Pangenome Graphs with Haplotype Paths

Jouni Sirén
UC Santa Cruz Genomics Institute

Data model

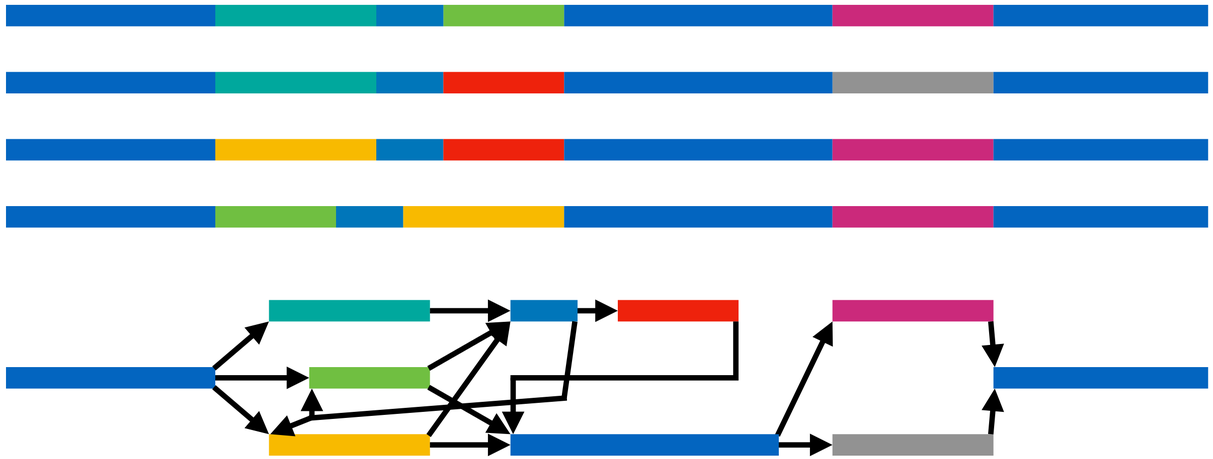
We have a representative set of **haplotypes** from the relevant population.

We **align** the haplotypes and build a **graph**, where each node represents aligned positions in the haplotypes.

Any **traversal** of the graph is a potential haplotype.

Traversals that are **locally consistent** with the original haplotypes are more likely to be **biologically plausible**.

For that reason, we store the original haplotypes as **paths**.



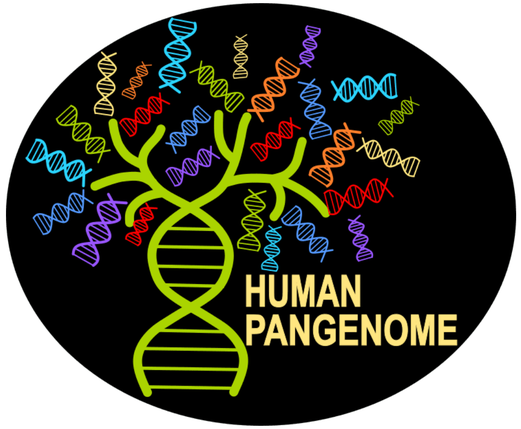
path ~ walk
path ~ stored traversal
traversal ~ emergent path

Human genomes

The methods I am going to present are general-purpose, but we always make **assumptions**.

I mostly work with **human genomes**, which has implications on things like:

- genome size;
- number and size of chromosomes;
- ploidy;
- repetitiveness of sequences; and
- similarity between haplotypes.



Human Pangenome Reference Consortium (HPRC)

Wang et al.: **The Human Pangenome Project: a global resource to map genomic diversity**. Nature, 2022.

https://humanpangenome.org/

~45 high-quality diploid de novo assemblies today and hundreds in the future.

Outline

Today: Data structures

- Bitvectors, rank, select
- Burrows–Wheeler transform (BWT)
- FM-index, RLBWT
- GBWT, construction algorithms

We will revisit topics Veli already talked about on Monday, but from a different perspective.

Tomorrow: GBWT applications

- Bidirected sequence graphs
- GBWTGraph
- Giraffe aligner
- GBZ file format

Focus on **algorithm engineering** and a little bit of software development, not on theoretical algorithms or biology.

Bitvectors

Notation

There are only two hard things in computer science: cache invalidation, naming things, and off-by-one errors.
(Phil Karlton, Leon Bambrick)

Off-by-one errors are often caused by incorrect translations between various array indexing **conventions**.

Many popular programming languages such as **C++** and **Rust** start array indexing from 0 and use **semi-open intervals** for representing substrings.

I am going to use the same conventions here.

Substring S[i..j) starts with S[i] and ends just before S[j].

S.rank(i, c) is the number of occurrences of character c in the prefix S[0..i).

Let A_c be the sorted array of positions of character c in string S.

S.select(i, c) = A_c[i] is the position of the occurrence of rank i.

Bitvectors

A **bitvector** represents a binary sequence B and supports efficient rank/select queries.

Bitvectors are often used for representing the **sorted integer array** A = A_1.

A common application is **partitioning** an interval [a..b) into subintervals [B.select(i, 1)..B.select(i + 1, 1)).

Offset j can be mapped to the subinterval containing it with a **predecessor** query B.pred(j) = (i, B.select(i, 1)), where i = B.rank(j + 1, 1) – 1.

B: 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 1 1 0 1 0 0
A: 2 3 7 8 12 13 16 17 19

B.rank(10, 1) = 4

B: 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 1 1 0 1 0 0
A: 2 3 7 8 12 13 16 17 19

B.select(5, 1) = 13

B: 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 1 1 0 1 0 0
A: 2 3 7 8 12 13 16 17 19

Rank on plain bitvectors

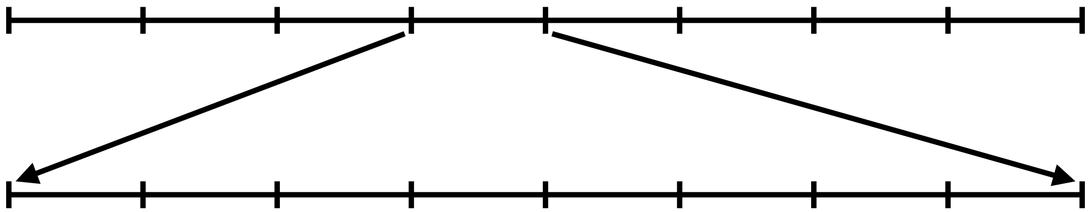
A **plain bitvector** stores binary sequence B as such. There are many structures that support rank queries in O(1) time.

The following is from SDSL: Gog, Petri: **Optimized succinct data structures for massive data**. Software – Practice and Experience, 2014.

A CPU can execute multiple **independent** operations in **parallel** using a single core.

Chained queries (such as in iterated LF-mapping) are bound by **memory latency**.

Partition the bitvector into 512-bit **blocks** and store the rank at the start of each block using 64 bits.



Partition each block into 64-bit **words** and store rank-within-block at the start of each word (except the first) using 9 bits.

Compute rank-within-word using popcnt and return the sum of the three ranks. A query takes two memory accesses and the space overhead is 25%.

Select on plain bitvectors

select queries are also $O(1)$ in theory, but practical implementations tend to have rare polylogarithmic worst cases.

The following is also from SDSL.

We partition the bitvector into **superblocks** of 4096 **values** (positions of ones) and store the first value in each superblock.

If a superblock is longer than $\log^4 |B|$ bits, we store all values in it explicitly.

Otherwise we partition the superblock into **blocks** of 64 values and store the first value in each block relative to the start of the superblock.

Within each block, we iterate popcnt to find the **word** containing the position we are interested in. This means $O(\log^3 |B|)$ iterations in the worst case.

Select-within-word uses uses somewhat complicated bit manipulation.

Space overhead is 18.75% in the worst case.

Elias–Fano encoding

Elias–Fano encoding is good for **sparse** bitvectors, where $|A| \ll |B|$. It is a mix between representations A and B.

For each value x, we store the lowest w bits in integer sequence low and assign the value to **bucket** $\text{floor}(x / 2^w)$.

We encode the buckets in **unary**: a bucket with k values becomes 1^k0. Concatenated buckets form binary sequence high.

By choosing $w \approx \log |B| - \log |A|$, the number of buckets will be close to |A|, making the density of high close to 0.5.

B: 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 1 1 0 1 0 0
A: 2 3 7 8 12 13 16 17 19

w = 2

Value	2	3	7	8	12	13	16	17	19
Low	2	3	3	0	0	1	0	1	3
Bucket	0	0	1	2	3	3	4	4	4

high: 1 1 0 1 0 1 0 1 1 0 1 1 1 0

Sparse bitvectors

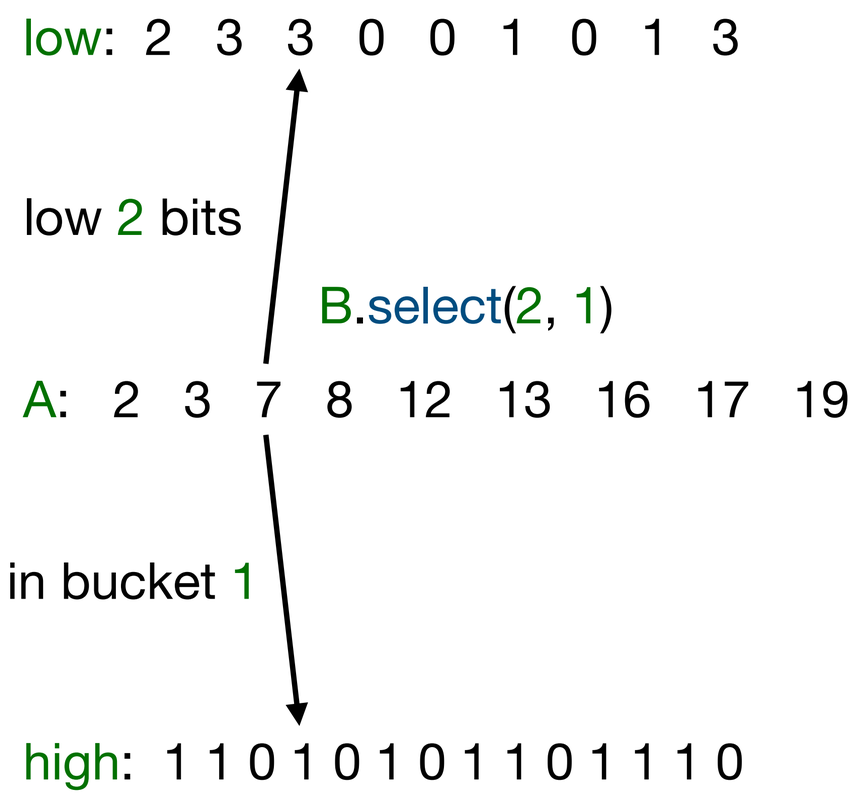
Accessing the original values is simple:
A[i] = (high.select(i, 1) – i) · 2^w + low[i].

We can **iterate** over A by iterating over high and low.

A B.rank(i, 1) query starts by finding the end of the bucket with high.select(floor(i / 2^w), 0). We then iterate backward as long as the values are too large.

B.pred(i) can be answered directly in a similar way.

Okanohara, Sadakane: **Practical Entropy-Compressed Rank/Select Dictionary**. ALENEX 2007.



Sparse bitvectors

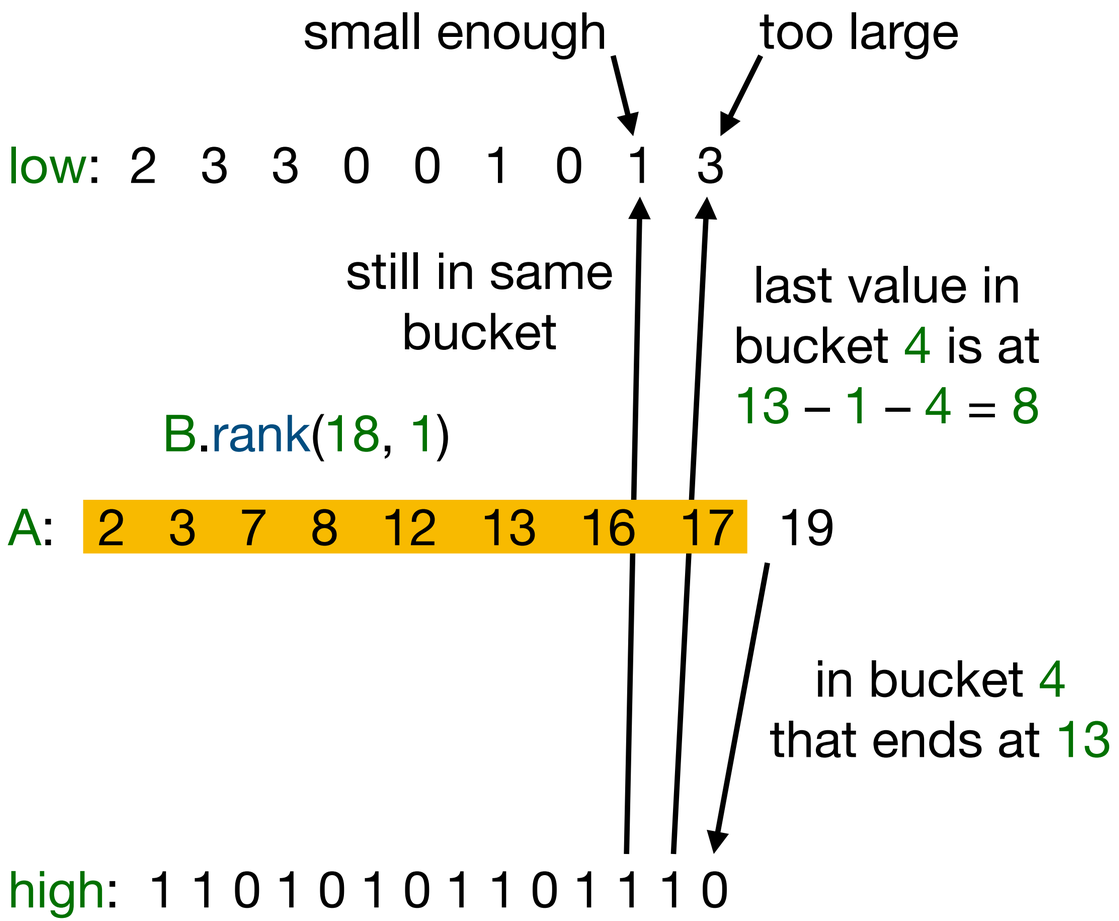
Accessing the original values is simple:
A[i] = (high.select(i, 1) – i) · 2^w + low[i].

We can **iterate** over A by iterating over high and low.

A B.rank(i, 1) query starts by finding the end of the bucket with high.select(floor(i / 2^w), 0). We then iterate backward as long as the values are too large.

B.pred(i) can be answered directly in a similar way.

Okanohara, Sadakane: **Practical Entropy-Compressed Rank/Select Dictionary**. ALENEX 2007.



SDSL versions

Succinct Data Structures Library (SDSL)

Gog et al.: **From Theory to Practice: Plug and Play with Succinct Data Structures**. Proc. SEA 2014.

https://github.com/simongog/sdsl-lite

Wide range of efficient and scalable data structures.

Unfortunately the original library (SDSL 2) has been abandoned.

SDSL 3

- https://github.com/xxsds/sdsl-lite
- Maintained (for now) by SeqAn people.
- Requires C++17.

vgteam fork

- https://github.com/vgteam/sdsl-lite
- Some improvements to SDSL 2.
- Limited support.

Simple-SDS

- https://github.com/jltsiren/simple-sds
- Limited scope, written in Rust.

Burrows–Wheeler transform

Empirical entropy

Entropy and **information** are based on the equation $H = -\sum_{c \in \Sigma} P(c) \log_2 P(c)$, where Σ is an **alphabet** (a finite set of symbols).

We can interpret this as the expected **number of bits** required for **encoding** one symbol $c \in \Sigma$.

If we use observed frequencies (in a specific text) as probabilities, we get the order-0 **empirical entropy** H_0, which is useful in data compression.

If we have an order-0 encoder (such as Huffman), we can **compress** a text of length n using nH_0 bits, plus **overhead** from the encoder and the **model** (probability distribution).

In (one version of) GRCh38, $H_0 \approx 2.17$:

Symbol	Count	Probability
A	866420001	0.279497
C	598683433	0.193129
G	600854940	0.193829
N	165045996	0.053242
T	868918077	0.280303

High-order entropy

If we know the **context** a symbol appears in, we can often **predict** it more accurately.

Given a set of possible contexts S, we can redefine entropy as the **weighted sum** $H = -\sum_{s \in S} P(s) \sum_{c \in \Sigma} P(c \mid s) \log_2 P(c \mid s)$ of entropies over contexts $s \in S$.

By using the k preceding characters as the context, we get the **order-k empirical entropy** H_k.

This does not help much with DNA, but the usual estimate for the entropy of English text is approximately 1 bit/character.

By using a separate order-0 **encoder for each context**, we can compress a text of length n using nH_k bits, plus overhead.

At $k \approx \log_{|\Sigma|} n$, **model overhead** becomes the dominant term in space usage.

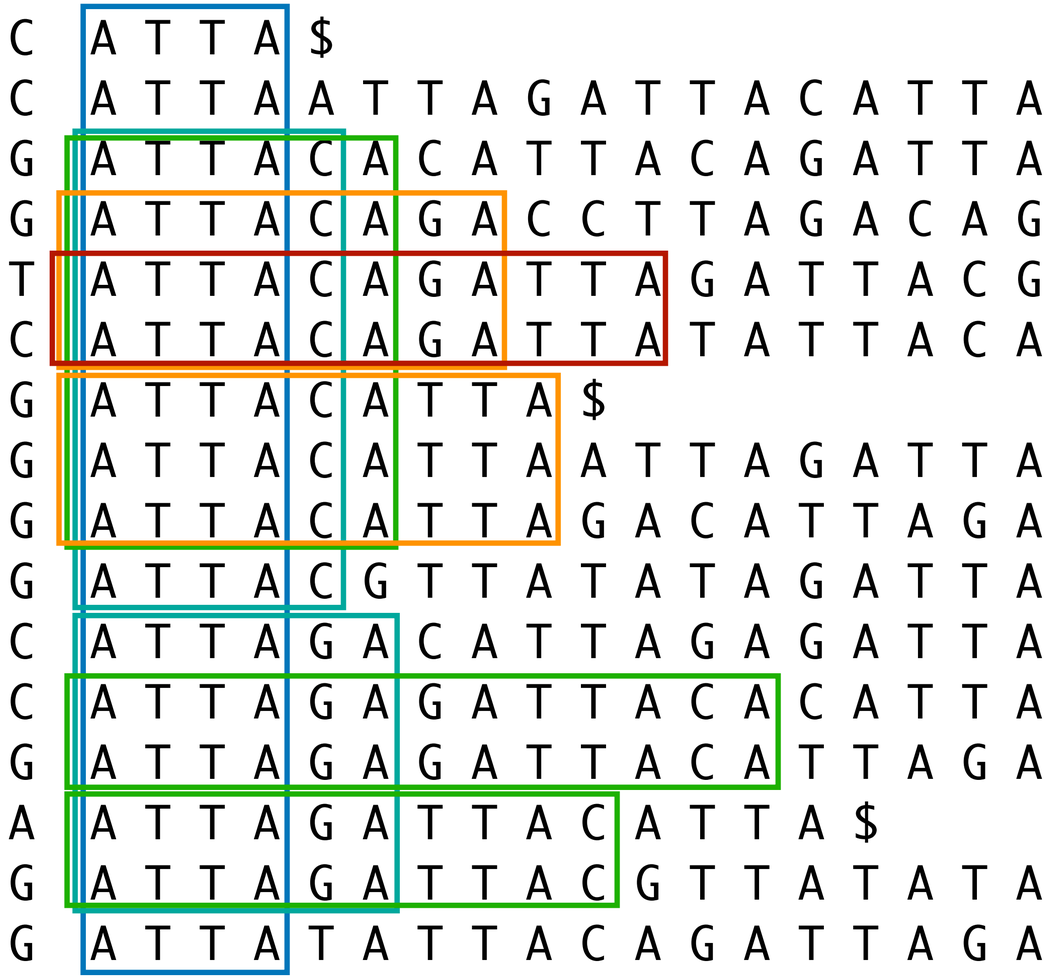
We need a **better encoding** for the model.

Burrows–Wheeler transform

The **Burrows–Wheeler transform** (BWT) is a permutation of the text that is useful for encoding both the text and the model.

If we **sort** each character occurrence by the **text following** it, we group them by the **order-k context for all k** simultaneously.

Burrows, Wheeler: **A Block-sorting Lossless Data Compression Algorithm**. Technical report, 1994.



Compression boosting

We can partition the BWT into **optimal contexts** (according to a cost function) using a greedy algorithm and compress each of them **separately** with an order-0 encoder.

This theoretical approach to BWT-based compression is called **compression boosting**.

Ferragina et al.: **Boosting Textual Compression in Optimal Linear Time**. Journal of the ACM, 2005.

We get similar results with a **specific order-0 encoder** (wavelet tree with RRR bitvectors) without any partitioning.

Mäkinen, Navarro: **Implicit compression boosting with applications to self-indexing**. SPIRE 2007.

Or we can just partition the BWT into **fixed-length blocks** and use any order-0 encoder.

Gog et al.: **Fixed Block Compression Boosting in FM-indexes: Theory and Practice**. Algorithmica, 2019.

From suffix array to BWT

Let T be a **text string** of length n over alphabet Σ = [0..|Σ|) such that T[n – 1] = $ = 0 and $ does not occur anywhere else.

The **suffix array** of T is an array SA[0..n) of pointers to the suffixes of T in **lexicographic order**.

The **BWT** of T is a permutation of the character occurrences BWT[0..n) that lists the character **preceding** each suffix:

- BWT[i] = T[SA[i] – 1] if SA[i] > 0; and
- BWT[i] = $ if SA[i] = 0.

BWT	SA	Suffix
A	7	$
C	6	A$
T	4	ACA$
G	1	ATTACA$
A	5	CA$
$	0	GATTACA$
T	3	TACA$
A	2	TTACA$

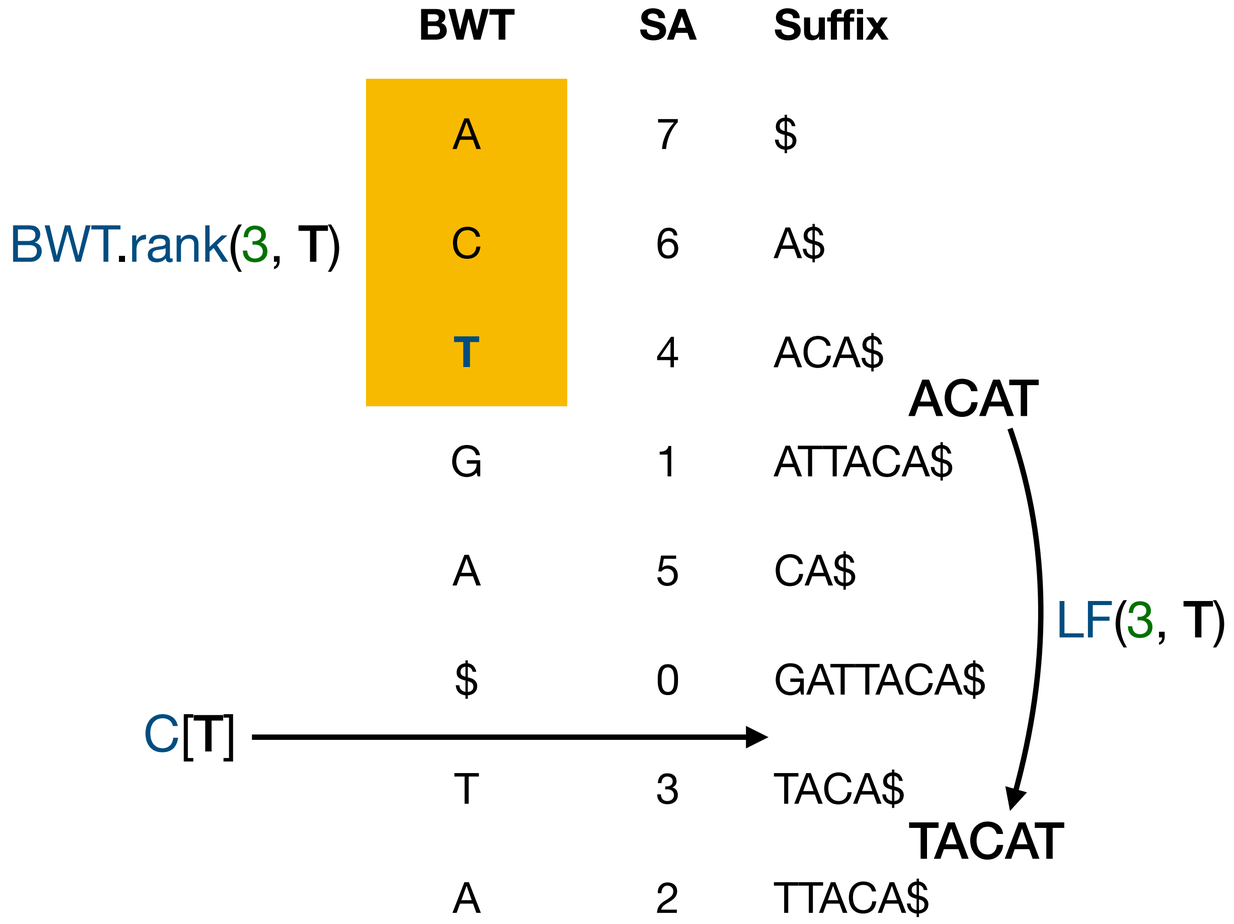
LF-mapping

The **lexicographic rank** of string X among the suffixes of text T is the number of suffixes Y such that Y < X in lexicographic order.

We define **LF-mapping** as a function such that if the lexicographic rank of string X is i, the lexicographic rank of string cX is LF(i, c).

We compute LF(i, c) = C[c] + BWT.rank(i, c):

- C[c] is the number of suffixes starting with a character c' < c; and
- BWT.rank(i, c) is the number of suffixes Y < X preceded by character c.



Inverting the BWT

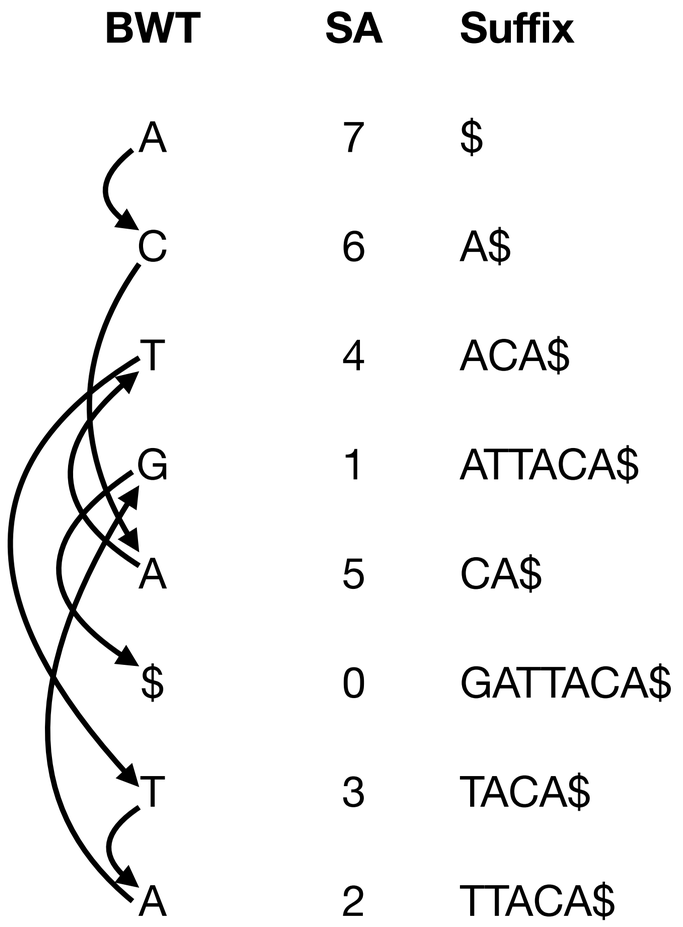
Because $ is the smallest character, we know that SA[0] = n – 1 and BWT[0] is the character preceding the **endmarker**.

We use LF(i) = LF(i, BWT[i]) for finding the **previous suffix**.

If BWT[i] ≠ $, it is the **previous character** in the text, and we continue iterating.

This way, we recover the text from the BWT **backwards**.

Jumping around in the BWT causes **cache misses**.



Multi-string BWT

Let T_0, ..., T_{m-1} be an **ordered collection** of m texts.

To make each suffix **unique**, we assume that the endmarker of T_i is smaller than that of T_j, for all $i < j$.

The BWT generalizes to this model easily, except that we cannot use LF-mapping with character $.

$SA[x] = (i, j)$ refers to suffix $T_i[j..)$ and points to the endmarker of T_x for $x < m$.

If $SA[x]$ refers to a suffix of text T_i, we have $DA[x] = i$ in the **document array**.

BWT	DA	SA	Suffix
A	0	(0, 7)	$
A	1	(1, 5)	**$**
C	0	(0, 6)	A$
T	1	(1, 4)	**A$**
T	0	(0, 4)	ACA$
C	1	(1, 1)	**ATTA$**
G	0	(0, 1)	ATTACA$
A	0	(0, 5)	CA$
$	1	(1, 0)	**CATTA$**
$	0	(0, 0)	GATTACA$
T	1	(1, 3)	**TA$**
T	0	(0, 3)	TACA$
A	1	(1, 2)	**TTA$**
A	0	(0, 2)	TTACA$

FM-index

Backward searching

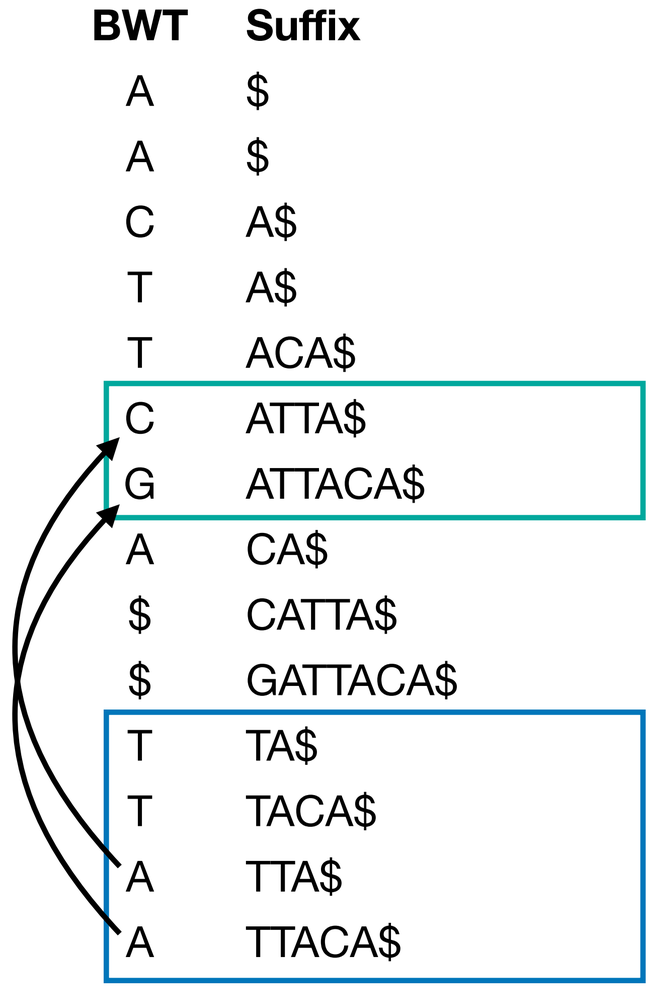
If SA[i..j) is the range of suffixes starting with string X, the range of suffixes starting with string cX is SA[LF(i, c)..LF(j, c)).

Given a **pattern** P, we can find the range of suffixes starting with it with **backward searching**:

- Start with [i..j) = [0..|SA|) matching an empty pattern.
- For k from |P| – 1 down to 0, update with [i..j) ← [LF(i, P[k])..LF(j, P[k])) to get the range matching pattern P[k..).

Range [5..7) = [LF(10, **A**)..LF(14, **A**)) matches pattern **AT**

Range [10..14) matches pattern **T**



FM-index

If we have the C array and the BWT with efficient rank queries, we can support the following:

- find(P) that returns the **lexicographic range** [i..j) starting with pattern P with O(|P|) rank queries.

- extract(i) that returns the text T_i with O($|T_i|$) rank queries.

This is the core functionality of the **FM-index**.

Ferragina, Manzini: **Indexing Compressed Text**. Journal of the ACM, 2005.

If we have **non-compressible** text over a **small alphabet** (such as DNA), we can simply partition the BWT into **fixed-length blocks** and store rank(i, c) at the start of each block for each character c.

Other common rank structures include:

- Bitvectors B_c that mark the positions where BWT[i] = c.

- **Wavelet trees** that reduce rank on the BWT to rank on log |Σ| bitvectors.

Locating the matches

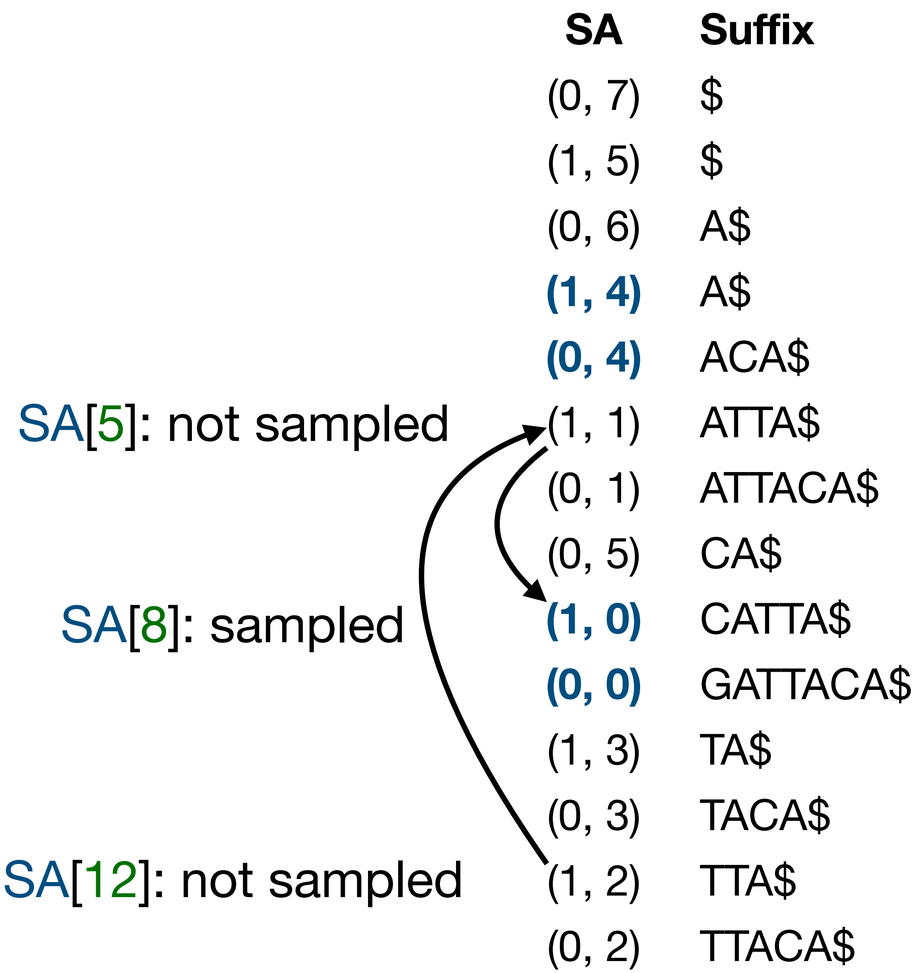
We **sample** some suffix array values in order to determine the **text positions** matching the pattern.

In **text order** sampling, we sample SA[i] if it is a multiple of d. Sampled positions are marked in a bitvector.

If SA[i] is not sampled, we **iterate** i ←LF(i) until we find a sampled position. If we need k iterations, the value we wanted is SA[i] + k.

Now we can support:

- locate(i) that returns SA[i] with O(d) rank queries and O(n / d) words of extra space.



Bidirectional FM-index

A **bidirectional FM-index** has an index F for the texts and an index R for the **reverse** texts.

For any **character** c, we have F.find(c) = R.find(c).

Because rev(cX) = rev(X) · c, range R.find(rev(cX)) is a **subrange** of R.find(rev(X)).

Because the occurrences of P in forward texts are occurrences of rev(P) in reverse texts, |R.find(rev(cX))| = |F.find(cX)|.

For any c' < c, we have find(Xc') < find(Xc).

Let o be the number of occurrences of characters c' < c in the BWT range F.find(X) and l = |F.find(cX)|. If R.find(rev(X)) = [i..j), we know that R.find(rev(cX)) = [i+o..i+o+l).

By extending the pattern **backward** in F, we also extend it **forward** in R, and the other way around.

Lam et al.: **High Throughput Short Read Alignment via Bi-directional BWT**. BIBM 2009.

Forward and backward

An **FMD-index** stores DNA sequences and their **reverse complements** in the same index and effectively matches **both orientations** of the pattern against both orientations of the texts.

It works in a similar way to bidirectional FM-indexes.

Li: **Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly**. Bioinformatics, 2012.

If we use the **forward** index F, we **sort suffixes** of the texts and match the pattern **backward**.

We can also use the **reverse** index R as an index of the original texts. Then we sort the **reverse prefixes** of the texts and match the pattern **forward**.

Sometimes using the reverse index is more natural.

FM-index in practice

Key features of FM-indexes:

- Reasonably fast find, slow locate.
- Very space-efficient.
- Cache misses for each character.
- Arbitrary pattern length.
- No need for word boundaries or other structure.

Short read alignment is the primary application of FM-indexes.

Space-efficiency is no longer that important if text size is only a few gigabytes.

Long reads (especially with high error rates) favor k-mer indexes with fast locate, because the alignment must be chained from many seed hits.

Information retrieval applications prefer using words or other tokens instead of characters, and they often have loose requirements for the relative order and spacing of query terms.

Run-length encoded BWT

Back to compression

Traditional BWT-based compressor:

1. BWT rearranges the symbols by context.
2. **Move-to-front** uses a list of symbols. Each symbol is encoded by its current position in the list, and the symbol is then moved to the front of the list.
3. **Run-length encoding** (RLE) replaces each run of symbols c^l with a pair (c, l).
4. Order-0 encoder such as Huffman encodes the pairs.

bzip2 adds some additional stages but uses the same basic idea.

Move-to-front is similar to implicit **compression boosting**, turning the global order-0 encoder into a local one.

Run-length encoding becomes useful with **highly repetitive data**.

BWT of identical texts

If there are R equal letter **runs** in the BWT of text T of length n, there are also R runs in the BWT of m copies of text T, for any $m > 0$.

If we run-length encode the BWT and use any reasonable encoding (such as a sparse bitvector) for run lengths, the **RLBWT** takes $R \log |\Sigma| + O(R \log (mn / R))$ bits.

Adding **duplicate** texts to the collection is almost **free**.

Entropy-based compression would require mnH bits, plus overhead.

BWT	Suffix
A	$
A	**$**
A	**$**
T	A$
T	**A$**
T	**A$**
C	ATTA$
C	**ATTA$**
C	**ATTA$**
$	CATTA$
$	**CATTA$**
$	**CATTA$**
T	TA$
T	**TA$**
T	**TA$**
A	TTA$
A	**TTA$**
A	**TTA$**

Edits in RLBWT

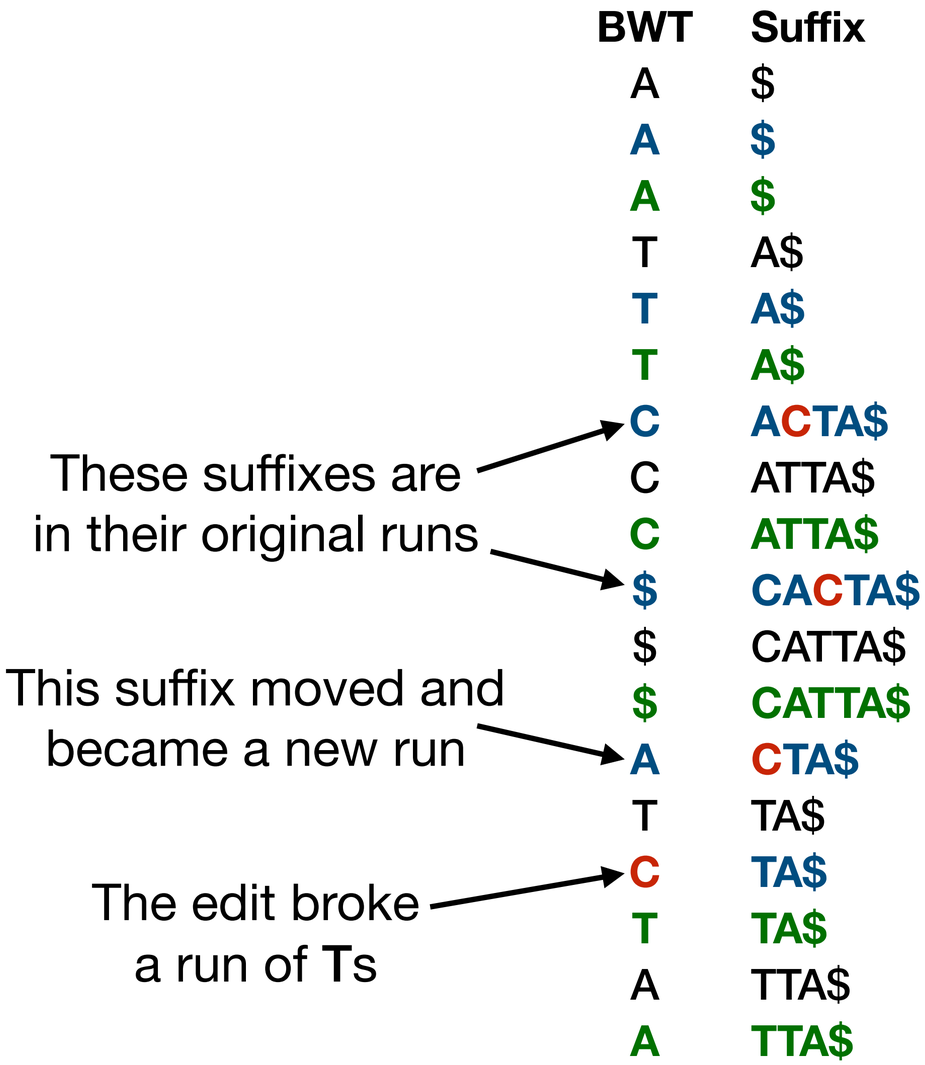
We changed a **single character** in the blue text.

The edit may **break** an existing run and **create** a new run where it appears in the BWT.

Some suffixes **preceding** the edit may **move** around and do the same.

Suffixes **far before** the edit remain in the **same runs** (but possibly in new positions).

In some models, an edit adds $O(\log_{|\Sigma|}(mn))$ runs in the **expected case**.



The "RLCSA model"

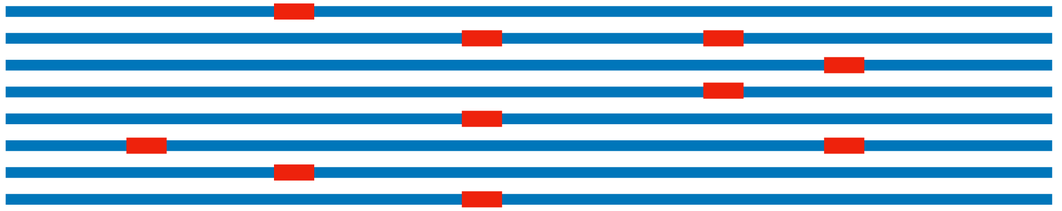
We start with m identical **aligned** texts.

Each edit **affects** some suffixes, but most remain unaffected.

If there are R runs in the BWT of the original text and s affected suffixes in total, there are at most R + O(s) runs in the BWT of the collection.

This implies a **graph** where unaffected columns can (usually) be merged into nodes.

Generalizations of the **BWT for graphs**, such as GCSA and Wheeler graphs, started from this model.



As the generalizations can grow **exponentially** in size, they were ultimately more useful in formal languages and automata than in bioinformatics.

Mäkinen et al.: **Storage and Retrieval of Highly Repetitive Sequence Collections**. Journal of Computational Biology, 2010.

FM-indexes based on RLBWT

With **highly repetitive** text collections, RLBWT can be **orders of magnitude** smaller than entropy-compressed BWT.

RLCSA and other early indexes showed that this also applies to FM-indexes, as long as the rank/select **overhead** scales proportionally to the **compressed size**.

While find queries were fast, locate queries still used SA samples, with the product of **query time** and **space overhead** scaling proportionally to mn rather than R.

The **r-index** finally solved the issue with a structure that can compute $SA[i + 1]$ from $SA[i]$ in $O(\log \log (mn))$ time and $O(R)$ words of space overhead.

(Veli and/or Christina should talk more about the r-index on Thursday.)

Gagie et al.: **Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space**. Journal of the ACM, 2020.

GBWT

Data model

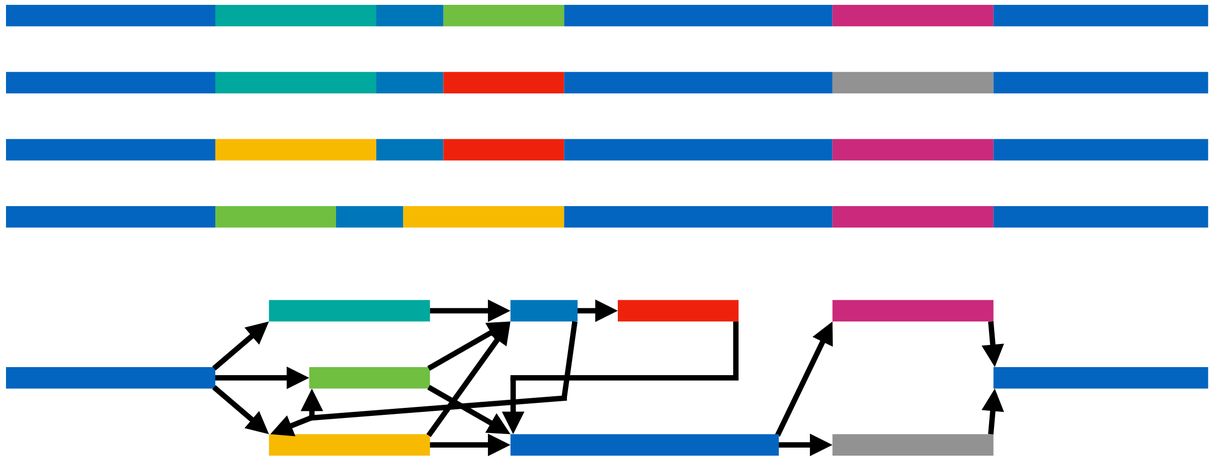
We have a representative set of **haplotypes** from the relevant population.

We **align** the haplotypes and build a **graph**, where each node represents aligned positions in the haplotypes.

Any **traversal** of the graph is a potential haplotype.

Traversals that are **locally consistent** with the original haplotypes are more likely to be **biologically plausible**.

For that reason, we store the original haplotypes as **paths**.



path ~ walk
path ~ stored traversal
traversal ~ emergent path

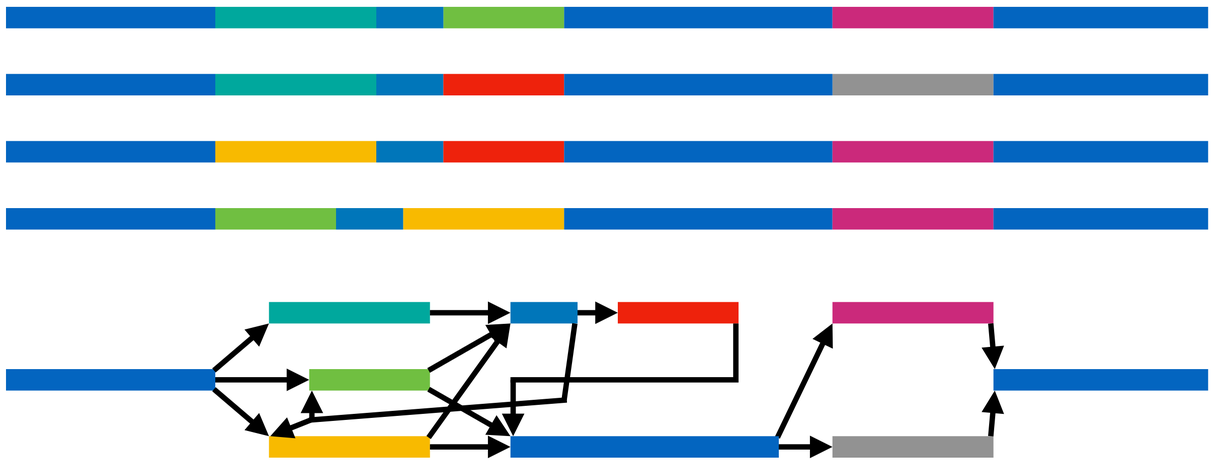
What do we need?

We want to store a collection of **haplotypes** as **paths** in a graph G = (V, E).

A path can be represented as a **sequence of nodes**, which can be interpreted as a string over alphabet V.

Because we expect the haplotypes to be highly similar, the collection is **highly repetitive**.

Therefore the data structure we choose is **RLBWT** tailored for strings over a **large alphabet** but where the **local alphabet** (adjacency list) is usually small.



path ~ walk
path ~ stored traversal
traversal ~ emergent path

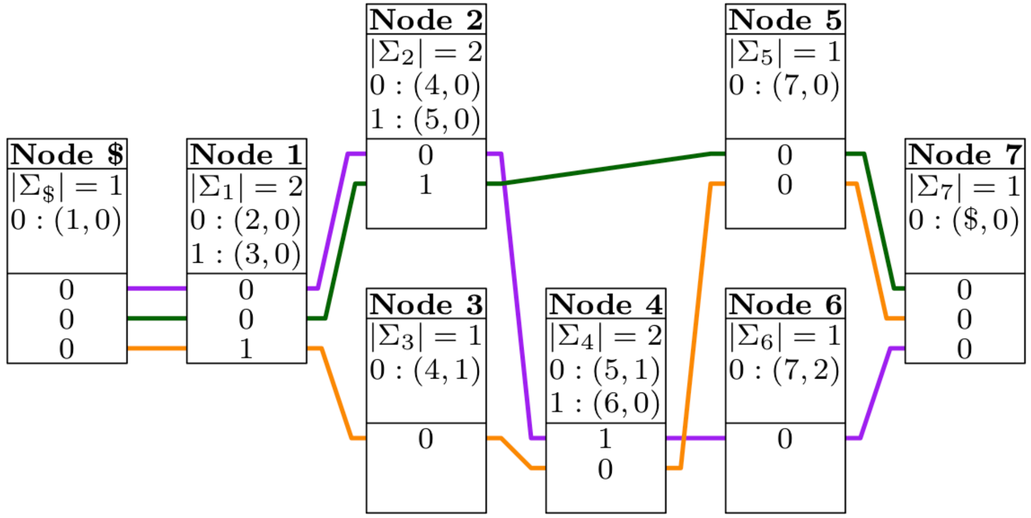
GBWT

The **GBWT** is a **reverse** RLBWT of paths in a **directed** graph.

We sort reverse prefixes of the paths and match patterns forward, following the **direction of the edges**.

To improve **memory locality**, we partition the BWT between the **nodes** and use the adjacency lists as rank structures.

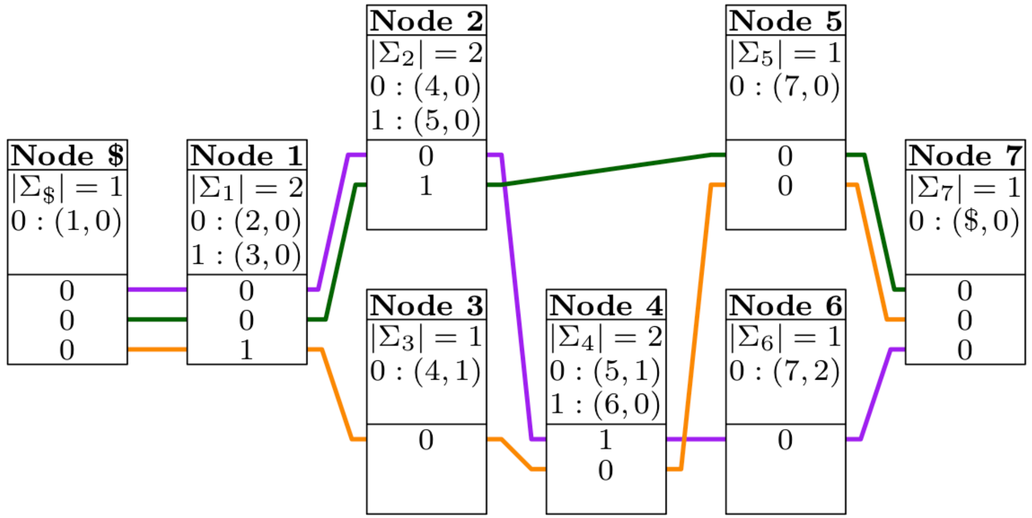
A find query determines how many indexed paths contain the corresponding **traversal** as a **subpath**.



Sirén et al.: **Haplotype-aware graph indexes**. Bioinformatics, 2020.

https://github.com/jltsiren/gbwt

BWT partitioning



Let BWT_v = BWT[C[v]..C[v + 1]).

That substring corresponds to prefixes where the **most significant** character in the sorting order (the last character) is v.

BWT_v tells where the path corresponding to each prefix **continues** after visiting node v.

BWT_4

Prefix	BWT
$	1
$	1
$	1
$ 1	2
$ 1	2
$ 1	3
$ 1 2	4
$ 1 2	5
$ 1 3	4
$ 1 2 4	6
$ 1 3 4	5
$ 1 2 5	7
$ 1 3 4 5	7
$ 1 2 4 6	7
$ 1 2 5 7	$
$ 1 3 4 5 7	$
$ 1 2 4 6 7	$

LF-mapping

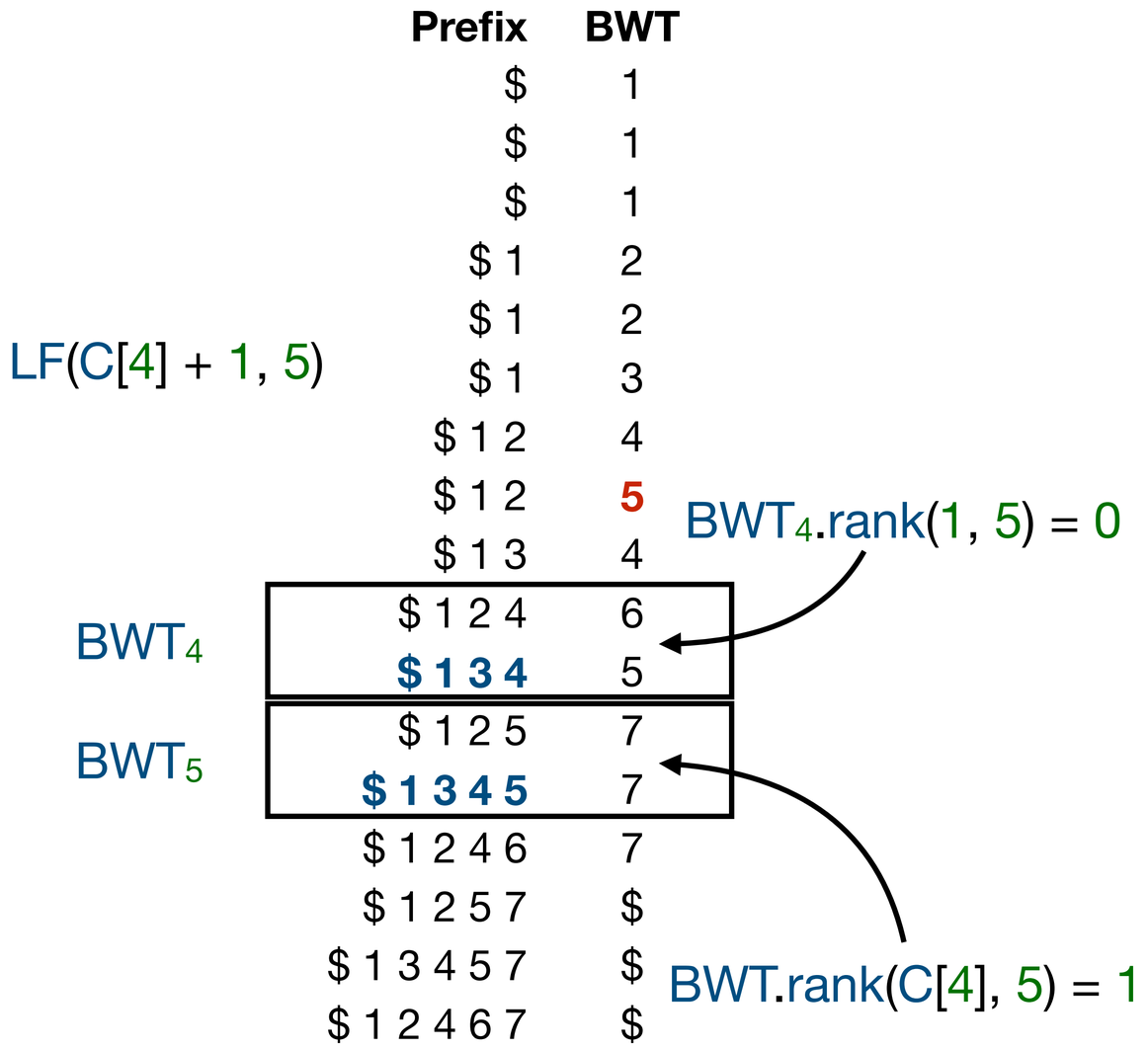
BWT offsets: (v, i) vs. C[v] + i vs. $BWT_v[i]$.

When we follow an edge (v, w), we use LF(C[v] + i, w) = C[w] + BWT.rank(C[v] + i, w).

C[w] is just a reference to node w.

We can partition BWT.rank(C[v] + i, w) into the sum of BWT.rank(C[v], w) and BWT_v.rank(i, w).

If we store BWT_v in node v and BWT.rank(C[v], w) in edge (v, w), we can compute LF-mapping using **local information** stored in the node.



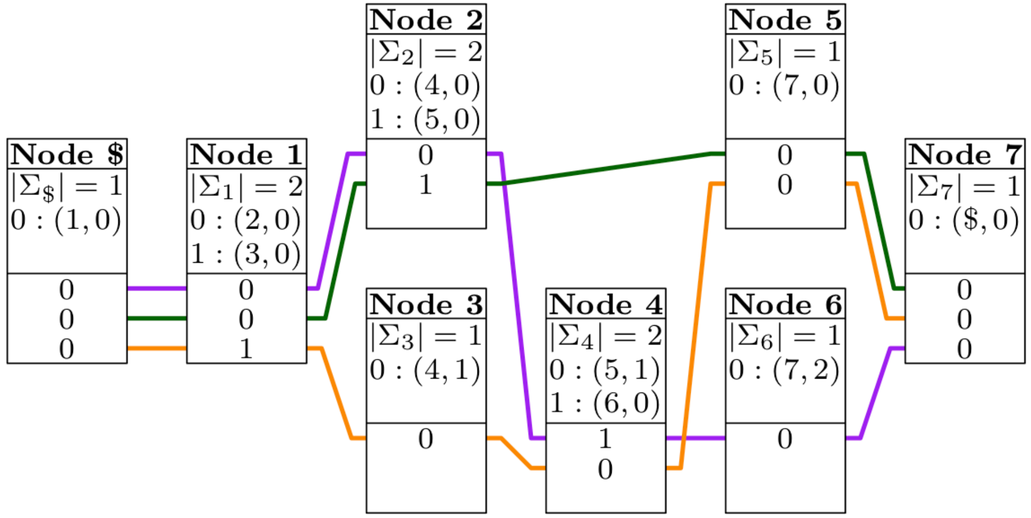
Node records

The **record** for node v contains a list of outgoing edges (v, w) and the BWT substring BWT_v.

For each edge (v, w), the **adjacency list** stores the destination node w as well as BWT.rank(C[v], w).

In BWT_v, nodes are replaced by their **ranks** in the adjacency list and and the substring is then **run-length encoded**.

The record is encoded as a **byte sequence**.



Node 1

- Outdegree 2 encoded as 2
- Edge to 2, offset 0 encoded as (2, 0)
- Edge to 3, offset 0 encoded as (1, 0)
- Run 0^2 encoded as 0 + 2 * (2 – 1) = 2
- Run 1^1 encoded as 1 + 2 * (1 – 1) = 1

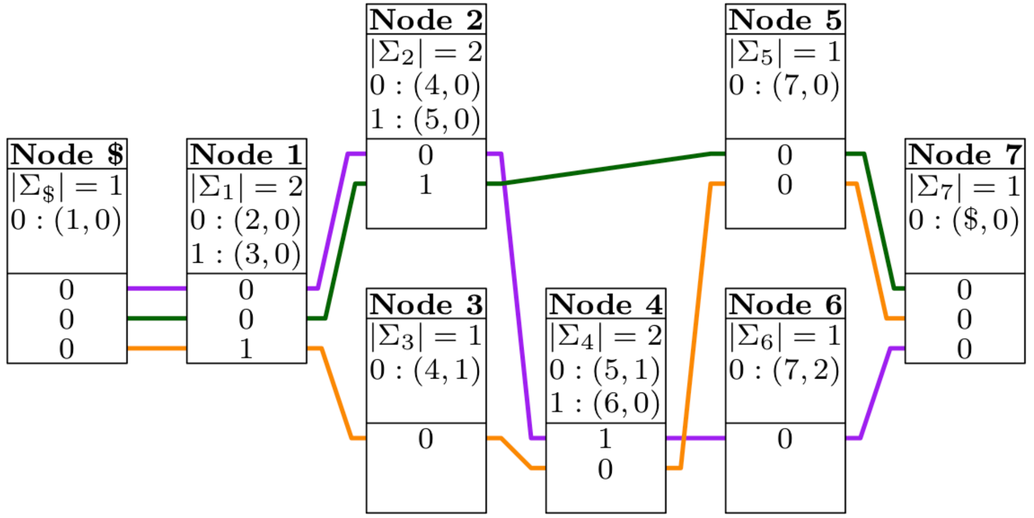
Using the GBWT

We **concatenate** the records and use a **sparse bitvector** B for finding the substring [B.select(v, 1)..B.select(v + 1, 1)) corresponding to node v.

When we compute LF-mapping from node v, we **decompress** the adjacency list and scan BWT_v sequentially.

This assumes that node degrees are not too high and paths do not visit the same nodes too many times.

Memory **locality** of iterated LF-mapping depends on the memory **layout** of the graph.




```
1102 2201021 2401001 1410 2511010 1701 1720 1002
1000 1000000 1000000 1000 1000000 1000 1000 1000
```

Encoding of the records and bitvector B (each byte is a single number).

More functionality

locate queries using **DA samples** for determining which haplotypes contain the given subpath.

r-index add-on as a larger and faster alternative for the same task.

Bidirectional GBWT implemented as a single index similar to the FMD-index.

Metadata mapping text identifiers to structured path names.

A **path name** consists of sample, contig, haplotype, and fragment identifiers.

Sample and contig identifiers map to **string names**.

Example path names

- (HG002, chr1, 1, 0)
- (HG002, chr1, 2, 0)
- (HG002, chr2, 1, 0)
- (HG002, chr2, 2, 0)
- (HG002, chr3, 1, 0)
- (HG002, chr3, 2, 0)
- (HG002, chr4, 1, 0)
- (HG002, chr4, 2, 0)

GBWT construction

Incremental BWT construction

Add an empty text

BWT	Suffix
A	$
$	$
C	A$
T	ACA$
G	ATTACA$
A	CA$
$	GATTACA$
T	TACA$
A	TTACA$

LF(· , **A**)

Prepend **A**

BWT	Suffix
A	$
A	**$**
C	A$
$	**A$**
T	ACA$
G	ATTACA$
A	CA$
$	GATTACA$
T	TACA$
A	TTACA$

LF(· , **T**)

Prepend **T**

BWT	Suffix
A	$
A	**$**
C	A$
T	**A$**
T	ACA$
G	ATTACA$
A	CA$
$	GATTACA$
$	**TA$**
T	TACA$
A	TTACA$

Hon et al.: **A space and time efficient algorithm for constructing compressed suffix arrays**. Algorithmica, 2007.

Batch insertion

The **BCR algorithm** builds the BWT for a collection of short reads incrementally.

It starts from the BWT of m empty texts and **extends each text** backward by a single character in each step.

Bauer et al.: **Lightweight algorithms for constructing and inverting the BWT of string collections**. Theoretical Computer Science, 2013.

RopeBWT2 inserts a **batch of texts** into an existing BWT using the same algorithm.

Li: **Fast construction of FM-index for long sequence reads**. Bioinformatics, 2014.

This is also the main GBWT construction algorithm.

During construction, we use a naive **dynamic** representation for the GBWT, where each node has an std::vector of edges and std::vector of runs.

In each step, we **rebuild** the node records for all nodes we touch.

Disjoint subgraphs

Paths are strings over the set of nodes V.

If we have two collections of paths in **disjoint subgraphs**, the strings in the collections are over disjoint alphabets.

We can build GBWTs for the collections **independently** and then **merge** them by simply reusing the node records.

More generally, we can partition the graph into weakly connected **components** and **parallelize** GBWT construction over the components.

We can easily build the GBWT for the 1000 Genomes Project (1000GP) data consisting of **5000 human haplotypes**.

A few years ago, the construction took **17 hours** on a system with 16 physical / 32 logical CPU cores and 244 GiB of memory.

```
Total length:     2194349057386
Sequences:        240232
Alphabet size:    612023760
Effective:        612023759
Runs:             2767709379
DA samples:       2143033346
BWT:              8636.28 MB
DA samples:       8368.48 MB
Total:            17006.6 MB
```

BWT merging

BWT of one text

BWT	Suffix
A	$
C	A$
T	ACA$
G	ATTACA$
A	CA$
$	GATTACA$
T	TACA$
A	TTACA$

BWT of another text

BWT	Suffix
A	**$**
T	**A$**
C	**ATTA$**
$	**CATTA$**
T	**TA$**
A	**TTA$**

Interleaved BWTs

BWT	Suffix
A	$
A	**$**
C	A$
T	**A$**
T	ACA$
C	**ATTA$**
G	ATTACA$
A	CA$
$	**CATTA$**
$	GATTACA$
T	**TA$**
T	TACA$
A	**TTA$**
A	TTACA$

GBWT merging

In order to **merge** the BWTs texts S and T, we must find the **interleaving** of their suffixes in lexicographic order.

By iterating **LF-mapping** in the BWT of S, we can determine the **lexicographic rank** of each suffix of T among the suffixes of S.

This produces the lexicographic ranks in an arbitrary order. We get the interleaving by **sorting** the ranks.

Sirén: **Compressed Suffix Arrays for Massive Data**. SPIRE 2009.

We can make BWT merging **fast and space-efficient** with a careful use of multiple search threads, buffering, compression, temporary files, and multithreaded sorting.

Sirén: **Burrows-Wheeler transform for terabases**. DCC 2016.

This allows us to build GBWTs for datasets larger than 1000GP.

It is unclear if indexing such large haplotype collections is useful, as recent projects such as HPRC are focusing on **quality over quantity**.

Bidirected sequence graphs

Data model

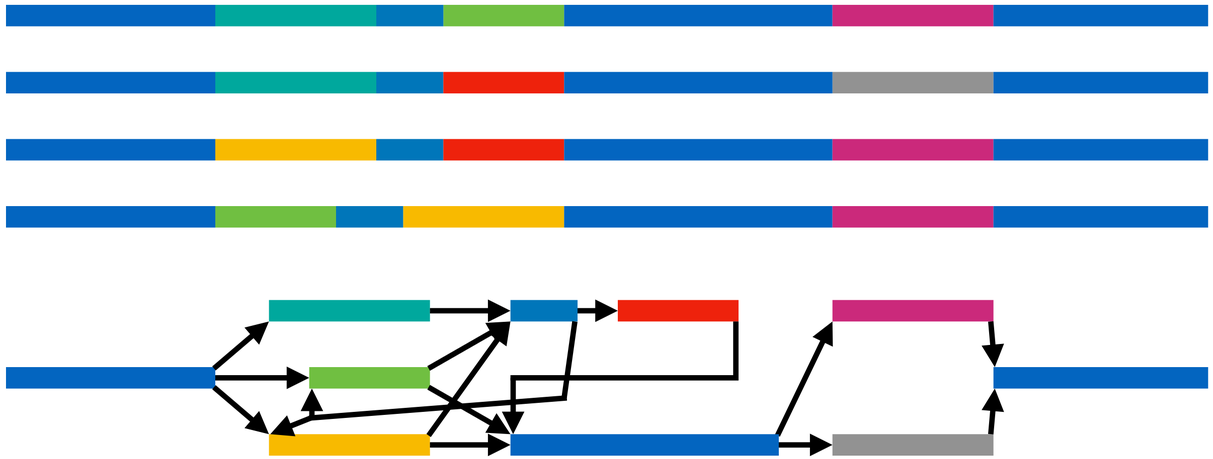
We have a representative set of **haplotypes** from the relevant population.

We **align** the haplotypes and build a **graph**, where each node represents aligned positions in the haplotypes.

Any **traversal** of the graph is a potential haplotype.

Traversals that are **locally consistent** with the original haplotypes are more likely to be **biologically plausible**.

For that reason, we store the original haplotypes as **paths**.



path ~ walk
path ~ stored traversal
traversal ~ emergent path

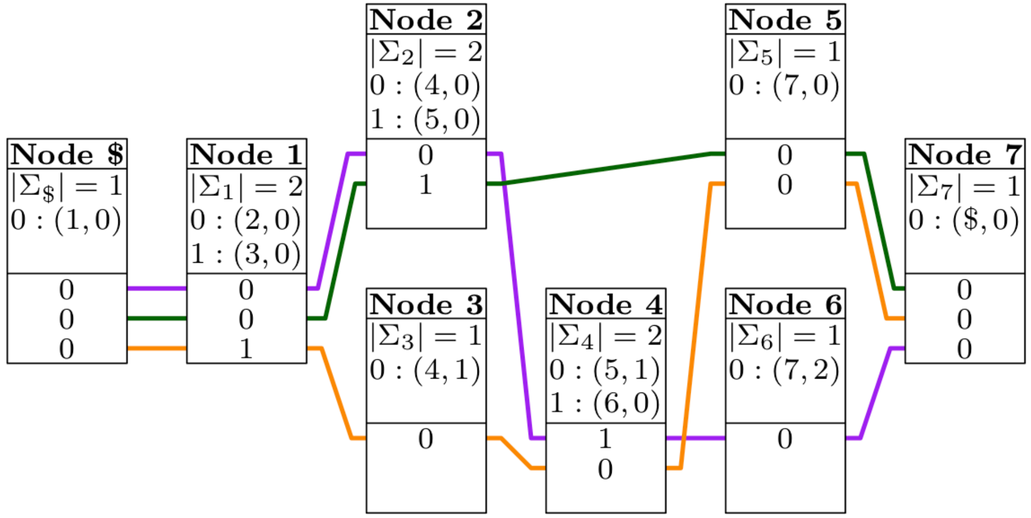
GBWT

The **GBWT** is a **reverse** RLBWT of paths in a **directed** graph.

We sort reverse prefixes of the paths and match patterns forward, following the **direction of the edges**.

To improve **memory locality**, we partition the BWT between the **nodes** and use the adjacency lists as rank structures.

A find query determines how many indexed paths contain the corresponding **traversal** as a **subpath**.



Sirén et al.: **Haplotype-aware graph indexes**. Bioinformatics, 2020.

https://github.com/jltsiren/gbwt

Three models

	GBWT	libhandlegraph / vg	GFA
Nodes	Simple	Bidirected	Bidirected
Node ids	Integers	Integers	Strings (but often integers)
Node labels	N/A	Any length but preferably short	Any length
Edges	Directed	Undirected	Undirected
Path ids	Integers (with optional metadata)	Strings or structured	Strings or structured
Path navigation	Forward only	Both directions	N/A

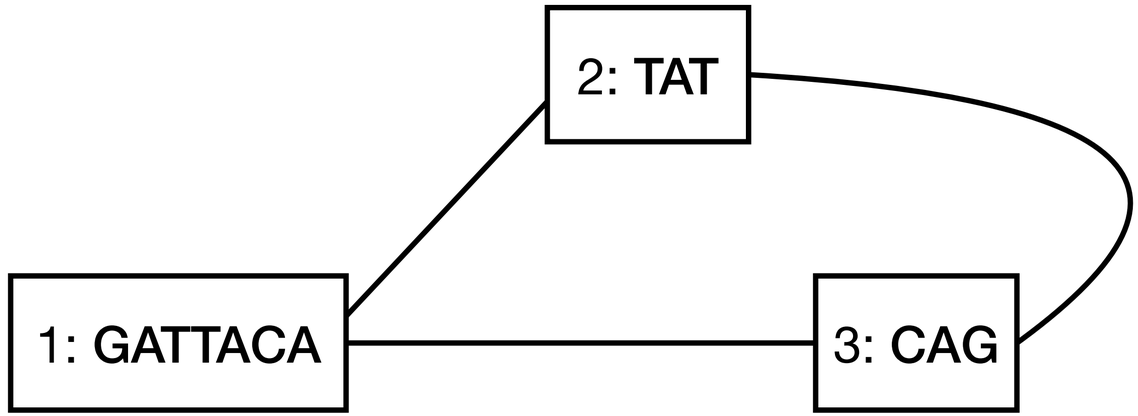
Bidirected sequence graphs

Each **node** has two **sides** and can be **visited** in two orientations.

A **forward** visit enters from the left, reads the **label**, and exits from the right.

A **reverse** visits enters from the right, reads the **reverse complement** of the label, and exits from the left.

Edges are **undirected** and connect two **node sides**.



Traversal >1 >2 <3 <1 reads **GATTACA**, **TAT**, **CTG**, and **TGTAATC**.

Traversal >1 >3 <2 <1 reads **GATTACA**, **CAG**, **ATA**, and **TGTAATC**.

libhandlegraph

Some graph implementations are mutable, others are immutable, and they make different time/space trade-offs.

A common **interface** reduces the need for rewriting code for each implementation.

In the **vg** ecosystem, that interface is provided by the **libhandlegraph** library.

Objects such as node visits, paths, and path steps have **handles** (opaque identifiers).

The interface uses functions that **iterate** over the relevant set of handles and call a **user-provided function** with each handle:

- for_each_handle(function, parallel)
- follow_edges(handle, backward, function)
- for_each_path_handle(function)
- for_each_step_in_path(path, function)
- for_each_step_on_handle(handle, function)

Eizenga et al.: **Efficient dynamic variation graphs**. Bioinformatics, 2020.

https://github.com/vgteam/libhandlegraph

GFA file format

GFA is a TSV-based **interchange format** for bidirected sequence graphs.

Originally intended for assembly graphs, recent extensions have made a subset of GFA suitable for **pangenome graphs**:

- **Segment**: name, sequence
- **Link**: from, orientation, to, orientation
- **Path**: name, node visits
- **Walk**: sample, haplotype, contig, interval, node visits

https://github.com/GFA-spec/GFA-spec/blob/master/GFA1.md

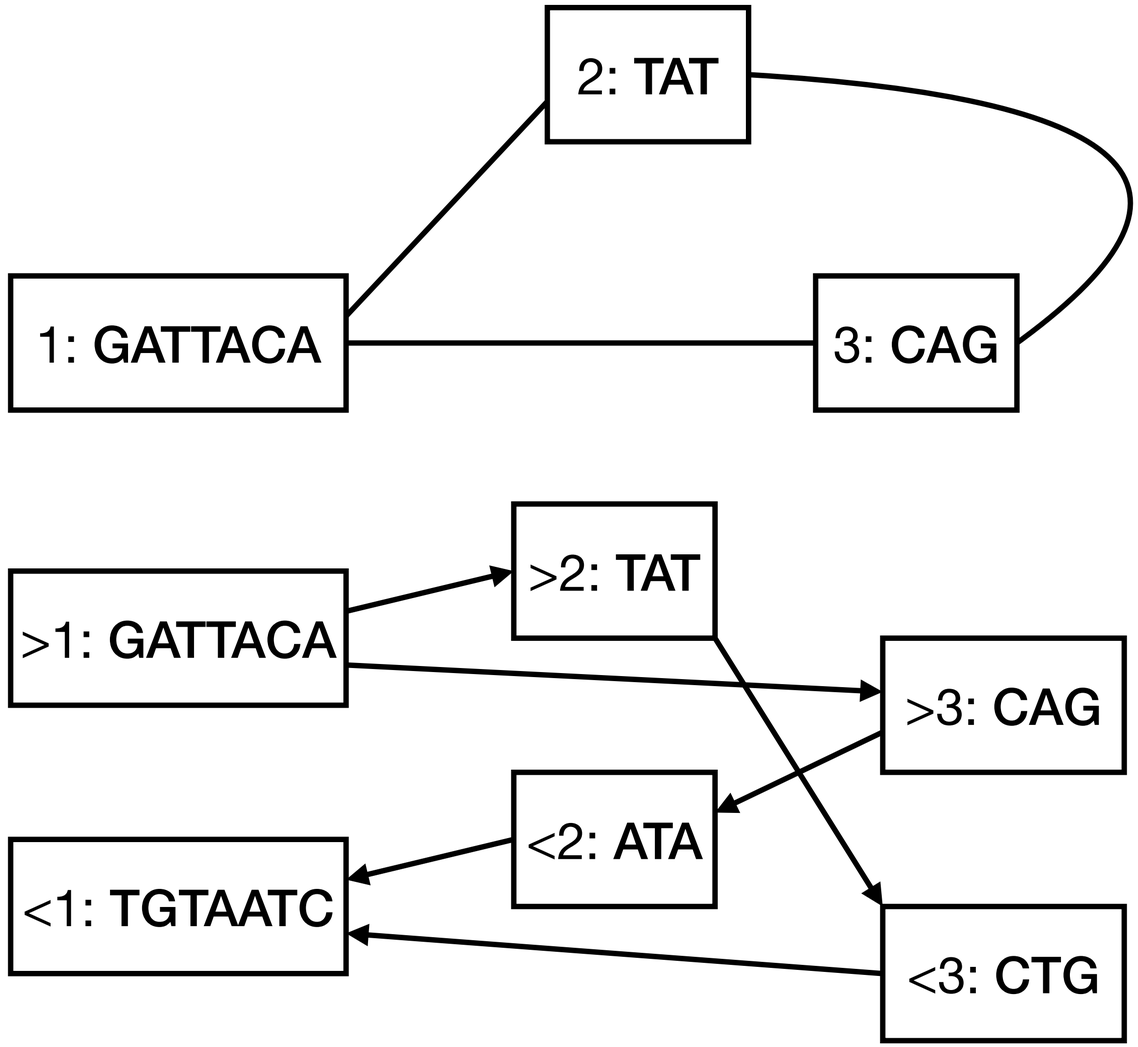
```
H       VN:Z:1.1
S       11      G
S       12      A
S       13      T
S       14      T
S       15      A
S       16      C
S       17      A
S       21      G
S       22      A
S       23      T
S       24      T
S       25      A
L       11      +       12      +       *
L       11      +       13      +       *
L       12      +       14      +       *
L       13      +       14      +       *
L       14      +       15      +       *
L       14      +       16      +       *
L       15      +       17      +       *
L       16      +       17      +       *
L       21      +       22      +       *
L       21      +       23      +       *
L       22      +       24      +       *
L       23      +       24      -       *
L       24      +       25      +       *
P       A       11+,12+,14+,15+,17+     *
P       B       21+,22+,24+,25+ *
W       sample  1       A       0       5       >11>12>14>15>17
W       sample  2       A       0       5       >11>13>14>16>17
W       sample  1       B       0       5       >21>22>24<23<21
W       sample  2       B       0       4       >21>22>24>25
```

Simulating bidirected graphs

We can **simulate** bidirected graphs with directed graphs by turning **node visits** into nodes.

Edges adjacent to the **right side** become outgoing edges from the **forward** node.

Edges adjacent to the **left side** become outgoing edges from the **reverse** node.



Path names

A GBWT **path name** is a **unique** combination of four numerical identifiers:

- **Samples** are top-level items and may correspond to string names.
- **Contigs** are non-overlapping parts of a sample and may correspond to string names.
- **Haplotypes** are overlapping sequences for the same (sample, contig).
- **Fragments** are non-overlapping parts of (sample, contig, haplotype).

External **string names** can be parsed or stored as (_gbwt_ref, name, 0, 0).

Sample	Contig	Haplotype	Fragment
_gbwt_ref	chr1	0	0
HG001	chr1	1	0
HG001	chr1	2	0
HG002	chr1	1	0
HG002	chr1	2	0
HG003	chr1	1	0
HG003	chr1	2	0
_gbwt_ref	chr2	0	0
HG001	chr2	1	0
HG001	chr2	2	0
HG002	chr2	1	0
HG002	chr2	2	0
HG003	chr2	1	0
HG003	chr2	2	0

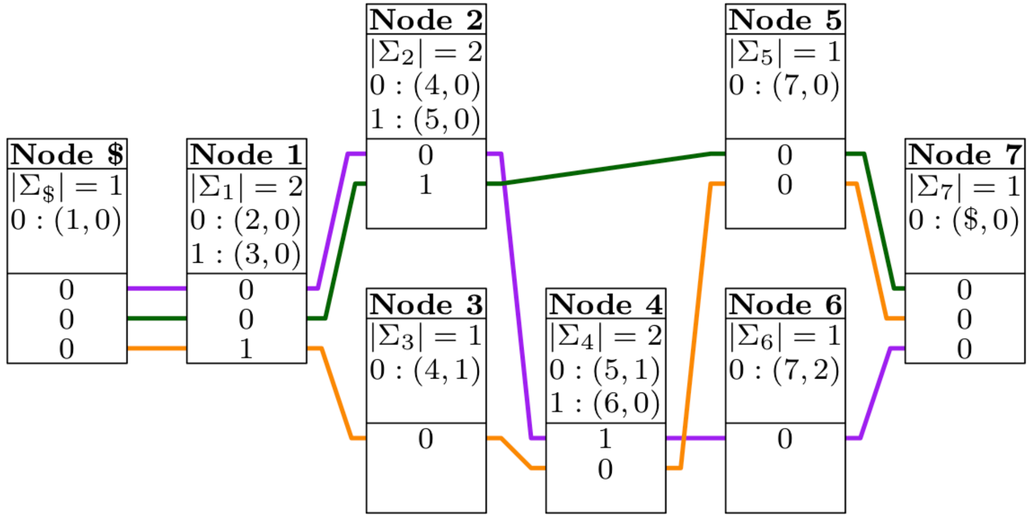
Path navigation

The GBWT does not support **inverse LF-mapping**, because the nodes do not know their **predecessors**.

In order to navigate **backward** on a path, we store each path in **both orientations** in a **bidirectional index** similar to the FMD-index.

We then find the predecessors by **flipping** node orientation, finding the successors, and flipping the results.

The **reverse path** visits the same nodes in reverse **order** and the other **orientation**.



reverse(>1 >2 > 4 >6 >7) = <7 <6 <4 <2 <1

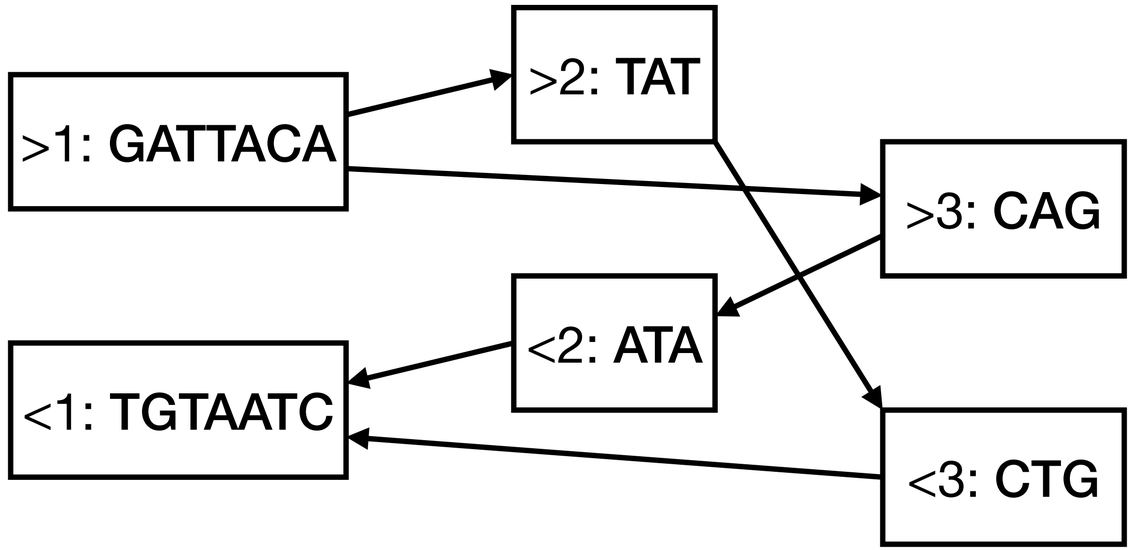
Node labels

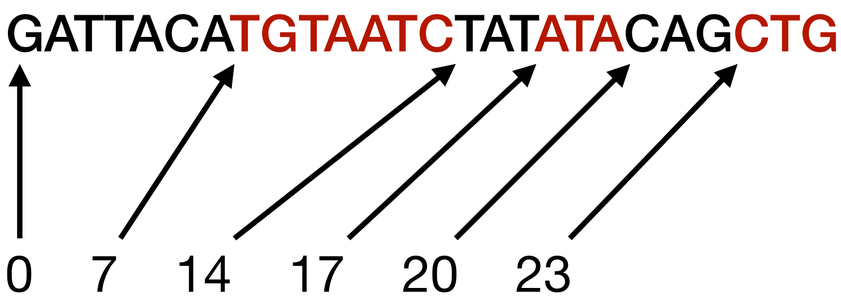
Because the GBWT does not store **node labels**, we need a separate structure for them.

As the graph is **immutable**, we can reduce memory and I/O **overhead** by **concatenating** the labels.

The **starting positions** can be an integer array or a sparse bitvector.

Storing vs. deriving the reverse complements is a relevant time/space trade-off.





Node-to-segment translation

GFA **segments** can be long in regions with no variation.

Long node labels are inconvenient in many applications:

- **Indexes** may want to store graph positions (node id, orientation, offset) in 64 bits.
- Long labels do not work in some graph **visualizations**.
- Interfaces may create **temporary copies** of node labels.

The usual solution is **chopping** long segments into at most 1024 bp nodes, so that each segment corresponds to an **interval** of node identifiers.

If **segment names** cannot be interpreted as integer identifiers, we also need a **translation** between the names and ranks of node id intervals.

GBWTGraph

GBWTGraph

We **simulate** a bidirected sequence graph using a directed graph and store the paths in a **bidirectional** GBWT index.

The GBWT represents the **topology** of the subgraph **induced** by the paths. Nodes and edges exist only if they are used on a path.

We store the **node labels** in a **string array** (concatenated strings + array of starting positions).

Another string array stores a mapping between nodes and **GFA segments**.

Paths corresponding to sample _gbwt_ref are exposed as named **libhandlegraph paths**.

Samples can be designated as **reference samples**, making the corresponding paths reference sequences.

Sirén et al.: **Pangenomics enables genotyping of known structural variants in 5202 diverse genomes**. Science, 2021.

https://github.com/jltsiren/gbwtgraph

Path operations

GBWT **search states** are pairs (v, [i..j)) representing intervals $BWT_v[i..j)$.

States are typically search results for a **pattern**; j – i is then the number of indexed paths containing the pattern as a **subpath**.

Bidirectional states contain search states for the pattern and its reverse. They can be used for extending the pattern in **both directions**.

The key operation follow_paths finds all non-empty single-node **extensions** of a search state.

```
// Get the most promising alignment from the
// priority queue.
GaplessExtension curr = extensions.top();
extensions.pop();

// Extend the alignment over all successor nodes.
graph.follow_paths(curr.state, false,
    [&](BidirectionalState next_state) {
    handle_t handle =
        node_to_handle(next_state.forward.node);
    GaplessExtension next { ... };
    size_t offset = match_forward(next, sequence,
        graph.get_sequence_view(handle),
        mismatch_limit);
    if (offset == 0) { return; }
    next.path = append(curr.path, handle);
    if (next.range.second >= sequence.length()) {
        next.reached_end();
    } else if (offset < graph.get_length(handle)) {
        next.set_right_maximal();
    }
    next.set_score();
    extensions.push(next);
});
```

Node record caching

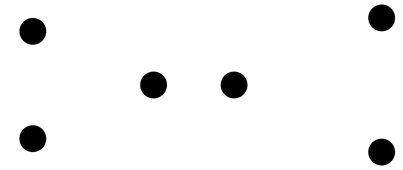
Accessing the outgoing edges or the BWT requires **finding** the GBWT node record and **decompressing** the edges.

This takes **hundreds of nanoseconds** (~1000 CPU cycles) with large GBWTs.

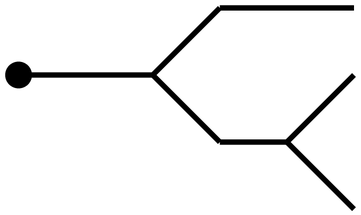
Some algorithms (such as aligning a read to a specific graph region) repeatedly access the nodes in a **small subgraph**.

We can speed such algorithms up significantly by **caching** the partially decompressed records.

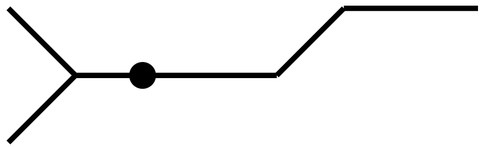
Cluster of seeds



Forward extensions of a seed



Backward extensions of an extension



Giraffe aligner

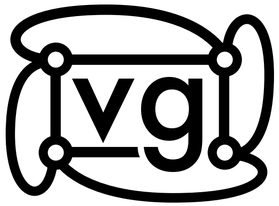
Giraffe aligner

Giraffe is a pangenomic short read aligner that combines the **speed** of linear aligners with the **accuracy** of vg map.

It maps reads to a **graph** but restricts its attention to paths that are locally consistent with the set of **reference haplotypes**.

The speed comes from ignoring **unlikely recombinations** and assuming that most **sequencing errors** are substitutions and most **real indels** are already in the reference.

Giraffe tries aligning the read **without gaps** before resorting to dynamic programming.



https://github.com/vgteam/vg

Sirén et al.: **Pangenomics enables genotyping of known structural variants in 5202 diverse genomes**. Science, 2021.

I gave a talk on Giraffe in Pangenomics Bio Hacking 2021. The recording and the slides can be found online.

https://pgbh2021.pangenome.eu/

Read alignment

Mapping: Approximate location of the read in the reference.

Alignment: The best base-to-base alignment between the read and the reference near a particular mapping.

Mapping quality: Estimated likelihood that the mapping is correct.

Fast and accurate read alignment is the easy part. The **hard part** is estimating mapping quality accurately without sacrificing speed.

Seed and extend approach:

1. Find **seeds** (partial alignments) using a text index.

2. **Cluster** the seeds that correspond to the same mapping (or **chain** them into rough alignments with long reads).

3. **Extend** the seeds into full alignments.

We continue **exploring** promising mappings until we are **confident** that we have found the best alignment.

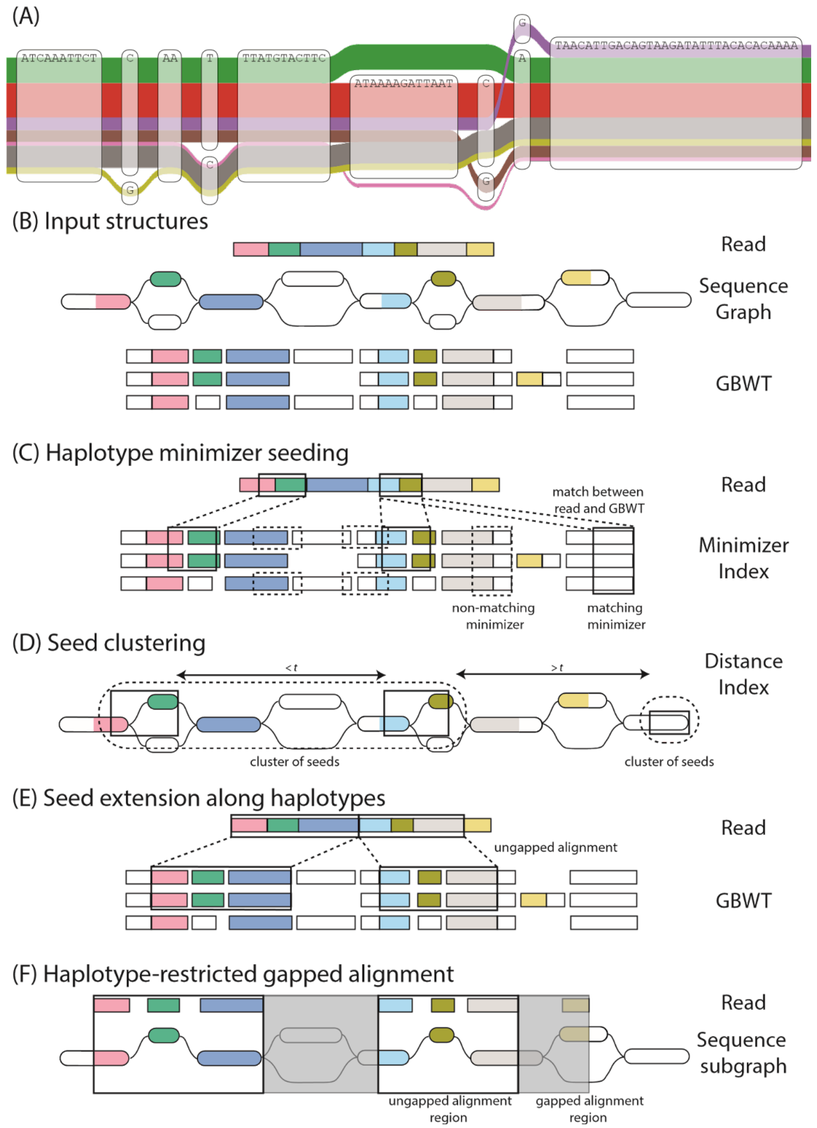
Giraffe algorithm

1. Find **seeds** using a **minimizer index** of the haplotypes.

2. **Cluster** the seeds using a **distance index** based on a hierarchical decomposition of the graph.

3. **Extend** the seeds over the **haplotypes**, allowing for a limited number of **mismatches**.

4. If we did not get enough full-length alignments, align the tails of best partial extensions using **dynamic programming** over the haplotypes.



Minimizer seeds

The cost of finding seeds for read P using an **FM-index** is O(|P|) **cache misses**, and the cost of listing occ seed hits is O(d · occ) cache misses.

Building the FM-index for the graph or the haplotypes is **expensive**.

We chose to use a **minimizer index** in Giraffe.

A (w, k)-minimizer is the k-mer with the **smallest hash value** among all k-mers and their reverse complements in a k + w – 1 bp **window**.

By using a **hash table**, we find minimizer seeds in O(|P| / w) cache misses in the expected case and list the hits **sequentially**.

For short reads, we use w = 11 and k = 29.

The index for a **human graph** typically takes 25–30 GiB.

If the index is on a network drive, **rebuilding** it may be faster than loading it from disk.

Index construction

Minimizer index **construction** iterates over all nodes using multiple **threads**.

At each **node**, the algorithm finds all **traversals** that start from the node and extend at least k + w – 2 bp beyond it.

The algorithm finds all **minimizers** in the traversal and stores them in a **buffer**.

Once the buffer is **full**, the thread acquires the lock and **inserts** the minimizers into the hash table.

```
// Start from both orientations of the initial
// node.
std::stack<GBWTTraversal> windows;
windows.push(GBWTTraversal(node, false));
windows.push(GBWTTraversal(node, true));

// Extend the windows until they are long enough.
while (!windows.empty()) {
    auto window = windows.top(); windows.pop();
    if (window.length >= target_length) {
        report(window); continue;
    }

    // Find all one-node extensions of the window.
    bool found = false;
    graph.follow_paths(window.state, false,
        [&](SearchState next_state) {
        handle_t handle =
            node_to_handle(next_state.node);
        auto next = window;
        next.append(handle);
        windows.push(next);
        found = true;
    });

    // Report maximal windows anyway.
    if (!found && window.length >= min_length) {
        report(window);
    }
}
```

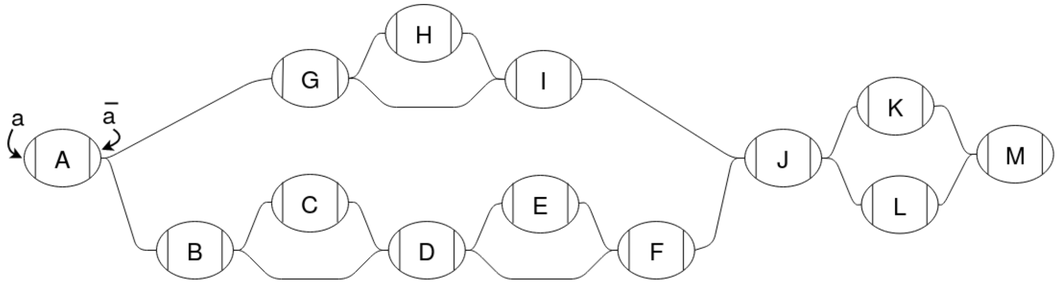
Seed clustering

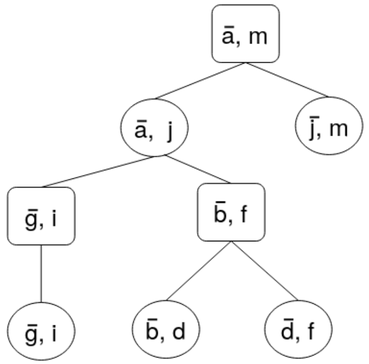
We use minimizer hits as **seeds** but avoid minimizers with too many hits.

Seeds close to each other form a **cluster** that likely corresponds to a single **alignment**.

Clustering uses a **distance index** that reduces computing distances in the graph to computing them in a tree.

Chang et al.: **Distance indexing and seed clustering in sequence graphs**. Bioinformatics, 2020.





Seed extension

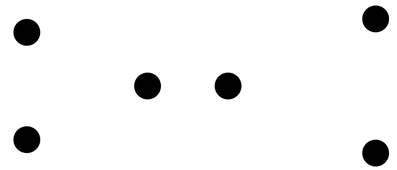
Giraffe **extends** most promising clusters into **gapless** alignments.

We merge **redundant seeds** that correspond to the same alignment between the read and a node.

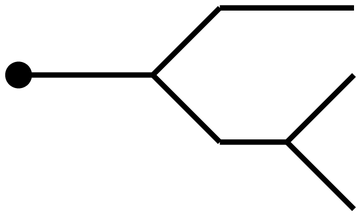
Each seed is extended over all **maximal paths** consisting of a left flank, the initial node, and the right flank.

There can be any number of **mismatches** in the initial node, up to 4 mismatches in total, and up to 2 mismatches in each flank regardless of the total.

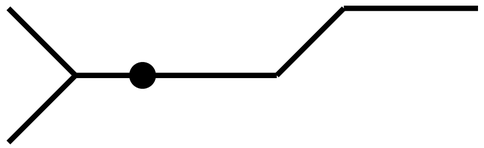
Cluster of seeds



Forward extensions of a seed



Backward extensions of an extension



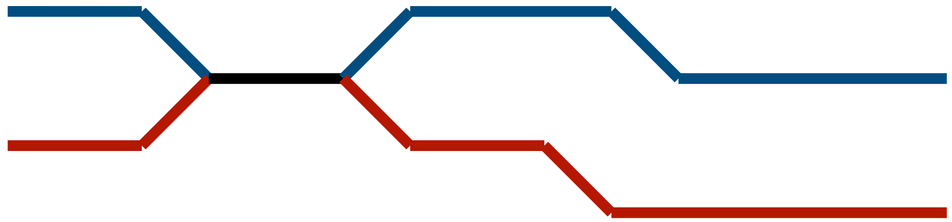
Alignment selection

The extensions are in a **priority queue** by **alignment score**: +1 per match, –4 per mismatch, and +5 for reaching the end.

For each seed, we find the **highest-scoring** maximal alignment.

If an alignment reaches **both ends** of the read, we return all full-length alignments that do not overlap too much.

Otherwise we **trim** the alignments to maximize the score and return all distinct alignments.



Non-overlapping alignments

+1 –4 +1 +1 +1 +1 +1 +1 +1 –4 +1 +1 +1 +1 +1 +1

M **X** M M M M M M M **X** M M M M M M

Alignment score 6

+1 –4 +1 +1 +1 +1 +1 +1 +1 –4 +1 +1 +1 +1 +1 +1

M X M M M M M M M **X** M M M M M M

Trimmed to alignment score 9

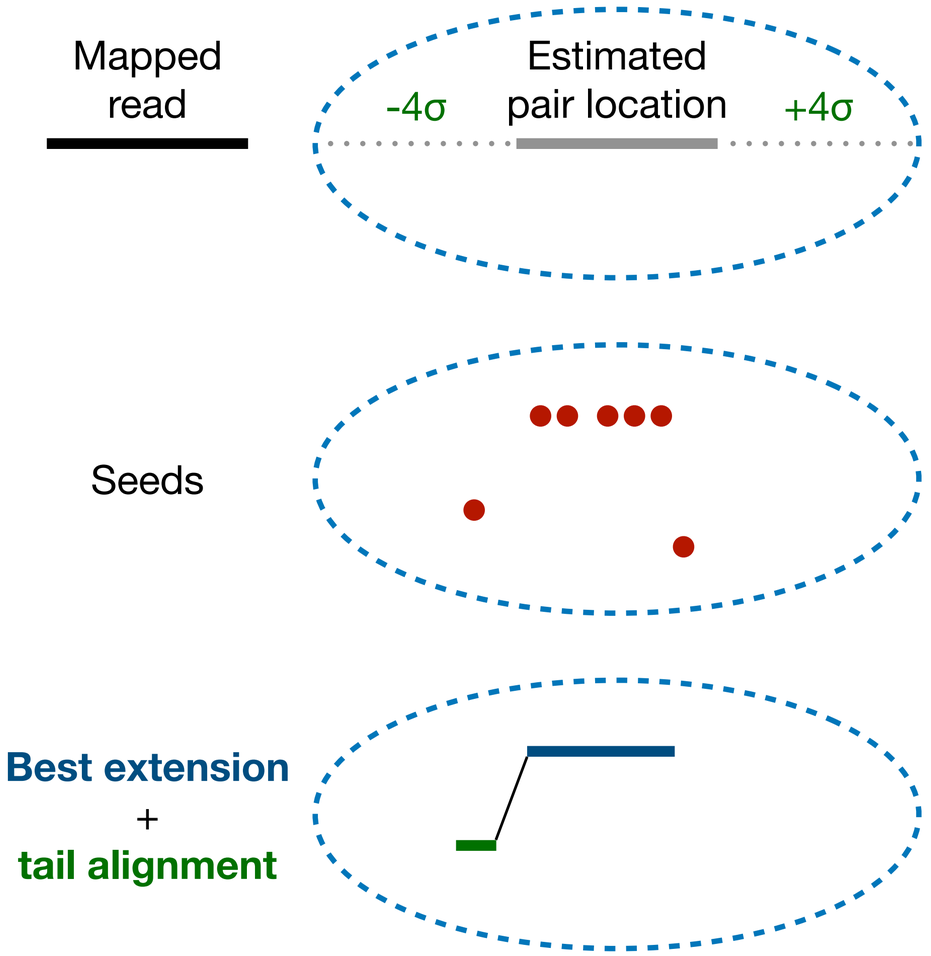
Final stages

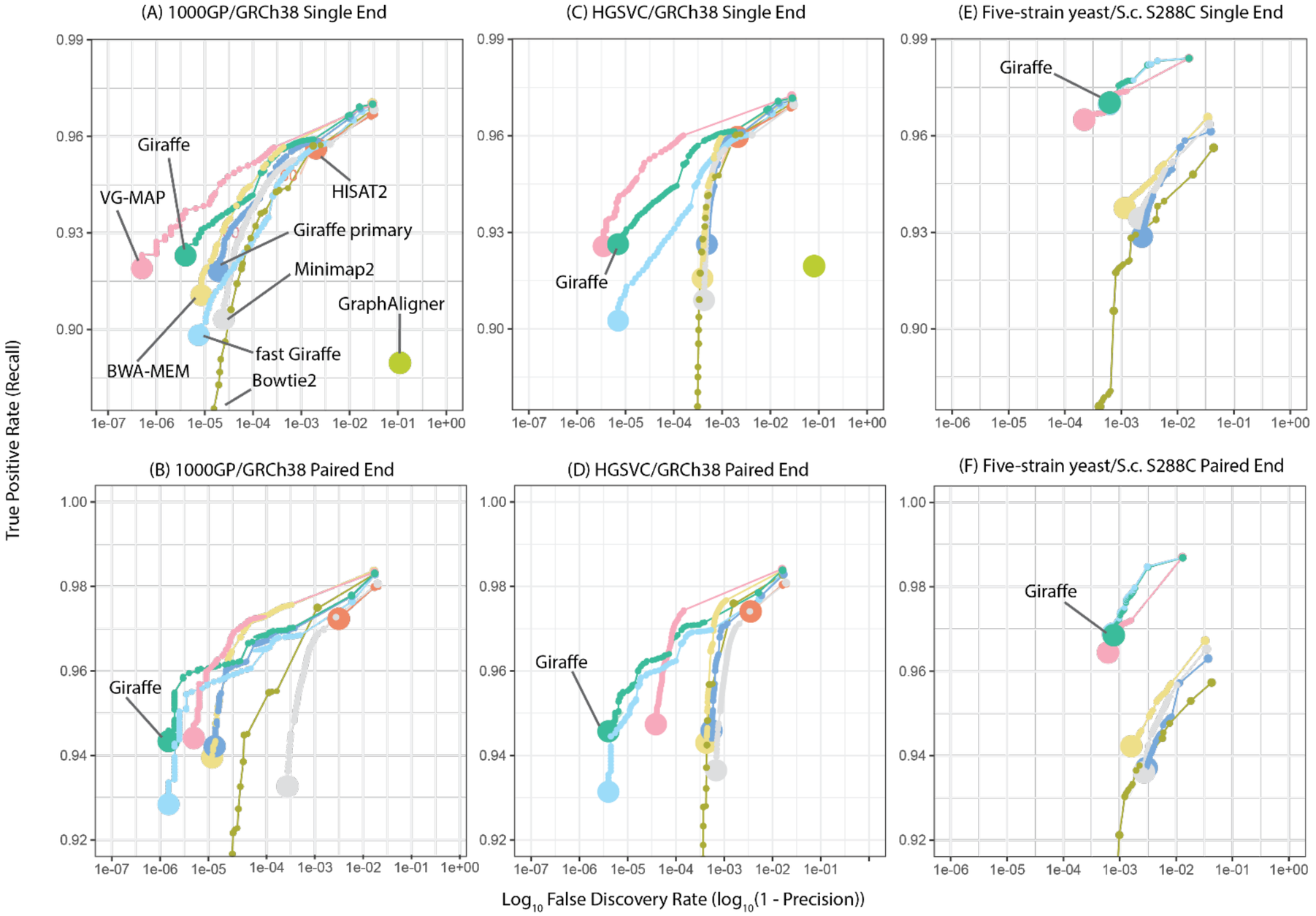
If we did not find enough full-length alignments without gaps, we use **dynamic programming** for aligning the **tails** of most promising gapless alignments.

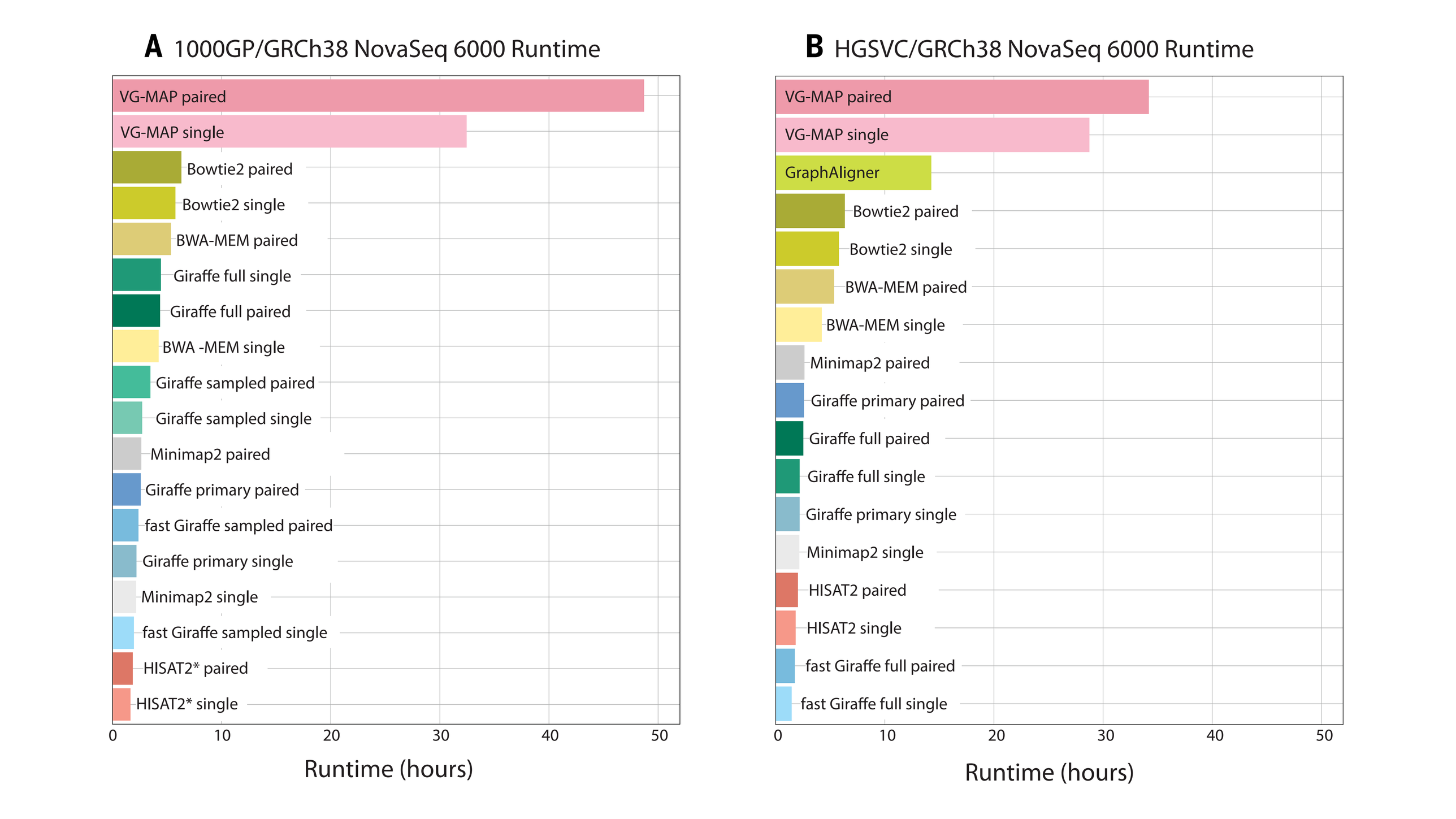
If we found a good alignment for a read but not for its pair, we may try **rescuing** the pair.

Rescue uses a **simplified version** of the Giraffe algorithm:

1. Find seeds in the relevant **subgraph**.
2. Extend them as a single cluster.
3. Align the tails of the best extension with gaps if necessary.




(A) 1000GP/GRCh38 Single End
(C) HGSVC/GRCh38 Single End
(E) Five-strain yeast/S.c. S288C Single End
(B) 1000GP/GRCh38 Paired End
(D) HGSVC/GRCh38 Paired End
(F) Five-strain yeast/S.c. S288C Paired End
True Positive Rate (Recall)
Log10 False Discovery Rate (log10(1 - Precision))
Giraffe
VG-MAP
HISAT2
Giraffe primary
Minimap2
GraphAligner
BWA-MEM
fast Giraffe
Bowtie2
250000
500000
750000


A 1000GP/GRCh38 NovaSeq 6000 Runtime
VG-MAP paired
VG-MAP single
Bowtie2 paired
Bowtie2 single
BWA-MEM paired
Giraffe full single
Giraffe full paired
BWA -MEM single
Giraffe sampled paired
Giraffe sampled single
Minimap2 paired
Giraffe primary paired
fast Giraffe sampled paired
Giraffe primary single
Minimap2 single
fast Giraffe sampled single
HISAT2* paired
HISAT2* single
0
10
20
30
40
50
Runtime (hours)
B HGSVC/GRCh38 NovaSeq 6000 Runtime
VG-MAP paired
VG-MAP single
GraphAligner
Bowtie2 paired
Bowtie2 single
BWA-MEM paired
BWA-MEM single
Minimap2 paired
Giraffe primary paired
Giraffe full paired
Giraffe full single
Giraffe primary single
Minimap2 single
HISAT2 paired
HISAT2 single
fast Giraffe full paired
fast Giraffe full single
0
10
20
30
40
50
Runtime (hours)

GBZ file format

GFA compression

GFA is the most common **interchange format** for pangenome graphs.

It does not **scale** well when the number of **haplotypes** increases.

While the haplotype paths are highly **similar**, they are too **long** for standard compressors to compress them together.

The **graph** itself is reasonably **small** for today's computers, but it also grows with the number of haplotypes, if we include **rare variants**.

The overall effect is **superlinear growth** with the number of haplotypes.

There is a need for a **compressed file format** for pangenome graphs with many haplotype paths.

The **GBWT** and the **GBWTGraph** already store the necessary information!

Goals and challenges

- **Stable** and **fully specified** file format.
- Good **compression**.
- Fast **loading** into in-memory data structures.
- Should not make too **specific requirements** for the in-memory data structures.
- Easy to handle as a **memory-mapped** file.

- Designing a **portable** file format based on **highly specialized** data structures?
- **Simple** enough for independent implementations vs. **compatibility** with existing files?
- **Different priorities** in the initial version and future versions?

File format basics

Element: Unsigned little-endian 64-bit integer.

File: Sequence of elements. Most objects are properly aligned in a memory-mapped file.

Limited number of **building blocks** to make implementation easier.

Serializable: Anything with size a multiple of 64 bits that can be serialized by copying the bits.

Vector: Length as an element, followed by concatenated items. Padded with 0-bits if necessary.

Optional structure: Size in elements as an element, followed by the structure. Can be passed through as a vector of elements. For implementation-dependent or application-dependent structures.

Simple-SDS
https://github.com/jltsiren/simple-sds

vgteam fork of SDSL
https://github.com/vgteam/sdsl-lite

Building blocks

Bitvector: Plain bitvector with optional rank/select structures.

Integer vector: Bit-packed integer array.

Sparse bitvector: Elias–Fano encoded bitvector with a bitvector as high and an integer vector as low.

String array: Concatenated alphabet-compacted ({ **A**, **C**, **G**, **N**, **T** } → [0..5)) strings as an integer vector and starting positions as a sparse bitvector. Usually decompressed as an in-memory structure.

Dictionary: Mapping between strings and their identifiers. Stored as a string array, with a permutation of the identifiers in lexicographic order as an integer vector. Usually decompressed in memory.

Tags: Key–value structure with case-insensitive keys. Stored as a string array. Key source identifies the library that wrote the file. The reader can use that information for determining if it can understand the optional structures.

GBZ file format

Full implementation in **C++**, partial implementation in **Rust**.

https://github.com/jltsiren/gbwt
https://github.com/jltsiren/gbwtgraph
https://github.com/jltsiren/gbwt-rs

The manuscript describing the file format is not available yet, as the benchmarks use HPRC graphs and the HPRC papers have not been submitted yet.

GBZ

Header: 16 bytes
Tags

GBWT

Header: 48 bytes
Tags
BWT: sparse bitvector, byte vector
DA samples: optional, unspecified

Optional metadata

Header: 40 bytes
Path names: vector of 16-byte items
Sample names: dictionary
Contig names: dictionary

GBWTGraph

Header: 24 bytes
Sequences: string array
Translation: string array, sparse bitvector

Compression algorithm

The input file is **memory-mapped** and the algorithm assumes that the order of the lines is reasonable.

1. Record the **starting position** and type of each line, determine if a translation is necessary, and determine GBWT construction buffer size.

2. Process **segments** and build the **translation** if necessary.

3. Process **links**, create a temporary graph, find weakly connected **components**, and determine GBWT construction **jobs**.

4. Process path and walk headers, build GBWT **metadata**.

5. Process **paths** and **walks**, running multiple GBWT construction jobs in **parallel**.

6. **Merge** partial GBWTs and build GBWTGraph.

.gfa: 44.9 GiB
.gz: 11.1 GiB
.gbz 3.11 GiB

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

Desktop: iMac 2020 with 128 GiB memory, 10/20 CPU cores.

Laptop: MacBook Air 2020 with 16 GiB memory, 4 + 4 CPU cores.

Intel Server: AWS i3.8xlarge with 244 GiB memory, 16/32 CPU cores.

ARM Server: AWS r6gd.8xlarge with 256 GiB memory, 32 CPU cores.

C++ implementation stores node labels in both orientations and uses more memory for faster decompression.

Rust implementation stores only forward labels and uses the query interface directly.

Memory usage is peak **resident set size**, which includes cached memory-mapped files but does not include pages swapped out to disk or to compressed memory.

Future ideas

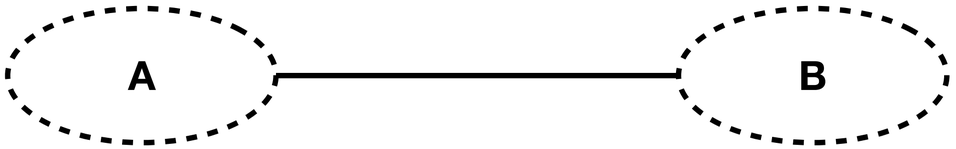
Typical distances

We often use the **shortest distance** in the graph as a proxy for the distance over the genome.

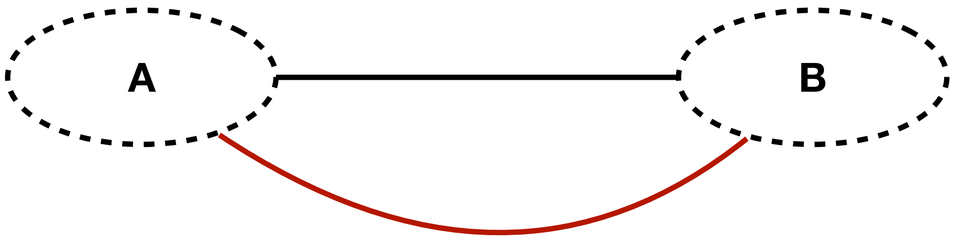
When structures such as long deletions are present, this may not reflect the **typical distance**.

How to **define** the typical distance, how to **compute** it efficiently, and how to **use** it?

Andrea Mariotti and Davide Piovani are working on this. Contact us if you have ideas!



Adding a single haplotype with a **long deletion** makes regions A and B close in the graph.



WFA over GBWT

Wavefront algorithm (WFA) is a sequence-to-sequence alignment algorithm generalizing the Myers' O(ND) algorithm to the **gap-affine** model (with mismatch, gap open, and gap extend penalties).

Myers: **An O(ND) Difference Algorithm and Its Variations**. Algorithmica, 1986.

Marco-Sola et al.: **Fast gap-affine pairwise alignment using the wavefront algorithm**. Bioinformatics, 2021.

Challenges using WFA over the haplotypes in a **GBWT** index:

- Is the end position even **reachable**?
- Is the **first visit** to the end position the right one or should we hope for a cycle?
- How to avoid **redundant work** with identical local haplotypes but branch when they diverge?
- Can we **meet in the middle** if we start from both directions?

Long read alignment

GraphAligner is the state of the art for aligning **long reads** to a general graph (not a DAG).

Rautiainen, Marschall: **GraphAligner: rapid and versatile sequence-to-graph alignment**. Genome Biology, 2020.

Error rates have recently gone down for both PacBio and ONT reads.

We also have **haplotype information** to take advantage of.

Rough idea for a new aligner:

1. Get **minimizer seeds**.
2. Try to use only **non-overlapping** seeds without too many hits.
3. **Chain** the seeds.
4. Connect the seeds using **WFA**.

Your ideas?