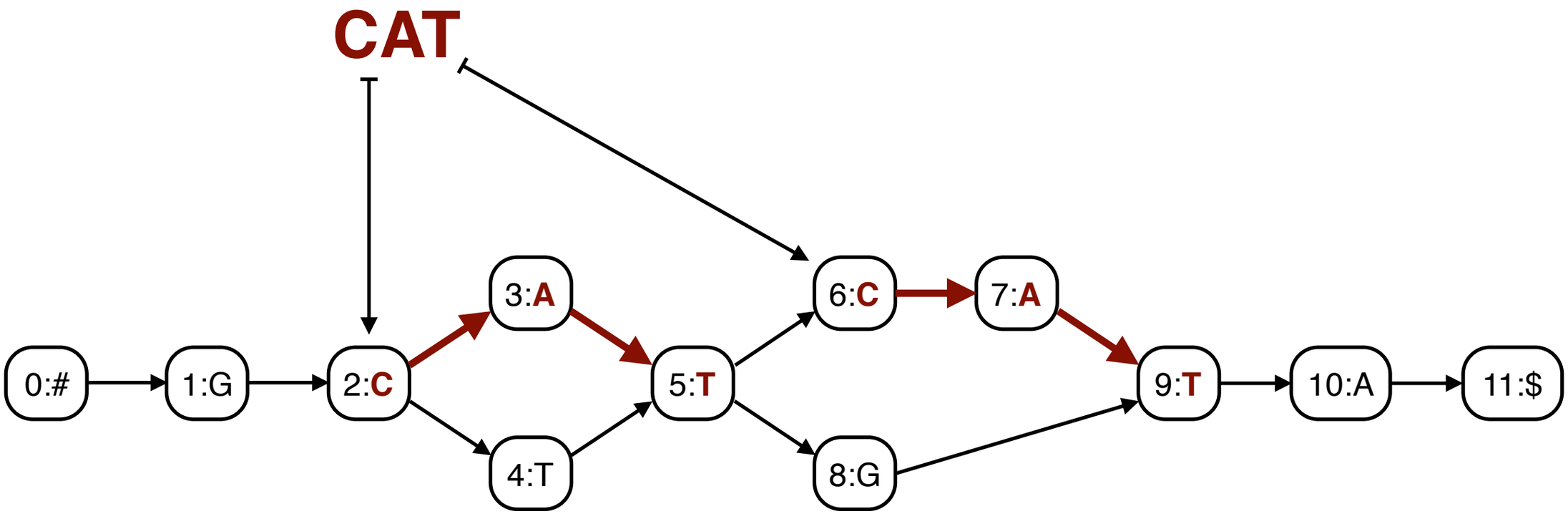
# Indexing Variation Graphs

Jouni Sirén
Wellcome Trust Sanger Institute




- Path indexes are text indexes for the path labels in a graph.
- The index finds the start nodes of the paths labeled by the query string.
- Indexing the paths themselves is not cost-effective.

# Trade-offs

- The number of kmers in a graph increases exponentially with k.

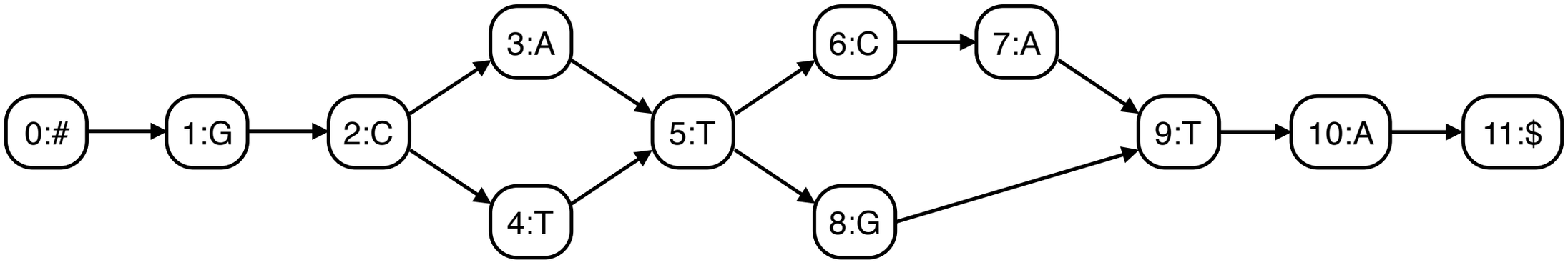
- k should be larger than the expected length of maximal exact matches.

- In one human variation graph, the number of kmers is $1.031^k \cdot 2.348$ billion, or 116 billion for $k = 128$.

- The design of a path index is a trade-off between index size, query performance, maximum query length, and ignoring complex regions of the graph.

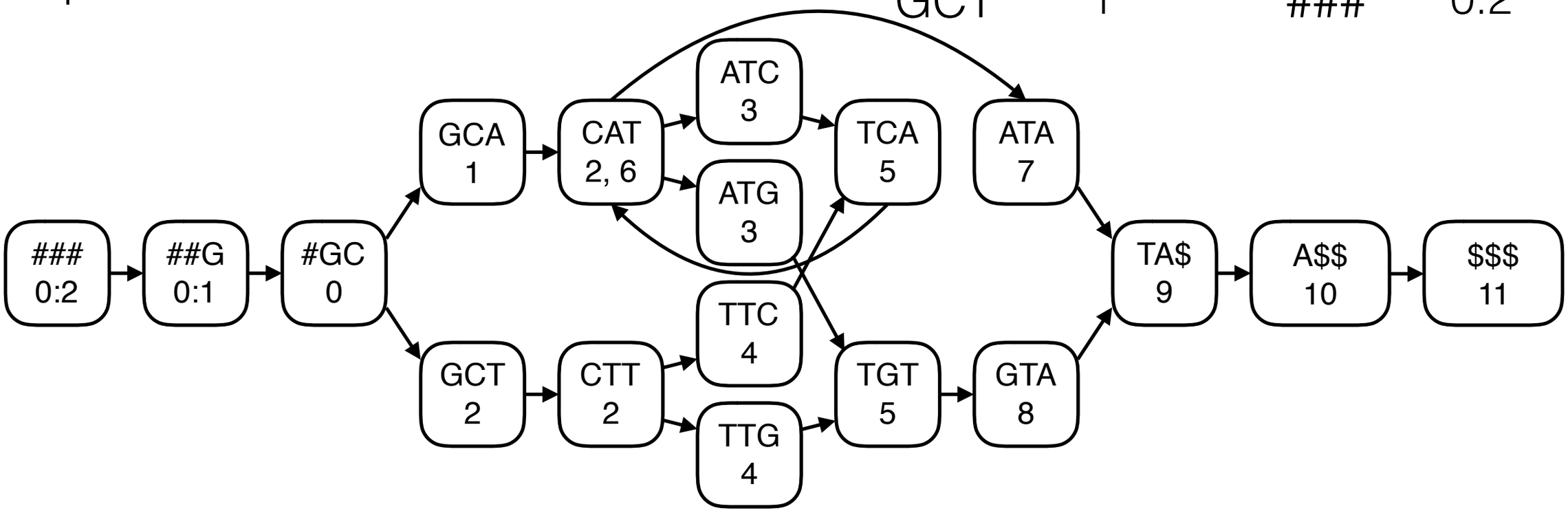
- The kmer index is a simple path index. It consists of a set of key-value pairs.
- A hash table supports fast kmer queries.
- Binary search in a sorted array is slower but supports queries shorter than k.
- Index size: terabytes.

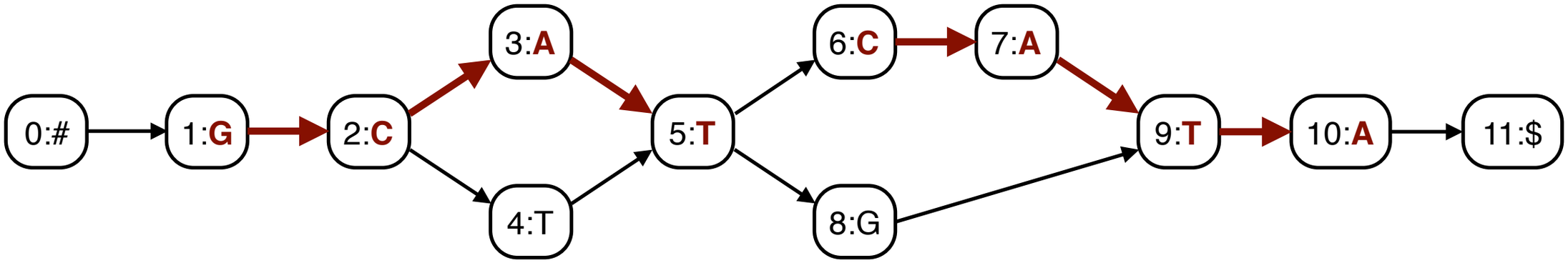
| Key | Value | Key | Value |
|---|---|---|---|
| $$$ | 11 | GTA | 8 |
| A$$ | 10 | TA$ | 9 |
| ATA | 7 | TCA | 5 |
| ATC | 3 | TGT | 5 |
| ATG | 3 | TTC | 4 |
| CAT | 2, 6 | TTG | 4 |
| CTT | 2 | #GC | 0 |
| GCA | 1 | ##G | 0:1 |
| GCT | 1 | ### | 0:2 |



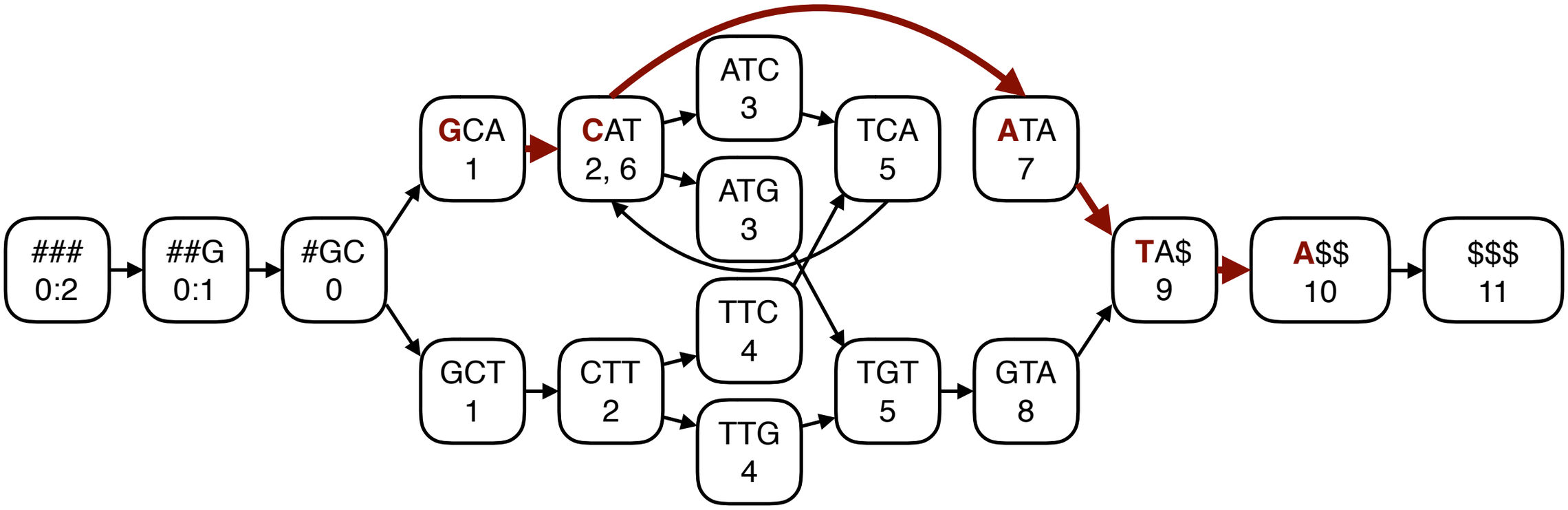
- We can represent the kmer index as a de Bruijn graph.
- We label each node with the first character of the key.
- The de Bruijn graph approximates the input graph. There are no false negatives, and no false positives shorter than k.

| Key | Value | Key | Value |
|---|---|---|---|
| $$$ | 11 | GTA | 8 |
| A$$ | 10 | TA$ | 9 |
| ATA | 7 | TCA | 5 |
| ATC | 3 | TGT | 5 |
| ATG | 3 | TTC | 4 |
| CAT | 2, 6 | TTG | 4 |
| CTT | 2 | #GC | 0 |
| GCA | 1 | ##G | 0:1 |
| GCT | 1 | ### | 0:2 |






Long paths may be false positives, but we can verify them in the input graph.

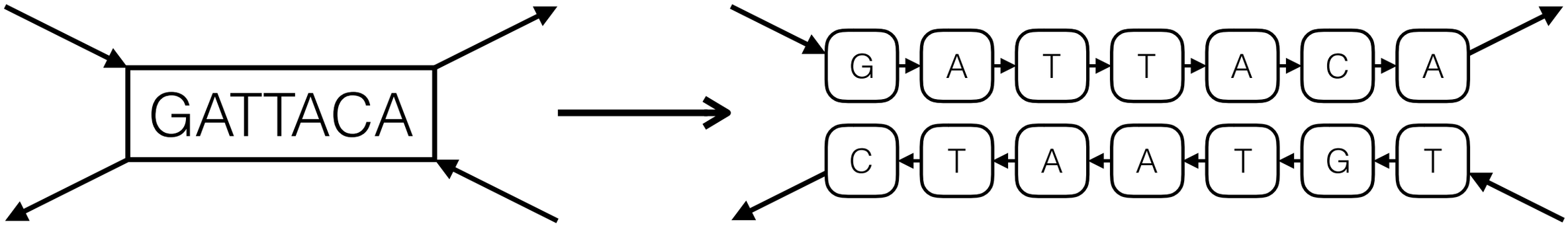


# Succinct de Bruijn graphs

| Node | BWT | IN | OUT |
|---|---|---|---|
| $$$ | A | 1 | 1 |
| A$$ | T | 1 | 1 |
| ATA | C | 1 | 1 |
| ATC | C | 1 | 1 |
| ATG | C | 1 | 1 |
| CAT | GT | 01 | 001 |
| CTT | G | 1 | 01 |
| GCA | # | 1 | 1 |
| GCT | # | 1 | 1 |
| GTA | T | 1 | 1 |
| TA$ | AG | 01 | 1 |
| TCA | AT | 01 | 1 |
| TGT | AT | 01 | 1 |
| TTC | C | 1 | 1 |
| TTG | C | 1 | 1 |
| #GC | # | 1 | 01 |
| ##G | # | 1 | 1 |
| ### | $ | 1 | 1 |

- Sort the nodes, write the predecessor labels to BWT, and encode the indegrees and the outdegrees in unary to bitvectors IN and OUT.
- The result is an FM-index for de Bruijn graphs.
- Bowe et al: Succinct de Bruijn graphs. WABI 2012.
- Index size: hundreds of gigabytes.

# GCSA2




1. Transform the original graph into a directed input graph with character labels.

2. Apply other transformations, such as determinization, pruning, or expanding haplotype paths.

3. Build a pruned de Bruijn graph for the path labels in the transformed input graph.

4. Use the pruned de Bruijn graph as an index of the original graph.

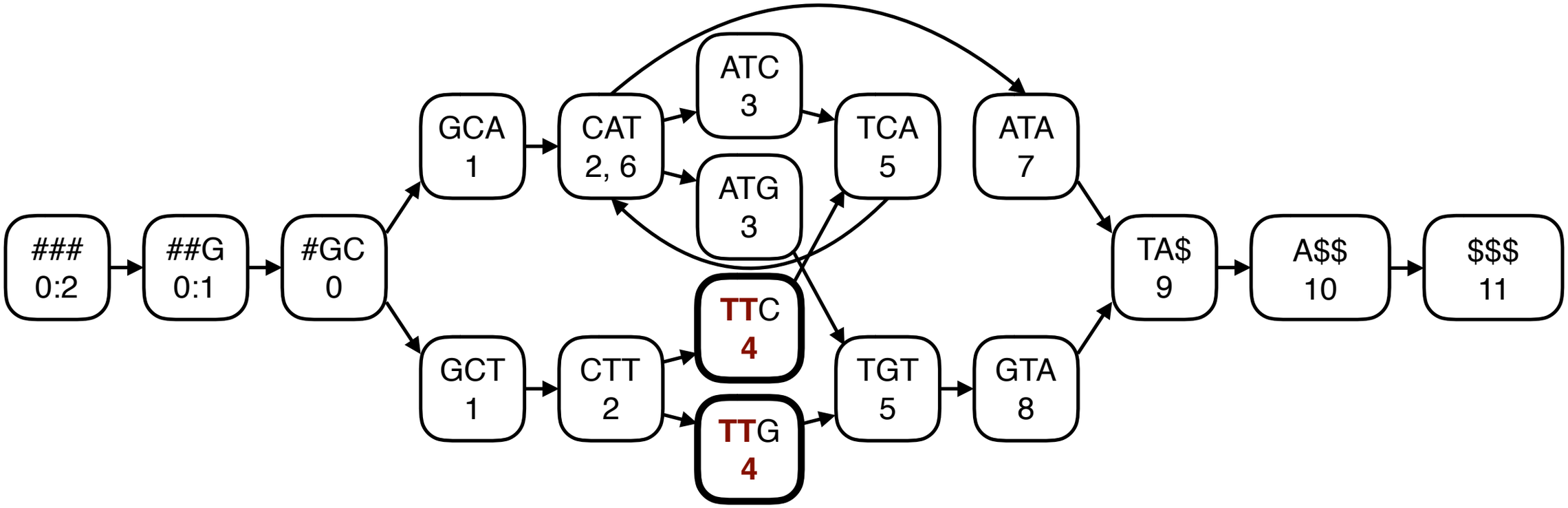
# Path graphs

- High-order de Bruijn graphs of a graph have redundant subgraphs, if shorter keys already specify the position uniquely.

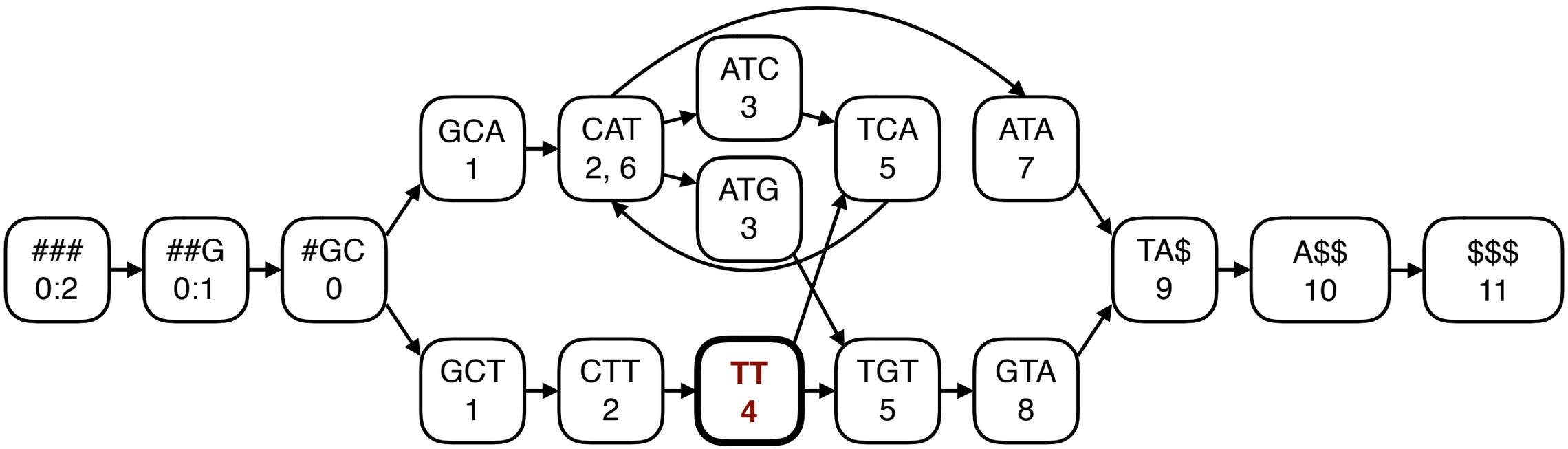
- We can compress the de Bruijn graph by merging such subgraphs.

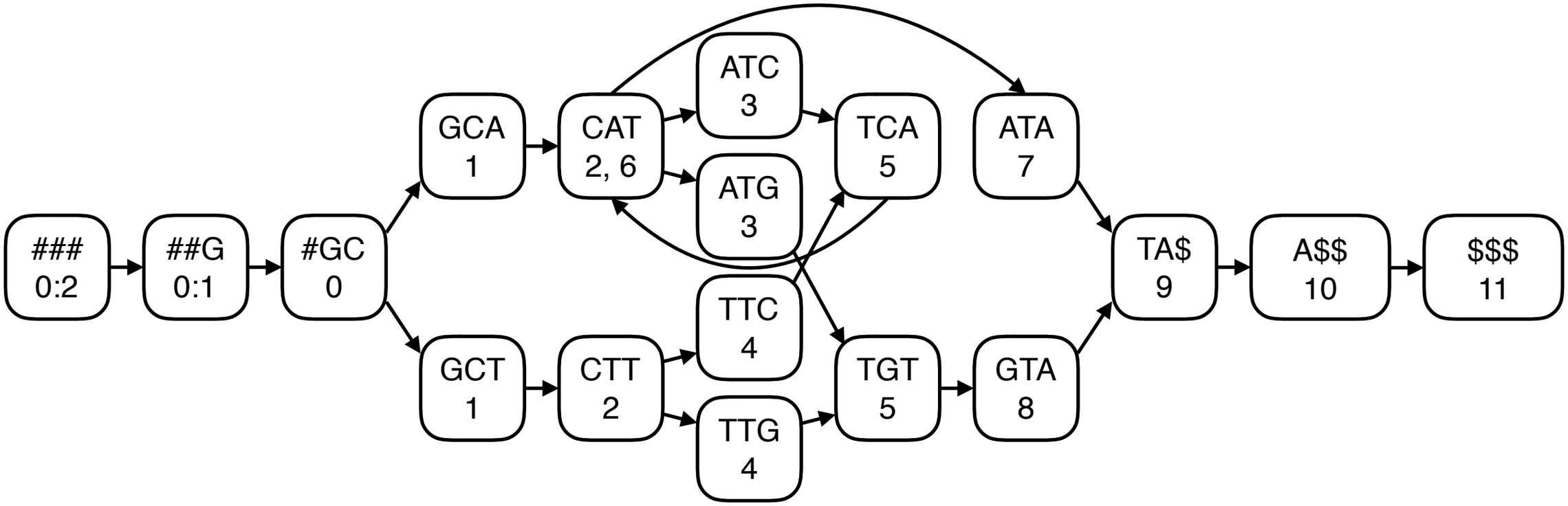
- Path graphs generalize de Bruijn graphs by using any prefix-free set of strings as keys.

- Inspired by: Sirén et al: Indexing Graphs for Path Queries with Applications in Genome Research. TCBB, 2014.

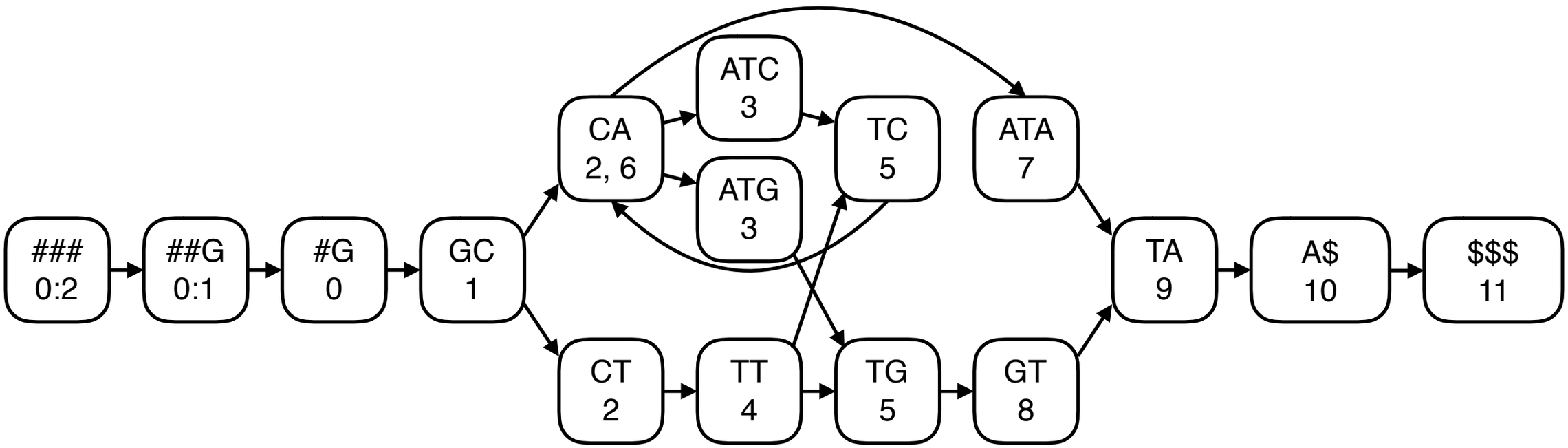


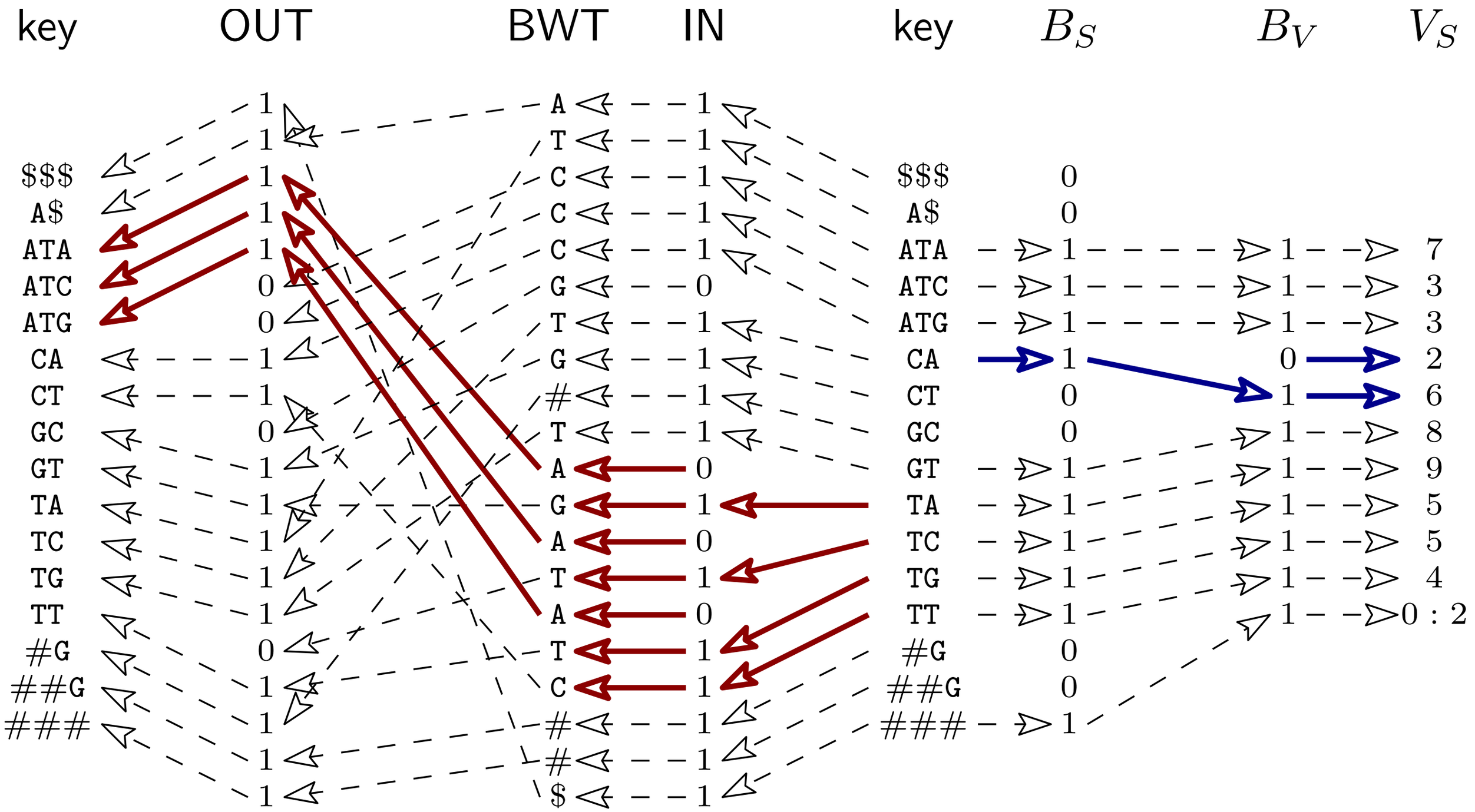

We can merge the nodes with a common prefix of keys without affecting queries, if the value sets are identical.



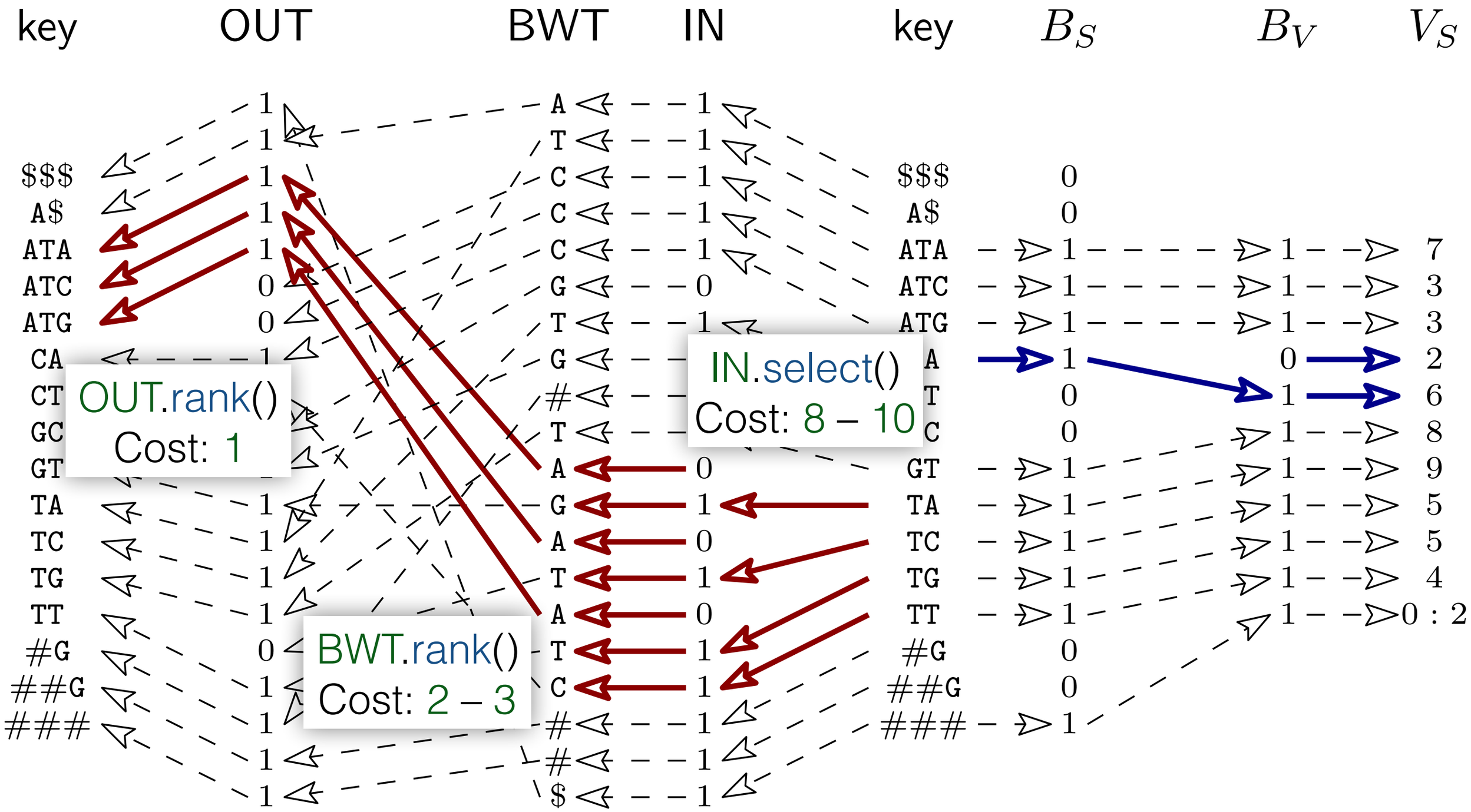



If we keep merging the nodes, we get a (maximally) pruned de Bruijn graph, which behaves intuitively.

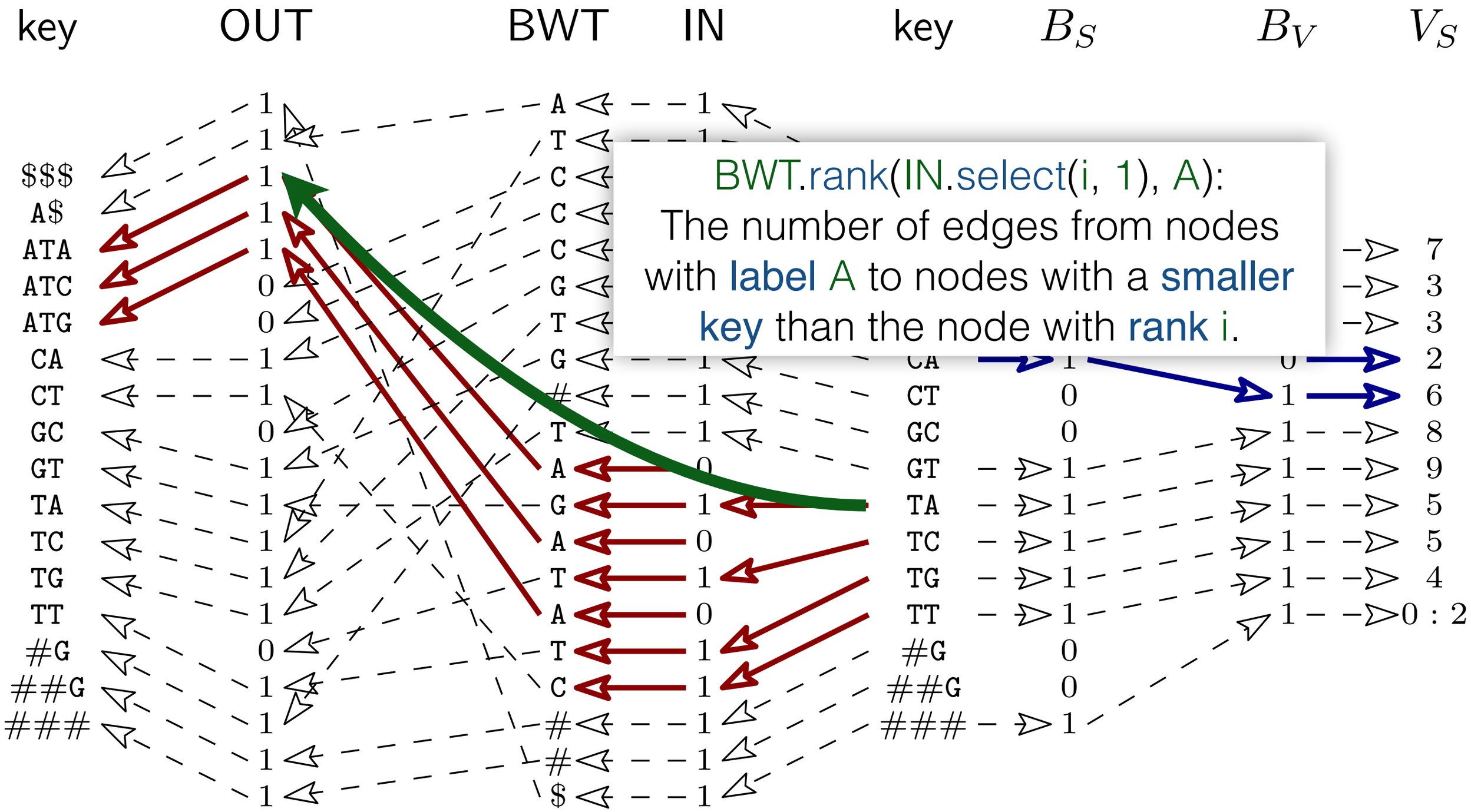





We can encode the result in the same way as in the succinct de Bruijn graph / GCSA.




This is much slower than in ordinary FM-indexes,
which use only BWT.rank().




We can use a faster encoding for reverse deterministic graphs, where each node has at most one predecessor with a given label.

(Maximally pruned) de Bruijn graphs are reverse deterministic.

$B_c$[i] tells whether the node with rank i has a predecessor with label c.

Now $B_c$.rank(i, 1) = BWT.rank(IN.select(i, 1), c).

With this encoding, GCSA2 is 2x to 4x faster.

If we compress the bitvectors for rare characters ($, N, #), the index is also a bit smaller.

| Key | BWT | $ | A | C | G | T | # |
|---|---|---|---|---|---|---|---|
| $$$ | A | 0 | 1 | 0 | 0 | 0 | 0 |
| A$ | T | 0 | 0 | 0 | 0 | 1 | 0 |
| ATA | C | 0 | 0 | 1 | 0 | 0 | 0 |
| ATC | C | 0 | 0 | 1 | 0 | 0 | 0 |
| ATG | C | 0 | 0 | 1 | 0 | 0 | 0 |
| CA | GT | 0 | 0 | 0 | 1 | 1 | 0 |
| CT | # | 0 | 0 | 0 | 0 | 0 | 1 |
| GC | G | 0 | 0 | 0 | 1 | 0 | 0 |
| GT | T | 0 | 0 | 0 | 0 | 1 | 0 |
| TA | AG | 0 | 1 | 0 | 1 | 0 | 0 |
| TC | AT | 0 | 1 | 0 | 0 | 1 | 0 |
| TG | AT | 0 | 1 | 0 | 0 | 1 | 0 |
| TT | C | 0 | 0 | 1 | 0 | 0 | 0 |
| #G | # | 0 | 0 | 0 | 0 | 0 | 1 |
| ##G | # | 0 | 0 | 0 | 0 | 0 | 1 |
| ### | $ | 1 | 0 | 0 | 0 | 0 | 0 |

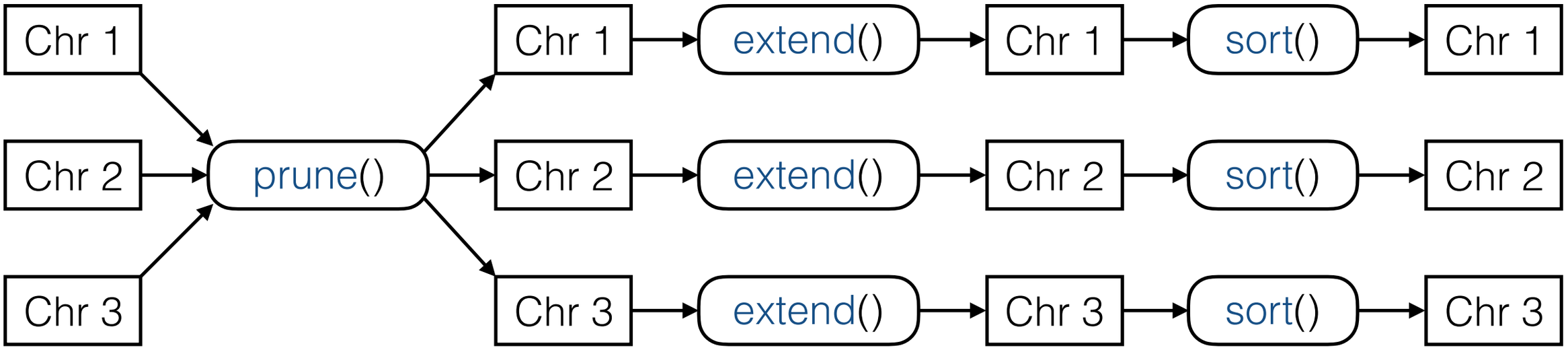
# Path graph construction

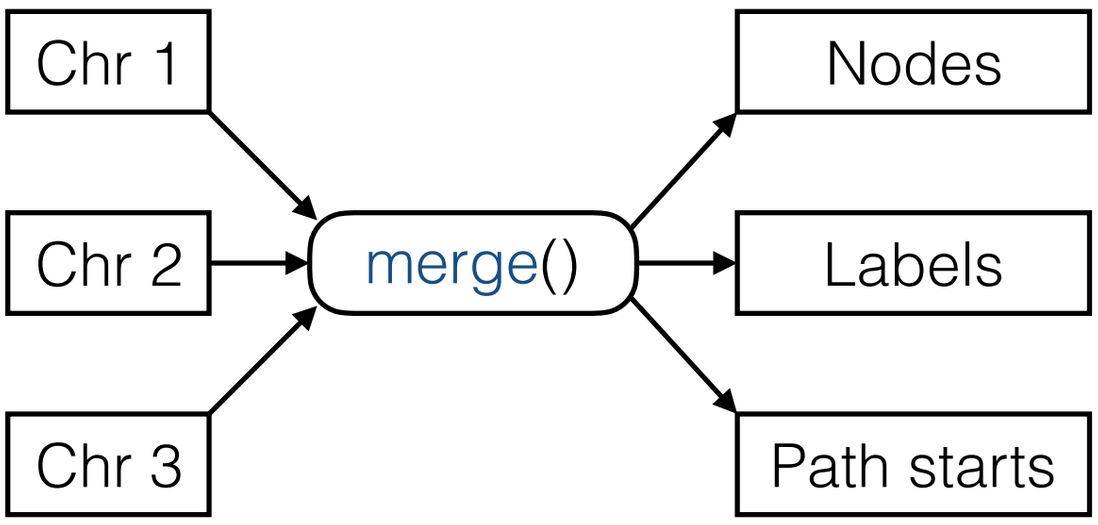
- Start from paths of length k and use a prefix-doubling algorithm to build the pruned de Bruijn graph.

- extend(): Double the path length by joining paths A→B and B→C into paths A→C.

- prune(): If all paths sharing a common prefix start from the same node, merge them into a single path.

- merge(): Merge all paths with the same label, and all paths sharing a prefix if their value sets are identical.




- prune() and merge() merge **sorted files** using a priority queue.
- extend() is done separately for each **chromosome**.
- Memory usage is often determined by extend() for the **most complex** chromosome.

# GCSA2 construction

- Index construction is essentially about determining the edges of the pruned de Bruijn graph.

- There is an edge from X to Y, if one of X and cY is a prefix of the other.

- One read pointer scans the destination nodes Y, while σ additional pointers scan the source nodes X starting with each character $c \in \sum$.

- We can determine the edges by using LF-mapping in the de Bruijn graph for the input kmers.

| Path length | 16→32 | 16→64 | 16→128 |
|---|---|---|---|
| **Kmers** | 6.20G | 16.7G | **116G** |
| **Nodes** | 4.37G | 5.24G | **5.73G** |
| **Index size** | 13.2 GB<br>18.2 bits / kmer | 13.5 GB<br>6.99 bits / kmer | **14.6 GB**<br>**1.08 bits / kmer** |
| **Construction:** | | | |
| **Time** | 7.44 h | 10.4 h | **14.1 h** |
| **Memory** | 59.8 GB | 51.9 GB | **52.3 GB** |
| **Disk** | 387 GB | 415 GB | **478 GB** |
| **I/O:** | | | |
| **Read** | 1.37 TB | 2.03 TB | **2.78 TB** |
| **Write** | 0.88 TB | 1.51 TB | **2.25 TB** |

1000GP human variation (forward strand only)

`vg mod -p -l 16 -e 4 | vg mod -S -l 100`

32 cores, 256 GB memory, distributed Lustre file system

| k | Index | kmers | Matched | find() | locate() |
|---|---|---|---|---|---|
| 16 | GCSA2 | 351584 | 347453 | 4.75 µs | 5.85 µs |
| | BWA | 351584 | 320764 | 3.64 µs | 4.65 µs |
| | csa_wt | 351584 | 301538 | 6.00 µs | 2.43 µs |
| 32 | GCSA2 | 351555 | 333258 | 10.8 µs | 5.44 µs |
| | BWA | 351555 | 156080 | 6.57 µs | 3.19 µs |
| | csa_wt | 351555 | 153957 | 10.9 µs | 2.16 µs |

GCSA2: Order-128 index for the pruned variation graph

BWA: The FM-index from BWA v0.7.15 for the reference and its reverse complement

csa_wt: Fast FM-index from SDSL for the reference

Average time for find queries (per query) and locate queries (per distinct occurrence) with kmers extracted from the non-pruned variation graph.

# Suffix Tree of a Path Graph

# Maximal exact matches

- Many read aligners are based on finding maximal exact matches between the read and the reference using the bidirectional BWT.

- The bidirectional BWT requires that the lexicographic range and the reverse range have the same length.

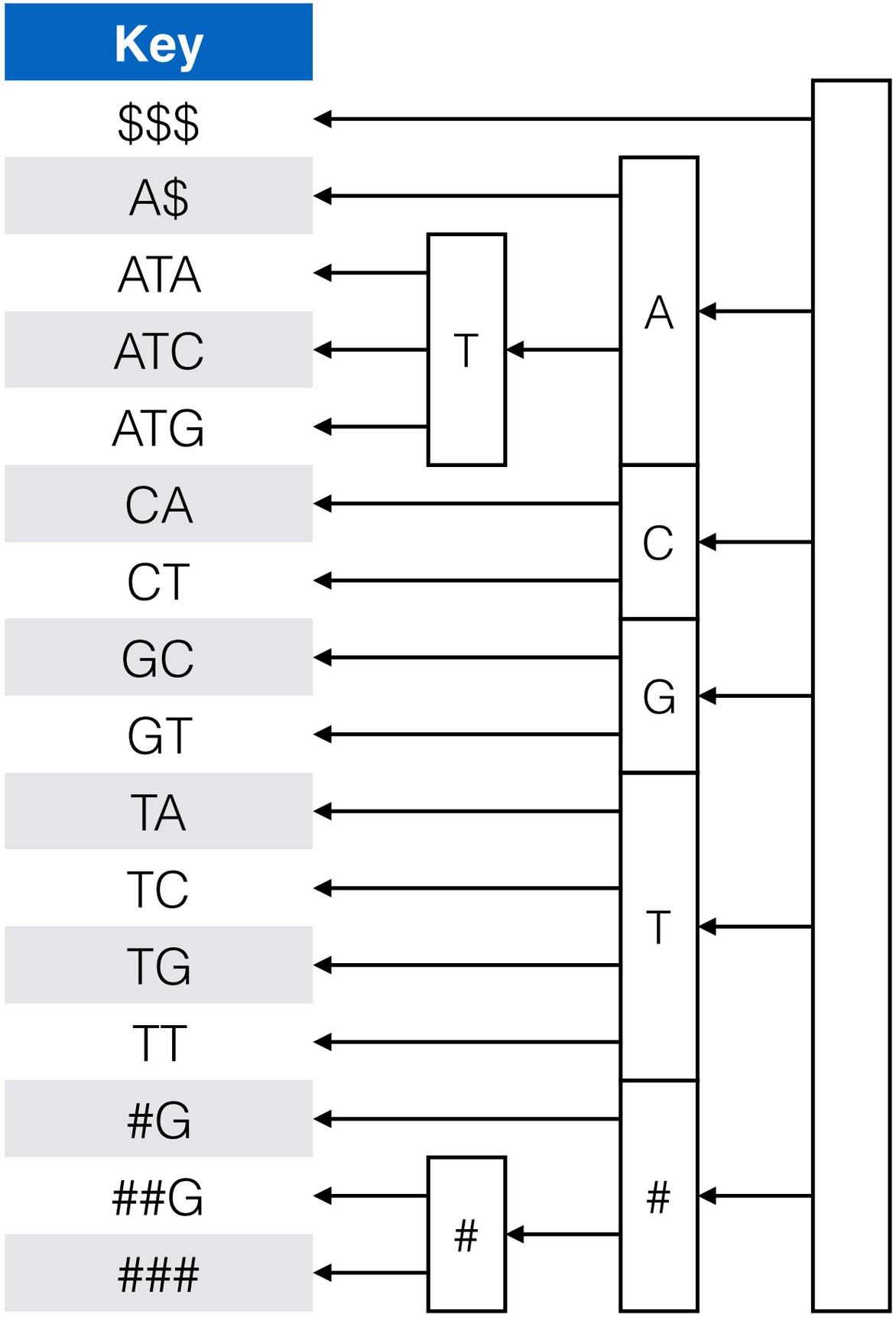
- The key set must contain the reverse complement of each key to guarantee this. We do not know how to do that efficiently.

- We can use compressed suffix trees instead.

The compacted trie of keys looks sufficiently similar to the suffix tree.

We can consider it the suffix tree of the path graph.

If the path graph is a maximally pruned de Bruijn graph, the suffix tree behaves intuitively.



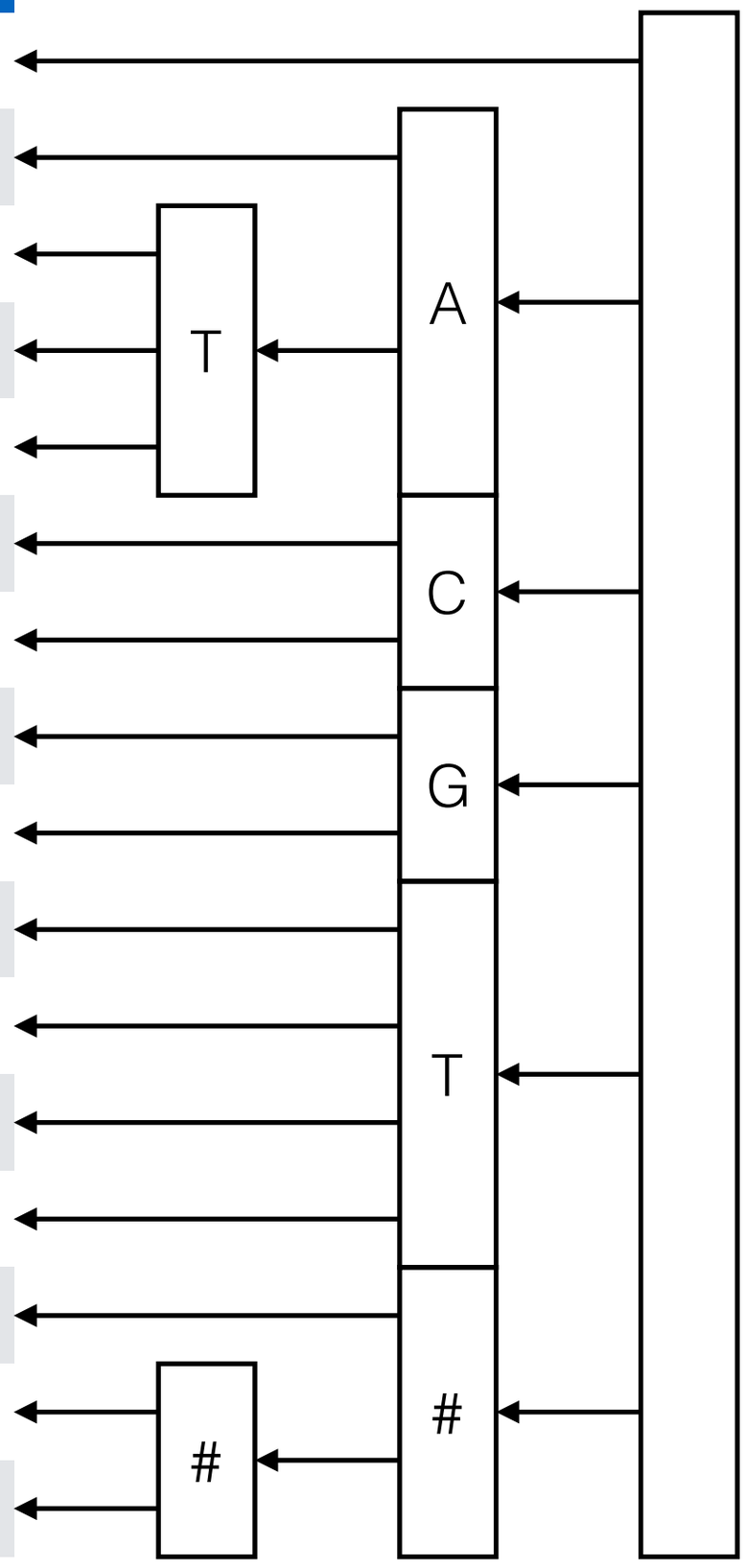
LCP[i…j] is an LCP-interval at depth d, if:

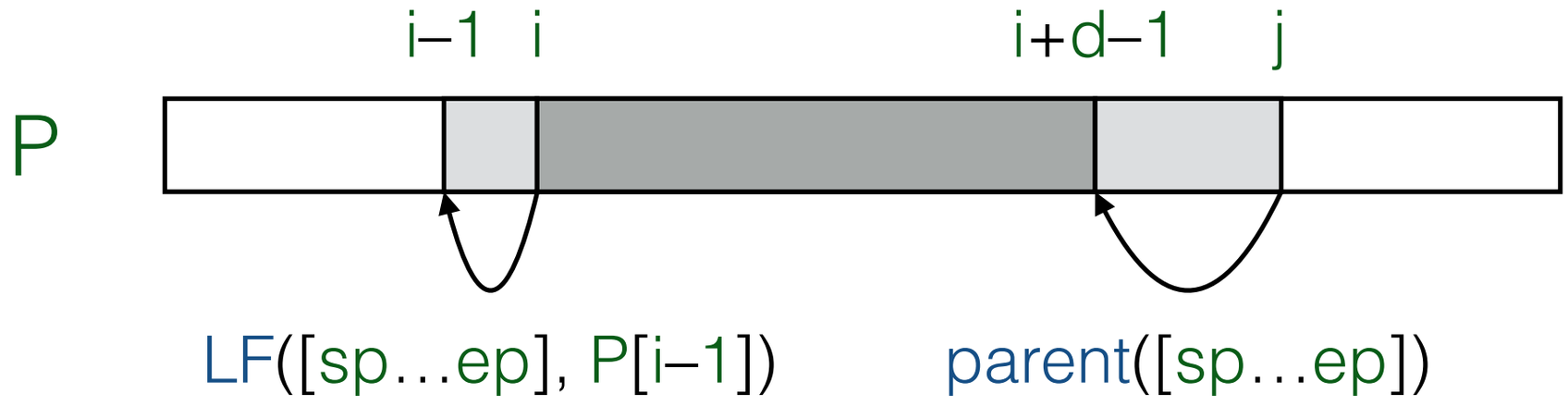
- LCP[i] < d;
- LCP[j+1] < d;
- LCP[i+1…j] ≥ d; and
- LCP[i+1…j] contains value d.

The LCP interval tree is equivalent to the suffix tree. (Abouelhoda et al: Replacing suffix trees with enhanced suffix arrays. JDA, 2004.)

We can simulate the suffix tree with next/previous smaller value queries and range minimum queries in the LCP array. (Fischer et al: Faster entropy-bounded compressed suffix trees. TCS, 2009)

| Key | LCP |
|---|---|
| $$$ | 0 |
| A$ | 0 |
| ATA | 1 |
| ATC | 2 |
| ATG | 2 |
| CA | 0 |
| CT | 1 |
| GC | 0 |
| GT | 1 |
| TA | 0 |
| TC | 1 |
| TG | 1 |
| TT | 1 |
| #G | 0 |
| ##G | 1 |
| ### | 2 |






If lexicographic range [sp…ep] **matches** substring P[i…j] of the **pattern**,

- lexicographic range LF([sp…ep], P[i–1]) matches substring P[i–1…j] of the pattern; and
- range parent([sp…ep]) matches P[i–1…i+d–1], where d < j+1–i is the depth of the **parent node**.

Ohlebusch et al: **Computing Matching Statistics and Maximal Exact Matches on Compressed Full-Text Indexes**. SPIRE 2010.

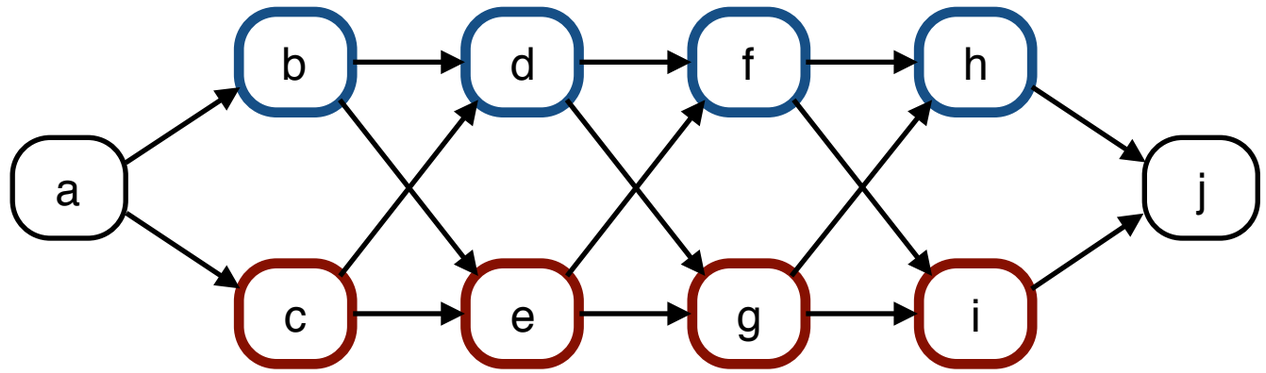
# Pruning the Variation Graph

# Complex regions

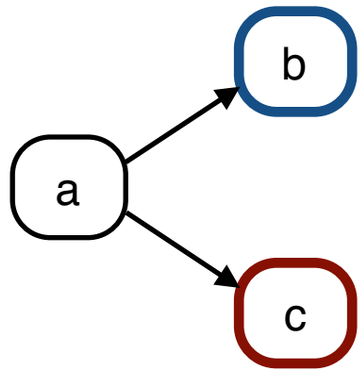
- A whole-genome human variation graph based on 1000GP variation contains trillions (quadrillions?) of distinct 128-mers.
- Almost all of them are from a few complex regions.
- We cannot index all potential recombinations in such regions.
- vg and GCSA2 have several ways for dealing with the complex regions.

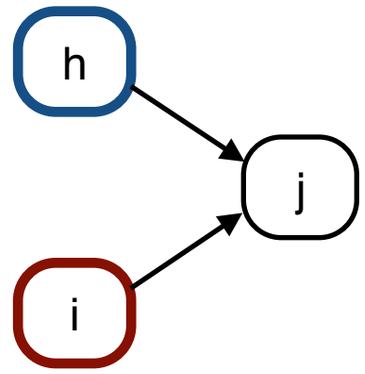
# Pruning

**`vg mod -p -l 16 -e 4`**
Remove paths of length 16 crossing more than 4 nontrivial edges.




**`vg mod -S -l 100`**
Remove subgraphs shorter than 100 bases.

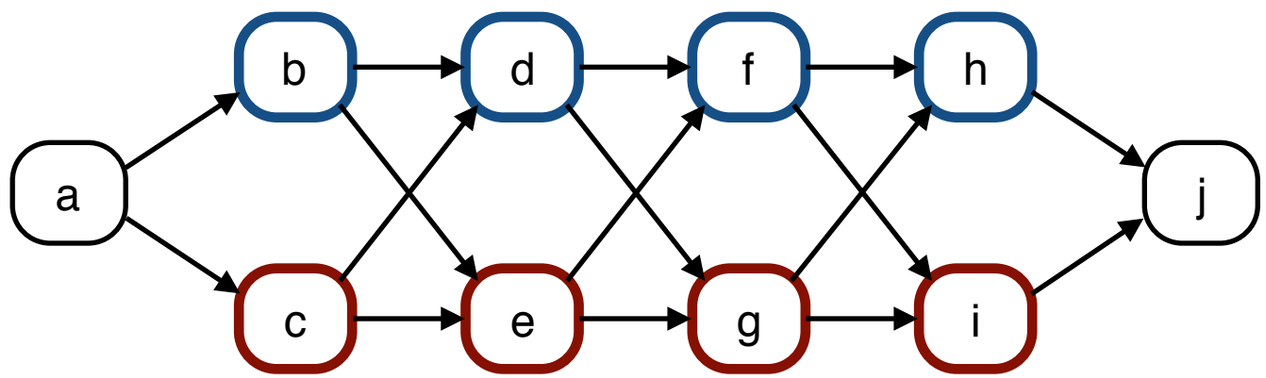


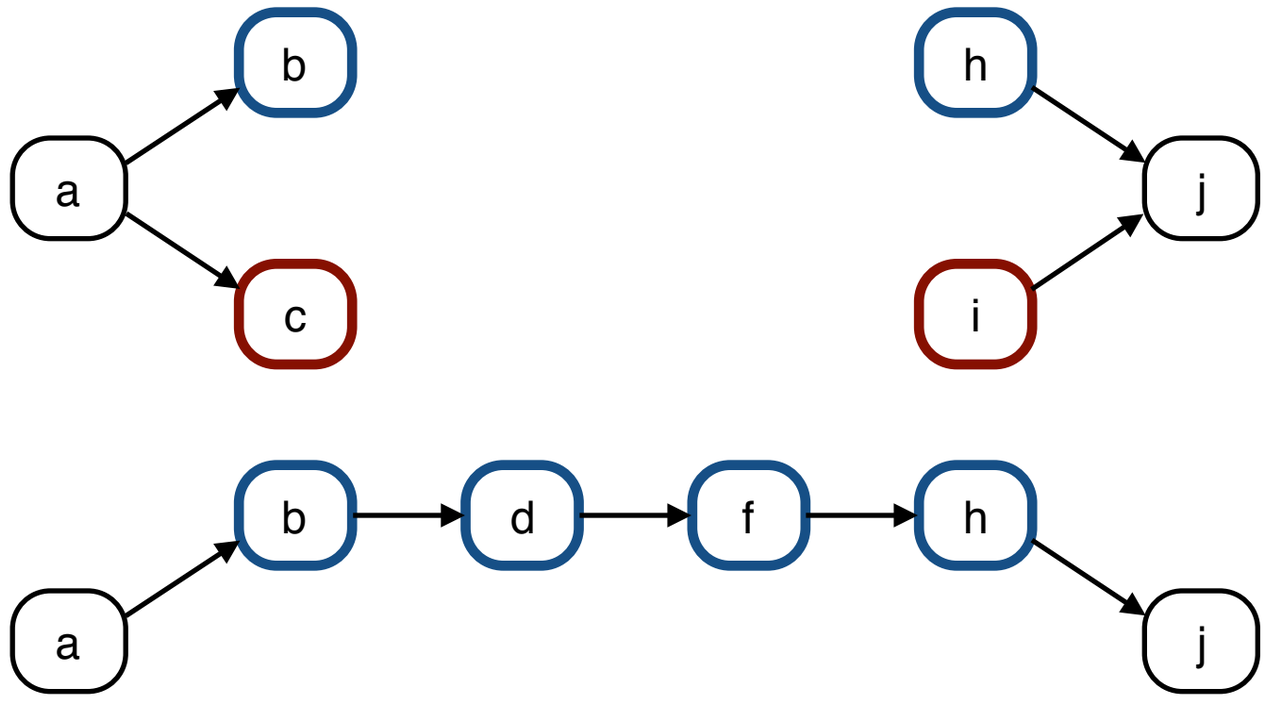

- Easy and efficient.
- Complex regions may be removed completely.

# Indexing subgraphs

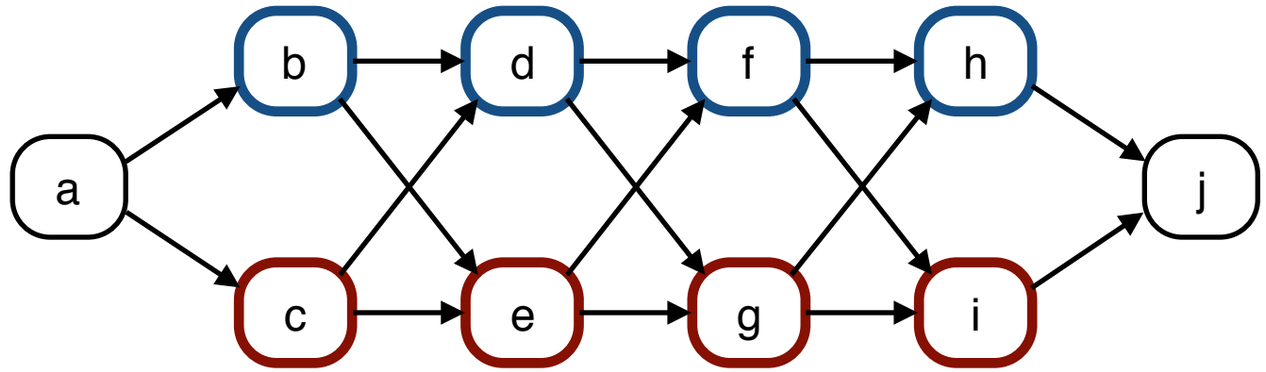
We can index overlapping subgraphs (e.g. a pruned variation graph and the reference path) and merge the results into a single index.




- Guarantees that the entire genome is indexed.
- Redundant paths can make index construction more expensive.
- Requires a reverse deterministic graph for the fast GCSA encoding.

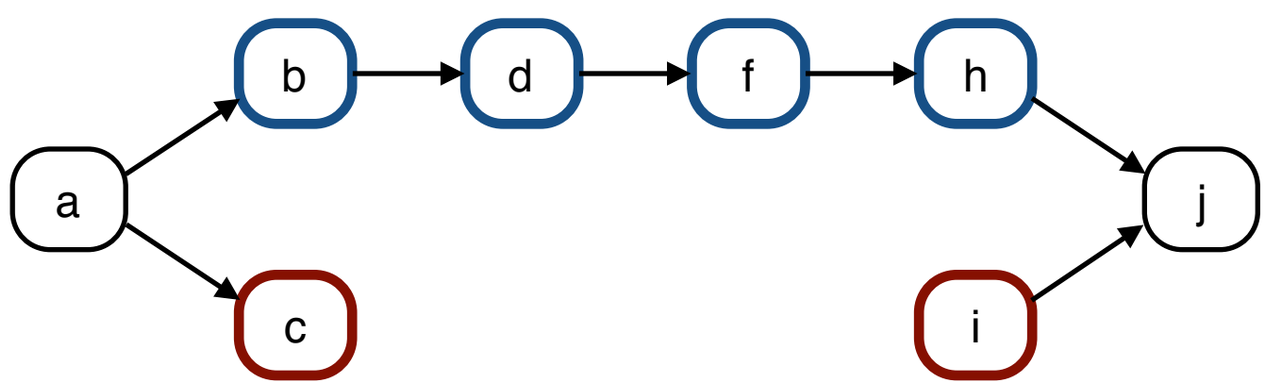


# Pruning



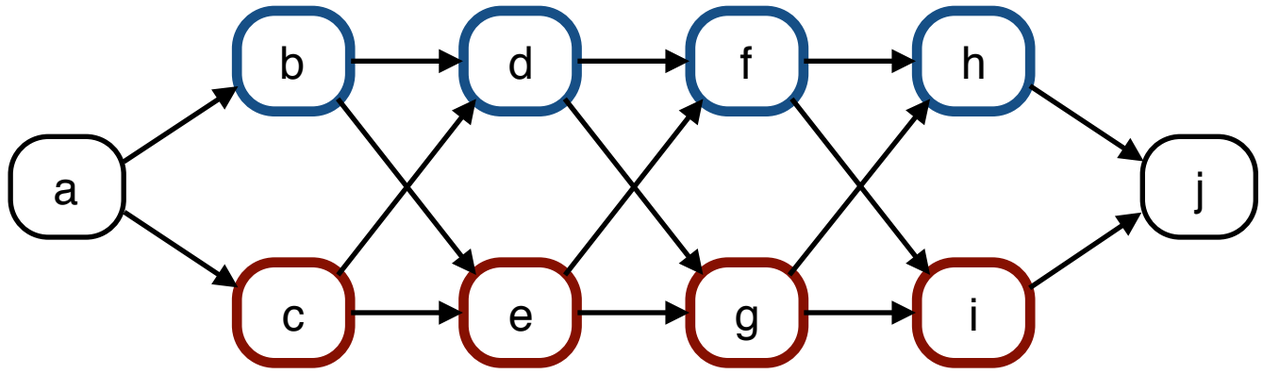

Remove paths crossing too many nontrivial edges, unless they are on the reference path.

- No need to determinize the graph.
- Things are getting quite complex.



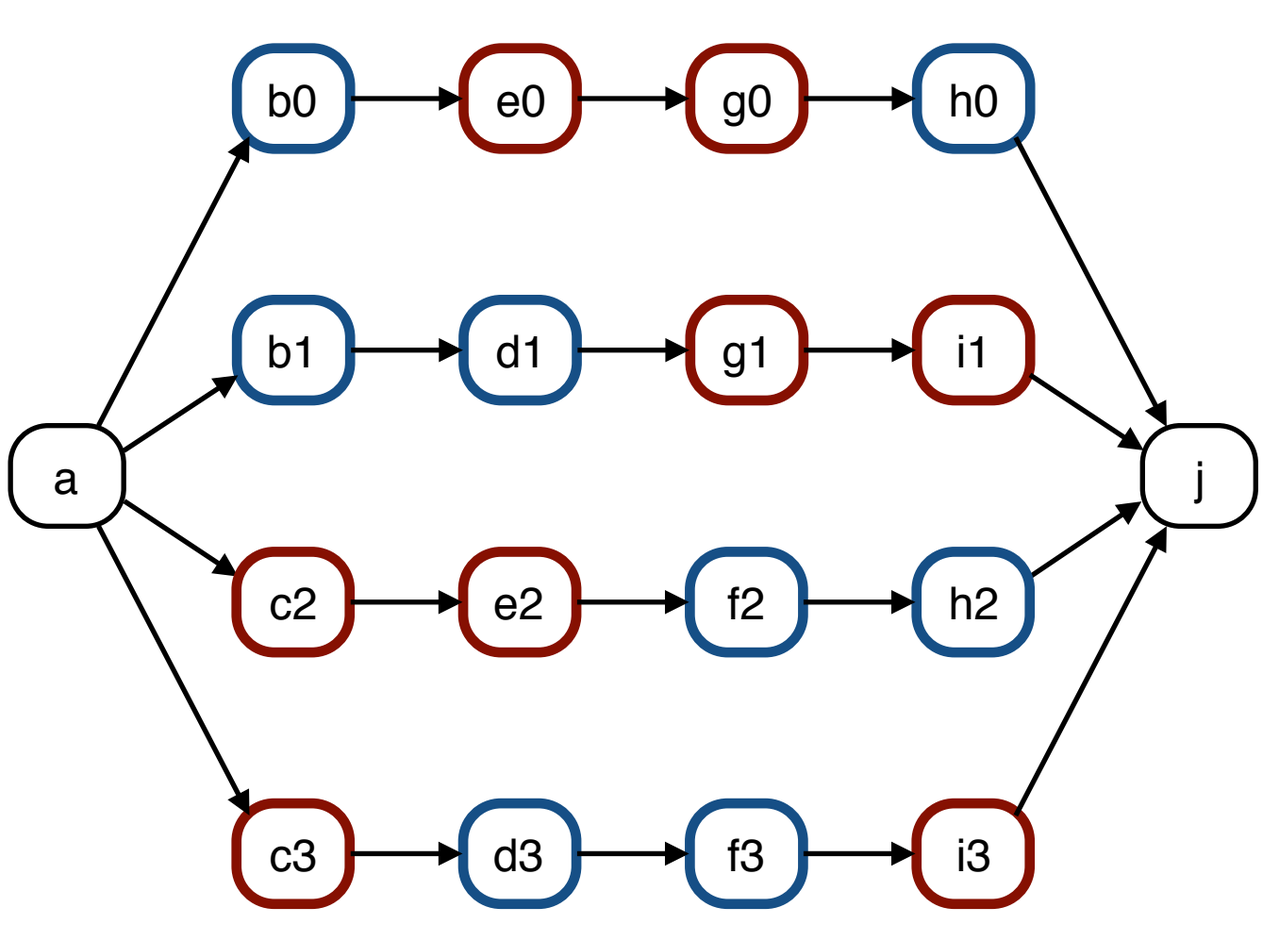
# Indexing haplotypes

Index only paths corresponding to known haplotypes in complex regions.




Multiple nodes of the input graph map to the same node in the variation graph.

- Guarantees that the entire genome and all observed variation is indexed.
- Not implemented yet in vg.



# Conclusions

- The design of a path index is a trade-off between index size, query performance, maximum query length, and ignoring complex regions of the graph.

- GCSA2 prioritizes performance and size, while supporting queries of length up to 128.

- It uses a de Bruijn graph as a kmer index, compresses it by merging redundant subgraphs, and encodes the result as a compressed suffix tree.

- Sirén: Indexing Variation Graphs. arXiv:1604.06605, 2016. Accepted to ALENEX 2017. https://github.com/jltsiren/gcsa2