GBZ File Format for Pangenome Graphs

Jouni Sirén and Benedict Paten
UCSC Genomics Institute

Overview

This talk is about the **GBZ file format** for pangenome graphs.

GBZ is based on the **GBWT index**, which stores a set of paths as sequences of node identifiers.

GBWT is a run-length encoded **FM-index** partitioned between the nodes of the graph.

FM-index is a space-efficient text index based on the **Burrows–Wheeler transform**.

Sirén and Paten: **GBZ file format for pangenome graphs**. Bioinformatics, 2022.

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

Ferragina and Manzini: **Indexing Compressed Text**. JACM, 2005.

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

Pangenome graphs

Terminology (a slight abuse of)

DNA sequences are strings over alphabet { **A**, **C**, **G**, **T**, **N** }, where **N** indicates that we do not know the actual base (character).

A **genome** is a collection of DNA sequences, most of which are **chromosomes**.

Human genomes are **diploid**: there are two copies of (almost) every chromosome.

The set of chromosomes inherited from the same parent is called a **haplotype**.

Human haplotypes are ~3 Gbp long.

On the average, a human genome can be derived from parental genomes with just hundreds of **edit operations**.

Sequences are **homologous** if they have been derived from the same ancestral sequence.

Sequence **alignment** is an attempt to match homologous substrings of related sequences.

An alignment can be represented as a **graph**.

The graph should be simple enough to be practical while representing the true homology between the sequences.

Directed acyclic graphs

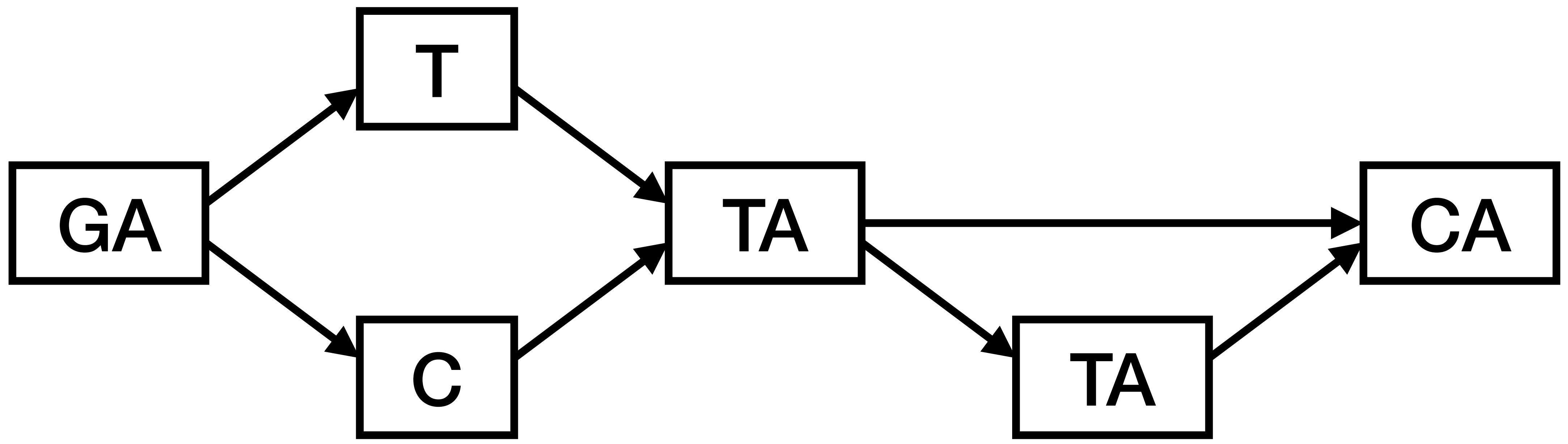
Directed acyclic graphs (DAG) are the simplest pangenome graph model.

They correspond to the edit distance model with **substitutions**, **insertions**, and **deletions**, but they also represent **recombinations** implicitly.

DAGs are easy to work with, but they cannot represent all biologically plausible alignments.

GATTACA
GACTATACA

```
GATTA--CA
GACTATACA
```



GATTATACA
GACTACA

Cycles and reversals

By allowing **cycles** in the graph, we can represent other edit operations, such as **rearrangements** and **repetitions**.

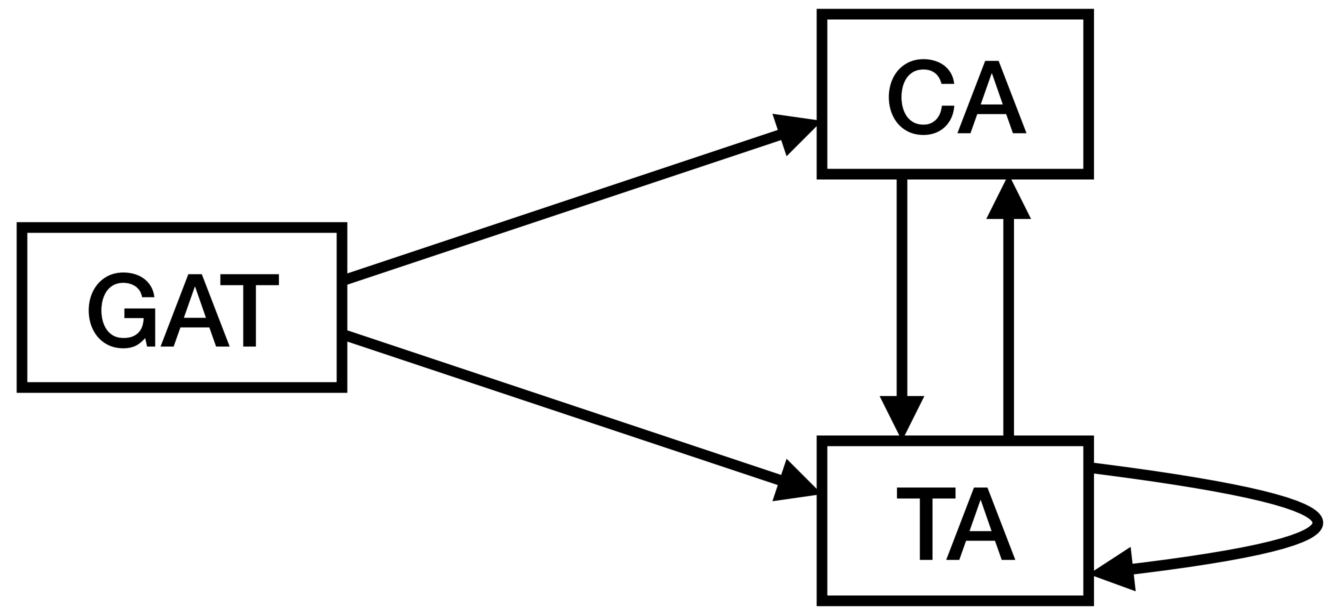
Cycles also allow completely implausible paths, unless we limit arbitrary iterations.

We can do that by storing the **aligned sequences** as paths and using them to guide us.

Cyclic graphs are more difficult to work with and reason about than DAGs.

One key operation is still missing: **reverse complement**.

GATCATACA
GATTATATACA



GATCATACATACATATATACATACATA

GATTACA → TGTAATC

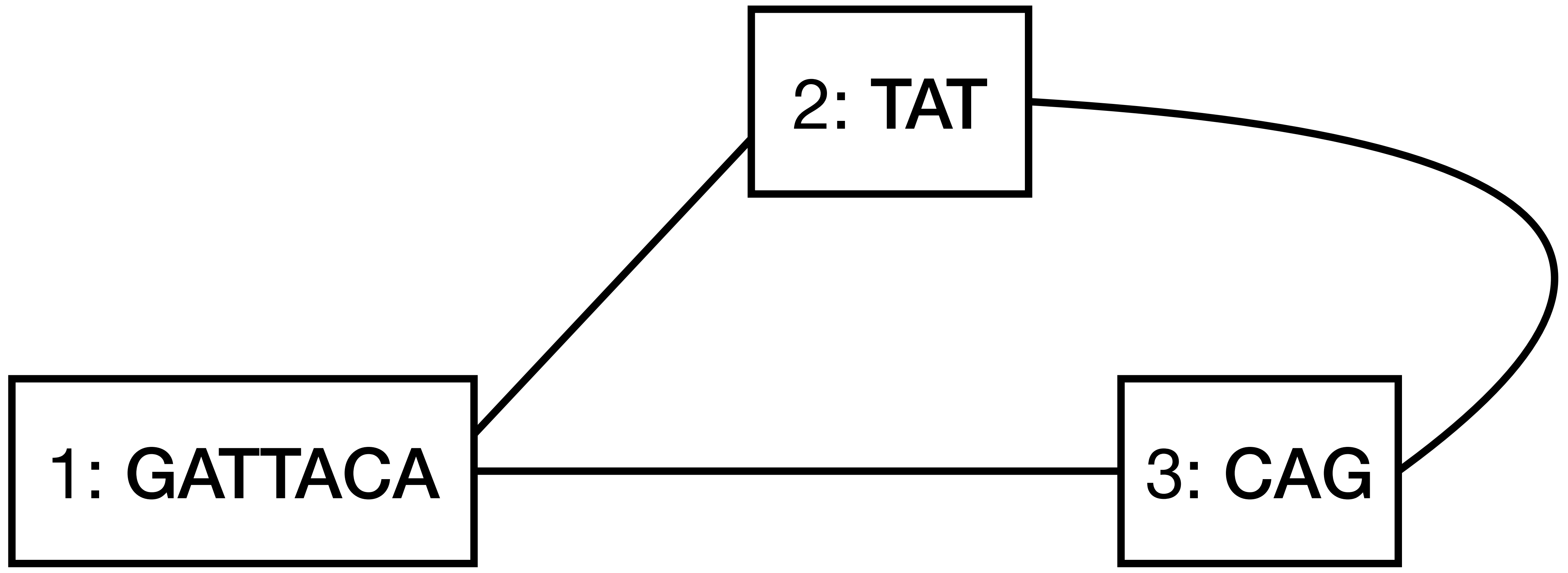
Bidirected sequence graphs

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**.

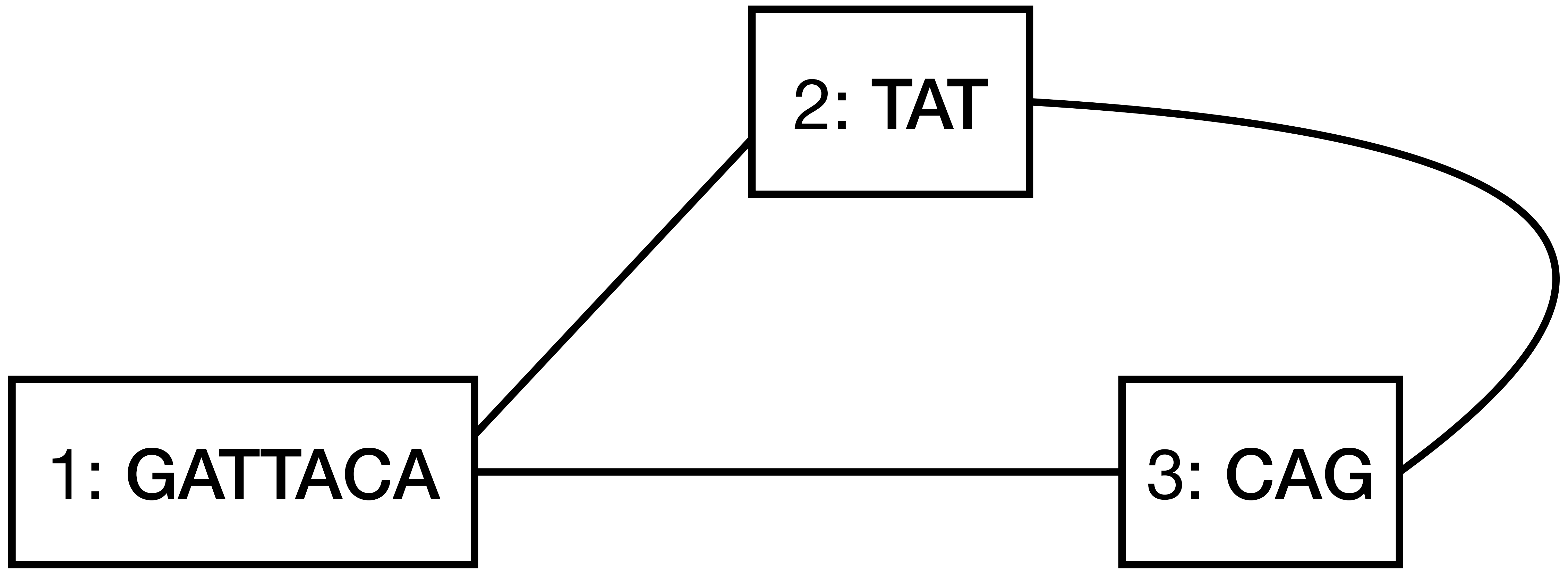
Traversal >1 >3 <2 <1 reads **GATTACA**, **CAG**, **ATA**, and **TGTAATC**.

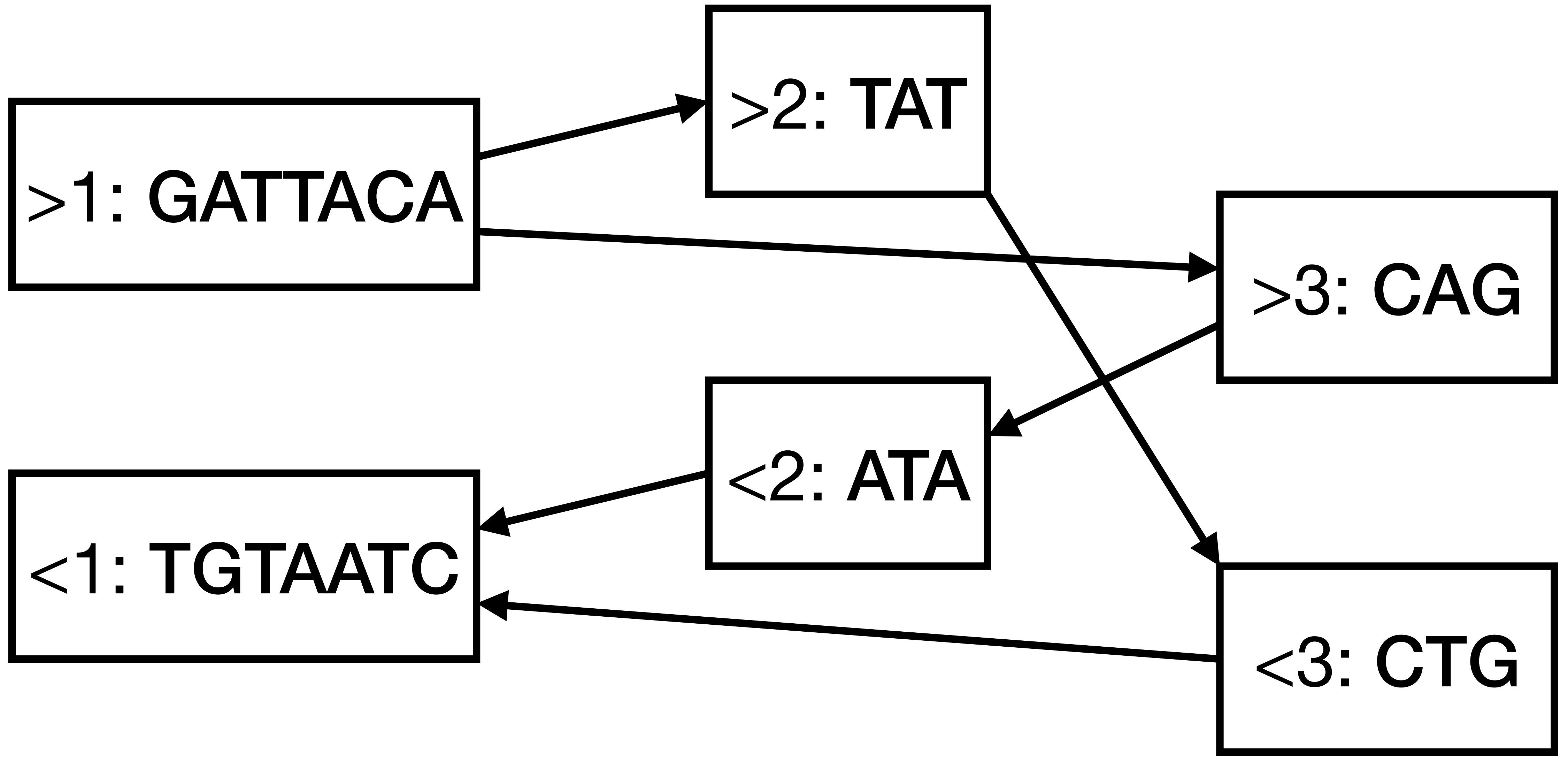
Simulating bidirected graphs

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.





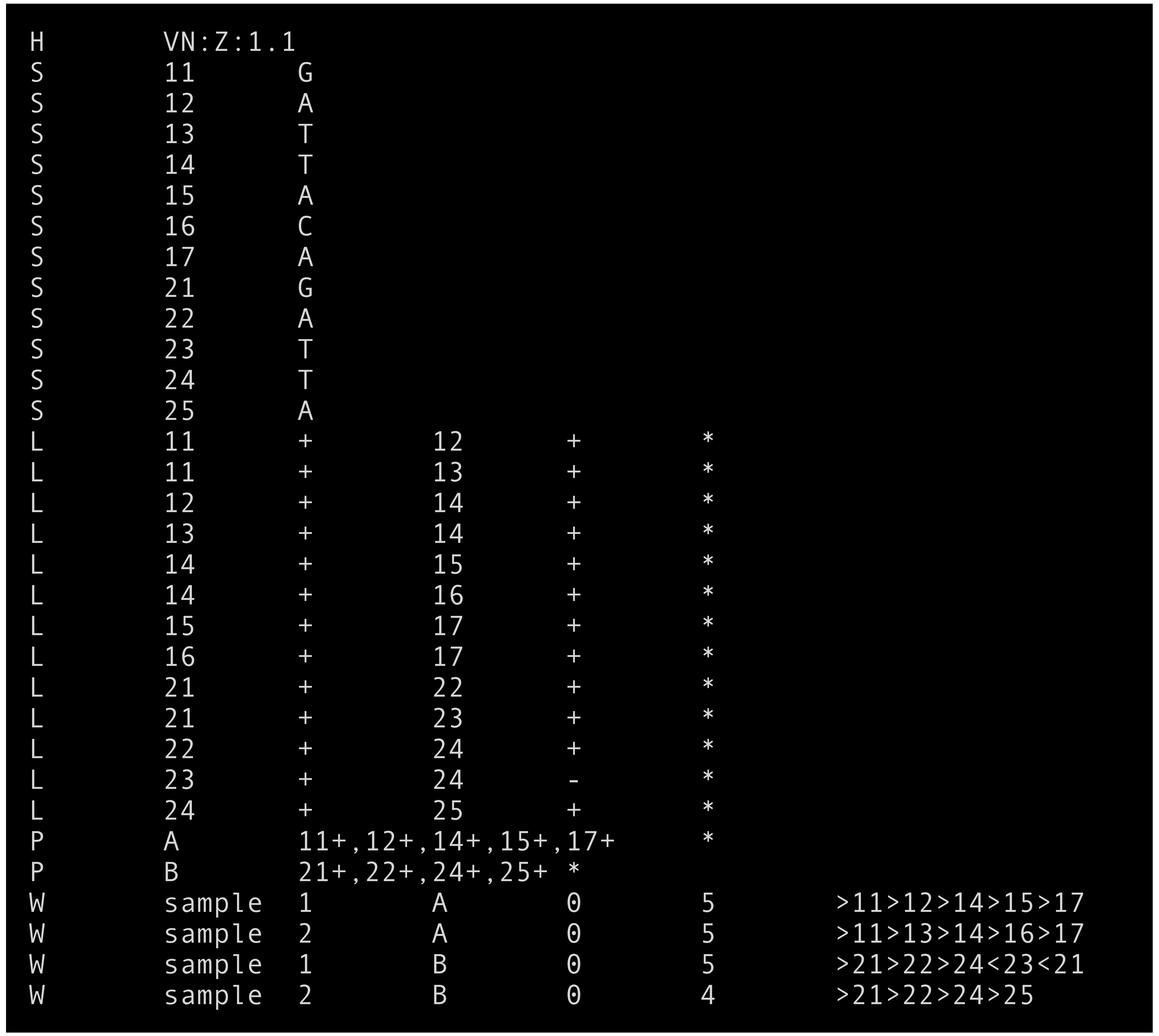
GFA file format

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

Originally intended for assembly graphs, a subset of GFA is 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
```

GFA compression

GFA 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.

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?

Rank, select, and bitvectors

Notation

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 $S[j]$.

$S.\text{rank}(i, c)$ 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.

$S.\text{select}(i, c) = A_c[i]$ 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

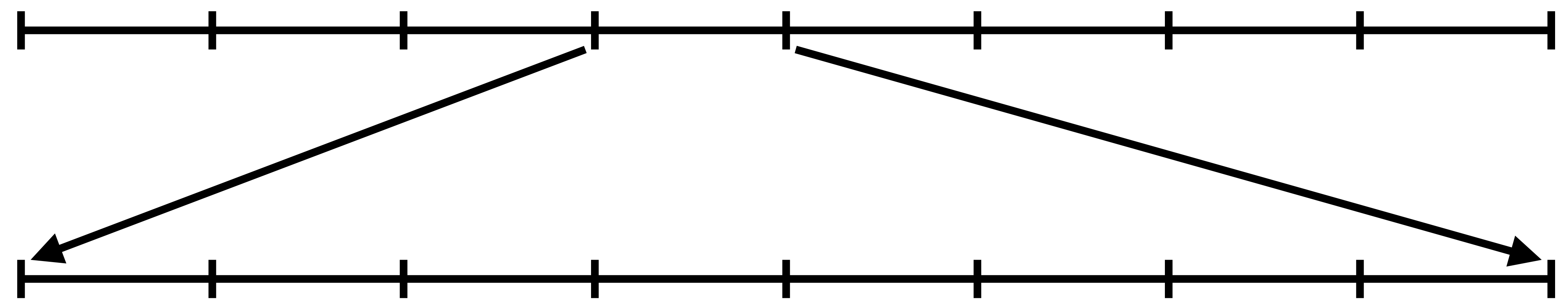
```
B:  0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 1 1 0 1 0 0
A:  2   3   7   8   12   13   16   17   19
```

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 and Petri: **Optimized succinct data structures for massive data.** Software – Practice and Experience, 2014.

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** 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

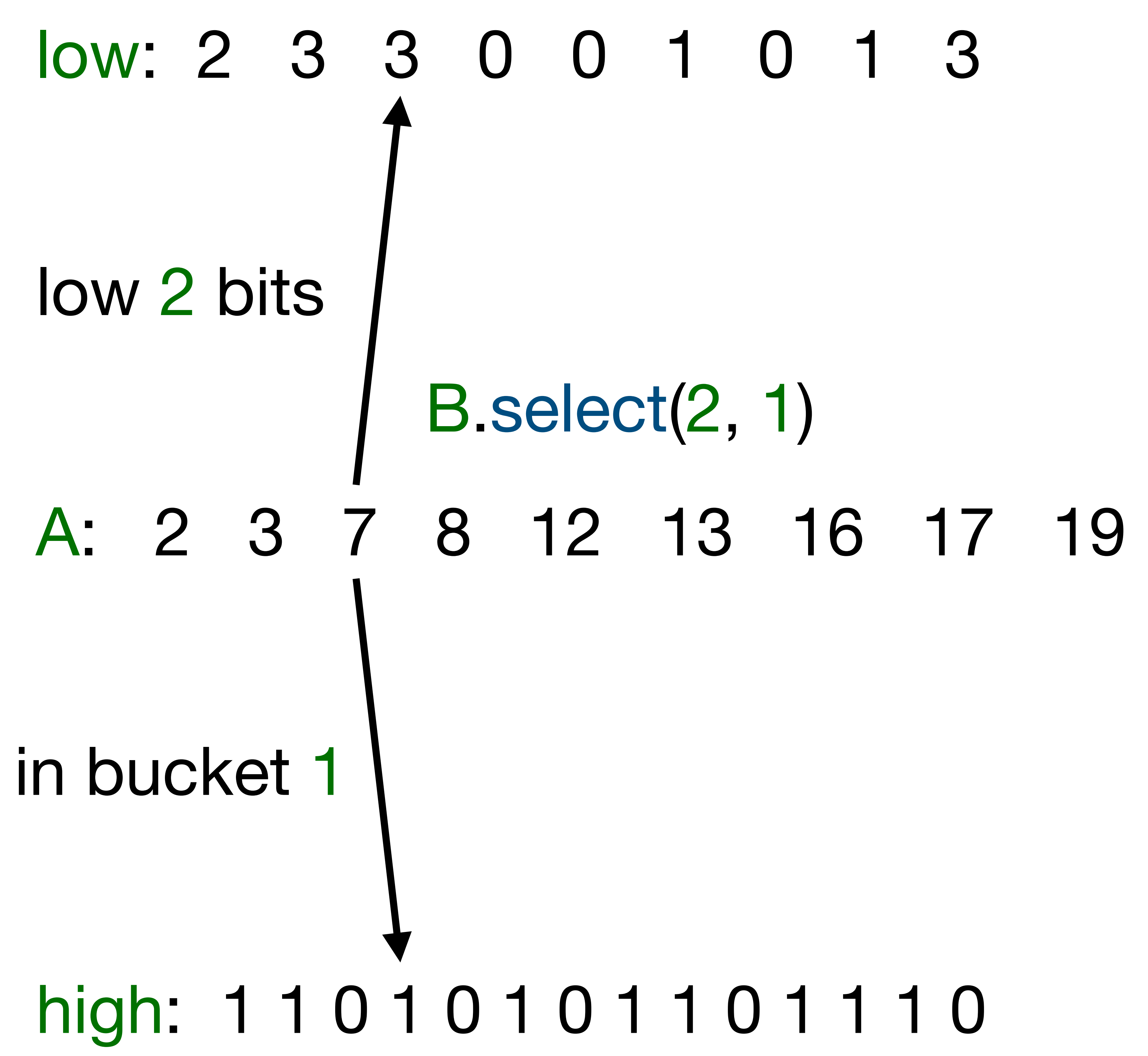
Accessing the original values is simple:
$A[i] = (high.select(i, 1) - i) \cdot 2^w + low[i]$.

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



Sparse bitvectors

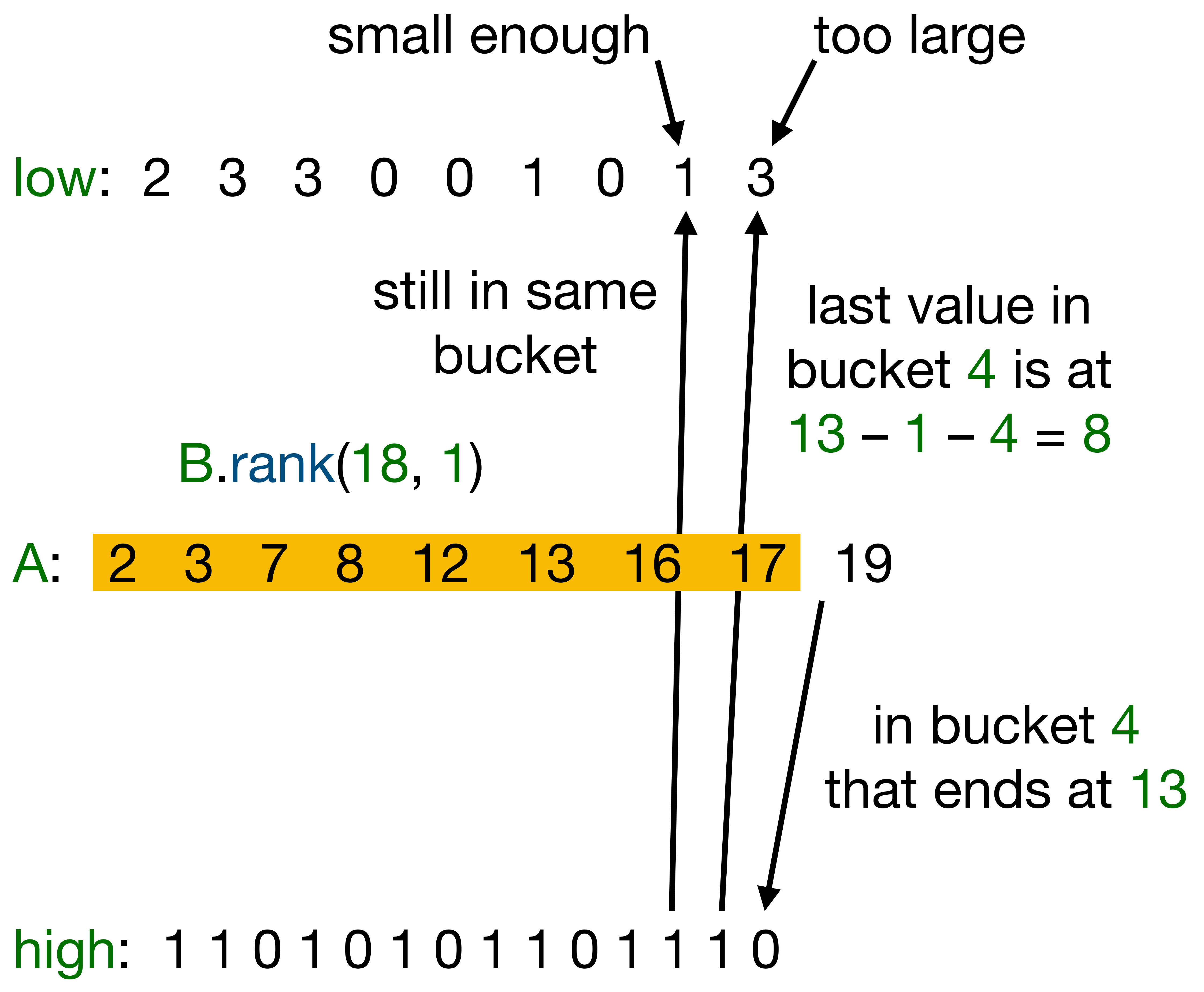
Accessing the original values is simple:
A[i] = (high.select(i, 1) – i) · 2^w + low[i].

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


small enough
too large
low: 2 3 3 0 0 1 0 1 3
still in same bucket
last value in bucket 4 is at 13 – 1 – 4 = 8
B.rank(18, 1)
A: 2 3 7 8 12 13 16 17 19
in bucket 4 that ends at 13
high: 1 1 0 1 0 1 0 1 1 0 1 1 1 0

Burrows–Wheeler transform

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$

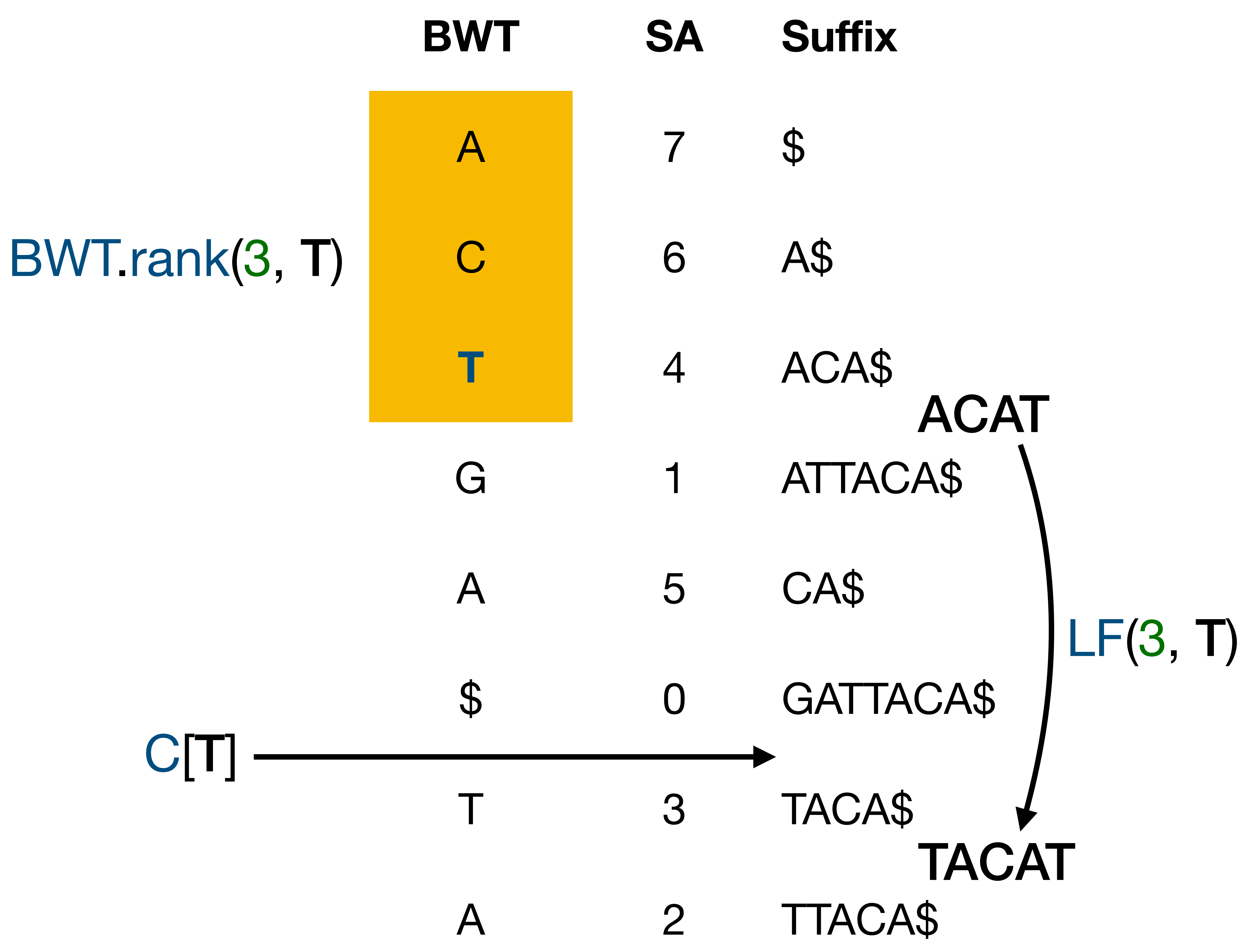
LF-mapping

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.



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$

Inverting the BWT

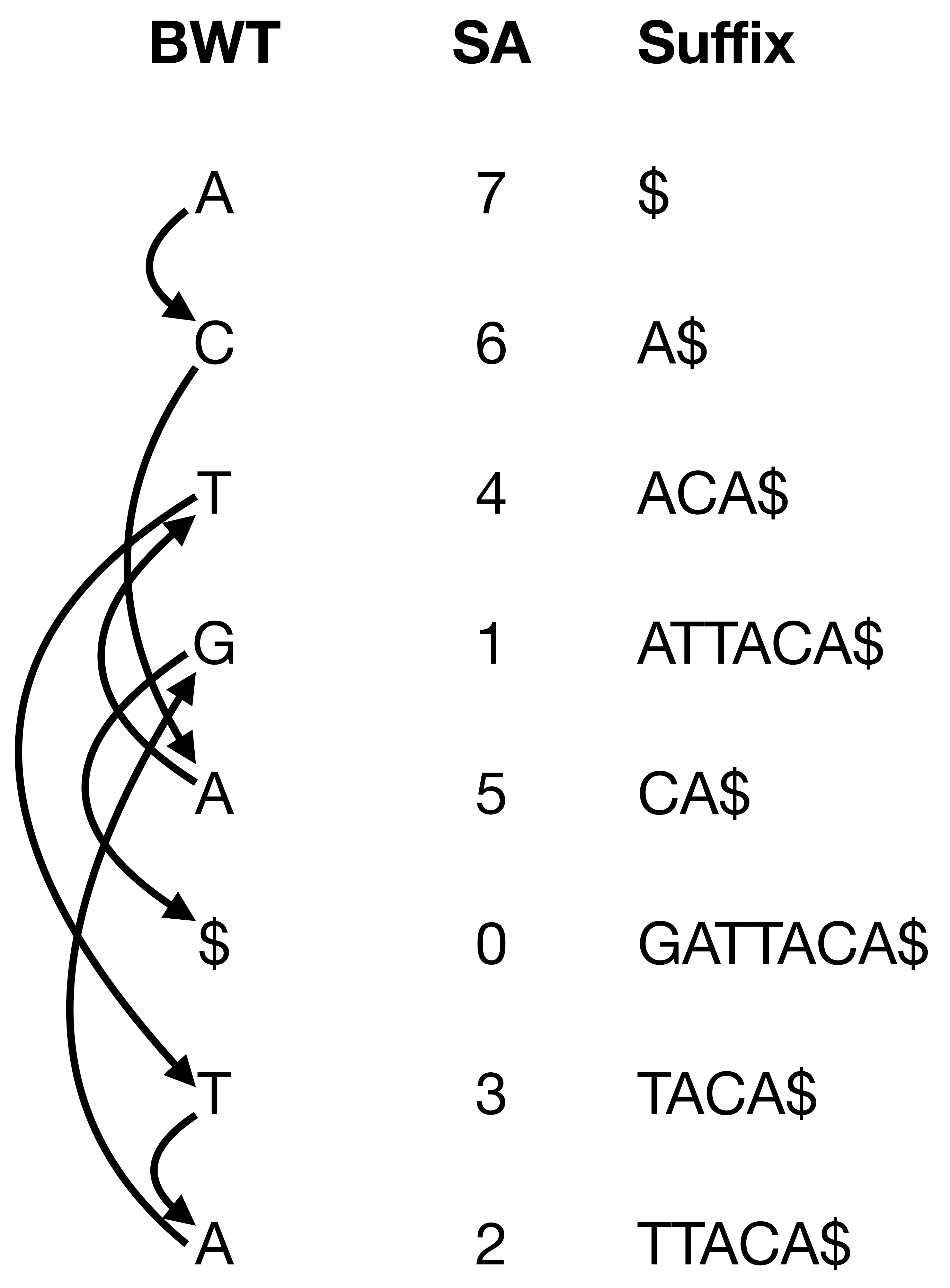
Because $ is the smallest character, we know that SA[0] = n – 1 and BWT[0] is the character preceding the **endmarker**.

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**.



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$

Multi-string BWT

Let $T_0, \ldots, T_{m-1}$ 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$**
A	0	(0, 2)	TTACA$

Backward searching

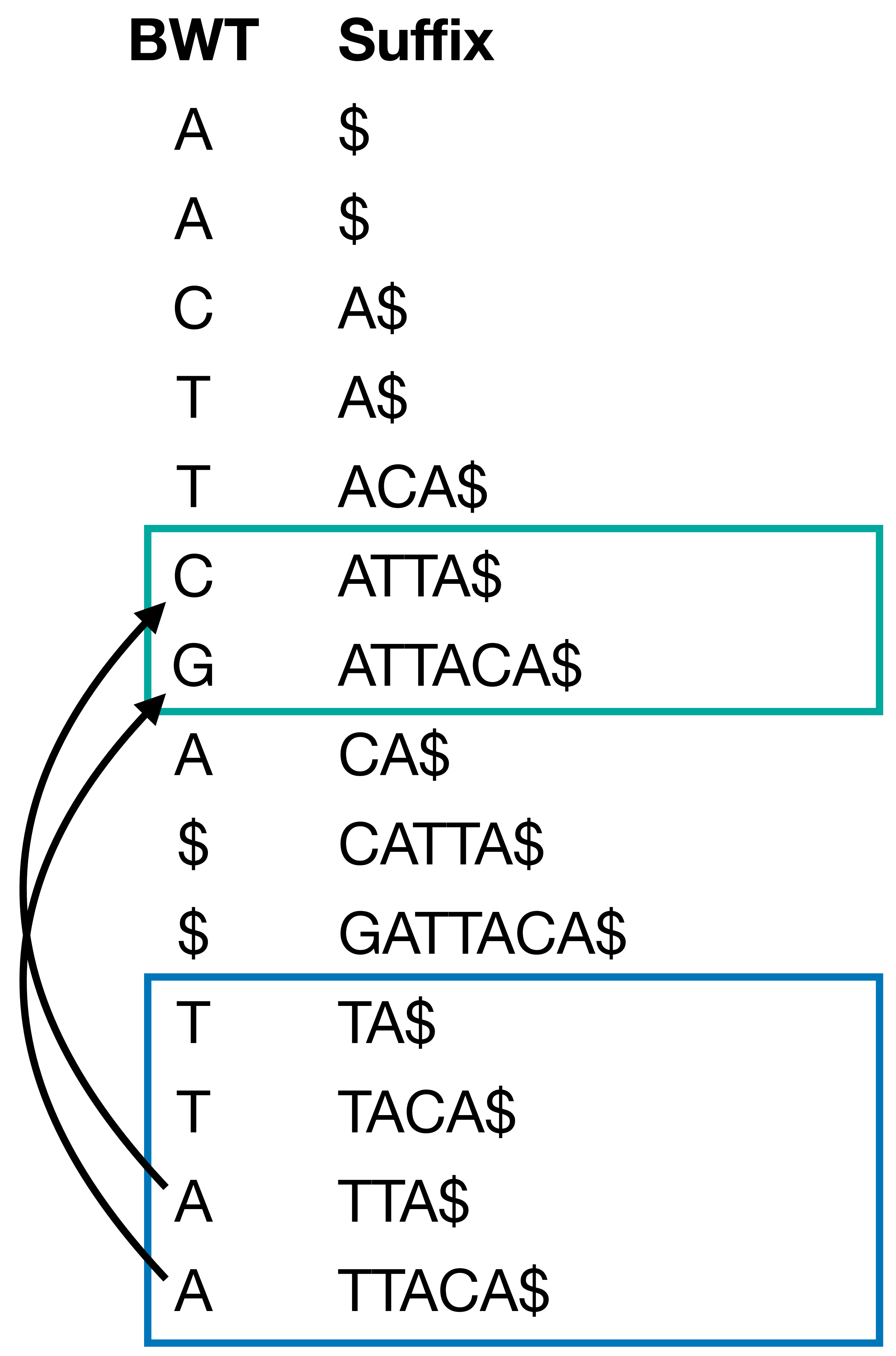
If SA[i..j) is the range of suffixes starting with string X, the range of suffixes starting with string cX is SA[LF(i, c)..LF(j, c)).

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**



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$

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 and Manzini: **Indexing Compressed Text**. JACM, 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.

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.

Runs in BWT

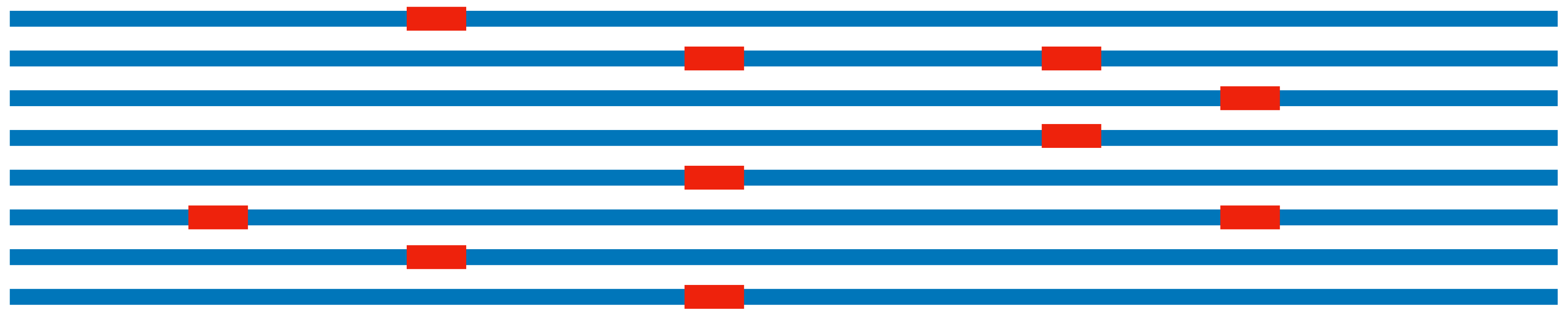
Repetitiveness in a text collection manifests as long equal letter **runs** in its BWT.

Adding a new **copy** of an existing text does not increase the number of runs.

Each **edit operation** creates $O(1)$ points of discontinuity and moves a number of suffixes preceding them in lexicographic order.

Suffixes far enough from the edits maintain their positions relative to **unrelated** suffixes.

An **insertion** of length k may create $O(k)$ additional runs.



If we start from a single text T and the total length of T and all insertions is n, there should be $O(n + s \log_\sigma n)$ runs after s edits, where σ is effective alphabet size.

Remember that there are only hundreds of edits in a human generation.

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

BWT ~ stable sorting

There are generalizations of the BWT for:

de Bruijn graphs: Nodes and edges represent substrings of length k and $k + 1$. (Bowe et al: **Succinct de Bruijn Graphs**. WABI 2012.)

DAGs, but potentially with an exponential blowup. (Sirén et al: **Indexing Graphs for Path Queries with Applications in Genome Research**. TCBB, 2014.)

Positional string collections: If $T_j[i] = c$, the effective character value is (i, c). (Durbin: **Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT)**. Bioinformatics, 2014.)

They rely on the following interpretation of $\mathrm{LF}(i, c) = C[c] + \mathrm{BWT.rank}(i, c)$:

- $C[c]$: Sort positions by the most significant character.
- $\mathrm{BWT.rank}(i, c)$: Break ties by maintaining the existing order.

Wheeler graphs

A directed edge-labeled graph is a **Wheeler graph**, if the nodes have an ordering such that:

1. Nodes with indegree 0 precede those with a positive indegree.

2. For any pair of edges (u, v) and (u', v') labeled a and a', respectively:

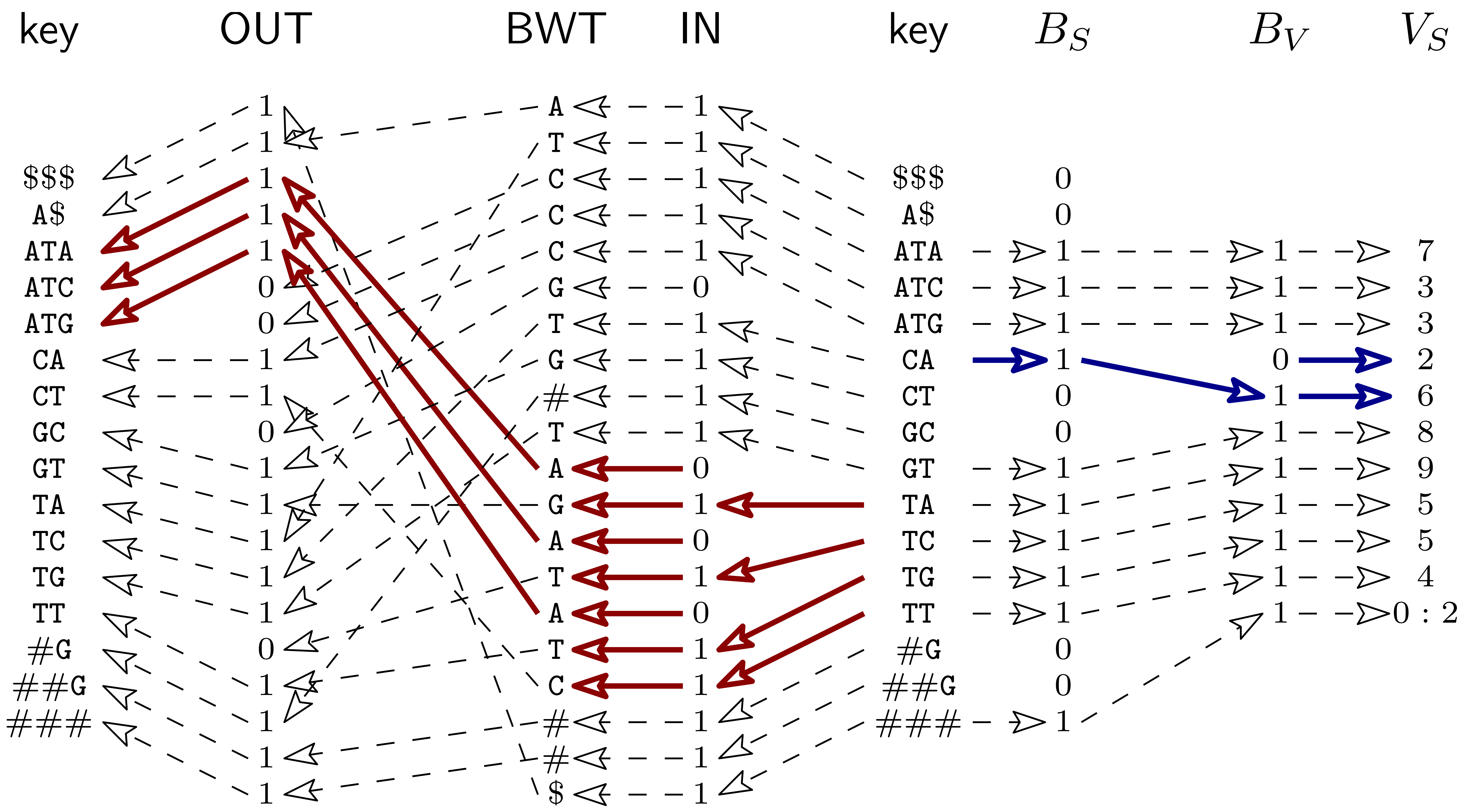
   A. $a < a' \Longrightarrow v < v'$,

   B. $a = a'$ and $u < u' \Longrightarrow v \leq v'$.

Wheeler graphs can be represented using the BWT and bitvectors encoding the indegrees and outdegrees in unary.

The BWT is that of **reverse** path labels, because we want LF-mapping to follow edges forward.

Gagie, Manzini, and Sirén: **Wheeler graphs: A framework for BWT-based data structures**. TCS, 2017.


key
OUT
BWT
IN
key
B_S
B_V
V_S
$$$ A$ ATA ATC ATG CA CT GC GT TA TC TG TT #G ##G ###
1 1 1 1 1 0 0 1 1 0 1 1 1 1 1 0 1 1 1 1
A T C C C G T G # T A G A T A T C # # $
1 1 1 1 1 0 1 1 1 1 0 1 0 1 0 1 1 1 1 1
$$$ A$ ATA ATC ATG CA CT GC GT TA TC TG TT #G ##G ###
0 0 1 1 1 1 0 0 1 1 1 1 1 0 0 1
1 1 1 0 1 1 1 1 1 1 1
7 3 3 2 6 8 9 5 5 4 0 : 2

GBWT

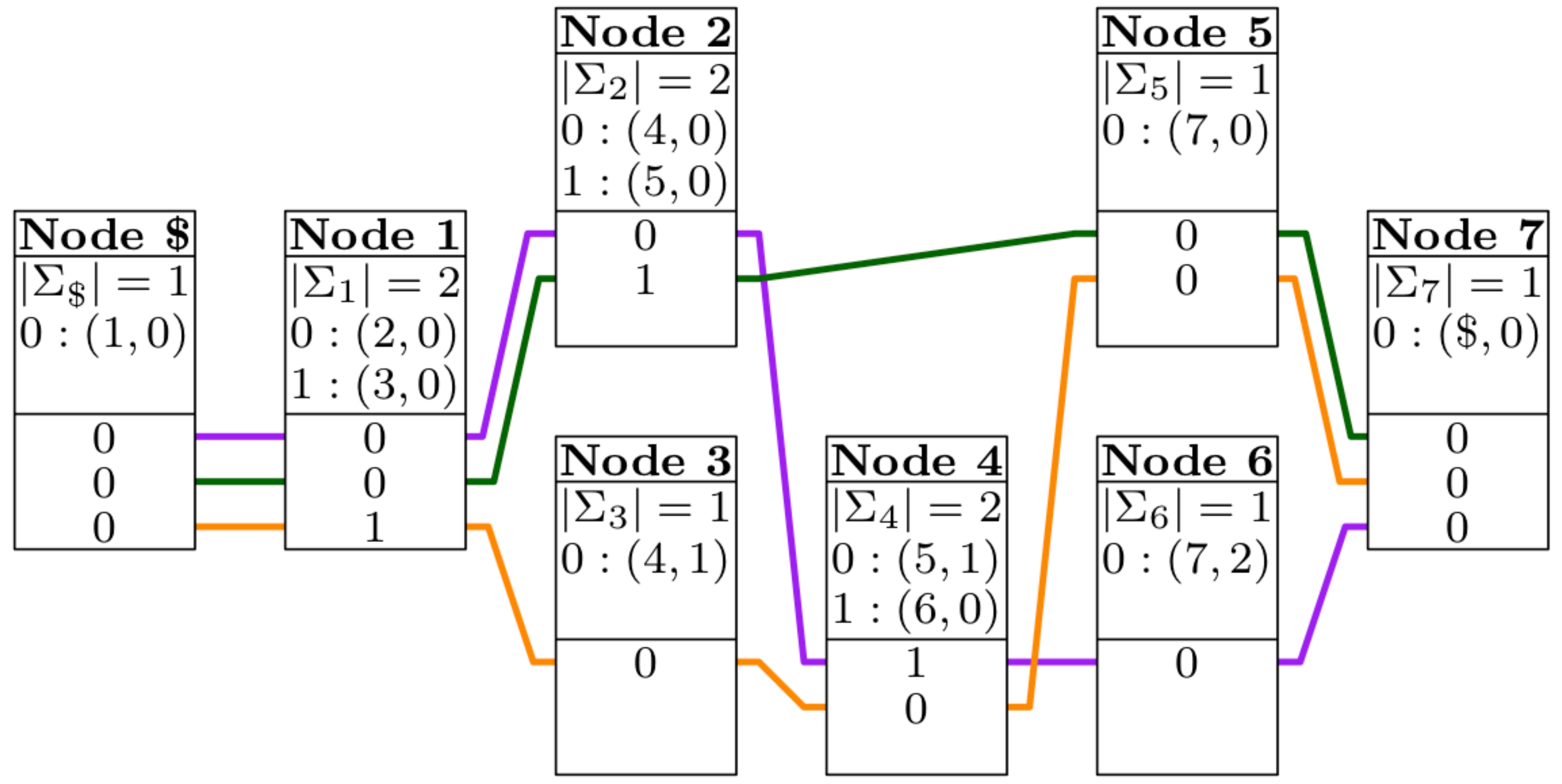
GBWT

The **GBWT** is a **reverse** FM-index (or FMD-index) 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.

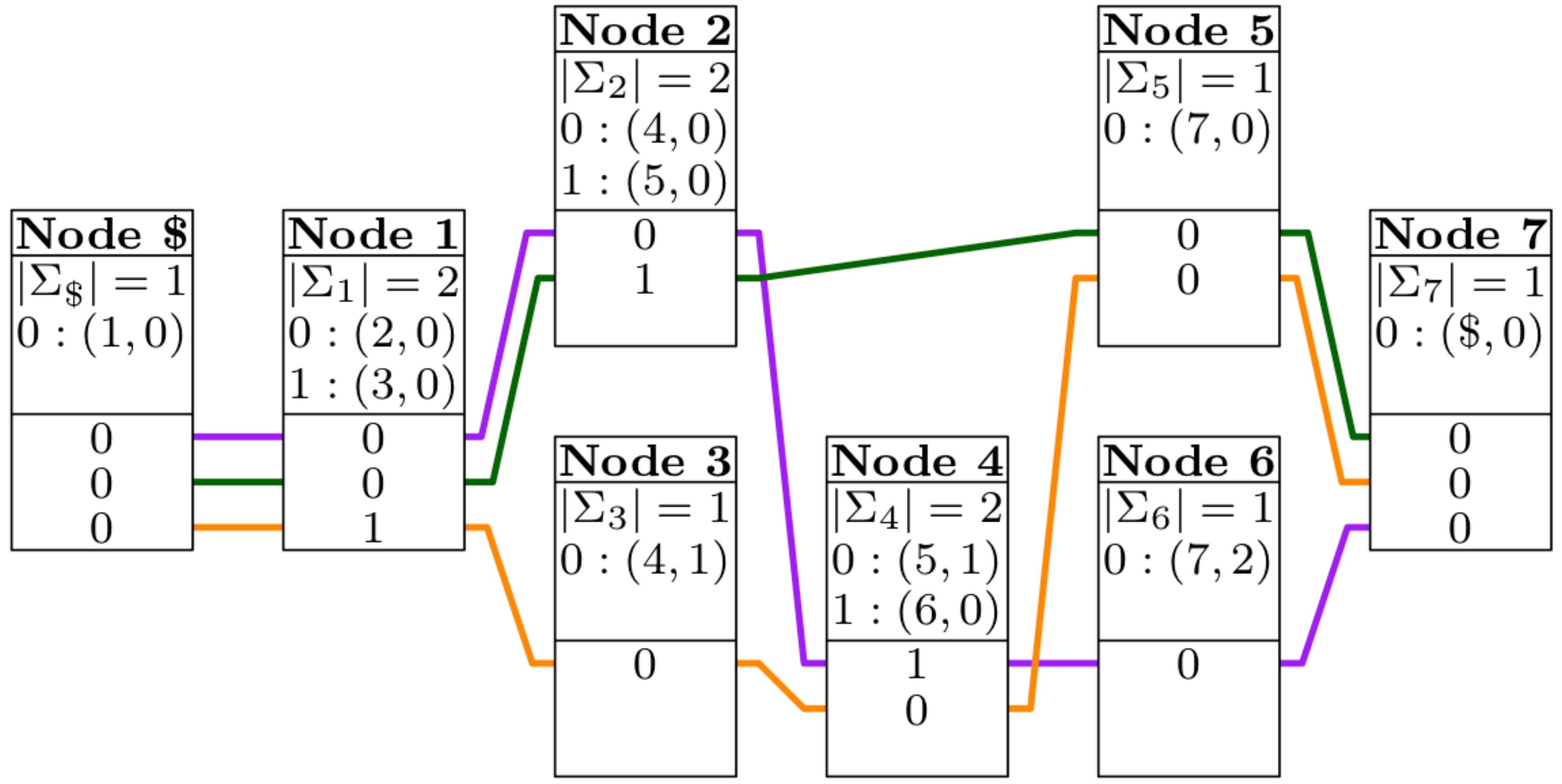
A find query determines how many indexed paths contain the corresponding **traversal** as a **subpath**.



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.

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	\$

BWT_4

LF-mapping

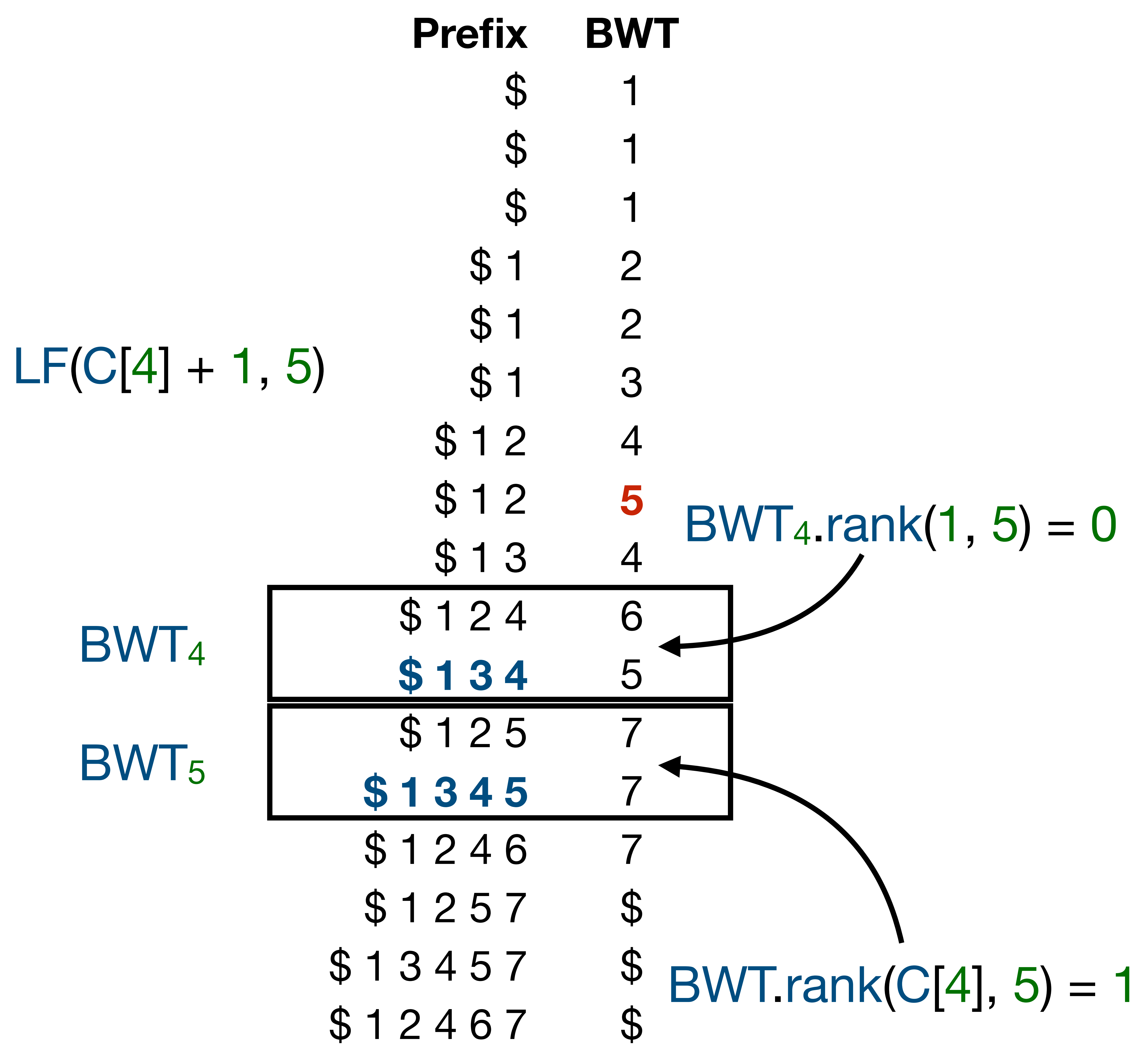
BWT offsets: (v, i) vs. C[v] + i vs. BWT$_v$[i].

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**.



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	$

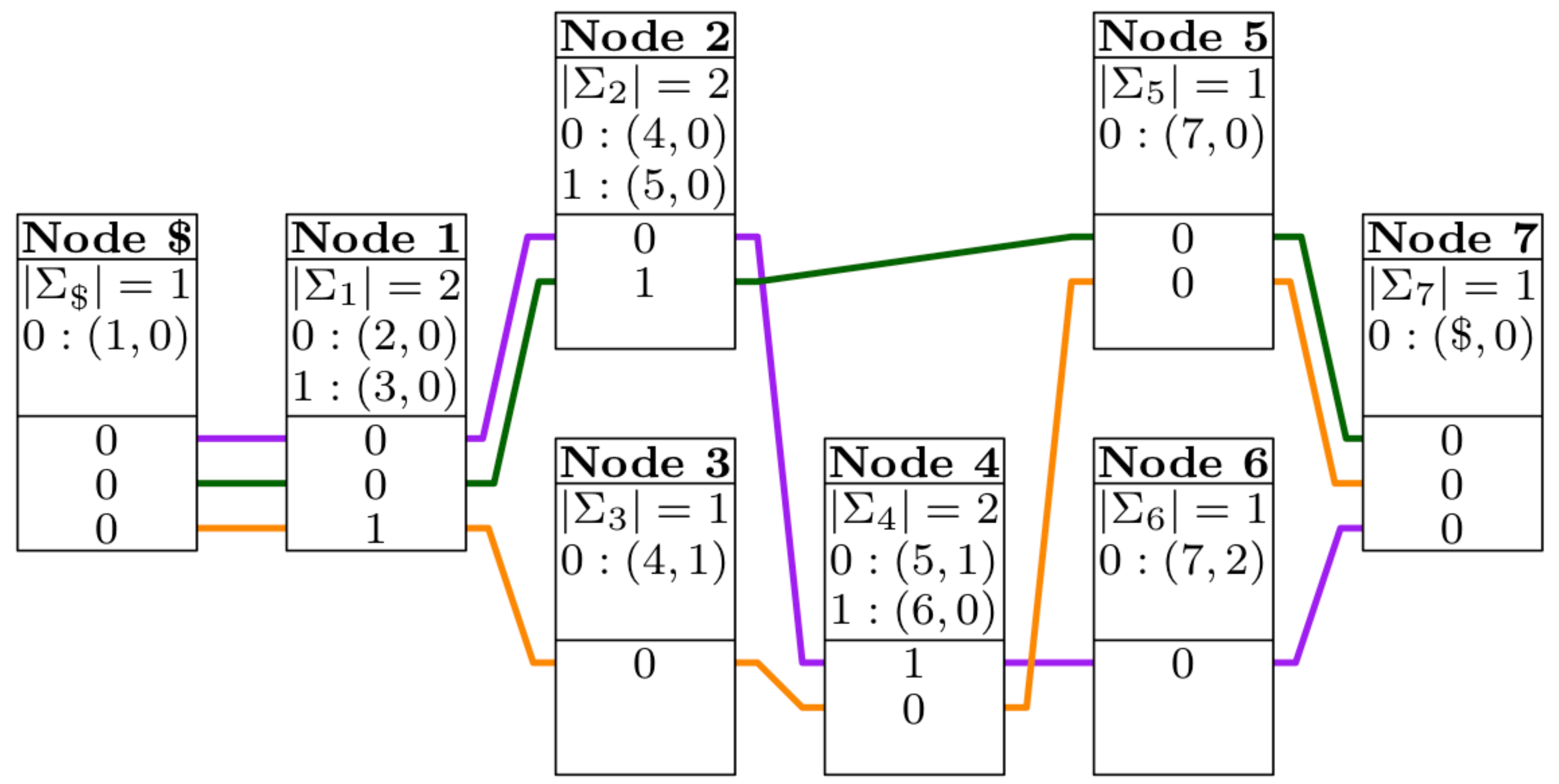
Node records

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**, using a 7+1-bit encoding for integers. The encoding for runs depends on the outdegree.



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

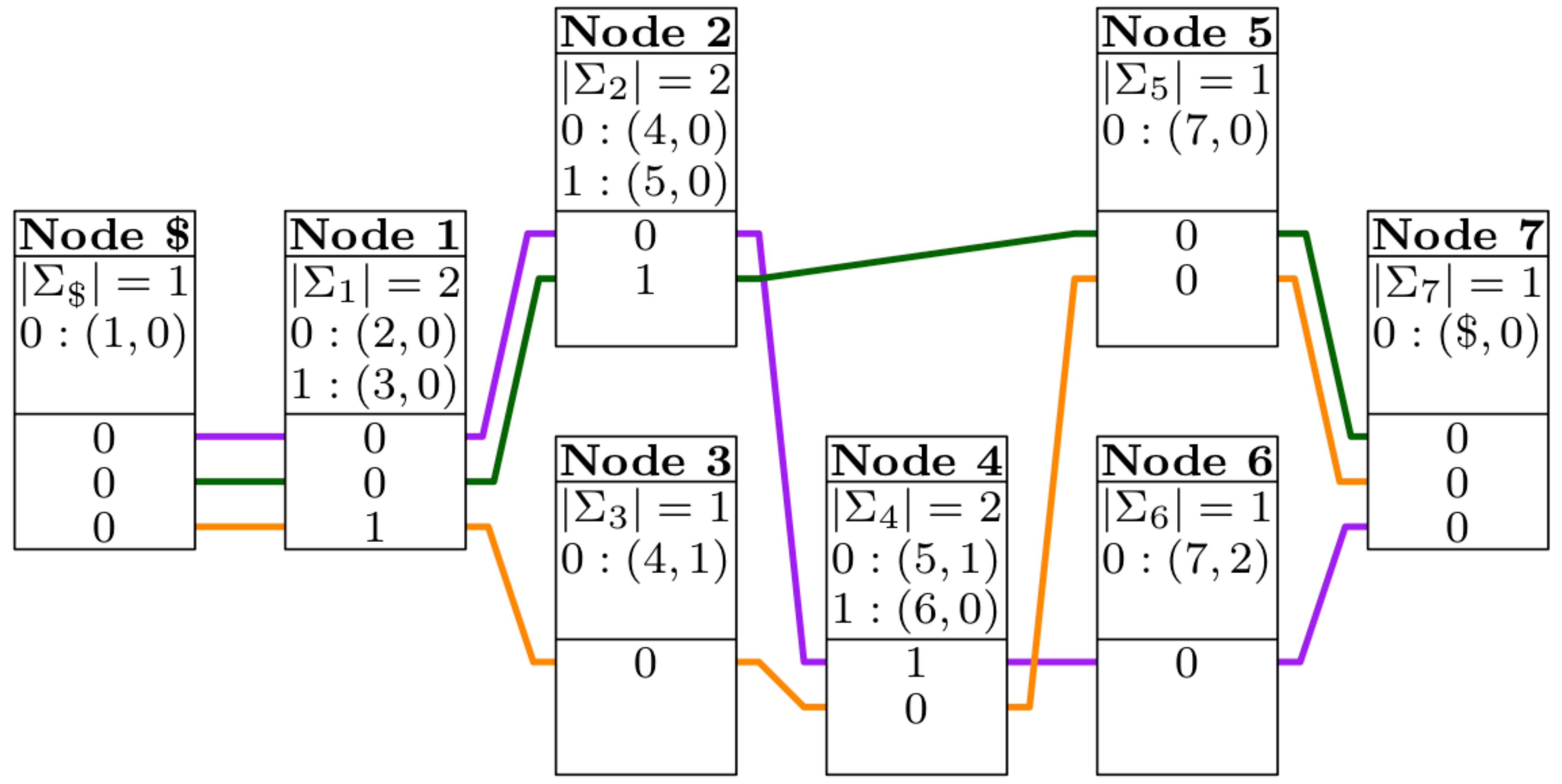
Using the GBWT

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
```

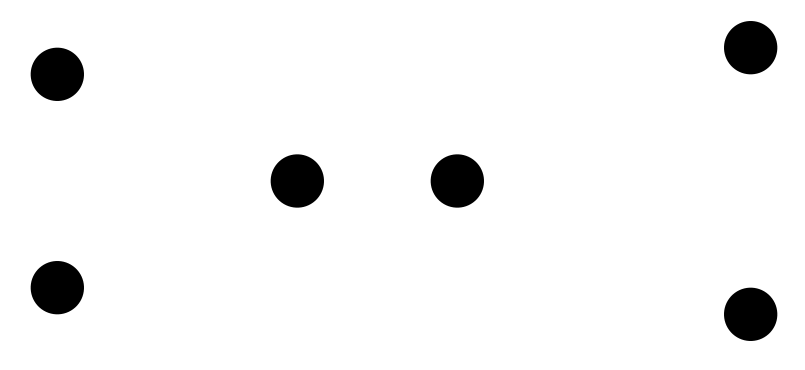
Encoding of the records and bitvector B (each byte is a single digit).

Path / subgraph traversals

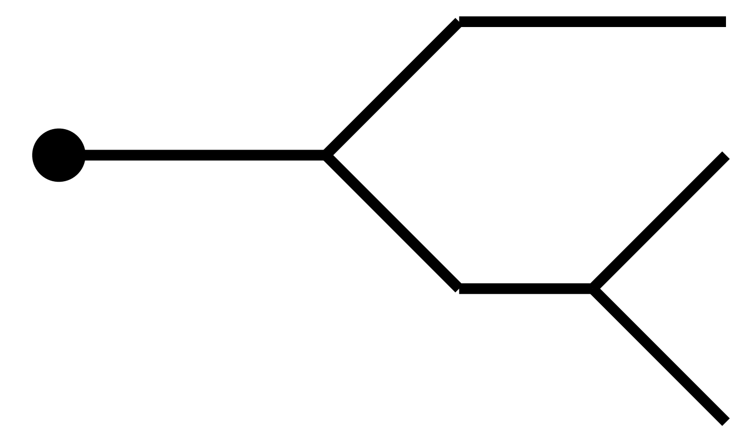
As we **traverse a path** in the graph using LF-mapping, the length of the BWT range tells the number of times the traversal occurs as a subpath in the haplotypes.

We often traverse **all possible extensions** in a subgraph, as long as some invariant holds and the traversal is supported by the haplotypes.

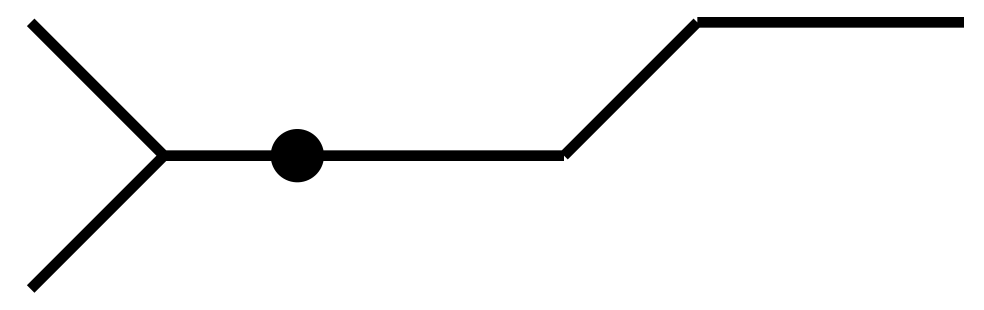
Cluster of seeds



Forward extensions of a seed



Backward extensions of an extension



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).

GFA segments have string **names**, while GBWT nodes have integer **identifiers**.

Segments can be arbitrarily long, but we may want to restrict the **length of nodes** for various reasons.

A **translation** between GFA segments and (ranges of) GBWT nodes can be stored using a string array for segment names and a sparse bitvector for the ranges.

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

https://github.com/jltsiren/gbwtgraph

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.** TCS, 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
```

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.

A 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.

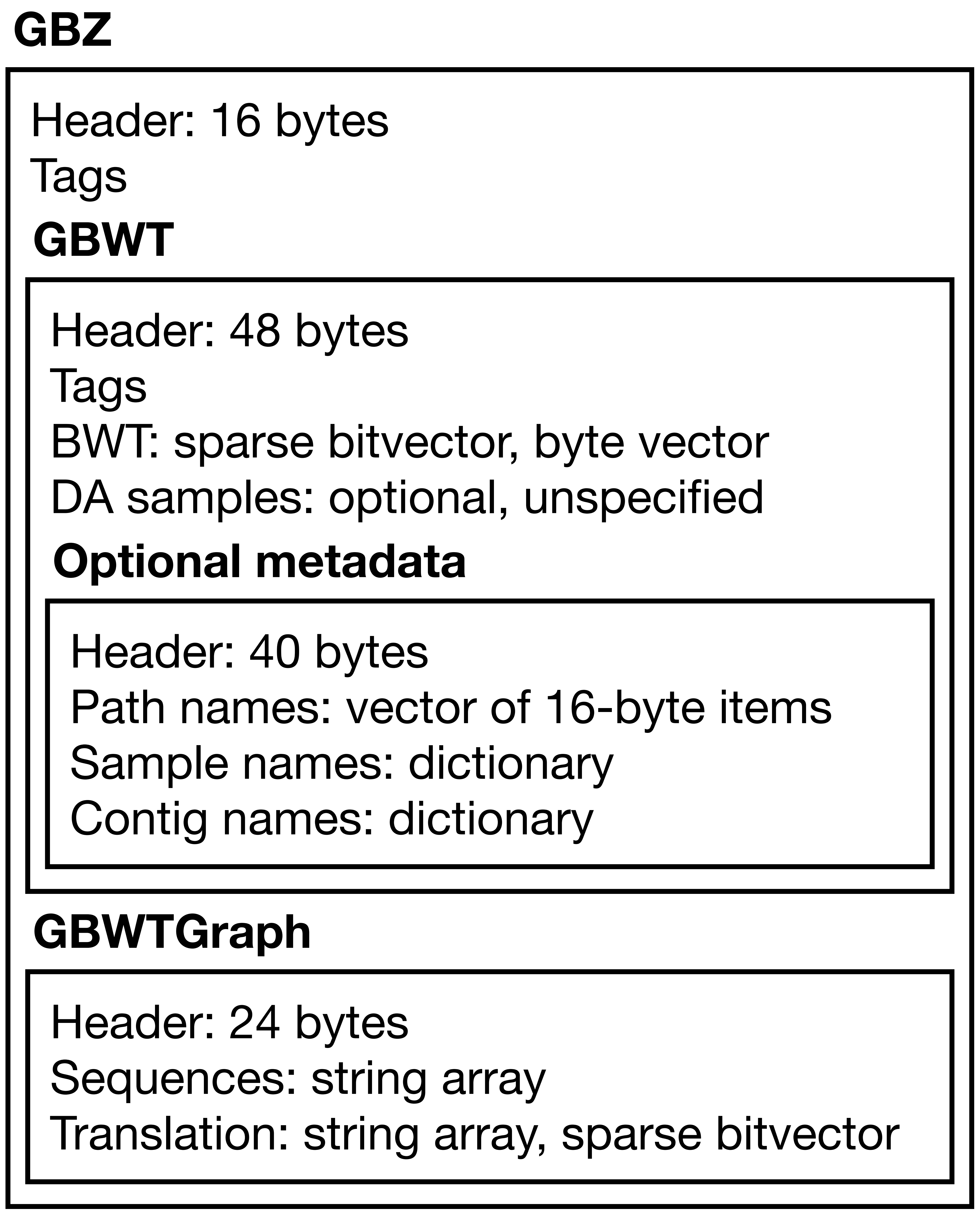
Tags: Key–value structure with case-insensitive keys. Stored as a string array. Key source identifies the library that wrote the file. The reader can use that information for determining if it can understand the optional structures.

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

Sirén and Paten: **GBZ file format for pangenome graphs**. Bioinformatics, 2022.



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.

GBZ benchmarks

Graph	Haplotypes	.gfa	.gfa.gz	.gbz	Compression	Decompression
HPRC	90	44.9 GiB	11.1 GiB	3.11 GiB	19 min 111.0 GiB	2 min 14.5 GiB
1000GP	~5000	9534.9 GiB	2231.3 GiB	16.84 GiB	779 min 489.2 GiB	124 min 49.3 GiB

HPRC: AWS i3.8xlarge

- 16 physical / 32 logical CPU cores
- 244 GiB RAM
- 16 parallel GBWT construction jobs
- 16 decompression threads

1000GP: AWS i4i.16xlarge

- 32 physical / 64 logical CPU cores
- 512 GiB RAM
- 32 parallel GBWT construction jobs
- 32 decompression threads

Sirén and Paten: **GBZ file format for pangenome graphs**. Bioinformatics, 2022.

Focus on data layout

- Designing a **portable** file format based on **highly specialized** data structures?

Two data structures sharing the same **layout** can often be built efficiently from each other.

We can then optimize the structures for different tasks.

GBWT: Compressed version for querying, dynamic version that supports inserting and deleting paths.

GBZ: C++ implementation focuses on fast access to sequences, while Rust implementation uses much less memory.

GBZ at its core:

- A collection of **node records** containing an adjacency list, a BWT fragment, and a sequence.
- The records are encoded as **byte sequences**.
- There is an **index** for finding a record based on its identifier.

GBZ in SQLite

We could do something like this:

```
CREATE TABLE Nodes (
    handle INTEGER PRIMARY KEY,
    edges BLOB,
    bwt BLOB,
    sequence BLOB
)
```

Inserting the key parts of an HPRC GBZ graph into a **SQLite database** takes ~90 seconds on this laptop.

Size increases from 3.06 GiB to 9.66 GiB without sequence compression. (Probably less than 6 GiB with alphabet compaction.)

Graph **traversal speed** is ~150k nodes per second, vs. a few million nodes/second with the in-memory GBZ graph.

The database is available **immediately**, vs. 15–20 seconds for loading the GBZ.

Potentially useful for **interactive** applications.

Overview

This talk was about the **GBZ file format** for pangenome graphs.

GBZ is based on the **GBWT index**, which stores a set of paths as sequences of node identifiers.

GBWT is a run-length encoded **FM-index** partitioned between the nodes of the graph.

FM-index is a space-efficient text index based on the **Burrows–Wheeler transform**.

Sirén and Paten: **GBZ file format for pangenome graphs**. Bioinformatics, 2022.

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

Ferragina and Manzini: **Indexing Compressed Text**. JACM, 2005.

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

Thank you!