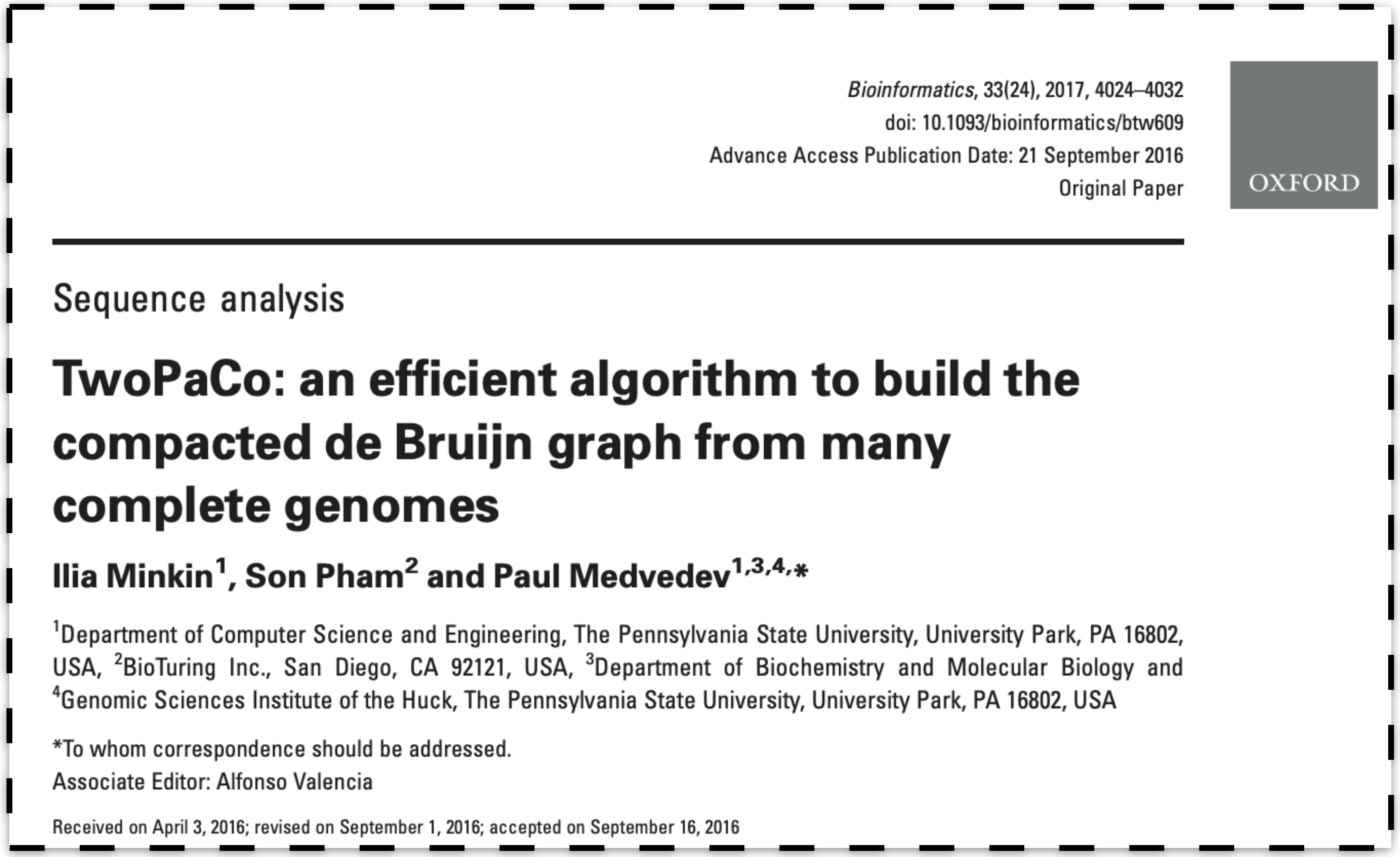
Building the compacted colored de Bruijn Graph

UNIVERSITY OF MARYLAND

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

OXFORD

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]

8th July 2016

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

Motivation

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

Motivation

- 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

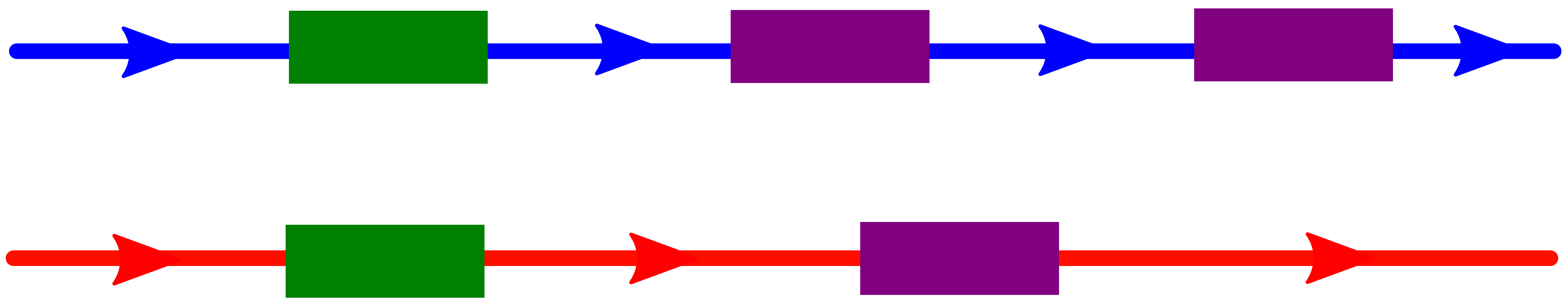
- More and more complete genomes
- Pan-genome: analysis within same species
- Mammalian-sized genomes are coming soon

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

The simplest way: string(s) of characters.

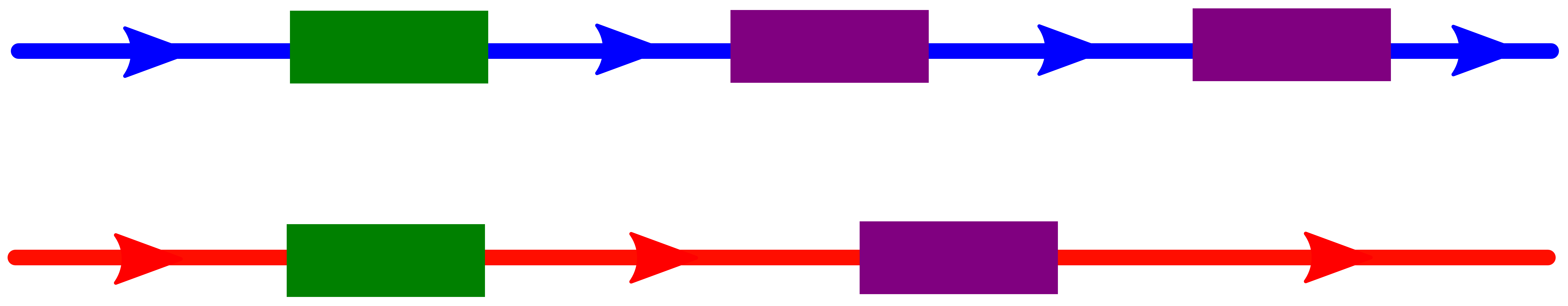
The Linear Representation

Two genomes:



The Linear Representation

Two genomes:

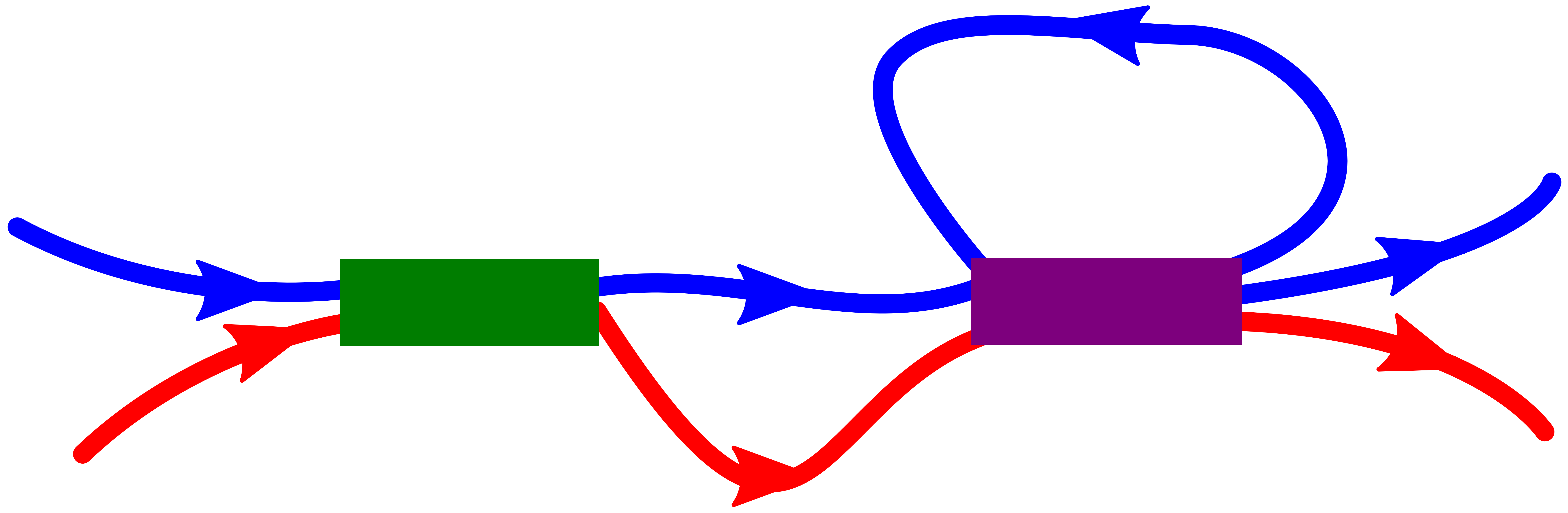


Issues:

- 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

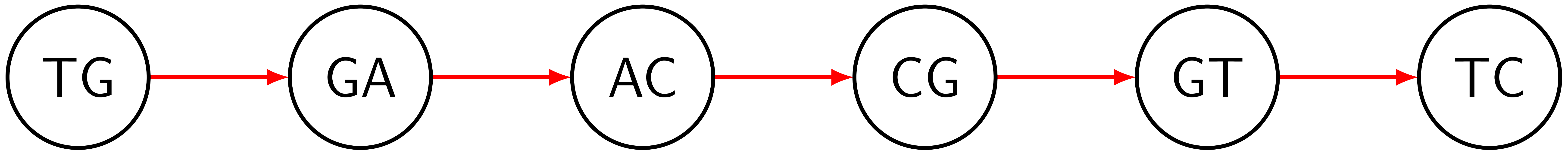
TGACTTC

The de Bruijn Graph

$k = 2$

TGACGTC

TGACTTC

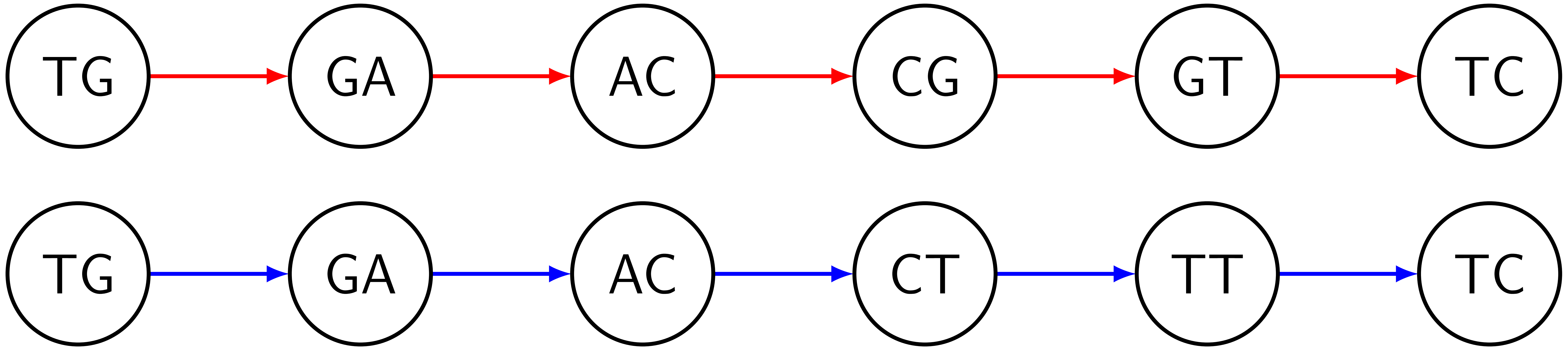


The de Bruijn Graph

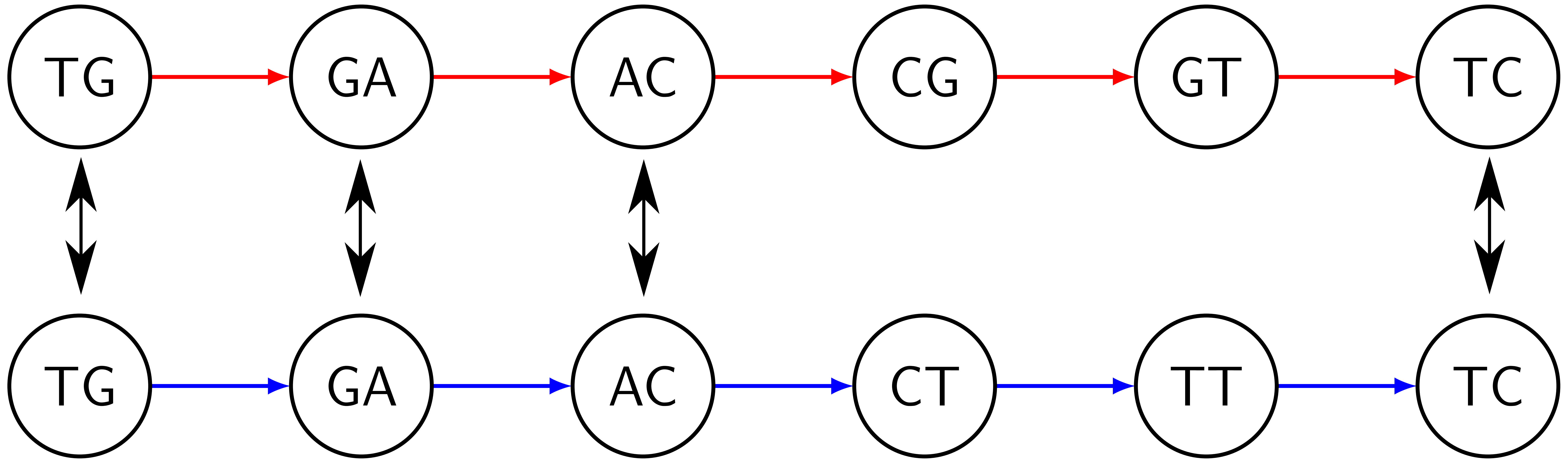
$k = 2$

TGACGTC

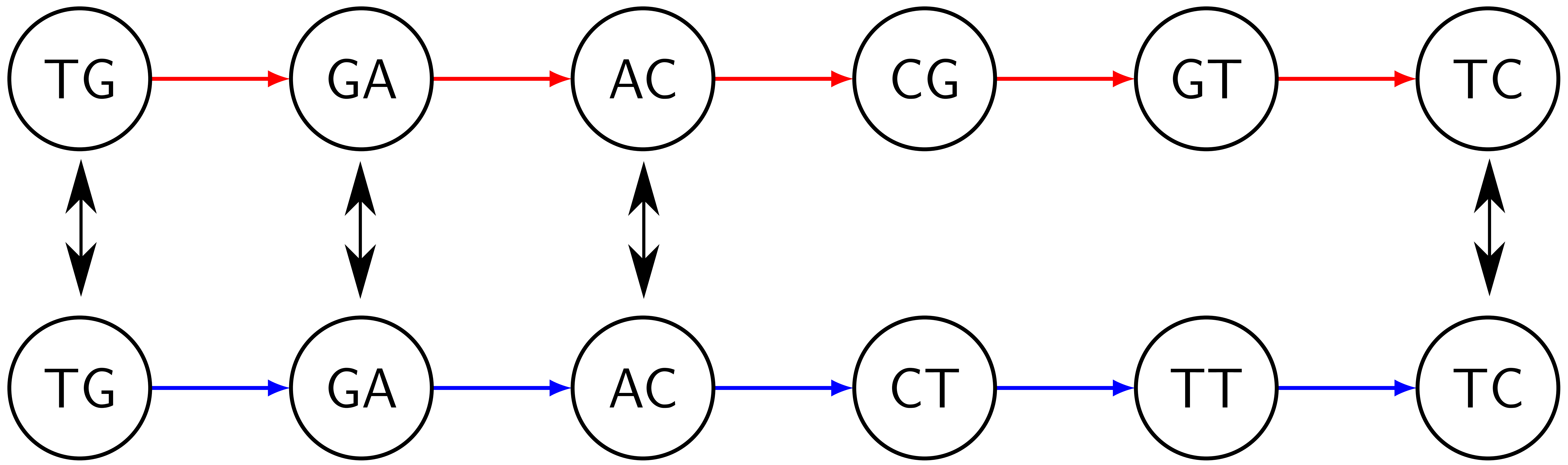
TGACTTC

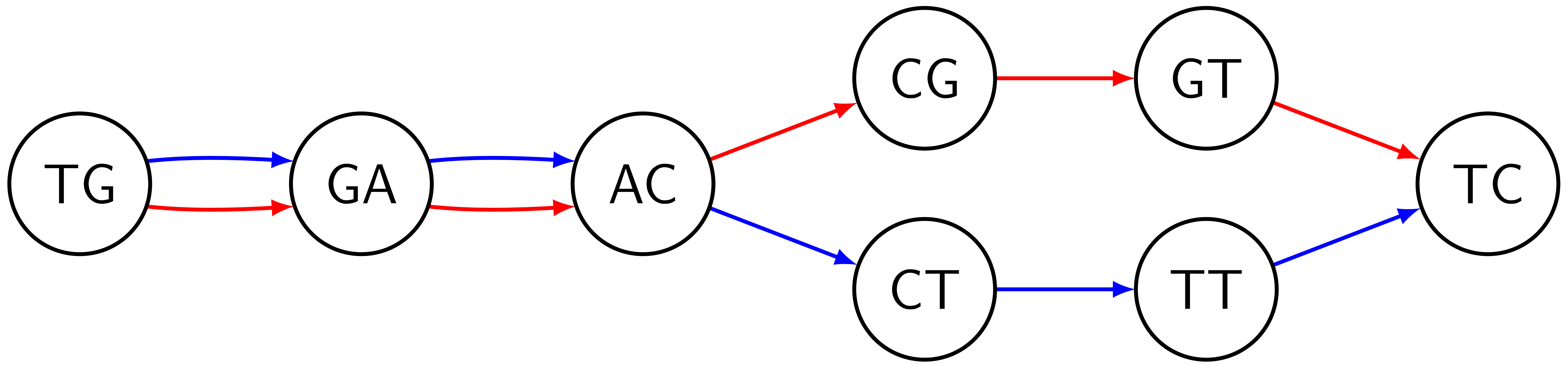

$$\begin{tikzcd}
TG \arrow[r] & GA \arrow[r] & AC \arrow[r] & CG \arrow[r] & GT \arrow[r] & TC \\
TG \arrow[r] & GA \arrow[r] & AC \arrow[r] & CT \arrow[r] & TT \arrow[r] & TC
\end{tikzcd}$$

The de Bruijn Graph

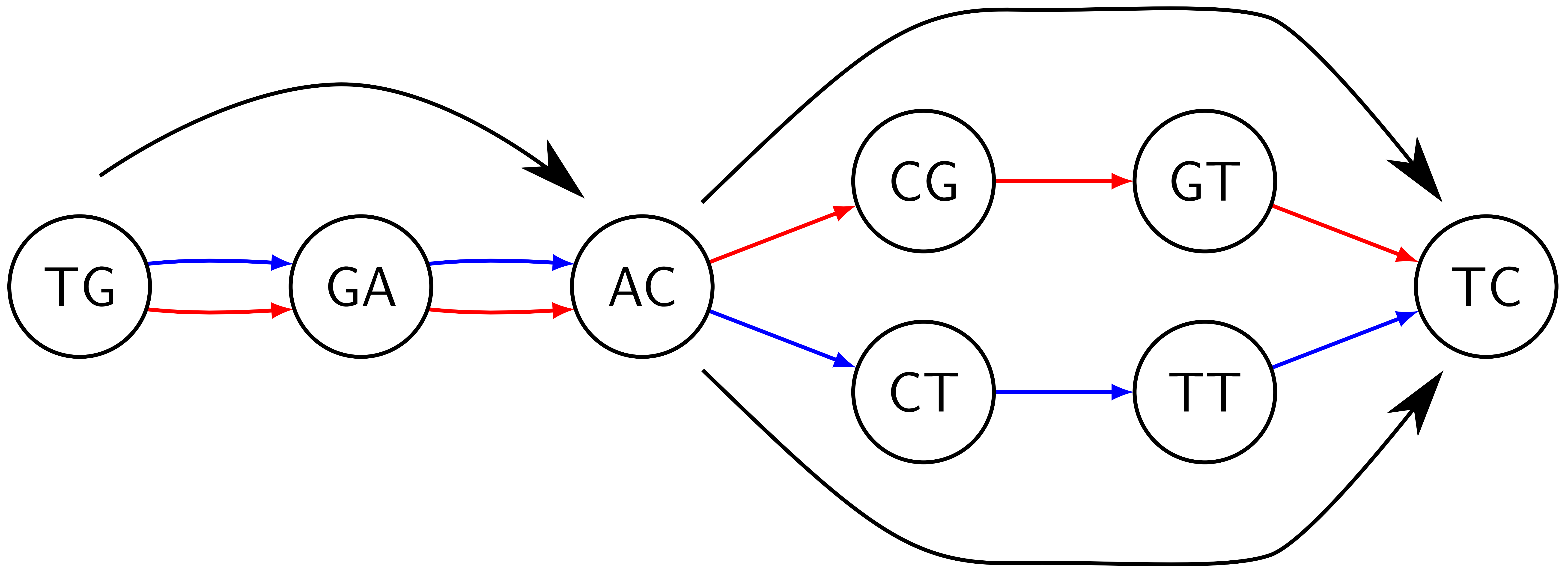


The de Bruijn Graph

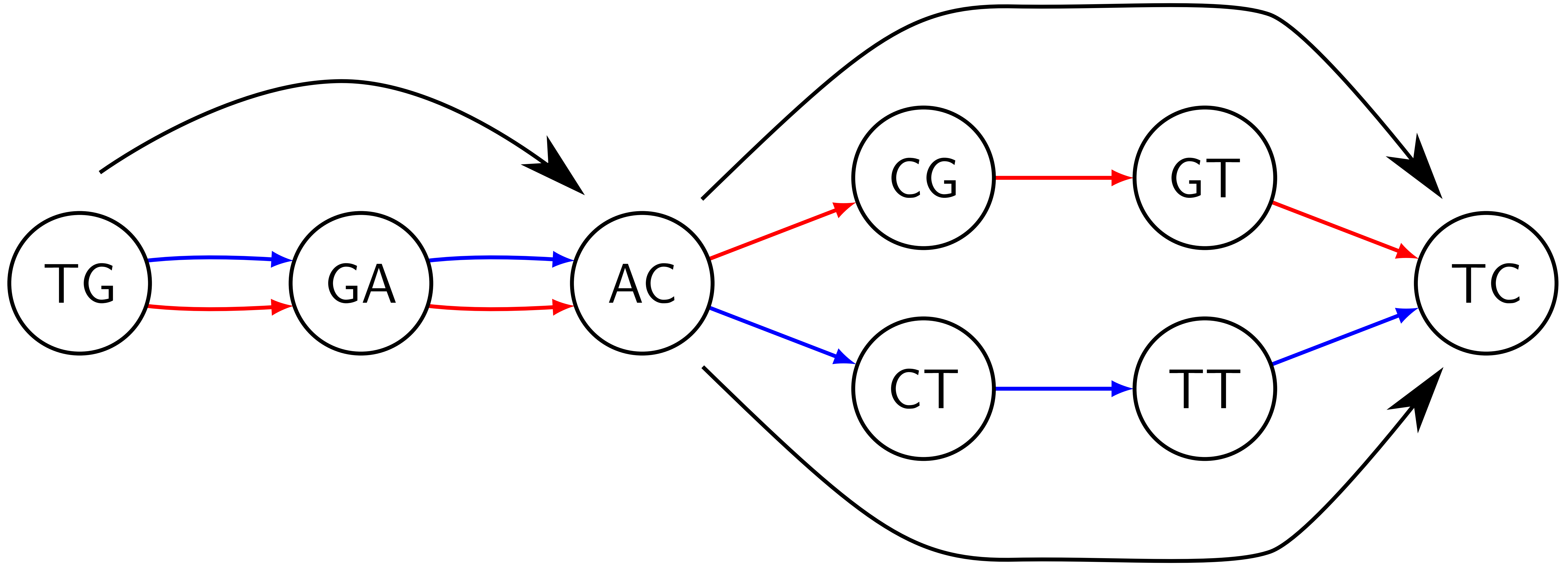




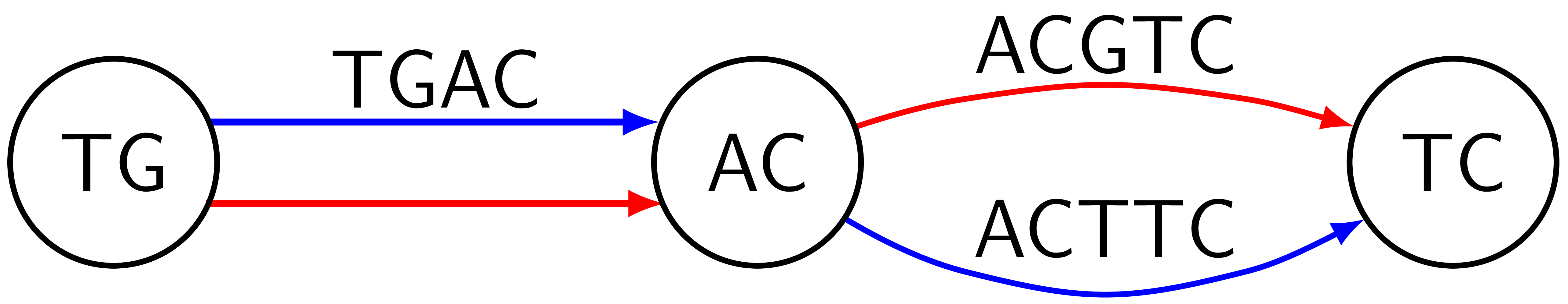
Compaction



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.

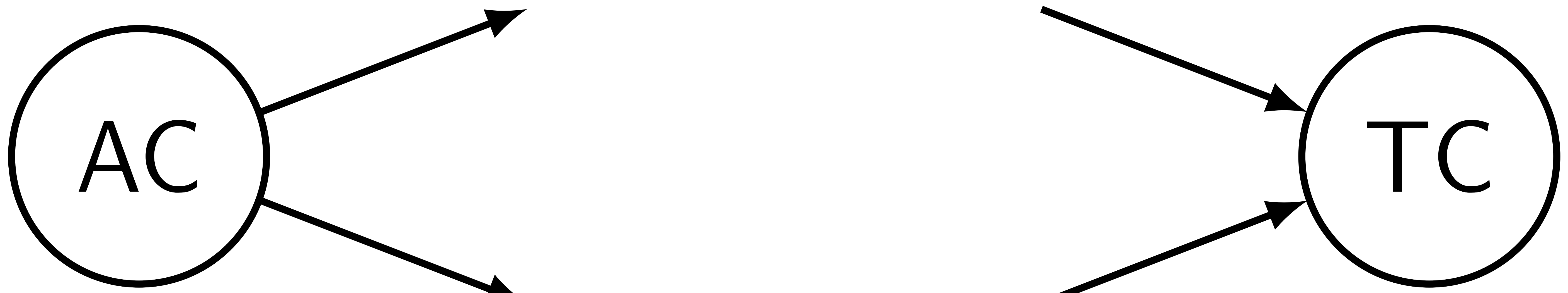
Earlier work: based on suffix arrays/trees Sibelia & SplitMEM handled > 60 E.Coli genomes.

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:

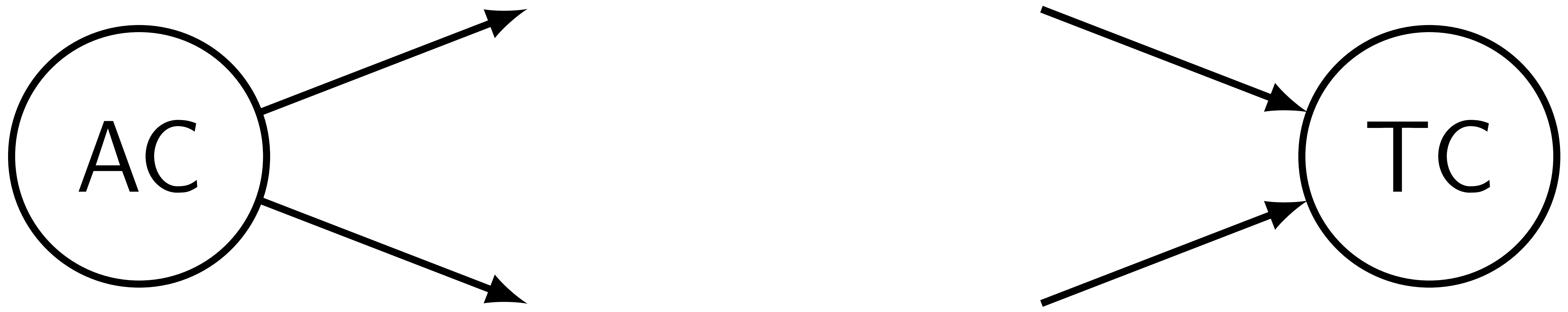
- v has ≥ 2 distinct outgoing or incoming edges:



Junctions

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

- v has ≥ 2 distinct outgoing or incoming edges:

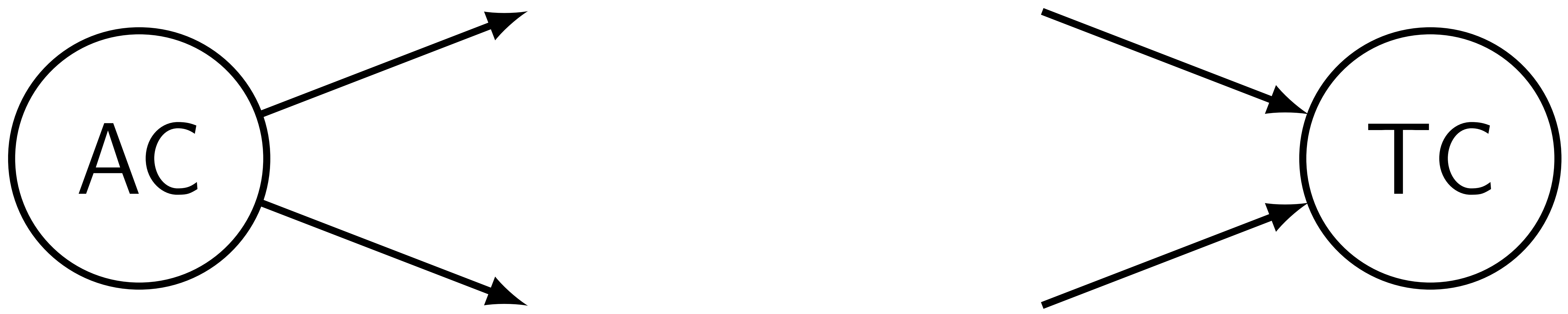


- v is the first or the last k-mer of an input string

Junctions

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

- v has ≥ 2 distinct outgoing or incoming edges:

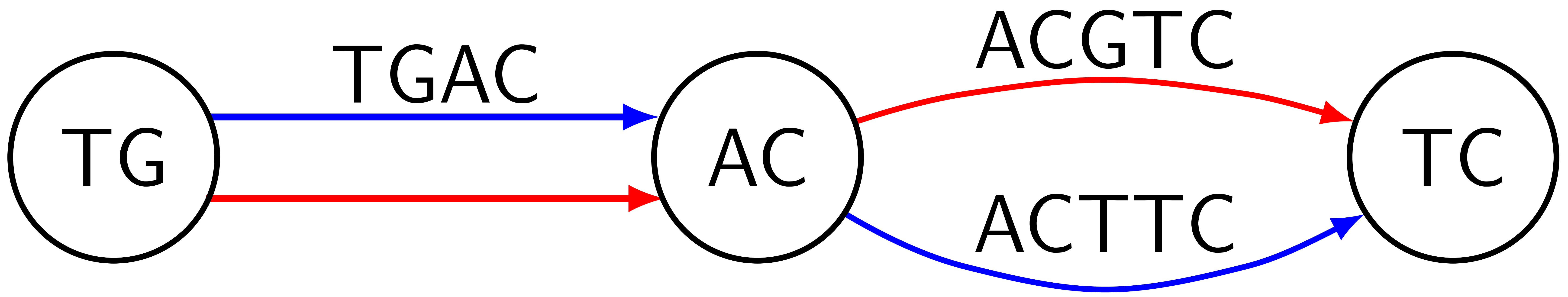


- v is the first or the last k-mer of an input string

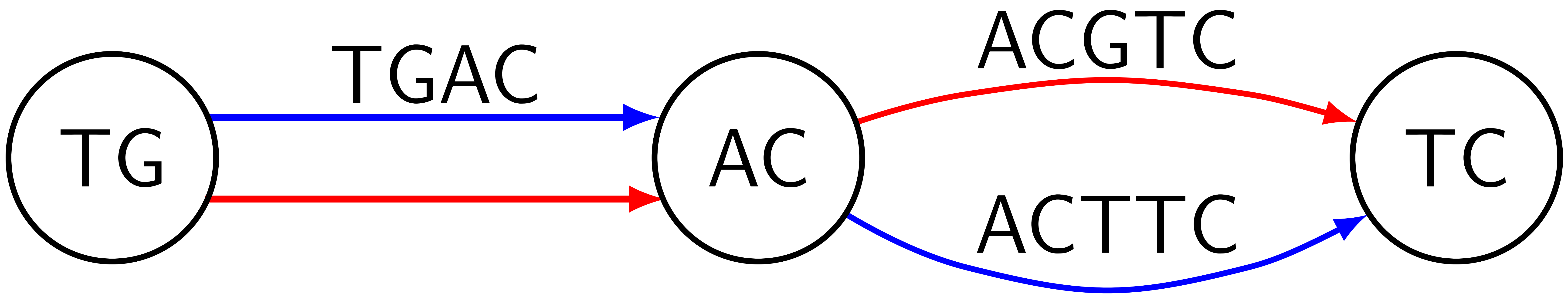
Facts:

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

Observations

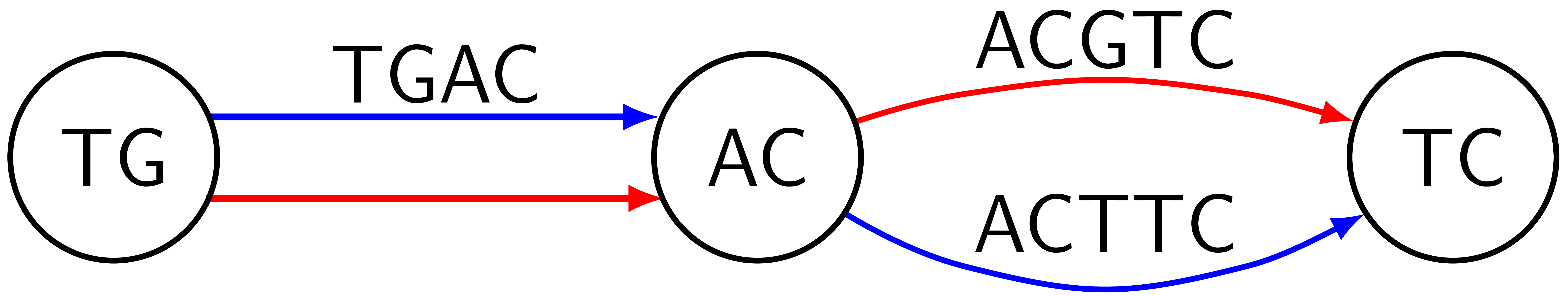


Observations



TG GA AC CG GT TC

Observations



TG GA AC CG GT TC

$$\mathrm{TG} \rightarrow \mathrm{AC} \rightarrow \mathrm{TC}$$

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 $s \in S$ do
2:     for $1 \leq i < |s| - k$ 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 $s \in S$ do
7:     for $1 \leq i < |s| - k$ do
8:         if $C[s, i]$ = marked and $s[i..i+k-1]$ is not a sentinel then
9:             $in \leftarrow 0$                ▷ Number of entering edges
10:            $out \leftarrow 0$               ▷ Number of leaving edges
11:            for $c \in \{A, C, G, T\}$ do    ▷ Consider possible edges and count how many of them exist
12:                if $v \cdot c \in E$ then    ▷ The symbol $\cdot$ depicts string concatenation
13:                    $out \leftarrow out + 1$
14:                if $c \cdot v \in E$ then
15:                    $in \leftarrow in + 1$
16:            if $in = 1$ and $out = 1$ then   ▷ If the $k$-mer at $i$ is not a junction.
17:                $C[s, i] \leftarrow$ Unmarked
18: return $C$
```

The Naive Algorithm

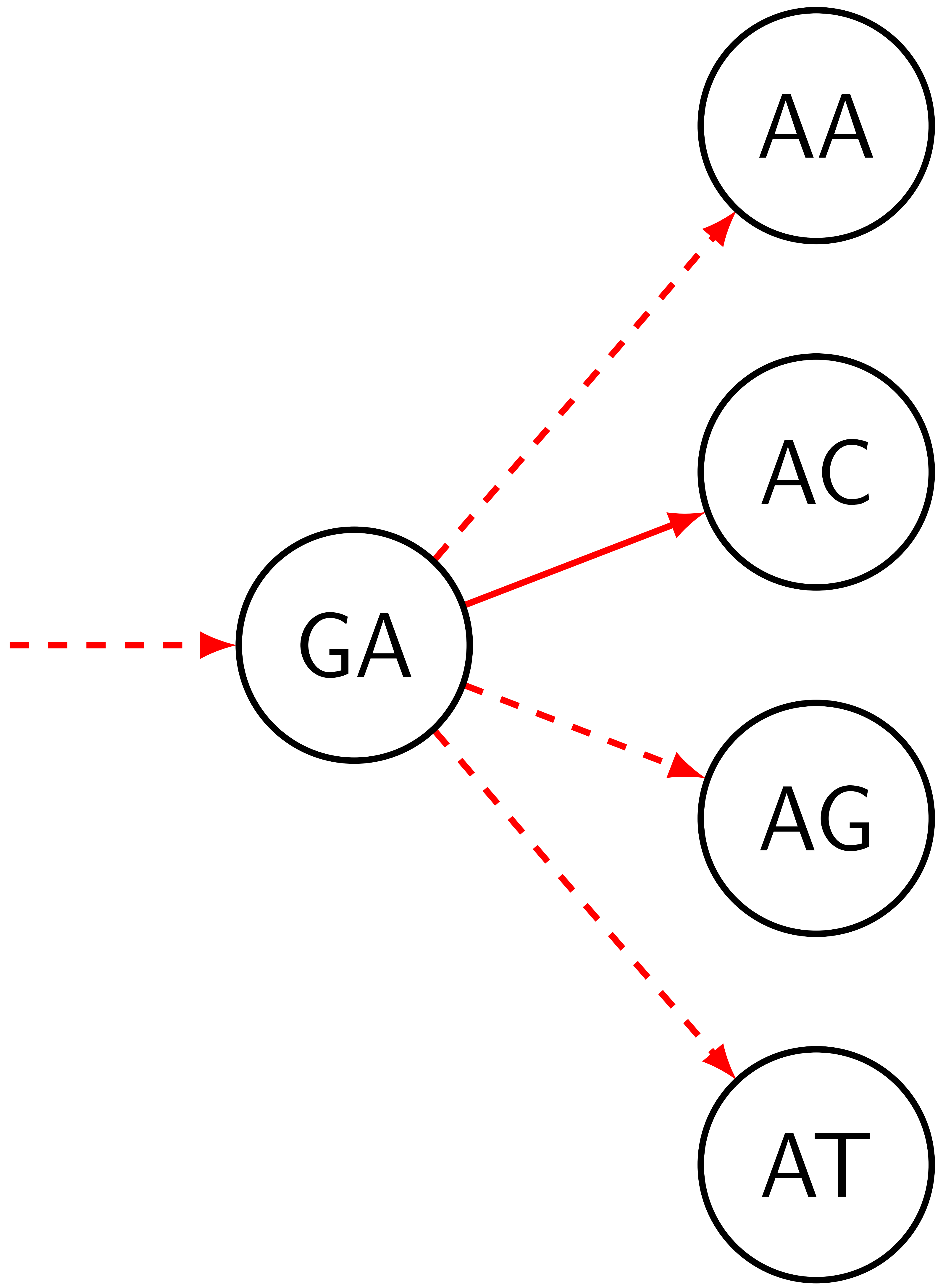
A naive way:

- 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

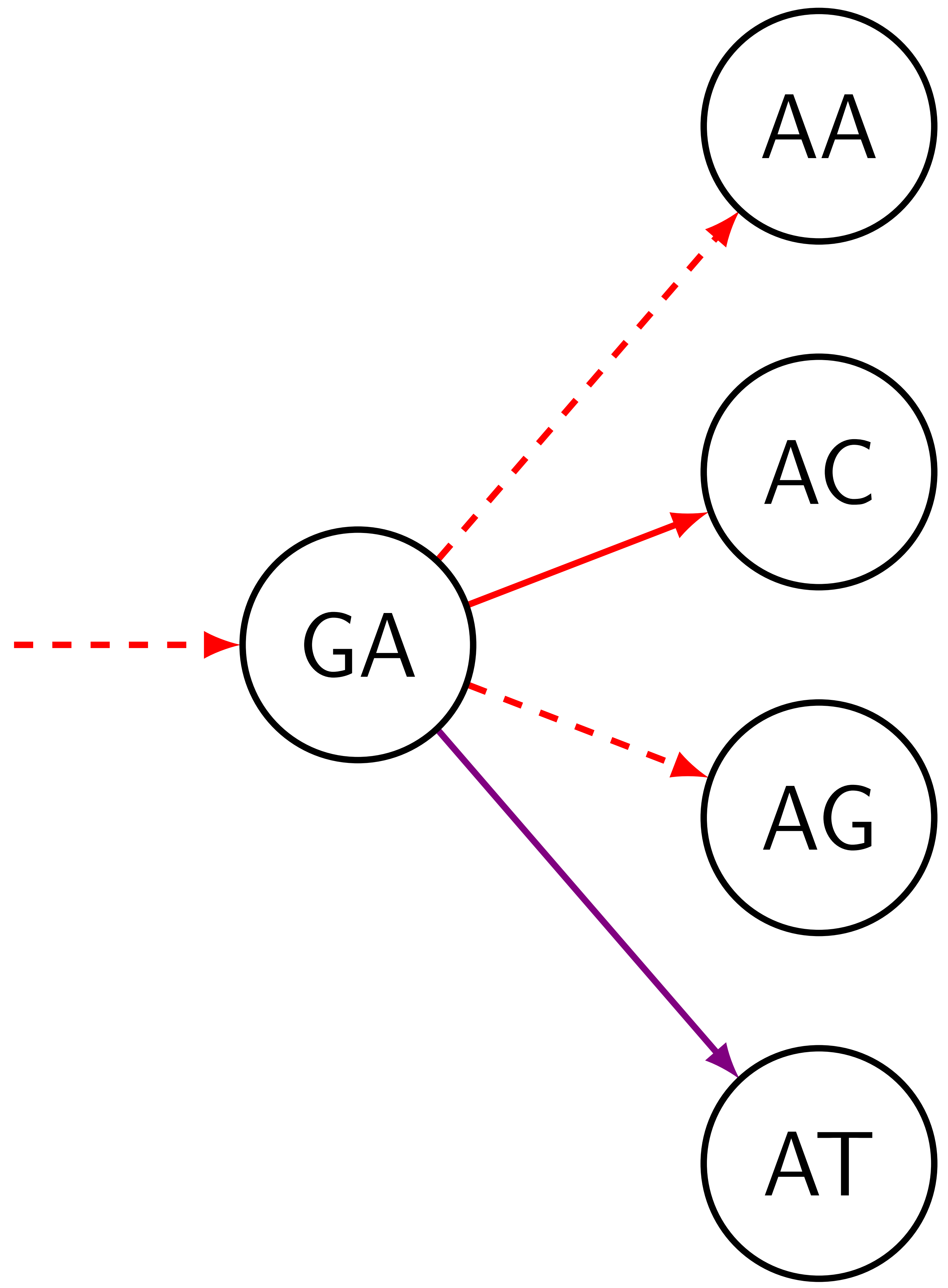
Example: Bloom Filter $= \{$ GA $\rightarrow$ AC $\}$

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

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

An Example

Bloom Filter = { GA → AC, GA → AT }



The purple edge is a false positive.

The Two Pass Algorithm

How to eliminate false positives?

The Two Pass Algorithm

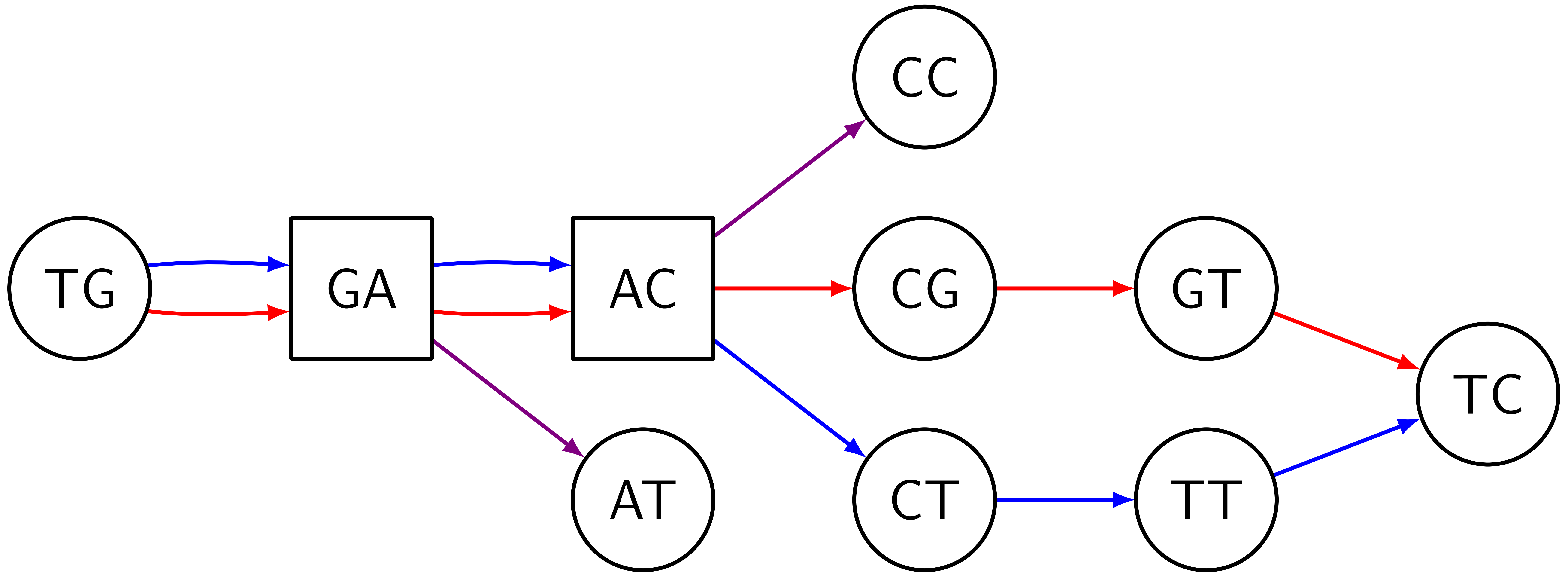
How to eliminate false positives?

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

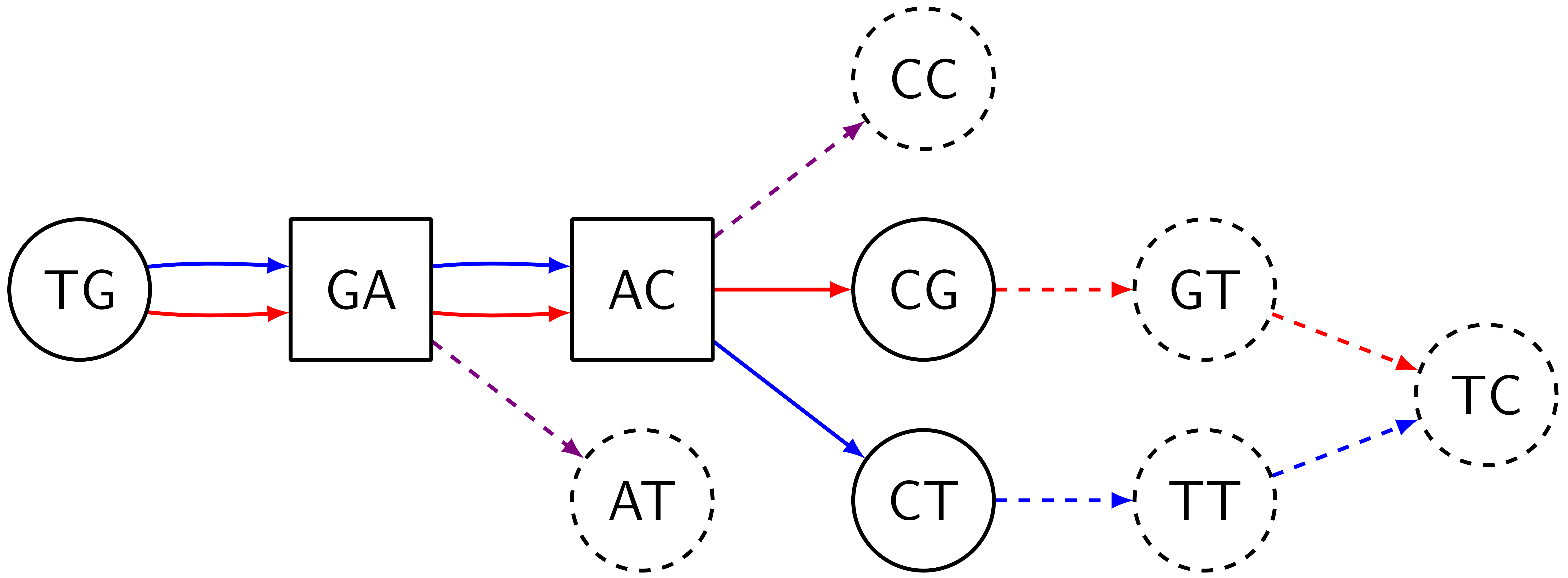
Here edges stored in the Bloom filter, purple ones are false positives:



Junction candidates: GA & AC

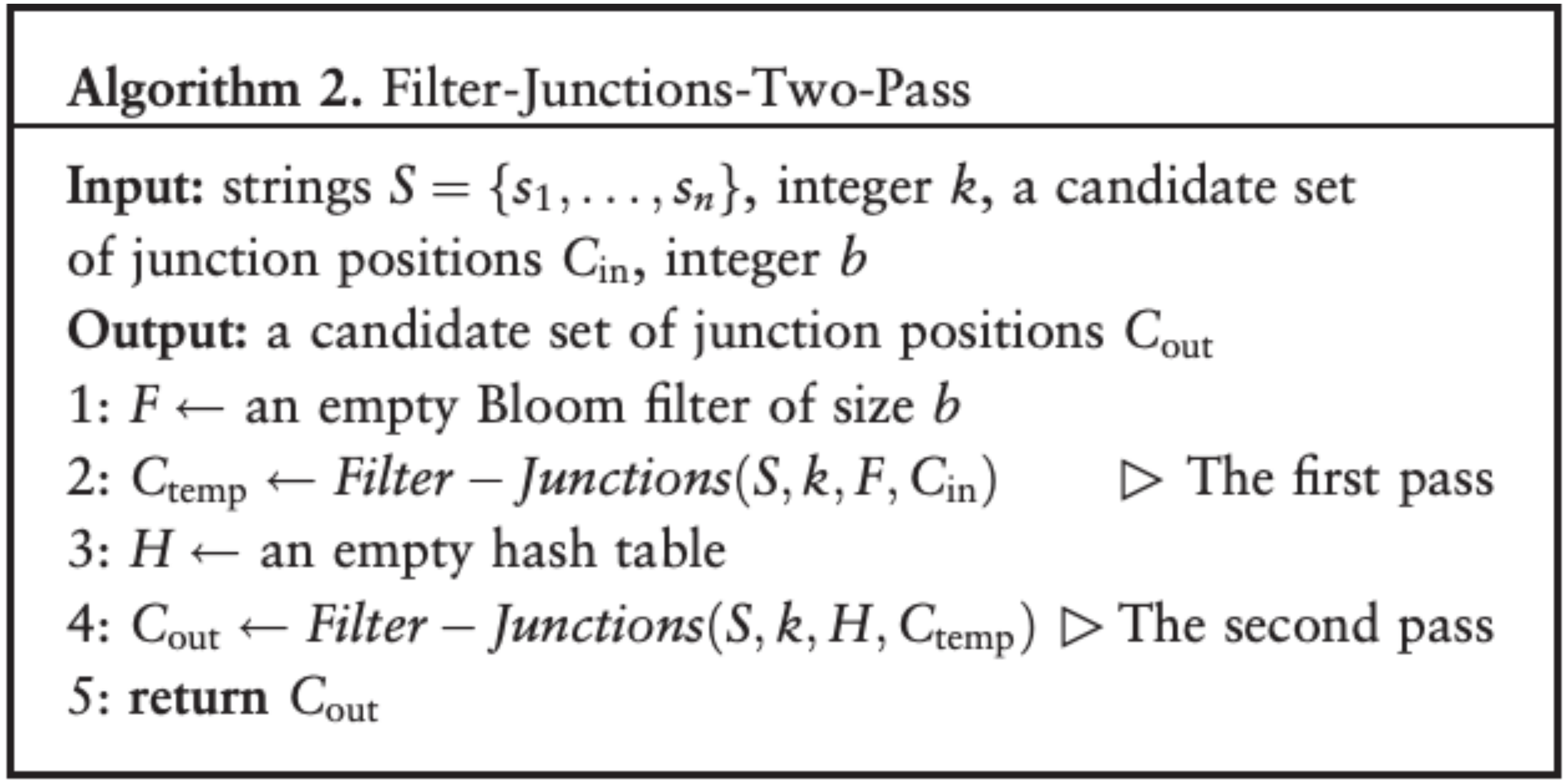
An Example: the Second Step

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



Junction: AC

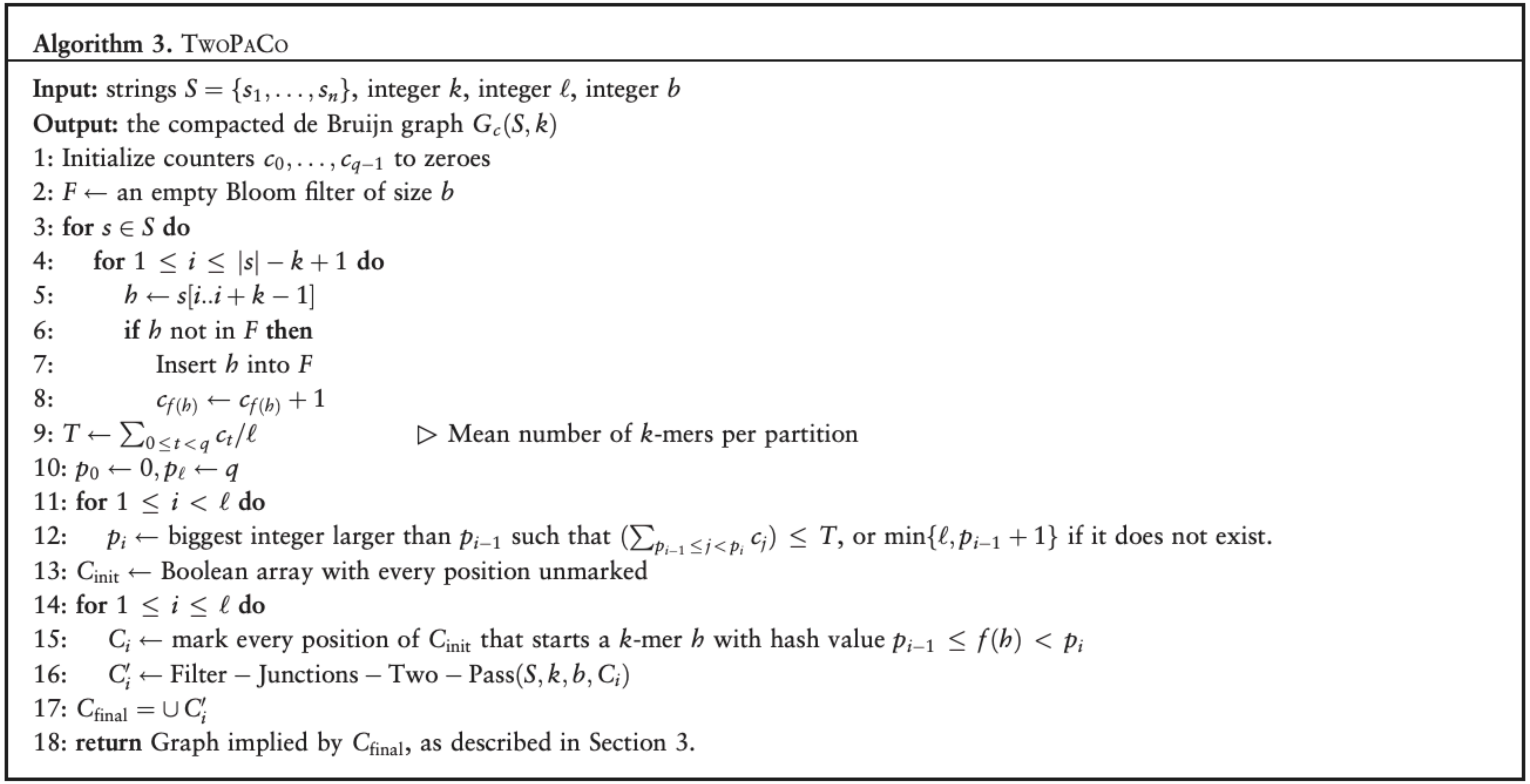
The TwoPass Algorithm



Algorithm 2. Filter-Junctions-Two-Pass

```
Input: strings $S = \{s_1, \dots, s_n\}$, integer $k$, a candidate set
of junction positions $C_{\text{in}}$, integer $b$
Output: a candidate set of junction positions $C_{\text{out}}$
1: $F \leftarrow$ an empty Bloom filter of size $b$
2: $C_{\text{temp}} \leftarrow \mathit{Filter-Junctions}(S, k, F, C_{\text{in}})$        $\triangleright$ The first pass
3: $H \leftarrow$ an empty hash table
4: $C_{\text{out}} \leftarrow \mathit{Filter-Junctions}(S, k, H, C_{\text{temp}})$ $\triangleright$ The second pass
5: return $C_{\text{out}}$
```

The TwoPaCo algorithm



Algorithm 3. TwoPaCo

```
Input: strings $S = \{s_1, \ldots, s_n\}$, integer $k$, integer $\ell$, 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 $C_{\text{init}}$ 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 $C_{\text{final}}$, 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)

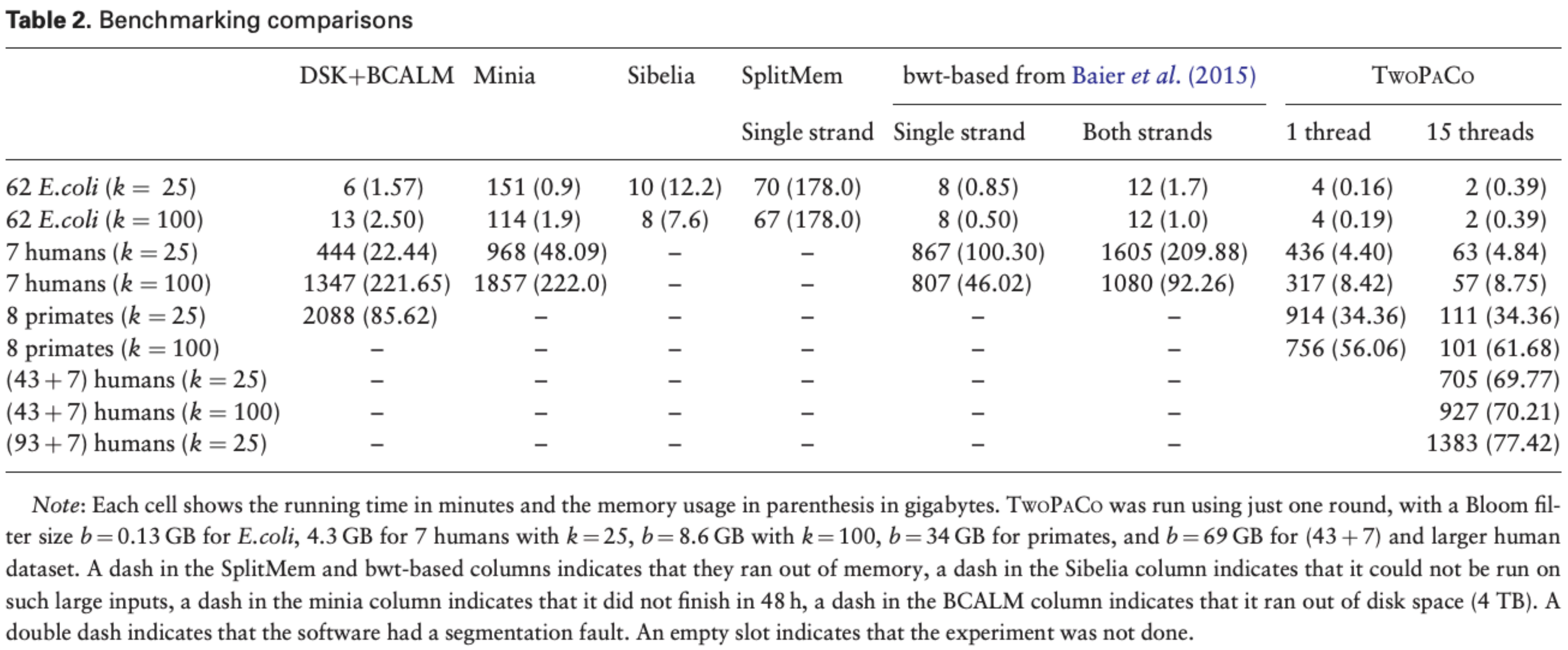


Table 2. Benchmarking comparisons

	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)

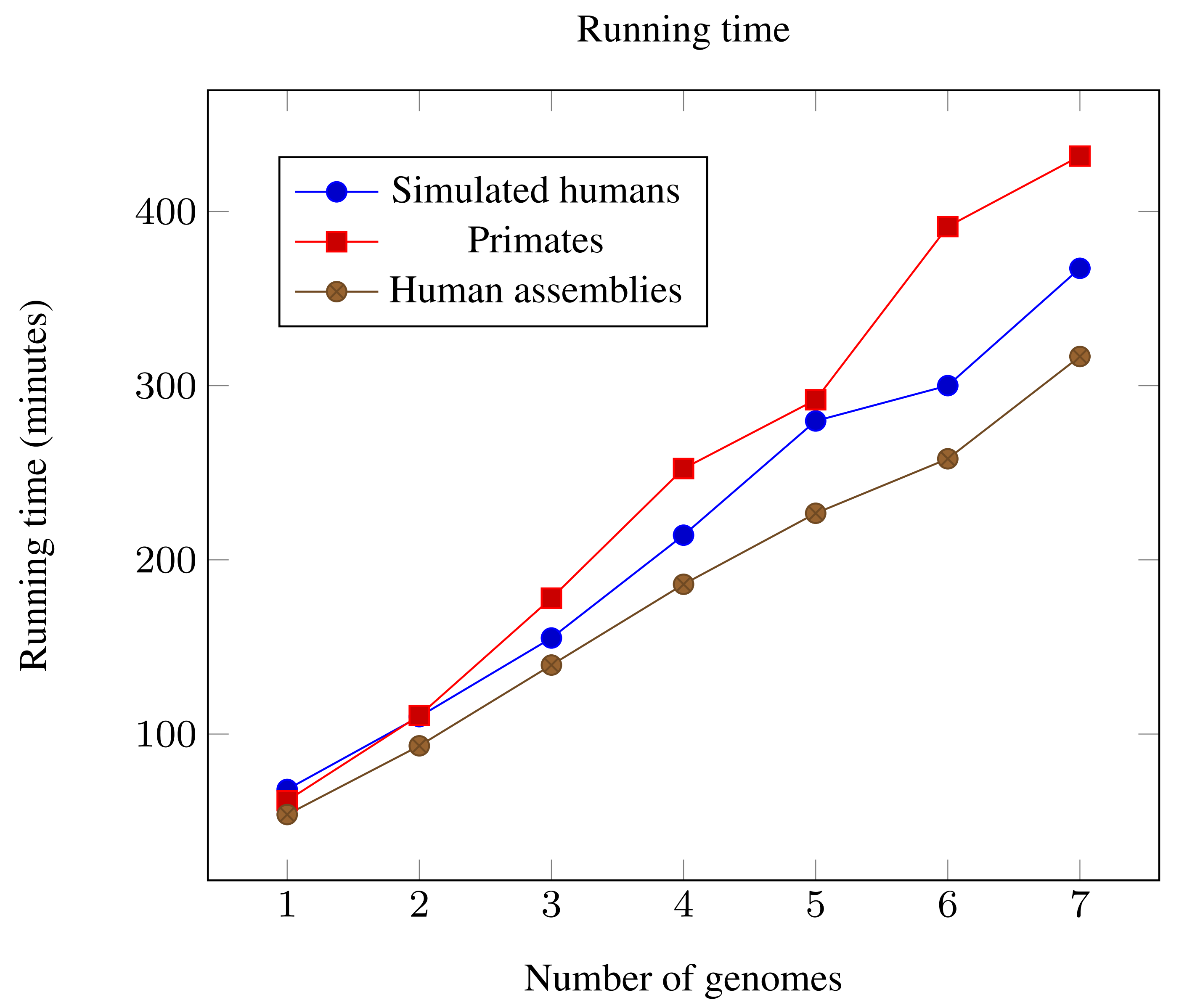
Note: Each cell shows the running time in minutes and the memory usage in parenthesis in gigabytes. TwoPaCo was run using just one round, with a Bloom filter size $b = 0.13$ GB for *E.coli*, 4.3 GB for 7 humans with $k = 25$, $b = 8.6$ GB with $k = 100$, $b = 34$ GB for primates, and $b = 69$ GB for (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.

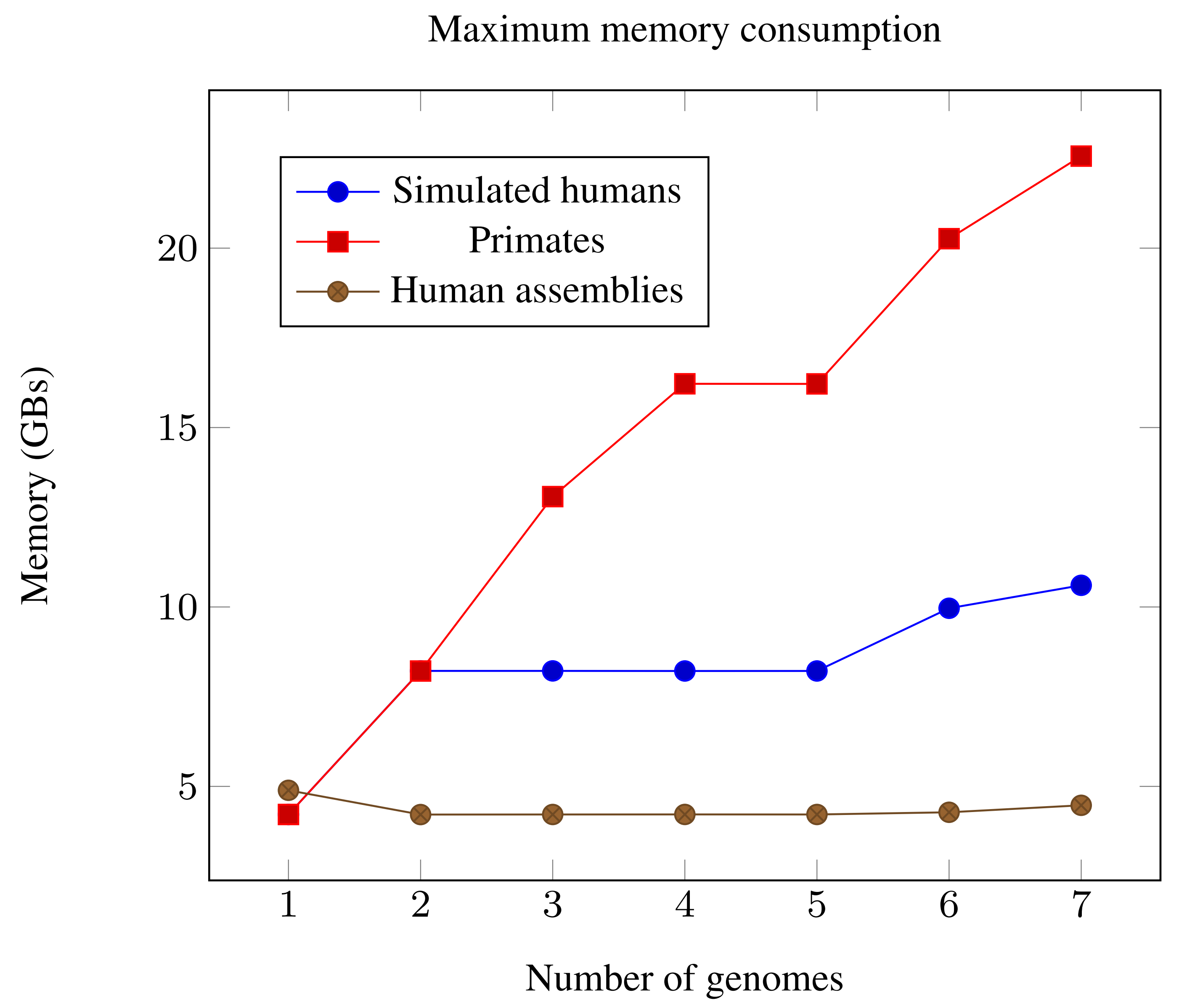
Conclusion & Future Work

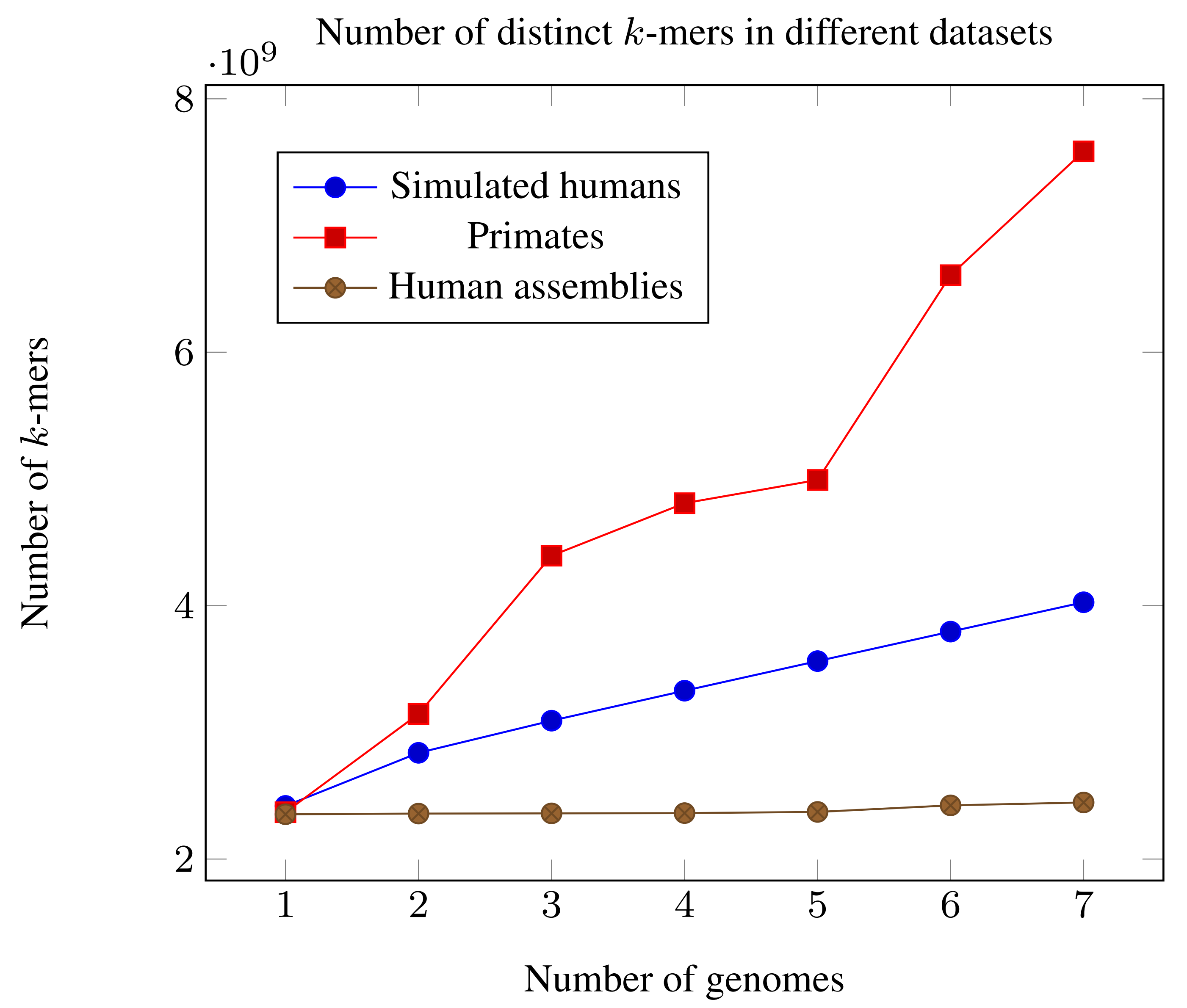
Can potentially facilitate:

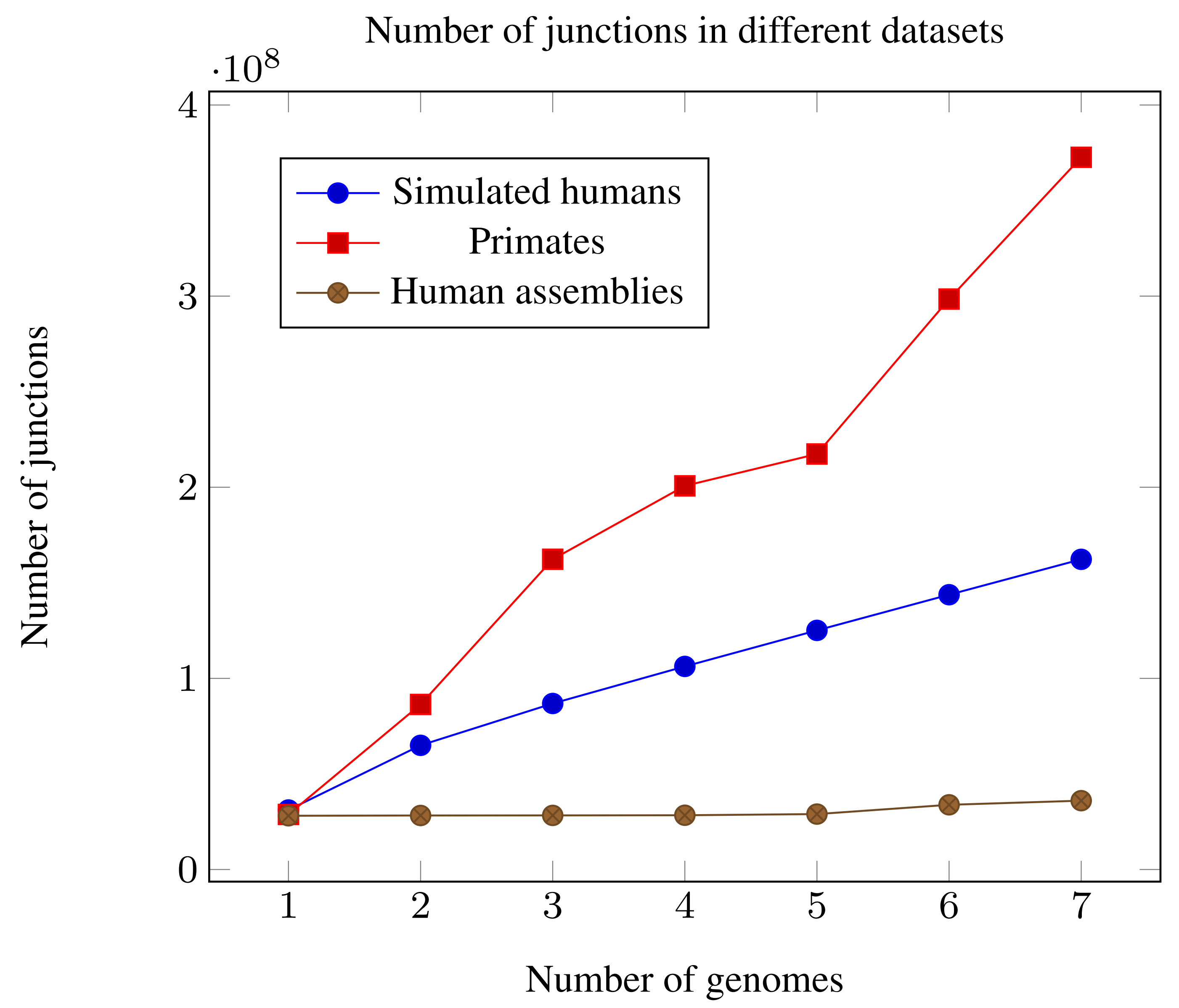
- Visualization
- Synteny mining (Sibelia)
- Structural variations analysis
- ...

Input Size vs. Performance

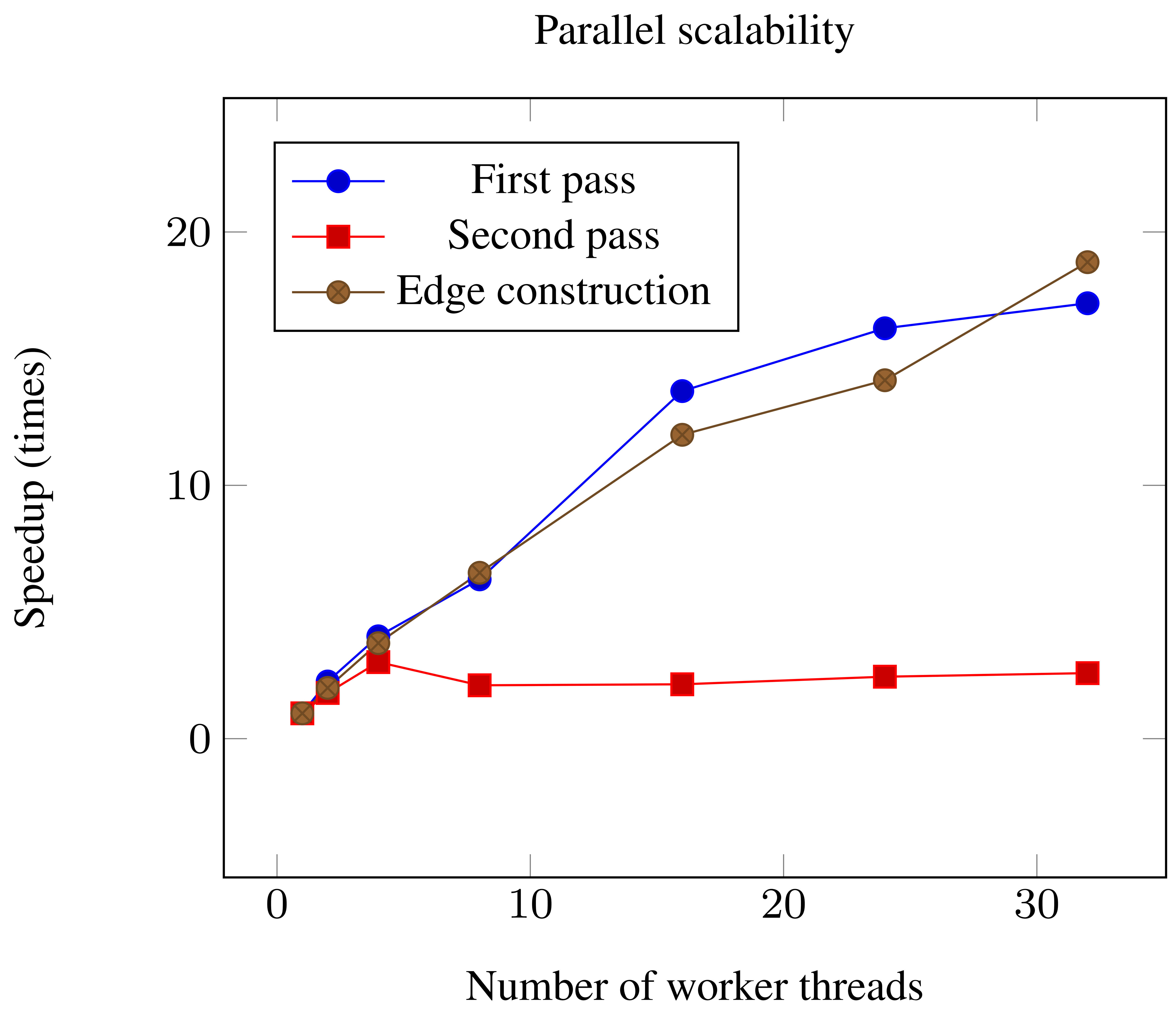








Parallel Scalability

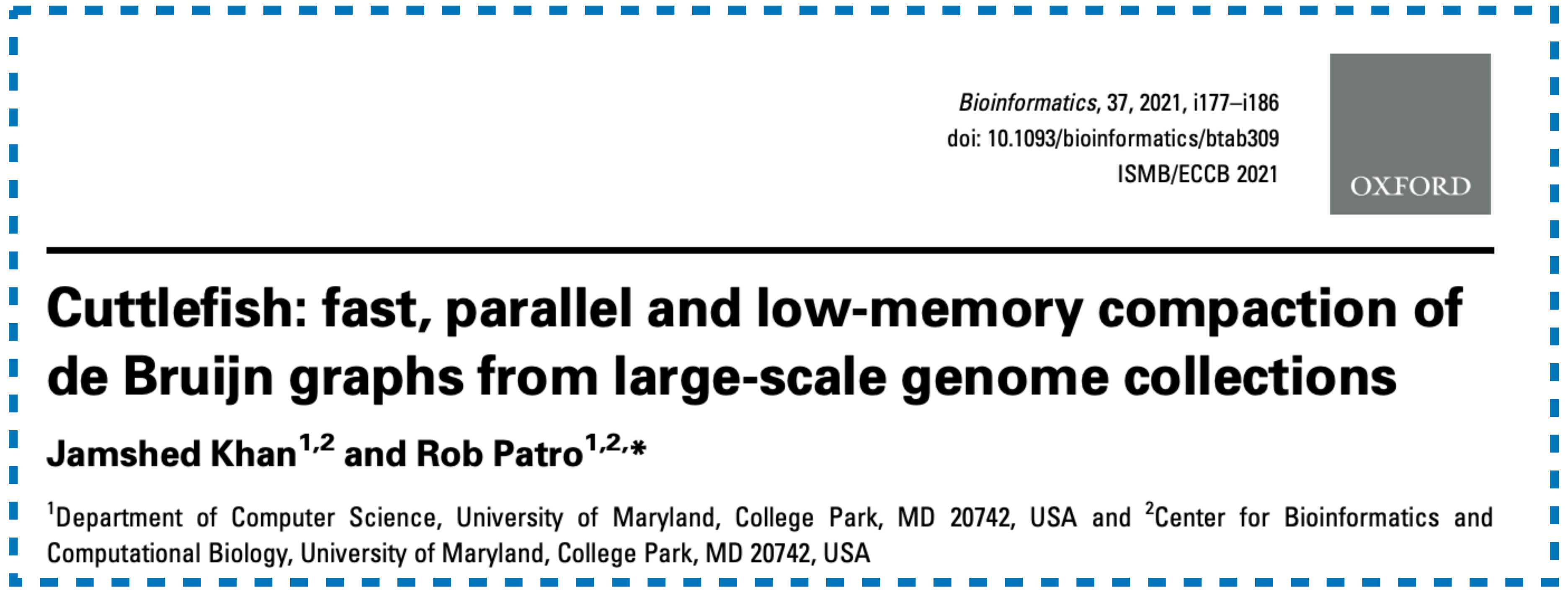


Splitting

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

Can we do even better?

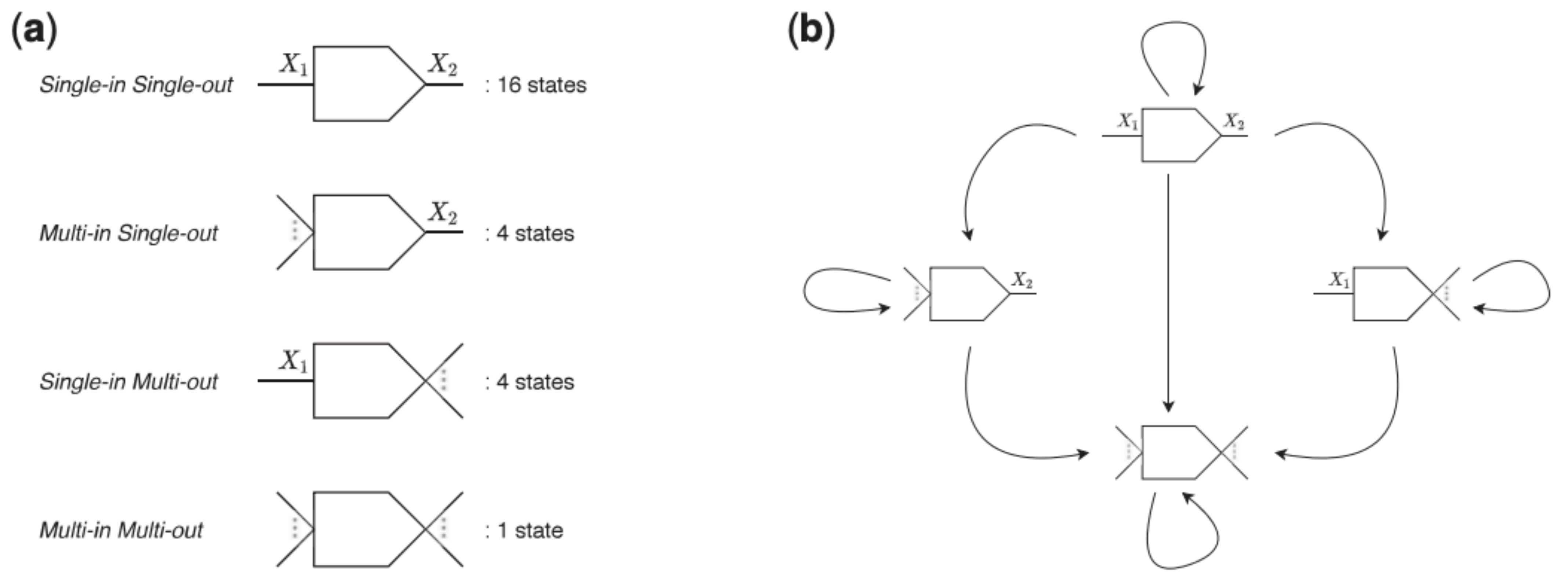

Bioinformatics, 37, 2021, i177–i186
doi: 10.1093/bioinformatics/btab309
ISMB/ECCB 2021

OXFORD

Cuttlefish: fast, parallel and low-memory compaction of de Bruijn graphs from large-scale genome collections

Jamshed Khan[1,2] and Rob Patro[1,2,*]

[1]Department of Computer Science, University of Maryland, College Park, MD 20742, USA and [2]Center for Bioinformatics and Computational Biology, University of Maryland, College Park, MD 20742, USA



Key ideas:

Just like TwoPaCo, make use of explicit traversal of references to identify junction nodes

Build a minimal perfect hash (BBhash here) over the set of k-mers

Associate each k-mer with a finite state automaton, denoting its topological status — 26 states requires 5-bits/k-mer

Walking the references and updating the states results in correct status for each k-mer, and unitigs can then be extracted as sequences between the junction nodes.

Cuttlefish

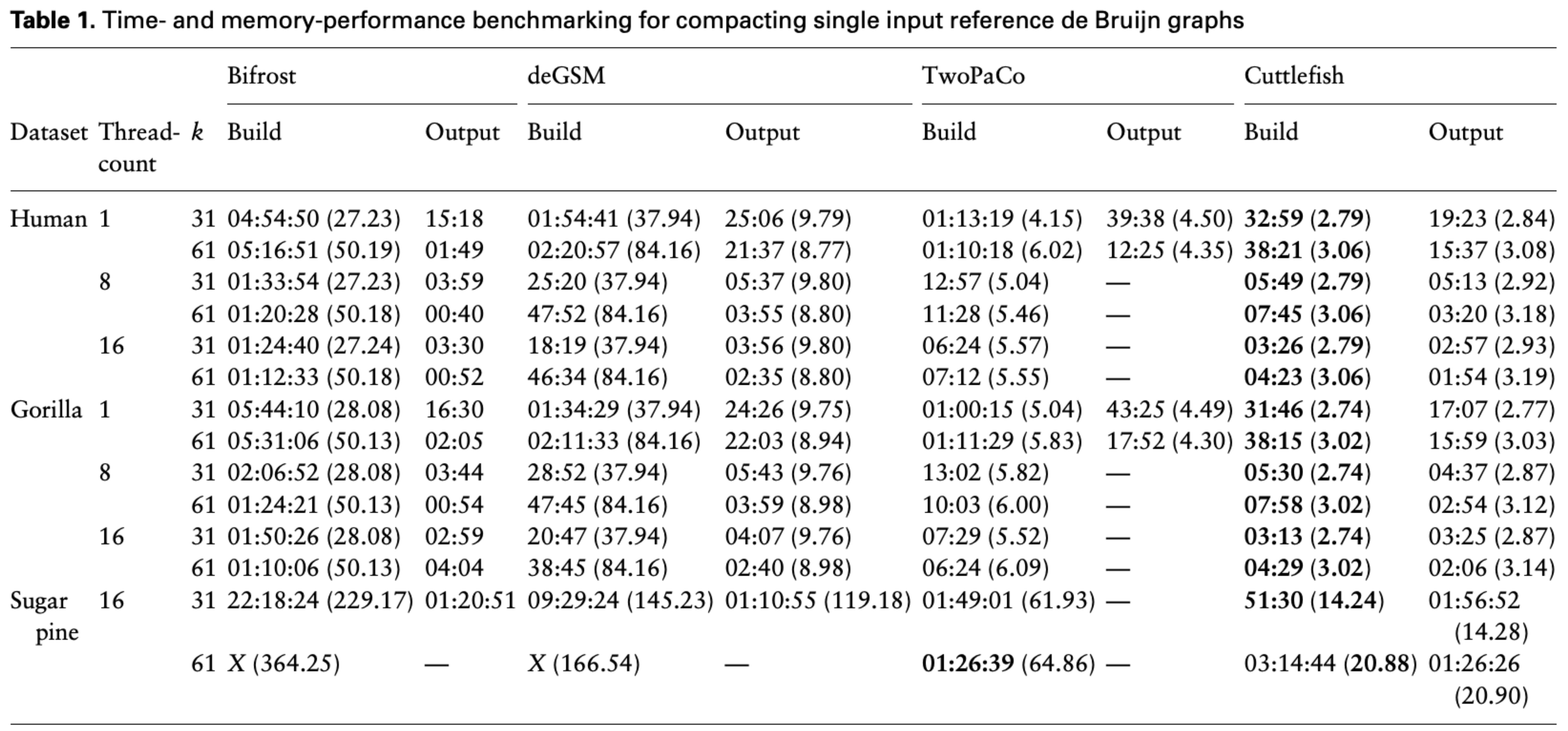


Table 1. Time- and memory-performance benchmarking for compacting single input reference de Bruijn graphs

			Bifrost		deGSM		TwoPaCo		Cuttlefish	
Dataset	Thread-count	*k*	Build	Output	Build	Output	Build	Output	Build	Output
Human	1	31	04:54:50 (27.23)	15:18	01:54:41 (37.94)	25:06 (9.79)	01:13:19 (4.15)	39:38 (4.50)	**32:59** (**2.79**)	19:23 (2.84)
		61	05:16:51 (50.19)	01:49	02:20:57 (84.16)	21:37 (8.77)	01:10:18 (6.02)	12:25 (4.35)	**38:21** (**3.06**)	15:37 (3.08)
	8	31	01:33:54 (27.23)	03:59	25:20 (37.94)	05:37 (9.80)	12:57 (5.04)	—	**05:49** (**2.79**)	05:13 (2.92)
		61	01:20:28 (50.18)	00:40	47:52 (84.16)	03:55 (8.80)	11:28 (5.46)	—	**07:45** (**3.06**)	03:20 (3.18)
	16	31	01:24:40 (27.24)	03:30	18:19 (37.94)	03:56 (9.80)	06:24 (5.57)	—	**03:26** (**2.79**)	02:57 (2.93)
		61	01:12:33 (50.18)	00:52	46:34 (84.16)	02:35 (8.80)	07:12 (5.55)	—	**04:23** (**3.06**)	01:54 (3.19)
Gorilla	1	31	05:44:10 (28.08)	16:30	01:34:29 (37.94)	24:26 (9.75)	01:00:15 (5.04)	43:25 (4.49)	**31:46** (**2.74**)	17:07 (2.77)
		61	05:31:06 (50.13)	02:05	02:11:33 (84.16)	22:03 (8.94)	01:11:29 (5.83)	17:52 (4.30)	**38:15** (**3.02**)	15:59 (3.03)
	8	31	02:06:52 (28.08)	03:44	28:52 (37.94)	05:43 (9.76)	13:02 (5.82)	—	**05:30** (**2.74**)	04:37 (2.87)
		61	01:24:21 (50.13)	00:54	47:45 (84.16)	03:59 (8.98)	10:03 (6.00)	—	**07:58** (**3.02**)	02:54 (3.12)
	16	31	01:50:26 (28.08)	02:59	20:47 (37.94)	04:07 (9.76)	07:29 (5.52)	—	**03:13** (**2.74**)	03:25 (2.87)
		61	01:10:06 (50.13)	04:04	38:45 (84.16)	02:40 (8.98)	06:24 (6.09)	—	**04:29** (**3.02**)	02:06 (3.14)
Sugar pine	16	31	22:18:24 (229.17)	01:20:51	09:29:24 (145.23)	01:10:55 (119.18)	01:49:01 (61.93)	—	**51:30** (**14.24**)	01:56:52 (14.28)
		61	*X* (364.25)	—	*X* (166.54)	—	**01:26:39** (64.86)	—	03:14:44 (**20.88**)	01:26:26 (20.90)

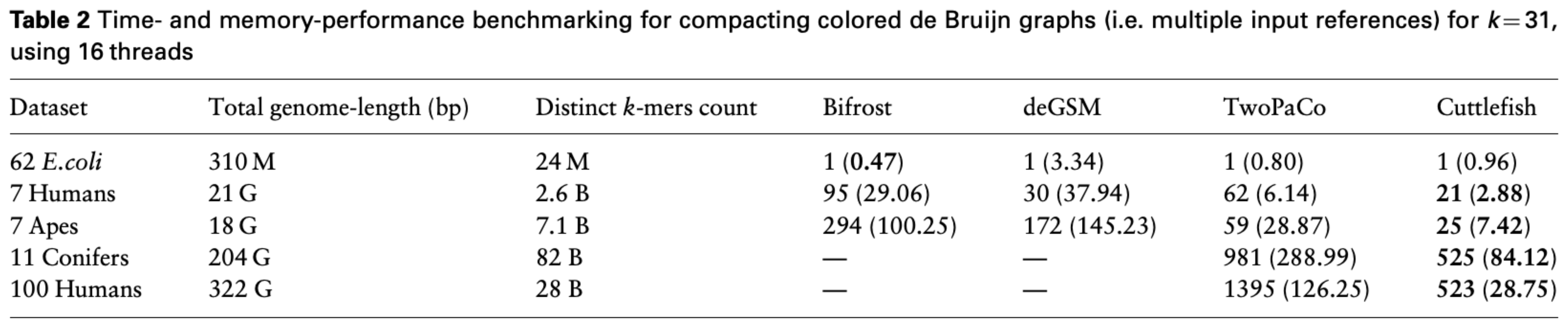
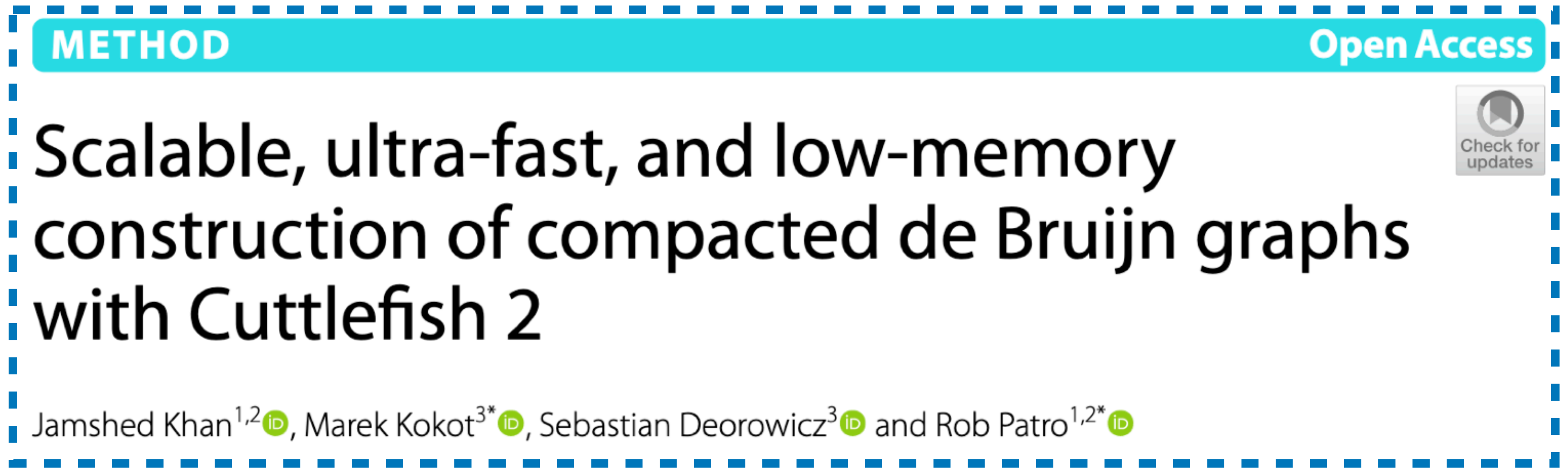


Table 2 Time- and memory-performance benchmarking for compacting colored de Bruijn graphs (i.e. multiple input references) for $k = 31$, using 16 threads

Dataset	Total genome-length (bp)	Distinct *k*-mers count	Bifrost	deGSM	TwoPaCo	Cuttlefish
62 *E.coli*	310 M	24 M	1 (**0.47**)	1 (3.34)	1 (0.80)	1 (0.96)
7 Humans	21 G	2.6 B	95 (29.06)	30 (37.94)	62 (6.14)	**21** (**2.88**)
7 Apes	18 G	7.1 B	294 (100.25)	172 (145.23)	59 (28.87)	**25** (**7.42**)
11 Conifers	204 G	82 B	—	—	981 (288.99)	**525** (**84.12**)
100 Humans	322 G	28 B	—	—	1395 (126.25)	**523** (**28.75**)

Cuttlefish 2



METHOD Open Access

Scalable, ultra-fast, and low-memory construction of compacted de Bruijn graphs with Cuttlefish 2

Jamshed Khan[1,2], Marek Kokot[3*], Sebastian Deorowicz[3] and Rob Patro[1,2*]

Can generalize the cuttlefish algorithm to work on raw sequencing data in addition to reference genomes. Leads to a state-of-the-art compacted dBG construction algorithm.

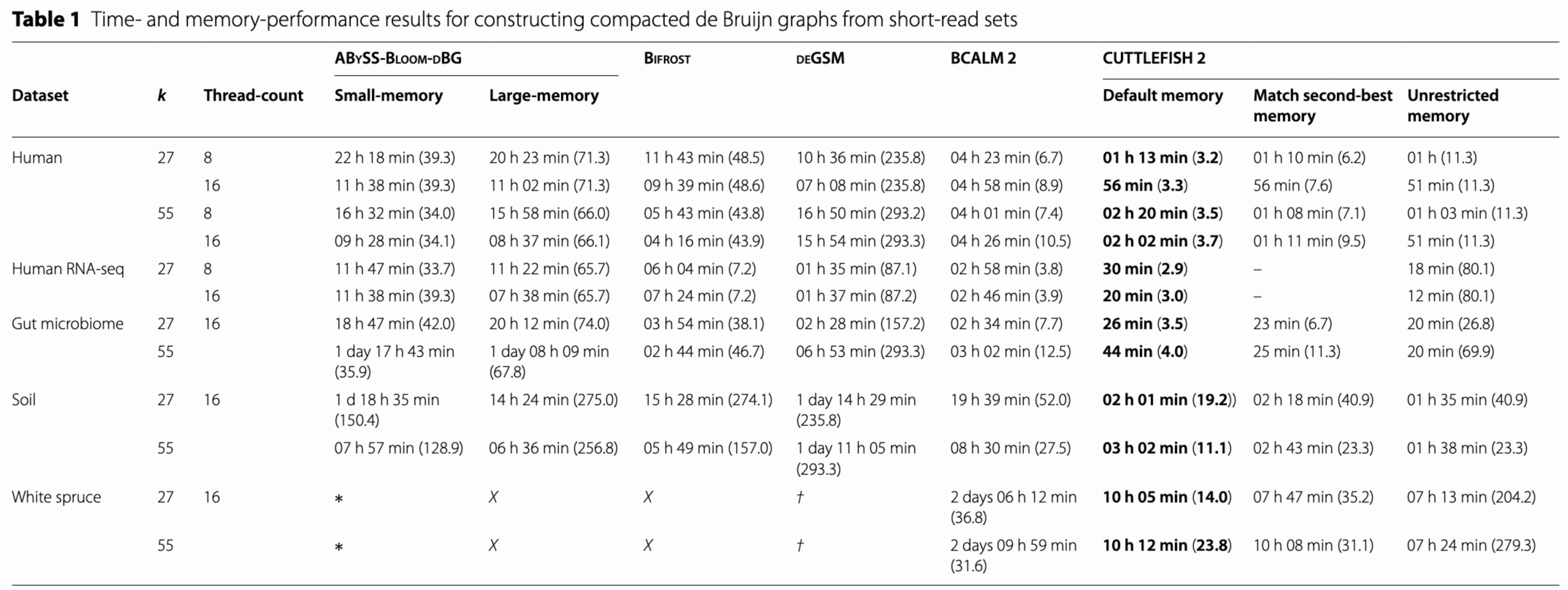


Table 1 Time- and memory-performance results for constructing compacted de Bruijn graphs from short-read sets

			ABySS-Bloom-dBG		Bifrost	deGSM	BCALM 2	CUTTLEFISH 2		
Dataset	***k***	**Thread-count**	**Small-memory**	**Large-memory**				**Default memory**	**Match second-best memory**	**Unrestricted memory**
Human	27	8	22 h 18 min (39.3)	20 h 23 min (71.3)	11 h 43 min (48.5)	10 h 36 min (235.8)	04 h 23 min (6.7)	**01 h 13 min** (**3.2**)	01 h 10 min (6.2)	01 h (11.3)
		16	11 h 38 min (39.3)	11 h 02 min (71.3)	09 h 39 min (48.6)	07 h 08 min (235.8)	04 h 58 min (8.9)	**56 min** (**3.3**)	56 min (7.6)	51 min (11.3)
	55	8	16 h 32 min (34.0)	15 h 58 min (66.0)	05 h 43 min (43.8)	16 h 50 min (293.2)	04 h 01 min (7.4)	**02 h 20 min** (**3.5**)	01 h 08 min (7.1)	01 h 03 min (11.3)
		16	09 h 28 min (34.1)	08 h 37 min (66.1)	04 h 16 min (43.9)	15 h 54 min (293.3)	04 h 26 min (10.5)	**02 h 02 min** (**3.7**)	01 h 11 min (9.5)	51 min (11.3)
Human RNA-seq	27	8	11 h 47 min (33.7)	11 h 22 min (65.7)	06 h 04 min (7.2)	01 h 35 min (87.1)	02 h 58 min (3.8)	**30 min** (**2.9**)	–	18 min (80.1)
		16	11 h 38 min (39.3)	07 h 38 min (65.7)	07 h 24 min (7.2)	01 h 37 min (87.2)	02 h 46 min (3.9)	**20 min** (**3.0**)	–	12 min (80.1)
Gut microbiome	27	16	18 h 47 min (42.0)	20 h 12 min (74.0)	03 h 54 min (38.1)	02 h 28 min (157.2)	02 h 34 min (7.7)	**26 min** (**3.5**)	23 min (6.7)	20 min (26.8)
	55		1 day 17 h 43 min (35.9)	1 day 08 h 09 min (67.8)	02 h 44 min (46.7)	06 h 53 min (293.3)	03 h 02 min (12.5)	**44 min** (**4.0**)	25 min (11.3)	20 min (69.9)
Soil	27	16	1 d 18 h 35 min (150.4)	14 h 24 min (275.0)	15 h 28 min (274.1)	1 day 14 h 29 min (235.8)	19 h 39 min (52.0)	**02 h 01 min** (**19.2**))	02 h 18 min (40.9)	01 h 35 min (40.9)
	55		07 h 57 min (128.9)	06 h 36 min (256.8)	05 h 49 min (157.0)	1 day 11 h 05 min (293.3)	08 h 30 min (27.5)	**03 h 02 min** (**11.1**)	02 h 43 min (23.3)	01 h 38 min (23.3)
White spruce	27	16	∗	*X*	*X*	†	2 days 06 h 12 min (36.8)	**10 h 05 min** (**14.0**)	07 h 47 min (35.2)	07 h 13 min (204.2)
	55		∗	*X*	*X*	†	2 days 09 h 59 min (31.6)	**10 h 12 min** (**23.8**)	10 h 08 min (31.1)	07 h 24 min (279.3)

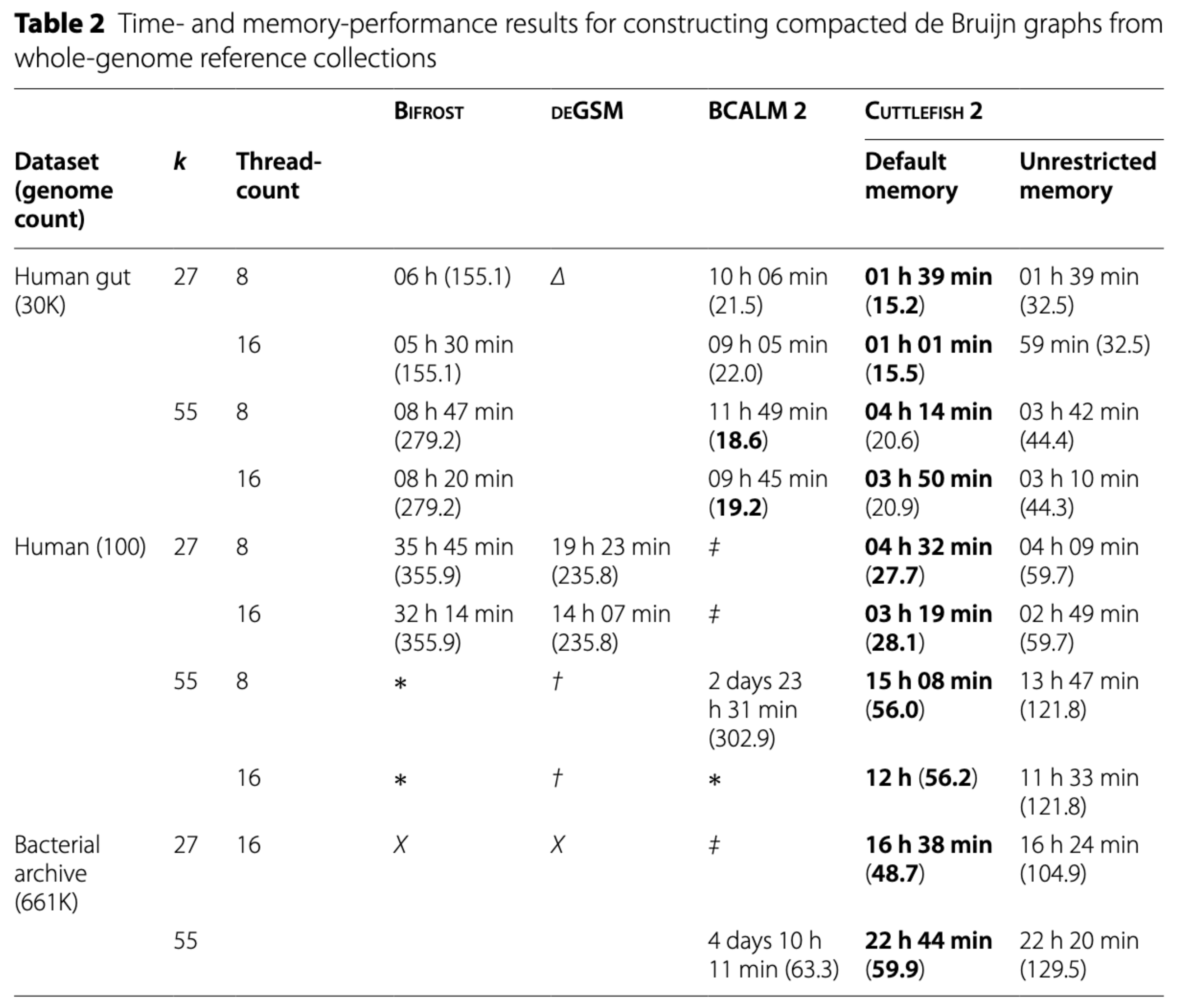


Table 2 Time- and memory-performance results for constructing compacted de Bruijn graphs from whole-genome reference collections

			Bifrost	deGSM	BCALM 2	Cuttlefish 2	
Dataset (genome count)	***k***	**Thread-count**				**Default memory**	**Unrestricted memory**
Human gut (30K)	27	8	06 h (155.1)	*Δ*	10 h 06 min (21.5)	**01 h 39 min** (**15.2**)	01 h 39 min (32.5)
		16	05 h 30 min (155.1)		09 h 05 min (22.0)	**01 h 01 min** (**15.5**)	59 min (32.5)
	55	8	08 h 47 min (279.2)		11 h 49 min (**18.6**)	**04 h 14 min** (20.6)	03 h 42 min (44.4)
		16	08 h 20 min (279.2)		09 h 45 min (**19.2**)	**03 h 50 min** (20.9)	03 h 10 min (44.3)
Human (100)	27	8	35 h 45 min (355.9)	19 h 23 min (235.8)	‡	**04 h 32 min** (**27.7**)	04 h 09 min (59.7)
		16	32 h 14 min (355.9)	14 h 07 min (235.8)	‡	**03 h 19 min** (**28.1**)	02 h 49 min (59.7)
	55	8	∗	†	2 days 23 h 31 min (302.9)	**15 h 08 min** (**56.0**)	13 h 47 min (121.8)
		16	∗	†	∗	**12 h** (**56.2**)	11 h 33 min (121.8)
Bacterial archive (661K)	27	16	*X*	*X*	‡	**16 h 38 min** (**48.7**)	16 h 24 min (104.9)
	55				4 days 10 h 11 min (63.3)	**22 h 44 min** (**59.9**)	22 h 20 min (129.5)

https://www.biorxiv.org/content/10.1101/2021.12.14.472718v1.full.pdf