GBZ File Format for Pangenome Graphs

Jouni Sirén [1,*] and Benedict Paten [1]

[1] Genomics Institute, University of California, Santa Cruz, CA 95064, USA

[*] To whom correspondence should be addressed.

Abstract

Motivation: Pangenome graphs representing aligned genome assemblies are being shared in the text-based Graphical Fragment Assembly format. As the number of assemblies grows, there is a need for a file format that can store the highly repetitive data space-efficiently.
Results: We propose the GBZ file format based on data structures used in the Giraffe short read aligner. The format provides good compression, and the files can be efficiently loaded into in-memory data structures. We provide compression and decompression tools and libraries for using GBZ graphs, and we show that they can be efficiently used on a variety of systems.
Availability: C++ and Rust implementations are available at https://github.com/jltsiren/gbwtgraph and https://github.com/jltsiren/gbwt-rs, respectively.
Contact: jouni.siren@iki.fi
Supplementary information: Supplementary data are available online.

1 Introduction

Pangenome graphs (or genome graphs) (Eizenga *et al.*, 2020b) are graphs representing sequence variation. Each node (or sometimes each edge) in the graph is labeled with a sequence, and each path represents the concatenation of the labels on it. Pangenome graphs are often used as reference genomes. Initial efforts have focused on using graphs as technical artifacts to improve accuracy in applications such as read alignment (Garrison *et al.*, 2018; Kim *et al.*, 2019; Rautiainen and Marschall, 2020; Li *et al.*, 2020; Sirén *et al.*, 2021) and genotyping (Eggertsson *et al.*, 2017; Kim *et al.*, 2019; Chen *et al.*, 2019; Hickey *et al.*, 2020; Sirén *et al.*, 2021; Ebler *et al.*, 2022).

There is increasing interest in using graphs as biologically meaningful references that represent sequence variation in the relevant population. The *Human Pangenome Reference Consortium* (HPRC) (Wang *et al.*, 2022) aims to create a human reference genome based on over 350 high quality haplotype-resolved assemblies. The reference will consist of the assembled genomes and pangenome graphs representing their alignments.

Software tools for pangenome graphs use the text-based *Graphical Fragment Assembly* (GFA) format as their data interchange format. GFA was originally intended for assembly graphs, but it is also viable for pangenome graphs with some restrictions and extensions. A GFA file representing only the graph itself is not too large, and it is often compressed further with standard data compressors such as gzip. The format becomes inadequate if we also want to represent the original sequences as paths. While the paths are often highly similar and should compress well, common general-purpose data compressors cannot handle long-distance repetitions. Decompressing a large text file and reading the graph and the paths into in-memory data structures can also be expensive.

We propose the GBZ file format for pangenome graphs representing aligned sequences. The format is based on the data structures used in the Giraffe aligner (Sirén *et al.*, 2021), and it uses a GBWT index (Sirén *et al.*, 2020) for storing a large collection of similar paths space-efficiently. GBZ is a path-based format. Paths representing the original sequences are the primary objects, while the existence of nodes and edges is inferred from usage. We provide a standalone C++ library and compressor for creating and using GBZ graphs, as well as a Rust library for using existing graphs. Both libraries can load GBZ graphs quickly into versatile data structures that are space-efficient enough to enable working with human pangenome graphs on a desktop or a laptop.

Our libraries use Elias–Fano encoded bitvectors for storing increasing integer sequences. We describe various improvements to their semantics and the query interface that make them more practical and may be of independent interest.

Pangenome graph construction pipelines may sometimes create highly collapsed regions that break the assumptions the GBWT makes to ensure fast and space-efficient queries. We show that decompressing and caching parts of the GBWT index is enough to restore performance in specific tasks when such regions are present. Based on this, we suggest changes to in-memory GBWT data structures to make them more robust with highly collapsed graph regions.

2 Methods

2.1 Background

2.1.1 Strings

In the following, we refer to semiopen and closed integer intervals as $[a \dots b) = [a, b) \cap \mathbb{N}$ and $[a \dots b] = [a, b] \cap \mathbb{N}$, respectively.

A *string* $S = s_0 \cdots s_{n-1}$ of length $|S| = n$ is a sequence of *characters* from an ordered *alphabet* Σ. For convenience, we often assume that the alphabet is the set of integers $[0 \dots |\Sigma|)$. We write $S[i] = s_i$ and $S[i \dots j) = s_i \cdots s_{j-1}$ to access individual characters and *substrings* of any sequence S. *Prefixes* and *suffixes* are substrings of the form $S[0 \dots j)$ and $S[i \dots n)$, respectively, and we write them as $S[\dots j)$ and $S[i \dots)$. A *text* string T ends with an *endmarker* $T[|T| - 1] = \$$ that does not occur in any other text position. The endmarker is the smallest character in the alphabet, and we often assume that $\$ = 0$.

Space-efficient data structures rely on *rank* and *select* queries on various sequences. For any string S, position $i \leq |S|$, and character $c \in \Sigma$, we define $S.\mathsf{rank}(i, c)$ as the number of occurrences of character c in the prefix $S[\dots i)$. If $S[i] = c$, we refer to it as the occurrence of rank $S.\mathsf{rank}(i, c)$ of character c. We also define the inverse operation $S.\mathsf{select}(i, c) = j$, where $S[j]$ is the occurrence of rank i of character c. (Note that we use 0-based select, where the first occurrence is of rank 0.)

2.1.2 Bitvectors

A *bitvector* is a data structure that encodes a binary sequence (a string over alphabet $\{0, 1\}$) and supports efficient rank/select queries. We call characters 0 and 1 *unset* and *set* bits, respectively. If A is a strictly increasing sequence of m non-negative integers, we can encode it as a bitvector B, where $B[A[i]] = 1$ for all $i < m$ and $B[j] = 0$ otherwise. If B is a binary sequence, we can decode it as integer sequence A with $A[i] = B.\mathsf{select}(i, 1)$. Query $B.\mathsf{rank}(j, 1)$ can then be understood as the number of values $A[i] < j$ in sequence A.

Plain bitvectors store the binary sequence without any further encoding. There are many efficient structures that support rank and select on plain bitvectors. We use the ones from SDSL (Gog and Petri, 2014).

A binary sequence is *sparse* if the number of set bits is much smaller than the number of unset bits. If a binary sequence is sparse, $|A| \ll |B|$ in the corresponding bitvector. In this paper, a sparse bitvector is a data structure that uses *Elias–Fano* encoding to store a sparse binary sequence space-efficiently (Okanohara and Sadakane, 2007).

Elias–Fano encoding stores the low and high parts of each value $A[i]$ separately. The lowest w bits are stored in an integer array low by setting $\mathsf{low}[i] = A[i] \bmod 2^w$. For the high part, we assign each value x to a bucket using $\mathsf{bucket}(x) = \lfloor x/2^w \rfloor$ and encode the buckets in unary. If a bucket contains k values, we encode it as a binary sequence $1^k 0$. We concatenate the binary sequences for each bucket to form bitvector high. The encoding works best with $w \approx \log_2|B| - \log_2|A|$, which makes the number of buckets similar to the number of values. Bitvector high then contains similar numbers of set and unset bits.

If $\mathsf{high}[j]$ is the set bit of rank i, we can compute

$$A[i] = \mathsf{bucket}(A[i]) \cdot 2^w + \mathsf{low}[i] = (j - i) \cdot 2^w + \mathsf{low}[i].$$

Given i, we can compute $j = \mathsf{high}.\mathsf{select}(i, 1)$. Hence

$$B.\mathsf{select}(i, 1) = A[i] = (\mathsf{high}.\mathsf{select}(i, 1) - i) \cdot 2^w + \mathsf{low}[i].$$

For $B.\mathsf{rank}(i, 1)$, we first find the end of the bucket that would contain value i using $h = \mathsf{high}.\mathsf{select}(\mathsf{bucket}(i), 0)$. The number of values in buckets up to $\mathsf{bucket}(i)$ is $l = h - \mathsf{bucket}(i)$. If $\mathsf{high}[h - 1] = 1$, it is the set bit of rank $l - 1$. If also $\mathsf{low}[l - 1] \geq i \bmod 2^w$, we know that $A[l - 1] \geq i$ and iterate with $h \leftarrow h - 1$ and $l \leftarrow l - 1$. If either of the conditions fails, we return $B.\mathsf{rank}(i, 1) = l$.

2.1.3 Burrows–Wheeler transform

Let T be a text string of length n. The *suffix array* of text T is an array SA of pointers to the suffixes of T in lexicographic order. For any $i < j$, we have $T[\mathsf{SA}[i] \dots) < T[\mathsf{SA}[j] \dots)$. Because the text ends with an endmarker, we always have $\mathsf{SA}[0] = n - 1$.

We can generalize the suffix array for an ordered collection $T_0, \dots, T_{m-1}$ of m texts. Each element of the suffix array is now a pair $\mathsf{SA}[i] = (j, j')$ that refers to suffix $T_j[j' \dots)$ of text T_j. To make the lexicographic order unique, we assume that the endmarker of T_i is smaller than that of T_j for any $i < j$. Then $\mathsf{SA}[i] = (i, |T_i| - 1)$ for $0 \leq i < m$.

The (multi-string) *Burrows–Wheeler transform* (BWT) (Burrows and Wheeler, 1994) of a collection of texts is a permutation BWT of character occurrences in the texts. For each suffix in lexicographic order, the permutation lists the character preceding the suffix. We define the BWT using the suffix array: $\mathsf{BWT}[i] = T_j[j' - 1]$ if $\mathsf{SA}[i] = (j, j')$ and $j' > 0$. If $j' = 0$, we set $\mathsf{BWT}[i] = \$$.

Let C be an array of length $|\Sigma| + 1$ such that $\mathsf{C}[c]$ is the total number of occurrences of all characters $c' < c$ in the texts. For any character c, if $\mathsf{C}[c] \leq i < \mathsf{C}[c + 1]$, and $\mathsf{SA}[i] = (j, j')$, we know that $T_j[j'] = c$. We later use this property for partitioning the BWT into substrings $\mathsf{BWT}_c = \mathsf{BWT}[\mathsf{C}[c] \dots \mathsf{C}[c+1])$ by the first character of the corresponding suffix.

Given a text collection $T_0, \dots, T_{m-1}$, the *lexicographic rank* of string X is the number of suffixes of the collection that are smaller than X. The key operation on the BWT is the *LF-mapping* that maps the lexicographic rank of a suffix to the lexicographic rank of the preceding suffix. In other words, it is a function such that if $\mathsf{SA}[i] = (j, j')$, then $\mathsf{SA}[\mathsf{LF}(i)] = (j, j' - 1)$. We leave this form of LF-mapping undefined when $j' = 0$. A more general form of LF-mapping takes a character $c \in \Sigma$ as a second argument. If the lexicographic rank of string X is i, the lexicographic rank of string cX is $\mathsf{LF}(i, c)$. In particular, $\mathsf{LF}(i) = \mathsf{LF}(i, \mathsf{BWT}[i])$. We leave the function undefined with $c = \$$.

The *FM-index* (Ferragina and Manzini, 2005) is a space-efficient text index based on the BWT. It computes LF-mapping as

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

Let $\mathsf{SA}[i \dots j)$ be the range of suffixes starting with *pattern* (string) P. Then the range of suffixes starting with pattern cP, for any character $c \in \Sigma$, is $\mathsf{SA}[\mathsf{LF}(i, c) \dots \mathsf{LF}(j, c))$. We can also use LF-mapping for extracting individual texts from the BWT. The last character of text T_i is $\mathsf{BWT}[i]$. As long as $\mathsf{BWT}[j] \neq \$$ at the current position j, there may be other characters in the text before the ones we have already extracted. We therefore move to $j \leftarrow \mathsf{LF}(j)$ and find the possible previous character at $\mathsf{BWT}[j]$.

2.1.4 Graphs

Let $G' = (V', E')$ be a *directed graph* with a finite set of *nodes* $V' \subset \mathbb{N}$ and a set of *edges* $E' \subseteq V' \times V'$. Because we often use value 0 for technical purposes and store some information for all integers between node identifiers, we assume that V' is a dense subset of interval $[a \dots b)$ for some $1 \leq a < b$. Let P be a string over alphabet V'. We say that P is a *path* in graph G' if $(P[i], P[i + 1]) \in E'$ for all $0 \leq i < |P| - 1$.

Genome graphs are often represented as *bidirected sequence graphs* $G = (V, E, \ell)$. Each node $v \in V$ has a non-empty string *label* $\ell(v)$, and it can be seen in two *orientations* $o \in \{+, -\}$. If o is an orientation, we use $\overline{o}$ to refer to the other orientation. When we *visit* node v in forward orientation $+$, we read string $\ell(v, +) = \ell(v)$. For a visit in reverse orientation $-$, we read $\ell(v, -) = \overline{\ell(v)}$, where $\overline{X}$ is the reverse complement of string X. The set of edges E is a subset of pairs of node visits $(V \times \{+, -\}) \times (V \times \{+, -\})$ such that $((v, o), (w, o')) \in E$ iff $((w, \overline{o'}), (v, \overline{o})) \in E$. Sequence P of node visits is a path in graph G if $(P[i], P[i+1]) \in E$ for

all i. If P is a path in graph G, the reverse path $\overline{P}$ visits the same nodes in reverse order and in the other orientation. That is, if $P[i] = (v_i, o_i)$, we have $\overline{P}[|P| - i - 1] = (v_i, \overline{o_i})$.

Graph representations often store the topology of a bidirected sequence graph $G = (V, E, \ell)$ as a directed graph $G' = (V', E')$. Each node visit $(v, o) \in V \times \{+, -\}$ becomes a node $w \in V'$ in the directed graph.

The *libhandlegraph* library (Eizenga *et al.*, 2020a) provides a common C++ interface for various bidirected sequence graph implementations. A *HandleGraph* is an immutable graph. *PathHandleGraph* extends it with a set of named paths, where each path has a unique non-empty string name.

2.1.5 GFA file format

We use a subset of the GFA file format version 1.1.[1] The subset is mostly compatible with the bidirected sequence graph data model:

- *S-lines* or *segments* are the nodes of the graph. They consist of a unique non-empty string name and a non-empty string label.
- *L-lines* or *links* are the edges of the graph. They connect two segment visits. We do not allow any overlaps between the connected segments.
- *P-lines* or *paths* are named paths in the graph. They consist of a unique non-empty string name and a non-empty sequence of segment visits.
- *W-lines* or *walks* are paths with metadata suitable for representing assembled genomes. They consist of a sample name (non-empty string), haplotype identifier (integer), sequence name (non-empty string), an optional interval of positions in the named sequence, and a non-empty sequence of segment visits.

In particular, we do not support overlapping segments and annotations with optional tags, as they are more useful with assembly graphs.

2.1.6 GBWT

The *GBWT* (Sirén *et al.*, 2020) is a run-length encoded FM-index that stores a collection of paths in directed graph $G' = (V', E')$ as strings over alphabet V'. Run-length encoding allows it to compress collections of similar paths well. Because we build the FM-index for reverse strings, LF-mapping with character w after character v follows edge (v, w) instead of going backwards on edge (w, v). We break the BWT into substrings BWT_v for each $v \in V'$ and store the substrings in the nodes. Each node $v \in V'$ also stores a list of outgoing edges (v, w) and some rank information that allows computing LF-mapping using locally stored information. We compress the nodes as byte sequences. Queries on the GBWT involve decompressing entire nodes. Hence we assume that nodes do not have too many neighbors and paths do not visit them too many times.

We use the GBWT for storing paths in a bidirected sequence graph $G = (V, E, \ell)$, using the approach described in Section 2.1.4. The GBWT index is usually *bidirectional* (Lam *et al.*, 2009). For each path P, we store both P and $\overline{P}$ in the index, which enables extending the pattern in both directions.

Each path in the graph (which may be stored as two paths in the GBWT index) may be associated with a structured name. The name consists of four integer fields: sample identifier, contig identifier, haplotype/phase identifier, and fragment identifier / running count. Each name is assumed to be unique. GBWT metadata may also contain lists of string names corresponding to sample and contig identifiers.

The *GBWTGraph* (Sirén *et al.*, 2021) is the graph representation used in the Giraffe aligner. It uses a bidirectional GBWT index for graph topology and stores the labels of both orientations of all nodes as a single concatenated string. The graph also caches some information that can be computed from the GBWT for faster access. A GBWTGraph is always a subgraph induced by the paths. Paths representing DNA sequences are the primary objects, while the existence of nodes and edges is inferred from usage. This is not a fundamental limitation of the data structures but a deliberate choice made due to the path-centric algorithms used in Giraffe.

[1] The software versions used in this paper output GFA 1.0, as the status of W-lines in the specification was still unclear when we started the (de)compression benchmarks.

2.2 Improved sparse bitvectors

Elias–Fano encoded sparse bitvectors are often used for partitioning integer intervals $[a \ldots b)$ into subintervals $[\mathsf{select}(i, 1) \ldots \mathsf{select}(i + 1, 1))$. For example, we may want to store an ordered collection of strings $S_0, \ldots, S_{m-1}$ as a single concatenated string X for faster disk I/O. A sparse bitvector B of length $|X|$ can then serve as an index. If we set $B[j] = 1$ at the starting position of each string, we can access the original strings as $S_i = X[B.\mathsf{select}(i, 1) \ldots B.\mathsf{select}(i + 1, 1))$ (with $B.\mathsf{select}(m, 1) = |X|$ for convenience). However, bitvector semantics and the rank/select query interface limit the practicality of this approach.

One common issue is that some of the strings $S[i]$ may be empty. Then $A[i] = A[i + 1]$, which would require that bit $B[A[i]]$ is set twice. However, Elias–Fano encoding is not limited to storing strictly increasing sequences. The encoding and $\mathsf{select}(\,\cdot\,, 1)$ queries work correctly as long as all values in bucket i occur before the values in bucket j in the integer sequence A, for all buckets $i < j$. For $\mathsf{rank}(\,\cdot\,, 1)$ queries, the algorithm works correctly as long as the sequence is non-decreasing. The semantics of $\mathsf{rank}(\,\cdot\,, 0)$ and $\mathsf{select}(\,\cdot\,, 0)$ queries become unclear when duplicate values are allowed.

When we want to retrieve string $S[i]$, we have to compute both $B.\mathsf{select}(i, 1)$ and $B.\mathsf{select}(i + 1, 1)$. Computing them directly is inefficient, as they involve relatively expensive $\mathsf{high.select}(\,\cdot\,, 1)$ queries. To avoid this, select queries should return an iterator with internal state $(i, \mathsf{high.select}(i, 1))$. We can compute $B.\mathsf{select}(i, 1)$ efficiently from those values. When we want to advance the iterator, we set $i \leftarrow i + 1$ and scan for the position of the next set bit in high. This is efficient on the average, as we have chosen a width w such that high contains similar numbers of set and unset bits. The iterator can also be used for decompressing the integer sequence A efficiently.

Given a value $j \in [a \ldots b)$, we may want to retrieve the subinterval $[\mathsf{select}(i, 1) \ldots \mathsf{select}(i + 1, 1))$ containing it, as well as the rank i of the subinterval. This can be achieved using a *predecessor* query $B.\mathsf{pred}(j) = (i, B.\mathsf{select}(i, 1))$, where i is the last position with $A[i] \leq j$. We can compute it naively by first finding the rank $i = B.\mathsf{rank}(j + 1, 1) - 1$ and then computing $B.\mathsf{select}(i, 1)$. This is inefficient, as rank queries on B involve computing $\mathsf{high.select}(\,\cdot\,, 0)$, while select queries on B require $\mathsf{high.select}(\,\cdot\,, 1)$. A more efficient implementation is similar to a rank query. We first find the end of the bucket that would contain value j, and then we iterate backward until we find a value $A[i] \leq j$ or run out of values.

2.3 GBZ file format

2.3.1 General

The GBZ file format is based on the serialization format used in the Simple-SDS library. A file is an array of *elements*: unsigned little-endian 64-bit integers. If the file is memory-mapped, all objects with alignment of 8 bytes or less are properly aligned in memory. A *vector* is serialized by writing the number of items as an element, followed by the concatenated items. If the size of an item is not a multiple of 64 bits, the vector must be padded with 0-bits. Optional or implementation-dependent structures may be serialized by writing the size of the serialized object in elements as an element, followed by the object itself. A reader can then easily skip the object or pass it through as a vector of elements.

Two general principles have guided the development of the GBZ file format: determinism and backward compatibility with the in-memory representation of existing GBWT indexes. The former means that the same data should always be compressed in the same way, without any variation that depends on the implementation or the execution environment. When these principles are in conflict, compatibility has prevailed in the first version of the file format. Compatibility also required making some complex requirements to support old GBWT indexes. Future versions of the file format may choose simplicity and determinism over compatibility.

See Supplement 1 for further details.

2.3.2 Building blocks

Simple-SDS provides a number of basic succinct data structures. They are serialized by encoding some header information as elements and storing the data in lower-level structures. The fundamental structure is the *raw bitvector* that stores a binary sequence of length n in a vector of elements. An *integer vector* stores a sequence of n integers of width w each as a raw bitvector. Plain bitvectors store the data as a raw bitvector, followed by optional rank/select support structures. The support structures are optional and unspecified, as they are implementation-dependent and can be built quickly from the bitvector itself. Sparse bitvectors use a plain bitvector for high and an integer vector for low.

String array stores an ordered collection of strings as a single concatenated string, as described in Section 2.2. The serialized structure uses a sparse bitvector as an index and compresses the strings using *alphabet compaction*. For example, if the strings only contain characters in $\{A, C, G, N, T\}$, they are stored as an integer vector containing values in $[0 \ldots 5)$. In-memory structures may decompress the index as an integer vector and the strings as a byte vector for faster access.

Dictionary encodes a bidirectional mapping between an ordered collection of distinct strings $S_0, \ldots, S_{m-1}$ and their identifiers $[0 \ldots m)$. In the serialized structure, the strings are stored as a string array. An integer vector stores the identifiers in lexicographic order. In-memory structures may decompress the dictionary into a more appropriate representation.

Tags are key-value pairs used for annotation. Keys are case-insensitive strings, and they are assumed to be distinct. Values can be arbitrary strings. Serialized tags are stored as a string array, while in-memory structures may use more appropriate representations. Key `source` identifies the library used for serializing the data. The reader may use it for determining if it can understand any optional structures that are present.

2.3.3 GBWT

The serialization format for the GBWT resembles the compressed in-memory structure. It starts with a header and tags. Node records are compressed as byte sequences using the original encoding, and the sequences are concatenated in a byte vector. A sparse bitvector is used as an index over the nodes.

The GBWT uses document array samples for mapping BWT positions to the identifiers of the corresponding paths. They are serialized as an optional structure in an unspecified format. The structure is optional, because many applications do not need that functionality. The format is left unspecified, as the original structure is complex and not particularly efficient. Future versions of the file format may specify a better structure based on the r-index (Gagie *et al.*, 2020).

GBWT metadata is stored in an optional structure. The structure is optional, because the GBWT may also be useful in applications outside genomics. Metadata starts with a header, followed by a vector of path names. Sample and contig names are serialized as dictionaries.

2.3.4 GBWTGraph

As the GBWTGraph uses a GBWT index for graph topology, it only needs to store a header and node labels. While the in-memory data structure used in Giraffe stores the labels in both orientations for faster access, serializing the reverse orientation is clearly unnecessary. The forward labels are concatenated and stored as a string array.

While GFA graphs have segments with string names, bidirected sequence graphs have nodes with integer identifiers. And while the original graph may have segments with long labels, it often makes sense to limit the length of the labels. For example, the minimizer index used in Giraffe supports nodes of length 1024 bp or less. Some algorithms may create temporary copies of node labels, and long labels may also be inconvenient to visualize.

For this purpose, the GBWTGraph includes a *node-to-segment translation*. The translation consists of a string array storing segment names $S_0, \ldots, S_{m-1}$ and a sparse bitvector B mapping node ranges to segments. If the translation is in use, the set of nodes is assumed to be $V' = [1 \ldots |V'|]$. Segment S_i is then the concatenation of nodes $v \in [B.\mathsf{select}(i, 1) \ldots B.\mathsf{select}(i + 1, 1))$.

2.3.5 GBZ

GBZ is a container that stores a bidirectional GBWT index and a GBWTGraph. It also includes a header and tags. The format interprets GBWT metadata in a specific way.

Paths corresponding to sample name `_gbwt_ref` are named paths. They correspond to paths in a PathHandleGraph and P-lines in GFA. The name of the path is stored as a contig name.

Other paths correspond to GFA W-lines. GBWT path names map to GFA walk metadata in the following way. Sample names and haplotype identifiers can be used directly. GBWT contig names become GFA sequence names. GBWT fragment identifier is used as the start of GFA sequence interval.

2.4 GFA compression

2.4.1 Compression algorithm

We propose an algorithm that compresses a GFA file in the GBZ format in several passes over a memory-mapped file. The algorithm works best when, for each weakly connected component in the graph and each line type, all lines from that component with that type are consecutive in the file.

In the first pass, the algorithm determines the starting position and type of each line in the file. It determines whether a node-to-segment translation is necessary. A translation is created if a segment contains a sequence longer than the user-defined threshold (1024 bp by default) or if segment names cannot be interpreted as integer identifiers. The algorithm also chooses an appropriate size for GBWT construction buffers based on the length of the longest path.

In the second pass, the algorithm processes all segments. It builds the node-to-segment translation if necessary and stores node labels in a temporary structure. The third pass processes links and creates a temporary graph. The algorithm finds the weakly ordered components of the graph and sorts them by the minimum node identifier. It then determines GBWT construction jobs by using large components directly as jobs and combining consecutive small components into larger jobs.

The fourth pass builds GBWT metadata from P-lines and W-lines. If there are W-lines present, the algorithm interprets P-lines as named paths, as described in Section 2.3.5. Otherwise the algorithm breaks P-line names info fields using a user-provided regex and maps the fields to the components of GBWT path names.

GBWT construction happens in the fifth and final pass that builds a separate GBWT index for each job. The jobs are started from the largest to

the smallest (by the number of nodes in the components), and multiple jobs can be run in parallel. Each job first processes P-lines and then W-lines in the corresponding components and appends the paths into a buffer. When the buffer is full, the paths are inserted into the GBWT index using the direct construction algorithm (Sirén *et al.*, 2020).

After all construction jobs have finished, the algorithm merges the partial GBWT indexes into a final index. Because the jobs are based on weakly connected components, the partial indexes do not overlap, and node records from them can be used directly in the merged index (Sirén *et al.*, 2020). The algorithm builds a GBWTGraph from the final GBWT index, node labels, and node-to-segment translation, and then it serializes everything in the GBZ format.

2.4.2 Decompression algorithm

Decompressing GFA from a GBZ file is straightforward. We first iterate over the node-to-segment translation (or nodes if there is no translation) and output the S-lines in that order. Then we iterate over all edges in an arbitrary order, select the ones that connect segments, and output them as L-lines. Because each edge $((v, o), (w, o'))$ of a bidirected sequence graph implies the existence of the reverse edge $((w, \overline{o'}), (v, \overline{o}))$, we only write the smaller of the two (in the usual tuple ordering) as an L-line. Writing L-lines is slower than writing S-lines, because it requires partial decompression of the node records. Hence it may be useful to write them using multiple threads.

Then we write named paths as P-lines and finally other paths as W-lines, both in an arbitrary order. We interpret GBWT metadata as described in Section 2.3.5. As extracting the paths from the GBWT index is the most expensive part of decompression, this should be done using multiple threads.

3 Results

3.1 Experimental setup

The main GBZ library is written in C++. It builds on the original GBWT implementation. The graph supports the the PathHandleGraph interface from libhandlegraph, exposing named paths through the interface. Other paths can be accessed using a custom interface. There is also a standalone Rust library that can use existing GBWT and GBZ structures but not build new ones.

There are some differences between the implementations. Because the C++ implementation uses Giraffe data structures, it decompresses and stores node labels in both orientations. The Rust implementation only stores the forward orientation. The GFA decompression algorithm in the C++ implementation uses additional memory to speed up decompression. It decompresses the node-to-segment translation and large (>1024-byte) GBWT node records for faster access. The Rust implementation uses the query interface directly without caching. Finally, the C++ implementation writes both edges and paths using multiple threads, while the Rust implementation only uses multiple threads for paths.

Both implementations use an external library for basic succinct data structures. The C++ implementation uses the vgteam fork of SDSL (Gog *et al.*, 2014), which includes sparse bitvector improvements and supports the Simple-SDS serialization format. The Rust implementation uses Simple-SDS.

As our test data, we used two 90-haplotype human graphs based on year 1 data from HPRC (Liao *et al.*, 2022). Cactus was built using the Minigraph–Cactus pipeline (Hickey *et al.*, 2022) with GRCh38 as the reference. PGGB was built using the Pangenome Graph Builder pipeline (Garrison *et al.*, 2022). We also created a 1000GP dataset with 5008 human haplotypes by combining the full 1000 Genomes Project (The 1000 Genomes Project Consortium, 2015) indexes from the Giraffe paper (Sirén *et al.*, 2021). See Table 1 for details.

Chromosome 16 of the PGGB graph contains a highly collapsed repetitive region, where the average haplotype visits a few nodes tens of thousands or even hundreds of thousands of times. These nodes also have a large number of neighbors. This breaks the assumptions made by the GBWT, making compression much slower than expected. The C++ implementation successfully mitigates the issue during decompression by caching large node records. Hence a better in-memory representation for large records should be enough to fix this and other similar issues.

We used four systems for the experiments: Desktop (iMac 2020), Laptop (MacBook Air 2020), Intel Server (AWS i3.8xlarge), and ARM Server (AWS r6gd.8xlarge). See Table 2 for details. All data was stored on a local SSD. While we used different C++ compilers on different systems, the Rust compiler was always rustc version 1.58.1.

See Supplement 2–4 for further details.

3.2 Compression

We compressed the Cactus and PGGB datasets in the GBZ format using the compressor included in the C++ implementation. File sizes were 3.6 times and 2.5 times smaller, respectively, than those achieved by the general-purpose gzip compressor (see Table 1). The compression ratio improves further as the number of haplotypes increases, as seen with the 1000GP dataset.

We measured the time and memory usage of GBZ compression and the time usage of gzip compression. (Memory usage of gzip compression is negligible.) The results can be seen in Table 3 and Table 4 for Cactus and PGGB, respectively. Because the input GFA is memory-mapped, any parts of the input stored in disk cache are included in the memory usage of the compressor. We did not use Laptop for this benchmark due to the limited amount of memory.

With the Cactus dataset, multi-threaded GBZ compression required similar time to single-threaded gzip compression. Intel Server and ARM Server were faster than Desktop, both with individual GBWT construction jobs and by the total running time multiplied by the number of parallel jobs. This was likely caused by the higher memory bandwidth of the server processors.

Compression speed of the PGGB dataset was dominated by the highly collapsed region of chromosome 16. Without chromosome 16, the compression would have finished in 60–90 minutes (depending on the system), which is slightly higher than gzip compression time. The GBWT construction job for chromosome 16 took an order of magnitude longer. Dealing with such collapsed regions will require changes to the in-memory data structures used in the GBWT (see Section 4). Desktop was faster than the servers due to the higher single-core speed of its processor. Its memory usage was also lower, because there was not enough memory for caching the entire input.

3.3 In-memory structures

The GBZ file format has been designed for fast loading into in-memory data structures. In the C++ implementation, these are the same GBWT and GBWTGraph structures that are used in the Giraffe aligner. The structures in the Rust implementation are similar. While some structures must be rebuilt when loading the file and other parts are decompressed for faster access, most of the GBWT can be copied from the file to the data structures.

We measured the time and memory usage of loading GBZ files. The results can be seen in Table 3 and Table 4. Loading the structures took tens of seconds. Desktop and Laptop were faster than Intel Server and ARM Server due to their higher single-core performance. The C++ implementation used more memory than the Rust implementation, because it stores node labels in both orientations.

Laptop used less memory than the other systems, especially with the PGGB dataset. This is because we defined memory usage as resident set

Table 1. Datasets and their properties. We list the size of the file in uncompressed and gzip-compressed GFA format as well as in the GBZ format. We also list the total length (in nodes) of the forward and reverse paths stored in the GBWT index and the number of lines of each type in the file.

Dataset	.gfa	.gfa.gz	.gbz	Total length	S-lines	L-lines	P-lines	W-lines
Cactus	44.9 GiB	11.1 GiB	3.11 GiB	8.8 billion	81.4 million	113.0 million	2580	24456
PGGB	88.6 GiB	14.6 GiB	5.73 GiB	16.3 billion	110.9 million	154.8 million	34796	0
1000GP	9534.9 GiB	—	16.84 GiB	2125.1 billion	293.2 million	372.9 million	0	115184

Table 2. Systems used for the experiments. Jobs indicates the number of parallel compression/decompression jobs.

System	Processor	CPU Cores	Jobs	RAM	OS	C++ Compiler
Desktop	Intel Core i9-10910	10 physical (20 logical)	10 / 10	128 GiB	macOS 12.2.1	GCC 11.2.0
Laptop	Apple M1	4 performance + 4 efficiency	– / 4	16 GiB	macOS 12.2.1	Apple Clang 13.0.0
Intel Server	Intel Xeon E5-2686 v4	16 physical (32 logical)	16 / 16	244 GiB	Ubuntu 20.04	GCC 9.3.0
ARM Server	AWS Graviton2	32	16 / 32	256 GiB	Ubuntu 20.04	GCC 9.3.0

Table 3. Wall clock time and peak memory usage for various tasks with the Cactus dataset.

System	Compression	gzip	Loading (C++)	Loading (Rust)	Decompression (C++)	Decompression (Rust)	gunzip
Desktop	40 min / 96.5 GiB	25 min	23 s / 11.8 GiB	19 s / 5.9 GiB	116 s / 15.5 GiB	239 s / 7.1 GiB	80 s
Laptop	—	—	23 s / 9.4 GiB	16 s / 5.9 GiB	186 s / 9.7 GiB	304 s / 6.5 GiB	80 s
Intel Server	19 min / 111.5 GiB	39 min	37 s / 11.7 GiB	35 s / 5.9 GiB	125 s / 14.5 GiB	193 s / 6.5 GiB	361 s
ARM Server	16 min / 111.0 GiB	48 min	33 s / 11.7 GiB	33 s / 5.9 GiB	86 s / 14.5 GiB	138 s / 7.1 GiB	350 s

Table 4. Wall clock time and peak memory usage for various tasks with the PGGB dataset.

System	Compression	gzip	Loading (C++)	Loading (Rust)	Decompression (C++)	Decompression (Rust)	gunzip
Desktop	890 min / 114.7 GiB	34 min	43 s / 27.2 GiB	39 s / 13.1 GiB	309 s / 32.7 GiB	7226 s / 14.2 GiB	121 s
Laptop	—	—	70 s / 8.0 GiB	40 s / 7.6 GiB	638 s / 7.2 GiB	18451 s / 12.5 GiB	116 s
Intel Server	1365 min / 194.6 GiB	54 min	73 s / 27.1 GiB	73 s / 13.0 GiB	335 s / 32.2 GiB	9194 s / 13.5 GiB	603 s
ARM Server	1232 min / 194.3 GiB	66 min	65 s / 27.1 GiB	68 s / 13.0 GiB	187 s / 33.5 GiB	3499 s / 14.7 GiB	551 s

size. When memory usage approaches memory capacity, the operating system starts swapping out inactive memory regions to compressed memory and ultimately to disk.

3.4 Decompression

We decompressed the Cactus and PGGB datasets from GBZ format to GFA format on all four system using the C++ and Rust decompressors. Time and memory usage of can be seen in Table 3 and Table 4. We also measured the time used by gzip decompression for a comparison.

With the Cactus dataset, the multi-threaded C++ decompressor was about as fast as the single-threaded gzip decompressor on macOS. The Linux version of gzip was several times slower. The Rust decompressor was also slower, because it uses the query interface directly without caching. ARM Server was faster than Intel Server due to having more CPU cores.

Chromosome 16 in the PGGB dataset caused issues again. The C++ decompressor managed to decompress it in a reasonable time, as it caches large GBWT node records. Decompression time was reasonable even on Laptop, which only had half the memory required for the in-memory data structures. This is because the memory access patterns during decompression are mostly sequential, and swapping does not slow it down too much. The Rust implementation was more than an order of magnitude slower than the C++ implementation. Gzip decompression was again slower in Linux than in macOS.

3.5 Scalability

The 1000GP dataset contains 55.6 times more haplotypes than the Cactus and PGGB datasets, and the total length of the paths is 240.7 times and 130.7 times higher, respectively. However, the GBZ file is only 5.41 times and 2.94 times larger, respectively. The GFA file would be too large to store explicitly on any of our systems, and even the gzip-compressed file would likely be too large for all systems except Intel Server.

We decompressed the GBZ file using the C++ implementation and piped the output to `wc` to determine the size of the GFA file. This took 12.8 hours / 45.6 GiB on Intel Server and 18.2 hours / 51.2 GiB on ARM Server. The `wc` tool was the bottleneck in decompression. On the average, only about 8 parallel decompression threads were active on Intel Server. On ARM Server, the average number of threads was approximately 5.5.

4 Discussion

We have proposed the GBZ file format for pangenome graphs representing aligned genomes. The file format is based on data structures used in the Giraffe aligner, and it is the preferred graph format for the aligner. GBZ graphs are widely supported in vg (Garrison *et al.*, 2018), and we also provide standalone libraries for using them in other software tools.

GBZ compresses GFA files with many similar paths well. The compression speed is competitive as long as the graph does not contain

certain degenerate structures. A GBZ file can be decompressed quickly into a GFA file for tools that do not support GBZ. The graph and the paths in the file can also be loaded quickly into in-memory data structures.

There are two main data models for representing paths metadata in pangenome graphs. The W-line / GBWT model is based on structured names appropriate for assembled genomes. The P-line / libhandlegraph model uses string names, which are harder to interpret but suitable for more diverse applications. GBZ supports both models. Some applications of pangenome graphs require specifying a set of paths that acts as a linear reference sequence. Neither model has a way of specifying whether the graph contains a preferred reference sequence or if any sample is an equally valid choice. If such a convention emerges, GBZ can support it using tags.

The GBWT assumes that the nodes of the graph do not have too many neighbors and the paths do not visit the same nodes too many times. Pangenome graph construction pipelines sometimes produce graphs that violate these assumptions. Because the GBWT has to decompress the node record every time it computes LF-mapping from that node, this can cause significant performance loss. However, as we saw in the decompression benchmarks (Section 3.4), caching large records in a more suitable format helps to avoid the issue. While the format we used was only appropriate for extracting paths from the GBWT, a proper solution such as a run-length encoded wavelet tree should work in most situations. The ones from the DYNAMIC library (Prezza, 2017) look promising. However, using them in the GBWT may require significant engineering effort and changes to the query interface.

Acknowledgements

We would like to thank members of the Human Pangenome Reference Consortium, in particular Glenn Hickey, Heng Li and Erik Garrison for providing pangenome graphs to experiment with. We also thank Adam M. Novak for the idea of a combined GBWT + GBWTGraph file format and for contributions to the C++ implementation. Finally, we would like to thank everybody in the VG Team.

Funding

This work was supported by the National Human Genome Research Institute of the National Institutes of Health [U01HG010961, R01HG010485, U41HG010972, U24HG011853, OT2OD026682]. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

References

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

Chen, S. *et al.* (2019). Paragraph: a graph-based structural variant genotyper for short-read sequence data. *Genome Biology*, **20**, 291.

Ebler, J. *et al.* (2022). Pangenome-based genome inference allows efficient and accurate genotyping across a wide spectrum of variant classes. *Nature Genetics*.

Eggertsson, H. P. *et al.* (2017). Graphtyper enables population-scale genotyping using pangenome graphs. *Nature Genetics*, **49**, 1654–1660.

Eizenga, J. M. *et al.* (2020a). Efficient dynamic variation graphs. *Bioinformatics*, **36**(21), 5139–5144.

Eizenga, J. M. *et al.* (2020b). Pangenome graphs. *Annual Review of Genomics and Human Genetics*, **24**, 139–162.

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

Gagie, T. *et al.* (2020). Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, **67**(1), 2.

Garrison, E. *et al.* (2018). Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, **36**(9), 875–879.

Garrison, E. *et al.* (2022). The pangenome graph builder (PGGB). Manuscript in preparation.

Gog, S. and Petri, M. (2014). Optimized succinct data structures for massive data. *Software: Practice and Experience*, **44**(11), 1287–1314.

Gog, S. *et al.* (2014). From theory to practice: Plug and play with succinct data structures. In *Proc. SEA 2014*, volume 8504 of *LNCS*, pages 326–337. Springer.

Hickey, G. *et al.* (2020). Genotyping structural variants in pangenome graphs using the vg toolkit. *Genome Biology*, **21**, 35.

Hickey, G. *et al.* (2022). Constructing pangenome graphs with minigraph/cactus. Manuscript in preparation.

Kim, D. *et al.* (2019). Graph-based genome alignment and genotyping with HISAT2 and HISAT-genotype. *Nature Biotechnology*, **37**(8), 907–915.

Lam, T. *et al.* (2009). High throughput short read alignment via bi-directional BWT. In *Proc. BIBM 2009*, pages 31–36. IEEE.

Li, H. *et al.* (2020). The design and construction of reference pangenome graphs with minigraph. *Genome Biology*, **21**, 265.

Liao, W.-W. *et al.* (2022). A draft human pangenome reference. bioRxiv.

Okanohara, D. and Sadakane, K. (2007). Practical entropy-compressed rank/select dictionary. In *Proc. ALENEX 2007*, pages 60–70. SIAM.

Prezza, N. (2017). A framework of dynamic data structures for string processing. In *Proc. SEA 2017*, volume 75 of *LIPIcs*, pages 11:1–11:15.

Rautiainen, M. and Marschall, T. (2020). GraphAligner: rapid and versatile sequence-to-graph alignment. *Genome Biology*, **21**, 253.

Sirén, J. *et al.* (2020). Haplotype-aware graph indexes. *Bioinformatics*, **36**(2), 400–407.

Sirén, J. *et al.* (2021). Pangenomics enables genotyping of known structural variants in 5202 diverse genomes. *Science*, **374**(6574), abg8871.

The 1000 Genomes Project Consortium (2015). A global reference for human genetic variation. *Nature*, **526**, 68–64.

Wang, T. *et al.* (2022). The human pangenome project: a global resource to map genomic diversity. *Nature*, **604**, 437–446.