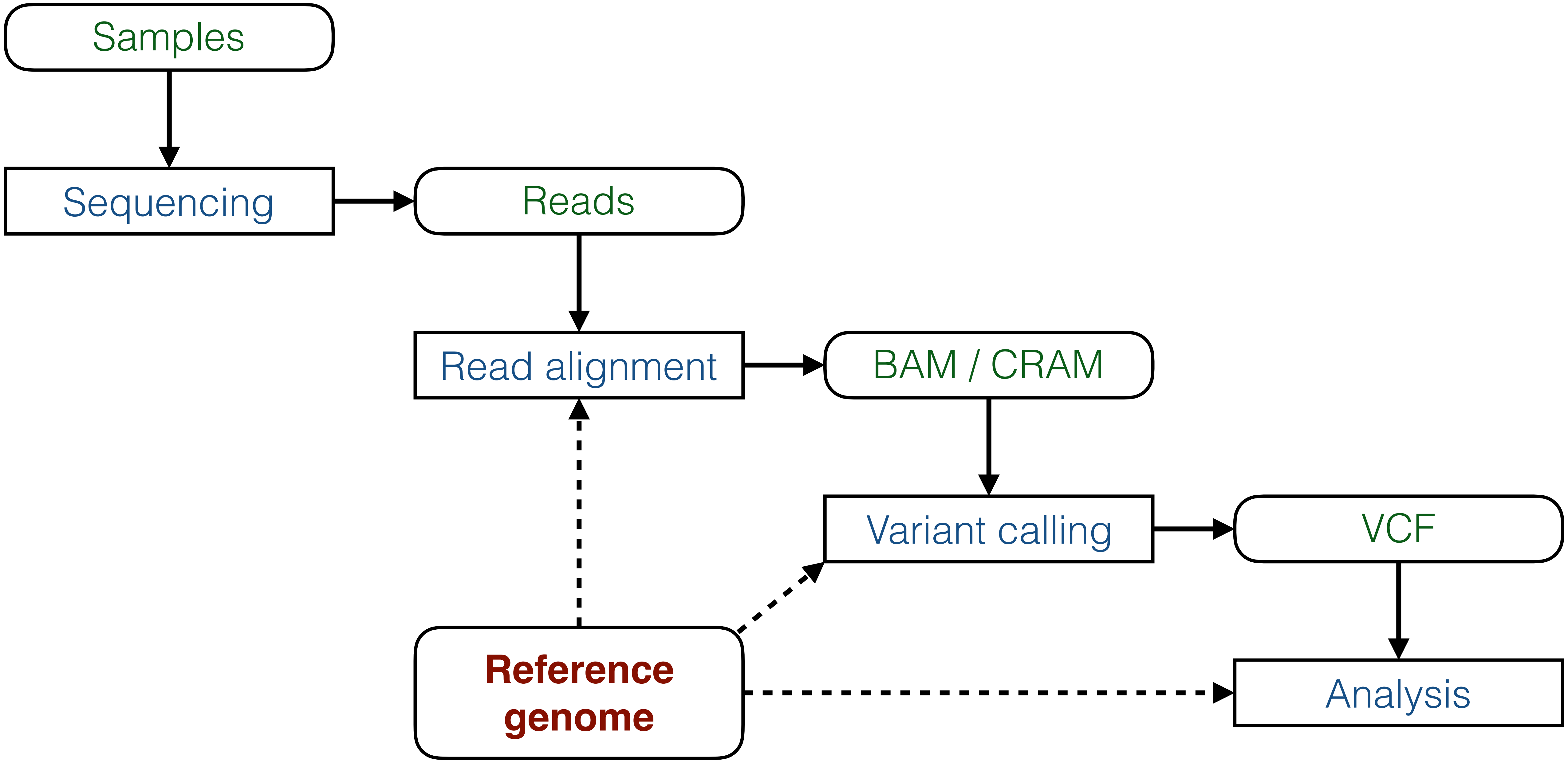
Indexing
Variation Graphs

Jouni Sirén
Wellcome Trust Sanger Institute

Typical pipeline



Reference bias

- Read alignment, variant calling, and the subsequent analysis all depend on the reference genome.
- Most reference genomes are based on the genomes of a small number of individuals or populations.
- The analysis may be biased towards those individuals and populations.

Avoiding reference bias

1. De novo assembly of individual genomes (not really possible with Illumina reads)
2. Reference-free analysis based on the reads (yesterday)
3. Using a graph reference (today)
4. Analysis based on assembly graphs (a combination of the above)

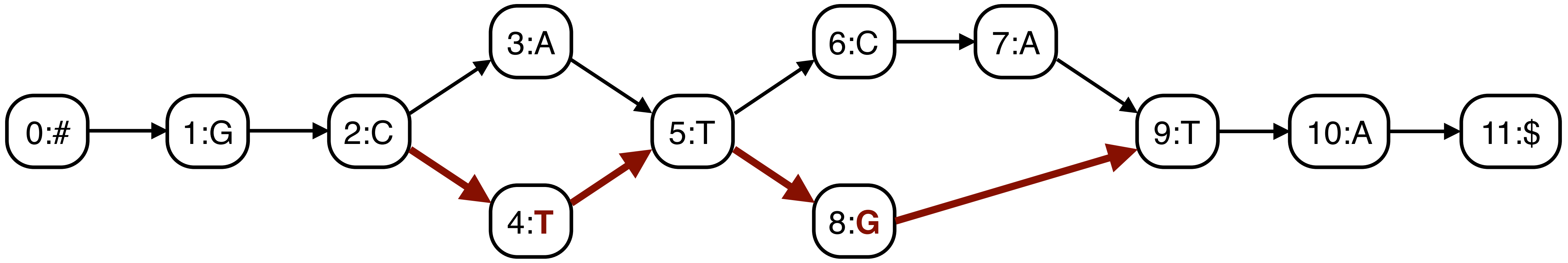
Contents

Variation Graphs

From alignments to graphs

```
GCATCATA
GCATG-TA
GCTTCATA
```

- We can represent a collection of aligned sequences as a graph.
- The graph generalizes the alignment by allowing for recombinations at aligned positions.



```
GCTTG-TA
```

Graph genomes

Global Alliance
for Genomics & Health

- A graph genome augments the reference sequence with known/ frequent variation in the relevant species/subspecies/population.

- Graph genome? Genome graph? Reference graph? Graph reference? Variation graph?

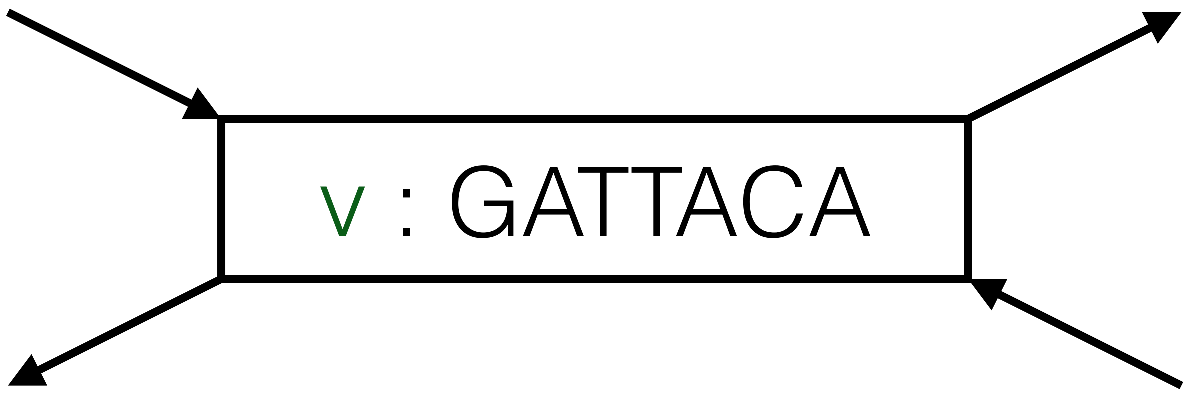
- The Global Alliance for Genomics and Health has been developing and evaluating approaches to including variation in reference genomes.

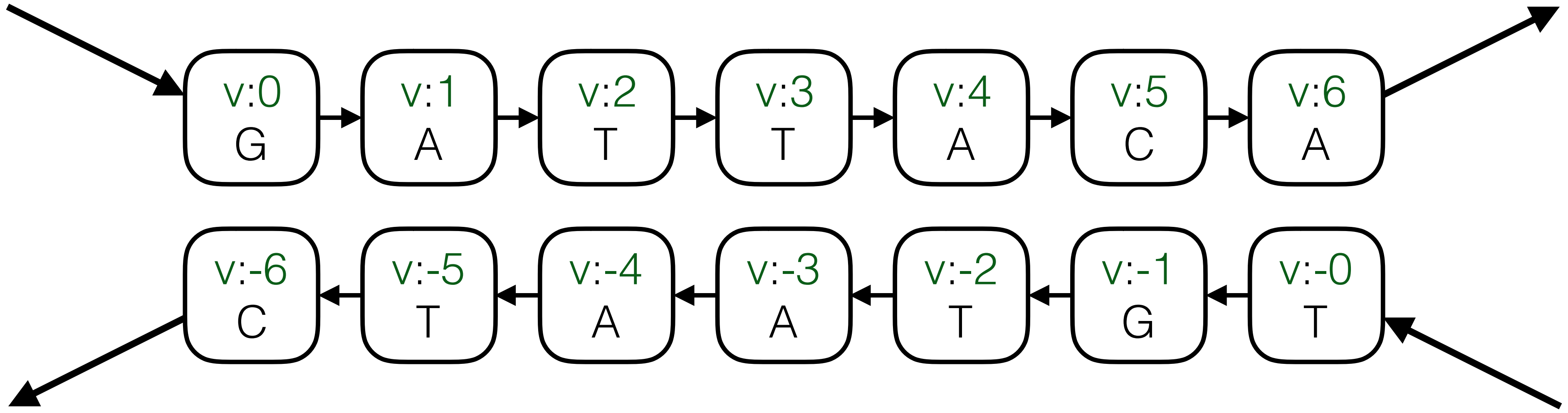
Variation graph toolkit vg

https://github.com/vgteam/vg

- Erik Garrison started developing the variation graph toolkit vg a couple of years ago.
- The toolkit has become a community effort to develop tools for working with variation graphs.
- Some companies offer similar products, but vg is the only free software graph-based sequence analysis pipeline anywhere near ready for production use.

Variation graphs

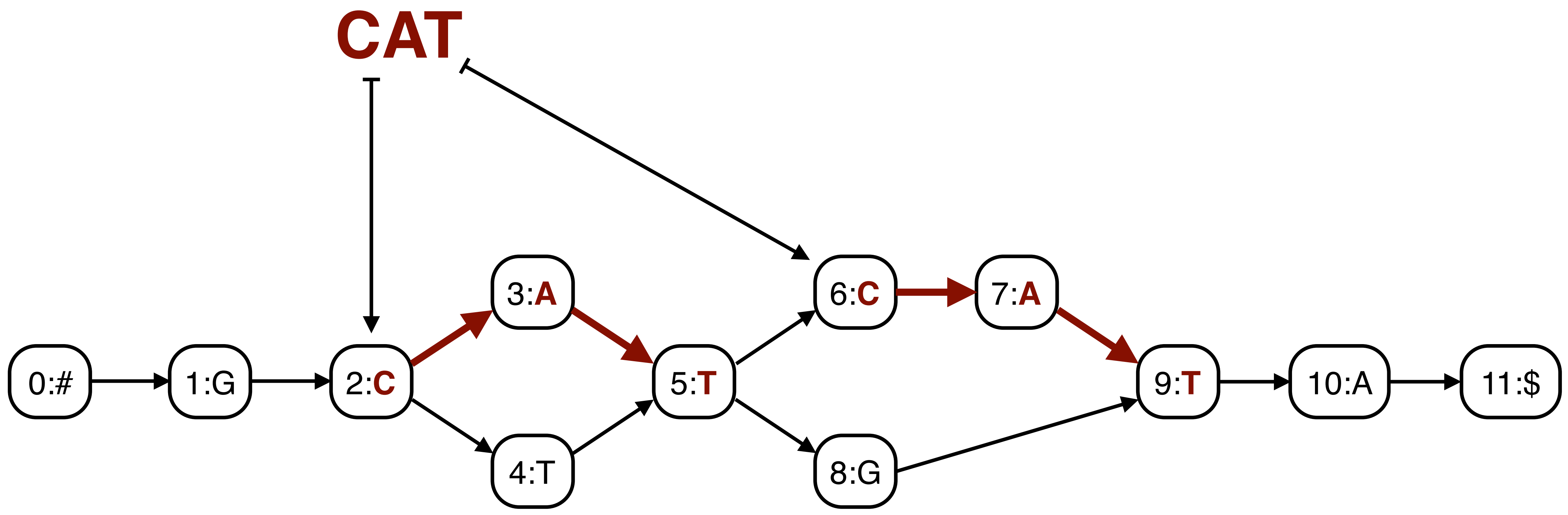




Simple directed graphs are easier to handle. The transformation is also useful for other purposes.

Path Indexes

Path indexes



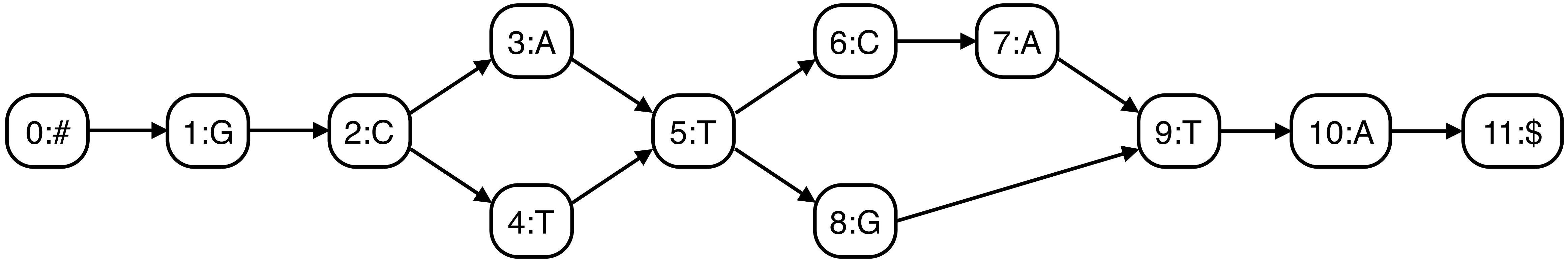
Path indexes are a central tool for working with variation graphs. They are text indexes for the path labels in a graph. The index finds the (start nodes of) the paths labeled by the query string.

Path indexes

- The number of kmers in a graph increases exponentially with k.
- 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.
- Query performance is probably the most critical issue, followed by index size.

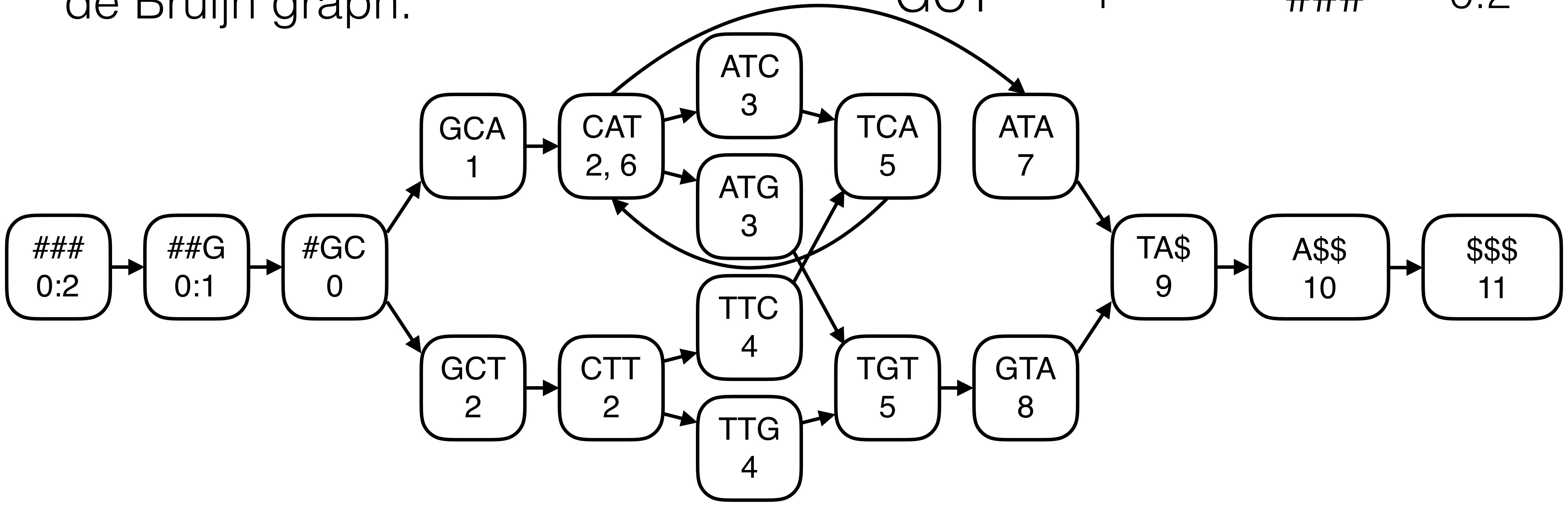
- The kmer index is a simple path index. It consists of a set of key-value pairs.
- A hash table supports fast kmer queries.
- Binary search in a sorted array is slower but supports queries shorter than k.

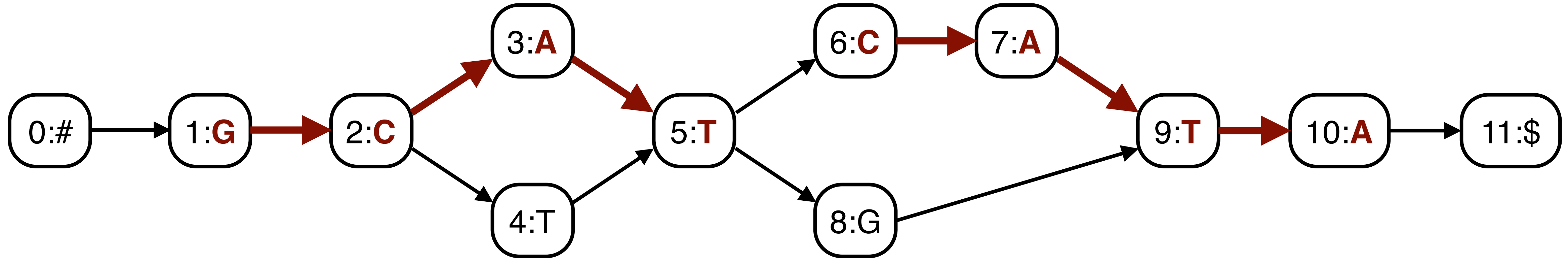
Key	Value	Key	Value
$$$	11	GTA	8
A$$	10	TA$	9
ATA	7	TCA	5
ATC	3	TGT	5
ATG	3	TTC	4
CAT	2, 6	TTG	4
CTT	2	#GC	0
GCA	1	##G	0:1
GCT	1	###	0:2



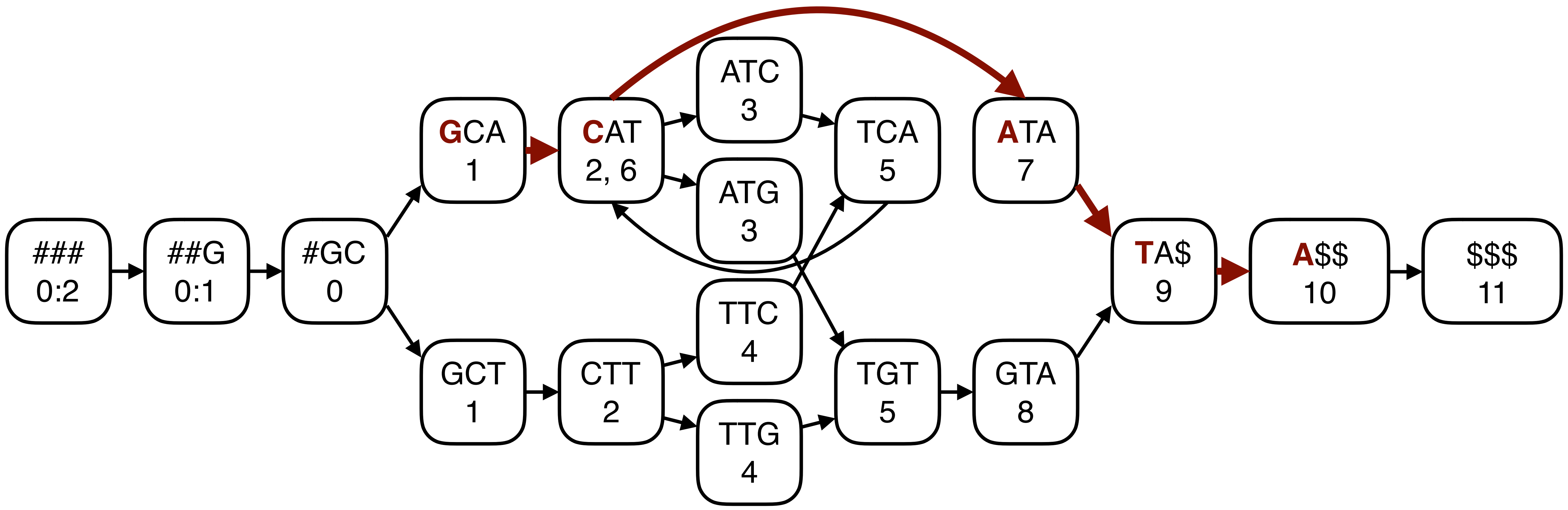
- We can represent the kmer index as a de Bruijn graph.
- Each edge in the de Bruijn graph is a (k+1)-mer in the input graph.
- We label each node with the first character of the key. All paths of length up to k+1 in the input graph exist in the de Bruijn graph.

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

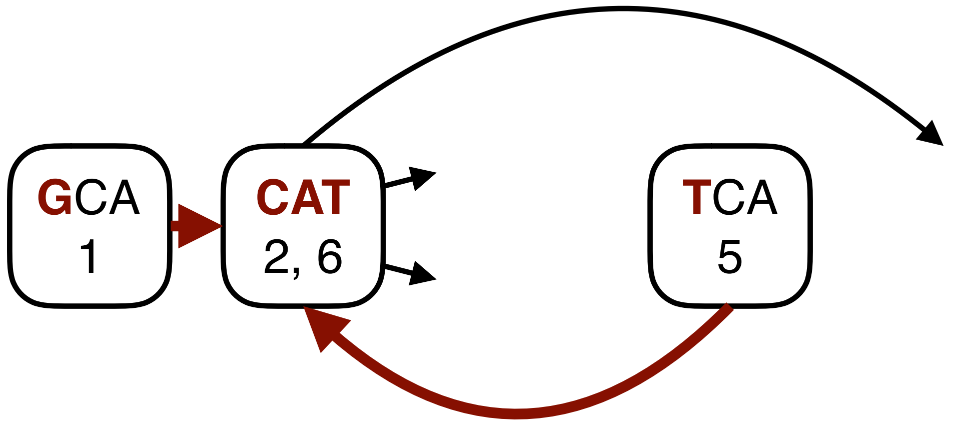


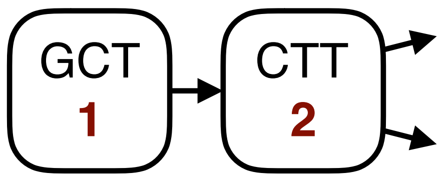


Paths longer than k+1 may be false positives, but we can verify them in the input graph.



Storing the de Bruijn graph





- If we know that node CAT exists, we can encode nodes GCA and TCA and the edges from them by listing the predecessor labels G and T.
- If nodes on unary paths have successive values, we can use a similar sampling scheme for the values as in the FM-index.
- This reduces the size of the kmer index from several integers per kmer to a couple of bytes per kmer.

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.

- **Backward searching** uses select on IN to find the incoming edges, LF on BWT to find the outgoing edges, and rank on OUT to find the predecessor nodes.

- Bowe et al: **Succinct de Bruijn graphs**. WABI 2012.

GCSA2

The index is still too large

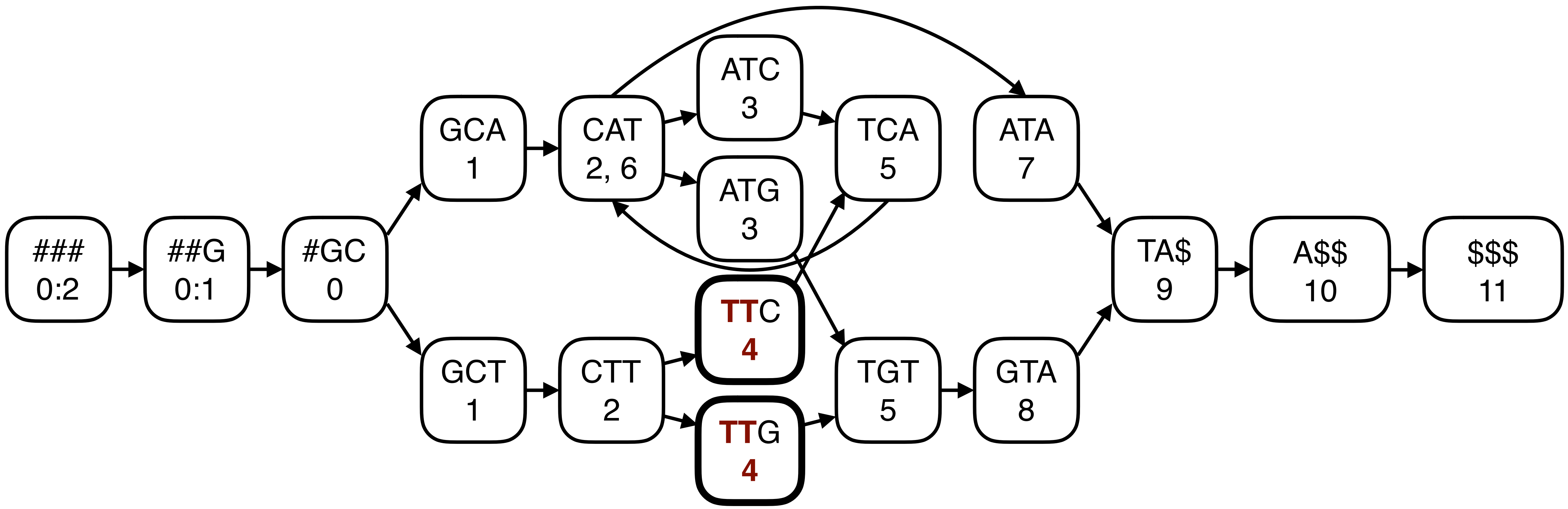
- Short reads are typically ~100 bp. The value of k in a kmer index should be more than that.
- There can be hundreds of billions of 128-mers in a whole-genome index for both strands.
- 2 bytes / kmer is too much for such indexes.
- We can use ideas from GCSA to compress the de Bruijn graph.
- Sirén et al: Indexing Graphs for Path Queries with Applications in Genome Research. TCBB, 2014.

Path graphs

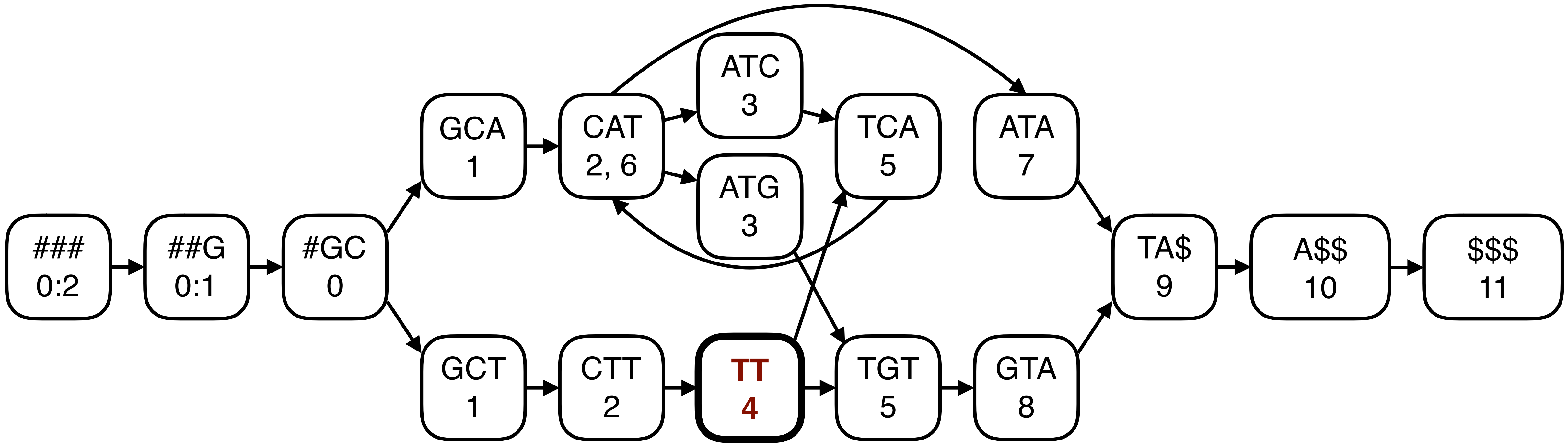
X	**GATTACA**
Y	ATTACAT
cY	**GATTACA**T

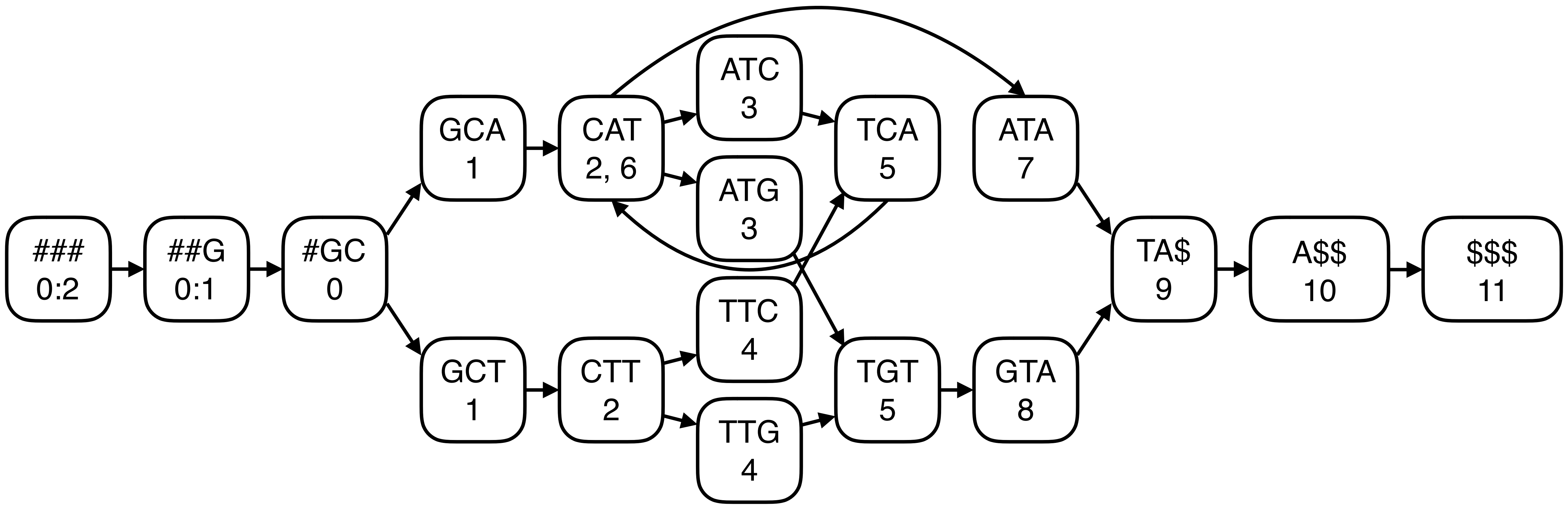
X	**GATTAC**A
Y	ATTAC
cY	**GATTAC**

- High-order de Bruijn graphs of a graph have redundant subgraphs, if shorter keys would 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.
- There is an edge from X to Y, if string cY exists in the source material and one of X and cY is a prefix of the other.

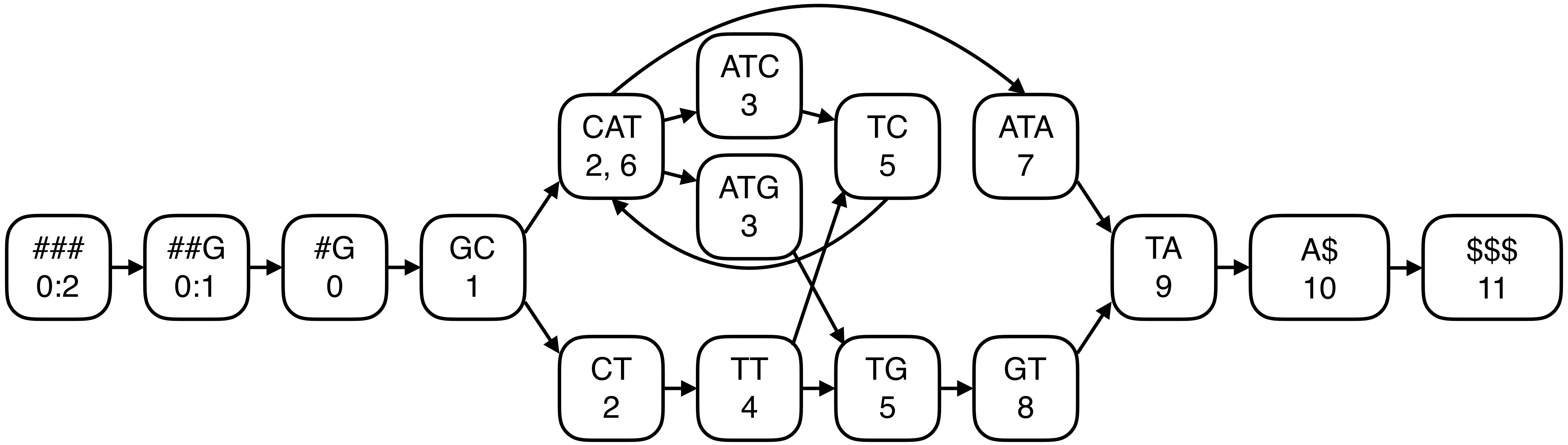


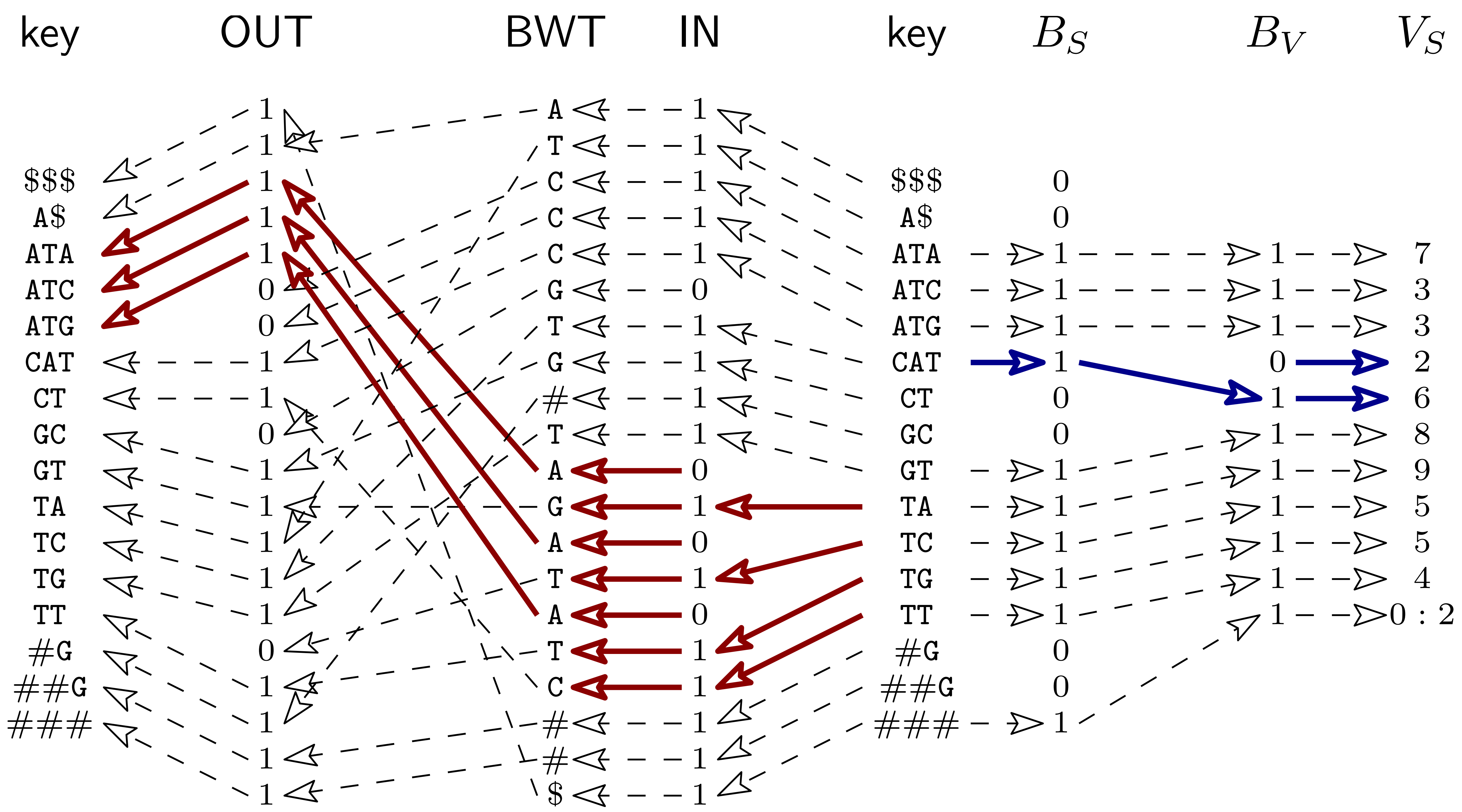
We can merge nodes sharing a prefix without affecting queries, if the value sets are identical.





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

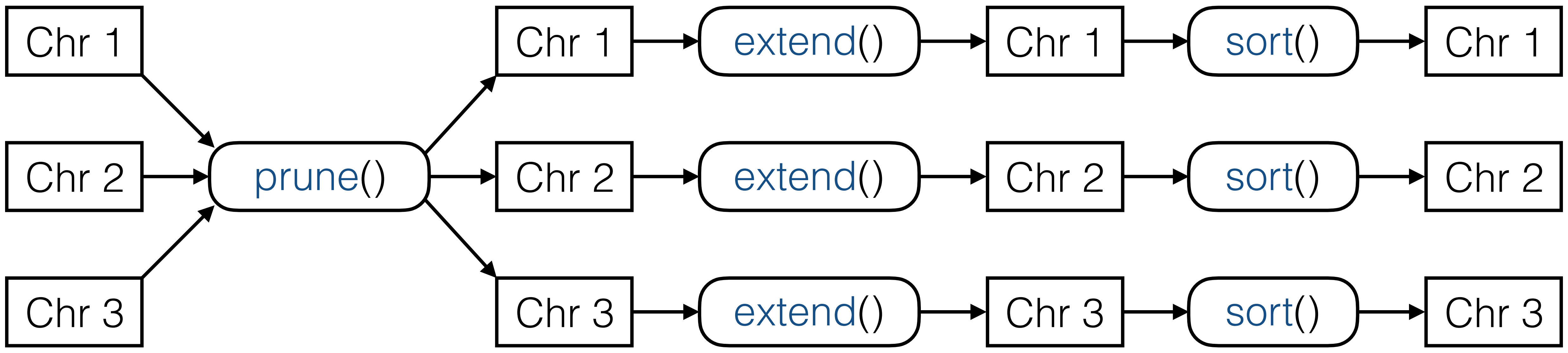


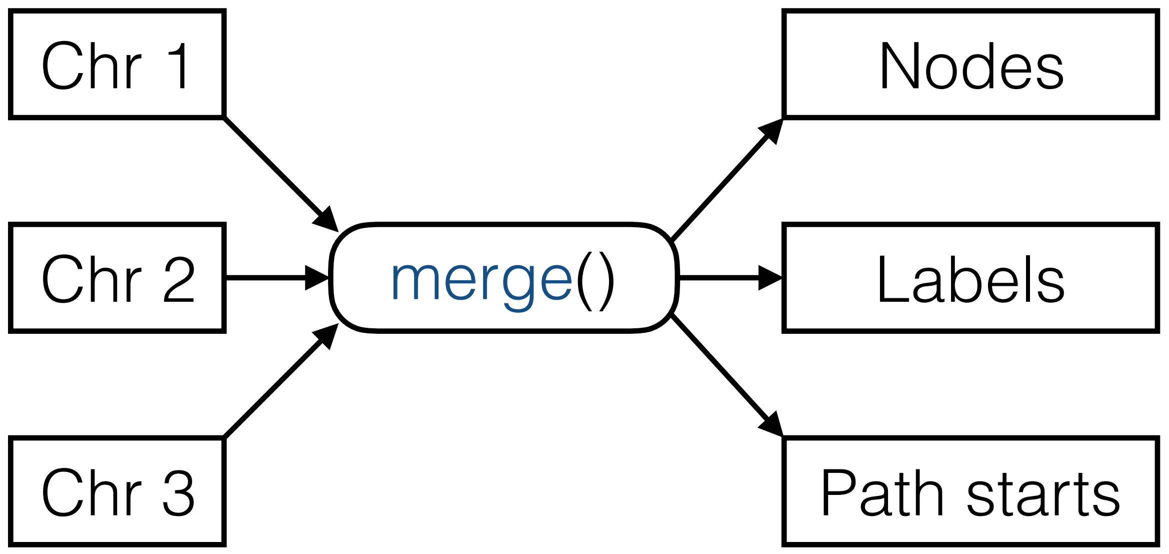


We can encode the result in the same way as in the succinct de Bruijn graph / GCSA.

GCSA 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.
- One read pointer scans the destination nodes Y, while σ additional pointers scan the source nodes X starting with each character $c \in \Sigma$.
- Path labels are stored as sequences of kmer ranks.
- LF-mapping in a de Bruijn graph for the original kmers transforms the sequence for Y into the sequence for cY.

Path length	16→32	16→64	16→128
Kmers Nodes	6.20G 4.37G	16.7G 5.24G	116G 5.73G
Index size	9.97 GB 13.8 bits / kmer	9.19 GB 4.74 bits / kmer	9.17 GB 0.68 bits / kmer
Construction: Time Memory Disk	 8.62 h 63.1 GB 387 GB	 12.1 h 55.9 GB 415 GB	 16.1 h 56.6 GB 478 GB
I/O: Read Write	 1.37 TB 0.88 TB	 2.03 TB 1.51 TB	 2.78 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

Index	k	Patterns	Found	find()	locate()
GCSA	16	351584	347453	18.3 µs	12.1 µs
GCSA	32	351555	333258	37.9 µs	11.1 µs
GCSA	64	351567	326101	76.1 µs	5.76 µs
GCSA	128	351596	316500	149 µs	5.98 µs
csa_wt	16	351584	301538	6.06 µs	2.46 µs
csa_wt	32	351555	153957	10.9 µs	2.19 µs
csa_wt	64	351567	88184	17.2 µs	1.91 µs
csa_wt	128	351596	35678	23.6 µs	3.48 µs

GCSA2 vs a similar FM-index from SDSL.
Patterns extracted from the non-pruned variation graph.
Time per find query / distinct occurrence.

Suffix Tree of a Path Graph

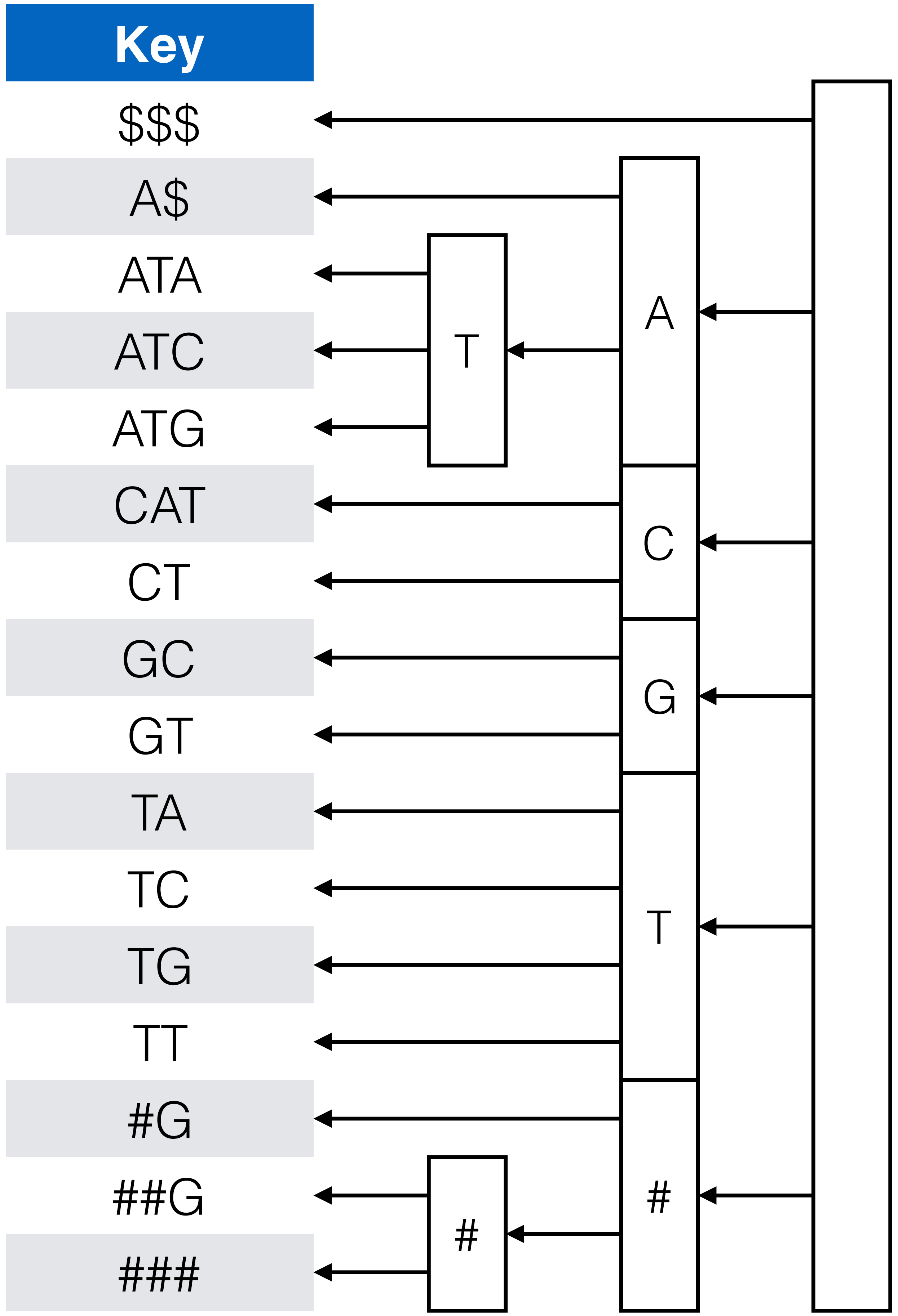
Maximal exact matches

- 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.



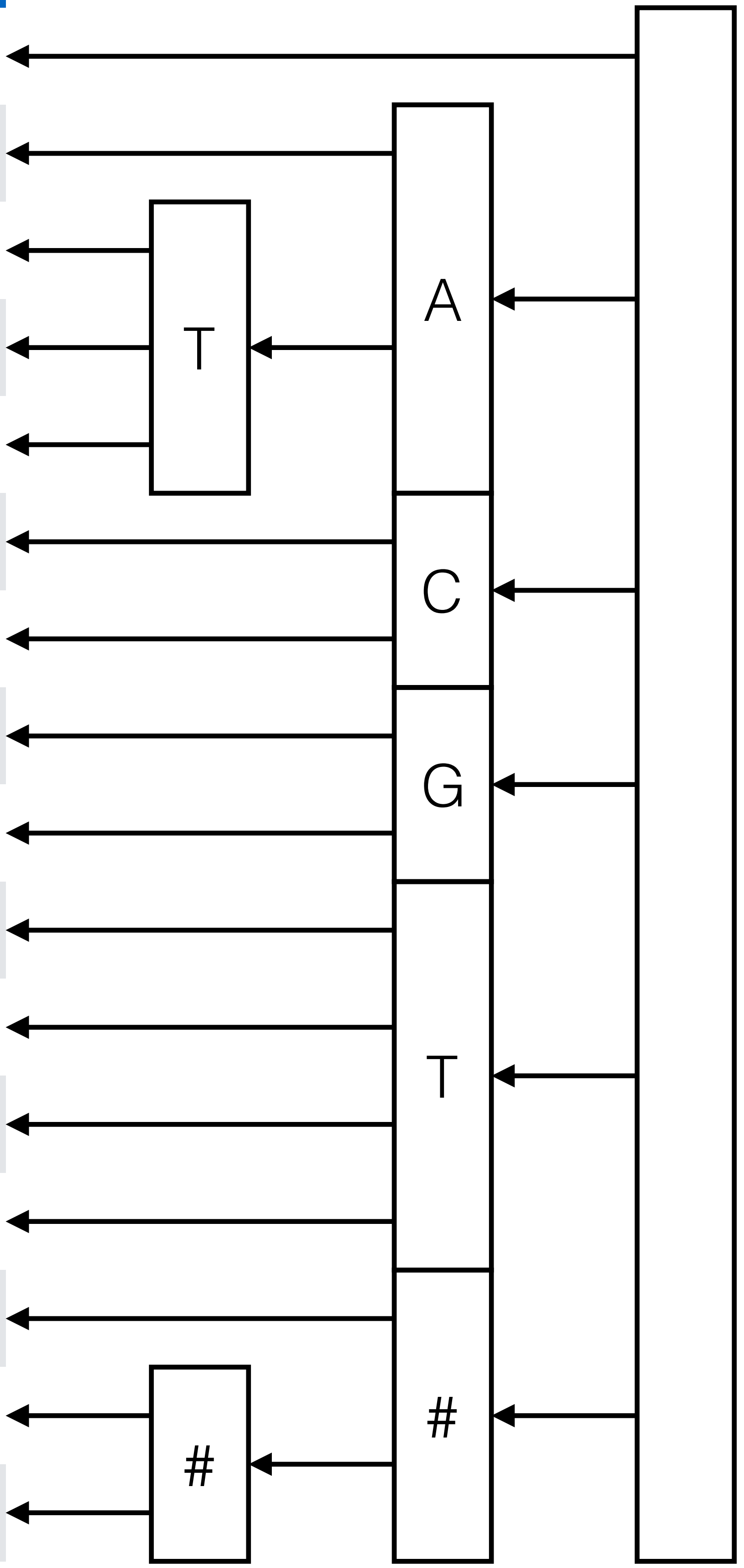
LCP[i…j] is an LCP-interval at depth d, if:

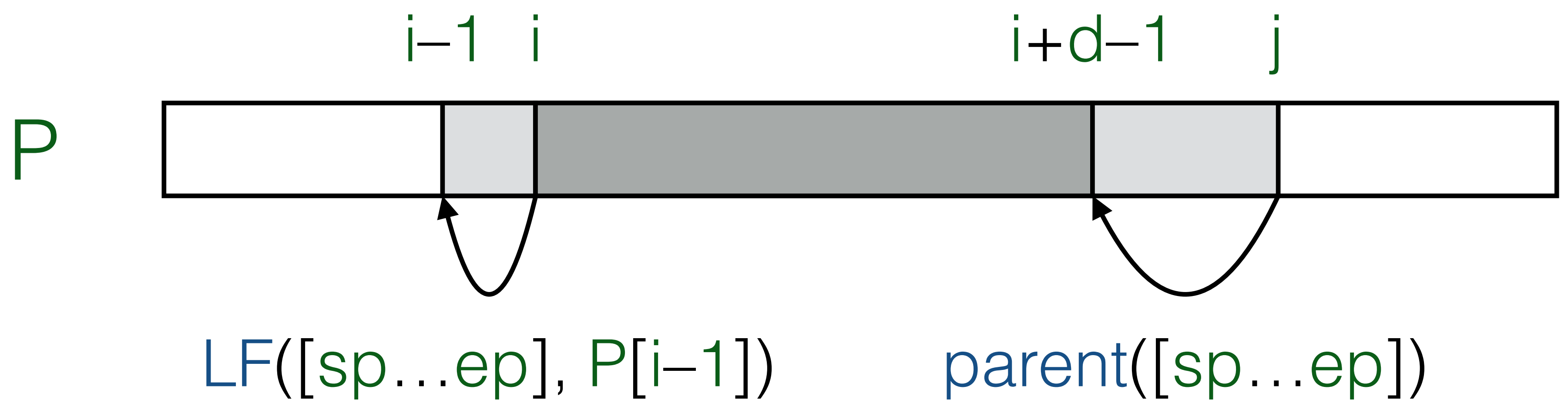
- LCP[i] < d;
- LCP[j+1] < d;
- LCP[i+1…j] ≥ d; and
- LCP[i+1…j] contains value d.

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
CAT	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.

Suffix tree implementation

- The LCP array requires 5n to 7n bits, depending on the number of doubling steps.
- To support NSV / PSV / RMQ, we build a 64-ary range minimum tree over the array.
- The result is a simple and fast compressed suffix tree, which could also be useful with sequences after some modifications.

Counting queries

- Determine the number of distinct start nodes of the paths matching the pattern.
- The length of the BWT range tells this in text indexes: each pointer in the suffix array is unique.
- A node of a path graph may contain multiple pointers to the original graph, and multiple nodes may contain the same pointer.

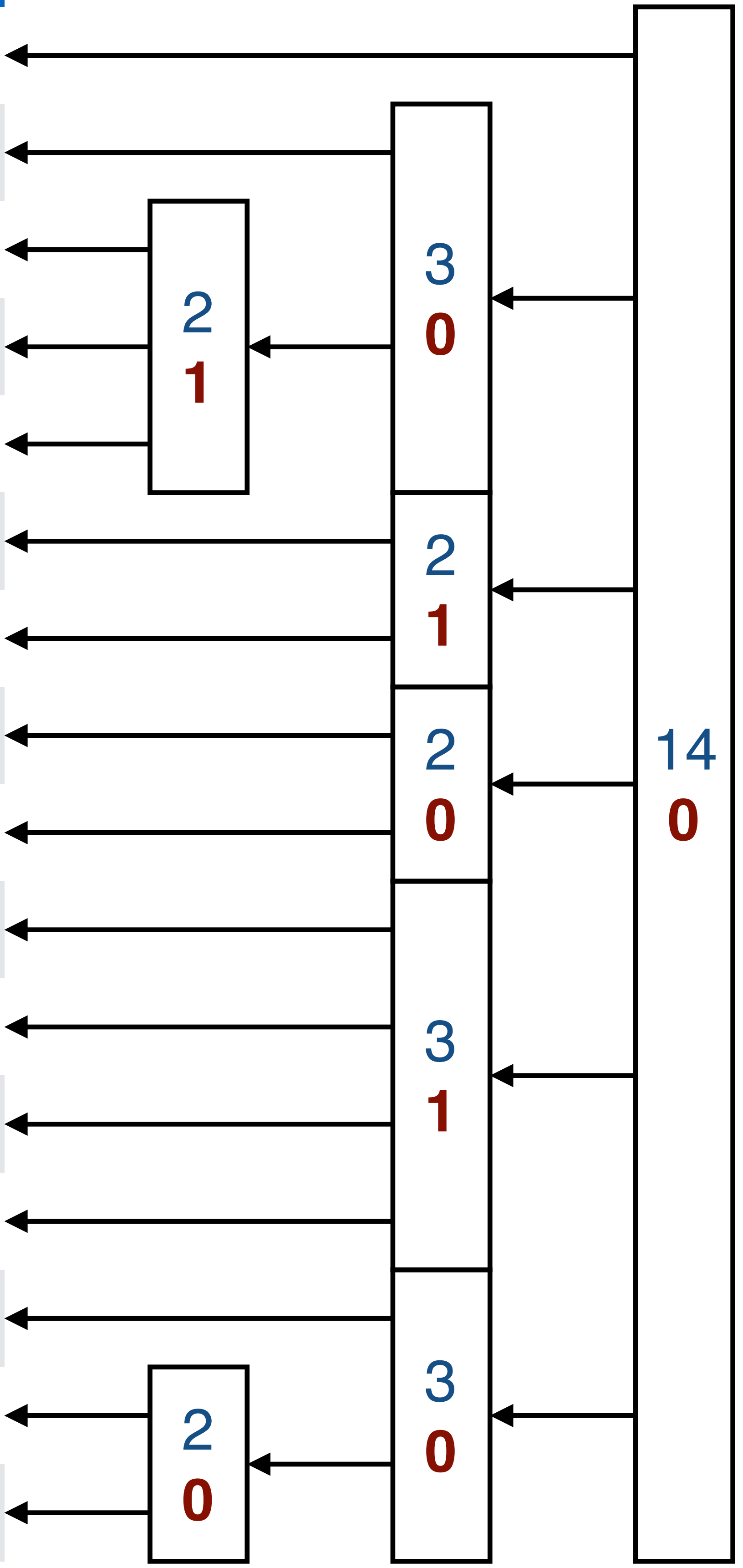
Determine the number of distinct and **redundant** values for each internal node of the suffix tree.

Traverse the tree in inorder and write down the number of redundant matches on the first visit to each node.

Lexicographic range [sp…ep]: inorder range [sp…ep–1].

count([sp…ep]): the total number of values in the lexicographic range minus the number of redundant values in the inorder range.

Key	Value	Red.
$$$	11	0
A$	10	0
ATA	7	1
ATC	3	0
ATG	3	0
CAT	2, 6	1
CT	2	0
GC	1	0
GT	8	0
TA	9	1
TC	5	0
TG	5	0
TT	4	0
#G	0	0
##G	0:1	0
###	0:2	–



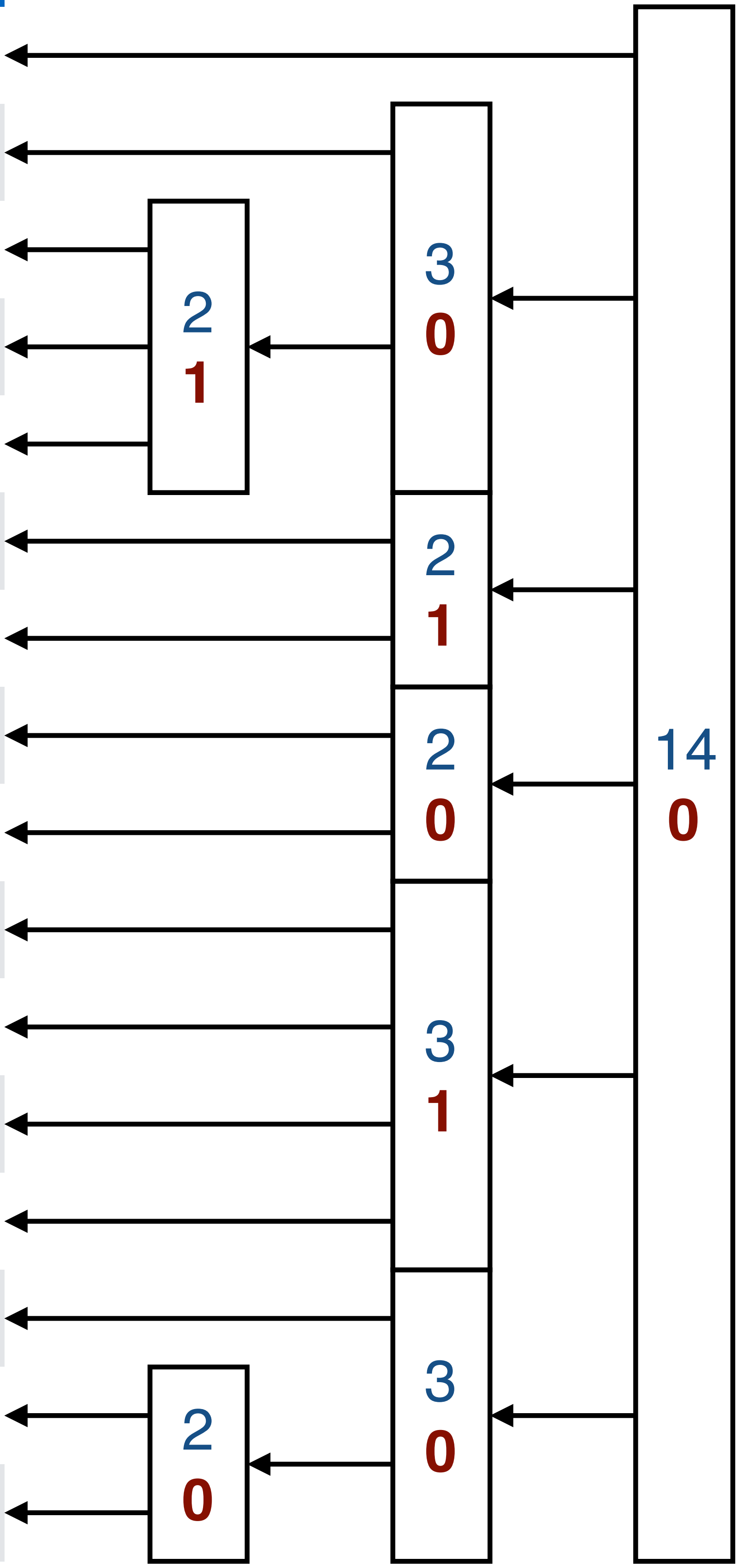
count([sp…ep]): the **total** number of values in the lexicographic range **minus** the number of **redundant** values in the inorder range.

Encode the integer arrays in **unary** to compute range sums with select queries.

Sadakane: **Succinct data structures for flexible text retrieval systems**. JDA, 2007.

The bitvectors are often highly compressible (Gagie et al: **Document Counting in Compressed Space**. DCC 2015).

Key	Value	Red.
$$$	11	0
A$	10	0
ATA	7	1
ATC	3	0
ATG	3	0
CAT	2, 6	1
CT	2	0
GC	1	0
GT	8	0
TA	9	1
TC	5	0
TG	5	0
TT	4	0
#G	0	0
##G	0:1	0
###	0:2	–



Path length	16→32	16→64	16→128
Kmers Nodes	6.20G 4.37G	16.7G 5.24G	116G 5.73G
Index with extensions	9.97 → 13.6 GB 18.9 bits / kmer	9.19 → 14.1 GB 7.27 bits / kmer	9.17 → 15.2 GB 1.12 bits / kmer
Construction: Time Memory Disk	8.62 h 63.1 GB 387 GB	12.1 h 55.9 GB 415 GB	16.1 h 56.6 GB 478 GB
I/O: Read Write	1.37 TB 0.88 TB	2.03 TB 1.51 TB	2.78 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

Index	k	find()	locate()	parent()	count()
GCSA	16	18.3 µs	12.1 µs	0.41 µs	0.87 µs
GCSA	32	37.9 µs	11.1 µs	0.28 µs	0.38 µs
GCSA	64	76.1 µs	5.76 µs	0.26 µs	0.28 µs
GCSA	128	149 µs	5.98 µs	0.26 µs	0.26 µs
csa_wt	16	6.06 µs	2.46 µs	–	–
csa_wt	32	10.9 µs	2.19 µs	–	–
csa_wt	64	17.2 µs	1.91 µs	–	–
csa_wt	128	23.6 µs	3.48 µs	–	–

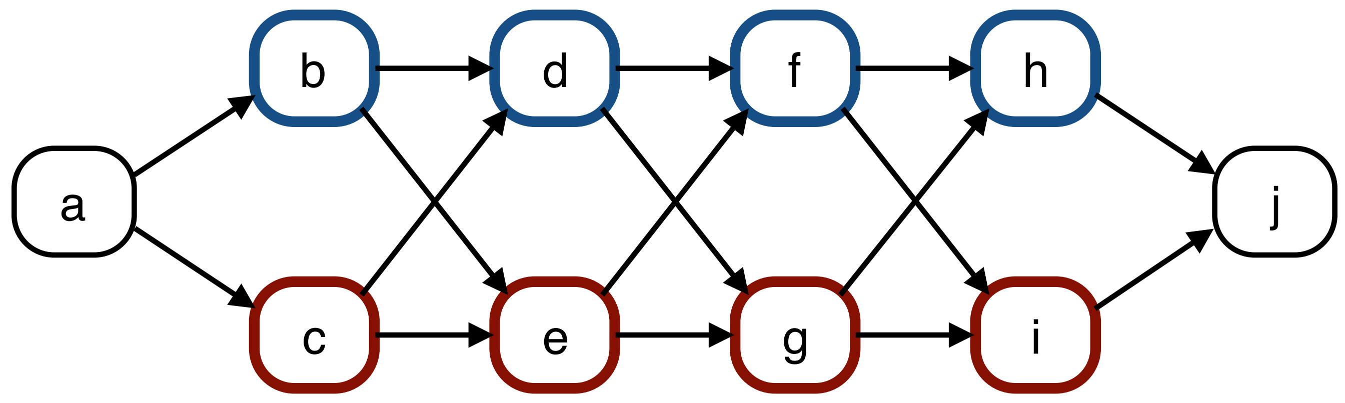
GCSA2 vs a similar FM-index from SDSL.
Patterns extracted from the non-pruned variation graph.
Time per query / distinct occurrence.

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

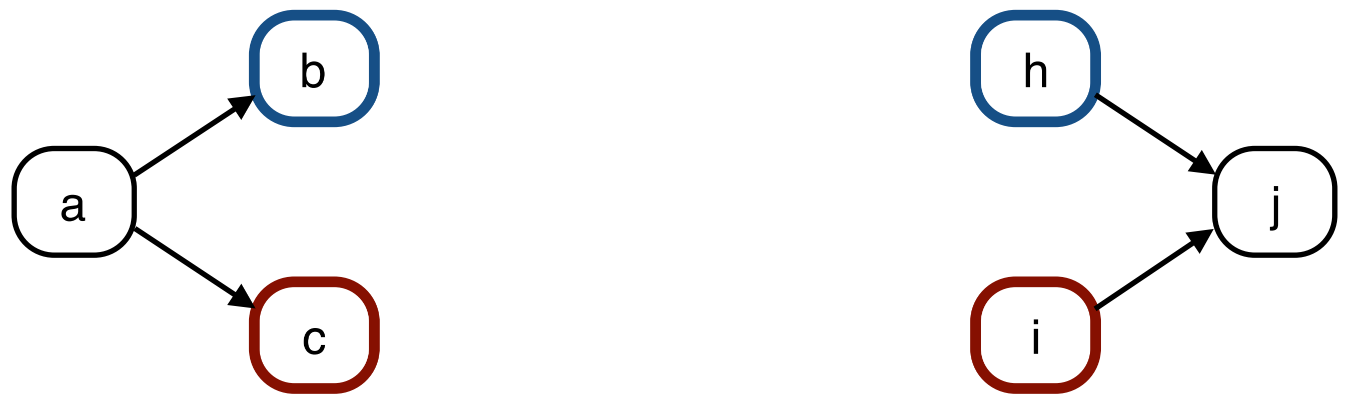


```
vg mod -p -l 16 -e 4
```

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.

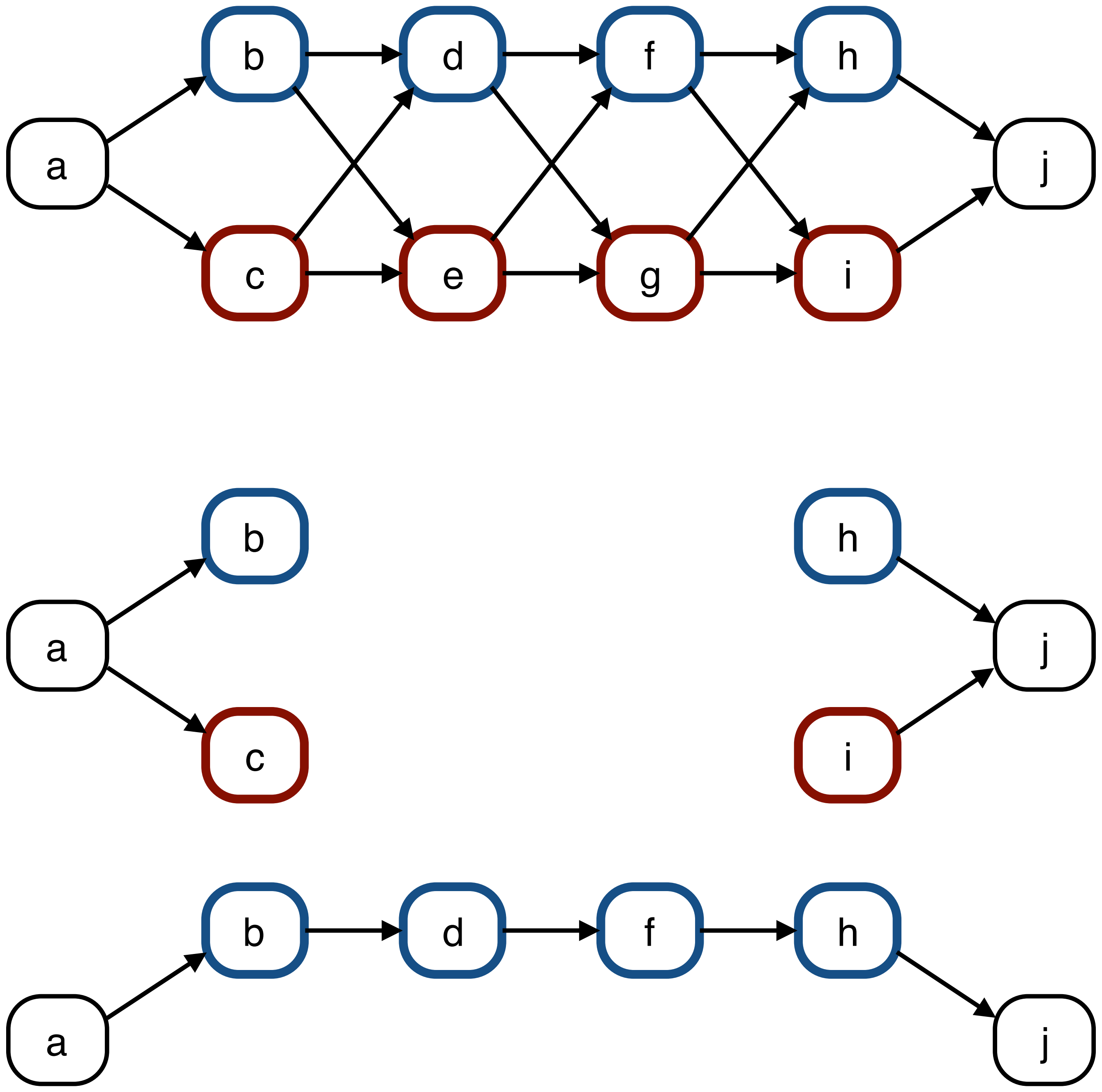
Indexing subgraphs

GCSA2 construction handles each chromosome (disjoint subgraph) separately.

We can also index overlapping subgraphs (e.g. a pruned variation graph and the reference path).

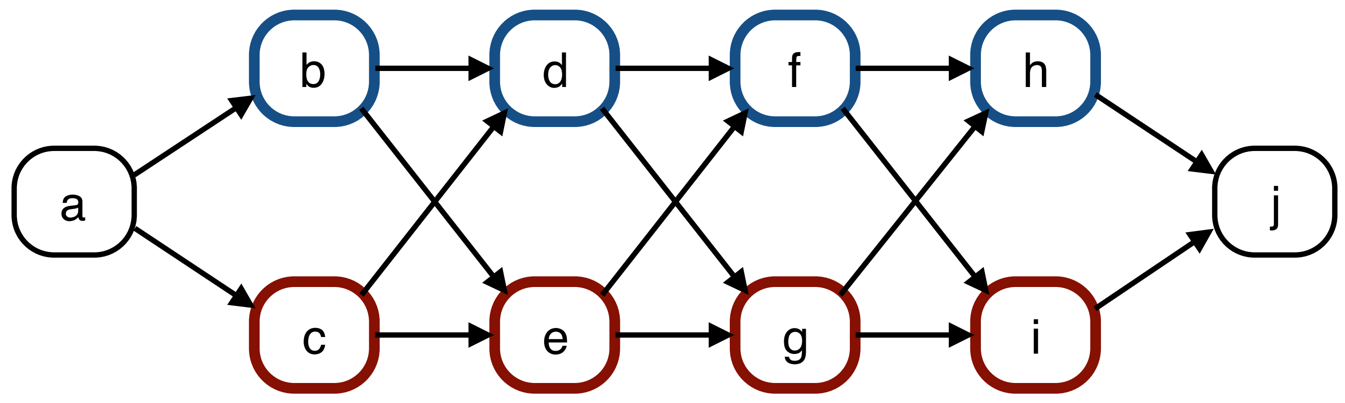
- Guarantees that the entire genome is indexed.

- Redundant paths can make index construction more expensive.



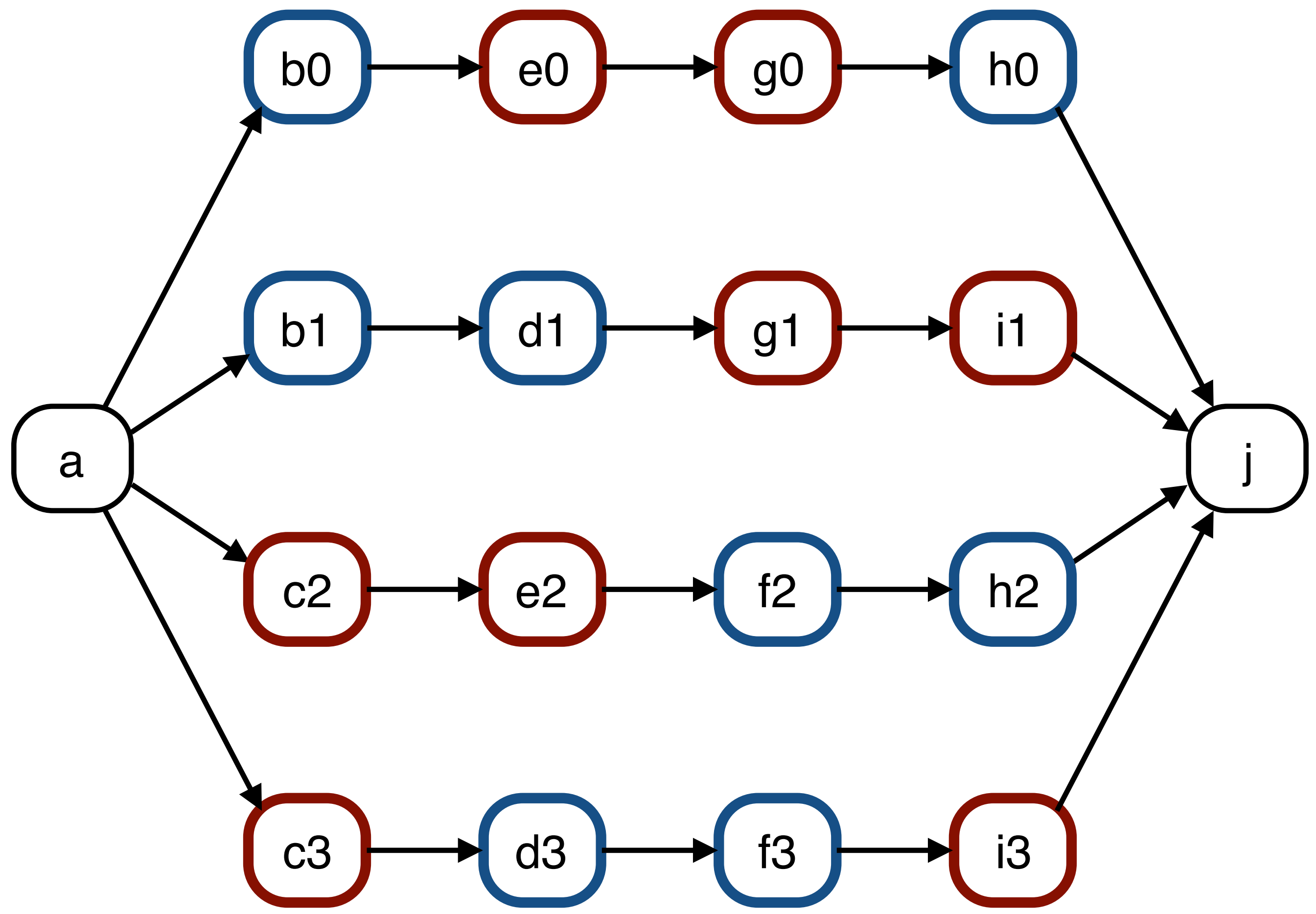
Indexing haplotypes

Duplicate nodes to 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. (How to do it efficiently?)



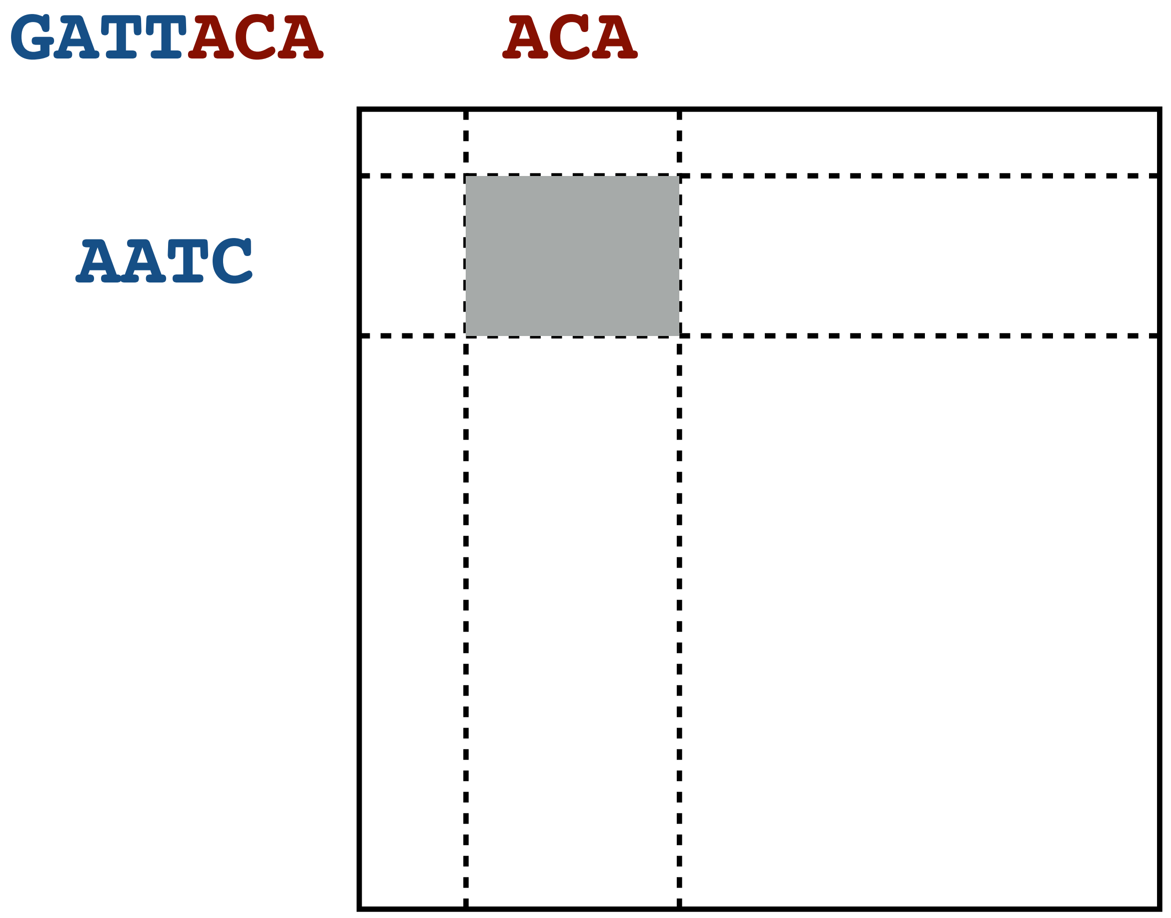
Hypertext Index

Hypertext index

We build an FMD-index for the node labels of a variation graph. In order to match for a pattern, we:

- Search for the pattern and its reverse complement.
- Combine the partial match for each suffix with the partial match for the reverse complement of the corresponding prefix with a range query in the edge matrix.
- Matches crossing multiple edges can also be found, though it may not be practical.

This is similar to LZ-indexes and grammar-compressed indexes.



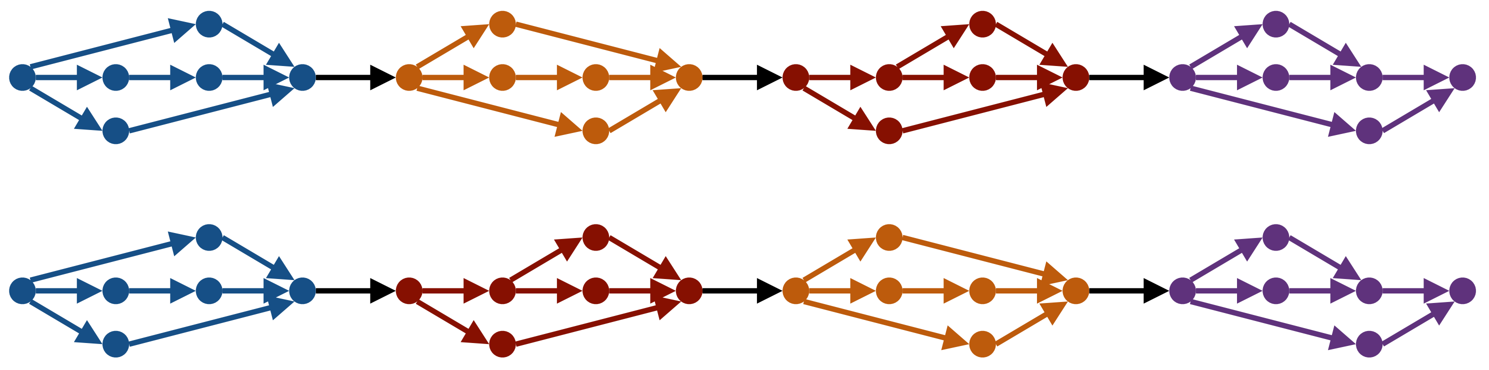
Chris Thachuk: Indexing Hypertext. JDA, 2013.

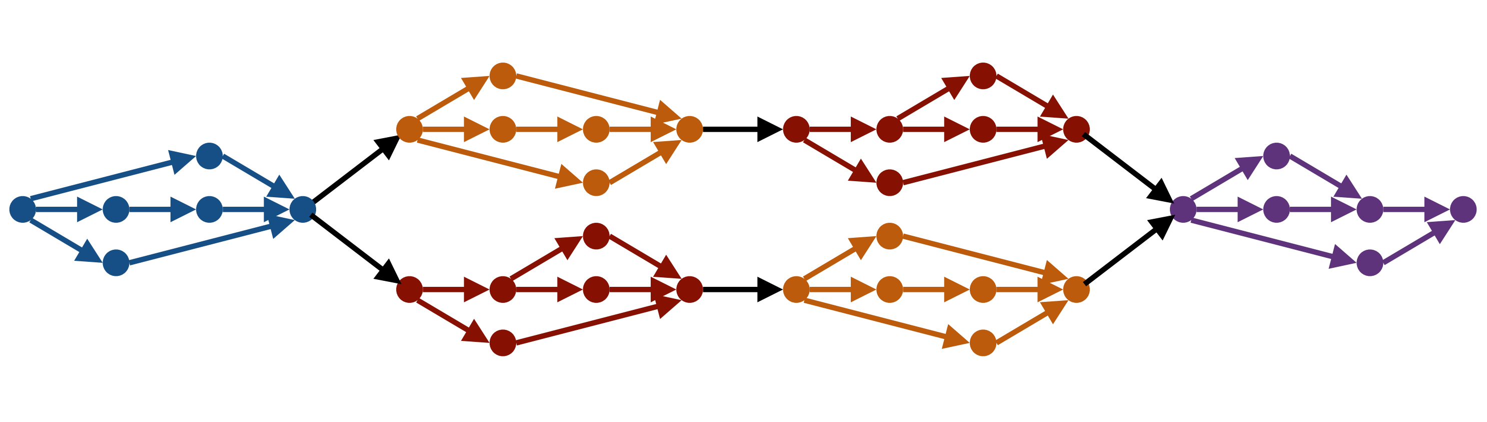
Potential recombinations

- We can use the hypertext index to add new edges to a path graph indexed by GCSA2.
- This can be used for example to add potential recombinations to complex regions, where we would otherwise index only known haplotypes.
- Include only edges from the reverse complement strand to the forward strand in the edge matrix.
- (I have not decided whether this is worth the effort.)

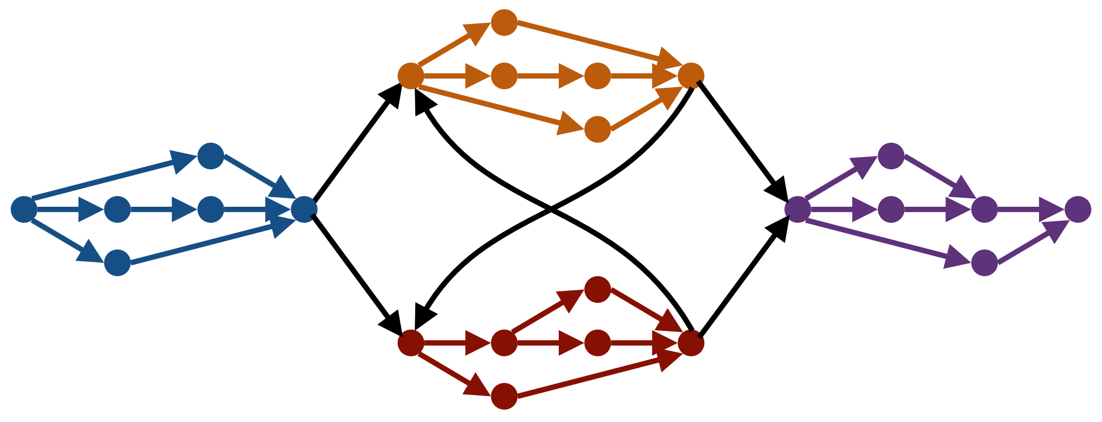
Representing rearrangements

We may have the same sequence or even subgraph in different positions.





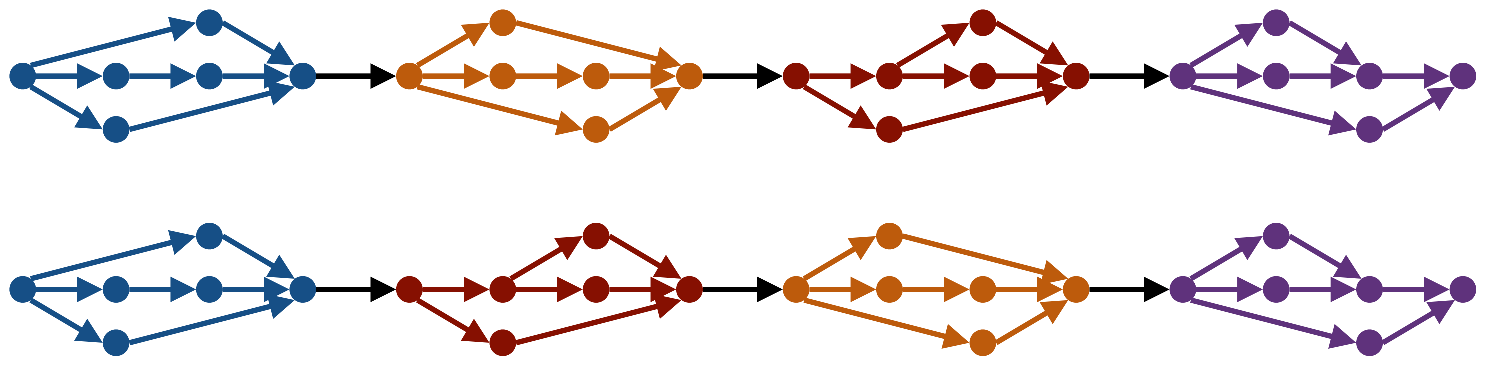
Duplicated subgraphs



Unsupported paths

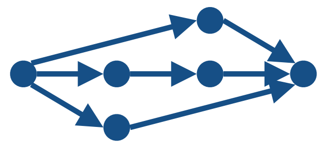
Neither option works well in a reference genome.

We may need something stronger than graphs, which correspond to regular languages.

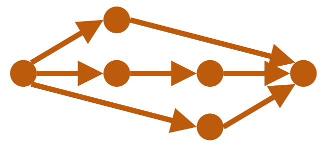


$S \rightarrow ABCD \mid ACBD$

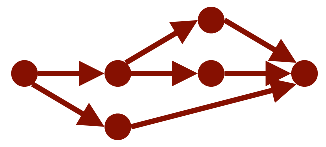
A →



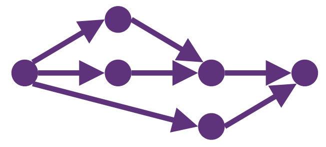
B →



C →



D →



	A	B1	B2	C1	C2	D
A	0	1	0	0	1	0
B1	0	0	0	1	0	0
B2	0	0	0	0	0	1
C1	0	0	0	0	0	1
C2	0	0	1	0	0	0
D	0	0	0	0	0	0

We can use the hypertext index, as long as the grammar is non-nested. For more general grammars, we may need a GCSA-like generalization of grammar-compressed indexes.

Embedding haplotypes

A series of theoretical papers on indexing similar sequences by embedding them in a graph.

Na et al: Suffix Tree of Alignment: An Efficient Index for Similar Data. IWOCA 2013.

Na et al: Suffix Array of Alignment: A Practical Index for Similar Data. SPIRE 2013.

Na et al: FM-index of alignment: A compressed index for similar strings. TCS, 2015.

Na et al: FM-index of Alignment with Gaps. arXiv: 1606.03897, 2016.

From alignment to graph

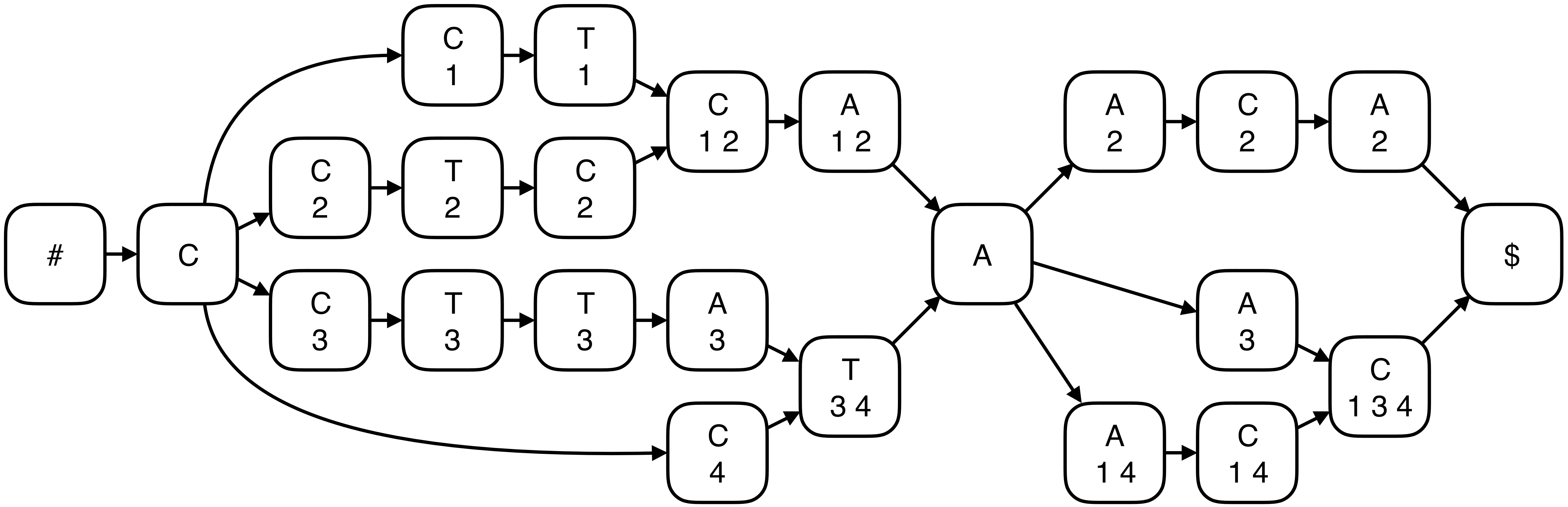
- Start from aligned sequences and split the alignment into shared and **diverging** segments.
- Move the **shortest globally unique suffix** of each shared segment to the next diverging segment.
- Justify the diverging segments to the right.

```
#CCT C-A- AAC C $
#CCT CCA- AAC A $
#CCT T-AT AAC - $
#CCT ---- AAC C $

#C CTC-A- A ACC $
#C CTCCA- A ACA $
#C CTT-AT A AC- $
#C CT---- A ACC $

#C --CTCA A ACC $
#C -CTCCA A ACA $
#C -CTTAT A -AC $
#C ----CT A ACC $
```

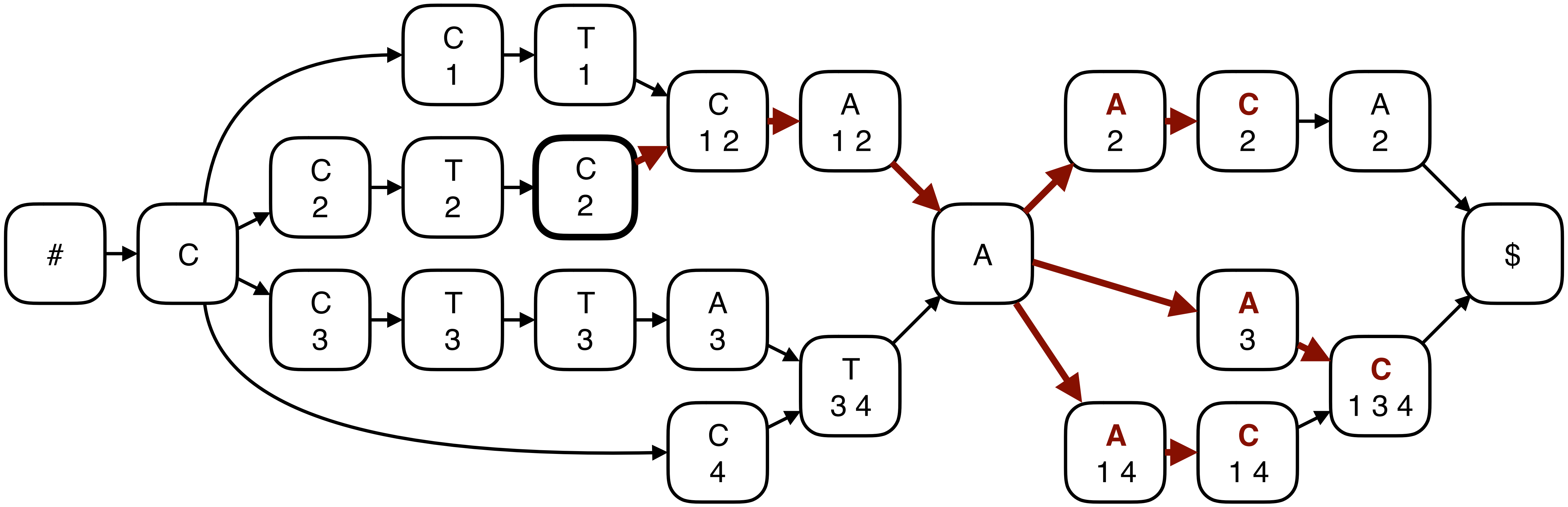
From alignment to graph

Each distinct suffix of each diverging segment becomes a node.

```
1: #C --CTCA A ACC $
2: #C -CTCCA A ACA $
3: #C -CTTAT A -AC $
4: #C ----CT A ACC $
```

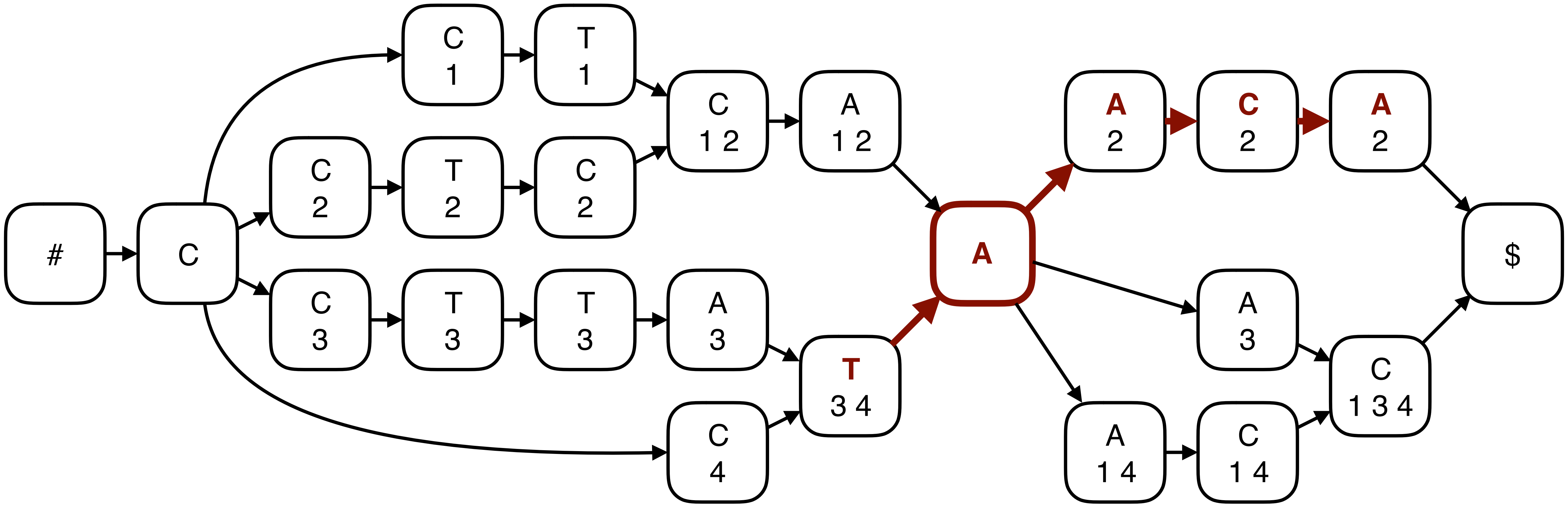


Indexing the graph



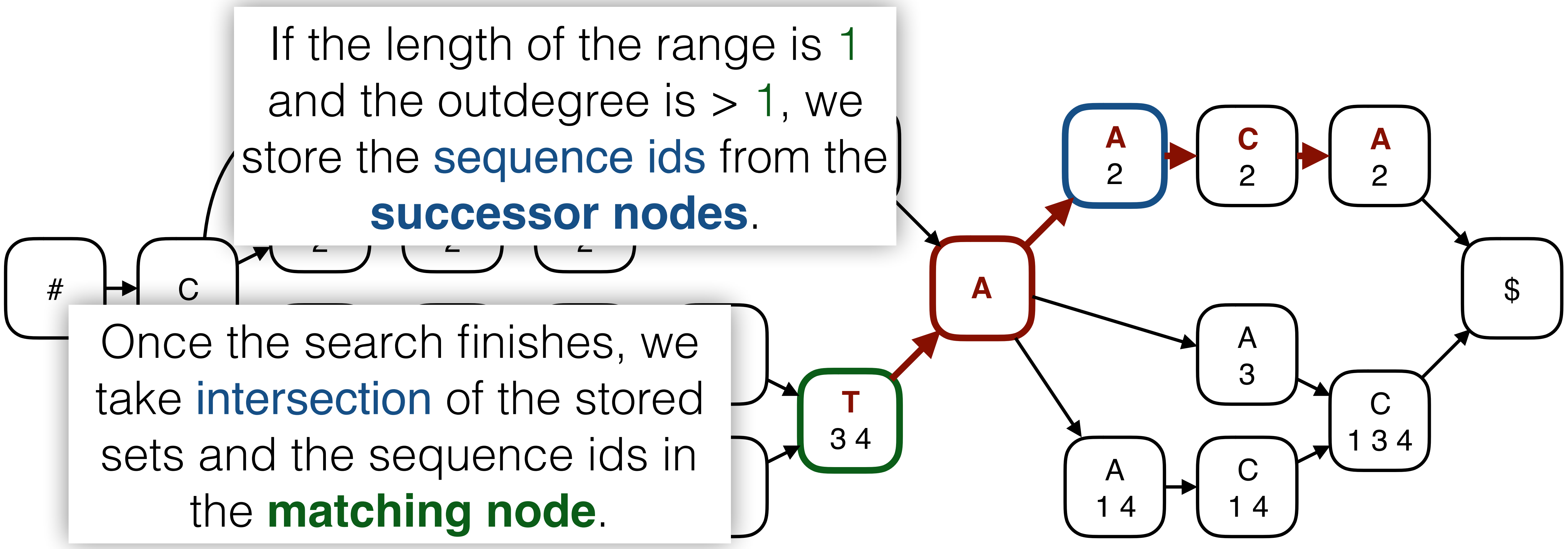
- The graph is reverse deterministic.
- All paths from a node share a prefix that ends with a **globally unique substring**.
- Hence the graph is a path graph and we can encode it with GCSA.

Indexing the sequences



- Paths that pass from one diverging segment to the next may be false positives, if they continue past the globally unique substring.
- The length of the lexicographic range is 1, once backward searching reaches the **shared segment**.
- Shared segment ends if and only if a node has outdegree > 1.

Indexing the sequences

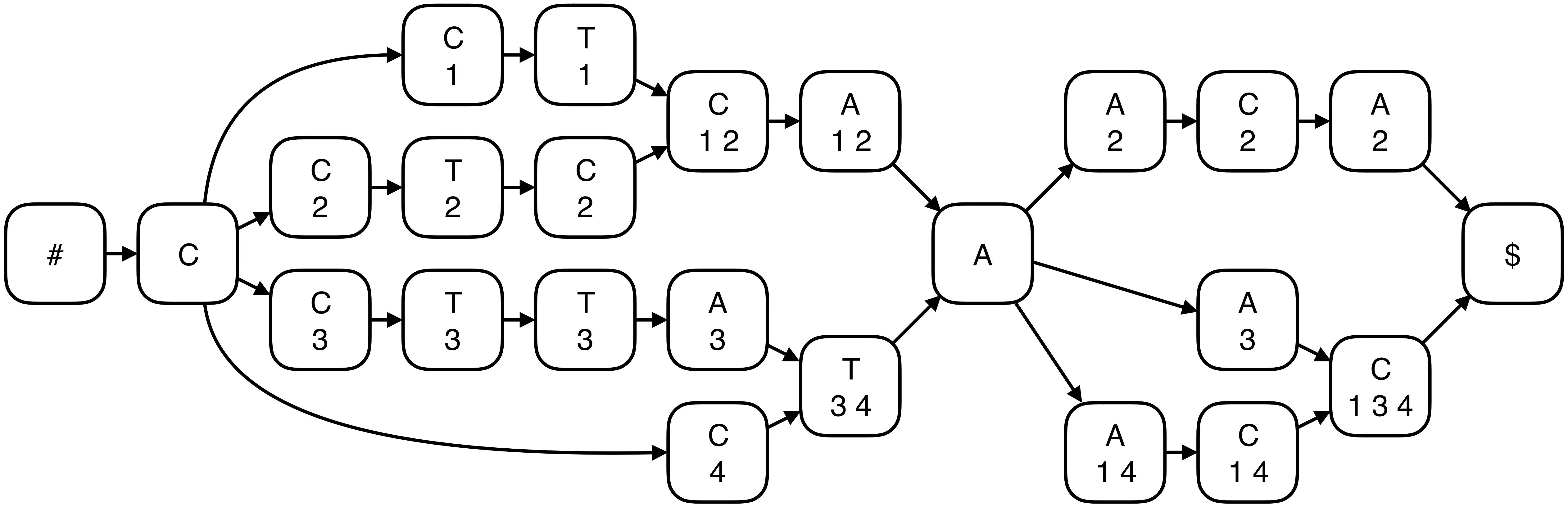


If the length of the range is 1 and the outdegree is > 1, we store the sequence ids from the **successor nodes**.

Once the search finishes, we take intersection of the stored sets and the sequence ids in the **matching node**.

- Paths that pass from one diverging segment to the next may be false positives, if they continue past the globally unique substring.
- The length of the lexicographic range is 1, once backward searching reaches the **shared segment**.
- Shared segment ends if and only if a node has outdegree > 1.

Indexing graph and sequences



- We have indexed the graph in a way that allows for restricting the search to the original sequences.
- This depends on the specific properties of the graph.
- Could we do the same efficiently with any GCSA?

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 long enough queries to map short reads in one piece. It ignores unobserved recombinations in complex regions.

- 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. https://github.com/jltsiren/gcsa2