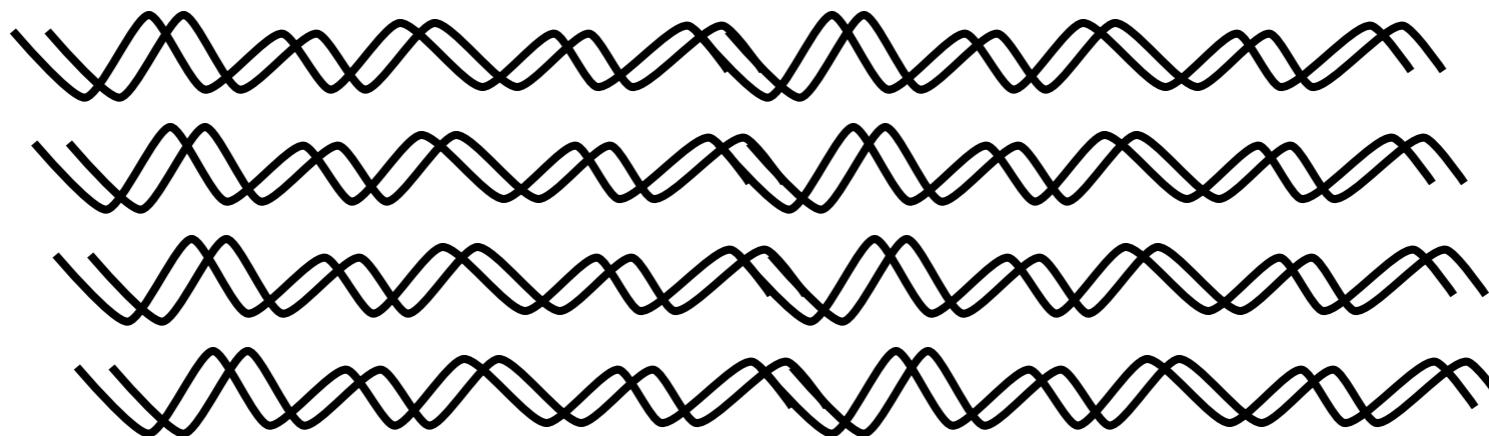


CSE 549: Genome Assembly

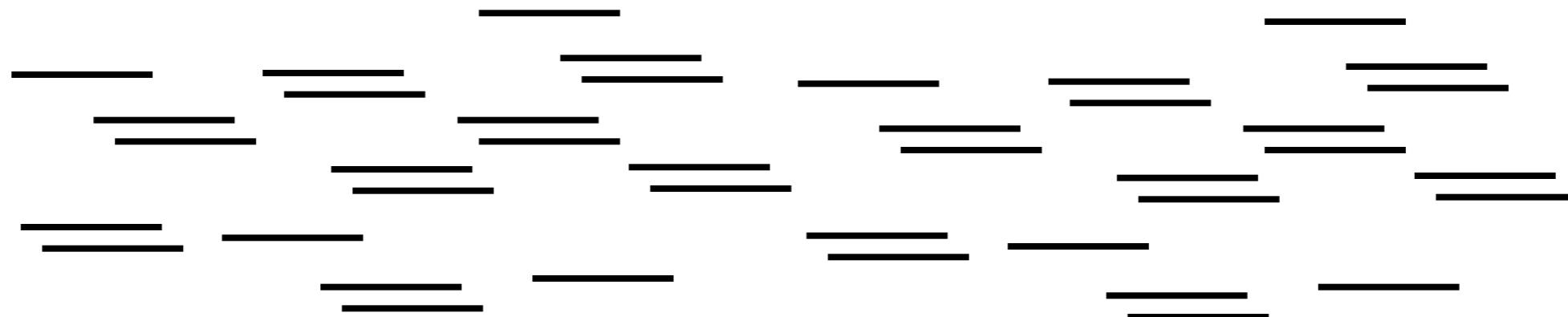
Intro & OLC

Shotgun Sequencing

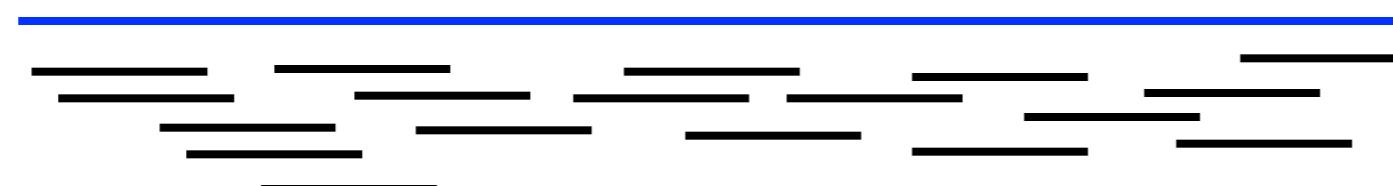
Many copies
of the DNA



Shear it, randomly breaking them into many small pieces,
read ends of each:



Assemble into original genome:



Milestones in Genome Assembly

Nature Vol. 265 February 24 1977

487

articles

Nucleotide sequence of bacteriophage Φ X174 DNA

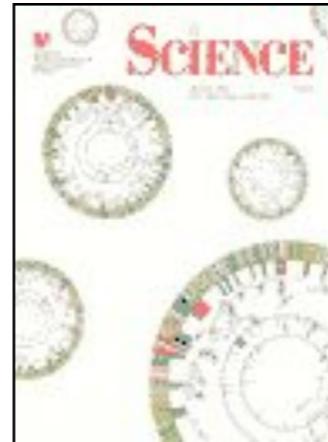
F. Sanger, G. M. Air*, B. G. Barrell, N. L. Brown*, A. R. Coulson, J. C. Fiddes, C. A. Hutchison III, P. M. Slocombe* & M. Smith*

MRC Laboratory of Molecular Biology, Hills Road, Cambridge CB2 2QH, UK

A DNA sequence for the genome of bacteriophage Φ X174 of approximately 5,375 nucleotides has been determined using the combined "sequencing by synthesis" method. The sequence identifies many of the features responsible for the production of the proteins of the nine known genes of the organism, including initiation and termination sites for the proteins and RNAs. Two pairs of genes are coded by the same region of DNA using different reading frames.

The genome of bacteriophage Φ X174 is a single-stranded, circular DNA of approximately 5,300 nucleotides coding for nine known genes. The order of genes, as determined by genetic techniques¹⁻³, is A-B-C-D-E-J-F-G-H. Genes F, G and H code for structural proteins of the virus capsid, and gene J (as defined by sequence work) codes for a small basic protein.

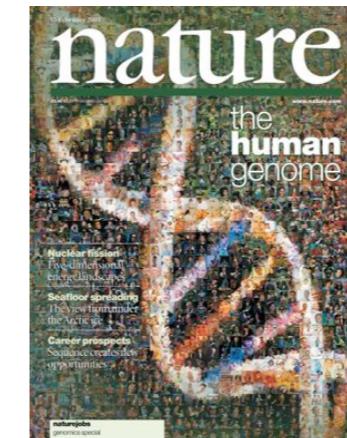
This paper describes the sequence analysis of the genome of bacteriophage Φ X174.



1977. Sanger et al.
1st Complete Organism
5375 bp

1995. Fleischmann et al.
1st Free Living Organism
TIGR Assembler. 1.8Mbp

1998. C.elegans SC
1st Multicellular Organism
BAC-by-BAC Phrap. 97Mbp



2000. Myers et al.
1st Large WGS Assembly.
Celera Assembler. 116 Mbp

2001. Venter et al., IHGSC
Human Genome
Celera Assembler/GigaAssembler. 2.9 Gbp

2010. Li et al.
1st Large SGS Assembly.
SOAPdenovo 2.2 Gbp

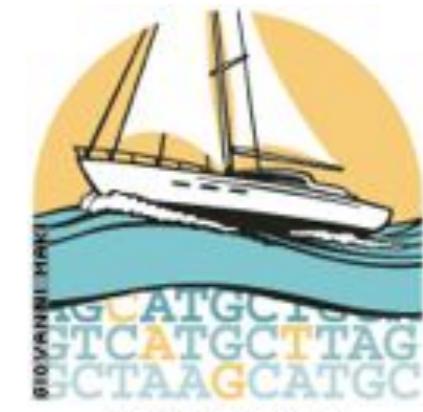
Like Dickens, we must computationally reconstruct a genome from short fragments

Assembly Applications

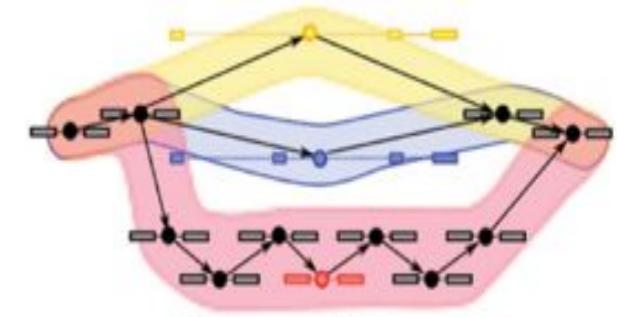
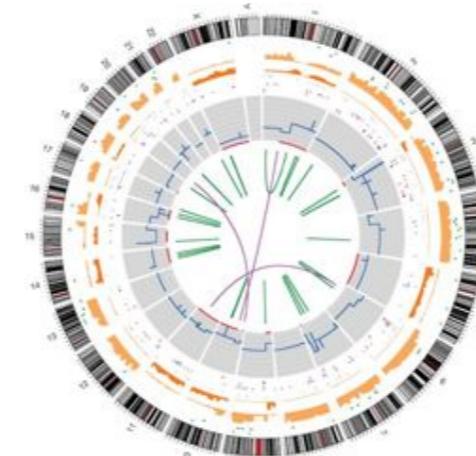
- Novel genomes



- Metagenomes

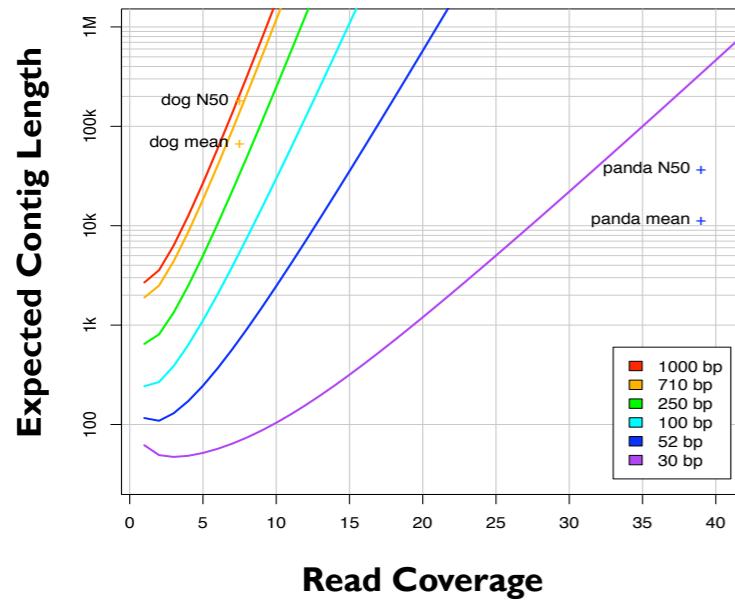


- Sequencing assays
 - Structural variations
 - Transcript assembly
 - ...



Ingredients for a good assembly

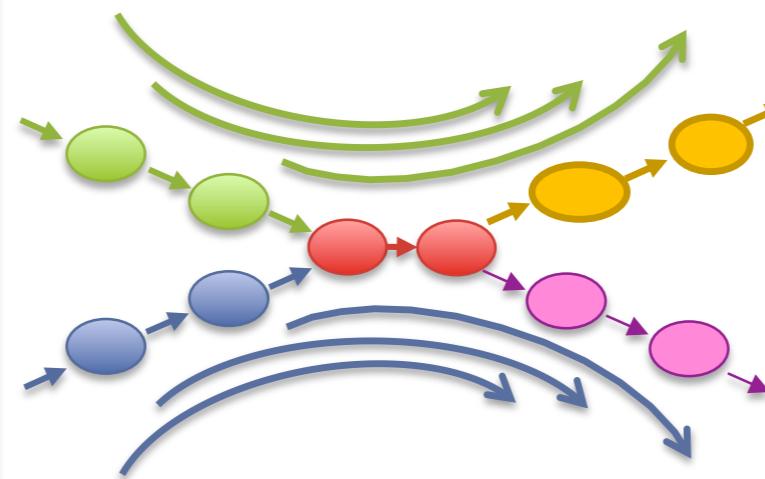
Coverage



High coverage is required

- Oversample the genome to ensure every base is sequenced with long overlaps between reads
- Biased coverage will also fragment assembly

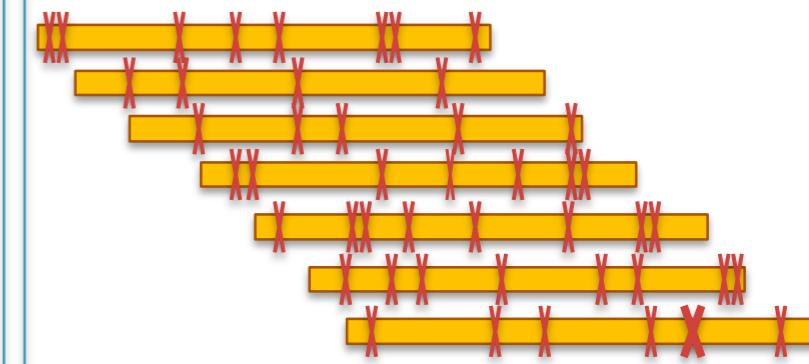
Read Length



Reads & mates must be longer than the repeats

- Short reads will have **false overlaps** forming hairball assembly graphs
- With long enough reads, assemble entire chromosomes into contigs

Quality



Errors obscure overlaps

- Reads are assembled by finding kmers shared in pair of reads
- High error rate requires very short seeds, increasing complexity and forming assembly hairballs

Current challenges in *de novo* plant genome sequencing and assembly

Schatz MC, Witkowski, McCombie, WR (2012) *Genome Biology*. 12:243

Assembly

Whole-genome “shotgun” sequencing starts by copying and fragmenting the DNA

(“Shotgun” refers to the random fragmentation of the whole genome; like it was fired from a shotgun)

Input: GGCGTCTATATCTGGCTCTAGGCCCTCATTTTT

Copy: GGCGTCTATATCTGGCTCTAGGCCCTCATTTTT
GGCGTCTATATCTGGCTCTAGGCCCTCATTTTT
GGCGTCTATATCTGGCTCTAGGCCCTCATTTTT
GGCGTCTATATCTGGCTCTAGGCCCTCATTTTT

Fragment: GGCGTCTA TATCTGG CTCTAGGCCCTC ATTTTTT
GGC GTCTATAT CTCGGCTCTAGGCCCTCA TTTTTT
GGCGTC TATATCT CGGCTCTAGGCCCT CATTTTTT
GGCGTCTAT ATCTGGCTTAG GCCCTCA TTTTTT

Assembly

Assume sequencing produces such a large # fragments that almost all genome positions are *covered* by many fragments...

Reconstruct
this

CTAGGCCCTCAATT
CTCTAGGCCCTCAATT
GGCTCTAGGCCCTCAATT
CTCGGCTCTAGCCCCTCAATT
TATCTCGACTCTAGGCCCTCA
TATCTCGACTCTAGGCC
TCTATATCTGGCTCTAG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT

From these

→ GGCGTCTATATCTGGCTCTAGGCCCTCAATT

Assembly

...but we don't know what came from where

Reconstruct
this

CTAGGCCCTCAATTTT
GGCGTCTATATCT
CTCTAGGCCCTCAATTTT
TCTATATCTGGCTCTAGG
GGCTCTAGGCCCTCATTTTT
CTCGGCTCTAGCCCCTCATT
TATCTCGACTCTAGGCCCTCA
GGCGTCGATATCT
TATCTCGACTCTAGGCC
GGCGTCTATATCTCG

From these

→ GGCGTCTATATCTGGCTCTAGGCCCTCATTTTT

Assembly

Key term: *coverage*. Usually it's short for *average coverage*: the average number of reads covering a position in the genome.

CTAGGCCCTCAATTTT	
CTCTAGGCCCTCAATTTT	
GGCTCTAGGCCCTCATTTTT	
CTCGGCTCTAGCCCCTCATTTT	
TATCTCGACTCTAGGCCCTCA	177 nucleotides
TATCTCGACTCTAGGCC	
TCTATATCTGGCTCTAGG	
GGCGTCTATATCTCG	
GGCGTCGATATCT	
GGCGTCTATATCT	
GGCGTCTATATCTGGCTCTAGGCCCTCATTTTT	35 nucleotides

$$\text{Average coverage} = 177 / 35 \approx 7x$$

Assembly

Coverage could also refer to the number of reads covering a particular position in the genome:

CTAGGCCCTCAATTTT
CTCTAGGCCCTCAATTTT
GGCTCTAGGCCCTCATTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
TATCTCGACTCTAGGCC
TCTATATCTGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
GGCGTCTATATCTGGCTCTAGGCCCTCATTTTT

Coverage at this position = 6

Assembly

Basic principle: the more similarity there is between the end of one read and the beginning of another...

TATCTCGACTCTAGGCC
||| ||| | | | | | |
TCTATATCTCGGCTCTAGG

...the more likely they are to have originated from overlapping stretches of the genome:

TATCTCGACTCTAGGCC
TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTT

Assembly

Say two reads truly originate from overlapping stretches of the genome. Why might there be differences?

TATCTGACTCTAGGCC
||||||| |||||
TCTATATCTCGGCTCTAGG
 ↑

1. Sequencing error
2. Difference between inherited *copies* of a chromosome

E.g. humans are diploid; we have two copies of each chromosome, one from mother, one from father. The copies can differ:

Read from Mother:

TATCTGACTCTAGGCC
||||||| |||||

Read from Father: TCTATATCTCGGCTCTAGG

Sequence from Mother: TCTATATCTCGACTCTAGGCC

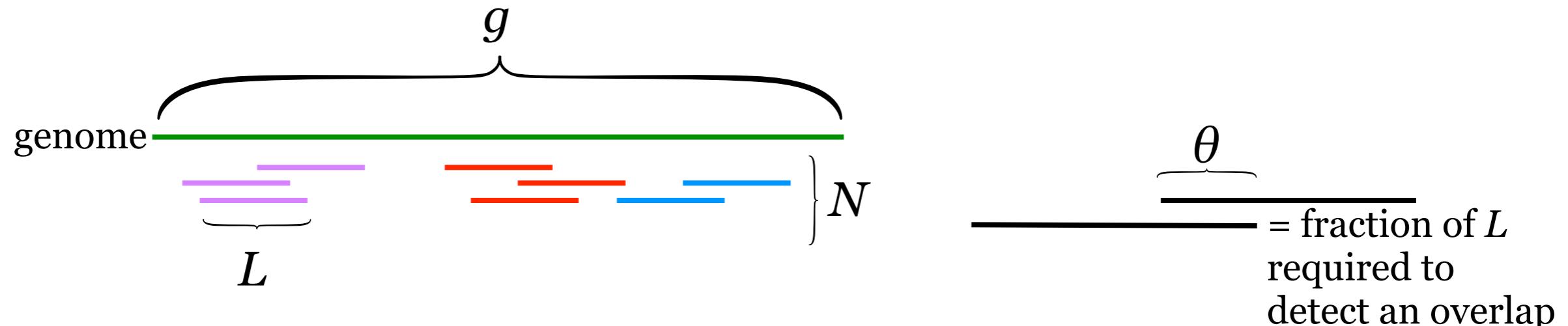
Sequence from Father: TCTATATCTCGGCTCTAGGCC

We'll mostly ignore ploidy, but real tools must consider it

How Much Coverage is Enough? Lander-Waterman Statistics

Lander ES, Waterman MS (1988). "Genomic mapping by fingerprinting random clones: a mathematical analysis". Genomics 2 (3): 231–239

How many reads do we need to be sure we cover the whole genome?



An **island** is a contiguous group of reads that are connected by overlaps of length $\geq \theta L$.
(Various colors above)

Want: Expression for expected # of islands given N, g, L, θ .

Expected # of Islands

$\lambda := N/g$ = probability a read starts at a given position
(assuming random sampling)

Pr(k reads start in an interval of length x)

x trials, want k “successes,” small probability λ of success

Expected # of successes = λx

Poisson approximation to binomial distribution:

$$\Pr(k \text{ reads in length } x) = e^{-\lambda x} \frac{(\lambda x)^k}{k!}$$

Expected # of islands = $N \times \Pr(\text{read is at rightmost end of island})$

$$\begin{aligned} \frac{(1-\theta)L}{\theta L} &= N \times \Pr(0 \text{ reads start in } (1-\theta)L) \\ &= Ne^{-\lambda(1-\theta)L} \frac{\lambda^0}{0!} \quad (\text{from above}) \\ &= Ne^{-\lambda(1-\theta)L} \\ &= Ne^{-(1-\theta)LN/g} \quad \leftarrow LN/g \text{ is called the } \mathbf{coverage} c. \end{aligned}$$

Expected # of Islands, 2

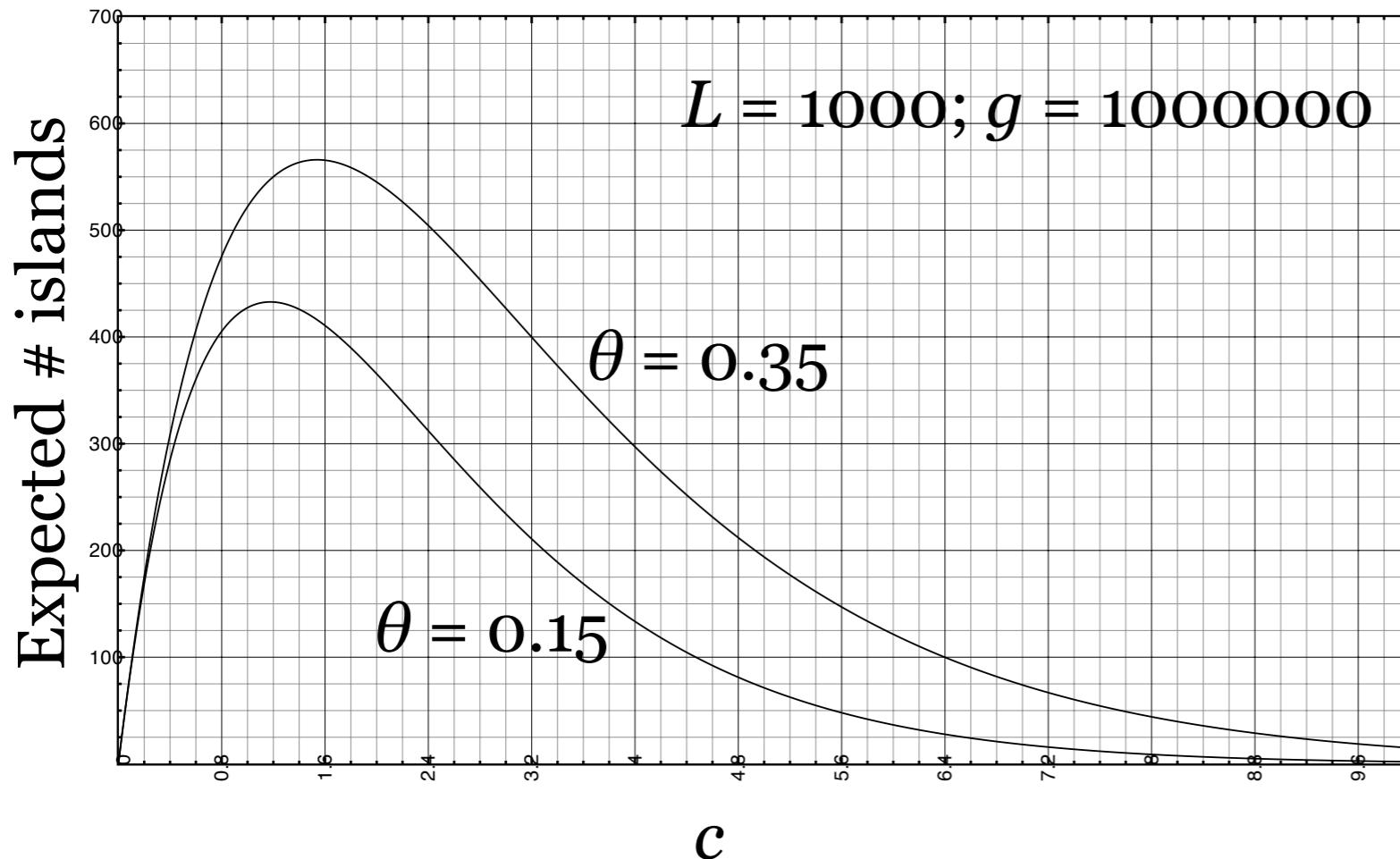
We can rewrite this expression to depend more directly on the things we can control: c and θ

$$\text{Expected # of islands} = Ne^{-(1-\theta)LN/g}$$

$$= Ne^{-(1-\theta)c}$$

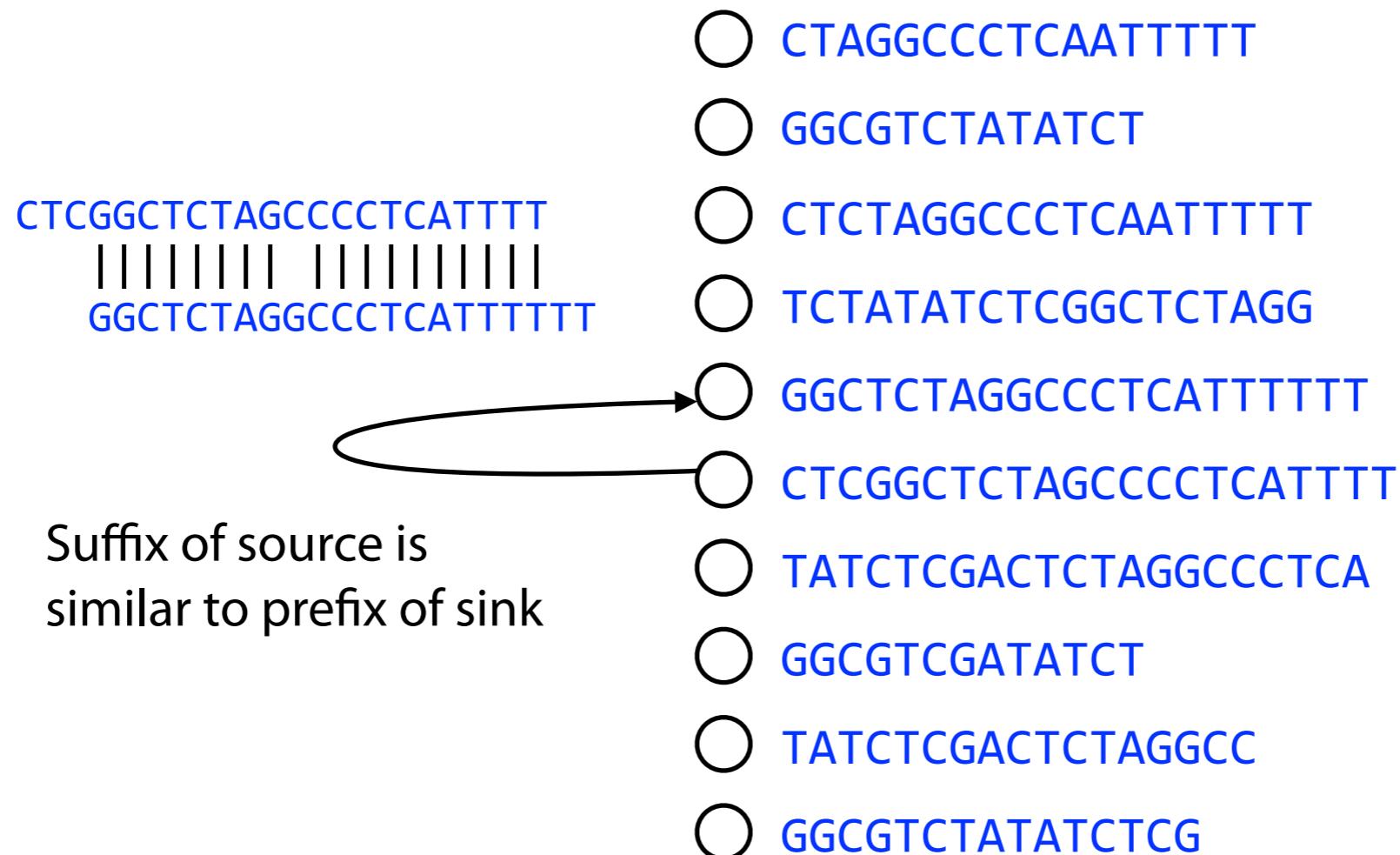
$$= \frac{L/g}{L/g} Ne^{-(1-\theta)c}$$

$$= \frac{g}{L} ce^{-(1-\theta)c}$$



Overlaps

Finding all overlaps is like building a *directed graph* where directed edges connect overlapping nodes (reads)



Directed graph review

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

Directed edge is an *ordered pair* of vertices.
First is the *source*, second is the *sink*.

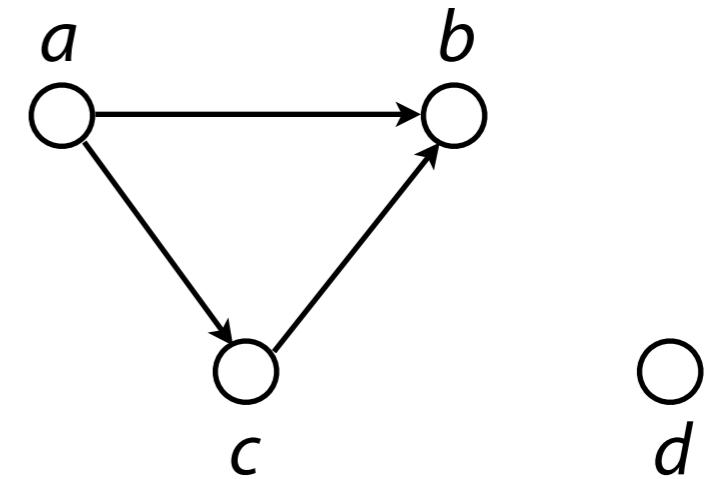
Vertex is drawn as a circle

Edge is drawn as a line with an arrow
connecting two circles

Vertex also called *node* or *point*

Edge also called *arc* or *line*

Directed graph also called *digraph*



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

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

Source Sink

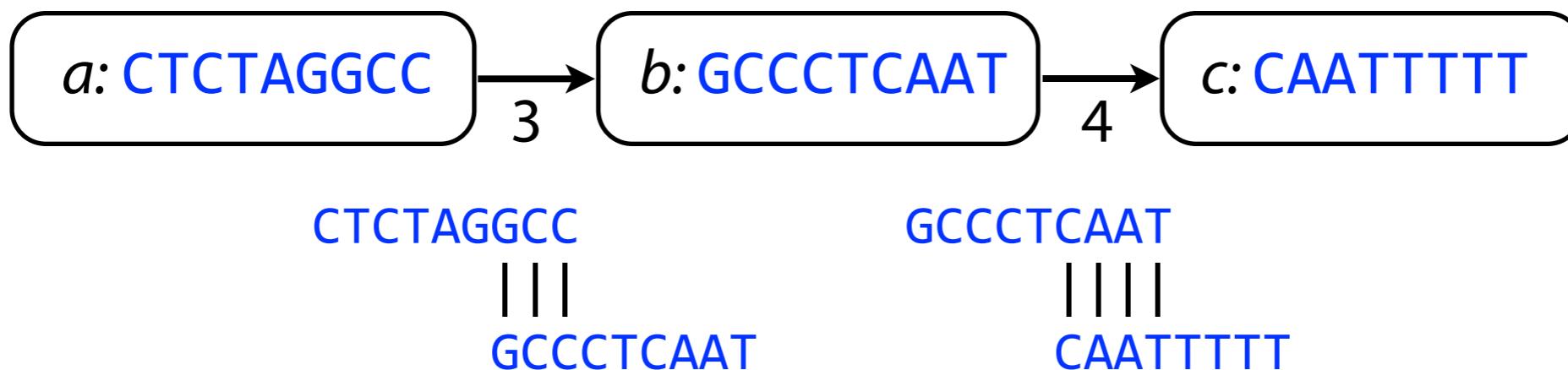
Overlap graph

Below: overlap graph, where an overlap is a suffix/prefix match of at least 3 characters

A vertex is a read, a directed edge is an overlap between suffix of source and prefix of sink

Vertices (reads): { a : CTCTAGGCC, b : GCCCTCAAT, c : CAATTTTT }

Edges (overlaps): { (a, b) , (b, c) }



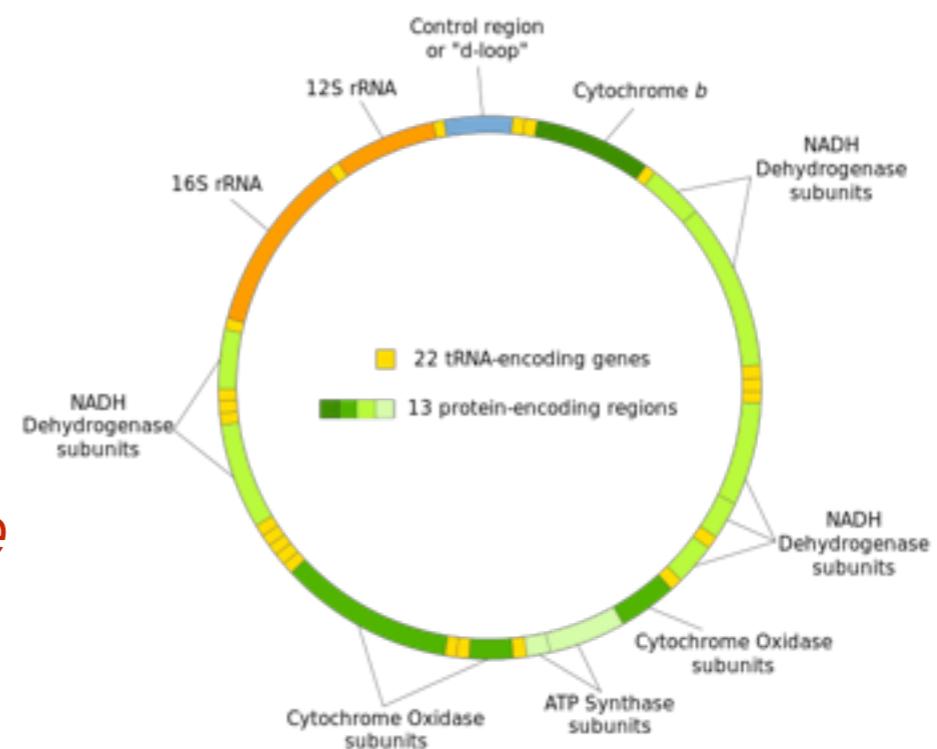
Overlap graph

Overlap graph could contain *cycles*. A cycle is a path beginning and ending at the same vertex.



These happen when the DNA string itself is circular. E.g. bacterial genomes are often circular; mitochondrial DNA is circular.

Cycles could also be due to *repetitive* DNA, as we'll see



Finding overlaps



How do we build the overlap graph?

What constitutes an overlap?

Assume for now an “overlap” is when a suffix of X of length $\geq l$ exactly matches a prefix of Y , where l is given

Finding overlaps

Overlap: length- l suffix of X matches length- l prefix of Y , where l is given

Simple idea: look in Y for occurrences of length- l suffix of X . Extend matches to the left to confirm whether entire prefix of Y matches.

Say $l = 3$

Look for this in Y ,
going right-to-left

$X:$ CTCTAGGCC
 $Y:$ TAGGCCCTC



$X:$ CTCTAGGCC
 $Y:$ TAGGCCCTC

Found it

Extend to left; in this case, we
confirm that a length-6 prefix
of Y matches a suffix of X

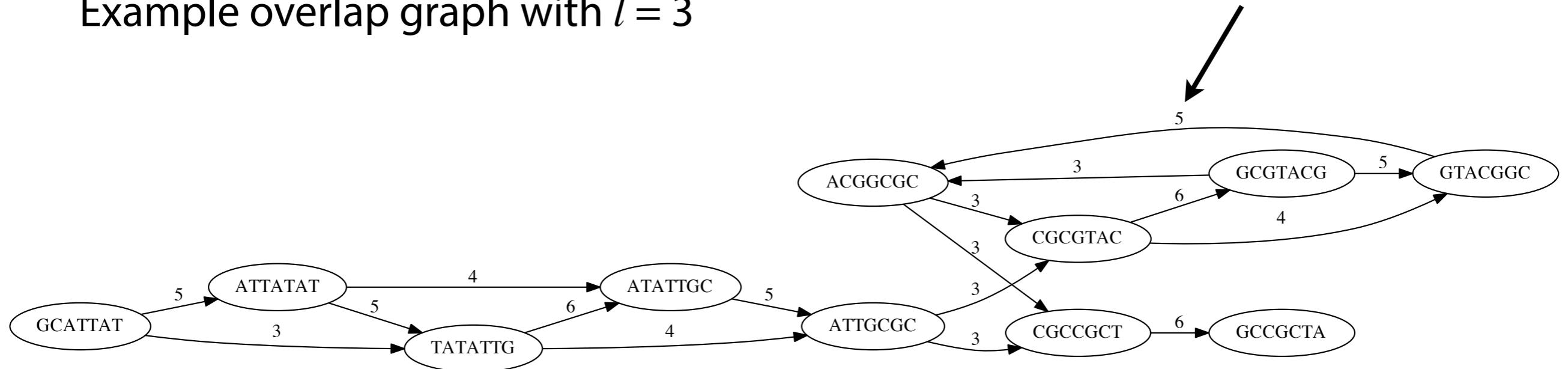
$X:$ CTCTAGGCC
 $Y:$ TAGGCCCTC



Finding overlaps

Edge label is
overlap length

Example overlap graph with $l = 3$



Original string: **GCATTATATATTGCGCGTACGGCGCCGCTACA**

Shortest common superstring

Given a collection of strings S , find $\text{SCS}(S)$: the shortest string that contains all strings in S as substrings

Without requirement of “shortest,” it’s easy: just concatenate them

Example: $S:$ **BAA AAB BBA ABA ABB BBB AAA BAB**

Concatenation: **BAAAABBBAABAABB BBBBAAAABAB**
 ————— 24 —————

$\text{SCS}(S):$ **AAABBBBABAA**
 ————— 10 —————

AAA
AAB
ABB
BBB
BBA
BAB
ABA
BAA

Shortest common superstring

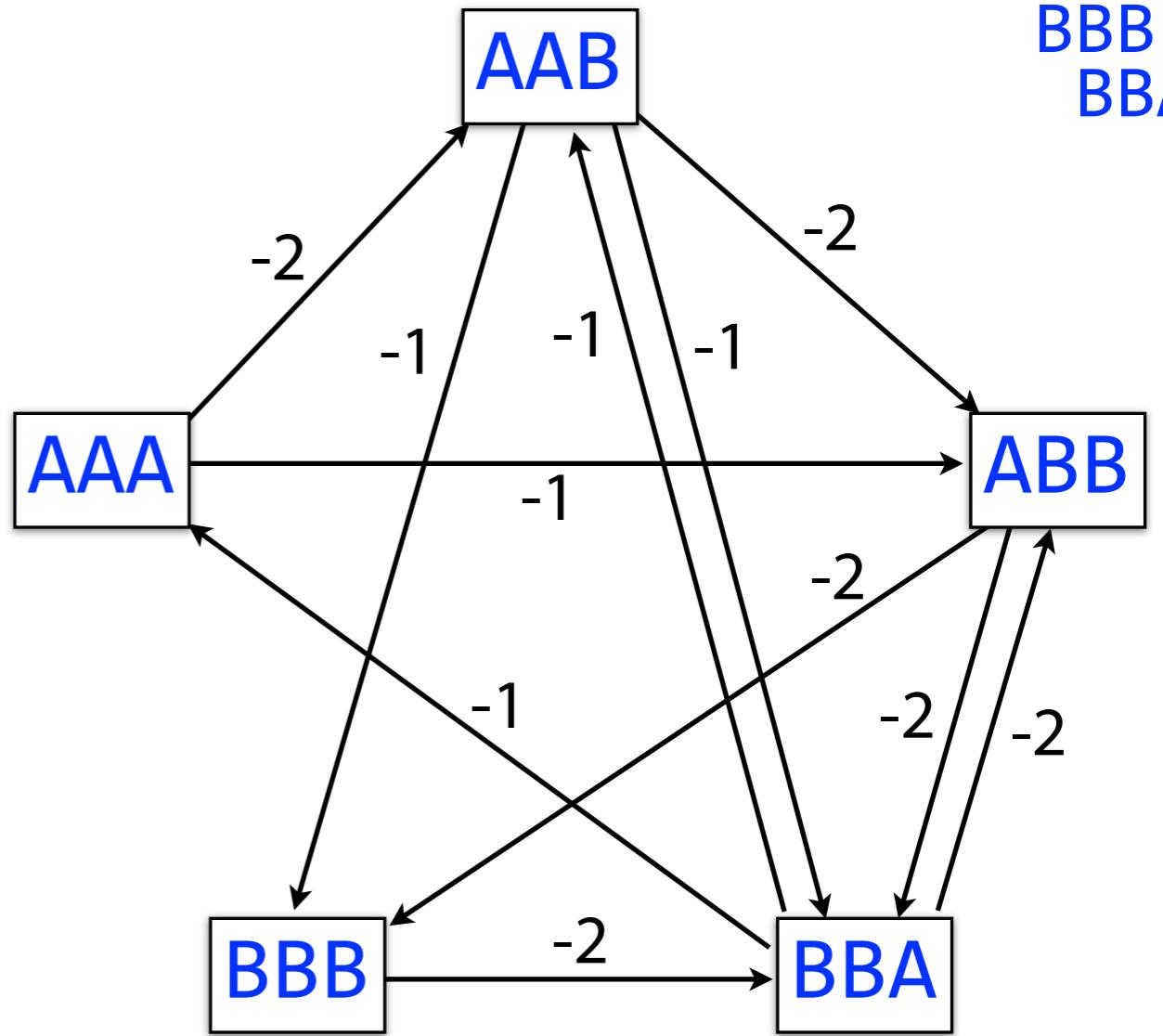
Can we solve it?

Imagine a modified overlap graph where each edge has cost = - (length of overlap)

SCS corresponds to a path that visits every node once, minimizing total cost along path

That's the *Traveling Salesman Problem (TSP)*, which is NP-hard!

$S: \text{AAA AAB ABB BBB BBA}$
 $\text{SCS}(S): \text{AAABBBAA}$
 AAA
 AAB
 ABB
 BBB
 BBA



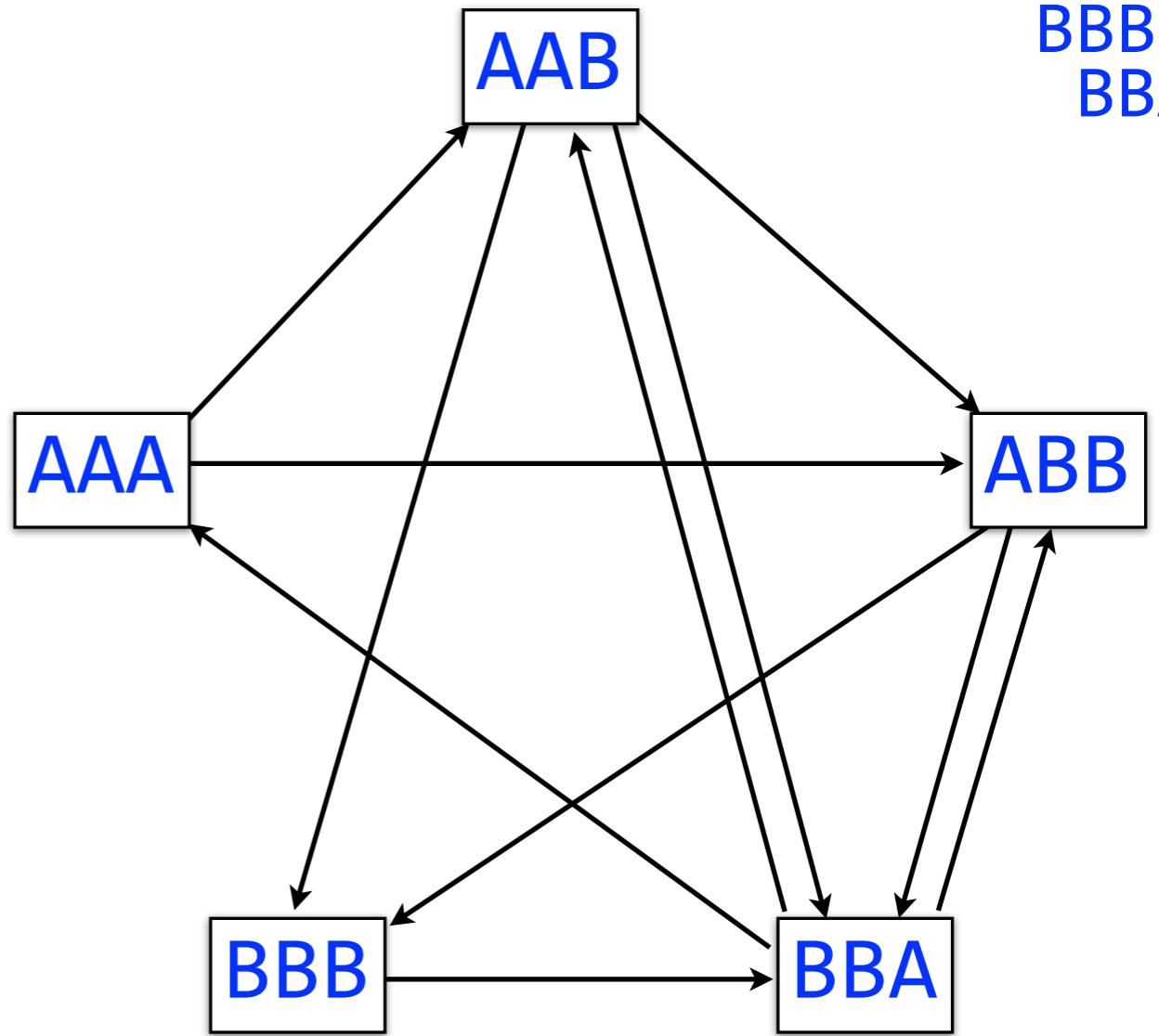
Shortest common superstring

Say we disregard edge weights and just look for a path that visits all the nodes exactly once

That's the *Hamiltonian Path* problem:
NP-complete

Indeed, it's well established that SCS is NP-hard

$S: \text{AAA } \text{AAB } \text{ABB } \text{ BBB } \text{ BBA}$
 $\text{SCS}(S): \text{AAABBBAA}$
 AAA
 AAB
 ABB
 BBB
 BBA



Shortest common superstring

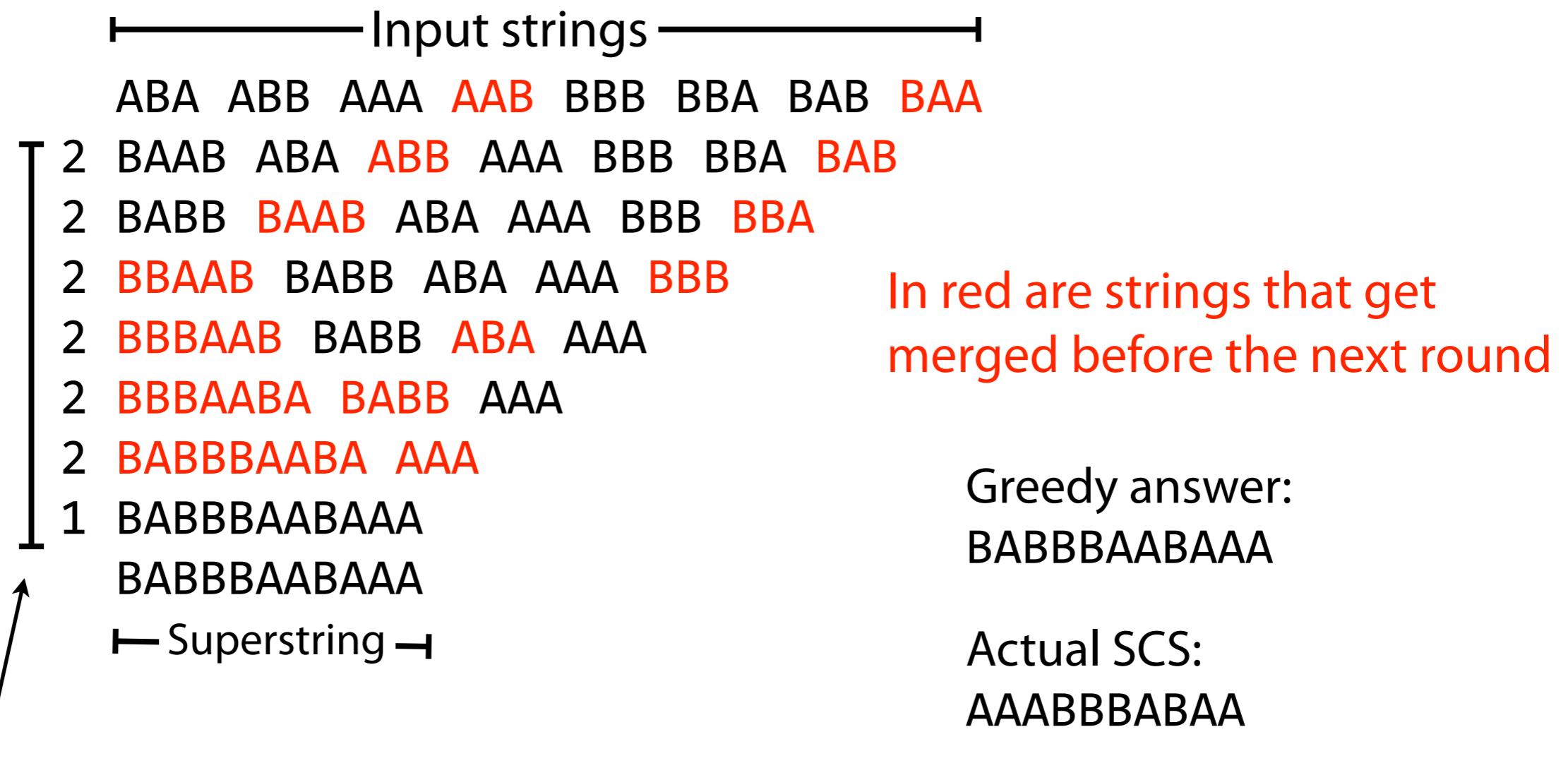
Let's take the hint give up on finding the *shortest possible* superstring

Non-optimal superstrings can be found with a *greedy* algorithm

At each step, the greedy algorithm “greedily” chooses longest remaining overlap, merges its source and sink

Shortest common superstring: greedy

Greedy-SCS algorithm in action ($l = 1$):



Rounds of merging, one merge per line.

Number in first column = length of overlap merged before that round.

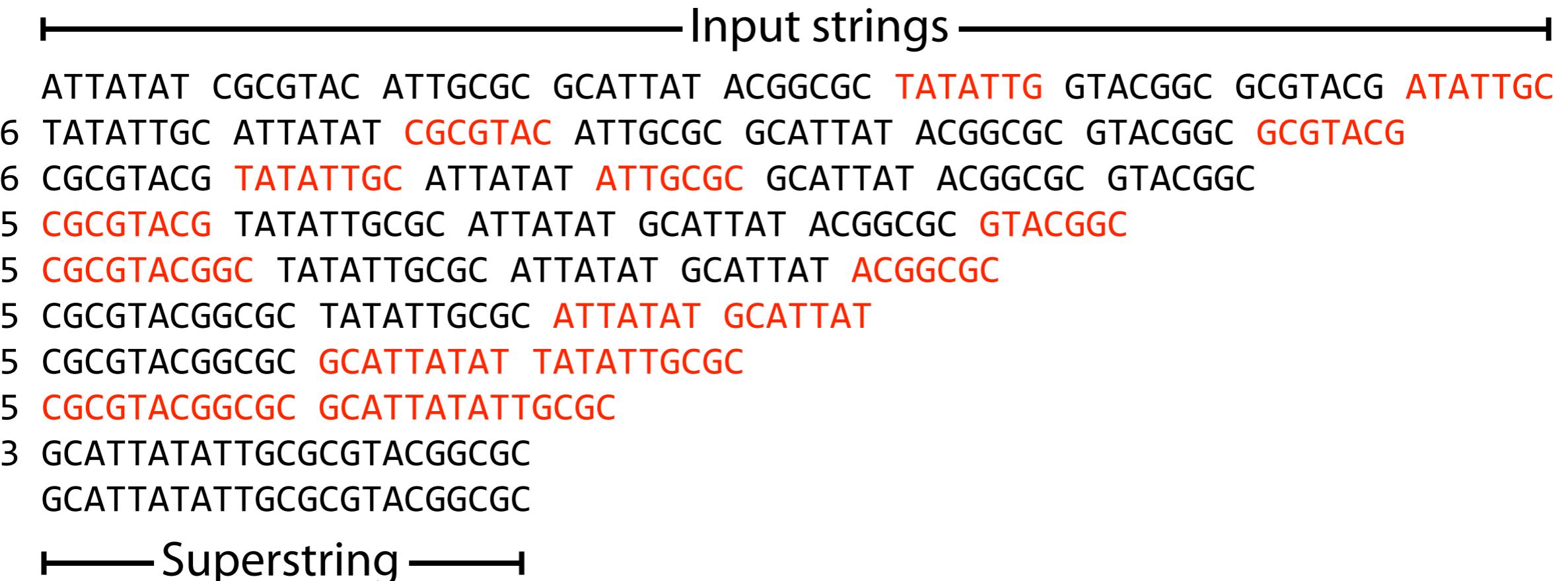
Shortest common superstring: greedy

Greedy algorithm is *not* guaranteed to choose overlaps yielding SCS

But greedy algorithm is a good *approximation*; i.e. the superstring yielded by the greedy algorithm won't be more than ~2.5 times longer than true SCS (see Gusfield 16.17.1)

Shortest common superstring: greedy

Greedy-SCS algorithm in action again ($l = 3$):



Shortest common superstring: greedy

Another setup for Greedy-SCS: assemble all substrings of length 6
from string [a_long_long_long_time](#). $l = 3$.

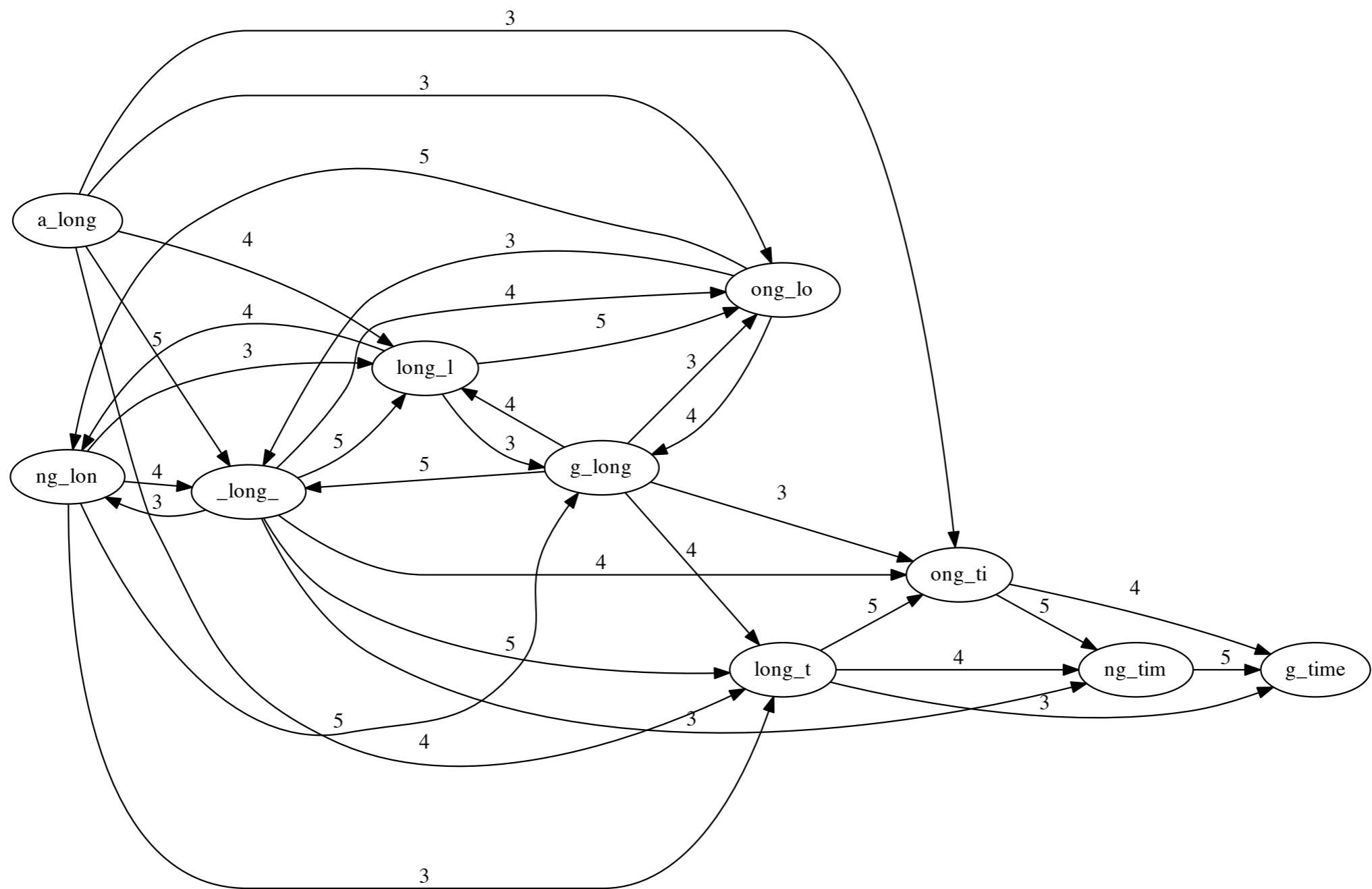
```
ng_lon_long_a_long long_l ong_ti ong_lo long_t g_long g_time ng_time
5 ng_time ng_lon long_a_long long_l ong_ti ong_lo long_t g_long
5 ng_time g_long_ng_lon a_long long_l ong_ti ong_lo long_t
5 ng_time long_ti g_long_ng_lon a_long long_l ong_lo
5 ng_time ong_lon long_ti g_long_a_long long_l
5 ong_lon long_time g_long_a_long long_l
5 long_lon long_time g_long_a_long
5 long_lon g_long_time a_long
5 long_long_time a_long
4 a_long_long_time
a_long_long_time
```

I only got back: [a_long_long_time](#) (missing a [long](#))

What happened?

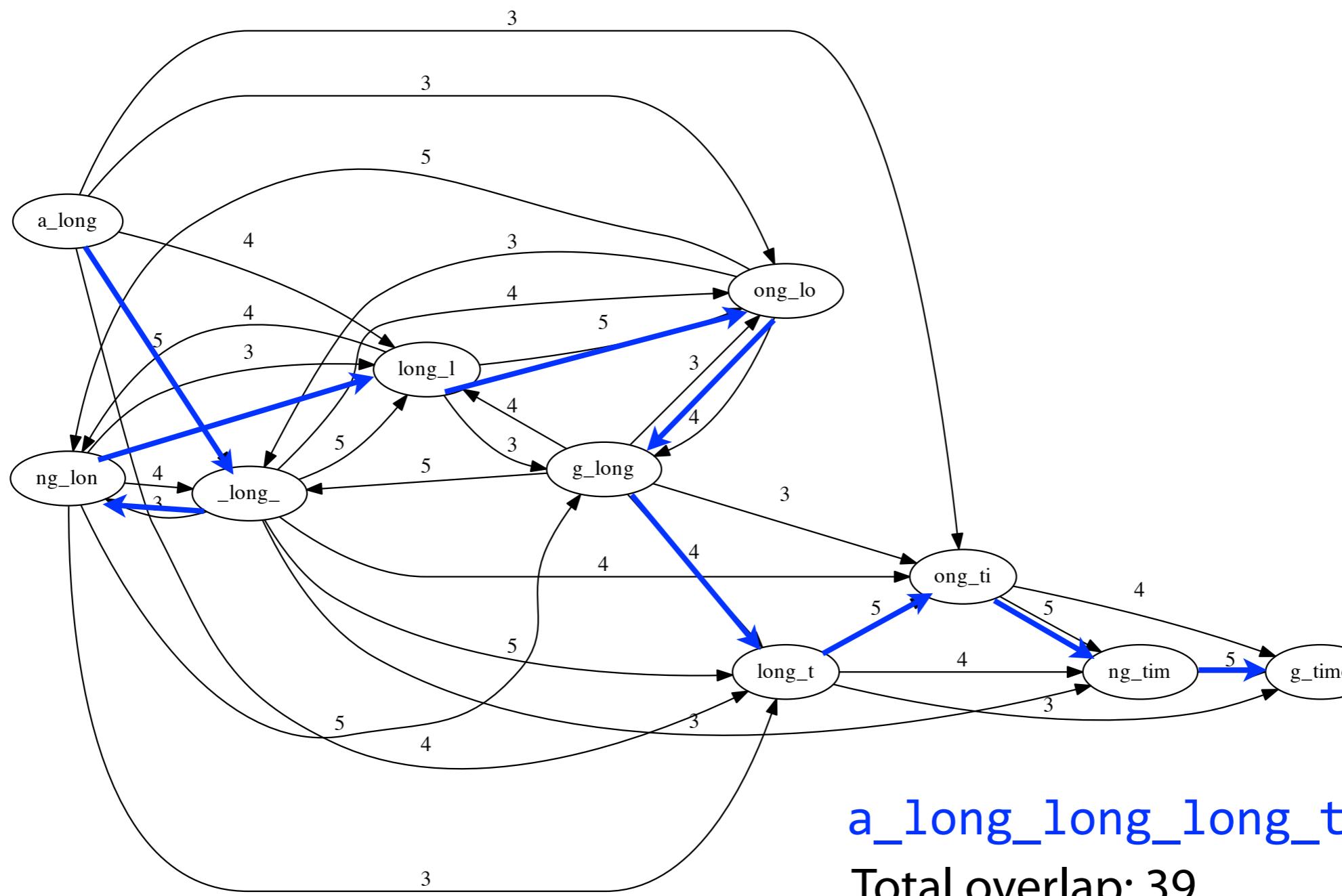
Shortest common superstring: greedy

The overlap graph for that scenario ($l = 3$):



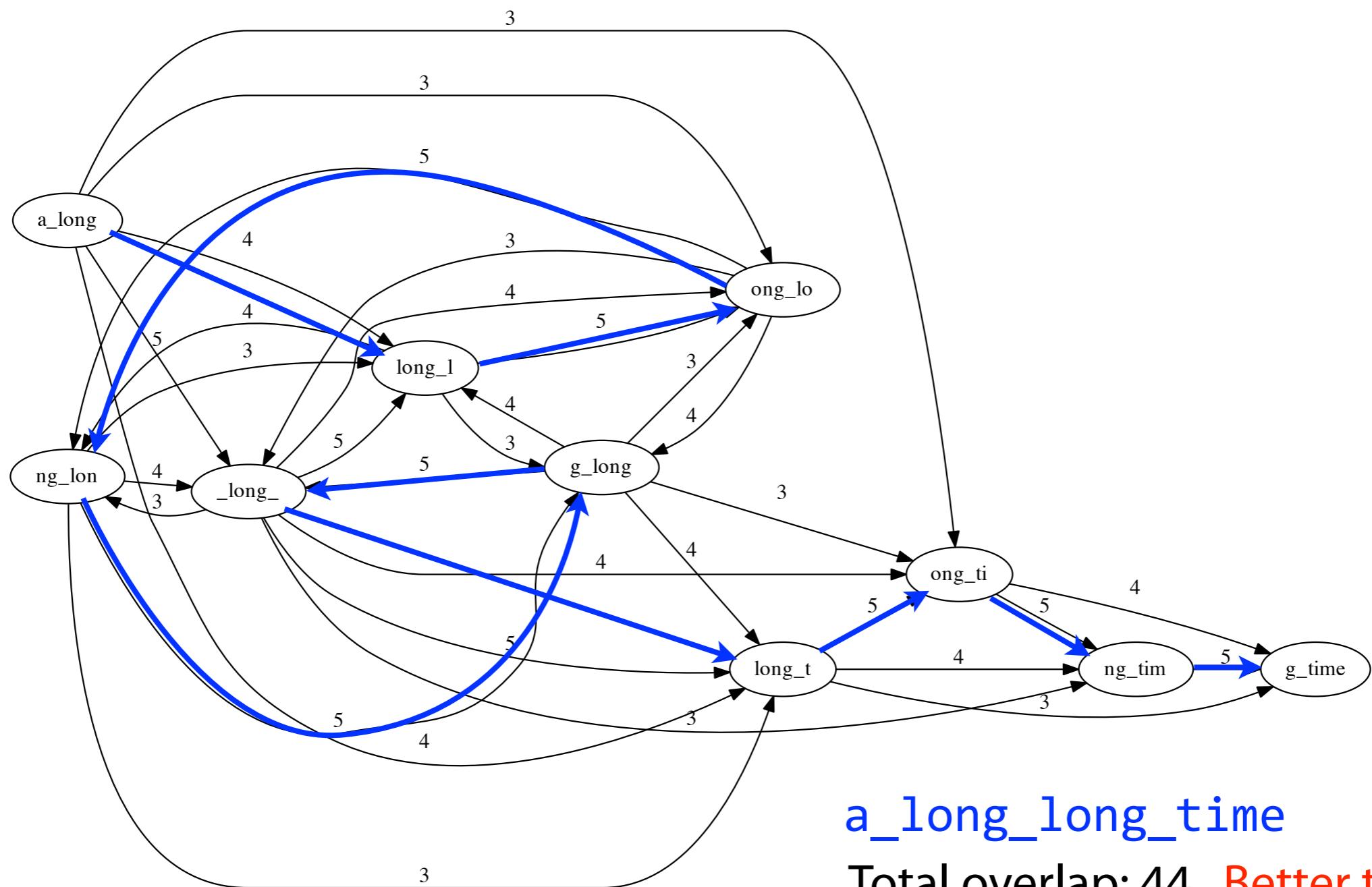
Shortest common superstring: greedy

The overlap graph for that scenario ($l = 3$):



Shortest common superstring: greedy

The overlap graph for that scenario ($l = 3$):



a_long_long_time

Total overlap: 44 Better than the
correct path!

Shortest common superstring: greedy

Same example, but increased the substring length from 6 to 8

```
long_lon ng_long_ _long_lo g_long_t ong_long g_long_l ong_time a_long_l _long_ti long_tim
7 long_time long_lon ng_long_ _long_lo g_long_t ong_long g_long_l a_long_l _long_ti
7 _long_time long_lon ng_long_ _long_lo g_long_t ong_long g_long_l a_long_l
7 _long_time a_long_lo long_lon ng_long_ g_long_t ong_long g_long_l
7 _long_time ong_long_ a_long_lo long_lon g_long_t g_long_l
7 g_long_time ong_long_ a_long_lo long_lon g_long_l
7 g_long_time ong_long_ a_long_lo long_lon g_long_l
7 g_long_time ong_long_l a_long_lon
7 g_long_time a_long_long_l
3 a_long_long_long_time
a_long_long_long_time
```

Got the whole thing: **a_long_long_long_time**

Shortest common superstring: greedy

Why are substrings of length 8 long enough for Greedy-SCS to figure out there are 3 copies of `long`?

a_long_long_long_time

g_long_l



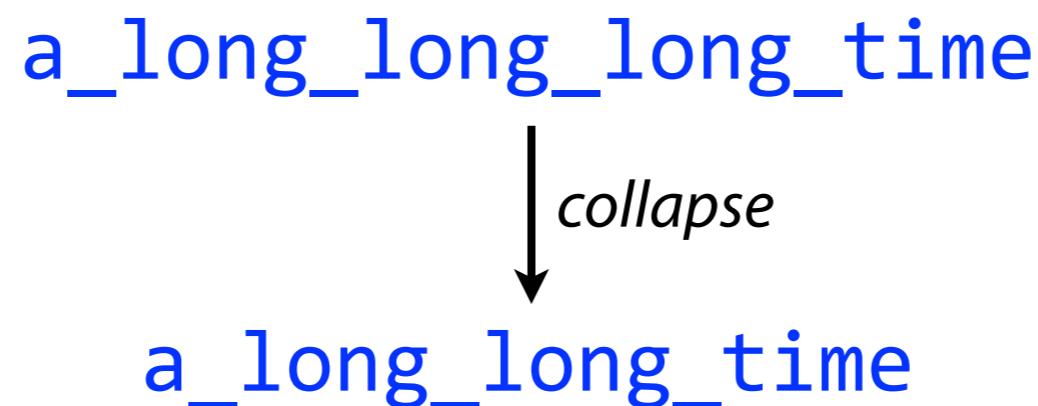
One length-8 substring spans all three `longs`

Repeats

Repeats often foil assembly. They certainly foil SCS, with its “shortest” criterion!

Reads might be too short to “resolve” repetitive sequences. This is why sequencing vendors try to increase read length.

Algorithms that don’t pay attention to repeats (like our greedy SCS algorithm) might *collapse* them



The human genome is ~ 50% repetitive!

Repeats

Basic principle: *repeats foil assembly*

Another example using Greedy-SCS:

Input: `it_was_the_best_of_times_it_was_the_worst_of_times`

Extract every substring of length k , then run Greedy-SCS.

Do this for various l (min overlap length) and k .

l, k	output
3, 5	<code>the_worst_of_times_it_was_the_best_o</code>
3, 7	<code>s_the_worst_of_times_it_was_the_best_of_t</code>
3, 10	<code>_was_the_best_of_times_it_was_the_worst_of_tim</code>
3, 13	<code>it_was_the_best_of_times_it_was_the_worst_of_times</code>

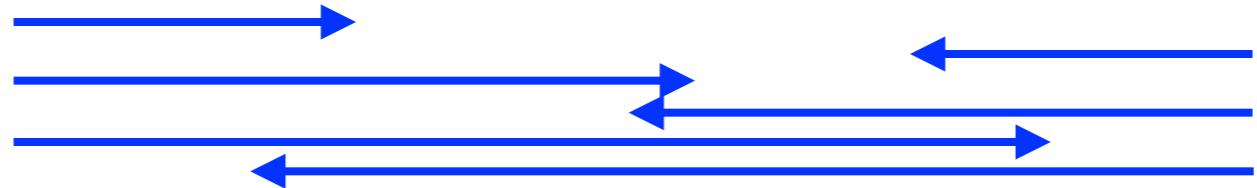
Repeats

Basic principle: *repeats foil assembly*

Longer and longer substrings allow us to “anchor” more of the repeat to its non-repetitive context:

swinging_and_the_ringing_of_the_bells_bells_bells_bells_bells_bells


Often we can “walk in” from both sides. When we meet in the middle, the repeat is resolved:

ringing_of_the_bells_bells_bells_bells_bells_to_the_rhyhming


Repeats

Basic principle: *repeats foil assembly*

Yet another example using Greedy-SCS:

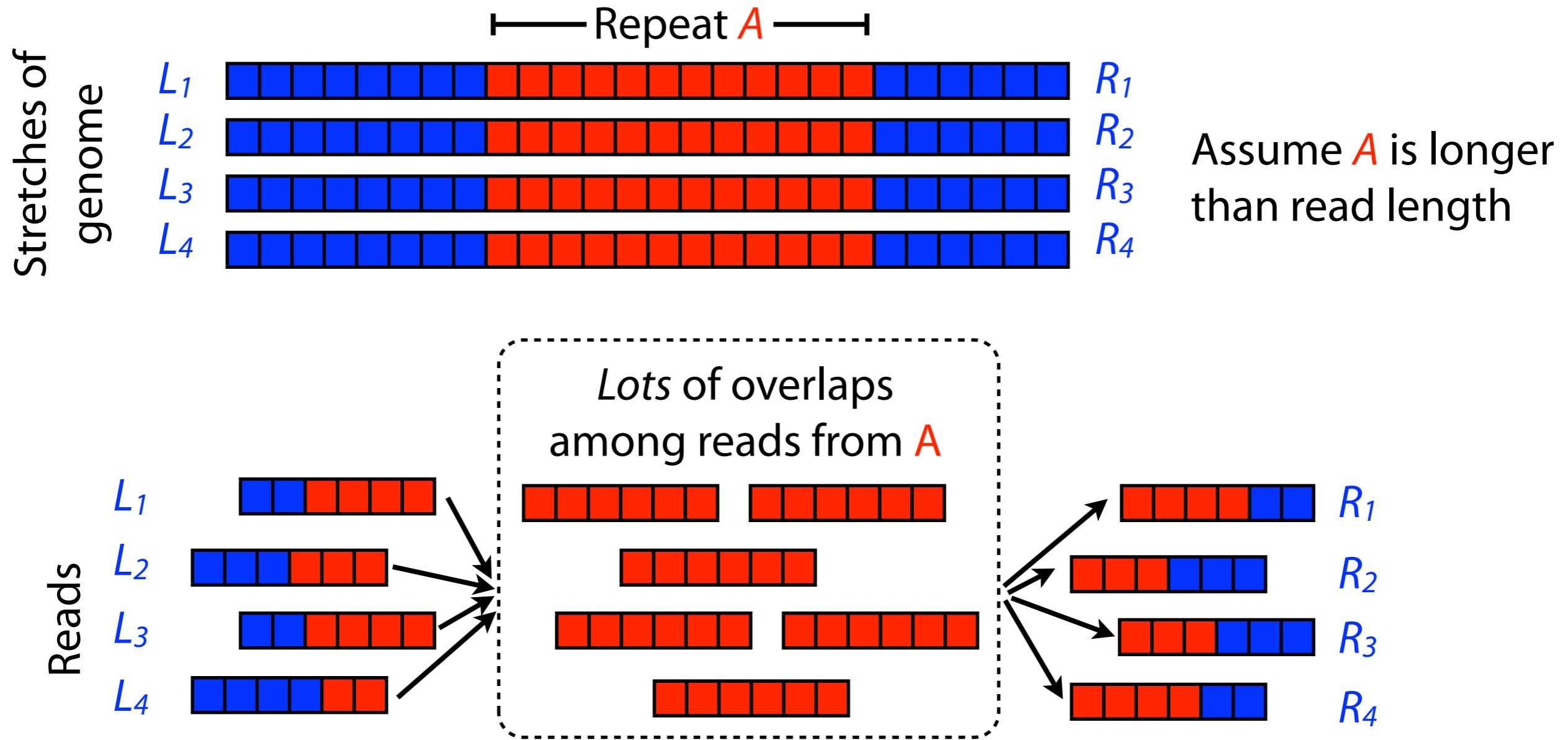
Input: swinging_and_the_ringing_of_the_bells_bells_bells_bells_bells

l, k	read length	output	
3, 7		swinging_and_the_ringing_of_the_bells_bells	
3, 13		swinging_and_the_ringing_of_the_bells_bells_bells	
3, 19		swinging_and_the_ringing_of_the_bells_bells_bells_b	
3, 25		swinging_and_the_ringing_of_the_bells_bells_bells_bells	

$\xrightarrow{\text{longer and longer substrings allow us to “reach” further into the repeat}}$

Repeats

Picture the portion of the overlap graph involving repeat A



Even if we avoid collapsing copies of A , we can't know which paths *in* correspond to which paths *out*

Shortest common superstring: post mortem

SCS is flawed as a way of formulating the assembly problem

No tractable way to find optimal SCS

Had to use Greedy-SCS. Answers might be too long.

SCS spuriously collapses repetitive sequences

Answers might be too short, by a lot!

Need formulations that are (a) tractable, and (b) handle repeats as gracefully as possible

Remember: repeats foil assembly no matter the algorithm. This is a property of read length and repetitiveness of the genome.

Taxonomy of assembly approaches

Search for most parsimonious explanation of the reads (shortest superstring)

Exact solutions are intractable (e.g. TSP), but a greedy approximation is possible

Any solution will collapse repeats spuriously

Search for “maximum likelihood” explanation of the reads; i.e. force solution to be consistent with uniform coverage

Boža, Vladimír, Broňa Brejová, and Tomáš Vinař. "GAML: Genome Assembly by Maximum Likelihood." Algorithms in Bioinformatics. Springer Berlin Heidelberg, 2014. 122-134.

Medvedev, Paul, and Michael Brudno. "Maximum likelihood genome assembly." Journal of computational Biology 16.8 (2009): 1101-1116.

Give up on unresolvable repeats and use a tractable algorithm to assemble the resolvable portions. **This is what real tools do.**

Real-world assembly methods

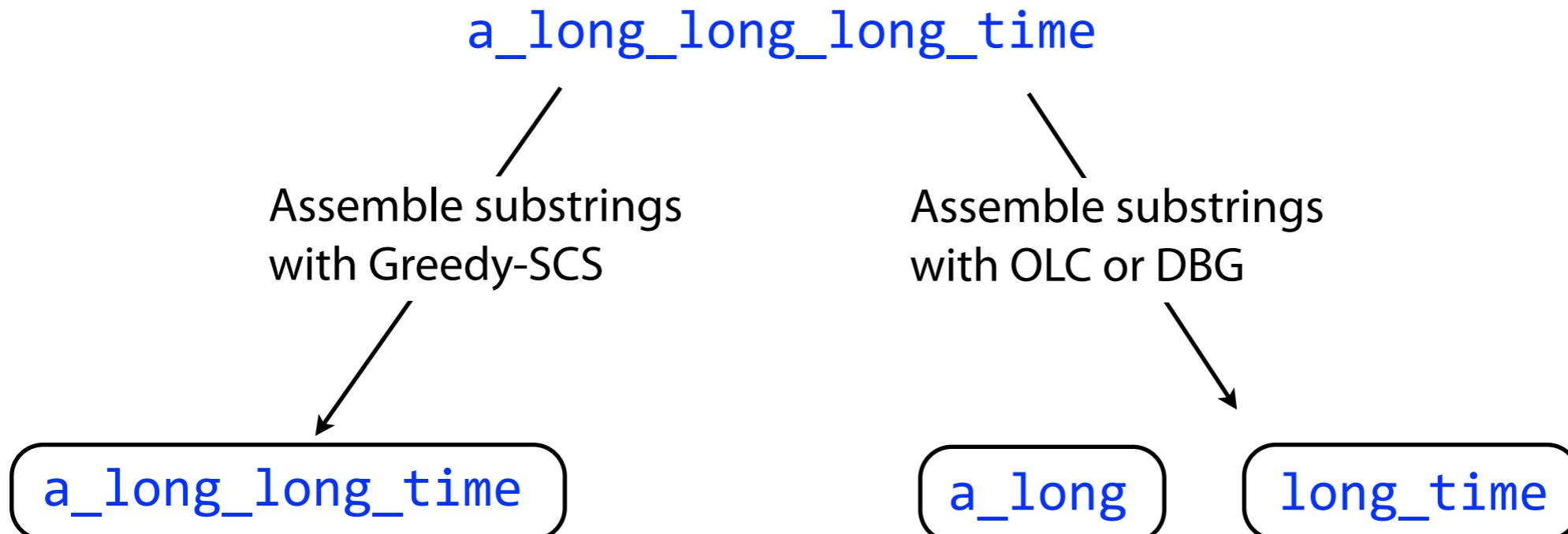
OLC: Overlap-Layout-Consensus assembly

DBG: De Bruijn graph assembly

Both handle unresolvable repeats by essentially *leaving them out*

Unresolvable repeats break the assembly into fragments

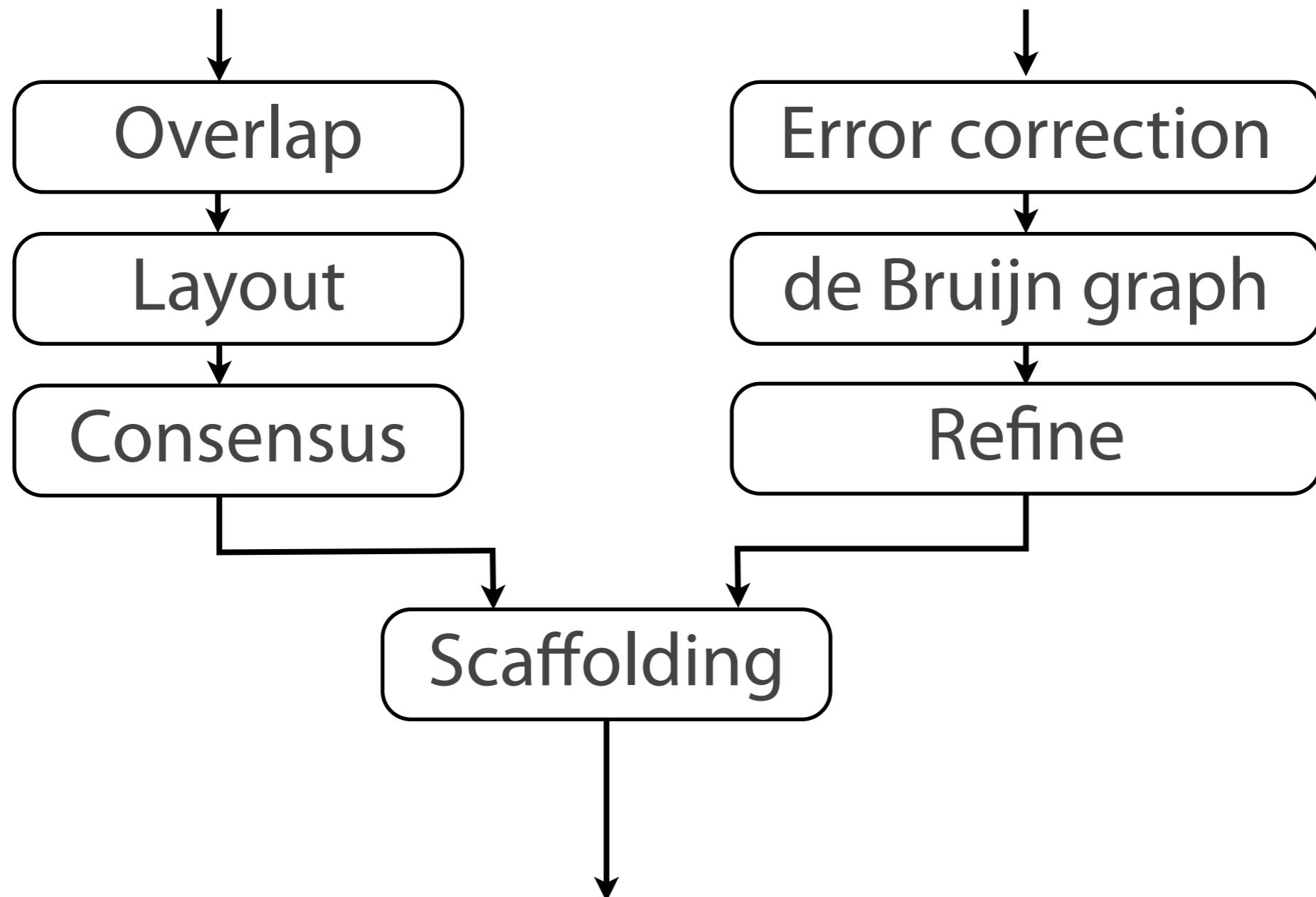
Fragments are *contigs* (short for *contiguous*)



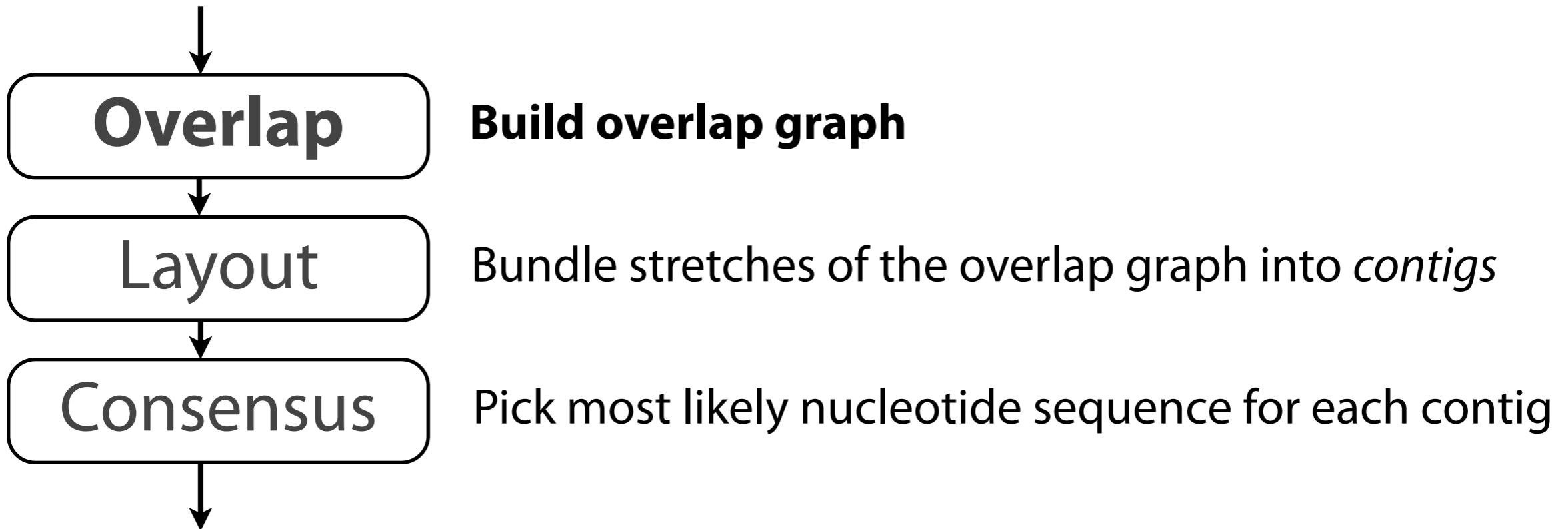
Assembly alternatives

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

Alternative 2: de Bruijn graph (DBG) assembly



Overlap Layout Consensus



Finding overlaps

Can we be less naive than this?

Say $l = 3$

Look for this in Y ,
going right-to-left

$X:$ CTCTAGGCC

$Y:$ TAGGCCCTC



$X:$ CTCTAGGCC

$Y:$ TAGGCCCTC



Found it

Extend to left; in this case, we
confirm that a length-6 prefix
of Y matches a suffix of X

$X:$ CTCTAGGCC

$Y:$ TAGGCCCTC



We're doing this for *every pair* of input strings

Finding overlaps

Can we use suffix trees for overlapping?

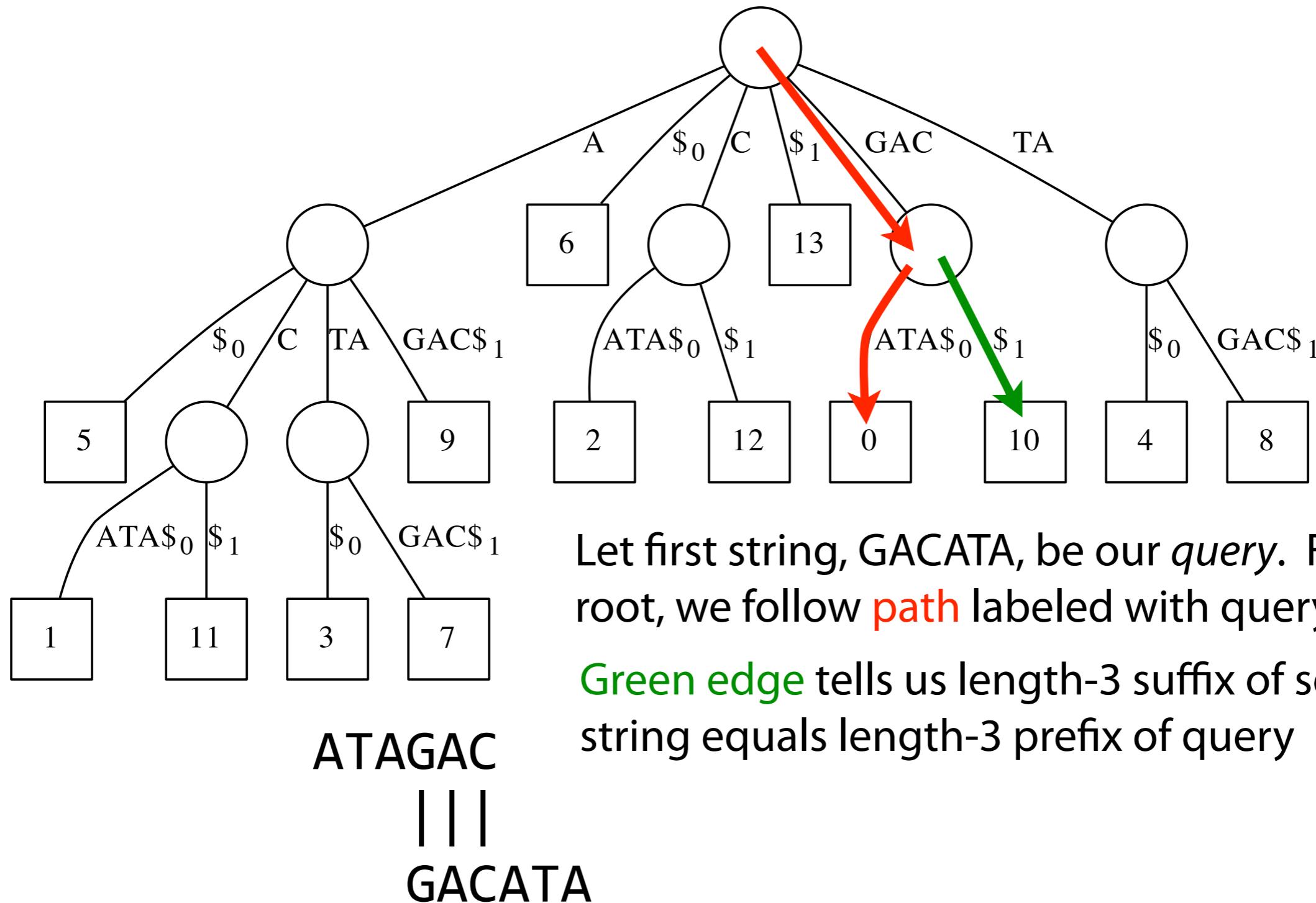
Problem: Given a collection of strings S , for each string x in S find all overlaps involving a prefix of x and a suffix of another string y

Hint: Build a generalized suffix tree of the strings in S

Finding overlaps with suffix tree

Generalized suffix tree for {"GACATA", "ATAGAC"}

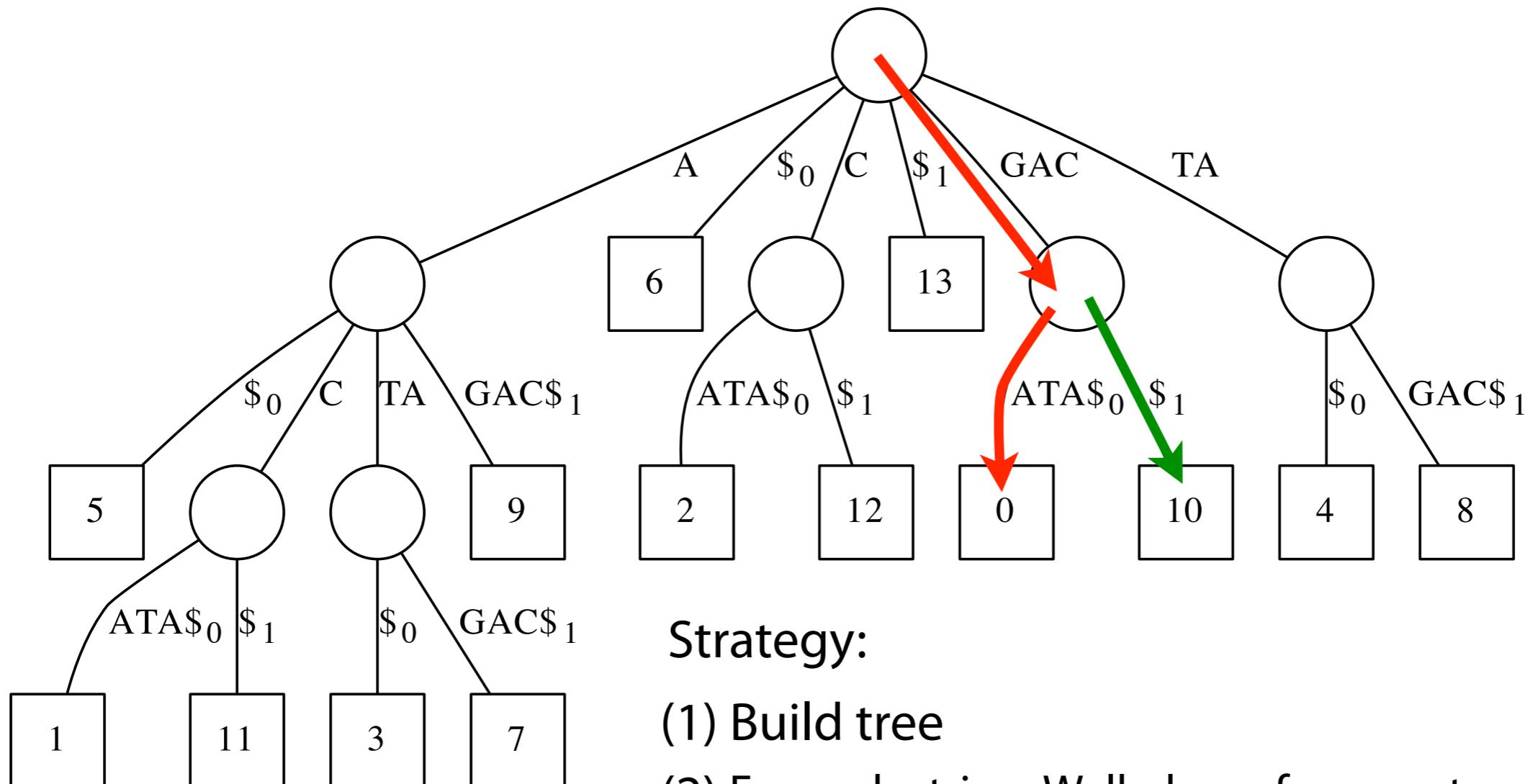
GACATA\$₀ATAGAC\$₁



Finding overlaps with suffix tree

Generalized suffix tree for {"GACATA", "ATAGAC"}

GACATA\$₀ATAGAC\$₁



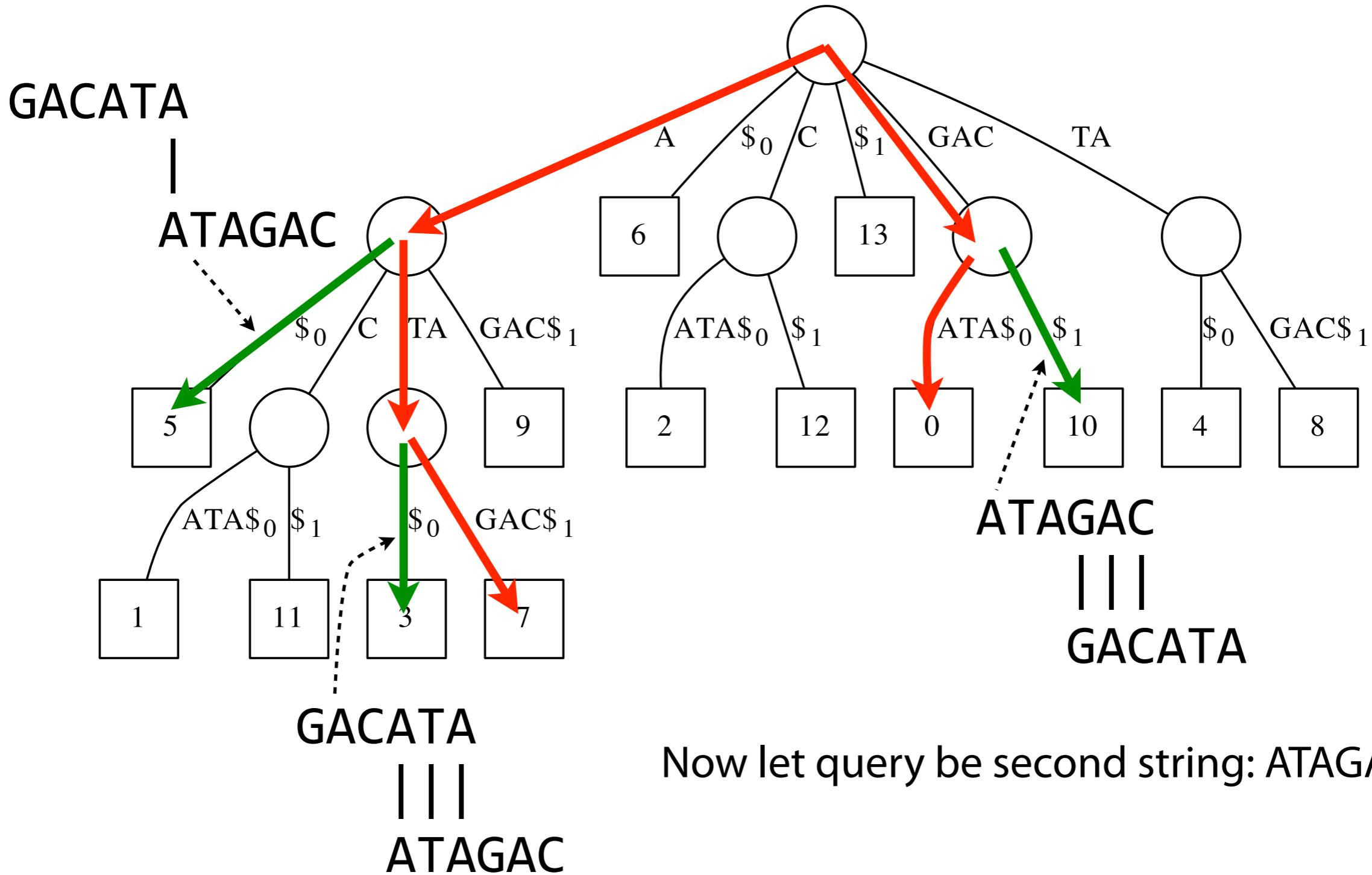
Strategy:

- (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

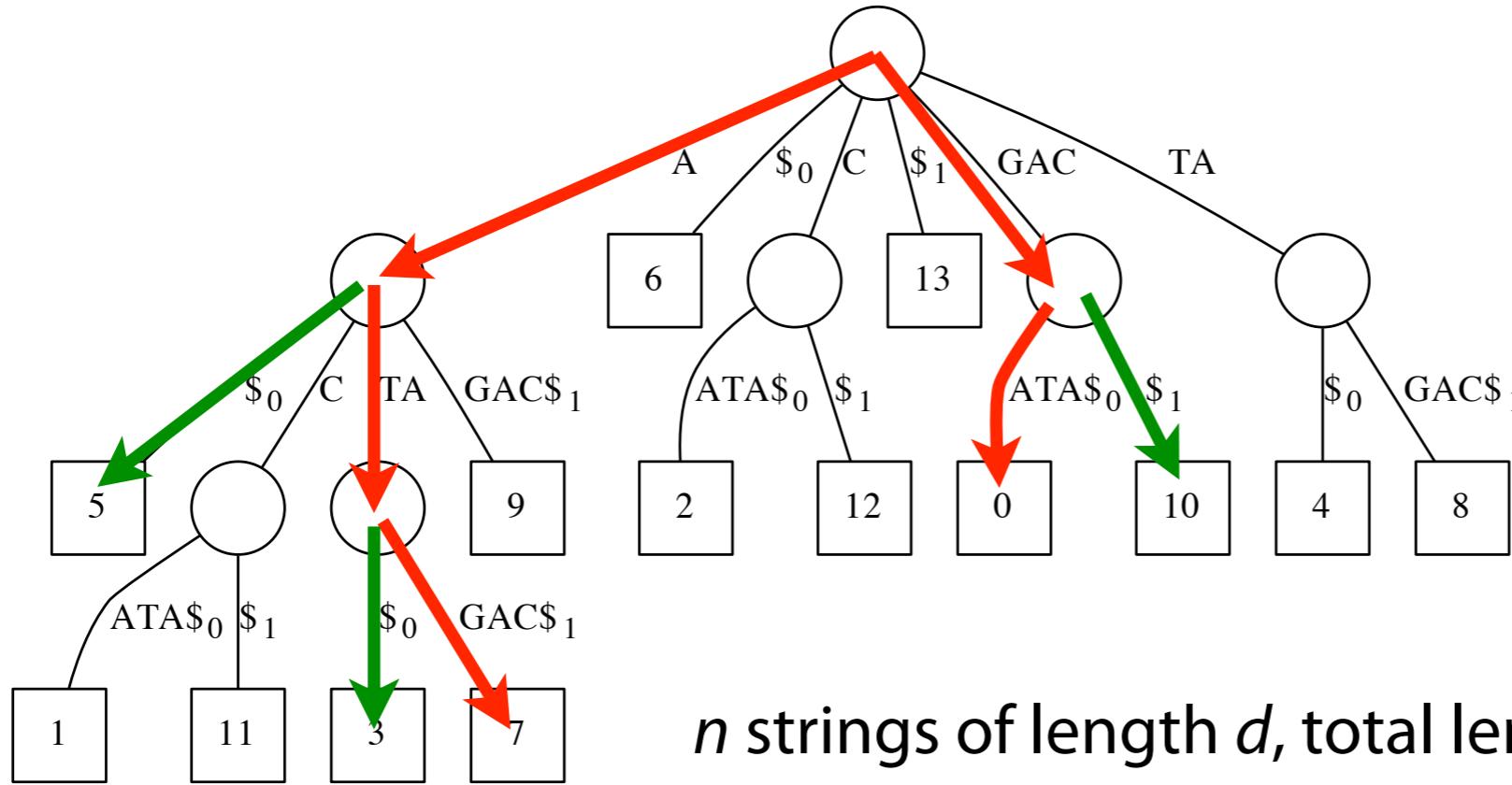
Generalized suffix tree for {"GACATA", "ATAGAC"}

GACATA\$₀ATAGAC\$₁



Finding overlaps with suffix tree

Generalized suffix tree for {"GACATA", "ATAGAC"}
GACATA\$₀ATAGAC\$₁



n strings of length d , total length $N = nd$, and
 $a = \#$ of string pairs that overlap

Time to build generalized suffix tree: $O(N)$

... to walk down red paths: $O(N)$

... to report all overlaps (green): $O(a)$

Overall: $O(N + a)$

Bounds don't include n^2 ,
but a is $O(n^2)$ in worst case

Finding overlaps

What if we want to allow mismatches and gaps in the overlap?

I.e. How do we find the best *alignment* of a suffix of X to a prefix of Y ?

$X:$ CTCGGCCCTAGG
 ||| |||||
 $Y:$ GGCTCTAGGCC

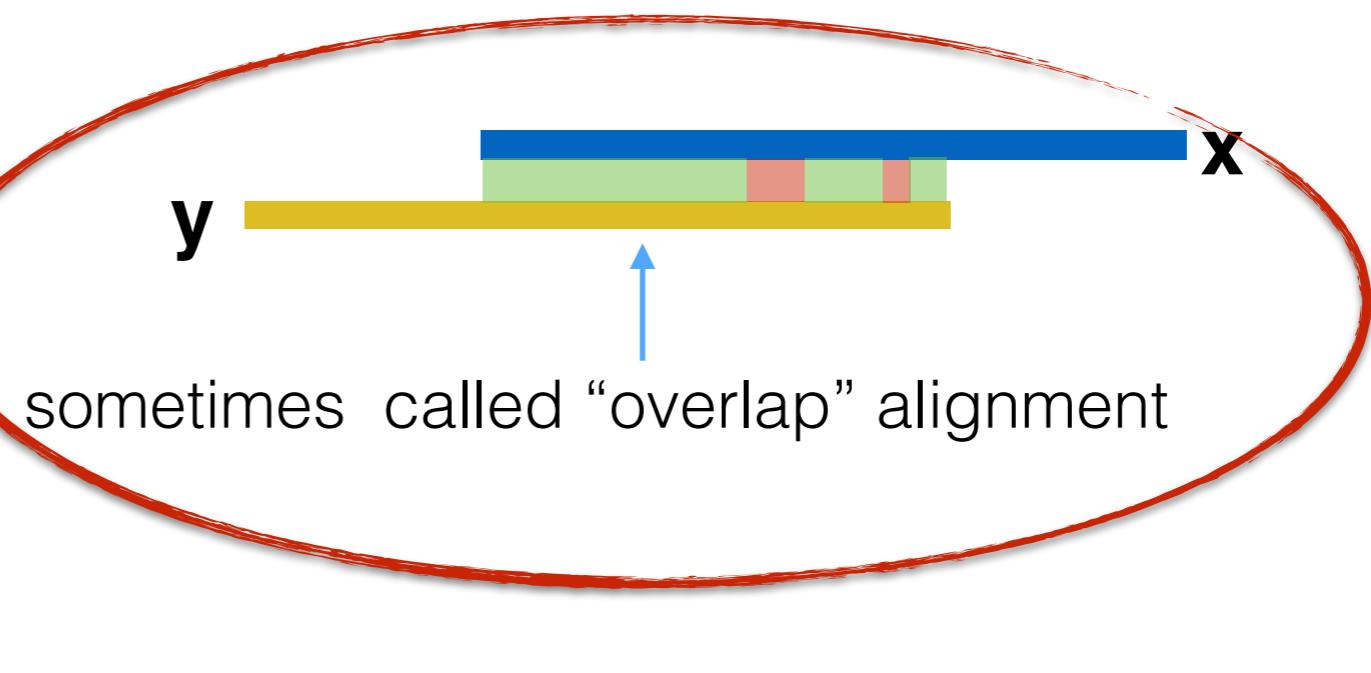
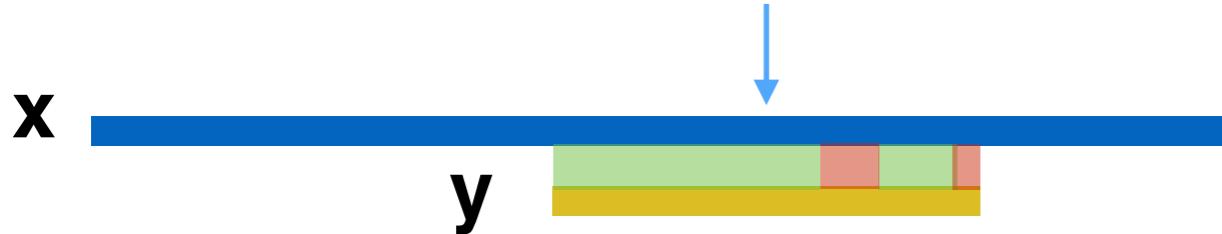
Dynamic programming

But we must frame the problem such that only backtraces involving a suffix of X and a prefix of Y are allowed

Recall: Semi-global Alignment

Semi-global (glocal): Gaps at the beginning or end of **x** or **y** are free. Useful when one string is significantly shorter than the other or we want to find an overlap between the suffix of one string and a prefix of the other

sometimes called “cost-free-ends” or “fitting” alignment



This variant is useful for our purposes here

Finding overlaps with dynamic programming

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

Number of overlaps to try: $O(n^2)$

Size of each dynamic programming matrix: $O(d^2)$

Overall: $O(n^2d^2) = O(N^2)$

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

But dynamic programming is more flexible, allowing mismatches and gaps

In practice, overlappers are between the two, using indexes to filter away non-overlapping pairs, then dynamic programming for the remainder

Finding overlaps

Overlapping is typically the slowest part of assembly

Consider a second-generation sequencing dataset with hundreds of millions or billions of reads!

Approaches from alignment unit can be adapted to finding overlaps

We saw adaptations of naive exact matching, suffix-tree-assisted exact matching, and dynamic programming

Could also have adapted efficient exact matching, approximate string matching, co-traversal, ...

Finding overlaps

Celera Assembler's overlapper is probably the best documented:

Inverted substring indexes built on batches of reads

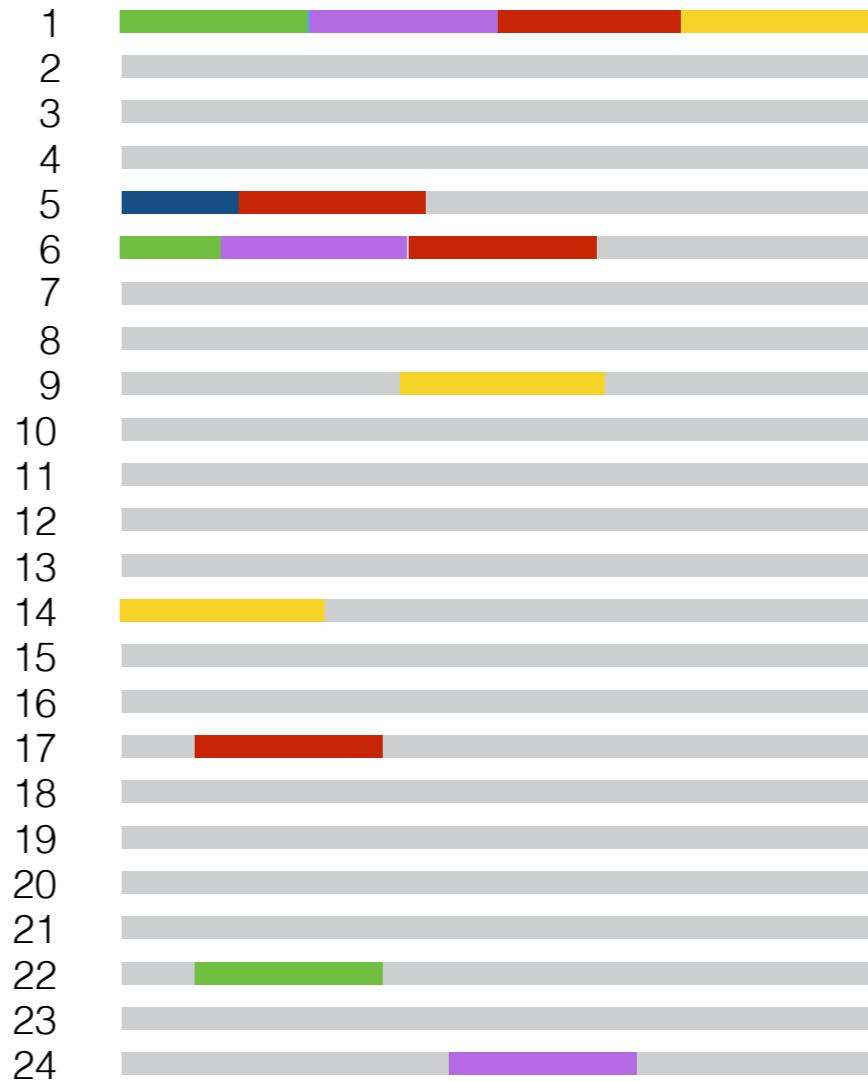
Only look for overlaps between reads that share one or more substrings of some length

<http://sourceforge.net/apps/mediawiki/wgs-assembler/index.php?title=RunCA#Overlapper>

Inverted substring index is a “k-mer” lookup table.
It maps every short fixed-length substring to the set of reads where it occurs.

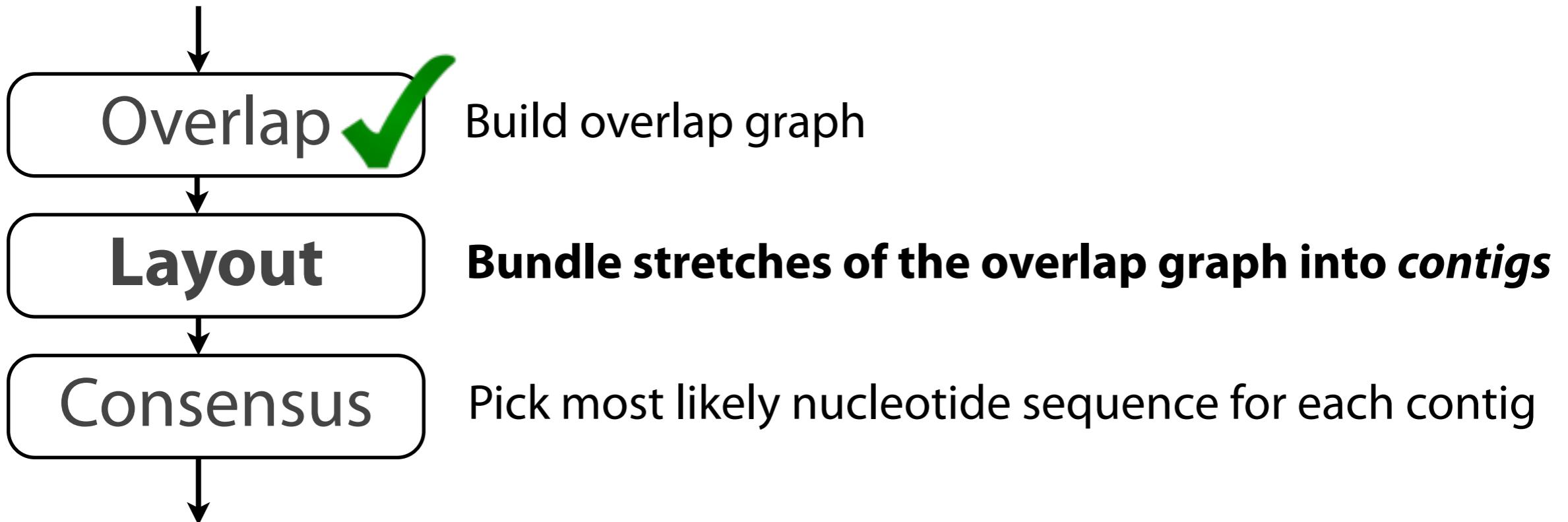
Utility of an inverted index

	1, 5, 6, 17
	1, 6, 24
	1, 6, 22
	▪
	▪
	▪



Only reads sharing at least 1 indexed substring can possibly have an exact overlap. Checking only these pairs *greatly* reduces the burden of detecting overlaps. However, overlapping can still be one of the slowest steps in an assembly.

Overlap Layout Consensus



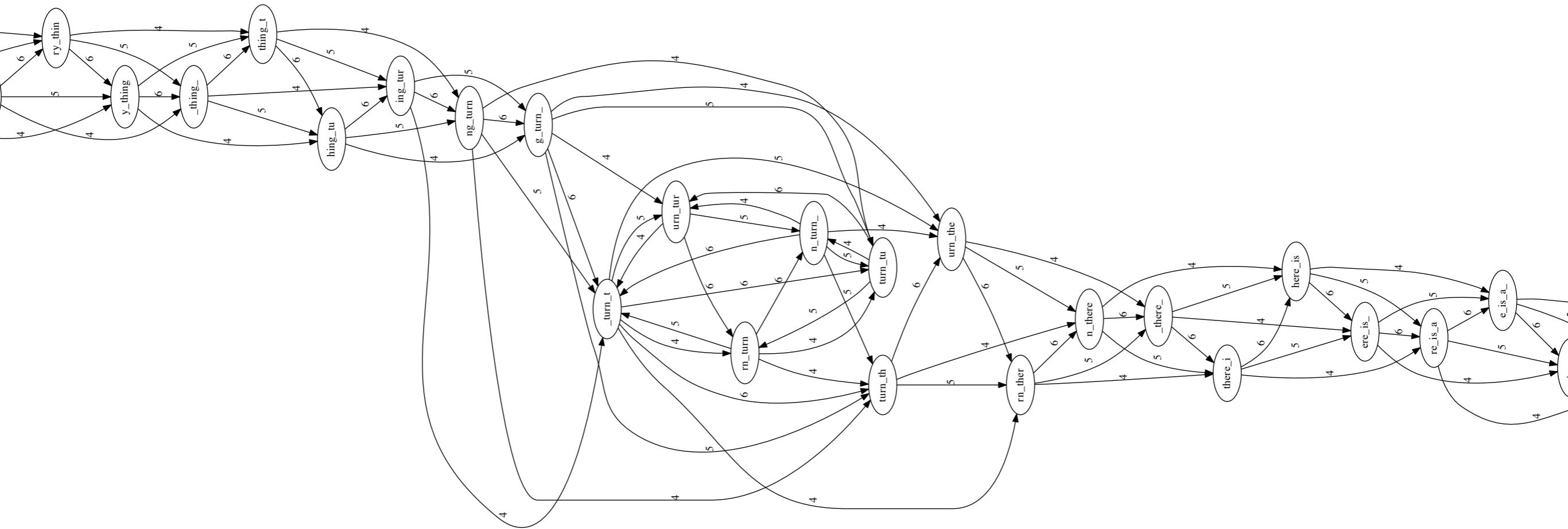
Layout

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

Below: part of the overlap graph for

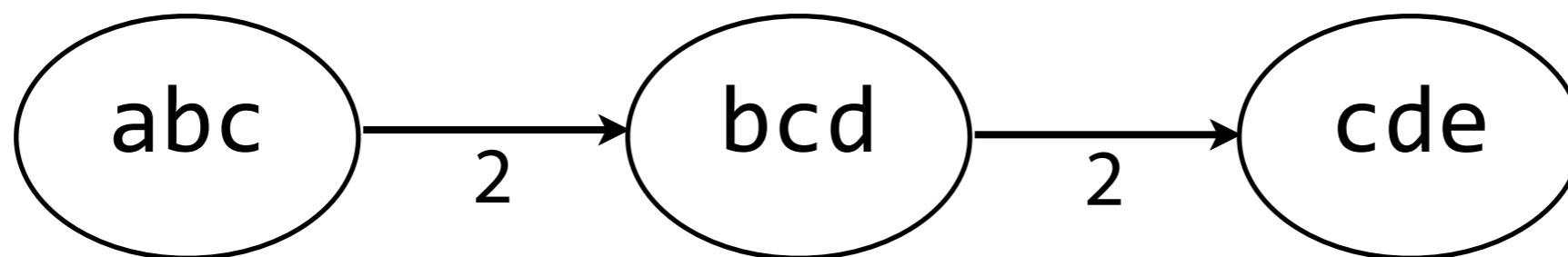
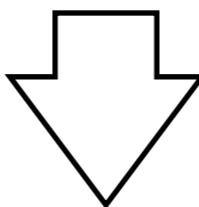
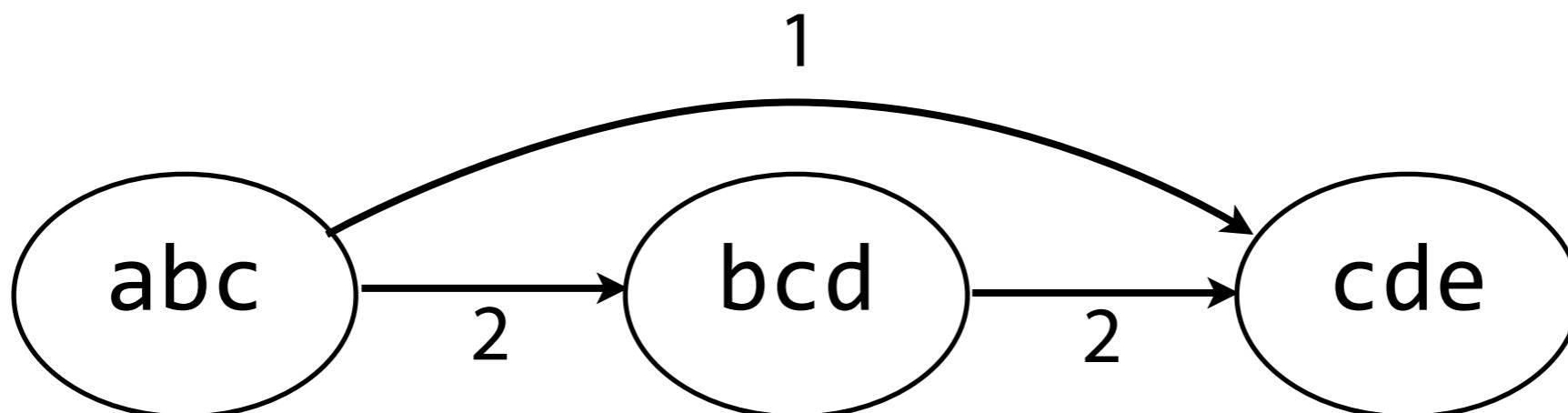
[to_every_thing_turn_turn_there_is_a_season](#)

$l = 4, k = 7$



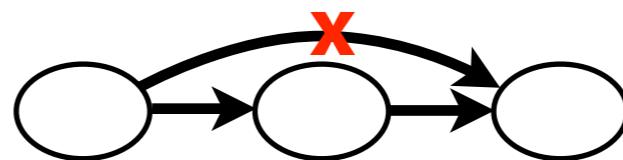
Layout

Picture gets clearer after removing some transitively-inferrible edges

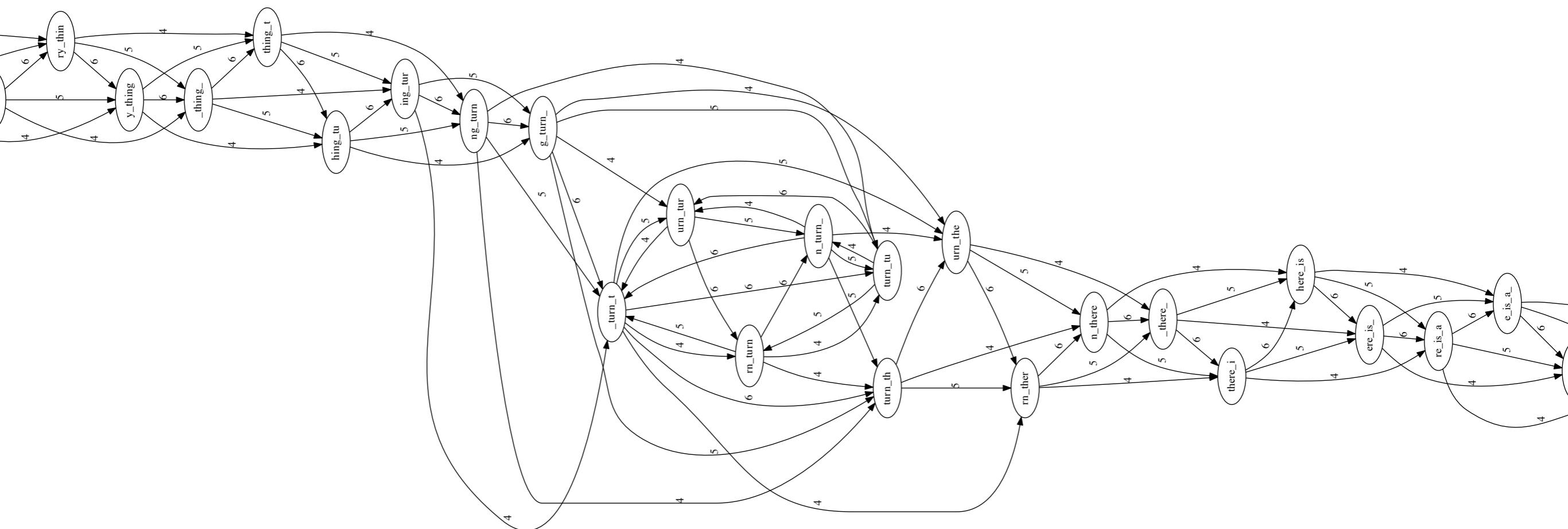


Layout

Remove transitively-inferrible edges, starting with edges that skip one node:

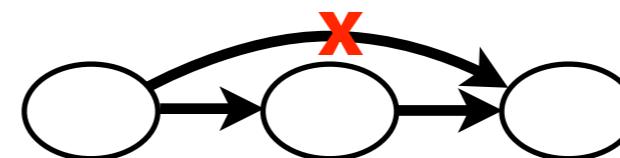


Before:

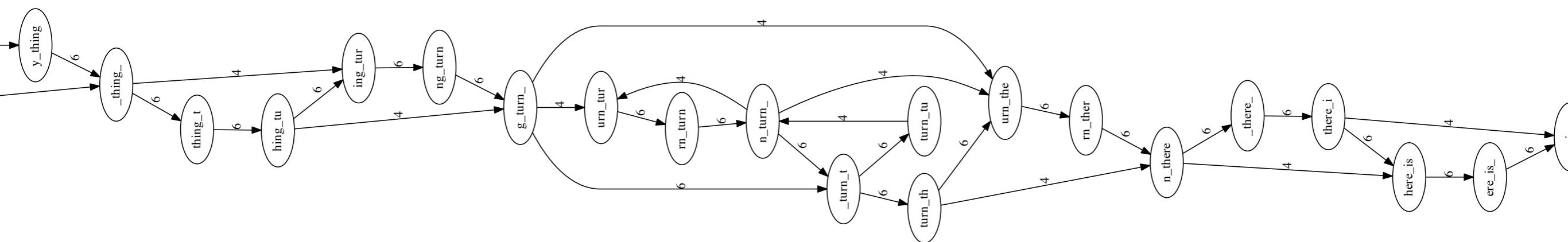


Layout

Remove transitively-inferrible edges, starting with edges that skip one node:



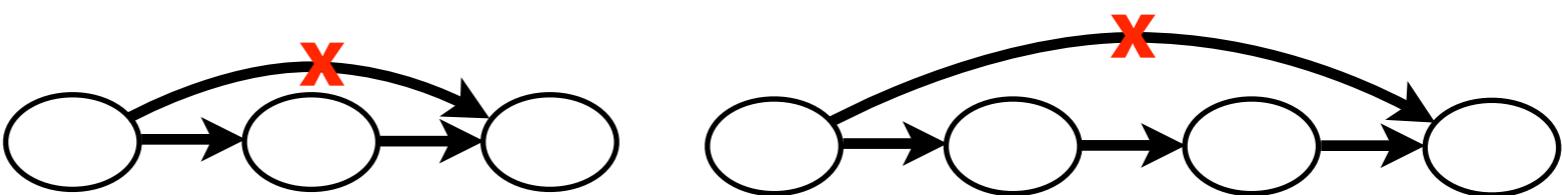
After:



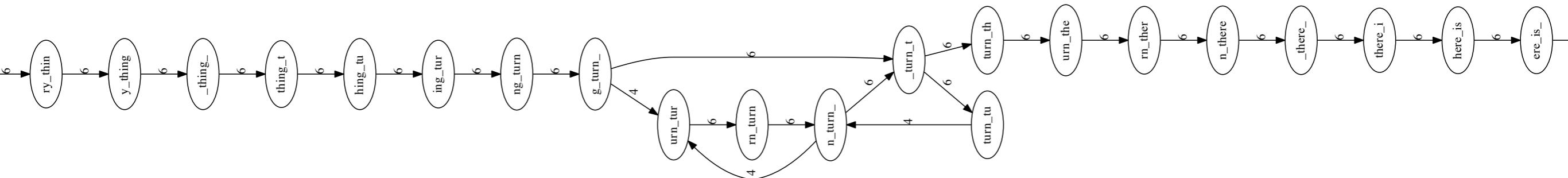
These edges are between reads whose overlaps completely encompass the center node.

Layout

Remove transitively-inferrible edges, starting with edges that skip one or two nodes:



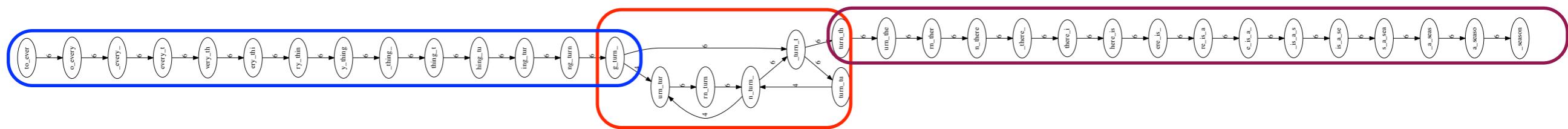
After:



Even simpler

Layout

Emit *contigs* corresponding to the non-branching stretches



Contig 1

to_every_thing_turn_

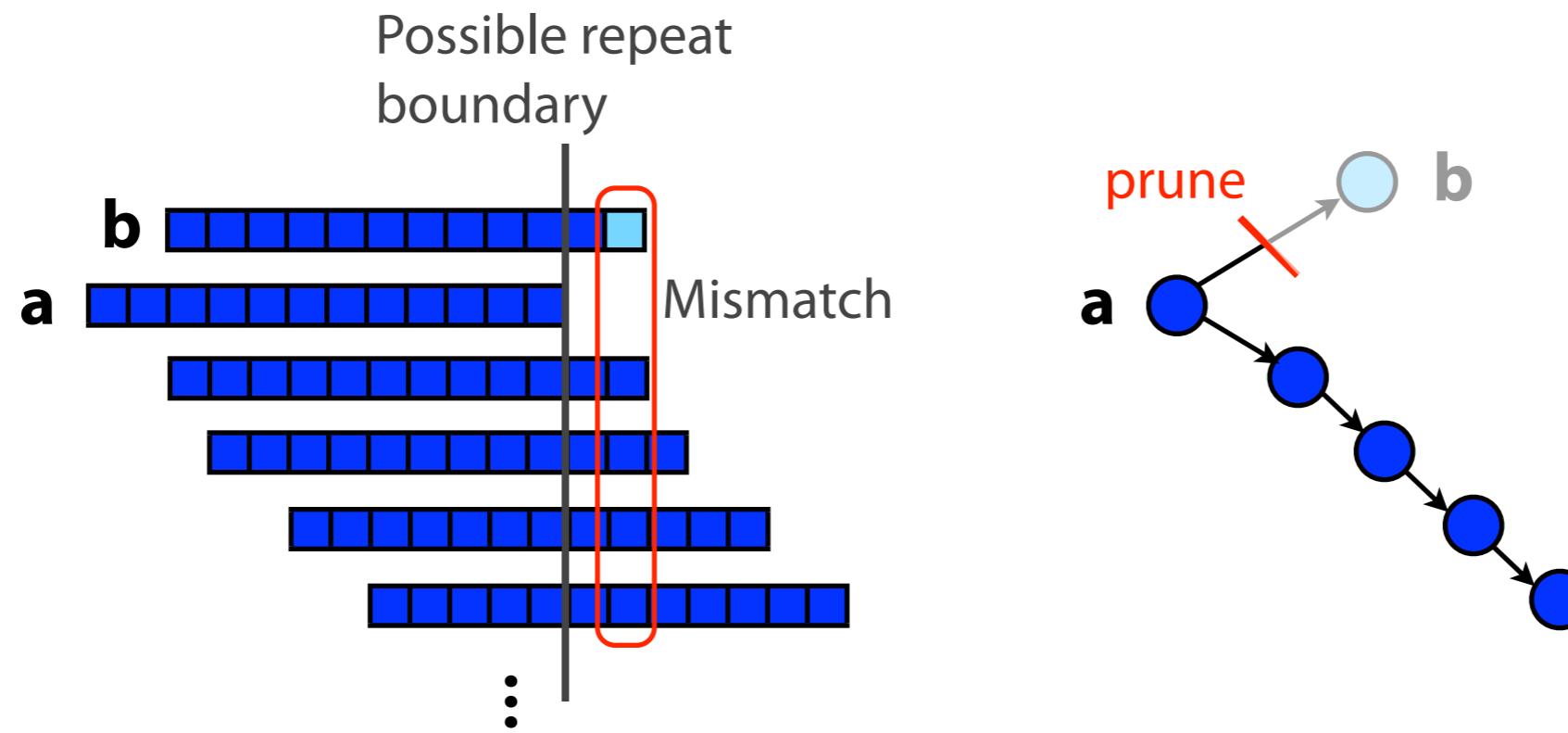
Contig 2

turn_there_is_a_season

Unresolvable repeat

Layout

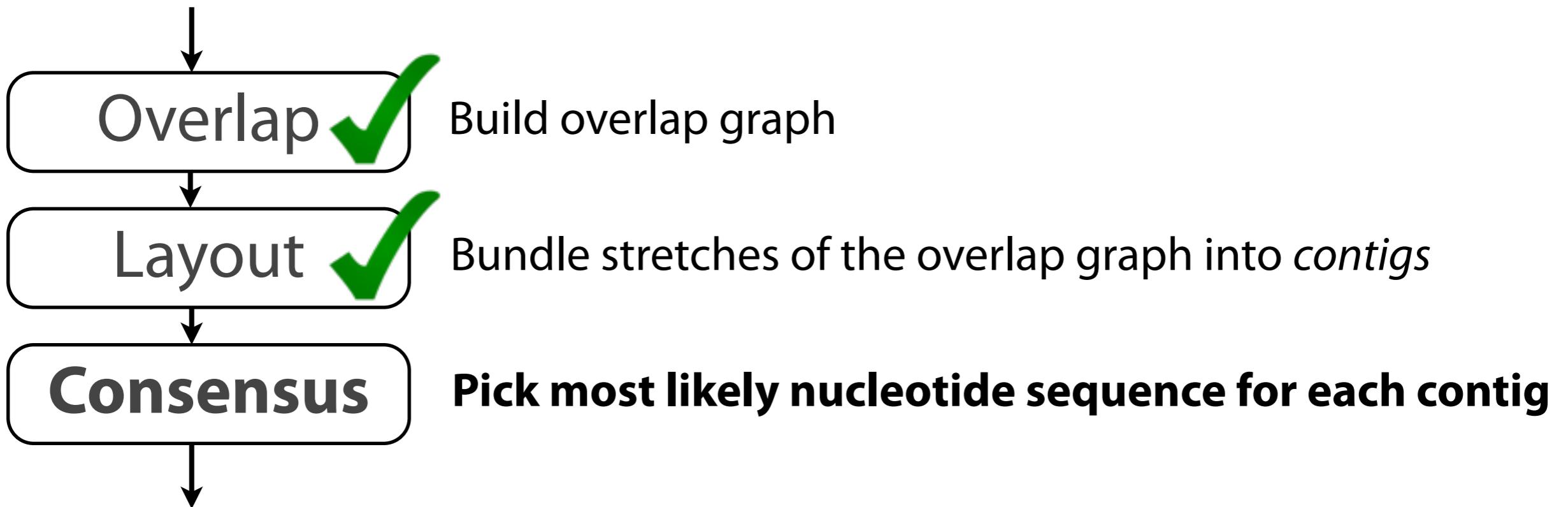
In practice, layout step also has to deal with spurious subgraphs, e.g. because of sequencing error



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**.

Modern assemblers are full of such “heuristics” — wisdom gained from running them on a lot of data.

Overlap Layout Consensus



Consensus

TAGATTACACAGATTACTGA TTGATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAAACTA
TAG TTACACAGATTATTGACTTCATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAA CTA

↓ ↓ ↓ ↓ ↓

TAGATTACACAGATTACTGACTTGATGGCGTAA CTA



Take reads that make up a contig and line them up

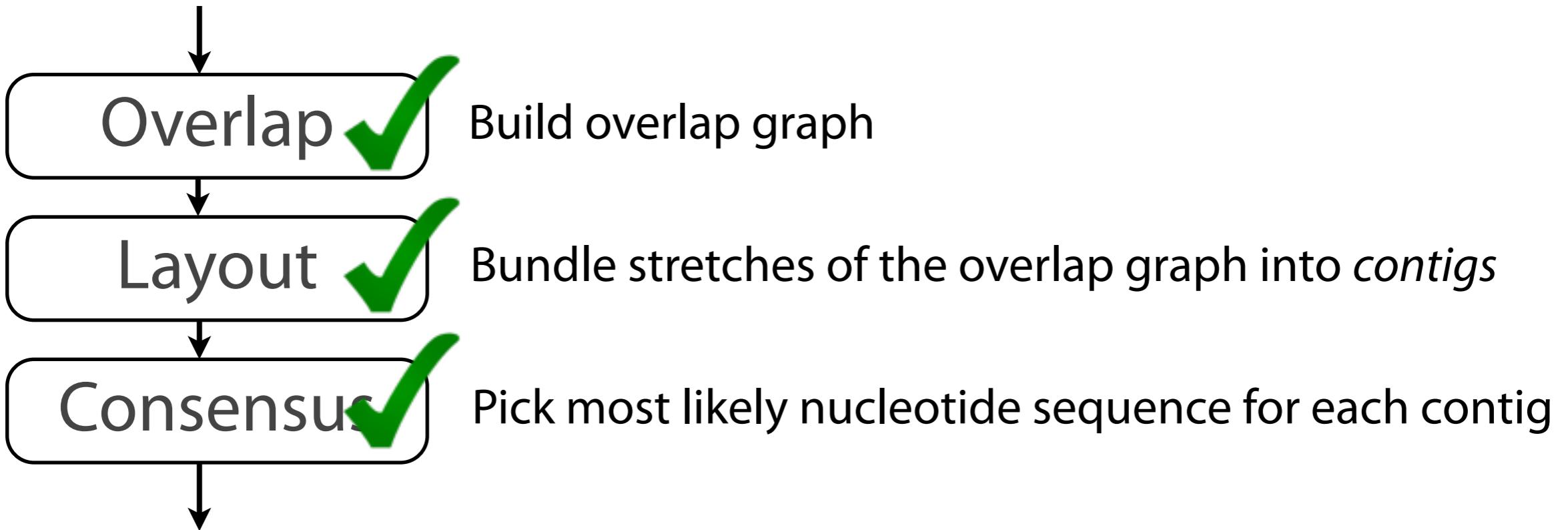
Take *consensus*, i.e. majority vote

At each position, ask: what nucleotide (and/or gap) is here?

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

Say the true genotype is AG, but we have a high sequencing error rate and only about 6 reads covering the position.

Overlap Layout Consensus



What's the main drawback of OLC?

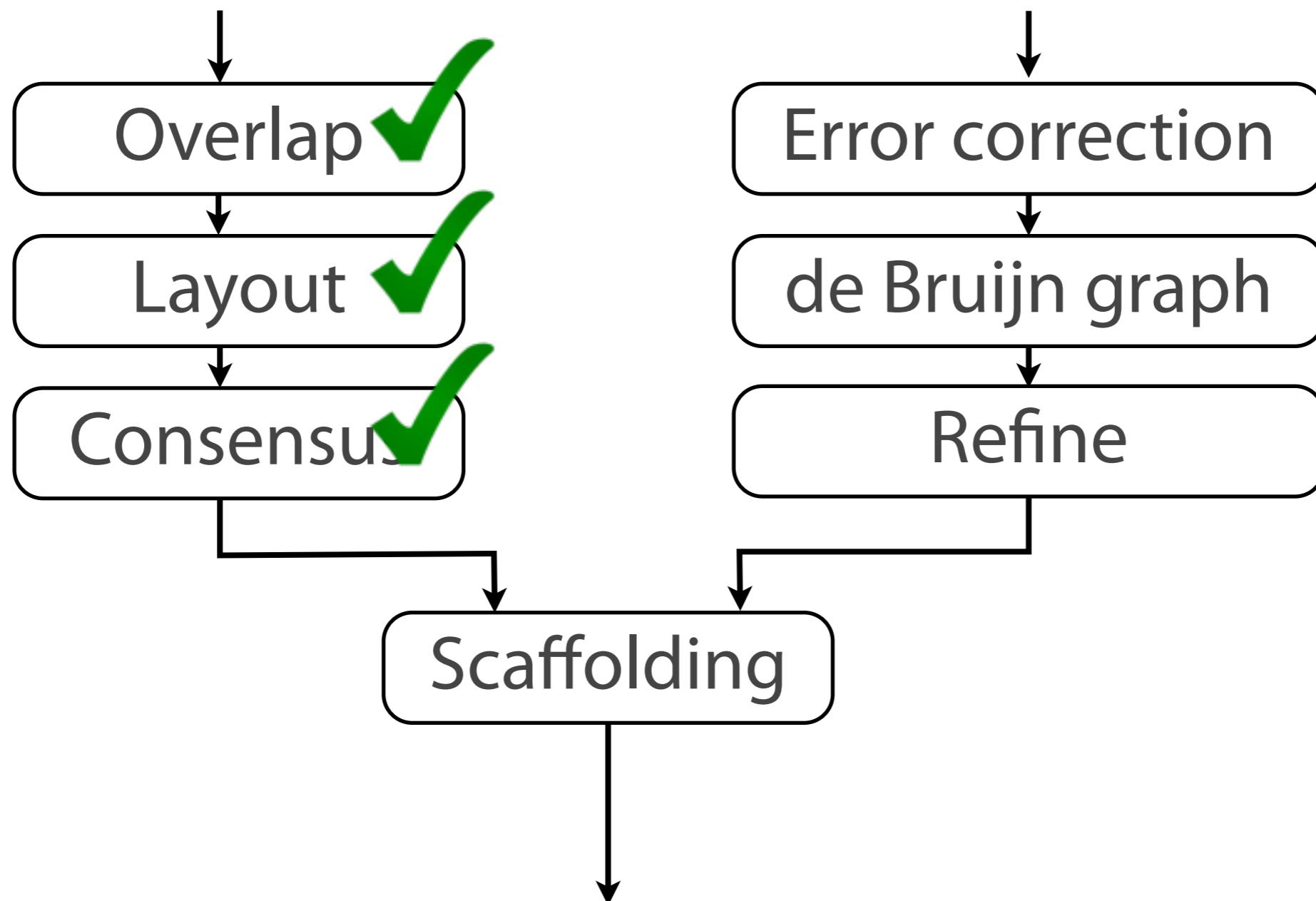
Building overlap graph is *slow*. We saw $O(N + a)$ and $O(N^2)$ approaches

2nd-generation sequencing datasets are ~ 100s of millions or billions of reads, hundreds of billions of nucleotides total

Assembly alternatives

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

Alternative 2: de Bruijn graph (DBG) assembly



Scaffolding with mate pair information

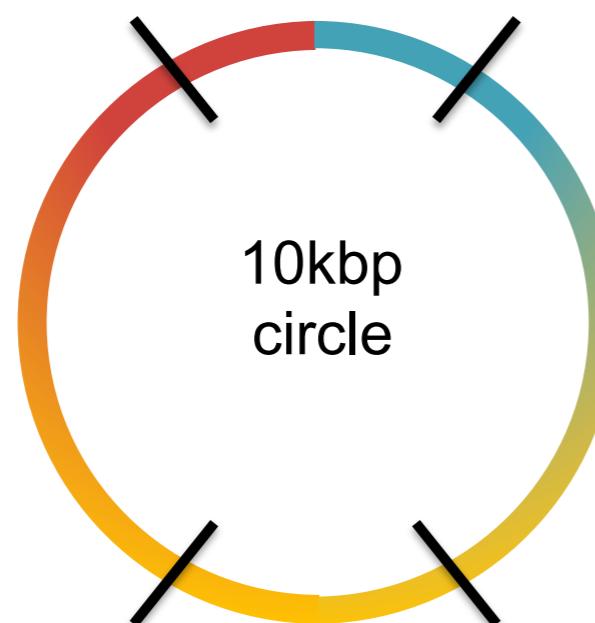
Paired-end sequencing

- Read one end of the molecule, flip, and read the other end
- Generate pair of reads separated by up to 500bp with inward orientation



Mate-pair sequencing

- Circularize long molecules (1-10kbp), shear into fragments, & sequence
- Mate failures create short paired-end reads



2x100 @ ~10kbp (outies)

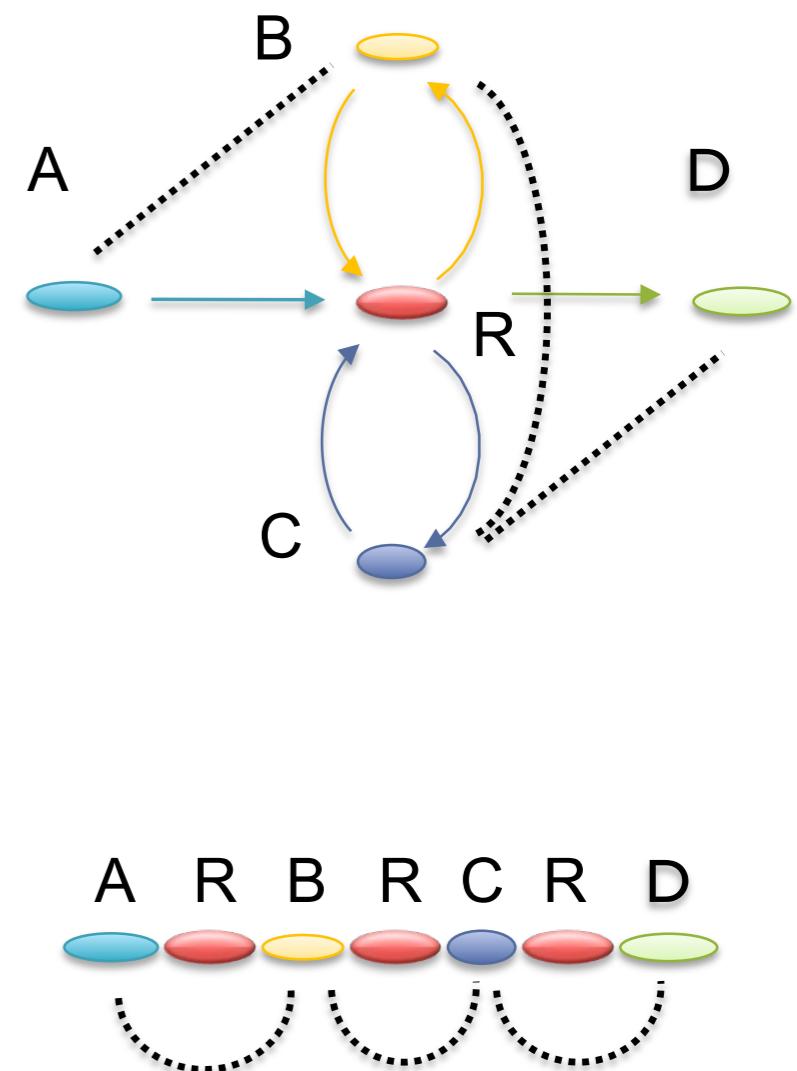


2x100 @ 300bp (innies)



Scaffolding

- Initial contigs (aka unipaths, unitigs) terminate at
 - Coverage gaps: especially extreme GC
 - Conflicts: errors, repeat boundaries
- Use mate-pairs to resolve correct order through assembly graph
 - Place sequence to satisfy the mate constraints
 - Mates through repeat nodes are tangled
- Final scaffold may have internal gaps called sequencing gaps
 - We know the order, orientation, and spacing, but just not the bases. Fill with Ns instead



Assembly alternatives

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

Alternative 2: de Bruijn graph (DBG) assembly

