# Indexing Variation Graphs[*]

Jouni Sirén[†]

**Abstract**

Variation graphs, which represent genetic variation within a population, are replacing sequences as reference genomes. Path indexes are one of the most important tools for working with variation graphs. They generalize text indexes to graphs, allowing one to find the paths matching the query string. We propose using de Bruijn graphs as path indexes, compressing them by merging redundant subgraphs, and encoding them with the Burrows-Wheeler transform. The resulting fast, space-efficient, and versatile index is used in the variation graph toolkit vg.

## 1 Introduction

Sequence analysis pipelines typically start with mapping the reads from the sequenced genome to a *reference genome* of the same species. As reference genomes are usually assembled from the genomes of a small number of individuals, they are biased towards those individuals. This *reference bias* may affect the results of subsequent analysis, especially when the sequenced individuals are from different populations than the ones behind the reference genome.

*Variation graphs* (also graph genomes, genome graphs, graph references, or reference graphs), which encode the genetic variation within a population as a graph, have been proposed as a solution to the reference bias [37, 28, 6, 8, 29]. These graphs are expected to replace sequences as reference genomes. The shift to graphs will likely take years, as new methods and tools are needed to replace those based on linear references.

The *variation graph toolkit vg* [14] is a community effort to develop such tools. This paper describes *GCSA2*, the path index used in vg. A *path index* is a generalization of text indexes for labeled graphs. Given a query string, the index finds the paths with a label matching the query. Indexing graphs is inherently hard, as the number of paths increases exponentially with path length. The design of a path index is hence a trade-off between maximum query length, index size, query performance, and pruning complex regions of the graph.

Mapping reads to a graph was first investigated by Schneeberger et al. [42]. Sirén et al. developed *GCSA* [43], which generalized the FM-index [10] (a text index based on the Burrows-Wheeler transform [4]) to directed acyclic graphs. GCSA depends on pruning the complex regions, as there are no limits on query length. Kim et al. [19] combined GCSA with the HISAT read aligner [20]. The result was the first practical graph-based read aligner, though it uses the graph for more accurate mapping to a linear reference.

*De Bruijn graphs* can be used as $k$-mer indexes of other graphs. There are already read aligners based on them [23, 25]. The *succinct de Bruijn graph* of Bowe et. al. [3] encodes the graph with a generalization of the FM-index. Rødland [39] proposed another similar generalization. Succinct de Bruijn graphs can simulate order-$k$ de Bruijn graphs for multiple values of $k$ [2], but they still need to store the graph explicitly for the largest value of $k$. Other representations include compacted [5] and compressed [28] de Bruijn graphs, which represent unary paths in the graph as single nodes or edges. Probabilistic de Bruijn graphs [38] use Bloom filters to support faster queries, at the expense of producing false positives. All these space-efficient representations require several bits per $k$-mer for the graph, and more for mapping back to the indexed graph.

Some path indexes store the graph as a collection of sequences. BWBBLE [17] uses the powerset alphabet for encoding substitutions and creates new sequences with a sufficient amount of context for other variants. vBWT [26] encodes variant sites explicitly in the sequence as $X(A|B|C)Y$, using distinct separator symbols for each site. Queries in both BWBBLE and vBWT are slower than in ordinary FM-indexes, as variant sites force the search to branch. The hypertext index [44] works with graphs that have string labels on the nodes. The strings are indexed using an FM-index. Partial matches in the strings are combined into full matches with range queries in the edge matrix. While matches crossing one edge are easy to find, the approach becomes impractical with matches crossing multiple edges.

There are also structures (e.g. [18, 46, 7, 32, 33]) using graphs as a space-efficient way of indexing similar sequences. While the problem is different from indexing the paths in a graph, the techniques used are similar.

The above methods can be classified in three categories based on the data models they use. Many deal

[*]Supported by the Wellcome Trust grant 098051.

[†]Wellcome Trust Sanger Institute, UK. `jouni.siren@iki.fi`.

with graphs arising from *aligned sequences*, assuming a shared global sequence with local variation. GCSA-based methods can index directed acyclic graphs, as well as cyclic graphs that are sufficiently similar to *de Bruijn graphs*. GCSA2, the hypertext index, and indexes based on de Bruijn graphs work with *arbitrary graphs*.

GCSA2 combines ideas from the original GCSA and from succinct de Bruijn graphs. Conceptually it uses a de Bruijn graph as a $k$-mer index of a variation graph. The de Bruijn graph is *pruned* (compressed structurally) by using strings shorter than $k$ characters as nodes, if the shorter strings identify the start nodes of the corresponding paths uniquely. The pruned graph is encoded with a generalization of the FM-index. GCSA2 often uses less space (e.g. less than 1 bit per $k$-mer) than other de Bruijn graph-based indexes, which have to store some information for each $k$-mer explicitly. The index also includes extensions based on suffix trees. The extensions are used for e.g. finding maximal exact matches in the vg read aligner.

The main differences to the original GCSA are:

1. The graph encoding in GCSA2 has been optimized for small alphabets, improving query performance by up to an order of magnitude.

2. The construction algorithm stores the graphs on disk, reducing the memory requirements of building a whole-genome human index from hundreds of gigabytes to tens of gigabytes.

3. GCSA2 can index denser graphs, including cyclic graphs, by limiting maximum query length.

4. GCSA2 extends the functionality of the FM-index with suffix tree operations.

The first two improvements can also be used with the original GCSA. The third point represents a different approach to indexing graphs. It was not possible the with the original GCSA, as the construction algorithm required a prefix-range-sorted graph. The extended functionality depends on the new construction algorithm and on limiting maximum query length.

This paper describes the GCSA2 data structure. Its uses in the vg toolkit will be discussed in the vg paper.

## 2 Background

**2.1 Strings** A *string* $S[0, n-1] = s_0 \cdots s_{n-1}$ of length $|S| = n$ is a sequence of *characters* over an *alphabet* $\Sigma = \{0, \ldots, \sigma-1\}$. *Text* strings $T[0, n-1]$ are terminated by an *endmarker* $T[n-1] = \$ = 0$ not found anywhere else in the text. A *substring* of string $S$ is a sequence of the form $S[i, j] = s_i \cdots s_j$. We call substrings of the type $S[0, j]$ and $S[i, n-1]$ *prefixes* and *suffixes*, respectively, and refer to substrings of length $k$ as $k$-mers. Substring $S[i, j]$ is a *proper* substring of string $S$, if $S \neq S[i, j]$. We say that string $S'$ is a substring of string collection $\mathcal{S}$, if it is a substring of a string $S \in \mathcal{S}$.

Sometimes we consider *infinite* character sequences $S = (s_i)_{i \in Z}$, where set $Z$ is a contiguous infinite subset of $\mathbb{Z}$. The notion of substring generalizes to infinite sequences in a natural way. A substring of an infinite sequence $S$ is *left-infinite* if it extends infinitely to the left, and *right-infinite* if it extends infinitely to the right. A substring of a finite or infinite sequence $S$ is *left-maximal* if it is left-infinite or a prefix, and *right-maximal* if it is right-infinite or a suffix.

We are primarily interested in sequences over the *DNA* alphabet $\{\$, \mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \mathtt{N}\}$. Characters A, C, G, and T are called *bases*, while character N represents an arbitrary or unknown base. The alphabet may contain other technical characters in addition to the endmarker \$. Each character $c$ of the DNA alphabet has a *complement* $\overline{c}$ defined as $\overline{\mathtt{A}} = \mathtt{T}$, $\overline{\mathtt{C}} = \mathtt{G}$, $\overline{\mathtt{G}} = \mathtt{C}$, $\overline{\mathtt{T}} = \mathtt{A}$, and $\overline{c} = c$ for other characters $c$. Given a DNA sequence $S$, its *reverse complement* is the sequence $\overleftarrow{S}$ obtained by reversing the non-technical parts of the sequence and replacing each character with its complement. For example, $\overleftarrow{\mathtt{GATTACA\$}} = \mathtt{TGTAATC\$}$.

Given a string $S[0, n-1]$, we define $S.\mathsf{rank}(i, c)$ as the number of occurrences of character $c$ in the prefix $S[0, i-1]$. We also define

$$S.\mathsf{select}(i, c) = \max\{j \leq n \mid S.\mathsf{rank}(j, c) < i\}$$

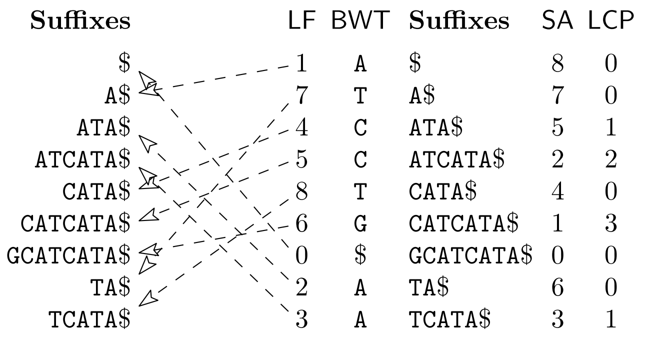
as the position of the occurrence of character $c$ with rank $i > 0$.[1] A *bitvector* is a binary sequence supporting efficient $\mathsf{rank}/\mathsf{select}$ queries. *Wavelet trees* [16] are space-efficient data structures that use bitvectors to support $\mathsf{rank}/\mathsf{select}$ queries on arbitrary strings.

Let $S$ be a string and $S'$ be a string or an infinite character sequence over alphabet $\Sigma$. We say that sequences $S$ and $S'$ *prefix-match*, if $S$ is a prefix of $S'$ or $S'$ is a prefix of $S$. Set $\mathcal{S}$ of strings is *prefix-free*, if no two strings $S, S' \in \mathcal{S}$ (with $S \neq S'$) prefix-match.

**2.2 Text Indexes** The *suffix tree* [47] is the most fundamental full-text index supporting substring queries. It is formed by taking the suffixes of the text, storing them in a trie, and compacting unary paths in the trie into single edges. Although fast and versatile, suffix trees are impractical with large texts, as they require much more space than the text itself.

*Suffix arrays* (SA) [27] were introduced as a space-efficient alternative to the suffix tree. The suffix array

[1] These definitions are used in the SDSL library [15]. We assume for convenience that $S.\mathsf{select}(0, c) = -1$.



| Suffixes | LF | BWT | Suffixes | SA | LCP |
|---|---|---|---|---|---|
| $ | 1 | A | $ | 8 | 0 |
| A$ | 7 | T | A$ | 7 | 0 |
| ATA$ | 4 | C | ATA$ | 5 | 1 |
| ATCATA$ | 5 | C | ATCATA$ | 2 | 2 |
| CATA$ | 8 | T | CATA$ | 4 | 0 |
| CATCATA$ | 6 | G | CATCATA$ | 1 | 3 |
| GCATCATA$ | 0 | $ | GCATCATA$ | 0 | 0 |
| TA$ | 2 | A | TA$ | 6 | 0 |
| TCATA$ | 3 | A | TCATA$ | 3 | 1 |

Figure 1: LF-mapping, BWT, suffix array, and LCP array for text GCATCATA$.

of text $T[0, n-1]$ is an array of pointers $\mathsf{SA}[0, n-1]$ to the suffixes of the text in *lexicographic order*. We find the occurrences of *pattern* string $X$ in the text in $\mathsf{O}(|X| \log n)$ time by using *binary search* in the suffix array. The suffix array requires $n \log n$ bits of space in addition to the text, while its functionality is more limited than that of the suffix tree. See Figure 1 for an example of the suffix array and related structures.

The *Burrows-Wheeler transform (BWT)* [4] of text $T[0, n-1]$ is a permutation $\mathsf{BWT}[0, n-1]$ such that $\mathsf{BWT}[i] = T[(\mathsf{SA}[i] - 1) \bmod n]$. Given the *lexicographic rank* $i$ of suffix $T[\mathsf{SA}[i], n-1]$, we can use *LF-mapping* on the BWT to find the lexicographic rank of the previous suffix $T[(\mathsf{SA}[i] - 1) \bmod n, n-1]$. Let

$$\mathsf{LF}(i) = \mathsf{C}[\mathsf{BWT}[i]] + \mathsf{BWT}.\mathsf{rank}(i, \mathsf{BWT}[i]),$$

where $\mathsf{C}[c]$ is the number of occurrences of characters $c' < c$ in the BWT. Then $\mathsf{SA}[\mathsf{LF}(i)] = (\mathsf{SA}[i] - 1) \bmod n$. We generalize the definition to any character $c \in \Sigma$:

$$\mathsf{LF}(i, c) = \mathsf{C}[c] + \mathsf{BWT}.\mathsf{rank}(i, c).$$

Let $X$ be a string. If there are $i$ suffixes $S'$ of text $T$ such that $S' < X$ in lexicographic order, then there are $\mathsf{LF}(i, c)$ suffixes $S'$ such that $S' < cX$.

We can use the BWT as a space-efficient text index. The *FM-index* [10] combines a representation of the BWT supporting rank/select queries, the C array, and a set of *sampled pointers* from the suffix array. It uses *backward searching* to find the *lexicographic range* of suffixes matching pattern $X$ (having $X$ as a prefix). If the lexicographic range matching suffix $X[i+1, |X|-1]$ of the pattern is $\mathsf{SA}[sp, ep]$, the range matching suffix $X[i, |X|-1]$ is $\mathsf{SA}[\mathsf{LF}(sp, X[i]), \mathsf{LF}(ep+1, X[i]) - 1]$. Matching the entire pattern takes $\mathsf{O}(|X|)$ rank queries.

We use the sampled suffix array pointers for finding the text positions containing the occurrences. If $\mathsf{SA}[i]$ is not sampled, we iterate $\mathsf{LF}(i)$ until we find a sampled pointer. If we find a sample at $\mathsf{SA}[\mathsf{LF}^k(i)]$, we know that

$$\mathsf{SA}[i] = (\mathsf{SA}[\mathsf{LF}^k(i)] + k) \bmod n.$$

If we have sampled one out of $d$ suffix array pointers at regular intervals, finding each occurrence takes $\mathsf{O}(d)$ rank queries. If we also sample one out of $d'$ *inverse suffix array* pointers[2], we can *extract* an arbitrary substring $X$ of the text using $\mathsf{O}(|X| + d')$ rank queries.

The *longest-common-prefix array* (LCP array) [27] is an integer array $\mathsf{LCP}[0, n-1]$, where each value $\mathsf{LCP}[i]$ tells the length of the longest common prefix of suffixes $T[\mathsf{SA}[i-1], n-1]$ and $T[\mathsf{SA}[i], n-1]$ (with $\mathsf{LCP}[0] = 0$). If we have the FM-index, the LCP array, and the topology of the suffix tree, we get the *compressed suffix tree*, which supports the full functionality of the suffix tree in a space-efficient manner [40].

## 2.3 Graphs

A *graph* $G = (V, E)$ consists of a set of *nodes* $V = \{0, \ldots, |V|-1\}$ and a set of *edges* $E \subseteq V \times V$. We say that $(u, v) \in E$ is an edge *from* node $u$ *to* node $v$, and assume that the edges are *directed*: $(u, v) \neq (v, u)$ for $u \neq v$. The *indegree* $G.\mathsf{in}(v)$ of node $v$ is the number of *incoming* edges to $v$, while the *outdegree* $G.\mathsf{out}(v)$ is the number of *outgoing* edges from $v$.

The graphs we use are *labeled* with alphabet $\Sigma$: each node $v \in V$ has a *label* $G.\mathsf{label}(v) \in \Sigma$. A *path* in a graph is a sequence of nodes $P = v_0 \cdots v_{|P|-1}$ such that $(v_i, v_{i+1}) \in E$ for $0 \leq i < |P| - 1$. We say that $v_0$ is the *start* node and $v_{|P|-1}$ is the *end* node of the path. The label of path $P$ is the concatenation of node labels $G.\mathsf{label}(P) = G.\mathsf{label}(v_0) \cdots G.\mathsf{label}(v_{|P|-1})$.

If the graph has nodes with indegree or outdegree 0, we add a *source* node $s$ and a *sink* node $t$ to it. To distinguish these technical nodes from the actual nodes, we label them with characters $G.\mathsf{label}(s) = \#$ and $G.\mathsf{label}(t) = \$$, which are not used anywhere else in the graph. We add an edge $(s, v)$ to all nodes $v \in V \setminus \{s\}$ with no incoming edges, and an edge $(v, t)$ from all nodes $v \in V \setminus \{t\}$ with no outgoing edges. We also add edge $(t, s)$ to guarantee that $G.\mathsf{in}(v) \geq 1$ and $G.\mathsf{out}(v) \geq 1$ for all nodes $v \in V$. However, this edge is not considered a real edge, and no path can cross it.

We also consider infinite paths $P = (v_i)_{i \in Z}$, generalizing the definition in a similar way as we did with infinite character sequences in Section 2.1. We say that path $P$ is *left-maximal* if it starts at the source node or extends infinitely to the left; *right-maximal* if it ends at the sink node or extends infinitely to the right; and *maximal* if it is both left-maximal and right-maximal.

[2]The suffix array is a permutation of $\{0, \ldots, n-1\}$, and the inverse suffix array is the inverse permutation.

Given a graph $G = (V, E)$, we may want to reason about the *predecessors* of a node with the given label. Let $v \in V$ be a node and $c \in \Sigma$ be a character. We write $G.\mathsf{pred}(v, c)$ to denote the set of nodes $u \in V$ such that $G.\mathsf{label}(u) = c$ and there is an edge $(u, v) \in E$.

We work with de Bruijn graphs and their generalizations. For that purpose, we define collections of (finite or infinite) sequences suitable for constructing order-$k$ de Bruijn graphs.

Definition 2.1. ($k$-collection) *Let $\mathcal{S}$ be a collection of character sequences over alphabet $\Sigma$, and let $k > 0$ be a parameter value. We say that $\mathcal{S}$ is a* $k$-collection*, if each sequence $S \in \mathcal{S}$ (a) is left-infinite or begins with $\#^k$; (b) is right-infinite or ends with $\$^k$; and (c) contains no other occurrences of characters $\#$ and $\$$.*

Definition 2.2. (de Bruijn graph) *Let $k > 0$, and let $\mathcal{S}$ be a $k$-collection. The order-$k$* de Bruijn graph *of $\mathcal{S}$ is a graph $G = (V, E)$ such that*

- *each node $v_X \in V$ represents a distinct $k$-mer $X$ occurring in $\mathcal{S}$, with $G.\mathsf{label}(v_X) = X[0]$;*
- *each node $v_X \in V$ has a* key $G.\mathsf{key}(v_X) = X$*; and*
- *each edge $(v_X, v_Y) \in E$ represents a $(k+1)$-mer $X[0]Y = XY[k-1]$ occurring in $\mathcal{S}$.*

*We use $\#^k$ as the source node $s$ and $\$^k$ as the sink node $t$, adding the edge $(t, s)$ in the usual way.*

De Bruijn graphs have several properties that make them useful for indexing purposes. Node keys prefix-match the labels of all paths starting from the node. This makes it possible to sort the nodes unambiguously by path labels. Every substring of the $k$-collection is the label of a path in the de Bruijn graph, and every path label of length at most $k + 1$ is a substring of the collection. In Section 3, we develop an index structure based on a generalization of de Bruijn graphs.

**2.4 FM-Index for Graphs** Suffix trees, suffix arrays, and the FM-index can be generalized to index multiple texts. There are also generalizations to other combinatorial structures. The *XBW transform* [9] is an FM-index for *labeled trees*. The nodes of the tree are sorted by the path labels from the node to the root. $\mathsf{BWT}$ stores the labels of the children of each node, while leaf nodes are marked with special characters. If a node has $k$ children, we encode that as a binary sequence $0^{k-1}1$. We concatenate these sequences to form the indegree bitvector $\mathsf{IN}$. The labels of the children of the $i$th node are found in $\mathsf{BWT}[\mathsf{IN}.\mathsf{select}(i, 1) + 1, \mathsf{IN}.\mathsf{select}(i + 1, 1)]$.

The *generalized compressed suffix array* (GCSA) [43] extends the XBW transform to a class of graphs that includes *directed acyclic graphs* and de Bruijn graphs. Before indexing, we transform the graph into an equivalent graph, where the nodes can be sorted unambiguously by the labels of the right-maximal paths starting from them. The transformation increases the size of the graph exponentially in the worst case. In addition to sequences $\mathsf{BWT}$ and $\mathsf{IN}$, GCSA also uses an outdegree bitvector $\mathsf{OUT}$, which is encoded in the same way as $\mathsf{IN}$. LF-mapping uses $\mathsf{select}$ queries on bitvector $\mathsf{IN}$ to map nodes to BWT ranges, ordinary LF-mapping with $\mathsf{BWT}$ to map incoming edges to the corresponding outgoing edges, and $\mathsf{rank}$ queries on bitvector $\mathsf{OUT}$ to map the outgoing edges to the predecessor nodes.

## 3 Path Indexes

A *path index* is a generalization of text indexes for *labeled graphs*. Given a path index for *input graph* $G = (V, E)$, we use the index to find the start nodes $v_0 \in V$ of the paths $P = v_0 \cdots v_{|X|-1}$ *matching* pattern $X$ (paths $P$ with $G.\mathsf{label}(P) = X$).

The proofs of the lemmas can be found in Appendix A.

**3.1 De Bruijn Graphs as Path Indexes** The *$k$-mer index* is the simplest path index. It consists of a set of *key-value pairs* $(X, V_X)$, where $X \in \Sigma^k$ is a $k$-mer and $V_X \subseteq V$ is the set of the start nodes of the paths matching the $k$-mer. If we store the pairs in a hash table, we can quickly search for patterns of length $k$. If we use binary search in a sorted list of pairs, queries become slower, but we gain the ability to search for patterns shorter than $k$ characters. The main drawback of these basic $k$-mer indexes is their size, as they store the key-value pairs explicitly.

We can represent $k$-mer indexes as de Bruijn graphs. For that purpose, we define the de Bruijn graph of graph $G = (V, E)$ by using the collection $\mathcal{S}$ of the labels of the maximal paths in the graph. If sequence $S \in \mathcal{S}$ is the label of path $P = (v_i)_{i \in Z}$, we set $\mathcal{S}.\mathsf{node}(S, i) = v_i$ for all positions $i \in Z$. We transform $\mathcal{S}$ into a $k$-collection by inserting characters $\#$ to the beginning of each non-left-infinite sequence when necessary, and characters $\$$ to the end of each non-right-infinite sequence. If $S[i]$ is a $\#$ we inserted, we set $\mathcal{S}.\mathsf{node}(S, i) = s : j$, where $s \in V$ is the source node and $j$ is the distance to the nearest non-inserted $\#$ in $S$.

Definition 3.1. (de Bruijn graph of a graph) *Let $G$ be a labeled graph, and let $\mathcal{S}$ be the $k$-collection of maximal path labels in $G$. The order-$k$ de Bruijn graph of $\mathcal{S}$ is the order-$k$ de Bruijn graph of graph $G$.*

Let $G' = (V', E')$ be a de Bruijn graph of graph $G = (V, E)$, and let $\mathcal{S}$ be the $k$-collection used to define it. We attach a set of nodes of graph $G$ to each node $v' \in V'$ as a *value* $G'.\mathsf{value}(v)$:

$$\{\mathcal{S}.\mathsf{node}(S,i) \mid S \in \mathcal{S}, S[i,i+k-1] = G'.\mathsf{key}(v)\}.$$

Apart from some technicalities near the source/sink nodes, $G'.\mathsf{value}(v)$ is the set of the start nodes of the paths matching pattern $G'.\mathsf{key}(v)$ in graph $G$. The index produces no *false negatives* (path labels that exist in the input graph but not in the index). There may be *false positives* (path labels that exist in the index but not in the input graph) with patterns longer than $k$, but we can avoid them by *verifying* the results in the input graph. See Figure 2 for an example.

We can encode the de Bruijn graph as a GCSA using $(|V'| + \mathsf{o}(|V'|))(\log \sigma + 2)$ bits. By using a similar sampling scheme for the values as in the FM-index, we get a $k$-mer index that uses a couple of bytes per $k$-mer for typical variation graphs (see Section 5). While this is much less than with the basic $k$-mer indexes, it is still too much for large variation graphs.

**3.2 Path Graphs** When a path of length $k' < k$ has a unique label, its extensions become *redundant* nodes in the order-$k$ de Bruijn graph, if the path branches after the first $k'$ characters. By using shorter keys when possible, we can build a smaller graph that is equivalent to the de Bruijn graph as a path index.

Definition 3.2. (Path graph) *Let $k > 0$, let $\mathcal{S}$ be the $k$-collection of the labels of the maximal paths in graph $G = (V,E)$, and let $\mathcal{K}$ be a prefix-free set of substrings of length $k$ or less from $\mathcal{S}$. Assume that each right-maximal substring $S \in \mathcal{S}$ prefix-matches a string $K \in \mathcal{K}$ and that $\$^k \in \mathcal{S}$. The order-$k$* path graph *of graph $G$ with* key set *$\mathcal{K}$ is a graph $G' = (V',E')$, where*

- *each node $v_K \in V'$ represents a distinct key $K \in \mathcal{K}$, with $G'.\mathsf{label}(v_K) = K[0]$;*
- *each node has a key $G'.\mathsf{key}(v_K) = K$ and a value $G'.\mathsf{value}(v_K) = V_K$, where $V_K$ is the set of nodes $\mathcal{S}.\mathsf{node}(S,i) \in V$ for $S \in \mathcal{S}$ and positions $i$ such that $S[i,i+|K|-1] = K$; and*
- *each edge $(v_K, v_{K'}) \in E'$ represents the occurrence of substring $K[0]K'$ in $\mathcal{S}$ such that strings $K$ and $K[0]K'$ prefix-match.*

*We use $\#^k$ as the source node $s$ and $\$^k$ as the sink node $t$, adding the edge $(t,s)$ in the usual way.*

Definition 3.3. (Path graph as an index) *Let $G = (V,E)$ be a graph, and let $G' = (V',E')$ be a path graph of $G$.*

- *Pattern $X \in \Sigma^*$* matches *node $v \in V'$, if there is a path $P'$ in $G'$ with $G'.\mathsf{label}(P') = X$. We use $G'.\mathsf{find}(X)$ to denote the set of nodes $V'_X \subseteq V'$ matching the pattern.*
- *If $V'_X \subseteq V'$ is the set of nodes matching pattern $X$, the set of* occurrences *for the pattern is $G'.\mathsf{locate}(V'_X) = \bigcup_{v' \in V'_X} G'.\mathsf{value}(v')$. We use $G'.\mathsf{locate}(X)$ to denote $G'.\mathsf{locate}(G'.\mathsf{find}(X))$.*

Lemma 3.1. (No false negatives) *Let $G' = (V',E')$ be a path graph of $G = (V,E)$, and let $X \in (\Sigma \setminus \{\#,\$\})^*$ be a pattern. Set $G'.\mathsf{locate}(X)$ contains the start nodes of all paths $P$ in graph $G$ with $G.\mathsf{label}(P) = X$.*

Lemma 3.2. (Context length) *Let $G' = (V',E')$ be a path graph, and let $X \in (\Sigma \setminus \{\#,\$\})^*$ be a pattern. Set $G'.find(X)$ consists of all nodes $v' \in V'$ such that $X[0,m-1]$ is a prefix of $G'.\mathsf{key}(v')$, for a context length $m$, which depends on the graph and the pattern.*

Lemma 3.3. (Short keys) *Let $G' = (V',E')$ be a path graph with $|G'.\mathsf{key}(u')| \leq |G'.\mathsf{key}(v')| + 1$ for all edges $(u',v') \in E'$. Then (a) $|G'.\mathsf{pred}(v',c)| \leq 1$ for all nodes $v' \in V'$ and characters $c \in \Sigma$; and (b) key $G'.\mathsf{key}(v')$ prefix-matches pattern $X \in (\Sigma \setminus \{\#,\$\})^*$ for all nodes $v' \in G'.\mathsf{find}(X)$.*

A path graph may produce false positives with patterns longer than $k'$ characters, where $k'$ is the length of the shortest key. In the next section, we define a class of path graphs that can be proven to be equivalent to de Bruijn graphs.

**3.3 Pruned de Bruijn Graphs** We can compress de Bruijn graphs structurally by merging keys sharing a common prefix, if the corresponding values are identical. These pruned de Bruijn graphs, which arise naturally from GCSA construction, are similar to manifold de Bruijn graphs [24]. As path indexes, they are equivalent to de Bruijn graphs with patterns of length up to $k$ characters.

Definition 3.4. (Equivalent path graphs) *Let $G'$ and $G''$ be two path graphs, and let $k > 0$ be a parameter value. We say that graphs $G'$ and $G''$ are $k$*-equivalent, *if we have $G'.\mathsf{locate}(X) = G''.\mathsf{locate}(X)$ for all patterns $X \in (\Sigma \setminus \{\#,\$\})^*$ with $1 \leq |X| \leq k$.*

Definition 3.5. (Pruned de Bruijn graph) *Let $G$ be a graph, and let $G'$ be an order-$k$ path graph of $G$. Path graph $G'$ is an order-$k$* pruned de Bruijn graph, *if it is $k$-equivalent to the order-$k$ de Bruijn graph of $G$.*

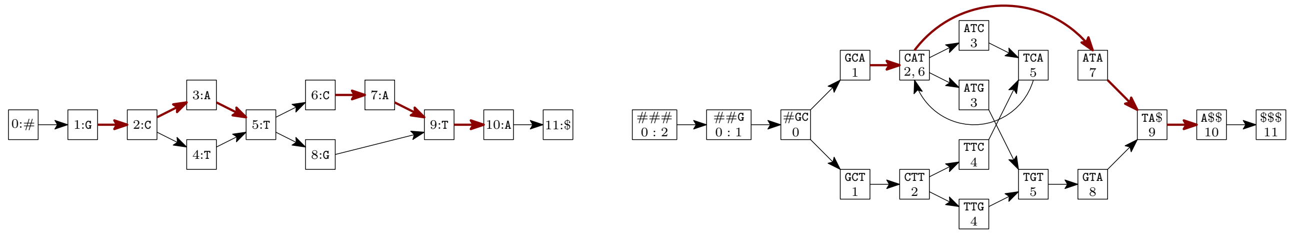


Figure 2: Left: Input graph $G = (V, E)$, with each node $v \in V$ labeled with $v : G.\mathsf{label}(v)$. Right: The order-3 de Bruijn graph $G' = (V', E')$ of graph $G$, with each node $v' \in V'$ labeled with $G'.\mathsf{key}(v')$ and $G'.\mathsf{value}(v')$. Both: Edges $(t, s)$ are not shown. The highlighted path in the de Bruijn graph is a false positive, as it consists of two disjoint paths in the input graph.

Lemma 3.4. (No short false positives) *Let $G = (V, E)$ be a graph, let $G' = (V', E')$ be an order-$k$ pruned de Bruijn graph of graph $G$, and let $X \in (\Sigma \setminus \{\#, \$\})^*$ be a pattern with $1 \leq |X| \leq k$. Then $G'.\mathsf{locate}(X)$ is a set of start nodes $v \in V$ of paths matching the pattern in graph $G$.*

Lemma 3.5. (Pruning) *Let $G = (V, E)$ be a graph, let $G' = (V', E')$ be the order-$k$ pruned de Bruijn graph of $G$ with key set $\mathcal{K}$, let $K \in \Sigma^*$ be a string of length $|K| > 0$, and let $V'_K$ be the set of nodes $v' \in V'$ having string $K$ as a proper prefix of $G'.\mathsf{key}(v')$. If $|V'_K| > 0$ and $G'.\mathsf{value}(u') = G'.\mathsf{value}(v')$ for all $u', v' \in V'_K$, the path graph with key set $(\mathcal{K} \setminus \{G'.\mathsf{key}(v') \mid v' \in V'_K\}) \cup \{K\}$ is an order-$k$ pruned de Bruijn graph of $G$.*

We can compress a de Bruijn graph structurally by merging sets of nodes sharing a common prefix of their keys, as long as the conditions of Lemma 3.5 hold. Let $G' = (V', E')$ be an order-$k$ pruned de Bruijn graph, and let $G'' = (V'', E'')$ be the same graph after further pruning. Each node $v'' \in V''$ is an *equivalence class* of nodes $V'(v'') \subseteq V'$ corresponding to a shared prefix $G''.\mathsf{key}(v'')$ of keys. For all $v' \in V'(v'')$, we have $G'.\mathsf{value}(v') = G''.\mathsf{value}(v'')$. See Figure 3 for an example of a pruned de Bruijn graph.

Definition 3.6. (Maximally pruned graph) *Let $G'$ be a a pruned de Bruijn graph of graph $G$. We say that $G'$ is* maximally pruned, *if we cannot prune it any further using Lemma 3.5.*

Lemma 3.6. (Maximal pruning) *Let $G' = (V', E')$ be a maximally pruned de Bruijn graph of $G = (V, E)$. Then $|G'.\mathsf{key}(u')| \leq |G'.\mathsf{key}(v')| + 1$ for all $(u', v') \in E'$.*

## 4 GCSA2

As in the original GCSA, we sort the nodes of the path graph in lexicographic order, encode the indegrees and outdegrees in bitvectors IN and OUT, and store the predecessor labels in BWT. See Figure 3 for an example. If lexicographic range $[sp_{i+1}, ep_{i+1}]$ matches suffix $X[i+1, |X|-1]$ of pattern $X$, we can find the range $[sp_i, ep_i]$ matching suffix $X[i, |X|-1]$ as

$$
\begin{aligned}
[sp_{in}, ep_{in}] &= [\mathsf{IN.select}(sp_{i+1}, 1) + 1, \\
&\qquad \mathsf{IN.select}(ep_{i+1} + 1, 1)]; \\
[sp_{out}, ep_{out}] &= [\mathsf{LF}(sp_{in}, X[i]), \\
&\qquad \mathsf{LF}(ep_{in} + 1, X[i]) - 1]; \\
[sp_i, ep_i] &= [\mathsf{OUT.rank}(sp_{out}, 1), \\
&\qquad \mathsf{OUT.rank}(ep_{out}, 1)].
\end{aligned}
$$

In order to support locate queries, we *sample* the values of a node, if the node has multiple incoming edges or the values cannot be derived from the predecessor. We may also sample other values to improve query performance. The sampled nodes are marked in bitvector $B_S$, the number of values in each sample is encoded in unary in bitvector $B_V$, and the sampled values are stored in integer array $V_S$. A detailed description of the data structure can be found in Appendix B.

GCSA2 construction starts from paths of length $k$ in the input graph. We build a maximally pruned order-$2k$, order-$4k$, or order-$8k$ de Bruijn graph using a similar *prefix-doubling* algorithm as in the original GCSA [43], and encode the result as a GCSA. To avoid excessive memory usage, we keep the paths and the graphs on disk, and read only a single chromosome at a time into memory. The details of the construction algorithm can be found in Appendix C.

We can improve the query performance with (maximally pruned) de Bruijn graphs by using a *simplified encoding* (Appendix D). We replace BWT and IN with bitvectors $B_c$ for all $c \in \Sigma$, where $B_c[j] = 1$ if and only if node $j$ in lexicographic order has a predecessor with

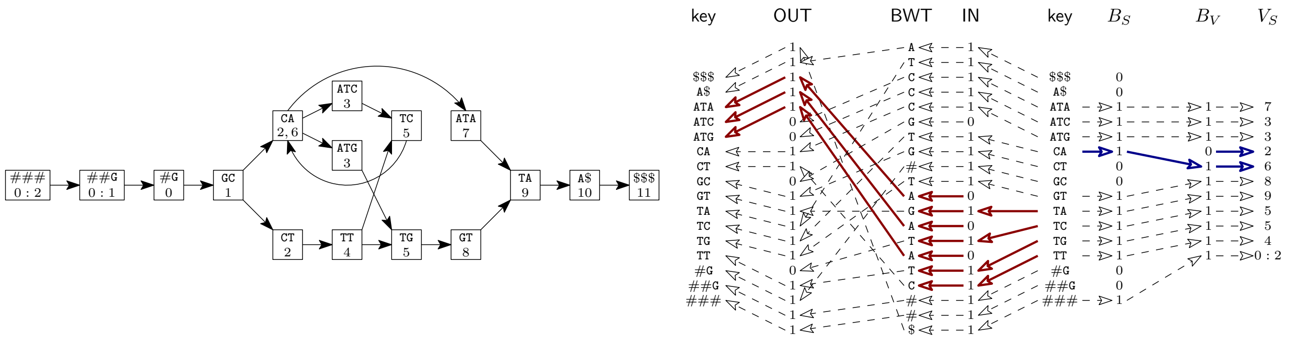


Figure 3: Left: An order-3 pruned de Bruijn graph $G''$ 3-equivalent to the de Bruijn graph in Figure 2. Right: GCSA for graph $G''$. Leftward arrows illustrate backward searching, with the red arrows showing it from `T` to `AT`. Rightward arrows mark the samples belonging to each node, with the blue ones showing them for node `CAT`.

label $c$. This simplifies backward searching to

$$\begin{aligned} sp_{out} &= C[X[i]] + B_{X[i]}.\mathsf{rank}(sp_{i+1}, 1); \\ ep_{out} &= C[X[i]] + B_{X[i]}.\mathsf{rank}(ep_{i+1}+1, 1) - 1; \\ [sp_i, ep_i] &= [\mathsf{OUT}.\mathsf{rank}(sp_{out}, 1), \mathsf{OUT}.\mathsf{rank}(ep_{out}, 1)]. \end{aligned}$$

The compacted trie of keys resembles a suffix tree. We can simulate it space-efficiently by using the LCP array [1, 11], and thus extend GCSA2 to support many suffix tree operations. For example, we can search for *maximal exact matches* by using LF-mapping and parent queries [35], and use that as a basis for a read aligner similar to BWA-MEM [21]. We can also use *document counting* techniques [41] to quickly count the number of distinct matches in a lexicographic range. Further details of these extensions can be found in Appendix E.

## 5 Implementation and Experiments

GCSA2 is the path indexing library of vg [14]. The implementation is written in C++, and the source code is available on GitHub.[3] It depends on *SDSL* [15] and *libstdc++ parallel mode*. We use the simplified encoding (Appendix D) with fast non-compressed bitvectors in most index components. Bitvectors $B_c$ for rare characters (`N`, #, and \$) are compressed as sparse bitvectors [36].

We used a system with two 16-core AMD Opteron 6378 processors and 256 gigabytes of memory for the experiments, and stored all files on a distributed Lustre file system. The system was running Ubuntu 12.04 on Linux kernel 3.2.0. We used vg version 1.3.0 for processing the graphs and GCSA2 version 0.8 using SDSL version 2.1.1 for the benchmarks. All code was compiled with gcc/g++ version 4.9.2.

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

### 5.1 Construction

*Variation graphs*, as defined in vg, use strings as node labels. A node can be traversed in both forward and reverse complement orientations, and edges may cross between the orientations. For indexing, the graph is implicitly converted into an input graph with single-character labels. We always sample the input graph nodes corresponding to the initial offsets of variation graph node labels.

We built vg graphs from the human reference genome (GRCh37) and 1000 Genomes Project variation [45]. To avoid excessive growth, we removed paths where 16-mers crossed more than 4 nontrivial edges with `vg mod -p -l -e 4`, and subgraphs shorter than 100 bases with `vg mod -S -l 100`. We extracted all paths of length 16 from the forward strand of the graph. There were a total of 4.80 billion paths with 1.53 billion distinct labels. We then built GCSA with 1–3 doubling steps, producing order-32, order-64, and order-128 indexes.

Tables 1 and 2 show construction requirements and index sizes, respectively. We can build a whole-genome index overnight using less than 96 gigabytes of memory, including disk cache. The index contains $1.031^k \cdot 2.348$ billion $k$-mers, but the path graph only uses 4.4–5.7 billion nodes to represent them. For $k = 128$, GCSA2 requires 0.63 bits per $k$-mer, out of which 0.28 bits is used for the path graph. Extensions based on suffix trees increase the size to 1.08 bits per $k$-mer.

Index construction uses more memory with $k = 32$ than with larger values of $k$. The order-32 path graph has more nodes, where we cannot derive the values from the predecessor node. As we sample more values, we need more memory in the final phase of construction. With larger values of $k$, the path graph resembles the input graph better, and we sample less values. For the same reason, index size decreases with larger values of $k$, even though the graph requires more space.

| $k$ | **Time** | **Memory** | **Disk** | **Read** | **Write** |
|---|---|---|---|---|---|
| 32 | 7.44 h | 59.8 GB | 387 GB | 1.37 TB | 0.88 TB |
| 64 | 10.4 h | 51.9 GB | 415 GB | 2.03 TB | 1.51 TB |
| 128 | 14.1 h | 52.3 GB | 478 GB | 2.78 TB | 2.25 TB |

Table 1: GCSA2 construction. Order of the path graph; construction time in hours; peak memory and disk usage in gigabytes; and disk I/O volume for reading and writing in terabytes.

**5.2 Queries** We compared the query performance of the order-128 GCSA2 to several FM-indexes for the reference sequence. SSA is the SDSL implementation (`csa_wt<>`) of the *succinct suffix array* [30], using a Huffman-shaped wavelet tree on top of the BWT. As the default FM-index in SDSL, it prioritizes query performance over compression. We used SSA with SA sample period 17 for good locate performance. BWA is the FM-index in the *Burrows-Wheeler Aligner* [22] (version 0.7.15 with the default SA sample period 32). Optimized for DNA sequences, BWA indexes both the reference and its reverse complement.

As building the original GCSA requires around $65n$ bytes of memory for a path graph with $n$ nodes, we could not compare GCSA and GCSA2 directly on a system with 256 gigabytes of memory. Instead, we used *RLCSA* [31] (May 2016 version) as a proxy. The RLCSA is an FM-index for repetitive sequence collections using the same basic components as the original GCSA. Under a mixed query load, RLCSA with SA sample period 32 is 1.5x to 3x faster than GCSA, depending on algorithmic overhead and the mix of find and locate queries [43].

We extracted $k$-mers for $k \in \{16, 32, 64, 128\}$ from the (non-pruned) vg graphs by using `vg sim`, filtered out $k$-mers consisting entirely of `N`s, and queried for the remaining $k$-mers using a single thread. The results can be seen in Table 3.

Backward searching in an FM-index stops early if there are no matches. In order to compare the find performance of the indexes reliably, we must hence concentrate on the 16-mers, where the fraction of matching patterns is similar for all indexes. GCSA2 and the fast FM-indexes (SSA and BWA) all have similar performance, while RLCSA is several times slower. As a result, we can estimate that find queries in GCSA2 are an order of magnitude faster than in GCSA.

When comparing the locate performance of different FM-indexes, the distribution of the query positions should be close to uniform. Otherwise the biases from e.g. different suffix array sampling strategies or the variation in the number of distinct occurrences per node in GCSA2 can make the results unreliable. As the $k$-mers have been sampled uniformly from the variation graph, we get the best results with the 16-mers, where all indexes can match most of the patterns.

GCSA2 uses denser SA sampling than the other indexes, with effective sample period 10.6. On the average, GCSA2 calls locate for 2.84 nodes per distinct value, making the amount of work comparable to sample period 30.2. SSA with sample period 17 is 2.4x faster than GCSA2, mostly because it has to do less work. BWA with sample period 32 is closer to GCSA2 in locate performance. RLCSA is slower than the other indexes, but the difference is smaller than with find queries due to the optimizations for retrieving suffix array ranges. Assuming that locate queries are 3x slower in GCSA than in RLCSA, as GCSA does not use the optimizations, we can estimate that GCSA2 is 4x faster than GCSA.

The remaining queries, parent and count, take a fraction of a microsecond. As a parent query takes comparable time to a single step of backward searching, it will not be a bottleneck in finding maximal exact matches. Counting the number of distinct occurrences with a count query is faster than retrieving even a single occurrence.

## 6 Discussion

GCSA2 is a path index for variation graphs. It uses a de Bruijn graph as a $k$-mer index of the variation graph, prunes it by merging redundant subgraphs, and encodes the result with a generalization of the FM-index. The index supports queries of length up to $k$ exactly, and longer queries with false positives. GCSA2 also includes extensions based on suffix trees; other extensions have been considered but not implemented (see Appendix F). The index is used in the variation graph toolkit vg for e.g. read alignment based on maximal exact matches.

We can build a whole-genome index overnight on a system with 96 gigabytes of memory and a few hundred gigabytes of fast disk space. The resulting index takes less than 15 gigabytes, or 1.08 bits per $k$-mer for the order-128 index with extensions. Query performance is comparable to that of fast FM-indexes for sequences.

The primary design goals for GCSA2 were query performance and index size. The index works with arbitrary graphs, supports queries that are long enough to map short reads in one piece without false positives, and provides several options for dealing with complex regions. Other path indexes work with a more restricted class of graphs [43, 17, 19, 26], are at least an order of magnitude slower [17, 26], require much more space [3, 38, 5, 28], or are theoretical proposals that have never been implemented [44].

We may want to determine whether a pattern matches known *haplotypes* or only their recombinations.

| $k$ | $k$-mers | Nodes | Graph | Index | With extensions |
|---|---|---|---|---|---|
| 32 | 6.20G | 4.37G | 2.89 GB / 4.00 bits | 9.50 GB / 13.2 bits | 13.2 GB / 18.2 bits |
| 64 | 16.7G | 5.24G | 3.46 GB / 1.78 bits | 8.64 GB / 4.46 bits | 13.6 GB / 6.99 bits |
| 128 | 116G | 5.73G | 3.78 GB / 0.28 bits | 8.58 GB / 0.63 bits | 14.6 GB / 1.08 bits |

Table 2: GCSA2 index sizes. Order of the path graph; number of $k$-mers and nodes in the path graph in billions; index size in gigabytes and in bits per $k$-mer for the graph ($B_c$ and $\mathsf{OUT}$), the index (the graph, $B_S$, $B_V$, and $V_S$), and the index with the extensions from Appendix E.

| $k$ | Patterns | Index | Found | Nodes | Occs | find() | parent() | count() | locate() |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 351584 | GCSA2 | 347453 | 2477M | 872M | 4.75 µs | 0.42 µs | 0.87 µs | 5.85 µs |
| | | SSA | 301538 | – | 782M | 6.00 µs | – | – | 2.43 µs |
| | | BWA | 320764 | – | 1564M | 3.64 µs | – | – | 4.65 µs |
| | | RLCSA | 301538 | – | 782M | 23.7 µs | – | – | 8.12 µs |
| 32 | 351555 | GCSA2 | 333258 | 112M | 34.3M | 10.8 µs | 0.28 µs | 0.38 µs | 5.44 µs |
| | | SSA | 153957 | – | 26.6M | 10.9 µs | – | – | 2.16 µs |
| | | BWA | 156080 | – | 52.9M | 6.57 µs | – | – | 3.19 µs |
| | | RLCSA | 153957 | – | 26.6M | 47.6 µs | – | – | 5.87 µs |
| 64 | 351567 | GCSA2 | 326101 | 2.63M | 1.35M | 22.5 µs | 0.26 µs | 0.29 µs | 2.92 µs |
| | | SSA | 88184 | – | 0.84M | 17.1 µs | – | – | 1.89 µs |
| | | BWA | 88786 | – | 1.60M | 10.3 µs | – | – | 2.34 µs |
| | | RLCSA | 88184 | – | 0.84M | 74.3 µs | – | – | 5.97 µs |
| 128 | 351596 | GCSA2 | 316500 | 0.32M | 0.37M | 45.3 µs | 0.26 µs | 0.26 µs | 3.13 µs |
| | | SSA | 35678 | – | 0.08M | 23.5 µs | – | – | 3.47 µs |
| | | BWA | 35741 | – | 0.12M | 14.0 µs | – | – | 3.46 µs |
| | | RLCSA | 35678 | – | 0.08M | 91.7 µs | – | – | 12.9 µs |

Table 3: Query benchmarks using an order-128 GCSA2 and various FM-indexes. Pattern length; number of patterns; index type; number matching patterns, matching nodes, and distinct occurrences; average time for $[sp, ep] = \mathsf{find}(X)$, $\mathsf{parent}(sp, ep)$, and $\mathsf{count}(sp, ep)$ queries in microseconds; and average time per value for $\mathsf{locate}(sp, ep)$ queries in microseconds.

As GCSA2 does not support this directly, vg must determine it afterwards using a separate structure [34]. The *FM-index of alignment* [32, 33] embeds the haplotypes directly in a GCSA-like index and reports the haplotypes matching the find query. While the solution depends on specific properties of the graph, it could be possible to extend it to work with any GCSA.

**Acknowledgements.** The author thanks Erik Garrison, Richard Durbin, and Adam Novak for the fruitful discussions while developing the GCSA2 index.

## References

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] C. Boucher et al. Variable-order de Bruijn graphs. In *Proc. 2015 IEEE Data Compression Conference (DCC 2015)*, pages 383–392. IEEE, 2015.

[3] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de Bruijn graphs. In *Proc. 12th Workshop on Algorithms in Bioinformatics (WABI 2012)*, volume 7534 of *LNCS*, pages 225–235. Springer, 2012.

[4] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[5] B. Cazaux, T. Lecroq, and E. Rivals. From indexing data structures to de bruijn graphs. In *Proc. 25th Annual Symposium on Combinatorial Pattern Matching (CPM 2014)*, volume 8486 of *LNCS*, pages 89–99. Springer, 2014.

[6] D. M. Church et al. Extending reference assembly models. *Genome Biology*, 16:13, 2015.

[7] A. Danek, S. Deorowicz, and S. Grabowski. Indexing large genome collections on a PC. *PLOS ONE*, 9(10):e109384, 2014.

[8] A. Dilthey et al. Improved genome inference in the MHC using a population reference graph. *Nature Genetics*, 47(6):682–688, 2015.

[9] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukr-

ishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1):4, 2009.

[10] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[11] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

[12] T. Gagie et al. Document counting in compressed space. In *Proc. 2015 IEEE Data Compression Conference (DCC 2015)*, pages 103–112. IEEE, 2015.

[13] T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427:25–41, 2012.

[14] E. Garrison et al. vg: the variation graph toolkit, 2014–2016. `https://github.com/vgteam/vg`.

[15] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA 2014)*, volume 8504 of *LNCS*, pages 326–337. Springer, 2014.

[16] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. fourteenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2003)*, pages 841–850. SIAM, 2003.

[17] L. Huang, V. Popic, and S. Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):i361–i370, 2013.

[18] S. Huang et al. Indexing similar DNA sequences. In *Proc. The Sixth International Conference on Algorithmic Aspects in Information and Management (AAIM 2010)*, volume 6124 of *LNCS*, pages 180–190. Springer, 2010.

[19] D. Kim et al. HISAT2, 2015–2016. `https://ccb.jhu.edu/software/hisat2/index.shtml`.

[20] D. Kim, B. Langmead, and S. L. Salzberg. HISAT: a fast spliced aligner with low memory requirements. *Nature Methods*, 12(4):357–360, 2015.

[21] H. Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. arXiv:1303.3997, 2013.

[22] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[23] A. Limasset, B. Cazaux, E. Rivals, and P. Peterlongo. Read mapping on de bruijn graphs. *BMC Bioinformatics*, 17:237, 2016.

[24] Y. Lin and P. A. Pevzner. Manifold de Bruijn graphs. In *Proc. 14th Workshop on Algorithms in Bioinformatics (WABI 2014)*, volume 8701 of *LNCS*, pages 296–310. Springer, 2014.

[25] B. Liu, H. Guo, M. Brudno, and Y. Wang. deBGA: read alignment with de Bruijn graph-based seed and extension. Accepted to Bioinformatics, 2016.

[26] S. Maciuca, C. del Ojo Elias, G. McVean, and Z. Iqbal. A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference. bioRxiv, 2016.

[27] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[28] S. Marcus, H. Lee, and M. C. Schatz. SplitMEM: a graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics*, 30(24):3476–3483, 2014.

[29] T. Marschall et al. Computational pan-genomics: Status, promises and challenges. BioRxiv, 2016.

[30] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[31] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[32] J. C. Na et al. FM-index of alignment: A compressed index for similar strings. *Theoretical Computer Science*, 638:159–170, 2016.

[33] J. C. Na et al. FM-index of alignment with gaps. arXiv:1606.03897, 2016.

[34] A. M. Novak, E. Garrison, and B. Paten. A graph extension of the positional Burrows-Wheeler transform and its applications. bioRxiv, 2016.

[35] E. Ohlebusch, S. Gog, and A. Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *Proc. 17th Symposium on String Processing and Information Retrieval (SPIRE 2010)*, volume 6393 of *LNCS*, pages 347–358. Springer, 2010.

[36] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. Ninth Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*, pages 60–70. SIAM, 2007.

[37] B. Paten, A. Novak, and D. Haussler. Mapping to a reference genome structure. arXiv:1404.5010, 2014.

[38] J. Pell et al. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proc. National Academy of Sciences of the United States of America*, 109(33):13272–13277, 2012.

[39] E. A. Rødland. Compact representation of k-mer de Bruijn graphs for genome read assembly. *BMC Bioinformatics*, 14:313, 2013.

[40] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[41] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.

[42] K. Schneeberger et al. Simultaneous alignment of short reads against multiple genomes. *Genome Biology*, 10(9):R98, 2009.

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

[44] C. Thachuk. Indexing hypertext. *Journal of Discrete Algorithms*, 18:113–122, 2013.

[45] The 1000 Genomes Project Consortium. A global

reference for human genetic variation. *Nature*, 526:68–64, 2015.

[46] S. Wandelt, J. Starlinger, M. Bux, and U. Leser. RCSI: Scalable similarity search in thousand(s) of genomes. *Proc. VLDB Endowment*, 6(13):1534–1545, 2013.

[47] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory (FOCS 1973)*, pages 1–11. IEEE, 1973.

## A Proofs of Lemmas

*Proof.* [Lemma 3.1: No false negatives] Let $\mathcal{S}$ be the $k$-collection used for building the path graph, and let $P$ be a path starting from $v_0 \in V$ with $G.\mathsf{label}(P) = X$. The collection contains a sequence $S \in \mathcal{S}$ such that $S[i, i + |X| - 1] = X$ and $\mathcal{S}.\mathsf{node}(S, i) = v_0$.

For all positions $j$ with $i \le j \le i + |X| - 1$, there is a node $v'_j \in V'$ with $G'.\mathsf{label}(v'_j) = S[j]$ and $G'.\mathsf{key}(v'_j) = S[j, j+|G'.\mathsf{key}(v'_j)|-1]$. By definition, path graph $G'$ has an edge $(v'_j, v'_{j+1}) \in E'$ for all such positions $j$. Hence $P' = v'_i \cdots v'_{i+|X|-1}$ is a path in $G'$ with $G'.\mathsf{label}(P') = X$. As path $P'$ starts from node $v'_i \in V'$, node $v'_i$ is included in the set $G'.\mathsf{find}(X)$. Furthermore, $v_0 = \mathcal{S}.\mathsf{node}(S, i) \in G'.\mathsf{value}(v'_i) \subseteq G'.\mathsf{locate}(X)$.

*Proof.* [Lemma 3.2: Context length] If $|X| \le 1$, the statement is true by definition for $m = |X|$. Now let $M_{i+1} = G'.\mathsf{find}(X[i + 1, |X| - 1])$ be the set of all nodes $v' \in V'$ such that substring $X[i + 1, i + m_{i+1}]$ is a prefix of key $G'.\mathsf{key}(v')$, and assume that the set is nonempty.

Consider the set

$$M_i = \bigcup_{v' \in M_{i+1}} G'.\mathsf{pred}(v', X[i]) = G'.\mathsf{find}(X[i, |X| - 1]).$$

There is an edge $(u', v') \in E'$ if and only if key $G'.\mathsf{key}(u')$ prefix-matches string $G'.\mathsf{label}(u') \cdot G'.\mathsf{key}(v')$. Hence key $G'.\mathsf{key}(u')$ prefix-matches string $X[i, i + m_{i+1}]$ for all nodes $u' \in M_i$.

Now let $u' \in V'$ be a node with key $G'.\mathsf{key}(u')$ prefix-matching string $X[i, i + m_{i+1}]$. If the key is a prefix of string $X[i, i + m_{i+1}]$, there is an edge $(u', v') \in E'$ to all nodes $v' \in M_{i+1}$, and hence $u' \in M_i$. Otherwise let $S[j, j + |G'.\mathsf{key}(u')| - 1] = G'.\mathsf{key}(u')$ be a substring of the $k$-collection used for building the path graph. As $S[j + 1, j + m_{i+1}] = X[i + 1, i + m_{i+1}]$, the substring starting at $S[j + 1]$ is represented by a node $v' \in M_{i+1}$, and hence $u' \in M_i$.

Set $M_i$ is the set of all nodes $u' \in V'$ such that substring $X[i, i + m_{i+1}]$ prefix-matches key $G'.\mathsf{key}(u')$. If $|M_i| > 1$, string $X[i, i+m_{i+1}]$ is a proper prefix of key $G'.\mathsf{key}(u')$ for all nodes $u' \in M_i$ due to the prefix-free property, and we can set $m_i = m_{i+1} + 1$. Otherwise we set $m_i = \min(m_{i+1} + 1, |G'.\mathsf{key}(u')|)$ for the only node $u' \in M_i$.

*Proof.* [Lemma 3.3: Short keys] (a) If node $v'$ has multiple predecessors with label $c$, the keys of the predecessors must be longer than string $c \cdot G'.\mathsf{key}(v')$, as the key set is prefix-free.

(b) By the construction in the proof of Lemma 3.2, the context length for pattern $X$ in graph $G'$ is $\min(|X|, |G'.\mathsf{key}(v')|)$.

*Proof.* [Lemma 3.4: No short false positives] We may assume without loss of generality that graph $G'$ is a de Bruijn graph. Let $v' \in G'.\mathsf{find}(X)$ be a node. By Lemma 3.3, pattern $X$ prefix-matches key $G'.\mathsf{key}(v')$. For every node $v \in G'.\mathsf{value}(v')$, there is a substring $S[i, i + k - 1] = G'.\mathsf{key}(v'_0)$ in the $k$-collection $\mathcal{S}$ used for building graph $G'$, with $\mathcal{S}.\mathsf{node}(S, i) = v$. Prefix $S[i, i + |X| - 1] = X$ of the substring corresponds to a path with label $X$ starting from node $v$ in graph $G$.

Definition A.1. (Equivalent paths) *Let $G'$ and $G''$ be path graphs of the same graph, and let $P' = v'_0 \cdots v'_{n-1}$ and $P'' = v''_0 \cdots v''_{n-1}$ be paths in graphs $G'$ and $G''$, respectively. We say that paths $P'$ and $P''$ are* equivalent, *if for $0 \le i < n$, keys $G'.\mathsf{key}(v'_i)$ and $G''.\mathsf{key}(v''_i)$ have a common prefix $K_i$ such that $G'.\mathsf{value}(v') = G''.\mathsf{value}(v'')$ for all nodes $v' \in V'$ and $v'' \in V''$ having $K_i$ as a prefix of their keys.*

*Proof.* [Lemma 3.5: Pruning] Let $G'' = (V'', E'')$ be the path graph corresponding to the new key set, let $v''_K \in V''$ be the node with key $K$, and let $\mathcal{S}$ be the $k$-collection used to define the path graphs.

Consider the edge $(u', v') \in E'$ defined by substring $S'$ of $\mathcal{S}$. The same substring also defines an edge $(u'', v'') \in E''$, where either (a) $G''.\mathsf{key}(u'') = G'.\mathsf{key}(u')$ or (b) $u'' = v''_K$ and $u' \in V'_K$, and the same holds for nodes $v''$ and $v'$. We can hence transform any path in graph $G'$ into an equivalent path in graph $G''$ by replacing nodes $v' \in V'_K$ with node $v''_K$.

Let $P'' = v''_0 \cdots v''_{|P''|-1}$ be a in graph $G''$. We transform it into an equivalent path $P' = v'_0 \cdots v'_{|P''|-1}$ in graph $G'$. There are two cases for $|P''| = 1$. If $v''_0 = v''_K$, we can replace it with any $v'_0 \in V'_K$. Otherwise we use the node $v'_0 \in V'$ with $G'.\mathsf{key}(v'_0) = G''.\mathsf{key}(v''_0)$.

In the general case $|P''| > 1$, assume that we have transformed the suffix $v''_1 \cdots v''_{|P''|-1}$ of path $P''$ into an equivalent path $v'_1 \cdots v'_{|P''|-1}$ in graph $G'$. Because $G'.\mathsf{value}(v'_1) = G''.\mathsf{value}(v''_1)$, node $v'_1$ must have a predecessor with label $c = G''.\mathsf{label}(v''_0)$. We can choose any such predecessor as node $v'_0$.

Consider the predecessors $v' \in G'.\mathsf{pred}(v'_1, c)$ and $v'' \in G''.\mathsf{pred}(v''_1, c)$. Their keys prefix-match string $Y = c \cdot K_1$. There are three cases:

1. If $Y$ is a prefix of $G'.\mathsf{key}(v')$, the key $G'.\mathsf{key}(w')$ of every successor $w'$ of node $v'$ prefix-matches $K_1$ and

hence has $K_1$ as a prefix. Therefore $G'.\mathsf{value}(v')$ is the union of sets $G.\mathsf{pred}(v,c)$ over $v \in G'.\mathsf{value}(v'_1)$. If $Y$ is also a prefix of $G''.\mathsf{key}(v'')$, nodes $v'$ and $v''$ have identical value sets by the same reasoning.

2. If $Y$ is a prefix of $G'.\mathsf{key}(v')$ and $G''.\mathsf{key}(v'')$ is a proper prefix of $Y$, there is only one possible predecessor $v'' = v''_0$. Hence $v' \in V'_K$ and $v''_0 = v''_K$.

3. If key $G'.\mathsf{key}(v')$ is a proper prefix of string $Y$, there is only one possible predecessor $v' = v'_0$. Because key $G''.\mathsf{key}(v''_0)$ prefix-matches string $Y$, it must also be a proper prefix of the string. Hence either $G'.\mathsf{key}(v'_0) = G''.\mathsf{key}(v''_0)$ or $v'_0 \in V'_K$ and $v''_0 = v''_K$.

In every case, $G'.\mathsf{value}(v'_0) = G''.\mathsf{value}(v''_0)$, and we can use shorter of strings $Y$ and $G''.\mathsf{key}(v''_0)$ as $K_0$.

We can transform any path $P'$ in graph $G'$ into an equivalent path $P''$ in graph $G''$, and the other way around. Because the labels of equivalent paths and the value sets of their start nodes are identical, we have $G'.\mathsf{locate}(X) = G''.\mathsf{locate}(X)$ for all patterns $X \in (\Sigma \setminus \{\#, \$\})^*$ with $|X| > 0$.

*Proof.* [Lemma 3.6: Maximal pruning] Let $G_d = (V_d, E_d)$ the de Bruijn graph of graph $G$ with the same order $k$ as graph $G'$, and let $\mathcal{S}$ be the $k$-collection used for building the path graphs. If $v' \in V'$ is a node, then $G_d.\mathsf{value}(v) = G'.\mathsf{value}(v')$ for all nodes $v \in V_d(v')$.

Assume that $|G'.\mathsf{key}(u')| > |G'.\mathsf{key}(v')| + 1$ for an edge $(u', v') \in E'$. String $G'.\mathsf{label}(u') \cdot G'.\mathsf{key}(v')$ must then be a prefix of key $G'.\mathsf{key}(u')$. There cannot be edges $(u', w')$ to other nodes $w' \neq v'$, as keys $G'.\mathsf{key}(v')$ and $G'.\mathsf{key}(w')$ would prefix-match.

Let $S[i, i + |G'.\mathsf{key}(u')| - 1] = G'.\mathsf{key}(u')$ be a substring of $\mathcal{S}$. Because $G'.\mathsf{key}(v')$ is a substring of $G'.\mathsf{key}(u')$ and $G'$ is a pruned de Bruijn graph, the set of nodes $\mathcal{S}.\mathsf{node}(S, i+1)$ over all occurrences of substring $G'.\mathsf{key}(u')$ in $\mathcal{S}$ is $G'.\mathsf{value}(v')$. As node $u'$ has no other successors, set $G'.\mathsf{value}(u')$ is the union of sets $G.\mathsf{pred}(v, G'.\mathsf{label}(u'))$ over all nodes $v \in G'.\mathsf{value}(v')$.

The above is true for all $x' \in G'.\mathsf{pred}(v', G'.\mathsf{label}(u'))$. Hence we can prune graph $G'$ further using string $G'.\mathsf{label}(u') \cdot G'.\mathsf{key}(v')$ as the new key in Lemma 3.5.

## B GCSA for Path Graphs

Let $G' = (V', E')$ be a path graph. We sort the nodes $V'$ by their keys in lexicographic order and generate the sequences $\mathsf{BWT}$, $\mathsf{IN}$, and $\mathsf{OUT}$ from the nodes in that order. For each node $v' \in V'$, we append $\mathsf{BWT}$ with the predecessor labels $G.\mathsf{label}(u')$ for all edges $(u', v') \in E'$; $\mathsf{IN}$ with the indegree encoded as $0^{G'.\mathsf{in}(v')-1}1$; and $\mathsf{OUT}$ with the outdegree as $0^{G'.\mathsf{out}(v')-1}1$.

If node $v' \in V'$ has lexicographic rank $i$, the range of incoming edges $(u', v') \in E'$ to that node is $[sp_{in}, ep_{in}] = [\mathsf{IN}.\mathsf{select}(i, 1) + 1, \mathsf{IN}.\mathsf{select}(i+1, 1)]$. The labels of the predecessor nodes are encoded in $\mathsf{BWT}[sp_{in}, ep_{in}]$. Sorting the incoming edges by pairs $(\mathsf{BWT}[j], i)$, where $\mathsf{BWT}[j]$ corresponds to edge $(u', v') \in E'$, is equivalent to sorting them by strings $G'.\mathsf{label}(u') \cdot G'.\mathsf{key}(v')$. As multiple edges may have the same sort key, our sorting algorithm must be stable. We get the desired sorting order by using LF-mapping: $j \mapsto \mathsf{LF}(j)$.

The range of outgoing edges $(u', v') \in E'$ from node $u' \in V'$ with lexicographic rank $i'$ is $[sp_{out}, ep_{out}] = [\mathsf{OUT}.\mathsf{select}(i', 1) + 1, \mathsf{OUT}.\mathsf{select}(i'+1, 1)]$. The edges are already sorted by keys $G'.\mathsf{key}(u')$. Because graph $G'$ is a path graph, we know that key $G'.\mathsf{key}(u')$ prefix-matches string $G'.\mathsf{label}(u') \cdot G'.\mathsf{key}(v')$. The sorting orders are therefore compatible. For every $j \in [sp_{in}, ep_{in}]$ for a node $v' \in V'$, having $\mathsf{LF}(j) \in [sp_{out}, ep_{out}]$ for a node $u' \in V'$ implies an edge $(u', v') \in E'$.

We use *backward searching* for query $G'.\mathsf{find}(X)$. Let $X \in (\Sigma \setminus \{\#, \$\})^*$ be a pattern. If $|X| = 0$, query $G'.\mathsf{find}(X)$ returns the lexicographic range $[0, |V'| - 1]$ containing all nodes. Now assume that $|X| \geq 1$ and that $G'.\mathsf{find}(X[i+1, |X|-1]) = [sp_{i+1}, ep_{i+1}]$. We want to find the lexicographic range $G'.\mathsf{find}(X[i, |X|-1])$, which is the union of sets $G'.\mathsf{pred}(v', X[i])$ over nodes $v' \in G'.\mathsf{find}(X[i+1, |X|-1])$. We map the node range $[sp_{i+1}, ep_{i+1}]$ to the range $[sp_{in}, ep_{in}]$ of incoming edges; the incoming edges to the corresponding range of outgoing edges $[sp_{out}, ep_{out}]$; and the outgoing edges to the range $[sp_i, ep_i] = G'.\mathsf{find}(X[i, |X|-1])$:

$$\begin{aligned}
[sp_{in}, ep_{in}] &= [\mathsf{IN}.\mathsf{select}(sp_{i+1}, 1) + 1, \\
&\qquad \mathsf{IN}.\mathsf{select}(ep_{i+1} + 1, 1)]; \\
[sp_{out}, ep_{out}] &= [\mathsf{LF}(sp_{in}, X[i]), \\
&\qquad \mathsf{LF}(ep_{in} + 1, X[i]) - 1]; \\
[sp_i, ep_i] &= [\mathsf{OUT}.\mathsf{rank}(sp_{out}, 1), \\
&\qquad \mathsf{OUT}.\mathsf{rank}(ep_{out}, 1)].
\end{aligned}$$

We can think this as a generalization of LF-mapping: $[sp_i, ep_i] = G'.\mathsf{LF}([sp_{i+1}, ep_{i+1}], X[i])$.

Query $G'.\mathsf{locate}(X)$ retrieves the values $G'.\mathsf{value}(v')$ for nodes $v' \in G'.\mathsf{find}(X)$ and filters out duplicates. Instead of storing the values explicitly for all nodes, GCSA uses a *sampling* scheme to save space. We assume that the nodes of the input graph $G = (V, E)$ are conveniently chosen integers. If $(u, v) \in E$ is the only outgoing edge from node $u$ and the only incoming edge to node $v$, it should be that $v = u + 1$.

We sample the values $G'.\mathsf{value}(v')$ for a node $v' \in V'$, (a) if there are multiple incoming edges to node $v'$; (b) if $v'$ is the source node $s$; or (c) if $G'.\mathsf{value}(v') \neq \{u + 1 \mid u \in G'.\mathsf{value}(u')\}$ for the only incoming edge $(u', v') \in E'$. We may also sample the values for some nodes on long unary paths for performance reasons. If the set $G'.\mathsf{value}(v')$ has not been sampled, we can derive it from sampled values by following the incoming edges.

If node $v' \in V'$ with lexicographic rank $i$ has only one predecessor, the lexicographic rank of the predecessor is $G'.\mathsf{LF}(i) = \mathsf{OUT}.\mathsf{rank}(\mathsf{LF}(\mathsf{IN}.\mathsf{select}(i,1)+1),1)$. If the lexicographic rank $G'.\mathsf{LF}^k(i)$ corresponding to node $w' \in V'$ is the first sampled node we encounter, we know that $G'.\mathsf{value}(v') = \{w + k \mid w \in G'.\mathsf{value}(w')\}$.

Let $B_S[0, |V'|-1]$ be a bitvector. If we have sampled the values for the node $v' \in V'$ with lexicographic rank $i$, we mark that as $B_S[i] = 1$. We can then determine the rank of node $v'$ among the sampled nodes as $j = B_S.\mathsf{rank}(i,1)$. For each sampled node $v' \in V$, we store the size of the value set $|G'.\mathsf{value}(v')|$ in another bitvector $B_V$, using the same encoding as for bitvectors $\mathsf{IN}$ and $\mathsf{OUT}$. We store the samples in array $V_S$ in the same order, using $\log|V|$ bits each. The sampled values for node $v'$ with rank $j$ among the sampled nodes can be found at $V_S[B_V.\mathsf{select}(j,1)+1, B_V.\mathsf{select}(j+1,1)]$.

## C Index Construction

GCSA construction [43] is based on the *prefix-doubling* algorithm for suffix array construction [27]. The original GCSA started from paths of length 1 in the input graph, and then repeatedly *joined* paths of length $k$ into paths of length $2k$, until each path had a distinct label. The resulting path graph was essentially an order-$\infty$ pruned de Bruijn graph and supported queries of any length.

We use a variant of that algorithm with GCSA2. Let $G = (V, E)$ be the input graph. We extract all paths of length $k$ (typically with $k = 16$) from graph $G$. For each path $P = v_0 \cdots v_{|P|-1}$, we store several fields. *Key* $P.\mathsf{key}$ encodes $G.\mathsf{label}(P)$ as a sequence of lexicographic ranks of $k$-mers. If $|P|$ is not an integer multiple of $k$, the key consists of the $k$-mer ranks for the lexicographically smallest $(\lceil |P|/k \rceil \cdot k)$-mer having $G.\mathsf{label}(P)$ as a prefix, followed by the rank of the last $k$-mer in the largest such $(\lceil |P|/k \rceil \cdot k)$-mer. *Value* $P.\mathsf{value}$ is the start node $v_0$ of the path. We store the set of *predecessor labels* $\{c \in \Sigma \mid |G.\mathsf{pred}(v_0, c)| > 0\}$ as $P.\mathsf{pred}$. For each possible *extension node* $v \in \{v \in V \mid (v_{|P|-1}, v) \in E\}$, we create a separate copy of the path and store the node as $P.\mathsf{ext} = v$.

The construction uses several supporting structures. We build an order-$k$ *de Bruijn graph* $G_d = (V_d, E_d)$ of the path labels and encode it as a GCSA, using the predecessor labels $P.\mathsf{pred}$ for determining the edges. Let $v_0, \ldots, v_{|V_d|-1}$ be the nodes of the de Bruijn graph in lexicographic order by their keys. We use two additional arrays: the *LCP array* $\mathsf{LCP}[0, |V_d|-1]$, where $\mathsf{LCP}[i]$ is the length of the longest common prefix of keys $G_d.\mathsf{key}(v_{i-1})$ and $G_d.\mathsf{key}(v_i)$ (with $\mathsf{LCP}[0] = 0$), and the *last character array* $L[0, |V_d|-1]$, where $L[i] = G_d.\mathsf{key}(v_i)[k-1]$. The LCP array is stored as a wavelet tree for fast *range minimum queries* [13].

Because we store path labels explicitly, we only do a limited number of *doubling steps*, typically 2 or 3. After $d$ doubling steps, the length of the paths is $2^d k$, and we can use them to build a maximally pruned order-$(2^d k)$ de Bruijn graph. Each doubling step consists of a *pruning* step, followed by an *extension* step. The pruning step applies a limited form of Lemma 3.5 to lexicographic ranges of paths. Given two paths $P$ and $P'$, we can determine the length of the longest common prefix of the path labels by using the keys $P.\mathsf{key}$ and $P'.\mathsf{key}$ and the LCP array. If all paths sharing a prefix start from the same node, we merge them into a single path $Q$ with $Q.\mathsf{key}$ based on the shared prefix and $Q.\mathsf{ext} = -1$.

The extension step transforms the current set of paths of length (up to) $k'$ into a set of paths of length (up to) $2k'$. If $P$ is a path with $P.\mathsf{ext} = -1$, we use it as such. If $P.\mathsf{ext} = P'.\mathsf{value}$ for paths $P$ and $P'$, we create a new path $PP'$. We set $(PP').\mathsf{key}$ according to the concatenation of the path labels, take $\mathsf{value}$ and $\mathsf{pred}$ from path $P$, and take $\mathsf{ext}$ from path $P'$. If we have another path $Q$ with $Q.\mathsf{key} = P.\mathsf{key}$ such that $QP'$ is a path, and if $P'.\mathsf{ext} = -1$, all possible $2k'$-mer extensions of label $G.\mathsf{label}(PP')$ are also labels of paths starting from node $Q.\mathsf{value}$, and the other way around. Hence paths $PP'$ and $QP'$ can be represented by a single node in a pruned de Bruijn graph.

The doubling steps are followed by the *merging step*, which transforms the paths into the nodes of a maximally pruned de Bruijn graph $G'' = (V'', E'')$. We merge the paths with identical keys into the nodes of a pruned de Bruijn graph $G' = (V', E')$. If paths $P_0, \cdots, P_{m-1}$ all have the same key, we create a node $v' \in V'$ with the shared key as $G'.\mathsf{key}(v')$ and with $G'.\mathsf{value}(v') = \bigcup_{i=0}^{m-1} P_i.\mathsf{value}$. We also store the union of predecessor labels as $v'.\mathsf{pred} = \bigcup_{i=0}^{m-1} P_i.\mathsf{pred}$. We then apply Lemma 3.5 maximally, transforming graph $G'$ into graph $G''$.

Storing the paths and the graphs may require hundreds of gigabytes of memory when indexing whole-genome variation graphs. To avoid that, we keep them on *disk* when possible. The subgraph corresponding to each chromosome is stored in a separate file, with paths sorted by their labels in lexicographic order. Extension steps are done separately for each of the chromosomes. The pruning step merges the sorted files. It keeps reading paths into a buffer, until it has found a maximal range of paths that can be merged. The merged path is written into the new file for that chromosome, and the original range of paths is removed from the buffer. The merging step works in a similar way.

After creating the nodes $V''$ of the maximally pruned order-$k'$ de Bruijn graph $G'' = (V'', E'')$, we *build* *the index.* Sequences $\mathsf{BWT}$ and $\mathsf{IN}$ can be generated from the predecessor sets $v''.\mathsf{pred}$, while the outdegree sequence $\mathsf{OUT}$ requires further processing. There is an edge $(u'', v'') \in E''$ if and only if string $c \cdot G''.\mathsf{key}(v'')$ prefix-matches key $G''.\mathsf{key}(u'')$ and $c \in v''.\mathsf{pred}$. We determine the edges and produce the outdegree sequence by scanning the node file sequentially with $\sigma + 1$ pointers. The pointer corresponding to node $v''$ scans the entire file, while each of the remaining $\sigma$ pointers scans only the range of nodes $u''$ with $G''.\mathsf{label}(u'') = c$ for a character $c \in \Sigma$. We also sample the nodes for $\mathsf{locate}$ queries during the same scans.

Checking whether key $G''.\mathsf{key}(u'')$ prefix-matches string $c \cdot G''.\mathsf{key}(v'')$ can be done by using the GCSA for de Bruijn graph $G_d = (V_d, E_d)$ and the last character array $L$. If the lexicographic rank of $k$-mer $X$ is $i$, the lexicographic rank of $k$-mer $(cX)[0, k-1]$ is $G_d.\mathsf{LF}(i, c)$. If $2k$-mer $X$ is encoded with $k$-mer ranks $(i, j)$, we can encode string $cX$ as a lexicographic range of $k$-mer rank sequences, with $(G_d.\mathsf{LF}(i, c), G_d.\mathsf{LF}(j, L[i]), G_d.\mathsf{LF}(0, L[j]))$ as the lower bound and $G_d.\mathsf{LF}(|V_d| - 1, L[j])$ as the last rank in the upper bound. String $c \cdot G''.\mathsf{key}(v'')$ prefix-matches key $G''.\mathsf{key}(u'')$ if and only if the lexicographic ranges of the $k$-mer rank sequences overlap.

*Complex regions* of the variation graph must be pruned before indexing. While this happens before index construction begins, the construction algorithm has features that can make the pruning less destructive. Pruning heuristics often create regions that are completely missing from the index. The same mechanism that saves memory by having each chromosome in a separate file can be used to index *overlapping subgraphs* without indexing any paths between them. By having the pruned graph in one file and the reference path in another file, we can guarantee that no region is completely missing from the index. We can also index *selected paths* in complex regions by duplicating nodes for prefix-doubling and mapping the duplicates back to the original nodes during the merging step. If we index the paths corresponding to *known haplotypes* in complex regions, we can guarantee that the index contains all observed variation.

## D Simplified GCSA Encoding

Let $G' = (V', E')$ be a path graph with $G'.\mathsf{pred}(v', c) \leq 1$ for all nodes $v' \in V'$ and characters $c \in \Sigma$. This is true for de Bruijn graphs, and also for maximally pruned de Bruijn graphs (Lemmas 3.6 and 3.3). We can use the simplified encoding of the original GCSA [43] with such path graphs.

We replace the sequences $\mathsf{BWT}$ and $\mathsf{IN}$ with *indicator bitvectors* $B_c[0, |V'| - 1]$ for all $c \in \Sigma$. If node $v' \in V$ with lexicographic rank $i$ has a predecessor with label $c \in \Sigma$, we set $B_c[i] = 1$. The backward step becomes:

$$\begin{aligned} sp_{out} &= C[X[i]] + B_{X[i]}.\mathsf{rank}(sp_{i+1}, 1); \\ ep_{out} &= C[X[i]] + B_{X[i]}.\mathsf{rank}(ep_{i+1} + 1, 1) - 1; \\ [sp_i, ep_i] &= [\mathsf{OUT}.\mathsf{rank}(sp_{out}, 1), \mathsf{OUT}.\mathsf{rank}(ep_{out}, 1)]. \end{aligned}$$

Two expensive queries ($\mathsf{IN}.\mathsf{select}()$ and $\mathsf{BWT}.\mathsf{rank}()$) are replaced with a cheap $B_c.\mathsf{rank}()$.

Computing $G'.\mathsf{LF}(i)$ can expensive, as we have to look at $B_c[i]$ for all $c \in \Sigma$ to determine the character used in the backward step. If the alphabet is small, this is still faster than the $\mathsf{select}$ queries in the general encoding. We can further improve the time/space trade-off by compressing the bitvectors $B_c$ for rare characters (e.g. N, #, and \$) and checking $B_c[i]$ first for the frequent characters (e.g. bases) when computing $G'.\mathsf{LF}(i)$.

## E Suffix Tree of a Path Graph

Let $G' = (V', E')$ be an order-$k$ path graph, and let $v'_0, \ldots, v'_{|V'|-1}$ be its nodes in lexicographic order. The *LCP array* of graph $G'$ is an array $\mathsf{LCP}[0, |V'| - 1]$, where $\mathsf{LCP}[i]$ is the length of the longest common prefix of keys $G'.\mathsf{key}(v'_{i-1})$ and $G'.\mathsf{key}(v'_i)$, with $\mathsf{LCP}[0] = 0$. If we build a trie of keys $G'.\mathsf{key}(v')$ for $v' \in V'$ and compact the unary paths into single edges, the resulting tree can be considered the *suffix tree* of graph $G'$.

The *LCP interval tree* [1] is an alternate representation of the suffix tree. Each node $v$ of the suffix tree is represented by the lexicographic range $[sp, ep]$ matching the path label $\ell(v)$ from the root to the node. These ranges can be determined from the LCP array. If we use the FM-index with an LCP array supporting *next/previous smaller value* queries and *range minimum queries*, we can support the full functionality of the suffix tree [11].

If we build an $x$-ary tree over the LCP array, with each internal node storing the minimum LCP value in the corresponding range, we can support the required queries in $\mathsf{O}(x \log_x |V'|)$ time with $\mathsf{O}(\log_x |V'|)$ random memory accesses, while using $\frac{x}{x-1} |V'| \log k$ bits of space. Because the nodes of a path graph may match patterns that do not prefix-match their keys, we have to be careful with the suffix tree operations we use. By Lemma 3.2, the ranges returned by $\mathsf{find}$ queries always correspond to prefixes of the pattern. Hence we can safely use $\mathsf{parent}$ queries with such ranges.[4]

BWA-MEM [21] aligns reads to a reference genome by finding *maximal exact matches* between the read and the reference. It indexes both the reference and its reverse complement, allowing it to extend the pattern in both directions. GCSA cannot use similar techniques, as

[4]The $\mathsf{shorter}$ query in the variable-order de Bruijn graph [2] is essentially a $\mathsf{parent}$ query.

we cannot guarantee that the length of the lexicographic range matching pattern $X$ is the same as the length of the range matching its reverse complement $\overleftarrow{X}$ (that the key set contains key $K$ if and only if it contains key $\overleftarrow{K}$). However, we can search for maximal exact matches by using LF-mapping and the $\mathsf{parent}$ operation [35].

In an ordinary FM-index, the length of a lexicographic range tells the number of distinct pointers in the range. In GCSA, each node $v' \in V'$ may have multiple values (pointers), and a value may occur in multiple nodes. As $\mathsf{locate}$ queries can be slow, we need another way to support efficient *counting queries*. The problem is similar to determining the *document frequency* of a pattern. Given a collection of documents, the document frequency of pattern $X$ is the number of distinct documents that contain occurrences of the pattern. We can use a bitvector of length $2n - d - 1$, where $n$ is the total size of the document collection and $d$ is the number of documents, to compute the frequencies efficiently [41].

Let $v$ be a suffix tree node corresponding to lexicographic range $[sp, ep]$, and let $\mathsf{count}(v)$ be the document frequency of the label $\ell(v)$. If nodes $v_0, \ldots, v_{m-1}$ are the children of node $v$, the number of *redundant* documents in them is $R(v) = \sum_{i=0}^{m-1} \mathsf{count}(v_i) - \mathsf{count}(v)$. We create an array $R[0, n-2]$ based on the *inorder traversal* of the suffix tree. If the $i$th internal node we encounter is our first visit to node $v$, we set $R[i] = R(v)$. We set $R[j] = 0$ for any subsequent visits to the same node. Range $R[sp, ep-1]$ covers the internal nodes in the subtree with node $v$ as the root. We can determine document frequencies as $\mathsf{count}(v) = (ep + 1 - sp) - \sum R[sp, ep-1]$. If we encode array $R$ in unary, with value $x$ becoming $0^x 1$, we get a bitvector $B_R$, where we can compute sums $\sum_{i=a}^{b} R[i]$ with $\mathsf{select}$ queries as

$$(B_R.\mathsf{select}(b+1, 1) - b) - (B_R.\mathsf{select}(a, 1) + 1 - a).$$

For value counting, we use array $R$ to store the number of redundant values as above. We use another array $A[0, |V'| - 1]$ to store the number of additional values in each node $v'_i \in V'$ as $A[i] = |G'.\mathsf{value}(v'_i)| - 1$, and encode it as a bitvector $B_A$ in the same way as array $R$ above. The number of distinct values in range $[sp, ep] = G'.\mathsf{find}(X)$ for a pattern $X$ is

$$\begin{aligned} G'.\mathsf{count}(X) &= G'.\mathsf{count}(sp, ep) \\ &= \sum_{i=sp}^{ep} (A[i] + 1) - \sum_{i=sp}^{ep-1} R[i]. \end{aligned}$$

The bitvectors are often highly compressible [12], but GCSA already uses one of the compression schemes implicitly when it prunes the de Bruijn graph.

## F Using the Hypertext Index

The hypertext index [44] is based on graphs $G = (V, E)$, where the label $G.\mathsf{label}(v)$ of a node $v \in V$ is a string over alphabet $\Sigma$. The labels are indexed in FM-index $F$, while the reverse labels are indexed in FM-index $R$. When we search for a pattern $X \in \Sigma^*$ of length $|X| \geq 2$, some of the matches may cross edges. In order to find matches crossing one edge, we search for suffixes $X[i, |X| - 1]$ in the forward index $F$ and the reverses of prefixes $X[0, i-1]$ in the reverse index $R$, for all $1 \leq i \leq |X| - 1$. For each value of $i$, we combine the partial matches into complete matches with a two-dimensional range query in the edge matrix $E$, using the lexicographic ranges for the reverse of $X[0, i-1]$ and for $X[i, |X| - 1]$ as the query ranges.

We often have to prune complex regions of the input graph before indexing it. This causes false negatives: paths that exist in the input graph but not in the index. We can avoid the false negatives with a generalization of the hypertext index. Instead of pruning the input graph heuristically, we create a primary graph based on known haplotypes and build a GCSA index for both strands of the graph. We then create a matrix of additional edges corresponding to potential *recombinations* in the path graph, always crossing from the reverse complement strand to the forward strand. We search for pattern $X$ and its reverse complement $\overleftarrow{X}$ in the index, and combine the results $\mathsf{find}(X[i, |X|-1])$ and $\mathsf{find}(\overleftarrow{X}[|X|-i, |X|-1])$ with a range query.

While graphs are a natural formalism for representing genetic variation, they cannot adequately represent certain types of *rearrangements*. For example, if sequence $S$ can occur in different positions of the genome (e.g. $ASBC$ and $ABSC$), we can either have a single copy or multiple copies of $S$ in the graph. Neither option is good in a reference genome. With a single copy, we create paths in the reference that do not correspond to any valid genome. With the second option, we lose the information that both copies of $S$ are the same sequence. One solution is to use *context-free grammars*. As long as the grammar is non-nested, we can handle it with the hypertext index. We build a hypertext index for a high level graph, where each node is labeled with a nonterminal symbol, while each nonterminal expands into a subgraph indexed in GCSA.