Module 2 - Assembly

Lecture 10: Genomics

Bioinformatics Algorithms CSC4181/6802

Most slides used are from Ben Langmead's Teaching Materials (www.langmead-lab.org/teaching-materials)

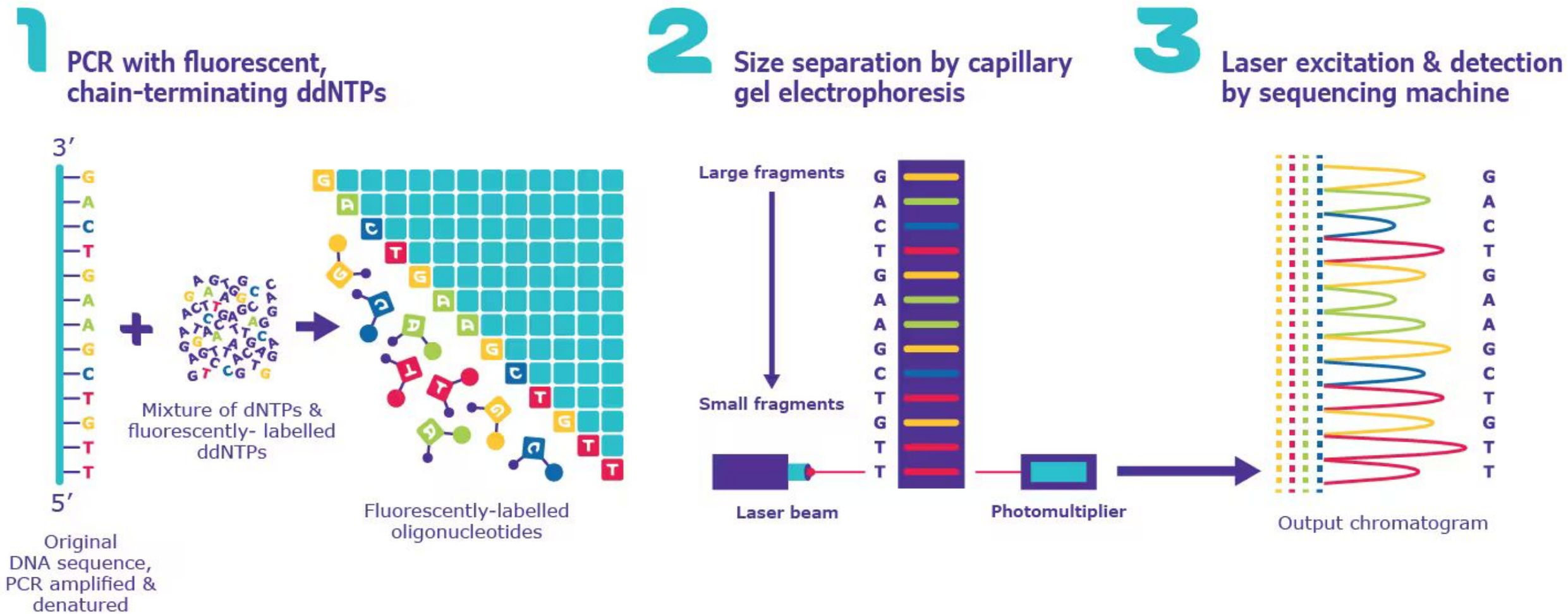
Sequencing Technology

First generation

Sanger sequencing
Maxam and Gilbert
Sanger chain termination

https://www.pacb.com/blog/the-evolution-of-dna-sequencing-tools/

Sanger Sequencing



https://www.sigmaaldrich.com/CA/en/technical-documents/protocol/genomics/sequencing/sanger-sequencing

Sequencing Technology

First generation

Sanger sequencing
Maxam and Gilbert
Sanger chain termination

Infer nucleotide identity using dNTPs, then visualize with electrophoresis

500–1,000 bp fragments

https://www.pacb.com/blog/the-evolution-of-dna-sequencing-tools/

Sequencing Technology

Sanger sequencing
Maxam and Gilbert
Sanger chain termination

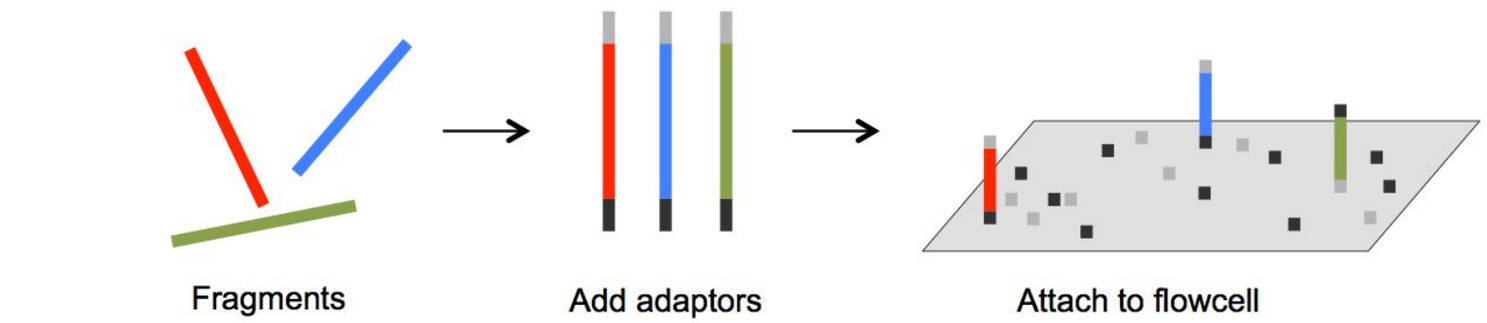
Infer nucleotide identity using dNTPs,
then visualize with electrophoresis

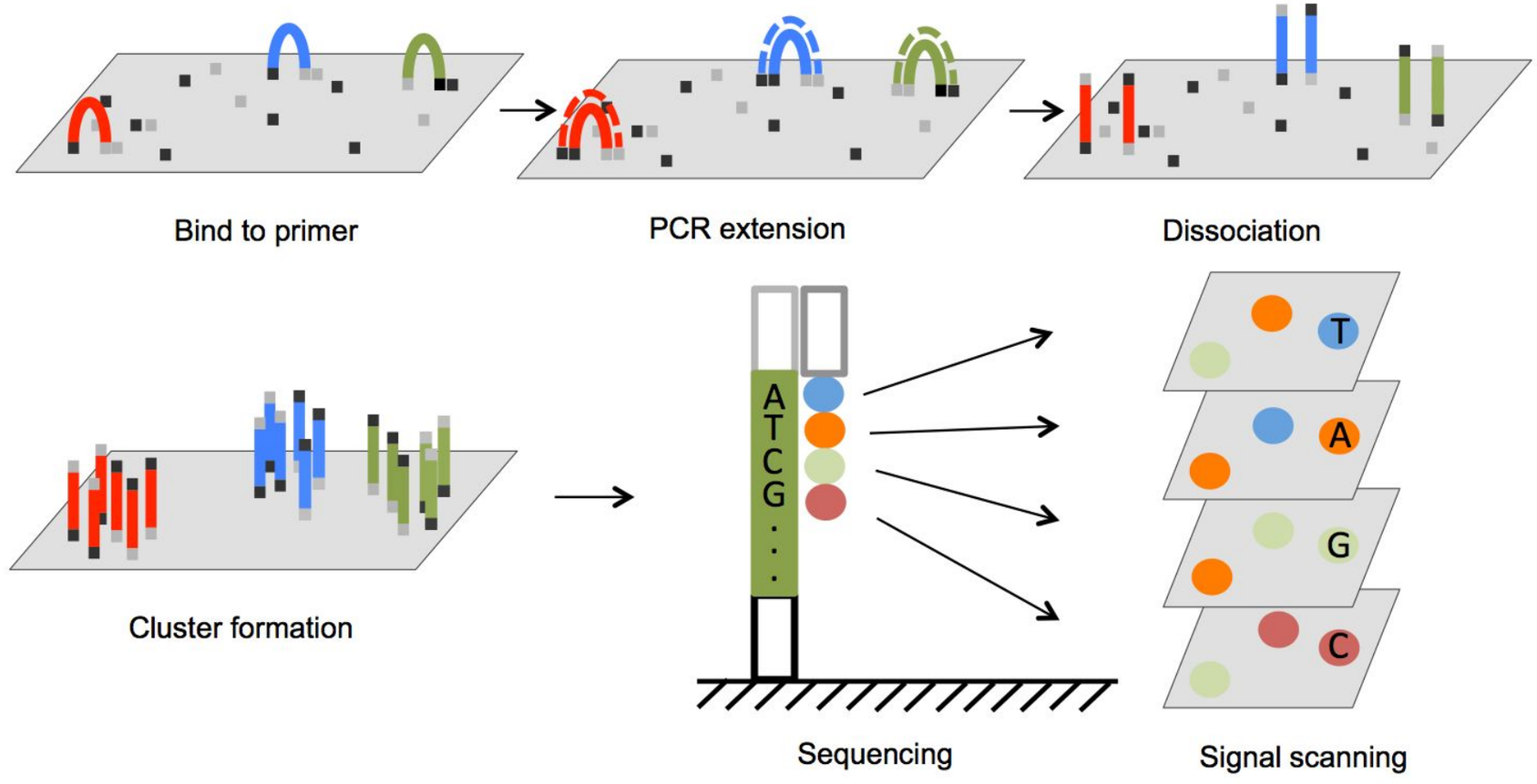
500–1,000 bp fragments

454, Solexa,
Ion Torrent,
Illumina

https://www.pacb.com/blog/the-evolution-of-dna-sequencing-tools/

Sequencing by Synthesis





https://www.intechopen.com/chapters/49419

Sequencing Technology

Sanger sequencing
Maxam and Gilbert
Sanger chain termination

Infer nucleotide identity using dNTPs,
then visualize with electrophoresis

500–1,000 bp fragments

454, Solexa,
Ion Torrent,
Illumina

High throughput from the
parallelization of sequencing reactions

~50–500 bp fragments

https://www.pacb.com/blog/the-evolution-of-dna-sequencing-tools/

Sequencing Technology

First generation

Second generation
(next generation sequencing)

Third generation








Sanger sequencing
Maxam and Gilbert
Sanger chain termination

Infer nucleotide identity using dNTPs,
then visualize with electrophoresis

500–1,000 bp fragments

454, Solexa,
Ion Torrent,
Illumina

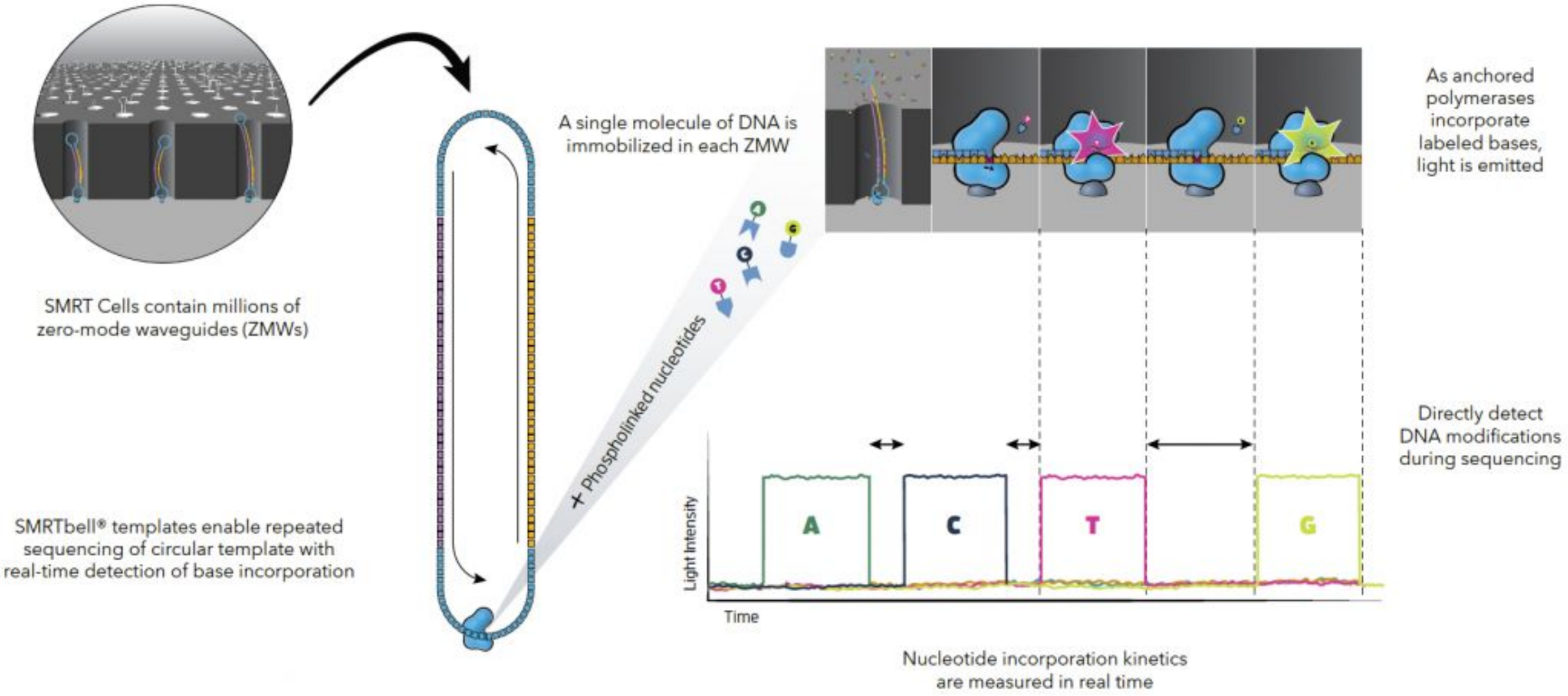
High throughput from the
parallelization of sequencing reactions

~50–500 bp fragments

PacBio
Oxford Nanopore

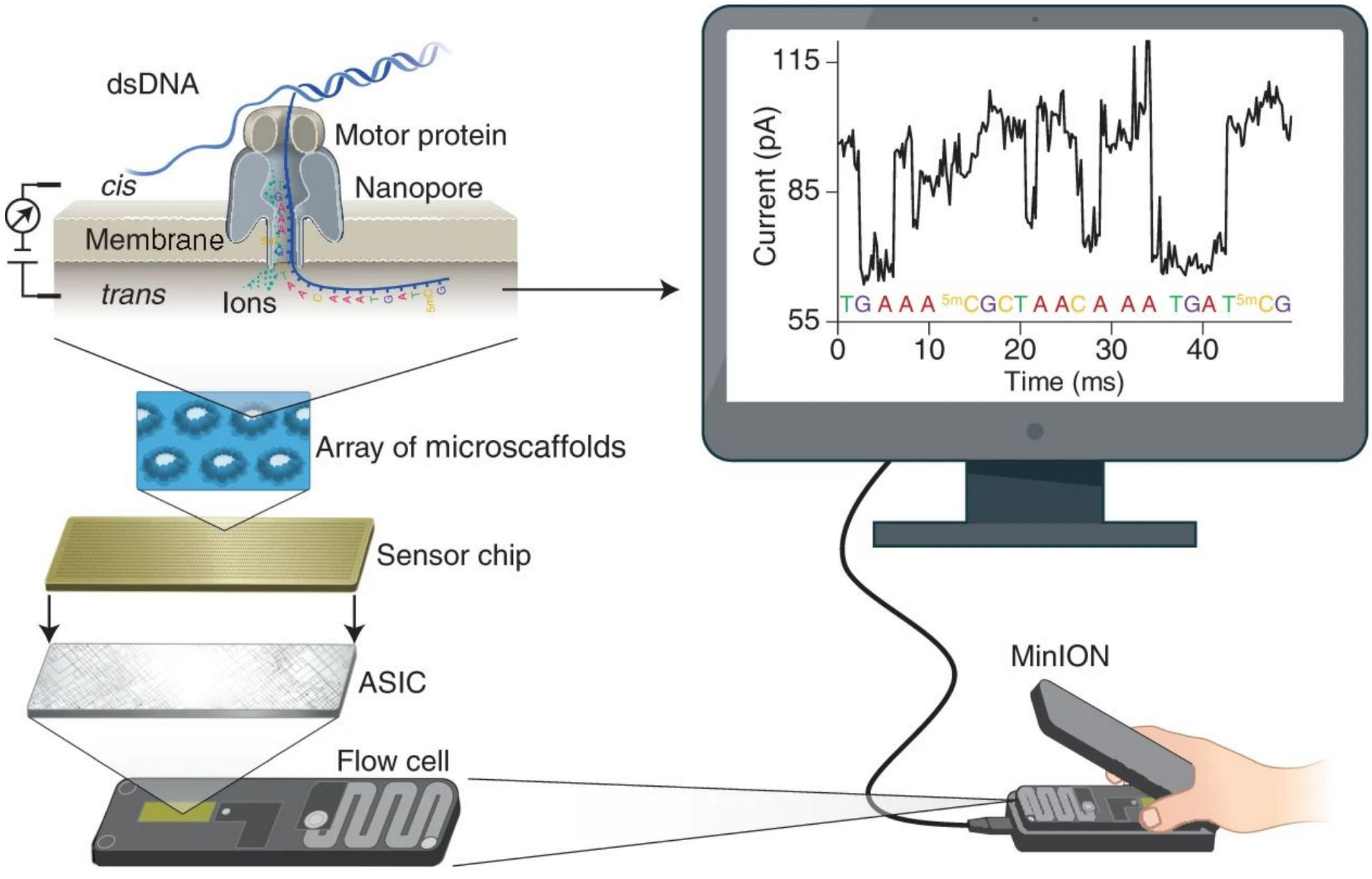
https://www.pacb.com/blog/the-evolution-of-dna-sequencing-tools/

PacBio Sequencing



https://www.pacb.com/wp-content/uploads/SMRT-Sequencing-Brochure-Delivering-highly-accurate-long-reads-to-drive-discovery-in-life-science.pdf

Nanopore Sequencing



https://www.nature.com/articles/s41587-021-01108-x/figures/1

Sequencing Technology

Sanger sequencing
Maxam and Gilbert
Sanger chain termination

Infer nucleotide identity using dNTPs,
then visualize with electrophoresis

500–1,000 bp fragments

454, Solexa,
Ion Torrent,
Illumina

High throughput from the
parallelization of sequencing reactions

~50–500 bp fragments

PacBio
Oxford Nanopore

Sequence native DNA in real time
with single-molecule resolution

Tens of kb fragments, on average

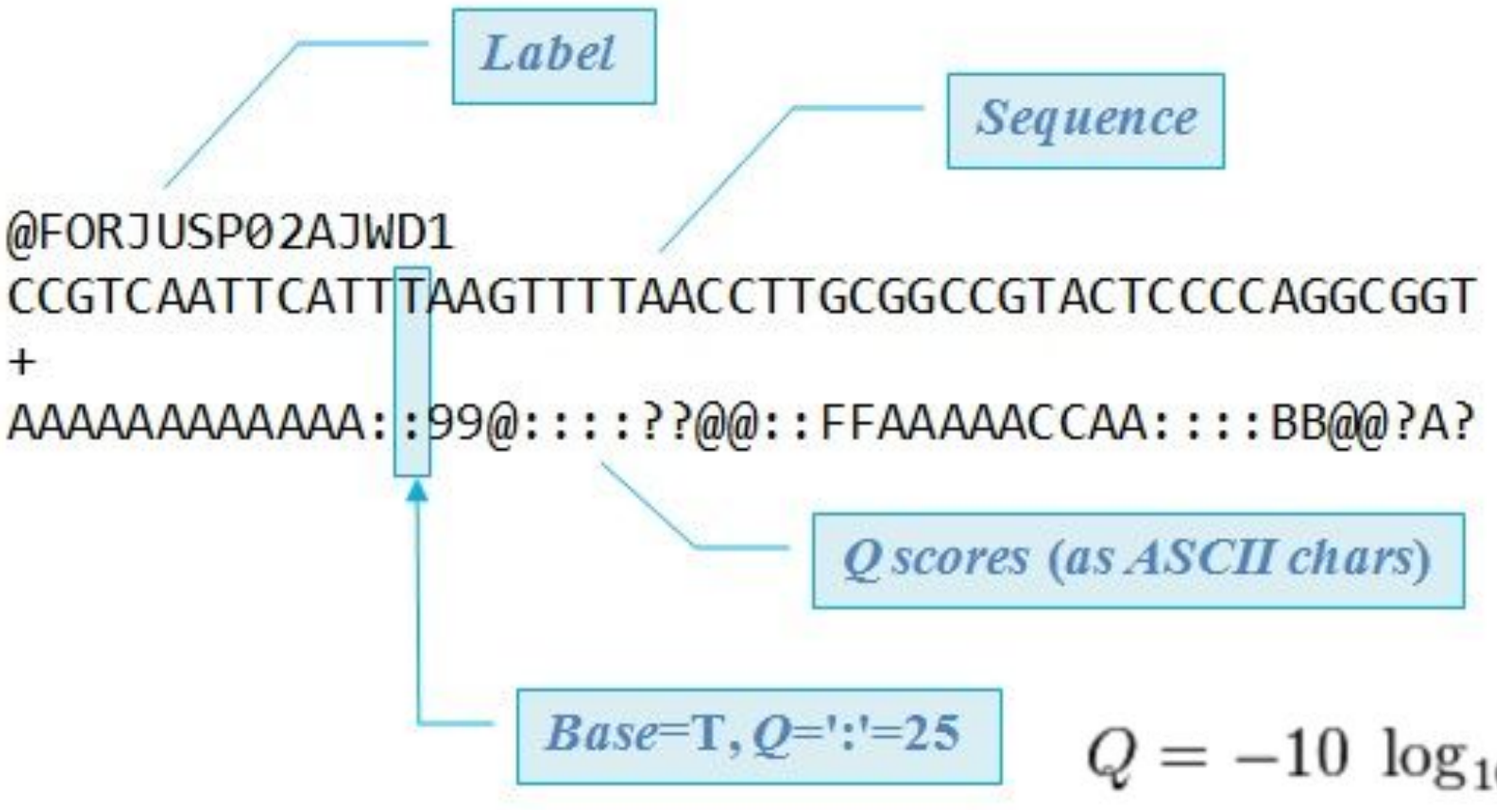
https://www.pacb.com/blog/the-evolution-of-dna-sequencing-tools/

Sequencing Technology

First generation	Second generation (next generation sequencing)	Third generation
Sanger sequencing Maxam and Gilbert Sanger chain termination	454, Solexa, Ion Torrent, Illumina	PacBio Oxford Nanopore
Infer nucleotide identity using dNTPs, then visualize with electrophoresis	High throughput from the parallelization of sequencing reactions	Sequence native DNA in real time with single-molecule resolution
500–1,000 bp fragments	~50–500 bp fragments	Tens of kb fragments, on average
Short-read sequencing		Long-read sequencing

https://www.pacb.com/blog/the-evolution-of-dna-sequencing-tools/

Capturing measurement error: FASTQ



$$Q = -10 \log_{10} P \quad \Longrightarrow \quad P = 10^{\frac{-Q}{10}}$$

Quality value Q is an integer representation of the probability p that a corresponding base call is incorrect

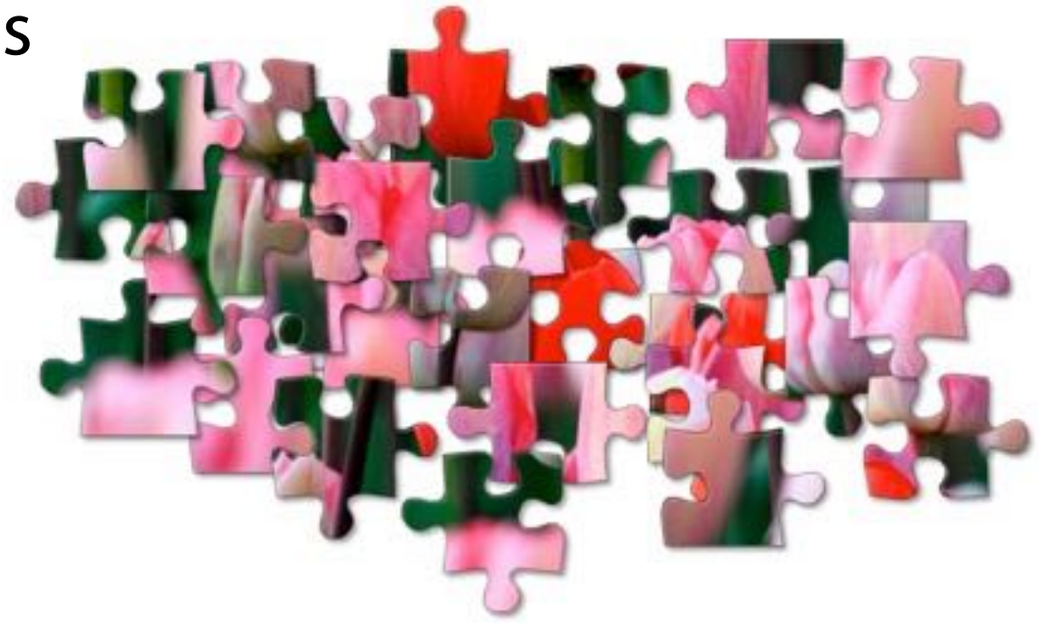
Phred Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90%
20	1 in 100	99%
30	1 in 1000	99.9%
40	1 in 10000	99.99%
50	1 in 100000	99.999%

https://www.drive5.com/usearch/manual/fastq_files.html

https://learn.gencore.bio.nyu.edu/ngs-file-formats/quality-scores/

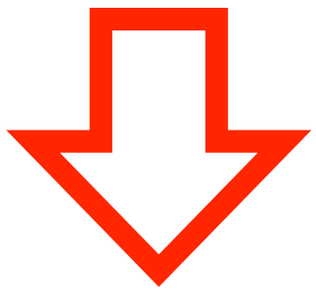
Assembly

Reads



+

Reference genome

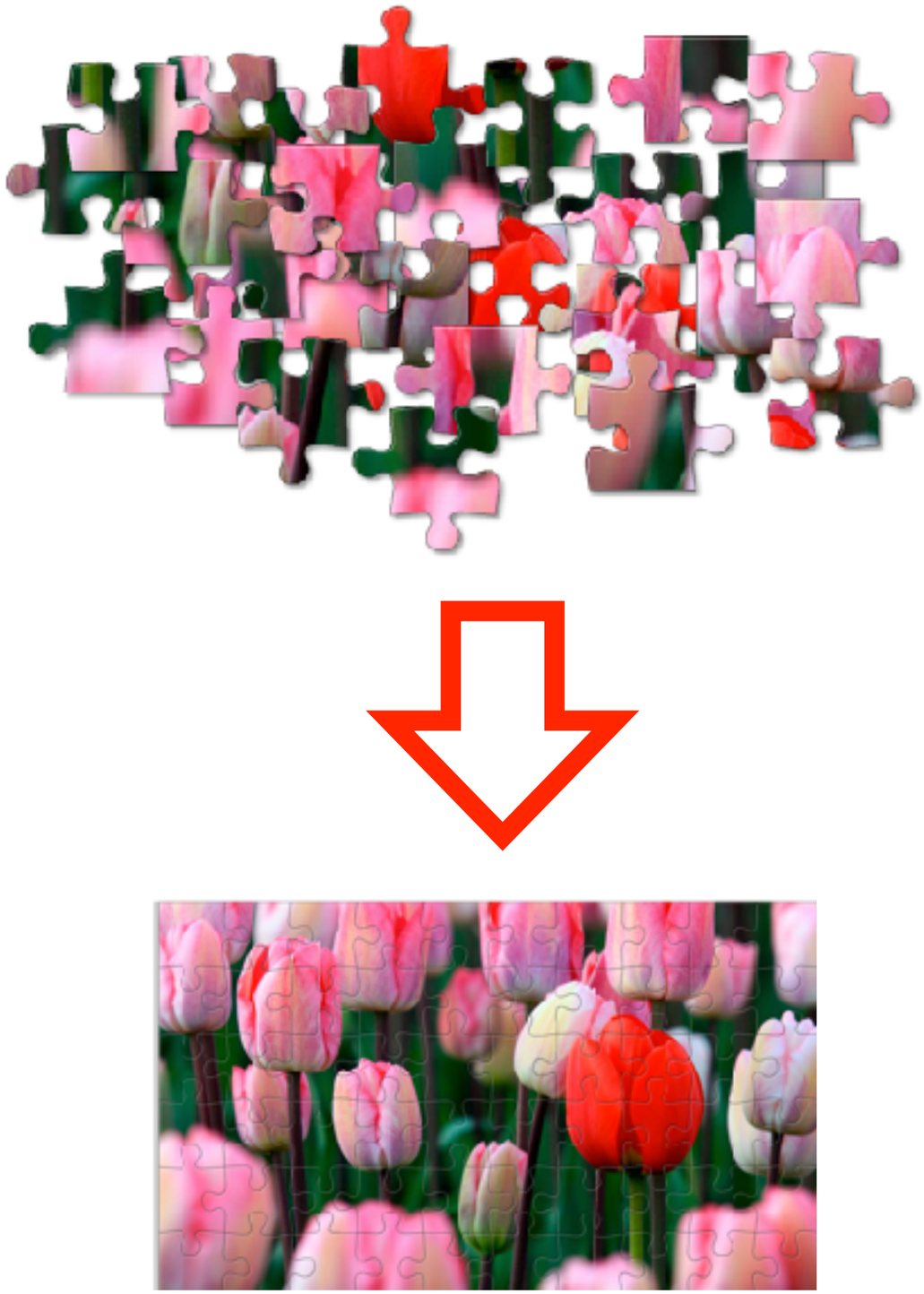


Input DNA

How do we assemble puzzle without the benefit of knowing what the finished product should look like?

(That's what the Human Genome Project had to do!)

De novo shotgun assembly



Assembly

Whole-genome "shotgun" sequencing first copies the input DNA:

Input: GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

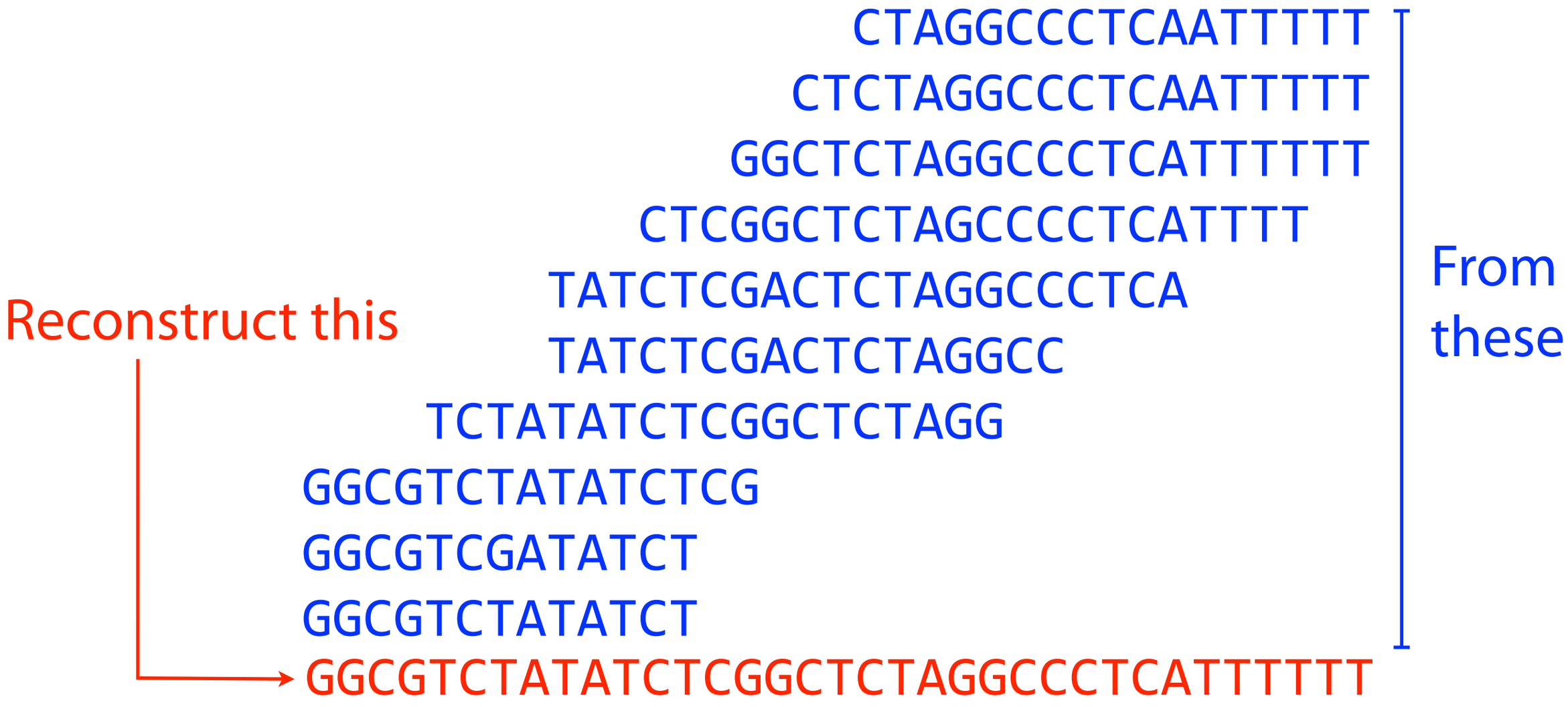
Copy: GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Then fragments it:

Fragment: GGCGTCTA TATCTCGG CTCTAGGCCCTC ATTTTTT
GGC GTCTATAT CTCGGCTCTAGGCCCTCA TTTTTT
GGCGTC TATATCT CGGCTCTAGGCCCT CATTTTTT
GGCGTCTAT ATCTCGGCTCTAG GCCCTCA TTTTTT

"Shotgun" refers to the random fragmentation of the whole genome; like it was fired from a shotgun

Assembly


Reconstruct this
From these
CTAGGCCCTCAATTTTT
CTCTAGGCCCTCAATTTTT
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
TATCTCGACTCTAGGCC
TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Assembly

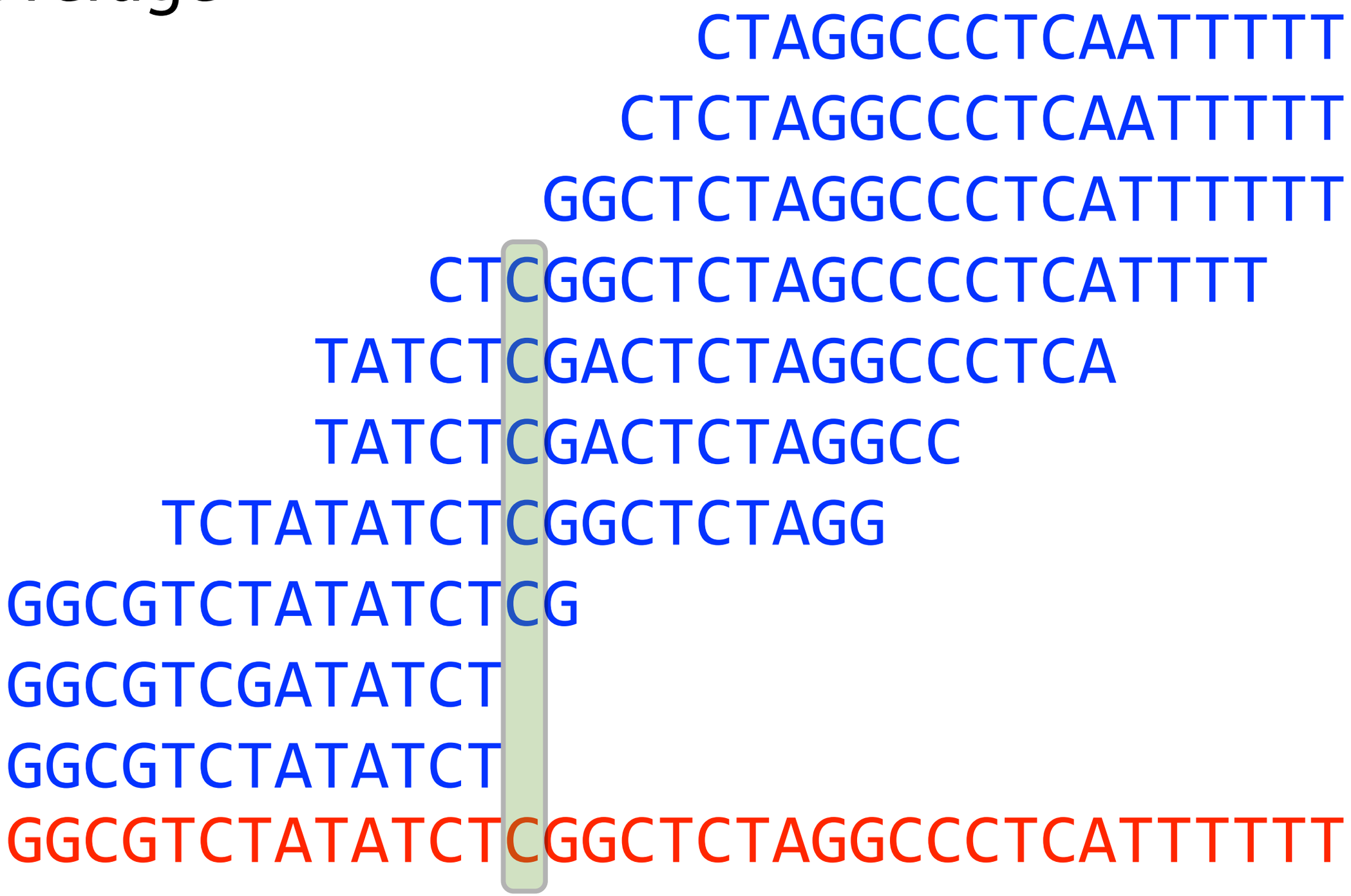
Reconstruct this

From these

CTAGGCCCTCAATTTTT
GGCGTCTATATCT
CTCTAGGCCCTCAATTTTT
TCTATATCTCGGCTCTAGG
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
GGCGTCGATATCT
TATCTCGACTCTAGGCC
GGCGTCTATATCTCG

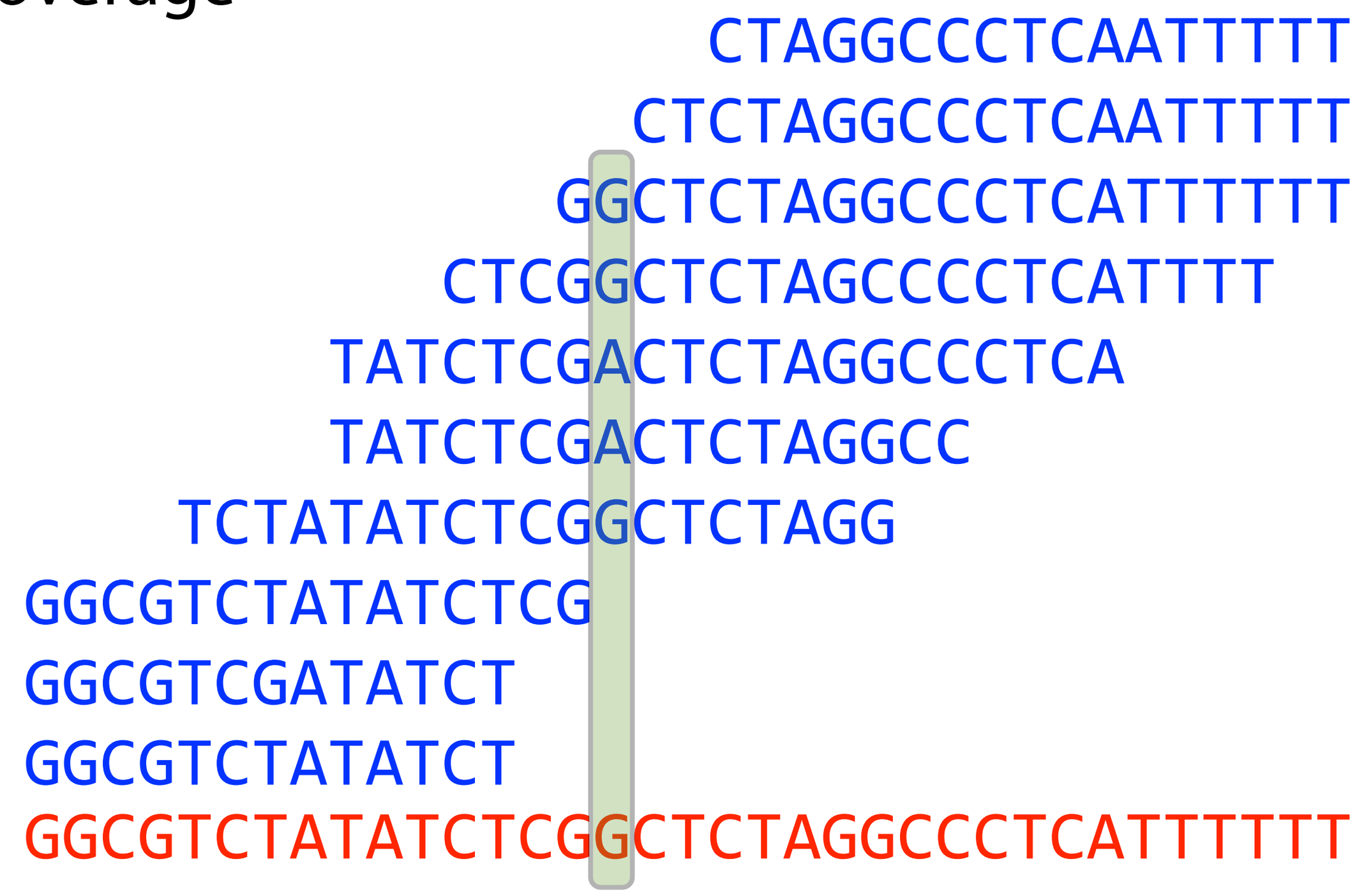
??

Coverage



Coverage = 5

Coverage



Coverage = 5

CTAGGCCCTCAATTTTT
CTCTAGGCCCTCAATTTTT
GGCTCTAGGCCCTCATTTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
TATCTCGACTCTAGGCC
TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

177 bases

35 bases

Average coverage = 177 / 35 ≈ 5-fold

TCTATATCTCGGCTCTAGG

TATCTCGACTCTAGGCC

TCTATATCTCGGCTCTAGG

TATCTCGACTCTAGGCC

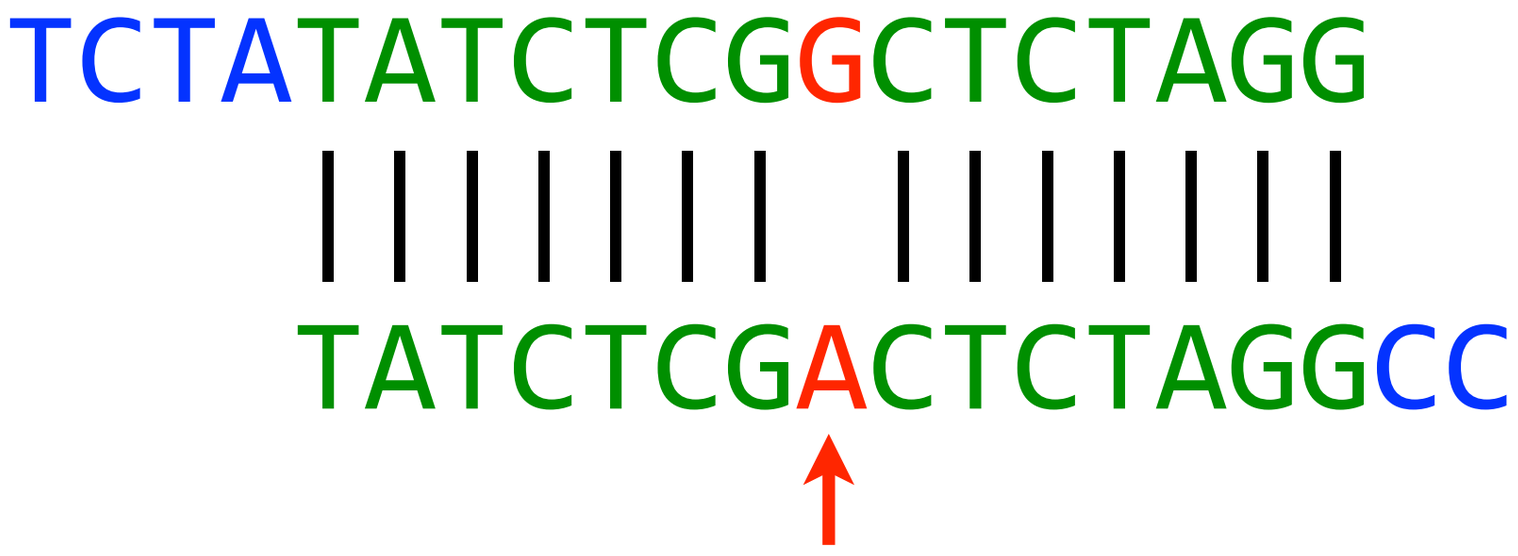
First law of assembly

If a suffix of read A is similar to a prefix of read B...

```
TCTATATCTCGGCTCTAGG
       ||||||| ||||||||
     TATCTCGACTCTAGGCC
```

...then A and B might *overlap* in the genome

```
          TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTTT
          TATCTCGACTCTAGGCC
```

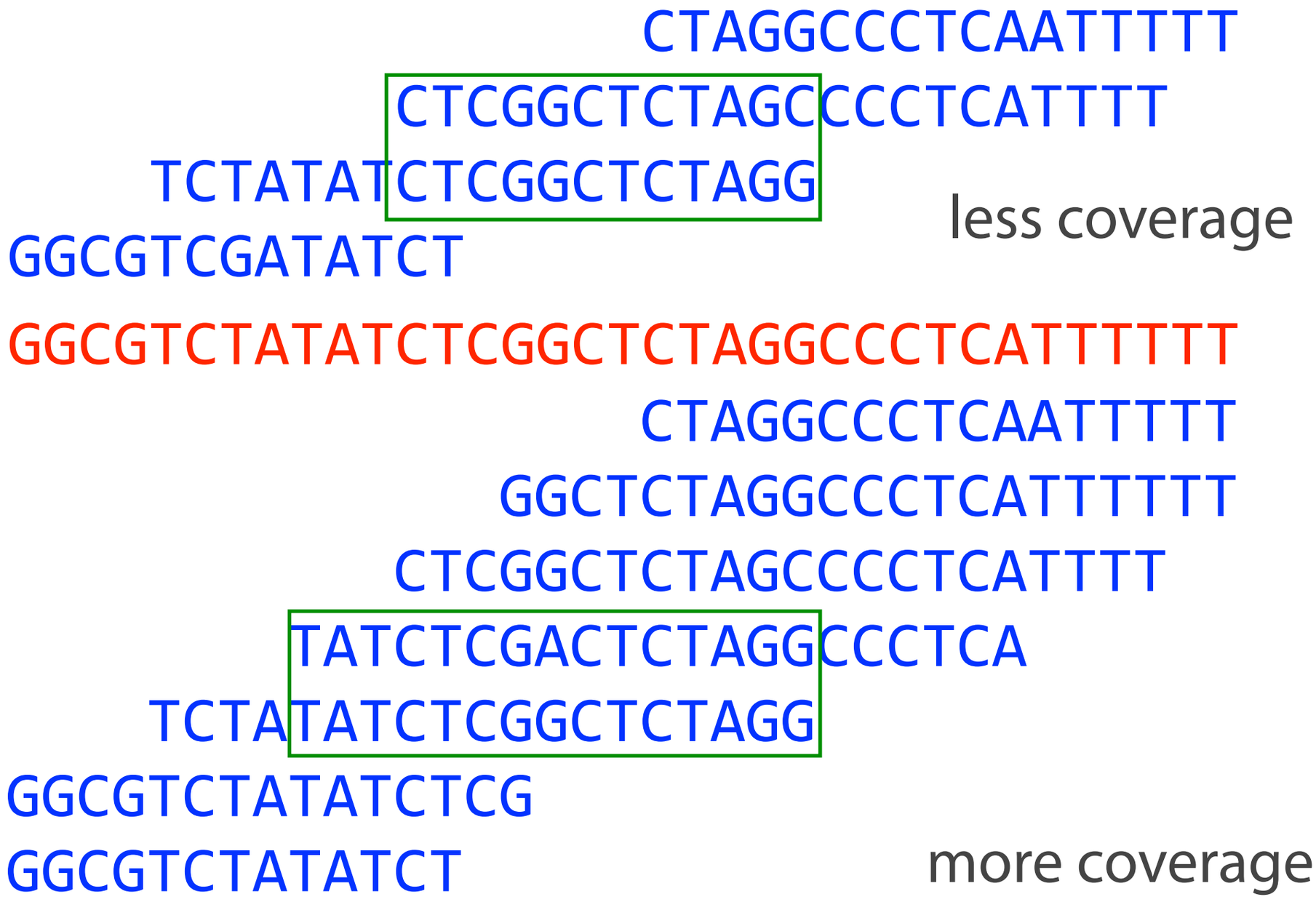


Why the differences?

1. Sequencing errors
2. Ploidy: e.g. humans have 2 copies of each chromosome, and copies can differ

Second law of assembly

More coverage leads to more and longer overlaps

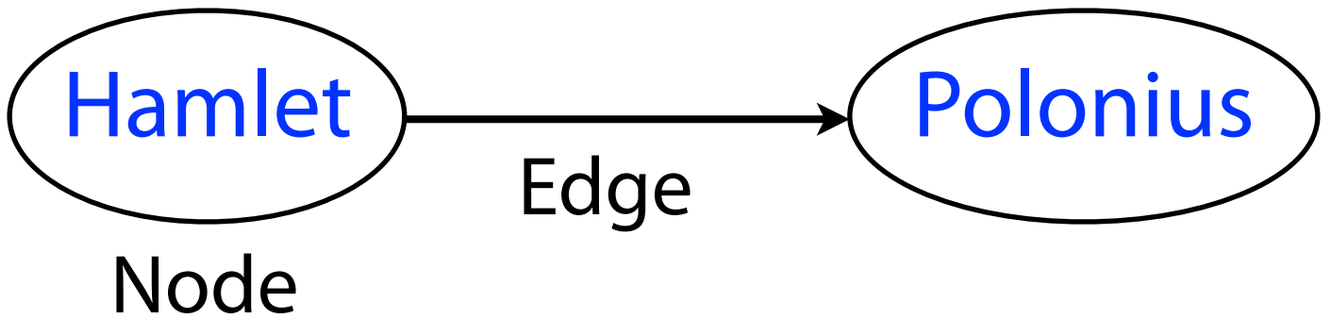


```
TCTATATCTCGGCTCTAGG
      ||||||| ||||||||
      TATCTCGACTCTAGGCC
```

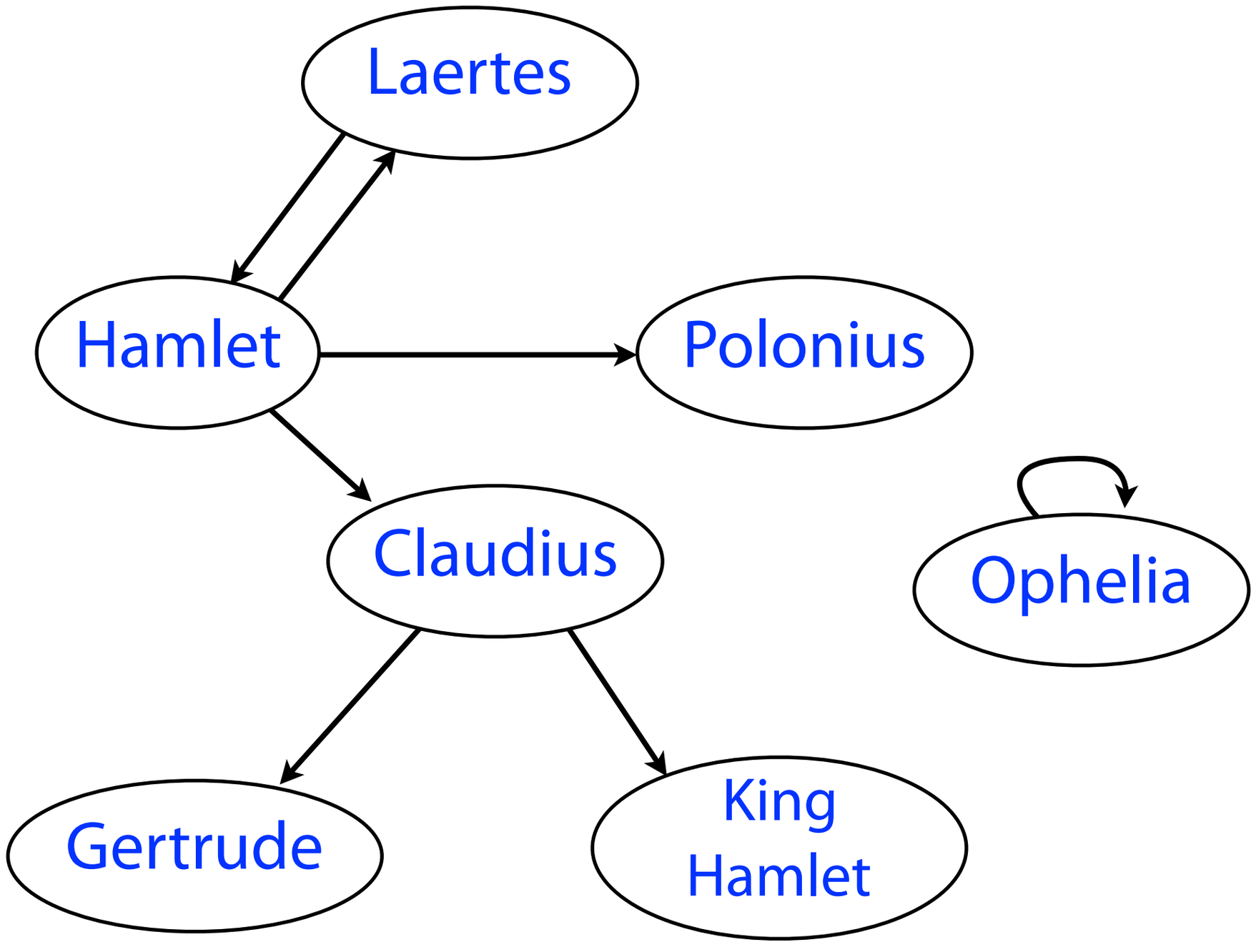
```
TCTATATCTCGGCTCTAGG
      ||||||| |||||||
       TATCTCGACTCTAGGCC
```

```
TATCTCGACTCTAGGCC
    |||| |||||| ||
     CTCGGCTCTAGCCCCTCAT
```

Directed graph



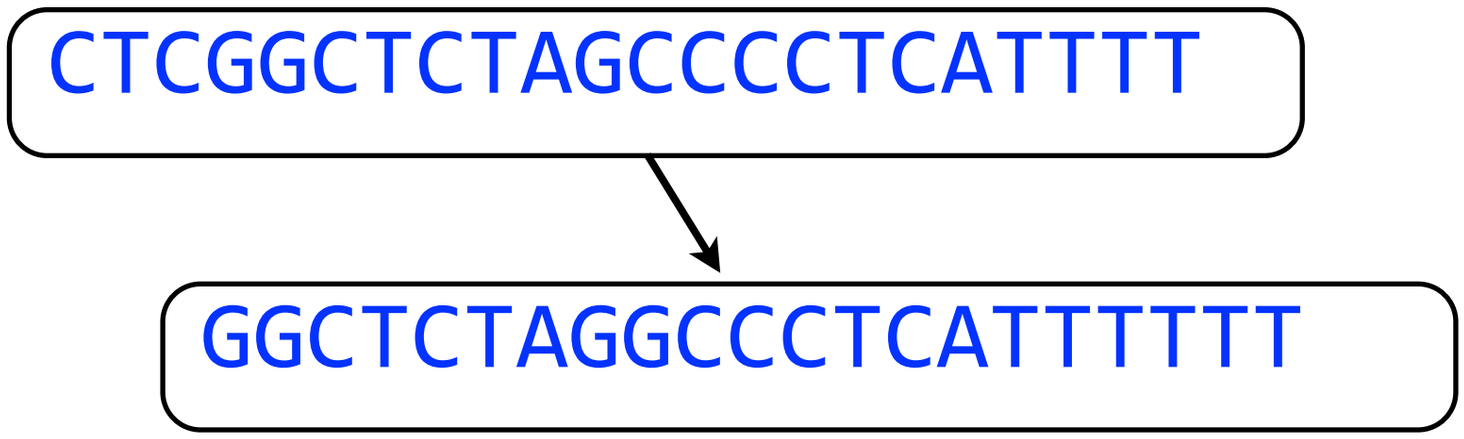
Directed graph



Overlap graph

Each node is a read

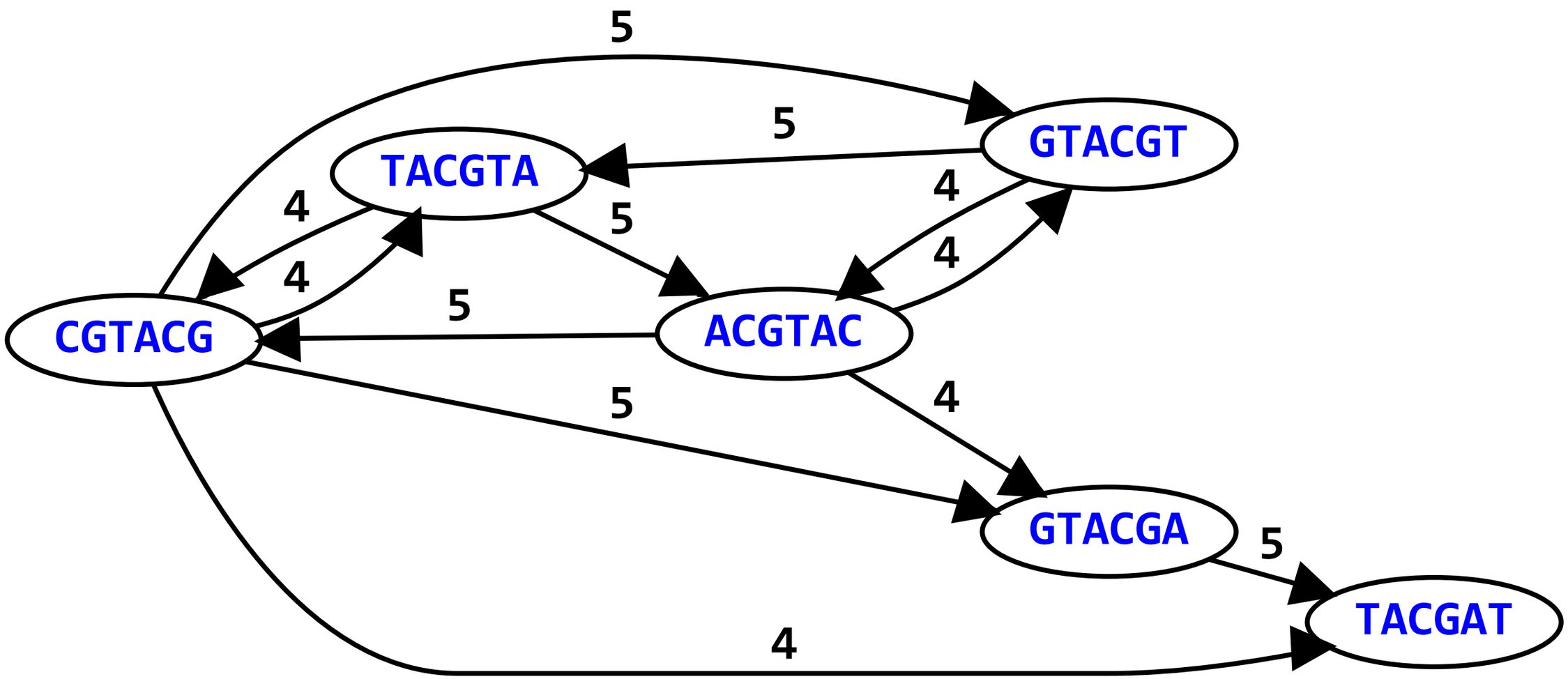
Draw edge A -> B when suffix of A overlaps prefix of B



Overlap graph

Nodes: all 6-mers from GTACGTACGAT

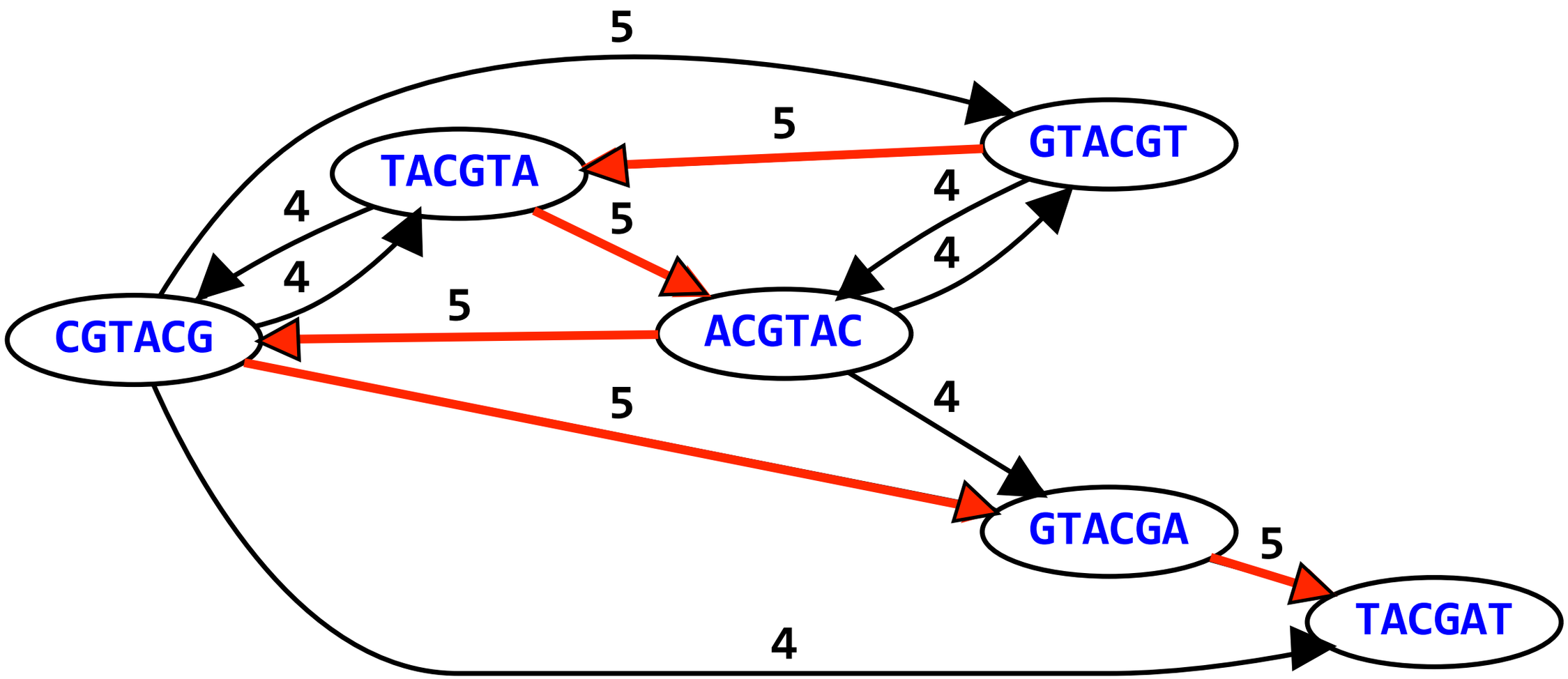
Edges: overlaps of length ≥4



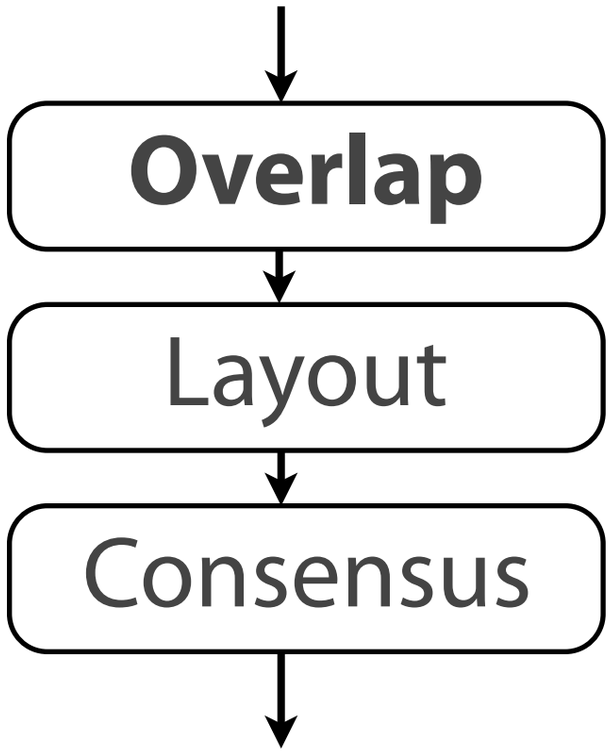
Overlap graph

Nodes: all 6-mers from GTACGTACGAT

Edges: overlaps of length ≥4



Overlap Layout Consensus



Build overlap graph

Coalesce paths into *contigs*

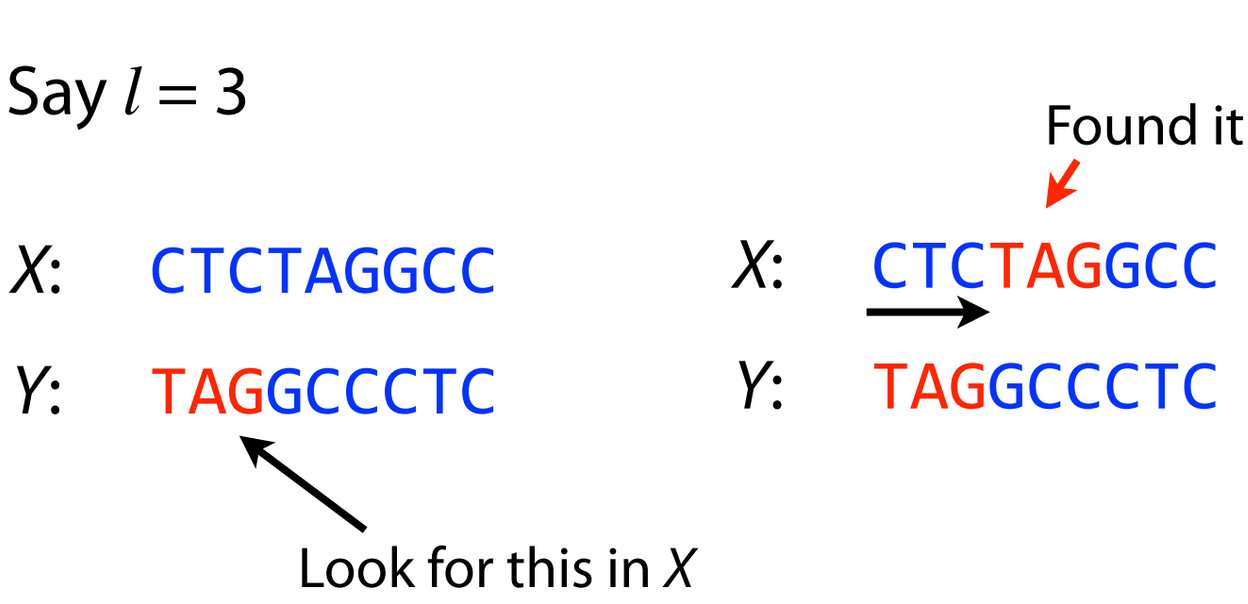
Pick nucleotide sequence for each contig

Finding overlaps

Overlap: Suffix of *X* of length $\geq l$ matches prefix of *Y; l* is given

Naive: look in *X* for occurrences of *Y*'s length-l prefix. Extend matches to the right to confirm whether entire suffix of *X* matches.

Say $l = 3$



See `suffixPrefixMatch` function in HW5 Q4 (Assembly Challenge)

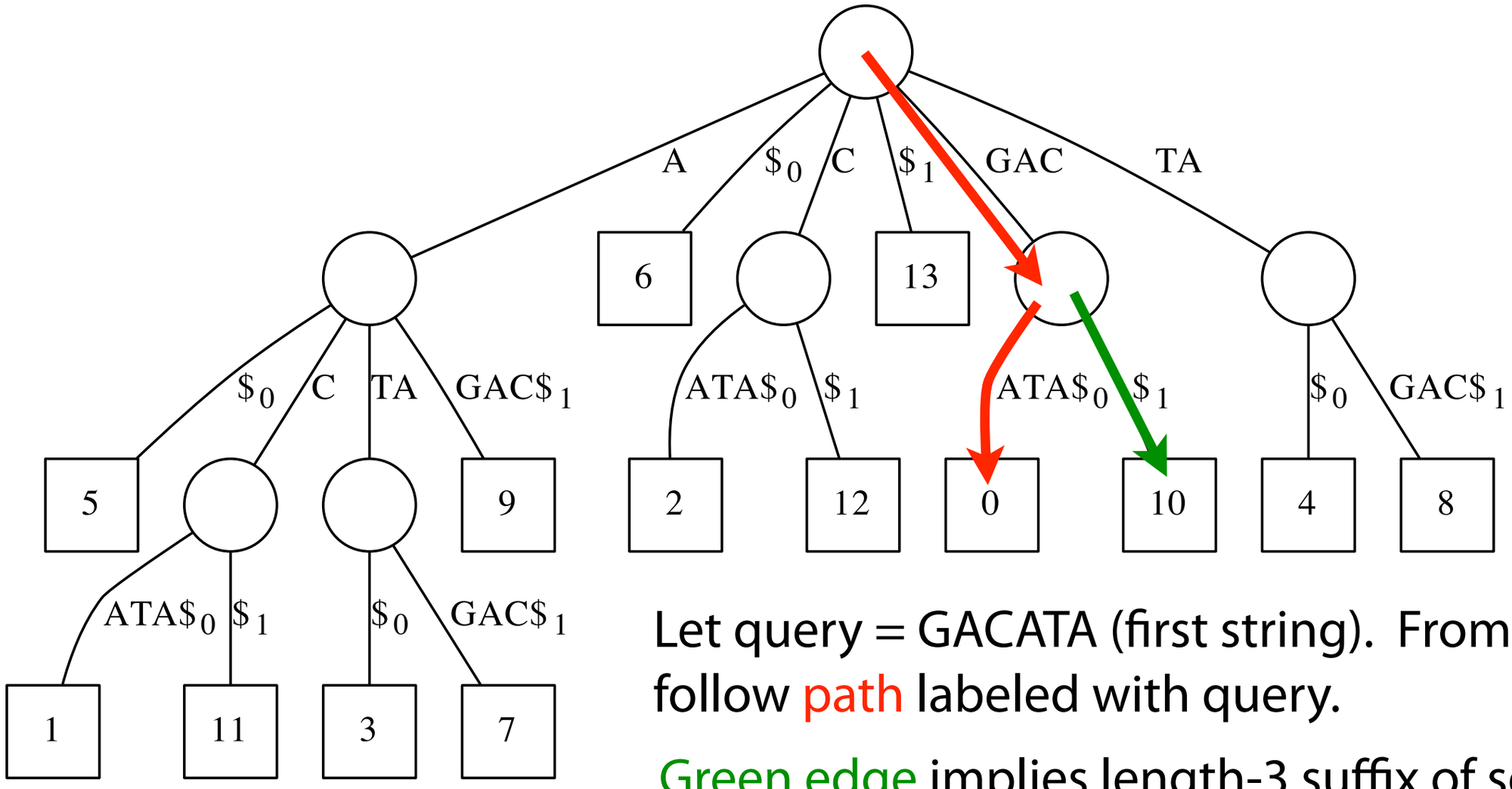
Finding overlaps

With suffix tree?

Given a collection of strings *S*, for each string *x* in *S* find all overlaps involving a prefix of *x* and a suffix of another string *y*

Finding overlaps with suffix tree

Generalized suffix tree for {"GACATA", "ATAGAC"} GACATA$_0$ATAGAC$_1$



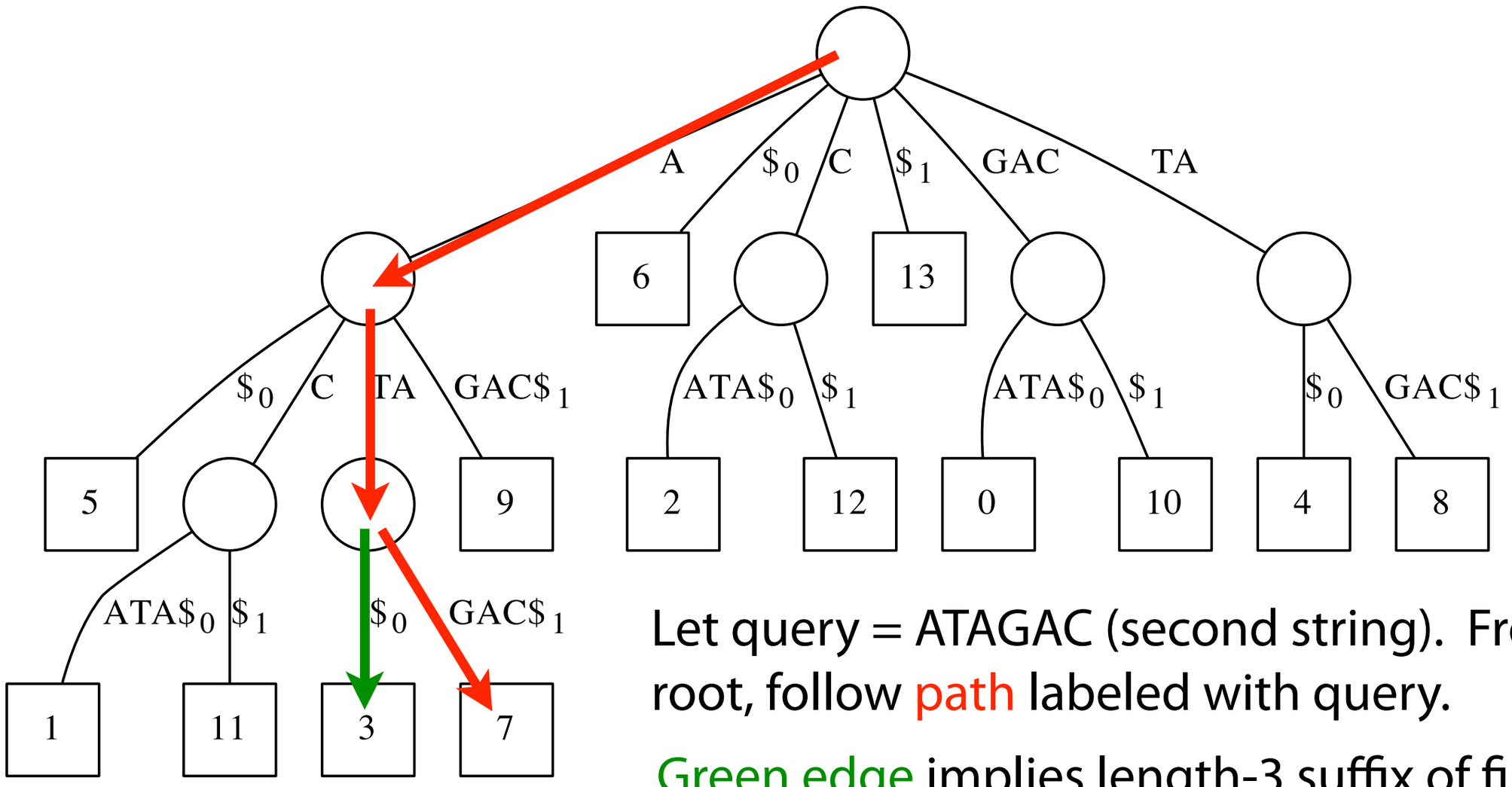
Let query = GACATA (first string). From root, follow path labeled with query.

Green edge implies length-3 suffix of second string equals length-3 prefix of query

ATAGAC
|||
GACATA

Finding overlaps with suffix tree

Generalized suffix tree for {"GACATA", "ATAGAC"} GACATA$_0$ATAGAC$_1$



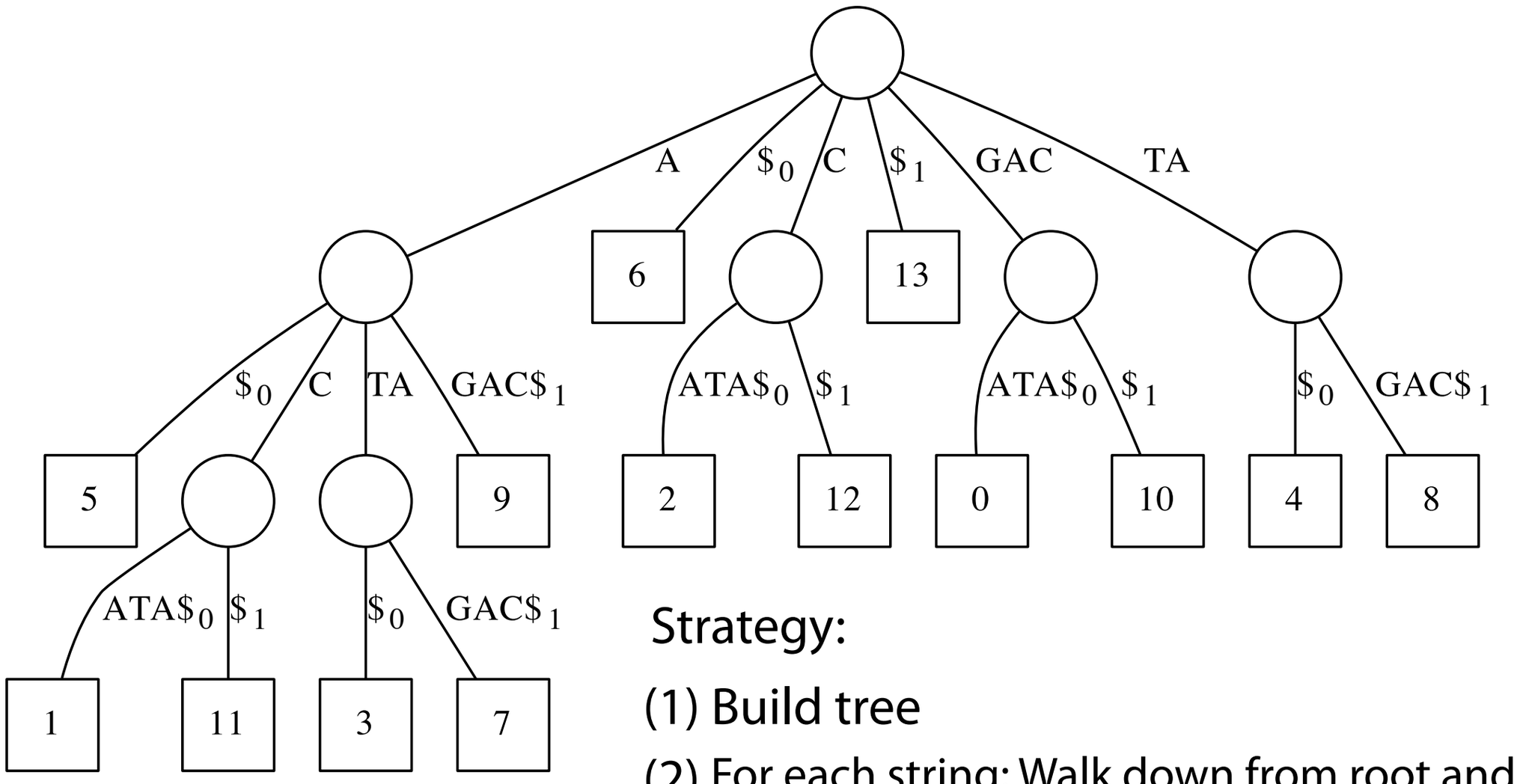
GACATA
|||
ATAGAC

Let query = ATAGAC (second string). From root, follow path labeled with query.

Green edge implies length-3 suffix of first string equals length-3 prefix of query

Finding overlaps with suffix tree

Generalized suffix tree for {"GACATA", "ATAGAC"} GACATA$_0$ATAGAC$_1$

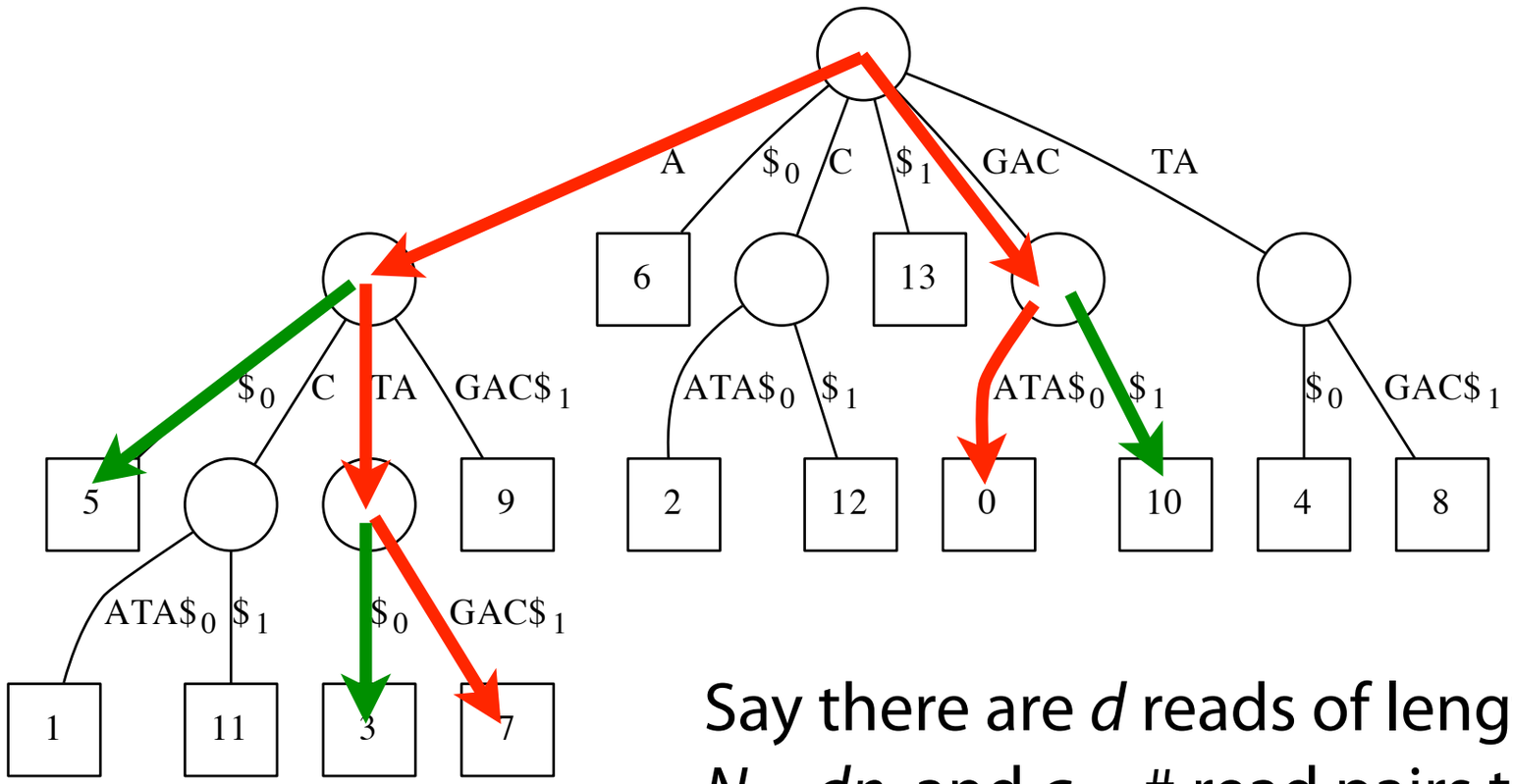


Strategy:

(1) Build tree

(2) For each string: Walk down from root and report any outgoing edge labeled with a separator. Each corresponds to a prefix/suffix match involving prefix of query string and suffix of string ending in the separator.

Finding overlaps with suffix tree



Say there are *d* reads of length *n*, total length $N = dn$, and a = # read pairs that overlap

Assume for given string pair we report only the longest suffix/prefix match

Time to build generalized suffix tree:	O(*N*)
... to walk down red paths:	O(*N*)
... to find & report overlaps (green):	O(*a*)
Overall:	O(*N* + *a*)

Finding overlaps

What about *approximate* suffix/prefix matches?

```
X: CTCGGCCCTAGG
       ||| |||||
Y:     GGCTCTAGGCCC
```

Dynamic programming

Finding overlaps with dynamic programming

```
X: CTCGGCCCTAGG
       |||  |||||
Y:     GGCTCTAGGCCC
```

Use *global alignment* recurrence and score function

$$D[i,j] = \min \begin{cases} D[i-1,j] + s(x[i-1], -) \\ D[i,j-1] + s(-, y[j-1]) \\ D[i-1,j-1] + s(x[i-1], y[j-1]) \end{cases}$$

$s(a,b)$

	A	C	G	T	-
A	0	4	2	4	8
C	4	0	4	2	8
G	2	4	0	4	8
T	4	2	4	0	8
-	8	8	8	8	

How do we force it to find prefix / suffix matches?

Finding overlaps with dynamic programming

$s(a, b)$

	A	C	G	T	-
A	0	4	2	4	8
C	4	0	4	2	8
G	2	4	0	4	8
T	4	2	4	0	8
-	8	8	8	8	

How to initialize first row & column so suffix of *X* aligns to prefix of *Y*?

First column gets 0s
(any suffix of *X* is possible)

First row gets ∞s
(must be a prefix of *Y*)

Backtrace from last row

Y

X		-	G	G	C	T	C	T	A	G	G	C	C	C
	-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
	C	0	4	12	20									
	T	0	4	8	14									
	C	0	4	8	8									
	G	0	0	4	12									
	G	0	0	0	8	16	16	24	26	30	36	44	52	60
	C	0	4	4	0	8	16	18	26	30	34	36	44	52
	C	0	4	8	4	0	8	16	22	30	34	34	36	44
	C	0	4	8	8	6	0	10	18	26	34	34	34	36
	T	0	4	8	10	8	8	0	10	18	26	34	36	36
	A	0	2	6	12	14	12	10	0	10	18	26	34	40
	G	0	0	2	10	16	18	16	10	0	10	18	26	34
	G	0	0	0	6	14	20	22	18	10	0	10	18	26

X: CTCGGCCCTAGG
||| |||||
Y: GGCTCTAGGCCC

Finding overlaps with dynamic programming

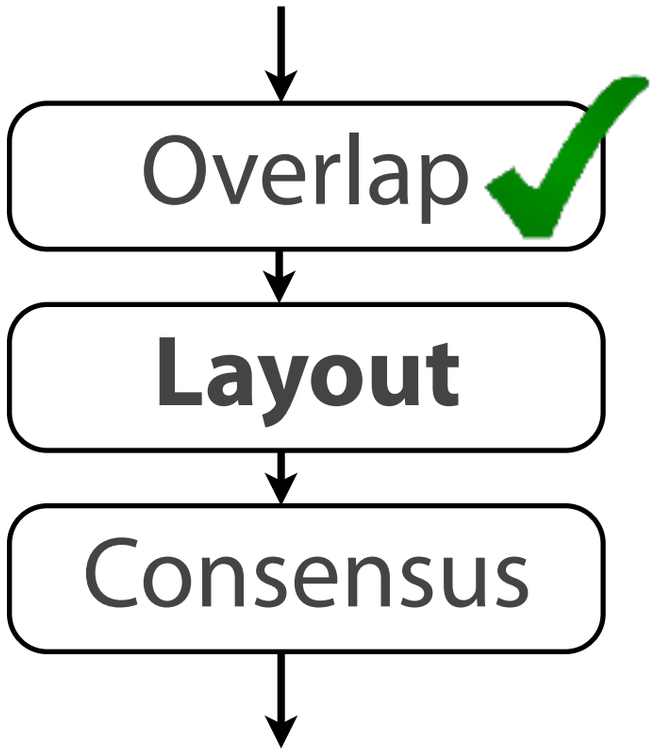
Say there are d reads of length n, total length $N = dn$, and a is total number of pairs with an overlap

# overlaps to try:	$O(d^2)$
Size of each DP matrix:	$O(n^2)$
Overall:	$O(d^2n^2)$, or $O(N^2)$

Contrast $O(N^2)$ with suffix tree: $O(N + a)$, but where a is worst-case $O(d^2)$

Real-world overlappers mix the two; index filters out vast majority of non-overlapping pairs, dynamic programming used for remaining pairs

Overlap Layout Consensus



Build overlap graph

Coalesce paths into *contigs*

Pick nucleotide sequence for each contig

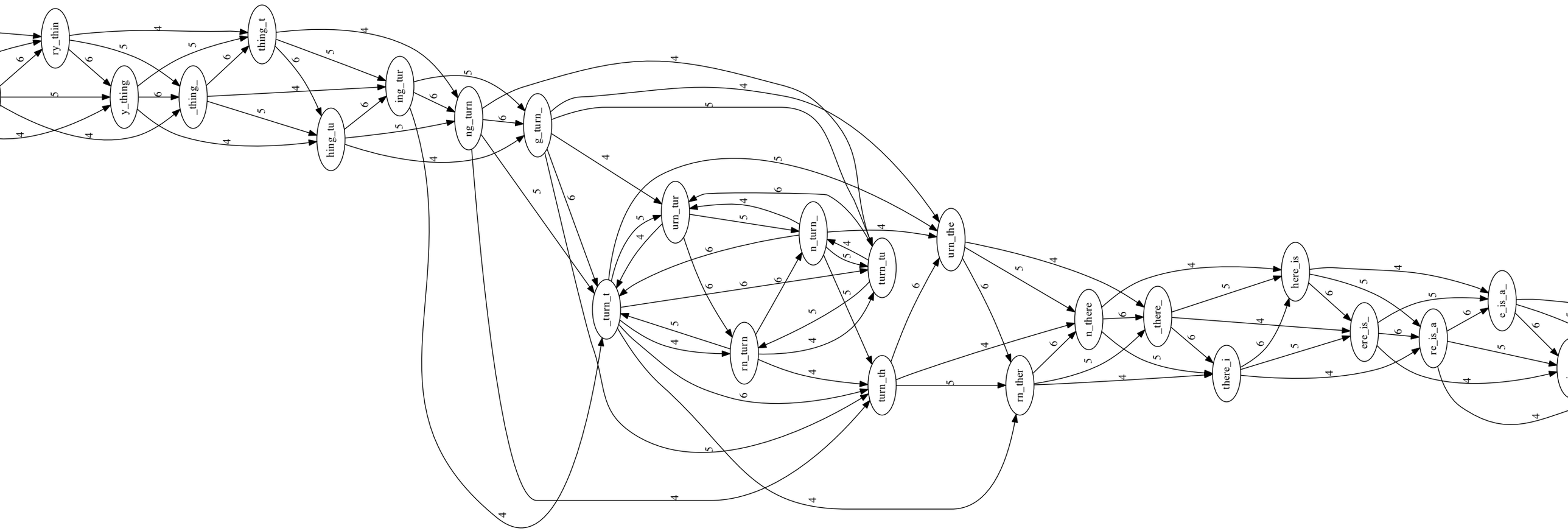
Layout

Overlap graph is big and messy. Contigs don't "pop out" at us.

Below: part of the overlap graph for

`to_every_thing_turn_turn_turn_there_is_a_season`

$l = 4, k = 7$

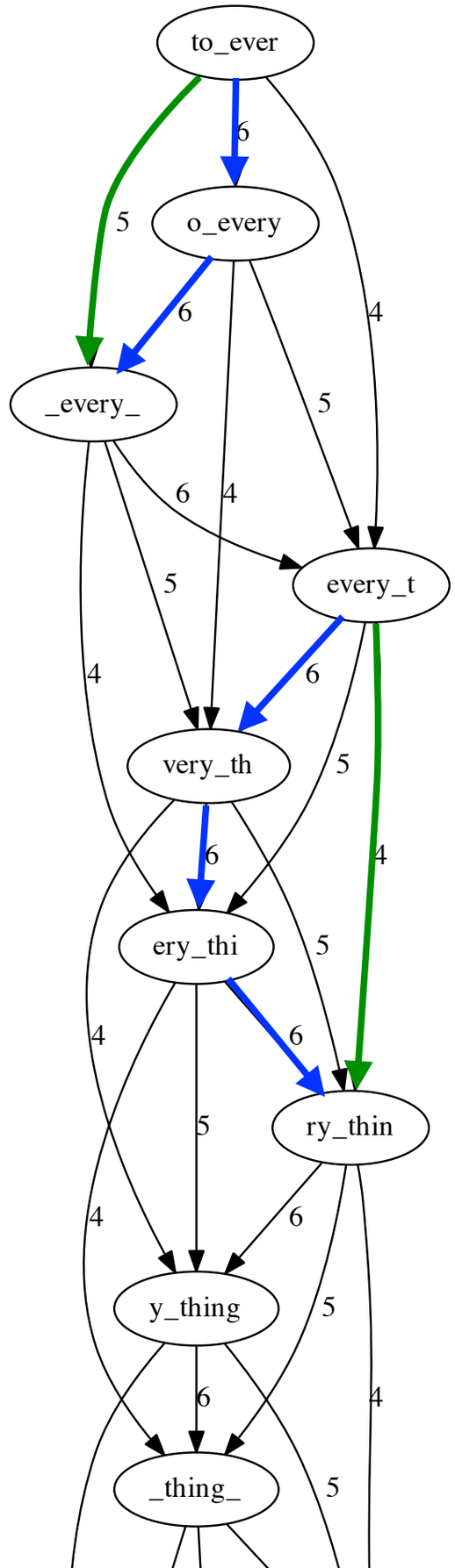


Layout

Anything redundant about this part of the overlap graph?

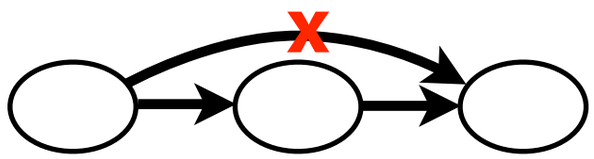
Some edges can be *inferred* (*transitively*) from other edges

E.g. green edge can be inferred from blue

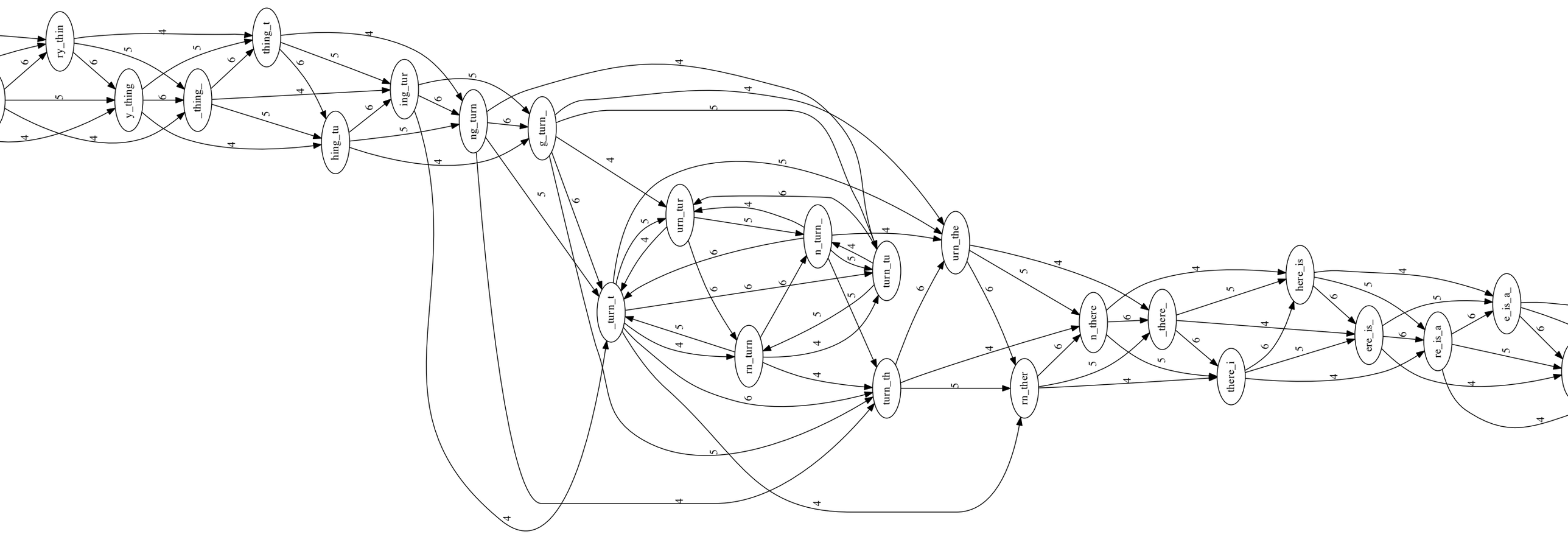


Layout

Remove transitively inferrable edges, starting with edges that skip one node:

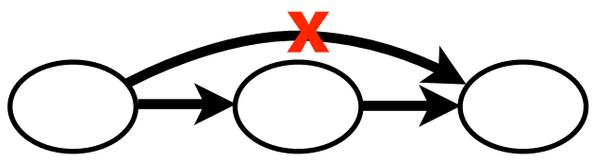


Before:

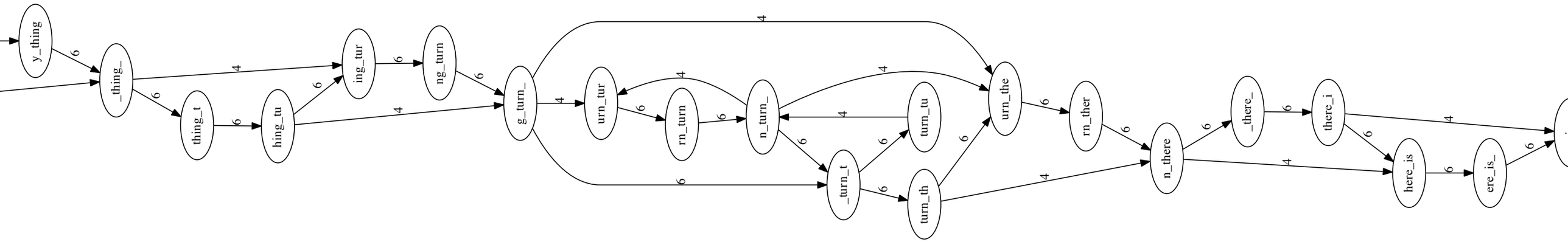


Layout

Remove transitively inferrable edges, starting with edges that skip one node:

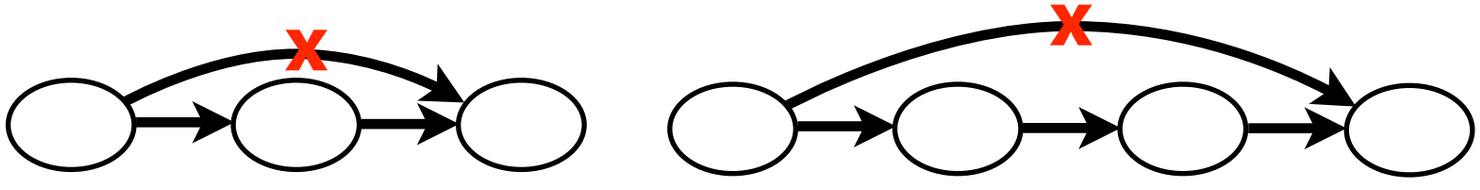


After:

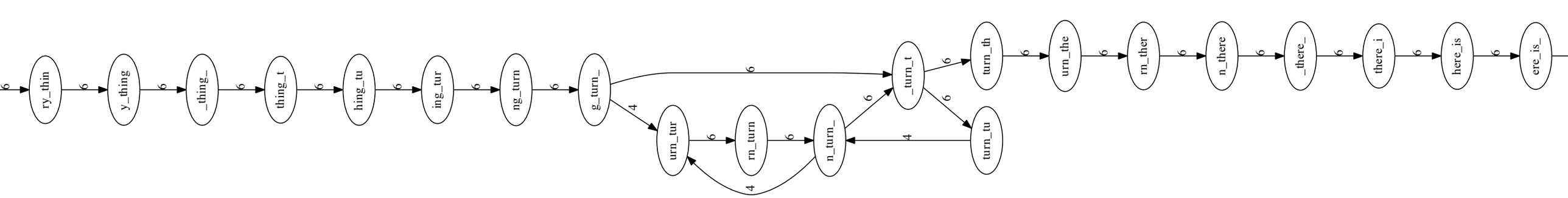


Layout

Now remove edges that skip one or two nodes:



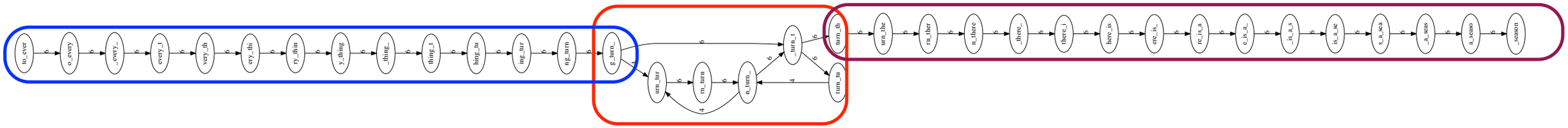
After:



Even simpler

Layout

Emit *contigs* corresponding to the non-branching stretches



Contig 1

`to_every_thing_turn_`

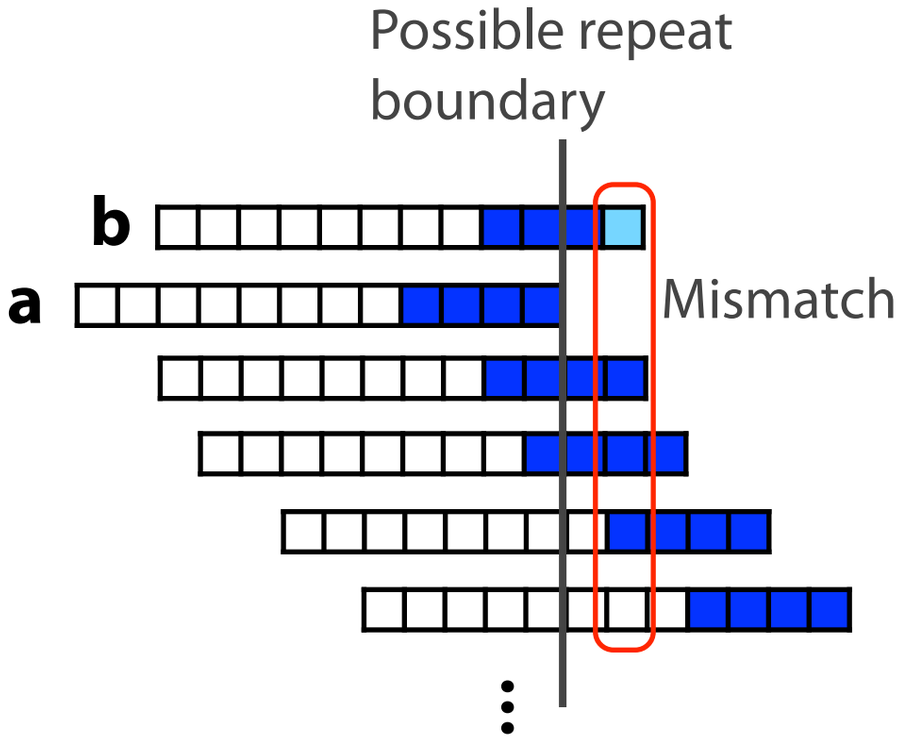
Contig 2

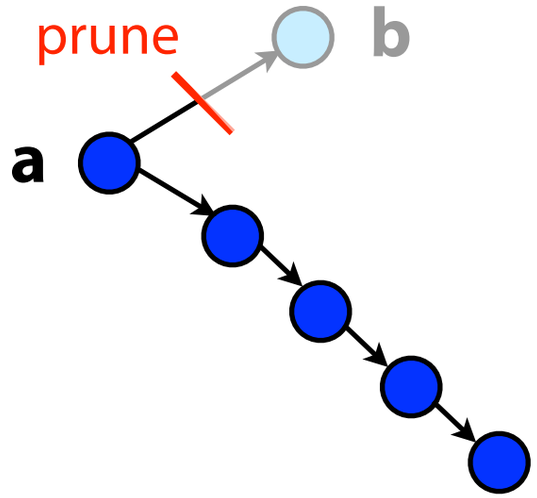
`turn_there_is_a_season`

Unresolvable repeat

Layout

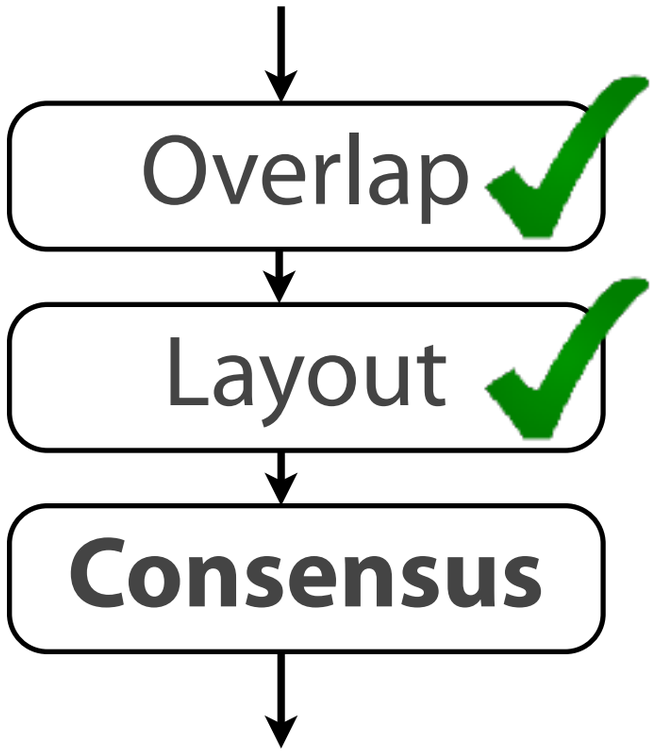
Must handle subgraphs that are spurious, e.g. because of sequencing error





Mismatch could be due to sequencing error or repeat. Since the path through **b** ends abruptly we might conclude it's an error and prune **b**.

Overlap Layout Consensus



Build overlap graph

Bundle stretches of the overlap graph into *contigs*

Pick most likely nucleotide sequence for each contig

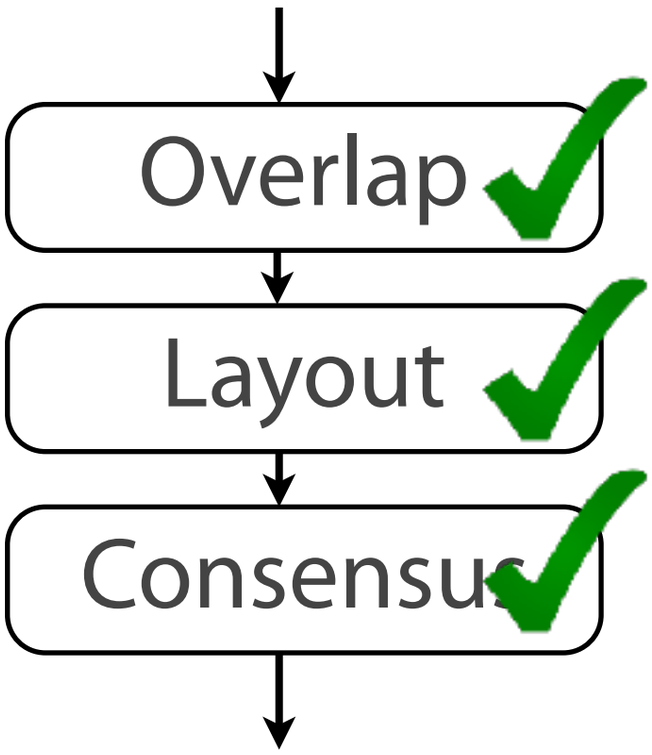
Consensus

Take reads that make up a contig and line them up

Take *consensus*, i.e. majority vote

Complications: (a) sequencing error, (b) ploidy

Overlap Layout Consensus



Build overlap graph

Coalesce paths into *contigs*

Pick nucleotide sequence for each contig

OLC drawbacks

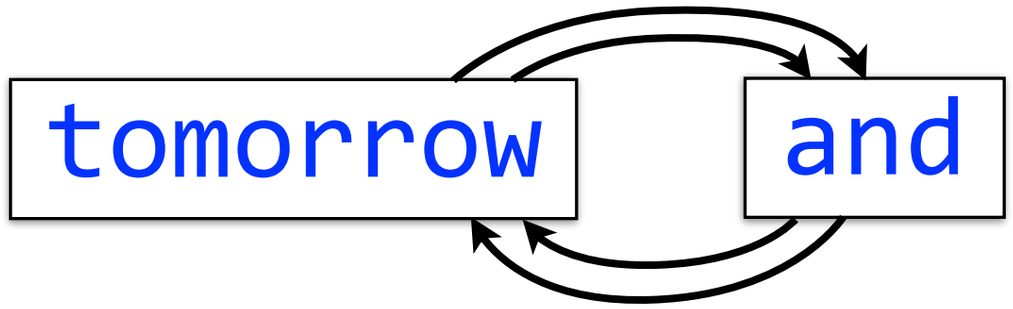
Building overlap graph is slow. We saw $O(N + a)$ and $O(N^2)$ approaches.

Overlap graph is big; one node per read, # edges can grow superlinearly with # reads

Sequencing datasets are ~ 100s of millions or billions of reads

Different kind of graph

"tomorrow and tomorrow and tomorrow"



An edge represents an ordered pair of adjacent words in the input

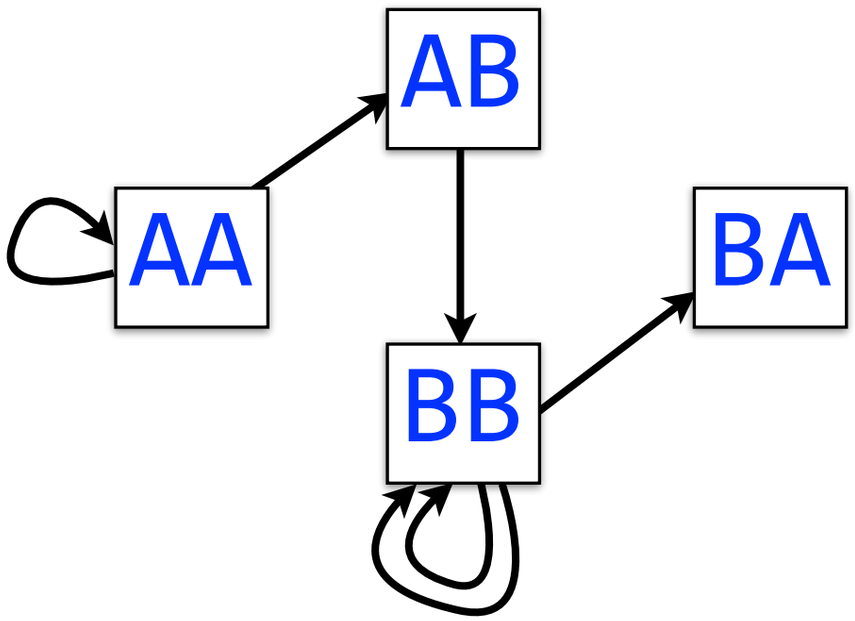
Multigraph: there can be more than one edge from node A to node B

De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

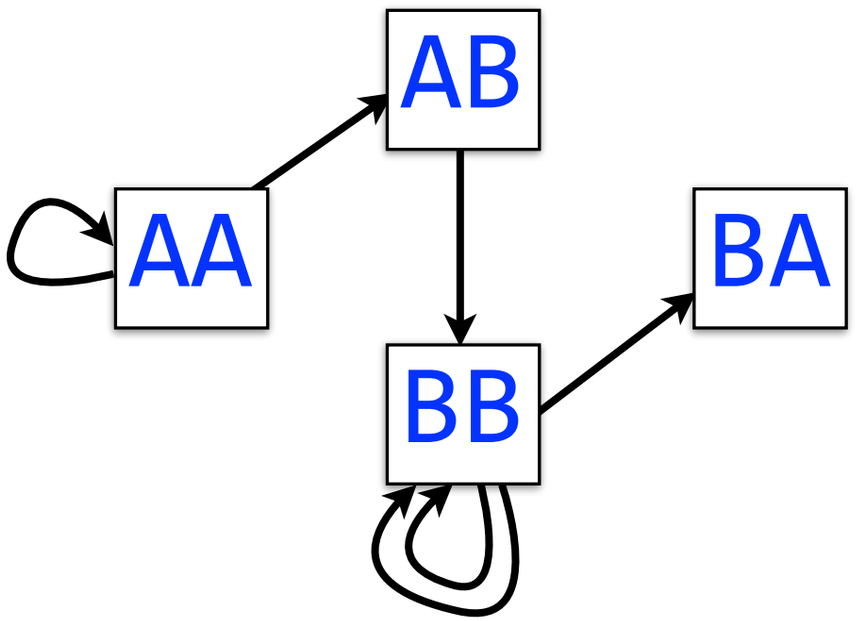
L/R 2-mers: AA, AA AA, AB AB, BB BB, BB BB, BB BB, BA



One edge per *k*-mer

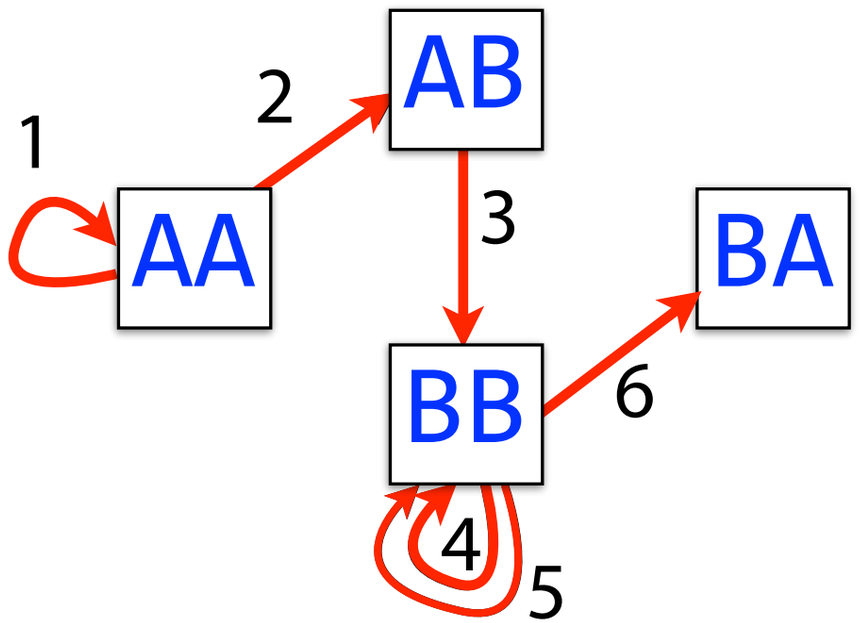
One node per distinct *k*-1-mer

De Bruijn graph



Walk crossing each edge exactly once gives a reconstruction of the genome

De Bruijn graph



AAABBBBA

Walk crossing each edge exactly once gives a reconstruction of the genome. This is an *Eulerian walk*.

De Bruijn graph

Aside: how do you pronounce "De Bruijn"?

There is debate:

https://www.biostars.org/p/7186/

I still don't quite know. I say "De Broin" (rhymes with "groin")

I asked a Dutch person once; his pronunciation sounded more like "De Brown"

Nicolaas Govert
de Bruijn
1918 -- 2012

Directed multigraph

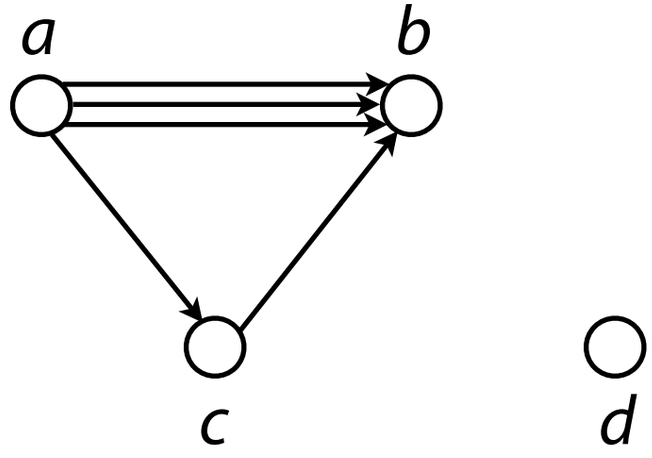
Directed **multigraph** $G(V, E)$ consists of set of *vertices, V* and **multiset** of *directed edges, E*

Otherwise, like a directed graph

Node's *indegree* = # incoming edges

Node's *outdegree* = # outgoing edges

De Bruijn graph is a directed multigraph



$V = \{ a, b, c, d \}$

$E = \{ (a, b), (a, b), (a, b), (a, c), (c, b) \}$

Repeated

Eulerian walk definitions and statements

Node is *balanced* if indegree equals outdegree

Node is *semi-balanced* if indegree differs from outdegree by 1

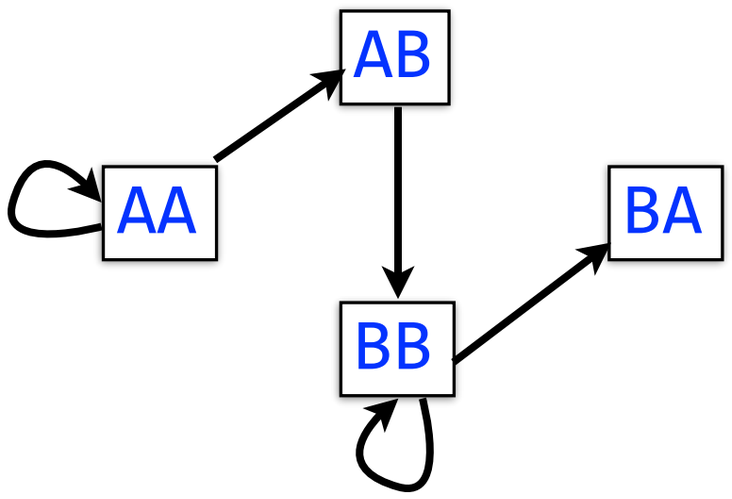
Graph is *connected* if each node can be reached by some other node

Eulerian walk visits each edge exactly once

Not all graphs have Eulerian walks. Graphs that do are *Eulerian*.
(For simplicity, we won't distinguish Eulerian from semi-Eulerian.)

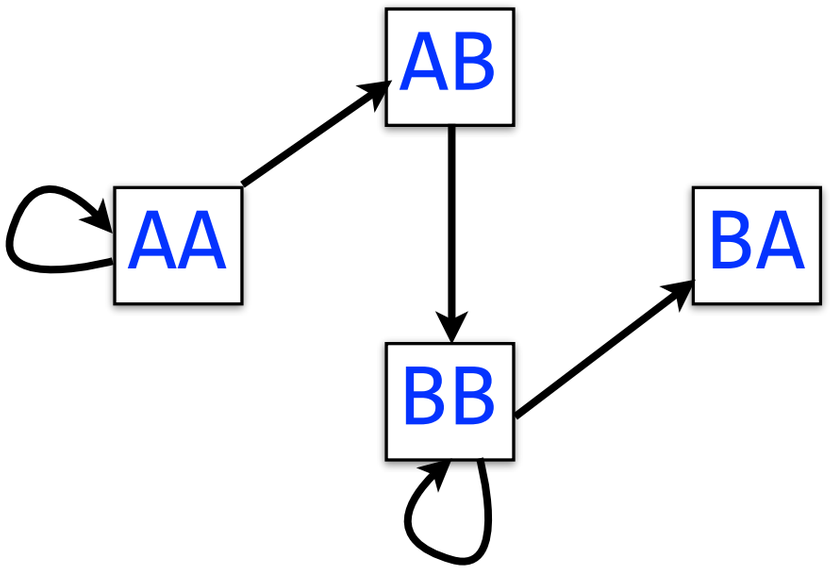
A directed, connected graph is Eulerian if and only if it has at most 2 semi-balanced nodes and all other nodes are balanced

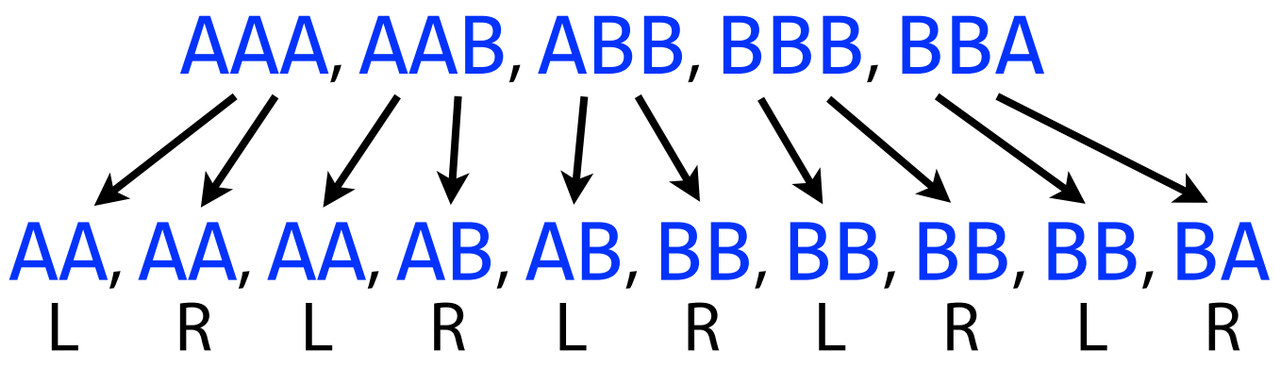
Jones and Pevzner section 8.8



De Bruijn graph

Back to de Bruijn graph





Is it Eulerian? Yes

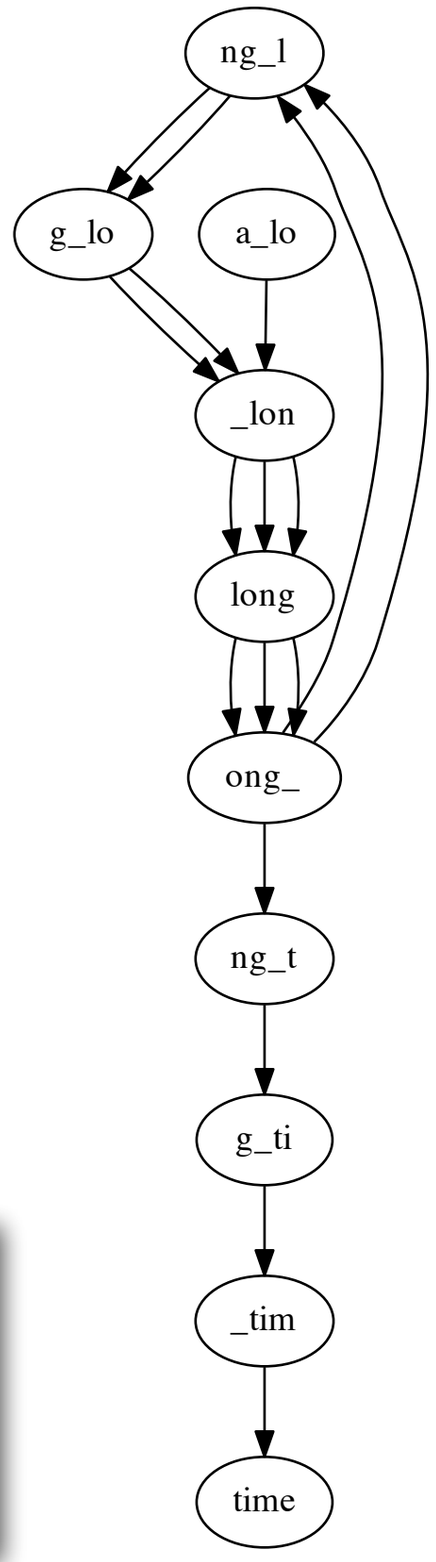
Argument 1: AA → AA → AB → BB → BB → BA

Argument 2: AA and BA are semi-balanced, AB and BB are balanced

De Bruijn graph

Full illustrative de Bruijn graph and Eulerian walk implementation:

http://bit.ly/CG_DeBruijn



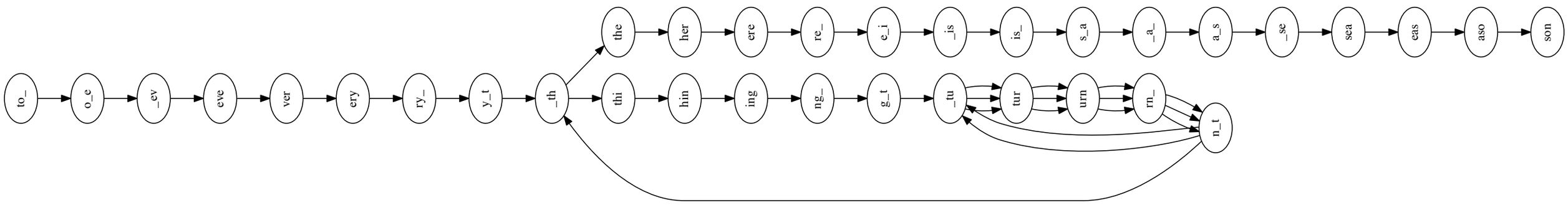
Example where Eulerian walk gives correct answer for small *k* whereas Greedy-SCS could spuriously collapse repeat:

```
>>> G = DeBruijnGraph(["a_long_long_long_time"], 5)
>>> print G.eulerianWalkOrCycle()
['a_lo', '_lon', 'long', 'ong_', 'ng_l', 'g_lo',
'_lon', 'long', 'ong_', 'ng_l', 'g_lo', '_lon',
'long', 'ong_', 'ng_t', 'g_ti', '_tim', 'time']
```

De Bruijn graph

```
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 4)
>>> path = G.eulerianWalkOrCycle()  # Fast! Linear in # edges
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
to_every_thing_turn_turn_turn_there_is_a_season
```

http://bit.ly/CG_DeBruijn

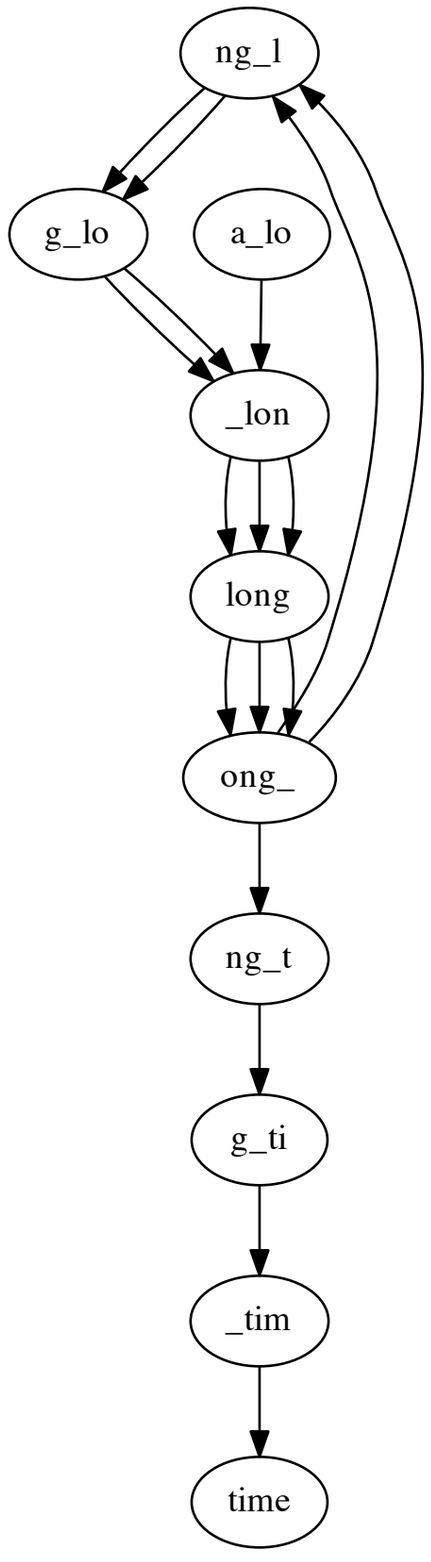


Recall: This is not generally possible or tractable in the overlap/SCS formulation

De Bruijn graph

Assuming perfect sequencing, procedure yields graph with Eulerian walk that can be found efficiently.

We saw cases where Eulerian walk corresponds to the original superstring. Is this always the case?

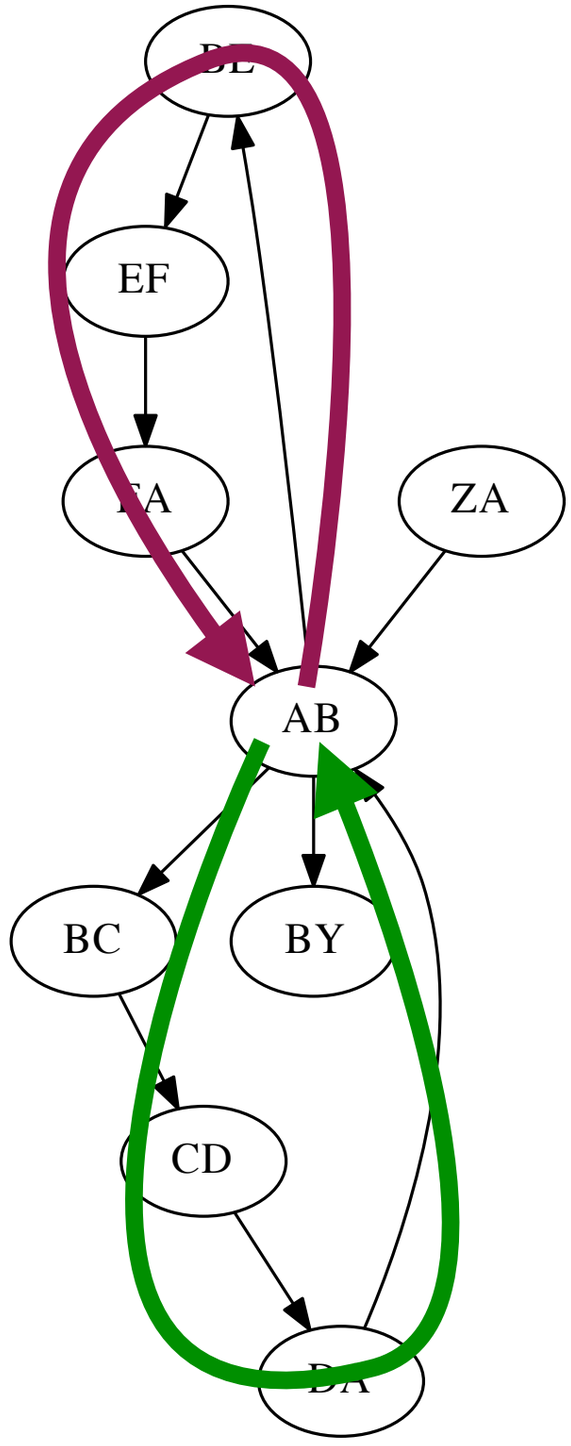


De Bruijn graph

Problem 1: Repeats still cause misassembles

ZA → AB → BE → EF → FA → AB → BC → CD → DA → AB → BY

ZA → AB → BC → CD → DA → AB → BE → EF → FA → AB → BY



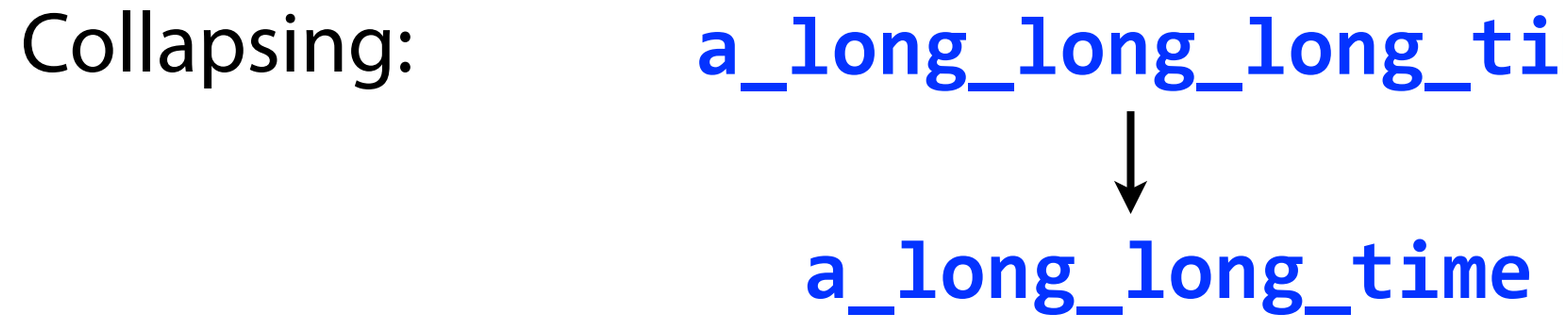
Problem 2:

We've been building DBGs assuming "perfect" sequencing: each *k*-mer reported exactly once, no mistakes. Real datasets aren't like that.

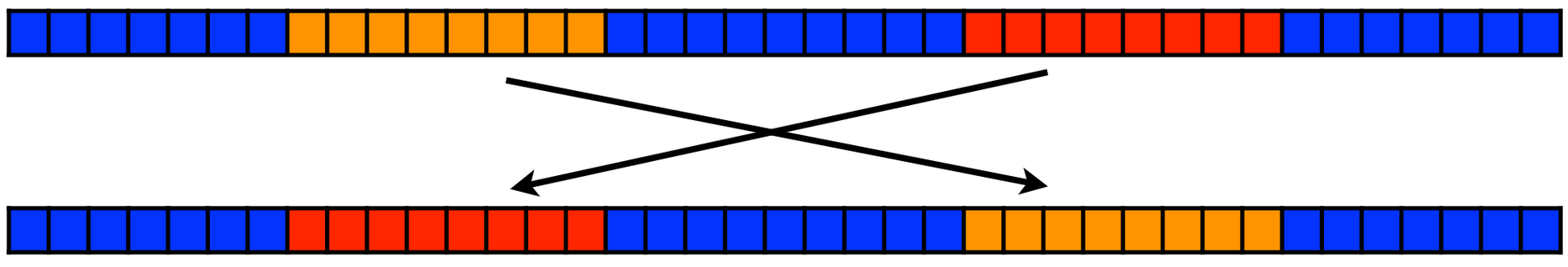
Third law of assembly

Repeats make assembly difficult; whether we can assemble without mistakes depends on length of reads and repetitive patterns in genome

Collapsing:



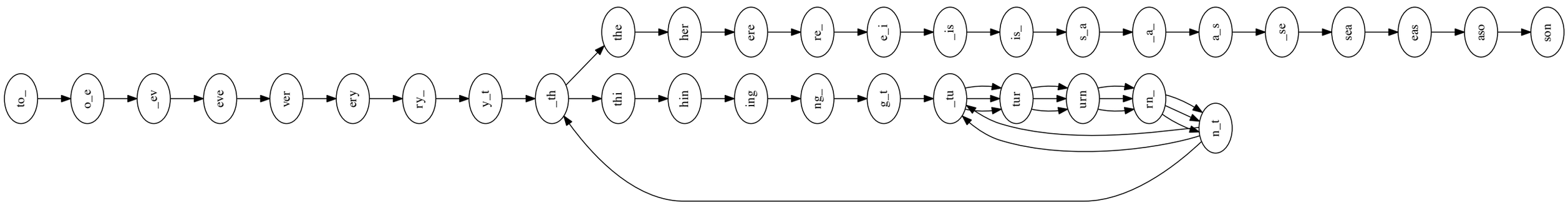
Shuffling:



De Bruijn graph

```
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 4)
>>> path = G.eulerianWalkOrCycle()  # Fast! Linear in # edges
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
to_every_thing_turn_turn_turn_there_is_a_season
```

http://bit.ly/CG_DeBruijn



De Bruijn graph

Case where $k = 4$ works:

```
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 4)
>>> path = G.eulerianWalkOrCycle()
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
to_every_thing_turn_turn_turn_there_is_a_season
```

But $k = 3$ does not:

```
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 3)
>>> path = G.eulerianWalkOrCycle()
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
```

De Bruijn graph

Case where $k = 4$ works:

```
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 4)
>>> path = G.eulerianWalkOrCycle()
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
to_every_thing_turn_turn_turn_there_is_a_season
```

But $k = 3$ does not:

```
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 3)
>>> path = G.eulerianWalkOrCycle()
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
to_every_turn_turn_thing_turn_there_is_a_season
```

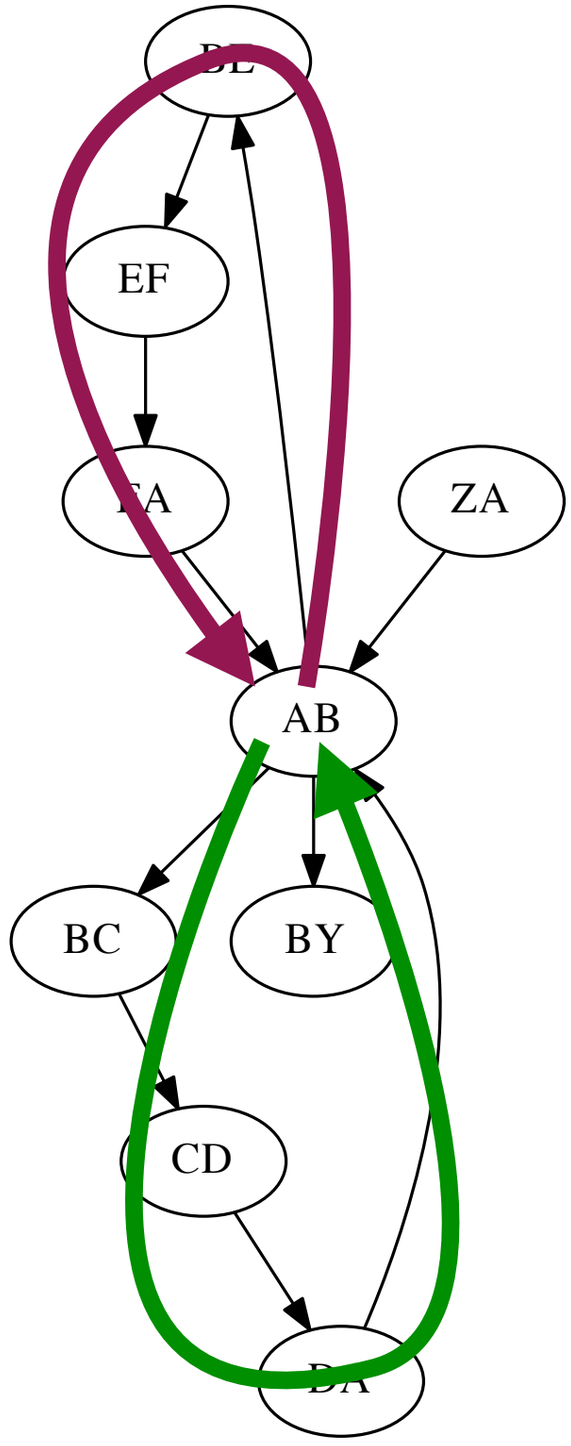
Due to repeats that are unresolvable at $k = 3$

De Bruijn graph

Problem 1: Repeats still cause misassembles

ZA → AB → BE → EF → FA → AB → BC → CD → DA → AB → BY

ZA → AB → BC → CD → DA → AB → BE → EF → FA → AB → BY



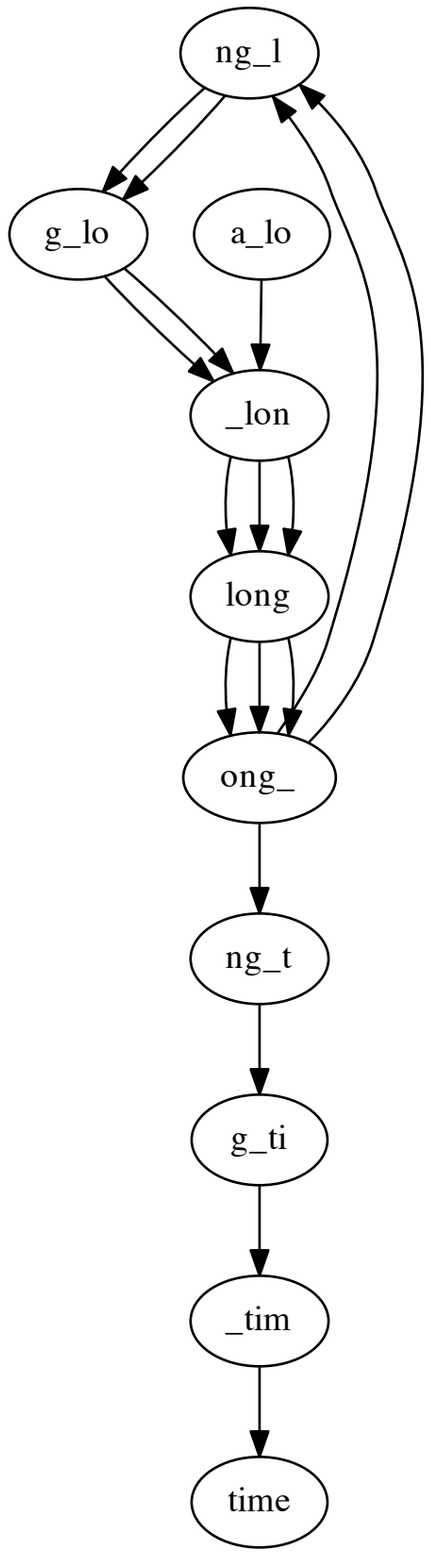
Problem 2:

We've been building DBGs assuming "perfect" sequencing: each *k*-mer reported exactly once, no mistakes. Real datasets aren't like that.

De Bruijn graph

Gaps in coverage (missing *k*-mers) lead to *disconnected* or non-Eulerian graph

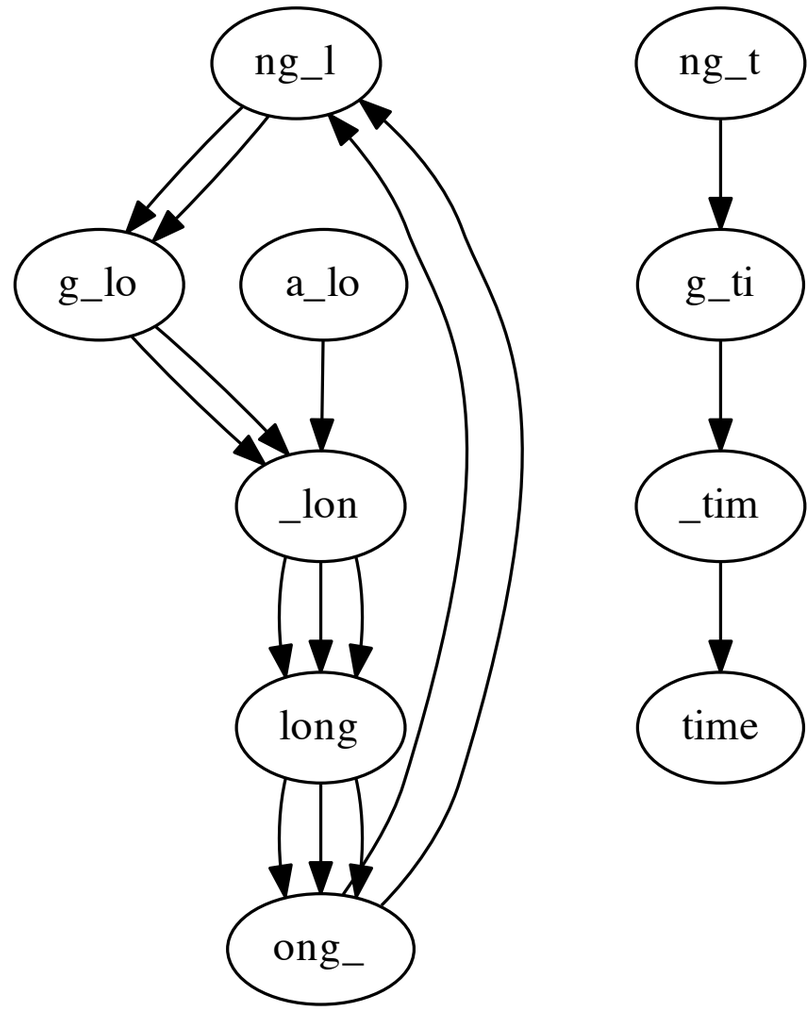
Graph for `a_long_long_long_time`, $k = 5$:



De Bruijn graph

Gaps in coverage (missing *k*-mers) lead
to *disconnected* or non-Eulerian graph

Graph for `a_long_long_long_time`, $k = 5$ but *omitting* `ong_t` :

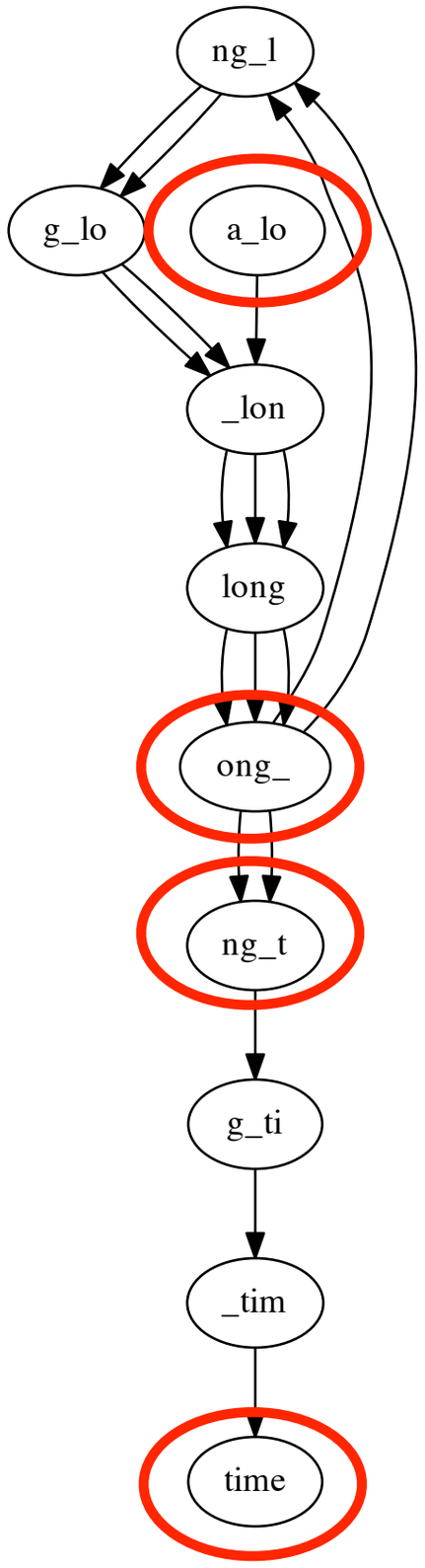


De Bruijn graph

Coverage *differences* make graph non-Eulerian

Graph for `a_long_long_long_time`,
$k = 5$, with *extra copy* of `ong_t` :

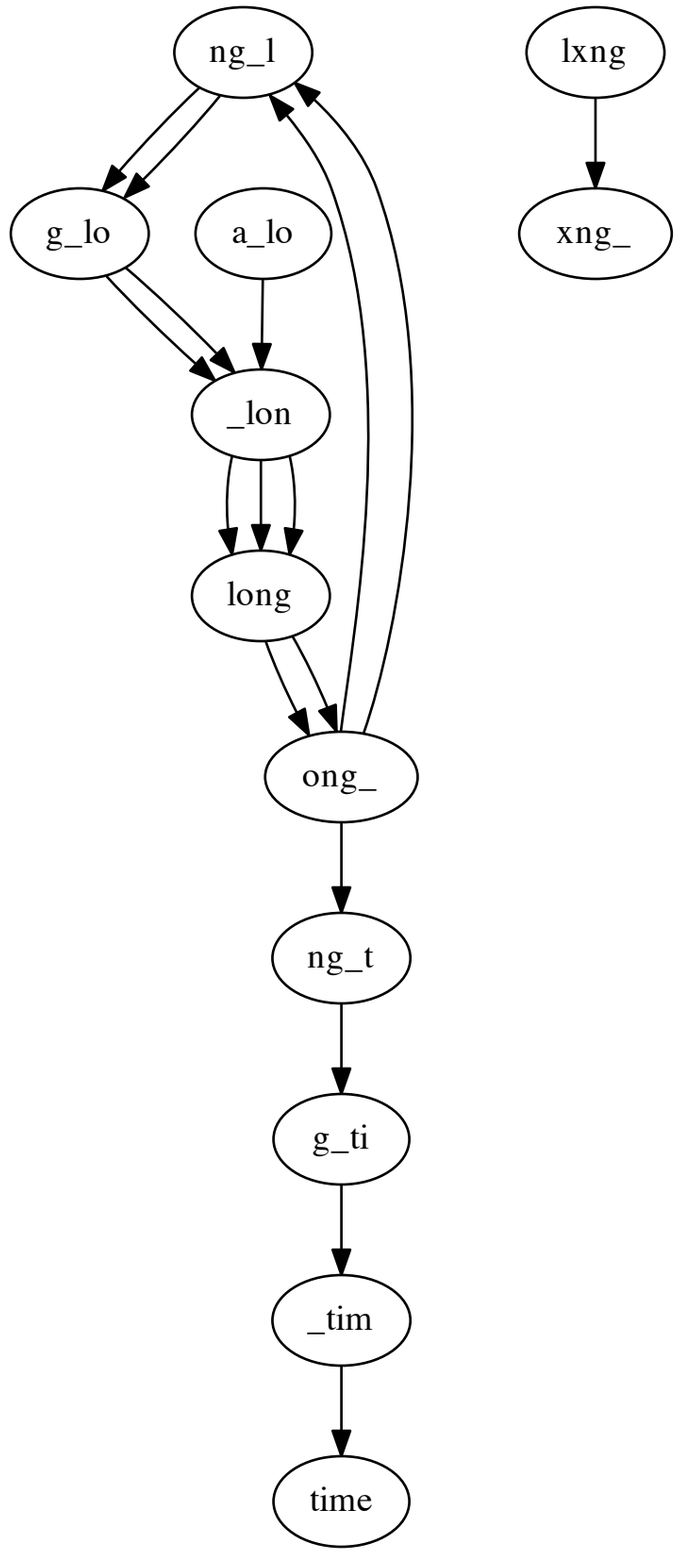
4 semi-balanced nodes



De Bruijn graph

Errors and differences between chromosomes also lead to non-Eulerian graphs

Graph for `a_long_long_long_time`, $k = 5$ but with error that turns one copy of `long_` into `lxng_`



De Bruijn graph

Casting assembly as Eulerian walk is appealing, but not practical

Uneven coverage, sequencing errors, etc make graph non-Eulerian

Even if graph were Eulerian, repeats yield many possible walks

Kingsford, Carl, Michael C. Schatz, and Mihai Pop. "Assembly complexity of prokaryotic genomes using short reads." *BMC bioinformatics* 11.1 (2010): 21.

De Bruijn Superwalk Problem (DBSP) seeks a walk over the De Bruijn graph, where walk contains each read as a *subwalk*

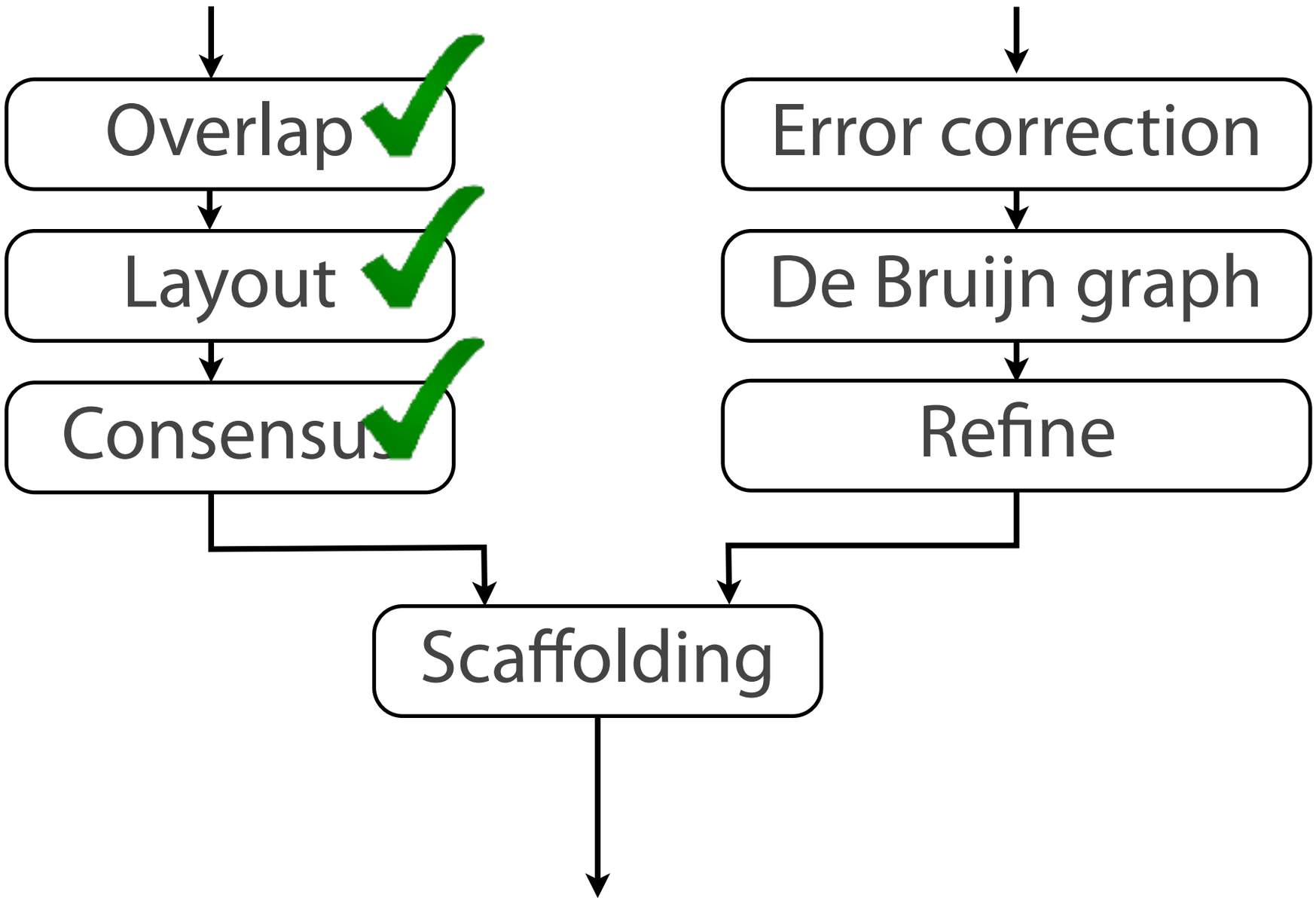
Proven NP-hard!

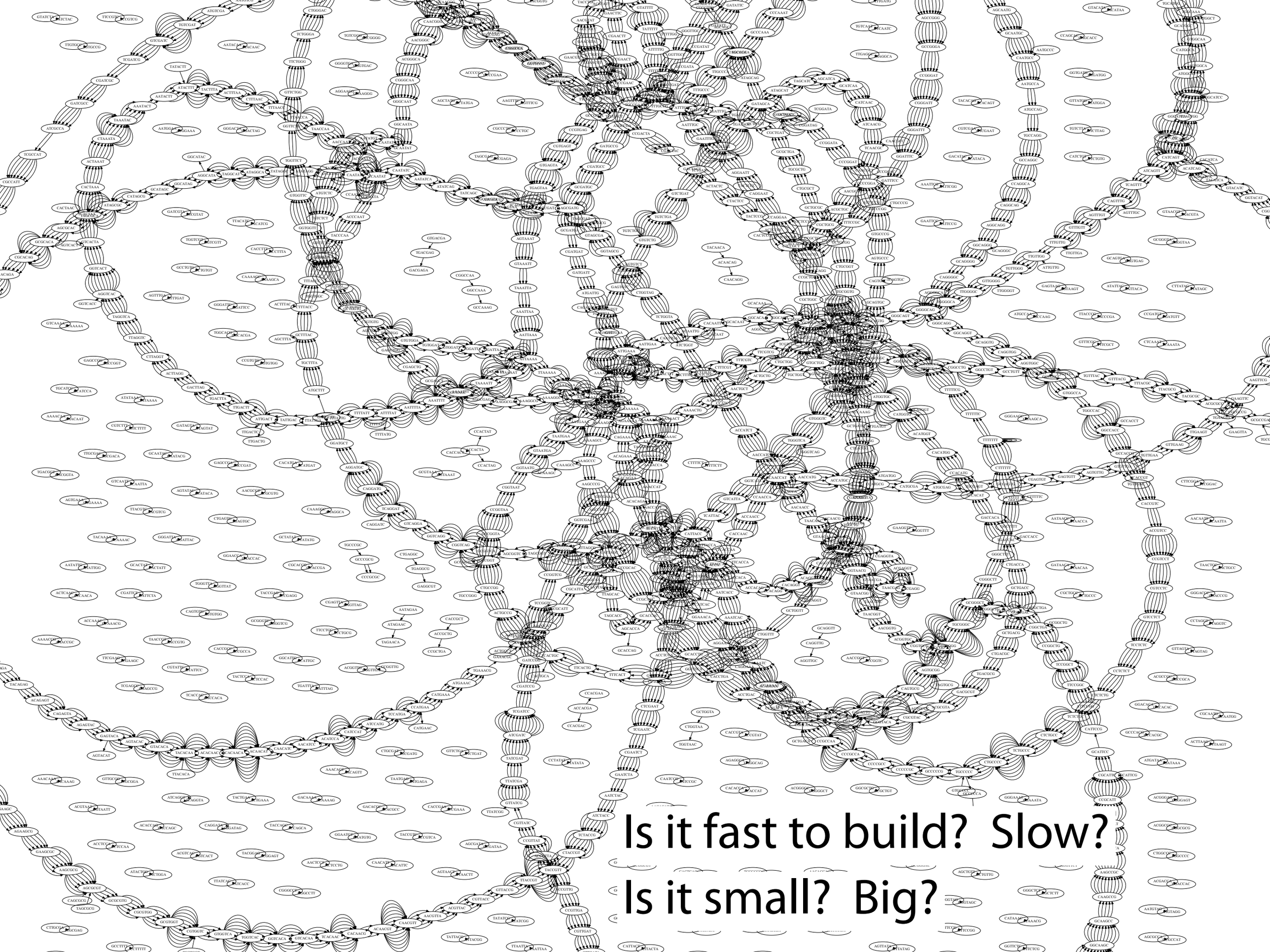
Medvedev, Paul, et al. "Computability of models for sequence assembly." *Algorithms in Bioinformatics*. Springer Berlin Heidelberg, 2007. 289-301.

Assembly alternatives

Alternative 1: Overlap-Layout-Consensus (OLC) assembly

Alternative 2: De Bruijn graph (DBG) assembly




Is it fast to build? Slow?
Is it small? Big?

De Bruijn graph

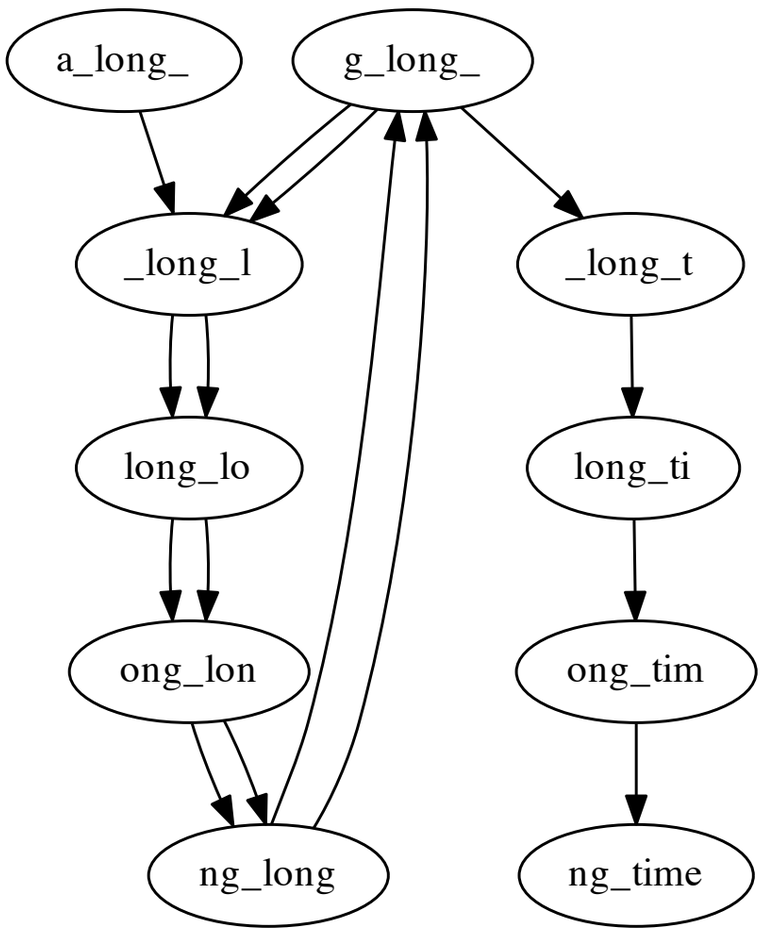
Pick $k = 8$

Genome: a_long_long_long_time

Reads: a_long_long_long, ng_long_l, g_long_time

k-mers:

```
a_long_l          ng_long_   g_long_t
 _long_lo          g_long_l   _long_ti
  long_lon                     long_tim
   ong_long                     ong_time
    ng_long_
     g_long_l
      _long_lo
       long_lon
        ong_long
```



For each read:

For each *k*-mer:

Add *k*-mer's left and right *k*-1-mers to graph if not there already. Draw an edge from left to right *k*-1-mer.

De Bruijn graph

d = 6 x 10^9 reads
n = 100 nt
} ≈ 1 week-long run of

Illumina HiSeq 2000

Sequencer outputs **d** reads of length **n**, total length N = **dn**.

To build graph: Pick k. Usually k is short relative to read length (k = 30 to 50 is common).

For each read:

For each k-mer:

Add k-mer's left and right k-1-mers to graph if not there already. Draw an edge from left to right k-1-mer.

De Bruijn graph

d = 6 x 10^9 reads
n = 100 nt
} ≈ 1 week-long run of

Illumina HiSeq 2000

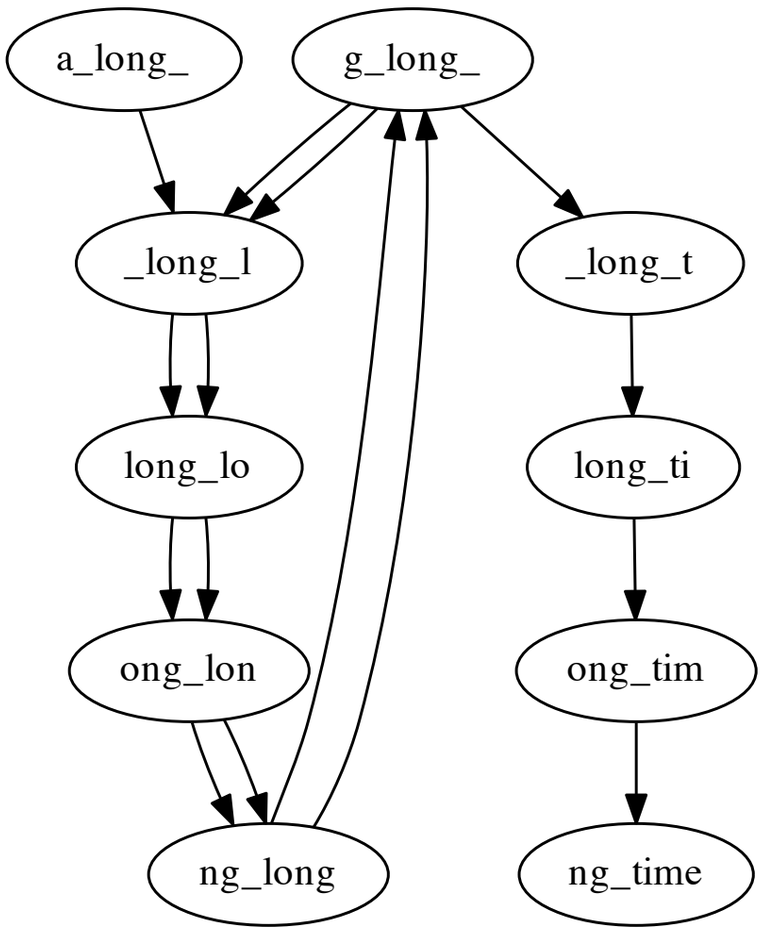
Sequencer outputs **d** reads of length **n**, total length N = **dn**.

To build graph: Pick k. Usually k is short relative to read length (k = 30 to 50 is common).

k-mers (edges): *O(N)*

nodes is at most 2 · (# edges); typically much smaller due to repeated k-1-mers *O(N)*

De Bruijn graph



How much work to build graph?

For each *k*-mer, add 1 edge and up to 2 nodes

Reasonable to say this is O(1) expected work

Say hash map holds nodes & edges

Say *k*-1-mers fit in O(1) machine words, and hashing O(1) words is O(1) work

Querying / adding a key is O(1) expected work

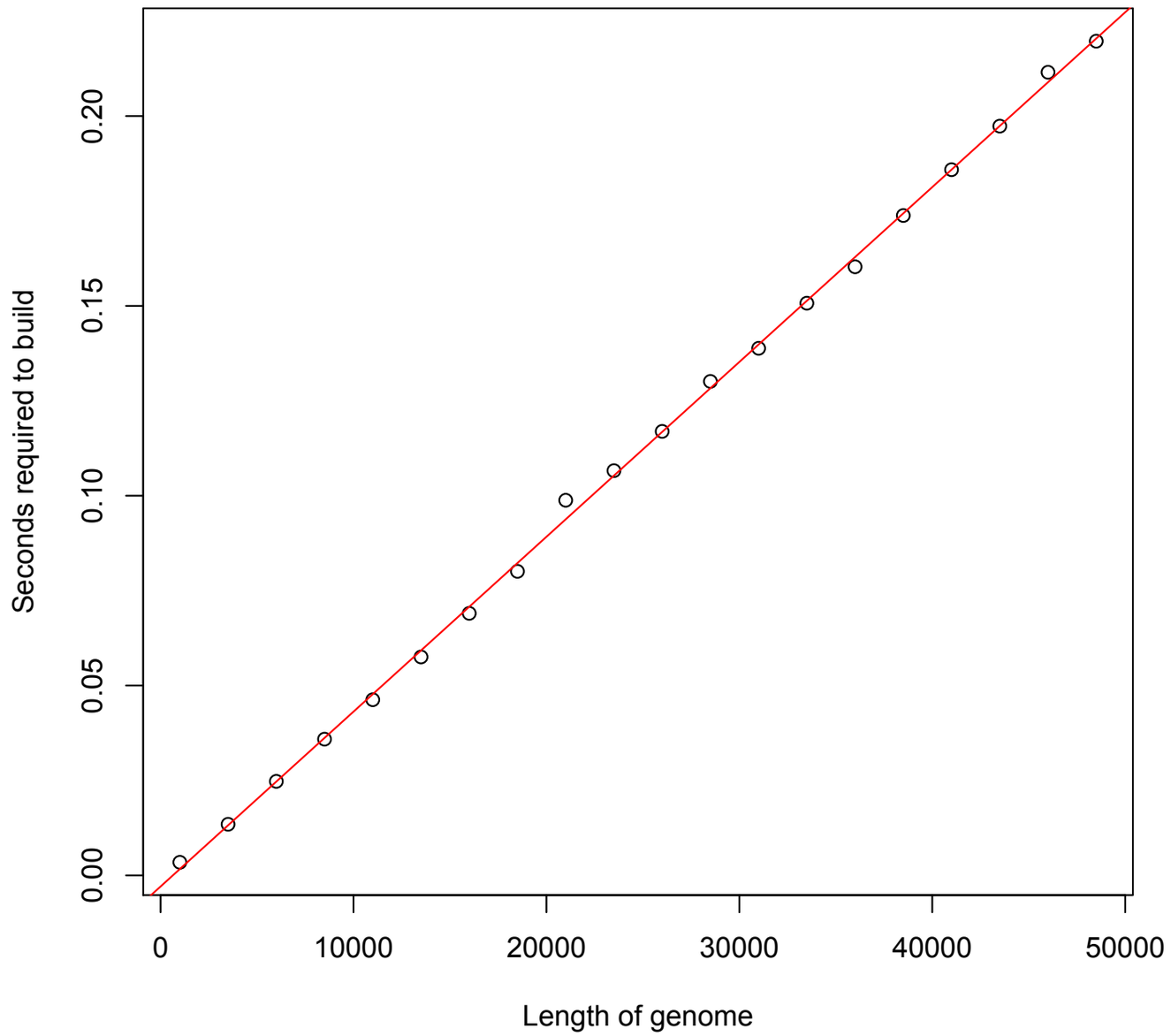
O(1) expected work for 1 *k*-mer, **O(*N*) overall**

De Bruijn graph

Timed De Bruijn graph construction applied to progressively longer prefixes of lambda phage genome, $k = 14$

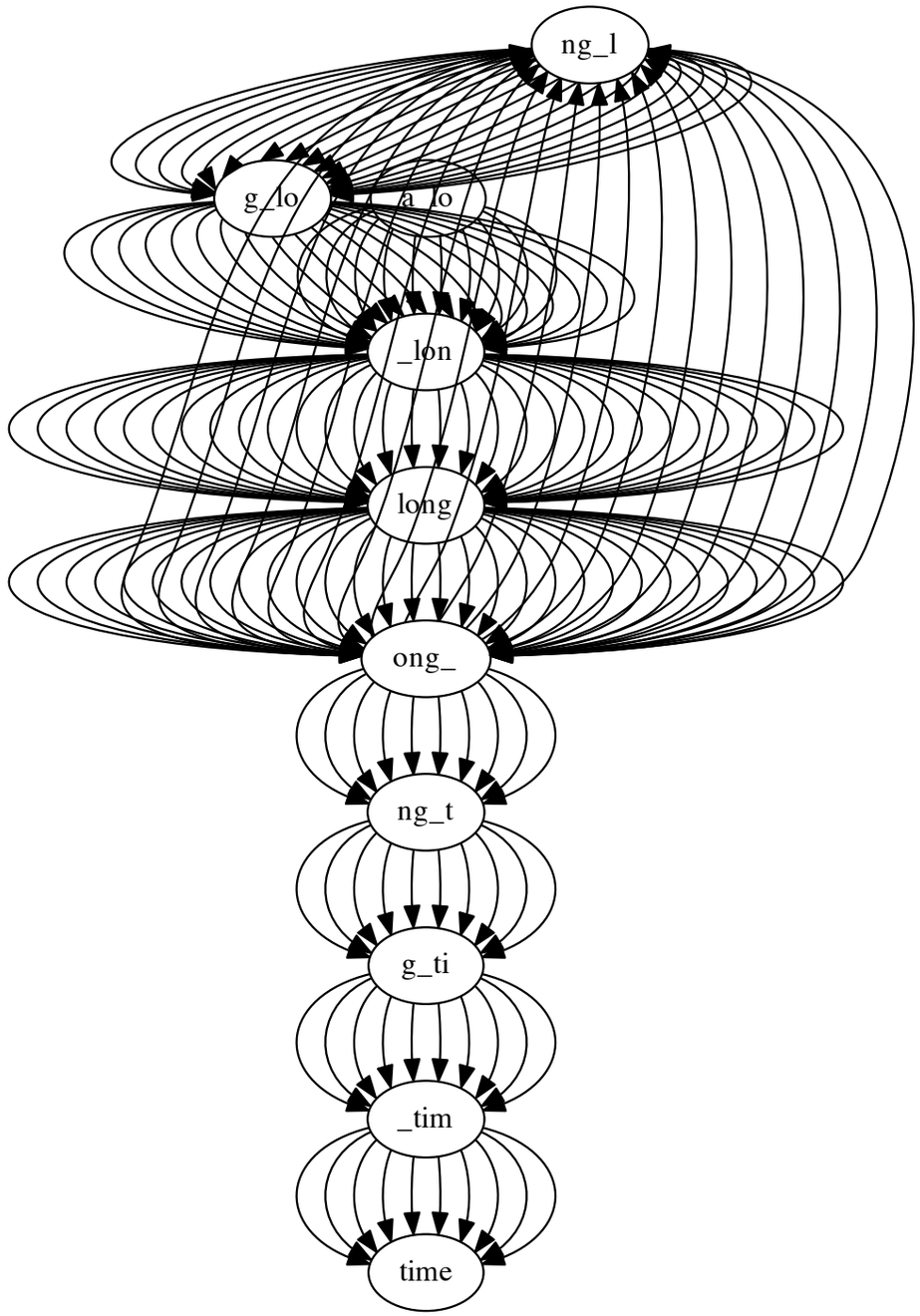
O(N) expectation works in practice

(in this case at least)

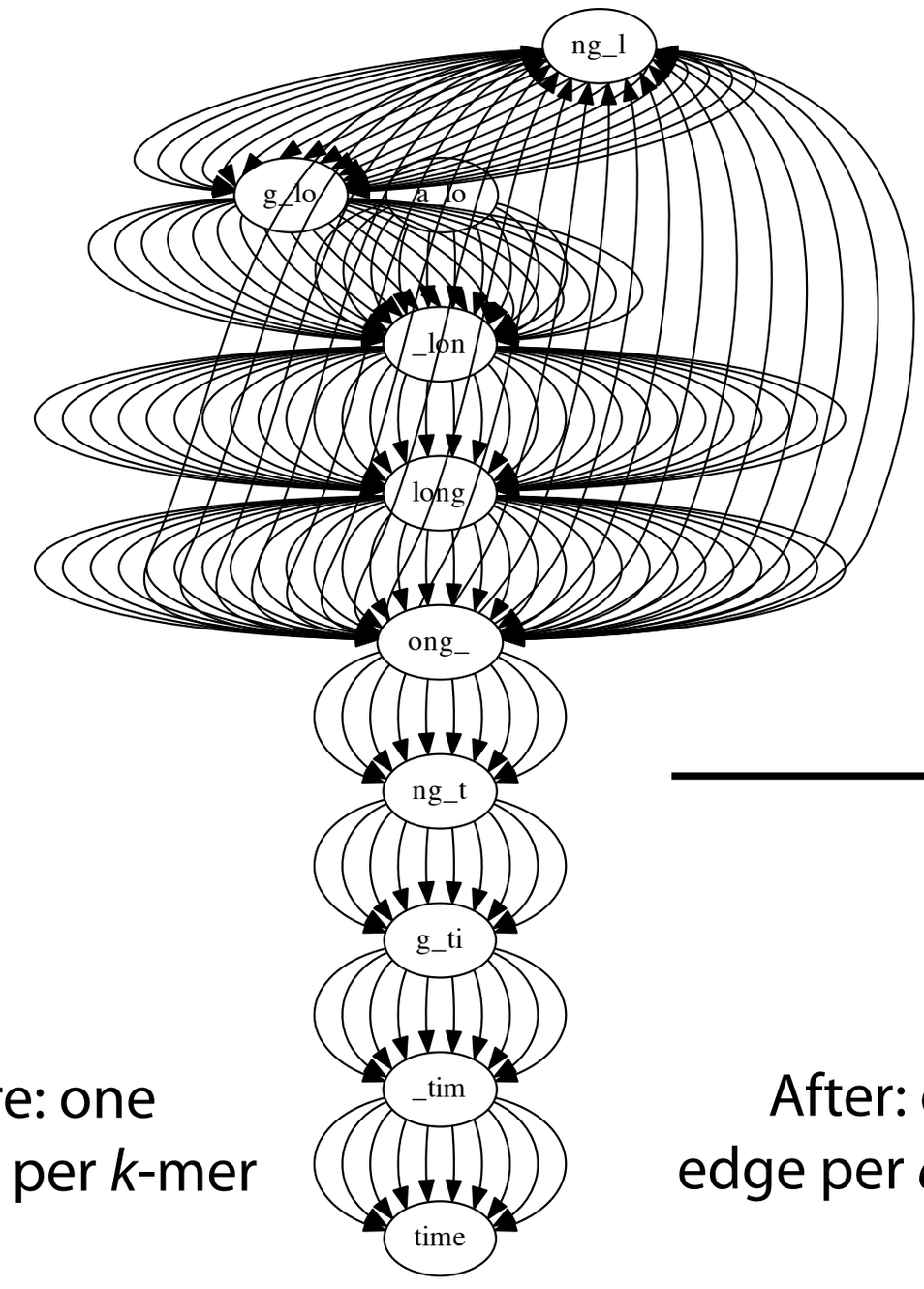


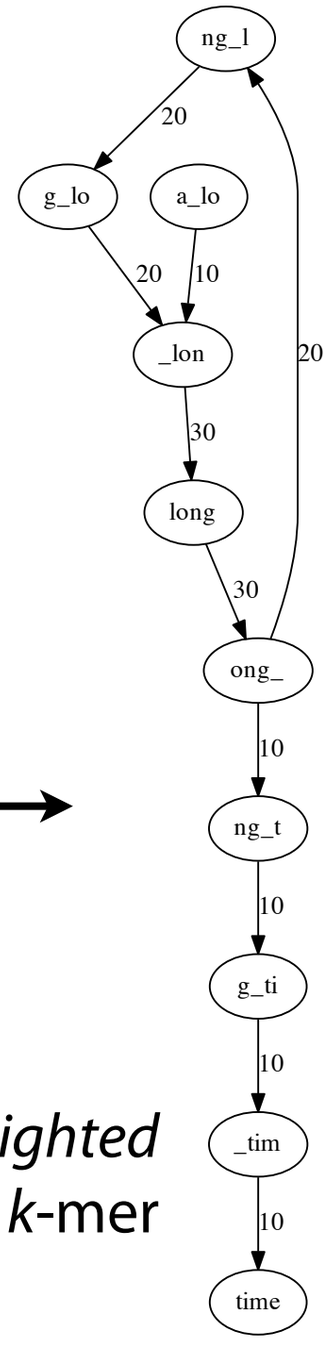
De Bruijn graph

In typical assembly projects,
average coverage is ~ 30 - 50



De Bruijn graph





Before: one edge per *k*-mer

After: one *weighted* edge per *distinct k*-mer

De Bruijn graph

of nodes and edges both O(*N*)

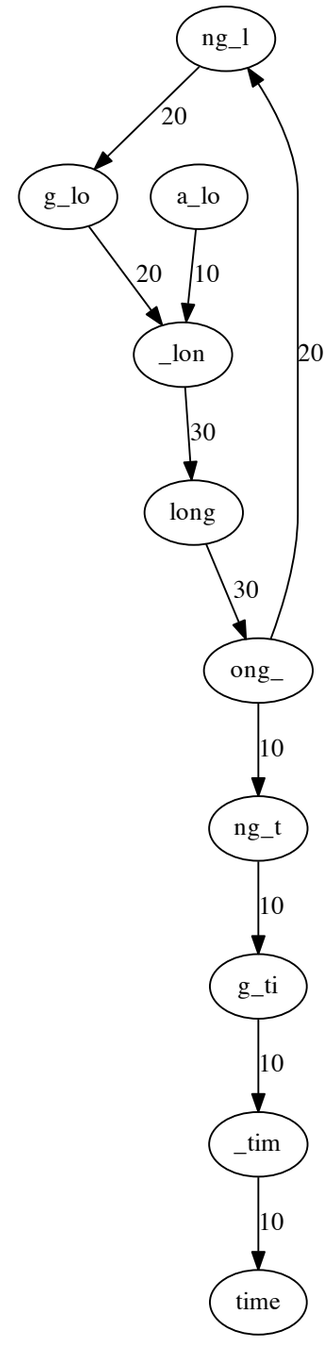
Say (a) reads are error-free, (b) we have one *weighted* edge for each *distinct k*-mer, and (c) length of genome is *G*

1 node per distinct *k*-1-mer, 1 edge per distinct *k*-mer

Can't have more distinct *k*-mers than *k*-mers in the genome; likewise for *k*-1-mers

So # of nodes and edges are both O(*G*)

Combine with the O(*N*) bound and the # of nodes and edges are both O(min(*N*, *G*))



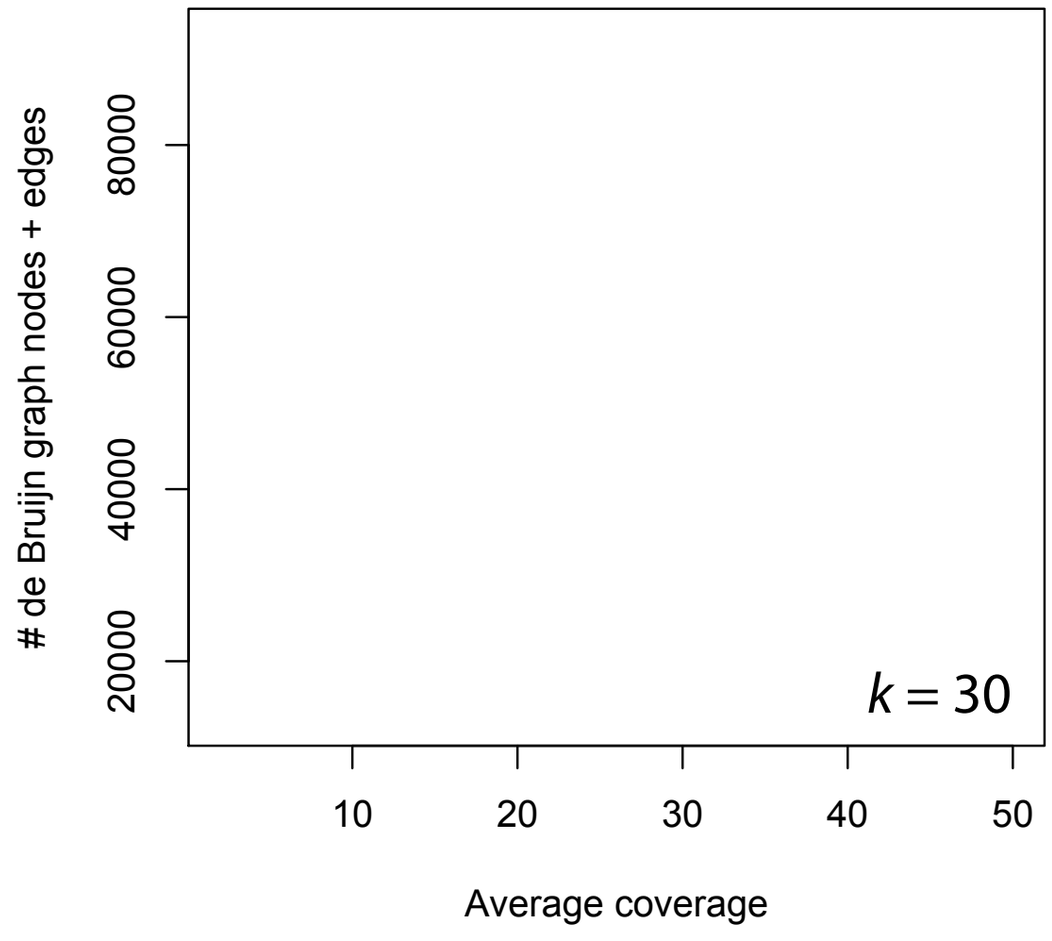
De Bruijn graph

At high coverage, O(min(*N*, *G*)) bound is advantageous

Genome: lambda phage (~48,500 bp)

Draw random *k*-mers until target average coverage (x axis) is reached

Build graph, sum # nodes and # edges (y axis)



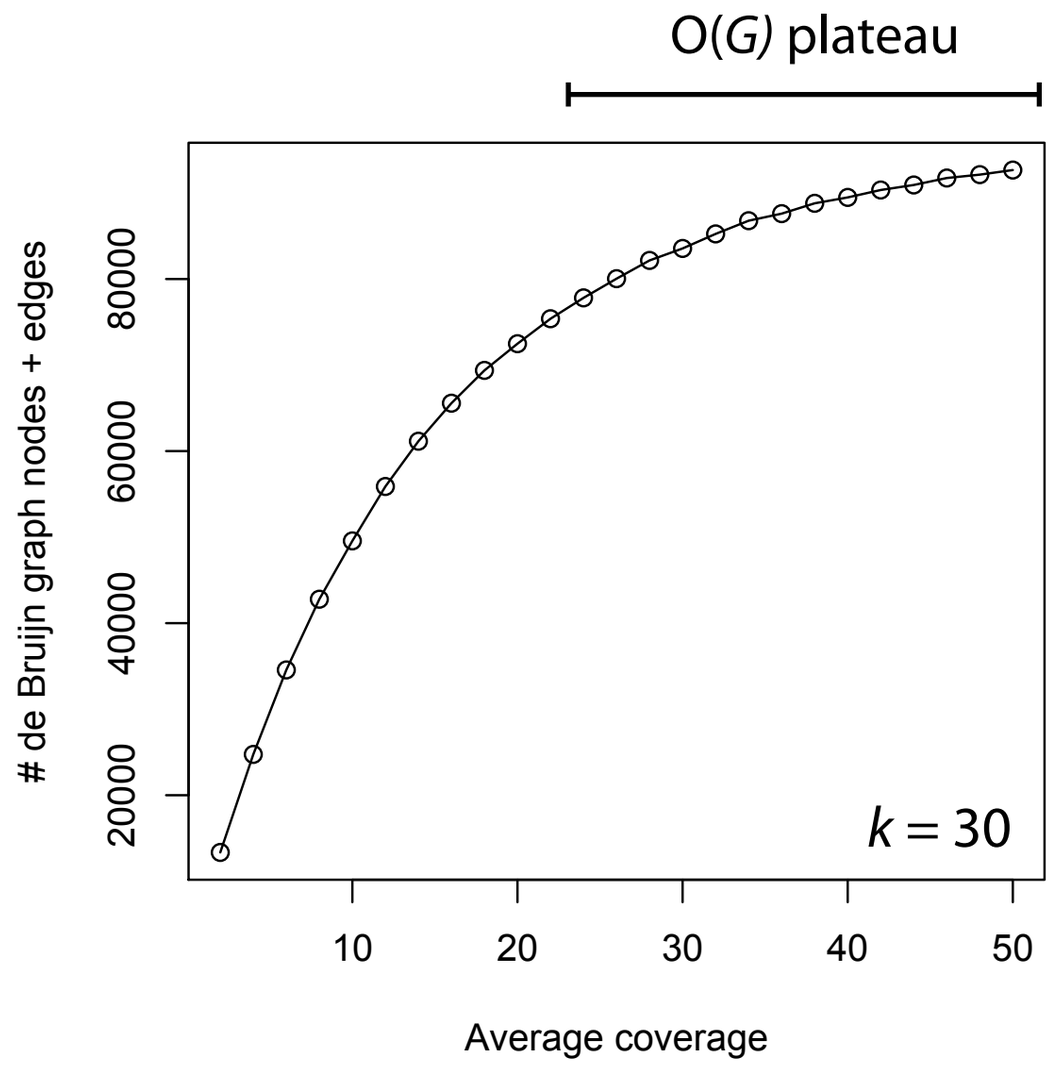
De Bruijn graph

At high coverage, O(min(*N*, *G*)) bound is advantageous

Genome: lambda phage (~48,500 bp)

Draw random *k*-mers until target average coverage (x axis) is reached

Build graph, sum # nodes and # edges (y axis)



De Bruijn graph

Advantages

Can build in O(N) expected time, N = total length of reads

With error-free data, space is O(min(N, G)); G = genome length

When average coverage is high, $G \ll N$

Compares favorably with overlap graph

Overlap graph has node for every read, edge for every overlap

Fast construction (suffix tree) is O($N + a$) time, where a is O(d^2)

De Bruijn graph

Disadvantages

Reads are immediately split into shorter *k*-mers, losing the ability to resolve some repeats resolvable by overlap graph

Only relatively short, exact overlaps are considered, which makes handling of sequencing errors more complicated

We lose *read coherence*. Some paths through De Bruijn graph are inconsistent with respect to input reads.

Assembly alternatives

	De Bruijn	Overlap
Time to build	$O(N)$	Suffix tree: $O(N + a)$ Dyn Prog: $O(N^2)$
Space	O(N) Error-free: $O(\min(N, G))$	$O(N + a)$

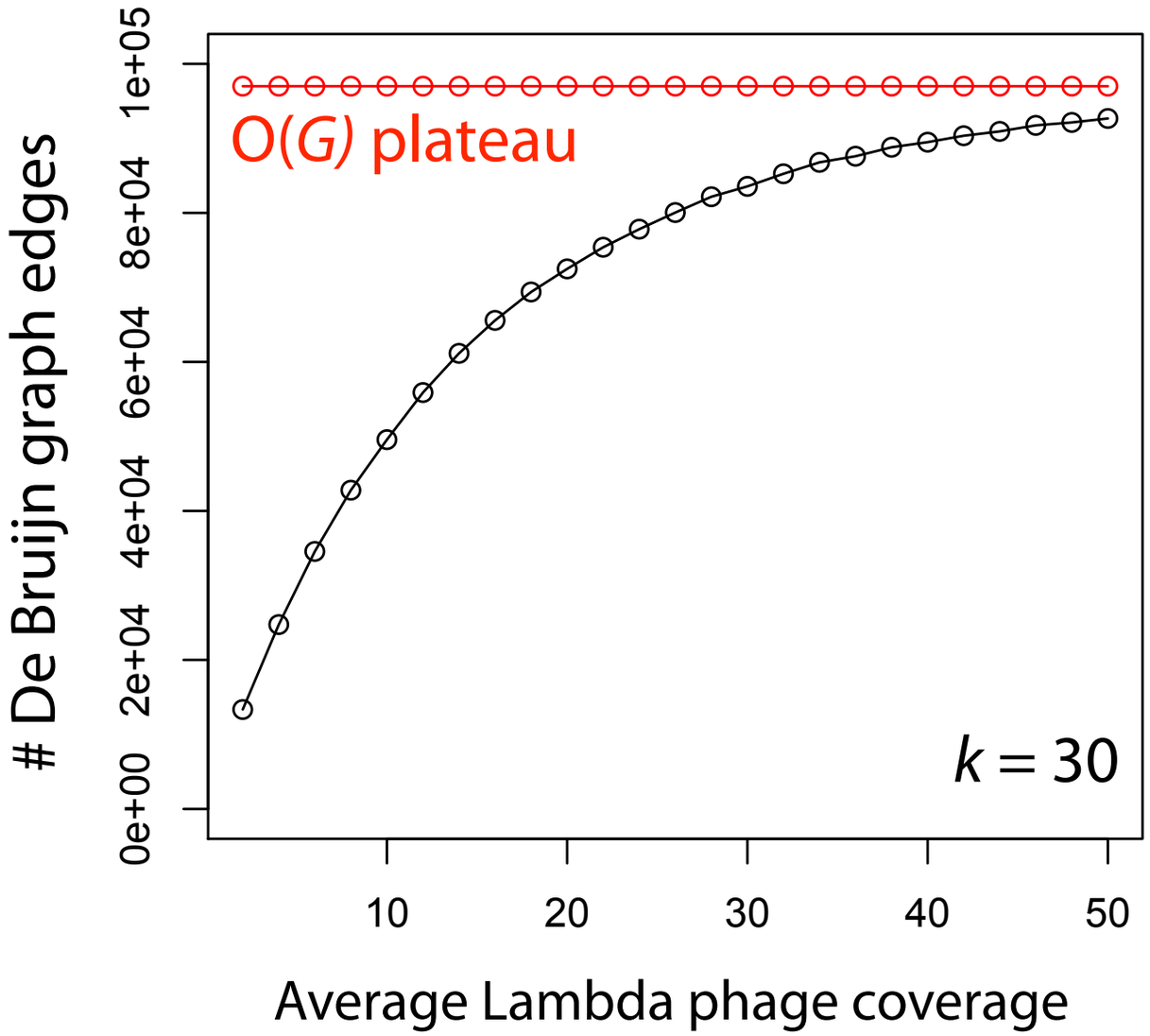
n = # reads
d = read length
$N = dn$ = # bases

a = # overlaps; $a \in O(n^2)$
G = source genome length

When average coverage is high, $G \ll N$ and the G is the more relevant bound for De Bruijn graph size

Error correction

When data is error-free, # nodes, edges in De Bruijn graph is O(min(*G*, *N*))



What about data with sequencing errors?

Error correction

How many possible DNA strings of length *k*? 4^k

How many possible DNA strings of length 20? $4^{20} = 2^{40} \approx 1$ trillion

How many strings of length 20 in human genome? ~3 billion

For large *k*, set of *k*-mers in genome is tiny subset of all 4^k *k*-mers

Errors tend to yield *new k-mers* that don't appear elsewhere

> Given *k*-mer from genome, we expect most of its *neighbors*
> (e.g. by Hamming distance) are not in the genome

Analogy: correctly / incorrectly spelled words in collection of documents

Error correction

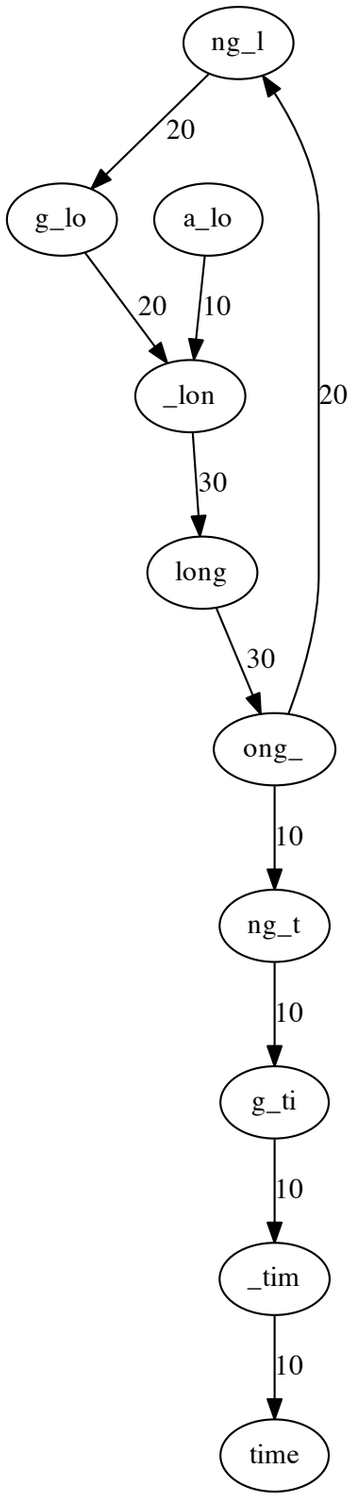
Correcting errors up-front prevents De Bruijn graph from growing far beyond O(*G)* plateau

How to correct?

Analogy: how to spell check a language you've never seen before?

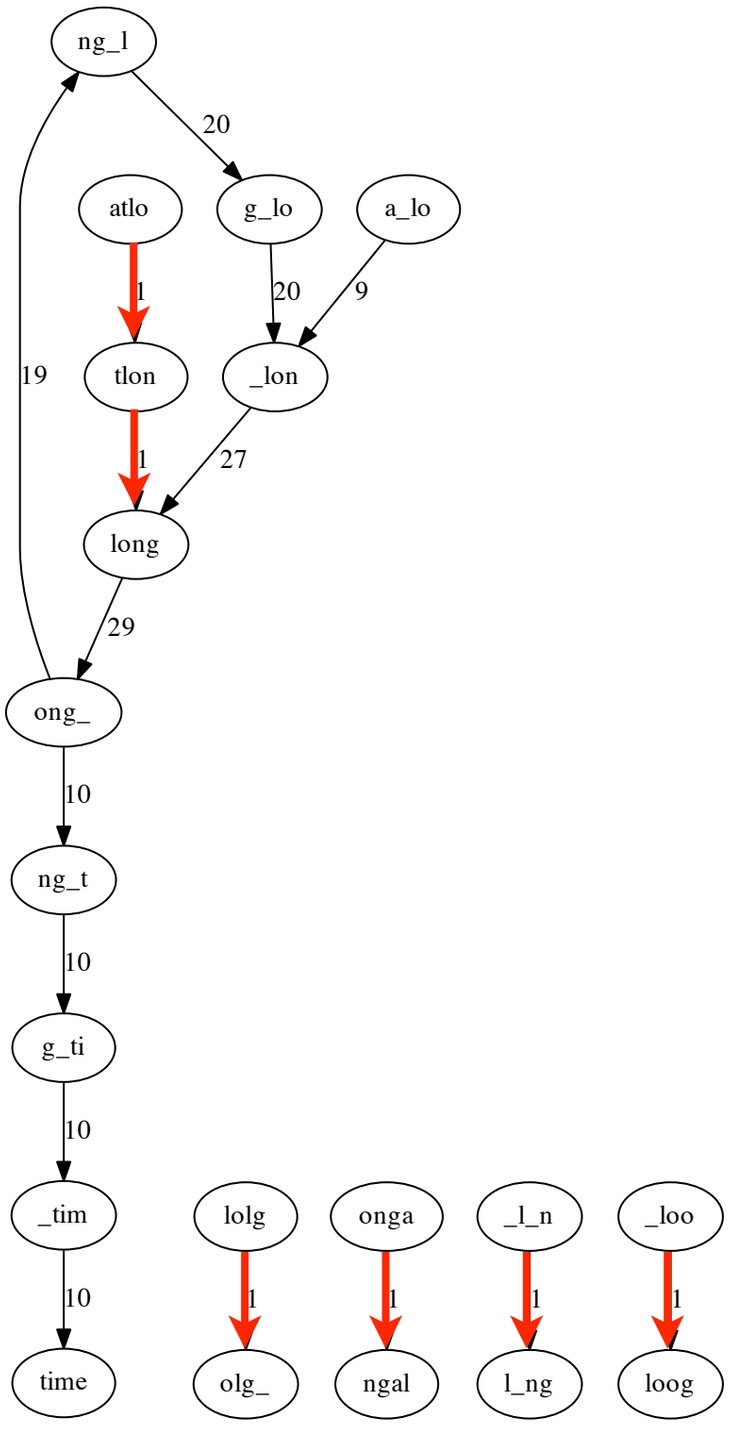
Errors tend to turn frequent words (*k*-mers) to infrequent ones. Corrections should do the reverse.

Error correction



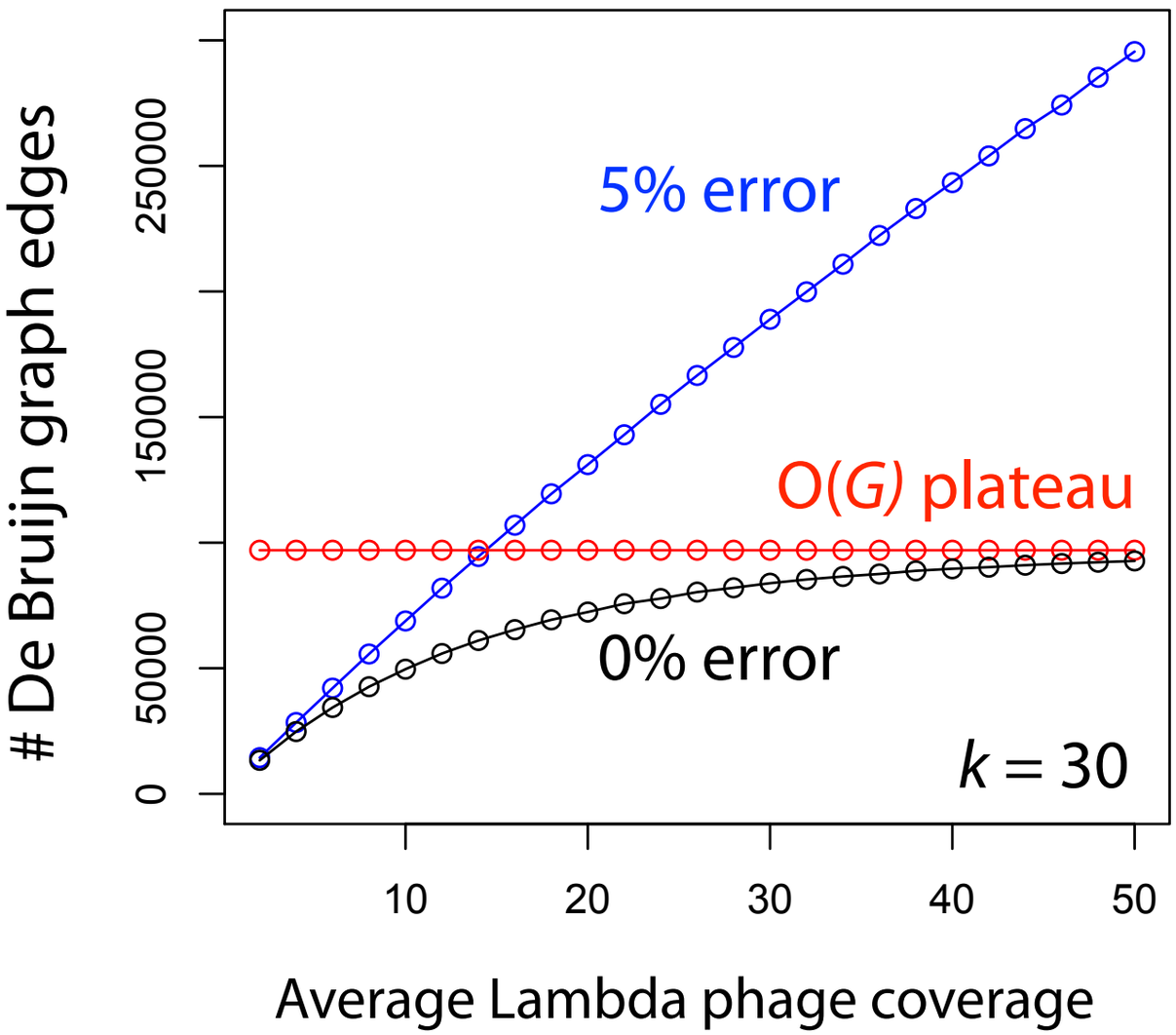
Left: Take example, mutate a *k*-mer character randomly with probability 1%

Right: 6 errors yield 10 new nodes, 6 new weighted edges, all with weight 1



Error correction

As more *k*-mers overlap errors, # nodes & edges approach *N*

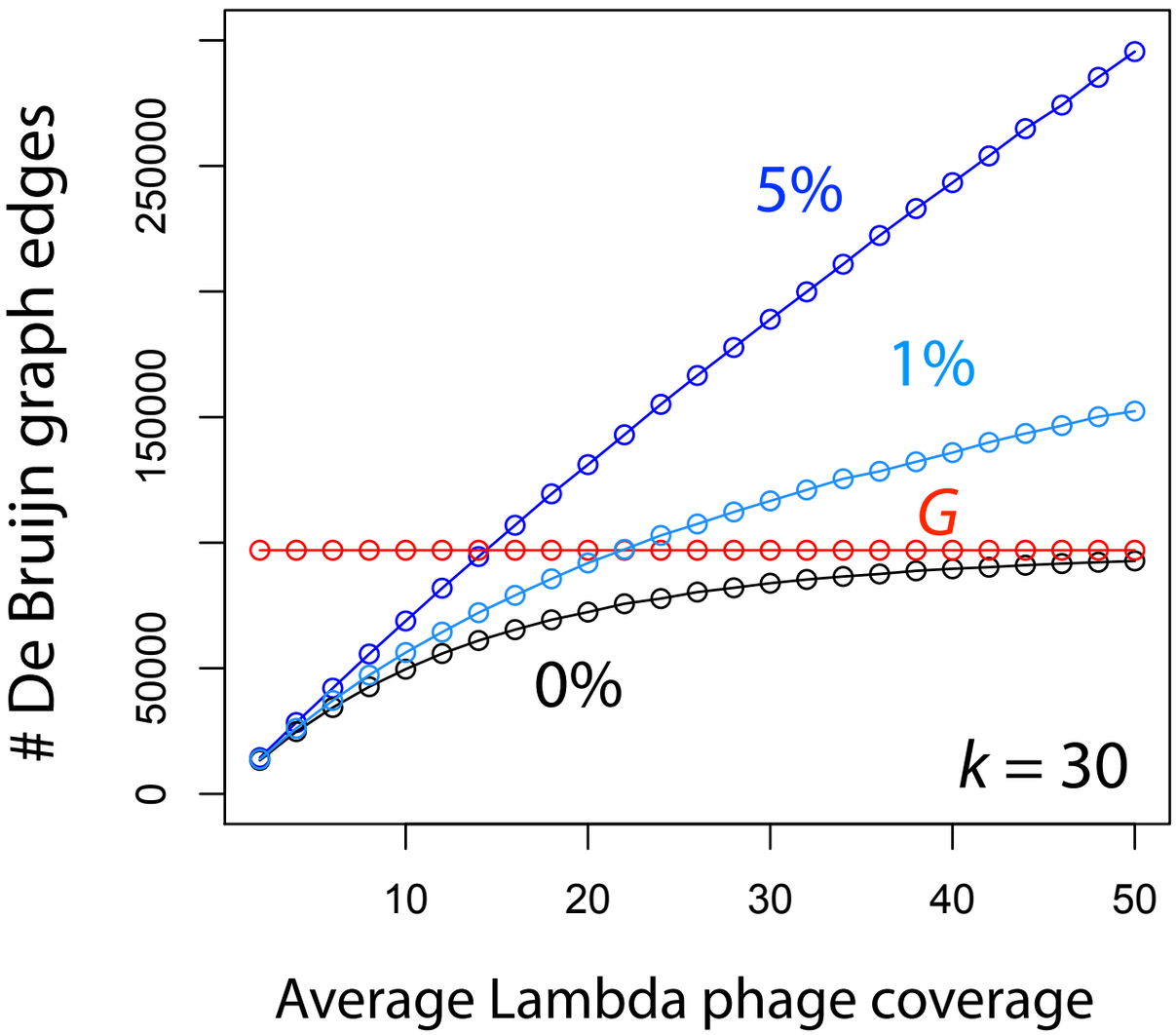


Same experiment as before, with 5% error added

Errors "push through" *G* bound

Error correction

As more *k*-mers overlap errors, # nodes & edges approach *N*



Same experiment as before, with 5% error added

Errors "push through" *G* bound

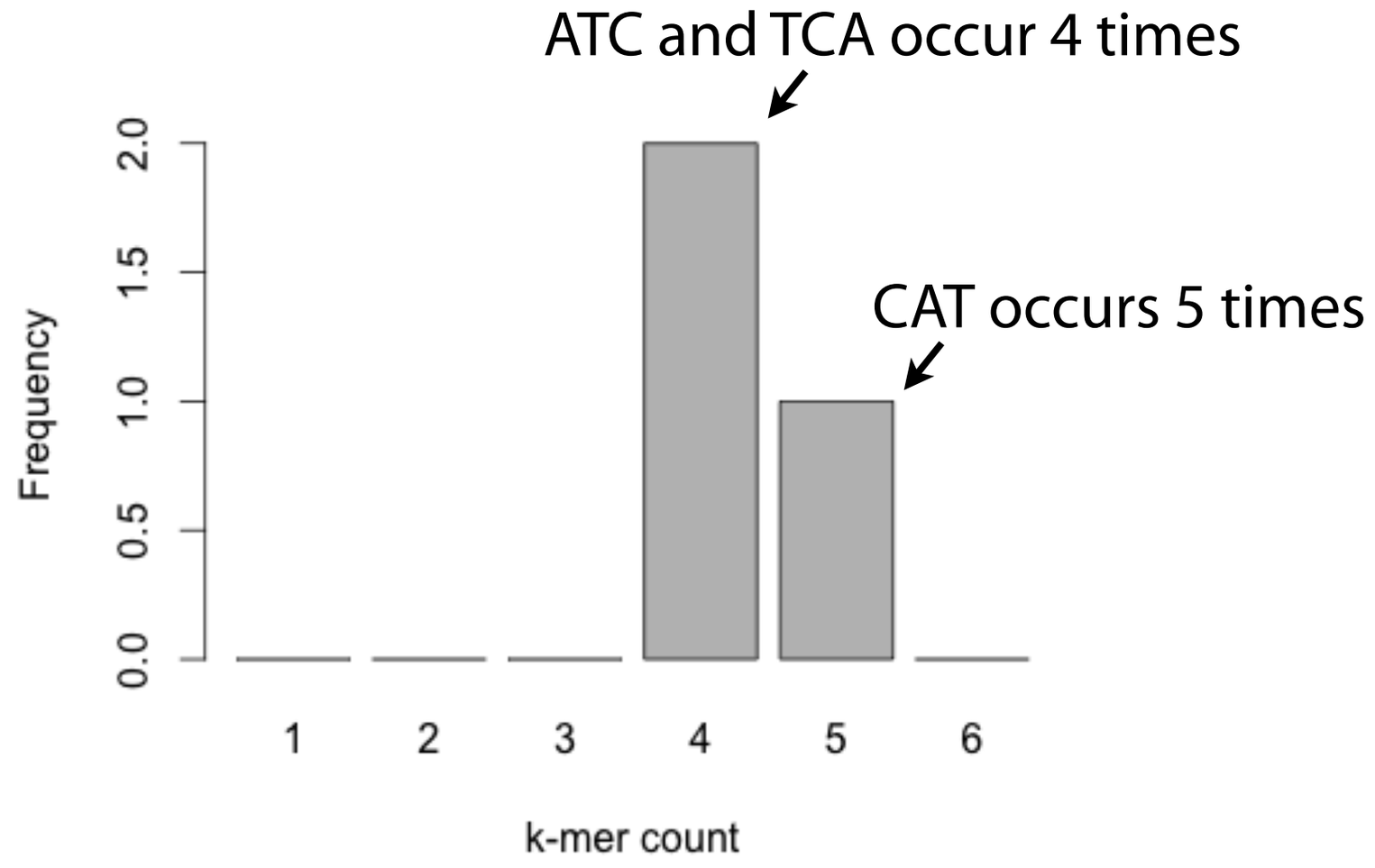
(Now with 1% error added)

Error correction

k-mer count histogram:

x axis is an integer *k*-mer count, y axis is # distinct *k*-mers with that count

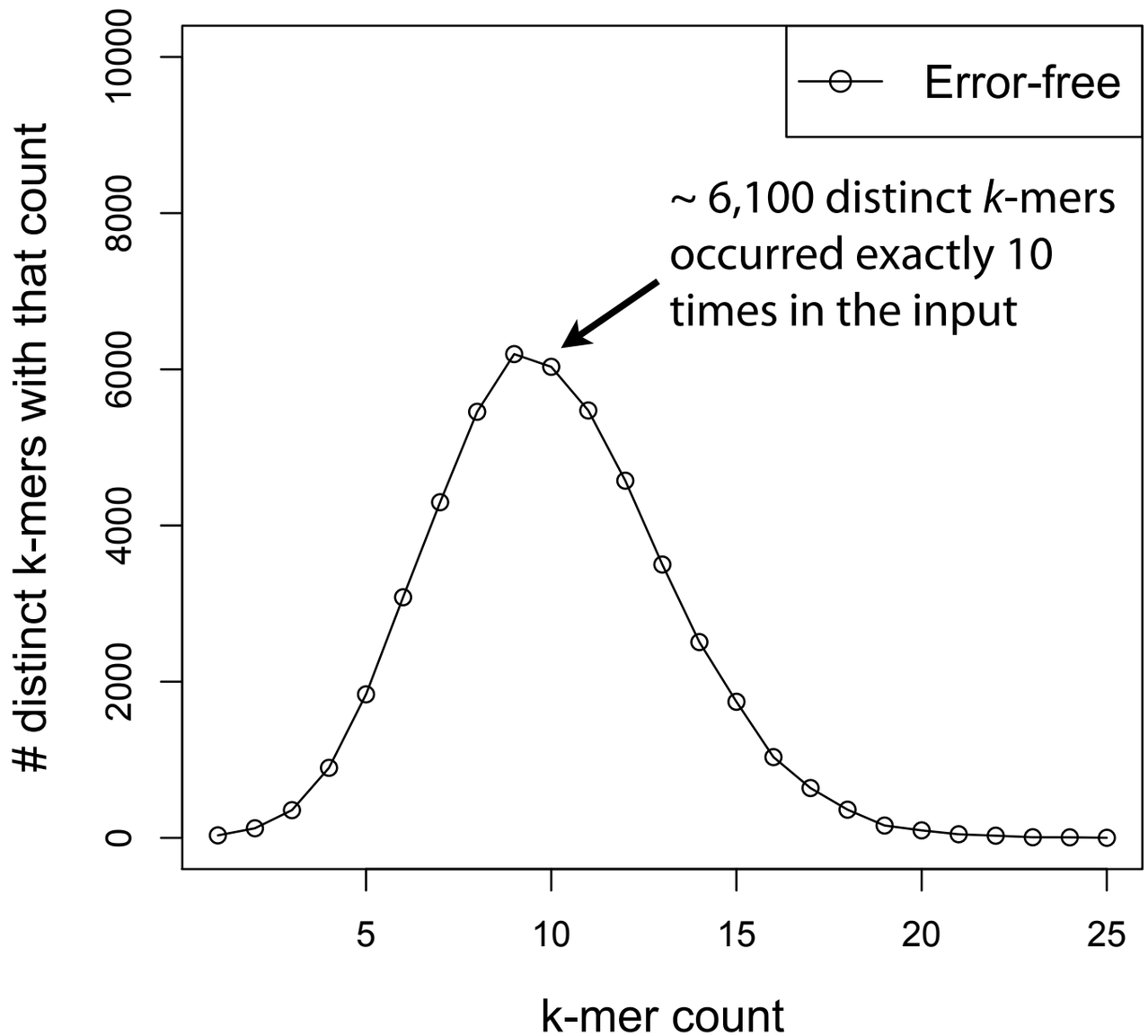
Right: such a histogram for 3-mers of CATCATCATCATCAT:



Error correction

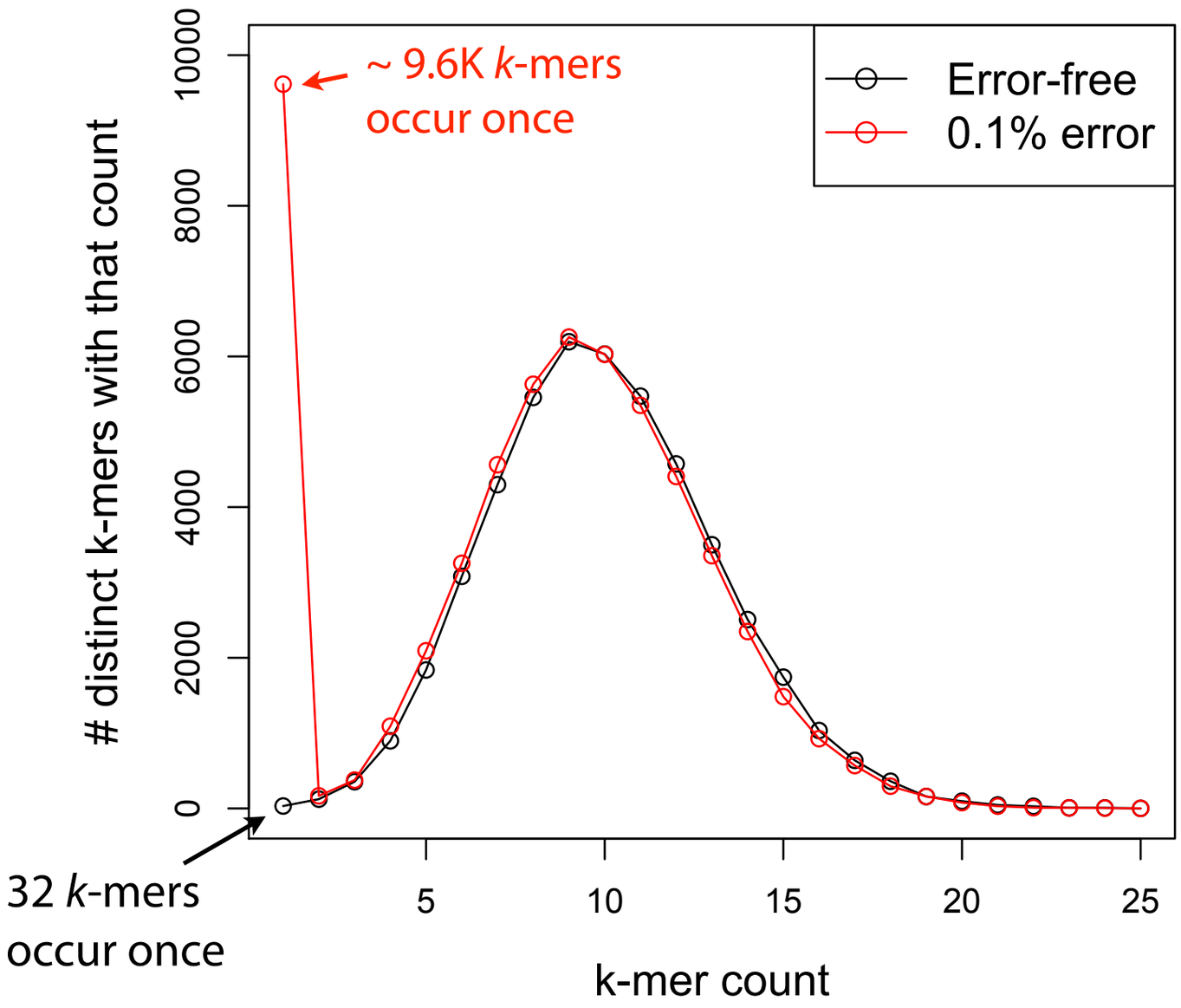
Draw 20-mers from genome randomly until each 20-mer has been drawn 10 times on average

How would the picture change for data with 1% error rate?



Error correction

k-mers with errors usually occur fewer times than error-free *k*-mers



Error correction

Idea: errors tend to turn frequent *k*-mers to infrequent *k*-mers, so corrections should do the reverse

Say each 8-mer occurs an average of ~10 times:

Read: `GCGTATTACGCGTCTGGCCT` (20 nt)

8-mers:

```
GCGTATTA: 8
CGTATTAC: 8
GTATTACG: 9
TATTACGC: 9
ATTACGCG: 10
TTACGCGT: 10
TACGCGTC: 11
ACGCGTCT: 11
CGCGTCTG: 10
GCGTCTGG: 10
CGTCTGGC: 11
GTCTGGCC: 9
TCTGGCCT: 8
```

times each 8-mer occurs in the reads.
"*k*-mer count profile"

All 8-mer counts are near average, suggesting read is error-free

Error correction

Suppose there's an error

Read: GCGTACTACGCGTCTGGCCT

```
GCGTACTA: 1
 CGTACTAC: 2
  GTACTACG: 1
   TACTACGC: 1
    ACTACGCG: 2
     CTACGCGT: 1
      TACGCGTC: 9
       ACGCGTCT: 8
        CGCGTCTG: 10
         GCGTCTGG: 10
          CGTCTGGC: 11
           GTCTGGCC: 9
            TCTGGCCT: 8
```

Below average

k-mer count profile has corresponding stretch of below-average counts

Around average

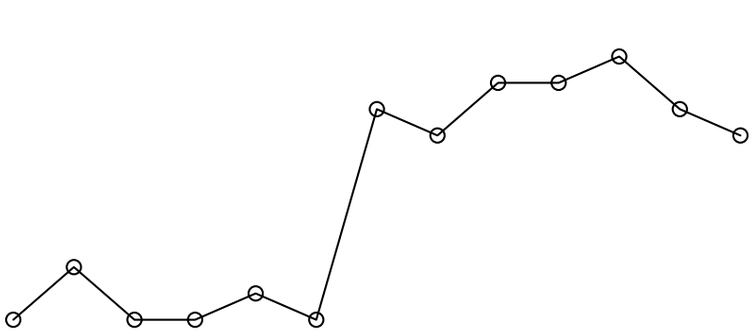
Error correction

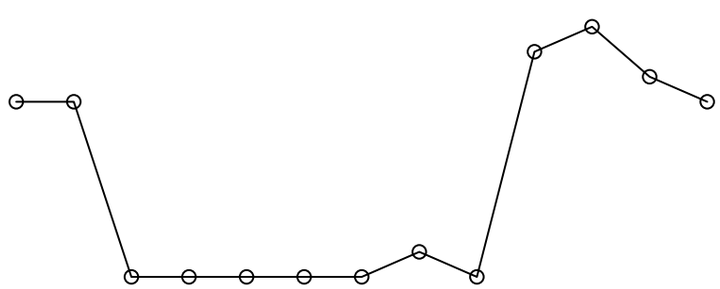
k-mer counts when errors are in different parts of the read:

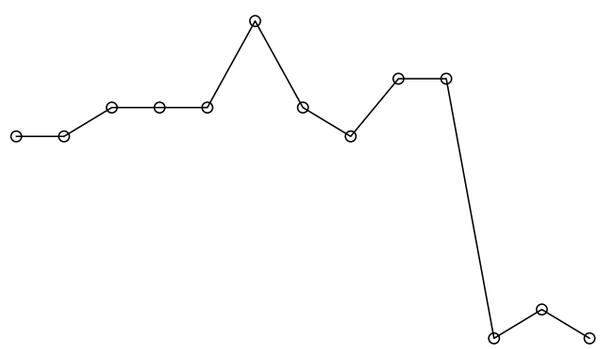
```
GCGTACTACGCGTCTGGCCT
GCGTACTA: 1
 CGTACTAC: 3
  GTACTACG: 1
   TACTACGC: 1
    ACTACGCG: 2
     CTACGCGT: 1
      TACGCGTC: 9
       ACGCGTCT: 8
        CGCGTCTG: 10
         GCGTCTGG: 10
          CGTCTGGC: 11
           GTCTGGCC: 9
            TCTGGCCT: 8
```

```
GCGTATTACACGTCTGGCCT
GCGTATTA: 8
 CGTATTAC: 8
  GTATTACA: 1
   TATTACAC: 1
    ATTACACG: 1
     TTACACGT: 1
      TACACGTC: 1
       ACACGTCT: 2
        CACGTCTG: 1
         ACGTCTGG: 1
          CGTCTGGC: 11
           GTCTGGCC: 9
            TCTGGCCT: 8
```

```
GCGTATTACGCGTCTGGTCT
GCGTATTA: 8
 CGTATTAC: 8
  GTATTACG: 9
   TATTACGC: 9
    ATTACGCG: 9
     TTACGCGT: 12
      TACGCGTC: 9
       ACGCGTCT: 8
        CGCGTCTG: 10
         GCGTCTGG: 10
          CGTCTGGT: 1
           GTCTGGTC: 2
            TCTGGTCT: 1
```

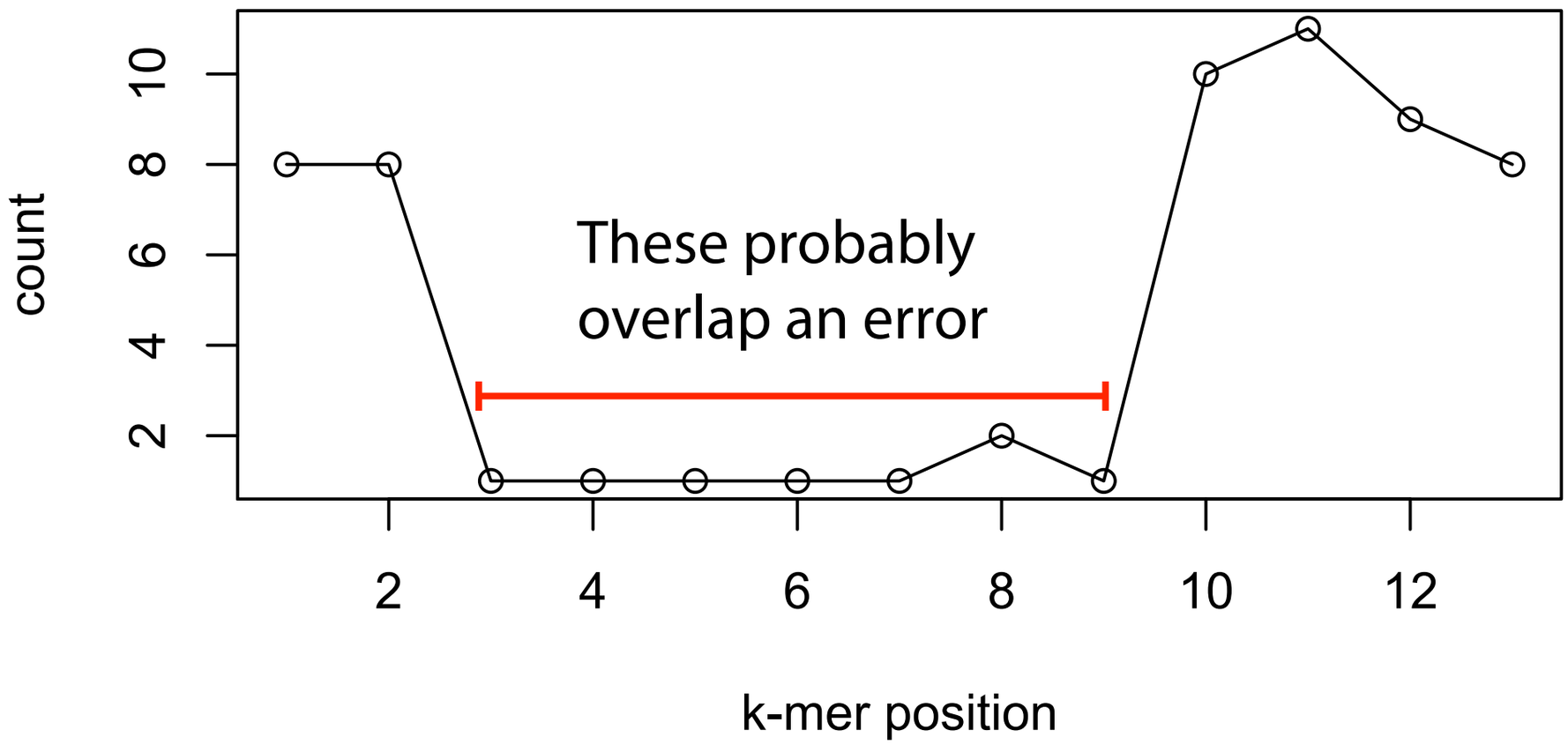






Error correction

Count profile indicates where errors are



Error correction

Simple algorithm, given a count threshold *t*:

For each read:

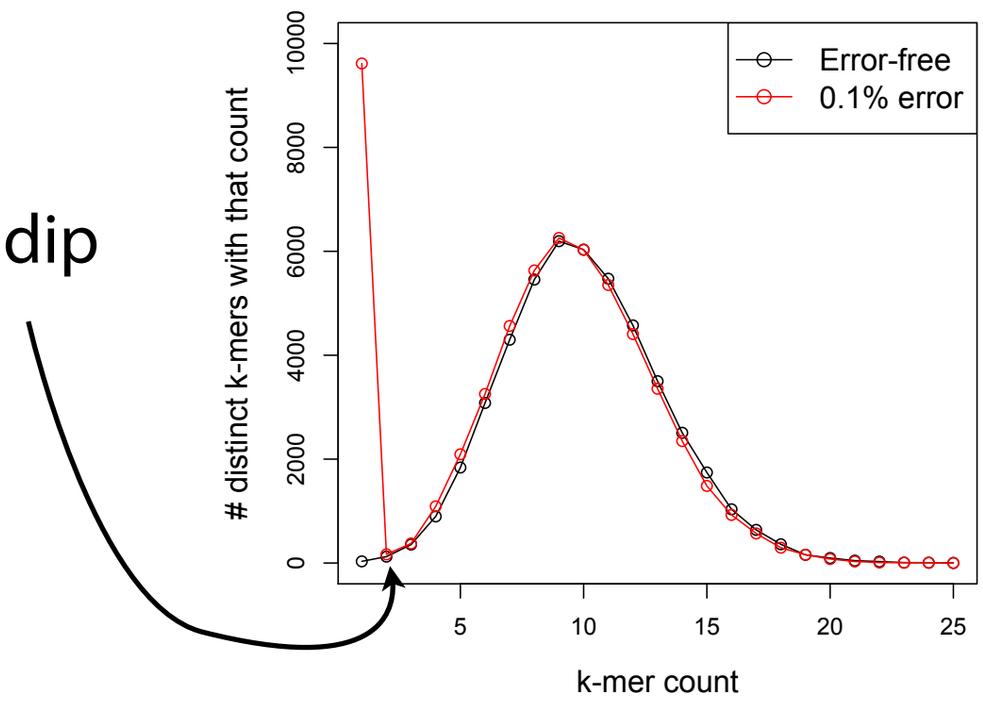
For each k-mer:

If *k*-mer count < *t*:

Examine *k*-mer's neighbors within some Hamming/edit distance.

If neighbor has count ≥ *t*, replace old *k*-mer with neighbor.

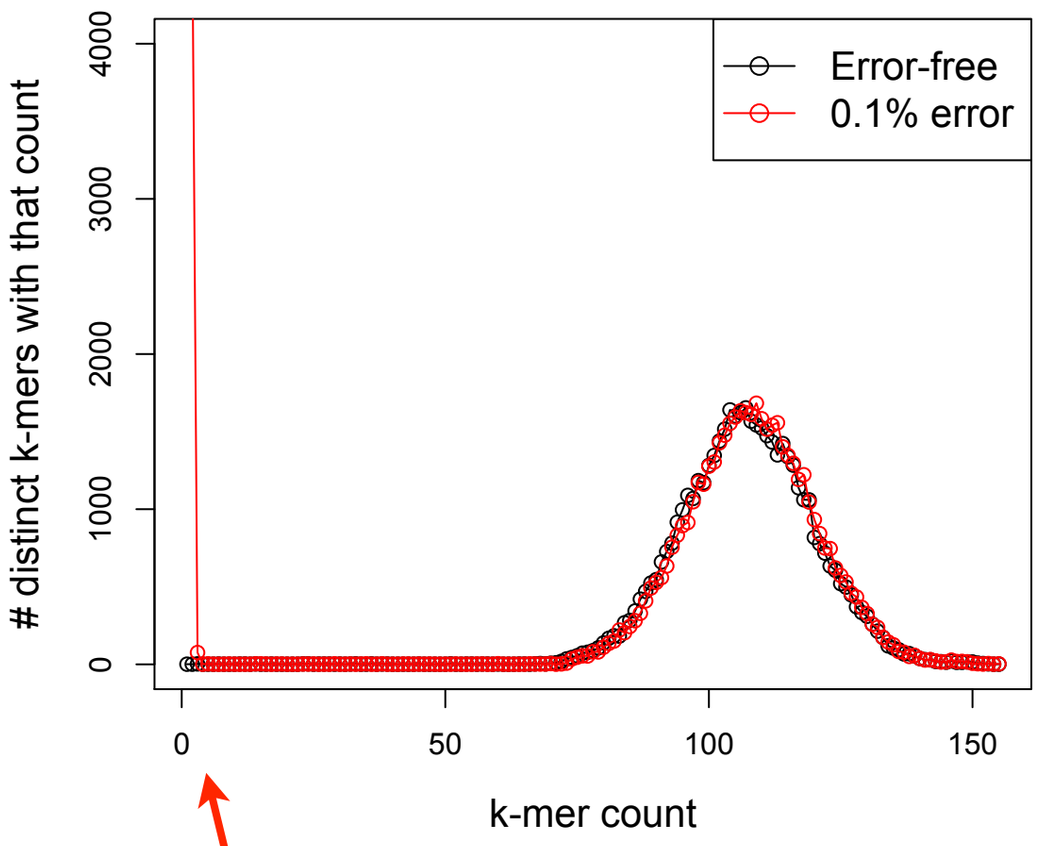
Pick *t* corresponding to dip between the peaks



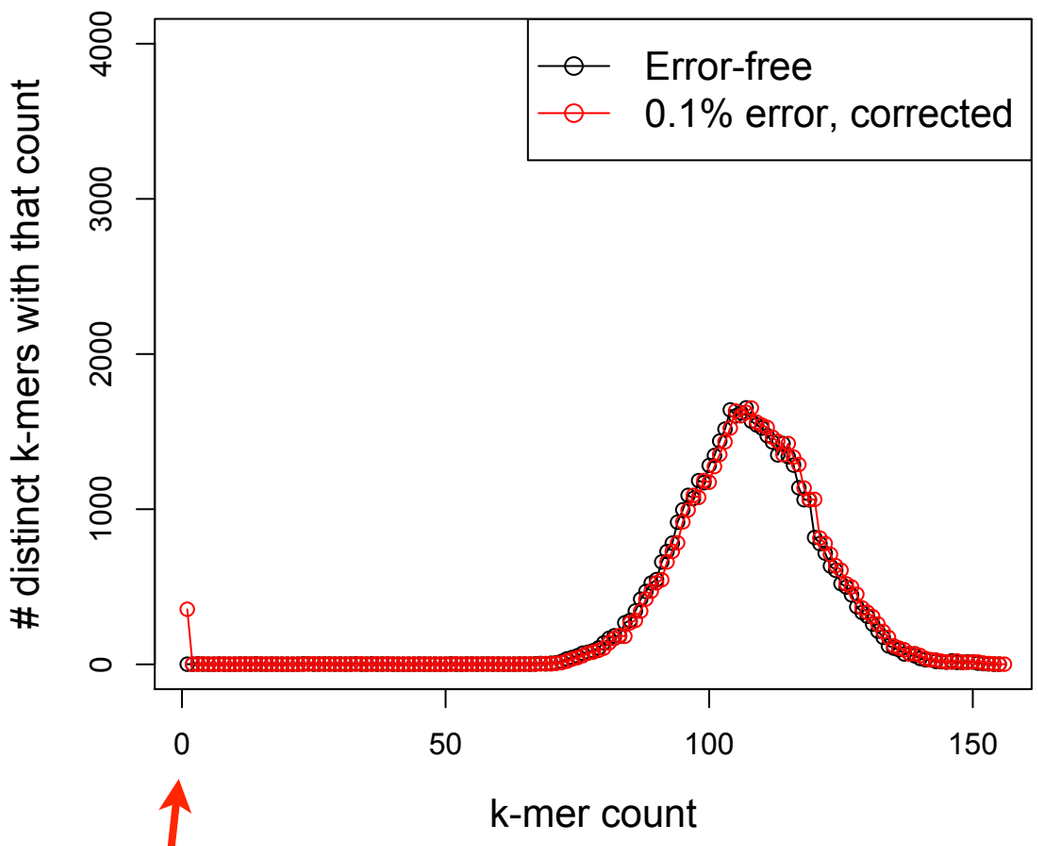
Error correction: results

Corrects 99.2% of errors in an example with 0.1% error added

Before



After



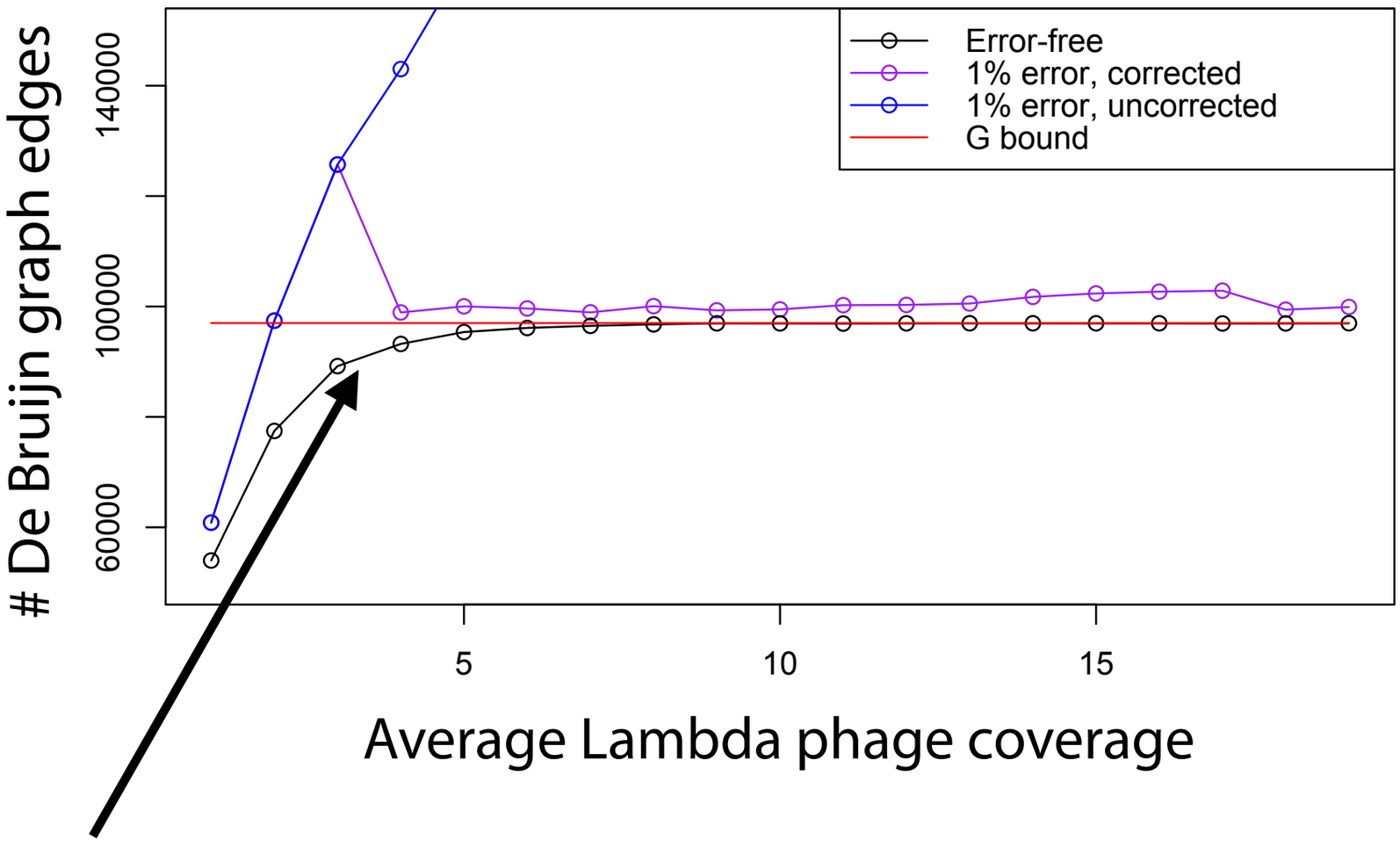
From 194K k-mers occurring exactly once to just 355

Error correction: results

Also works for 1% error...

Uncorrected, graph size is off the chart

Corrected, graph size is near G bound



...provided enough coverage to distinguish frequent/infrequent

Error correction

To work well:

Average coverage & *k* must be such that we can distinguish frequent from infrequent *k*-mers

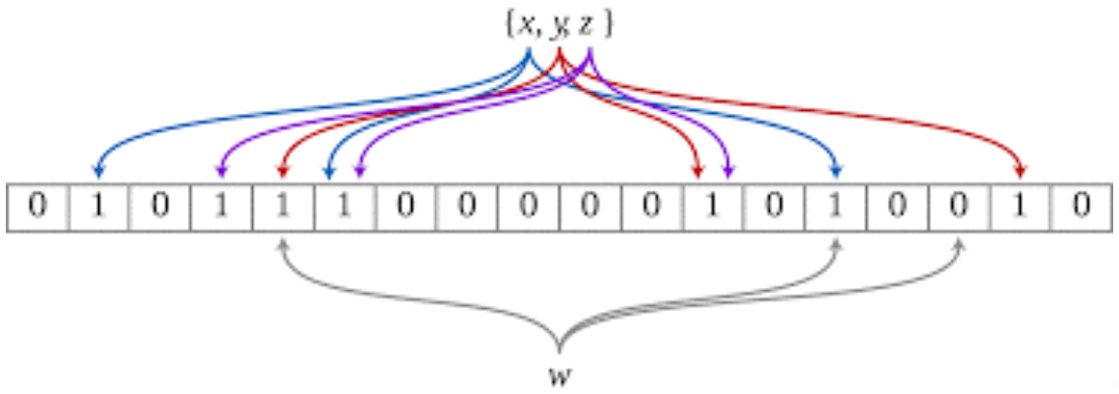
k-mer neighborhood explored must be broad enough to find frequent neighbors. Depends on error rate and *k*.

Alternately, we might give up on correcting and simply remove bad k-mers

Data structure for storing *k*-mer counts should be smaller than the De Bruijn graph

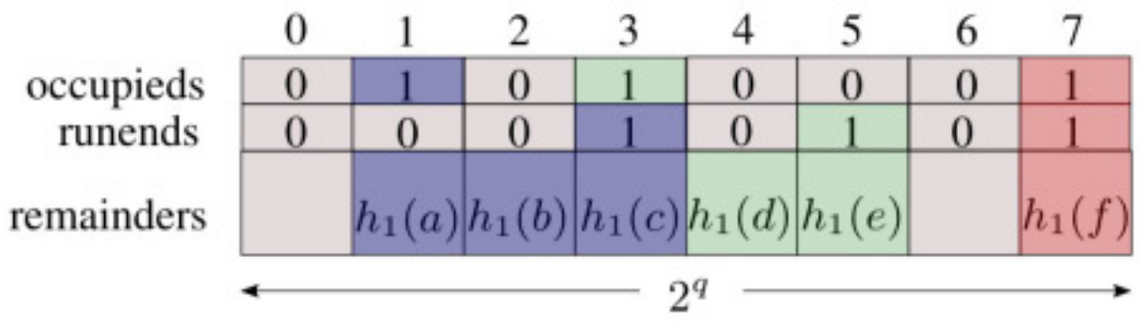
Otherwise, what's the point? 🙂

Data structures for error correction



Bloom filters

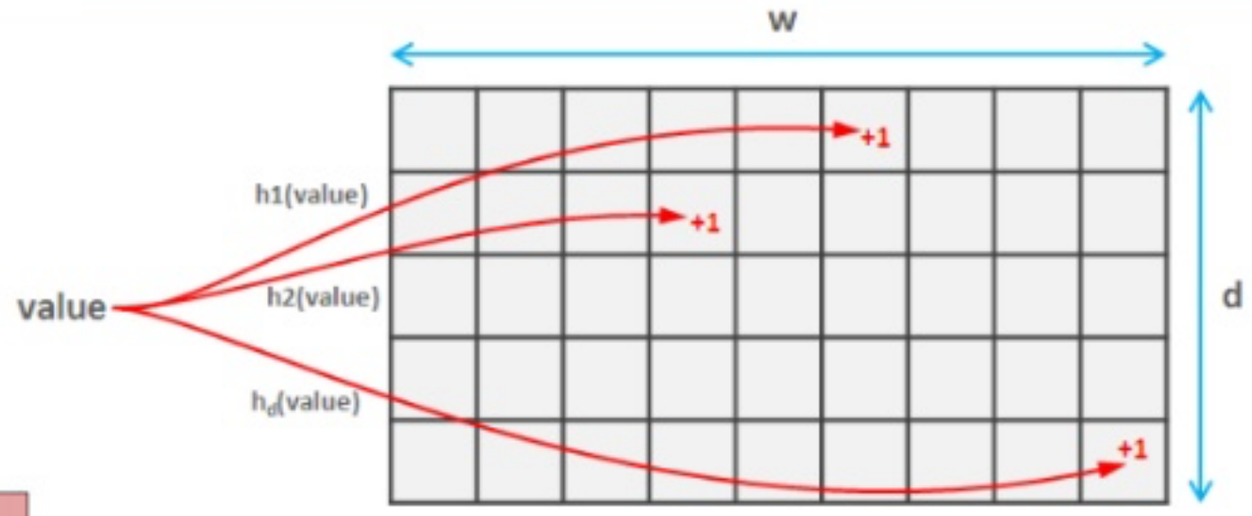
Song L, Florea L, Langmead B. Lighter: fast and memory-efficient sequencing error correction without counting. *Genome Biology*. 2014;15(11):509.



Counting quotient filters

Pandey P, Bender MA, Johnson R, Patro R. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics*. 2017; btx636.

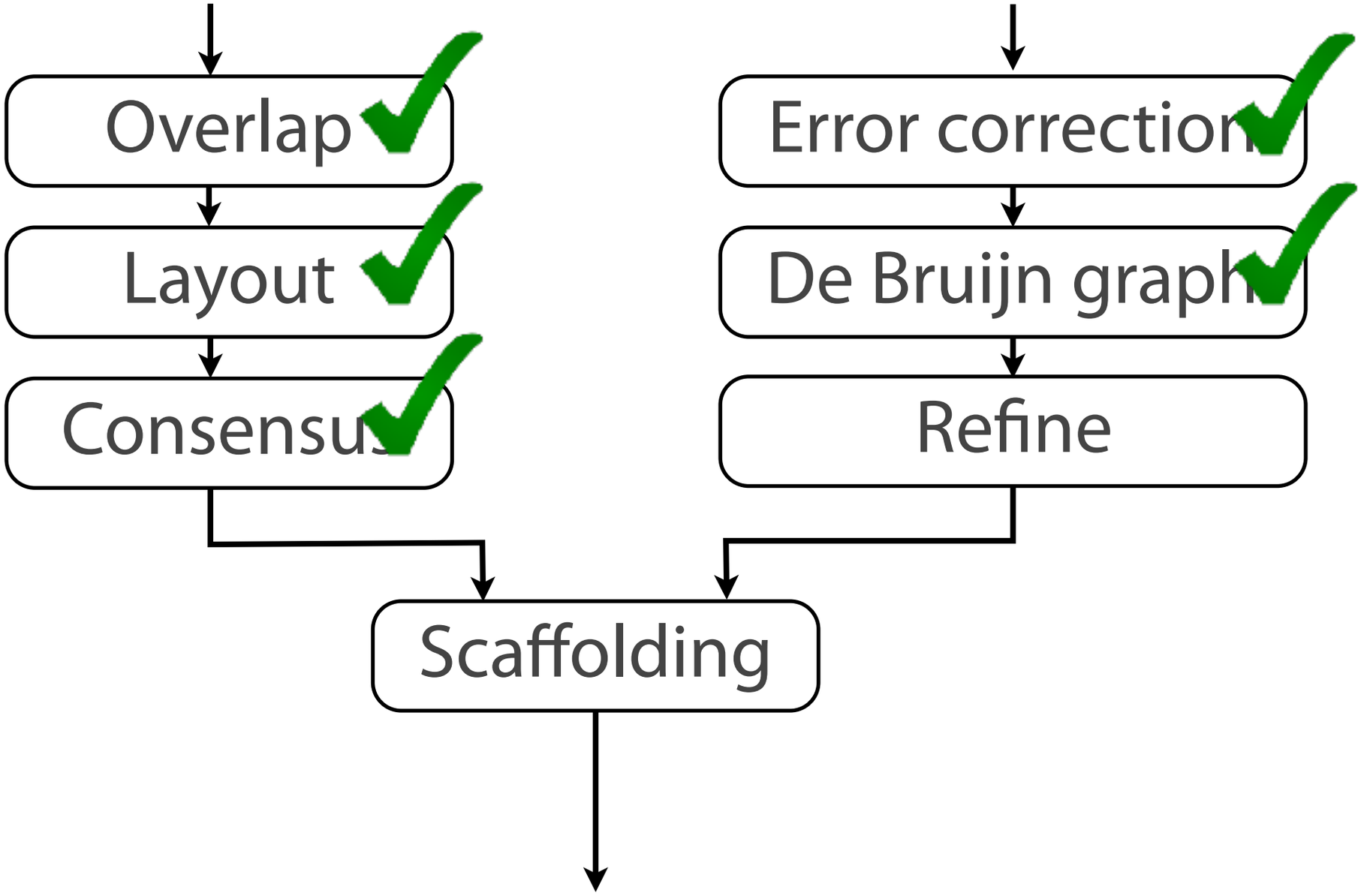
Don't need 100% accurate *k*-mer counts; just have to distinguish frequent and infrequent



CountMin sketches

Crusoe MR, Alameldin HF, Awad S, Boucher E, ..., Brown CT. The khmer software package: enabling efficient nucleotide sequence analysis. *F1000 Research*. 2015 Sep 25;4:900.

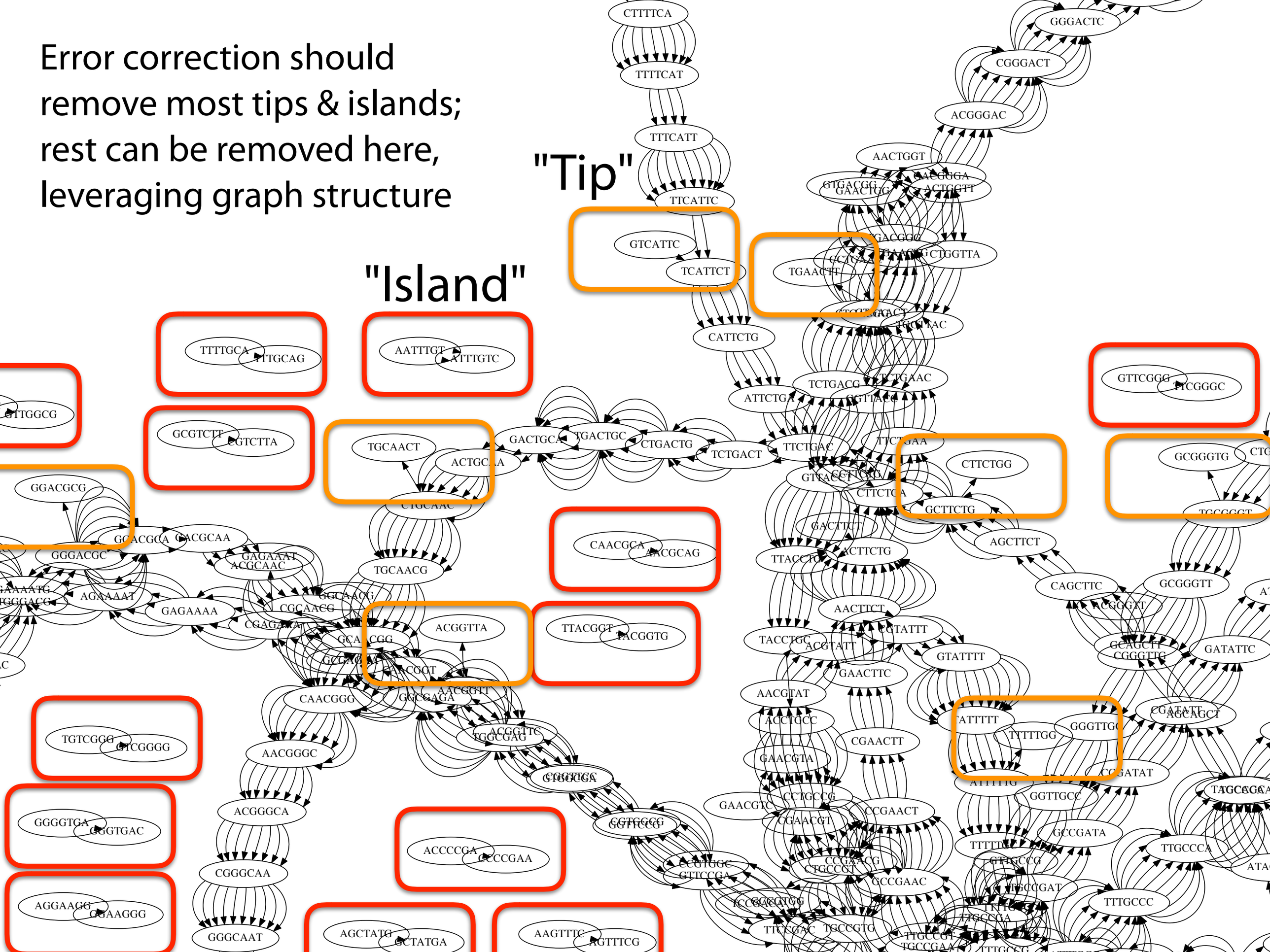
Assembly alternatives

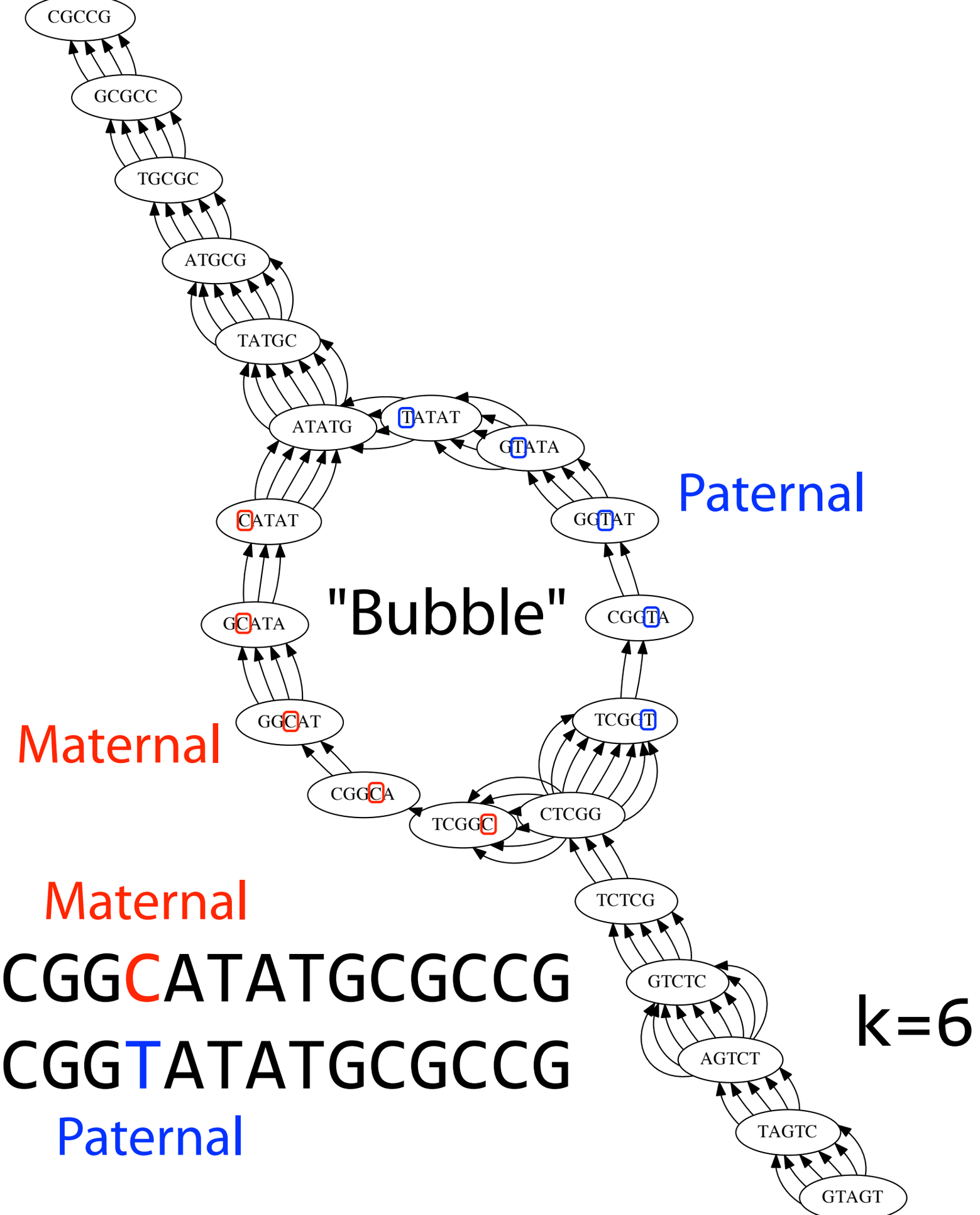


Error correction should remove most tips & islands; rest can be removed here, leveraging graph structure

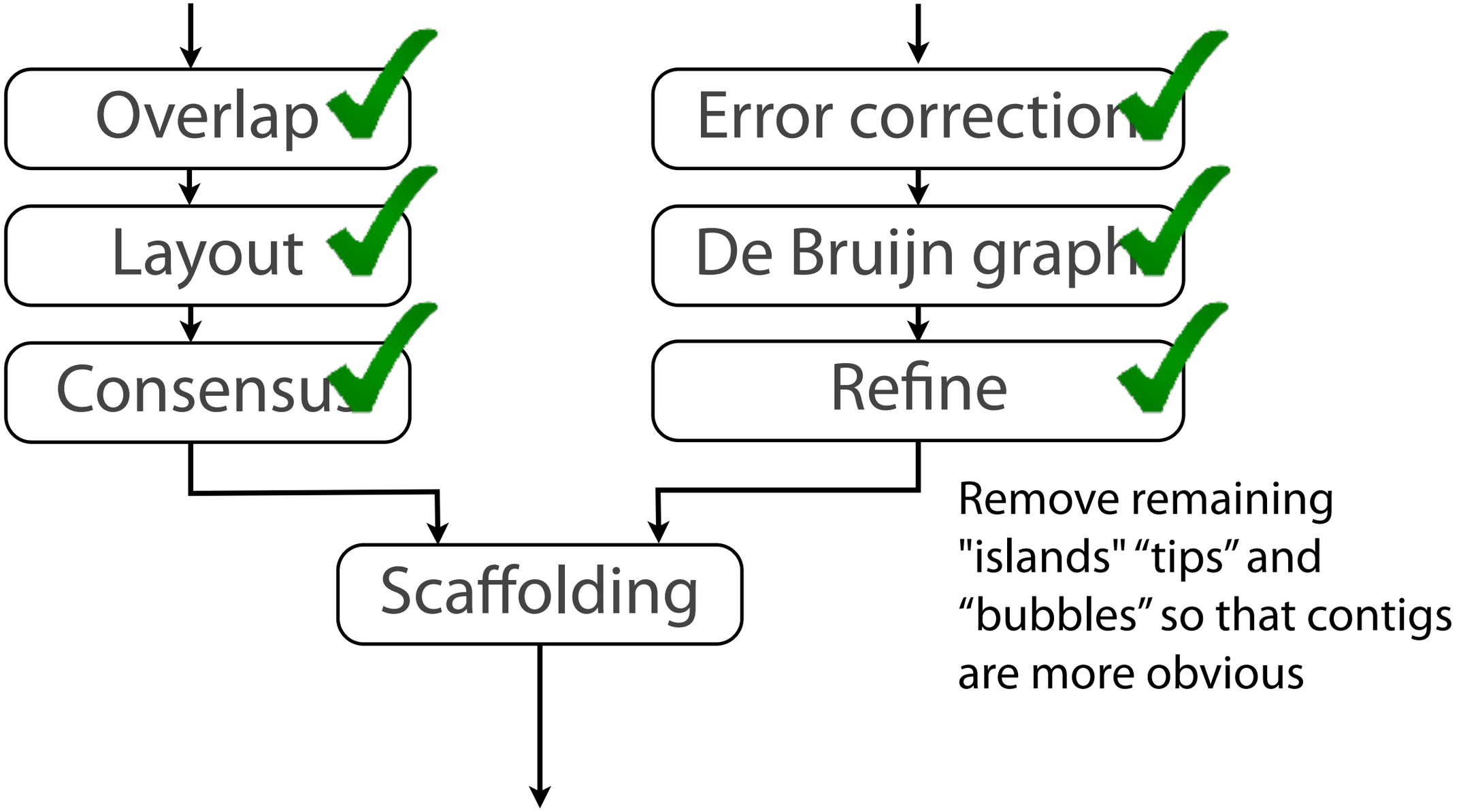
"Tip"

"Island"




CGCCG
GCGCC
TGCGC
ATGCG
TATGC
ATATG
TATAT
GTATA
GGTAT
CATAT
GCATA
CGGTA
GGCAT
TCGGT
CGGCA
TCGGC
CTCGG
TCTCG
GTCTC
AGTCT
TAGTC
GTAGT
Paternal
"Bubble"
Maternal
Maternal
GTAGTCTCGGCATATGCGCCG
GTAGTCTCGGTATATGCGCCG
Paternal
k=6

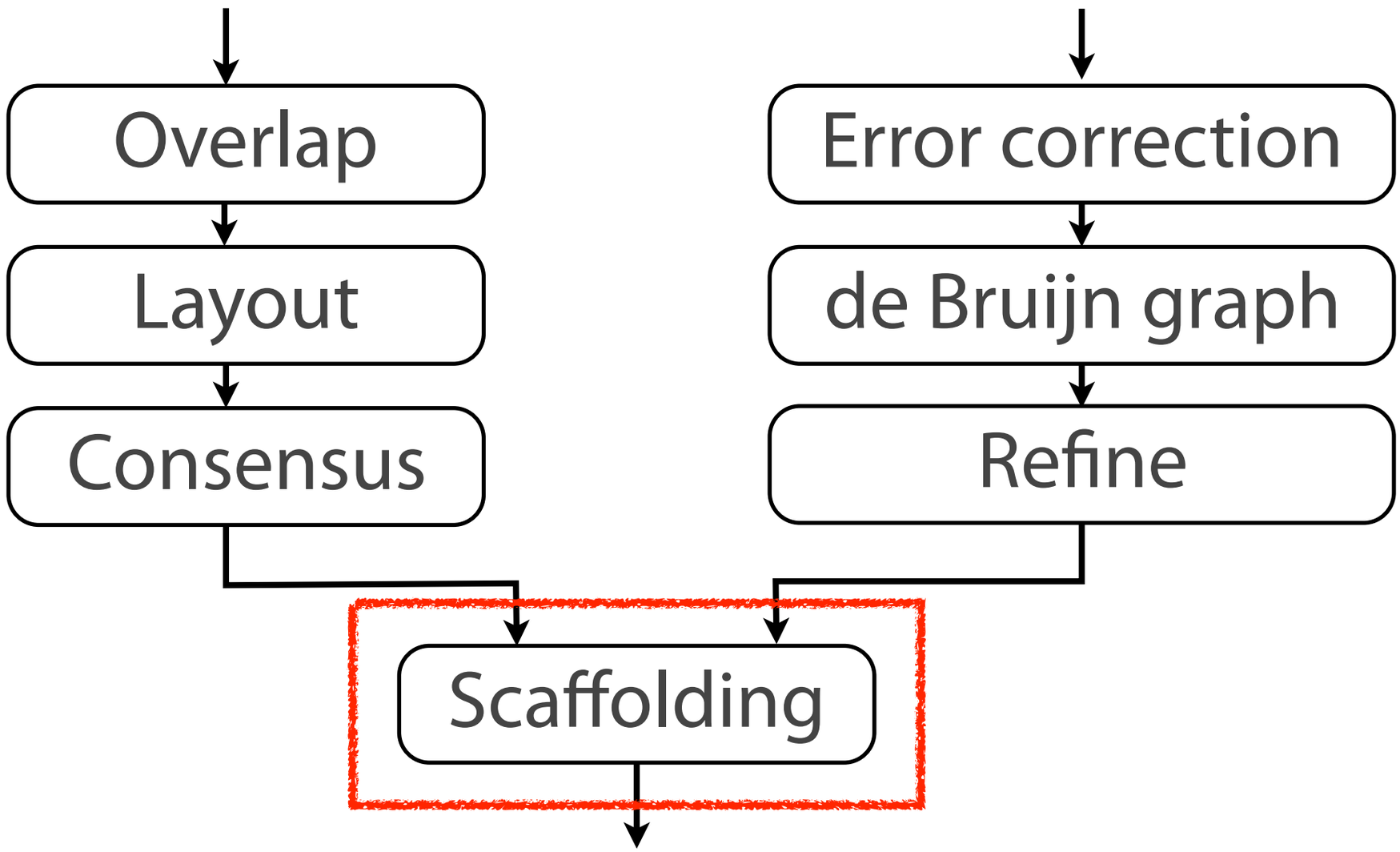
Assembly alternatives



Assembly paradigms

1: Overlap-Layout-Consensus (OLC) assembly

2: de Bruijn graph (DBG) assembly



Scaffolding

Both OLC and DBG are concerned with constructing the longest, most accurate *contigs* possible

Contig is a stretch of unambiguously assembled sequence

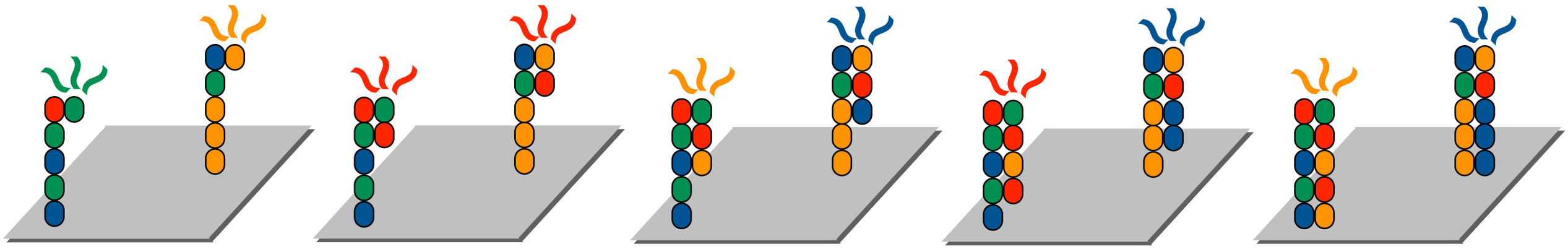
Scaffolding *orders* and *orients* contigs with respect to each other

For this we can use data from various sources, especially *paired ends*

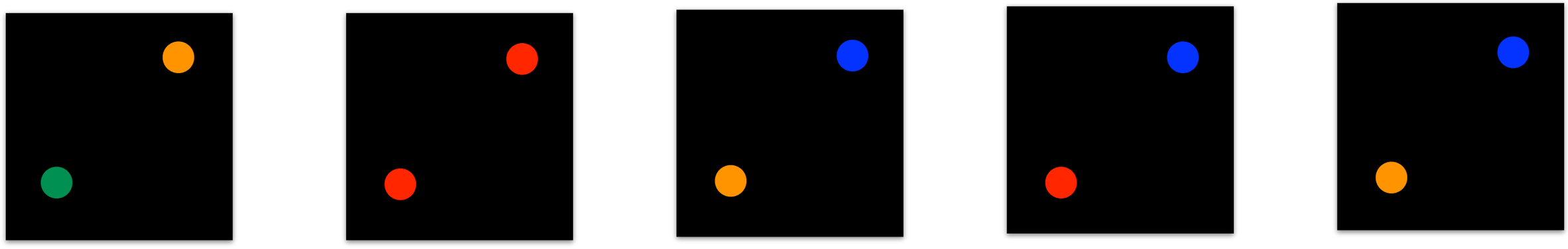
Scaffolding: paired-end sequencing

We discussed sequencing by synthesis

Process we discussed produces one contiguous read sequence



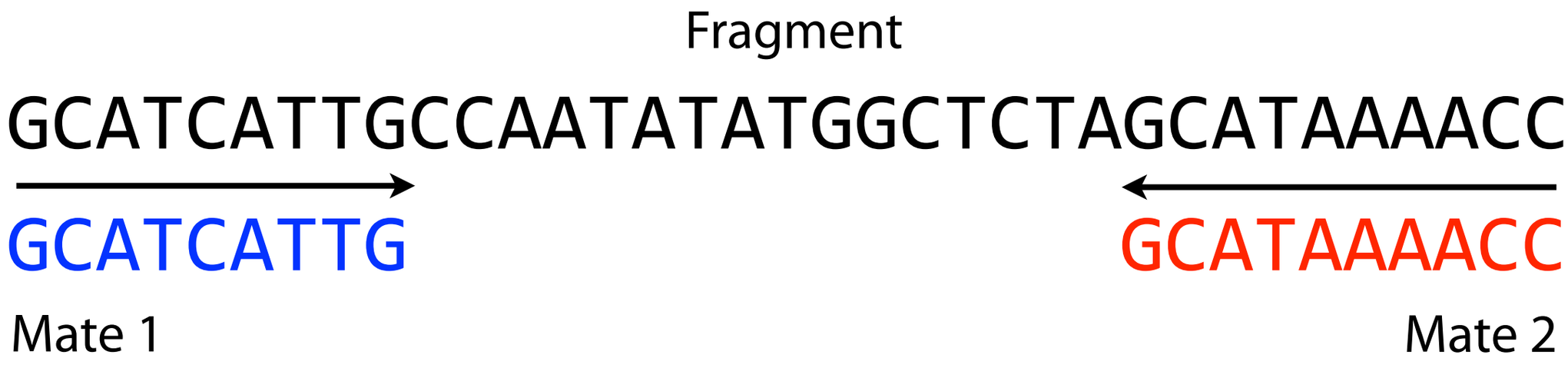
Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5



Scaffolding: paired-end sequencing

Alternative protocol produces a *pair* of reads taken from either end of a longer *fragment*

Paired reads are also called *mates* to distinguish them from the *unpaired* reads we've been discussing

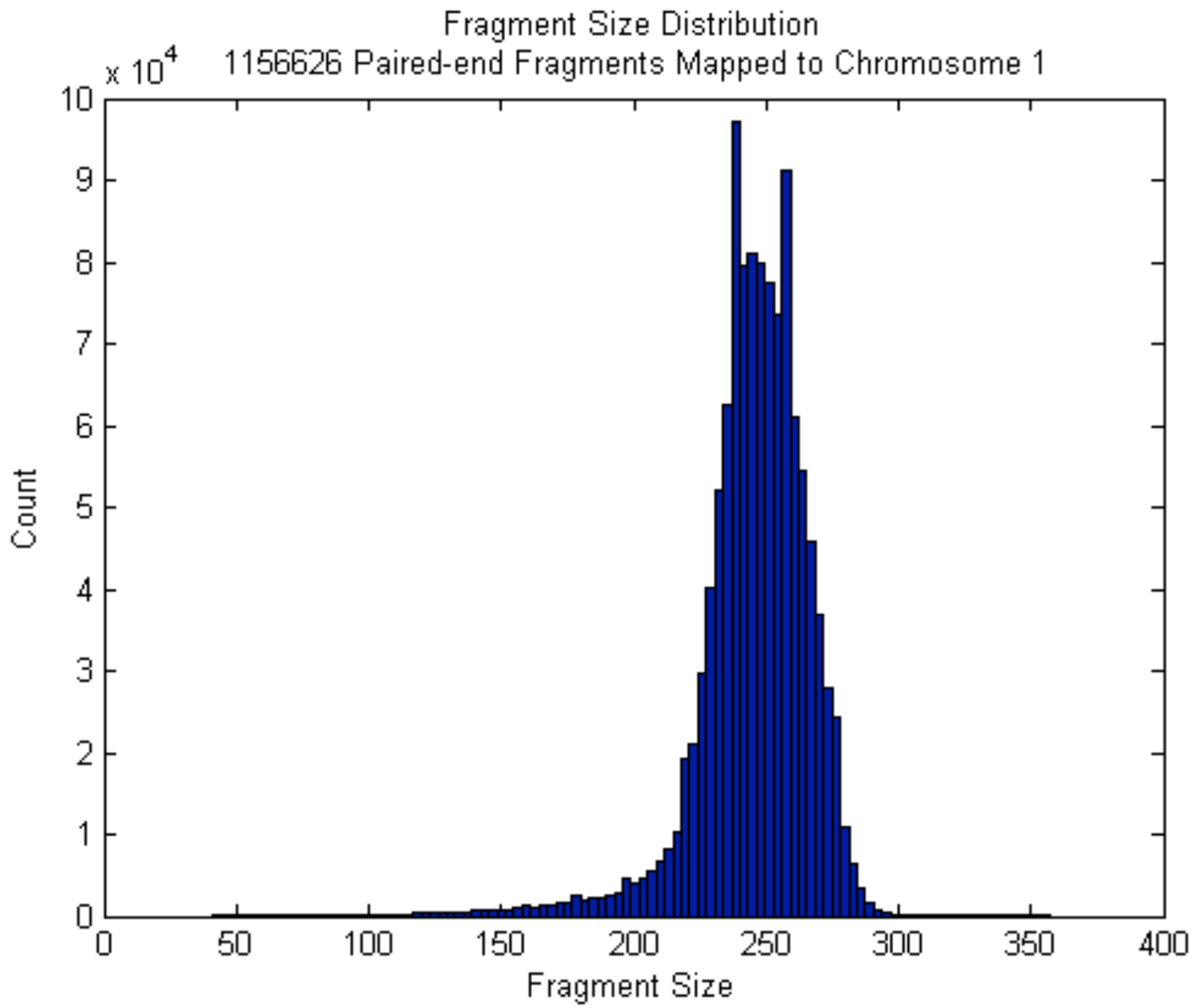


Depending on lengths, mates might overlap in the middle of the fragment

Scaffolding: paired-end sequencing

Example fragment length distribution

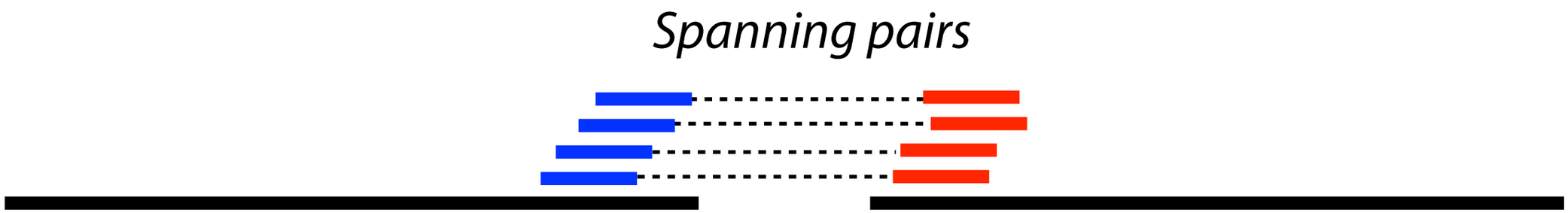
Fragments are not exactly the same length, but there's a clear peak around 250 nt, very few < 150 nt or > 300 nt



Scaffolding: paired-end sequencing

Say we have a collection of pairs and we assemble them as usual

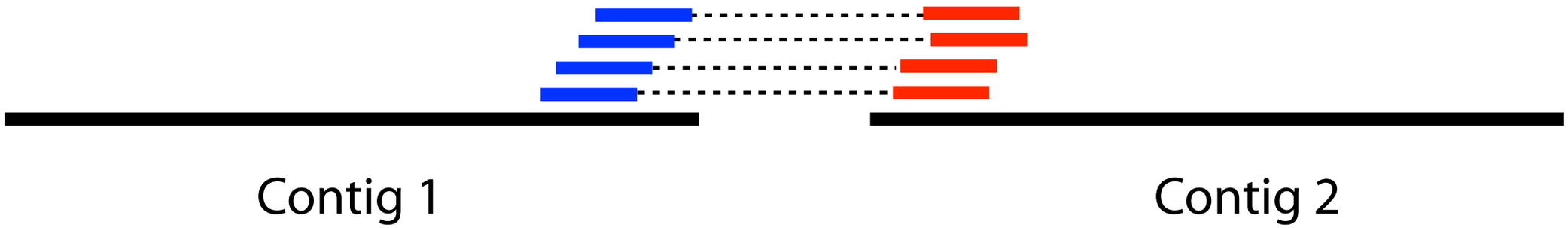
Assembly yields two contigs:



...and we discover that some of the mates at one edge of contig 1 are paired with mates in contig 2

Call these *spanning pairs*

Scaffolding: paired-end sequencing

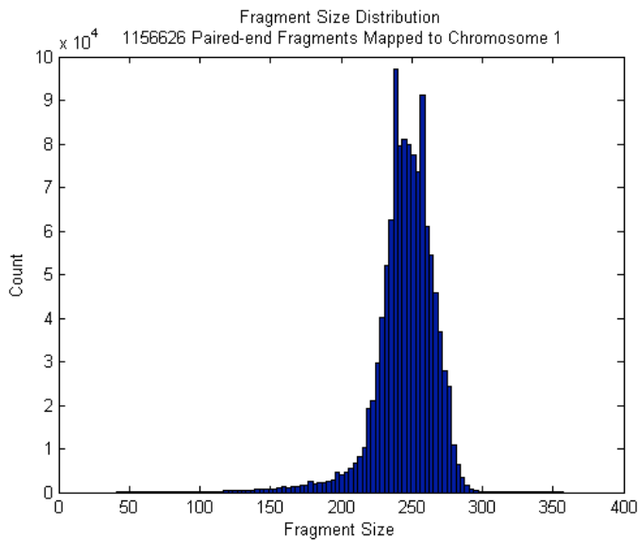


What does this tell us?

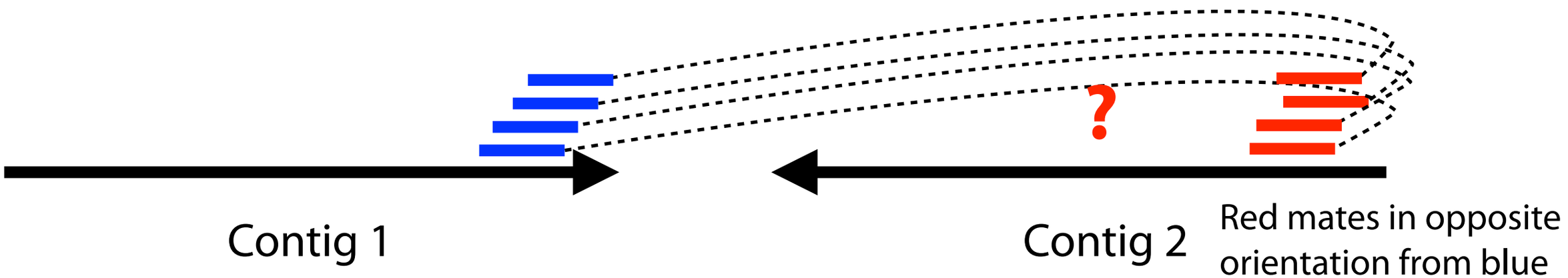
Contig 1 is close to contig 2 in the genome

In fact, we can *estimate distance between contigs* using what we know about fragment length distribution

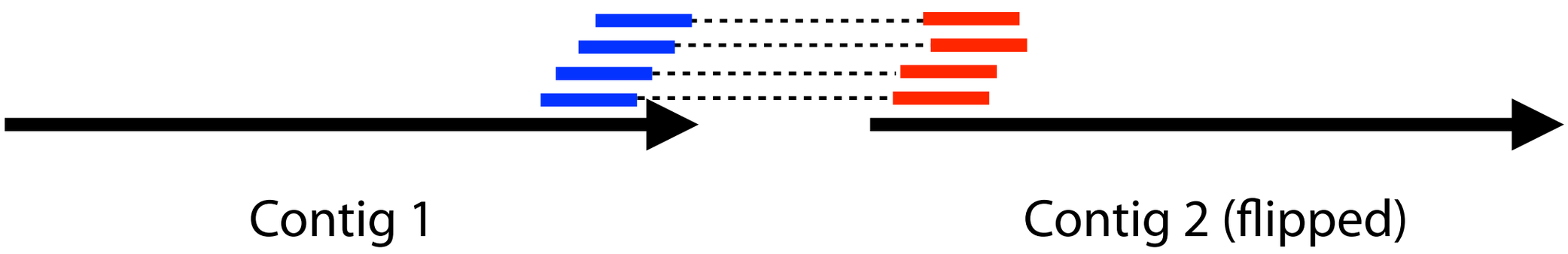
The more spanning pairs we have, the better our estimate



Scaffolding: paired-end sequencing



What does the picture look like if contigs 1 and 2 are close, but we assembled contig 2 "backwards" (i.e. reverse complemented)



Pairs also tell us about contigs' relative *orientation*

Scaffolding

Scaffolding output: collection of *scaffolds*, where a scaffold is a collection of contigs related to each other with high confidence using pairs

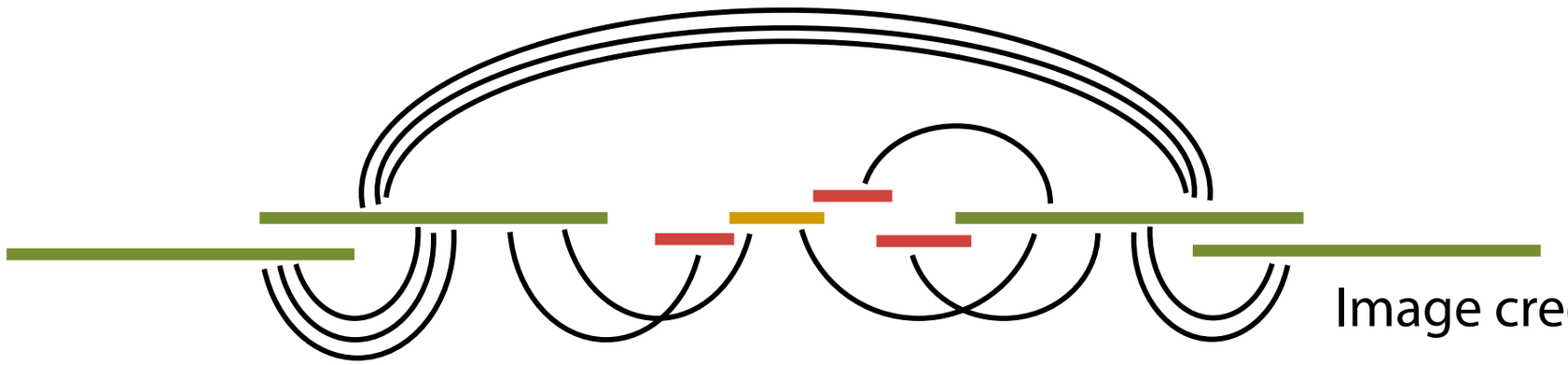


Image credit: Mike Schatz