Haplotype-aware graph indexes

Jouni Sirén[1]

University of California, Santa Cruz, USA
Wellcome Sanger Institute, Hinxton, UK
jouni.siren@iki.fi
 https://orcid.org/0000-0001-5828-4139

Erik Garrison[2]

Wellcome Sanger Institute, Hinxton, UK
eg10@sanger.ac.uk
 https://orcid.org/0000-0003-3821-631X

Adam M. Novak[3]

University of California, Santa Cruz, USA
anovak@soe.ucsc.edu
 https://orcid.org/0000-0001-5828-047X

Benedict J. Paten[4]

University of California, Santa Cruz, USA
bpaten@ucsc.edu
 https://orcid.org/0000-0001-8863-3539

Richard Durbin[5]

Department of Genetics, University of Cambridge, UK
Wellcome Sanger Institute, Hinxton, UK
rd@sanger.ac.uk
 https://orcid.org/0000-0002-9130-1006

Abstract

The variation graph toolkit (VG) represents genetic variation as a graph. Each path in the graph is a potential haplotype, though most paths are unlikely recombinations of true haplotypes. We augment the VG model with haplotype information to identify which paths are more likely to be correct. For this purpose, we develop a scalable implementation of the graph extension of the positional Burrows–Wheeler transform. We demonstrate the scalability of the new implementation by indexing the 1000 Genomes Project haplotypes. We also develop an algorithm for simplifying variation graphs for k-mer indexing without losing any k-mers in the haplotypes.

---

1 Introduction

Sequence analysis pipelines often start by mapping the sequence reads to a *reference genome* of the same species. A read aligner first uses a *text index* to find candidate positions for the read. Then it aligns the read to the candidate positions, trying to find the best mapping.

The reference genome may represent the sequenced individual poorly, e.g. when it is a single sequence that diverges substantially at some location. Mapping reads to such a reference can introduce *reference bias* to the subsequent analysis. Richer reference models can help to avoid the bias, but challenges remain in choosing the right model and working with it effectively [28].

We can replace the single reference sequence with a *collection of haplotypes*. Because individual genomes are similar, compressed text indexes can store such collections in very little space [18]. However, due to this similarity, most reads map equally well to many haplotypes. If the reference model is a simple collection, we cannot tell whether a read maps to the same position in different haplotypes or not.

If the haplotypes are *aligned*, we can use the alignment to determine whether the mappings are equivalent. Text indexes can also take advantage of the alignment by storing shared substrings only once [13]. The FM-index of alignment [19, 20] goes one step further by collapsing the multiple alignment into a *directed acyclic graph* (DAG), where each node is labeled by a sequence. It indexes the graph and stores some additional information for determining which paths correspond to valid haplotypes.

We can also build a reference graph directly from a reference sequence and a set of *variants* [24]. This approach has been used in many tools such as GCSA [26], BWBBLE [12], vBWT [16], the Seven Bridges Graph Pipeline [23], and Graphtyper [5]. Algorithms for working with sequences are often easy to generalize to DAGs. On the other hand, because an acyclic graph imposes a global alignment on the haplotypes, allowing only matches, mismatches, and indels, it cannot represent structural variation such as duplications or inversions adequately.

Assembly graphs such as *de Bruijn graphs* collapse sequences by local similarity instead of global alignment. They are better suited to handling structural variation than DAGs. However, the lack of a global coordinate system limits their usefulness as references.

Graph-based reference models share certain weaknesses. Because they collapse sequences between variants, they represent both the original haplotypes and their *recombinations*, that is paths that switch between haplotypes. This may cause false positives when a read maps better to an unobserved recombination than to the correct path. Graph regions with many variants in close proximity can give rise to very large numbers of recombinant paths, and be *too complex* to index completely. Graph tools try to deal with such regions by, for example, limiting the amount of variation in the graph, artificially simplifying complex regions, and making trade-offs between query performance, index size, and maximum query length.

CHOP [17] embeds haplotypes into a graph and indexes the corresponding paths. For a given parameter k, the graph is transformed into a collection of short strings such that adjacent strings overlap by $k-1$ characters. Each haplotype can be represented as a sequence of adjacent strings. Any read aligner can be used to map reads to the strings. However, because the aligner sees only short strings, it cannot map long reads or paired-end reads.

The *variation graph toolkit* (VG) [9] works with many kinds of graphs. While the tools mentioned earlier use graphs to represent other information (e.g. sequences or variants),

the graph itself is the primary object in the VG model. A global coordinate system can be provided by designating certain paths as *reference paths*.

VG uses GCSA2 [25] as its text index. GCSA2 represents a *k*-mer index as a de Bruijn graph and compresses it structurally by merging redundant nodes. VG handles complex graph regions by indexing a *simplified graph*, although the final alignment is done in the original graph. The drawback of this approach is that simplification can break paths corresponding to known haplotypes, while leaving paths representing recombinations intact.

In this paper, we augment the VG model with haplotype information. We develop the GBWT, a scalable implementation of the graph extension of the positional Burrows–Wheeler transform (gPBWT) [4, 21], to store the haplotypes as paths in the graph. To demonstrate the scalability of the GBWT, we build an index for the the 1000 Genomes Project haplotypes. We also describe an algorithm that adds the haplotype paths back to the simplified graph, without reintroducing too much complexity.

The main differences to the old gPBWT implementation [21] are:

- We use *local structures* for each node instead of global structures for the graph. The index is smaller and faster and takes better advantage of memory locality.
- The GBWT is implemented as an ordinary text index instead of a special-purpose index for paths. Most FM-index algorithms can be used with it. For example, we can use the GBWT as an FMD-index [14] and support *bidirectional search*.
- We have a fast and space-efficient *incremental construction* algorithm that does not need access to the entire collection of haplotypes at the same time.
- Our implementation can be used *independently* of VG.

The haplotype information stored in GBWT can also be used to improve read mapping, and potentially, variant inference. We leave this investigation to a subsequent paper.

2 Background

2.1 Strings and graphs

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* $T[n-1] = \$ = 0$ that does not occur anywhere else in the text. *Substrings* of string S are sequences of the form $S[i, j] = s_i \cdots s_j$. We call substrings of length k *k*-mers and substrings of the type $S[0, j]$ and $S[i, n-1]$ *prefixes* and *suffixes*, respectively.

Let $S[0, n-1]$ be a string. 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 $S.\mathsf{select}(i, c) = \max\{j \le n \mid S.\mathsf{rank}(j, c) < i\}$ as the position of the occurrence of rank $i > 0$. A *bitvector* is a data structure that stores a binary sequence and supports efficient $\mathsf{rank}/\mathsf{select}$ queries over it.

A *graph* $G = (V, E)$ consists of a finite set of *nodes* $V \subset \mathbb{N}$ and a set of *edges* $E \subseteq V \times V$. We assume that the edges are *directed*: $(u, v) \in E$ is an edge *from* node u *to* node v. The *indegree* of node v is the number of *incoming* edges to v, while the *outdegree* is the number of *outgoing* edges from v. Let $P = v_0 \cdots v_{|P|-1}$ be a string over the set of nodes V. We say that P is a *path* in graph $G = (V, E)$, if $(v_i, v_{i+1}) \in E$ for all $0 \le i < |P| - 1$.

The VG model [9] is based on *bidirected* graphs, where each node has two *orientations*. We simulate them with directed graphs. We partition the set of nodes V into *forward* nodes V_f and *reverse* nodes V_r, with $V_f \cap V_r = \emptyset$ and $|V_f| = |V_r|$. We match each forward node $v \in V_f$ with the corresponding reverse node $\overline{v} \in V_r$, with $\overline{\overline{v}} = v$ for all $v \in V_f$. For all nodes $u, v \in V$, we also require that $(u, v) \in E \iff (\overline{v}, \overline{u}) \in E$.

2.2 FM-index

The *suffix array* $\mathsf{SA}[0, n-1]$ of text $T[0, n-1]$ is an array of pointers to the suffixes of the text in *lexicographic order*. For all $i < j$, we have $T[\mathsf{SA}[i], n-1] < T[\mathsf{SA}[j], n-1]$. The *Burrows–Wheeler transform* (BWT) [2] is a permutation of the text with a similar combinatorial structure. We define it as string $\mathsf{BWT}[0, n-1]$, where $\mathsf{BWT}[i] = T[(\mathsf{SA}[i] - 1) \bmod n]$. Let $\mathsf{C}[c]$ be the number of occurrences of characters $c' < c$ in the text. The main operation in BWT is the *LF-mapping*, which we define as $\mathsf{LF}(i, c) = \mathsf{C}[c] + \mathsf{BWT.rank}(i, c)$. We use shorthand $\mathsf{LF}(i)$ for $\mathsf{LF}(i, \mathsf{BWT}[i])$ and note that $\mathsf{SA}[\mathsf{LF}(i)] = (\mathsf{SA}[i] - 1) \bmod n$

Let X be a string and let c be a character. If $T' < X$ for i suffixes T' of text T, we say that string X has *lexicographic rank i* among the suffixes of text T. The number of suffixes starting with any character $c' < c$ is $\mathsf{C}[c]$, and the number of suffixes $T' < X$ preceded by character c is $\mathsf{BWT.rank}(i, c)$. Hence the lexicographic rank of string cX is $\mathsf{LF}(i, c)$.

The FM-index [6] is a text index based on the BWT. Assume that we can compute $\mathsf{BWT.rank}(i, c)$ in t_r time. Further assume that we have stored $(i, \mathsf{SA}[i])$ for all $\mathsf{SA}[i]$ divisible by some integer $d > 0$. The FM-index supports the following queries:

- $\mathsf{find}(X)$: Return the *lexicographic range* $[sp, ep]$ of suffixes starting with *pattern X*. If $[sp_{i+1}, ep_{i+1}]$ is the lexicographic range for pattern $X[i+1, |X|-1]$, the range for pattern $X[i, |X|-1]$ is $[\mathsf{LF}(sp_{i+1}, X[i]), \mathsf{LF}(ep_{i+1}+1, X[i]) - 1]$. By extending the pattern backwards, we can support $\mathsf{find}(X)$ in $\mathsf{O}(|X| \cdot t_r)$ time.
- $\mathsf{locate}(sp, ep)$: Return the *occurrences* $\mathsf{SA}[sp, ep]$. For each $i \in [sp, ep]$, we iterate $\mathsf{LF}(i)$ until we find a stored pair $(\mathsf{LF}^k(i), \mathsf{SA}[\mathsf{LF}^k(i)])$. Then $\mathsf{SA}[i] = \mathsf{SA}[\mathsf{LF}^k(i)] + k$. Locating each occurrence $\mathsf{SA}[i]$ takes $\mathsf{O}(d \cdot t_r)$ time.
- $\mathsf{extract}(j, j')$: Return the substring $T[j, j']$. We start from the nearest stored $(i, \mathsf{SA}[i])$ with $\mathsf{SA}[i] > j'$ and iterate $(i, \mathsf{SA}[i]) \leftarrow (\mathsf{LF}(i), \mathsf{SA}[i] - 1)$ until $\mathsf{SA}[i] = j + 1$. As $\mathsf{BWT}[i] = T[\mathsf{SA}[i] - 1]$, we extract the substring backwards in $\mathsf{O}((d + j' - j) \cdot t_r)$ time.

We can generalize the FM-index to indexing *multiple texts* $T_0, \ldots, T_{m-1}$. Each text T_j is terminated by a distinct endmarker $\$_j$, where $\$_j < \$_{j+1}$ for all j. As the suffixes of the texts are all distinct, we can sort them unambiguously. In the final BWT, we replace each $\$_j$ with $\$$ in order to reduce alphabet size. The index works as with a single text, except that we cannot compute $\mathsf{LF}(i, \$)$. We also define the *document array* DA as an array of *text identifiers*. If $\mathsf{SA}[i]$ points to a suffix of text T_j, we define $\mathsf{DA}[i] = j$.

3 Indexing haplotypes

3.1 Positional BWT

Assume that we have m haplotype strings $S_0, \ldots, S_{m-1}$ of equal length over alphabet Σ. At each variant site i, character $S_j[i]$ tells whether haplotype j contains the reference allele ($S_j[i] = 0$) or an alternate allele ($S_j[i] > 0$). Given a pattern X and a range of sites $[i, i']$, we want to find the haplotypes S_j matching the pattern at the specified sites ($S_j[i, i'] = X$). Ordinary FM-indexes do not support such queries, as they find all occurrences of the pattern.

The *positional BWT* (PBWT) [4] is an FM-index that supports *positional queries*. We can interpret it as the FM-index of texts $T_0, \ldots, T_{m-1}$ such that $T_j[i] = (i, S_j[i])$ [7]. If we want to search for pattern X in range $[i, i']$, we search for pattern $X' = (i, X[0]) \cdots (i', X[|X|-1])$ in the FM-index. The texts are over a large alphabet, but their *first-order empirical entropy* is low. We can encode the BWT using alphabet Σ with a simple model. Assume that $\mathsf{SA}[x]$ points to a suffix starting with $(i + 1, c)$. We often know the character from a previous query, and we can determine it using the C array. Then $\mathsf{BWT}[x] = (i, c')$ for a $c' \in \Sigma$, and we can

encode it as c'. (Note that we build the rank structure for the original BWT, not the encoded BWT.) Because the collection of haplotype strings is usually *repetitive*, we can compress the PBWT further by *run-length encoding* the BWT [18].

3.2 Graph extension

Haplotypes correspond to paths in the VG model. Because chromosome-length phasings are often not available, there may be multiple paths for each haplotype. The *graph extension* of the PBWT [21] generalizes the PBWT to indexing such paths. While the original extension was specific to VG graphs, we present a simplified version over directed graphs. We call this structure the *Graph BWT* (GBWT), as it encodes the BWT using the graph as a model.

Let $P_0, \ldots, P_{m-1}$ be paths in graph $G = (V, E)$. We can interpret the paths as strings over alphabet V. Assume that $0 \notin V$, as we use it as the endmarker. We build an FM-index for the *reverse strings*. We sort reverse prefixes in lexicographic order, so the LF-mapping traverses edges in the correct direction, and place the endmarker before the string.

For each node $v \in V$, we define the *local alphabet* $\Sigma_v = \{w \in V \mid (v, w) \in E\}$. We also add $ to Σ_v if v is the last node on a path, and define $\Sigma_\$$ as the set of the initial nodes on each path. We partition the BWT into substrings BWT_v corresponding to the prefixes ending with v, and encode each substring BWT_v using the local alphabet Σ_v. If $w \in \Sigma_v$ is the kth character in the local alphabet in sorted order, we encode it as $\Sigma_v(w) = k$.

The GBWT supports the following variants of the standard FM-index queries:

- find(X) returns the lexicographic range of reverse prefixes starting with the reverse pattern (the range of prefixes ending with the pattern).
- locate(sp, ep) returns the haplotype identifiers $\mathsf{DA}[sp, ep]$. We do not return text offsets, as the node corresponding to the range $[sp, ep]$ already provides similar information.
- extract(j) returns the path P_j. We save memory by not supporting substring extraction.

These queries should be understood as examples of what we can support. Because the GBWT is an FM-index for multiple texts, most algorithms using an FM-index can be adapted to use the GBWT. For example, let $P = v_0 \cdots v_{|P|-1}$ be a path. The *reverse path* of P is the path $\overline{P} = \overline{v_{|P-1|}} \ldots \overline{v_0}$ traversing the reverse nodes in the reverse order. If we also index $\overline{P}$ for every path P, the GBWT becomes an FMD-index [14] that supports *bidirectional search*.

3.3 Records

We develop a GBWT representation based on the following assumptions:

1. Almost all nodes $v \in V$ have a low outdegree, making the local alphabet Σ_v small. Hence we can afford storing the rank of all $w \in \Sigma_v$ at the start of BWT_v. Decompressing that information every time we access the node does not take too much time either.
2. The number of occurrences of almost all nodes is bounded by the number of haplotypes. As the length of BWT_v is bounded for almost all $v \in V$, we can afford scanning it every time we compute rank within it.
3. The collection of paths is repetitive. Run-length encoding compresses the BWT well, reducing both index size and the time required for scanning BWT_v.
4. The set of nodes V is a dense subset of a range $[a, b]$. Hence we can afford storing some information for all $i \in [a, b]$ without using too much space.
5. The graph is almost linear and almost topologically sorted. The closer to topological order we can store the nodes, the less space we need for graph topology, and the better we can take advantage of *memory locality*.
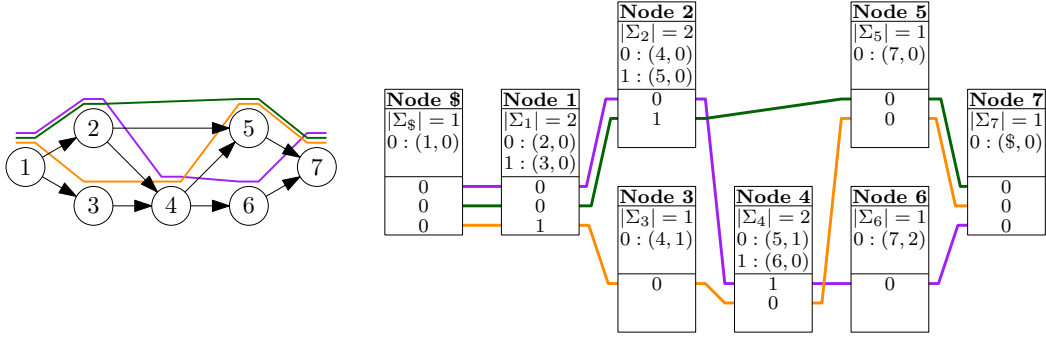
Figure 1 Left: A graph with three paths. Right: GBWT of the paths.

We store a *record* consisting of a *header* and a *body* for each node $v \in V$ and for the endmarker \$. For each character $w \in \Sigma_v$ in sorted order, the header stores a pair $(w, \mathsf{BWT.rank}(v, w))$, where $\mathsf{BWT.rank}(v, w)$ is the total number of occurrences of character w in all $\mathsf{BWT}_{v'}$ with $v' < v$. The body run-length encodes BWT_v, representing a run of ℓ copies of character w as a pair $(\Sigma_v(w), \ell)$. See Figure 1 for an example.

Because the BWT is a set of records, we use node/offset pairs as *positions*. Pair (v, i) refers to offset $\mathsf{BWT}[\mathsf{C}[v] + i] = \mathsf{BWT}_v[i]$. We define rank queries over positions as

$$\mathsf{BWT.rank}((v, i), w) = \mathsf{BWT.rank}(v, w) + \mathsf{BWT}_v.\mathsf{rank}(i, w).$$

Similarly, we define $\mathsf{LF}((v, i), w) = (w, \mathsf{BWT.rank}((v, i), w))$ and use it in place of ordinary LF-mapping in the FM-index.

The FM-index is based on iterating LF-mapping. Because LF-mapping tends to jump randomly around the BWT, this can be a significant bottleneck. GBWT achieves better memory locality, if we store the records for adjacent nodes close to each other. When we iterate LF-mapping over a path in the graph, we traverse adjacent memory regions.

As a run-length encoded FM-index, GBWT supports the *fast* locate() algorithm [18]. The *direct algorithm*, as described in Section 2.2, locates each position $i \in [sp, ep]$ separately. If we instead process the entire range at once, advancing every position by one step of LF-mapping at the same time, we achieve better memory locality. We can also compute LF-mapping for an entire run $\mathsf{BWT}_v[x, y] = w^{y+1-x}$ in the same time as for a single position $i \in [x, y]$.

3.4 GBWT encodings

Dynamic GBWT is intended for index construction, where speed is more important than size. We have an array of fixed-size records for characters \$ and $v \in [a, b]$, including character values $v \notin V$. The record for v has four pointers to arrays: header, body, incoming edges, and haplotype identifiers. For each incoming edge $(u, v) \in E$, the incoming edges array stores a pair $(u, \mathsf{BWT}_u.\mathsf{rank}(|\mathsf{BWT}_u|, v))$, recording the number of paths crossing from u to v.

Let SA_v and DA_v be the parts of SA and DA corresponding to BWT_v. The haplotype identifiers array for node v stores, in sorted order, pairs $(i, \mathsf{DA}_v[i])$ for which $\mathsf{SA}_v[i]$ points to either the last node on a path or a path position divisible by $d > 0$. These pairs are used for locate() queries, like stored SA pointers in an ordinary FM-index.

Compressed GBWT balances query performance with index size. We use it when the set of haplotypes is fixed and for storing the index on disk. Each record is a byte array. We encode integers as sequences of bytes, where the lower 7 bits contain data and the high bit

tells whether the encoding continues. The header starts with $|\Sigma_v|$. We encode the outgoing edges $(w_i, \mathsf{BWT.rank}(v, w_i))$ *differentially*, replacing w_i with $w_i - w_{i-1}$. If the local alphabet is large, each run (k, ℓ) in the body is encoded as an integer pair. Otherwise we encode k and as much of ℓ as possible in the first byte, and continue with the remaining run length in subsequent bytes. We concatenate all records and mark their starting positions in a sparse bitvector [22]. The records can be accessed with select queries on the bitvector.

Each compressed record must be decompressed sequentially. As the stored haplotype identifiers tend to cluster in certain nodes, storing them in records would make these records large and slow to decompress. Instead, we use a global structure for the haplotype identifiers. The structure consists of three bitvectors and an array of identifiers:

- Uncompressed bitvector B_s marks the records with stored identifiers. If the ith record contains identifiers, we set $B_s[i] = 1$. This allows us to skip checking the identifiers in most records when iterating $\mathsf{LF}()$.
- Sparse bitvector B_r is defined over the concatenated offset ranges of the records with stored identifiers. If $B_s[i] = 1$, the range for the record starts at $B_r.\mathsf{select}(B_s.\mathsf{rank}(i, 1) + 1, 1)$.
- Sparse bitvector B_o covers the same range as B_r. If $B_o[i + j] = 1$ and the range for the record starts at $B_o[i]$, we have an identifier for offset j at array position $B_o.\mathsf{rank}(i + j, 1)$.

4   GBWT construction

The assumptions in Section 3.3 make the GBWT easier to build than an ordinary FM-index. *Inserting* new texts into the collection updates adjacent records, just like searching traverses adjacent records. Because the local alphabet is small, because the number of occurrences of each character is limited, and because run-length encoding compresses the BWT well, records tend to be small. Hence we can afford *rebuilding* a record each time we update it.

On the other hand, the GBWT is harder to build than the PBWT. In the PBWT, all strings are of the same length and have the same variant site at the same position. Hence we can build the final record for a site in a single step. In the GBWT, indels in the haplotypes become indels on the haplotype paths, and hence we have to update the same record multiple times. We also have to buffer the strings instead of indexing them as we generate them.

4.1   Basic construction

The following algorithm [11] updates the BWT of text T to be the BWT of text cT, where c is a character. It forms the basis of many incremental BWT construction algorithms.

1. Find the offset i where $\mathsf{BWT}[i] = \$$ and replace the endmarker with character c.
2. Compute $i' = \mathsf{LF}(i, c)$ and insert a new endmarker between offsets $i' - 1$ and i'.

If we have a BWT for m texts, we can insert a new empty text by inserting an endmarker between offsets $m - 1$ and m. By iterating the above algorithm, we can then insert the actual text. If we have a *dynamic FM-index* [3], this can be quite efficient in practice.

The BCR algorithm [1] builds BWT for m texts. It starts with the BWT for m empty texts and then extends each text by one character in each step. Originally intended for indexing short reads, the BCR algorithm is also used for PBWT construction.

Our GBWT construction algorithm is similar to RopeBWT2 [15]. We have a dynamic GBWT and insert multiple texts into the index in a single *batch* using the BCR algorithm. In each step, we extend each text by one character. In the following, v and w are the current and the next character in the current text T_j and i is a record offset. If v is the last character of the text (the endmarker is at $T_j[0]$), we set $w = \$$. In each step, we:

1. **Rebuild records:** The texts are sorted by positions (v, i) such that the endmarker of that text should be at $\mathsf{BWT}_v[i]$. (We do not write the temporary endmarkers to the records.) We process all texts at the same node v to rebuild the record.
   a. If the record does not contain the edge (v, w), we add $(w, 0)$ to the header.
   b. We add BWT runs and haplotype identifiers until offset i to the new record. If we have inserted k characters so far, we replace haplotype identifier (i', j') with $(i' + k, j')$.
   c. If $w = \$$ or the text position is divisible by d, we insert haplotype identifier (i, j).
   d. We insert w to the BWT and set $i \leftarrow \mathsf{BWT}_v.\mathsf{rank}(i, w)$.
   e. If $w \neq \$$, we increment the number of paths from v to w in the incoming edges of w.
2. **Sort:** We sort the texts by (w, v, i), which is the order we need in the next step. If $w = \$$, the text is now fully inserted, and we remove it from further processing.
3. **Rebuild offsets:** For each distinct node w, we rebuild the $\mathsf{BWT}.\mathsf{rank}(v', w)$ fields in the outgoing edges of predecessor nodes v' using the path counts in the incoming edges of w. Then we set $i \leftarrow i + \mathsf{BWT}.\mathsf{rank}(v, w)$ to have the correct offset in the next step.

4.2   Construction in VG

GBWT construction in VG requires a VCF file with *phasing information*. We expect a diploid genome, though some regions may be haploid. Because we need two layers of buffering, we process the VCF file in batches of s samples (default 200) in order to save memory.

At each variant site and for every haplotype, we determine the path from the previous site to the current site, and extend the buffered path for that haplotype with it. If there is no phasing information at the current site or if we cannot otherwise extend the path P, we insert both the path P and its reverse $\overline{P}$ into the GBWT construction buffer and start a new path. Once the GBWT construction buffer is full (the default size is 100 million), we launch a background thread to insert the batch into the index.

We can *merge* GBWT indexes quickly if the node identifiers do not overlap (e.g. indexes for different chromosomes). The records for all nodes $v \in V$ can be reused in the merged index. In the endmarker record $\$$, we merge the local alphabets and concatenate the record bodies in some order. When we interleave the global haplotype identifier structures, we have to update the identifiers according to the order we used in the endmarker.

## 5   Haplotype-aware graph simplification

VG uses *pruning heuristics* to simplify graphs for k-mer indexing. First we remove edges on k-mers that make too many edge choices (e.g. more than 3 choices in a 24-mer). Edges on unary paths are not deleted, as there is no choice in taking them. Then we delete *connected components* with too little sequence (e.g. less than 33 bases). Finally, if the graph contains reference paths, we may restore them to the pruned graph.

Heuristic pruning often breaks paths taken by known haplotypes. This may cause errors in read mapping, if we cannot find candidate positions for a read in the correct graph region. On the other hand, indexing too many recombinations may increase the number of false positives. Hence we would like to prune recombinations while leaving the haplotypes intact.

We describe an algorithm that *unfolds* the haplotype paths in pruned regions, *duplicating* nodes when necessary. Our algorithm works with any pruning algorithm that removes nodes from the graph. See Figure 2 for an example. We work with bidirected VG graphs, unless otherwise noted. Reference paths can also be unfolded with a similar algorithm.

Let $G_i = (V_i, E_i)$ be the graph induced by GBWT paths and $G_p = (V_p, E_p)$ be a pruned graph. We build a *complement graph* induced by edges $E_i \setminus E_p$ and consider each connected
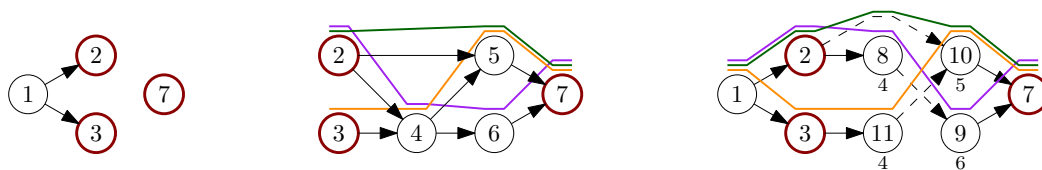
■ **Figure 2** Unfolding the paths in the graph in Figure 1. Border nodes have been highlighted. Left: The graph after removing nodes 4, 5, and 6. Center: Complement graph. The maximal paths are $(2, 4 \mid 6, 7)$, $(2 \mid 5, 7)$, and $(3, 4 \mid 5, 7)$, with the bar splitting a path into a prefix and a suffix. Right: Unfolded graph. Dashed edges cross from prefixes to suffixes. Duplicated nodes have the original ids below the node.

component $G_c = (V_c, E_c)$ in it separately. The set $V_b = V_c \cap V_p$ is the *border* of the component, as the nodes exist both in the component and in the pruned graph. Nodes in the set $V_c \setminus V_b$ are *internal nodes*.

Each connected component G_c represents a graph region that was removed from the original graph. We build an unfolded component consisting of the paths in G_c supported by GBWT paths and insert it into the pruned graph G_p. We achieve this by duplicating the internal nodes that would otherwise cause recombinations.

In order to build the unfolded component, we must find all *maximal* paths P of length $|P| \geq 2$ supported by GBWT paths in the component. A path starting from a border node is maximal if it reaches the border again or cannot be extended any further. GBWT paths consisting entirely of internal nodes of the component are also maximal.

Let v be a GBWT node and $vg(v) \in V_c$ the corresponding VG node. If $vg(v)$ is a border node, we create a *search state* $(v, \mathsf{find}(v))$ consisting of a pattern and a range. For internal nodes, we create state $(v, \mathsf{find}(\$v))$. Then, for each search state $(X, [sp, ep])$, with $x = X[|X| - 1]$:

1. If $|X| \geq 2$ and the last node $vg(x)$ is a border node, we stop the search for this state. If $vg(X[0])$ is also a border node, X is a maximal path, and we output it.

2. We try to extend the search with all GBWT nodes v corresponding to the successors $u \in V_c$ of $vg(x)$, taking the orientation of v from the VG edge. If $[sp', ep'] = [\mathsf{LF}((x, sp), v), \mathsf{LF}((x, ep + 1), v) - 1] \neq \emptyset$, we create a new state $(Xv, [sp', ep'])$.

3. If no extension was successful and $|X| \geq 2$, path X is maximal, and we output it.

Let P be a maximal path we output. If P is not a border-to-border path, we try to extend the lexicographically smaller of P and $\overline{P}$ with reference paths, replacing P with the extended path. To avoid having the same path in both orientations, we replace each path P with the smaller of P and $\overline{P}$.

We could create new duplicates of all internal nodes on P and insert the path into G_p, but this would create too much nondeterminism for GCSA2.[6] Instead, we split each path into a prefix and a suffix of equal length and build a trie of the prefixes and a trie of the reverse suffixes. Every edge in the tries becomes a node in the unfolded component.

Let v be the label of a trie edge starting from the root. If $vg(v)$ is a border node, it already exists in G_p. Otherwise we add a new duplicate of $vg(v)$. Now let v and v' be the

---

[6] If VG node v has predecessors u and u' with identical labels, k-mers starting from u and u' and passing through v cannot be distinguished. GCSA2 construction has to extend these k-mers until the order of the index (e.g. $k = 256$), which may increase the size of the temporary files significantly.

labels of two successive trie edges, and let u be the VG node we used for v. We create a new duplicate u' of $vg(v')$ and add node u' to G_p. We also add edge (u, u') or (u', u), depending on whether we are in a prefix or a suffix. Finally, if we used VG node u for the end of a prefix and VG node u' for the start of the corresponding suffix, we add edge (u, u') to G_p.

After we have handled all components, the simplified graph G_p contains all GBWT paths. The GCSA2 index of G_p contains all k-mers (e.g. 256-mers) in the haplotypes. This allows us to prune the graph more aggressively, removing more k-mers corresponding to recombinations. In order to map reads to the original graph $G = (V, E)$ instead of the simplified graph G_p, we replace the node identifiers $v \in V_p$ in the GCSA2 index with the original identifiers $v' \in V$.

6   Experiments

We have implemented GBWT in C++ using the SDSL library [10]. The following experiments were done using VG v1.7.0 with prerelease versions of GBWT v0.4 and GCSA2 v1.2. All code was compiled using GCC 5.4. We used a single Amazon EC2 i3.8xlarge instance with 16 physical (32 logical) cores of an Intel Xeon E5 2686 v4 and 244 GiB[7] of memory. The system was running Ubuntu 16.04 with Linux kernel 4.4.0. The temporary files in GCSA2 construction were stored on a local RAID 0 volume consisting of four 1.9 TB SSDs.

6.1   GBWT construction

We built VG graphs from the GRCh37 human reference genome and the *1000 Genomes Project* (1000GP) final phase data [27]. The VCF files had phasings for 2504 humans over approximately 80 million variants. VG transformed the phasings into 29.3 million paths of total length 1.62 trillion in a graph with 493 million nodes (including the reverse paths).

GBWT construction is space-efficient and uses two threads. We first built separate GBWTs for each chromosome, running 12 jobs in parallel. The jobs were ordered $X, 1, \ldots, 22, Y$, as large chromosomes take longer to finish. Total construction time was 29.0 hours. The longest job was 27.1 hours for chromosome 2. The bottleneck was generating haplotype paths, as the insertion threads were running less than half of the total time.

Merging the GBWTs into a single index took less than 9 minutes. The merged index took 14.6 GiB, out of which 7.4 GiB was for the GBWT itself and 7.2 GiB for the haplotype identifiers ($d = 1024$). The dynamic GBWT was roughly 10x larger. Its exact size is not well-defined due to a large number of memory allocations and unused space in the arrays.

All the assumptions in Section 3.3 were valid for our dataset:
1. The average outdegree of a record is 1.34 and the maximum is 13, excluding the endmarker.
2. As the graphs built from a VCF file are acyclic, no haplotype can visit the same node twice.
3. The GBWT takes 0.04 bits per character, excluding the haplotype identifiers.
4. VG construction avoids leaving gaps between node identifiers.
5. The VG graphs built from a VCF file are almost in topological order.

6.2   GBWT benchmarks

For various pattern lengths $|X|$ from 2 to 50, we extracted 100,000 patterns from the whole-genome index and used them for find() queries. The average query times in the compressed GBWT start from 460 ns/character with $|X| = 2$ and go down to 300 ns/character with

---

7 Sizes measured in MiB, GiB, and TiB are based on 1024-byte kibibytes. Sizes measured in MB, GB, and TB are based on 1000-byte kilobytes.

Table 1 GCSA2 indexes for simplified 1000GP graphs. Index size in GiB and construction time in hours. Number of distinct 64-mers (in billions) shared with pruned-256, additional haplotype 64-mers, and additional recombination 64-mers.

| | GCSA2 index | | 64-mers | | |
Graph	Size	Construction	Shared	Haplotype	Recombination
pruned-128	35.4 GiB	25.4 h	27.0 G	3.11 G	11.4 G
pruned-256	29.3 GiB	25.5 h	27.0 G	–	–
unfolded-256	33.7 GiB	28.9 h	27.0 G	3.46 G	–

$|X| = 50$ due to memory locality. This is somewhat slower than in FM-indexes over small alphabets [25]. For the dynamic GBWT, query times were 130 ns/character with $|X| = 2$ and 80 ns/character with $|X| = 50$, or 2–3 times faster than in ordinary FM-indexes.

The locate() performance suffers from the long distance $d = 1024$ between stored identifiers. We extracted 20,000 patterns of length 20 from the index and used the ranges returned by find() queries for locate() benchmarks. The total length of the ranges was 69.1 million. The average query times in the compressed GBWT were 110 µs/position (direct algorithm) and 14 µs/position (fast algorithm). For the dynamic index, the times were 19 µs/position (direct) and 11 µs/position (fast). FM-indexes for non-repetitive text typically use $d = 16$ or $d = 32$ and take a few microseconds to locate each position.

We also extracted 100,000 paths of total length 5.50 billion from the index. The average time per character was 1,800 ns in the compressed index and 410 ns in the dynamic index. This is several times slower than in find() queries: find() uses $\mathsf{LF}((v, i), c)$ instead of the slower $\mathsf{LF}(v, i)$, while extract() queries start from the very large record for the endmarker \$. While the direct locate() algorithm also uses $\mathsf{LF}(v, i)$, it benefits from memory locality, as it traverses the same graph region for each position in the query range.

6.3   Haplotype-aware graphs

The typical pruning parameters for a 128-mer GCSA2 index are 4 edge choices in a 16-mer. When building 256-mer indexes, we need to prune more aggressively: 3 edge choices in a 24-mer. This removes more k-mers corresponding to both haplotypes and their recombinations. In the following, graph pruned-k has been pruned with the parameters for a k-mer index, and the reference paths have been restored afterwards. Similarly, unfolded-k is a graph, where the haplotype paths and reference paths have been unfolded after pruning.

We created simplified whole-genome graphs pruned-128, pruned-256, and unfolded-256. The simplification took 3–4 hours for unfolded-256 and slightly less for the other graphs. We then built GCSA2 indexes for both orientations of the simplified graphs. The results can be seen in Table 1. The index for pruned-256 is a few GiB smaller than the others, while the index for unfolded-256 takes a few hours longer to build. Graph pruned-128 contains 90 % of the haplotype k-mers missing from pruned-256 but included in unfolded-256. (Some haplotype k-mers may be recombinations crossing between simple and unfolded regions.) It also contains a large number of additional recombination k-mers not present in unfolded-256.

7   Discussion

We have developed GBWT, a scalable implementation of the graph extension of the PBWT. The earlier implementation used 9.3 hours and 278 GiB of memory to index the 1000GP chromosome 22 using a single thread [21]. In comparison, our implementation takes 4.1 hours and less than 10 GiB of memory using two threads. We also reduced the final index size from

321 MiB to 110 MiB (without haplotype identifiers). By running multiple jobs in parallel, we were able to build a whole-genome GBWT in less than 30 hours on a single system.

Contemporary sequencing projects are sequencing in excess of 100,000 diploid genomes. We intend to scale GBWT to allow working with such large collections, providing a compressed, indexed and searchable representation that should fit into the memory of a single server. Potential applications in genome inference and imputation, as well as for powering population genomic queries, are myriad. The main bottleneck here is construction time. We currently parse the VCF file and find the paths between variant sites once for every 200 samples, which takes the bulk of the time. By parsing the file once and storing the information in a directly usable format, we should be able to double the construction speed.

Storing the haplotype identifiers for locate() queries is another bottleneck. With 5,000 haplotypes, the identifiers use roughly as much space as the GBWT itself. If we increase the number of haplotypes to 50,000, GBWT size should not increase too much, while the identifiers will take 10x more space. We can save space by increasing the distance between stored identifiers, at the expense of increased query times. There is a theoretical proposal for supporting fast locate() queries in space proportional to the size of the run-length encoded BWT [8]. Building the proposed structure for large text collections is still an open problem.

We used the haplotype information in GBWT to simplify VG graphs for k-mer indexing. This allowed us to prune the k-mers corresponding to recombinations more aggressively, while still having all k-mers from the haplotypes in the index. CHOP, the other haplotype-aware graph indexing approach, can only use short-range haplotype information in read mapping. Because VG graphs are connected, we can use the long-range information in the GBWT for mapping long reads and paired-end reads. We will investigate this in a subsequent paper.

References

1 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theoretical Computer Science*, 483:134–148, 2013. `doi:10.1016/j.tcs.2012.02.002`.

2 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994. URL: `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html`.

3 Ho-Leung Chan et al. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2):21, 2007. `doi:10.1145/1240233.1240244`.

4 Richard Durbin. Efficient haplotype matching and storage using the Positional Burrows–Wheeler transform (PBWT). *Bioinformatics*, 30(9):1266–1272, 2014. `doi:10.1093/bioinformatics/btu014`.

5 Hannes P. Eggertsson et al. Graphtyper enables population-scale genotyping using pangenome graphs. *Nature Genetics*, 49:1654–1660, 2017. `doi:10.1038/ng.3964`.

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

7 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017. `doi:10.1016/j.tcs.2017.06.016`.

8 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. ALENEX 2018*, pages 1459–1477. SIAM, 2018. `doi:10.1137/1.9781611975031.96`.

9 Erik Garrison et al. Sequence variation aware genome references and read mapping with the variation graph toolkit. bioRxiv, 2017. `doi:10.1101/234856`.

10 Simon Gog et al. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA 2014*, volume 8504 of *LNCS*, pages 326–337. Springer, 2014. `doi:10.1007/978-3-319-07959-2_28`.

11 Wing-Kai Hon et al. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, 2007. `doi:10.1007/s00453-006-1228-8`.

12 Lin Huang, Victoria Popic, and Serafim Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):i361–i370, 2013. `doi:10.1093/bioinformatics/btt215`.

13 Songbo Huang et al. Indexing similar DNA sequences. In *Proc. AAIM 2010*, volume 6124 of *LNCS*, pages 180–190. Springer, 2010. `doi:10.1007/978-3-642-14355-7_19`.

14 Heng Li. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, 2012. `doi:10.1093/bioinformatics/bts280`.

15 Heng Li. Fast construction of FM-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014. `doi:10.1093/bioinformatics/btu541`.

16 Sorina Maciuca et al. A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference. In *Proc. WABI 2016*, volume 9838 of *LNCS*, pages 222–233. Springer, 2016. `doi:10.1007/978-3-319-43681-4_18`.

17 Tom O. Mokveld et al. CHOP: Haplotype-aware path indexing in population graphs. bioRxiv, 2018. `doi:10.1101/305268`.

18 Veli Mäkinen et al. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010. `doi:10.1089/cmb.2009.0169`.

19 Joong Chae Na et al. FM-index of alignment: A compressed index for similar strings. *Theoretical Computer Science*, 638:159–170, 2016. `doi:10.1016/j.tcs.2015.08.008`.

20 Joong Chae Na et al. FM-index of alignment with gaps. *Theoretical Computer Science*, 710(148-157), 2018. `doi:10.1016/j.tcs.2017.02.020`.

21 Adam Novak, Erik Garrison, and Benedict Paten. A graph extension of the positional Burrows–Wheeler transform and its applications. *Algorithms for Molecular Biology*, 12:18, 2017. `doi:10.1186/s13015-017-0109-9`.

22 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. ALENEX 2007*, pages 60–70. SIAM, 2007. `doi:10.1137/1.9781611972870.6`.

23 Goran Rakocevic et al. Fast and accurate genomic analyses using genome graphs. bioRxiv, 2017. `doi:10.1101/194530`.

24 Korbinian Schneeberger et al. Simultaneous alignment of short reads against multiple genomes. *Genome Biology*, 10(9):R98, 2009. `doi:10.1186/gb-2009-10-9-r98`.

25 Jouni Sirén. Indexing variation graphs. In *Proc. ALENEX 2017*, pages 13–27. SIAM, 2017. `doi:10.1137/1.9781611974768.2`.

26 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`.

27 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–64, 2015. `doi:10.1038/nature15393`.

28 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 2018. `doi:10.1093/bib/bbw089`.