Lesson 7

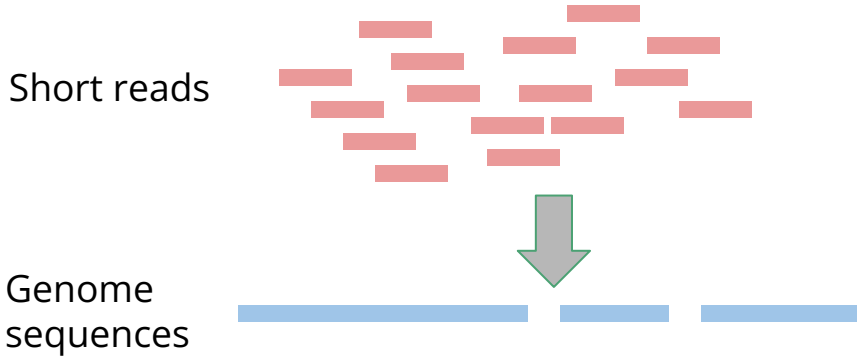
De novo genome assembly

By the end of this lesson you will...

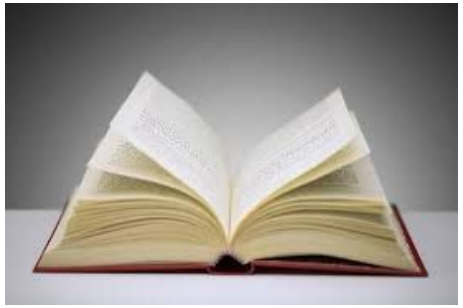
- Understand the basics of de novo genome assembly
- Be familiar with the DeBruijn graph method
- Know several methods and metrics for genome assembly QA
- Be able to perform assembly QA using QUAST and BUSCO

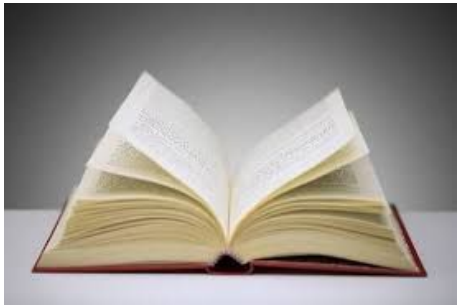
What is de novo genome assembly?

- Genome assembly - constructing long genomic sequences from shorter ones
- De novo = “from scratch”
- In NGS context - short reads ➔ whole genome, without any external reference

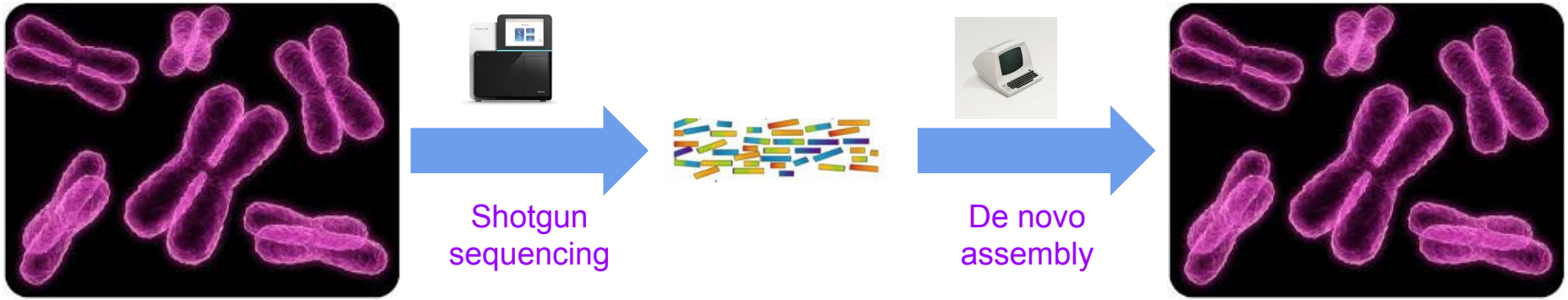


The assembly problem



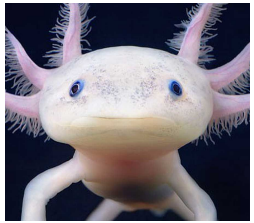


The assembly problem



Why do we need de novo assembly?

- Completely new organism



- Existing reference is too different from what we are interested in
- Identify large structural variation

- Detect novel sequences not present in the reference
- Cancer genomics

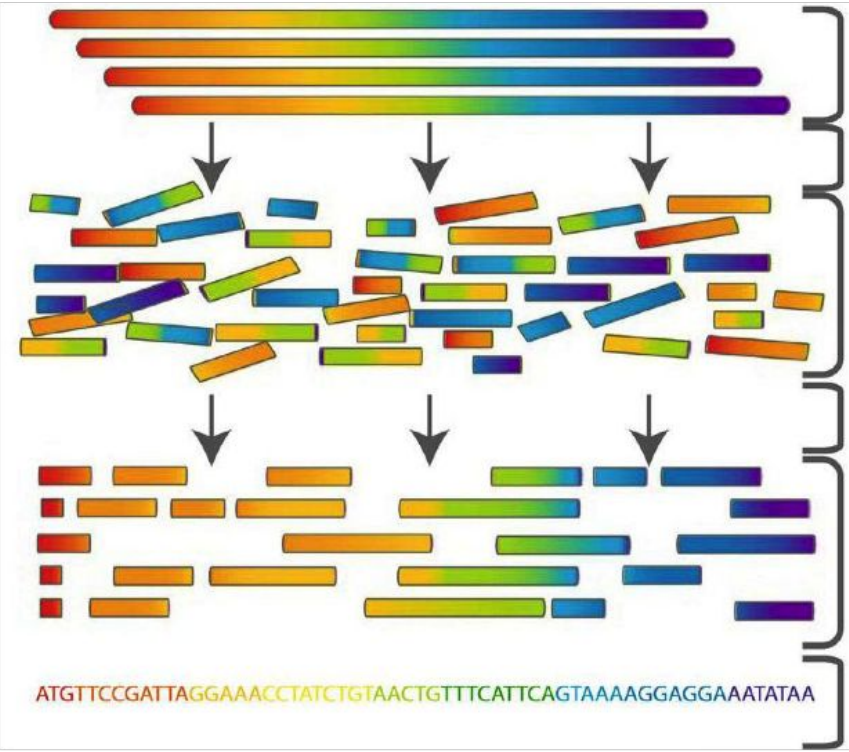
Genome assembly by reads overlap - challenges

- Do we believe short overlaps?
- How do we handle sequencing errors?
- Computationally ineffective for large data sets
- “Greedy” - high chance for non-optimal results

Bottom line: not good enough for large and complex genomes (e.g. human)

We need something smarter!

Genome assembly by reads overlap - challenges



Genomic DNA

Fragmentation + Sequencing

Sequence reads

Assembly

Connection between reads found

Consensus sequence

Suffix Prefix Matching

TCTATATCTCGGCTCTAGG

TATCTCGACTCTAGGCC

Suffix Prefix Matching

TCTATATCTCGGCTCTAGG

TATCTCGACTCTAGGCC

Suffix Prefix Matching

TCTATATCTCGGCTCTAGG

GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

TATCTCGACTCTAGGCC

If a suffix of read A is similar to a prefix of read then A and B might overlap in the genome

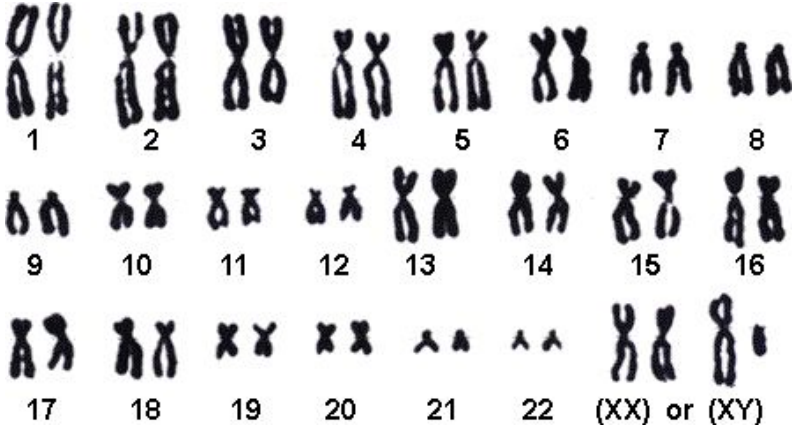
Suffix Prefix Differences

TCTATATCTCGGCTCTAGG

TATCTCGACTCTAGGCC

Why the differences?

1. Sequencing errors
2. Polyploidy



High and Low Coverage

CTAGGCCCTCAATTTT
GGCTCTAGGCCCTCATTTTT
CTCGGCTCTAGGCCCTCATTT
TATCTCGACTCTAGGCC
TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCTATATCT

More coverage

GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
CTAGGCCCTCAATTTT
TATCTCGACTCTAGGCCCTCA
GGCGTCTATATCT

Less coverage

More coverage leads to more and longer overlaps

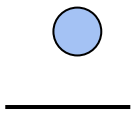
Overlap Consensus

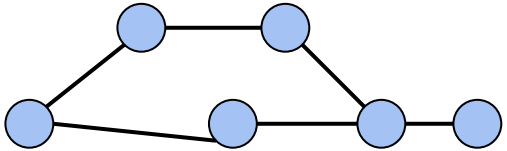
What is the best representation to the set of sequences?

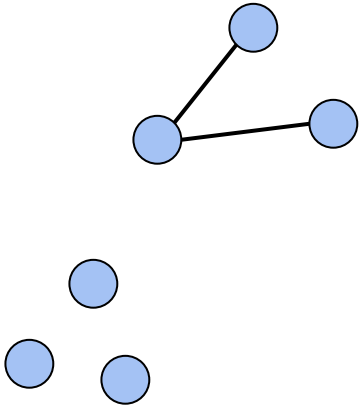
Graph

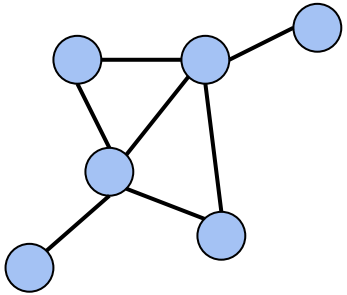
A graph is a set of:

- Nodes (vertices)
- Edges - connecting two nodes



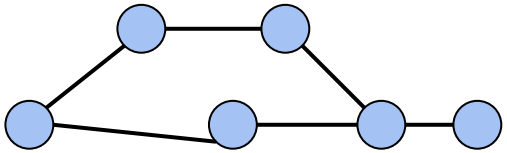


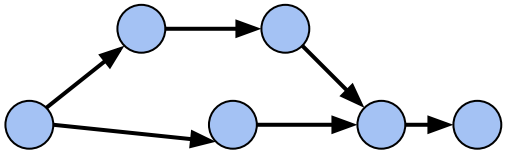


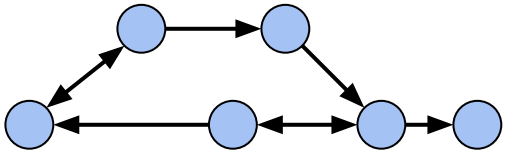


Directed and Weighted Graphs

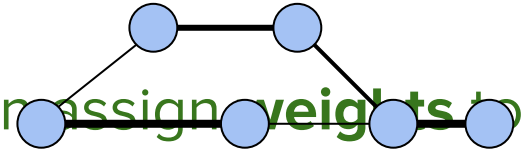
Graphs can be **directed** or **non-directed**

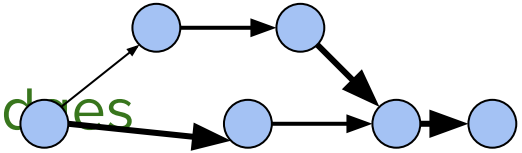






We can assign **weights** to edges



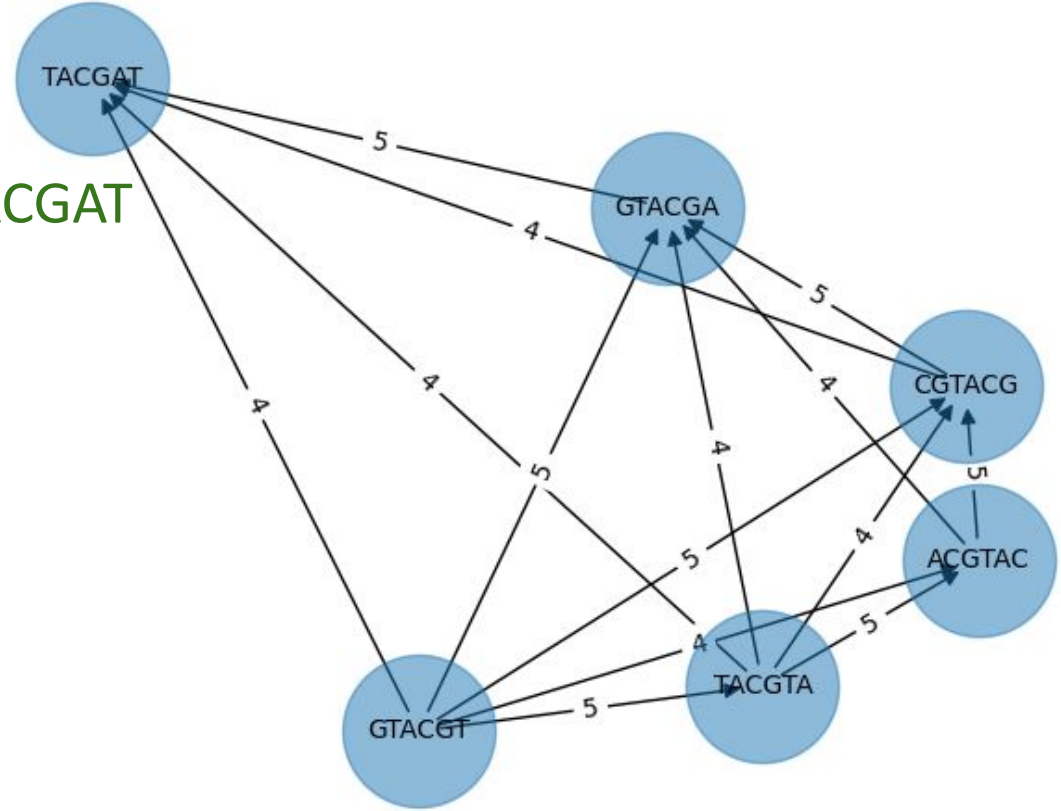


Overlap Consensus Graph

Nodes: all 6-mers in GTACGTACGAT

Edges: overlaps of length > 3

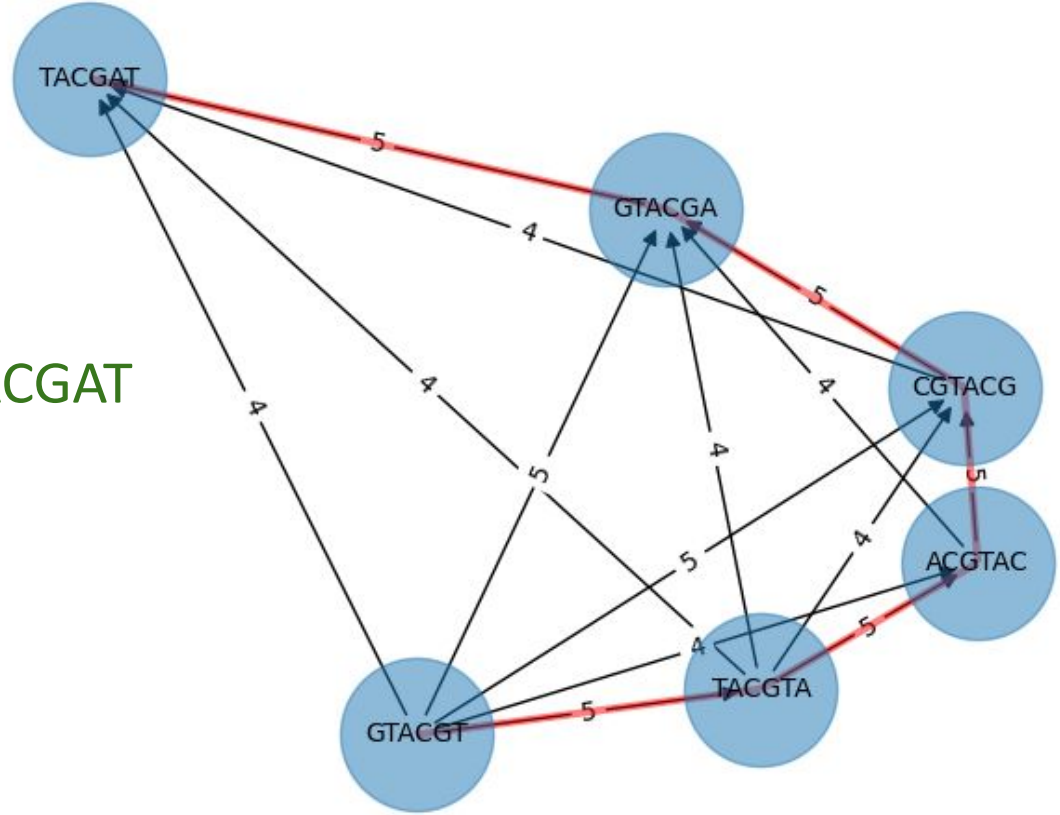
GTACGT
TACGTA
ACGTAC
CGTACG
GTACGA
TACGAT



Overlap Consensus Graph

Nodes: all 6-mers in GTACGTACGAT

Edges: overlaps of length > 3



Shortest Common Superstring

The Shortest Common Superstring problem (SCS) aim to find the shortest possible string that contains every string in a given set as substrings

Example: BAA AAB BBA ABA ABB BBB AAA BAB

Concatenation: BAAAABBBAABAABBBBBAAABAB

24

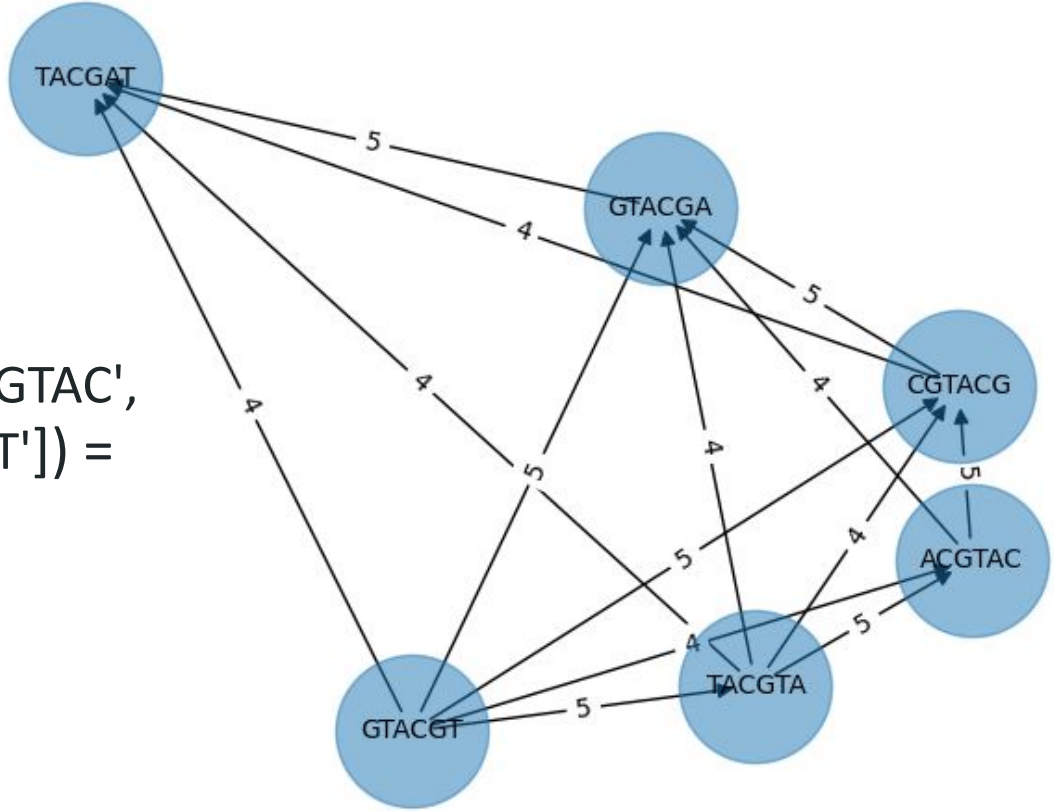
SCS: AAABBBABAA

10

AAA
AAB
ABB
BBB
BBA
BAB
ABA
BAA

Shortest Common Superstring

SCS(['GTACGT', 'TACGTA', 'ACGTAC', 'CGTACG', 'GTACGA', 'TACGAT']) = GTACGTACGAT



Shortest Common Superstring

Brute Force

Order 1: AAA AAB ABA ABB BAA BAB BBA BBB

AAABABBAABABBABBB Superstring 1

Order 2: AAA AAB ABA BAB ABB BBB BAA BBA

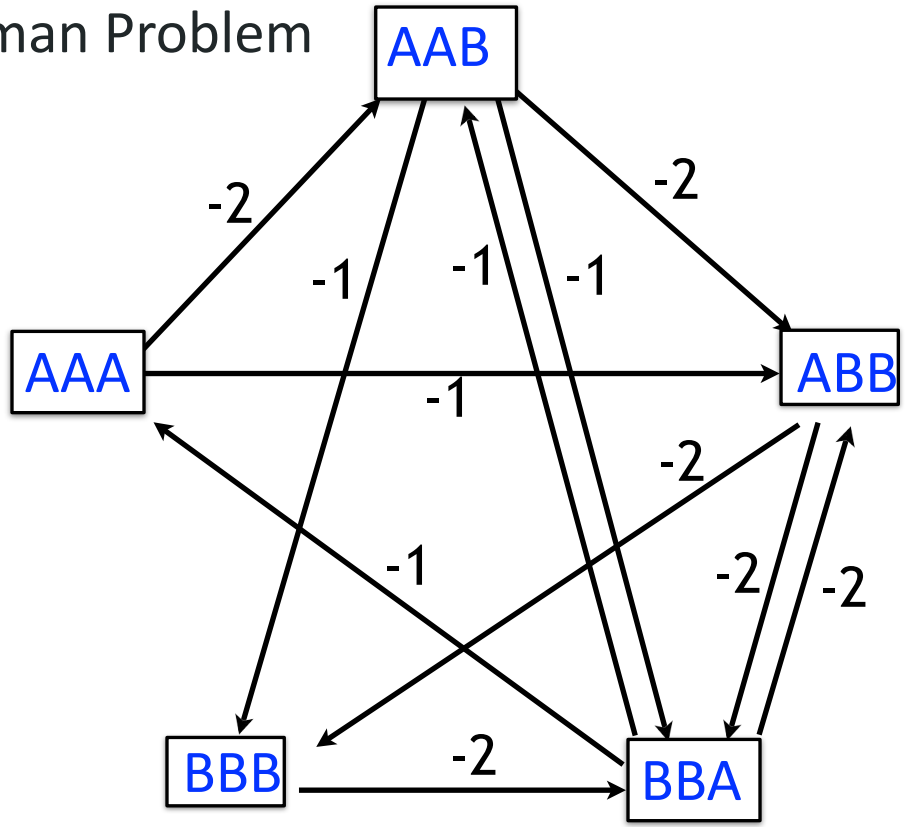
AAABABBBAABBA Superstring 2

O(n!)

Shortest Common Superstring

Traveling Salesman Problem

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



Shortest Common Superstring

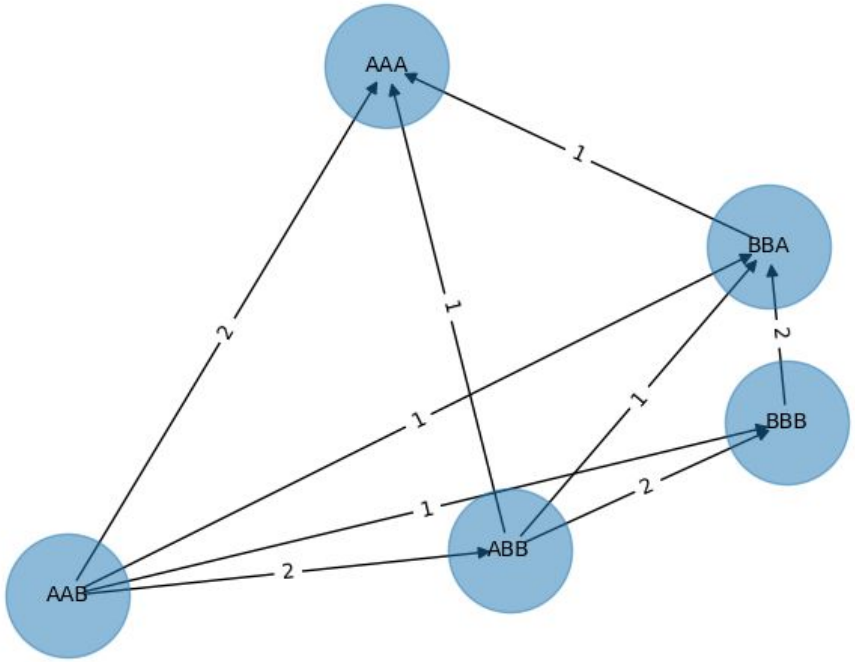
NP-complete

No efficient solution algorithm has been found

Shortest Common Superstring

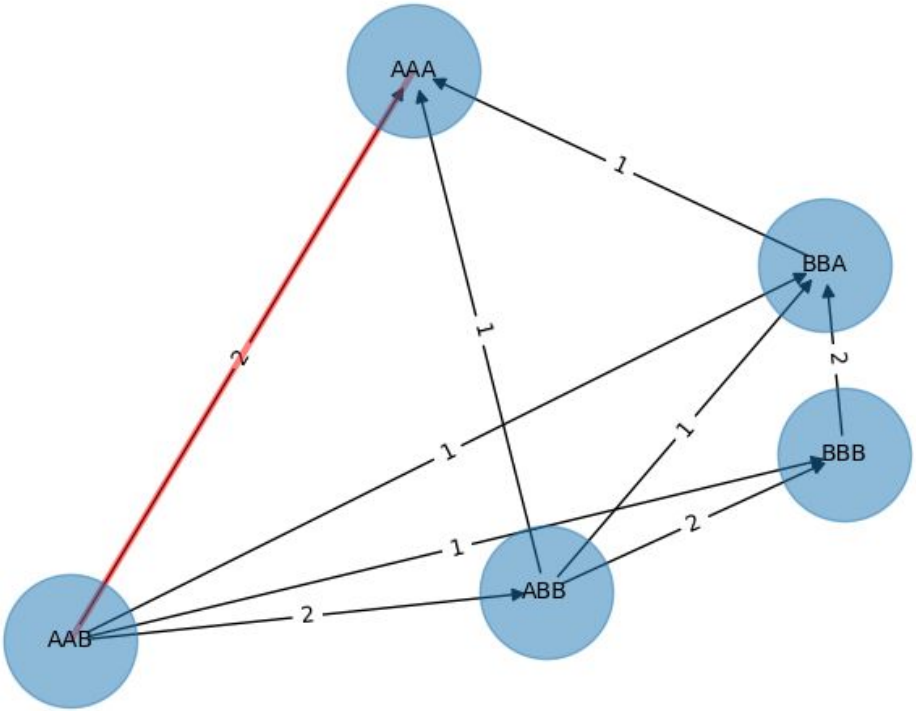
Greedy

Example: BAA AAB BBA ABA ABB BBB AAA BAB



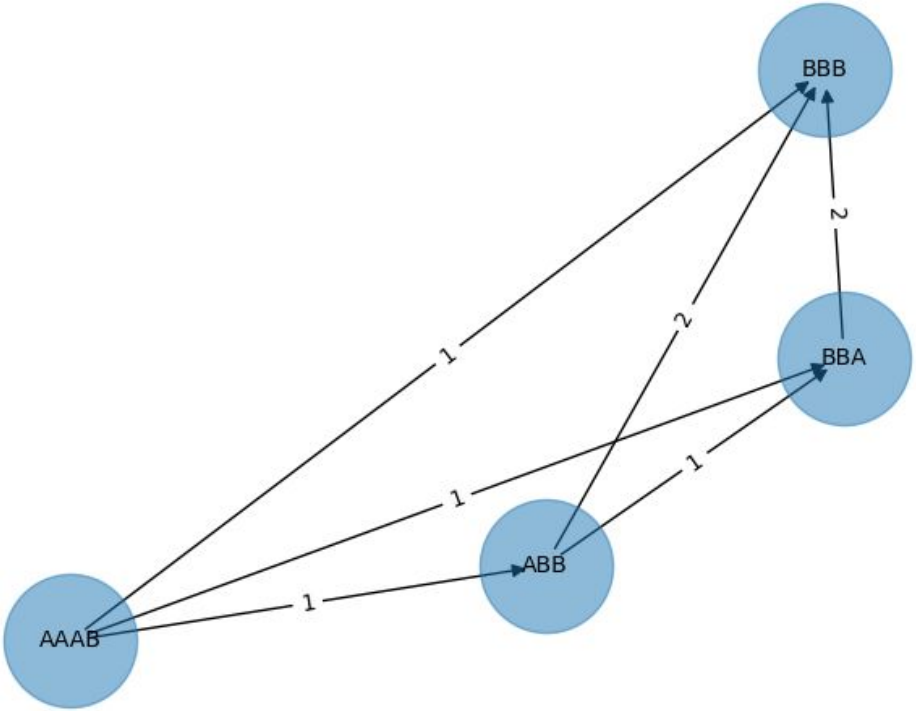
Shortest Common Superstring

Greedy



Shortest Common Superstring

Greedy



Shortest Common Superstring

Greedy

Superstring, length 7

Alternative Shuffling

Superstring, length 9

Greedy answer isn’t necessarily optimal

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

Gusfield, Dan. "Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology." (1997).

Shortest Common Superstring

Greedy

a_*long*_*long*_*long*_time *l=6*

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

Missing a _long

Shortest Common Superstring

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

a_long_long_long_time

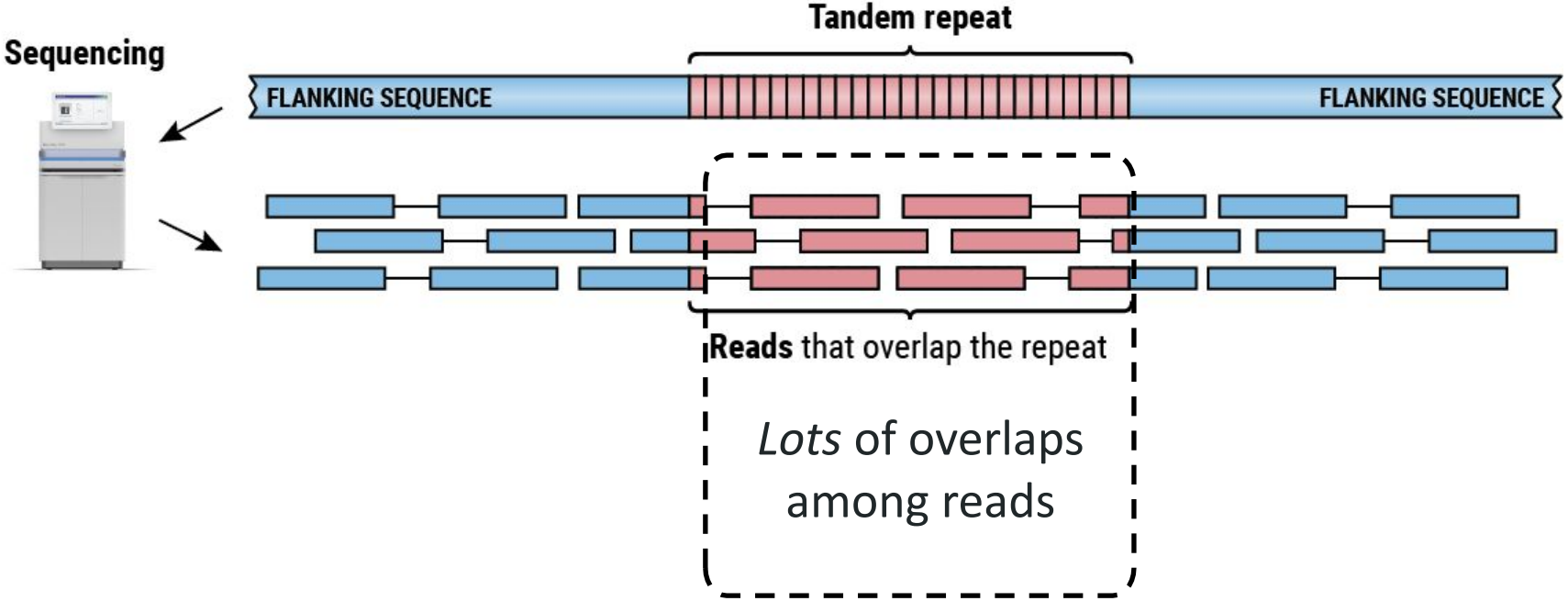
collapse

a_long_long_time

The human genome is ~ 50% repetitive!

Shortest Common Superstring

Repeats

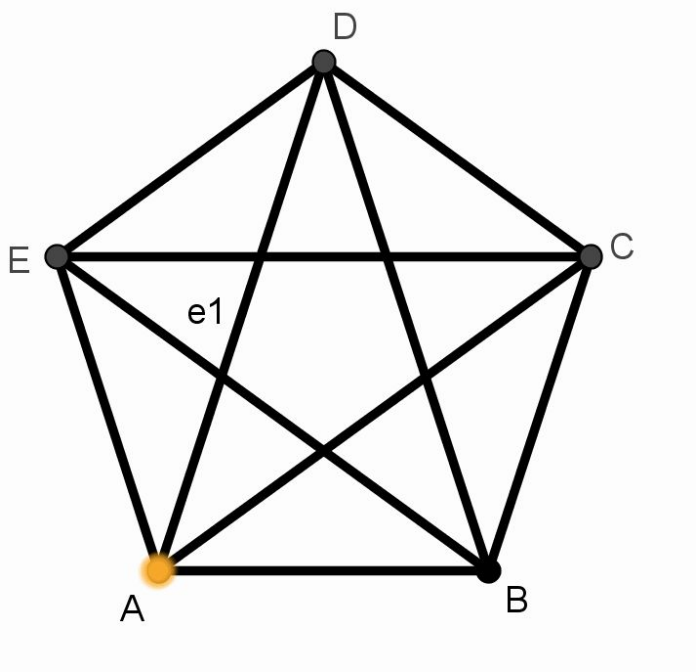


De Bruijn Graph assembly

- Currently the most popular assembly method
- Many software tools use variants of the method
  - SPAdes
  - SOAPdenovo
  - AbySS
  - MEGAHIT
- Based on graph theory
- Specifically **k-mer graphs**

The Eulerian path

- A path in a graph that visits every **edge** exactly once
  - Must visit **all** edges
  - Can’t visit an edge twice
  - Can visit a node more than once
- A common problem in graph theory



K-mers

AGATCCAGCGAGGTCGCTATCCGTTAATTG

5-mers
AGATC
GATCC
ATCCA
…
AATTG

K-mers

AGATCCAGCGAGGTCGCTATCCGTTAATTG

5-mers
AGATC
GATCC
ATCCA
…
AATTG

How many 21-mers are in a 100 bp read?

7-mers
AGATCCA
GATCCAG
ATCCAGC
...
TTAATTG

De Bruijn Graph assembly - the basic algorithm

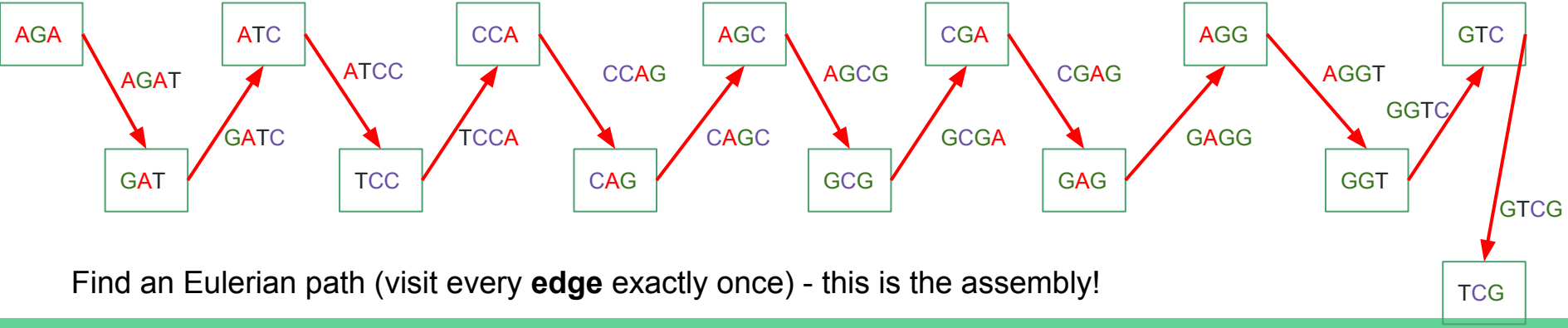
Genome (G=30): AGATCCAGCGAGGTCGCTATCCGTTAATTG

Reads (L=10): AGATCCAGCG
AGCGAGGTCG
GCTATCCGTT
CCGTTAATTG

Break into k-mers (k=4):
AGATCCAGCG → AGAT GATC ATCC TCCA CCAG CAGC AGCG
AGCGAGGTCG → AGCG GCGA CGAG GAGG AGGT GGTC GTCG
...

Create graph nodes - each **unique prefix and suffix** of length k-1 of k-mers

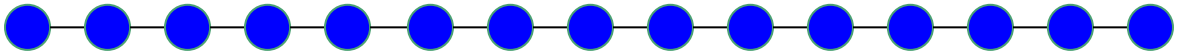
Add directed edge between node x and node y if k-mer exists with prefix x and suffix y



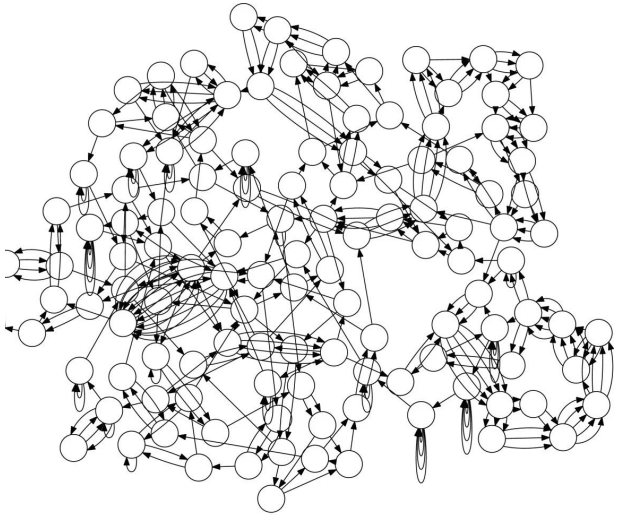
Find an Eulerian path (visit every **edge** exactly once) - this is the assembly!

If only life was that simple...

Ideally, we want our graph to look like this:

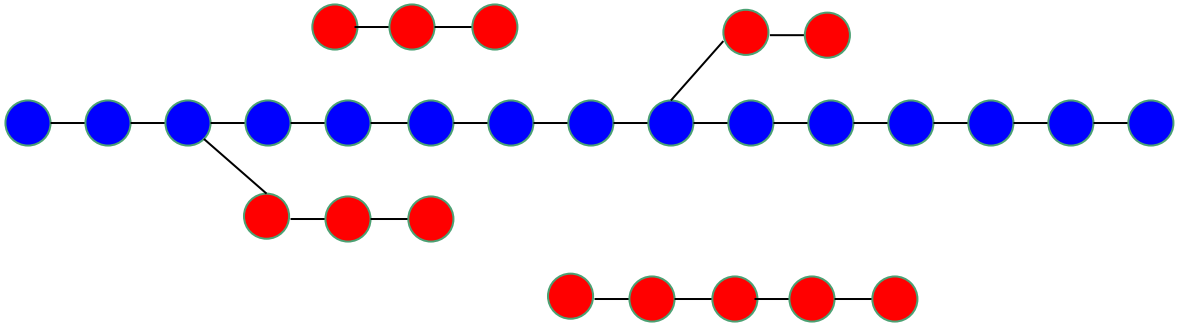


But in practice:



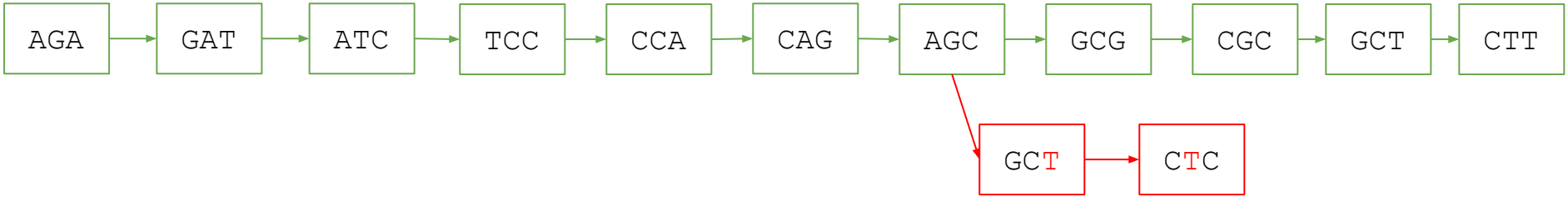
Sequencing errors

- Side branches
- Disconnected bits



Sequencing errors

```
AGATCCAGCG      → AGAT GATC ATCC TCCA CCAG CAGC AGCG
 GATCCAGCTC     → GATC ATCC TCCA CCAG CAGC AGCT GCTC
   TCCAGCGCTT   → TCCA CCAG CAGC AGCG GCGC CGCT GCTT
```

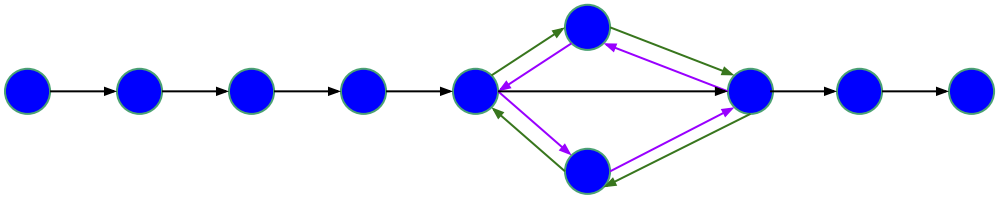


Genomic repeats

- Tandem duplication

- Interspersed duplications

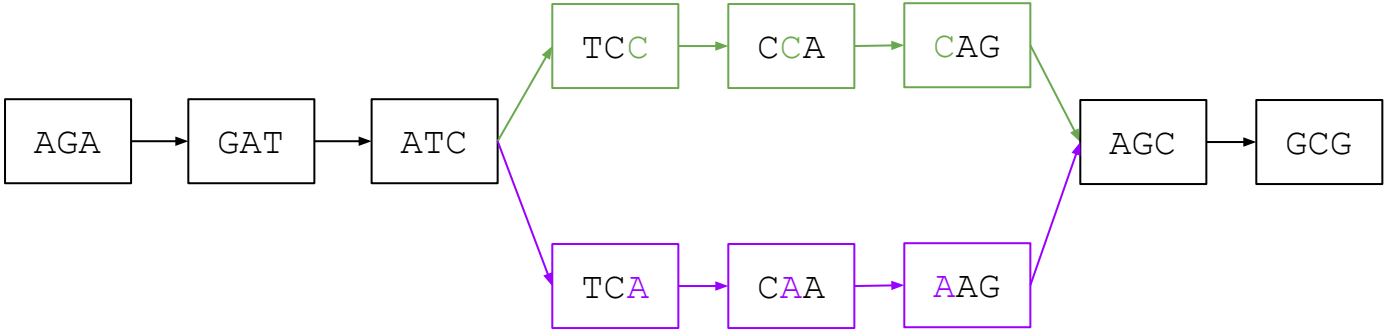
- Might create ambiguity - multiple possible eulerian paths



Heterozygosity

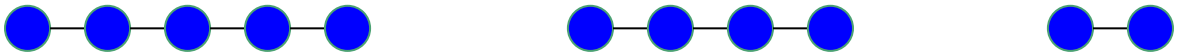
- Creates “bubbles” in the graph

Maternal AGATCCAGCG → AGAT GATC ATCC TCCA CCAG CAGC AGCG
Paternal AGATCAAGCG → AGAT GATC ATCA TCAA CAAG AAGC AGCG

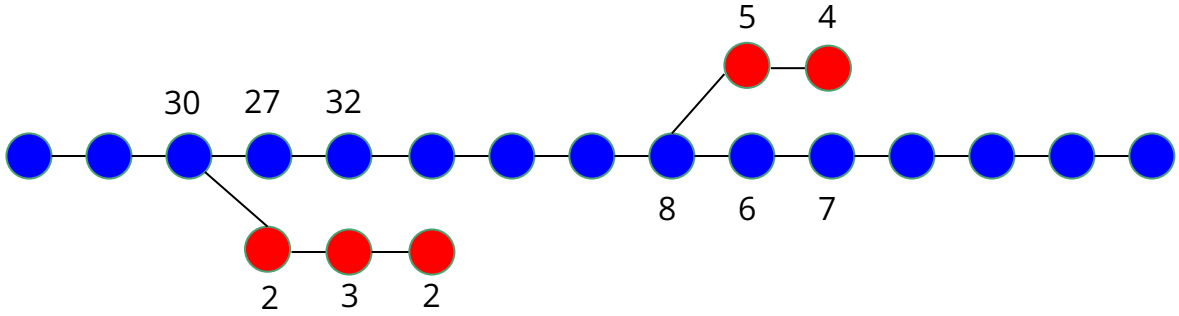


Uneven sequencing depth

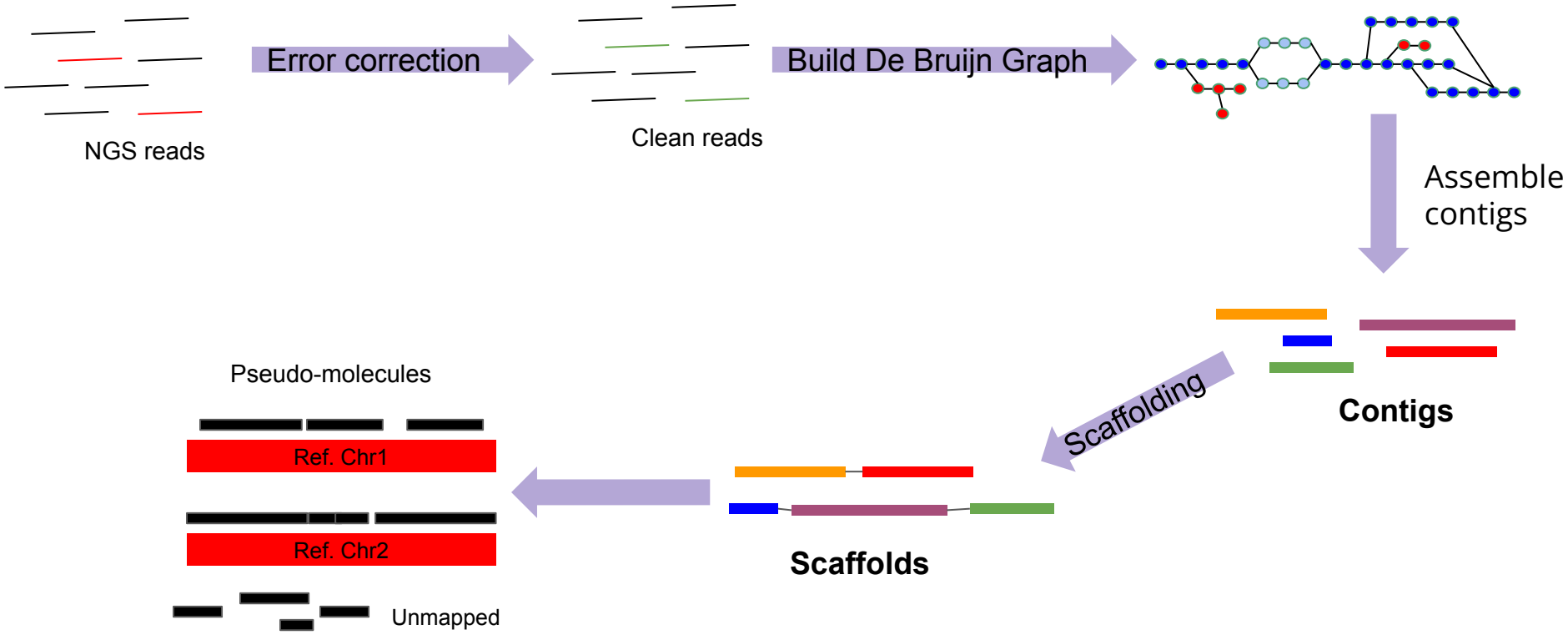
Very low depth (e.g. low complexity regions) can fragment the graph:



Uneven depth can make it hard to determine which branches are true and which are noise



General assembly workflow

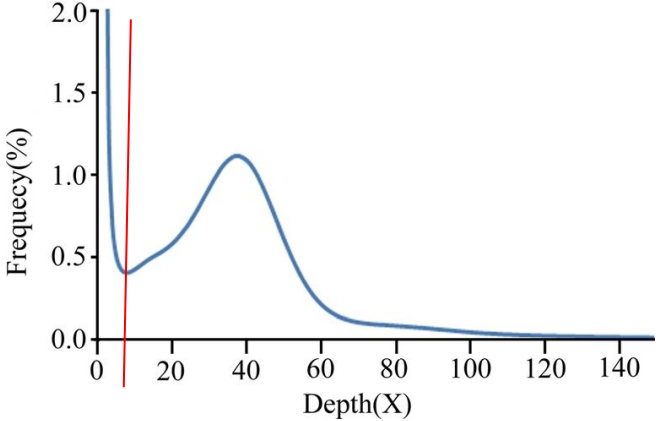


Error correction/filtration

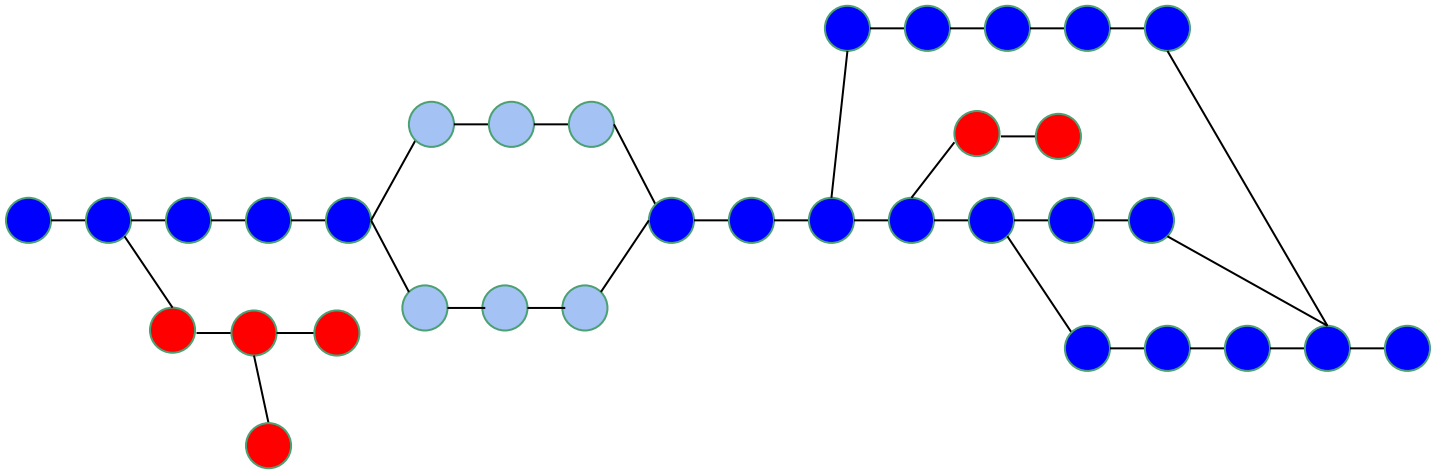
1. Extract all k-mers from all reads
2. Count how many times each k-mer was observed
3. Label rare k-mers as error k-mers
4. Find reads from which error k-mers came
5. Discard or correct the reads with error k-mers

Count(GGATAGGCACCAGTTAT) = 30

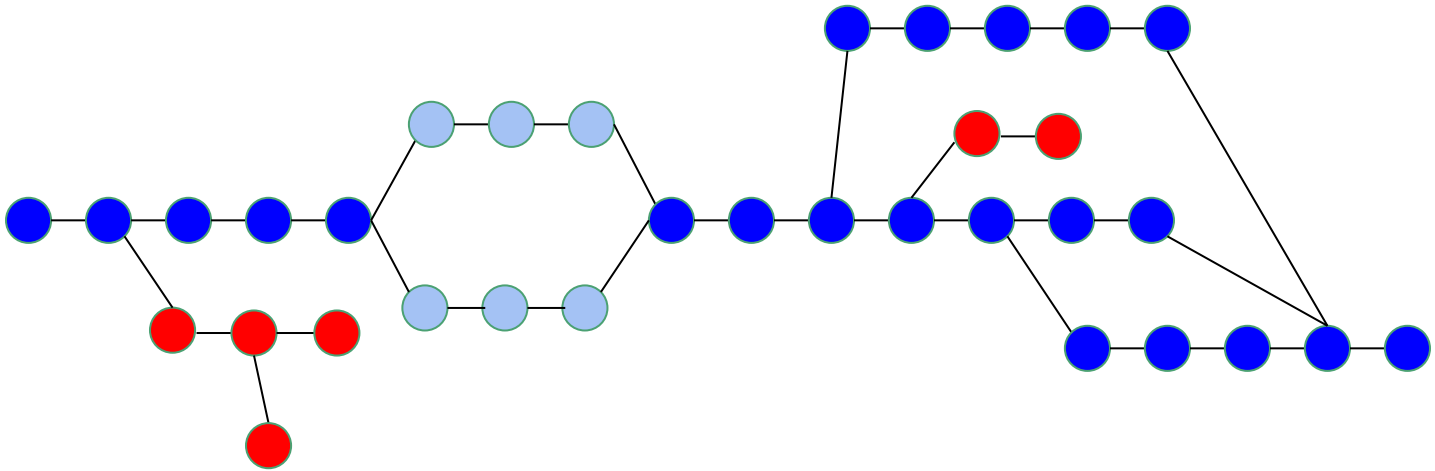
Count(GGATAGGTACCAGTTAT) = 1



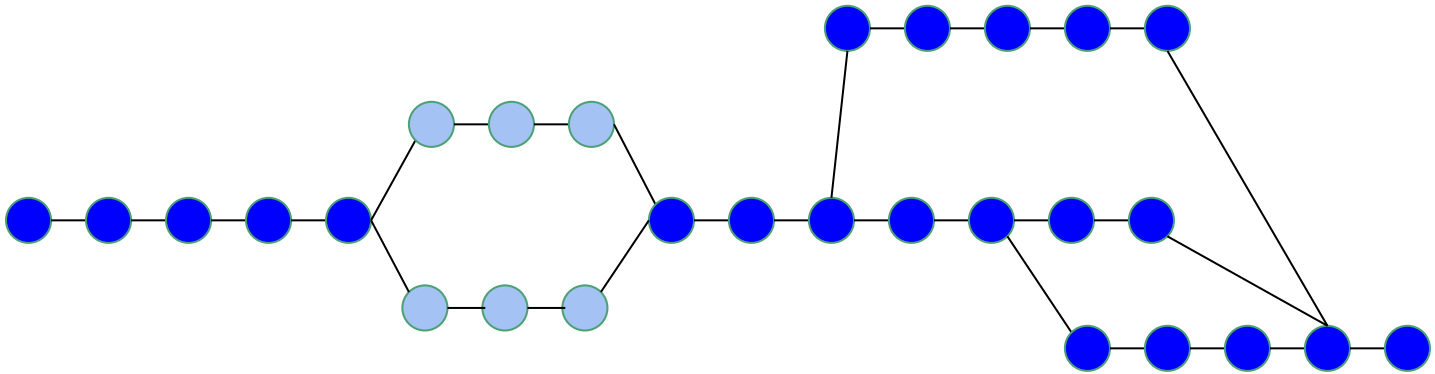
Build De Bruijn graph



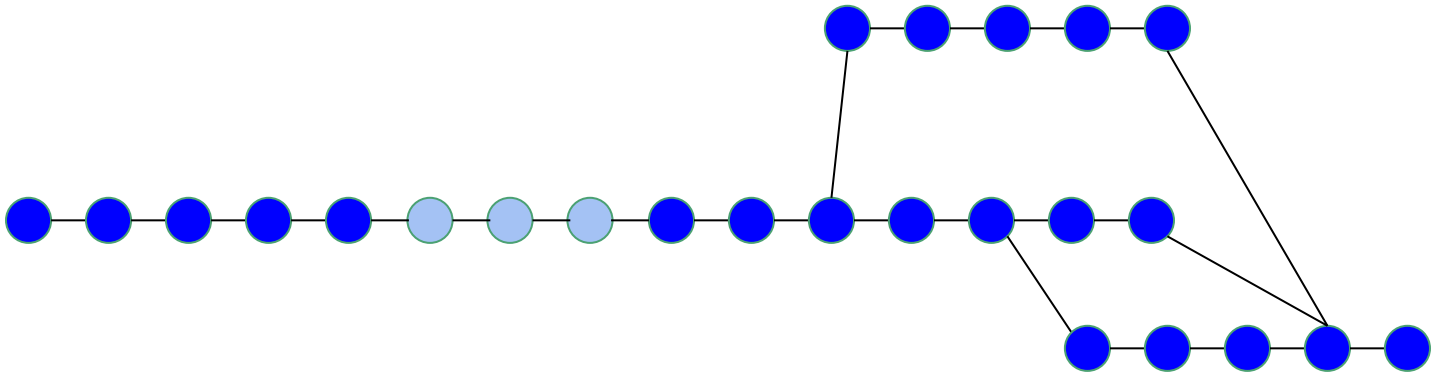
Prune error branches



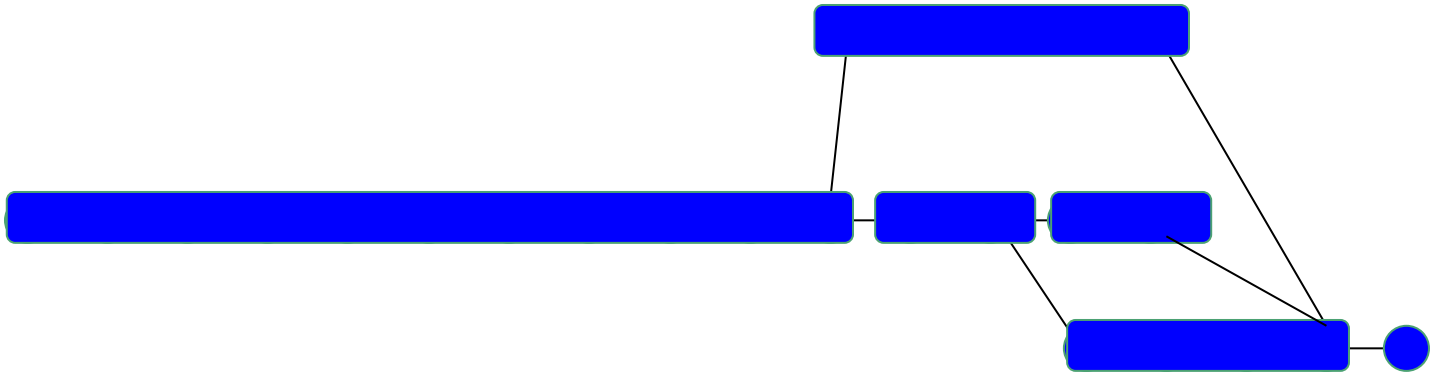
Resolve bubbles



Resolve bubbles



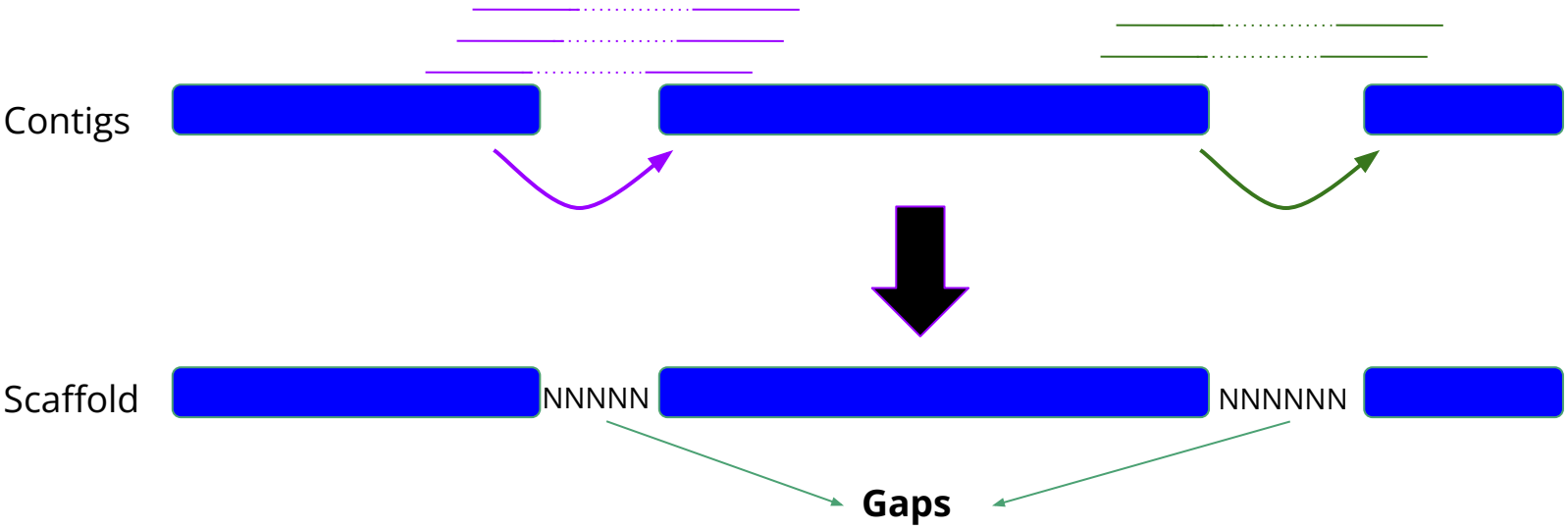
Create contigs - break on branching points



Scaffolding

Mate Pair Sequencing

- Use paired end information
- Look for evidence of two contigs linked together



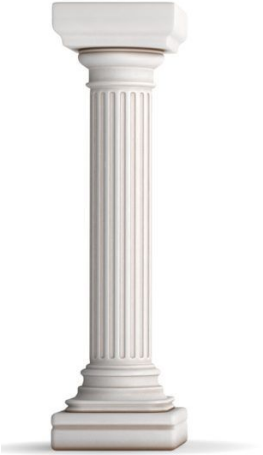
The SPAdes assembler

- A popular assembler for small-medium size genomes
- Can use multiple types of data
  - Short reads - paired/single end
  - Mate pairs
  - Long reads
- Includes several modules:
  - Error correction
  - Contigs assembly
  - Scaffolding
- Does not require k-mer choice

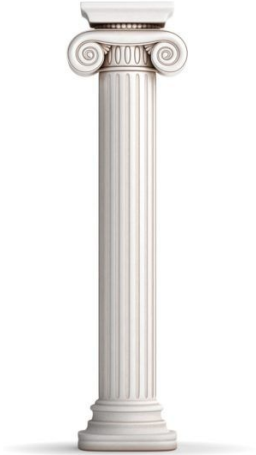
Beyond scaffolds?

- Scaffolds are the best we can do *de novo*
- We can always produce more data - won't always help
- To get to pseudomolecules we need external data
  - A similar reference genome that we can map to
  - Hi-C data
  - Long reads

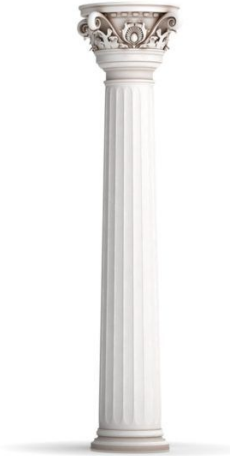
The “three pillars” of assembly quality



Completeness



Contiguity



Accuracy

A good assembly is...

- **Complete** - contains all or most of the genome
- **Contiguous** - built of large scaffolds
- **Accurate** - includes few mis-assemblies

How can we assess these for a given assembly?

Assembly statistics

- **Completeness**
  - Assembly size - compared to expected genome size
  - % of gaps in assembly (‘N’ bases)
- **Contiguity**
  - Number of contigs/scaffolds
  - Mean/median scaffold/contig size

- **Always consider all stats together**

Assembly statistics - example

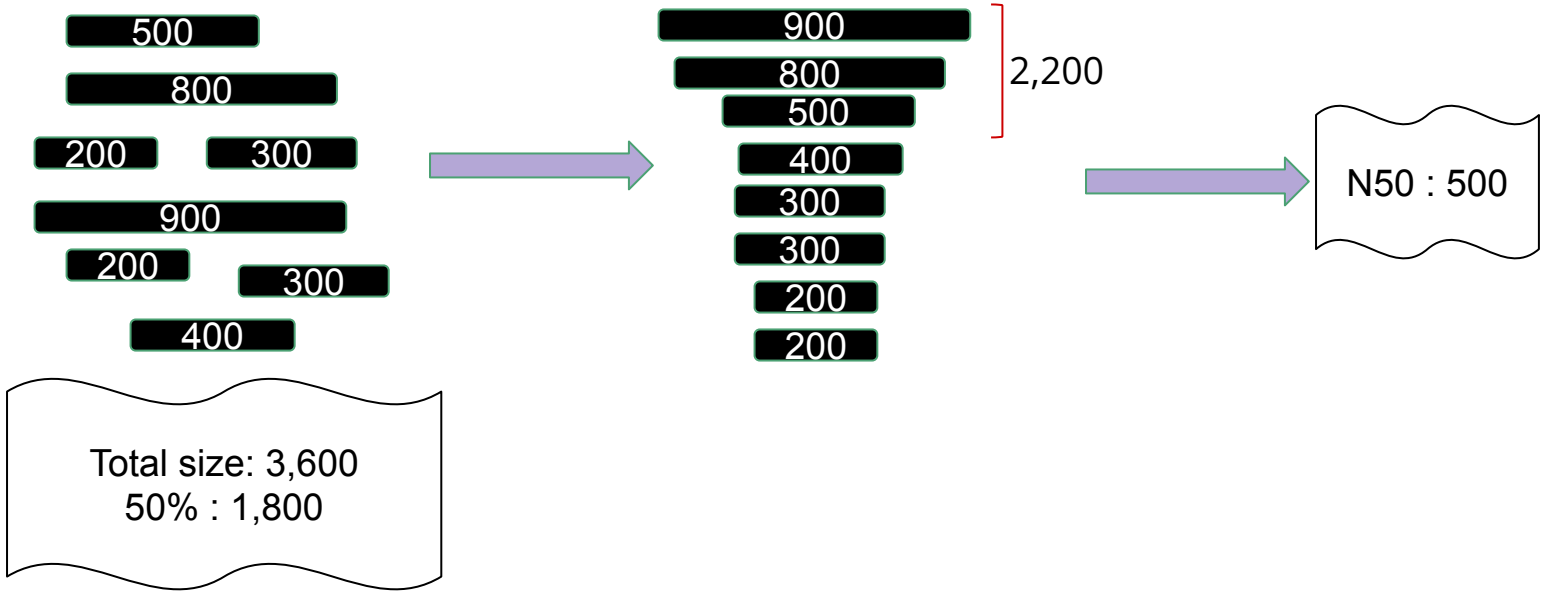
Expected genome size: 100 Mb

	Assembly A	**Assembly B**
Total size	90 Mb	80 Mb
Mean contig length	50 Kb	100 Kb
% gaps	2%	0.5%

What can you say about assemblies A and B in terms of completeness and contiguity?

Completeness - N50

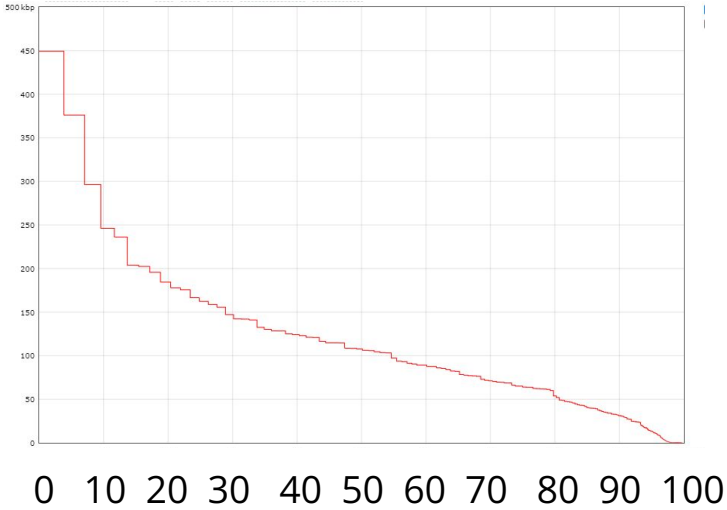
Length of the shortest contig for which the total size of all longer contigs is 50% of the assembly size.



N50 et al.

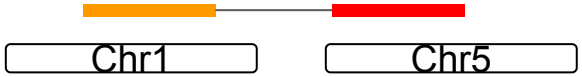
- N90 - contig length required to cover 90% of assembly size
- L50 - Number of contigs longer than N50
- NG50 - use expected genome size instead of assembly size

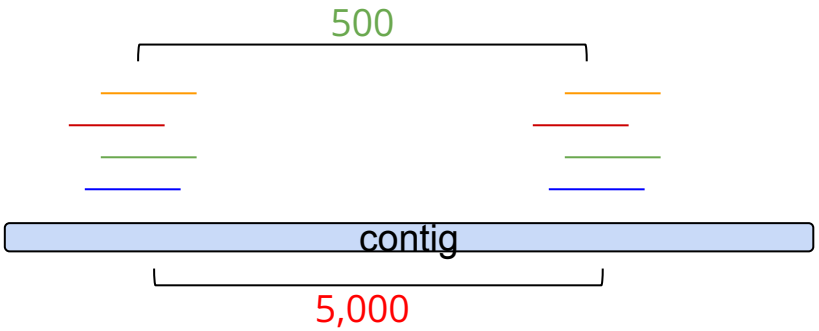
Nx plot



Accuracy - Detecting mis-assemblies

- Map reads to assembly (BWA)
- Assembled regions with low reads coverage are suspicious
- Paired read information can detect further errors
- If a reference assembly exists, map your scaffolds to detect **chimeras**





Genome assembly QA with QUAST

- Calculate assembly statistics
- Use reads and reference to perform QA
- Produce a graphical HTML report
- To run:

```
quast assembly.fasta -o out_dir
```

`-m <x>` - ignore contigs smaller than x (default: 500)

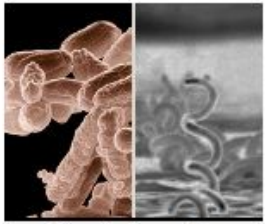
`-r <ref.fasta>` - use reference sequence for QA

`-1 <R1.fastq> -2 <R2.fastq>` - use paired end reads for QA

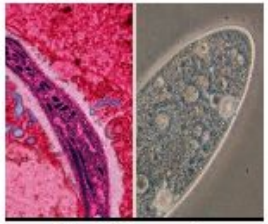
BUSCO

What assembly QA pillar does BUSCO test?

- **B**enchmarking **U**niversal **S**ingle-**C**opy **O**rthologs
- Provides a set of gene profiles expected to be found in any assembly
- Sets are available for various taxonomic groups
- Searches for BUSCOs in a given assembly and calculates % of BUSCOs detected.
- A good assembly should usually have >90% BUSCOs


Bacteria sets


Eukaryota sets


Protists sets


Metazoa sets


Fungi sets


Plants set

The BUSCO report

```
***** Results: *****

C:99.8%[S:99.6%,D:0.2%],F:0.2%,M:0.0%,n:450
449     Complete BUSCOs (C)
448     Complete and single-copy BUSCOs (S)
1       Complete and duplicated BUSCOs (D)
1       Fragmented BUSCOs (F)
0       Missing BUSCOs (M)
450     Total BUSCO groups searched
```

Running BUSCO

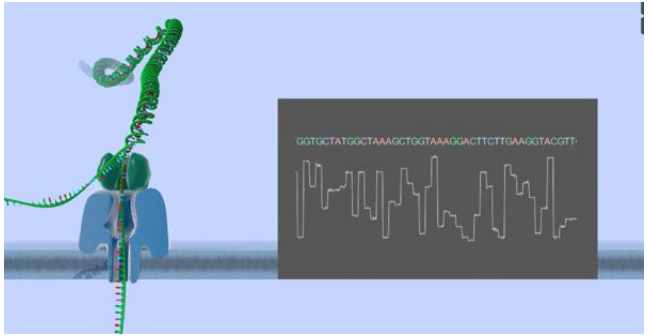
- Inputs:
  - Assembly (fasta)
  - BUSCOs set (can be downloaded from the official website)
- Output:
  - Short_summary - how many BUSCOs were found (txt file)
  - Full_summary - status of each BUSCO in the selected set (txt file)
- Running:

```
run_busco --in assembly.fasta --out <name> --mode genome
--lineage_path /path/to/BUSCOs_set/
```

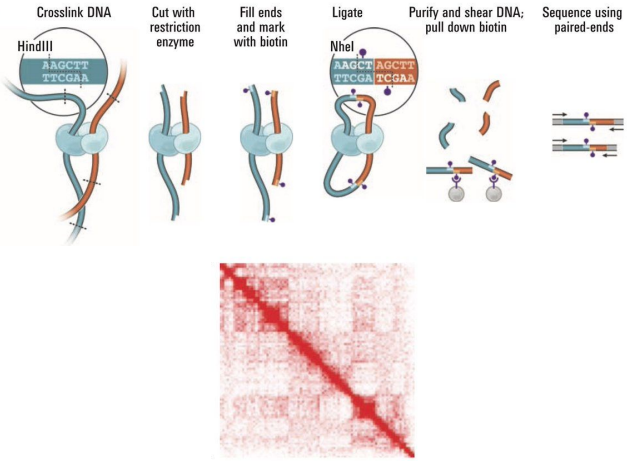
What factors affect assembly quality?

- Input data
  - Sequencing depth
  - SE/PE
  - Read length
  - Insert size
  - Sequencing quality and preprocessing
- Choice of assembler
- Choice of parameters
  - Size of *k*
  - K-mer depth cutoff for error filtration
- Genome complexity
  - Fraction of repeats
  - Low complexity regions

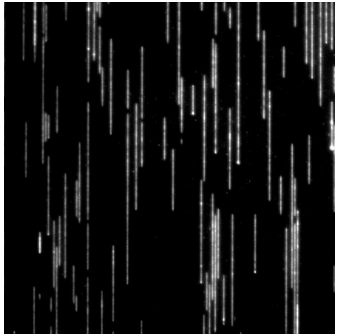
Emerging technologies for genome assemblies



Long reads



Hi-C



Optical mapping