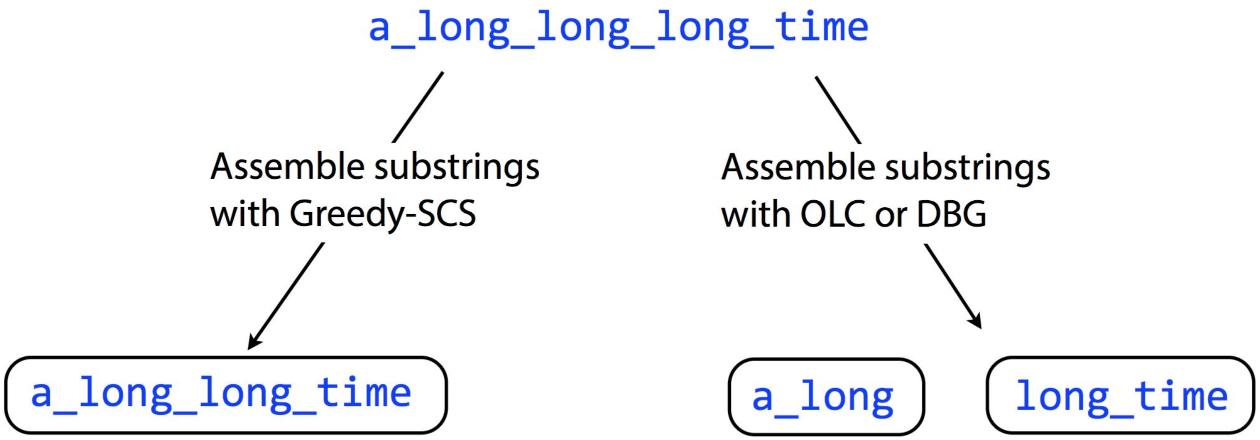
Overlap-layout-consensus

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).



a_long_long_long_time

Assemble substrings with Greedy-SCS

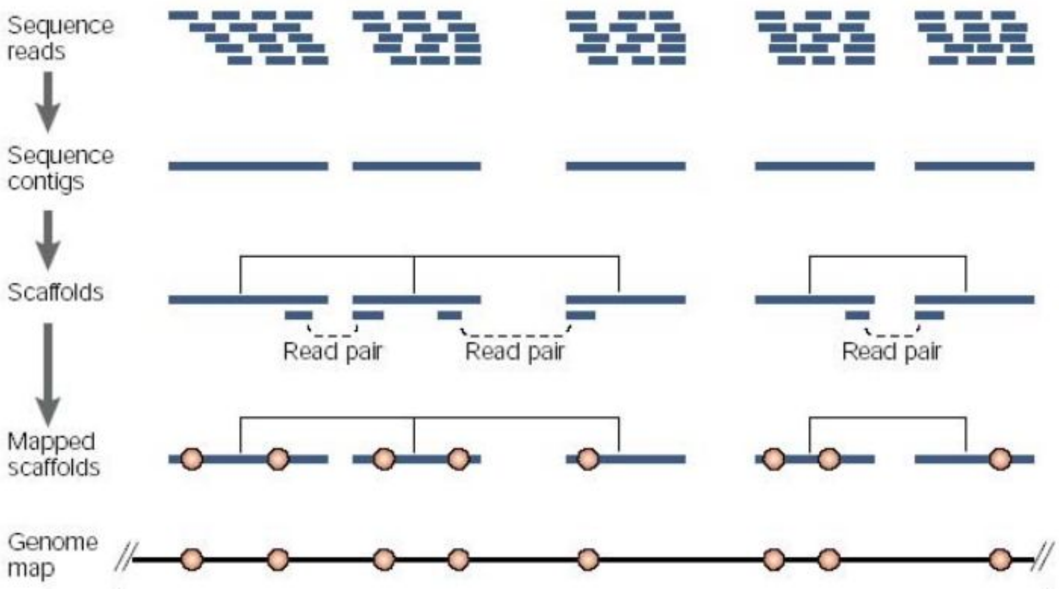
Assemble substrings with OLC or DBG

a_long_long_time

a_long

long_time

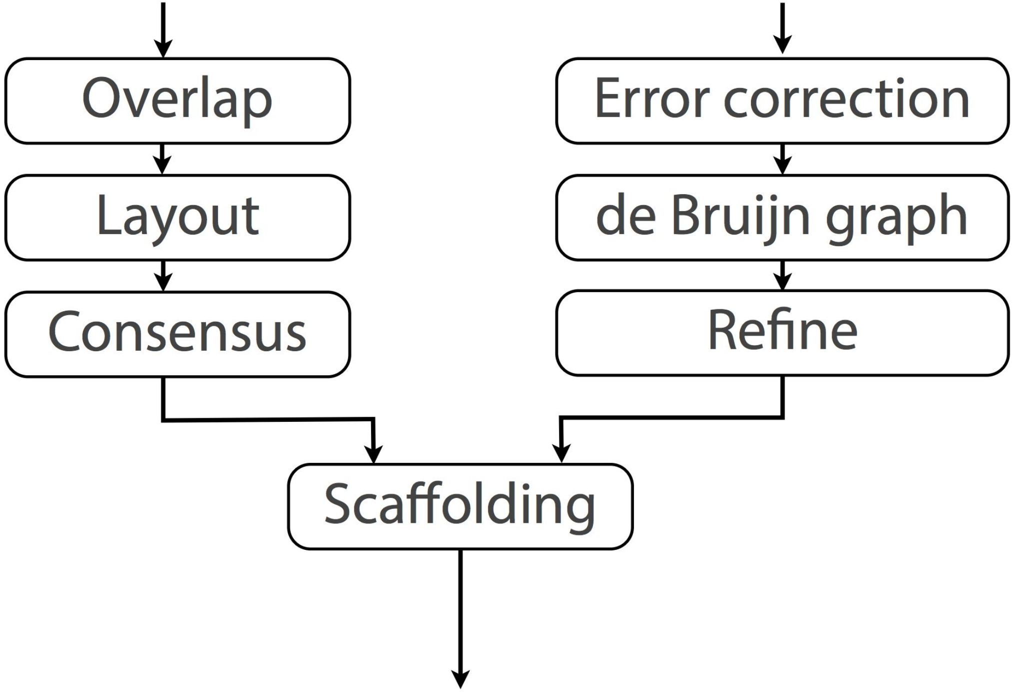
de novo whole-genome shotgun assembly



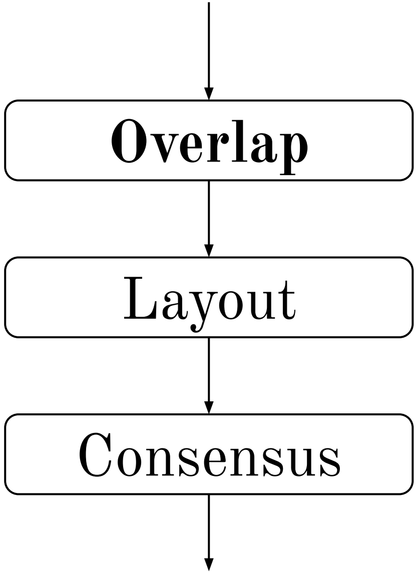
Assembly alternatives

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

Alternative 2: de Bruijn graph (DBG) assembly



Overlap Layout Consensus



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)

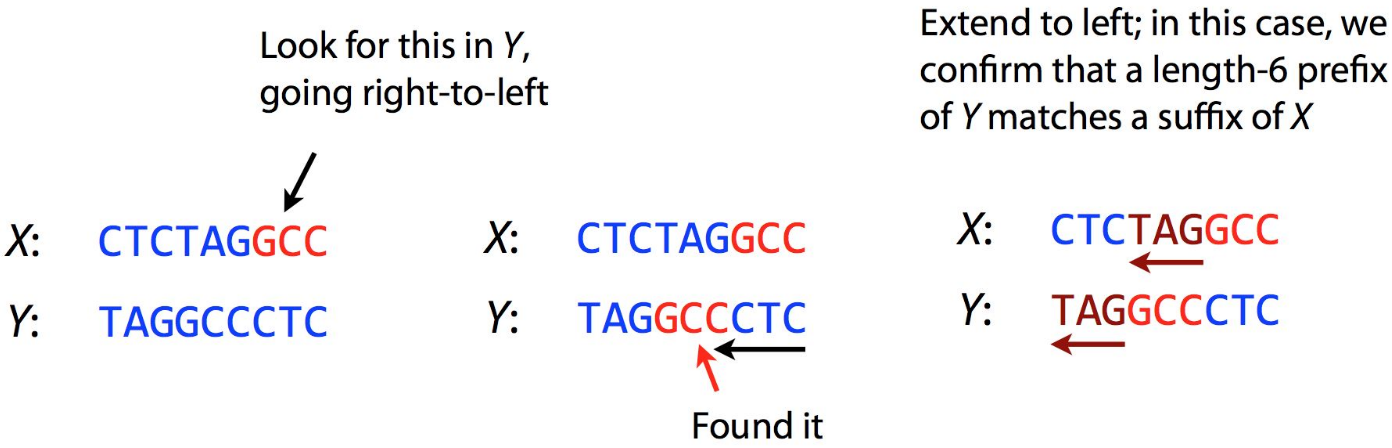
1. Naive exact matching
2. Suffix trees
3. FM index
4. Dynamic programming - inexact matching

Exact matching

Finding overlaps

Naive approach:

For 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?

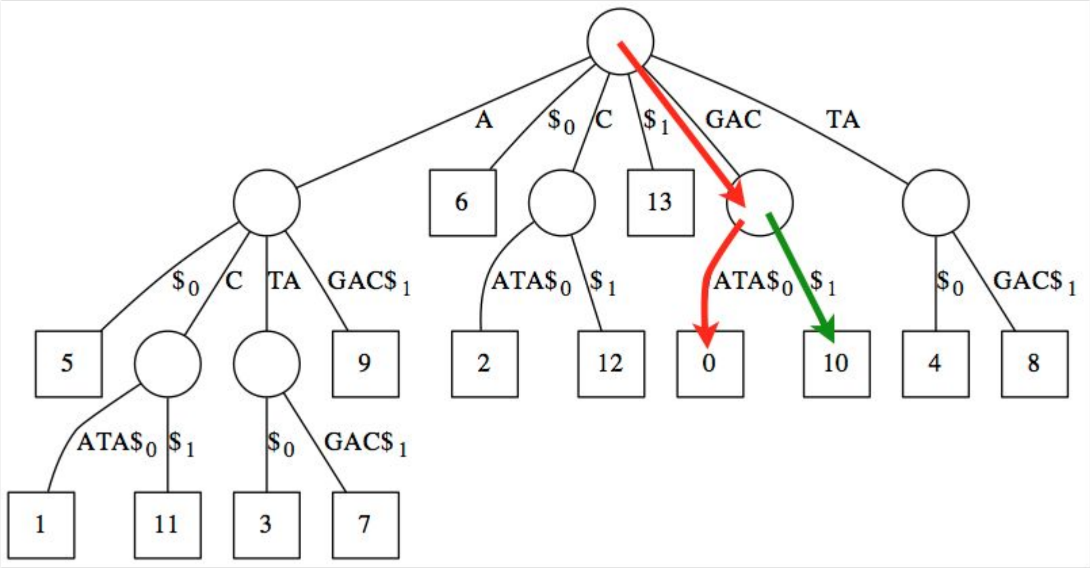
Problem: Given a collection of strings *S*, for each string *x* in *S* find all overlaps involving a prefix of *x* and a suffix of another string *y.*

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

Finding overlaps with suffix tree

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

$$\text{GACATA}\$_0\text{ATAGAC}\$_1$$



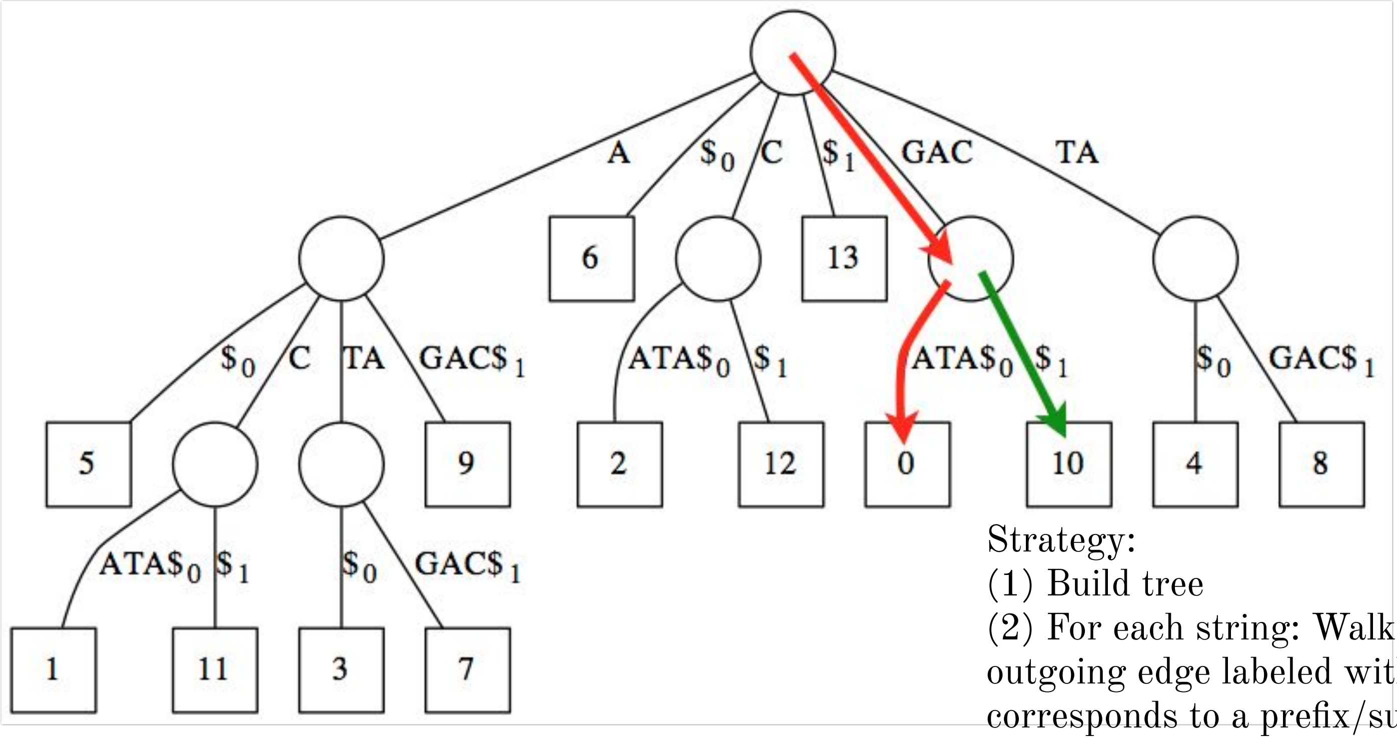
```
ATAGAC
   | | |
   GACATA
```

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” }

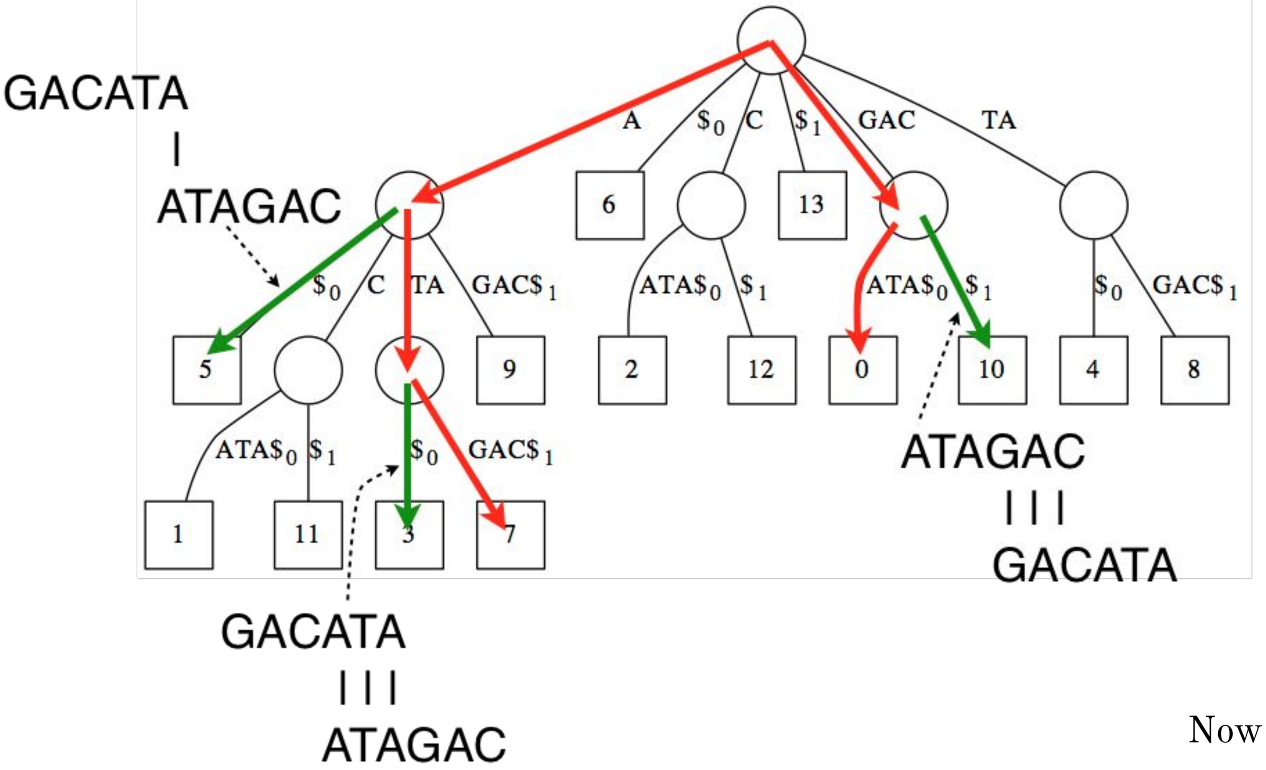


GACATA$\$_0$ATAGAC$\$_1$

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

Finding overlaps with suffix tree

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



GACATA
|
ATAGAC

ATAGAC
|||
GACATA

GACATA
|||
ATAGAC

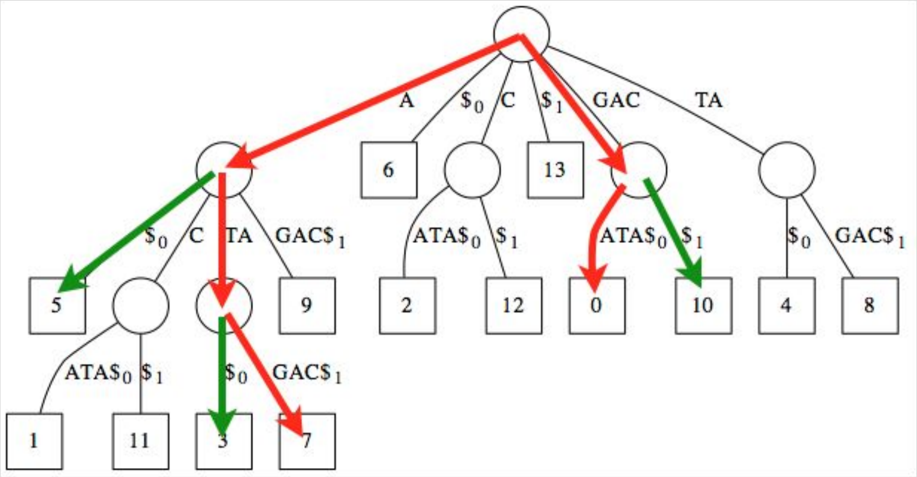
$\text{GACATA\$}_0\text{ATAGAC\$}_1$

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: $O(N)$

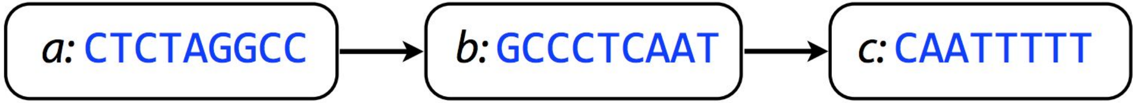
... to walk down red paths: $O(N)$

... 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?

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

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

Dynamic programming

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

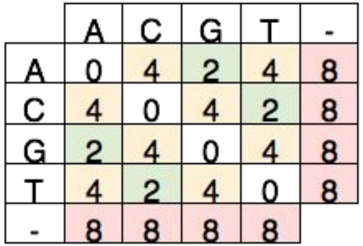
Finding overlaps with dynamic programming

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

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

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

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



	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.*

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

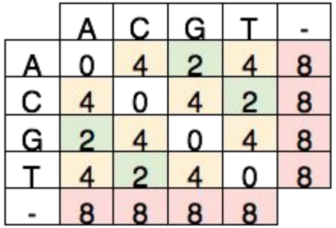
How to initialize first row & column so suffix of *X* aligns to prefix of *Y*? (remember *end-space free* variant)

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

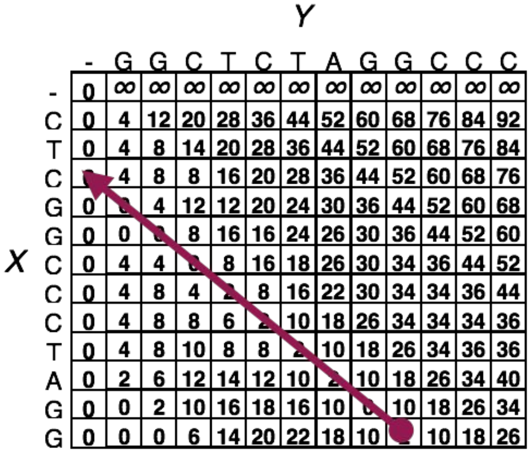
First row gets ∞s (must be a prefix of *Y*).

Backtrace from last row.

```
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

X

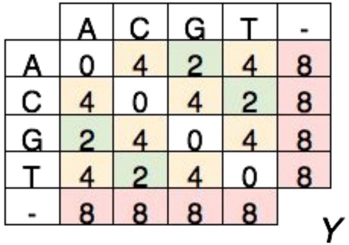
	-	G	G	C	T	C	T	A	G	G	C	C	C
-	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
C	0	4	12	20	28	36	44	52	60	68	76	84	92
T	0	4	8	14	20	28	36	44	52	60	68	76	84
C	0	4	8	8	16	20	28	36	44	52	60	68	76
G	0	0	4	12	12	20	24	30	36	44	52	60	68
G	0	0	0	8	16	16	24	26	30	36	44	52	60
C	0	4	4	0	8	16	18	26	30	34	36	44	52
C	0	4	8	4	2	8	16	22	30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

Finding overlaps with dynamic programming

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

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

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

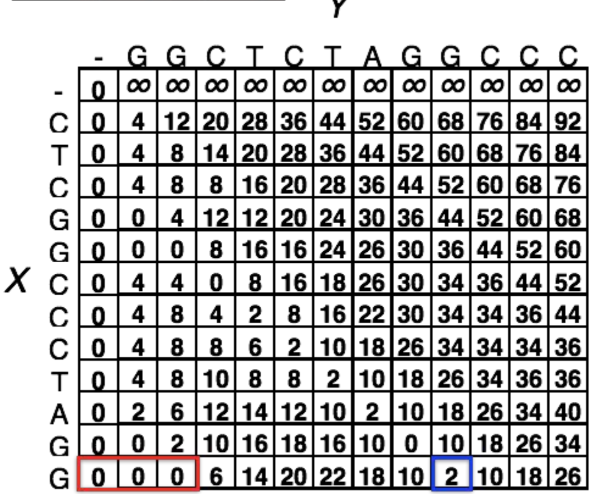


	A	C	G	T	-
A	0	4	2	4	8
C	4	0	4	2	8
G	2	4	0	4	8
T	4	2	4	0	8
-	8	8	8	8	

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

X

	-	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

Finding overlaps with dynamic programming

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

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

Solve by initializing certain additional cells to ∞

Cells whose values changed highlighted in red

Now the relevant match is the best candidate

```
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

		-	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
X	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

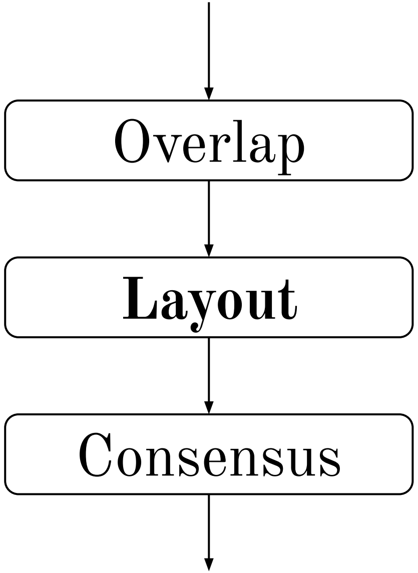
Celera Assembler overlapper is probably the best documented:

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

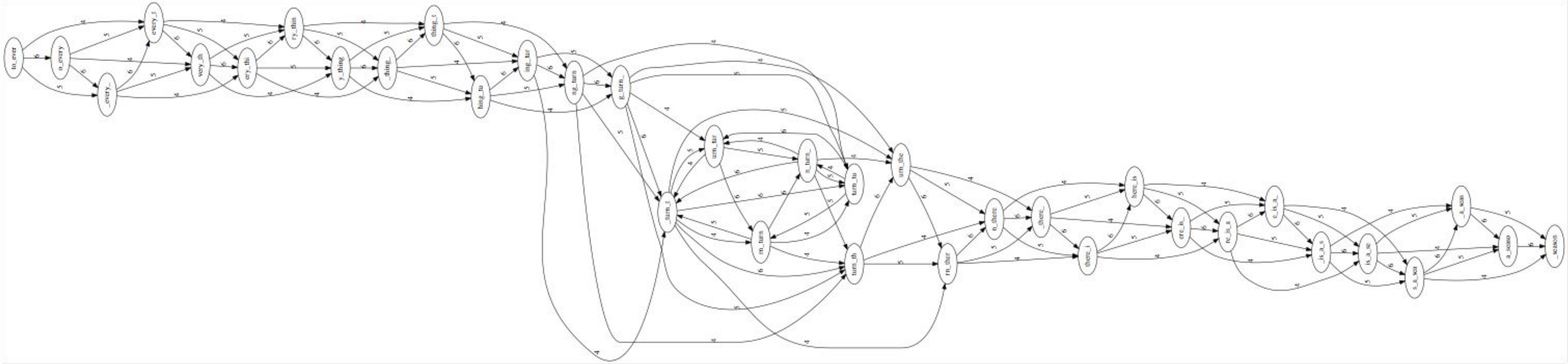
Layout

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

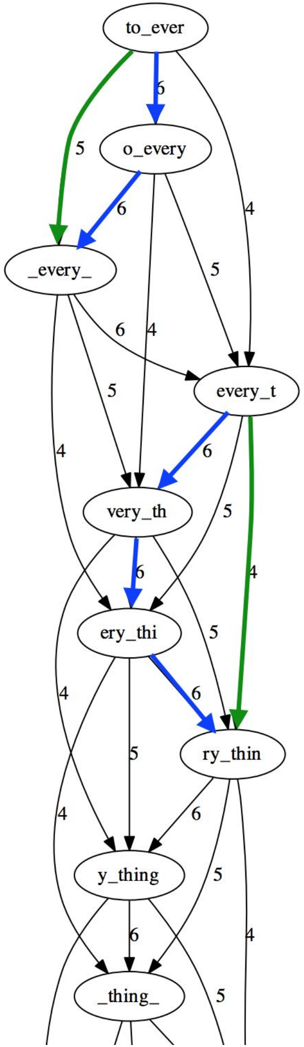


Layout

Anything redundant about this part of the overlap graph?

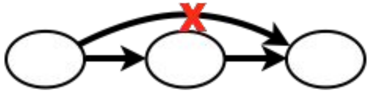
Some edges can be inferred (transitively) from other edges

E.g. green edge can be inferred from blue

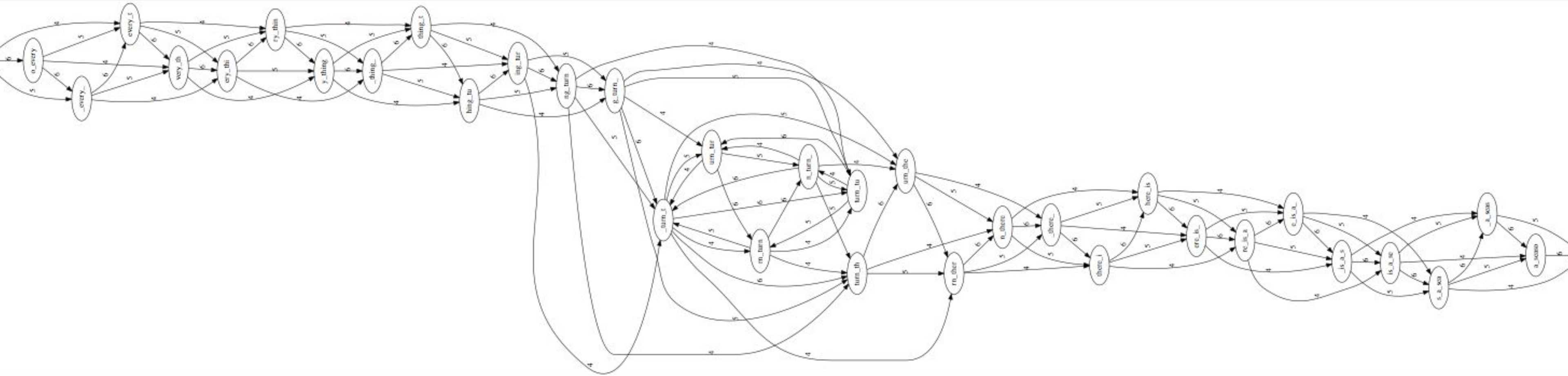


Layout

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

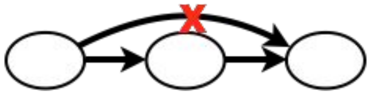


Before:

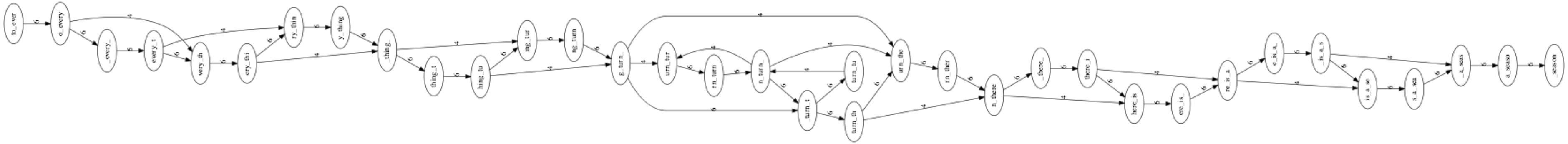


Layout

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

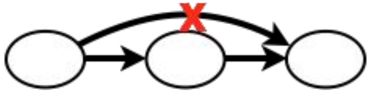


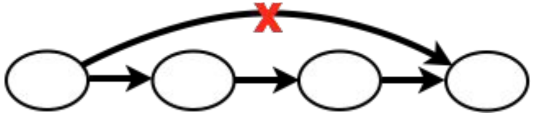
After:



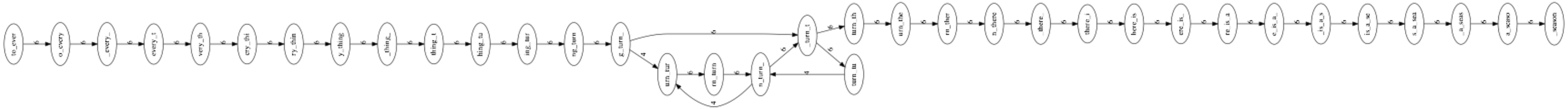
Layout

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





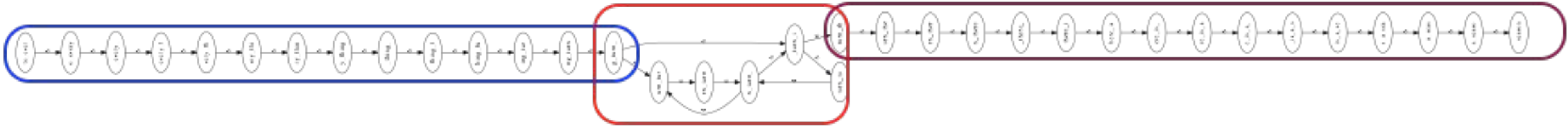
After:



Even simpler.

Layout

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



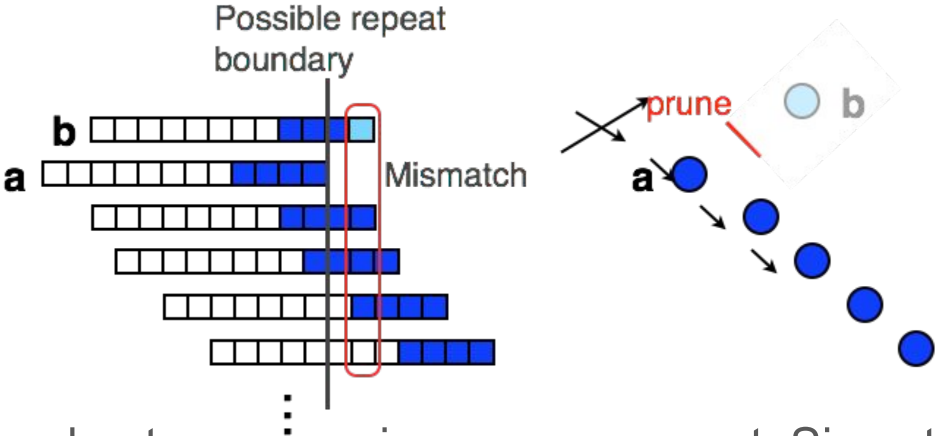
Contig 1
to_every_thing_turn_

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

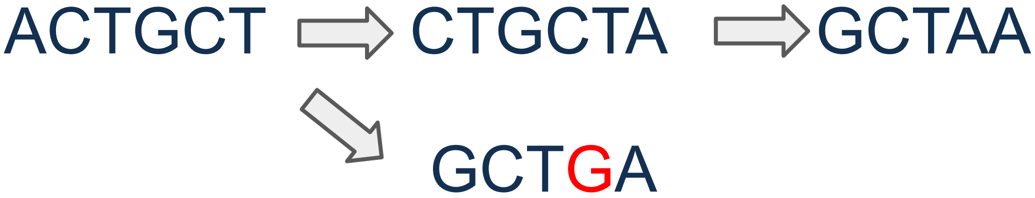


Mismatch could be due to sequencing error or repeat. Since the path through **b** ends abruptly we might conclude it’s an error and prune **b**.

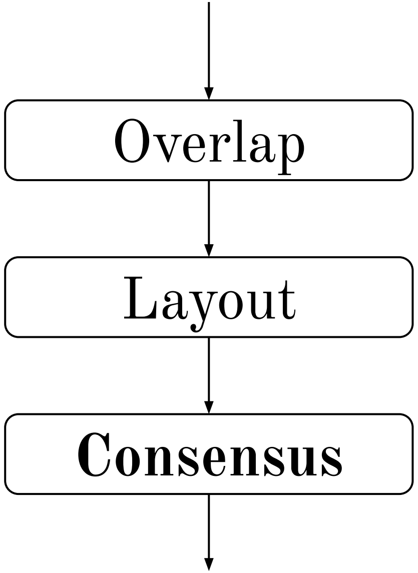
Layout

Effect of error on graph.

AACTGCT
CTGCTA
GCTAA
GCTGA



Overlap Layout Consensus

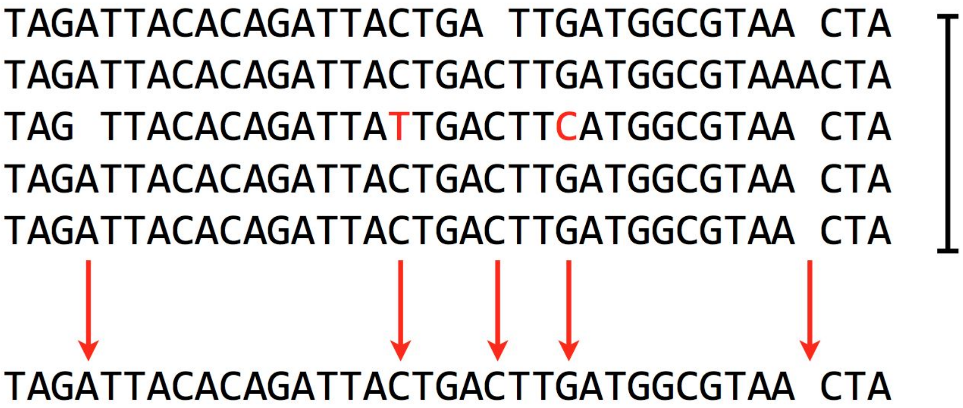


Overlap — Build overlap graph

Layout — Bundle stretches of the overlap graph into *contigs*

Consensus — **Pick most likely nucleotide sequence for each contig**

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

OLC drawbacks

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

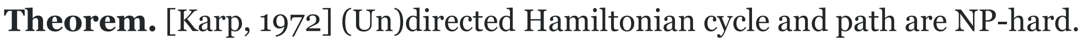
Overlap graph is big; one node per read, and in practice # edges grows superlinearly with # reads

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!

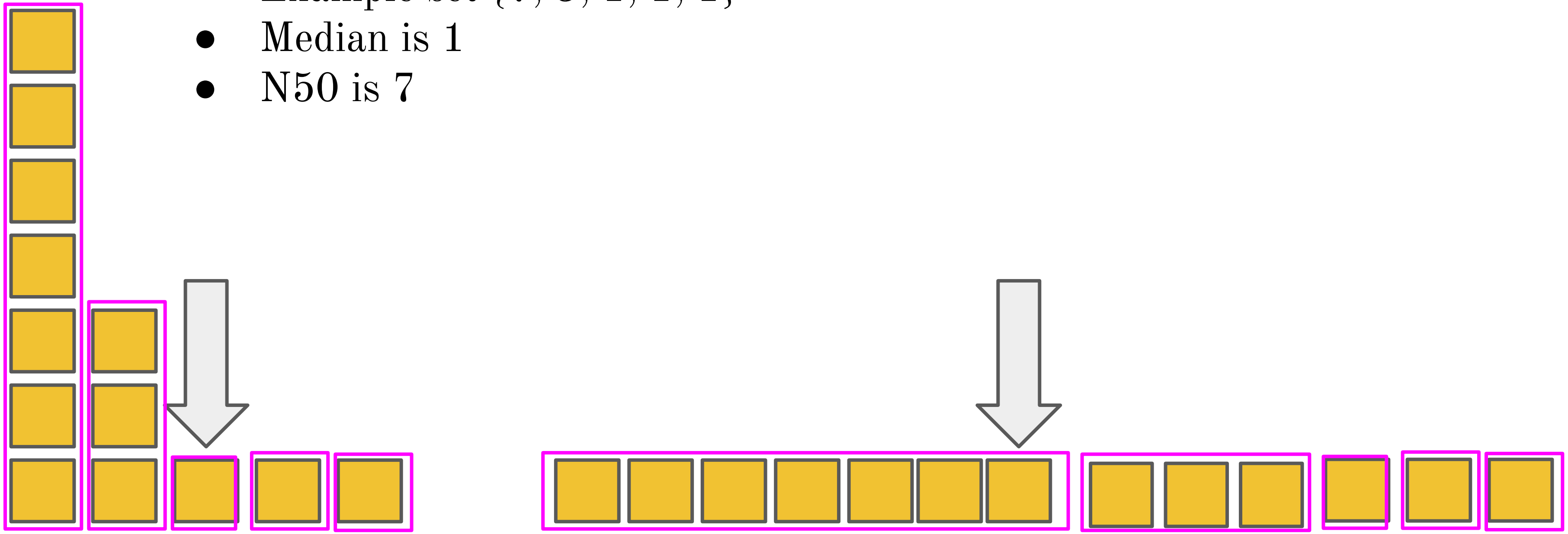
Assembly metrics

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

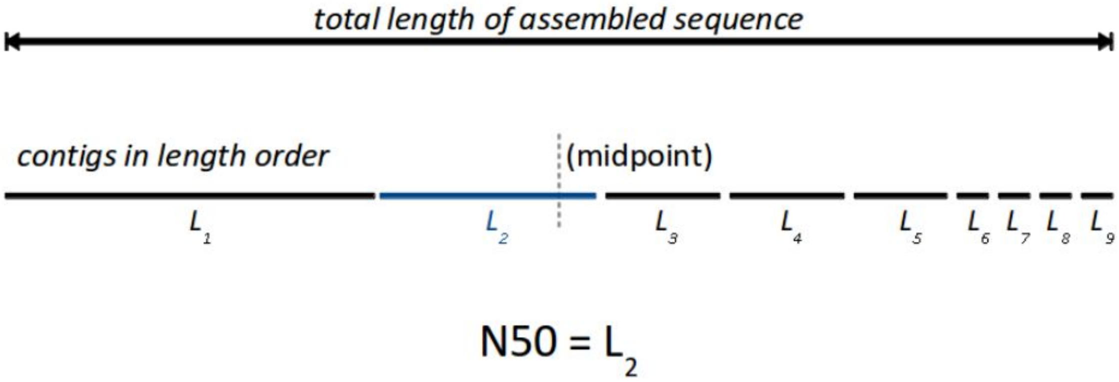
Nx / NGx of contigs / scaffolds

- 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

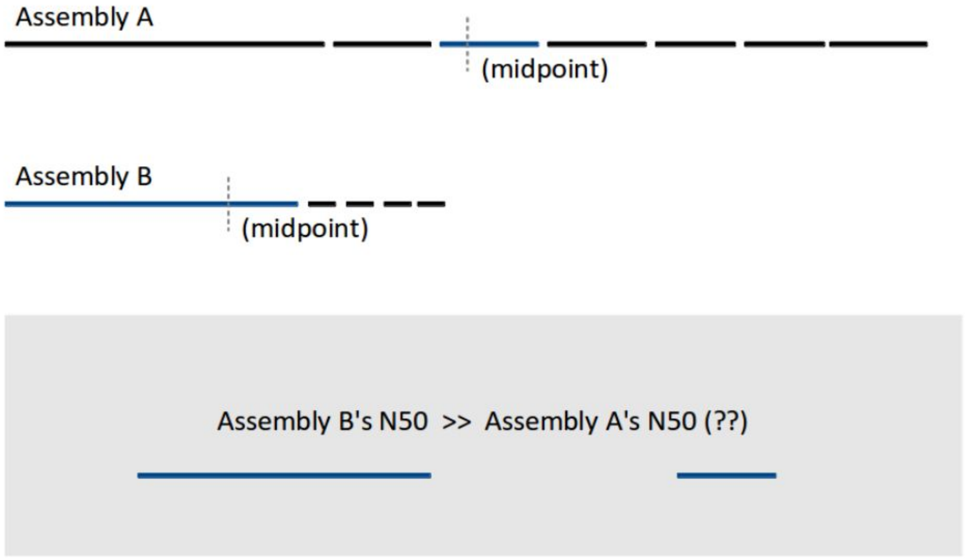
- **Nx** is largest contig length at which longer contigs cover **x%** of the total assembly length



- 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?



Assembly A

(midpoint)

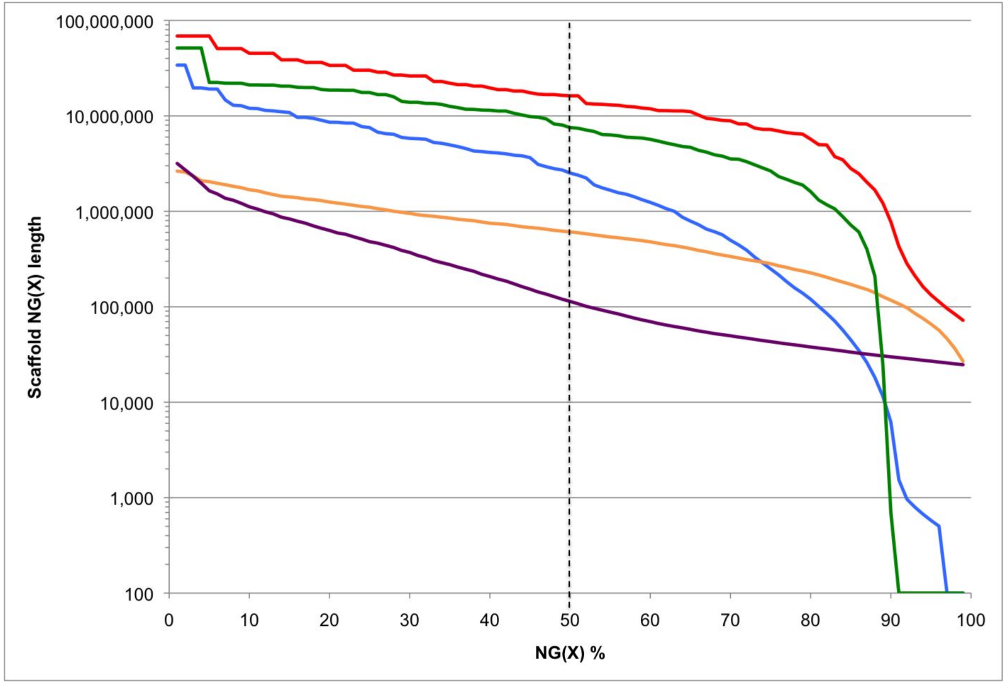
Assembly B

(midpoint)

Assembly B's N50 >> Assembly A's N50 (??)

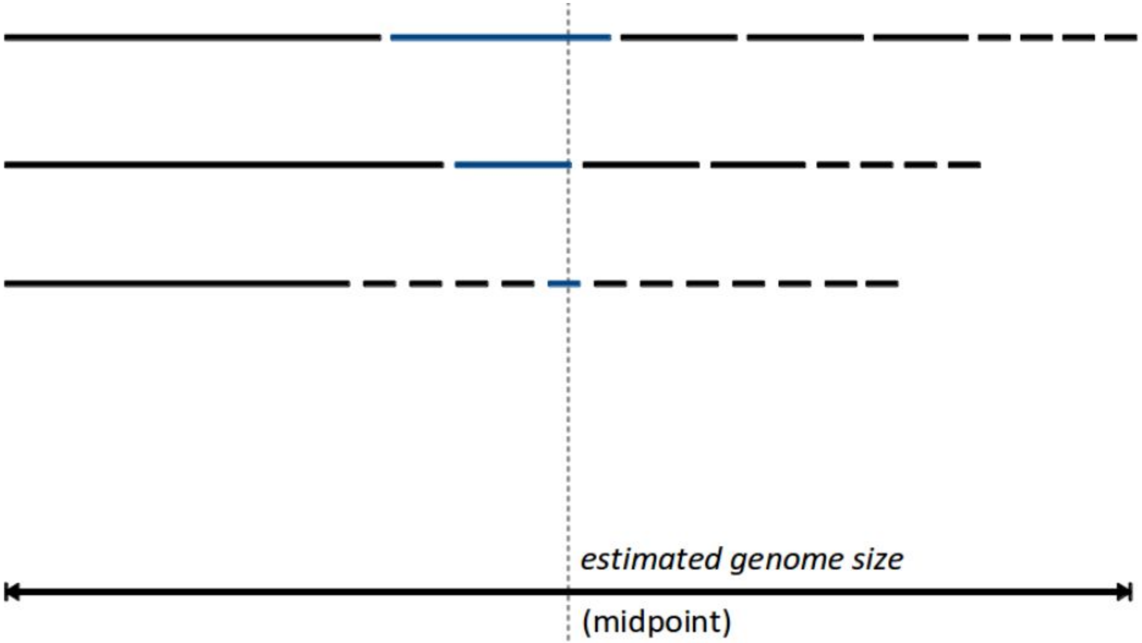
Nx / NGx of contigs / scaffolds

Assemblathon 2



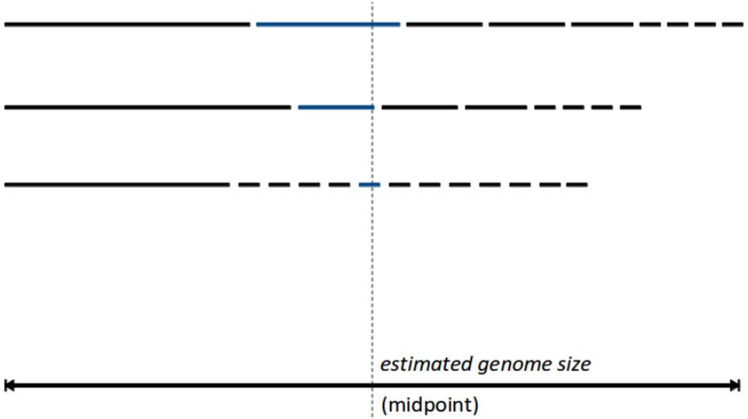
Nx / NGx of contigs / scaffolds

Solution - NGx



Nx / NGx of contigs / scaffolds

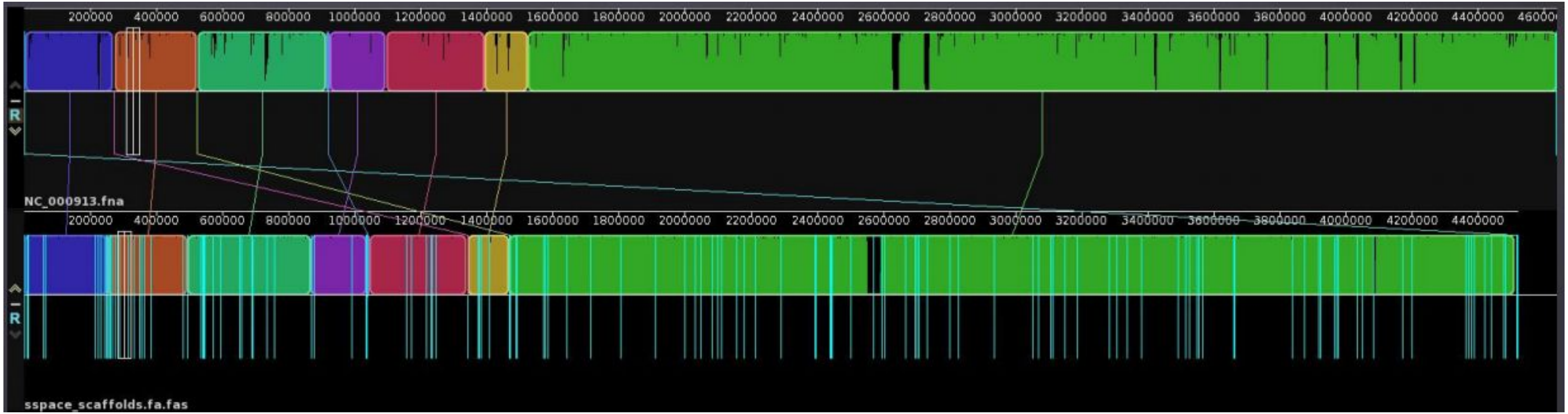
- **NGx** is largest contig length at which longer contigs cover **x%** of the total genome length



- 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



200000
400000
600000
800000
1000000
1200000
1400000
1600000
1800000
2000000
2200000
2400000
2600000
2800000
3000000
3200000
3400000
3600000
3800000
4000000
4200000
4400000
460000
R
NC_000913.fna
sspace_scaffolds.fa.fas