Genome Graphs and BWT-based Data Structures

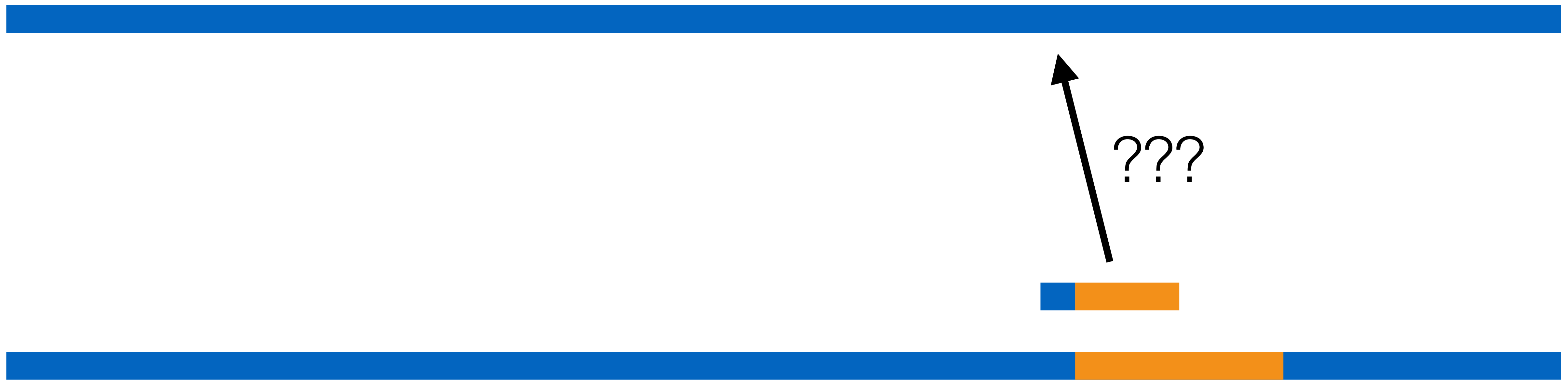
Jouni Sirén
UCSC Genomics Institute

Variation graph toolkit VG

- Most of this talk is based on my work on the VG toolkit (Garrison et al, 2018), available at https://github.com/vgteam/vg.
- In addition to the published work, the codebase contains prototype implementations of many genome graph algorithms, data structures, and workflows.
- We are in the process of moving the good parts into reusable modules outside the main VG codebase.

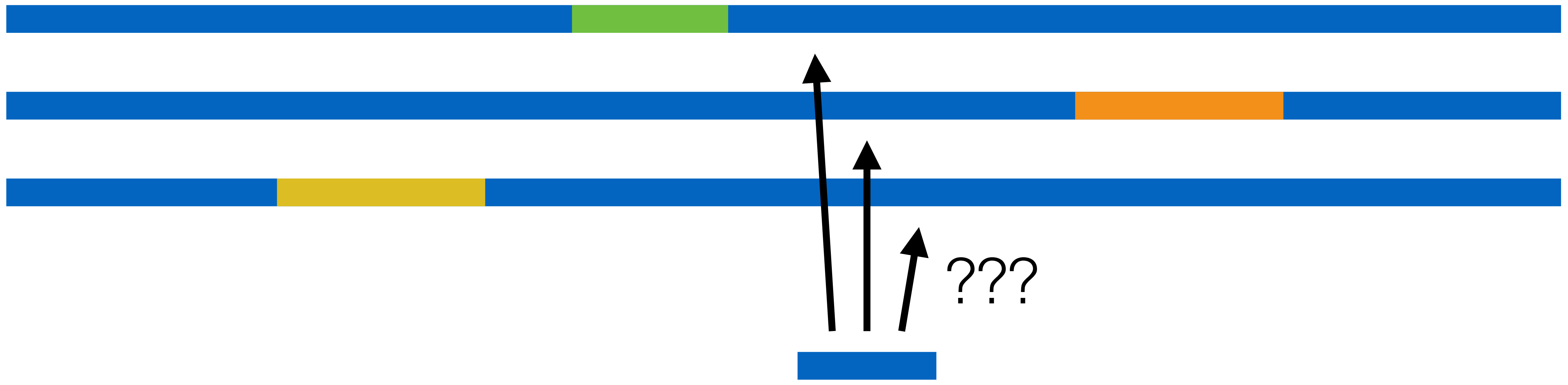
Why genome graphs?

Reference bias



- Reference sequences are easy to work with.
- When the sequenced sample diverges from the reference, using the reference may bias our results.

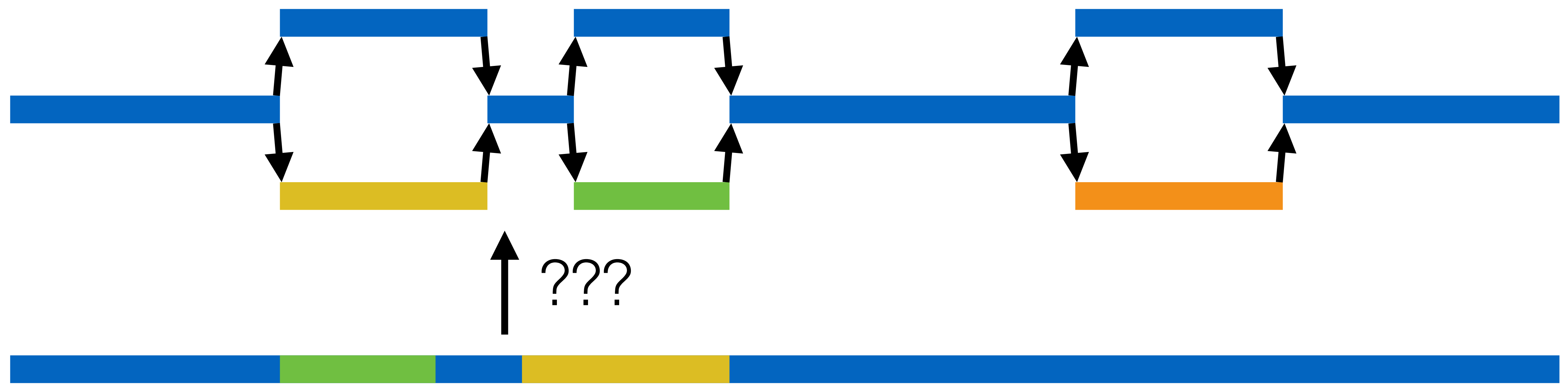
Collection of haplotypes



- We can try to reduce the reference bias by using a collection of haplotypes as the reference.

- Multiple hits: Same position in several haplotypes (useful) or several different positions (less useful)?

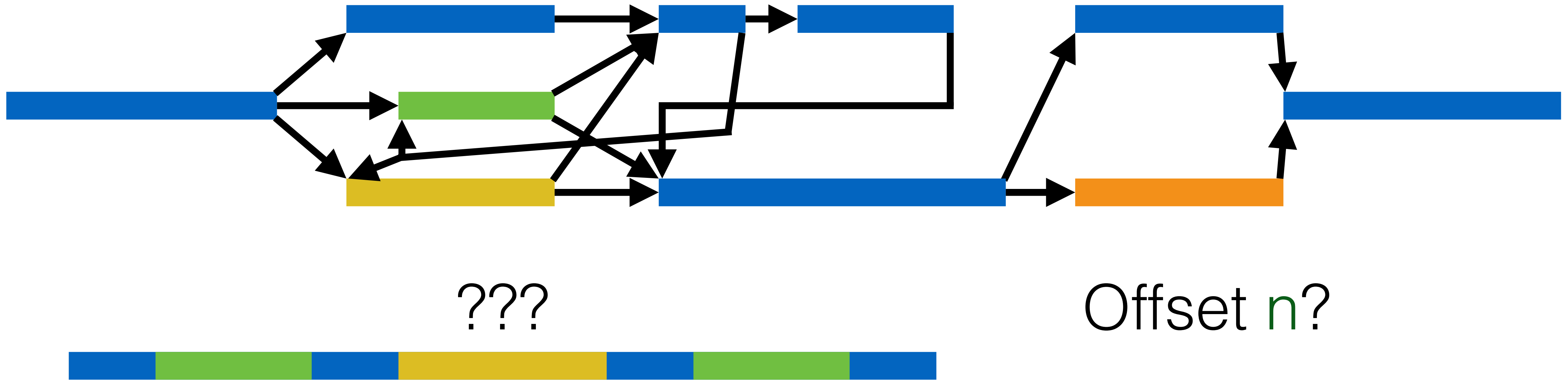
Global alignment / DAG



- A global alignment helps with reads mapping to multiple haplotypes. If we collapse shared regions, we get a directed acyclic graph.

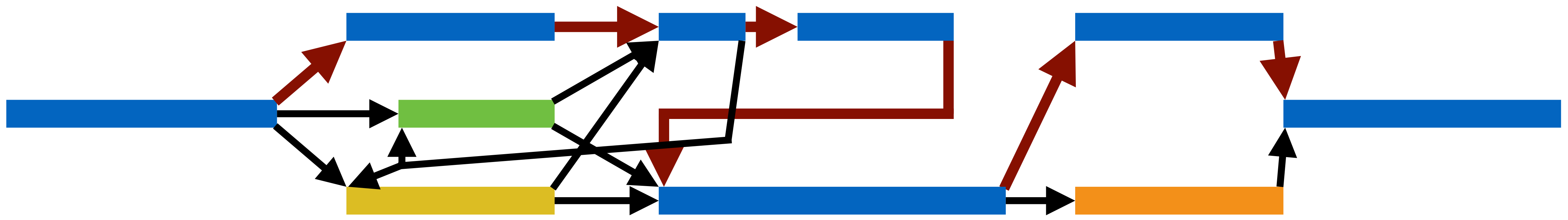
- How to deal with structural variation?

Local alignments



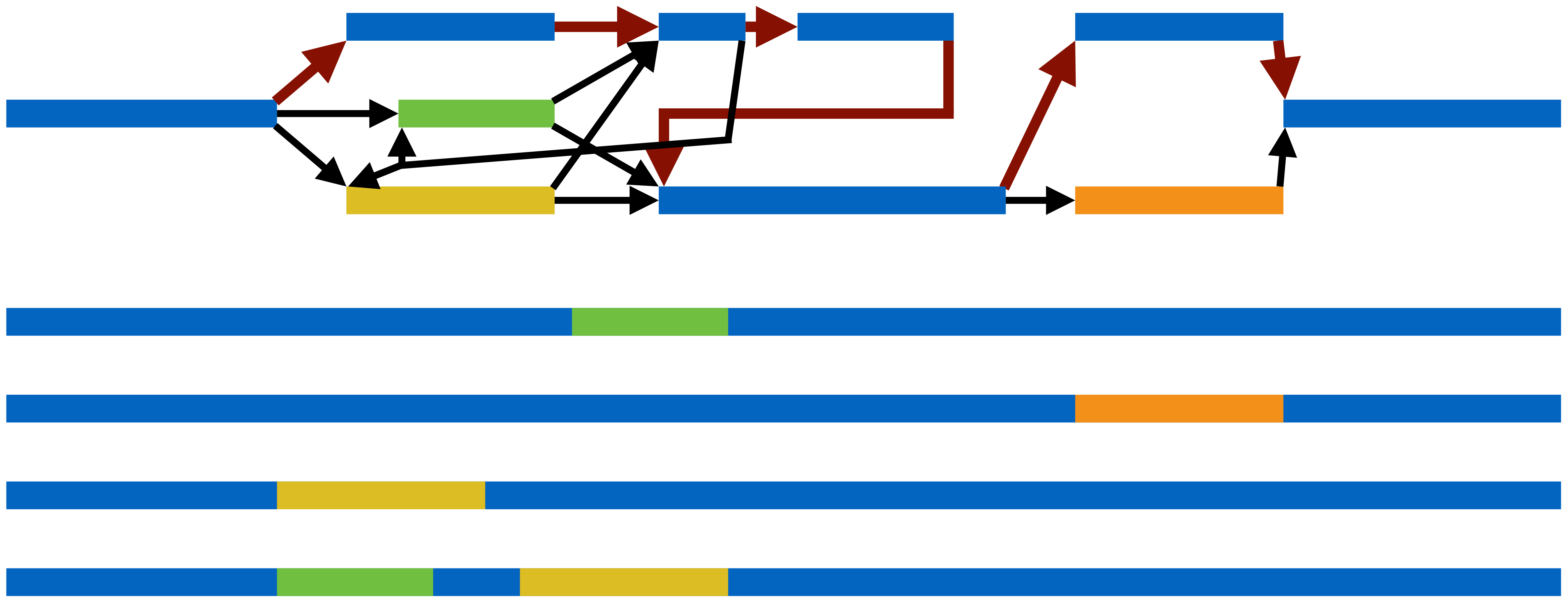
- If we use local alignments instead, we get assembly graphs that can handle structural variation.
- They contain nonsensical paths and lack a global coordinate system.

Graph + path



- A **primary path** can provide a coordinate system.
- We still cannot deal with structural variation in DAGs or with nonsensical paths in assembly graphs.
- This was the initial VG model.

Graph + path + haplotypes



Graph: These positions are equivalent.

Haplotypes: These paths make sense.

Indexing graphs

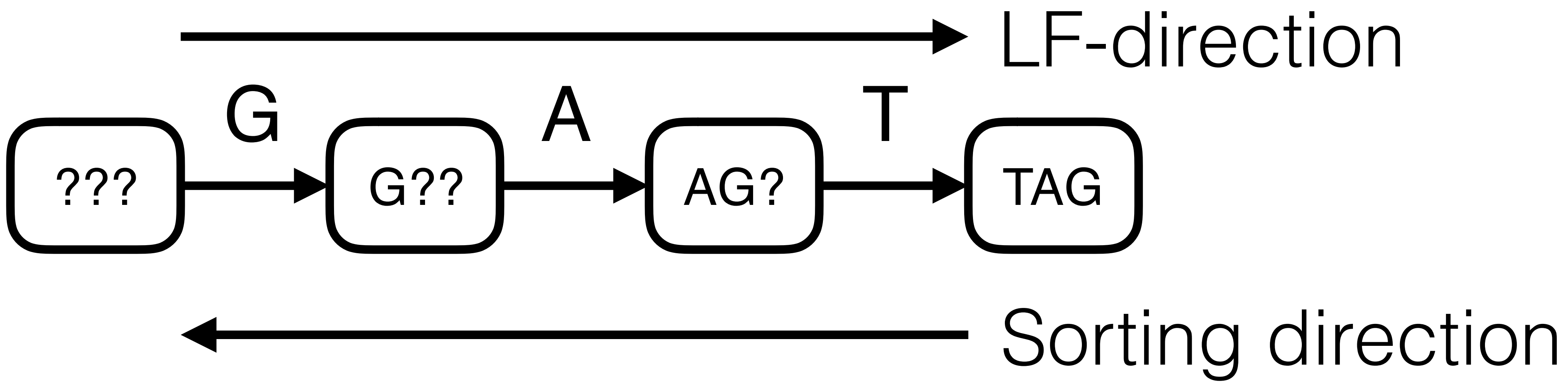
Wheeler graphs

- Wheeler graphs (Gagie et al, 2017) are edge-labeled directed graphs, where the nodes are ordered by a generalization of the lexicographic order.

- Node rank is determined by sorting by:
  1. Incoming edge labels (the first character)
  2. Predecessor node ranks (the following suffix)

- Useful subclass (generalizes de Bruijn graphs):
  – Nodes are a prefix-free set of strings.
  – Node order is the lexicographic order of the strings.
  – Path labels start with the string corresponding to the initial node (in the sorting direction).

LF/sorting directions



- **LF-direction:** LF-mapping moves **forward**; node order is based on **reverse prefixes**; and locate() returns the **endpoint** of the match.

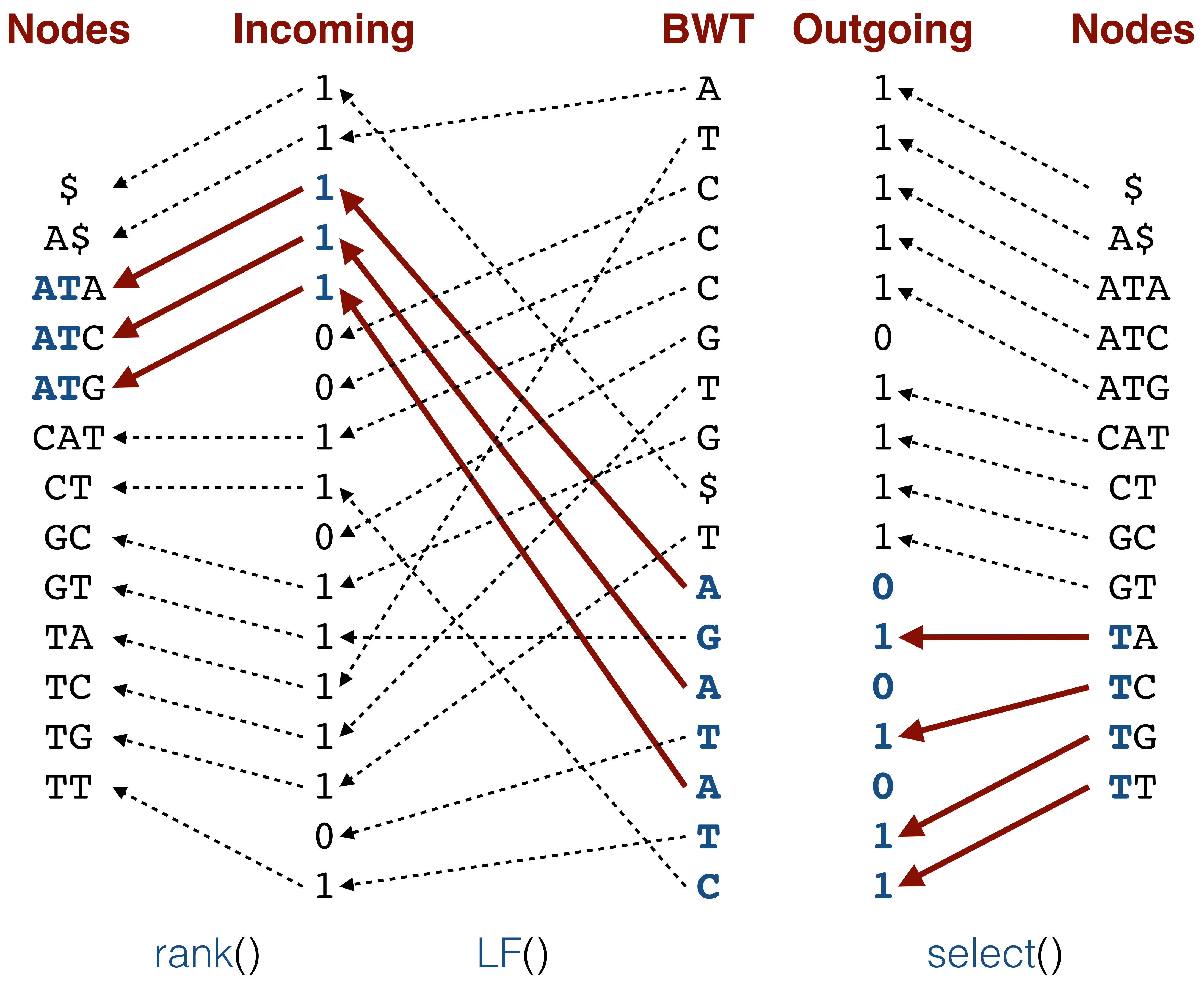
- **Sorting direction:** LF-mapping moves **backward**; node order is based on **suffixes**; and locate() returns the **starting point** of the match.

Indexing Wheeler graphs

- As the node order is based to the lexicographic order, we can use a generalization of the FM-index.

- One search step (in LF-direction):
  – Map the range of nodes into a range of outgoing edges using select() queries on a bitvector.
  – Edge labels form the BWT. Transform the range of outgoing edges into a range of incoming edges using LF-mapping.
  – Map the range of incoming edges into a range of nodes using rank() queries on a bitvector.

- Based on GCSA (Sirén et al, 2014) and the succinct de Bruijn graph (Bowe et al, 2012).


Nodes
Incoming
BWT
Outgoing
Nodes
$
A$
ATA
ATC
ATG
CAT
CT
GC
GT
TA
TC
TG
TT
1
1
1
1
1
0
0
1
1
0
1
1
1
1
1
0
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
$
A$
ATA
ATC
ATG
CAT
CT
GC
GT
TA
TC
TG
TT
rank()
LF()
select()

Faster searching

- If the Wheeler graph is **deterministic**, we can avoid the select() queries by using **indicator bitvectors**.
- $B_c[i] = 1$, if the node with rank i has an outgoing edge with label c.
- LF-mapping is just two rank() queries on a bitvector, making the index almost **as fast as any FM-index**.
- **GCSA2** (Sirén, 2017) can find **MEMs** between short reads and a 1000GP graph at **3 Mbp/s** and locate **200,000 occurrences/second**.

More functionality

- Assume that the nodes of the Wheeler graph are a prefix-free **set of strings**.
- We can use **CST techniques** to represent the **trie** of the strings.
- shorter() and longer() in the **variable-order de Bruijn graph** (Boucher et al, 2015).
- parent(), depth(), and count() in **GCSA2**.

Graph transformations

Indexing general graphs

- We want to index alignment graphs, but we can only index Wheeler graphs. The intersection of these two classes consists of de Bruijn graphs.

- In order to index a general graph, we must transform into an (almost) equivalent Wheeler graph.

- As we want to align reads to the original graph, we index the transformed graph but make the index map to the original graph.

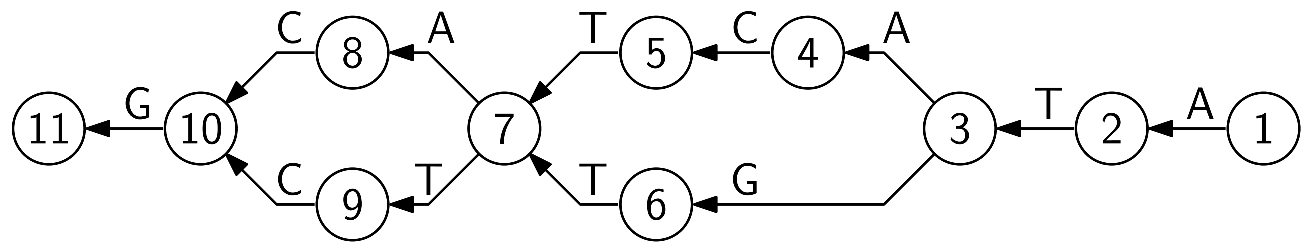
Transforming DAGs

- We can transform a DAG into an equivalent (but potentially much larger) Wheeler graph using prefix-doubling.
- The nodes of intermediate graphs correspond to paths of length k in the original graph.
- Prefix-doubling: Extend paths of length k into paths of length 2k. If all paths in a lexicographic range start from the same original node, merge them.
- Used in GCSA.

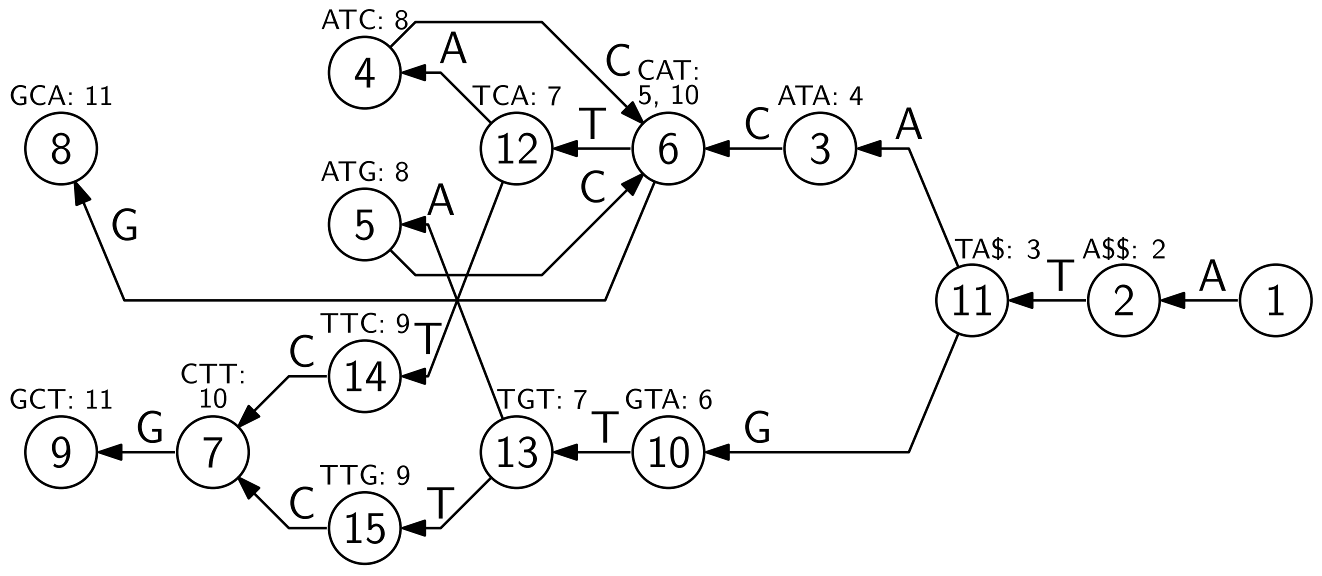
Approximating general graphs

- Graphs with cycles may not have equivalent Wheeler graphs.
- If we stop the prefix-doubling at length k and merge only ranges corresponding to a shared prefix, the graph is equivalent to an order-k de Bruijn graph.
- All original paths exist in the Wheeler graph, and all Wheeler graph paths of length $\leq$ k exist in the original graph.
- Used in GCSA2.

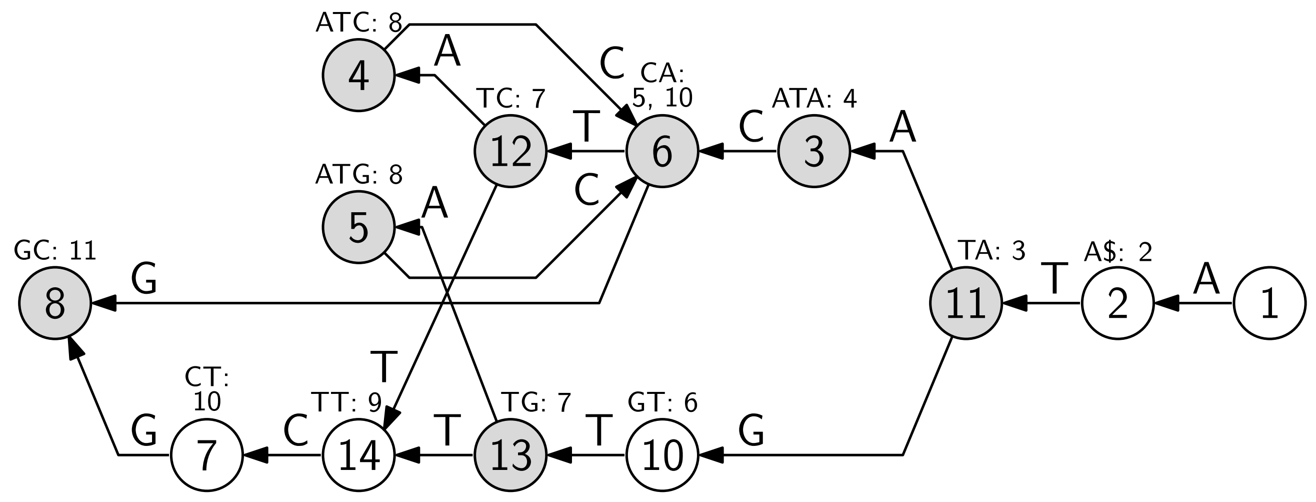
Original graph



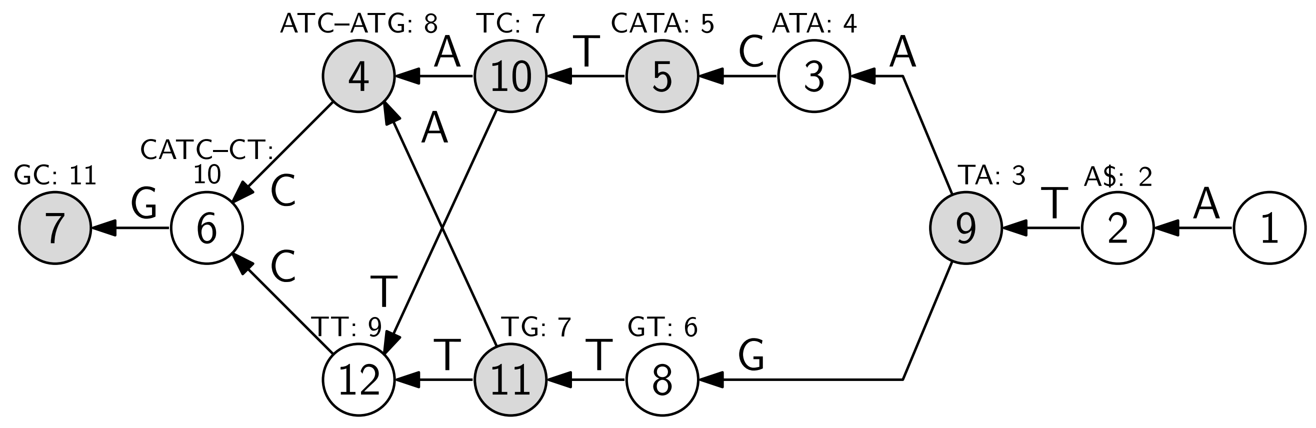
Order-3 de Bruijn graph



Order-3 pruned de Bruijn graph (GCSA2)



Prefix-range-sorted graph (GCSA)

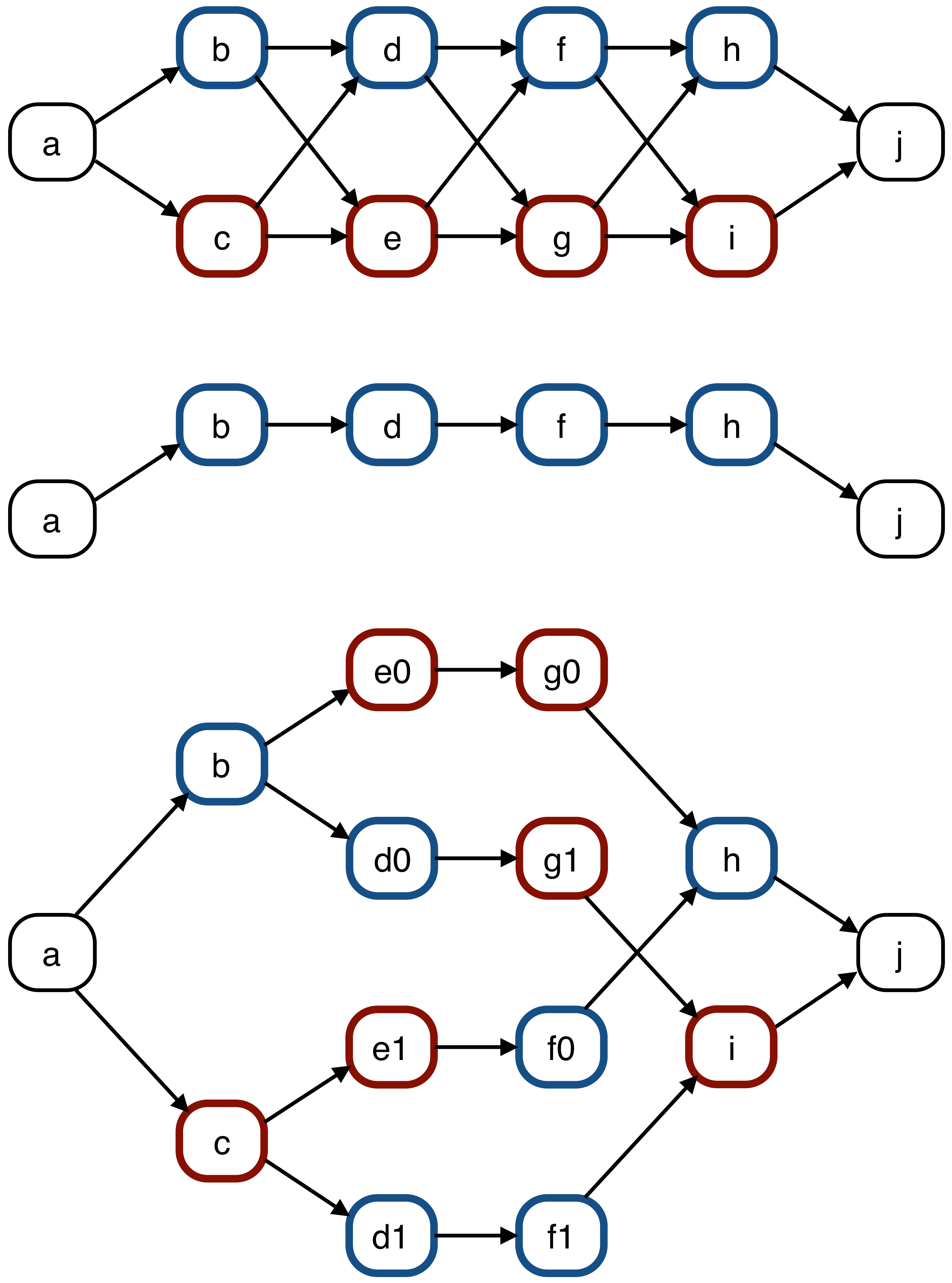


Graph simplification

Complex graph regions cannot be indexed using Wheeler graph-based methods, because they contain too many paths of length k.

VG removes regions with too many paths in a short window and replaces them with the reference sequence.

If we have the original haplotypes, we can unfold them in the complex region (Sirén et al, 2018).



GBWT

Are FM-indexes too slow?

- Iterated LF-mapping jumps randomly around the BWT. We usually get cache misses for each character of the pattern.
- Once the pattern is unique, it should be faster to extend it in the graph than in the index.
- Do we need an FM-index if we only match short patterns?
- Minimizer indexes (sparse k-mer indexes) are 10x faster in 2x space.

GBWT

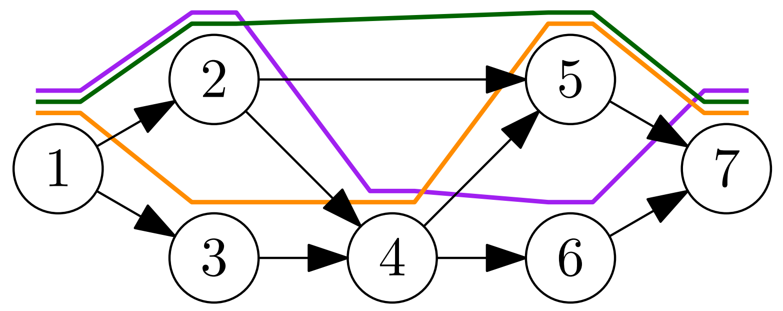
- GBWT (Sirén et al, 2018) is the haplotype index used in VG. It is based on the graph extension (Novak et al, 2017) of the PBWT (Durbin, 2014).

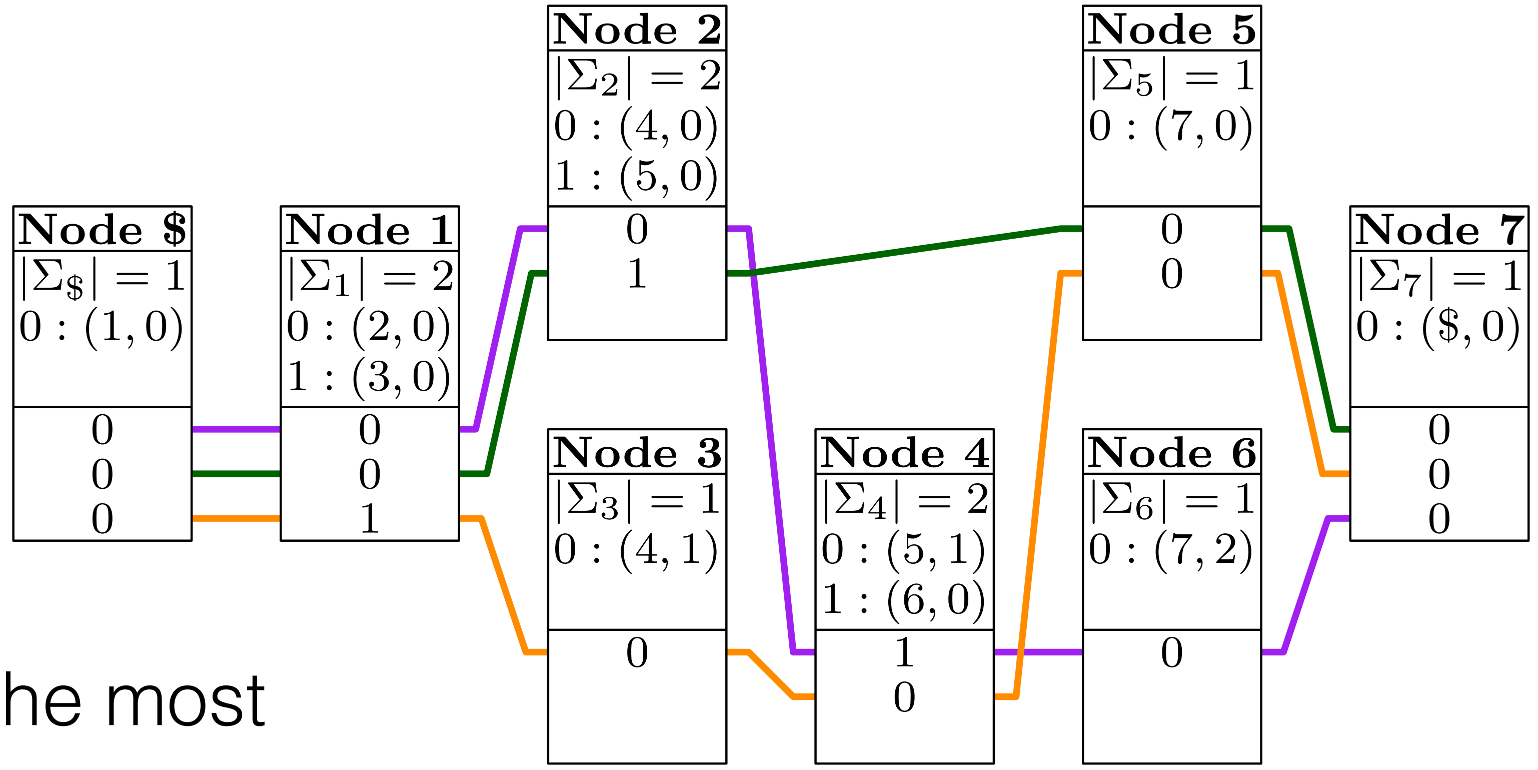
- We represent the haplotypes as paths in the graph and store the node sequences in RLBWT.

- Index construction is straightforward at 1000GP scale (5,000 human haplotypes, $n \approx 2^{41}$).

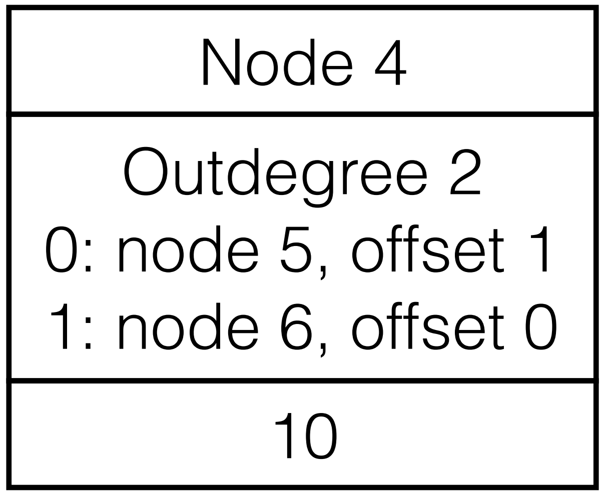
- Indexing 100x larger datasets ($n \approx 2^{48}$) is feasible but expensive.

GBWT details





- We partition the BWT by the most significant character.
- Each node contains the corresponding part of the BWT and a local rank() structure.
- If the graph layout is cache-friendly, iterated LF-mapping is also cache-friendly.
- One iteration of LF-mapping per node vs per character.



Node 4
Outdegree 2 0: node 5, offset 1 1: node 6, offset 0
10

GBWT construction

- Basic construction is like in RopeBWT2 (Li, 2014): We insert a batch of paths into a dynamic FM-index using the BCR algorithm (Bauer et al, 2013).

- When the basic algorithm is too slow, we can build partial indexes in parallel and merge them using BWT-merge (Sirén, 2016). (This is unnecessary at 1000GP scale.)

- Different chromosomes use different node ids, so we can index them in parallel and merge the indexes by concatenating the BWTs.

GBWT benchmarks

AWS i3.8xlarge instance: 16 physical / 32 logical CPU cores, 244 GiB memory.

1000GP haplotypes: 240,232 paths of total length 2.19 trillion nodes in a graph with 612 million nodes.

Index construction: 17 hours.

Index size: 8.43 GiB for bidirectional GBWT, 8.17 GiB for DA samples ($d = 1024$).

Bidirectional search: 2 million nodes/second (short patterns), 4 million nodes/second (long patterns).

Some GBWT applications

- Haplotype unfolding for GCSA2 construction.
- Minimizer index construction: 10 minutes for 1000GP haplotypes (>30 hours with GCSA2).
- Gapless seed extension: Illumina sequencing errors are mostly substitutions, and most real indels are already in the haplotypes.

Faster document listing?

- With the default **DA sample rate** 1024, GBWT can list the **matching haplotypes** at 10,000 (single positions) to 100,000 (ranges of positions) hits/second.
- It would be nice to use the fast locate() structure from the **r-index** (Gagie et al, 2018).
- Can we maintain the r-index locate() structure when **inserting/deleting** strings and **merging** indexes?