Building the compacted colored de Bruijn Graph

UNIVERSITY OF MARYLAND

NOTE: This lecture is being recorded

Construction of the compacted colored De Bruijn Graph from *reference* sequence

Bioinformatics, 33(24), 2017, 4024–4032
doi: 10.1093/bioinformatics/btw609
Advance Access Publication Date: 21 September 2016
Original Paper

Sequence analysis

TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes

Ilia Minkin[1], Son Pham[2] and Paul Medvedev[1,3,4,*]

[1]Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, USA, [2]BioTuring Inc., San Diego, CA 92121, USA, [3]Department of Biochemistry and Molecular Biology and [4]Genomic Sciences Institute of the Huck, The Pennsylvania State University, University Park, PA 16802, USA

*To whom correspondence should be addressed.

Associate Editor: Alfonso Valencia

Received on April 3, 2016; revised on September 1, 2016; accepted on September 16, 2016

TwoPaCo: An efficient algorithm to build the compacted de Bruijn graph from many complete genomes

Ilia Minkin[1], Son Pham[2], Paul Medvedev[1]

Pennsylvania State University[1]
Salk Institute for Biological Studies[2]

TwoPaCo slides, unless otherwise noted, are from Ilia's presentation

Motivation

- Efficient Sequence Indexing
- More and more complete genomes
- Pan-genome: analysis within same species
- Mammalian-sized genomes are coming soon

Motivation

- Efficient Sequence Indexing
- More and more complete genomes
- Pan-genome: analysis within same species
- Mammalian-sized genomes are coming soon

Key question: what is a handy data structure to represent genomes?

Motivation

- Efficient Sequence Indexing
- More and more complete genomes
- Pan-genome: analysis within same species
- Mammalian-sized genomes are coming soon

Key question: what is a handy data structure to represent genomes?

The simplest way: string(s) of characters.

The Linear Representation

Two genomes:

The Linear Representation

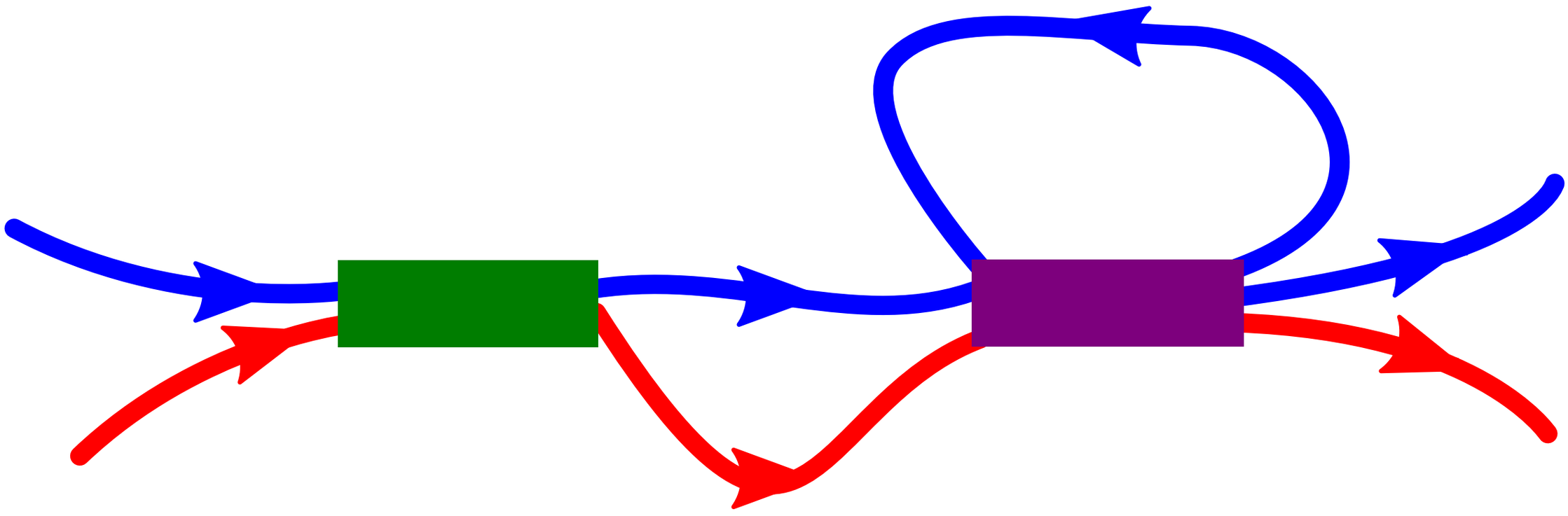
Two genomes:

Issues:

- Homology between genomes?
- Duplications?
- Rearrangements?

Solution: a Graph Representation

What we want to see:



Why de Bruijn graph?

A simple object.

Demonstrated utility in:

- Assembly
- Read mapping
- Synteny identification

The de Bruijn Graph

$k = 2$

TGACGTC

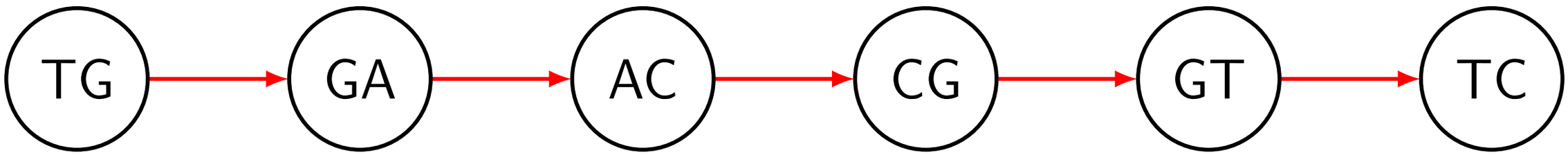
TGACTTC

The de Bruijn Graph

$k = 2$

TGACGTC

TGACTTC

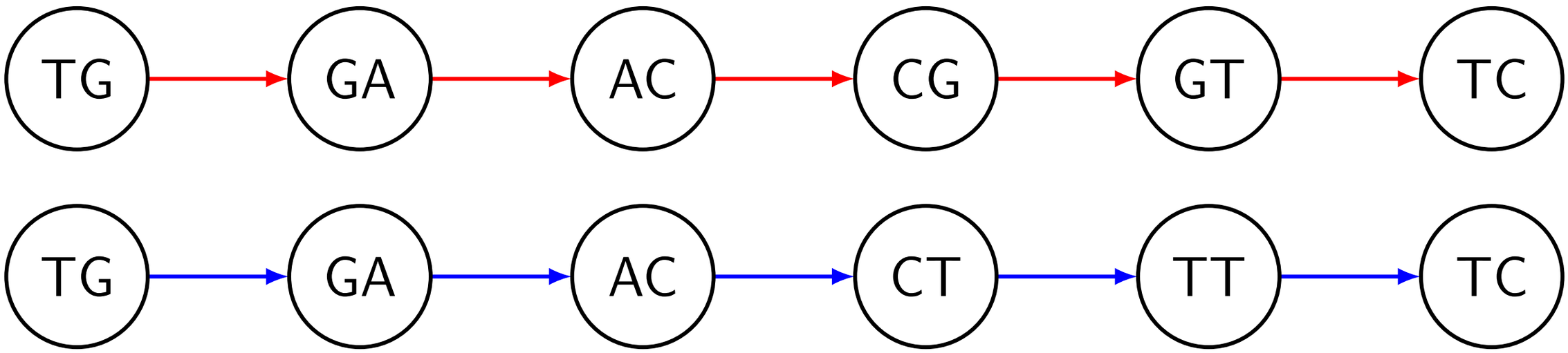


The de Bruijn Graph

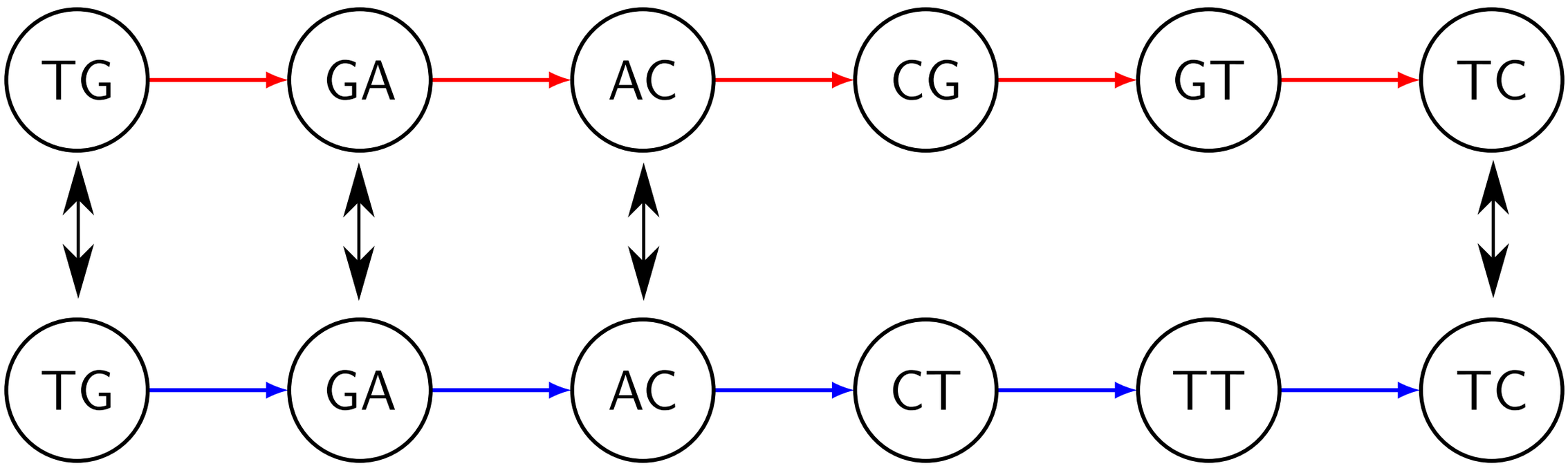
$k = 2$

TGACGTC

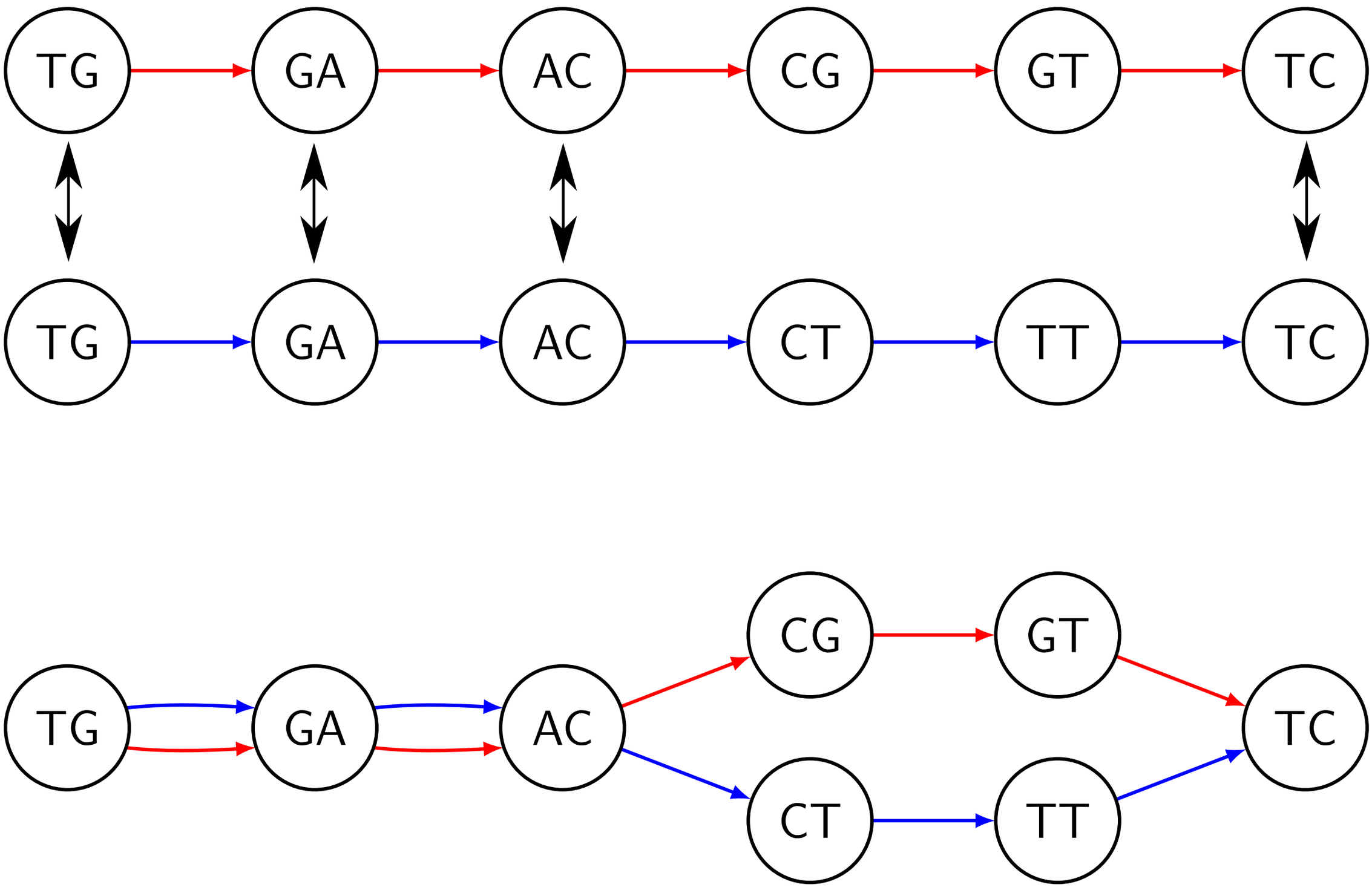
TGACTTC



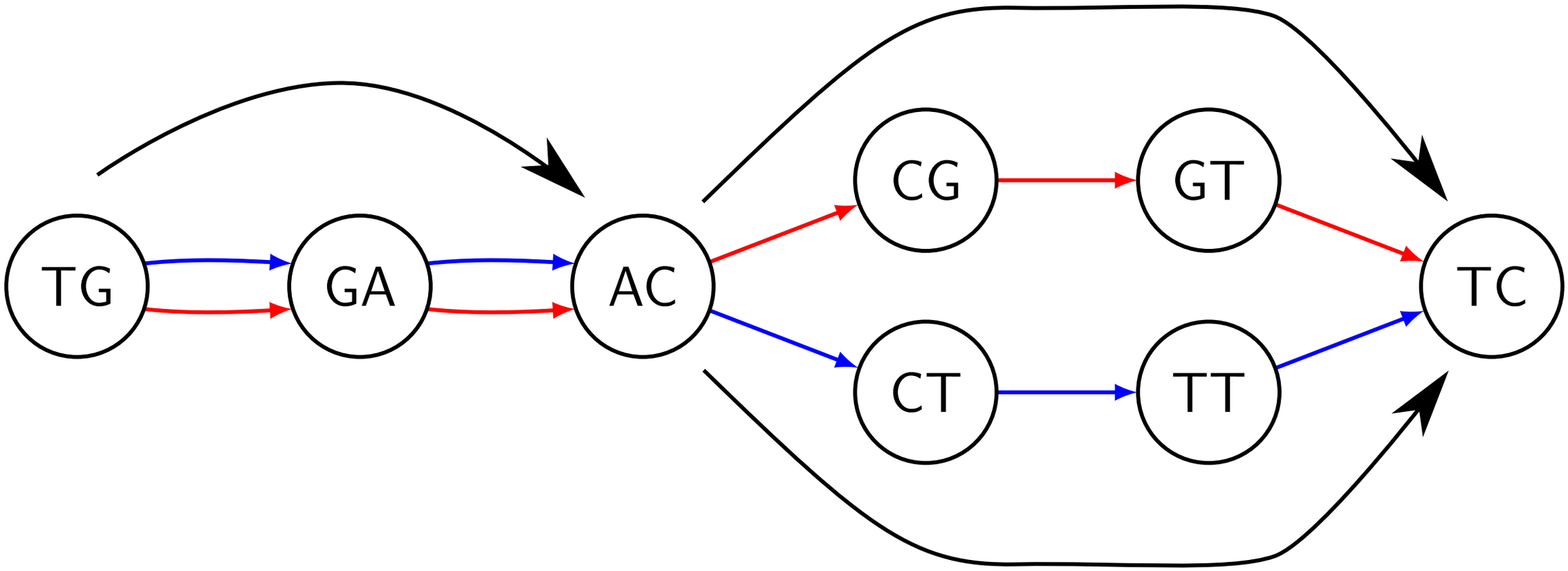
The de Bruijn Graph



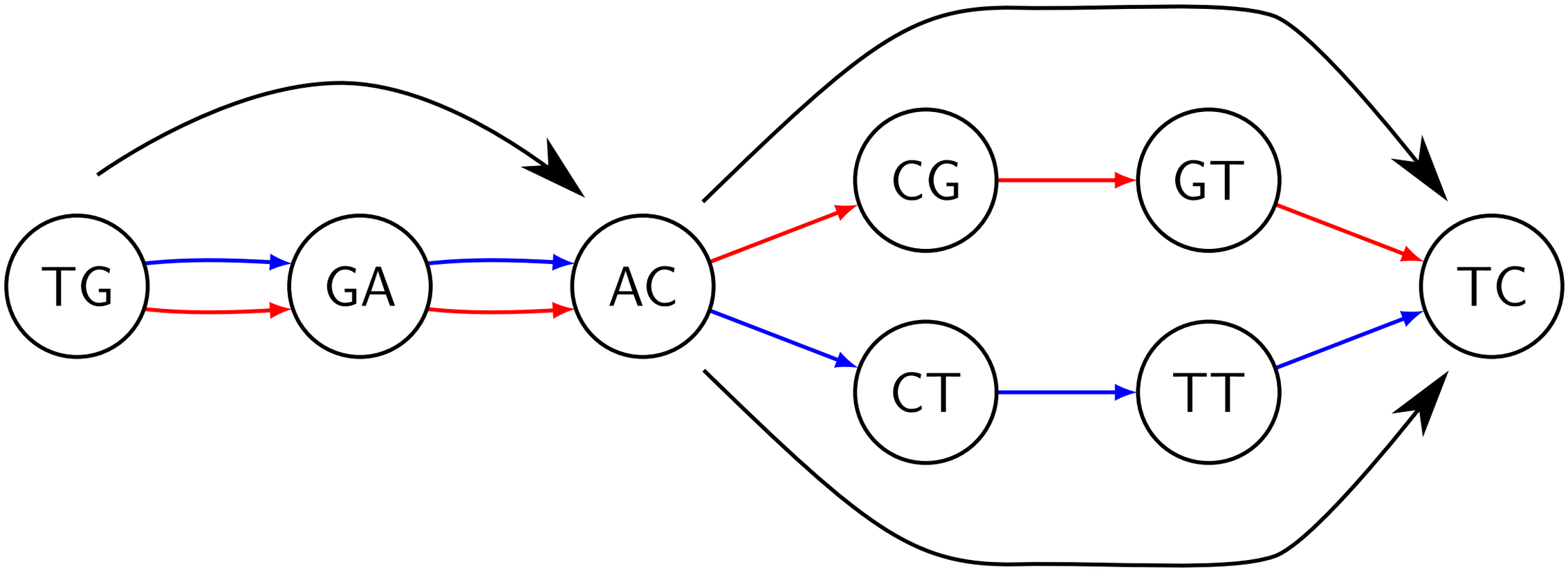
The de Bruijn Graph



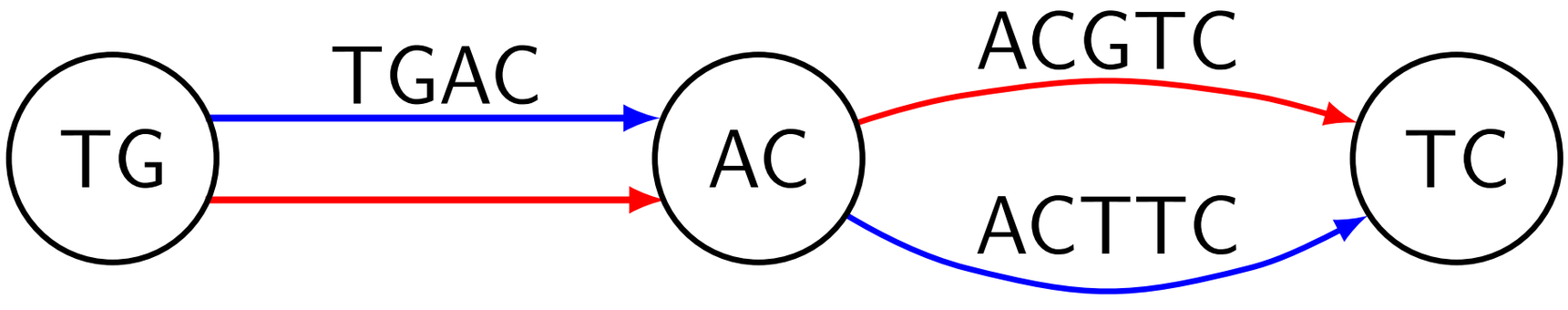
Compaction



Compaction



After compaction:



The Challenge

Construct the compacted graph from many large genomes **bypassing** the ordinary graph traverse.

The Challenge

Construct the compacted graph from many large genomes **bypassing** the ordinary graph traverse.

Earlier work: based on suffix arrays/trees Sibelia & SplitMEM handled > 60 E.Coli genomes.

The Challenge

Construct the compacted graph from many large genomes **bypassing** the ordinary graph traverse.

Earlier work: based on suffix arrays/trees Sibelia & SplitMEM handled $>$ 60 E.Coli genomes.

A recent advance: 7 Humans in 15 hours using 100 GB of RAM using a BWT-based algorithm by Baier *et al.*, 2015, Beller *et al.*, 2014.

Junctions

A vertex v is a **junction** if:

- v has ≥ 2 distinct outgoing or incoming edges:

Junctions

A vertex v is a **junction** if:

- v has ≥ 2 distinct outgoing or incoming edges:

- v is the first or the last k-mer of an input string

Junctions

A vertex v is a **junction** if:

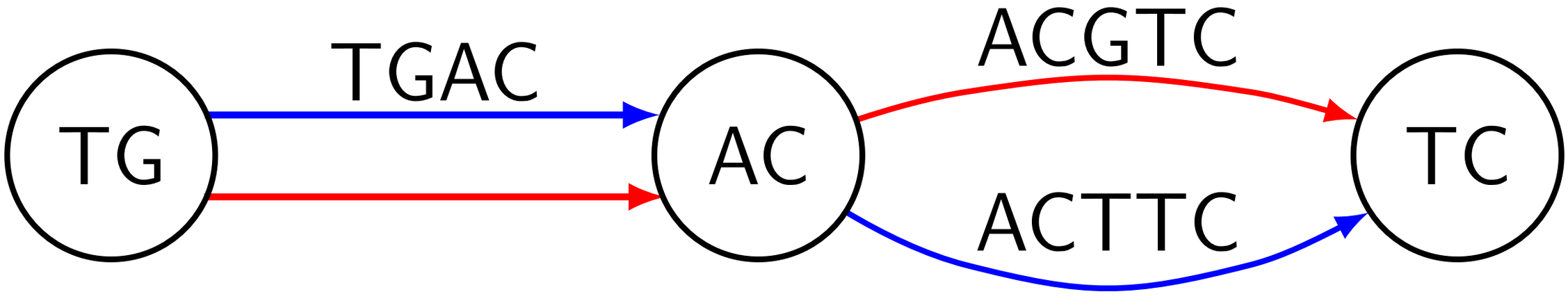
- v has ≥ 2 distinct outgoing or incoming edges:

- v is the first or the last k-mer of an input string

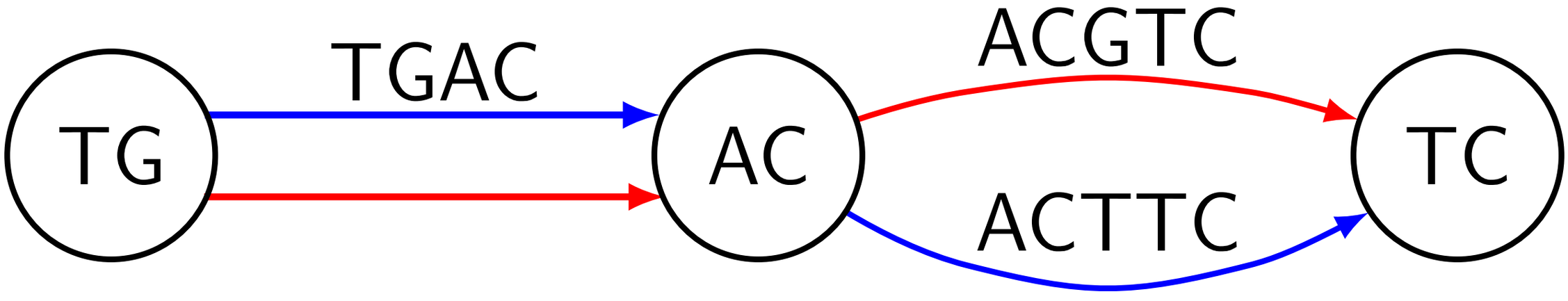
Facts:

- Junctions $=$ vertices of the compacted graph
- Compaction $=$ finding positions of junctions

Observations

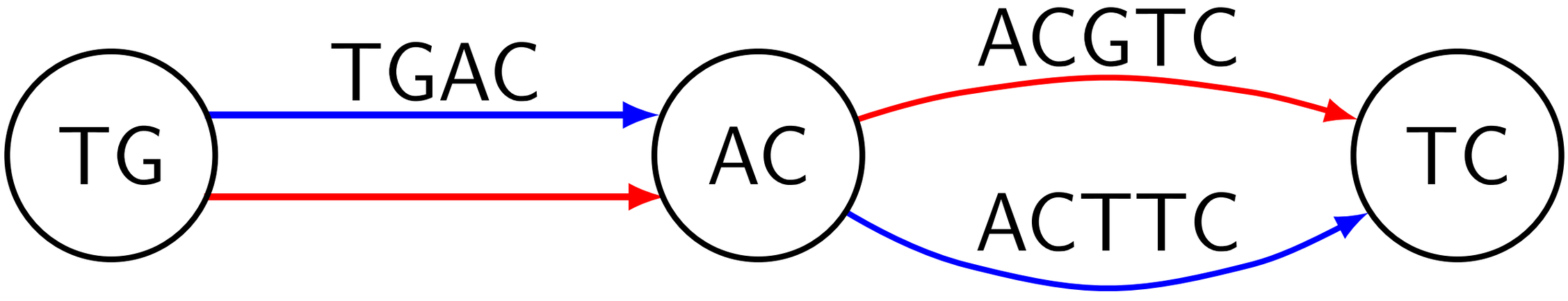


Observations



TG GA AC CG GT TC

Observations



TG GA AC CG GT TC

TG $\rightarrow$ AC $\rightarrow$ TC

The Observation

The observation only works when we have complete genomes.

Once we know junctions, construction of the edges is simple.

We can simply traverse input strings and record junctions in the order they appear.

How to identify junctions?

The Naive Algorithm

A naive way:

- Store all $(k+1)$-mers (edges) in a hash table
- Consider each vertex one by one
- Query all possible edges from the table
- If found > 1 edge, mark vertex as a junction

Simple algorithm in more detail

Algorithm 1. *Filter-Junctions*

Input: strings $S = \{s_1, \ldots, s_n\}$, integer k, and an empty set data structure E. A candidate set of marked junction positions $C \supseteq J(S,k)$ is also given. When the algorithm is run naively, all the positions would be marked.
Output: a reduced candidate set of junction positions.

```
1: for s ∈ S do
2:     for 1 ≤ i < |s| − k do
3:         if C[s, i] = marked then                ▷ Insert the two (k+1)-mers containing the k-mer at i into E.
4:             Insert s[i..i + k] into E.
5:             Insert s[i − 1..i − 1 + k] into E.
6: for s ∈ S do
7:     for 1 ≤ i < |s| − k do
8:         if C[s, i] = marked and s[i..i + k − 1] is not a sentinel then
9:             in ← 0                              ▷ Number of entering edges
10:            out ← 0                             ▷ Number of leaving edges
11:            for c ∈ {A, C, G, T} do             ▷ Consider possible edges and count how many of them exist
12:                if v · c ∈ E then               ▷ The symbol · depicts string concatenation
13:                    out ← out + 1
14:                if c · v ∈ E then
15:                    in ← in + 1
16:            if in = 1 and out = 1 then          ▷ If the k-mer at i is not a junction.
17:                C[s, i] ← Unmarked
18: return C
```

The Naive Algorithm

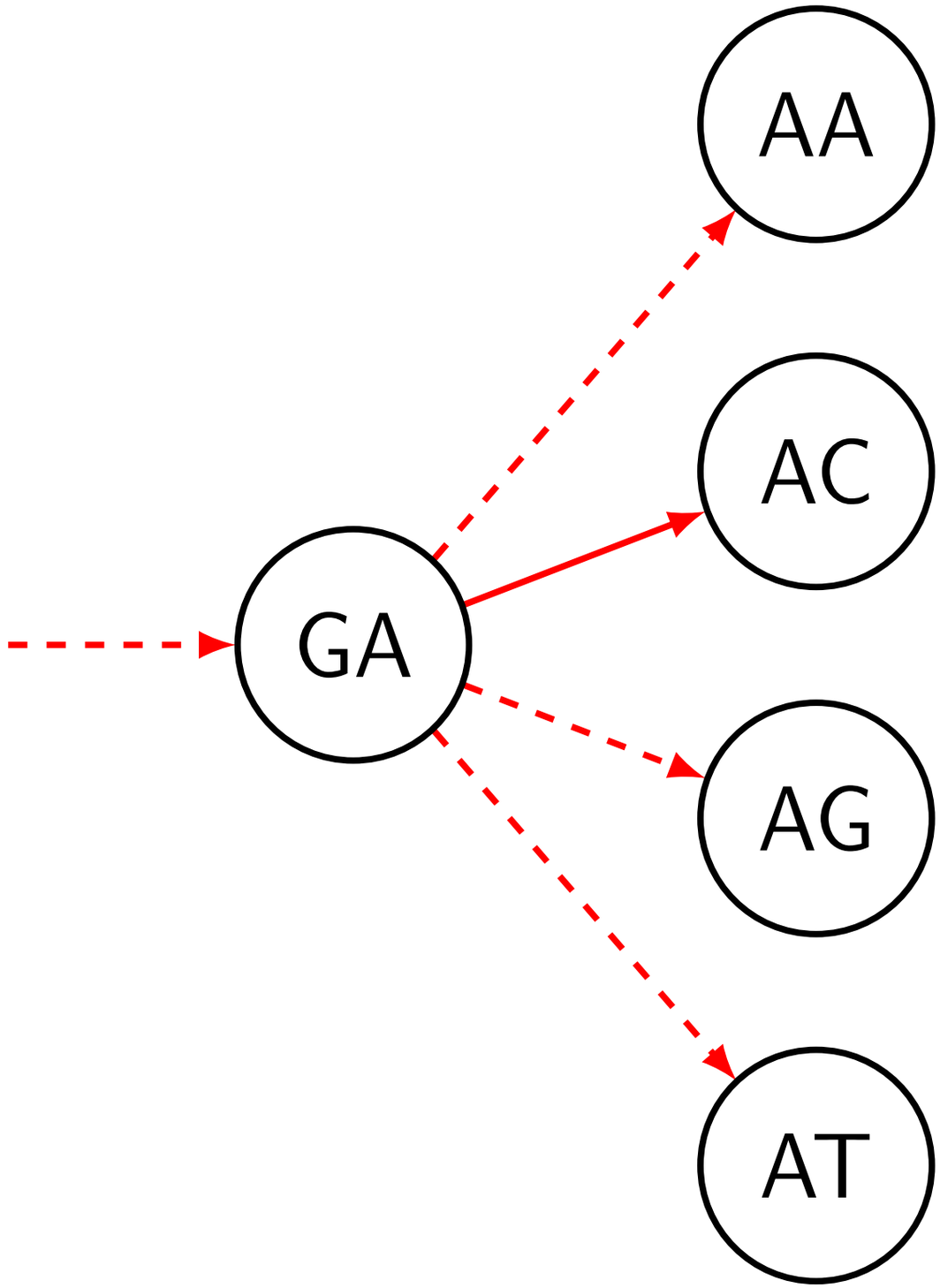
A naive way:

- Store all $(k+1)$-mers (edges) in a hash table
- Consider each vertex one by one
- Query all possible edges from the table
- If found > 1 edge, mark vertex as a junction

Problem: the hash table can be too large.

An Example

Hash table $= \{$ GA $\rightarrow$ AC $\}$



What is the Bloom filter

A probabilistic data structure representing a set

Properties:

- Occupies fixed space
- May generate false positives on queries
- False positive rate is low

What is the Bloom filter

A probabilistic data structure representing a set

Properties:

- Occupies fixed space
- May generate false positives on queries
- False positive rate is low

Example: Bloom Filter $= \{$ GA $\rightarrow$ AC $\}$

Is GA $\rightarrow$ AC in the set? Yes.

What is the Bloom filter

A probabilistic data structure representing a set

Properties:

- Occupies fixed space
- May generate false positives on queries
- False positive rate is low

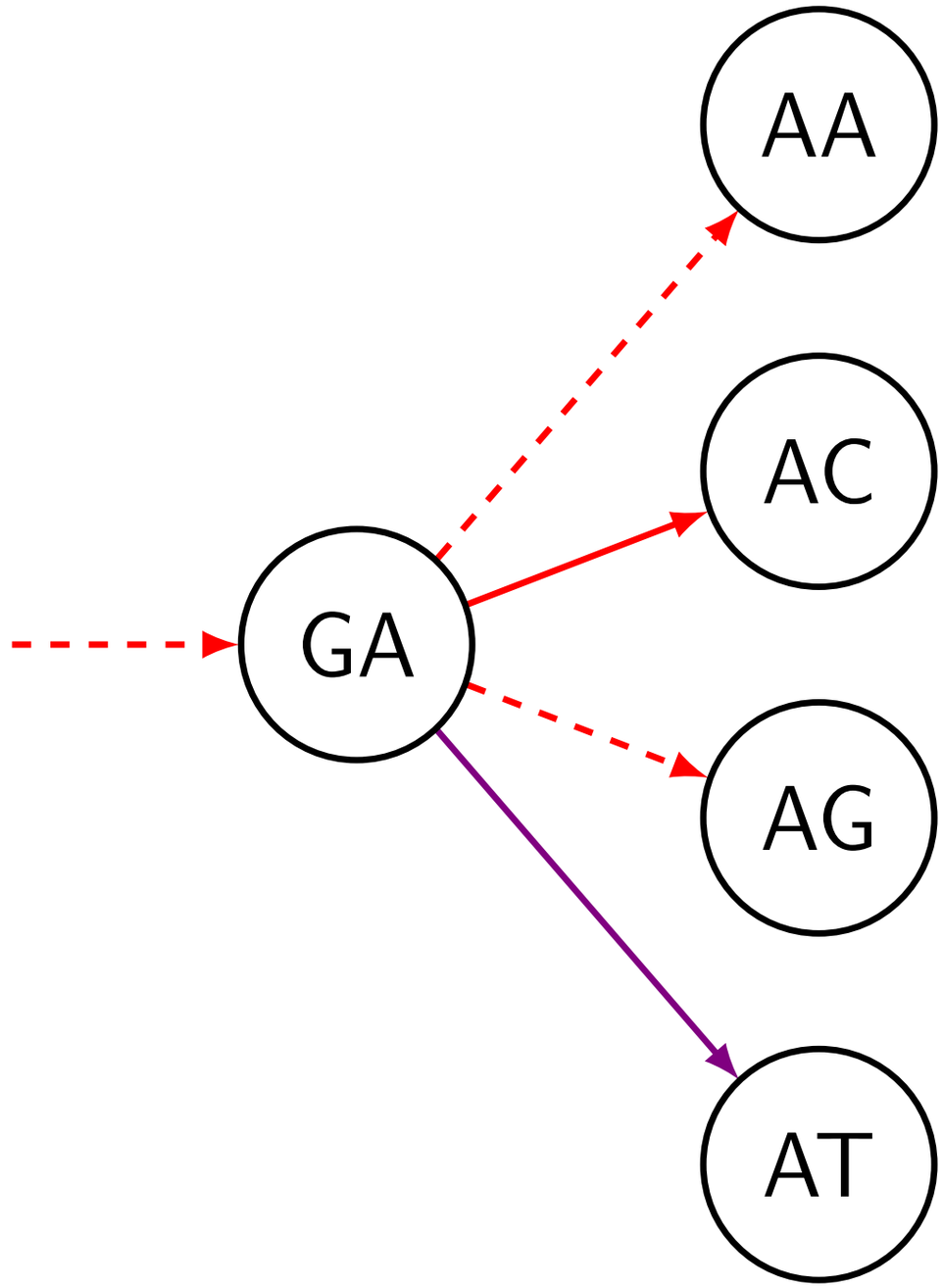
Example: Bloom Filter $= \{$ GA $\rightarrow$ AC $\}$

Is GA $\rightarrow$ AC in the set? Yes.

Is GA $\rightarrow$ AT in the set? **Maybe** no.

An Example

Bloom Filter $= \{$ GA $\rightarrow$ AC, GA $\rightarrow$ AT $\}$



The purple edge is a false positive.

The Two Pass Algorithm

How to eliminate false positives?

The Two Pass Algorithm

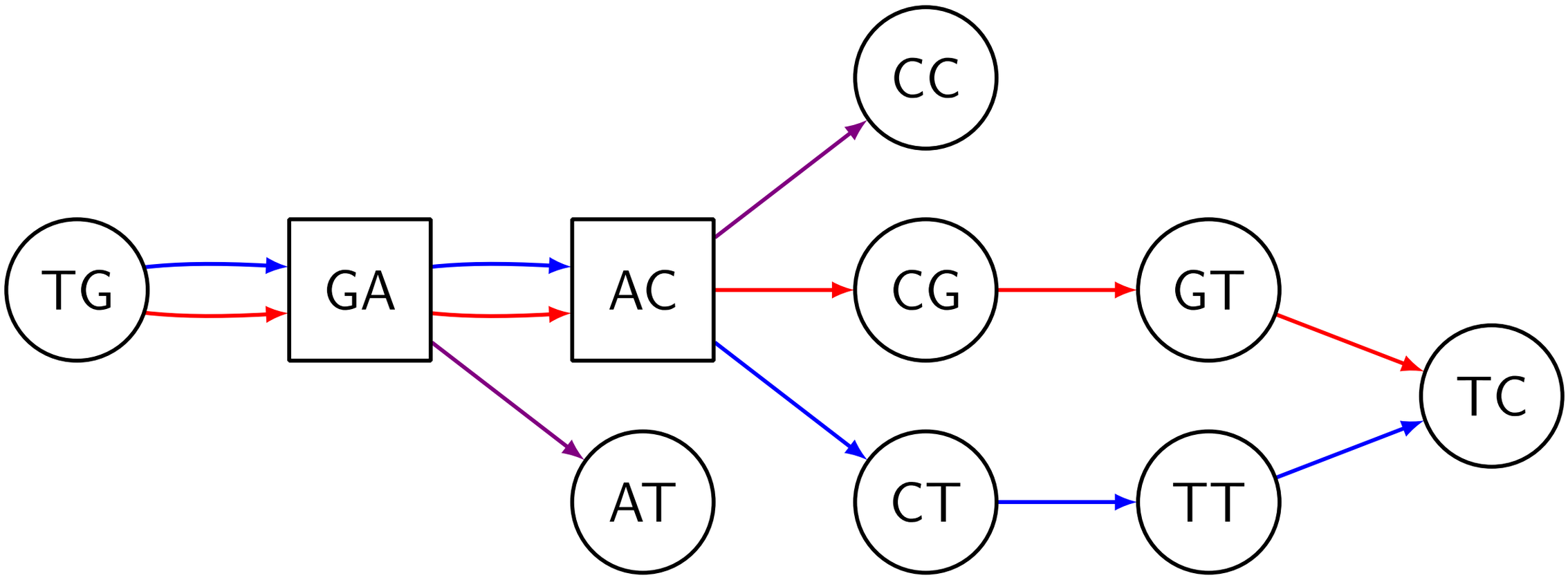
How to eliminate false positives?

Two-pass algorithm:

1. Use the Bloom filter to identify **junction candidates**
2. Use the hash table, but store **only edges that touch candidates**

An Example: the First Step

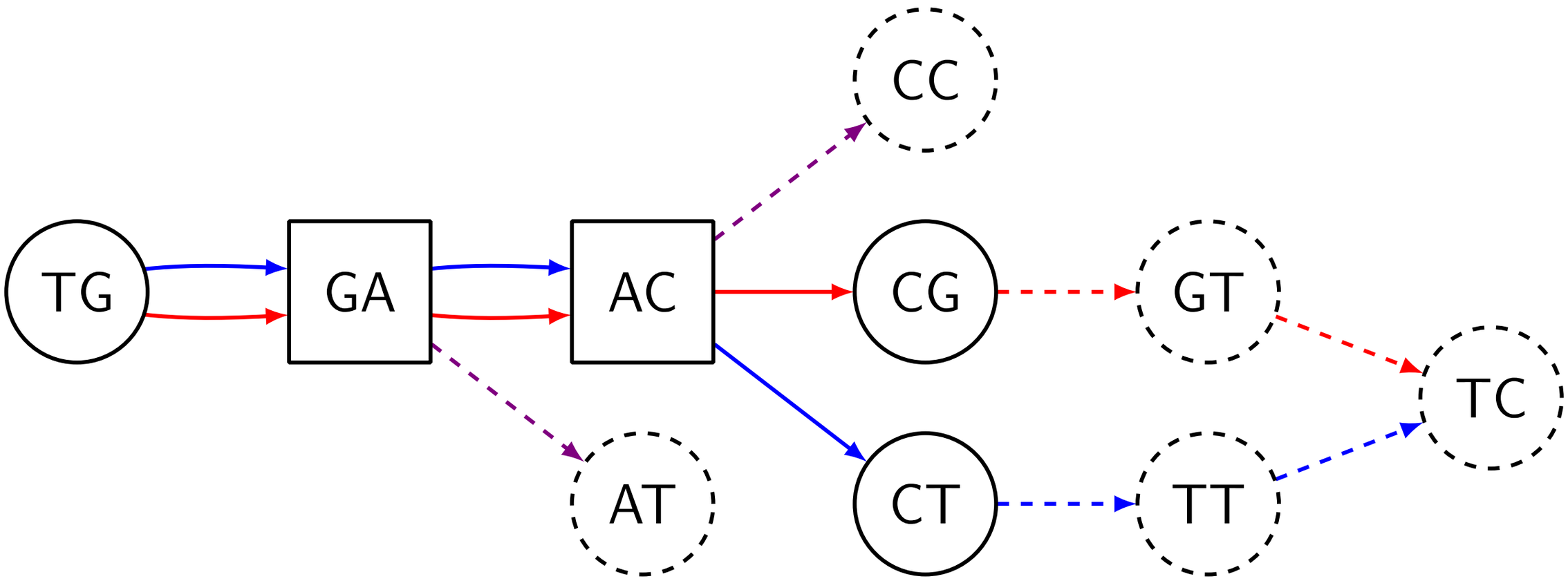
Here edges stored in the Bloom filter, purple ones are false positives:



Junction candidates: GA & AC

An Example: the Second Step

Edges stored in the hash table. We kept only edges touching junction candidates:



Junction: AC

The TwoPass Algorithm

Algorithm 2. Filter-Junctions-Two-Pass

Input: strings $S = \{s_1, \ldots, s_n\}$, integer k, a candidate set of junction positions C_{in}, integer b

Output: a candidate set of junction positions C_{out}

1: $F \leftarrow$ an empty Bloom filter of size b

2: $C_{\text{temp}} \leftarrow \mathit{Filter-Junctions}(S, k, F, C_{\text{in}})$ ▷ The first pass

3: $H \leftarrow$ an empty hash table

4: $C_{\text{out}} \leftarrow \mathit{Filter-Junctions}(S, k, H, C_{\text{temp}})$ ▷ The second pass

5: **return** C_{out}

The TwoPaCo algorithm

Algorithm 3. TwoPaCo

Input: strings $S = \{s_1, \ldots, s_n\}$, integer k, integer ℓ, integer b
Output: the compacted de Bruijn graph $G_c(S, k)$
1: Initialize counters $c_0, \ldots, c_{q-1}$ to zeroes
2: $F \leftarrow$ an empty Bloom filter of size b
3: **for** $s \in S$ **do**
4: **for** $1 \leq i \leq |s| - k + 1$ **do**
5: $h \leftarrow s[i..i + k - 1]$
6: **if** h not in F **then**
7: Insert h into F
8: $c_{f(h)} \leftarrow c_{f(h)} + 1$
9: $T \leftarrow \sum_{0 \leq t < q} c_t / \ell$ ▷ Mean number of k-mers per partition
10: $p_0 \leftarrow 0, p_\ell \leftarrow q$
11: **for** $1 \leq i < \ell$ **do**
12: $p_i \leftarrow$ biggest integer larger than p_{i-1} such that $(\sum_{p_{i-1} \leq j < p_i} c_j) \leq T$, or $\min\{\ell, p_{i-1} + 1\}$ if it does not exist.
13: $C_{\text{init}} \leftarrow$ Boolean array with every position unmarked
14: **for** $1 \leq i \leq \ell$ **do**
15: $C_i \leftarrow$ mark every position of C_{init} that starts a k-mer h with hash value $p_{i-1} \leq f(h) < p_i$
16: $C'_i \leftarrow \text{Filter} - \text{Junctions} - \text{Two} - \text{Pass}(S, k, b, C_i)$
17: $C_{\text{final}} = \cup C'_i$
18: **return** Graph implied by C_{final}, as described in Section 3.

Results

Datasets:

- 7 humans: 5 versions of the reference +
  2 haplotypes of NA12878 from 1000 Genomes
- 93 simulated humans (FIGG)
- 8 primates available in UCSC genome browser

Results

Format: minutes (GB)

Table 2. Benchmarking comparisons

	DSK+BCALM	Minia	Sibelia	SplitMem	bwt-based from Baier *et al.* (2015)		TwoPaCo	
				Single strand	Single strand	Both strands	1 thread	15 threads
62 *E.coli* ($k = 25$)	6 (1.57)	151 (0.9)	10 (12.2)	70 (178.0)	8 (0.85)	12 (1.7)	4 (0.16)	2 (0.39)
62 *E.coli* ($k = 100$)	13 (2.50)	114 (1.9)	8 (7.6)	67 (178.0)	8 (0.50)	12 (1.0)	4 (0.19)	2 (0.39)
7 humans ($k = 25$)	444 (22.44)	968 (48.09)	–	–	867 (100.30)	1605 (209.88)	436 (4.40)	63 (4.84)
7 humans ($k = 100$)	1347 (221.65)	1857 (222.0)	–	–	807 (46.02)	1080 (92.26)	317 (8.42)	57 (8.75)
8 primates ($k = 25$)	2088 (85.62)	–	–	–	–	–	914 (34.36)	111 (34.36)
8 primates ($k = 100$)	–	–	–	–	–	–	756 (56.06)	101 (61.68)
$(43 + 7)$ humans ($k = 25$)	–	–	–	–	–	–		705 (69.77)
$(43 + 7)$ humans ($k = 100$)	–	–	–	–	–	–		927 (70.21)
$(93 + 7)$ humans ($k = 25$)	–	–	–	–	–	–		1383 (77.42)

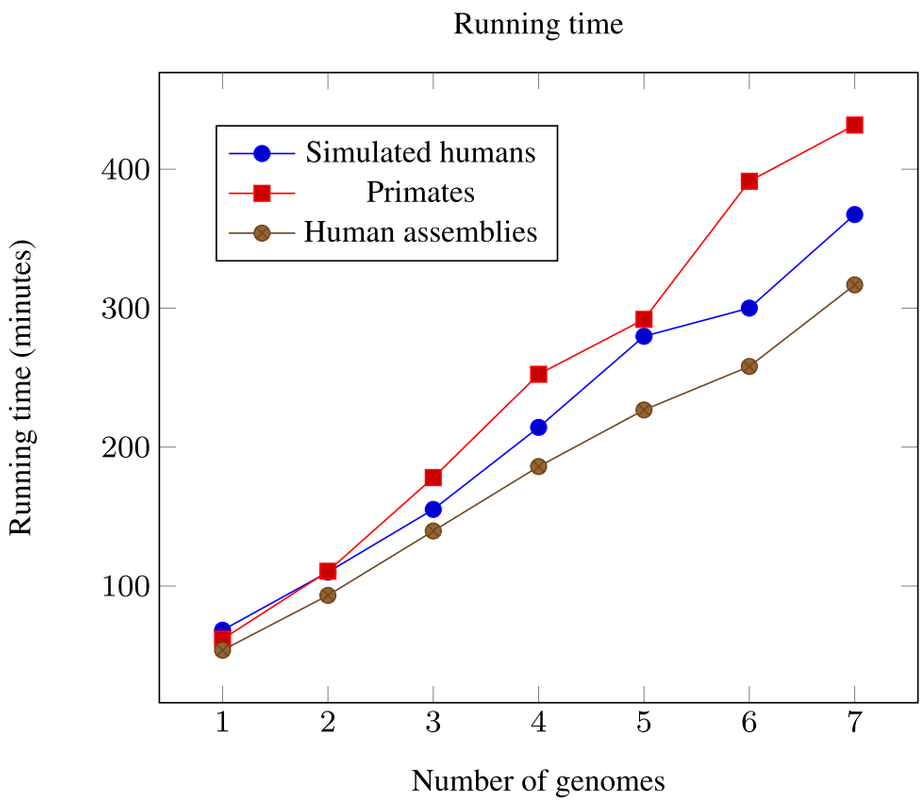
Note: Each cell shows the running time in minutes and the memory usage in parenthesis in gigabytes. TwoPaCo was run using just one round, with a Bloom filter size $b = 0.13$ GB for *E.coli*, 4.3 GB for 7 humans with $k = 25$, $b = 8.6$ GB with $k = 100$, $b = 34$ GB for primates, and $b = 69$ GB for $(43 + 7)$ and larger human dataset. A dash in the SplitMem and bwt-based columns indicates that they ran out of memory, a dash in the Sibelia column indicates that it could not be run on such large inputs, a dash in the minia column indicates that it did not finish in 48 h, a dash in the BCALM column indicates that it ran out of disk space (4 TB). A double dash indicates that the software had a segmentation fault. An empty slot indicates that the experiment was not done.

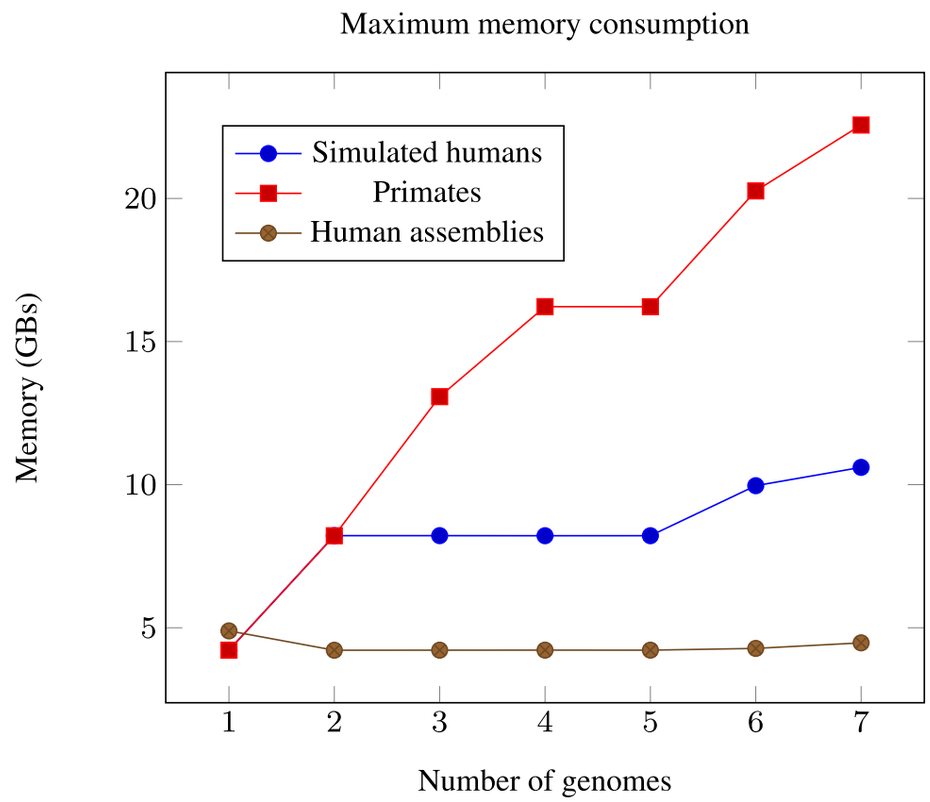
Conclusion & Future Work

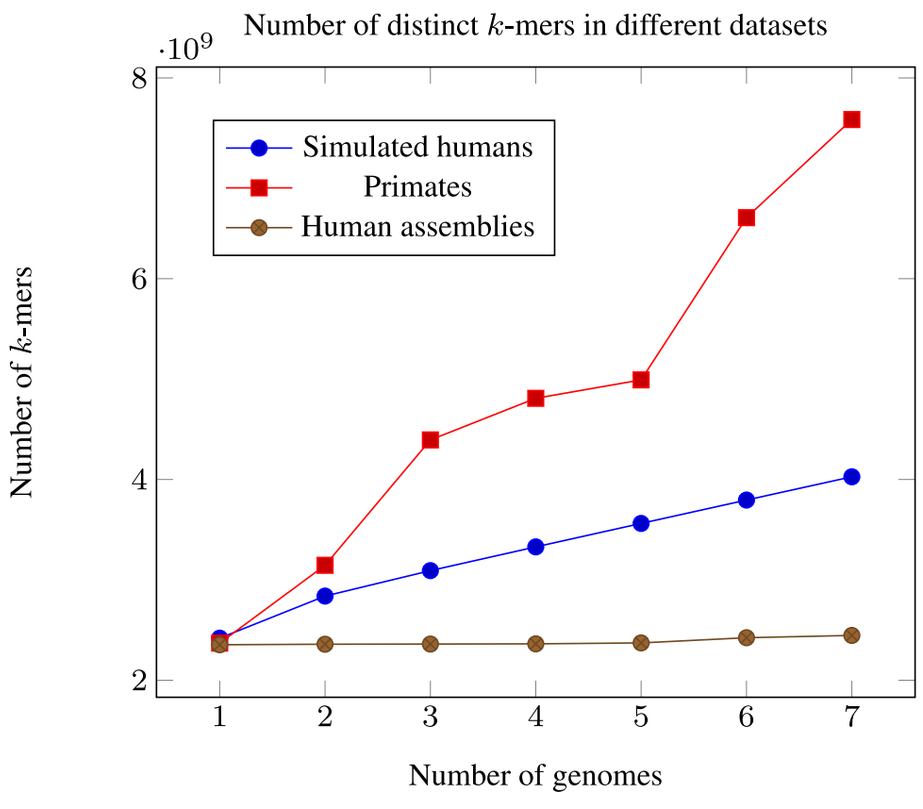
Can potentially facilitate:

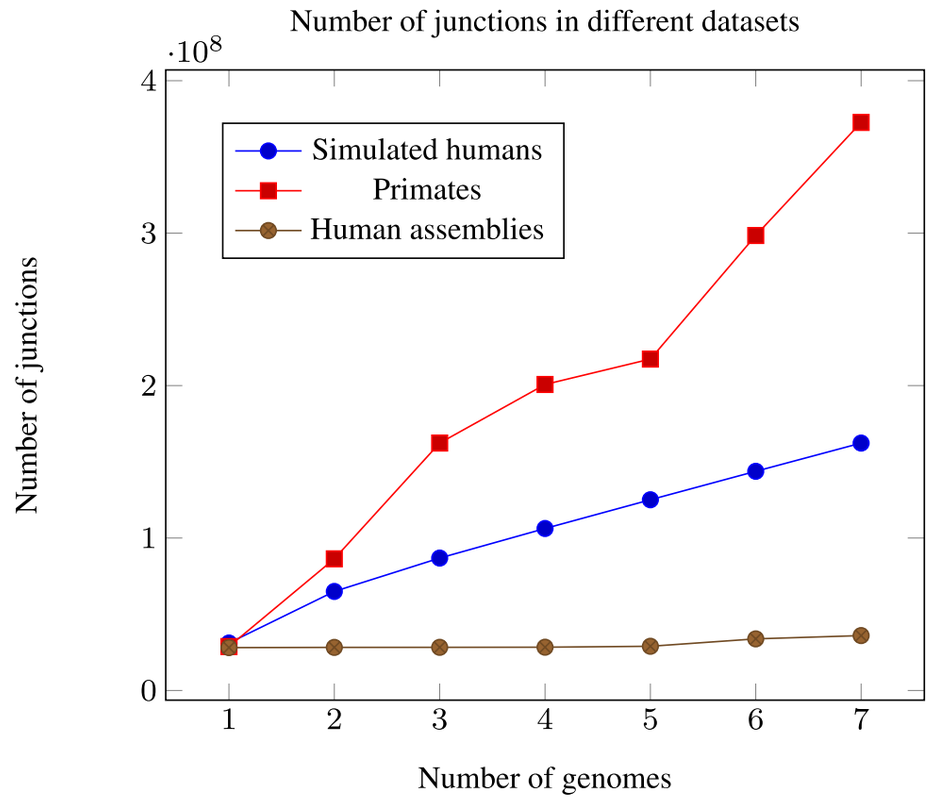
- Visualization
- Synteny mining (Sibelia)
- Structural variations analysis
- ...

Input Size vs. Performance

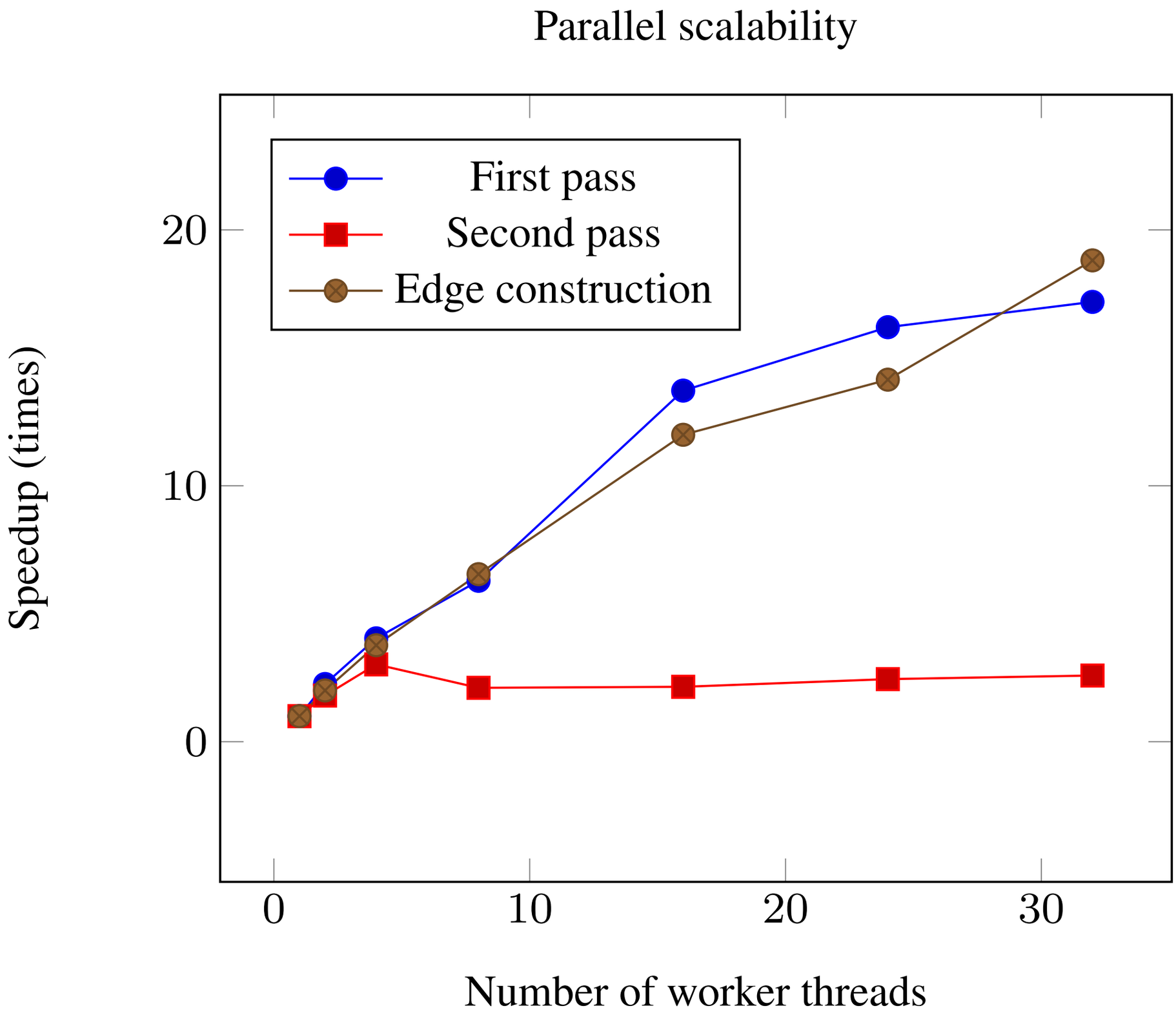








Parallel Scalability



Splitting

Table 1: The minimal number of rounds it takes to compress the graph without exceeding a given memory threshold.

Memory threshold	Used memory	Bloom filter size	Running time	Rounds
10	8.62	8.59	259	1
8	6.73	4.29	434	3
6	5.98	4.29	539	4
4	3.51	2.14	665	6

One small caveat

Thanks for the detailed explanation and reference. I *think* I get it now. So the reason that the "palindromic" sequences have duplicate (k+1)-mers (assuming canonicalization) is because their in/out edges degrees are only counted once --- that is, if I encounter that k-mer in a plaindromic sequence, and I look at the two (k+1)-mers extending it, they are the same after canonicalization.

For example, take the sequence S = `AACTGACA|TGTCAGTT`, and let k=3. The `|` denotes where the sequence reflects over itself to form the palindrome. Clearly, there are repeated canonical (k+1)-mers in this sequence, since :

`AACT = AGTT`,
`ACTG = CAGT`,
`CTGA = TCAG`,
`TGAC = GTCA`,
`GACA = TGTC`

However, as I scan through the k-mers as the TwoPaCo algorithm does, each time I check for the (k+1)-mers that contain a k-mer, and canonicalize them, I end up with only a single in and out edge for these k-mers (I think this is the case for all of the, I only spot-checked a few). Thus, we won't find junction nodes and the entire sequence will be considered as a contig. Pictorially, it looks like:

```
[AAC] -> [ACT] -> [TGA] -> [GAC] -> [ACA] -> [CAT] ->
                                                \
                                                 |
                                                /
[GTT]<- [AGT] <- [TCA] <- [GTC] <- [TGT] <- [ATG] <-
```

because, were we to reflect the first half of this sequence about the `|`, we would get the second half --- i.e., there is a self-edge at the [CAT | ATG] node. Thus, I was originally expecting we'd get a contig +1 = `AACTGACA` and we would spell out S as +1,-1. Instead, since we are "junction-free", we simply say +1 = `AACTGACATGTCAGTT` and spell out S = +1, despite the fact that this contig contains duplicate (k+1)-mers if we consider canonicalization. Is this interpretation correct? If so, it seems that duplicate k-mers and k+1-mers in palindromic contigs are an inherent aspect of the junction-based approach adopted by TwoPaCo. Fortunately, it also seems that these are the only such cases where duplicate k+1-mers can occur.