b : y_b]$ of an MSA. We denote this graph $G(\mathcal{X})$.

► **Definition 22** ([20]). *Segment $[x : y]$ of $\text{MSA}[1 : m, 1 : n]$ is semi-repeat-free if for any $i, i' \in [1 : m]$ string $\text{spell}(\text{MSA}[i, x : y])$ occurs in gaps-removed row $\text{spell}(\text{MSA}[i', 1 : n])$ only at position $g(i', x)$, where $g(i', x)$ is equal to x minus the number of gaps in $\text{MSA}[i', 1 : x - 1]$.²*

► **Lemma 23** (Characterization of semi-repeat-free segments [20]). *The graph $G(\mathcal{X})$ induced from a segmentation \mathcal{X} is semi-repeat-free if and only if all segments of \mathcal{X} are semi-repeat-free.*

Consider you have found a semi-repeat-free segmentation until column x and want to extend it with a new semi-repeat-free segment into a longer segmentation. The following definition tells the minimum length of such extension.

► **Definition 24** (Minimal right extensions [20]). *Given $\text{MSA}[1 : m, 1 : n]$, for each $0 \leq x \leq n - 1$ we define value $f(x)$ as the smallest integer greater than x such that segment $[x + 1 : f(x)]$ is semi-repeat-free. If such integer does not exist, we define $f(x) = \infty$.*

To see how to compute these $f(x)$ values, let us for now assume our MSA is gapless. One can build a generalized suffix tree of the rows of the MSA. Consider the set of leaves of the suffix tree corresponding to suffixes of the rows starting at column $x + 1$. The goal is to find a set of suffix tree vertices that cover these leaves and only them. Among such sets, one should find the one that contains a vertex with the smallest string depth, as that will determine $f(x)$. This is called the *exclusive ancestor set* problem and can be solved in linear time in the number of leaves in the input set [43]. Updating the set of leaves when moving from column x to $x - 1$ can also be done in linear time, but with general MSAs, one needs to tailor the algorithm to take into gaps (e.g. in string depth), which is non-trivial but possible.

² The original definition considers mistakenly gaps in $\text{MSA}[i', 1 : x]$.

► **Theorem 25** ([43]). *Given $\text{MSA}[1 : m, 1 : n] \in \{\Sigma \cup \{-\}\}^{m \times n}$ with $\Sigma = [1 : \sigma]$ and $\sigma \leq mn$, we can compute the minimal right extensions $f(x)$ for $0 \leq x < n$ (Definition 24) in time $O(mn)$.*

► **Corollary 26** ([43]). *Given $\text{MSA}[1 : m, 1 : n] \in \{\Sigma \cup \{-\}\}^{m \times n}$ with $\Sigma = [1 : \sigma]$ and $\sigma \leq mn$, the construction of an optimal semi-repeat-free segmentation maximizing the number of blocks can be done in time $O(mn)$.*

Proof. Algorithm [20, Algorithm 1] by Equi et al. solves the problem in $O(n)$ time, assuming minimal right extensions given (Theorem 25). ◀

6.4 Repeat-free graphs: generalizing de Bruijn graphs with vertex uniqueness

We generalize iEFGs respecting the repeat-free property to overlap graphs respecting an equivalent and homonymous property. We show that graphs where long strings are constrained to have a unique starting location in the graph are a straightforward generalization of de Bruijn graphs, and are still indexable in polynomial time for linear-time pattern matching.

We define labeled graphs where each vertex is labeled with a non-empty string of variable length, and each directed edge specifies a suffix-prefix overlap between the connected vertices, as follows.

► **Definition 27** (Overlap graph). *An overlap graph is a quadruple $G = (V, E, \Sigma, \ell, \text{ov})$ where V is a finite set of vertices, $\ell: V \rightarrow \Sigma^+$ labels the vertices, $E \subseteq V \times V$ is the set of directed edges, and $\text{ov}: E \rightarrow \mathbb{N}$ defines the length of a (possibly empty) suffix-prefix overlap between the vertices of each edge $uv \in E$ such that the following holds: $0 \leq \text{ov}(uv) < \min(\|u\|, \|v\|)$ and*

$$\ell(u)[\|u\| - \text{ov}(uv) + 1 :] = \ell(v)[: \text{ov}(uv)].$$

Condition $\text{ov}(uv) < \min(\|u\|, \|v\|)$ imposes the overlap to be less than one full vertex.

See Figure 9. We extend ℓ to label paths of G as follows: given path $P[1 : k]$ ($(P[i], P[i+1]) \in E$ for all $i \in [1 : k-1]$), we define path label $\ell(P[1] \cdots P[k])$ as $\ell(u_1)$ if $k = 1$, otherwise it is equal to

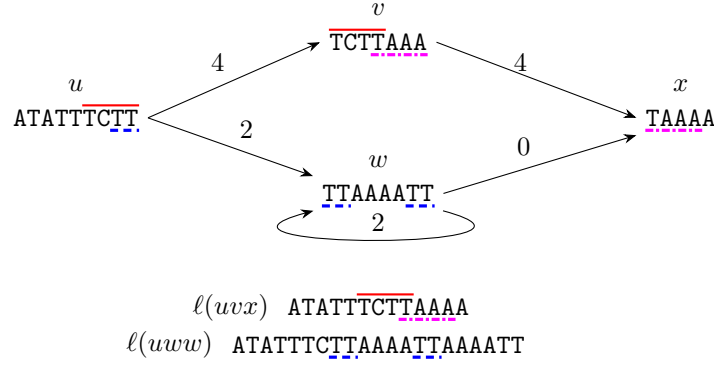
$$\ell(u_1)[\|u_1\| - \text{ov}(u_1 u_2) + 1 :] \cdots \ell(u_{k-1})[\|u_{k-1}\| - \text{ov}(u_{k-1} u_k) + 1 :] \cdot \ell(u_k),$$

that is, the string spelled by path P considers the overlaps between adjacent vertices only once.

6.4.1 Hardness of exact matching in overlap graphs

We say that a pattern $Q \in \Sigma^+$ occurs in an overlap graph G if Q is a substring of $\ell(P)$ for some path P in G . Then, it is immediate to see that the problem of matching a query in an overlap graph is a generalization of string matching in labeled graphs, since the latter are vertex-labeled graphs with overlap equal to 0 for all edges. Then, the conditional lower bounds of [19, 18] holds also for overlap graphs. However, two classes of graphs that bypass the conditional lower bounds by having a *uniqueness property* of the possible strings spelled in the graph are *de Bruijn graphs* and *iEFGs* we considered earlier.

► **Observation 28.** *Overlap graphs (Definition 27) are a generalization of EFGs (Definition 15), since the latter are directed graphs, the vertices are labeled with non-empty strings, and the edges do not present any overlap ($\text{ov}(uv) = 0$ for all $uv \in E$).*



■ **Figure 9** An overlap graph with four vertices u, v, w , and x , where the overlaps are represented above each edge. Note that in the spelling of edges and paths we do not repeat the overlapping substrings.

► **Observation 29.** *Overlap graphs (Definition 27) are a generalization of de Bruijn graphs (Definition 7) and compacted de Bruijn graphs (see e.g. [13]), since these can be represented as k -mers or unitigs connected by directed edges with a fixed overlap of $k - 1$ ($\text{ov}(uv) = k - 1$ for all $uv \in E$).*

► **Observation 30** (Uniqueness property of dBGs). *In a (compacted) de Bruijn Graph $G = (V, E, \Sigma, \ell, \text{ov})$, we have that for every possible k -mer $S \in \Sigma^k$, if S occurs in G then S is uniquely associated to a single node in G .*

6.4.2 Indexable overlap graphs

We study the natural extension of the repeat-free property (Definition 19) and the uniqueness property of de Bruijn Graphs (Observation 30) to overlap graphs, where the occurrences of patterns ignore overlaps between nodes. For this, we need to redefine the interpretation of occurrence positions, as an occurrence starting inside the overlapping part of an edge (u, v) can be seen to start from u or from v .

► **Definition 31.** *Let query Q have an occurrence $(i, u_1 u_2 \cdots u_k, j)$ in an overlap graph G with $\|u_1\| - \text{ov}(u_1 u_2) < i \leq \|u_1\|$. We say that such occurrence is redundant, as it can be written equivalently as $(i - (\|u_1\| - \text{ov}(u_1 u_2)), u_2 \cdots u_k, j)$. Occurrences that are not redundant are called non-redundant.*

► **Definition 32.** *Query Q occurs in an overlap graph G starting inside a node if it has a non-redundant occurrence $(i, u_1, u_2, \dots, u_k, j)$ with $i > 1$. Query Q occurs in an overlap graph G starting from the beginning of a node if all its non-redundant occurrences are of the form $(1, u_1, u_2, \dots, u_k, j)$.*

► **Definition 33.** *We say that an overlap graph $G = (V, E, \Sigma, \ell, \text{ov})$ is repeat-free if each non-redundant occurrence $(i, u_1 \cdots u_k, j)$ of $\ell(u)$ for $u \in V$ is such that $i = 1$ and $u_1 = u$, that is, $\ell(u)$ occurs in G only starting from the beginning of u .*

Then, we argue that the *expanded backward search* algorithm [20] for repeat-free EFGs is correct also for overlap graphs, since it performs the following steps:

1. preprocess G by indexing string

$$T_{\text{edges}} = \# \prod_{(u,v) \in E} \ell(uv) \#$$

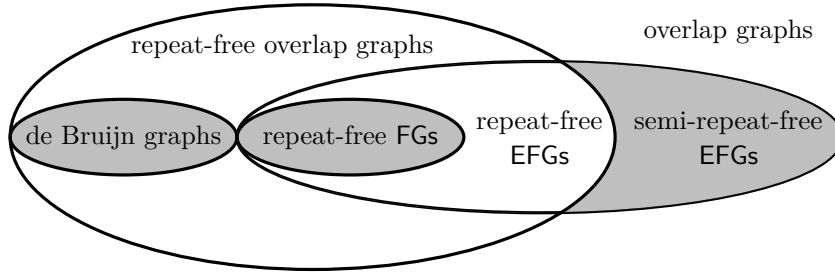
for pattern matching queries and bit arrays $B[1 : |T_{\text{edges}}| + 1]$, $E[1 : |T_{\text{edges}}| + 1]$ for rank and select queries, where B and E are defined such as $B[k] = 1$ and $E[r] = 1$ if and only if $[s : e]$ is the lexicographic interval of some node label $\ell(v)$, $v \in V$; then, given a non-empty lexicographical interval $[s : e]$ of string S in T_{edges} , we can answer in constant time whether $\ell(v)$ is prefix of S for some node $v \in V$;

2. perform backward search for Q in T_{edges} and, when the current lexicographical interval is fully contained in the lexicographical interval of some $\ell(v)$, *expand* the interval to that of $\ell(v)$ and continue the search.

The correctness follows from the repeat-free property and the lexicographic order: whenever we have read a substring $Q[i : j]$ of Q – initially, a suffix of Q – whose lexicographical interval is contained in that of some $\ell(v)$, it must be that $\ell(v)$ is a prefix of $Q[i : j]$ and by the repeat-free property all matches of $Q[i : j]$ in G start from the beginning of v . By expanding the lexicographic range to that of $\ell(v)$, we are discarding the characters $Q[i + |\ell(v)| : j]$ and we can continue with the search without losing any occurrence.

► **Theorem 34.** *A repeat-free overlap graph $G = (V, E, \Sigma, \ell, \text{ov})$ can be indexed in $O(|T_{\text{edges}}|) = O(LD)$ time, where $L = \max_{uv \in E} |\ell(uv)|$ and $D = \max_{v \in V} \max(\delta^+(v), \delta^-(v))$, where $\delta^+(v)$ and $\delta^-(v)$ are the indegree and outdegree of vertex v , to answer whether $Q \in \Sigma^+$ occurs in G in $O(|Q|)$ time.*

We leave it for future work how to construct such repeat-free overlap graphs, which is at least as hard as constructing repeat-free founder graphs from MSA with gaps, for which we do not have an efficient construction either.



■ **Figure 10** Hierarchy of vertex-labeled graph (with non-empty long labels) admitting a poly-time index for linear-time pattern matching. The classes of graphs with known linear-time construction algorithms are highlighted in gray. Note that semi-repeat-free EFGs are a superclass of repeat-free EFGs, that in turn contain repeat-free non-elastic founder graphs, and that the class of overlap graphs (Definition 27) contains all shown classes.

7 Conclusions

This chapter has reviewed BWT-inspired indexing of more complex structures, such as tries, finite languages, de Bruijn graphs, and aligned sequences. As we have seen, much can be gained by tailoring the BWT to these specific types of objects. The next chapter covers a more general indexing framework for labeled graphs, via the notion of Wheeler Graphs.

Even for the special cases we have covered above, many open problems remain. Although enormous progress has been made toward making these data structures practical – in many cases to the point that they are used routinely a variety of bioinformatics pipelines (e.g., the SBWT lies at the heart of the Themisto pseudoalignment software [6]) – further improvements

in performance practical performance are desirable. Construction algorithms in particular have room for improvement, and precisely how best to add parallelization to both construction and querying algorithms is currently under-explored.

References

- 1 Jarno Alanko, Giovanna D’Agostino, Alberto Policriti, and Nicola Prezza. Regular languages meet prefix sorting. In *Proc. SODA 2020*, pages 911–930. SIAM, 2020. doi:10.1137/1.9781611975994.55.
- 2 Jarno Alanko, Giovanna D’Agostino, Alberto Policriti, and Nicola Prezza. Wheeler languages. *Information and Computation*, 281:104820, 2021. doi:10.1016/j.ic.2021.104820.
- 3 Jarno N. Alanko, Elena Biagi, and Simon J. Puglisi. Longest common prefix arrays for succinct k-spectra. In *Proc. SPIRE*, LNCS 14240, pages 1–13. Springer, 2023. doi:10.1007/978-3-031-43980-3_1.
- 4 Jarno N. Alanko, Elena Biagi, Simon J. Puglisi, and Jaakko Vuohtoniemi. Subset wavelet trees. In *Proc. of the 21st International Symposium on Experimental Algorithms (SEA)*, LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SEA.2023.4.
- 5 Jarno N. Alanko, Simon J. Puglisi, and Jaakko Vuohtoniemi. Small searchable k-spectra via subset rank queries on the spectral Burrows-Wheeler transform. In *Proc. ACDA*, pages 225–236. SIAM, 2023. doi:10.1137/1.9781611977714.20.
- 6 Jarno N. Alanko, Jaakko Vuohtoniemi, Tommi Mäklin, and Simon J. Puglisi. Themisto: a scalable colored k -mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. *Bioinform.*, 39(Supplement-1):260–269, 2023. doi:10.1093/BIOINFORMATICS/BTAD233.
- 7 Uwe Baier. On undetected redundancy in the Burrows-Wheeler transform. In *Proc. CPM 2018*, volume 105 of *LIPIcs*, pages 3:1–3:15, 2018. doi:10.4230/LIPIcs.CPM.2018.3.
- 8 Andrej Baláž, Travis Gagie, Adrián Goga, Simon Heumos, Gonzalo Navarro, Alessia Petescia, and Jouni Sirén. Wheeler maps. In *Proc. LATIN 2024*, volume 14578 of *LNCS*, pages 178–192. Springer, 2024. doi:10.1007/978-3-031-55598-5_12.
- 9 Christina Boucher, Alex Bowe, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane. Variable-order de Bruijn graphs. In *Proc. DCC 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 *International workshop on algorithms in bioinformatics*, pages 225–235. Springer, 2012. doi:10.1007/978-3-642-33122-0_18.
- 11 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994. URL: <http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html>.
- 12 Bastien Cazaux and Eric Rivals. Linking BWT and XBW via Aho-Corasick automaton: Applications to run-length encoding. In *Proc. CPM 2019*, volume 128 of *LIPIcs*, pages 24:1–24:20, 2019. doi:10.4230/LIPIcs.CPM.2019.24.
- 13 Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinform.*, 32(12):201–208, 2016. doi:10.1093/BIOINFORMATICS/BTW279.
- 14 David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage (extended abstract). In *Proc. SODA 1996*, pages 383–391. SIAM, 1996. doi:10.1145/313852.314087.
- 15 Alessio Conte, Nicola Cotumaccio, Travis Gagie, Giovanni Manzini, Nicola Prezza, and Marinella Sciortino. Computing matching statistics on Wheeler DFAs. In *Proc. DCC 2023*, pages 150–159. IEEE, 2023. doi:10.1109/DCC55655.2023.00023.
- 16 Nicola Cotumaccio, Giovanna D’Agostino, Alberto Policriti, and Nicola Prezza. Co-lexicographically ordering automata and regular languages - part I. *Journal of the ACM*, 70(4):1–73, 2023. doi:10.1145/3607471.

- 17 Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. On the complexity of string matching for graphs. In *46th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 132 of *LIPICs*, pages 55:1–55:15, 2019. doi:10.4230/LIPICs.ICALP.2019.55.
- 18 Massimo Equi, Veli Mäkinen, and Alexandru I. Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. *Theor. Comput. Sci.*, 975:114128, 2023. doi:10.1016/J.TCS.2023.114128.
- 19 Massimo Equi, Veli Mäkinen, Alexandru I. Tomescu, and Roberto Grossi. On the complexity of string matching for graphs. *ACM Trans. Algorithms*, 19(3):21:1–21:25, 2023. doi:10.1145/3588334.
- 20 Massimo Equi, Tuukka Norri, Jarno Alanko, Bastien Cazaux, Alexandru I. Tomescu, and Veli Mäkinen. Algorithms and complexity on indexing founder graphs. *Algorithmica*, 85(6):1586–1623, 2023. doi:10.1007/S00453-022-01007-W.
- 21 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1):4, 2009. doi:10.1145/1613676.1613680.
- 22 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.
- 23 Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009. doi:10.1016/j.tcs.2009.09.012.
- 24 Travis Gagie, Aleksi Hartikainen, Kalle Karhu, Juha Kärkkäinen, Gonzalo Navarro, Simon J. Puglisi, and Jouni Sirén. Document retrieval on repetitive string collections. *Information Retrieval Journal*, 20(3):253–291, 2017. doi:10.1007/s10791-017-9297-7.
- 25 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017. doi:10.1016/J.TCS.2017.06.016.
- 26 Erik Garrison, Jouni Sirén, Adam M. Novak, Glenn Hickey, Jordan M. Eizenga, Eric T. Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F. Lin, Benedict Paten, and Richard Durbin. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 36(9):875–879, 2018. doi:10.1038/nbt.4227.
- 27 Daniel Gibney and Sharma V. Thankachan. On the complexity of recognizing wheeler graphs. *Algorithmica*, 84(3):784–814, 2022. doi:10.1007/S00453-021-00917-5.
- 28 Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Softw. Pract. Exp.*, 44(11):1287–1314, 2014. doi:10.1002/SPE.2198.
- 29 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. SODA 2003*, pages 841–850. SIAM, 2003. URL: <http://dl.acm.org/citation.cfm?id=644108.644250>.
- 30 Roberto Grossi, Costas S. Iliopoulos, Chang Liu, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, Giovanna Rosone, Fatima Vayani, and Luca Versari. On-line pattern matching on similar texts. In *Proc. CPM 2017*, volume 78 of *LIPICs*, pages 9:1–9:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.CPM.2017.9.
- 31 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.
- 32 Lin Huang, Victoria Popic, and Serafim Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):i361–i370, 2013. doi:10.1093/bioinformatics/btt215.
- 33 Songbo Huang, T.W. Lam, W.K. Sung, S.L. Tam, and S.M. Yiu. Indexing similar DNA sequences. In *Proc. AAIM 2010*, volume 6124 of *LNCS*, pages 180–190. Springer, 2010. doi:10.1007/978-3-642-14355-7_19.
- 34 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 *Proc. WABI 2016*, volume 9838 of *LNCS*, pages 222–233. Springer, 2016. doi:10.1007/978-3-319-43681-4_18.
- 35 Giovanni Manzini. XBWT tricks. In *Proc. SPIRE 2016*, volume 9954 of *LNCS*, pages 80–92. Springer, 2016. doi:10.1007/978-3-319-46049-9_8.
 - 36 Miguel A. Martínez-Prieto, Nieves Brisaboa, Rodrigo Cánovas, Francisco Claude, and Gonzalo Navarro. Practical compressed string dictionaries. *Information Systems*, 56:73–108, 2016. doi:10.1016/j.is.2015.08.008.
 - 37 J. Ian Munro. Tables. In *Proc. FSTTCS 1996*, volume 1180 of *LNCS*. Springer, 1996. doi:10.1007/3-540-62034-6_35.
 - 38 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010. doi:10.1089/cmb.2009.0169.
 - 39 Joong Chae Na, Hyunjoon Kim, Seunghwan Min, Heejin Park, Thierry Lecroq, Martine Léonard, Laurent Mouchard, and Kunsoo Park. FM-index of alignment with gaps. *Theoretical Computer Science*, 710:148–157, 2018. doi:10.1016/j.tcs.2017.02.020.
 - 40 Joong Chae Na, Hyunjoon Kim, Heejin Park, Thierry Lecroq, Martine Léonard, Laurent Mouchard, and Kunsoo Park. FM-index of alignment: A compressed index for similar strings. *Theoretical Computer Science*, 638:159–170, 2016. doi:10.1016/j.tcs.2015.08.008.
 - 41 Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *Proc. SPIRE 2010*, volume 6393 of *LNCS*, pages 347–358. Springer, 2010. doi:10.1007/978-3-642-16321-0_36.
 - 42 Daisuke Okanohara and Kunihiro Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. ALENEX 2007*, pages 60–70. SIAM, 2007. doi:10.1137/1.9781611972870.6.
 - 43 Nicola Rizzo, Massimo Equi, Tuukka Norri, and Veli Mäkinen. Elastic founder graphs improved and enhanced. *Theor. Comput. Sci.*, 982:114269, 2024. doi:10.1016/J.TCS.2023.114269.
 - 44 Kunihiro Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007. doi:10.1016/j.jda.2006.03.011.
 - 45 Jouni Sirén. Sampled longest common prefix array. In *Proc. CPM 2010*, volume 6129 of *LNCS*, pages 227–237. Springer, 2010. doi:10.1007/978-3-642-13509-5_21.
 - 46 Jouni Sirén. Indexing variation graphs. In *Proc. ALENEX 2017*, pages 13–27. SIAM, 2017. doi:10.1137/1.9781611974768.2.
 - 47 Jouni Sirén, Erik Garrison, Adam M. Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinformatics*, 36(2):400–407, 2020. doi:10.1093/bioinformatics/btz575.
 - 48 Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing finite language representation of population genotypes. In *Proc. WABI 2011*, volume 6833 of *LNCS*, pages 270–281. Springer, 2011. doi:10.1007/978-3-642-23038-7_23.
 - 49 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. doi:10.1109/TCBB.2013.2297101.
 - 50 Esko Ukkonen. Finding founder sequences from a set of recombinants. In Roderic Guigó and Dan Gusfield, editors, *Algorithms in Bioinformatics, Second International Workshop, WABI 2002, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2452 of *Lecture Notes in Computer Science*, pages 277–286. Springer, 2002. doi:10.1007/3-540-45784-4_21.
 - 51 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005. doi:10.1016/J.TCS.2005.09.023.