

Genome Assembly Algorithms

Dr. Jared Simpson
Ontario Institute for Cancer Research
&
Department of Computer Science
University of Toronto

Introduction

- Last month: algorithms to map and align reads to a reference genome
- What do we do if we don't have a reference genome?
 - we need to reconstruct the sequence of the genome from the reads
 - this is called *de novo* genome assembly and we'll discuss algorithms to solve this problem for the next two weeks

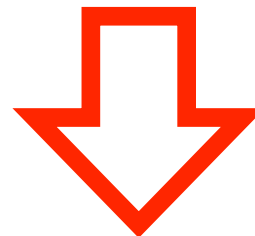
Assembly

Reads



+

Reference genome



Input DNA



How to assemble
puzzle without the
benefit of knowing
what the finished
product looks like?

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)

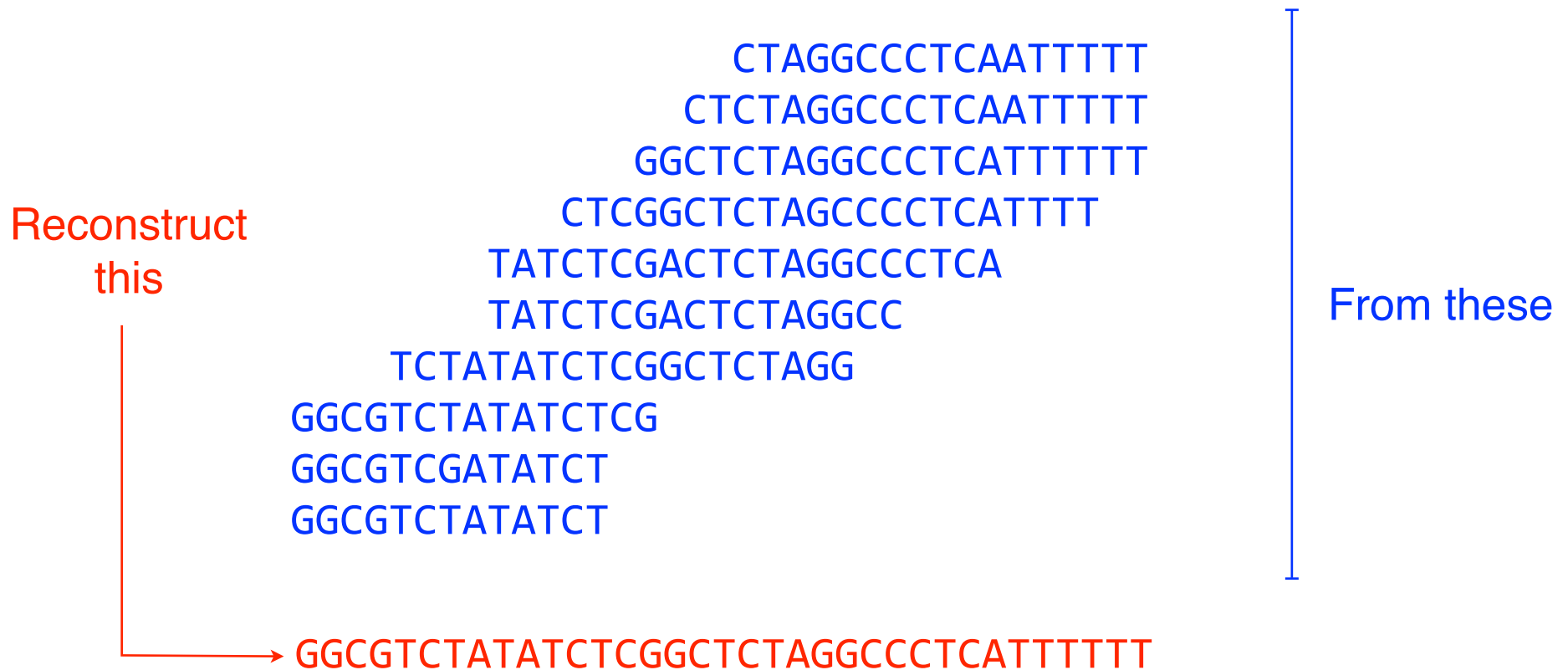
Input: GCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Copy: GCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

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

Assembly

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



Assembly

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

Reconstruct
this

CTAGGCCCTCAATTTTT
GGCGTCTATATCT
CTCTAGGCCCTCAATTTTT
TCTATATCTCGGCTCTAGG
GGCTCTAGGCCCTCATTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
GGCGTCGATATCT
TATCTCGACTCTAGGCC
GGCGTCTATATCTCG

From these

GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTT

Assembly

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

CTAGGCCCTCAATTTTT
CTCTAGGCCCTCAATTTTT
GGCTCTAGGCCCTCATTTTT
CTCGGCTCTAGCCCCTCATTTT
TATCTCGACTCTAGGCCCTCA
TATCTCGACTCTAGGCC
TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT

177 nucleotides

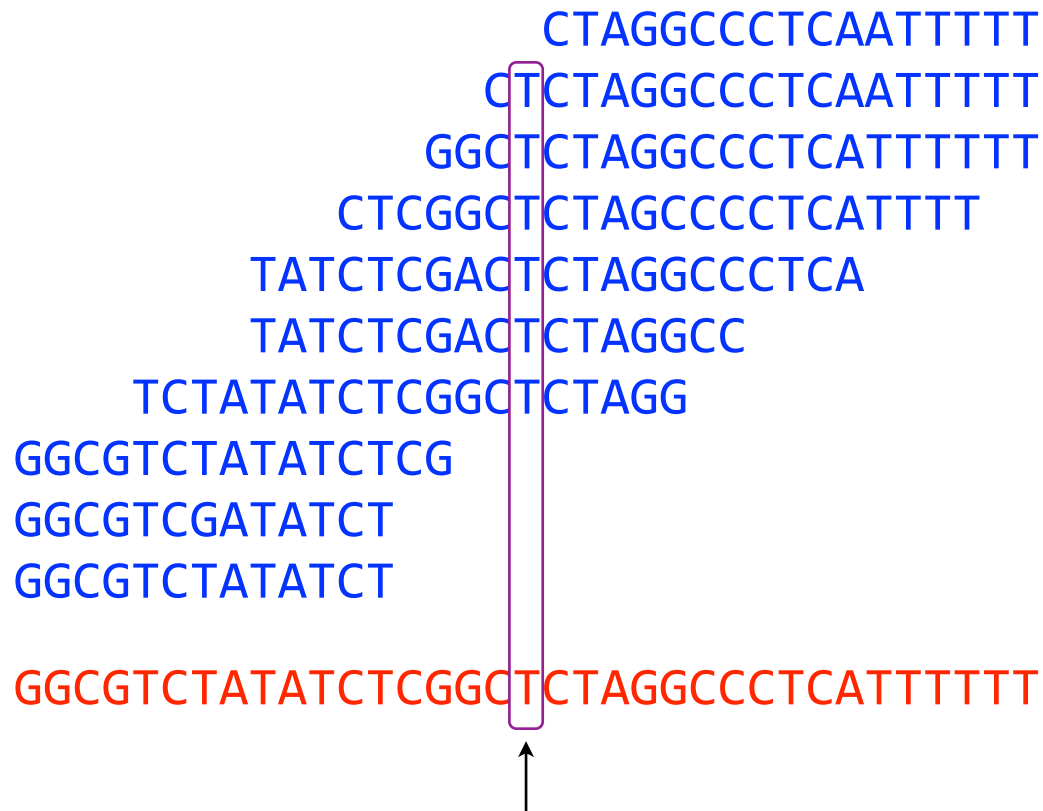
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

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
GGCTCTAGGCCCTCATTTT
CTCGGCTCTAGCCCCTCATTT
TATCTCGACTCTAGGCCCTCA
TATCTCGACTCTAGGCC
TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTT

Coverage at this position = 6

Assembly

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

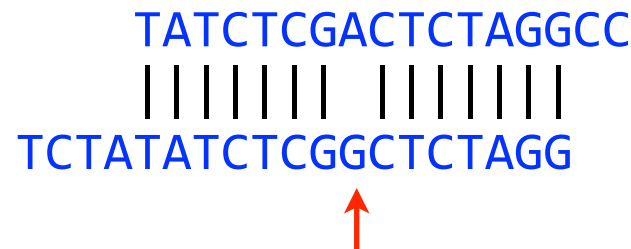
```
TATCTCGACTCTAGGCC
||| ||| ||| |||
TCTATATCTCGGCTCTAGG
```

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

```
TATCTCGACTCTAGGCC
TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

Assembly

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



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: TATCTCGACTCTAGGCC
 ||||| |||||

Read from Father: TCTATATCTCGGCTCTAGG

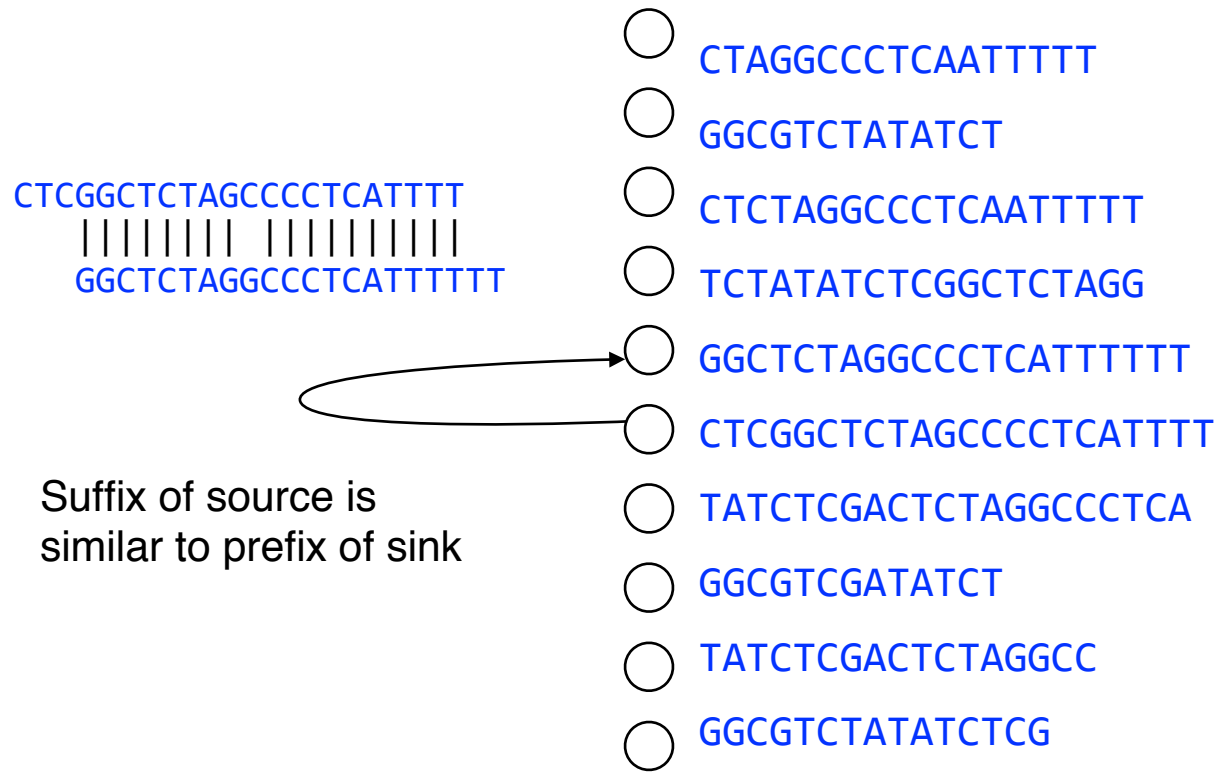
Sequence from Mother: TCTATATCTCGACTCTAGGCC

Sequence from Father: TCTATATCTCGGCTCTAGGCC

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

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*.

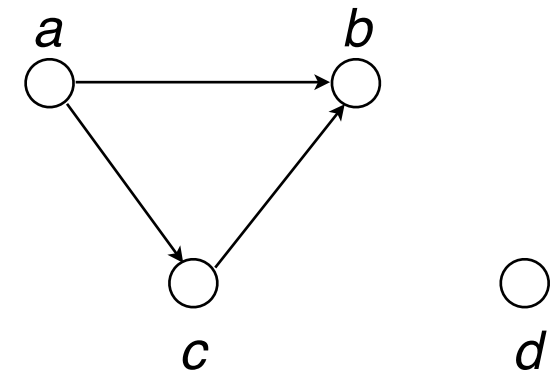
Vertex is drawn as a circle

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

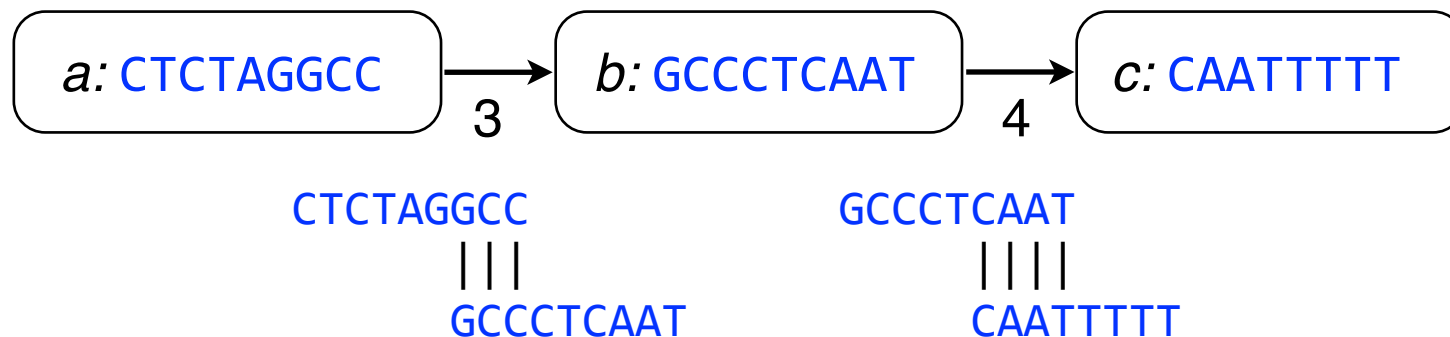
Overlap graph

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

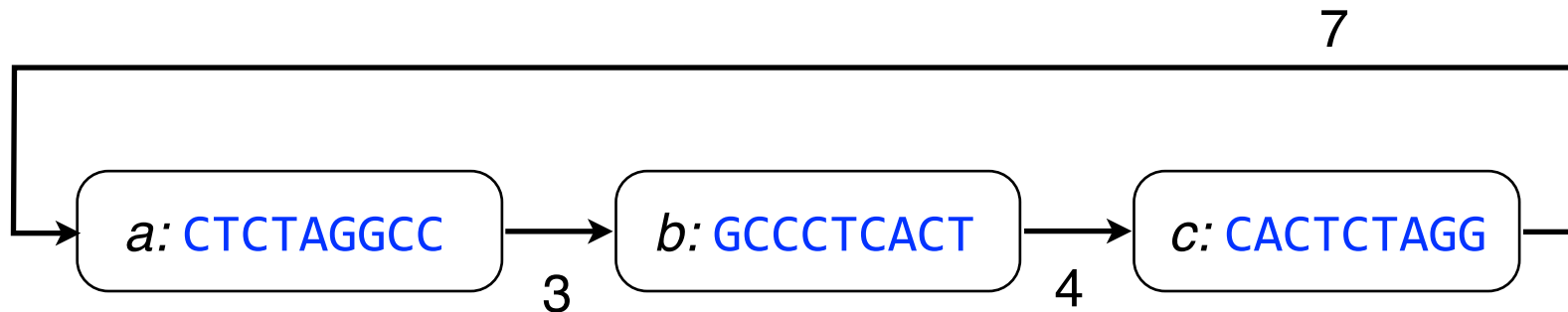
Vertices (reads): { a : CTCTAGGCC, b : GCCCTCAAT, c : CAATTTT }

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



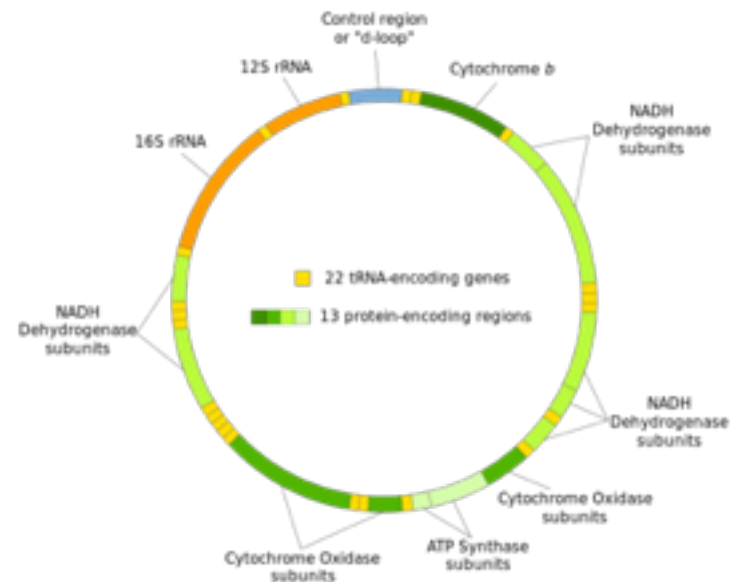
Overlap graph

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

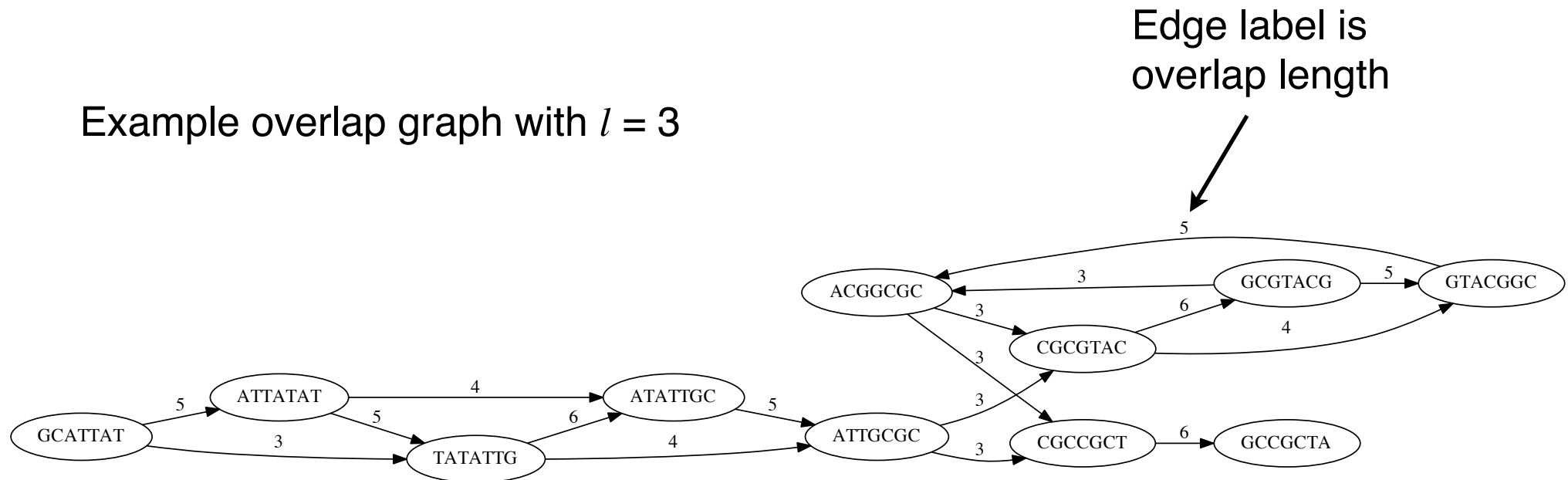
Finding overlaps: implementation

```
def suffixPrefixMatch(x, y, k):  
    ''' Return length of longest suffix of x of length at least k that  
        matches a prefix of y. Return 0 if there no suffix/prefix  
        match has length at least k. '''  
    if len(x) < k or len(y) < k:  
        return 0  
    idx = len(y) # start at the right end of y  
    # Search right-to-left in y for length-k suffix of x  
    while True:  
        hit = string.rfind(y, x[-k:], 0, idx)  
        if hit == -1: # not found  
            return 0  
        ln = hit + k  
        # See if match can be extended to include entire prefix of y  
        if x[-ln:] == y[:ln]:  
            return ln # return length of prefix  
        idx = hit + k - 1 # keep searching to left in Y  
    return -1
```

Python example: <http://nbviewer.ipython.org/7089885>

Finding overlaps

Example overlap graph with $l = 3$



Original string: **GCATTATATATTGCGCGTACGGCGCCGCTACA**

Formulating the assembly problem

Finding overlaps is important, and we'll return to it, but our ultimate goal is to recreate (assemble) the genome

How do we formulate this problem?

First attempt: the *shortest common superstring (SCS)* problem

Shortest common superstring

Given a collection of strings S , find $SCS(S)$: 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: BAAAABBBBAABAABBBBBBAAABAB
|-----24-----|

$SCS(S)$: 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)

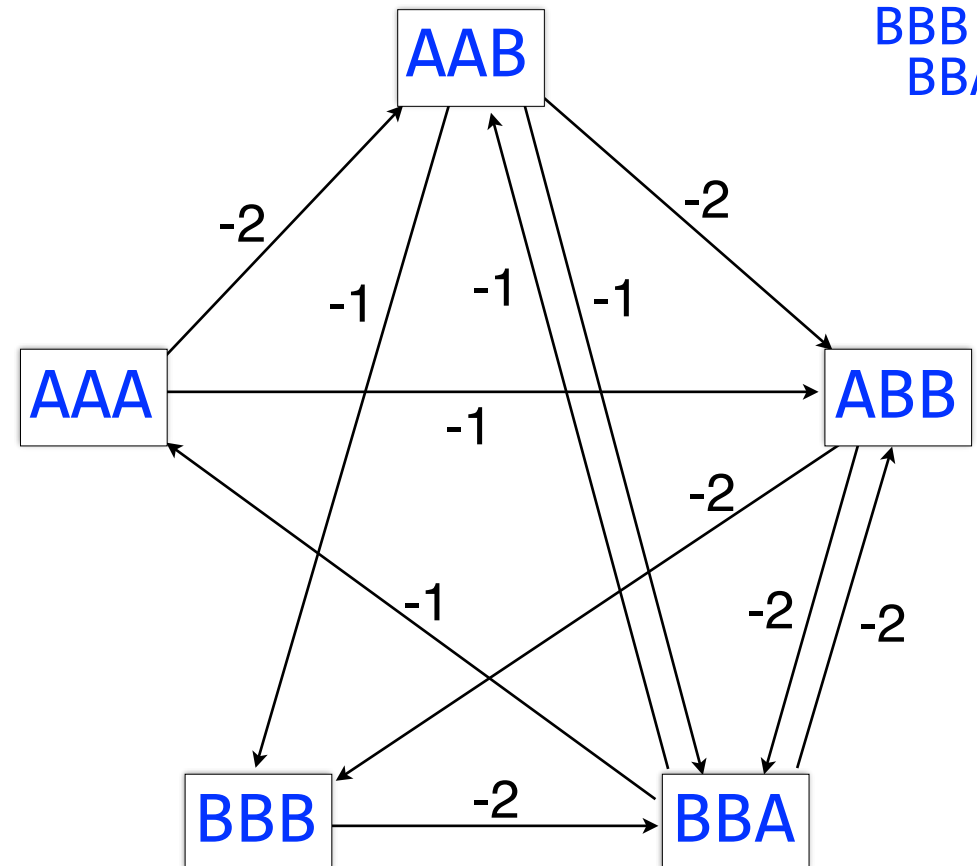
SCS corresponds to a path that visits every node once, minimizing total cost along path

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

S: AAA AAB ABB BBB BBA

SCS(S): AAABBBBA

AAA
AAB
ABB
BBB
BBA



Shortest common superstring

Say we disregard edge weights and just look for a path that visits each node exactly once

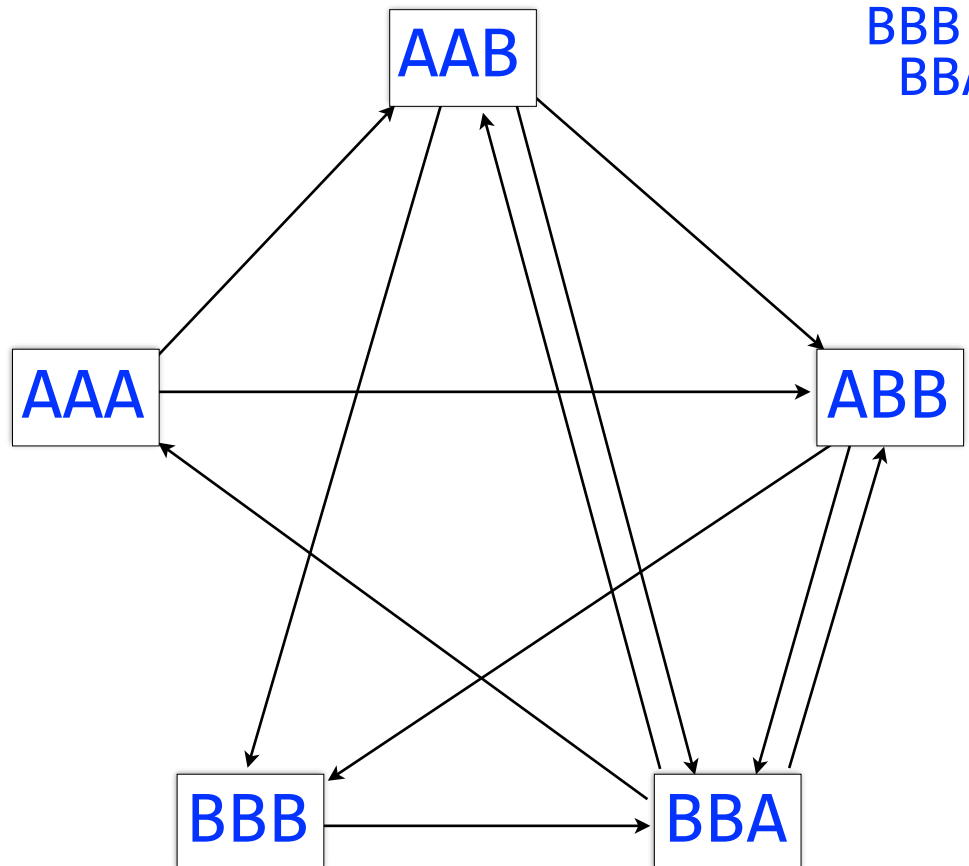
That's the *Hamiltonian Path* problem: NP-complete

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

S : AAA AAB ABB BBB BBA

$SCS(S)$: AAABBBBA

AAA
AAB
ABB
BBB
BBA



Shortest common superstring

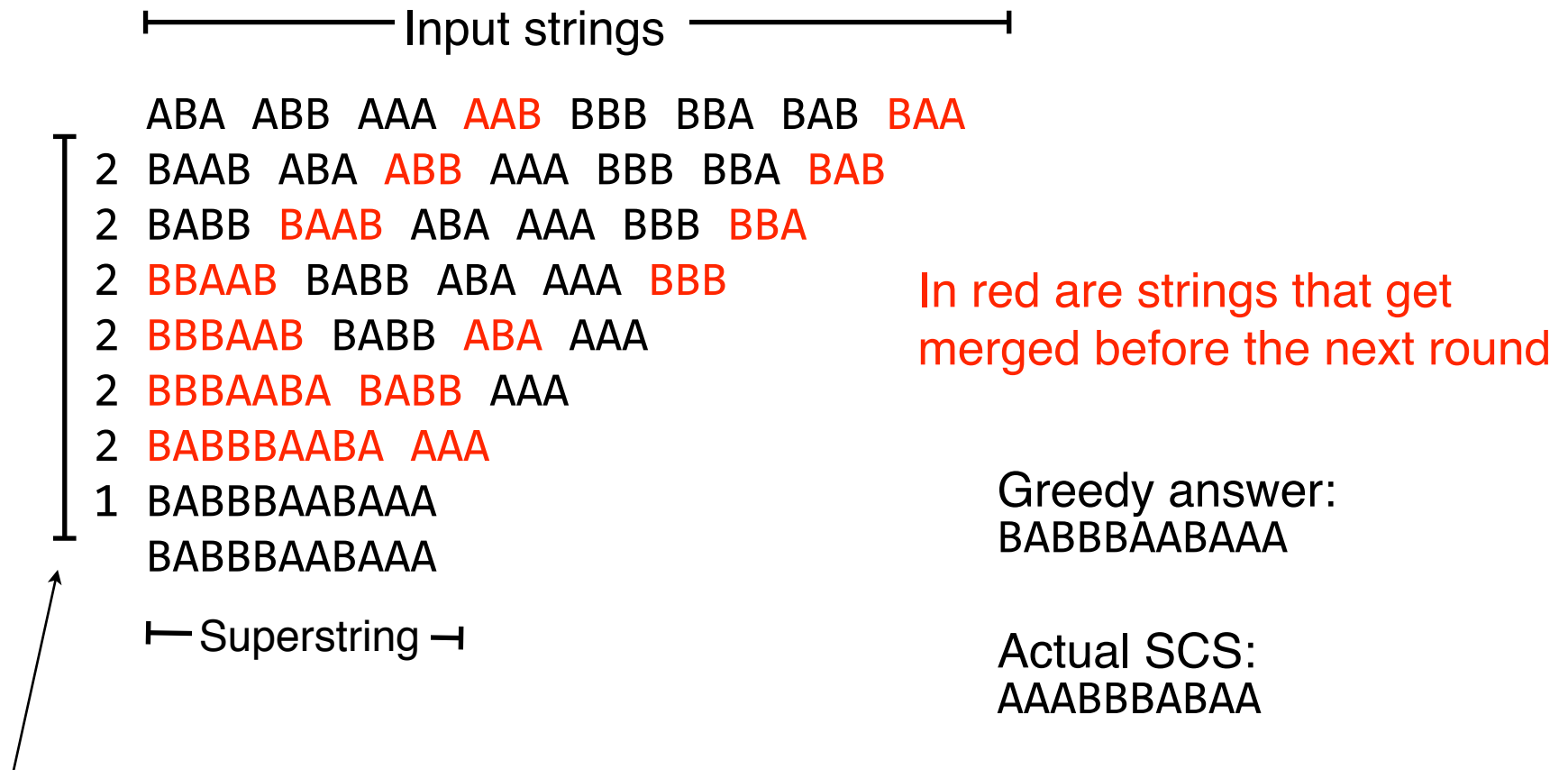
Let's take the hint give up on finding the *shortest possible* superstring

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

Another setup for Greedy-SCS: assemble all substrings of length 6 from string `a_long_long_long_time`. $l = 3$.

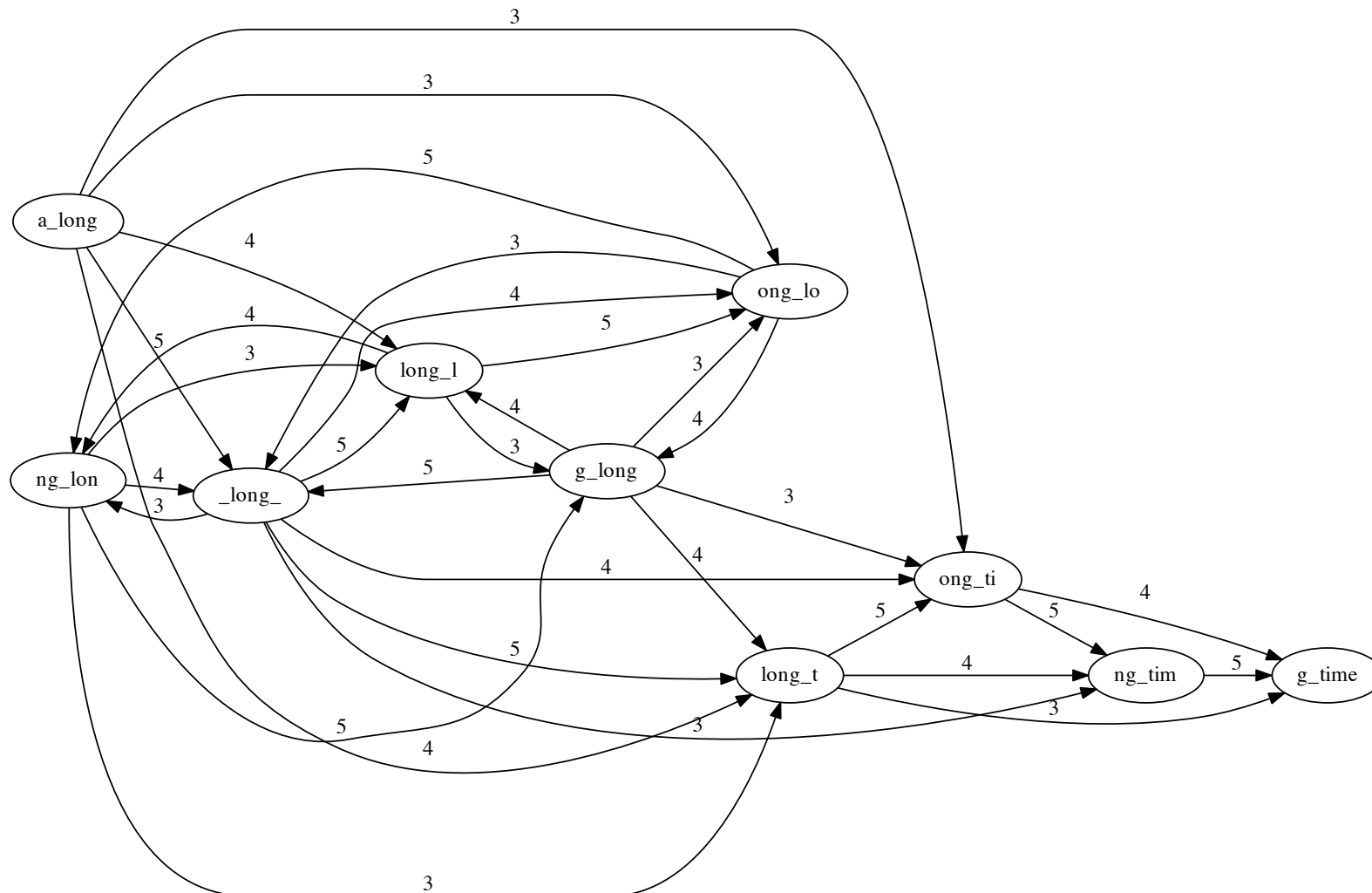
```
ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long g_time ng_tim
5 ng_time ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long
5 ng_time g_long_ ng_lon a_long long_l ong_ti ong_lo long_t
5 ng_time long_ti g_long_ ng_lon a_long long_l ong_lo
5 ng_time ong_lon long_ti g_long_ a_long long_l
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?

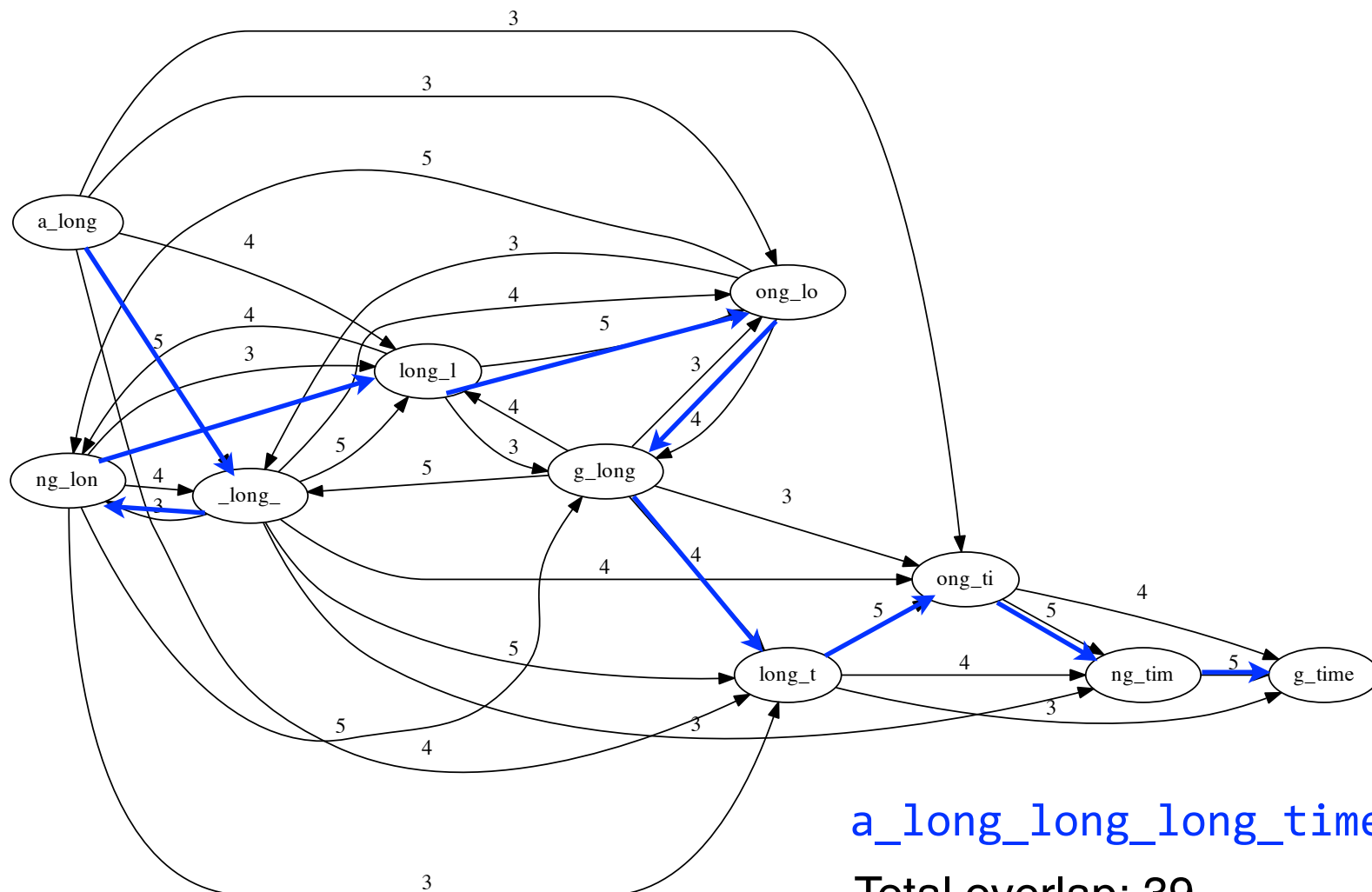
Shortest common superstring: greedy

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



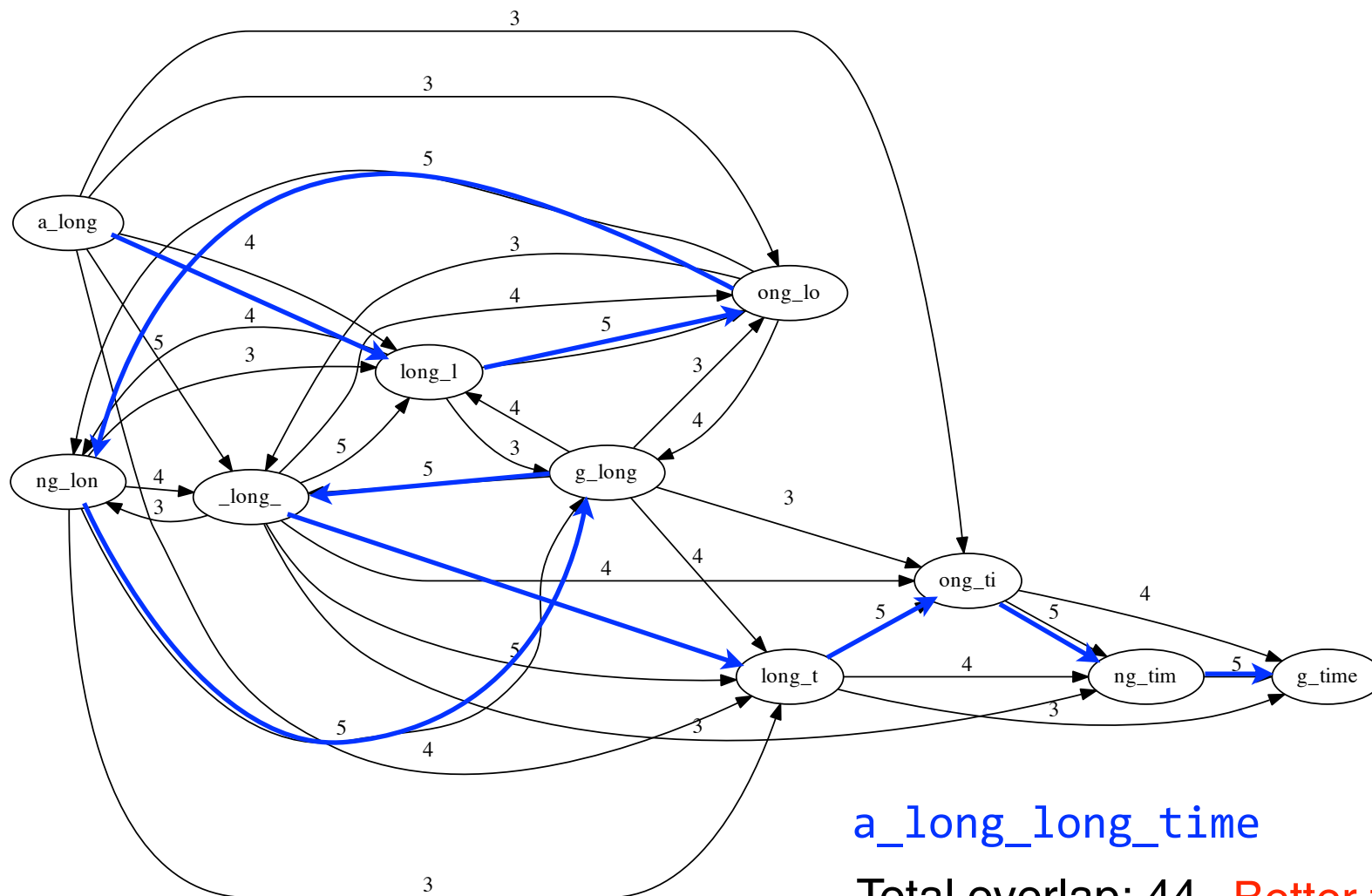
Shortest common superstring: greedy

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_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



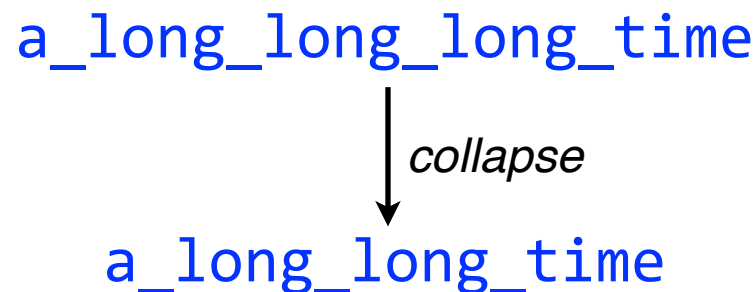
One length-8 substring spans all three **longs**

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*

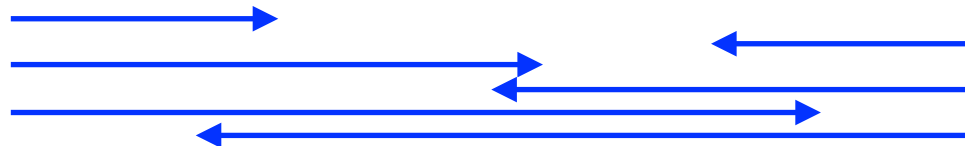
Longer and longer substrings allow us to “anchor” more of the repeat to its non-repetitive context:

swinging_and_the_ringing_of_the_bells_bells_bells_bells_bells



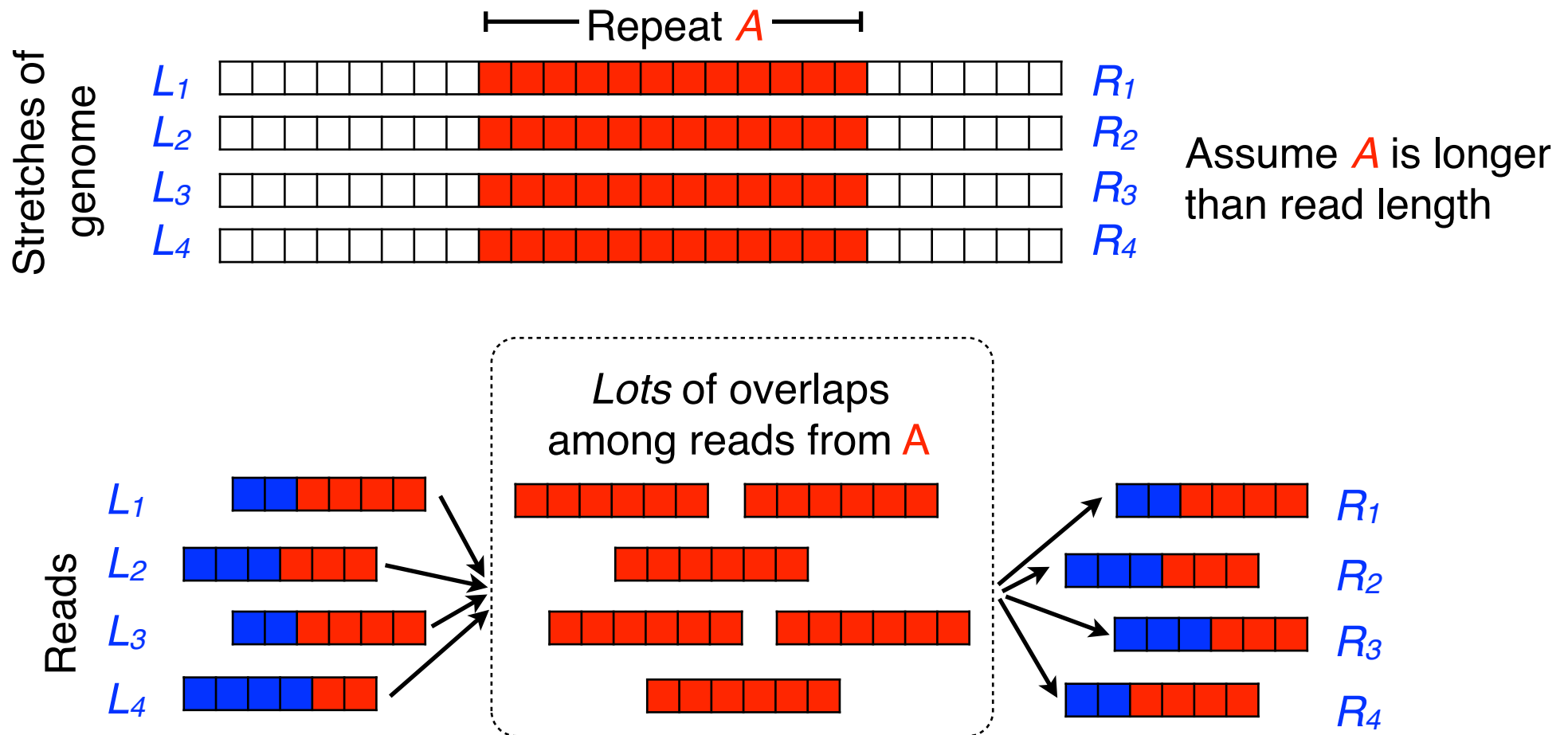
Often we can “walk in” from both sides. When we meet in the middle, the repeat is resolved:

ringing_of_the_bells_bells_bells_bells_bells_to_the_rhyhming



Repeats

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.

Real-world assembly methods

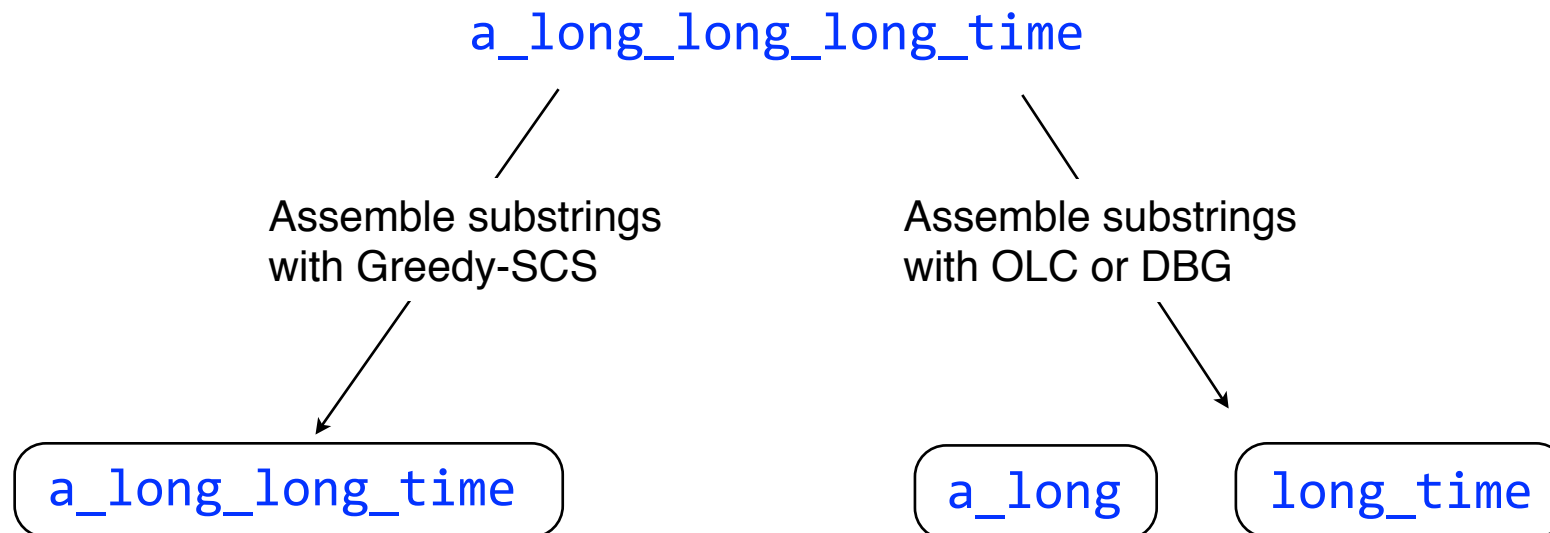
OLC: Overlap-Layout-Consensus assembly

DBG: De Bruijn graph assembly

Both handle unresolvable repeats by essentially *leaving them out*

Unresolvable repeats break the assembly into fragments

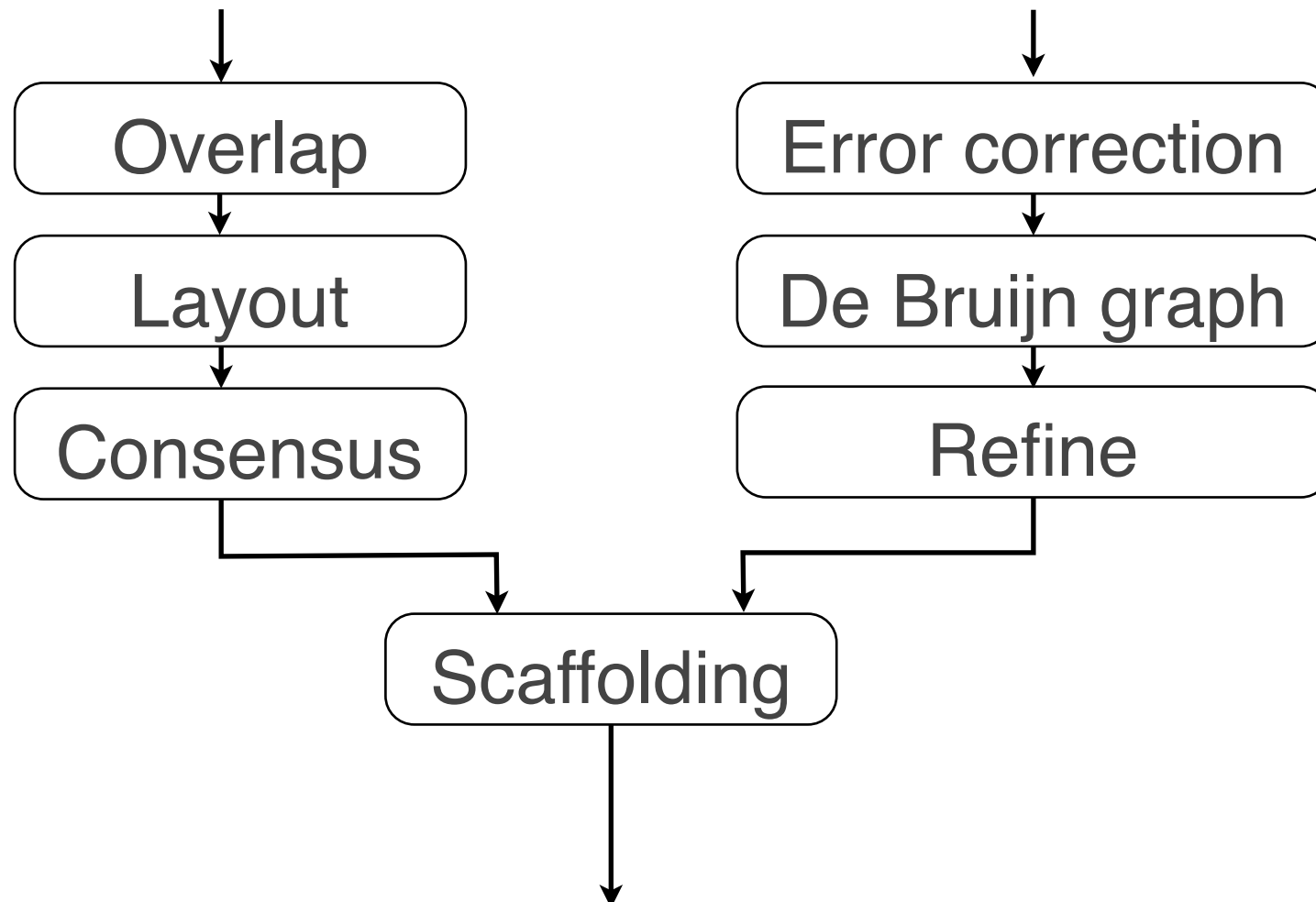
Fragments are *contigs* (short for *contiguous*)



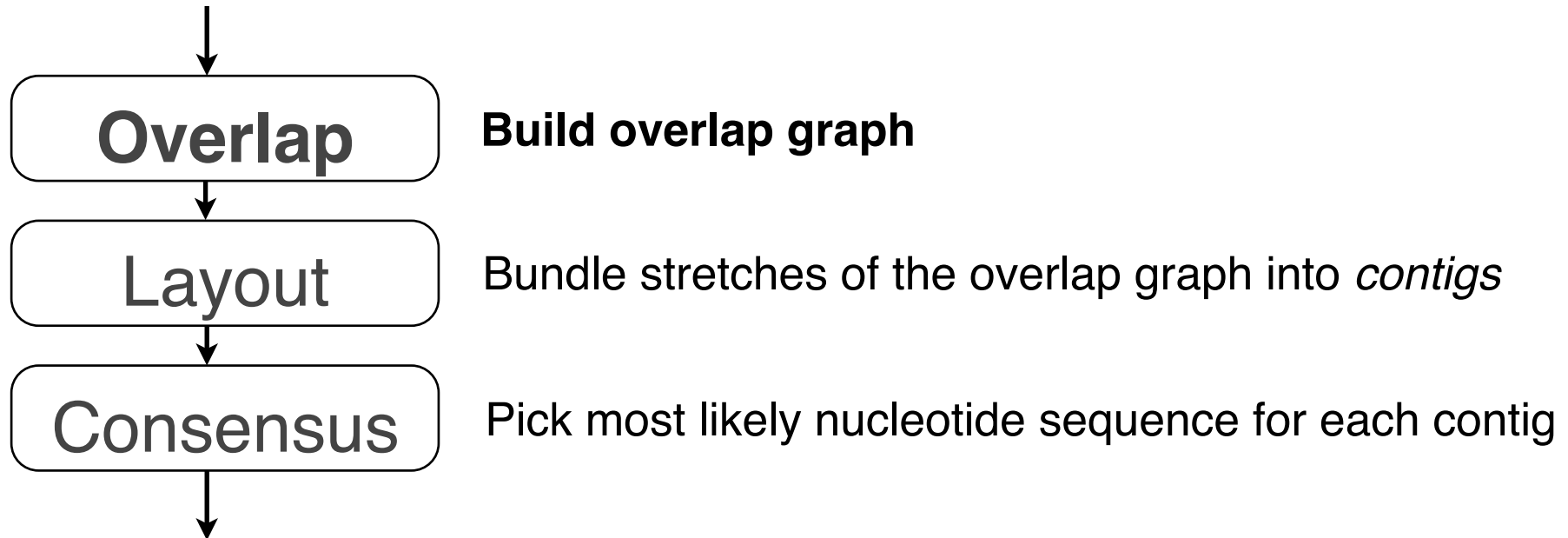
Assembly alternatives

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

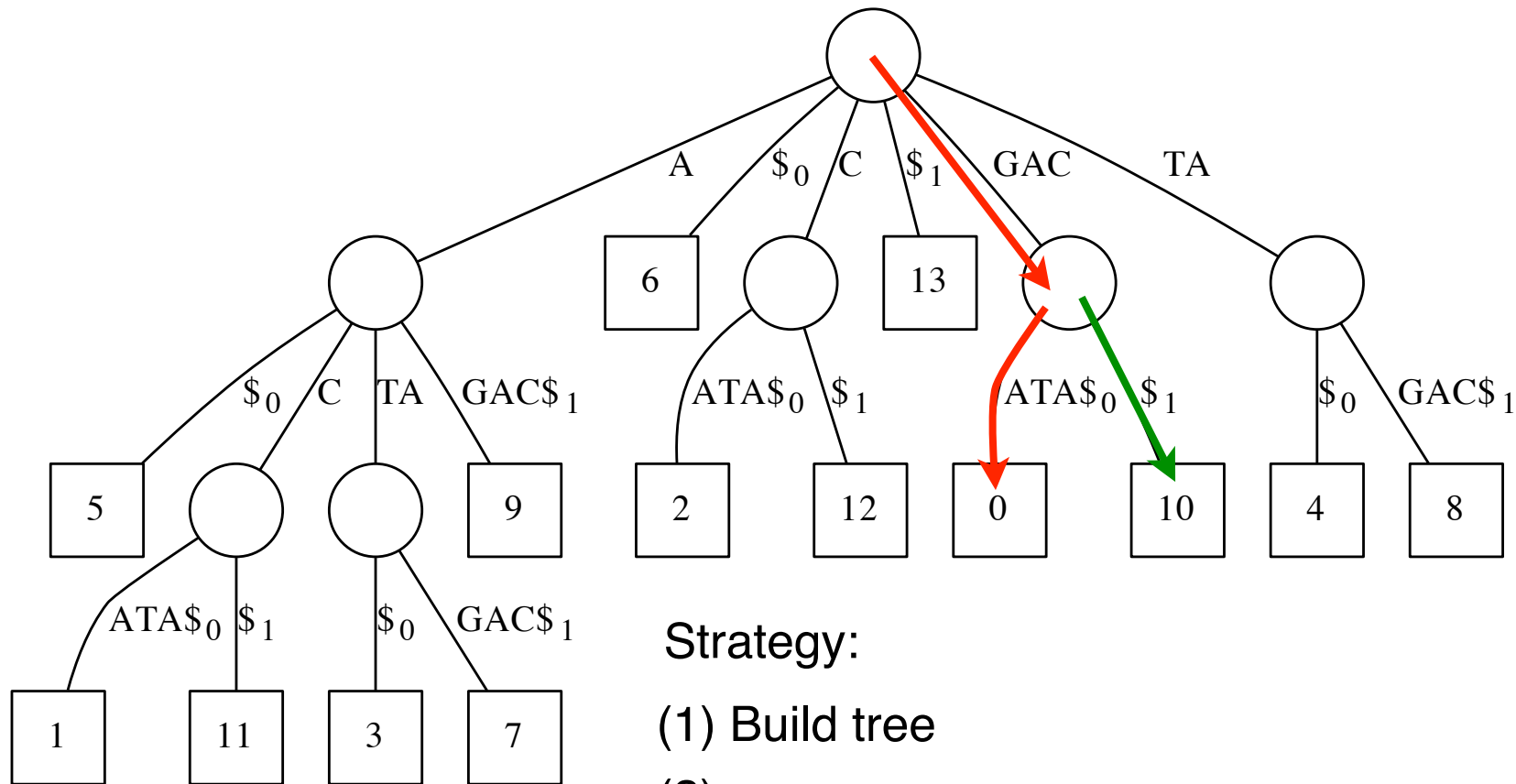
Hint: Build a generalized suffix tree of the strings in S

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\$₀ATAGAC\$₁



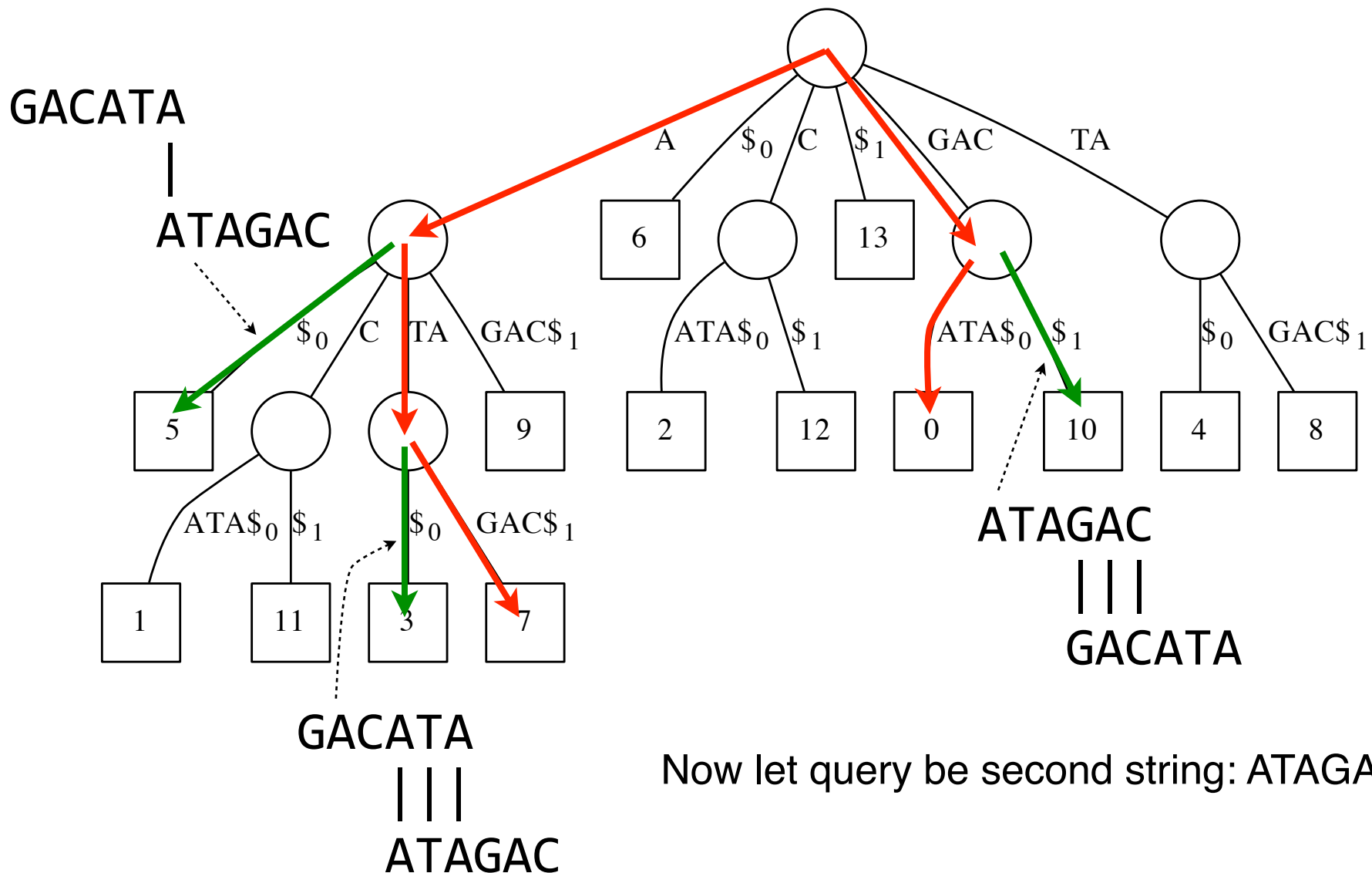
Strategy:

(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





Time to build generalized suffix tree:	$O(N)$	d^2 doesn't appear explicitly, but a is $O(d^2)$ in worst case
... to walk down red paths:	$O(N)$	
... to find & report overlaps (green):	$O(a)$	
Overall:	$O(N + a)$	

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

Finding overlaps with dynamic programming

Find the best alignment of a suffix of X to a prefix of Y

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

We'll 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		

But how do we force it to find prefix / suffix matches?

Finding overlaps with dynamic programming

Find the best alignment of a suffix of X to a prefix of Y

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

$$s(a, b)$$

	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	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

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

X : CTCGGCCCTAGG

Y : GGCTCTAGGCC

Finding overlaps with dynamic programming

Find the best alignment of a suffix of X to a prefix of Y

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

 $s(a, b)$

	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	8

Problem: **very short** matches got high scores by chance...

...which might obscure the **more relevant match**

Say we want to enforce minimum overlap length $l = 5$

Y

	-	G	G	C	T	C	T	A	G	G	C	C	C
-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
C	0	4	12	20	28	36	44	52	60	68	76	84	92
T	0	4	8	14	20	28	36	44	52	60	68	76	84
C	0	4	8	8	16	20	28	36	44	52	60	68	76
G	0	0	4	12	12	20	24	30	36	44	52	60	68
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	2	8	16	22	30	34	34	36	44
C	0	4	8	8	6	2	10	18	26	34	34	34	36
T	0	4	8	10	8	8	2	10	18	26	34	36	36
A	0	2	6	12	14	12	10	2	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	2	10	18	26

X

Finding overlaps with dynamic programming

Find the best alignment of a suffix of X to a prefix of Y

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

 $s(a, b)$

	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	8

Solve by initializing certain additional cells to ∞

Cells whose values changed highlighted in red

Now the relevant match is the best candidate

		Y												
		-	G	G	C	T	C	T	A	G	G	C	C	C
X	-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
	C	0	4	12	20	28	36	44	52	60	68	76	84	92
	T	0	4	8	14	20	28	36	44	52	60	68	76	84
	C	0	4	8	8	16	20	28	36	44	52	60	68	76
	G	0	0	4	12	12	20	24	30	36	44	52	60	68
	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	2	8	16	22	30	34	34	36	44
	C	0	4	8	8	6	2	10	18	26	34	34	34	36
	T	∞	4	8	10	8	8	2	10	18	26	34	36	36
	A	∞	12	6	12	14	12	10	2	10	18	26	34	40
	G	∞	20	12	10	16	18	16	10	0	10	18	26	34
	G	∞	∞	∞	∞	∞	20	22	18	10	2	10	18	26

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

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

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

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

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

But dynamic programming is more flexible, allowing mismatches and gaps

Real-world overlappers mix the two, using indexes to filter out vast majority of non-overlapping pairs, then using dynamic programming for remaining pairs

Finding overlaps

Overlapping is typically the slowest part of assembly

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

Finding overlaps

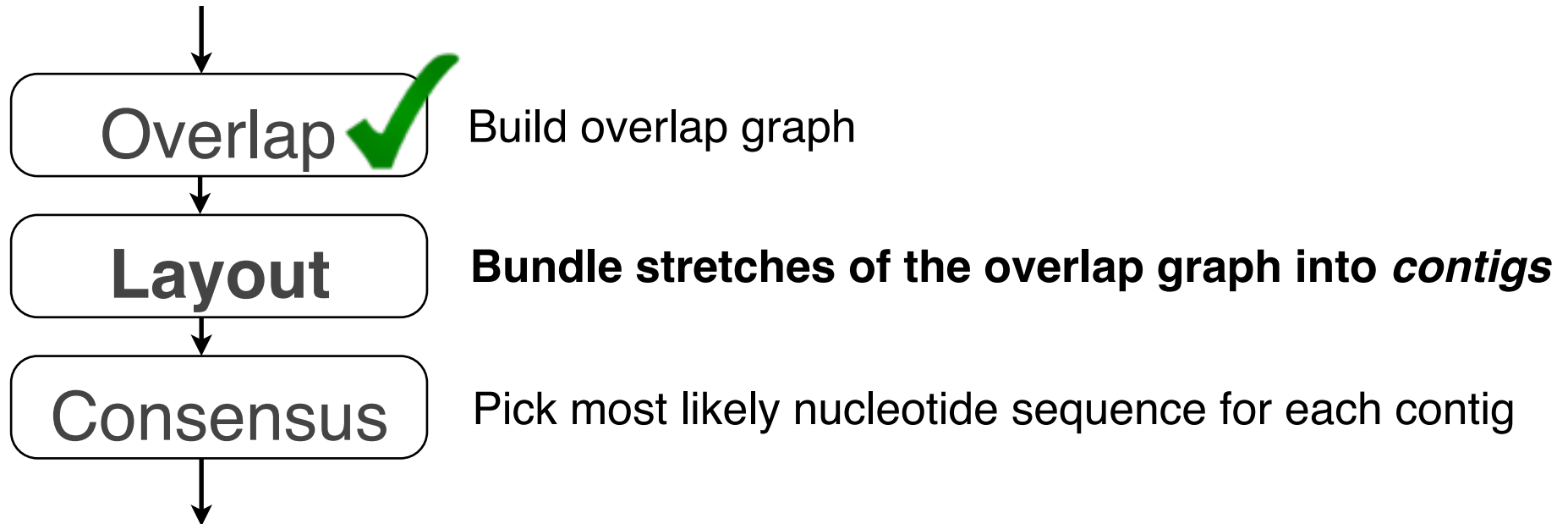
Celera Assembler's overlapper is probably the best documented:

Inverted substring indexes built on batches of reads

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

<http://wgs-assembler.sourceforge.net/wiki/index.php/RunCA#Overlapper>

Overlap Layout Consensus



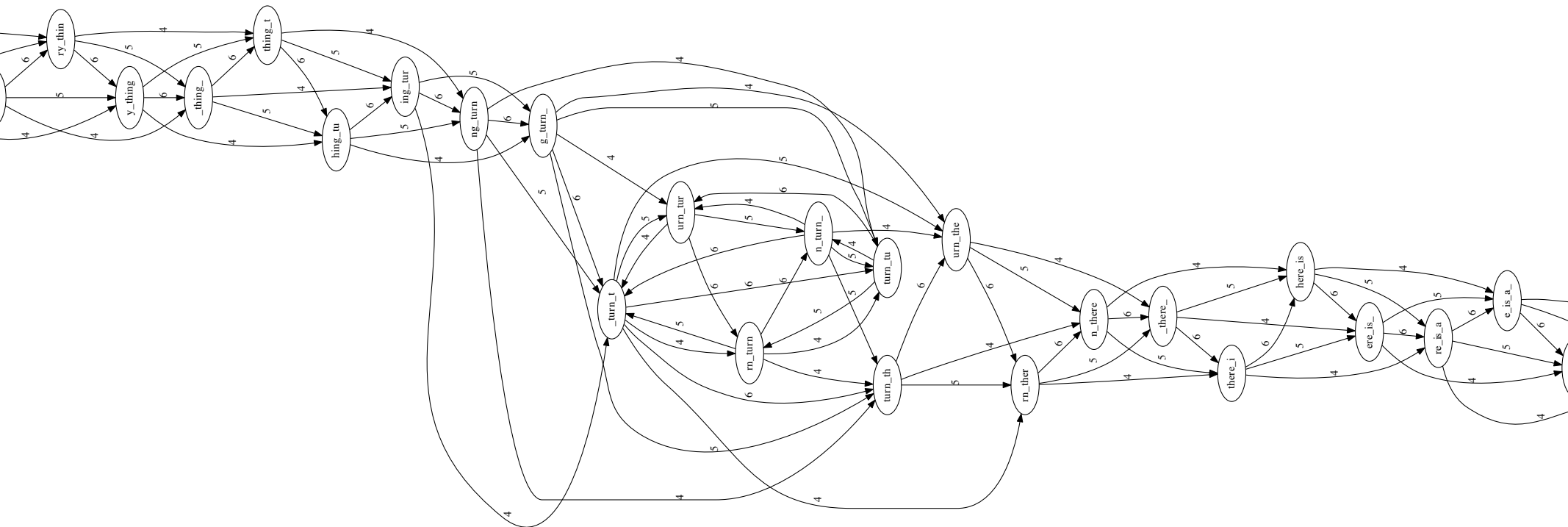
Layout

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

Below: part of the overlap graph for

`to_everything_turn_turn_turn_there_is_a_season`

$l = 4, k = 7$

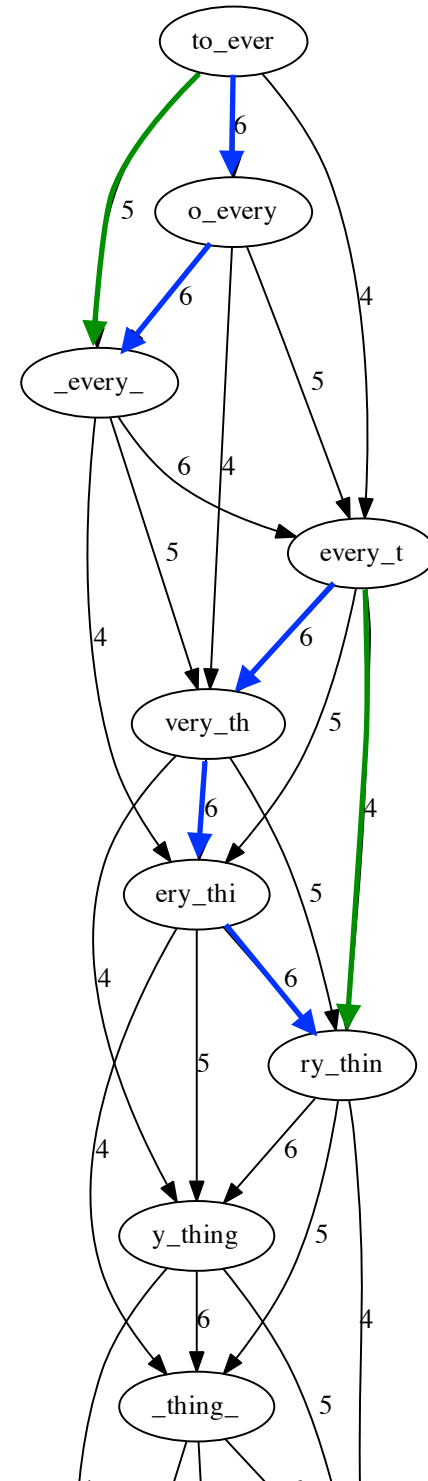


Layout

Anything redundant about this part of the overlap graph?

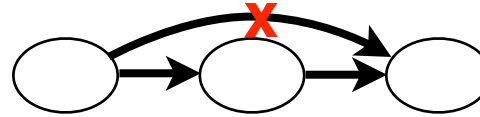
Some edges can be *inferred (transitively)* from other edges

E.g. **green** edge can be inferred from **blue**

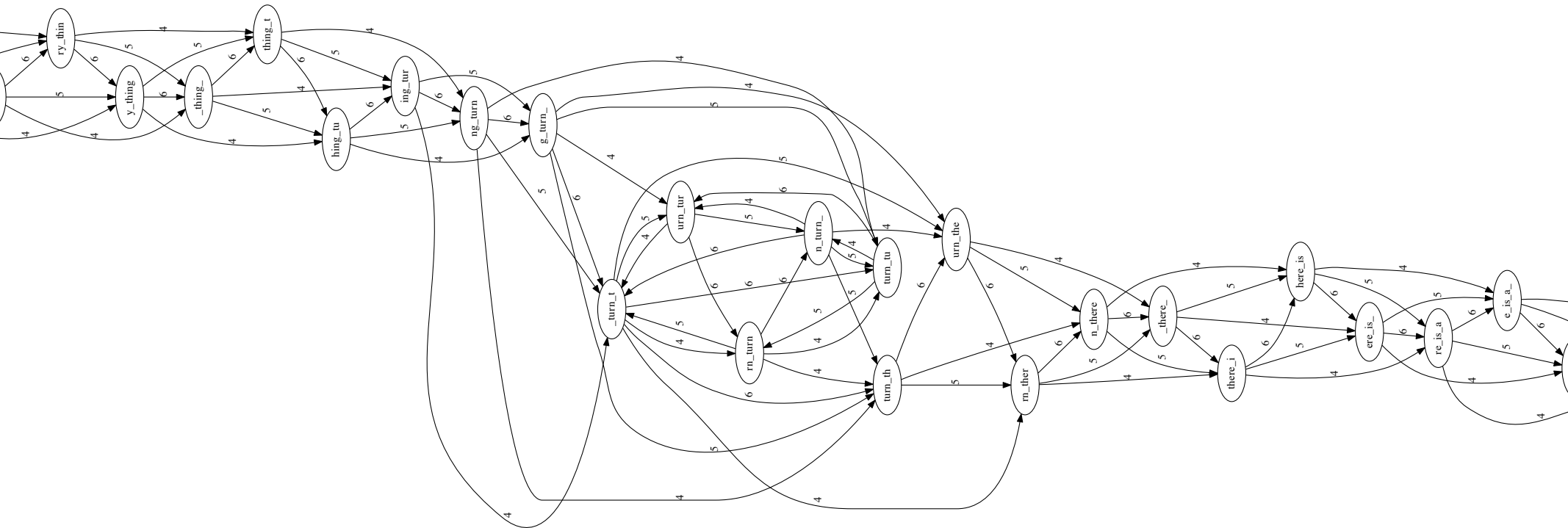


Layout

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

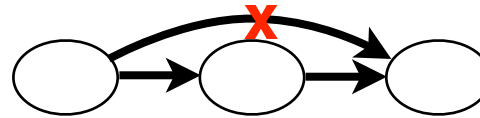


Before:

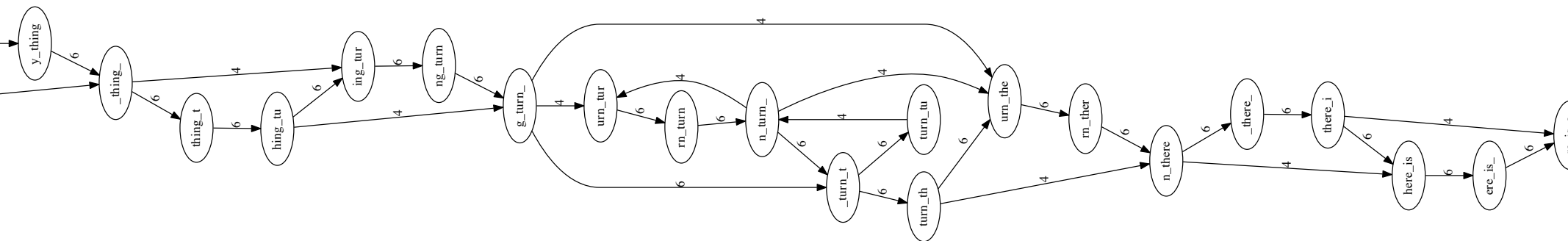


Layout

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

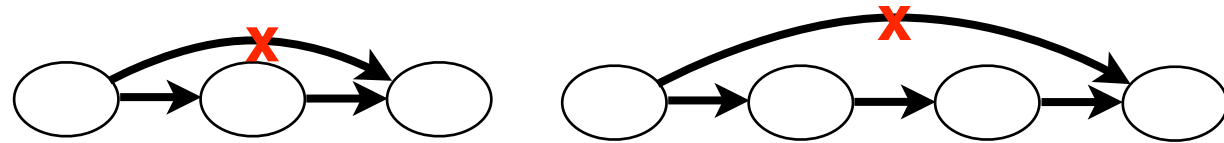


After:

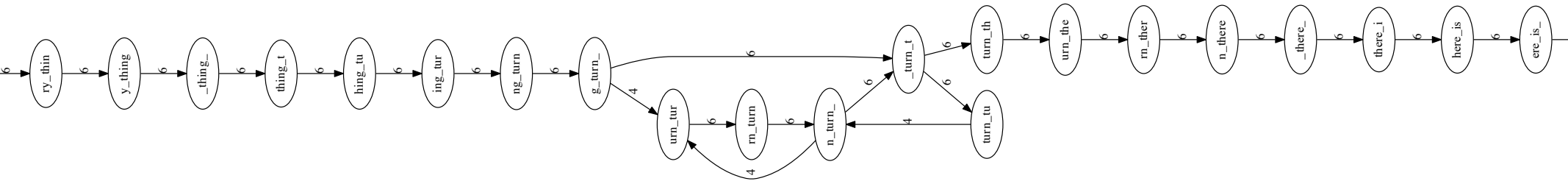


Layout

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



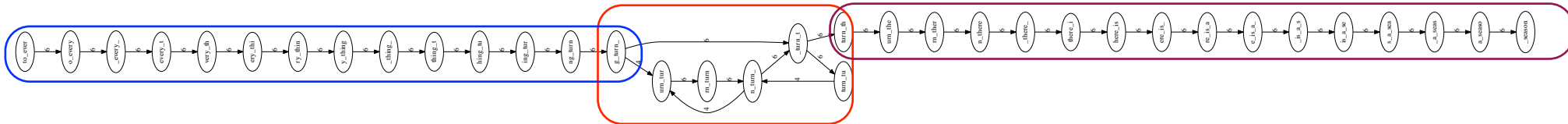
After:



Even simpler

Layout

Emit *contigs* corresponding to the non-branching stretches



Contig 1

to_every_thing_turn_

Contig 2

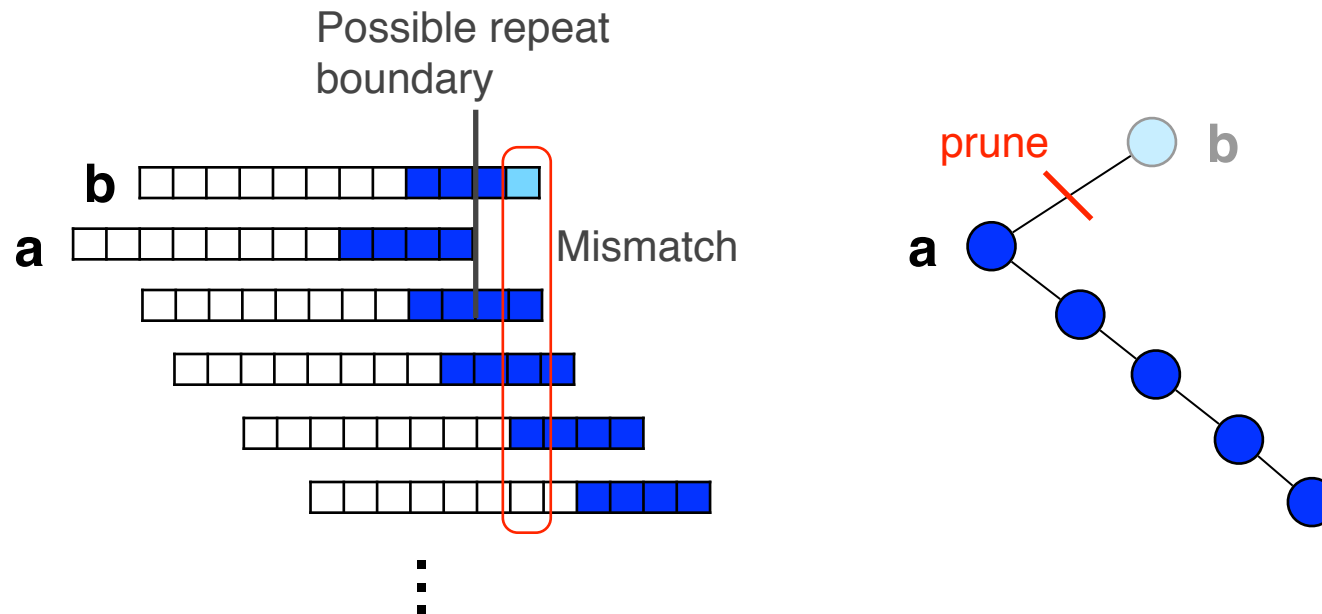
turn_there_is_a_season



Unresolvable repeat

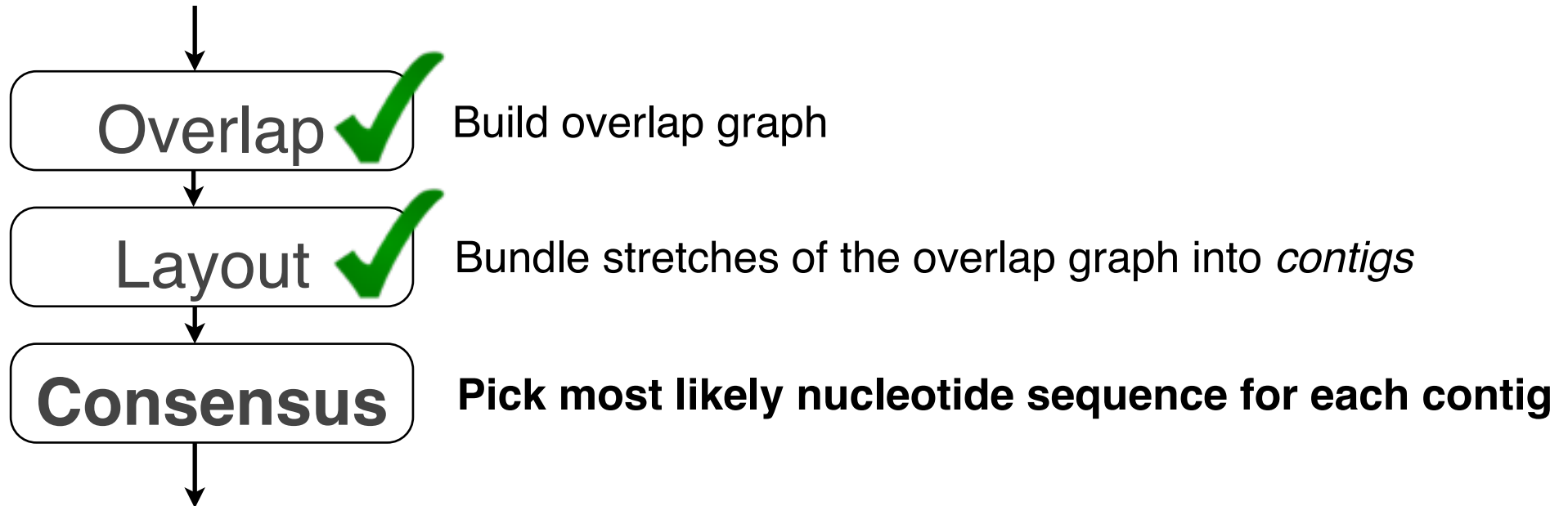
Layout

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**.

Overlap Layout Consensus



Consensus

TAGATTACACAGATTACTGA TTGATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAACTA
TAG TTACACAGATTATTGACTTCATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAA CTA

Take reads that make
up a contig and line
them up

↓ ↓ ↓ ↓ ↓
TAGATTACACAGATTACTGACTTGATGGCGTAA CTA

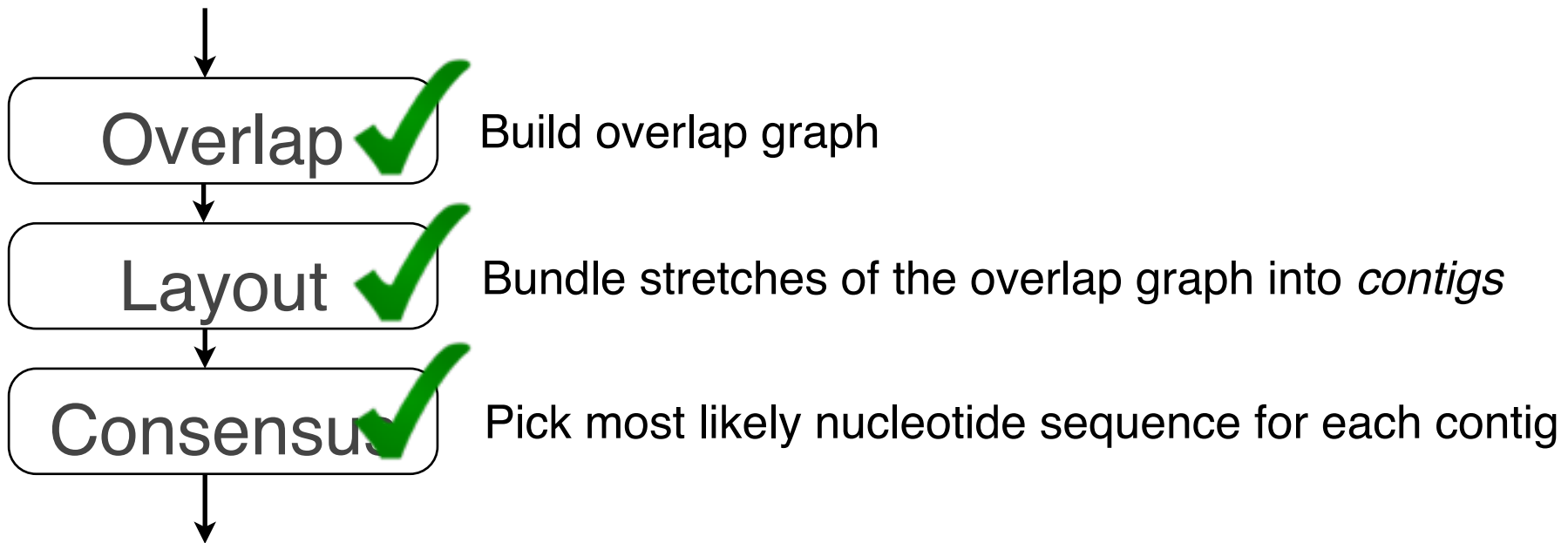
Take *consensus*, i.e.
majority vote

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



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, and in practice # edges grows superlinearly with # reads

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

Summary

- Discussed assembly as SCS problem
- Discussed OLC assembly
- Next week: de Bruijn assembly and memory efficiency

