# Computational Molecular Biology and Bioinformatics

## Genome Assembly

Malay Bhattacharyya

Associate Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

August, 2024

# The biology behind genome assembly

The entire genome cannot be sequenced in one go. Genome assembly (also known as `sequence assembly` or `fragment assembly`) refers to the process of aligning and merging fragments from a longer DNA sequence in order to reconstruct the original sequence.

- `Whole-genome shotgun sequencing` starts by copying and fragmenting (in random order) the DNA followed by sequencing of those fragments, and reassembling them to form a genomic sequence.
- `Next-generation sequencing` starts by fragmenting DNA/RNA into multiple pieces (non-overlapping), followed by adding adapters, sequencing the libraries in parallel, and reassembling them to form a genomic sequence.

# The biology behind genome assembly

**Whole-genome shotgun sequencing:**

| | |
|---|---|
| **Input:** | AAGTCCGTAACCGTCAATTTCGAGGGCA |
| **Copies:** | AAGTCCGTAACCGTCAATTTCGAGGGCA |
| | AAGTCCGTAACCGTCAATTTCGAGGGCA |
| | AAGTCCGTAACCGTCAATTTCGAGGGCA |
| **Fragments:** | AAGT\|CCGTAACCGT\|CAATTTCGA\|GGGCA |
| | AAGTCCG\|TAACCGTCAA\|TTTCGAGGGCA |
| | AAGTCCG\|TAACCGTCAATTTCG\|AGGGCA |

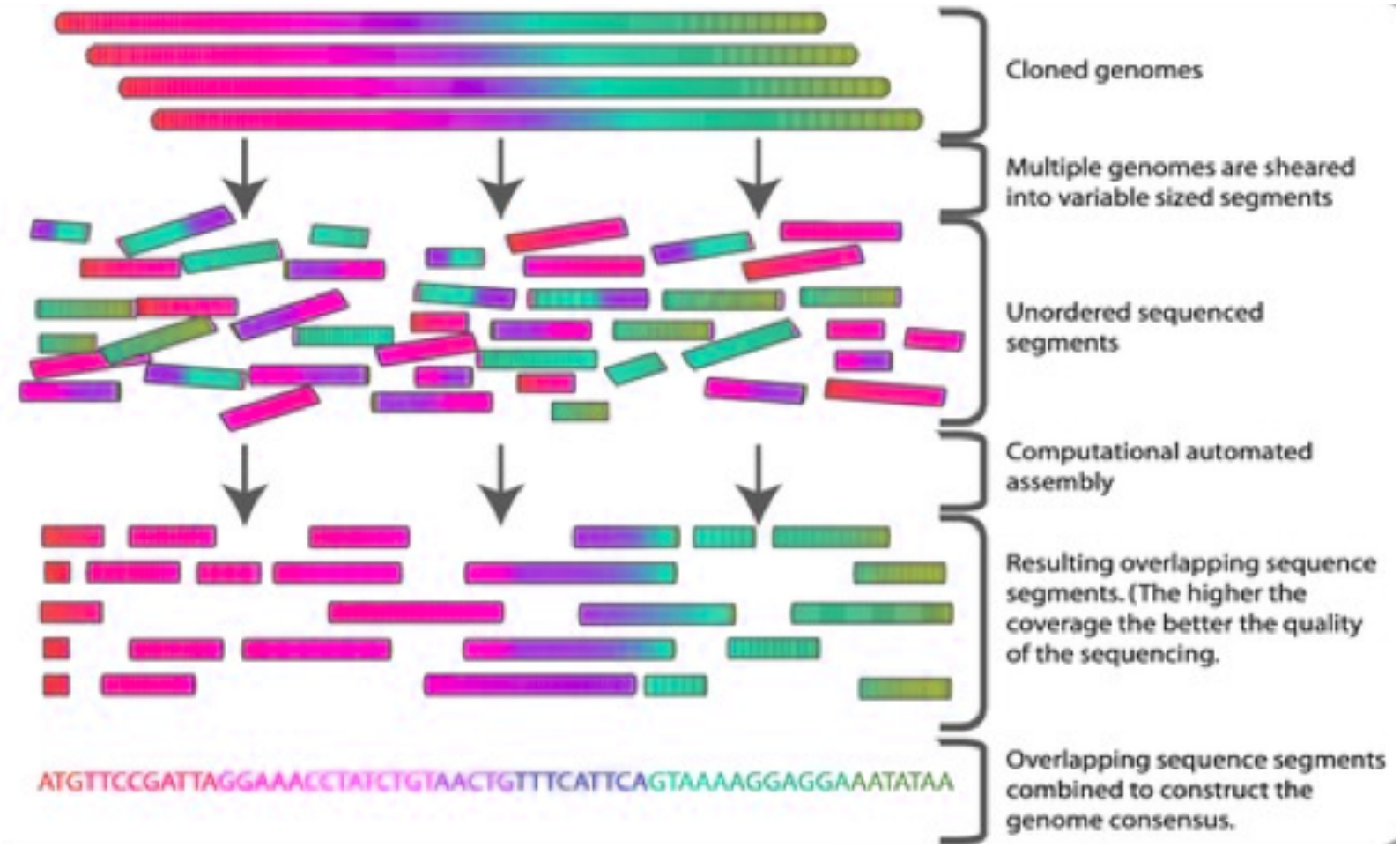
**Next-generation sequencing:**

| | |
|---|---|
| **Input:** | AAGTCCGTAACCGTCAATTTCGAGGGCA |
| **Fragments:** | AAGT\|CCGT\|AACC\|GTCA\|ATTT\|CGAG\|GGCA |

# Basic terminologies

- **Read:** A 200 to 800 character long word that comes out of a sequencer.
- **Mate pair:** A pair of reads from two ends of the same insert fragment.
- **Contig:** A contiguous sequence formed by several overlapping reads with no gaps.
- **Supercontig (scaffold):** An ordered and oriented set of contigs, usually by mate pairs.
- **Consensus sequence:** A sequence derived from the multiple alignment of reads in a contig.

# The mechanism




The entire genome is randomly fragmented and then reassembled

# The coverage

Coverage (also known as `read depth` or `depth`) is defined by the average number of reads representing a given character in the consensus sequence. Given the length of the original genome (G), number of reads (N), and average read length (L), one can calculate the coverage ($\lambda$) as follows.

$$\lambda = \frac{N \times L}{G}$$

For example, a hypothetical genome with 10,000 characters (bases) reconstructed from 25 reads with an average length of 800 characters will have 2x redundancy.

# Estimating uncovered bases (Lander-Waterman theory)

Let us assume that the probability that a base is not covered is represented with the following Poisson distribution.

$$\text{Posisson}(0, \lambda) = e^{-\lambda}$$

Then the number of uncovered characters (bases) can be estimated as

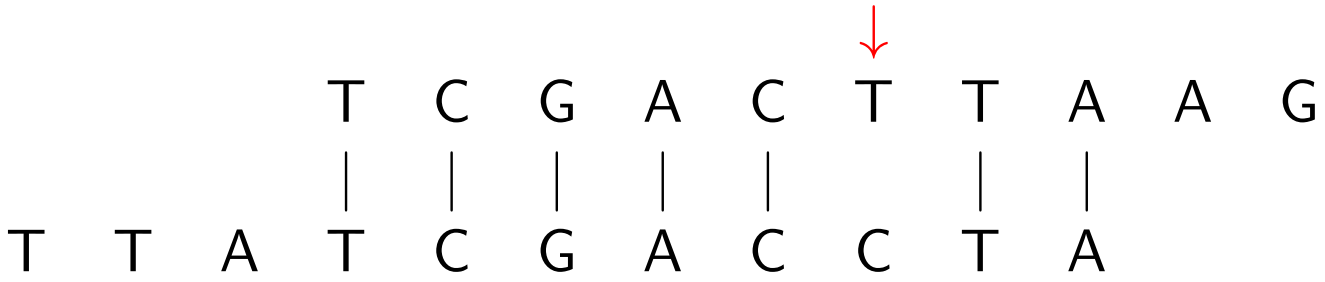
$$Ge^{-\lambda}.$$

Similarly, the number of gaps can be estimated as

$$Ne^{-\lambda}$$

# The approach of assembly

Even if a pair of reads actually originate from overlapping stretches of the genome, there might be differences.



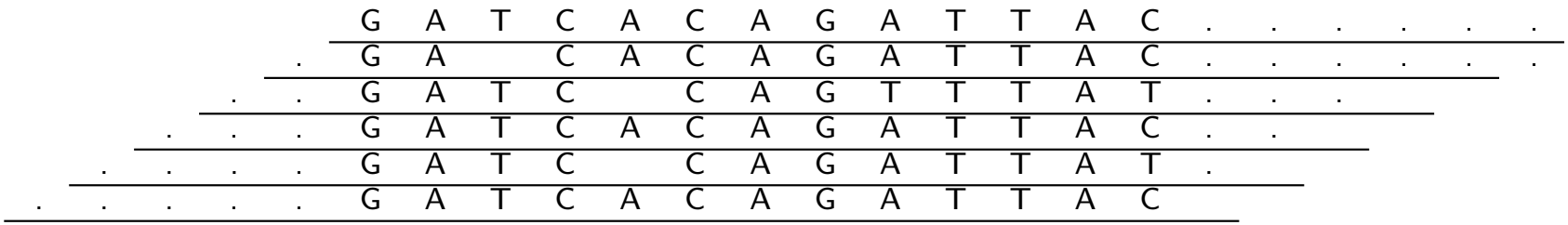

As a result, we have the following issues.

1. Sequencing error
2. Difference between inserted copies of chromosome

**Note:** Humans are diploid (have two copies of each chromosome – one from mother, one from father). The copies can differ.

# The approach of assembly

**Step 1:** Creating local multiple alignments



# The approach of assembly

**Step 2:** Correcting errors

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| G | A | T | C | A | C | A | G | A | T | T | A | C |
| G | A | | C | A | C | A | G | A | T | T | A | C |
| G | A | T | C | | C | A | G | T | T | T | A | T |
| G | A | T | C | A | C | A | G | A | T | T | A | C |
| G | A | T | C | | C | A | G | A | T | T | A | T |
| G | A | T | C | A | C | A | G | A | T | T | A | C |

Insert T  Replace T with A

**Note:** The error correction step can remove up to 98% of the errors in practice.

# The approach of assembly

**Step 2:** Correcting errors (continued)

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| G | A | T | C | A | C | A | G | A | T | T | A | C |
| G | A | | C | A | C | A | G | A | T | T | A | C |
| G | A | T | C | | C | A | G | T | T | T | A | T |
| G | A | T | C | A | C | A | G | A | T | T | A | C |
| G | A | T | C | | C | A | G | A | T | T | A | T |
| G | A | T | C | A | C | A | G | A | T | T | A | C |

Correlated errors – probably due to repeats

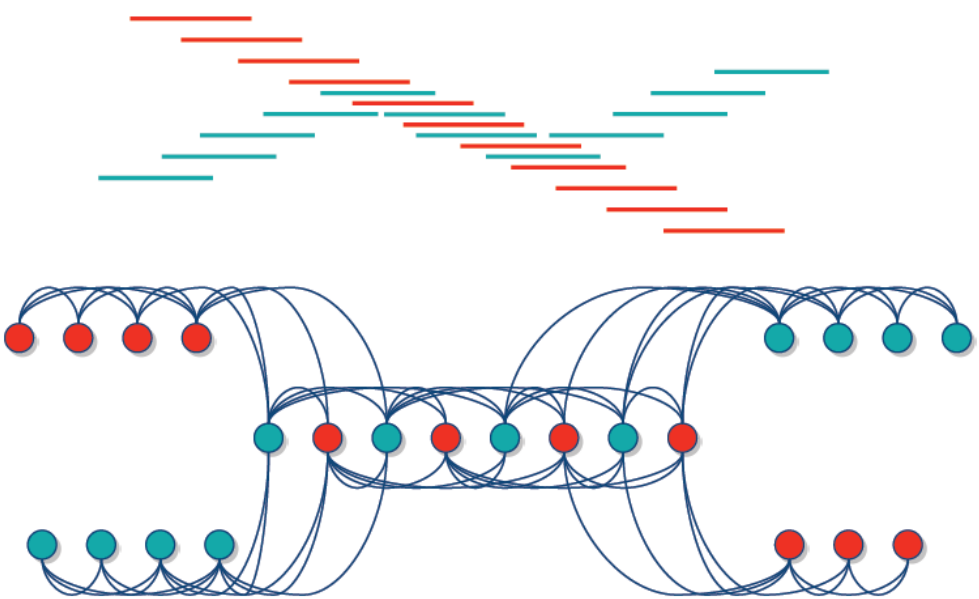
Disentangle overlaps

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| G | A | T | C | A | C | A | G | A | T | T | A | C |
| G | A | | C | A | C | A | G | A | T | T | A | C |
| G | A | T | C | A | C | A | G | A | T | T | A | C |
| G | A | T | C | A | C | A | G | A | T | T | A | C |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| G | A | T | C | | C | A | G | T | T | T | A | T |
| G | A | T | C | | C | A | G | A | T | T | A | T |

# The approach of assembly

**Step 3:** Merging reads



Reads that come from two regions of the genome (blue and red) that contain the same repeat

Note:
of course, we don't know the "color" of these nodes

# A note

Next-generation sequencing can be of two types as listed below.

- Short-read sequencing (e.g., Illumina sequencing, Roche 454 sequencing)
- Long-read sequencing (e.g., Nanopore sequencing)

The methods of genome assembly are same in general for both the aforementioned types. However, for obvious reasons, assembly of shorter reads are more challenging.

# Basic terminologies

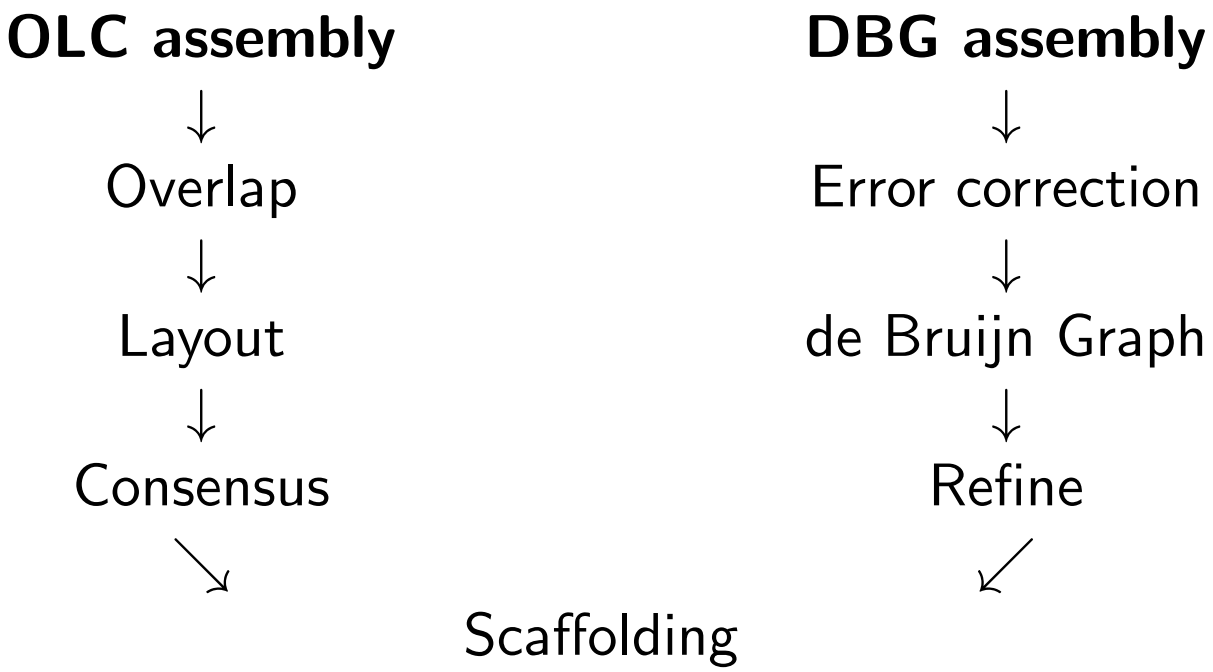
- **Read:** A 50 to 300 character long (for short-read sequencing) or 500 to 2.3 million character long (for long-read sequencing) word that comes out of a sequencer.
- **Contig:** A contiguous sequence formed by several overlapping reads with no gaps.
- **Scaffold:** An ordered and oriented set of contigs.
- **Consensus sequence:** A sequence derived from the multiple alignment of reads.

# The approach of assembly

The fragment assembly methods for short reads are of two types:

1. Overlap Layout Consensus (OLC) assembly
   - Employs string graph-based assemblers
   - Constructs overlap graph from short reads
   - Eliminates redundant reads
   - Traces paths in the graph for assembly
   - Examples include SGA, Fermi, etc.
2. de Bruijn Graph-based (DBG) assembly
   - Employs de Bruijn graph-based assemblers
   - Constructs k-mer graph from short reads
   - Discards original reads
   - Traces paths in the graph for assembly

# The approach of assembly



# The OLC assembly

**Step 1 – Overlap:** Building an overlap graph

An overlap graph is a directed graph which is constructed with the following considerations.

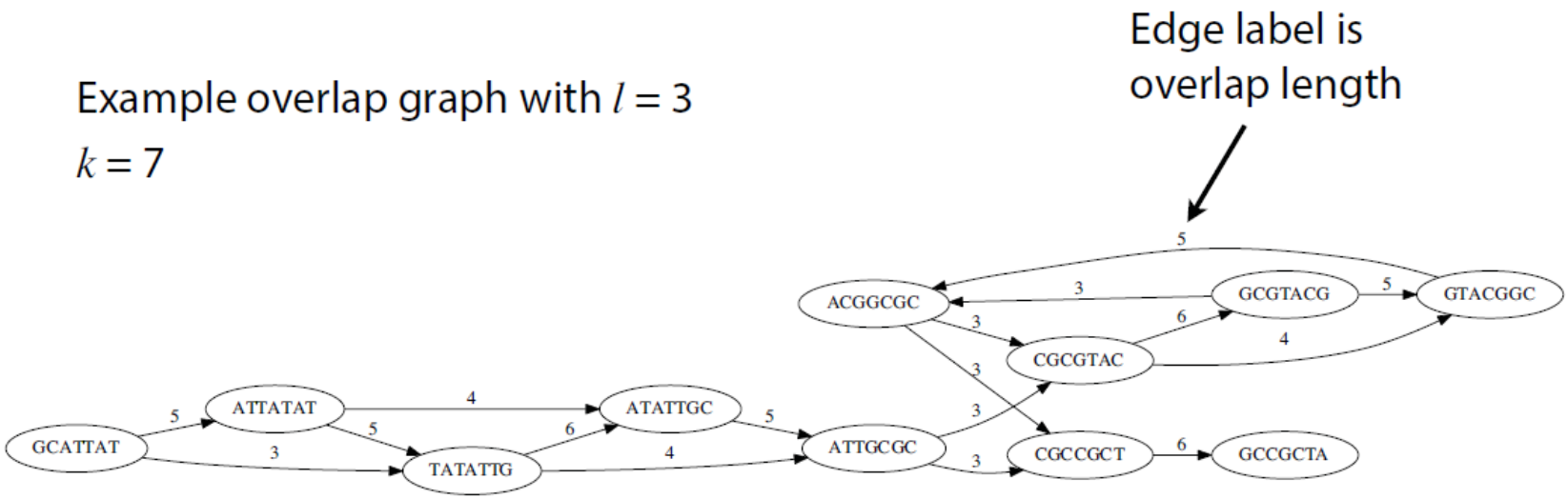
- A node $\equiv$ A read
- A edge $\equiv$ An overlap of reads

An edge $(r1, r2)$ signifies that a suffix of the read $r1$ (source) is similar to a prefix of the read $r2$ (sink).

**Note:** An overlap graph may contain a cycle.

# The OLC assembly

**Step 1 – Overlap:** Building an overlap graph




**Note:** A merged FM index of all reads enables to match prefixes and suffixes of reads to explore the overlaps.

# The OLC assembly

**Step 2 – Layout:** Putting together stretches of the overlap graph into contigs

We solve the shortest common superstring problem on the overlap graph by revising the graph with edges having a cost of −(length of overlap). The shortest common superstring corresponds to a path that visits every node once (Hamiltonian path), minimizing the cost of traversal.

**Note:** Hamiltonian cycle problem is NP-complete.

# The OLC assembly

**Step 3 – Consensus:** Choose the most probable nucleotide sequence for each contig

We remove the transitive edges from the graph.

# The FM-index

An FM-index is a compressed full-text substring index based on the Burrows–Wheeler transform (BWT). The BWT algorithm and its inverse version work as follows.

```
function BWT(string s){
    R <- All possible rotations of s as elements // Right circular shift
    SortedR <- The elements of R in lexicographic order
    return(Last character of each element in SortedR in the order they appear)
}

function InverseBWT(string s){
    SortedT <- {NULL, NULL, ... length(s) times}
    for(i = 1 to length(s)){
        T <- Insert characters of s as prefixes of elements in SortedT
        SortedT <- The elements in T in lexocographic order
    }
    return(The element in SortedT that ends with \$)
}
```

# Burrows–Wheeler Transform (BWT) – An example

| Input | Rotate | Sort | Output |
|---|---|---|---|
| ^CAGAGA$ | ^CAGAGA$ | AGAGA$^C | CGG^AA$A |
| | $^CAGAGA | AGA$^CAG | |
| | A$^CAGAG | A$^CAGAG | |
| | GA$^CAGA | CAGAGA$^ | |
| | AGA$^CAG | GAGA$^CA | |
| | GAGA$^CA | GA$^CAGA | |
| | AGAGA$^C | ^CAGAGA$ | |
| | CAGAGA$^ | $^CAGAGA | |

**Note:** Two-dimensional array can be used as a data structure.

# Inverse BWT – An example

| Input | Insert 1 | Sort 1 | Insert 2 | Sort 2 | ... | Sort 8 |
|---|---|---|---|---|---|---|
| CGG^AA$A | C | A | CA | AG | | AGAGA$^C |
| | G | A | GA | AG | | AGA$^CAG |
| | G | A | GA | A$ | | A$^CAGAG |
| | ^ | C | ^C | CA | | CAGAGA$^ |
| | A | G | AG | GA | ... | GAGA$^CA |
| | A | G | AG | GA | | GA$^CAGA |
| | $ | ^ | $^ | ^C | | ^CAGAGA$ |
| | A | $ | A$ | $^ | | $^CAGAGA |

# The DBG assembly

**Step 1 – Error Correction:** Correcting errors in the reads

Similar to assembly for whole-genome shotgun sequencing

# The DBG assembly

**Step 2 – de Bruijn Graph:** Building a de Bruijn graph

Given a sequence $s$ and the k-mer length $k$, we can construct a de Bruijn graph with the following characteristics:

- the nodes are ($k$-1)-mers, and
- an edge is present between a pair of nodes if they have ($k$-2)-long overlap.

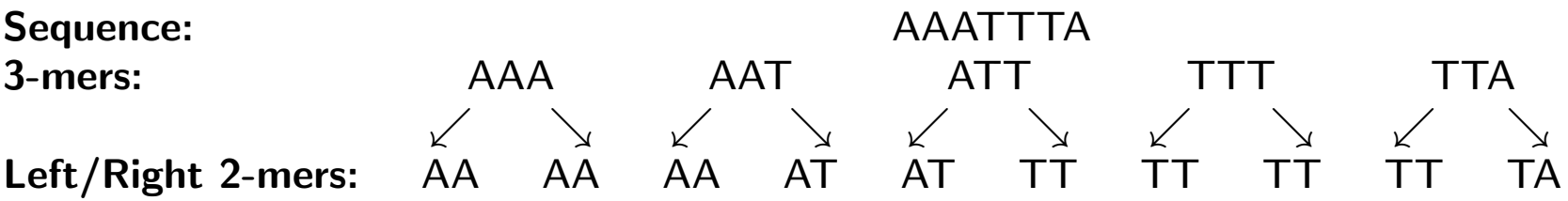
Consider the sequence `AAATTTA` and the value of $k = 3$. Let us construct the corresponding de Bruijn graph.

**Note:** The value of $k$ is taken as odd.

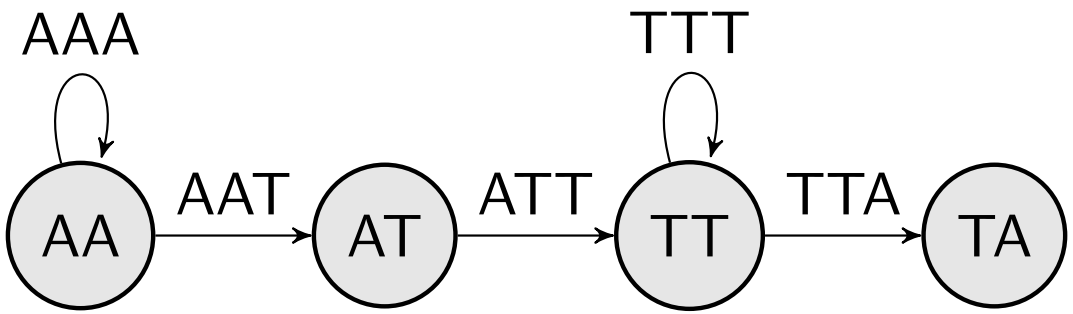
# The DBG assembly

**Step 2 – de Bruijn Graph:** Building a de Bruijn graph (continued)

We can obtain the left and right ($k$-1)-mers of each $k$-mer present in the sequence as follows.



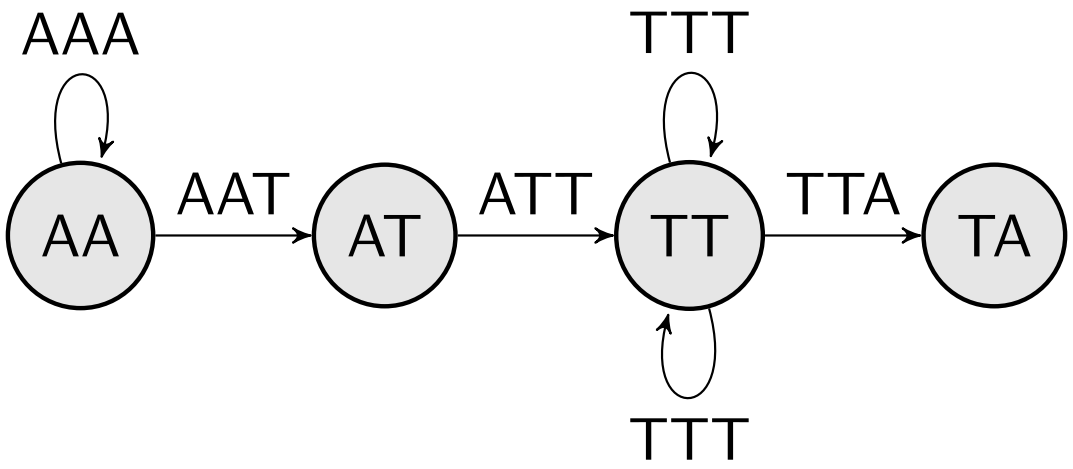

We then construct the corresponding de Bruijn graph as follows.



# The DBG assembly

**Step 2 – de Bruijn Graph:** Building a de Bruijn graph (continued)

Note that, if we add one more `T` to our input sequence such that $s$ becomes `AAATTTTA`, and reconstruct the corresponding de Bruijn graph, we get a *multiedge* in the graph as follows.



# The DBG assembly

**Step 2 – de Bruijn Graph:** Building a de Bruijn graph (continued)

A graph is *connected* if there is a path between every pair of vertex. An *Eulerian walk* visits each edge exactly once. A directed, connected graph is *Eulerian* if it contains an Eulerian walk.

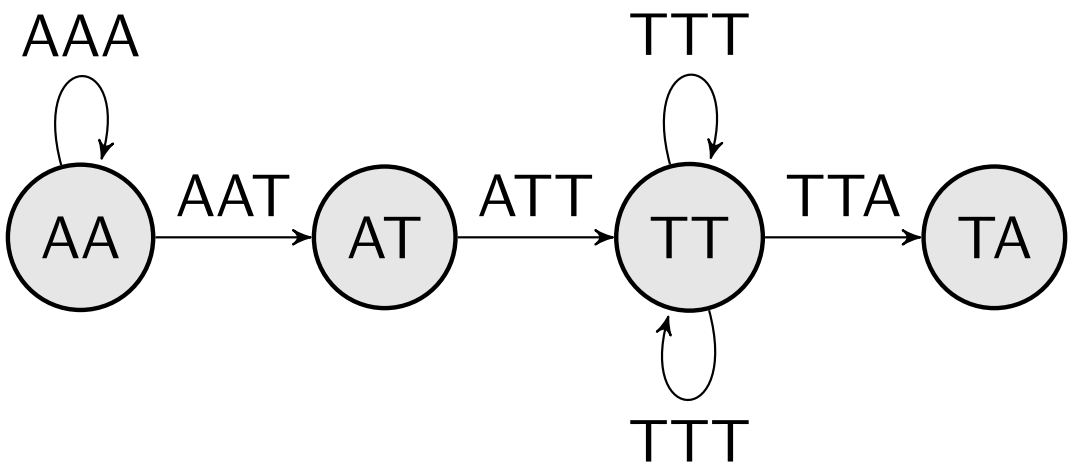
A node is *balanced* if its in-degree = out-degree. A node is *semi-balanced* if its |in-degree - out-degree| = 1. A directed, connected graph is Eulerian if and only if it has at most 2 semi-balanced nodes and all other nodes are balanced.

**Note:** For simplicity, we do not distinguish Eulerian from semi-Eulerian.

# The DBG assembly

**Step 2 – de Bruijn Graph:** Building a de Bruijn graph (continued)

Note that, the following graph is Eulerian.




Arguments:

1. AA $\rightarrow$ AA $\rightarrow$ AT $\rightarrow$ TT $\rightarrow$ TT $\rightarrow$ TA
2. AA and TA are semi-balanced, AT and TT are balanced

# The DBG assembly

**Step 2 – de Bruijn Graph:** Building a de Bruijn graph (continued)

For genome assembly, each $k$-mer is recorded in *twin* nodes – one node in forward direction and one node in reverse complement.

With perfect sequencing, this procedure always yields an Eulerian graph unless the genome is circular. Hence, we just need to find out the Eulerian walk in the graph.

**Note:** No node can be its own reverse complement because $k$ is odd.

# The DBG assembly

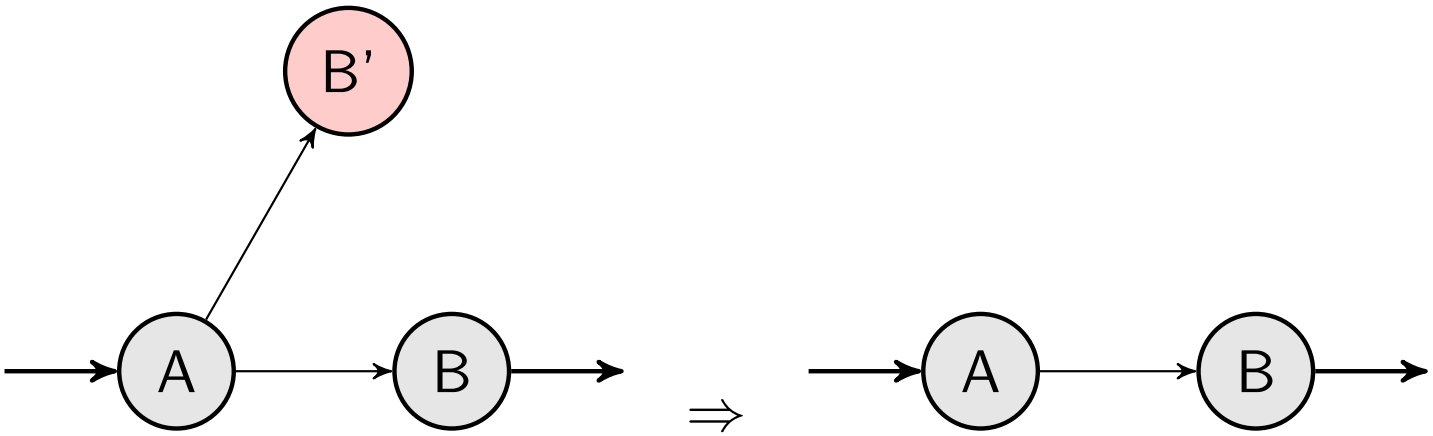
**Step 2 – de Bruijn Graph:** Building a de Bruijn graph (continued)

Note that, there are further challenges to deal with as listed below.

1. A de Bruijn graph can have multiple Eulerian walks, *only one* of which corresponds to original superstring.
2. Having gaps in the coverage can lead to a disconnected de Bruijn graph. The connected components are individually Eulerian, but not the overall graph.
3. Differences in coverage might lead to non-Eulerian de Bruijn graphs.
4. Errors and differences between chromosomes might lead to non-Eulerian de Bruijn graphs.

# The DBG assembly

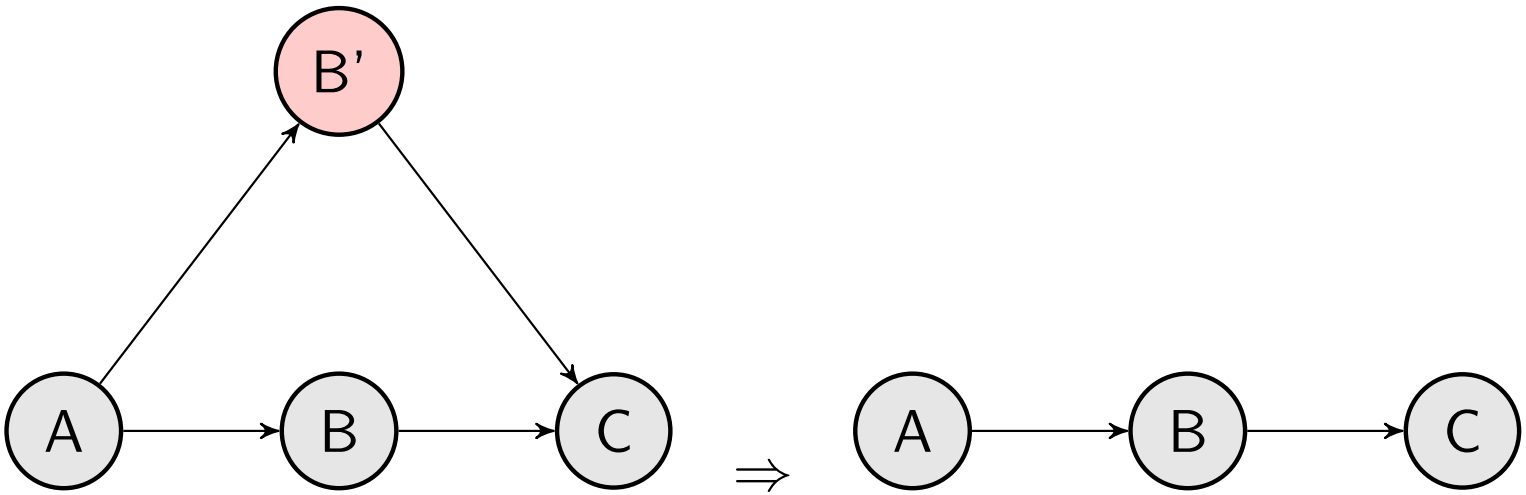
**Step 3 – Refine:** Refining the de Bruijn graph




If errors are present at the end of read, remove 'dead-end' tips

# The DBG assembly

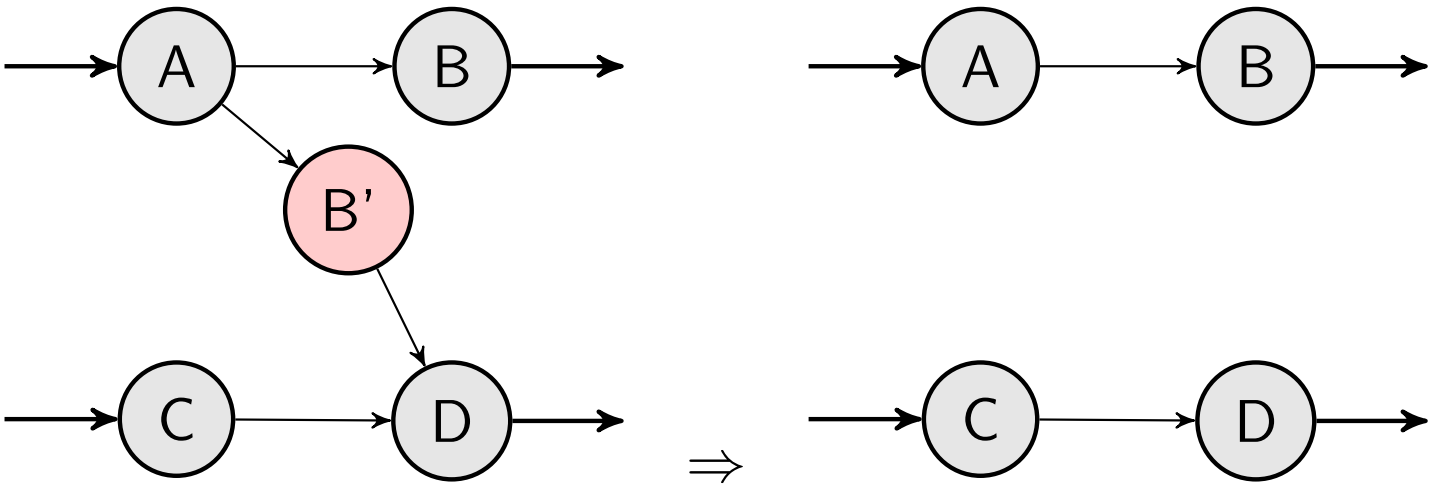
**Step 3 – Refine:** Refining the de Bruijn graph (continued)




If errors are present in the middle of read, remove 'bubbles'

# The DBG assembly

**Step 3 – Refine:** Refining the de Bruijn graph (continued)




If chimeric edges are present, remove short and low coverage nodes

# Comparing OLC and DBG assembly

DBG is faster and simple to implement but suffers from the following limitations.

- It cannot resolve repeats as well as overlap graphs because reads are immediately split into shorter k-mers.
- It cannot deal with complicated errors because only a specific type of overlap is considered.
- Some paths through de Bruijn graphs are inconsistent with respect to input reads. Hence, the read coherence is lost.

# Hands-on

1. Download the following paper and do the following:
   Kuronen, J., Horsfield, S.T., Pöntinen, A.K., Mallawaarachchi, S., Arredondo-Alonso, S., Thorpe, H., Gladstone, R.A., Willems, R.J., Bentley, S.D., Croucher, N.J. and Pensar, J., Pangenome-spanning epistasis and coselection analysis via de Bruijn graphs. Genome Research, 34(7), pp.1081-1088, 2024.
   i) Get the implementation from:
   `https://genome.cshlp.org/content/34/7/1081/suppl/DC1`
   ii) Look at the supplementary details for a better understanding of the implementation.
   iii) Could you identify some limitations of this implementation? How can you overcome that? Any suggestions?
2. Attempt one of the challenges given in the following paper:
   Pibiri, G.E., Fan, J. and Patro, R., Meta-colored compacted de Bruijn graphs. In RECOMB, pp. 131-146, 2024.