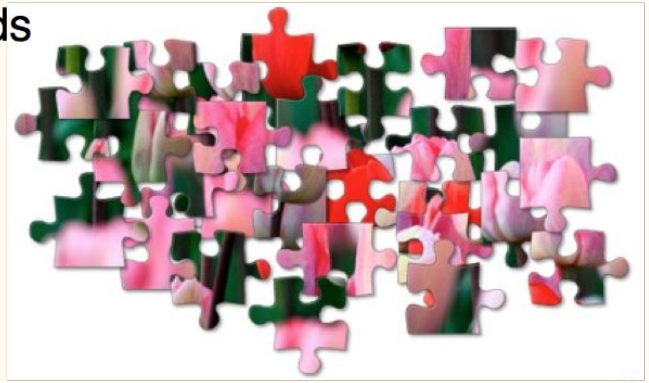
Assembly

Shortest common superstring

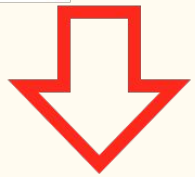
Assembly

Reads



+

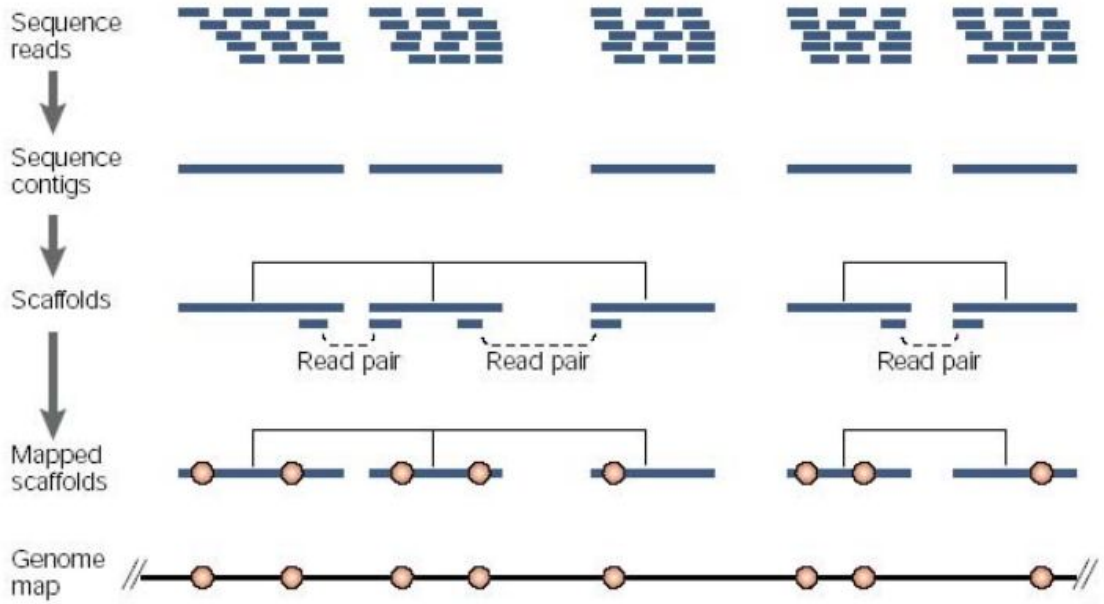
Reference genome



Input DNA

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

de novo whole-genome shotgun assembly



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)

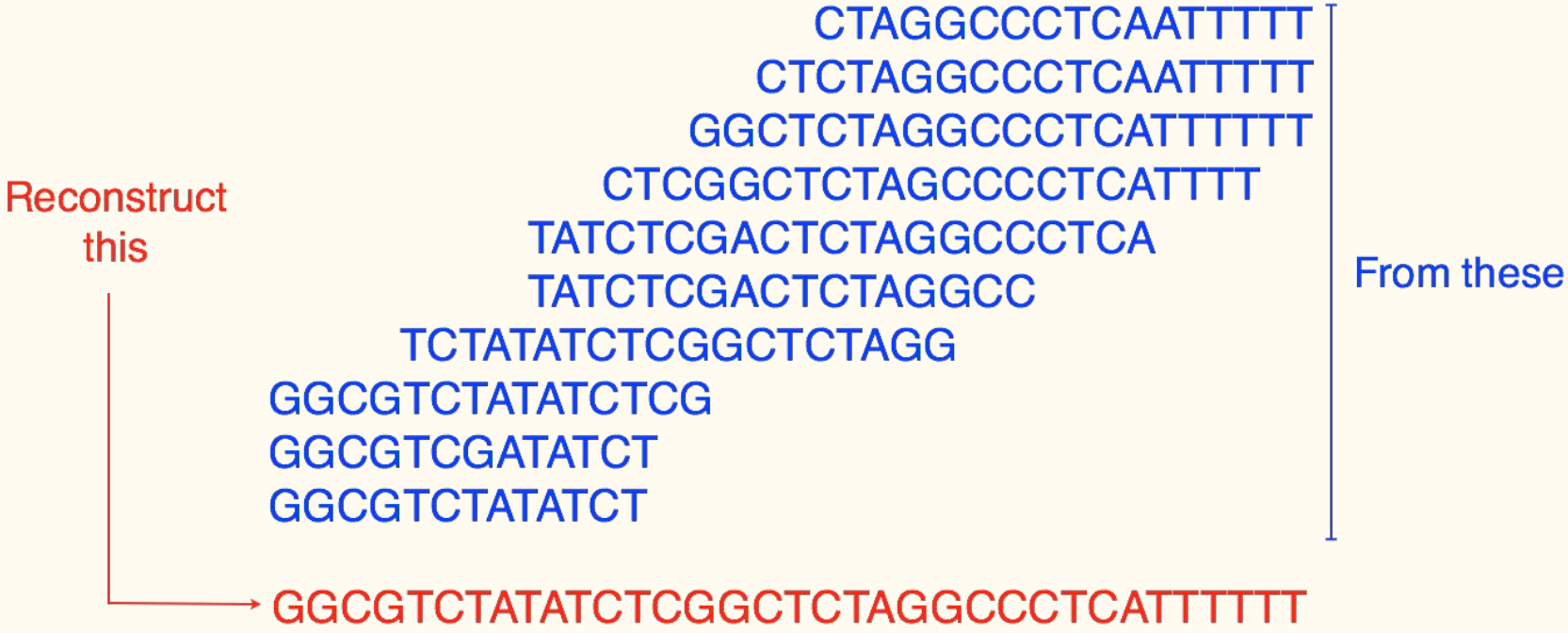
Input: GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Copy:
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

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

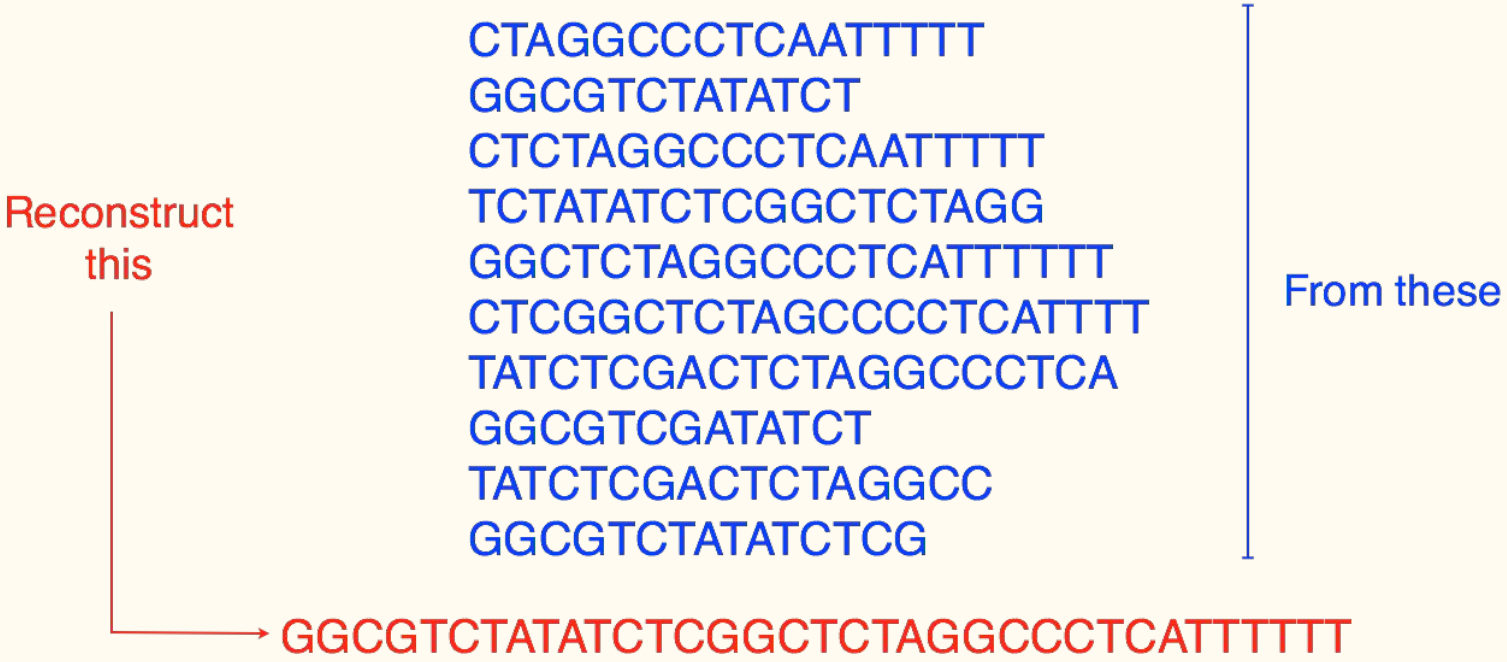
Assembly

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



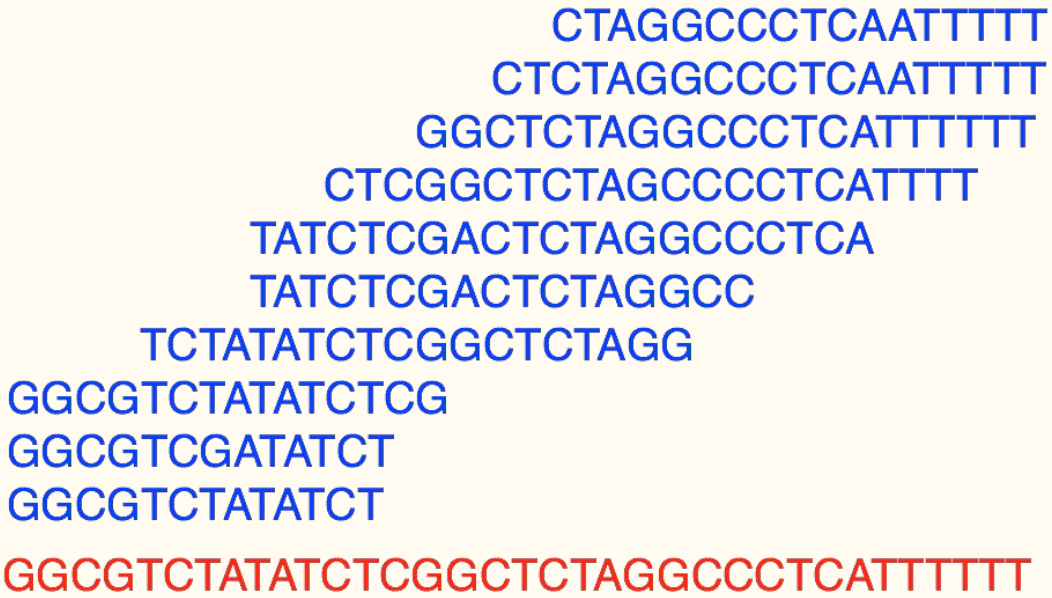
Assembly

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



Assembly

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



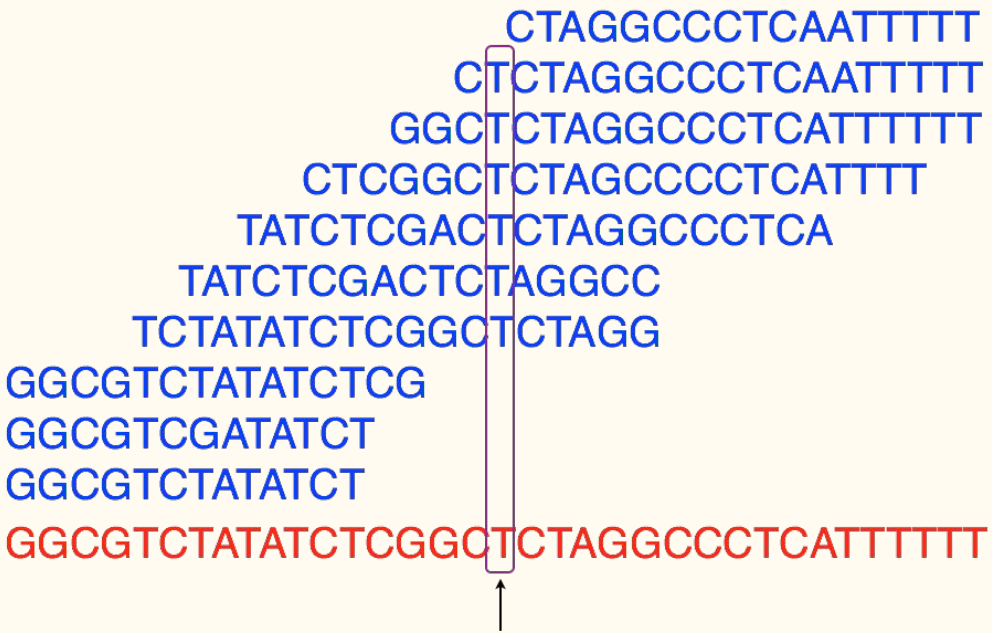
177 nucleotides

35 nucleotides

Average coverage = 177 / 35 ≈ 5x

Assembly

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



Coverage at this position = 6

Assembly

- Best case scenario: check every pair of (long) reads for overlaps
- Computationally expensive: **n** reads, **~n**2** operations
- Our task: how to get the **best assemblies** at the **smallest expense** – in terms of sequencing and computational expenses

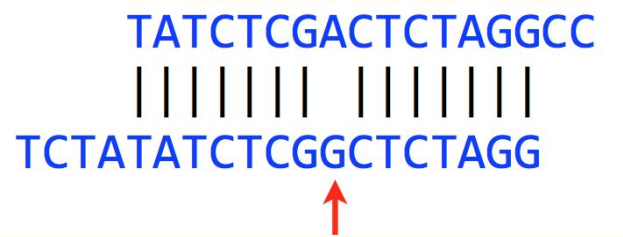
Assembly

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

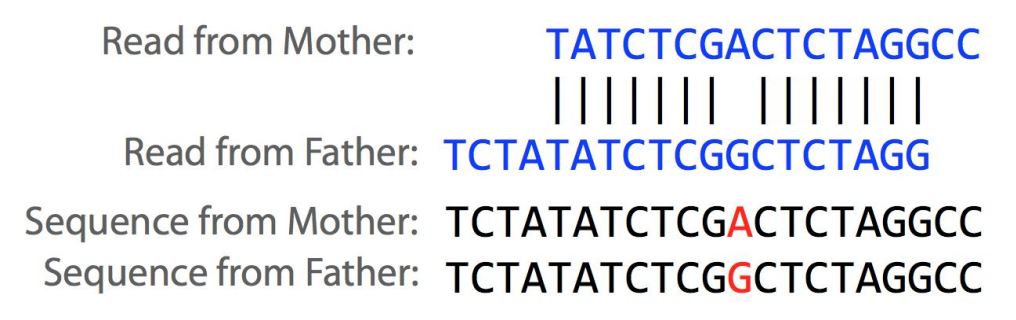
...the more likely they are to have originated from overlapping stretches of the genome:

Assembly

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



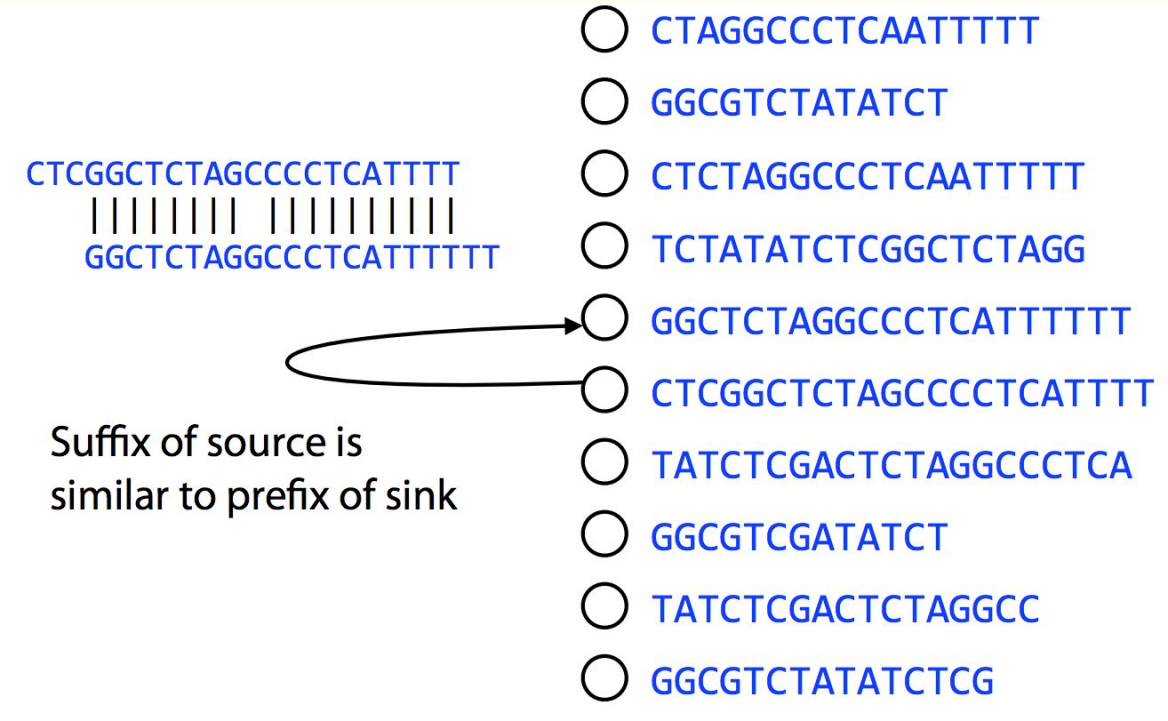
1. Sequencing error
2. Difference between inherited *copies* of a chromosome. E.g. humans are diploid; we have two copies of each chromosome, one from mother, one from father. The copies can differ:



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

Assembly

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



Graphs

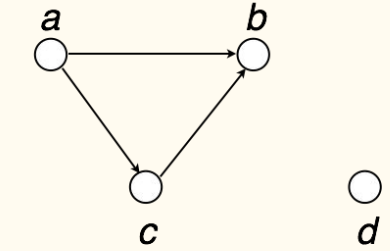
Directed graph G=(V, E) is a pair consisting of *node set* (or vertex set) V and *edge set (*or *arc set)* E ⊆ VxV.

An **edge** $e = (u, v) \in E$ represents a connection from u to v.
Directed edge is an ordered pair of vertices.
We call u and v the *source* and the *target (sink)*, respectively, of e.

Directed graph also called *digraph*.

An *outdegree* of the node is the number of edges leaving it, and *indegree* of the node is the number of edges ending at it.

Graph theory developed by Euler (more info about this later...)



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

Source Sink

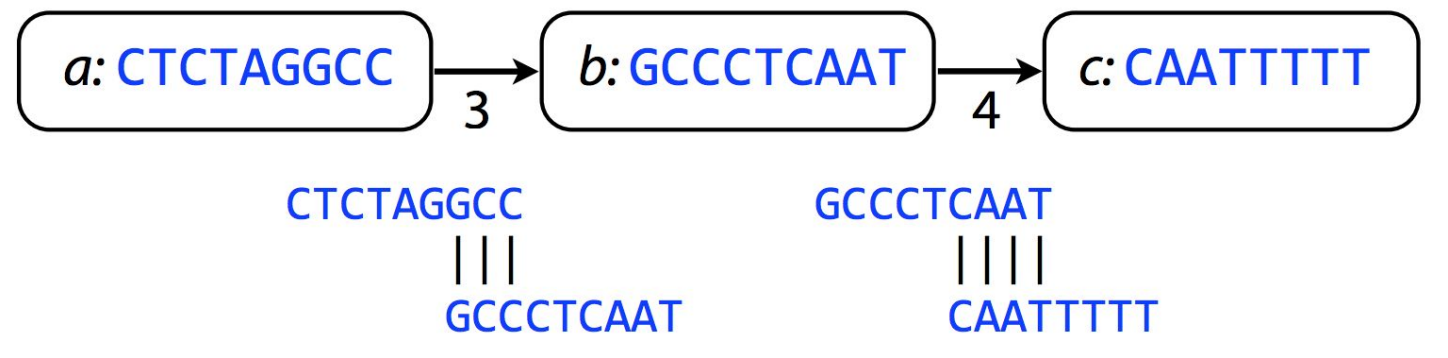
Overlap graph

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

In **overlap graph**:
A **vertex** is a read.
A **directed edge** is an overlap between suffix of source and prefix of sink.

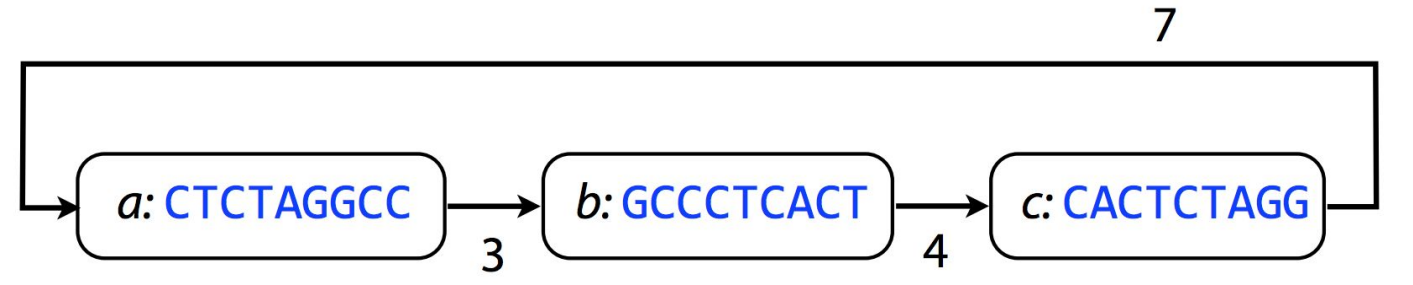
Vertices (reads): { *a:* CTCTAGGCC, *b:* GCCCTCAAT, *c:* CAATTTTT }

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



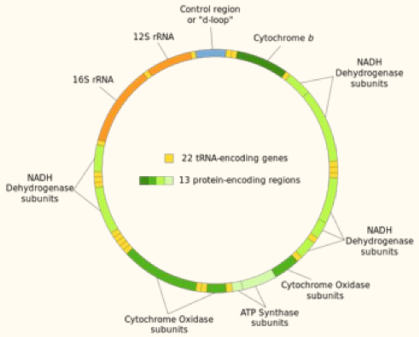
Overlap graph - cycles in graph

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?

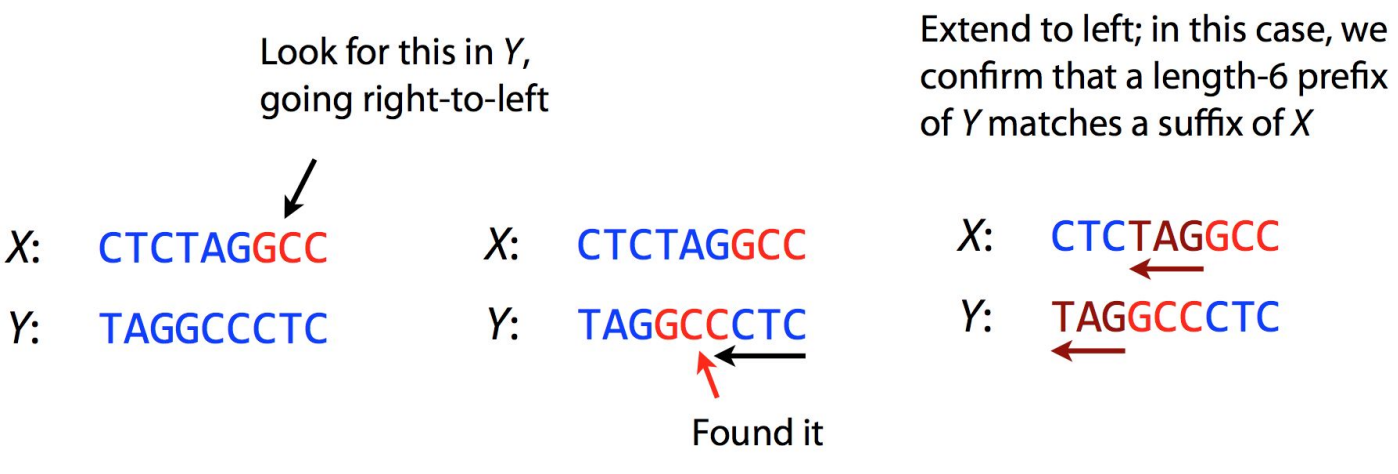
(**Def:**?) Assume for now an "overlap" is when a suffix of ***X*** of **length** $\geqslant$ ***l*** exactly matches a prefix of ***Y***, where l is given.

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.

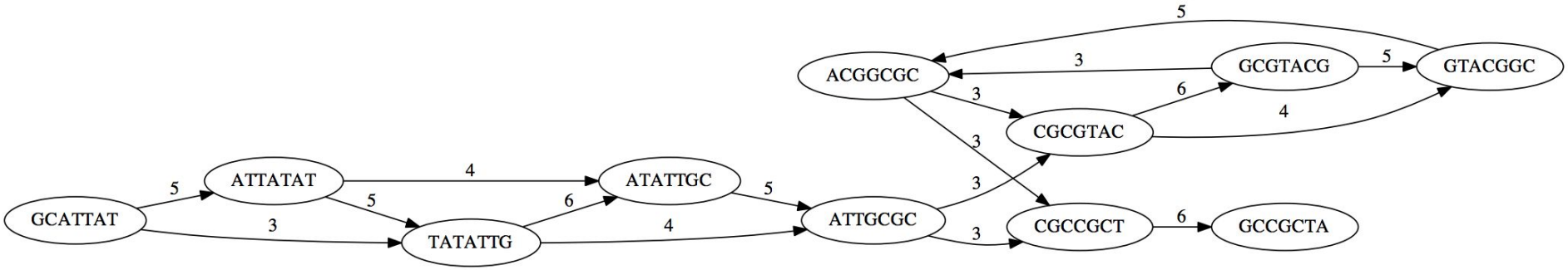
For l=3



Finding overlaps

Edge label is overlap length

Example overlap graph with $l = 3$



Original string: GCATTATATATTGCGCGTACGGCGCCGCTACA

Shortest common superstring

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

Every sequence (read) is a part of the genome, and we're trying to recreate a genome from reads. We need a *superstring* of the reads - string (genome) that contains all reads we have. Reasonable approximation is that we want not any, but *shortest superstring*.

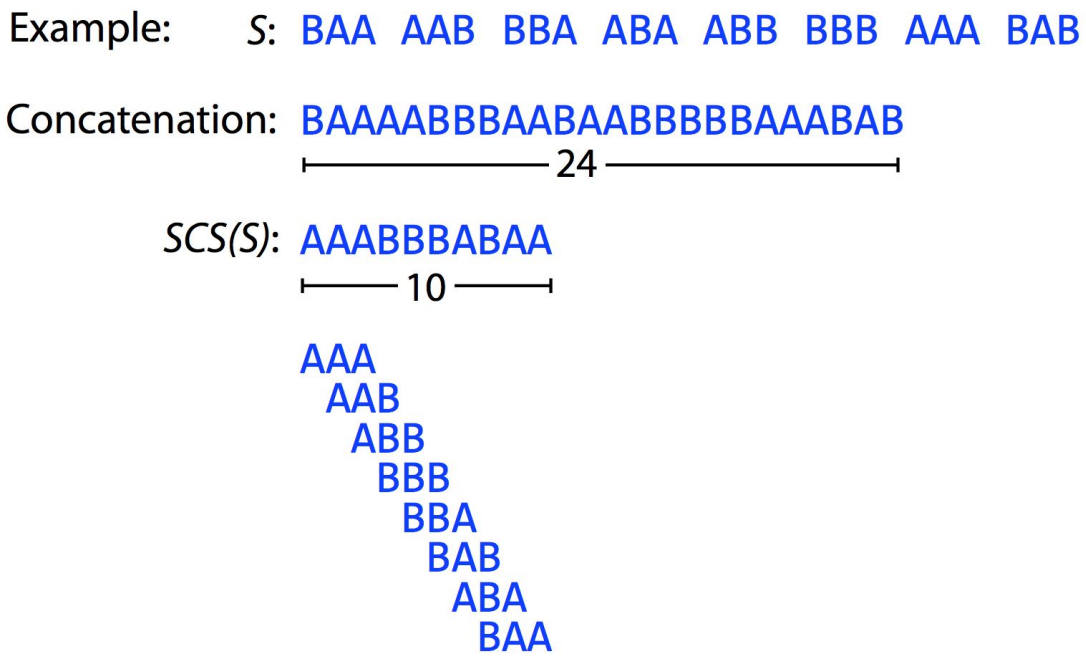
How do we formulate this problem?

First attempt: the s**hortest common superstring (SCS)** problem:
Given a set of strings, find a shortest string that contains all of them.

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

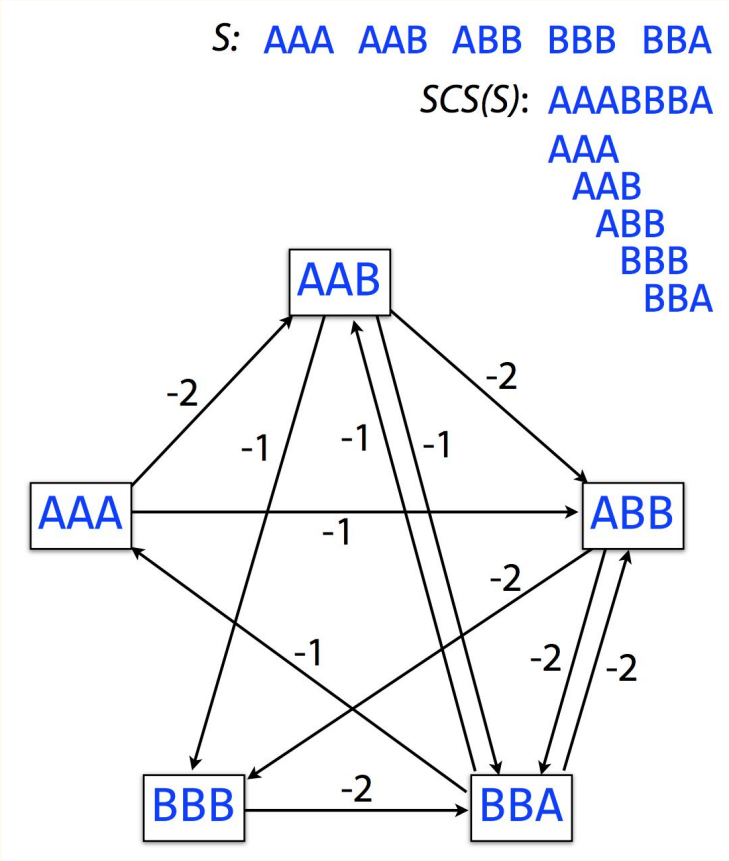


Shortest common superstring

How 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!

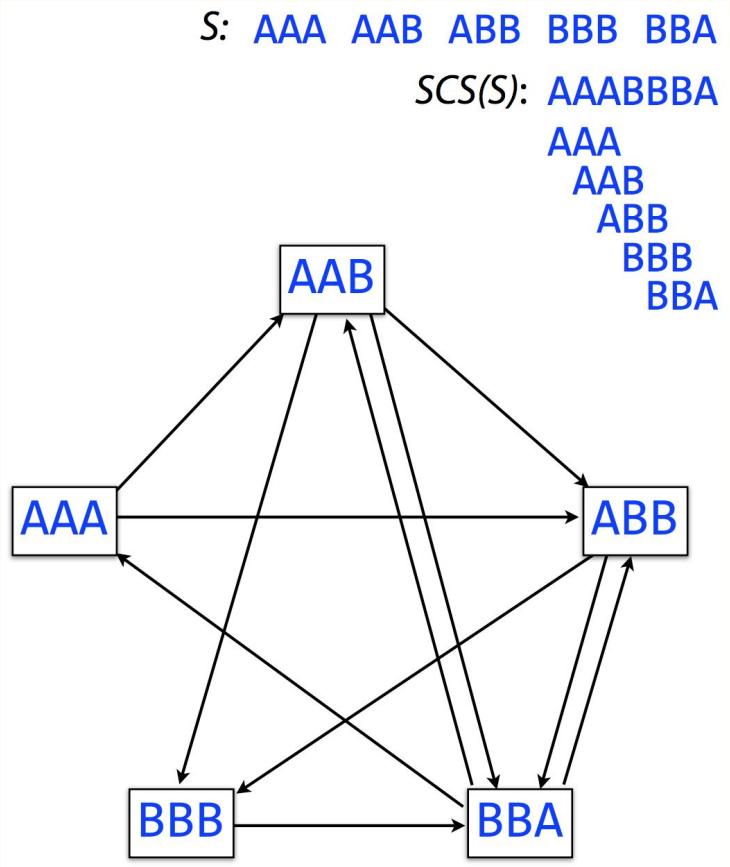


Shortest common superstring

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



NP problems

NP: Decision problems (yes or no answers), if “yes” can be checked in polynomial time

NP complete: Solve one, solved them all!

- Traveling salesman problem
- Knapsack problem
- Integer linear programming
- SAT problem ...

(Karp’s 21 NP-complete problems)

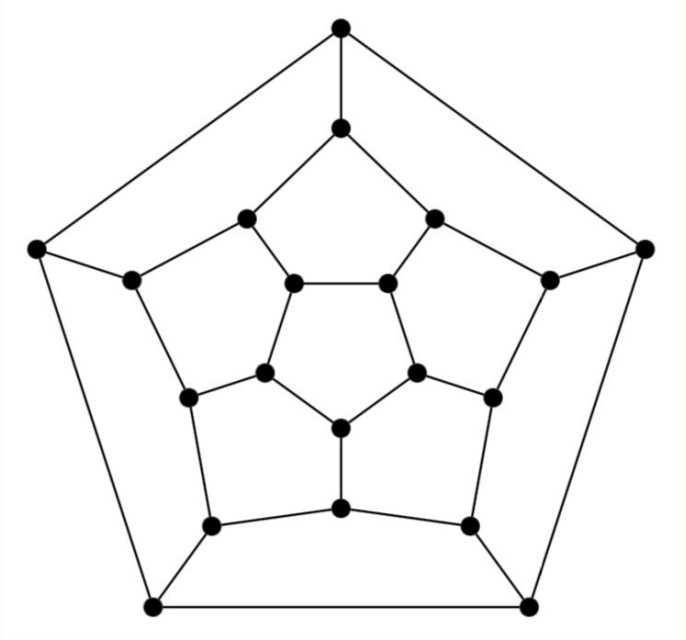
NP hard: at least hard as NP

Hamiltonian Path

Hamiltonian Cycle Problem:
Find a cycle in a graph that visits every vertex exactly once.

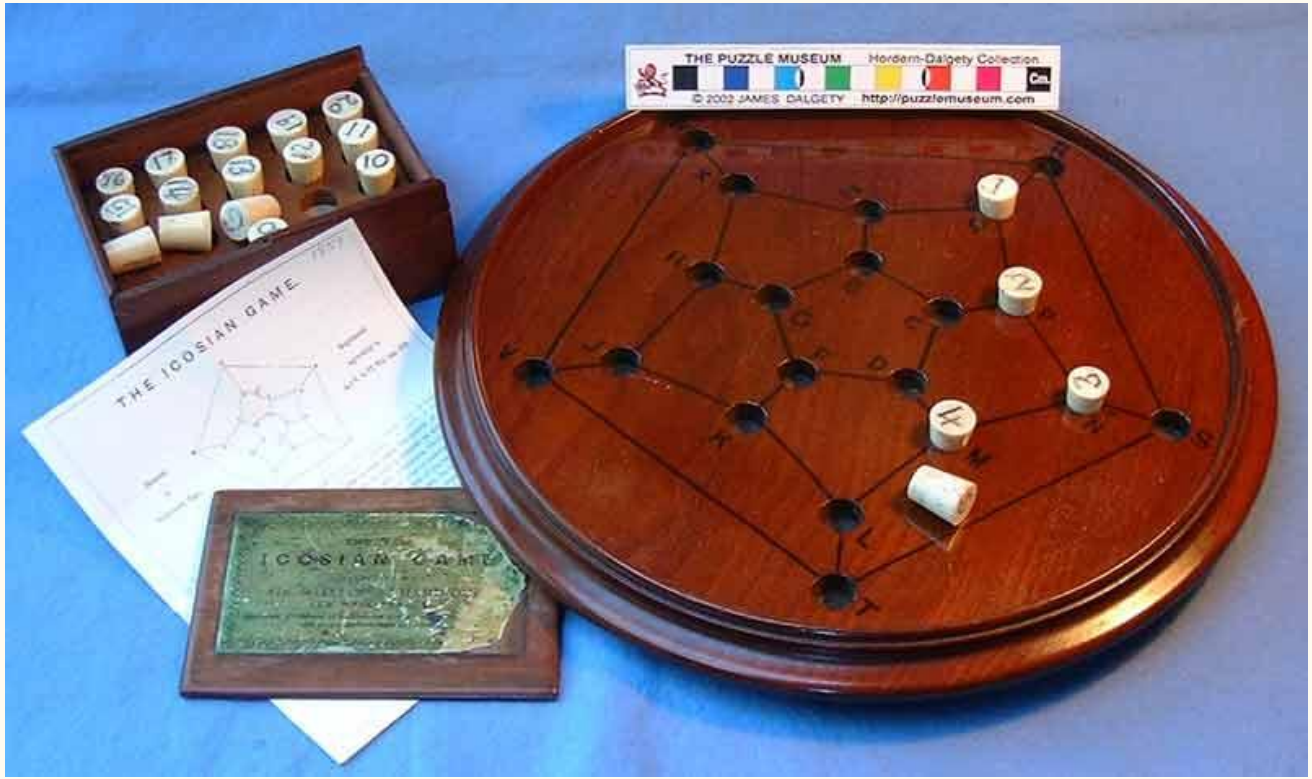
Input: A graph G.
Output:A cycle in G that visits every vertex exactly once (if exists).

- Sir Richard Hamilton created a game:
  - Graph whose vertices are labeled with the name of 25 cities
  - Visit every city only once
  - Sold via London game dealer for 25 pounds
  - Failed miserably, but defined “Hamiltonian Path” problem which occupied mathematicians and engineers in the times to come
  - NP-complete



Sir Richard Hamilton’s Icosian Game

Hamiltonian Path



Sir Richard Hamilton's Icosian Game

NP, NP-hard, NP-complete

Shortest common superstring & friends: For refreshers on Traveling Salesman, Hamiltonian Path, NP-hardness and NP-completeness, see Chapters 34 and 35 of “Introduction to Algorithms” by Cormen, Leiserson, Rivest and Stein;

or Chapters 8 and 9 of “Algorithms” by Dasgupta, Papadimitriou and Vazirani (free online: http://www.cs.berkeley.edu/~vazirani/algorithms)

Tl;dr - stack overflow - What are the differences between NP, NP-Complete and NP-Hard?

Greedy approach

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 ($l = 1$):

In red are strings that get merged before the next round

Greedy answer:
BABBBAABAAA

Actual SCS:
AAABBBABAA

Rounds of merging, one merge per line.
Number in first column = length of overlap merged before that round.

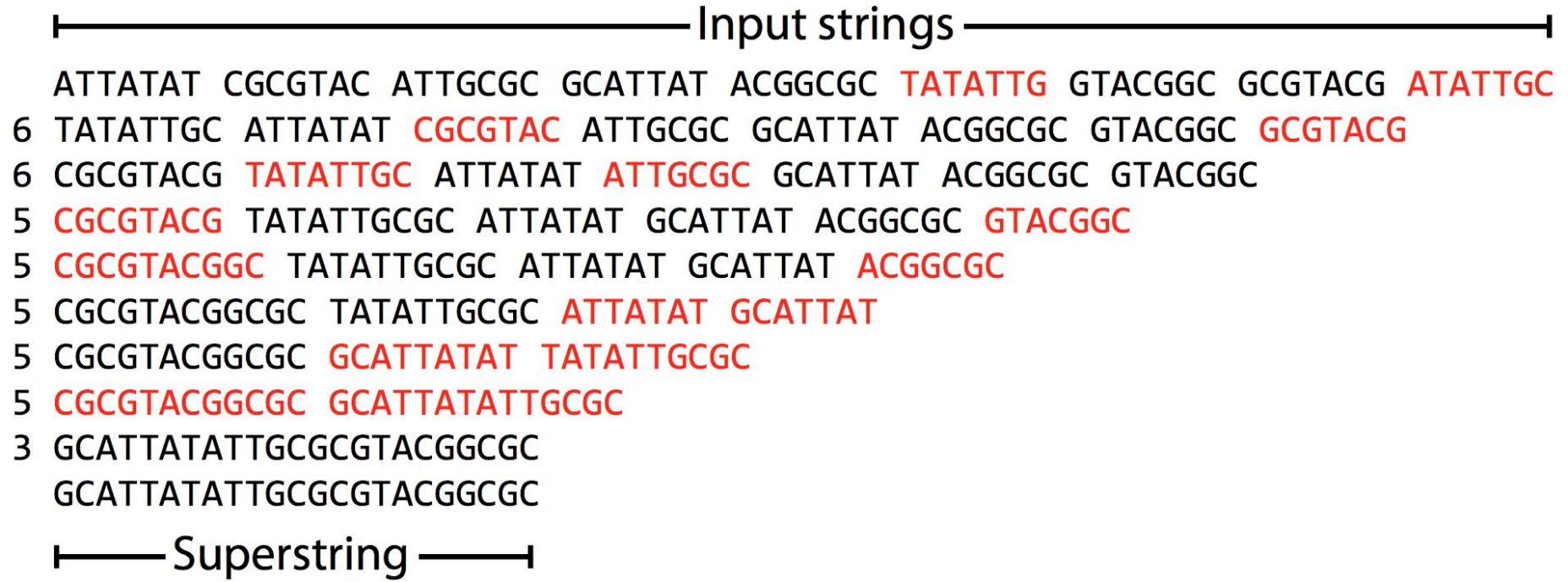
Shortest common superstring: greedy

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. $l = 3$.

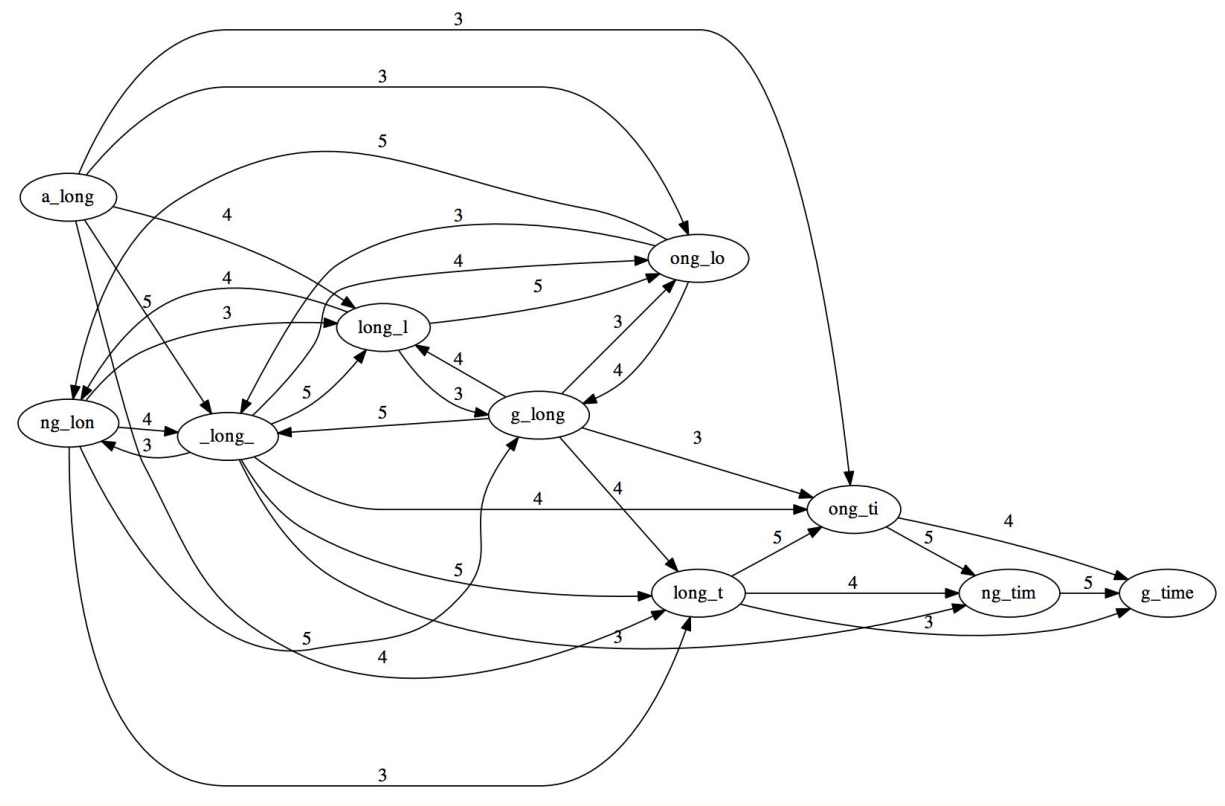
```
  ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long g_time ng_tim
5 ng_time ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long
5 ng_time g_long_ ng_lon a_long long_l ong_ti ong_lo long_t
5 ng_time long_ti g_long_ ng_lon a_long long_l ong_lo
5 ng_time ong_lon long_ti g_long_ a_long long_l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

What happened?

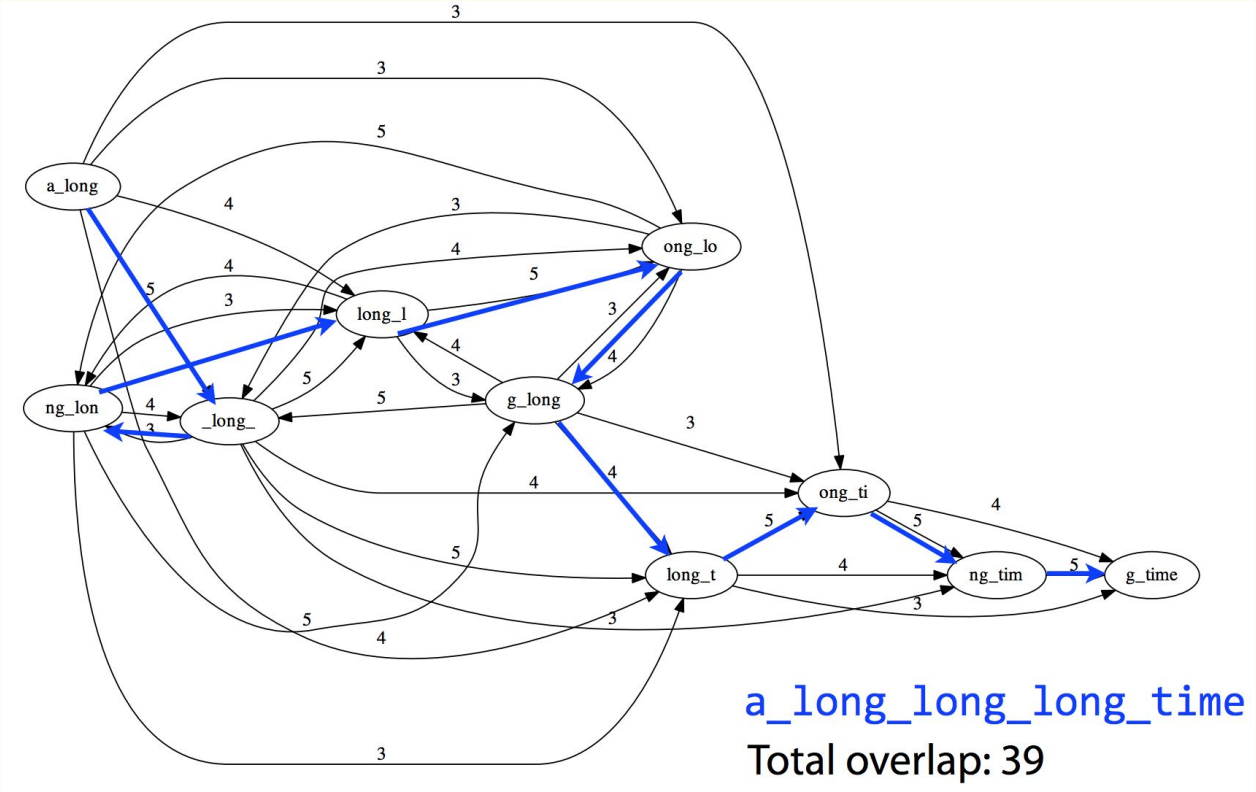
Shortest common superstring: greedy

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



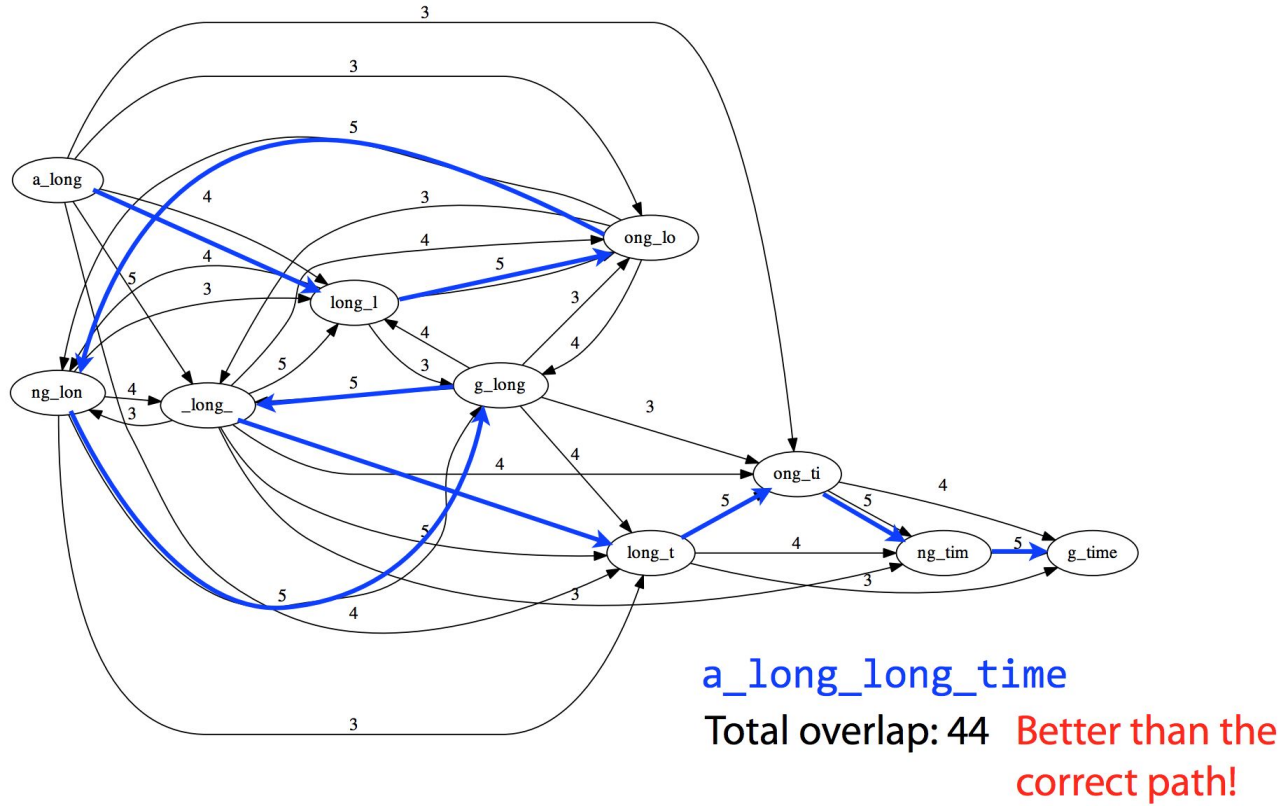
Shortest common superstring

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



Shortest common superstring: greedy

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



Shortest common superstring: greedy

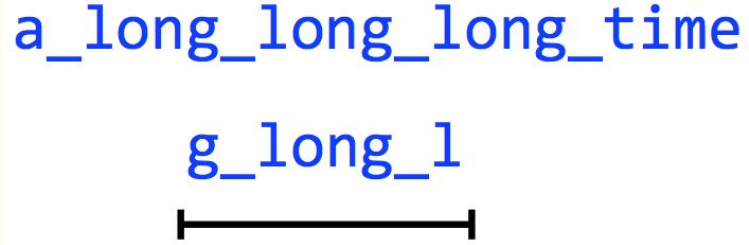
Same example, but increased the substring length from 6 to 8

```
  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

Repeats

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

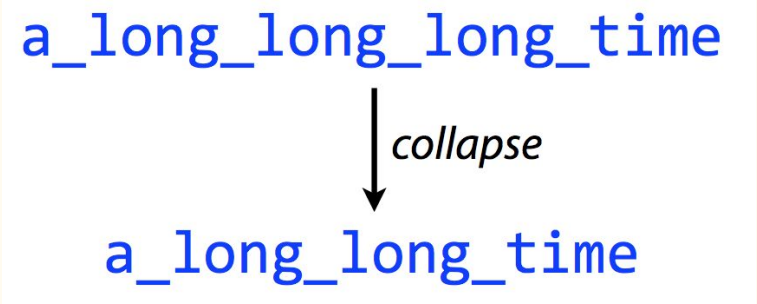


One length-8 substring spans all three longs

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

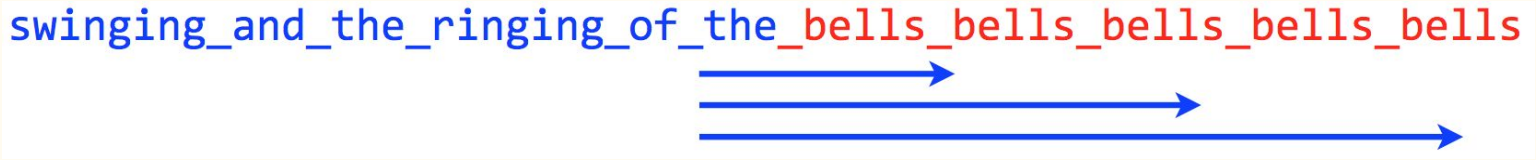
Extract every substring of length k, then run Greedy-SCS. Do this for various l (min overlap length) and k.

l, k	output
3, 5	the_worst_of_times_it_was_the_best_o
3, 7	s_the_worst_of_times_it_was_the_best_of_t
3, 10	_was_the_best_of_times_it_was_the_worst_of_tim
3, 13	it_was_the_best_of_times_it_was_the_worst_of_times

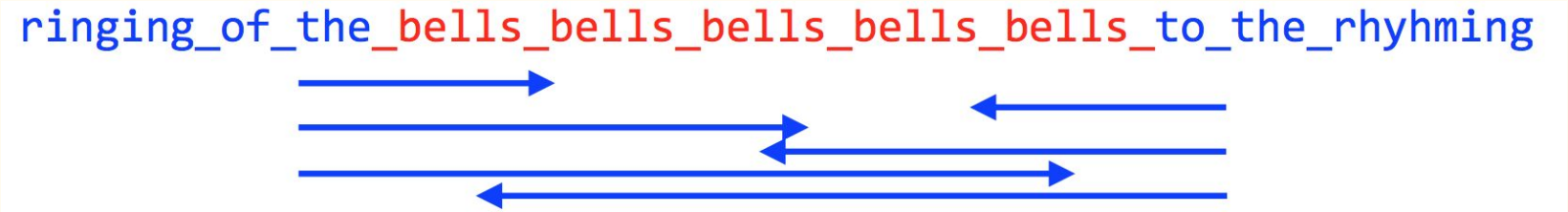
Repeats

Basic principle: repeats foil assembly

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



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



Repeats

Basic principle: repeats foil assembly

Yet another example using Greedy-SCS:

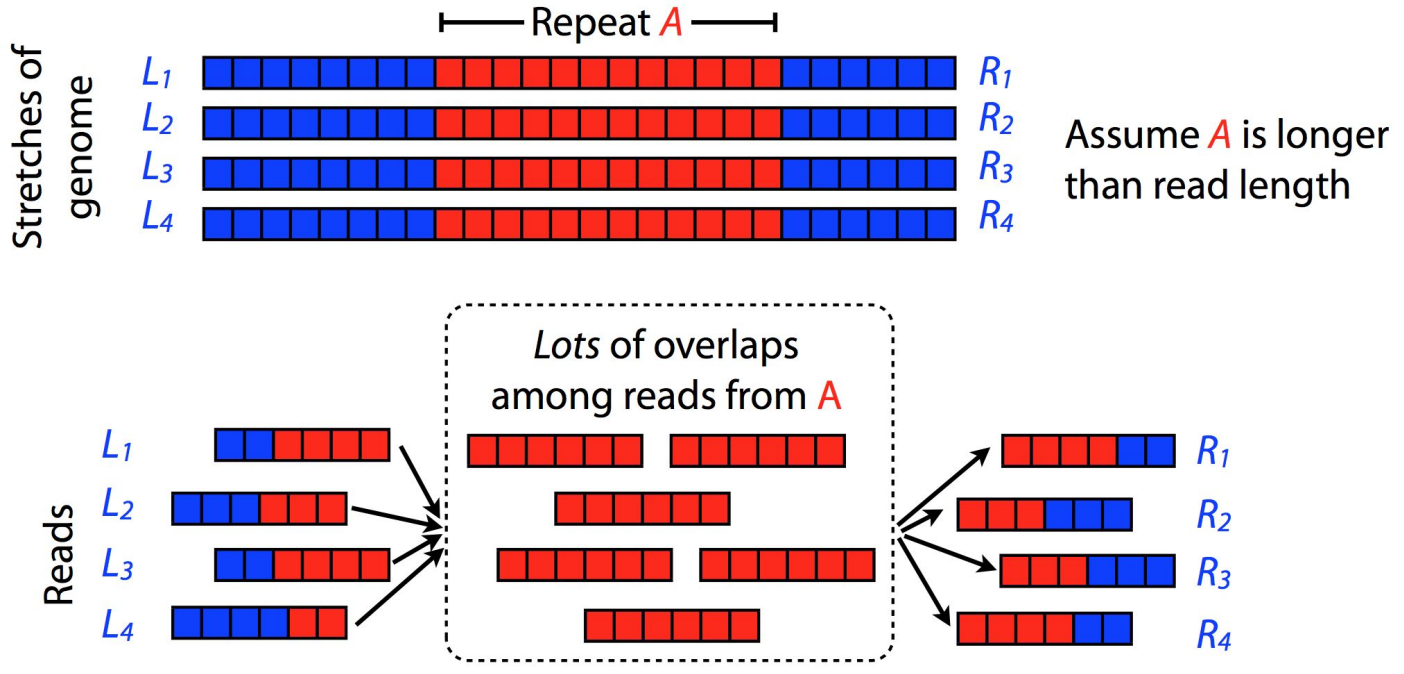
Input: `swinging_and_the_ringing_of_the_bells_bells_bells_bells_bells`

l, k	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_bells_b`
3, 25	`swinging_and_the_ringing_of_the_bells_bells_bells_bells_bells`

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

SCS is too computationally complex as a way of formulating the assembly problem

- There is no practical way to find SCS
- Greedy approach: we might get too long answers (at least ~2.5 times)
- Repat handling: SCS might collapse repeat sequences
    - In this case answers might get too short

Need solutions that are: a) tractable b) handle repeats as best as possible

Note: Repeats are problem in any assembly 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:

- No solutions (that I know of) are tractable

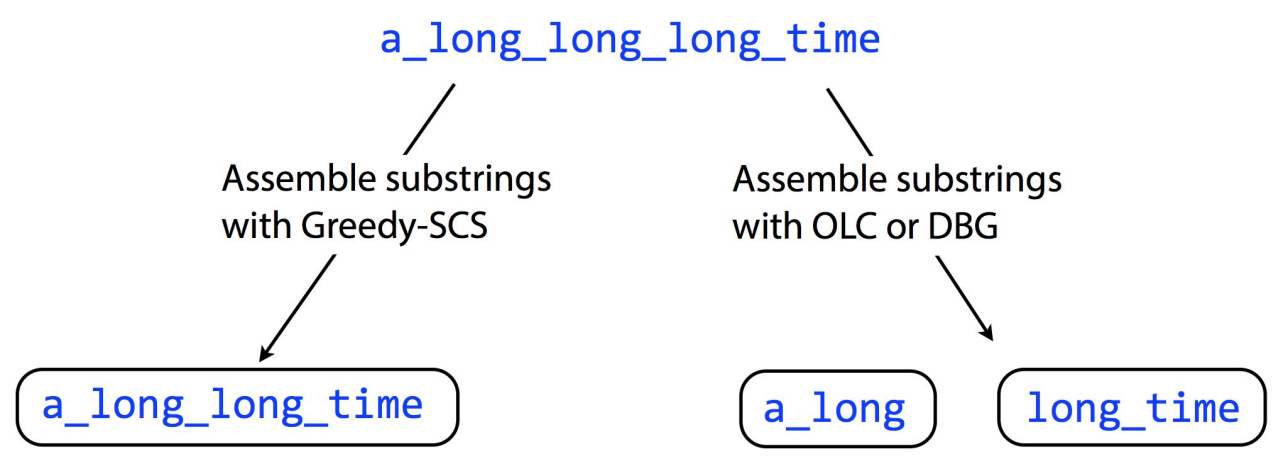
Give up on unresolvable repeats and use a tractable algorithm to assemble the resolvable portions. This is what real tools do.

Overlap-layout-consensus

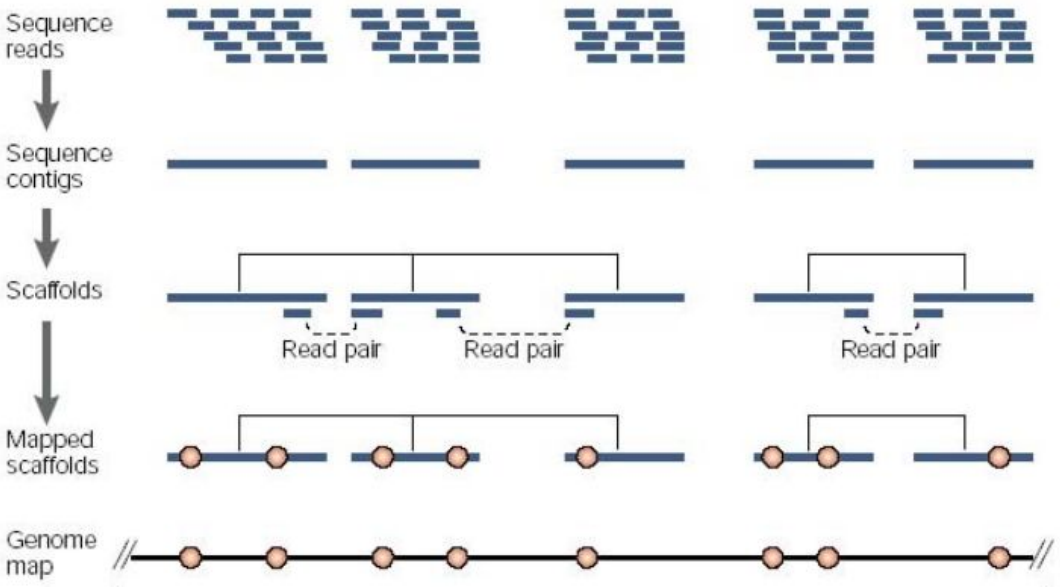
Real-world assembly methods

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).



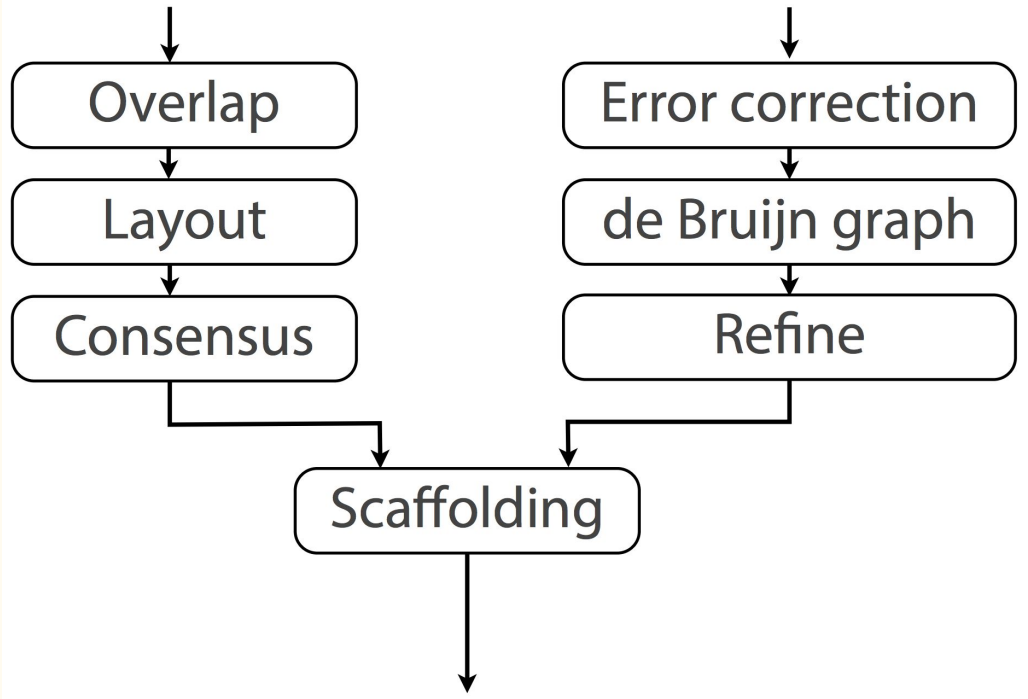
de novo whole-genome shotgun assembly



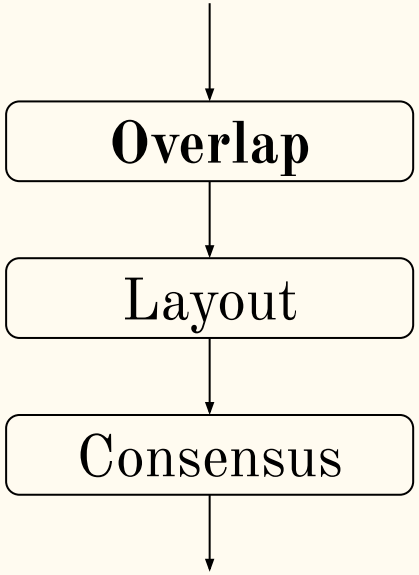
Assembly alternatives

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

Alternative 2: de Bruijn graph (DBG) assembly



Overlap Layout Consensus



Build overlap graph

Bundle stretches of the overlap graph into *contigs*

Pick most likely nucleotide sequence for each contig

Finding overlaps

We have several approaches (all covered in previous lectures)

1. Naive exact matching
2. Suffix trees
3. FM index

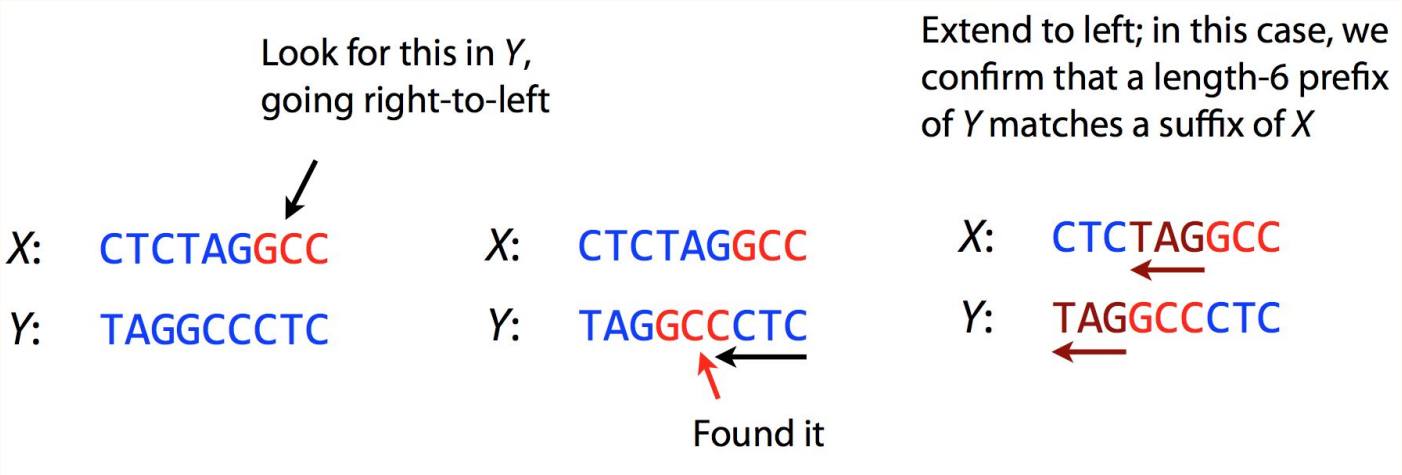
} Exact matching

4. Dynamic programming - inexact matching

Finding overlaps

Naive approach:

For l=3



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

Finding overlaps

Can we use suffix trees for overlapping?

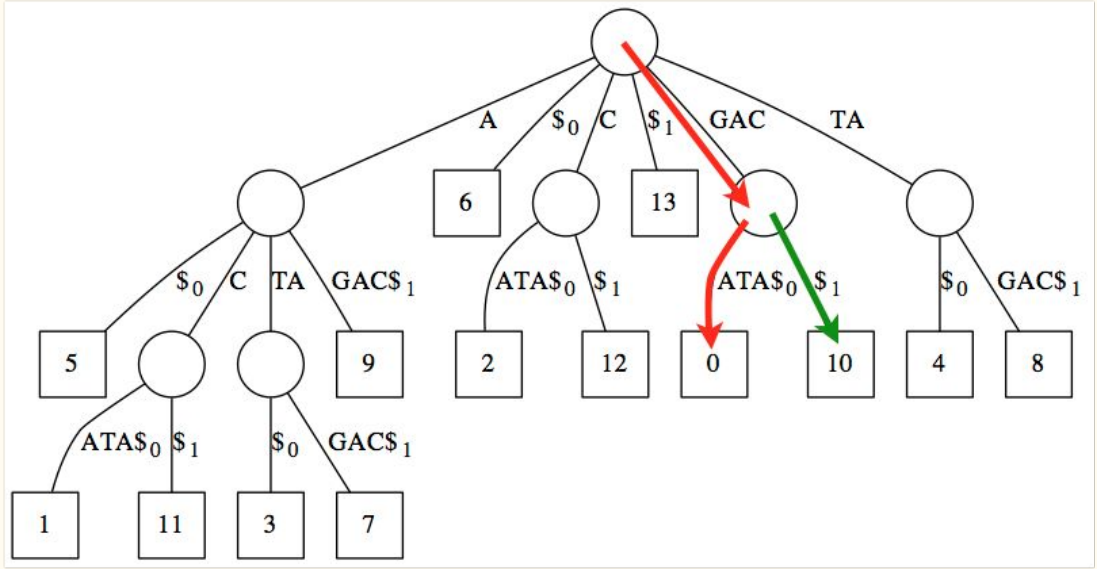
Problem: Given a collection of strings S, for each string x in S find all overlaps involving a prefix of x and a suffix of another string y.

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

Finding overlaps with suffix tree

Generalized suffix tree for { “GACATA”, “ATAGAC” }

$GACATA\$_0ATAGAC\$_1$



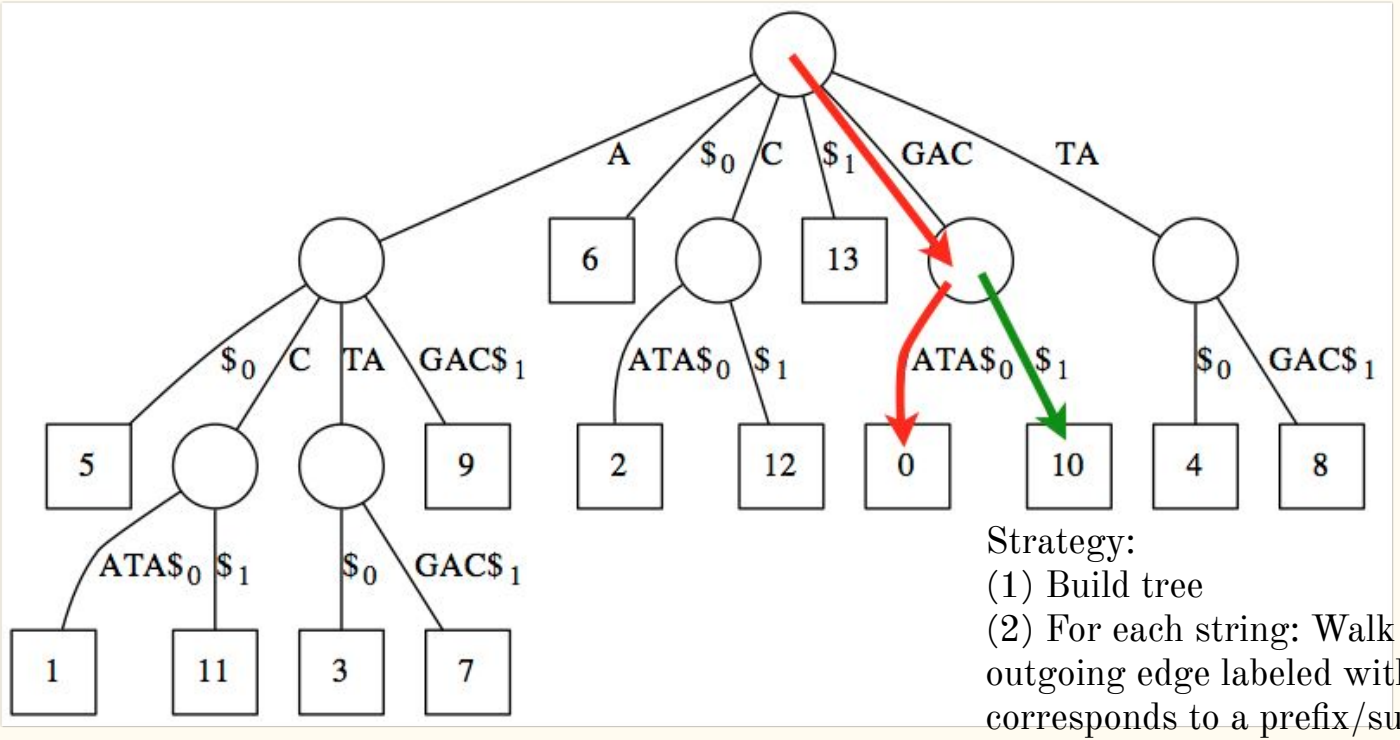
ATAGAC
| | |
GACATA

Say query = GACATA. From root, follow path labeled with query.

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

Finding overlaps with suffix tree

Generalized suffix tree for { “GACATA”, “ATAGAC” }

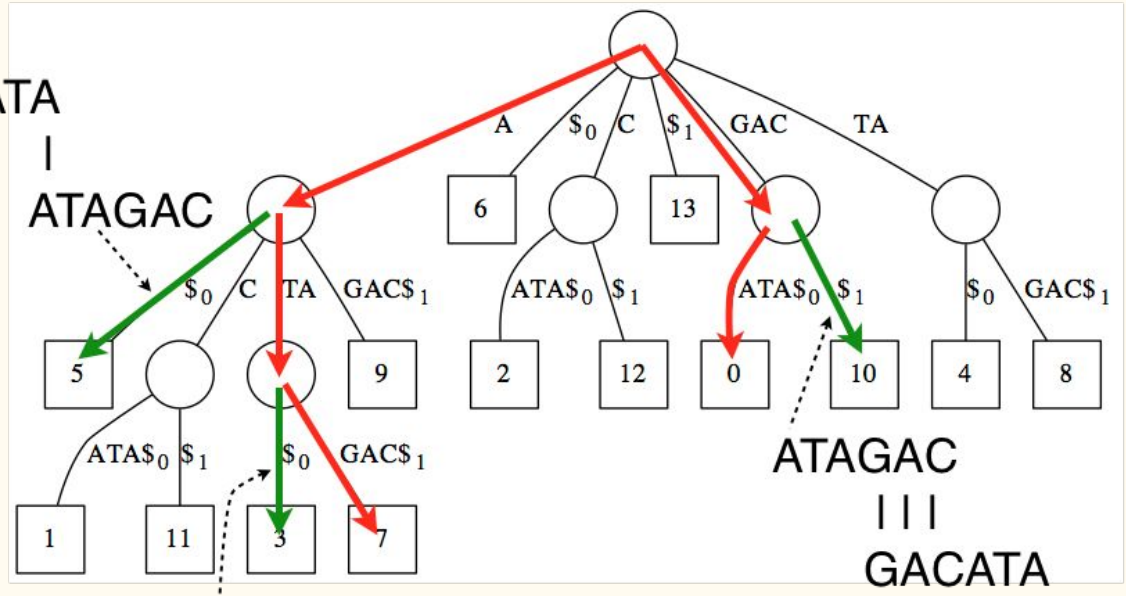


$GACATA\$_0ATAGAC\$_1$

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” }



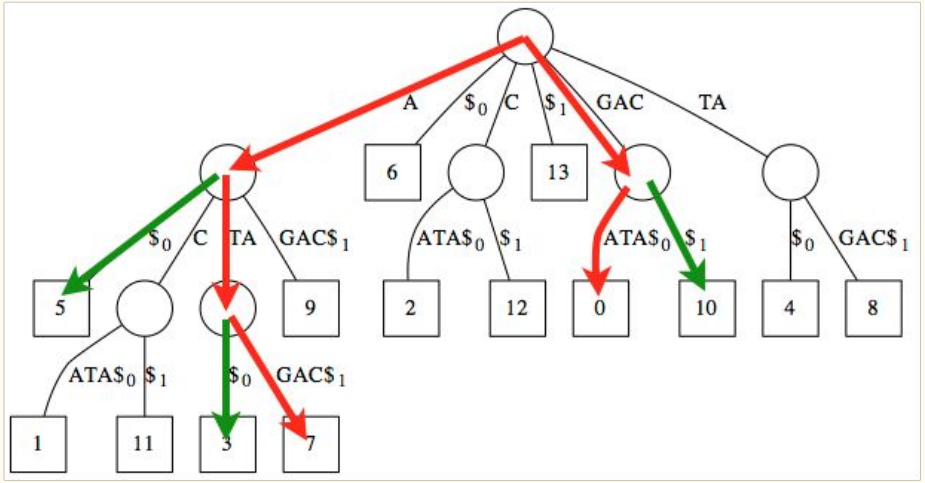
$GACATA\$_0ATAGAC\$_1$

Now let query be second string: ATAGAC

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: $O(a+N)$

d^2 doesn't appear explicitly, but a is $O(d^2)$ in worst case

FM Index for exact matching

We can also use FM index for exact matching. We need to build a joint FM index of all reads, and then search for read overlap by using FM index.

Index: CTCTAGGCC$GCCCTCAAT$CAATTTTT$$

A merged FM index of all reads allows us to match read prefixes and suffixes to discover overlaps.

T Simpson, Jared & Durbin, Richard. (2011). Efficient de novo assembly of large genomes using compressed data structures. Genome research. 22. 549-56. 10.1101/gr.126953.111.

SGA algorithm excludes redundant (transitive) edges.

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:     GGCTCTAGGCCC
```

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

We'll use *global alignment* recurrence and score function.

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

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

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

	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	

Finding overlaps with dynamic programming

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

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

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

	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? (remember *end-space free* variant)

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	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	4	4	12	12	20	24	30	36	44	52	60	68
	G	0	0	8	8	16	16	24	26	30	36	44	52	60
X	C	0	4	4	8	8	16	18	26	30	34	36	44	52
	C	0	4	8	4	8	8	16	22	30	34	34	36	44
	C	0	4	8	8	6	10	10	18	26	34	34	34	36
	T	0	4	8	10	8	8	12	10	18	26	34	36	36
	A	0	2	6	12	14	12	10	12	10	18	26	34	40
	G	0	0	2	10	16	18	16	10	12	10	18	26	34
	G	0	0	0	6	14	20	22	18	10	12	10	18	26

Finding overlaps with dynamic programming

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

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

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

	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	

Y

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

...which might obscure the more relevant match.

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

X \ 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.

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

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

	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	

Y

Solve by initializing certain additional cells to ∞

Cells whose values changed highlighted in red

Now the relevant match is the best candidate

X \ 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	∞	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!

Approaches from alignment unit can be adapted to finding overlaps

- We saw adaptations of naive exact matching, suffix-tree assisted exact matching, FM-index and dynamic programming
- Could also have adapted efficient exact matching, approximate string matching, co-traversal, ...

Heuristic speedup

One idea: check if longest common substring is high enough (example: check substrings only up to first quarter of string length), and filter out reads based on this

Finding overlaps

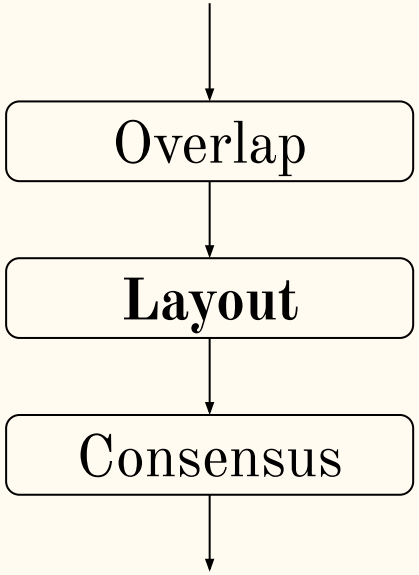
Celera Assembler overlapper is probably the best documented:

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



Build overlap graph

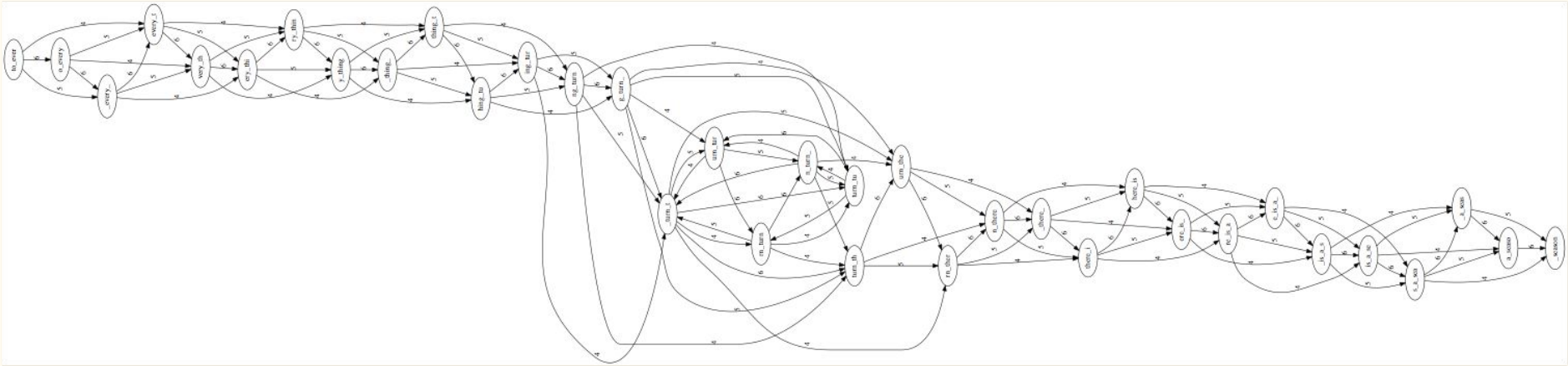
Bundle stretches of the overlap graph into *contigs*

Pick most likely 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
$l = 4, k = 7$

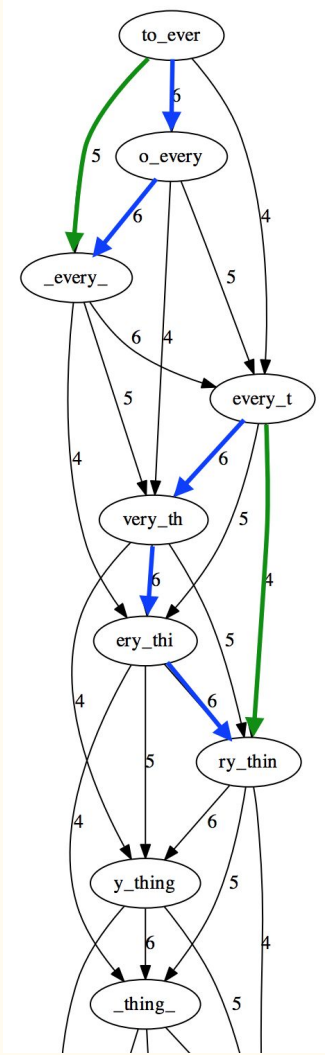


Layout

Anything redundant about this part of the overlap graph?

Some edges can be inferred (transitively) from other edges

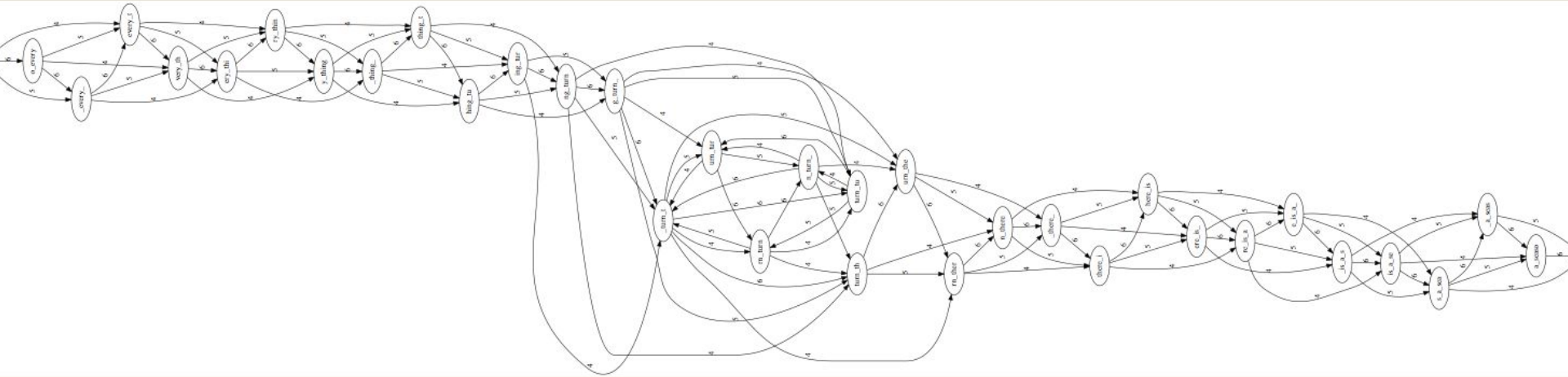
E.g. green edge can be inferred from blue



Layout

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

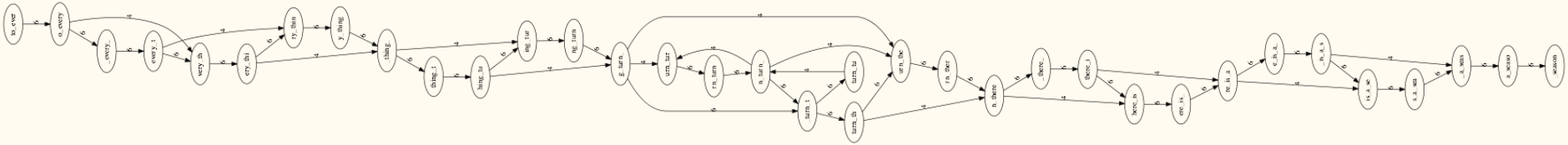
Before:



Layout

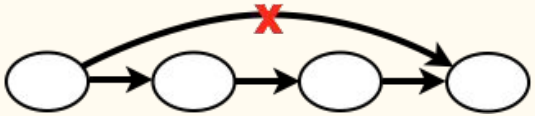
Remove transitively-inferrible edges, starting with edges that skip one node:

After:

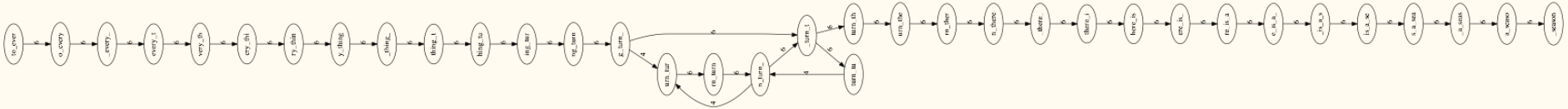


Layout

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

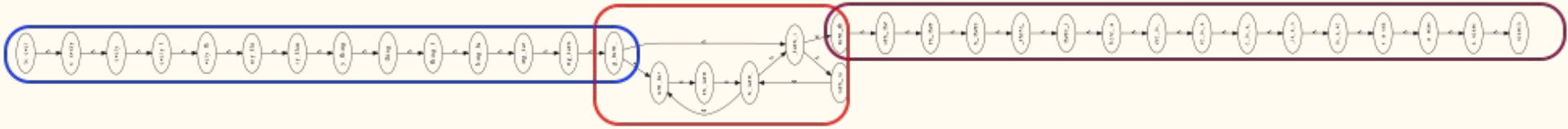


After:



Layout

Emit contigs corresponding to the non-branching stretches (graph traversal).



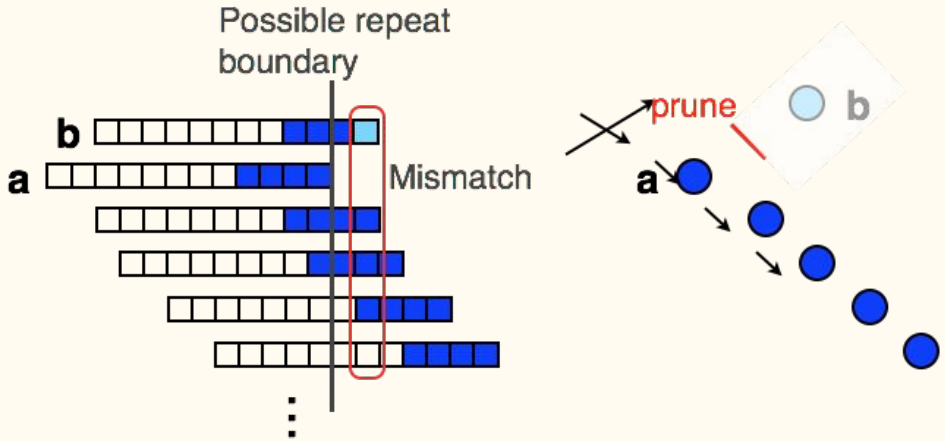
Contig 1
to_every_thing_turn_

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



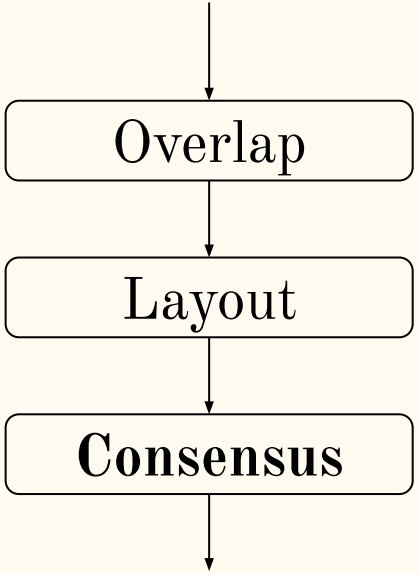
Mismatch could be due to sequencing error or repeat. Since the path through **b** ends abruptly we might conclude it's an error and prune **b**.

Layout

Effect of error on graph.

AACTGCT
A CTGCTA
ACT GCTAA
ACT GCTGA

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

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

Overlap Layout Consensus

OLC drawbacks

> In order to resolve overlap graph we need to traverse it in such way that we visit every node exactly once.
>
> > **Theorem.** [Karp, 1972] (Un)directed Hamiltonian cycle and path are NP-hard.
>
> This is Traveling Salesman Problem - NP-complete! (Hamiltonian cycle can be reduced to TSP)

As mentioned in the previous slide, in typical NGS (next-generation-sequencing) datasets, we have millions or billions of reads. This yields large graphs which are typically hard to traverse!

Assembly metrics

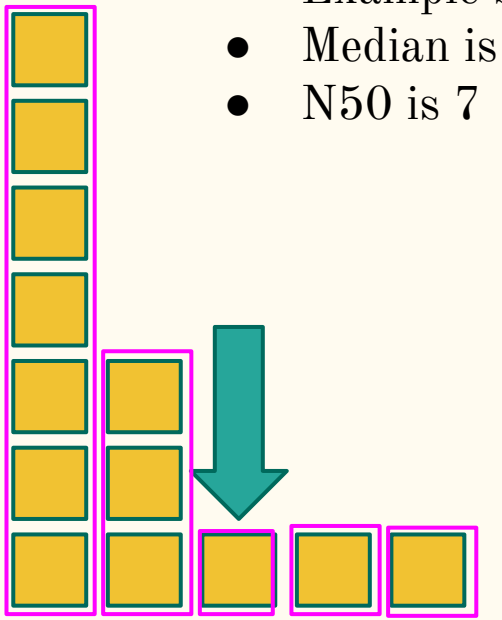
—

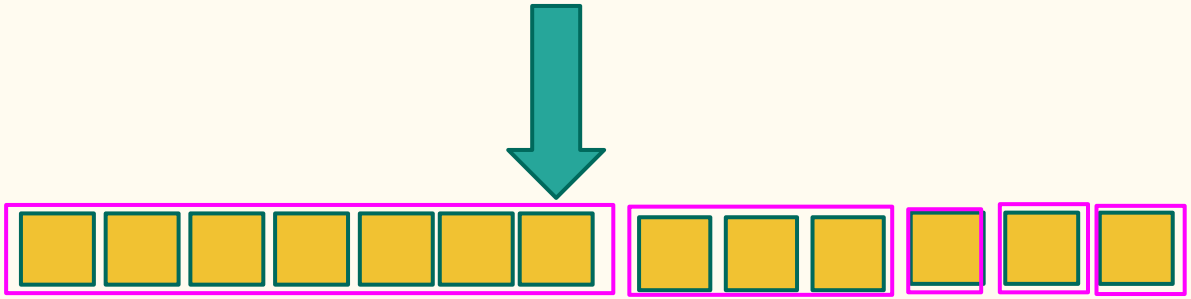
Metrics

- How do we evaluate different assemblies?
- There is no trivial ranking between assemblies
- Reference-free metrics:
  - Number of contigs / scaffolds
  - Total length of the assembly
  - Length of the largest contig / scaffold
  - Percentage of gaps in scaffolds
  - **Nx, NGx of contigs / scaffolds**
  - **Internal consistency**
  - Number of predicted genes
- Reference-using metrics:
  - **Coverage**
  - Assembly errors

Nx / NGx of contigs / scaffolds

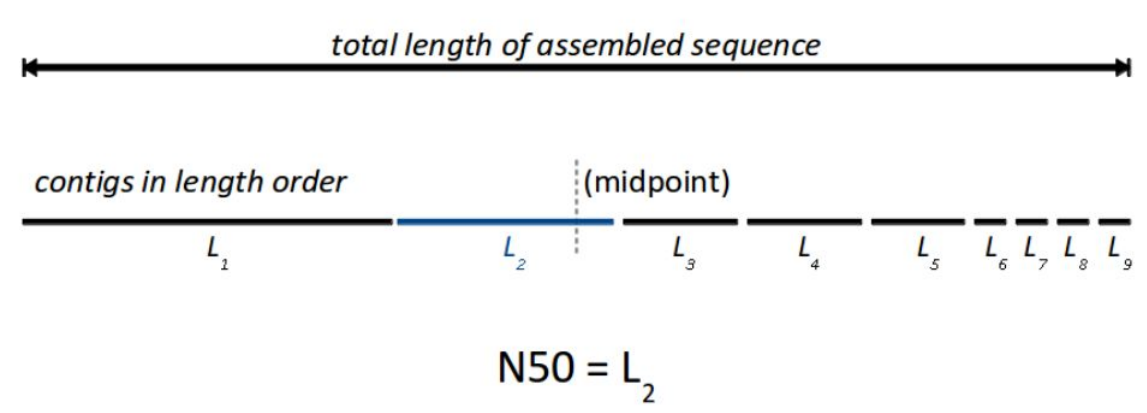
- Nx, where x is percentage value
- **N50** is analogous to **median**
- Example set {7, 3, 1, 1, 1}
- Median is 1
- N50 is 7





Nx / NGx of contigs / scaffolds

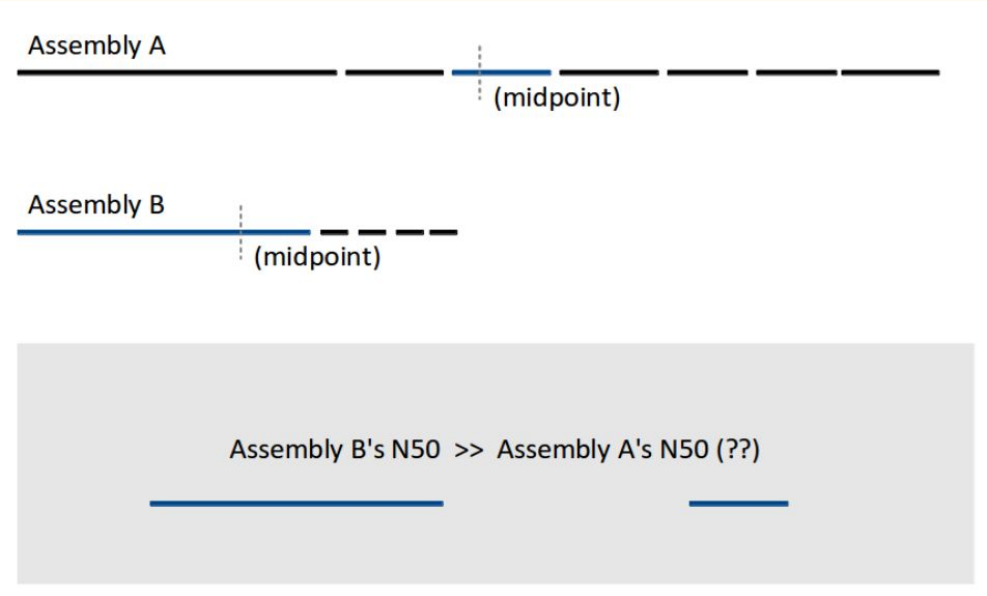
- **Nx** is largest contig length at which longer contigs cover **x%** of the total assembly length



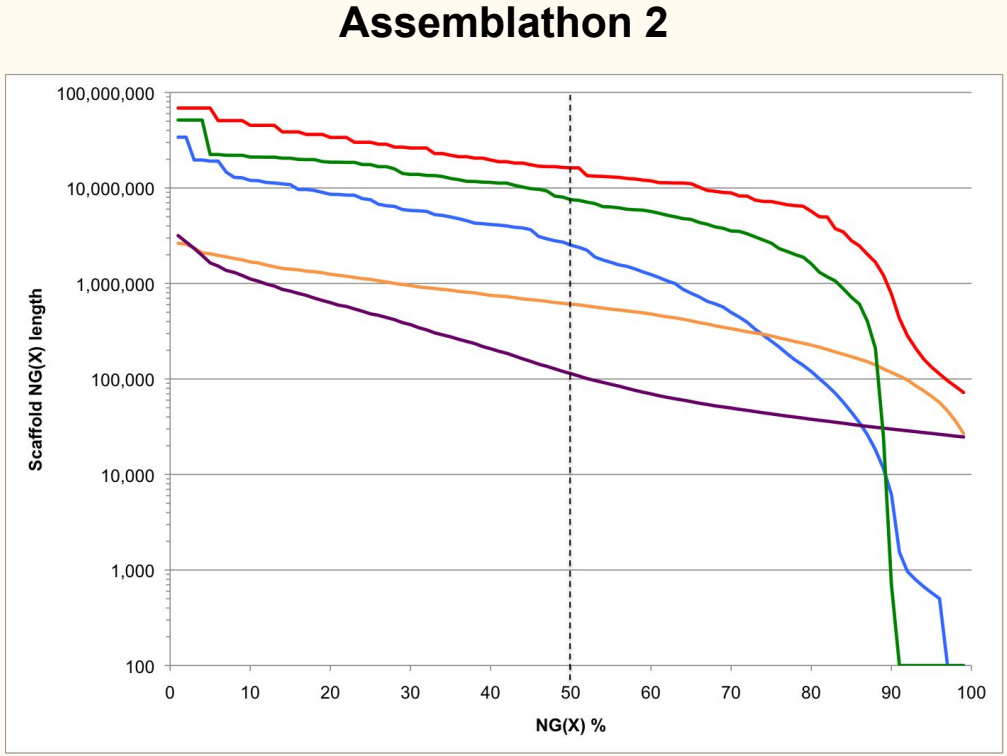
- A practical way to compute Nx:
  - Sort contigs by decreasing lengths
  - Take the first contig (the largest): does it cover x% of the assembly?
  - If yes, this is the Nx value. Else, repeat by trying the next one (the second largest)

Nx / NGx of contigs / scaffolds

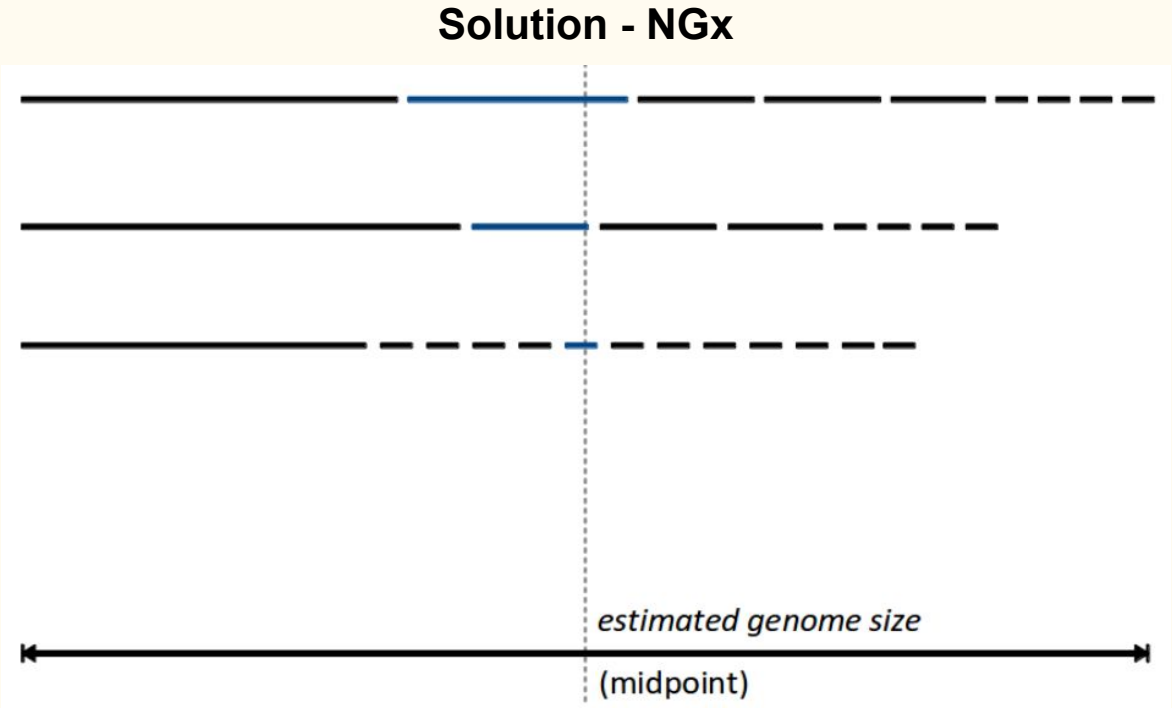
What's the problem with Nx?



Nx / NGx of contigs / scaffolds

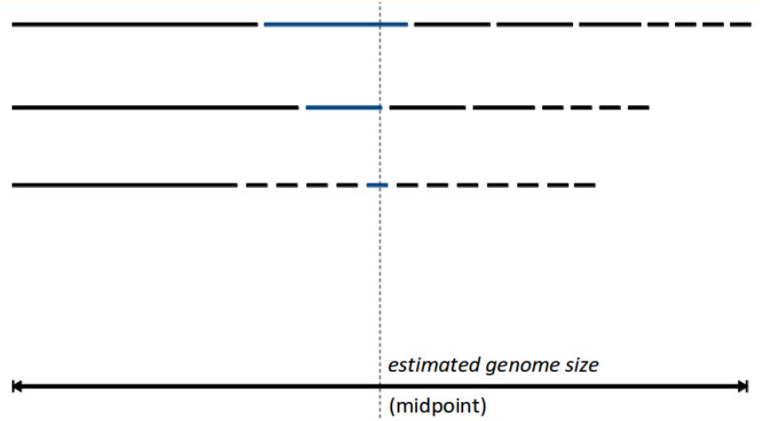


Nx / NGx of contigs / scaffolds



Nx / NGx of contigs / scaffolds

- **NGx** is largest contig length at which longer contigs cover **x%** of the total genome length



- A practical way to compute NGx:
  - Sort contigs by decreasing lengths
  - Take the first contig (the largest): does it cover x% of the genome?
  - If yes, this is the NGx value. Else, repeat by trying the next one (the second largest)

Metrics

- **Internal consistency:** percentage of paired reads correctly aligned back to the assembly (happy pairs)
- **Coverage:** percentage of bases in the reference which are covered by the alignment