

DNA Sequencing and Data Analysis

Prof Noam Shomron
Hadas Volkov

Lecture 6, December 9, 2022

DNA Sequencing and Data Analysis

Friday 8:45 AM to 11:15 AM
Arazi-Ofer Building, C.L03

nshomron@gmail.com

hadas.volkov@post.runi.ac.il

DNA Sequencing and Data Analysis

The De-novo Shotgun Assembly Problem

Class	Title	Content/assignments	Activity, location
1, 4.11	Introduction to Cells and DNA	Basic knowledge of biology	In the lecture hall, Noam
2, 11.11	DNA Sequencing past and present	Basic knowledge of molecular DNA	In the lecture hall, Noam
3, 18.11	Genomics technologies	DNA, RNA, technologies	In the lecture hall, Noam
4, 25.11	Introduction to Bioinformatics challenges in reading DNA	Focus on three methods: WES/WGS, RNA-seq, cell-free DNA	In the lecture hall, Noam
5, 2.12	Modern DNA Sequencing, 2nd wave File Formats, tools.	Analysis approaches for WES/WGS, RNA-seq, cell-free DNA	In the lecture hall, Hadas and Noam
6, 9.12	De novo Shotgun Assembly	The algorithms and methods behind the assembly problem	In computer class, Hadas and Noam
7, 16.12	Sequence Mapping and Alignment	The algorithms behind mapping and alignment, fast and heuristics	In computer class, Hadas and Noam
8, 23.12	Variant Calling and Somatic Variant Analysis	The bioinformatics behind discovery of novel mutations in cancer	In computer class, Hadas and Noam
9, 30.12	Nanopore data analysis introduction and class activity	The bioinformatics behind Nanopore analysis, activity on computers	In computer class, Hadas and Noam
10, 6.1	Practice molecular biology techniques	Pipetting, transferring small amounts of fluids, running a dry Nanopore experiment	In biology class, Meitar and Noam
11, 13.1	Nanopore DNA sequencing	Nanopore DNA sequencing, experimental run	In biology class, Meitar, Hadas, Assaf
12, 20.1	Nanopore data analysis	Nanopore DNA analysis, experimental run	In computer class, Hadas and Noam
13, 27.1	Nanopore data analysis and presentations	Groups present their results	In the lecture hall, Hadas and Noam

Basic Command Line & Conda

Live Session AWS EC2 ubuntu@52.198.154.52



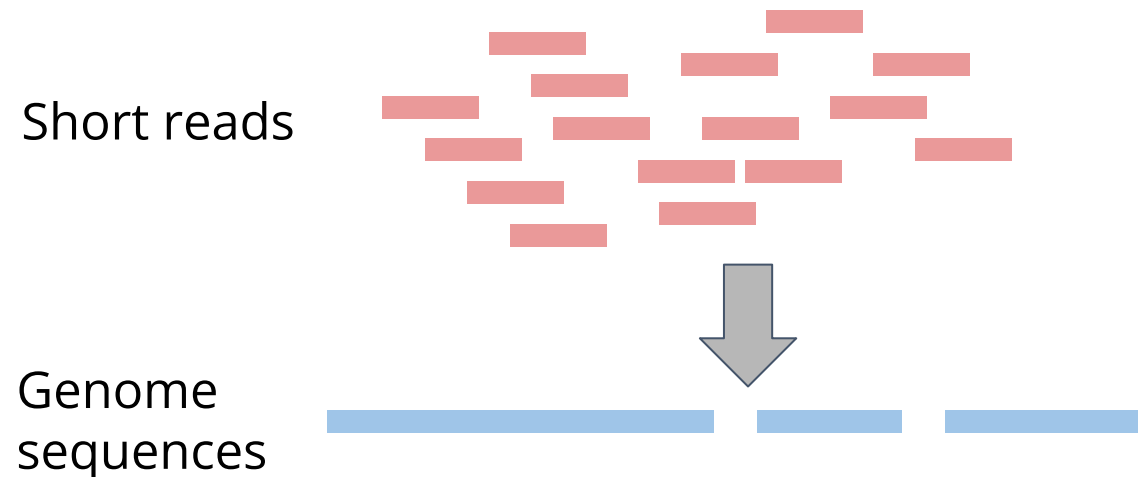
A word cloud of various Linux command-line utilities. The words are arranged in a dense, overlapping cluster. The colors of the words include shades of orange, yellow, green, blue, and purple. The words are of varying sizes, with some being larger than others. The words include: alias, service, sort, whoami, uname, cp, zip, dules, cat, pwd, sudo, vim, ln, rpm, rm, mkdir, tail, curl, git, touch, zcat, gzip, grep, sed, ssh, apt, chown, mv, head, htop, nano, diff, clear, awk, wget, cd, man, unzip, ps, chmod, df, killall, dnfs.

What is De Novo Genome Assembly?

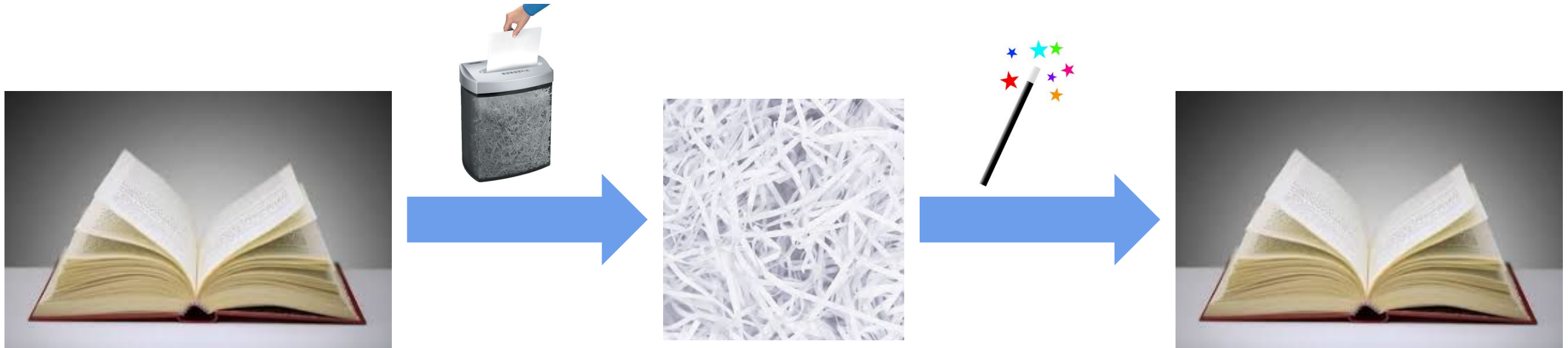
Genome assembly - constructing long genomic sequences from shorter ones

De novo = “from scratch”

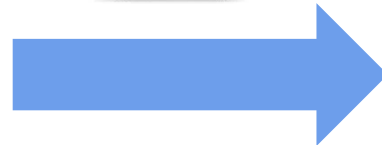
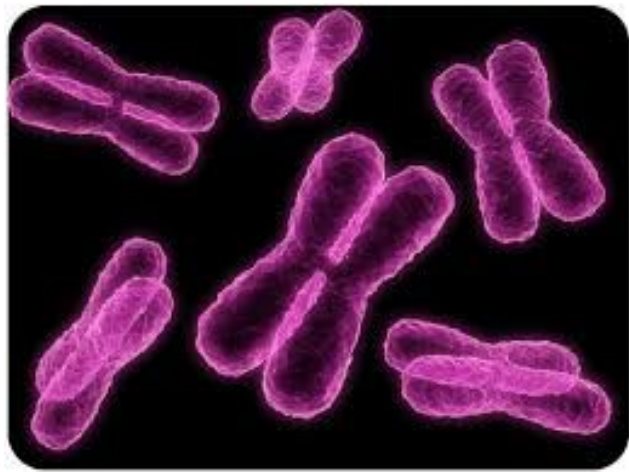
In NGS context - short reads → whole genome, without any external reference



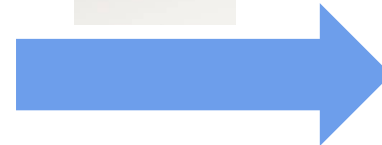
The Assembly Problem



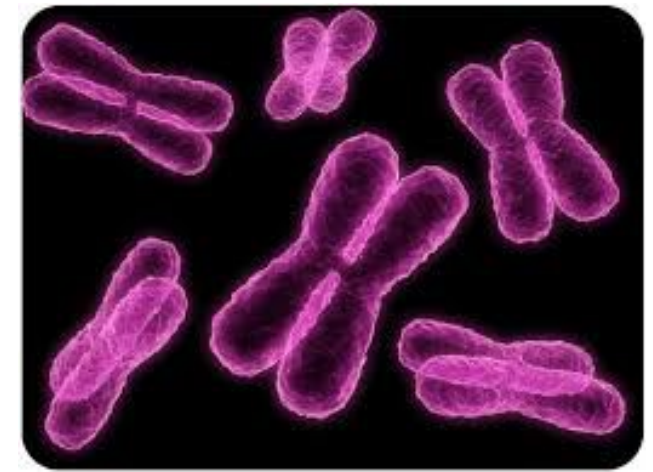
The Assembly Problem



Shotgun
sequencing

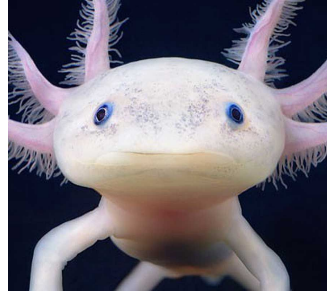


De novo
assembly



Why Do We Need De Novo Assembly?

Completely new organism



Existing reference is too different from what we are interested in



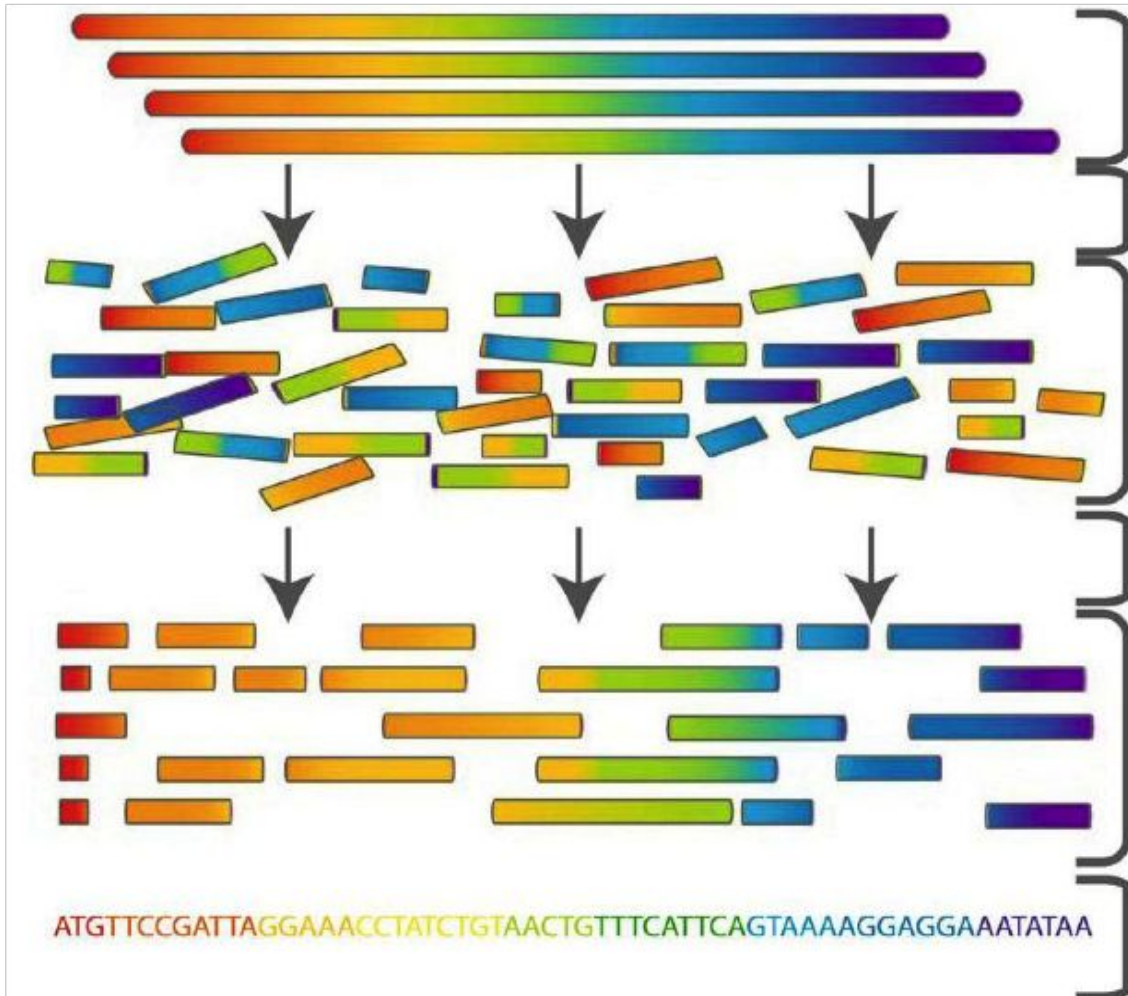
Identify large structural variation

Detect novel sequences not present in the reference

Cancer genomics



Genome Assembly by Reads Overlap



Genomic DNA

Fragmentation + Sequencing

Sequence reads

Assembly

Connection between reads
found

Consensus sequence

Coverage Definition

Diagram illustrating sequence alignment and coverage. The sequences are aligned as follows:

```
          CTAGGCCCTCAATTTT
        CTCTAGGCCCTCATTTTT
      GGCTCTAGGCCCTCATTTTT
    CTCGGCTCTAGGCCCTCATTT
  TATCTCGACTCTAGGCCCTCA
  TATCTCGACTCTAGGCC
TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCTATATCT
GGCGTCTATATCT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTT
```

Two vertical blue boxes highlight the region from position 10 to 14 of the bottom sequence, which is also highlighted in red. This region corresponds to the sequence `CGGCTC` in the red sequence.

Coverage = 5

Average Coverage

CTAGGCCCTCAATTTT
CTCTAGGCCCTCATTTTT
GGCTCTAGGCCCTCATTTTT
CTCGGCTCTAGGCCCTCATTT
TATCTCGACTCTAGGCCCTCA
TATCTCGACTCTAGGCC
TCTATATCTCGGCTCTAGG 177 bases
GGCGTCTATATCTCG
GGCGTCTATATCT
GGCGTCTATATCT 35 bases
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT

Average Coverage = $177/35 \approx 5$ -fold (5x)

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

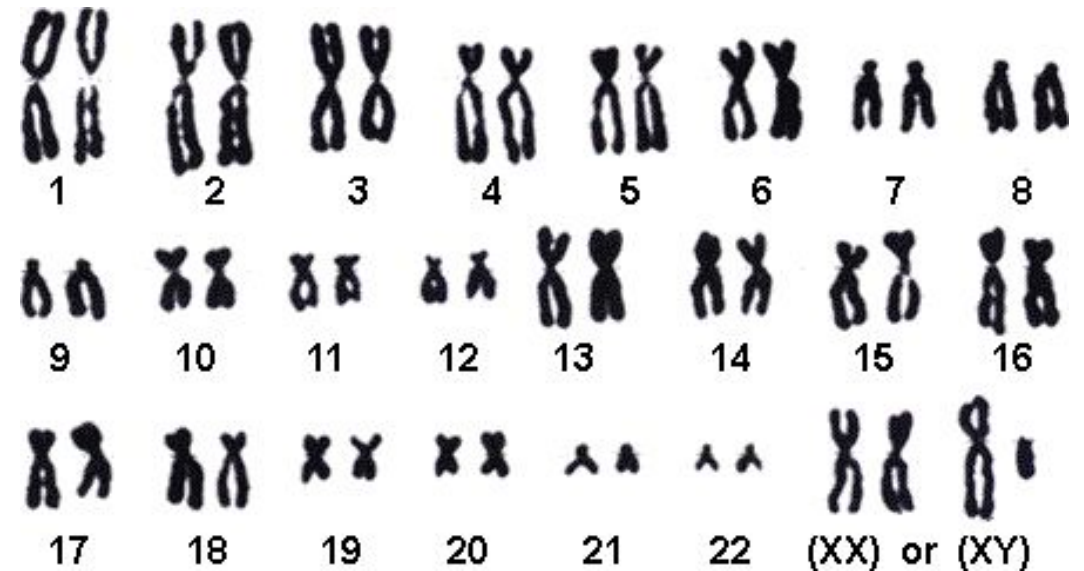
TCTATATCTCGGCTCTAGG

TATCTCGACTCTAGGCC



Why the differences?

1. Sequencing errors
2. Polyploidy



High and Low Coverage

CTAGGCCCTCAATTTT
GGCTCTAGGCCCTCATTTTT
CTCGGCTCTAGGCCCTCATTT
TATCTCGACTCTAGGCC
TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCTATATCT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
CTAGGCCCTCAATTTT
TATCTCGACTCTAGGCCCTCA
GGCGTCTATATCT

More coverage

Less coverage

More coverage leads to more and longer overlaps

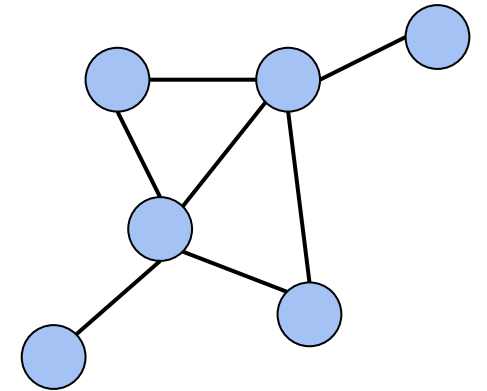
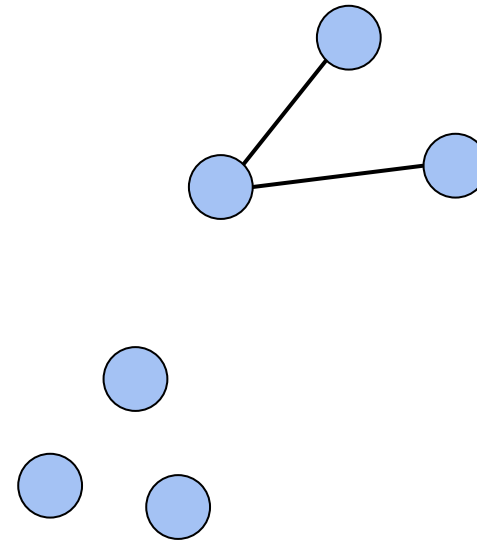
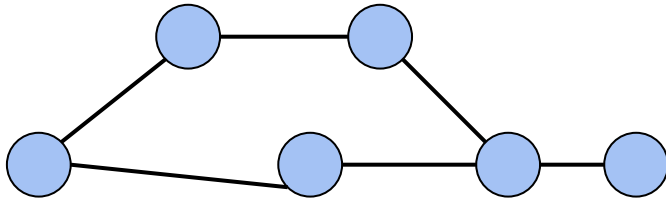
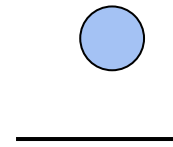
Overlap Consensus

What is the best representation to the set of sequences?

Graph

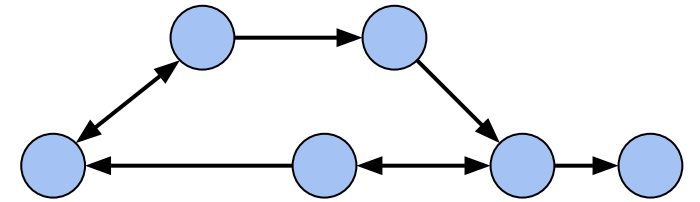
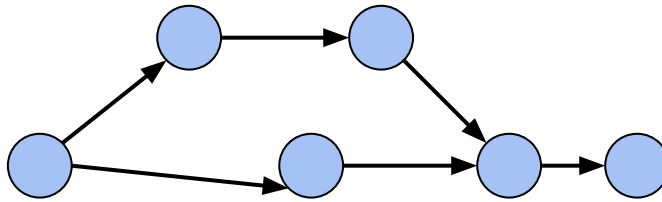
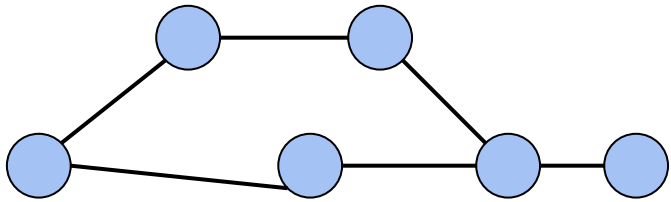
A graph is a set of:

- Nodes (vertices)
- Edges - connecting two nodes

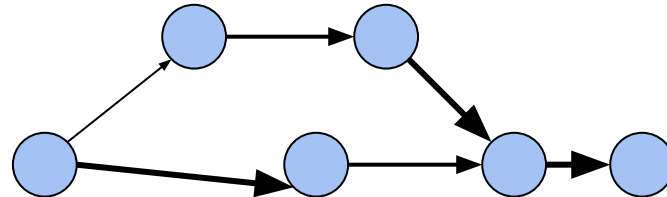
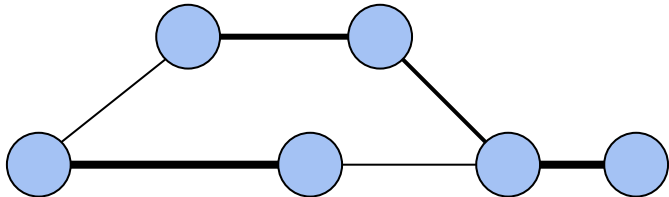


Directed and Weighted Graphs

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

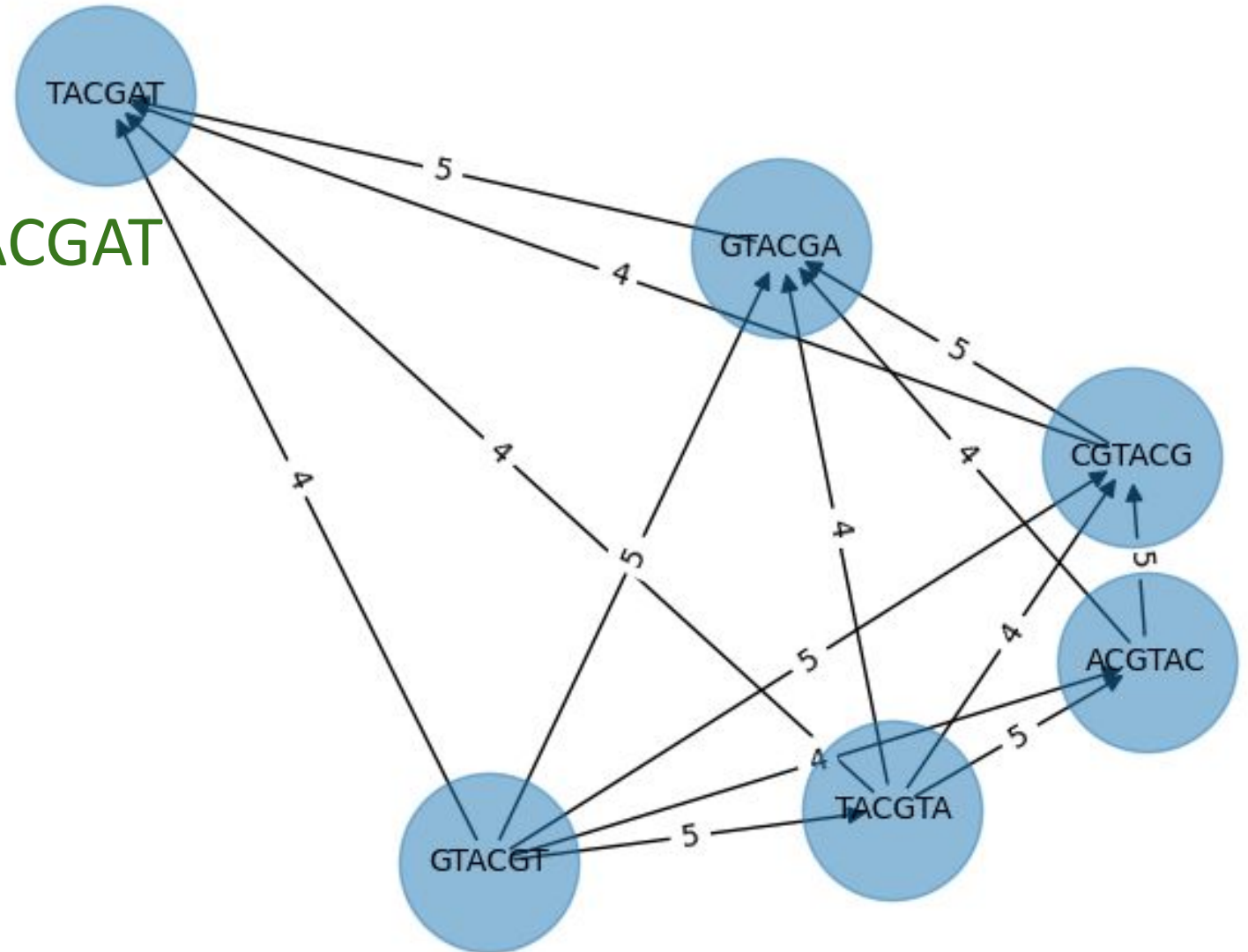
TACGTA

ACGTAC

CGTACG

GTACGA

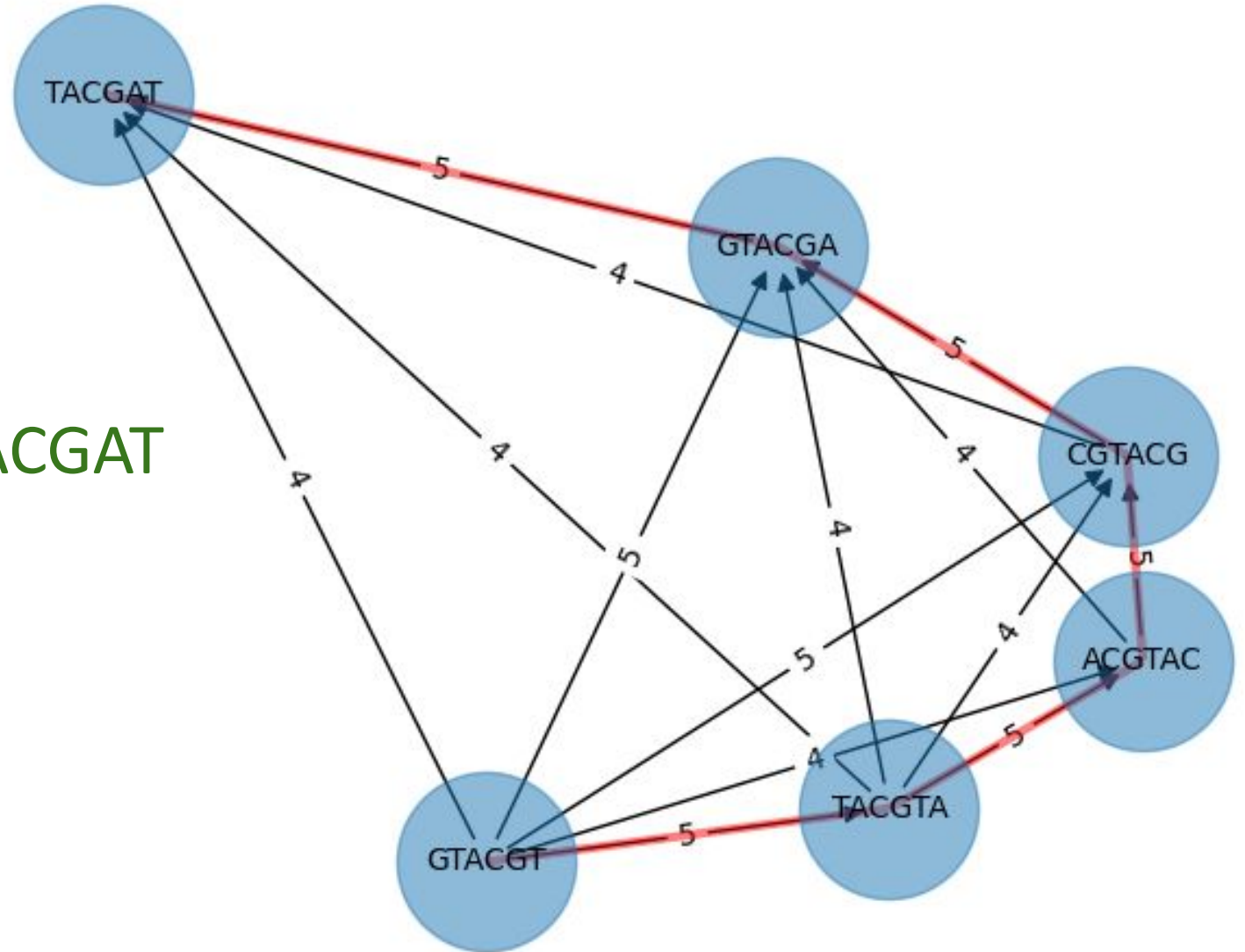
TACGAT



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: BAAABBBAABAABBBBBAABAB

AAA

AAB

ABB

BBB

BBA

BAB

ABA

BAA

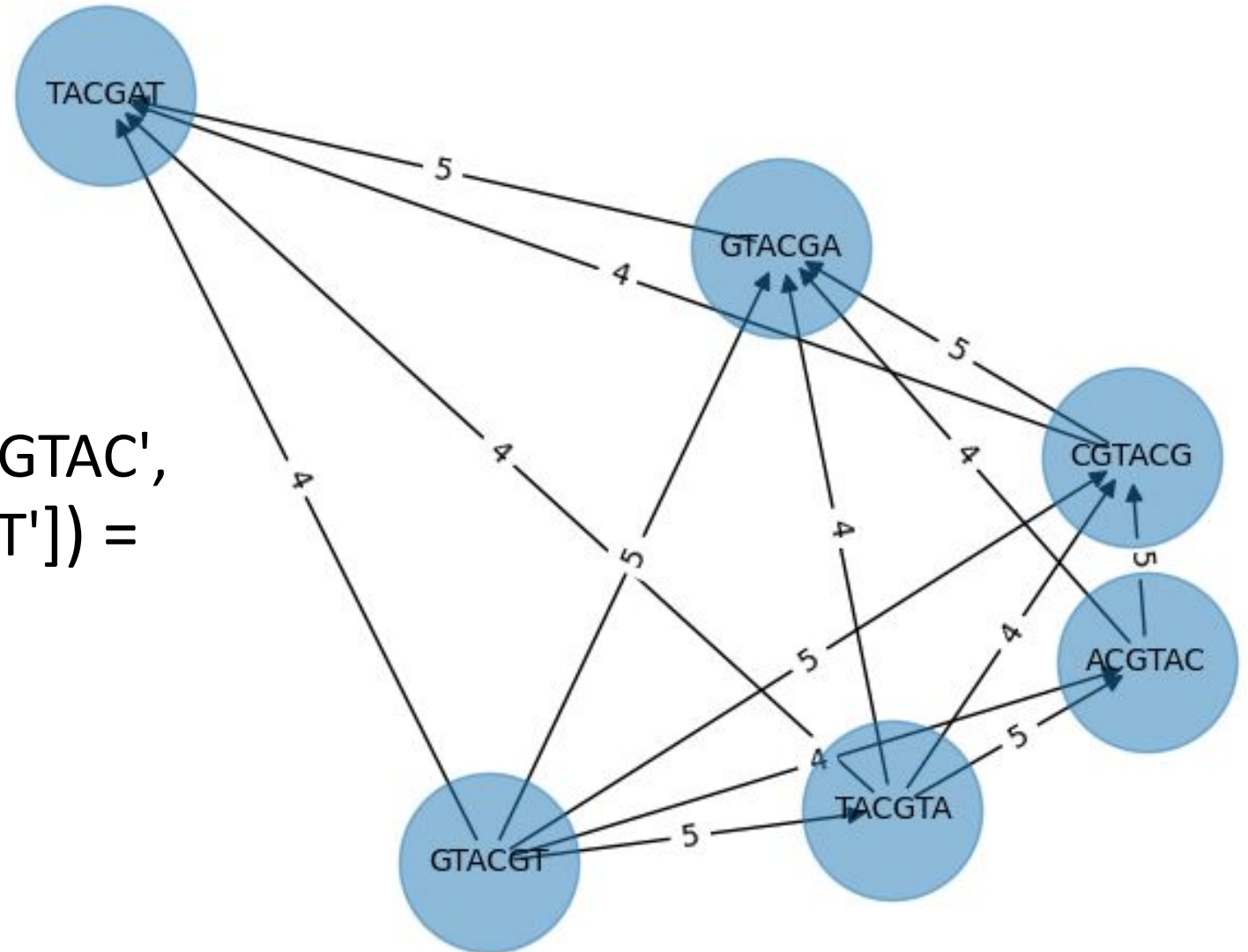
24

SCS: AAABBBABAA

10

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

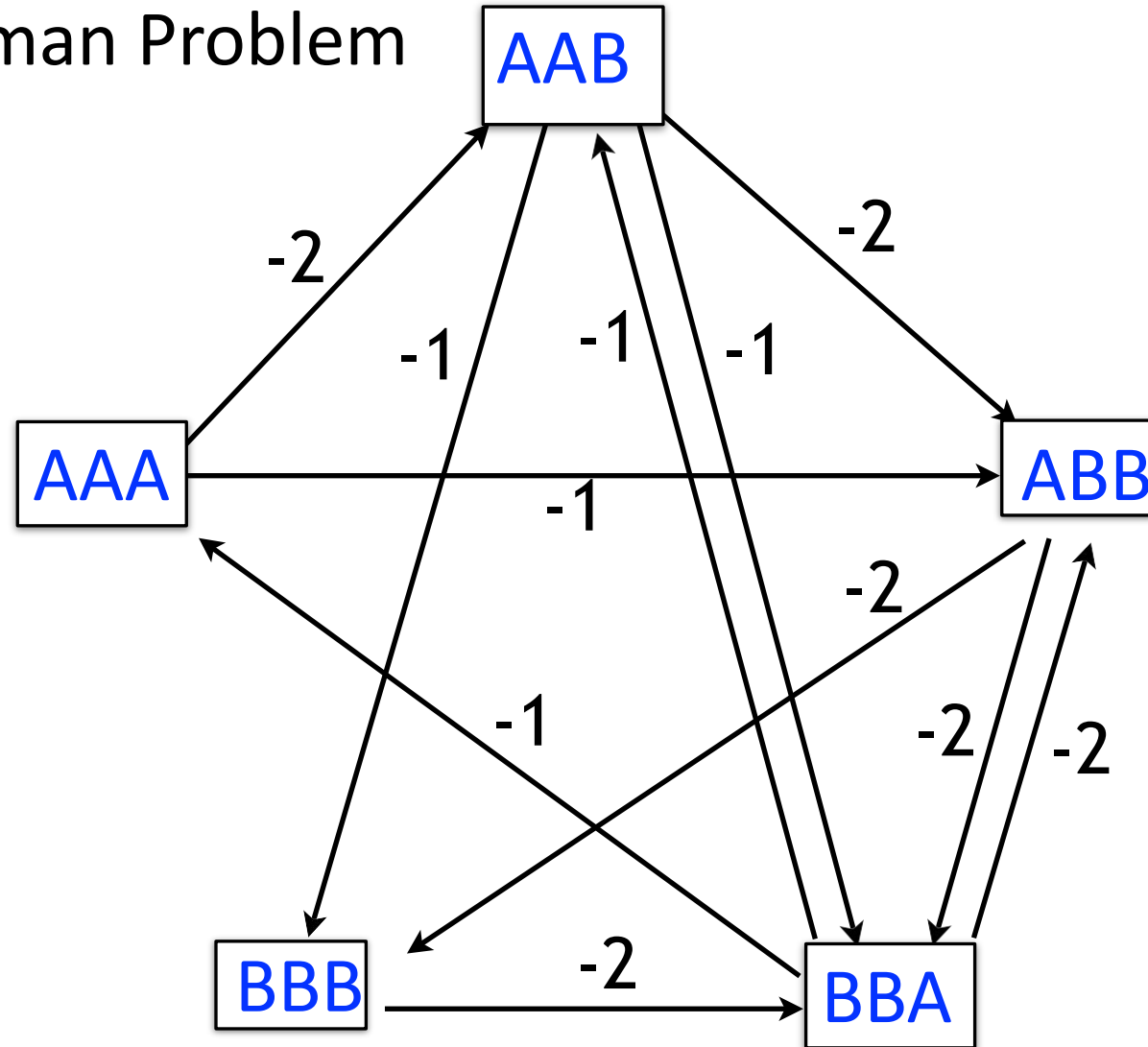
Order 2: AAA AAB ABA BAB ABB BBB BAA BBA
AAABABBBBAABBA Superstring 2

$O(n!)$

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

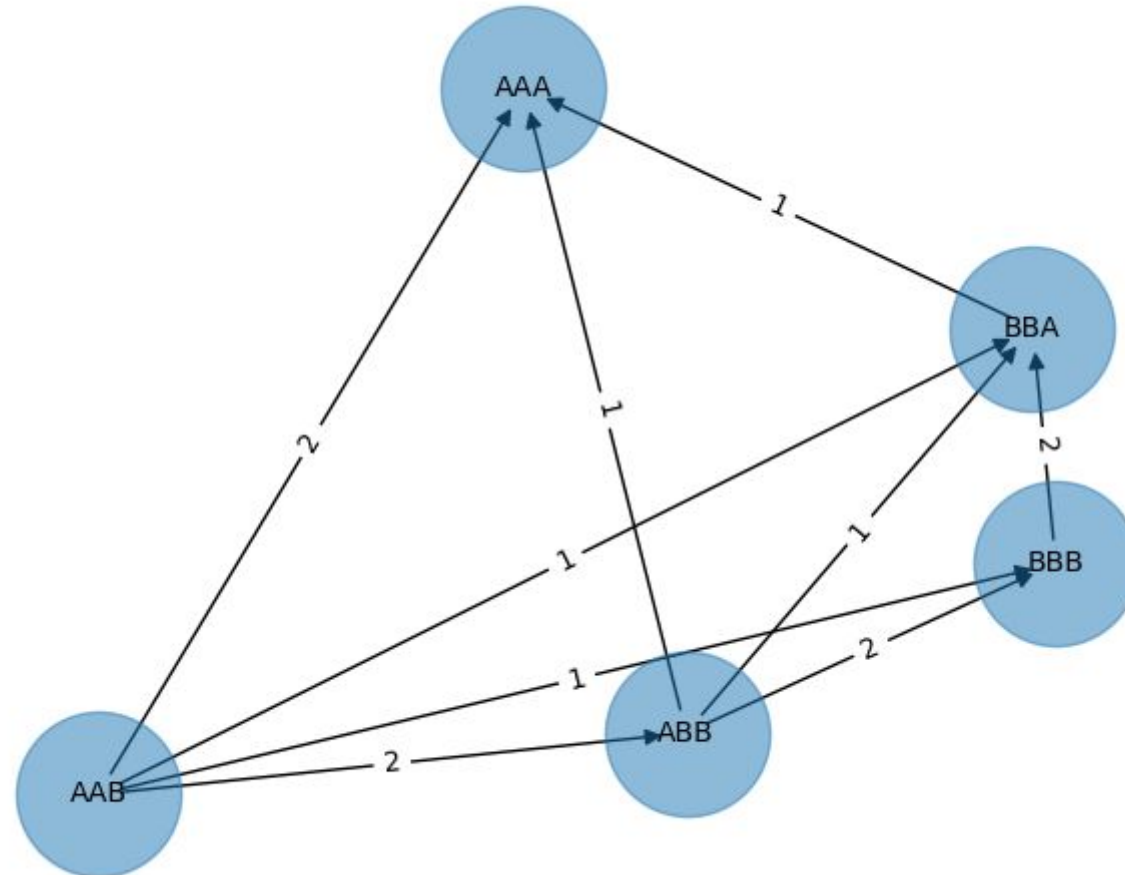
NP-complete

No efficient solution algorithm has been found

Shortest Common Superstring

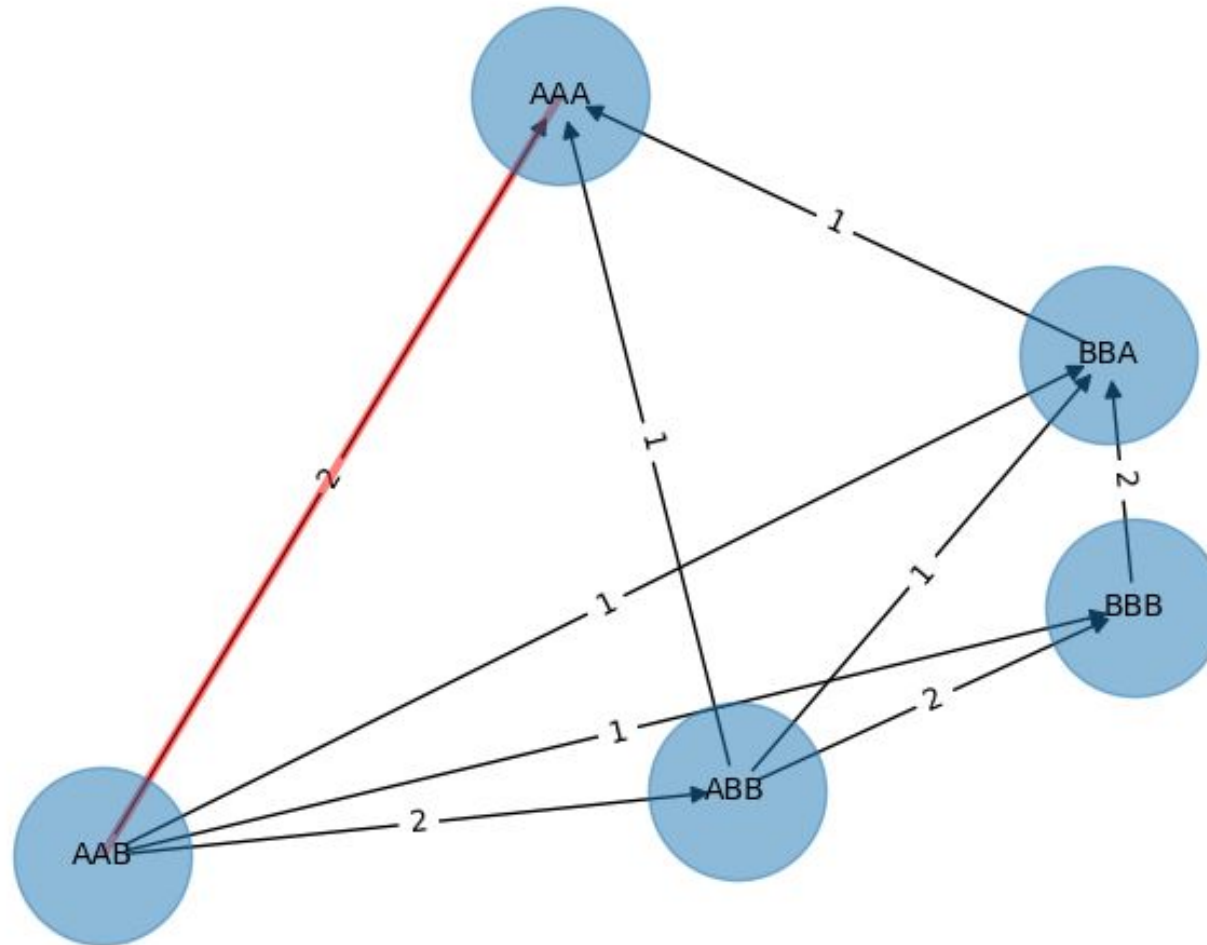
Greedy

Example: BAA AAB BBA ABA ABB BBB AAA BAB



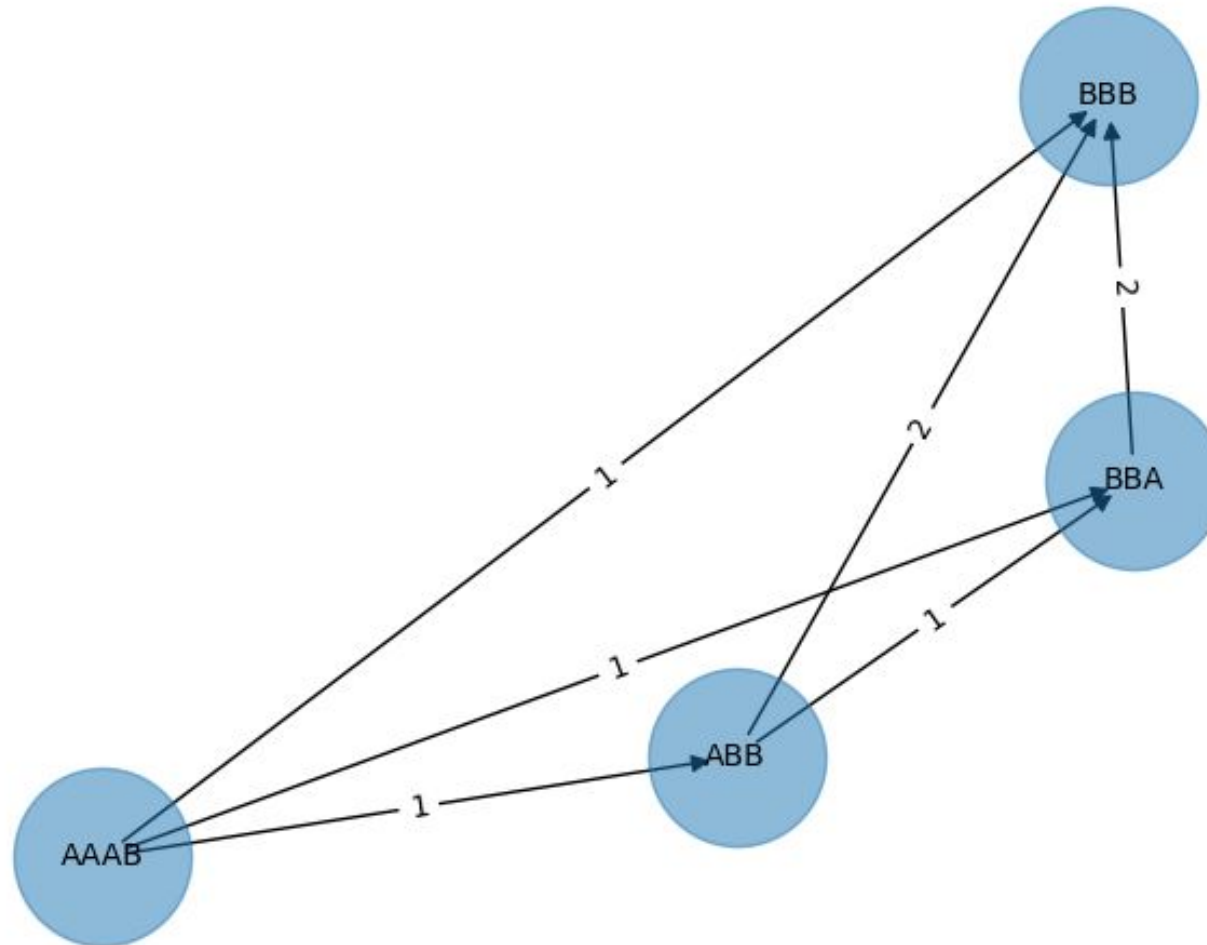
Shortest Common Superstring

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 **Missing 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

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

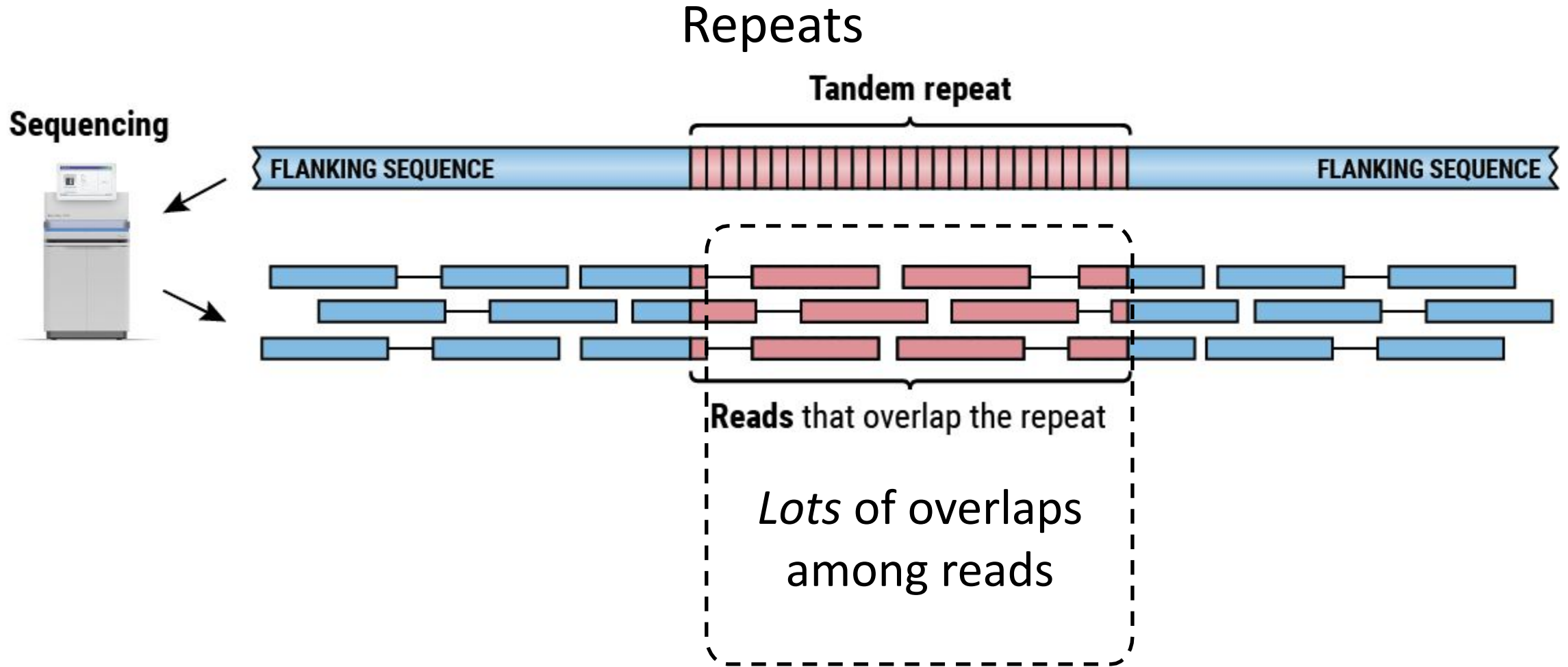
a_long_long_long_time

collapse

a_long_long_time

The human genome is ~ 50% repetitive!

Shortest Common Superstring



Genome Assembly by Reads Overlap - Challenges

Do we believe short overlaps?

How do we handle sequencing errors?

Computationally ineffective for large data sets

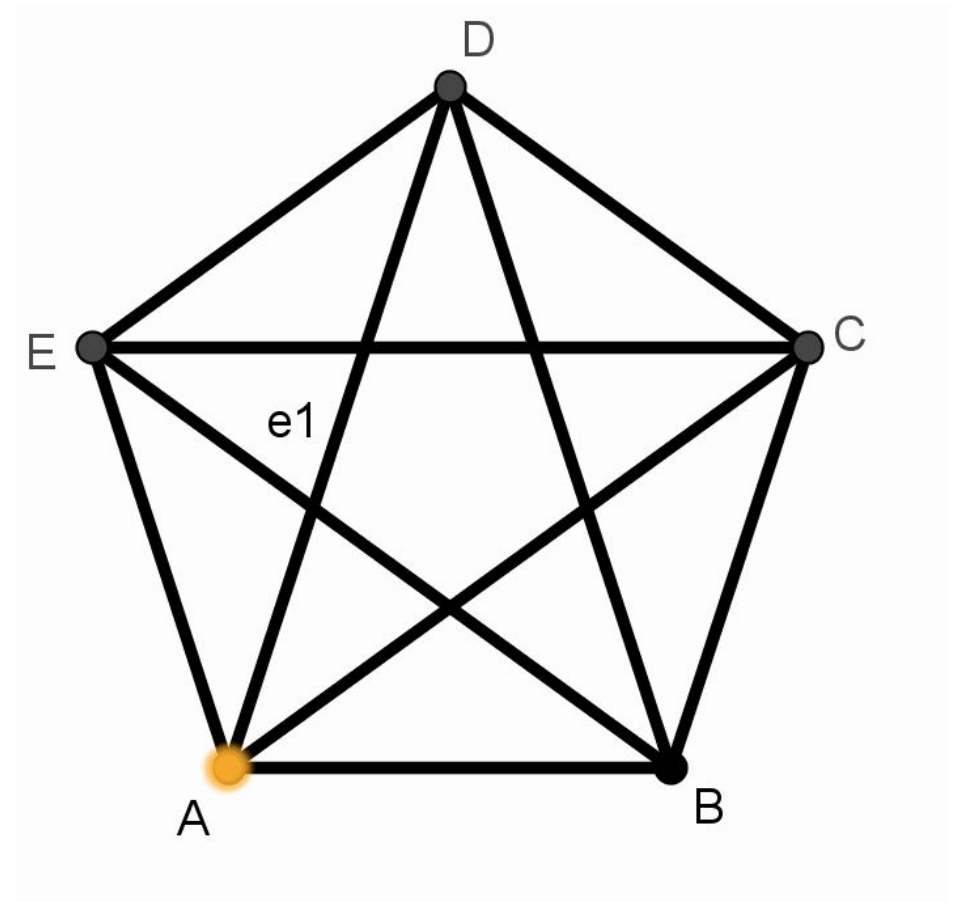
Bottom line: not good enough for large and complex genomes (e.g. human)

We need something smarter!

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



K-mers

AGATCCAGCGAGGTCGCTATCCGTTAATTG

5-mers

AGATC

GATCC

ATCCA

...

AATTG

K-mers

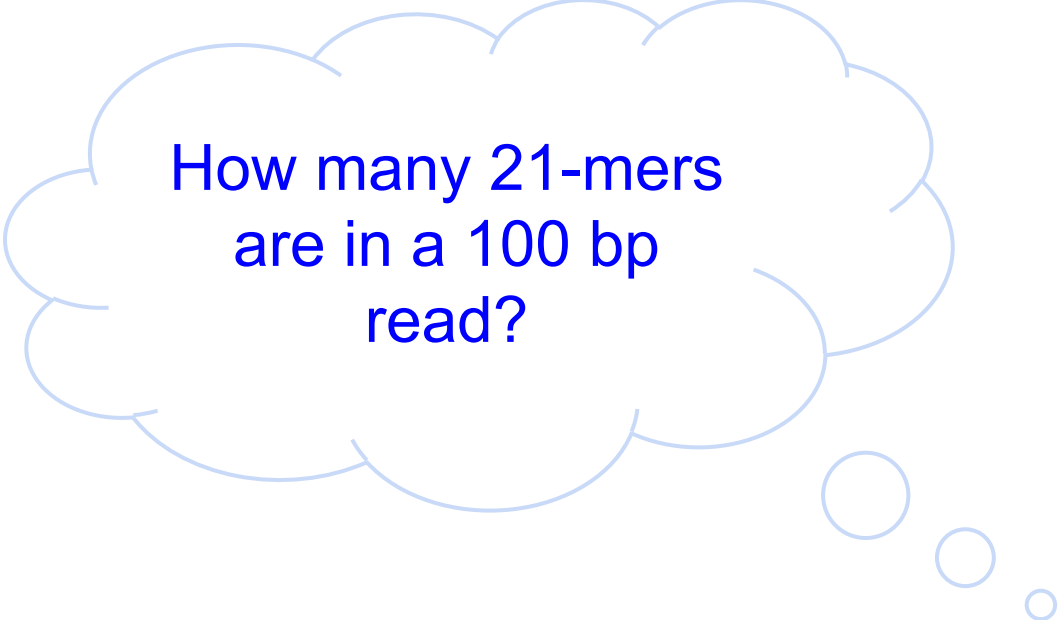
AGATCCAGCGAGGTCGCTATCCGTTAATTG

5-mers

AGATC
GATCC
ATCCA
...
AATTG

7-mers

AGATCCA
GATCCAG
ATCCAGC
...
TTAATTG



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

De Bruijn Graph

Genome (G=30): AGATCCAGCGAGGTCGCTATCCGTTAATTG

Reads (L=10): AGATCCAGCG

AGCGAGGTCG

GCTATCCGTT

CCGTTAATTG

Break into k-mers (k=4):

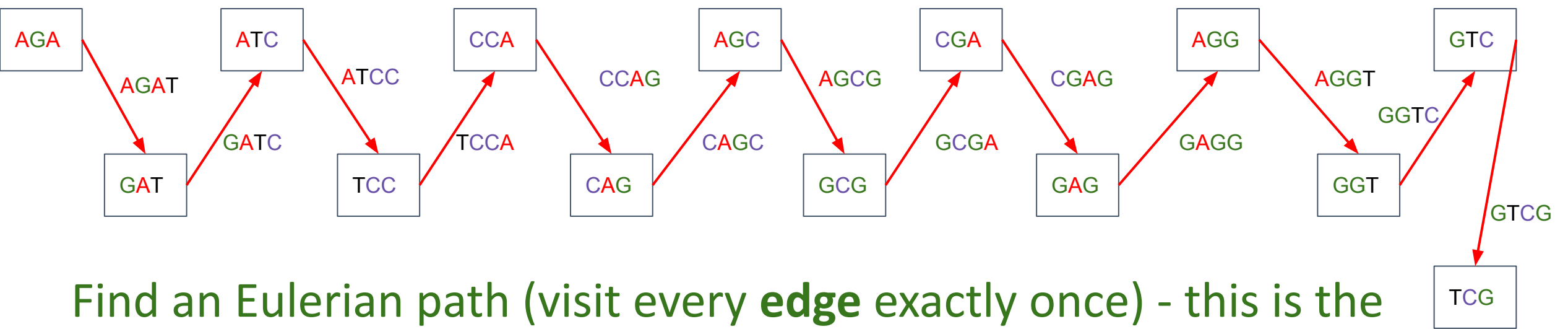
AGATCCAGCG → AGAT GATC ATCC TCCA CCAG CAGC AGCG
AGCGAGGTCG → AGCG GCGA CGAG GAGG AGGT GGTC GTCG

...

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!

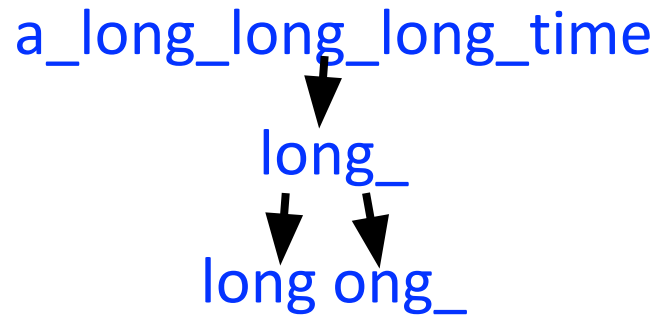
De Bruijn Graph

A procedure for making a De Bruijn graph for a genome

Assume perfect sequencing where each length-k substring is sequenced exactly once with no errors

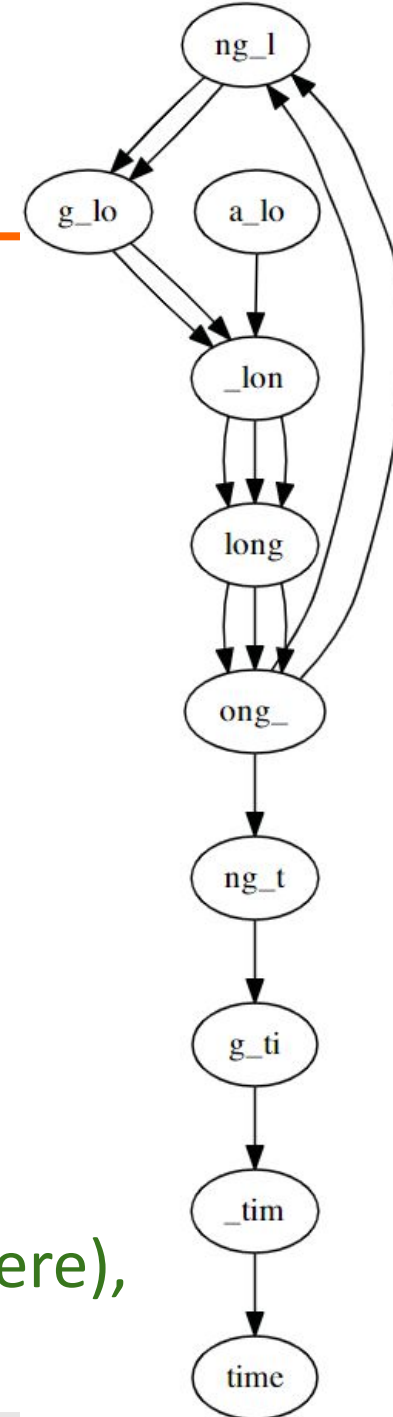
Pick a substring length k: 5

Start with each read:



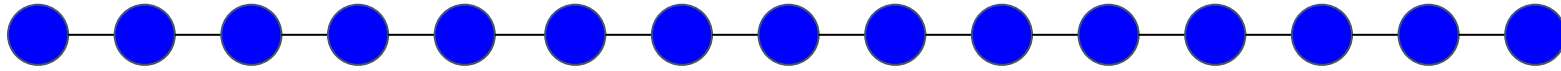
Take each k mer and split into left and right k-1 mers

Add k-1 mers as nodes to De Bruijn graph (if not already there), add edge from left k-1 mer to right k-1 mer

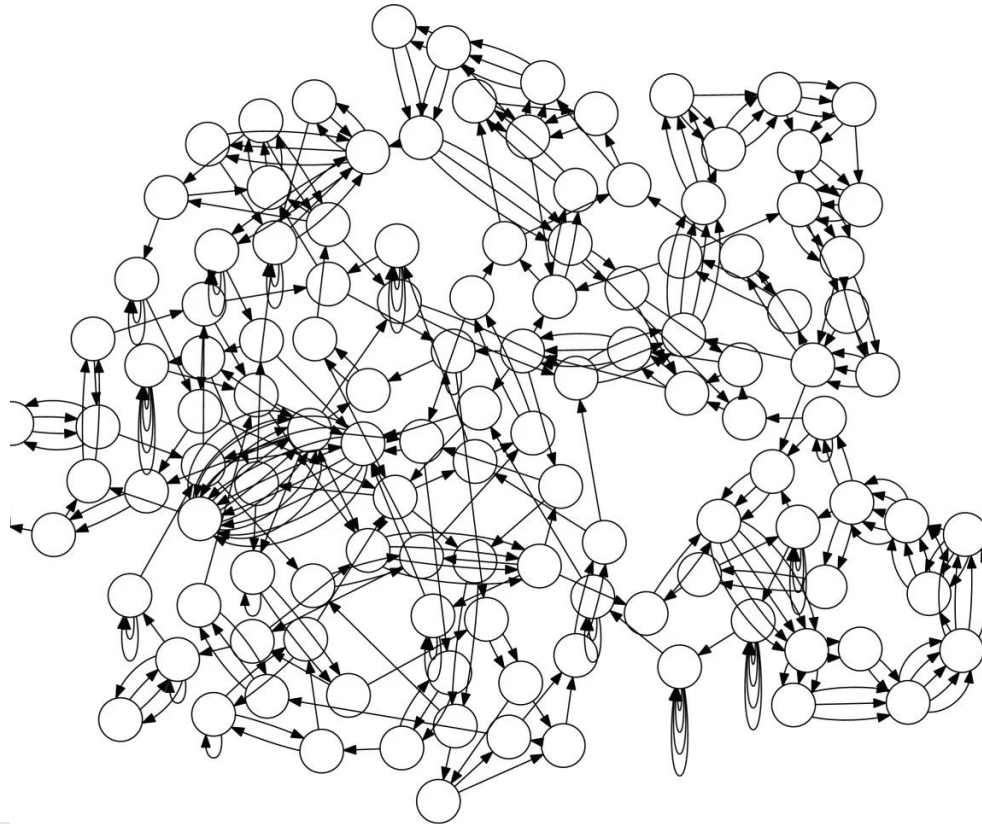


De Bruijn Graph

Ideally, we want our graph to look like this:



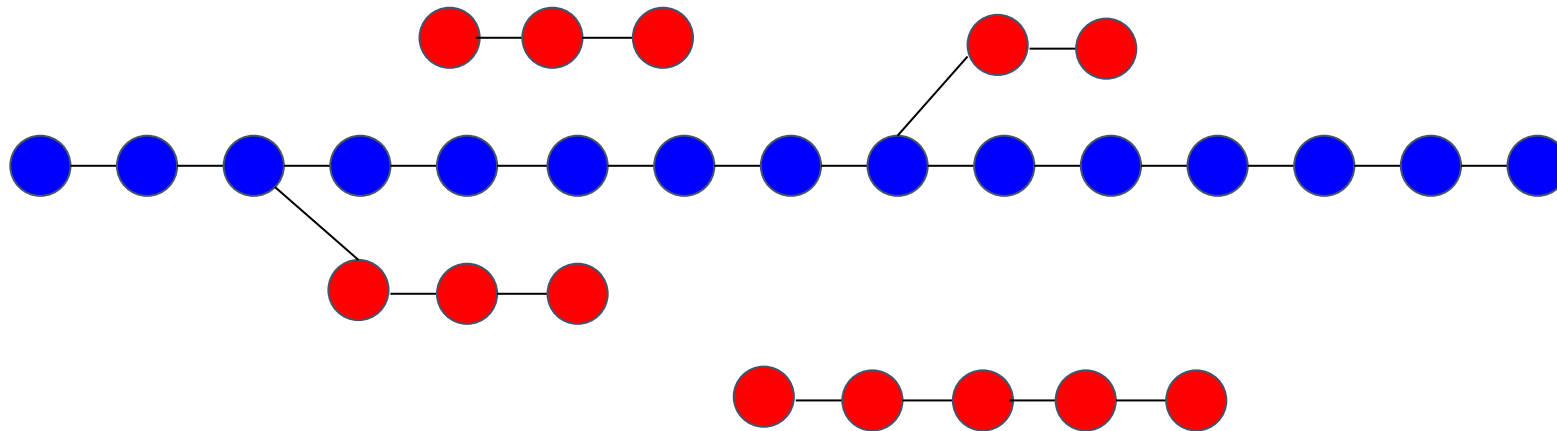
But in practice:



De Bruijn Graph

Sequencing errors

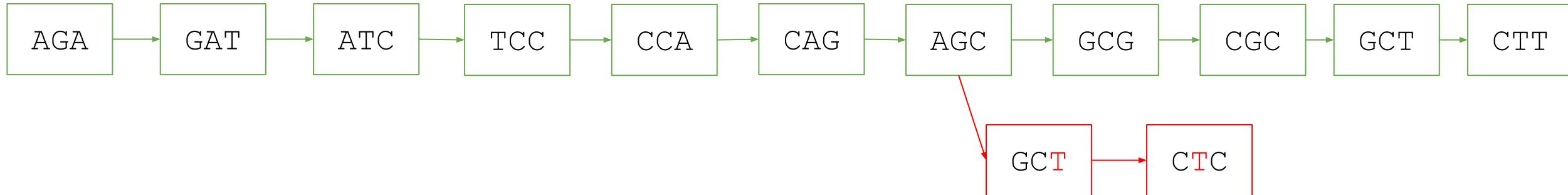
- Side branches
- Disconnected bits



De Bruijn Graph

Sequencing errors

AGATCCAGCG → AGAT GATC ATCC TCCA CCAG CAGC AGCG
GATCCAGC**T**C → GATC ATCC TCCA CCAG CAGC AGC**T** GCT**C**
TCCAGCGCTT → TCCA CCAG CAGC AGCG GCGC CGCT GCTT



De Bruijn Graph

Genomic Repeats

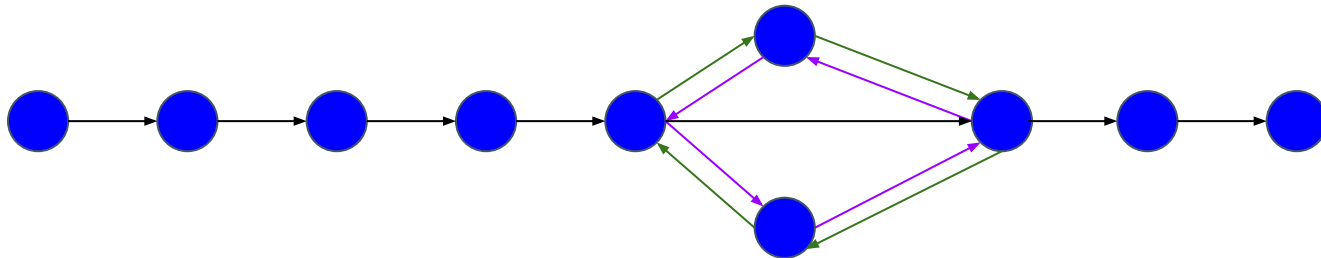
- Tandem duplication



- Interspersed duplications



- Might create ambiguity - multiple possible eulerian paths

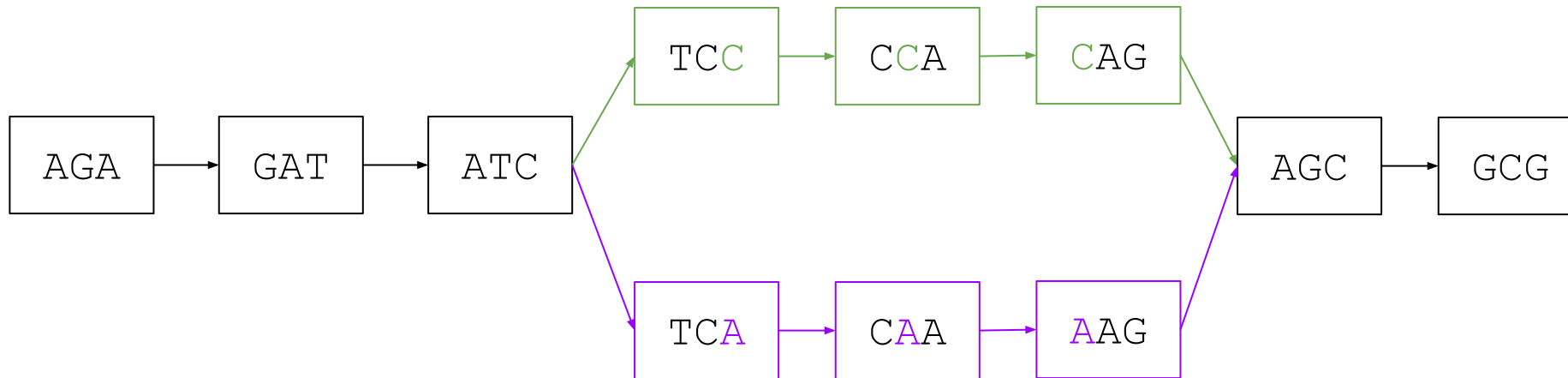


De Bruijn Graph

Heterozygosity

- Creates “bubbles” in the graph

Maternal AGATC**C**AGCG → AGAT GATC ATC**C** TC**C**A CCAG **C**AGC AGCG
Paternal AGATC**A**AGCG → AGAT GATC ATC**A** TC**A**A CA**A**AG **A**AGC AGCG



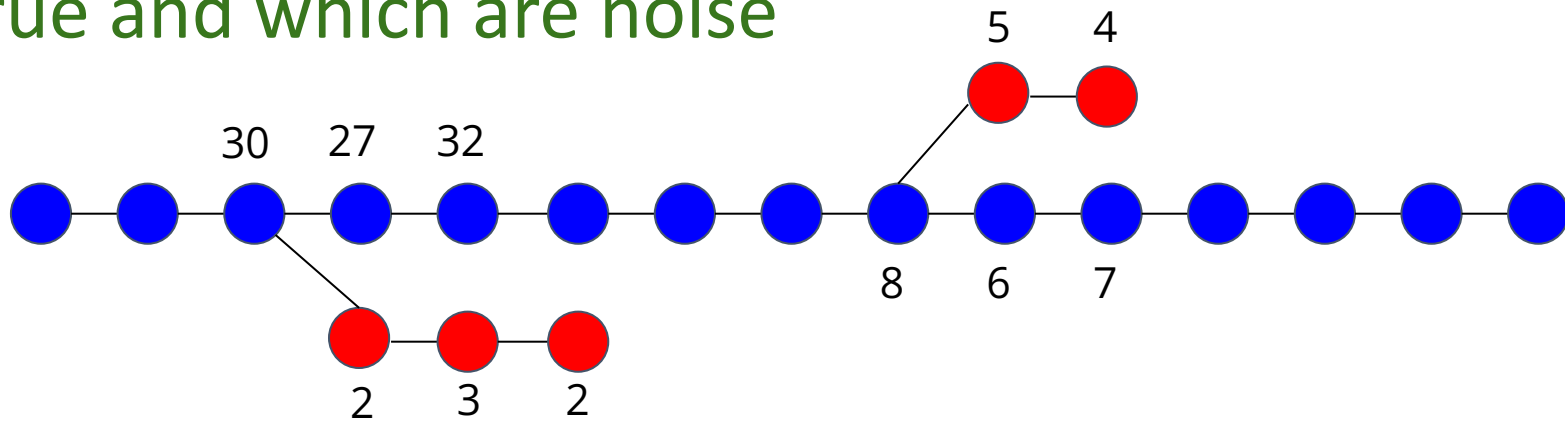
De Bruijn Graph

Uneven sequencing depth

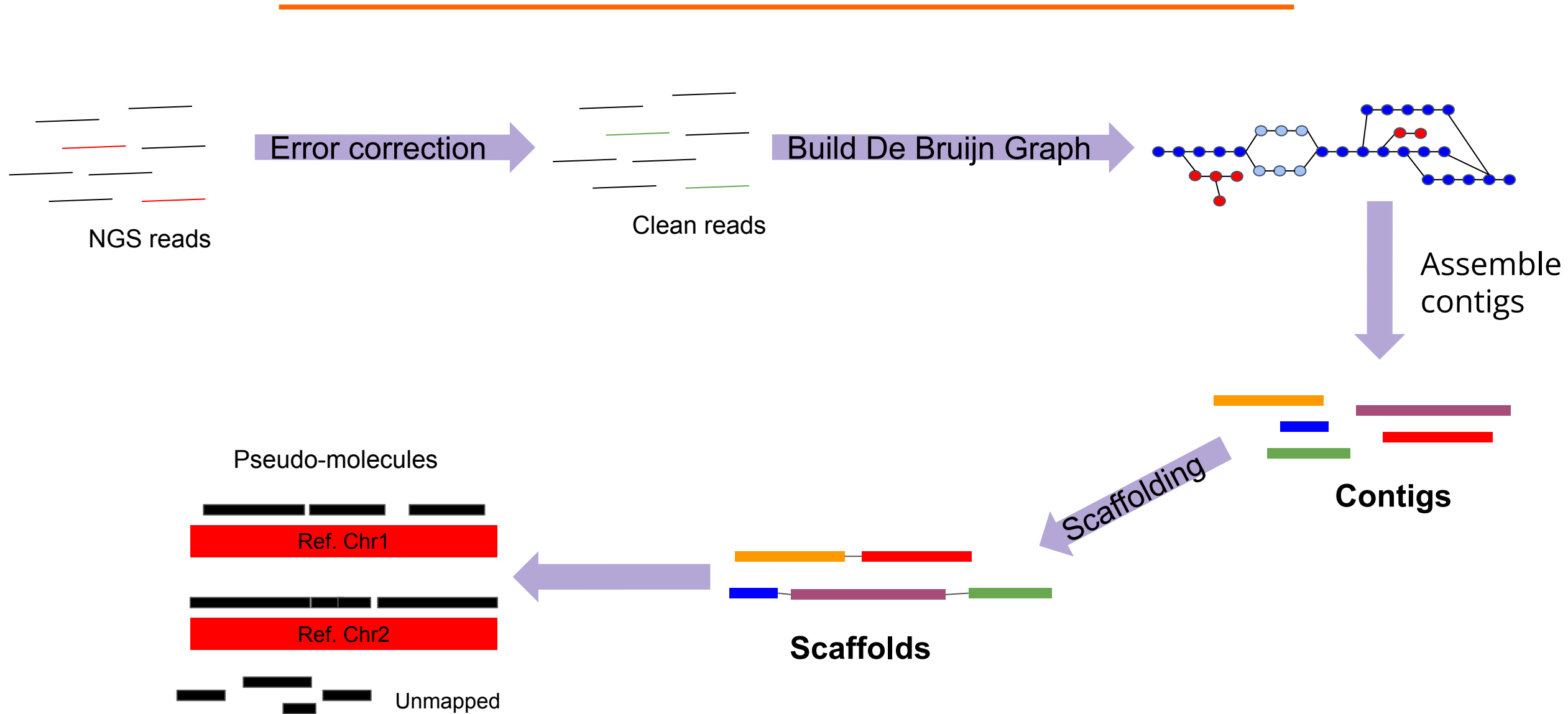
Very low depth (e.g. low complexity regions) can fragment the graph:



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



Workflow



Error Correction

Extract all k-mers from all reads

Count how many times each k-mer was observed

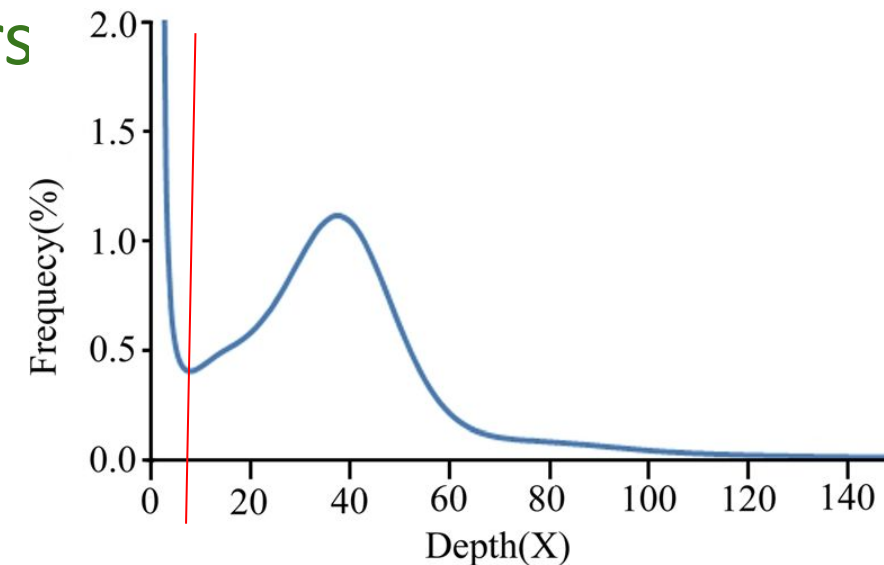
Label rare k-mers as error k-mers

Find reads from which error k-mers came

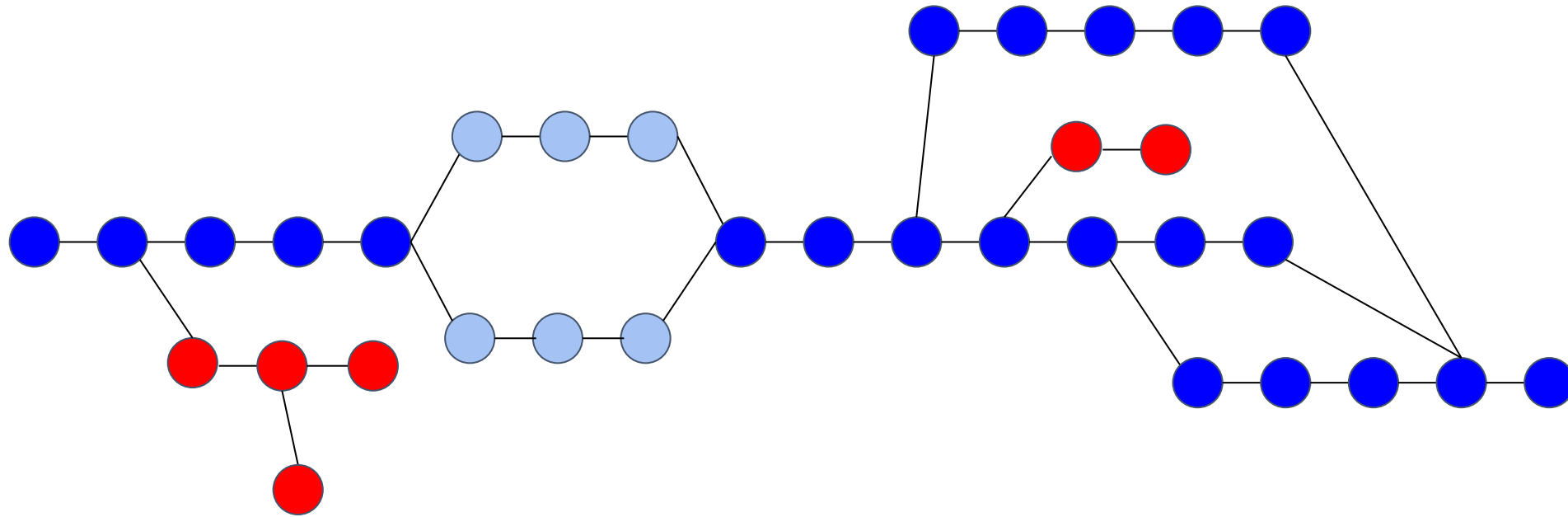
Discard or correct the reads with error k-mers

Count(GGATAGGCACCAGTTAT) = 30

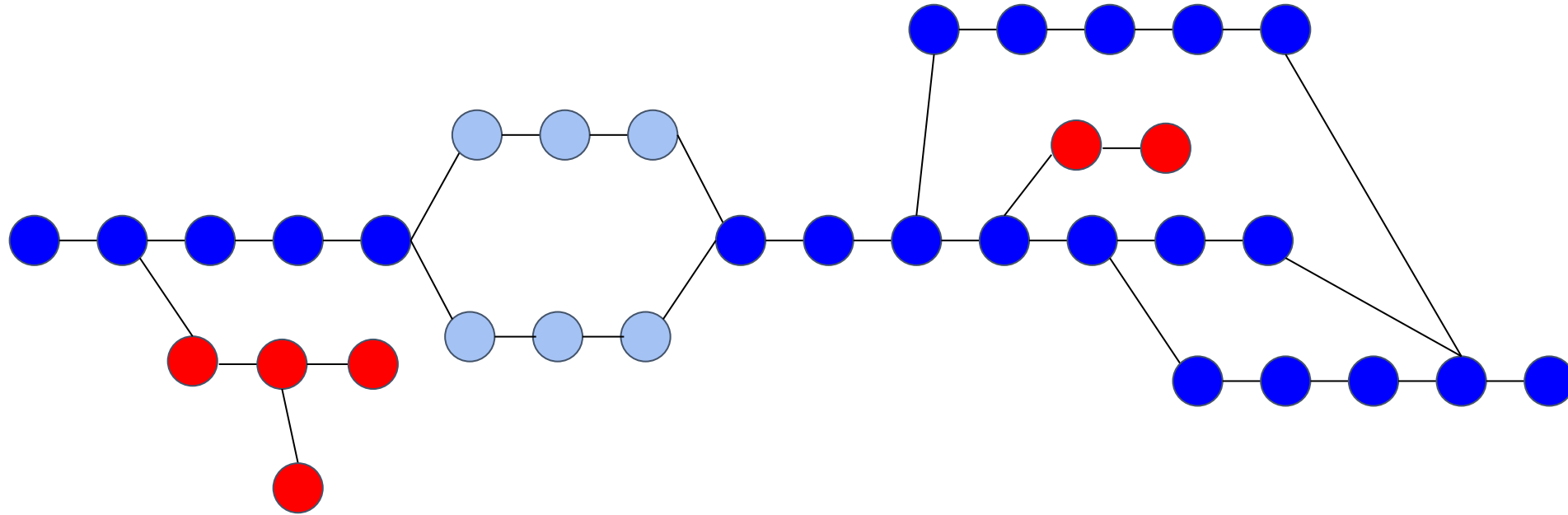
Count(GGATAGG**T**ACCAGTTAT) = 1



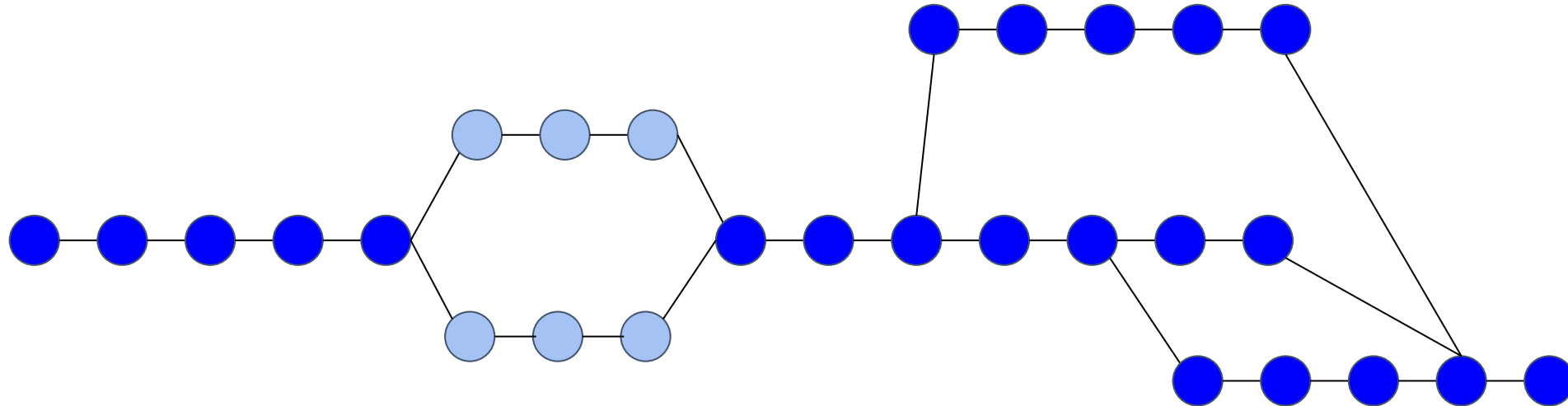
Build De Bruijn Graph



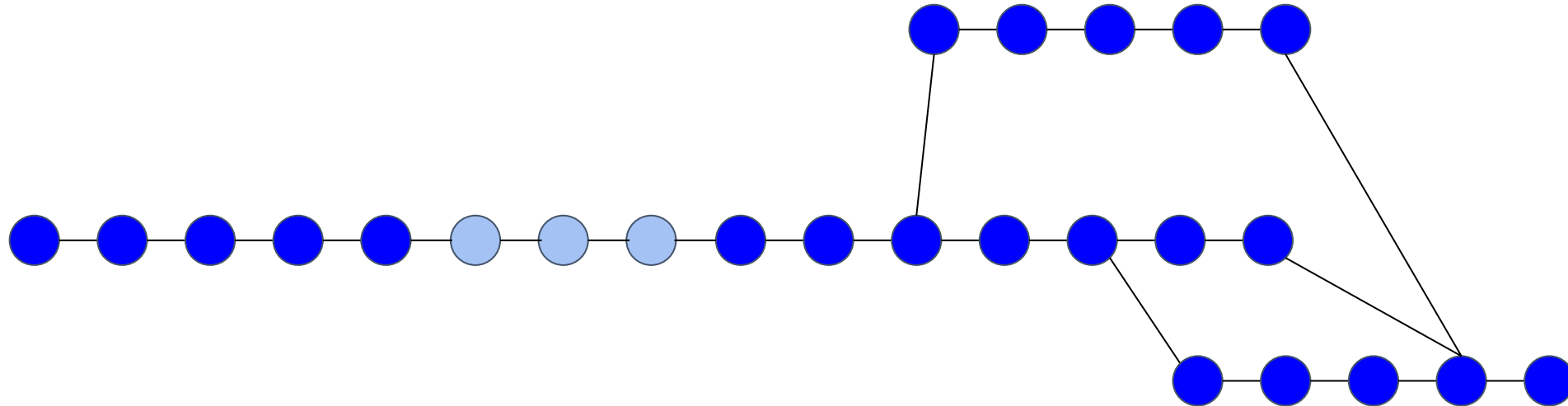
Prune Error Branches



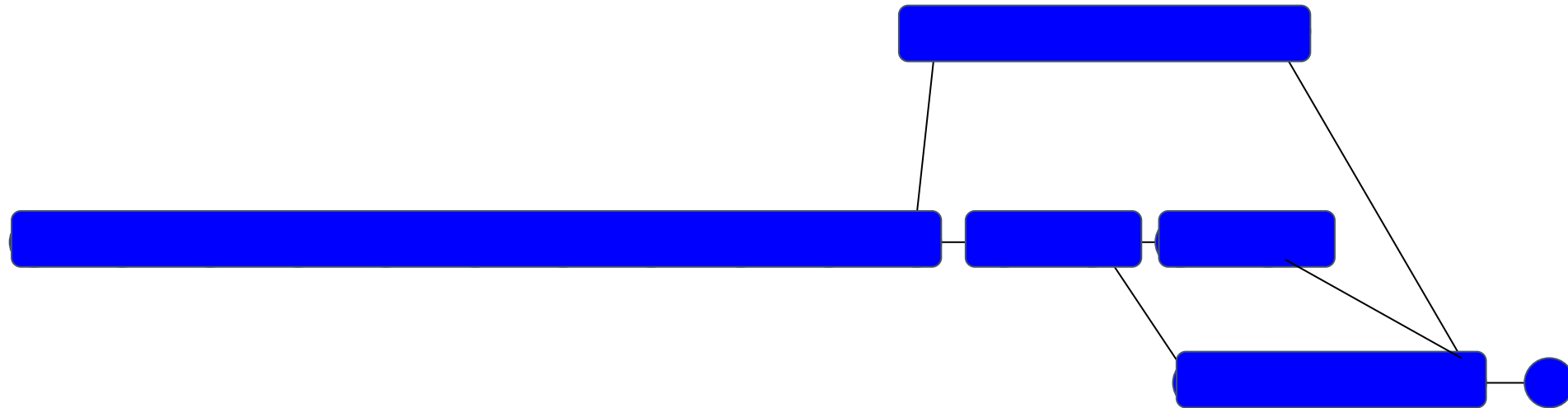
Resolve Bubbles



Resolve Bubbles



Create Contigs

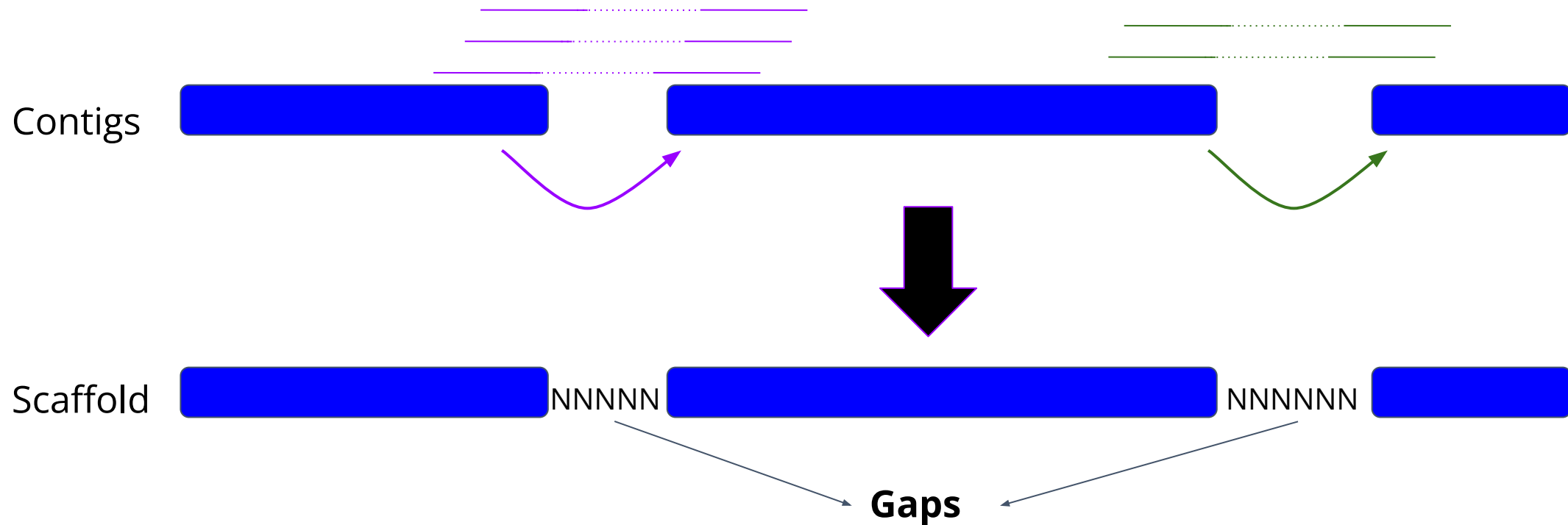


Scaffolding

Mate Pair Sequencing

Use paired end information

Look for evidence of two contigs linked together



De Bruijn Graph

Currently the most popular assembly method

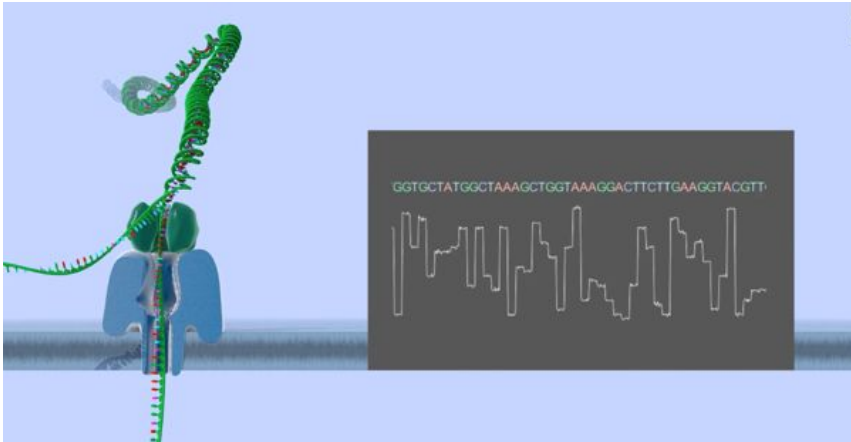
Many software tools use variants of the method

- SPAdes
- SOAPdenovo
- AbySS
- MEGAHIT

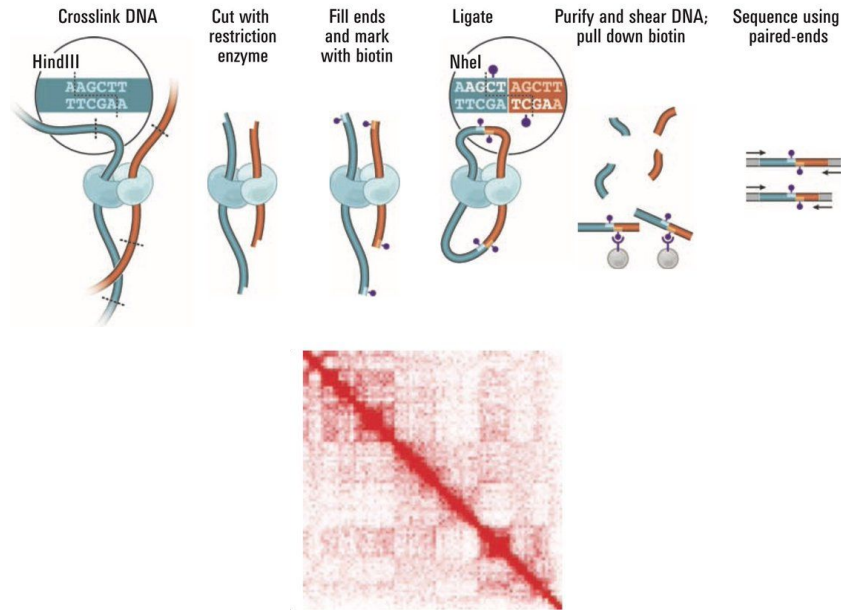


SPAdes

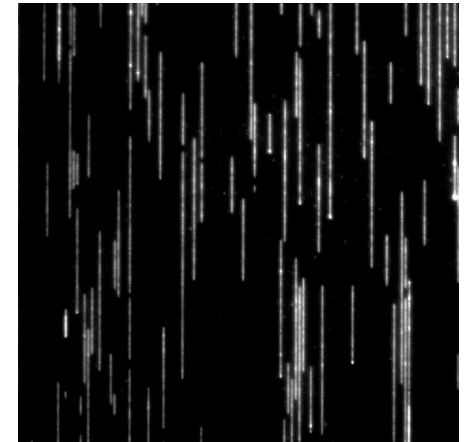
Emerging Technologies for Genome Assemblies



Long reads



Hi-C



Optical mapping