Indexing Paths in Genome Graphs

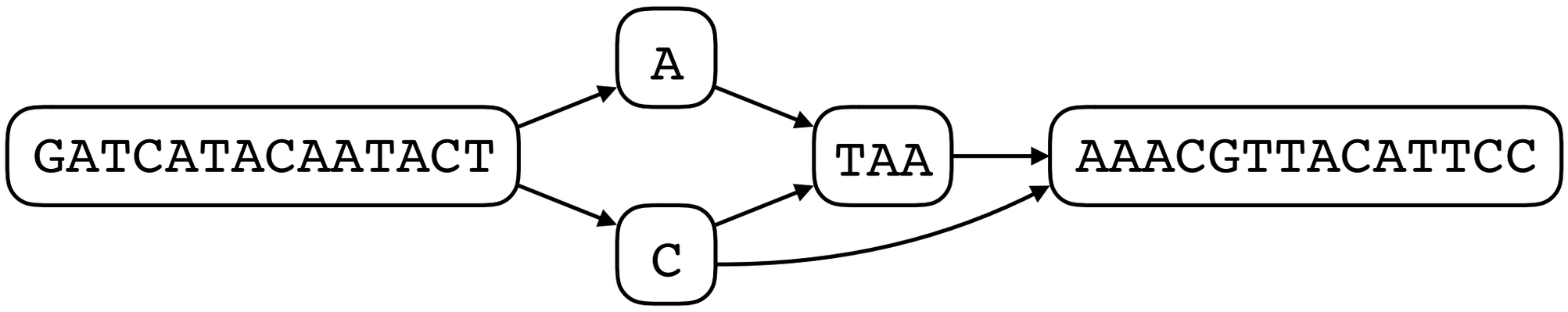
Jouni Sirén

Reference bias

```
Reference   GATCATACAATACTATAAAAACGTTACATTCC
Sample 1    GATCATACAATACTCTAAAAACGTTACATTCC
Sample 2    GATCATACAATACTC---AAACGTTACATTCC
```

- Sequence reads are much **shorter** than the genome, and they often contain **errors**.
- A **reference genome** makes it easier to infer the sequenced genome from the reads.
- The inference is **biased** towards the reference, while there may be **systematic differences** between the reference and the samples.

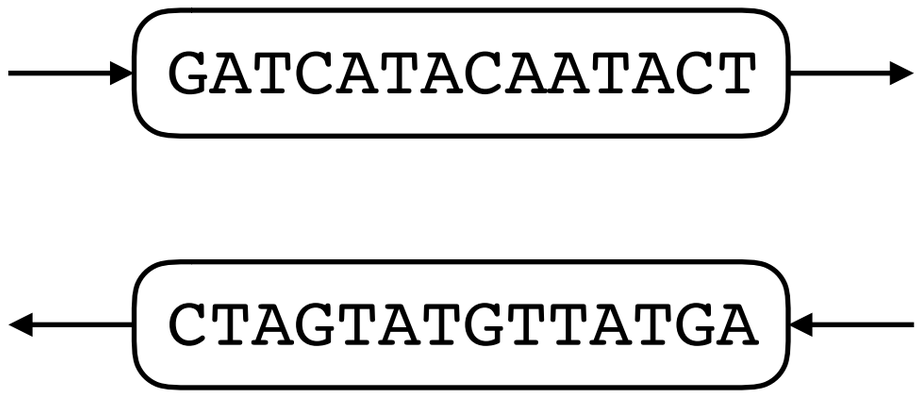
Genome graphs



- Genome graphs provide a richer reference model.
- Every path through the graph is a potential haplotype.
- If we choose the variation we include in the graph carefully, we can reduce the reference bias significantly.

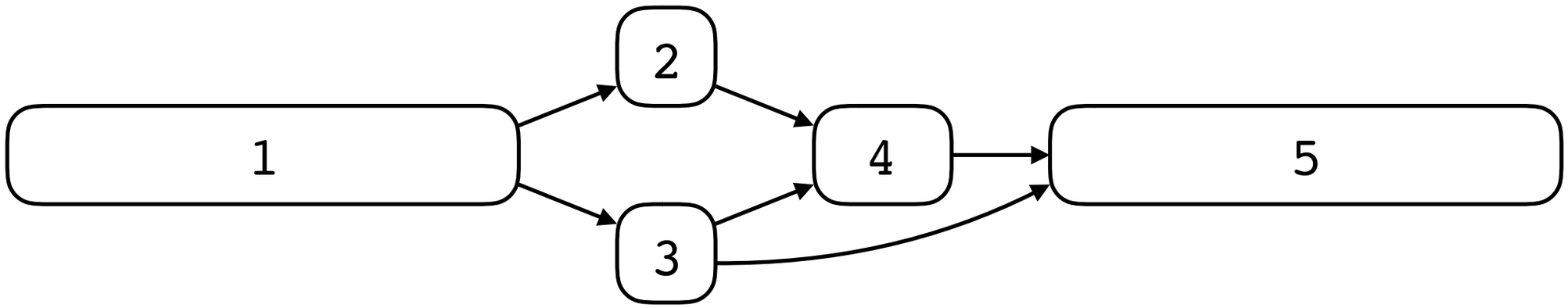
VG

- VG is an open source toolkit for genome graphs.
- Each node is effectively two nodes: the sequence and its reverse complement.



- https://github.com/vgteam/vg
- Garrison, Sirén, Novak, Hickey, Eizenga, Dawson, Jones, Lin, Paten, Durbin: Sequence variation aware genome references and read mapping with the variation graph toolkit.
- (The preprint should be out soon.)

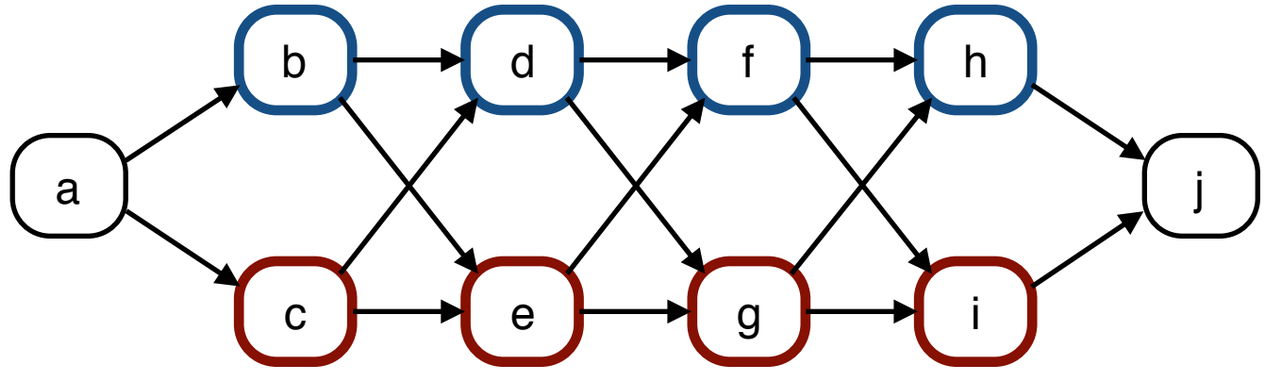
Haplotype storage

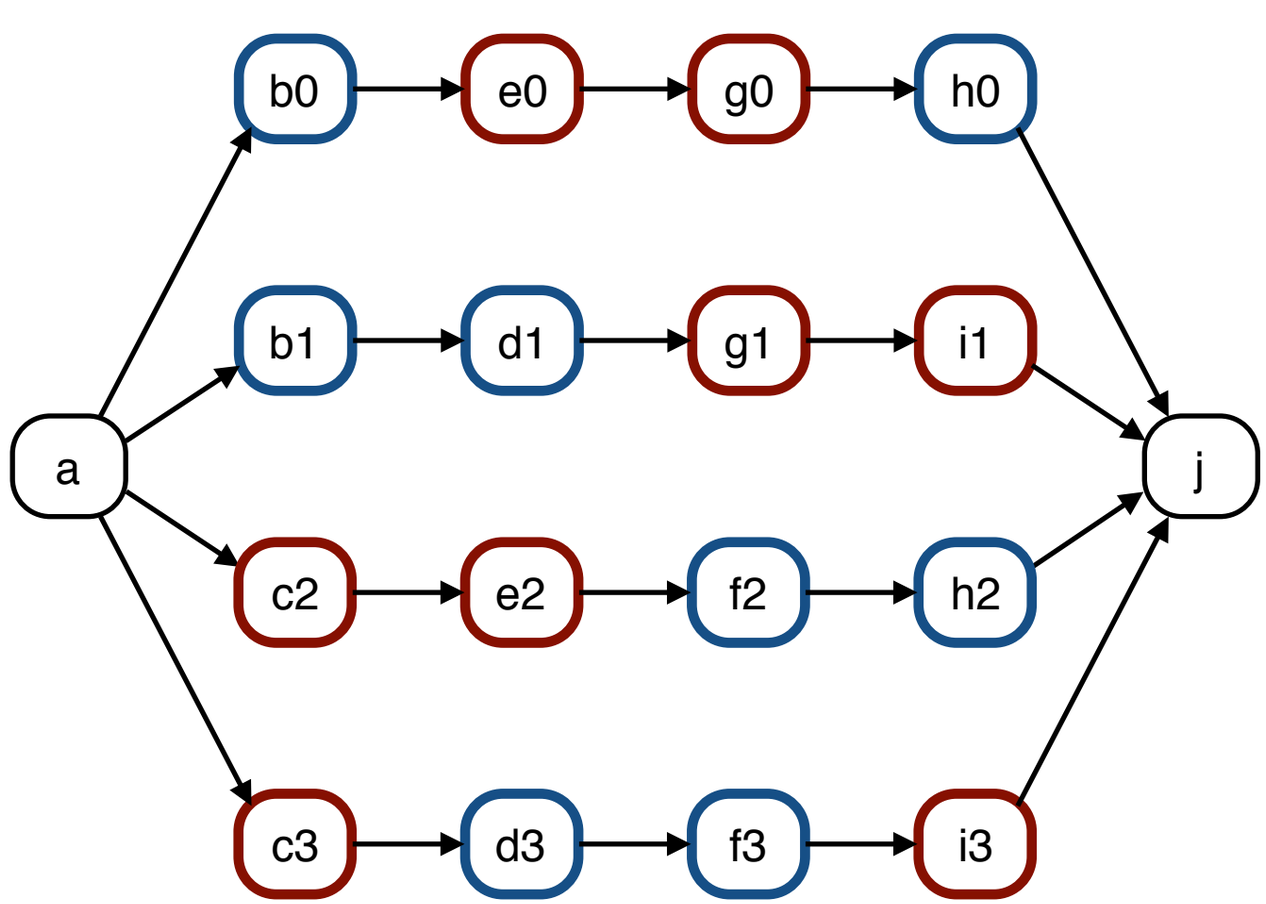


- We want to store haplotypes / paths in the graph / sequences of node identifiers.
- As the haplotypes should be similar, we can store large collections in small space.
- Typical queries: count the number of sequences matching the pattern; determine the identifiers of the matching sequences; extract a sequence.

Application: Indexing sequences

- Complex regions cause problems when we index the sequences in the graph.
- Various heuristics exist to simplify such regions, but they often lose the variation we are interested in.
- We can preserve the variation by unfolding the known haplotypes in the region.
- We index the unfolded graph, but the index maps to the original graph.





Application: Read mapping

- We noticed in the VG paper that mapping short reads to a reference sequence yields better results than mapping them to the 1000GP graph.

- False positives outweigh the reduced reference bias, if the graph contains too many rare variants.

- We believe we can reduce false positives by introducing a recombination penalty: reduce the mapping quality if the alignment does not match any haplotype.

FM-index

Burrows–Wheeler transform

- Add a unique terminator ($) to the end of the text, sort the suffixes in lexicographic order, and output the preceding character for each suffix.

- The permutation is easily reversible and makes the text easier to compress (Burrows & Wheeler, 1994).

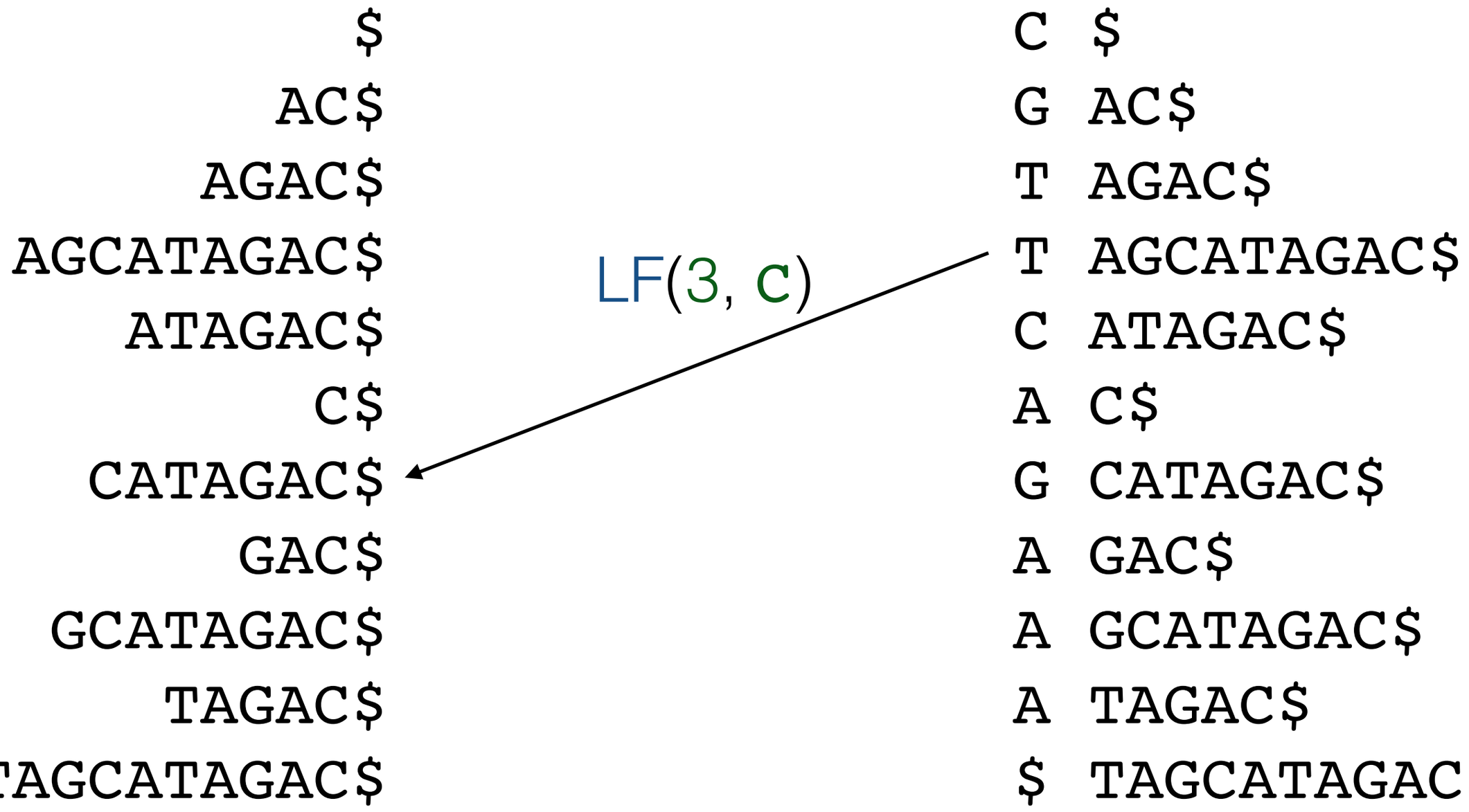
- The combinatorial structure is similar to the suffix array, which makes the BWT useful as a space-efficient text index (Ferragina & Manzini, 2000, 2005).

- There is a straightforward generalization for multiple strings by using distinct terminators during sorting.

```
TAGCATAGAC$

C $
G AC$
T AGAC$
T AGCATAGAC$
C ATAGAC$
A C$
G CATAGAC$
A GAC$
A GCATAGAC$
A TAGAC$
$ TAGCATAGAC$
```

LF-mapping



Interpretation: LF(i, c) = C[c] + BWT.rank(i, c) suffixes are **strictly before** the hypothetical suffix.

Incremental construction

TAGCATAGAC$ ⟶ **C**TAGCATAGAC$

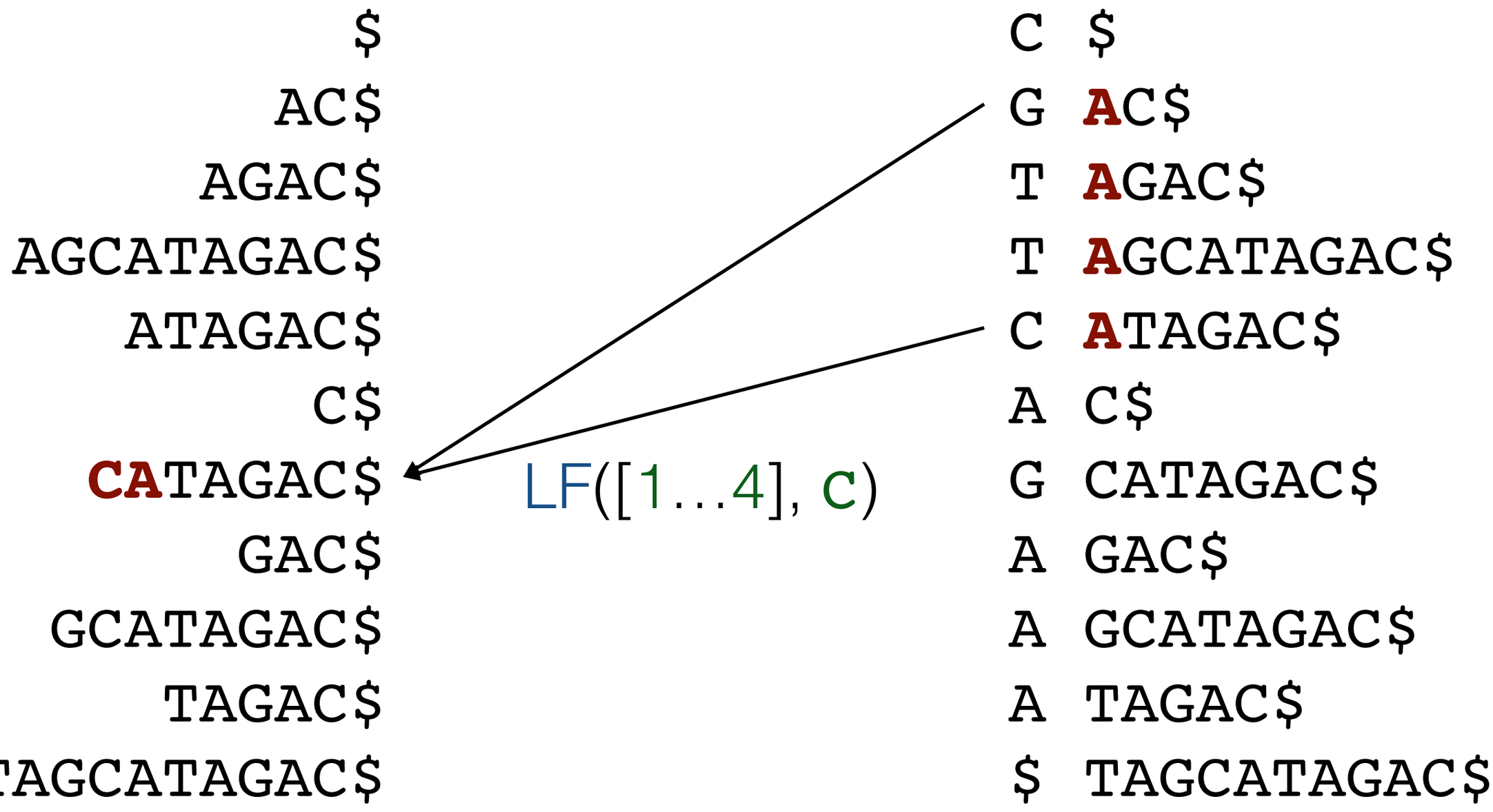
Insert **C** to the beginning

$	C	$	C
AC$	G	AC$	G
AGAC$	T	AGAC$	T
AGCATAGAC$	T	AGCATAGAC$	T
ATAGAC$	C	ATAGAC$	C
C$	A	C$	A
CATAGAC$	G	CATAGAC$	G
		CTAGCATAGAC$	**$**
GAC$	A	GAC$	A
GCATAGAC$	A	GCATAGAC$	A
TAGAC$	A	TAGAC$	A
TAGCATAGAC$	$	TAGCATAGAC$	**C**

2. Insert **$** after LF(i, **C**) suffixes.

1. Replace the **$** at position i with the inserted **C**.

Backward searching



LF([sp…ep], c) = [LF(sp, c)…LF(ep+1, c) −1]

Locating the occurrences

```
                $
               AC$
              AGAC$
           AGCATAGAC$
            ATAGAC$
                C$
              CATAGAC$
          GCATAGAC$
             TAGAC$
        TAGCATAGAC$
```

LF(4, C)

```
C $
G AC$
T AGAC$
T AGCATAGAC$
C ATAGAC$
```

Text position 4

```
G CATAGAC$
A GAC$
A GCATAGAC$
A TAGAC$
$ TAGCATAGAC$
```

Text position 3

SA[LF(i, BWT[i])] = SA[i] – 1

FM-index

- find() returns the **range of suffixes** matching the pattern: [sp_{cX}…ep_{cX}] = LF([sp_X…ep_X], c).

- locate() stores SA[i] for some offsets i and iterates LF(i, BWT[i]) to determine the **text position** SA[i] for other offsets.

- We can modify locate() to return **sequence identifiers**.

- extract() iterates LF(i, BWT[i]) and produces the **string** backwards, getting the previous character from BWT[i].

Bidirectional FM-index

- Pattern matching proceeds backwards.
- If we build the index for the reverse of the text, we can search forward instead.
- If we have the ranges for X in the forward index and reverse(X) in the reverse index, we can (Lam, 2009):
  - update them to cX and reverse(cX) by searching in the forward index; and
  - update them to Xc and reverse(Xc) by searching in the reverse index.

FMD-index

- In bioinformatics, the text and/or the patterns are often a mix of forward and reverse complement orientations.

- We can simplify the situation by indexing the text in both orientations in the same FM-index (Li, 2012).

- By using the same techniques as in the bidirectional index, we can

  - search for both orientations of the pattern in both orientations of the text; and

  - support bidirectional search.

Positional BWT

Haplotype storage

Sample 1 `0000100011100100110001101101010 1`

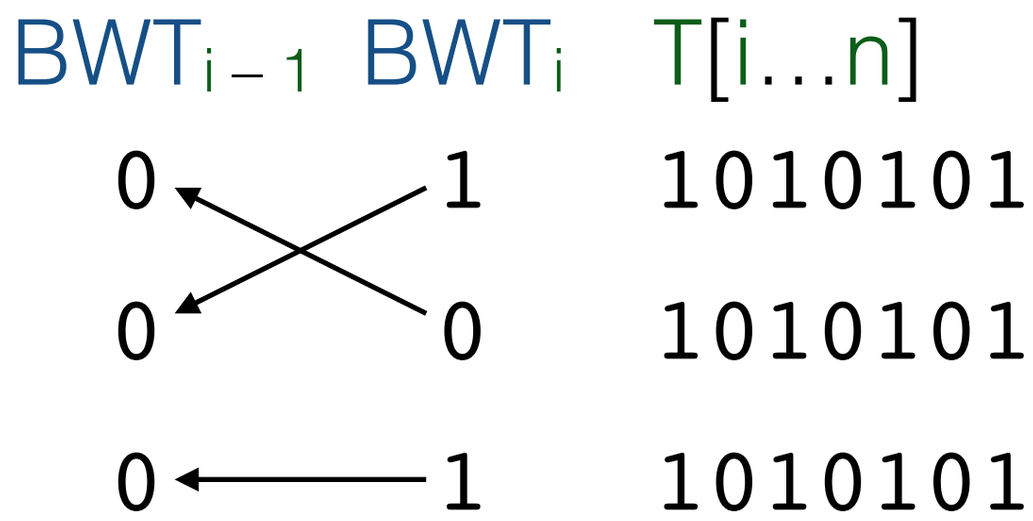
Sample 2 `0000100101100100110001100101010 1`

Sample 3 `0000100101100100110001101101010 1`

- We can encode haplotypes as binary sequences over variant sites, with 0 for the reference allele and 1 for the alternate allele.
- If we index the sequences in a normal substring index, we may have **false positives** when the pattern occurs in other positions.

Positional BWT

- We can avoid false positives by storing the subsequence of the BWT corresponding to each position separately.
- At each position i, the suffixes are sorted in lexicographic order.
- LF-mapping in (stable sorting of) BWT_i reorders the suffixes from position i to position i – 1.
- Durbin: Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). Bioinformatics, 2014.



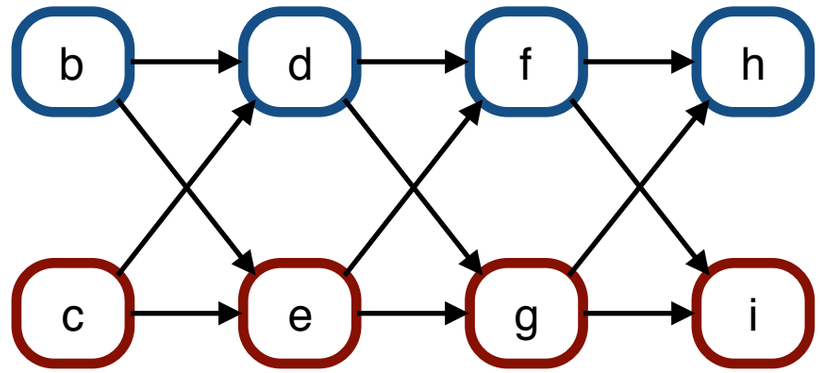
Because the collection of sequences is repetitive, run-length encoding compresses the BWTs well.

Graph extension

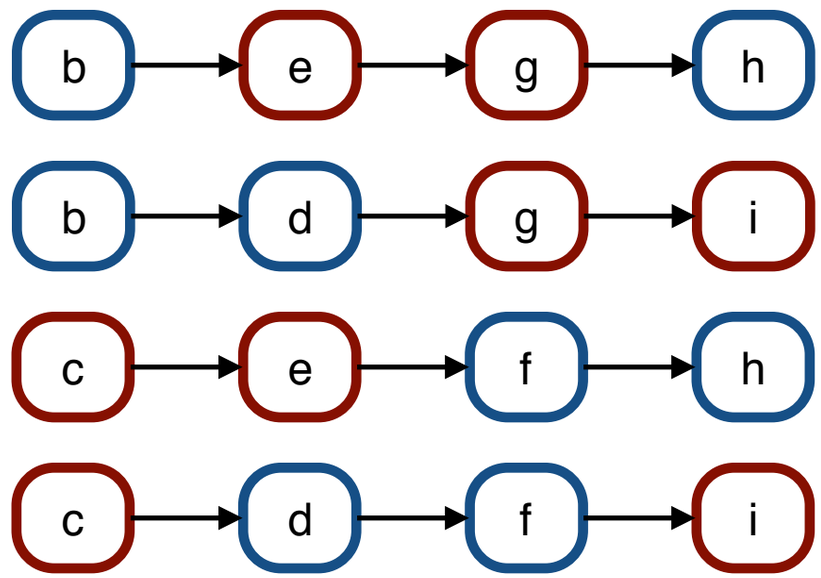
- There is an extension of the PBWT that stores haplotypes as paths in a graph.
- The extension is specific to VG graphs, making it quite complicated.
- The structure itself is general, but the construction algorithm only works with DAGs.
- Novak, Garrison, Paten: A graph extension of the positional Burrows–Wheeler transform and its applications. Algorithms for Molecular Biology, 2017.

Graph interpretation

- PBWT partitions the BWT by positions. We can achieve the same effect by using a separate alphabet for each position.
- This is equivalent to a multi-string BWT of node sequences in a graph with two nodes per position.



- We use the graph as a first-order model: if the current node is d, the next node is f or g, and one bit is enough to encode it.
- The generalization to arbitrary graphs is obvious.
- Because we have a normal multi-string BWT, we use any techniques that work with FM-indexes.



Graph BWT

Some assumptions

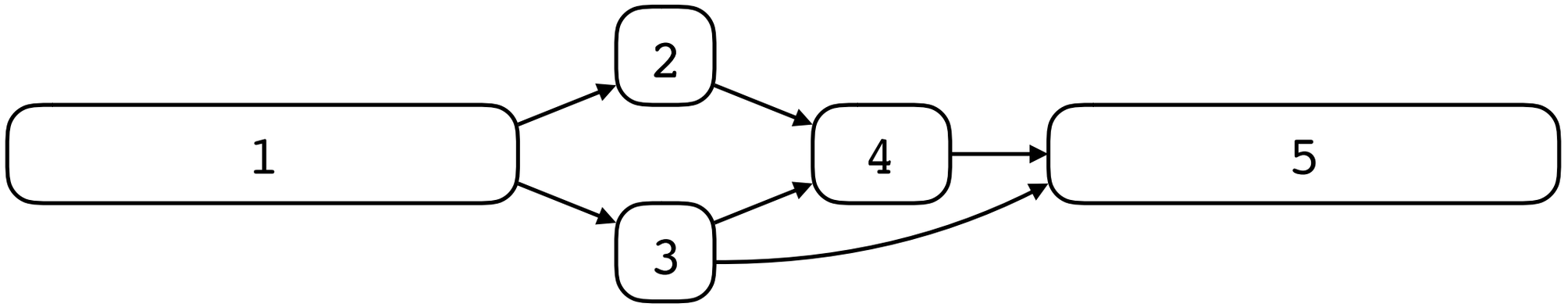
- We have a strings over a large alphabet (e.g. hundreds of millions of nodes in whole-genome human graph) with low first-order entropy.

- The number of occurrences of almost every character is proportional to the number of samples.

- While there may be cycles, the graph is still mostly linear and topologically sorted.

- We index reverse paths, as it is more intuitive to have LF-mapping following the direction of the edges.

Records



- We have a separate record for each node (each character in the alphabet).
- The header stores the outgoing edges and the rank information needed for LF-mapping.
- The body stores the part of the BWT corresponding to the prefixes ending with the current node.

Node 3
Outdegree 2 0: node 4, offset 5 1: node 5, offset 0
0000111100

FM-index $LF(i, c) = C[c] + BWT.rank(i, c)$

GBWT $LF((u, i), v) = (v, u.offset(v) + u.rank(i, u.edge(v)))$

Some consequences

- Query performance depends on the size of the local alphabet, not on the global alphabet.
- As the graph is (almost) topologically sorted, search tends to scan the BWT linearly instead of jumping around randomly.
- Because the rank structure is local, memory access takes almost constant time regardless of text size.
- We might even use a memory-mapped file for BWT.

Compressed GBWT

- Records are concatenated in a byte array, with an sd_vector (Okanohara & Sadakane, 2009) as an index.

- Header fields are stored using a 7+1 bit encoding.

- The body is run-length encoded. Short runs fit in one byte, while long runs use the 7+1 bit encoding for the remaining run length.

- The sequence identifiers (for locate()) are stored in a global structure, as some records would otherwise have too many identifiers to scan quickly through.

Dynamic GBWT

- The construction uses a dynamic FM-index, because we do not want to store the sequences explicitly.
- We rewrite a record every time we modify it.
- Speed is more important than compression, as run-length encoding already compresses the BWT well.
- I wrote a quick placeholder implementation, which turned out to be good enough…

A good enough dynamic FM-index

```
struct DynamicRecord
{
  size_type                    body_size;
  std::vector<edge_type>       incoming, outgoing;
  std::vector<run_type>        body;
  std::vector<sample_type> ids;
};

class DynamicGBWT
{
  GBWTHeader                   header;
  std::vector<DynamicRecord> bwt;
};
```

GBWT construction

- An incremental algorithm based on BCR (Bauer, Cox & Rosone, 2013) and RopeBWT2 (Li, 2014).

- Inserts a batch of sequences by extending each of them by one character at a time.

- Larger batches use more memory but reduce the total number of record rewrites.

- Some buffering is usually required, as we generate the sequences one variant site at a time, not one node at a time.

Construction in VG

```
for each batch of 200 samples
    for each site in VCF
        for each sample in batch
            for each phase in sample
                if not S[phase].extend(site)
                    GBWT.insert(S[phase])
                    S[phase].clear()
                    S[phase].extend(site)
    for each sample in batch
        for each phase in sample
            GBWT.insert(S[phase])
            S[phase].clear()
```

- Separate process for each chromosome.
- Memory usage ~1 GB / 10 Mbp with 1000GP data.
- Insert the sequences in both orientations.
- Sequences are buffered (size 100 million) and inserted in a background thread.
- Sequence generation is the bottleneck: the construction thread is running only 50% of the time.

Benchmarks

Construction

- 1000GP data: 2504 samples, 88 million variants.
- 5008 haplotypes, 29.3 million sequences.
- Text size 2.19 trillion, alphabet size 612 million.
- AWS r3.8xlarge instance: 32 cores, 244 GB memory.
- 12 parallel jobs for 24 chromosomes.
- Store sequence ids for one out of 1024 positions.
- Final merging is fast, as node ids do not overlap.

Chromosome	Time (h)	GBWT (MB)	locate() (MB)	Total (MB)
1	29.44	658	711	1369
2	**32.20**	685	744	1429
3	27.07	562	596	1158
4	26.15	553	603	1156
5	24.17	504	543	1046
6	23.18	487	516	1003
7	21.80	476	485	962
8	21.44	466	463	929
9	17.31	391	387	778
10	18.05	420	415	835
11	18.49	416	416	833
12	17.40	407	405	812
13	13.35	309	304	613
14	12.19	293	284	577
15	11.27	277	265	542
16	12.39	291	270	561
17	10.30	255	235	490
18	10.29	239	228	467
19	8.16	202	183	385
20	8.18	193	180	373
21	4.99	132	120	252
22	4.78	133	122	256
X	14.46	396	326	722
Y	0.19	42	12	54
Total	**34.32**	**8788**	**8814**	**17602**
Merged	**0.16**	**8849**	**9886**	**18736**

Query performance

- find() speed starts at 2M characters/second and increases to 4–5M as pattern length increases.

- GCSA2: 3M; BWA: 4–5M; but these are bases instead of nodes.

- locate() speed is 6k positions/second for individual positions and 50k for ranges of positions.

- GCSA2: 200–300k; BWA: 200–400k.

- Gagie, Navarro, and Prezza have a faster locate() structure in SODA 2018, but we cannot build it yet.

Conclusions

Conclusions

- GBWT is an FM-index for repetitive collections of paths in low-degree graphs.
- It is practical with (at least) thousands of human haplotypes.
- find() speed is comparable to fast FM-indexes.
- locate() speed is 1–2 orders of magnitude slower.
- Joint work with Richard Durbin, Erik Garrison, Adam M. Novak, and Benedict Paten.