Pangenome Graph Indexing via the Multidollar-BWT

Davide Cozzi
University of Milano-Bicocca, Milan, Italy

Brian Riccardi
University of Milano-Bicocca, Milan, Italy

Luca Denti
Comenius University, Bratislava, Slovakia

Simone Ciccolella
University of Milano-Bicocca, Milan, Italy

Kunihiko Sadakane
University of Tokyo, Japan

Paola Bonizzoni
University of Milano-Bicocca, Milan, Italy

Abstract

Indexing pangenome graphs is a major algorithmic challenge in computational pangenomics, a recent and active research field that seeks to use graphs as representations of multiple genomes. Since these graphs are constructed from whole genome sequences of a species population, they can become very large, making indexing one of the most challenging problems.

In this paper, we propose `gindex`, a novel indexing approach to solve the Graph Pattern Matching Problem based on the multidollar-BWT. Specifically, `gindex` aims to find all occurrences of a pattern in a sequence-labeled graph by overcoming two main limitations of GCSA2, one of the most widely used graph indexes: handling queries of arbitrary length and scaling to large graphs without pruning any complex regions. Moreover, we show how a smart preprocessing step can optimize the use of multidollar-BWT to skip small redundant sub-patterns and enhance `gindex`'s querying capabilities.

We demonstrate the effectiveness of our approach by comparing it to GCSA2 in terms of index construction and query time, using different preprocessing modes on three pangenome graphs: one built from Drosophila genomes and two produced by the Human Pangenome Reference Consortium.

The results show that `gindex` can scale on human pangenome graphs – which GCSA2 cannot index using large amounts of RAM – with acceptable memory and time requirements. Moreover, `gindex` achieves fast query times, although not as fast as GCSA2, which may produce false positives.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Multidollar-BWT, Graph Index, Graph Pattern Matching, Pangenome Graph

Digital Object Identifier 10.4230/LIPIcs.SEA.2025.13

Supplementary Material *Software (Source code)*: `https://github.com/dlcgold/graph_index.git` archived at `swh:1:dir:453dcdfdd6a0f323fdcc387312bc4c505d1aab3f`

Funding This research work was initially developed in collaboration with S.K. during a secondment of P.B, B.R. and D. C. at the University of Tokyo in June 2024, under the project PANGAIA. L. D. has received funding from the European Union's Horizon programme under the Horizon Europe grant agreement (ASVA-CGR No. 101180581). P.B., B.R, D.C. and S.C., have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement PANGAIA No. 872539. and ITN ALPACA N.956229. P.B. is also supported by the grant MIUR 2022YRB97K, PINC, Pangenome Informatics: from Theory to Applications, funded by the EU, Next-Generation EU, Mission 4. This work was supported by project 101160008 (FORGENOM II) funded by the Horizon Europe program.

1 Introduction

The indexing of a pangenome graph for pattern matching is a fundamental goal in computational pangenomics. Formally, a pangenome graph can be described as a directed graph with nodes labeled by strings. In the framework of computational biology, a pangenome graph aims to represent the multiple pairwise comparison of genomic sequences. It has been proposed as a new paradigm that replaces the traditional view of a reference genome as a linear sequence, to a sequence labeled graph able to summarize the main variations among a population of evolutionary-related sequences or genomes [1]. Indexing and querying a pangenome graph is a fundamental step for developing procedures that work with a graph in place of a sequence linear reference genome. The problem of indexing and searching general graphs is well-studied both from a theoretical and an application-oriented perspective [6, 7].

In this direction, the Burrows-Wheeler Transform (BWT) [3] plays a crucial role in developing an index for a pangenome graph, as GCSA [23] the first pangenome graph indexing is an extension of the BWT to query paths in a sequence labeled graph. Let us briefly recall that the BWT of a text T is a reversible permutation of symbols of T that allows compressing the original text T while it uses the self-indexing scenario to perform efficiently in time and space queries over the text T. While the original idea of GCSA [23] is indexing the paths of an automata representing the pangenome graph, a later implementation, GCSA2 [21, 22], represents a k-mer index as a succinct de-Bruijn graph, inspired by the well-known BOSS data structure [2]. Variants of GCSA2 have been further developed based on specific goals such as the Graph BWT [19], in short GBWT, aiming to query haplotypes in a pangenome graph. In a general formal setting, a pangenome graph can be modeled as a node-labelled graph.

In this paper, we propose `gindex` a novel method for indexing a sequence labeled graph that is based on the BWT for indexing the collection of labels. More precisely, we face the problem of finding all occurrences of a pattern P in a generic graph G, where such occurrences are given by providing either the path where the pattern matches the concatenation of their labels or the initial node where the match of the pattern P to a path label starts.

The main advantages of `gindex` rely on a simple and clean theoretical modeling of the graph combined with a deep insight into several aspects of the implementation of the method. One of these remarkable contributions regards the use of a preprocessing step to avoid redundant operations on short strings that occur many times in the input graph. Furthermore an original contribution of `gindex` is the idea of precomputing a *cache* of the first search intervals - which are the most complex - allowing for a noticeable improvement of querying time at the expense of negligible memory usage.

Moreover, `gindex` can perform space-efficient indexing even on large graphs because the construction is not affected by complex regions inside the graph, while state-of-the-art tools struggle when dealing with such regions.

A main advantage in terms of queries of `gindex` w.r.t. GCSA2 is that `gindex` does not pose any limitation on the length of queries. This is remarkable when dealing with particularly long queries, as those represented by long reads generated by the new sequencing technologies, such as *PacBio* and *Nanopore*. In particular, `gindex` is able to scale on large pangenome graphs such as the ones built by the *Human Pangenome Reference Consortium* [24] built from a large collection of human genomes [14]. In particular `gindex` allows indexing of and fast querying on large graphs, even on machines with limited amount of available memory, which is currently not possible with prevailing indexing tools.

During the development of `gindex` a similar indexing for *Elastic Degenerate Texts* has recently been proposed [5] using a generalization of the extended-BWT, that is the BWT for a collection of sequences. We note that Elastic Degenerate Texts are a restriction of directed acyclic labeled graphs, and that the similarities are related only to technique of backward-intervals merging.

2 Preliminaries

2.1 Pattern matching on strings

Let $\Sigma = \{\mathsf{c}_1, \ldots, \mathsf{c}_\sigma\}$ be a finite, ordered alphabet of cardinality $\sigma = |\Sigma|$, and let $\$, \$_1, \$_2, \ldots$ be an infinite number of terminal symbols. Let $\prec$ denote the lexicographic order; we assume that $\$ \prec \$_1 \prec \$_2 \prec \cdots \prec \mathsf{c}_1 \prec \ldots \prec \mathsf{c}_{|\Sigma|}$. We define a string S over Σ to be a concatenation of n symbols $S = S[1..n]$. We denote the i-th prefix of S as $S[1..i]$, the i-th suffix as $S[i..n]$, and the substring spanning positions i through j as $S[i..j]$. The set of all non-empty strings made of symbols from Σ is denoted by Σ^+. If P and T are two strings, we say that P occurs in T at position $1 \leq i \leq |T|$ (or, equivalently, that i is an occurrence of P in T) if $T[i..i + |P| - 1] = P$.

Given an input string S ending with symbol $\$ \notin \Sigma$, we consider all rotations of S in lexicographical order. We denote the matrix of these rotations, which is commonly referred to as the *Burrows-Wheeler Transform Matrix* [3], as $\mathcal{M}$. Since the rows of $\mathcal{M}$ are stored lexicographically all occurrences of any pattern P occur together in a contiguous range, implying that finding all occurrences of P in S correspond to finding the contiguous range of rows beginning with P. Although $\mathcal{M}$ requires $\mathcal{O}(n^2)$-space to be stored, it is sufficient to store and use only the first column (denoted as F) and the last column (denoted as L) of $\mathcal{M}$: this crucial property is known in the literature as the LF-mapping [8], which maps the i-th occurrence of a symbol c in L to the i-th occurrence of a symbol c in F. In fact, such two occurrences of c in columns F and L, correspond to the same symbol in the original text T.

The *Burrows-Wheeler Transform* of S, denoted as BWT_S, is in fact column L. We denote BWT_S as simply BWT if there is no ambiguity. The starting positions of all sorted rotations is a permutation of $\{1, \ldots, n\}$, and is referred to as the *suffix array* [17] of S since it is equivalent to the lexicographical sorting of all the suffixes of S. We denote this by SA_S, or just SA if S is clear from the context. It is not difficult to see that $\mathsf{BWT}[i]$ is the symbol preceding $\mathsf{F}[i]$ in S and thus BWT can be computed from the suffix array: $\mathsf{BWT}[i] = S[\mathsf{SA}[i] - 1]$, where, due to the rotational nature of the transform, we set $\mathsf{BWT}[i] = \$$ in case of $\mathsf{SA}[i] = 1$.

A classic problem in computer science is the string matching problem: given strings T (the text) and P (the pattern), the goal is to count and/or locate all occurrences of P in T. Arguably one of the most successful ideas to solve this problem, especially in the presence of large texts, is the celebrated FM-index [8] which heavily relies on the LF-mapping. Consider the interval $[i..j]$ of rows of $\mathcal{M}$ that start with P. As already mentioned above, $\mathsf{BWT}_T[i..j]$ corresponds to the symbols preceding $\mathsf{F}[i..j]$ in T, hence, for any $\mathsf{c} \in \Sigma$, the interval $[i'..j']$ of rows that start with $\mathsf{c} \cdot P$ can be retrieved via the LF-mapping. This operation, which is referred to c-*backward step* or c-*backward extension* (and for which we will omit the c-prefix when there is no ambiguity), can be implemented to run in either constant time [8] or, in case one would like to lessen the memory burden, in $\mathcal{O}(\log \sigma)$ using suitable data-structures [10]. In the following, any such interval $[i..j]$ in BWT will also be called c-*backward interval*, and the complete sequence of backward steps to retrieve the entire P as the *backward search*.

There has been extensive work in the literature to generalize the Burrows-Wheeler Transform from single strings to *collections* of strings. We will use one such generalization, namely the *Multidollar Burrows-Wheeler Transform* [18, 4].

▶ **Definition 1** (Multidollar-BWT). *The Multidollar-*BWT *of a multiset of strings* $X = \{x_1, \ldots, x_h\}$ *is the Burrows-Wheeler Transform of* $T_X = x_1\$_1x_2\$_2 \ldots x_h\$_h$*, still denoted by* BWT.

The following two simple properties, which are direct consequences of Definition 1 and which are discussed in Example 9, will be used in our algorithm.

▶ **Property 2.** *Let* $T = x_1\$_1 \cdots x_h\$_h$*, and let* $1 \leq i, j \leq |T|$ *be such that* $\mathsf{BWT}[i] = \$_j$*. If* $j = h$*, then* $T[\mathsf{SA}[i]] = x_1[1]$*; otherwise,* $T[\mathsf{SA}[i]] = x_{j+1}[1]$.

▶ **Property 3.** *Let* $T = x_1\$_1 \cdots x_h\$_h$*. Then, for every* $1 \leq i \leq h$ *it holds that* $\mathsf{BWT}[i] = x_i[|x_i|]$*. That is, the first* h *symbols of* BWT *correspond to the last symbols of* $x_1, \ldots, x_h$.

2.2 Pattern matching on graphs

In this paper, we are concerned with a more general version of the pattern matching problem where a pattern is searched inside a labelled graph.

▶ **Definition 4** (Node-Labelled Graph). *A* node-labelled graph *is a triplet* $G = (V, E, \ell)$ *such that* V *is a set of* nodes*,* $E \subseteq V \times V$ *is a set of* directed edges*, and* $\ell : V \to \Sigma^+$ *is a* labelling function *that assigns to each node* $v \in V$ *a* label $\ell(v) \in \Sigma^+$.

In the following, whenever we mention the term graph we are assuming that it is node-labelled. We extend the labelling function $\ell(\cdot)$ to any path $\pi = (v_1, \ldots, v_k)$ by simply representing $\ell(\pi) = \ell(v_1) \cdots \ell(v_k)$.

In this context, in order to locate where a pattern P occurs in the graph G we define two notions: (i) the *coarse* occurrence, in which we are interested only in the starting node (say v) of a path where P occurs; (ii) the *fine* occurrence, where we are also interested in the full path where P occurs. Formally:

▶ **Definition 5** (Coarse and Fine Occurrences). *Let* $G = (V, E, \ell)$ *be a graph, and* P *be a pattern. A* fine occurrence *of* P *in* G *is a path* $\pi = (v_1, v_2, \ldots, v_k)$ *such that* P *occurs in* $\ell(\pi)$*, starting in* v_1 *and ending in* v_k*. In such a case, we also say that the sole node* v_1 *is a* coarse occurrence *of* P *in* G.

Notice that, in contrast to the string-against-string matching, both a fine and a coarse occurrence can be testified by more than one path.

The goal of this work is to solve the following:

▶ **Problem 6** (Coarse (Fine) Graph Pattern Matching Problem). *Given as input a graph* $G = (V, E, \ell)$*, and a pattern* P*, the goal is to return as output the set of all coarse (fine) occurrences of* P *in* G.

2.3 GCSA2

GCSA (Generalized Compressed Suffix Array) [23] and GCSA2 [21, 22] are BWT and FM-index generalizations developed initially for directed acyclic graphs (GCSA) and later extended to support also cyclic graphs (GCSA2). These approaches were inspired by similar BWT-based indexes for *de Bruijn graphs* [2]. The latter is currently integrated as a part of the `VG toolkit` [9].

Previous experimental evaluations proposed in GCSA2's publication proved the efficiency of this index in terms of performance; according to the authors, GCSA2 performs pattern matching on graph path label concatenations almost as well as classical FM-index approaches on texts of similar sizes [21].

In this paper, we propose an alternative graph indexing method with the goal of overcoming two of the main limitations of GCSA2: (i) the high memory requirements for indexing the graph, especially in the presence of complex regions and (ii) the possibility of obtaining false positive results for long queries. Regarding (i), this behavior is strongly related to the GCSA2 construction algorithm which relies on various iterations of the prefix-doubling algorithm to build the underlying de-Bruijn graph. This approach consists of repetitively joining k-length paths into paths of length $2k$ in the input graph; even with two or three iterations, on large graphs, the algorithm requires hundreds of gigabytes of RAM even if it stores all the possible temporary data on disk, as stated in [21]. With respect to issue (ii), the length of the queries is bounded by k, where k is the k-mer size used in the underlying de-Bruijn graph, currently the maximum size possible is $k = 256$, due to technical details in the implementation[1].

3 The Algorithm

In its full generality, the algorithm we propose can be used to solve Problem 6 for any node-labelled graph. In this work, a particular focus will be given to *pangenome graphs*, which are graphs labelled by strings from the alphabet $\Sigma = \{A, C, G, T\}$. For simplicity of exposition, we assume that the nodes are ordered as $v_1, v_2, \ldots, v_{|V|}$ in a way that for every $i = 1, 2, \ldots, |V|$ it holds $L_i = \ell(v_i)$.

The high level idea is the following. Given the graph G and the pattern P, we start by searching simultaneously in all labels for the existence of an occurrence of P, in a backward-search manner. Whenever a label, say L_i, is *exhausted*, we jump to all incoming neighbours of v_i and continue the search inside their labels. In order to do so efficiently, we first construct $\mathsf{BWT}_{\mathscr{L}}$: the multidollar-BWT of the collection of labels $\mathscr{L} = \{\ell(v) : v \in V\}$, from now on simply denoted by BWT. Our index, `gindex`, will in fact be the pair (BWT, G).

As already said in Section 2.1, we leverage Properties 2 and 3, which we now restate in a way that suits better our graph-oriented application.

▶ **Property 7.** *Let $G = (V, E, \ell)$ be a graph,* BWT *be the multidollar-*BWT *of the collection of labels of G, and $[b, e]$ be a* c*-backward interval obtained after a backward step, for* $\mathsf{c} \in \Sigma$*. Let $b \leq i \leq e$ such that* $\mathsf{BWT}[i] = \$_j$*, for some j. Then, if $j = |V|$ label L_1 has been exhausted. Otherwise, label L_{j+1} is exhausted.*

▶ **Property 8.** *Let $G = (V, E, \ell)$ be a graph,* BWT *be the multidollar-*BWT *of the collection of labels of G. Then, the symbols* $\mathsf{BWT}[1], \ldots, \mathsf{BWT}[|V|]$ *correspond to the last symbols of $L_1, \ldots, L_{|V|}$.*

The graph is explored similarly to how we explore the text using the classical single-text BWT, scanning the pattern P from the right to the left. While on a text a backward interval is transformed into a new backward interval via the LF-mapping, this property is lost when dealing with a graph. Indeed, in case a label is exhausted, say L_i, the incoming neighbours of v_i must be explored afterwards: this will create multiple new intervals. For this reason, we will maintain a list $\mathcal{L}$ of all such backward intervals: (i) those obtained by finishing to

[1] `https://github.com/jltsiren/gcsa2`

visit a node in the graph using Property 8, and (ii) at most one additional backward-interval that represents all the occurrences that fall entirely inside a label. The latter is the same backward-interval produced by each iteration in a classical multidollar-BWT where the pattern can be shared between two strings of the concatenation, due to the absence in the pattern alphabet of the dollar symbol that separates them. In the graph context, this interval represents every possible match of the current suffix of the pattern in each label, without falling over two or more labels.

Our approach to solve Problem 6, finding all coarse occurrences, is given in Algorithm 1. It is divided into two functions: the main function Match and the auxiliary procedure MultiExtend.

To query a pattern P, we start by performing the classic backward search algorithm (lines 3–4). MultiExtend$(\mathcal{L}, \mathsf{c})$ is responsible for updating the backward-interval set $\mathcal{L}$. After each interval extension, whose result is possibly added to $\mathcal{L}$ (lines 21–23), we check the presence of the dollar symbols in the set of the backward-intervals $\mathcal{L}$. If any dollar is found, it means that we finished visiting a node label and we need to continue the visit in all the predecessors of that node. According to Property 8, we know the symbols in the BWT that correspond to the last symbols of the predecessor nodes, therefore we add a new interval of size one to $\mathcal{L}$ that covers exactly this index position for each predecessor node, if this predecessor node ends with the next pattern symbol (lines 24–27).

Note that in Algorithm 1 we do not include this additional check for the sake of simplicity, but we leave a comment in its place.

If we want to return all the fine occurrences, we need to augment each backward-interval with the ordered list of the already traversed nodes. This list is updated each time we jump from one node to another. Note that by merging two consecutive backward-intervals I_1 and I_2 into a bigger interval I_3 to disambiguate I_1 and I_2 inside I_3 we would require additional information of all the lists of all the visited nodes, thus greatly incrementing the memory requirement. Not being able to perform merging optimization means having many more backward-intervals that need to be extended at each iteration, resulting in slower performances in practice. Furthermore, with this method if the same interval is visited by different paths we have to add to the backward-intervals set the same interval for each visited path, resulting in redundant backward operations to perform. Notice that in this scenario $\mathcal{L}$ may contain multiple instances of the same backward interval, becoming technically a multiset.

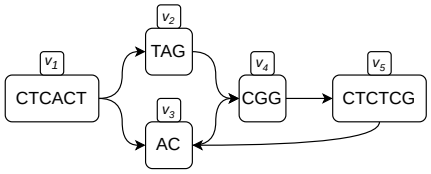
We will not consider this variant in Algorithm 1 description, however, we will present some experimental results in the dedicated section.

On the other hand, if we require just the starting node, we can merge all the consecutive backward-intervals at the end of each iteration, reducing the number of backward extensions needed (line 28). After scanning the entire pattern, we obtain a final set of backward-intervals $\mathcal{L}$ for which we can compute all the starting nodes of the exact matches of P against the graph as follows: for each interval in $\mathcal{L}$, we scan the corresponding set of symbols in the multidollar-BWT representing the starting symbol of a match, however to obtain the node that contains it we need to iteratively perform the classical BWT text reconstruction algorithm until we reach a dollar, from which we can get the corresponding node according to Property 7 (lines 7–16). Conversely, if the set $\mathcal{L}$ is empty then the pattern is not found (lines 5–6).

Algorithm 1 High-level view of all coarse occurrences finding algorithm.

```
1: function Match(P, BWT, G)
2:     L ← {(1, |BWT|)}
3:     for i = |P| downto 1 do        ▷ Scan the pattern and update the backward-intervals set L
4:         L ← MultiExtend(L, P[i])
5:     if L = ∅ then
6:         P not found in G
7:     for all (b, e) ∈ L do                          ▷ Find all starting nodes of the matches
8:         for all i ∈ BWT[b..e − 1] do
9:             c ← BWT[i]
10:            while c ≠ $_j for some j do
11:                (i, i + 1) ← Extend(i, i + 1, c)          ▷ Classic BWT backward extension
12:                c ← BWT[i]
13:            if c = $_{|V|} then
14:                Report occurrence at node v_1
15:            else
16:                Report occurrence at node j + 1, where c = $_j
17:
18: function MultiExtend(L, c)
19:     L_new ← ∅
20:     for all (b, e) ∈ L do
21:         (b′, e′) ← Extend(b, e, c)                    ▷ Classic BWT backward extension
22:         if e′ > b′ then
23:             PushInterval(b′, e′, L_new)               ▷ Add interval (b′, e′) to set L
24:         Nodes ← GetEndNodes(b, e)          ▷ Get ending nodes by dollars in [b..e − 1]
25:         for all v ∈ Nodes do
26:             for all v_a ∈ Predecessors(v) do          ▷ Jump to all the predecessors of v
27:                 PushInterval(a, a + 1, L_new)
28:     L_new ← Merge(L_new)                              ▷ Merge consecutive intervals
29:     return L_new
```

▶ **Example 9.** To better understand our approach we present here a simple example. Consider the following labelled graph:



We extract the five labels from the nodes: L_1 = CTCACT, L_2 = TAG, L_3 = AC, L_4 = CGG, L_5 = CTCTCG. According to the algorithm description, we concatenate all of them in a text T separating them with five unique terminating symbols:

T = CTCACT$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$,

for which we build the corresponding multidollar-BWT. To better visualize the approach, we show the whole multidollar-BWT, including the rotations matrix and highlighting the F array, which is the first column of the rotation matrix and contains all the symbols of T in lexicographical order.

i	F		**BWT**
1	$\$_1$	TAG$\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$CTCAC	**T**
2	$\$_2$	AC$\$_3CGG\$_4$CTCTCG$\$_5$CTCACT$\$_1$TA	**G**
3	$\$_3$	CGG$\$_4$CTCTCG$\$_5$CTCACT$\$_1TAG\$_2$A	**C**
4	$\$_4$	CTCTCG$\$_5$CTCACT$\$_1TAG\$_2AC\$_3$CG	**G**
5	$\$_5$	CTCACT$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTC	**G**
6	A	C$\$_3CGG\$_4$CTCTCG$\$_5$CTCACT$\$_1$TAG	$\$_2$
7	A	CT$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$CT	**C**
8	A	G$\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$CTCACT$\$_1$	**T**
9	C	$\$_3CGG\$_4$CTCTCG$\$_5$CTCACT$\$_1TAG\$_2$	**A**
10	C	ACT$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$C	**T**
11	C	G$\$_5$CTCACT$\$_1TAG\$_2AC\$_3CGG\$_4$CTC	**T**
12	C	GG$\$_4$CTCTCG$\$_5$CTCACT$\$_1TAG\$_2$AC	$\$_3$
13	C	T$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$CTC	**A**
14	C	TCACT$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTCG	$\$_5$
15	C	TCG$\$_5$CTCACT$\$_1TAG\$_2AC\$_3CGG\$_4$C	**T**
16	C	TCTCG$\$_5$CTCACT$\$_1TAG\$_2AC\$_3$CGG	$\$_4$
17	G	$\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$CTCACT$\$_1$T	**A**
18	G	$\$_4$CTCTCG$\$_5$CTCACT$\$_1TAG\$_2AC\$_3$C	**G**
19	G	$\$_5$CTCACT$\$_1TAG\$_2AC\$_3CGG\$_4$CTCT	**C**
20	G	G$\$_4$CTCTCG$\$_5$CTCACT$\$_1TAG\$_2AC\$_3$	**C**
21	T	$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$CTCA	**C**
22	T	AG$\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$CTCACT	$\$_1$
23	T	CACT$\$_1TAG\$_2AC\$_3CGG\$_4$CTCTCG$\$_5$	**C**
24	T	CG$\$_5$CTCACT$\$_1TAG\$_2AC\$_3CGG\$_4$CT	**C**
25	T	CTCG$\$_5$CTCACT$\$_1TAG\$_2AC\$_3CGG\$_4$	**C**

To see the extension step consider for example the interval [13..17) and suppose we want to extend it with a certain pattern symbol $\mathsf{c} \in \Sigma$. We have $\$_4$ and $\$_5$ in the BWT, marking that we can reach, respectively, the beginning of node 5 and node 1. Node 1 has no

predecessors while node 5 has only node 4, therefore we can add the interval [4..4] to our backward-intervals $\mathcal{L}$ set and continue the backward search. Finally, we need to add the extension of [13..16] by σ to $\mathcal{L}$ to continue the pattern search.

3.1 Complexity analysis

Consider now one step in Algorithm 1. The complexity of this step scale in the size of the backward-intervals set, i.e. we have to perform one backward-extension for each interval. If we have merged all the consecutive intervals, solving coarse Problem 6, in the worst case we need to consider $|\mathsf{BWT}|/2$ for the next step. On the other hand, if we consider the backward-intervals multiset in the case of fine occurrences we can not bind its cardinality, making it difficult to establish a time and space complexity bound for finding all fine occurrences.

Focusing on the first case, having to iterate the pattern P and having to perform at most $|\mathsf{BWT}|/2$ extension for each symbol of P, we have in total $|P|(|\mathsf{BWT}|/2)$ backward steps. In addition, the time required to retrieve the ending nodes in each backward-interval and their predecessors is bounded by the number of possible backward intervals multiplied by the number of nodes, i.e. $(|\mathsf{BWT}|/2) \cdot |V|$. Having $|\mathsf{BWT}| \gg |V|$, we can conclude that our algorithm runs in $\mathcal{O}(|P||\mathsf{BWT}|B)$ time, where B is the time of a single backward step. Regarding space, we are bounded by the size of the BWT and the size of the graph. We need to store the multidollar-BWT index, which requires $\mathcal{O}(|\mathsf{BWT}|)$ space, the graph adjacent list, which requires $\mathcal{O}(|V| + |E|)$ space, and the set of backward-intervals, which requires $\mathcal{O}(|\mathsf{BWT}|)$ space.

3.2 Preprocessing

Recalling how we designed our approach, after the first iteration we will add to the backward-intervals set one interval for each predecessor of every node that starts with the last pattern symbol. In fact, without considering the merging step or any further optimization, we have to check all the dollar symbols in the backward-interval we get after the first extension. In Example 9, suppose that the pattern ends with the symbol C. With the first backward-extension, we fall into the interval $[9, 17)$, which contains three dollars: $\$_3$, $\$_5$ and $\$_4$. By Property 7, C is the first symbol of labels L_4, L_5 and L_1 and we need to add one interval for each predecessor of these three nodes.

We can expect the cardinality of this new set of backward-intervals to be $\approx |V|/|\Sigma|$, assuming a uniform distribution of the first symbols on the labels L. Iteration by iteration the size of the set decreases exponentially, having discarded all the mismatching paths. In Figure 1 we show this behaviour. One of the main bottlenecks of this algorithm is the extension of the backward-intervals set in the first iterations, as the cardinality of the set is exponentially larger when we are searching short suffixes of the pattern. To alleviate this issue and allow to scale on large pangenome graphs, such as the ones from HPRC, we implemented a preprocessing step that computes the $\mathcal{L}$ sets for any possible x-length string over the alphabet Σ, for fixed x during the indexing phase. In detail, we precompute all the matches for all $|\Sigma|^x$ strings. We refer to this precomputed set as *cache of length* x. This cache allows us to skip all the first x iterations during the querying phase and access directly the backward-interval sets for $P[|P| - x]$, thus skipping the exponential search. Therefore, trade-off the initial computation time and disk usage to gain a considerable speedup in time during the querying. The entire *cache* needs to be stored on disk and it needs to be entirely loaded into memory when we query the index. Increasing the value of x allows us to skip more iterations, but at the cost of disk and RAM usage, therefore we utilized the results from Figure 1 to select a threshold that gives us a compromise.

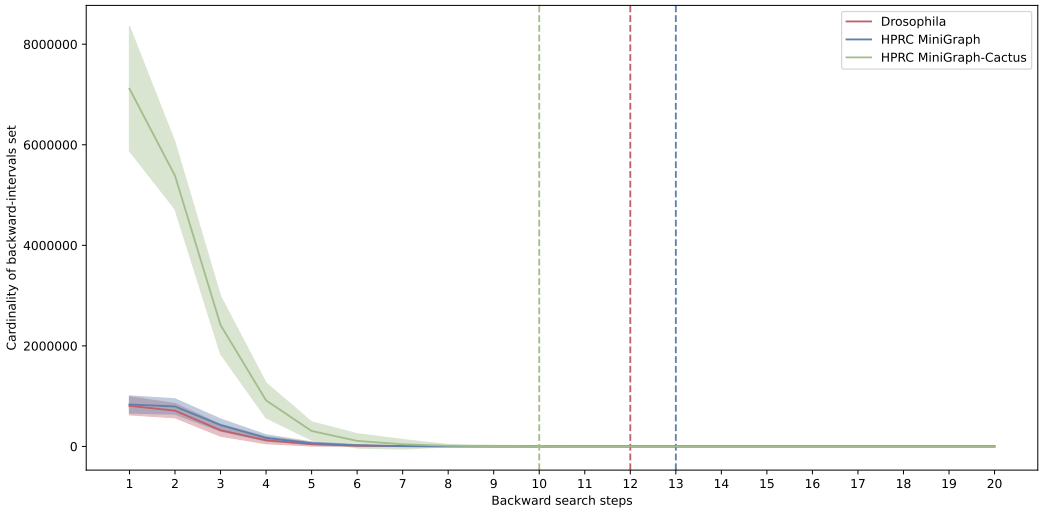


Figure 1 Average cardinality of backward-intervals set $\mathcal{L}$ for the three input graphs. The average is computed from a simulation of 100 random queries. The vertical lines represent the average positions (over the 100 queries) where the cardinality of the backward-interval sets reaches 1 for the first time.

By computing the cache the indexing time is increased, thus to optimize this task, we developed a parallel algorithm that starts from the backward-intervals of each $\sigma \in \Sigma$ which is then extended with each $\sigma' \in \Sigma$; such extensions can be computed in parallel. The process is repeated x times until the matches of all the x-length strings are computed. With a naïve approach we would have many repeated extensions to compute that are not necessary; for example two patterns $CAAA$ and $TAAA$ all have the common AAA suffix, that does not require to be recomputed for each pattern, since the backward-interval set is the same for all common suffixes. Therefore we implemented a tree-based approach in which each iteration extends a suffix with the all the σ without the need to recompute already processed patterns. With this approach, we perform only the necessary $\sum_{i=1}^{x} |\Sigma|^i$ extensions, instead of the all the $x \cdot |\Sigma|^x$; furthermore, each of these extensions can be executed in parallel, since they are independent. Thanks to this optimization we can heavily reduce the impact of the cache computation time.

Figure 1 clearly shows the trend of the cardinality of the backward-interval sets $\mathcal{L}$ - further motivating the use of the caching step - that exponentially decreases at each iteration, reaching a small enough value at around step 5. The vertical lines in the plot indicate the average position where the cardinalities reach a value of 1. In the experimentation, we tested various cache lengths to evaluate its impact, suggesting to use a value of around 7 to obtain a good trade-off between computational time, memory usage and querying speed. Increasing the length too much would also result in computing patterns that would rarely occur when performing pattern matching (due to their higher granularity) that results in wasted resources.

While technically the cardinality of $\mathcal{L}$ is not strictly monotonic decreasing, as in the presence of multiple predecessors with the same ending symbol dimension may increase for a few iterations, it is also likely to drop shortly after or to stabilize to the number of actual occurrences of the pattern.

4 Results

We evaluated the performances of `gindex` by comparing it against GCSA2 on real pangenome-graphs to solve Problem 6 using simulated queries. We evaluated maximum memory usage and execution time for both the graph indexing step and querying step. In the experimental setting, we used `VG toolkit` [9] to index the graphs for GCSA2 and we randomly simulated fixed-length queries extracting them directly from the input graph. In detail, we randomly select a starting node and a starting offset inside its label. Then we visit a path starting from this node, randomly selecting each time the successor node, until the label concatenation reaches the required length.

We implemented `gindex` in C++ using MFMI[2], a RopeBwt2 [12] and RLCSA [20, 16] fork, as multidollar-BWT index implementation. We ran our experiments on a machine equipped with an Intel Xeon CPU E5-4610 v2 (2.30GHz), 256 GB of RAM, 8 GB of swap, and Ubuntu 20.04.6 LTS 64-bit with kernel 5.15.0. `gindex` is compiled with the `-O3` flag. Execution times and memory peaks are retrieved using the Unix `time` command. The source code is available at: `https://github.com/dlcgold/graph_index.git`.

4.1 Input data

In our experiments, we tested three different pangenome graphs:

- **Drosophila pangenome graph** with 10M nodes, 12M edges, a maximum max degree of 15, an average degree of 1.18 and 144M bp. It was built by `VG` using 205 haplotypes [15]. This graph has at most 32 bp per node and has been pruned to delete complex regions with many variants that prevent indexing using GCSA2[3];
- **Human Pangenome Reference Consortium (HPRC) pangenome graph** [14] built using **MiniGraph** [13]. This graph has 12M nodes, 13M edges, a maximum degree of 74, an average degree of 1.02 and 3.2B bp. We slightly modified this graph to have all nodes with at most 256 bp, trying to avoid issues with GCSA2 indexing[4] and fixing the labels' maximal length to the maximum k-mers size supported by the tool (256bp);
- **HPRC pangenome graph** built using **MiniGraph-Cactus** [11]. This graph has 80M nodes, 110M edges, a maximum degree of 39, an average degree of 1.39, and 3.1B bp. No modification has been necessary to use this graph.

4.2 Experiments

Table 5 shows the file sizes on disk of all `gindex` components for the three input graphs. In detail, we need to store the RopeBwt2 index, the topology of the graph as the adjacency list of the predecessors, the label names and the BWT dollars order. The latter is needed because RopeBwt2 does not distinguish different dollars.

To test the efficacy of `gindex` we tested the method on all the pangenome graphs with three different modes of execution:

- **Cache length** x: indexing with a cache computation of length x and returning the starting nodes of each match (coarse occurrence);
- **No Cache**: indexing without using any cache and returning the starting nodes of each match (coarse occurrence);

[2] `https://github.com/ldenti/mfmi.git`
[3] `https://github.com/vgteam/vg/issues/4404`
[4] `https://github.com/jltsiren/gcsa2/issues/44`

- **Full Path**: indexing without using any cache and returning the complete paths that match the query (fine occurrence).

These settings would allow us to explore in detail the properties of both the method and its implementation.

Note that we do not perform a full path match with caching mode because this would require to include in the cache also all the possible paths in the precomputed intervals, therefore greatly affecting memory usage, computation time and implementation complexity without a clear advantage, considering the fact that the only other available implementation (to the best of our knowledge) GCSA2 only reports the starting node of the match.

4.2.1 Drosophila

Table 1, shows the indexing results for the Drosophila pangenome graph using cache lengths values of 1, 4, 7, 10 and no cache; note that cache 10 is presented for the demonstrative purpose of scaling, we do not expect it to be used in a real scenario. Considering *no cache* as baseline, using cache length 10 results in a 120x increase in computational time, while cache length 7 only a 3x increase. On the other hand, while a disk space increase is inevitable, it is not a noticeable amount, with at most 2.41 GB (for cache 10) which is negligible in modern systems. Regarding RAM usage, increasing the cache length requires more resources, but, as shown in the table, this usage scales smoother than the time requirements, with a maximum value of 24.9 (~3x for length 10 compared to the baseline). The trade-off between indexing time, memory usage and size on the disk indicates that length 7 is most likely the best value for this experiment. For this drosophila graph, all the other `gindex` index files require less than 500MB on disk. In this case, a cache of length 7 seems to be a fair trade-off between execution times and memory requirements.

GCSA2, on smaller graphs like this one, is able to fully index the input pangenome; however, as shown in Table 1, it tends to be not as efficient as `gindex` in terms of both memory usage and indexing time (with the sole exception of cache 10). Such requirements will also affect the experiments on larger graphs. Regarding disk usage, GCSA2 index requires about 1.5GB, resulting in a $\sim$ 3x increase with respect to `gindex` without cache and about 70% of the space necessary for cache length 7.

Table 2 shows the results of the querying phase using 10000 random generated queries for each fixed length of 100, 5050 and 10000. GCSA2 outperforms `gindex` both in time and memory usage, even when using a length 10 cache; however as already pointed out, due to its implementation constraints, GCSA2 also could report some false positive matches for queries longer than 256bp; therefore, in similar cases, the performance limits of GCSA2 are evident only in the index phase and not in the querying phase.

Regarding `gindex`, an increase in the length of the queries does not lead to significant increases in RAM usage or execution time . Short patterns, due to their nature, can have larger amounts of matches in the graph, explaining why in some cases queries of length 100 may require more time, since each of these occurrences, once computed, must also be reported in output.

Memory requirements are naturally increased with larger cache lengths, since it is necessary to load in RAM more precomputed interval sets.

The full path output requires inevitably more time than just outputting the first node, because at each backward-extension it is necessary to follow each individual interval to obtain - at the end - the entire path; however the impossibility of performing intervals merging operation results in lower memory usage, since such operations require auxiliary

Table 1 Index time and max memory result on drosophila pangenome graph.

Tool	Mode	Wall Clock (s)	Max Mem (GB)	Cache Size (GB)
gindex	Cache length 1	282	7.6	0.37
	Cache length 4	344	16.5	1.06
	Cache length 7	877	19.1	1.73
	Cache length 10	30185	24.9	2.41
	No Cache	251	7.5	–
GCSA2	–	3174	20.1	–

data structures. Given its limitations, the full path output is currently only usable on small graphs.

Table 2 Query time and max memory results on drosophila pangenome graph.

Tool	Mode	Query length	Wall Clock (s)	Max Mem (GB)
gindex	Cache length 1	100	128207	4.80
		5050	117212	4.68
		10000	129999	4.68
	Cache length 4	100	6871	6.58
		5050	6219	6.58
		10000	7196	6.58
	Cache length 7	100	1036	9.70
		5050	1576	9.70
		10000	1202	9.70
	Cache length 10	100	887	13.25
		5050	940	13.25
		10000	1130	13.25
	No Cache	100	56008	1.54
		5050	64691	1.50
		10000	66479	1.49
	Full Path	100	69195	1.24
		5050	66636	1.24
		10000	67955	1.24
GCSA2	–	100	1.72	0.61
		5050	52.80	0.61
		10000	96.58	0.61

4.2.2 HPRC graphs

Based on the results on the smaller graph, we decided to compute `gindex` only with no cache and cache length 7, that we found to be a good trade-off between indexing costs and querying speed-up.

Table 3 shows the indexing performance results on HPRC pangenome graphs. On our machine, we were not able to index either of these two graphs using GCSA2 as after more than 20 hours of computation time (on 64 threads) it failed to index the graphs exceeding the available amount of RAM of 256GB. As shown in the table, for MiniGraph pangenome graph, precomputing the cache of length 7 requires only a 1.3x time increase with respect to the baseline (no cache), with a negligible increase in RAM and disk usage.

Table 3 HPRC pangenome graphs indexing results.

HPRC Graph	Mode	Wall Clock (s)	Max Mem (GB)	Cache Size (GB)
MiniGraph	No Cache	5580	42	–
	Cache length 7	7380	45	2.4
MiniGraph-Cactus	No Cache	8880	104	–
	Cache length 7	15540	183	16

Table 4 HPRC graphs query results with 100 queries of various lengths.

HPRC Graph	Mode	Query length	Wall Clock (s)	Max Mem (GB)
MiniGraph	Cache length 7	100	67	13.7
		5050	65	13.7
		10000	68	13.7
	No Cache	100	747	3.1
		5050	849	3.1
		10000	786	3.1
	Full Path	100	741	2.8
		5050	900	2.8
		10000	753	2.8
MiniGraph-Cactus	Cache length 7	100	550	84.6
		5050	469	83.3
		10000	482	83.3
	No Cache	100	5406	11.7
		5050	4535	11.7
		10000	4506	11.7
	Full Path	100	4563	10.7
		5050	4372	10.7
		10000	4486	17.5

Similar behaviours are also present in the pangenome graph computed by MiniGraph-Cactus, where precomputing cache with length 7 requires almost double time and RAM. Note that the size of the graph is not the only aspect to consider for estimating resources, because the topology of the graph - frequency of bubbles and repetitions in the labels - also make an important impact on the computation.

Table 4 shows querying performance results on the HPRC pangenome graphs using 100 randomly generated queries of lengths equal to 100, 5050 and 10000. Running `gindex` with a cache length 7 yields in ~10x querying speed-up at the cost of ~4x more RAM. Finally, the full path output takes about the same time and memory as running `gindex` without cache, except for queries of length 10000 on the MiniGraph-Cactus graph, which required more RAM.

It is clear, Table 4 that in some cases the cache might result in a large amount of memory required, such as in the MiniGraph-Cactus graph, depending on the characteristics of the graph; however, its cost is heavily offset by the benefit in time. We advise the user to test different values of the cache length based on the application context and the resources available.

Table 5 Auxiliary files produced by the methods, GCSA2 failed to compute on the two HPRC graph on our machine.

Method	Drosophila	MiniGraph	MiniGraph-Cactus
`gindex`	`bwt`: 79MB `graph`: 174MB `labels`: 80MB `dollars`: 160MB	`bwt`: 1.5GB `graph`: 200MB `labels`: 99MB `dollars`: 198MB	`bwt`: 1.5GB `graph`: 1.5GB `labels`: 611MB `dollars`: 1.2GB
GCSA2	`gcsa`: 566MB `lcp`: 358MB	– –	– –

5 Conclusions

In this manuscript, we presented `gindex`, a novel graph indexing approach that extends the use of the multidollar-BWT to solve the graph pattern matching problem, in a pangenomics context. In particular, we presented an approach able to scale on human pangenome graphs; additionally, we proposed further optimizations leveraging the use of a preprocessing caching step which allows to avoid the on-the-fly computation of the more expensive steps of our approach, alleviating the running time in the most crucial phase.

Our experimental results show that, when dealing with small graphs, GCSA2 outperforms `gindex` in querying efficiency, albeit requiring a huge amount of time and memory in indexing and retaining its querying constraint on the pattern length. In contrast, our approach scales on much larger pangenome graphs, such as the HPRC ones, which require GCSA2 a very large amount of RAM and time, we were not able to compute them using the available 256GB of RAM in our experimental setting.

Future developments should be devoted to a more succinct representation of the backward-interval set and a different merging approach of consecutive intervals that avoid the loss of information about the full matching path.

Additionally, a different mode of execution that consider matches on a colored graph could be developed; in the context of indexing pangenome graphs a main distinction has been described in the literature between variation and sequence graphs. In variation graphs, differently from sequence graphs, a number of distinguished paths are specified in the input data that could biologically represent distinct haplotypes or distinct genomes. While GCSA2 is not meant for indexing this types of graphs; `gindex` can be extended by tweaking the full path mode to also store the colors of the visited nodes and check whether they match the color of each predecessor node before expanding on it. Optimizing the use of colors is however a critical issue for dealing with variation graphs.

Lastly, it would be interesting to test `gindex` as the base of downstream analysis on human pangenomes, even on lighter machines, leveraging the lower requirements of our approach, even on the analysis of larger graphs.

References

1 Jasmijn A Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangenomics: a tutorial on data structures and their applications. *Natural Computing*, 21(1):81–108, 2022. doi:10.1007/S11047-022-09882-6.

2 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.

3 Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm, 1994.

4 Davide Cenzato and Zsuzsanna Lipták. A survey of bwt variants for string collections. *Bioinformatics*, page btae333, 2024.

5 Lapo Cioni, Veronica Guerrini, and Giovanna Rosone. The burrows-wheeler transform of an elastic-degenerate string. In *Proceedings of the 25th Italian Conference on Theoretical Computer Science, Torino, Italy, September 11-13, 2024*, volume 3811 of *CEUR Workshop Proceedings*, pages 66–80, 2024. URL: https://ceur-ws.org/Vol-3811/paper240.pdf.

6 Massimo Equi, Veli Mäkinen, and Alexandru I Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless seth fails. *Theoretical Computer Science*, 975:114128, 2023. doi:10.1016/J.TCS.2023.114128.

7 Massimo Equi, Veli Mäkinen, Alexandru I Tomescu, and Roberto Grossi. On the complexity of string matching for graphs. *ACM Transactions on Algorithms*, 19(3):1–25, 2023. doi:10.1145/3588334.

8 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

9 Erik Garrison, Jouni Sirén, Adam M Novak, Glenn Hickey, Jordan M Eizenga, Eric T Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F Lin, et al. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature biotechnology*, 36(9):875–879, 2018.

10 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, USA, 2003. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

11 Glenn Hickey, Jean Monlong, Jana Ebler, Adam M Novak, Jordan M Eizenga, Yan Gao, Tobias Marschall, Heng Li, and Benedict Paten. Pangenome graph construction from genome alignments with minigraph-cactus. *Nature biotechnology*, 42(4):663–673, 2024.

12 Heng Li. Fast construction of fm-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014. doi:10.1093/BIOINFORMATICS/BTU541.

13 Heng Li, Xiaowen Feng, and Chong Chu. The design and construction of reference pangenome graphs with minigraph. *Genome biology*, 21:1–19, 2020.

14 Wen-Wei Liao, Mobin Asri, Jana Ebler, Daniel Doerr, Marina Haukness, Glenn Hickey, Shuangjia Lu, Julian K Lucas, Jean Monlong, Haley J Abel, et al. A draft human pangenome reference. *Nature*, 617(7960):312–324, 2023.

15 Trudy FC Mackay, Stephen Richards, Eric A Stone, Antonio Barbadilla, Julien F Ayroles, Dianhui Zhu, Sònia Casillas, Yi Han, Michael M Magwire, Julie M Cridland, et al. The drosophila melanogaster genetic reference panel. *Nature*, 482(7384):173–178, 2012.

16 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.

17 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

18 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the burrows–wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007. doi:10.1016/J.TCS.2007.07.014.

19 Adam M Novak, Erik Garrison, and Benedict Paten. A graph extension of the positional burrows–wheeler transform and its applications. *Algorithms for Molecular Biology*, 12:1–12, 2017. doi:10.1186/S13015-017-0109-9.

20 Jouni Sirén. Compressed suffix arrays for massive data. In *International Symposium on String Processing and Information Retrieval*, pages 63–74. Springer, 2009. doi:10.1007/978-3-642-03784-9_7.

21 Jouni Sirén. Indexing variation graphs. In *2017 Proceedings of the ninteenth workshop on algorithm engineering and experiments (ALENEX)*, pages 13–27. SIAM, 2017. doi:10.1137/1.9781611974768.2.

22 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.

23 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.

24 Ting Wang, Lucinda Antonacci-Fulton, Kerstin Howe, Heather A Lawson, Julian K Lucas, Adam M Phillippy, Alice B Popejoy, Mobin Asri, Caryn Carson, Mark JP Chaisson, et al. The human pangenome project: a global resource to map genomic diversity. *Nature*, 604(7906):437–446, 2022.