Indexing Graphs for Path Queries

Jouni Sirén
Wellcome Trust Sanger Institute

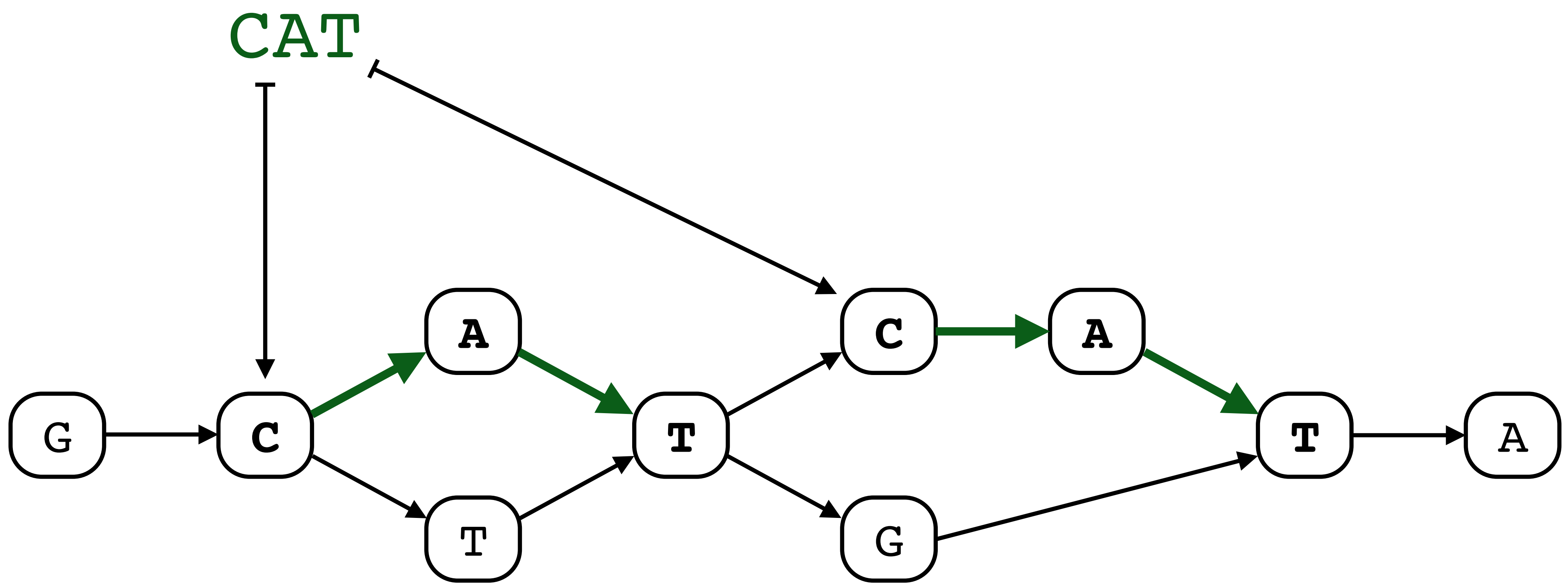
J. Sirén, N. Välimäki, V. Mäkinen: Indexing Graphs for Path Queries with Applications in Genome Research. WABI 2011, TCBB 2014.

A. Bowe, T. Onodera, K. Sadakane, T. Shibuya: Succinct de Bruijn Graphs. WABI 2012.

J. Sirén: GCSA2. https://github.com/jltsiren/gcsa2

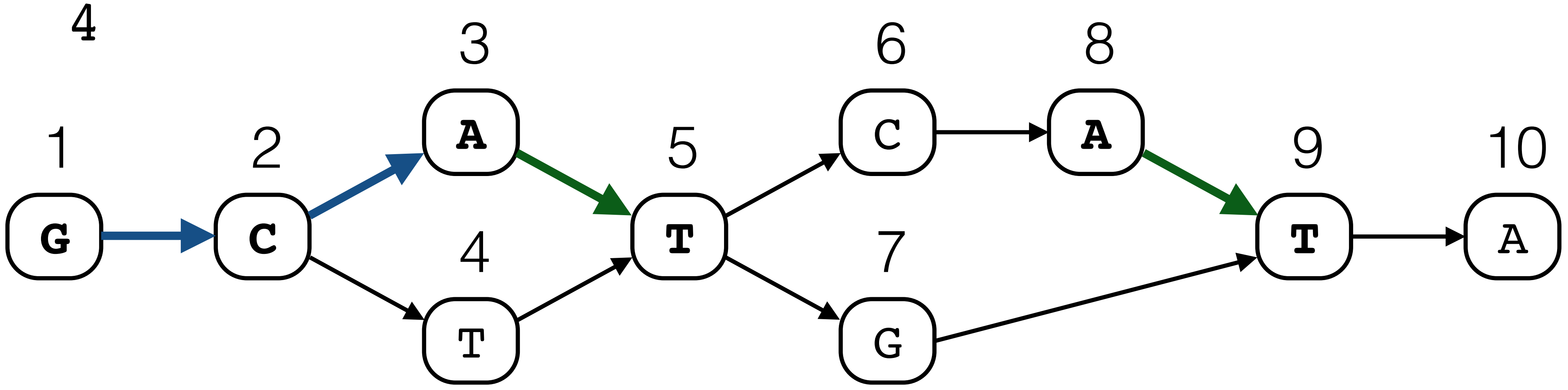
Given a graph where paths are labeled by strings, a path index is a text index for the strings.

A path query finds the (start nodes of) the paths labeled by a kmer.

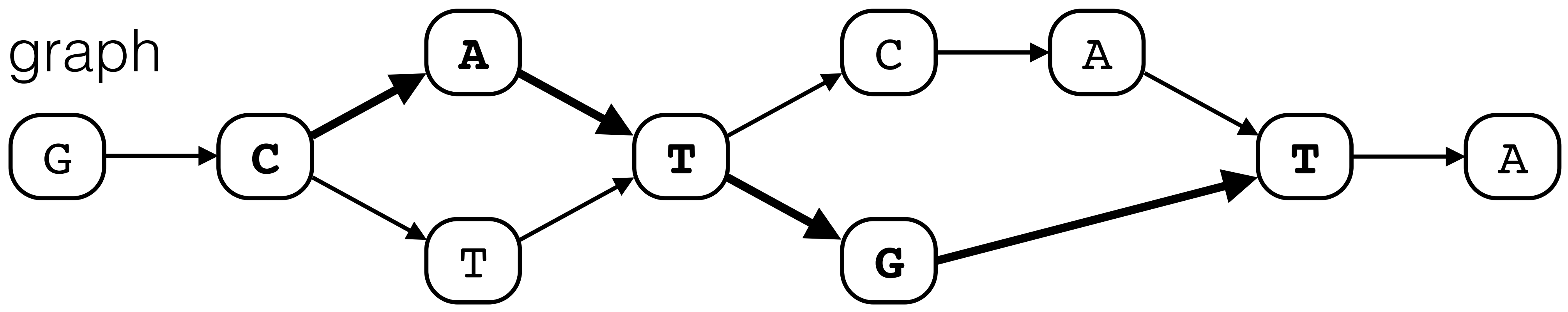


A$$	10
ATA	**8**
ATC	**3**
ATG	**3**
CAT	2, 6
CTT	2
GCA	**1**
GCT	1
GTA	7
TA$	9
TCA	5
TGT	5
TTC	4
TTG	4

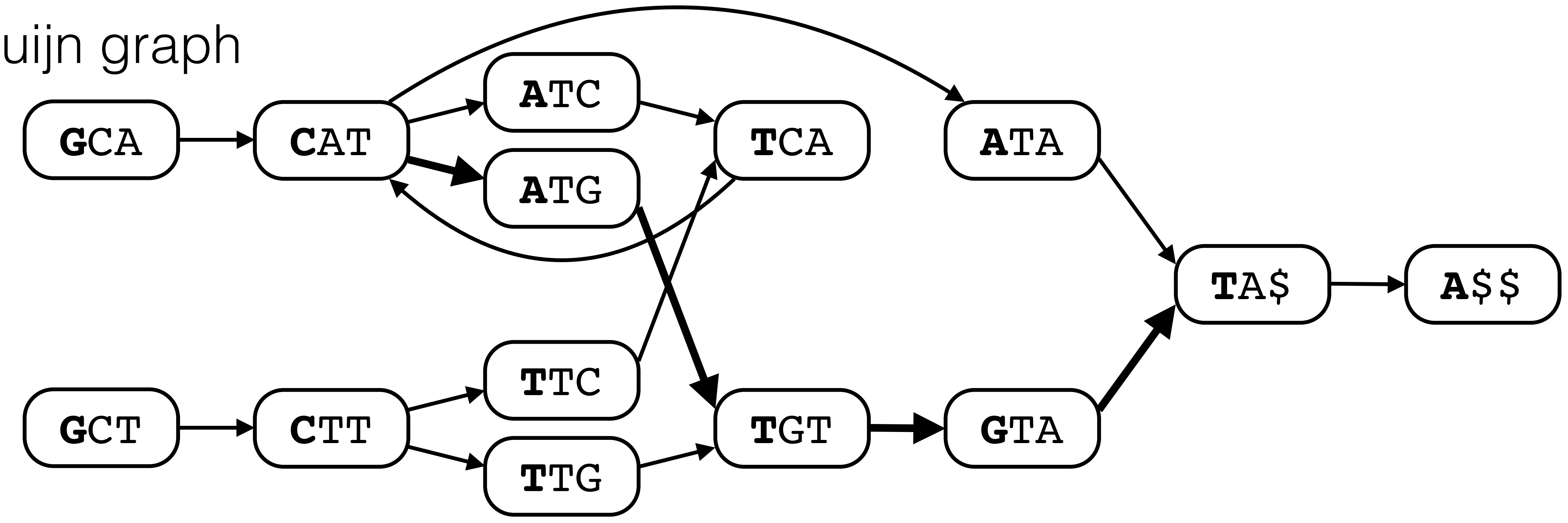
- A kmer index based on a hash table supports queries of length k efficiently.
- If we sort the kmers, we can use them as a suffix array-like index for shorter queries.
- The kmer index can also simulate a de Bruijn graph.



Original graph

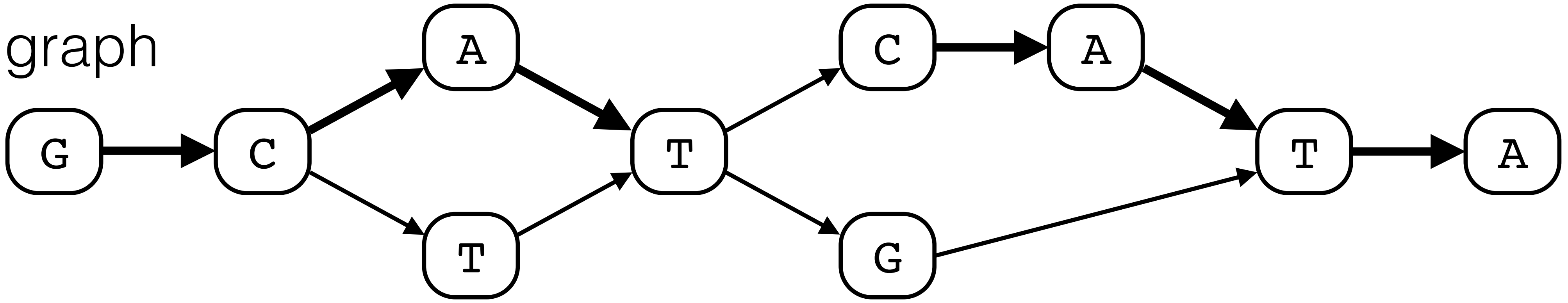


de Bruijn graph

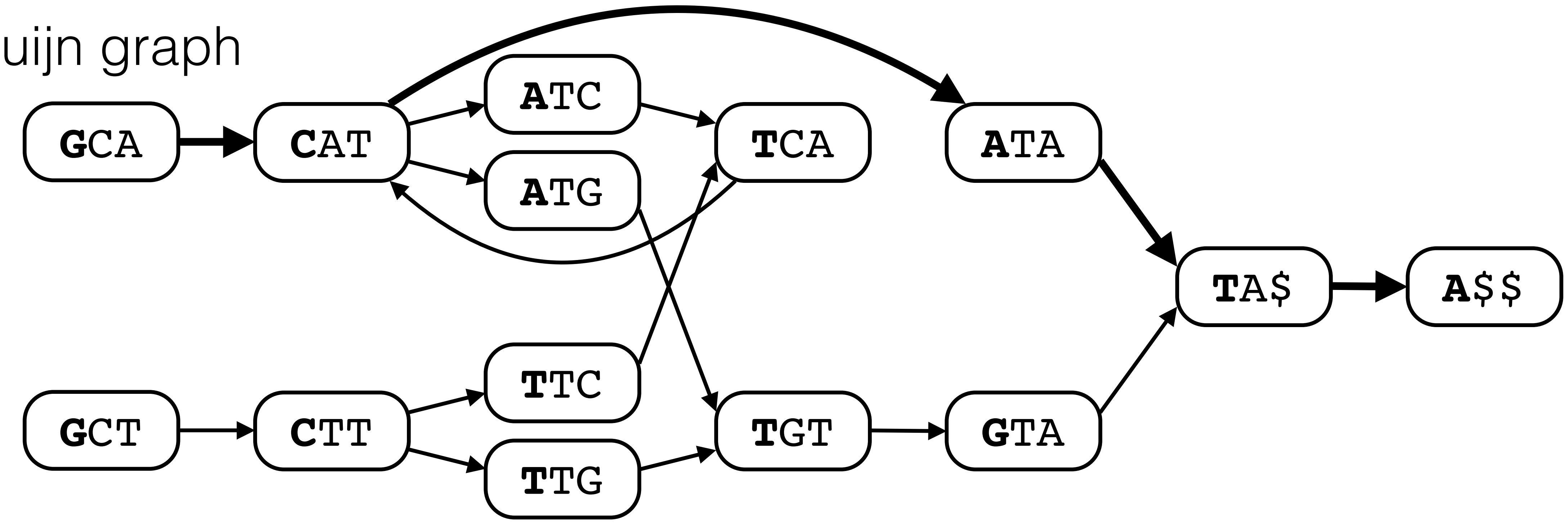


We can search for longer patterns by representing the kmer index as a de Bruijn graph.

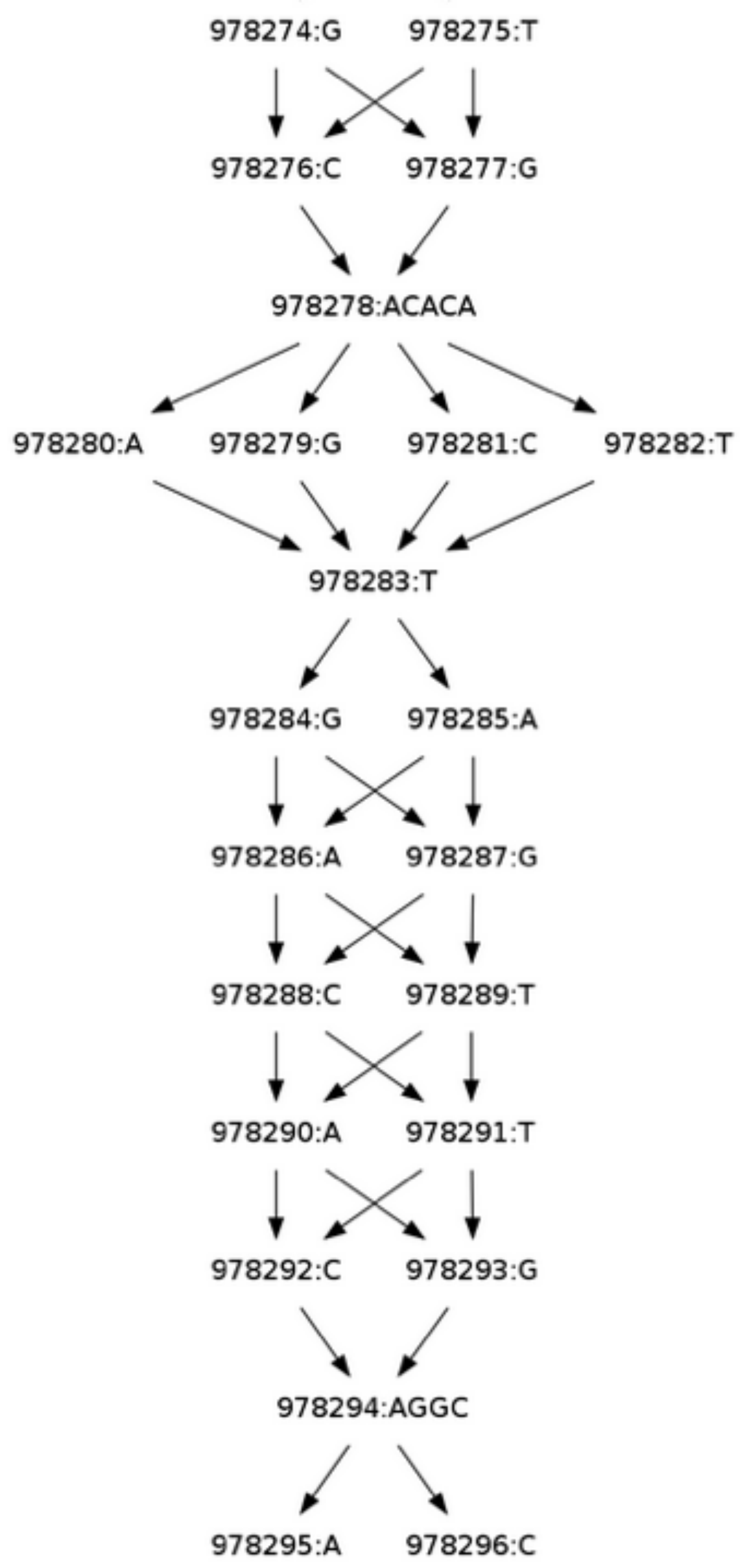
Original graph



de Bruijn graph



The results of long queries must be verified in the original graph to avoid false positives.

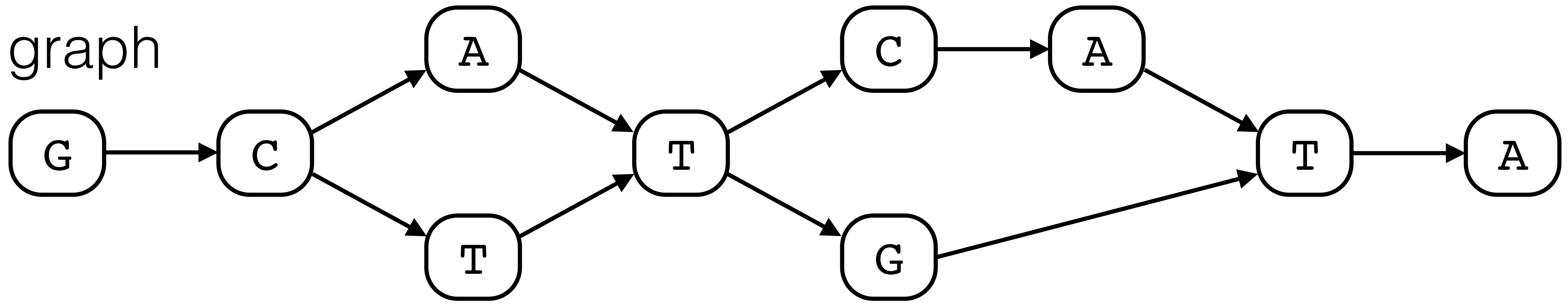


Some parts of the original graph may have too many paths through them. Those parts must be pruned before indexing.

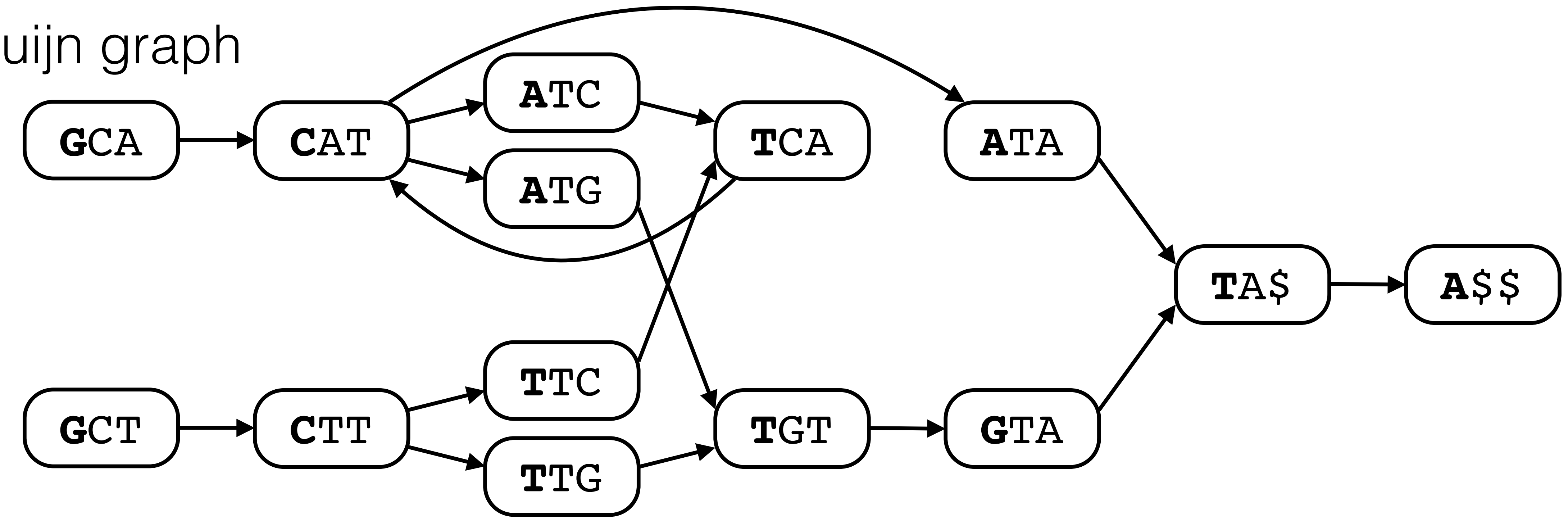
The de Bruijn graph can also be pruned by merging the nodes with a common prefix of the label, if:

1. the shorter label uniquely defines the start node in the original graph; or
2. the start nodes cannot be distinguished by length-k extensions of the label.

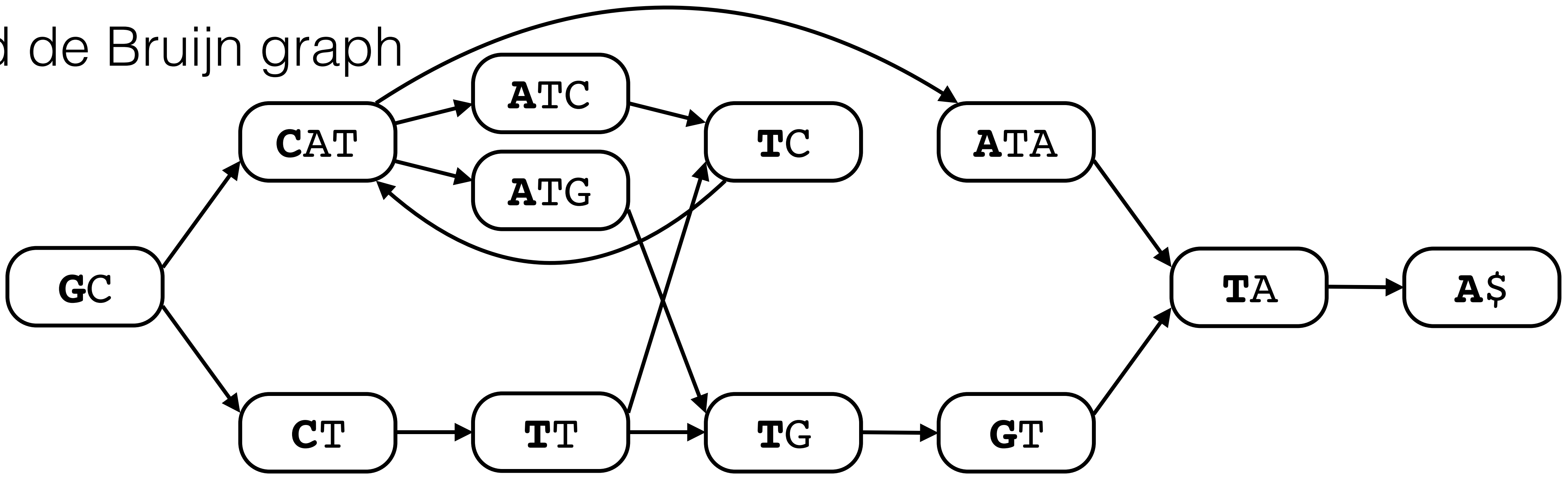
Original graph

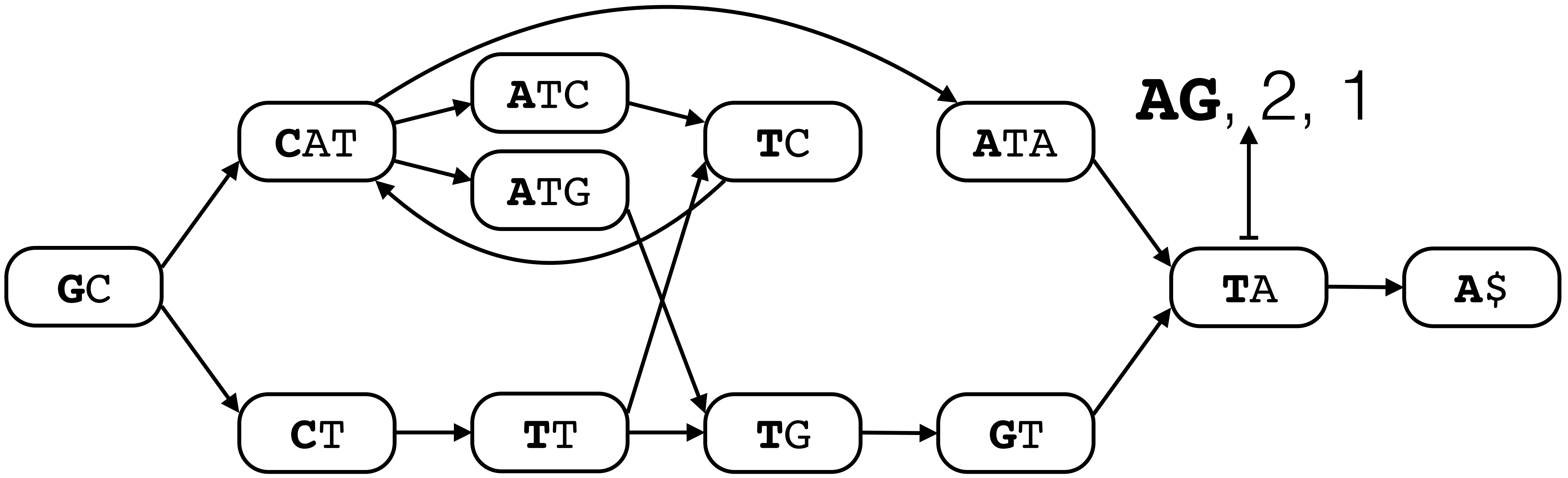


de Bruijn graph



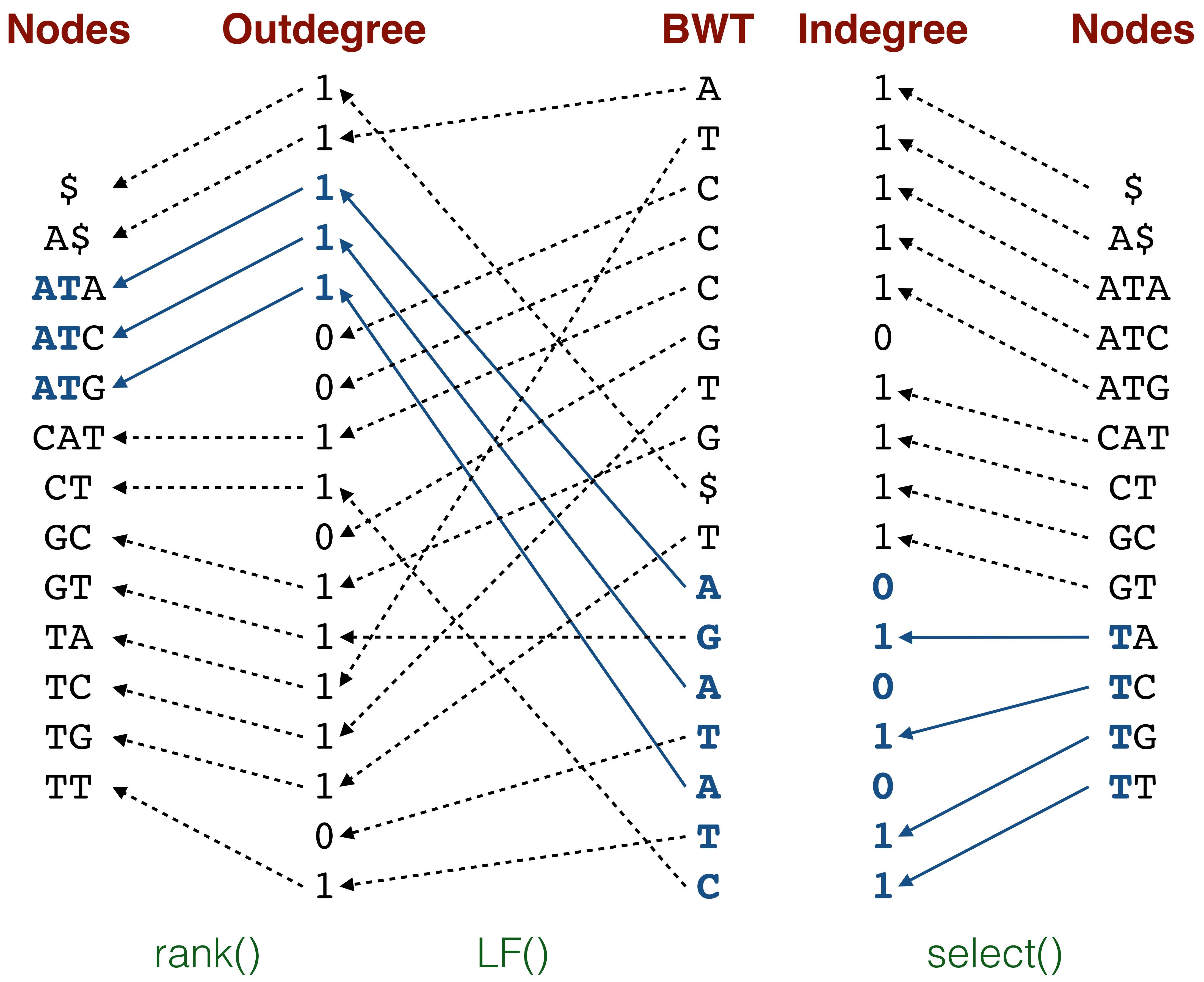
Pruned de Bruijn graph





We store predecessor labels, indegree, and outdegree for each node. For the nodes at the beginning of unary paths, we also store pointers to the original graph. Edges can be determined if the nodes are stored in sorted order.

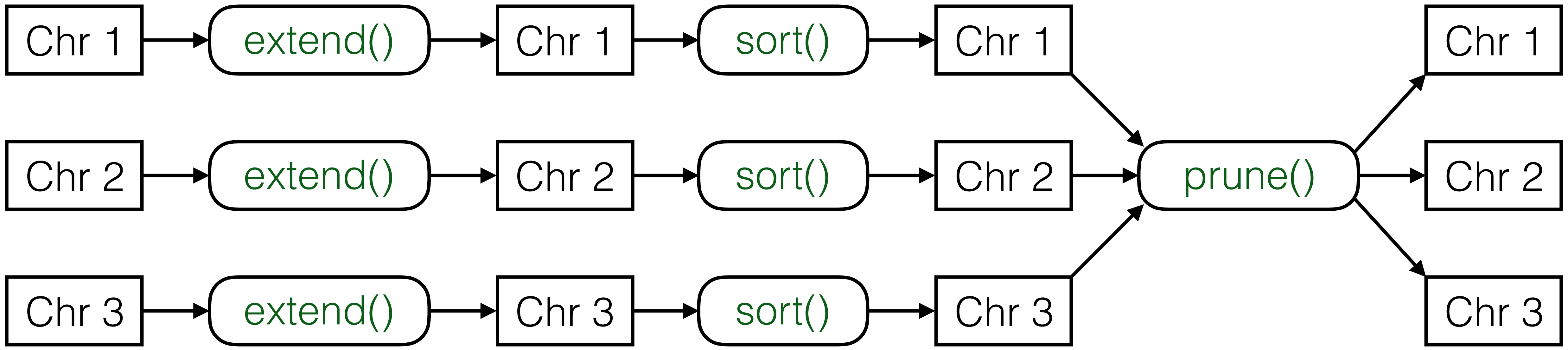
The encoding is similar to the Burrows-Wheeler transform and the FM-index. Typical space usage is 1–2 bytes/node.

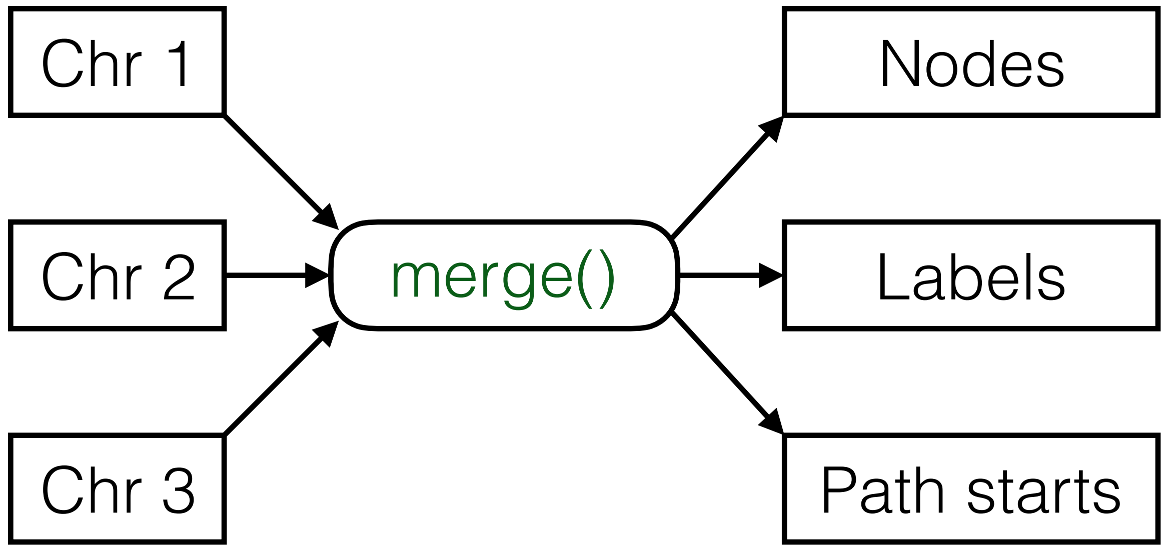


Nodes
Outdegree
BWT
Indegree
Nodes
$
A$
ATA
ATC
ATG
CAT
CT
GC
GT
TA
TC
TG
TT
1
1
1
1
1
0
0
1
1
0
1
1
1
1
1
0
1
A
T
C
C
C
G
T
G
$
T
A
G
A
T
A
T
C
1
1
1
1
1
0
1
1
1
1
0
1
0
1
0
1
1
$
A$
ATA
ATC
ATG
CAT
CT
GC
GT
TA
TC
TG
TT
rank()
LF()
select()

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.





The files are sorted by path labels. GCSA construction determines the lexicographic ranges of potential predecessors of each node in the pruned de Bruijn graph and creates an edge from each node intersecting with the range.

Path length	16→32	16→64	16→128
Nodes:			
de Bruijn graph	6.23G	16.9G	118G
Pruned	4.39G	5.27G	5.76G
Index size:			
Full index	9.99 GB	9.22 GB	9.23 GB
Without pointers	4.10 GB	4.84 GB	5.27 GB
Construction:			
Time	7.20 h	11.4 h	15.5 h
Memory	43.8 GB	43.8 GB	43.8 GB
Disk	401 GB	424 GB	489 GB
I/O:			
Read	1.43 TB	2.11 TB	2.89 TB
Write	1.05 TB	1.71 TB	2.47 TB

1000GP human variation, vg mod -p -l 16 -e 4 | vg mod -S -l 100
32 cores, 256 GB memory, distributed Lustre file system

Conclusions

- We can use pruned de Bruijn graphs encoded using the BWT to index variation graphs.
- GCSA2 is a practical implementation for whole-genome graphs and queries of length up to 128.
- The index is an FM-index: We can extend it with many techniques from text indexing literature.

Graph pruning

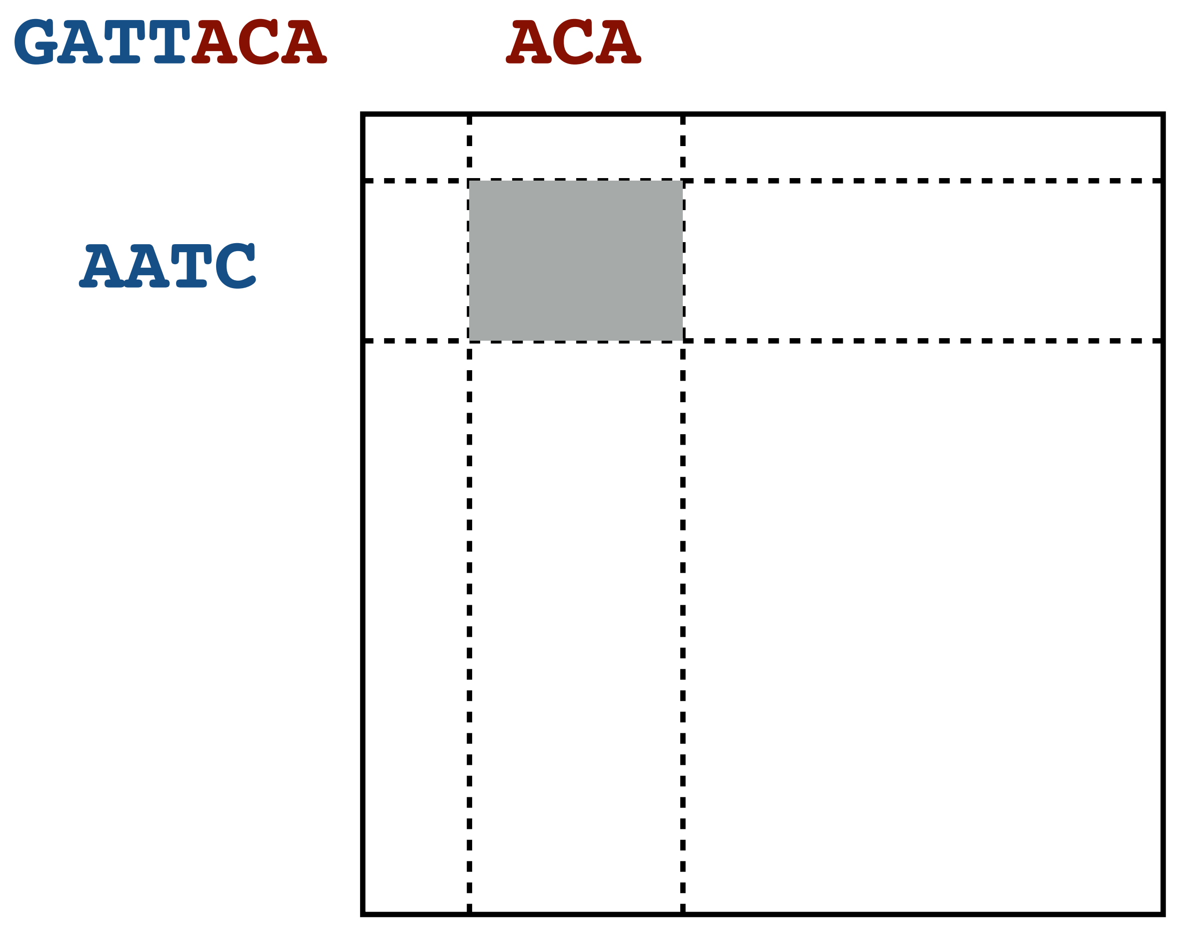
- Split the graph into two layers: primary graph and additional edges.
- Index the forward and reverse complement strands of the primary graph.
- If x is a node in the original graph, let V(x) and V'(x) be the sets forward and reverse complement paths with x as the start node.
- For each additional edge (a,b), create edges from V'(a) to V(b).

In order to match kmers with one recombination, we:

- Split the kmer into prefix-suffix pairs.
- Search for the reverse complement of the prefix and for the suffix.
- Combine the partial matches with a 2D range query over the matrix of created edges.

In practice, we search for the kmer and its reverse complement, and do k–1 range queries. This finds both forward and reverse complement occurrences.

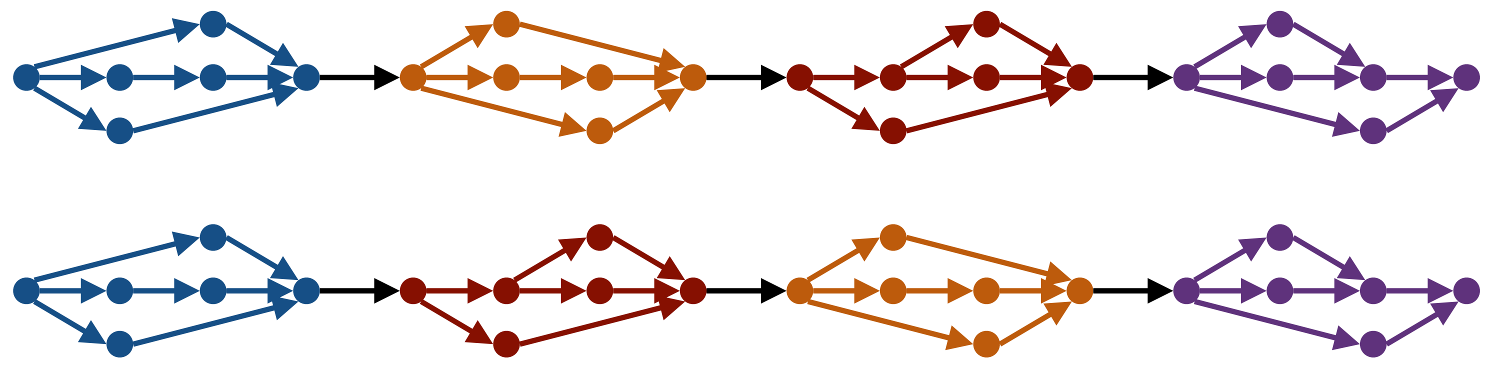
There may be false positives from paths with multiple start nodes.

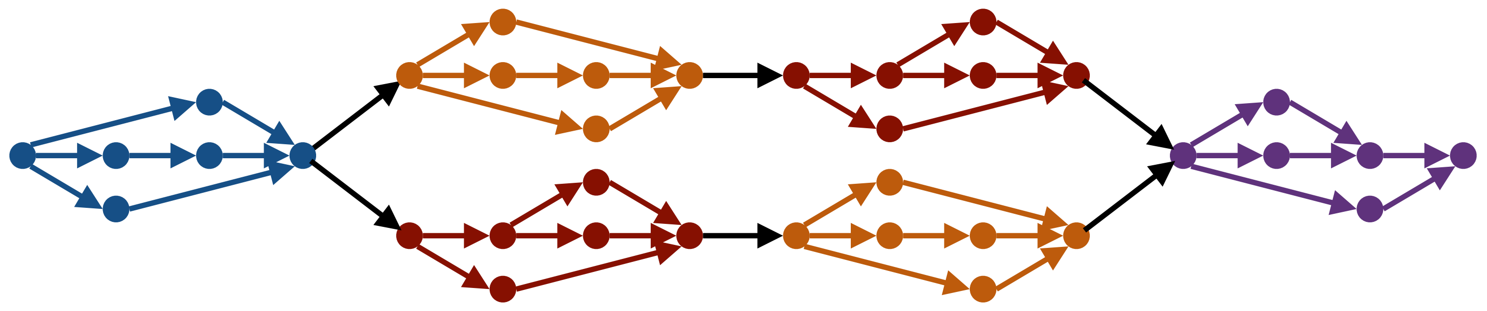


Chris Thachuk: Indexing Hypertext. JDA, 2013.

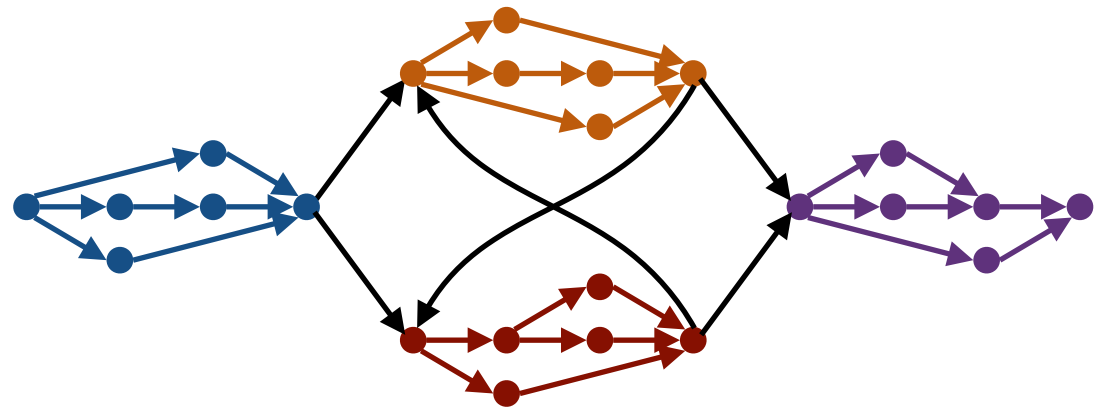
Representing rearrangements

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



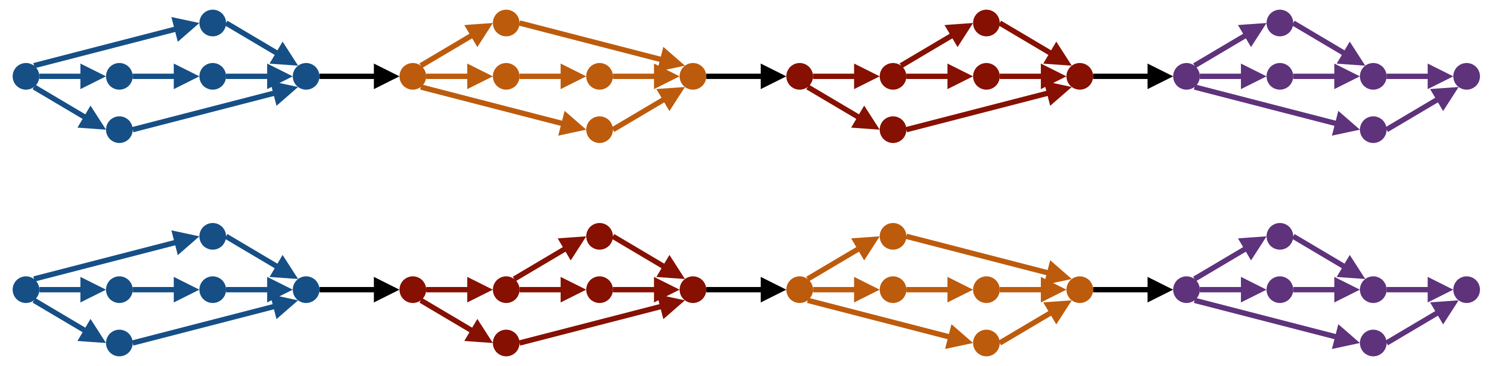


Duplicated subgraphs

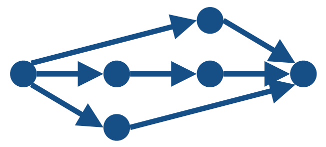


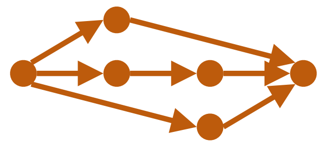
Unsupported cycles

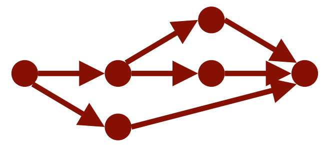
We may need something stronger than graphs, which correspond to regular languages.

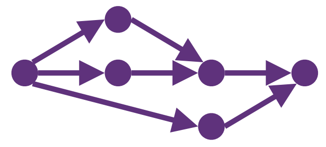


S → ABCD | ACBD

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

The solution from Indexing Hypertext can be made to work with non-nested grammars. In the nested case, we need a way to index arbitrary context-free grammars.