Melbourne Bioinformatics
BIOINFORMATICS + DATA SERVICES + INFRASTRUCTURE, FOR LIFE SCIENCES TODAY

THE UNIVERSITY OF MELBOURNE

# COMP90014

Algorithms for Bioinformatics

Week 8A: Considerations & Graph Simplification

# Assembly Algorithms

## Recap

### Overlap Layout Consensus

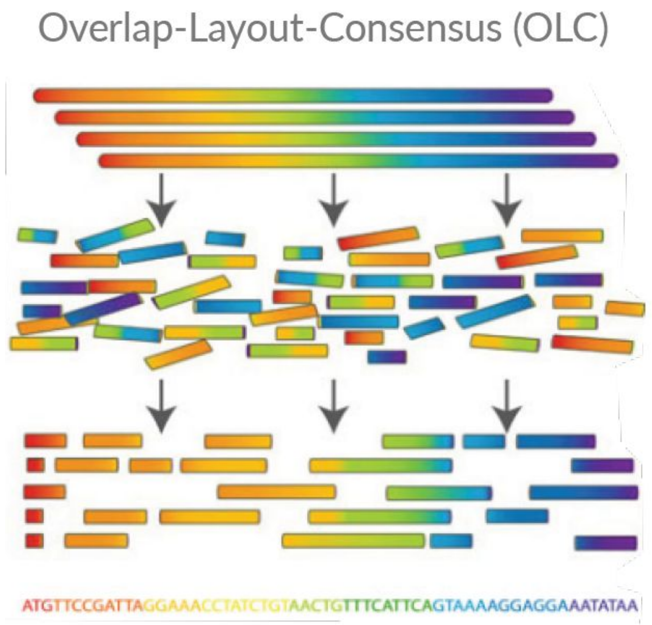
Strategy of choice for long read assembly

| | |
|---|---|
| Overlap | Build overlap graph |
| Layout | Group parts of graph into contigs |
| Consensus | Squash reads to generate consensus contigs |

Overlap（重叠）：在这个阶段，计算所有 reads 之间的重叠关系。目标是找出哪些 reads 可能连续出现在原始基因组中。

Layout（布局）：基于重叠关系，将 reads 排列成一个或多个连续的序列。在这一步，重叠的 reads 被组合在一起，形成一个更长的序列。

Consensus（共识）：由于测序错误和其他因素，重叠的部分可能在某些位置上有差异。共识阶段的目的是解决这些差异，得到一个最可能的单一序列。


Overlap-Layout-Consensus (OLC)
ATGTTCCGATTAGGAAACCTATCTGTAACTGTTTCATTCAGTAAAAGGAGGAAATATAA

Commins et al., 2009

# Recap

## Overlap Layout Consensus

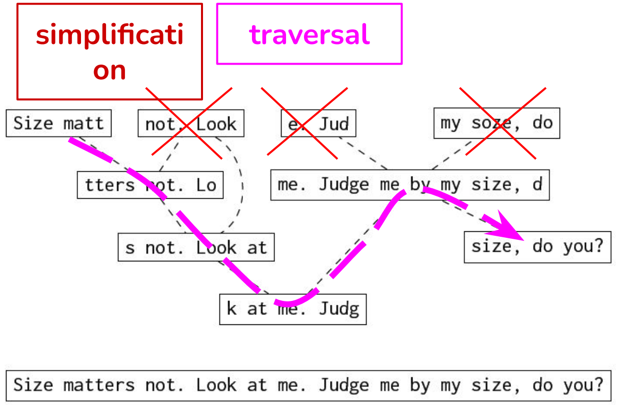
Strategy of choice for long read assembly

| | |
|---|---|
| Overlap | Build overlap graph |
| Layout | Group parts of graph into contigs |
| Consensus | Squash reads to generate consensus contigs |

Graph nodes = reads
Graph edges = overlaps

alignment
找到overlap
布局
共识

### Overlap, Layout



### Consensus

**reads:**

```
ATCGATGCTAGCTGA---
-----TGCTAGCTGATGA
-TCGAAG-TAG-TGATG-
ATAGATGCTAGCTGA-GA
```

**consensus:**

```
ATCGATGCTAGCTGATGA
```

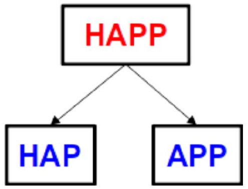
# Recap

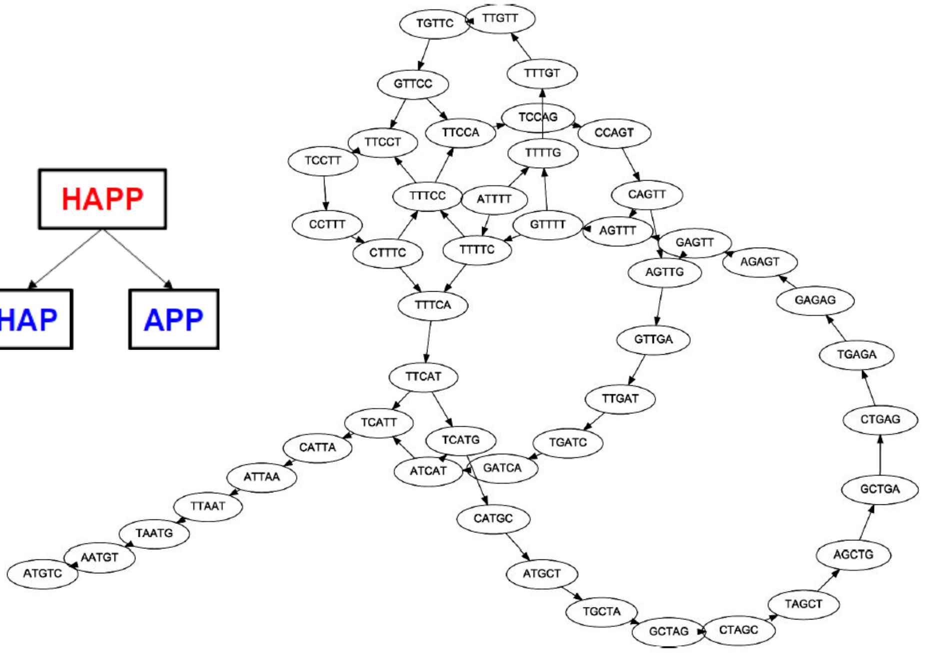
## De Bruijn Graphs

Strategy of choice for short read assembly

1. Break reads into kmers
2. Separate kmers into prefix - suffix
3. Add prefix / suffix and edge to graph

Graph nodes = prefixes / suffixes
Graph edges = kmers





# Recap

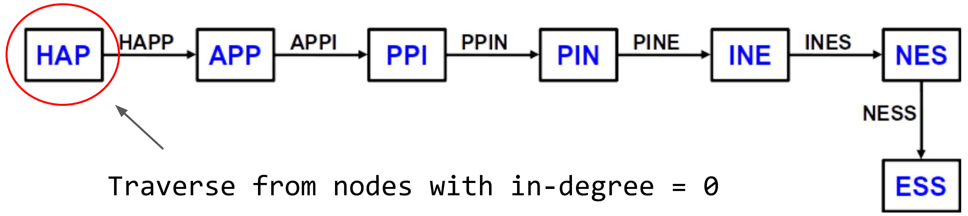
## De Bruijn Graphs

Strategy of choice for short read assembly

1. Break reads into kmers
2. Separate kmers into prefix - suffix
3. Add prefix / suffix and edge to graph

Graph nodes = prefixes / suffixes

Graph edges = kmers



```
Traverse from nodes with in-degree = 0
```

```
HAPP
 APPI
  PPIN
   PINE
    INES
     NESS
---------
HAPPINESS
```

The reads assemble to
HAPPINESS
Easy!

## Assembly Algorithms

Recap

<u>OLC vs De Bruijn Assembly</u>

Considerations

OLC Simplification

## OLC vs De Bruijn

| | OLC | De Bruijn |
|---|---|---|
| Preprocessing | All-vs-all alignment (slow) | Kmer counting (fast) |
| Graph nodes | Reads | Kmer prefixes / suffixes |
| Graph edges | Overlaps | Kmers |
| Traversal | Hamiltonian (slow) | Eulerian (fast) |
| Read coherence | Yes | No |
| Repeat resolution | Better | Worse |
| Read errors | No problem | A problem |

Almost like OLC is well suited to long reads, and De Bruijn is suited to short reads…

# OLC vs De Bruijn

Both OLC and De Bruijn approaches use graphs as intermediate.

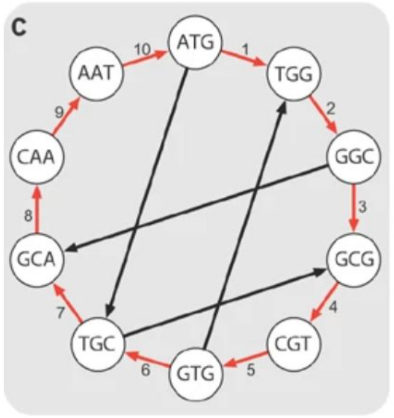
OLC has some pros:

- Better tolerance of read errors
- Better repeat resolution power

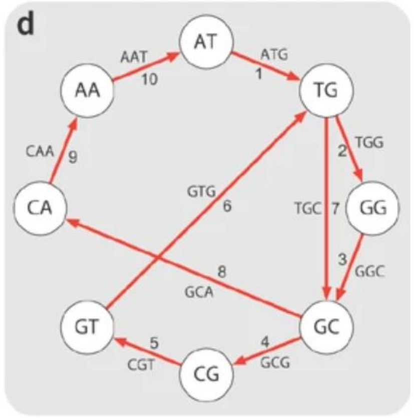
But enormous complexity hit!

- All vs all alignment (get overlaps)
- Hamiltonian vs Eulerian cycles

Why is De Bruijn much more scalable?



**Hamiltonian cycle**
Visit each vertex once
(harder to solve)



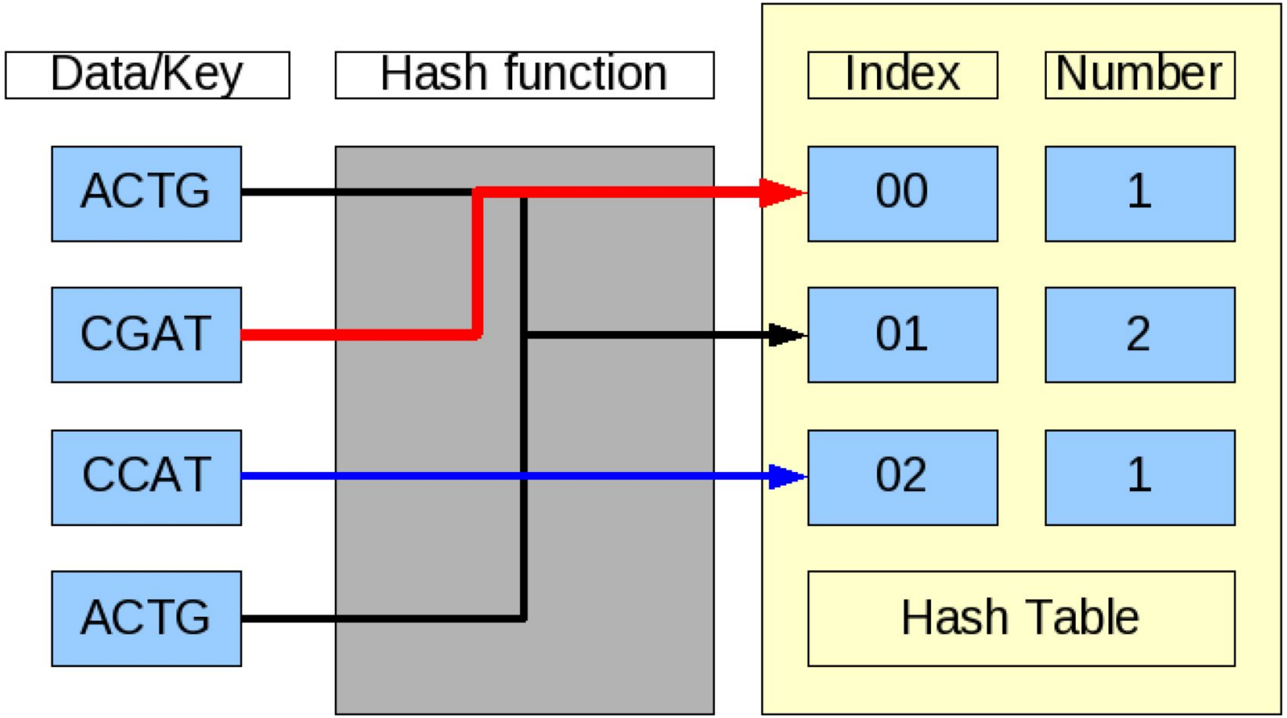
**Eulerian cycle**
Visit each edge once
(easier to solve)

Compeau, Pevzner, Tesler (2011)

## De Bruijn Efficiency

1. Select a value for *k*
2. ***Hash*** the reads (break them into *k*-mers)
3. Count the *k*-mers
4. Make the de Bruijn graph and annotate with *k*-mer frequency
5. Simplify the graph
6. Read off contigs from simplified graph

## De Bruijn Efficiency



## De Bruijn Efficiency

- Make k-mers in $O(N_r)$ time ($N_r$ = number of reads)
- Using de Bruijn graphs is fast
  - exact matches: ***no alignments to compute***!
  - look up *k*-mers in index (***hash***) in linear time
  - can build graph in $O(N_k)$ time ($N_k$ = number of unique *k*-mers)
- Eulerian pathing is fast
  - make contigs in $O(E)$ time ($E$ = number of edges)

## De Bruijn Efficiency

- What de Bruijn graph advantages have we discovered?
  - Can be built in $O(N)$ expected time
    $N$ = total number of reads
  - With perfect data, graph uses $O(\min(N, G))$ space
    $G$ = genome length
  - Note: when depth is high, $G \ll N$
- Compares favourably with overlap graph:
  - Fast overlap graph construction (suffix tree) is $O(N + a)$ time
    $a$ = number of alignments
  - Space is $O(N + a)$.
  - **Remember**: $a$ is $O(N^2)$

# Assembly Algorithms

Recap

OLC vs De Bruijn Assembly

<u>Considerations</u>

OLC Simplification

## Considerations

Last lecture - Toy examples.

In practice, graphs get very complex due to many considerations

## Considerations

Last lecture - Toy examples.

In practice, graphs get very complex due to many considerations

OLC specific

- Graph has lots of edges
  Intrinsic to overlap graphs

De Bruijn specific

- Effect of kmer size, use of kmers
- Read errors
- Ploidy

图有很多边（Graph has lots of edges）：
当使用 OLC 策略进行基因组组装时，我们首先需要寻找 reads 之间的所有可能重叠。这导致产生的图中有大量的边，因为每一个 read 可能与多个其他 reads 重叠。这使得图的结构非常复杂，并增加了计算的复杂性和内存需求。

在构建重叠图时，可能会因为一些内在的原因（如重复的基因组区域或测序错误）导致图结构变得复杂。例如，基因组中的重复区域可能导致一个 read 与多个其他 reads 重叠，这会使得图的结构变得更加复杂。

kmer 的大小效应（Effect of kmer size）：
De Bruijn 图的构建是基于 kmers 的。kmer 的大小（k的值）直接影响图的复杂性和组装的准确性。较小的 k 值可能会导致更多的连接，但可能更容易受到测序错误的影响。较大的 k 值可能会减少连接，但可能错过某些重要的重叠关系。

使用 kmers（use of kmers）：
为了构建 De Bruijn 图，reads 被切割成长度为 k 的片段，称为 kmers。然后基于这些 kmers 构建图，其中每个 kmer 是图中的一个节点，而边表示两个 kmer 的重叠。

测序错误（Read errors）：
由于测序技术的不完美，输入的 reads 中可能存在错误。这些错误在 De Bruijn 图中可能导致错误的连接或断开，使得图的解读更加困难。

Ploidy（倍性）：

倍性是一个生物学概念，指的是一个细胞中染色体的集合数量。例如，人类的体细胞是二倍体，因为它们有两套染色体，一套来自母亲，一套来自父亲。在基因组组装中，考虑倍性很重要，因为不同的倍性水平（如二倍体、四倍体等）可能导致组装过程中的困难，特别是在处理多倍体生物时，因为它们可能有多个非常相似但略有不同的基因组版本。

# Considerations

Last lecture - Toy examples.

In practice, graphs get very complex due to many considerations

OLC specific

- Graph has lots of edges

  Intrinsic to overlap graphs

De Bruijn specific

- Effect of kmer size, use of kmers
- Read errors
- Ploidy

Both

- Repeats
- Strandedness
- Will mention, then promptly forget these.

Repeats（重复）：
重复是基因组组装中的一个主要挑战。基因组中可能存在大量的重复序列，这些重复序列会导致 reads 的误配和组装图的复杂性增加。当一个 read 可以与多个位置匹配时，确定其在基因组中的正确位置变得困难。这是 OLC 和 De Bruijn 等组装方法都需要面对的问题。

Strandedness（链特异性）：
链特异性是指 DNA 的两个互补链中，哪一个链被用来生成 reads。在某些测序技术中，可以确定 reads 来自哪一个链（正链或负链），这称为“链特异性测序”。链特异性的信息有助于提高组装的准确性，因为它可以帮助确定 reads 的方向。然而，不是所有的测序数据都包含链特异性信息

## OLC Specific

### Many edges

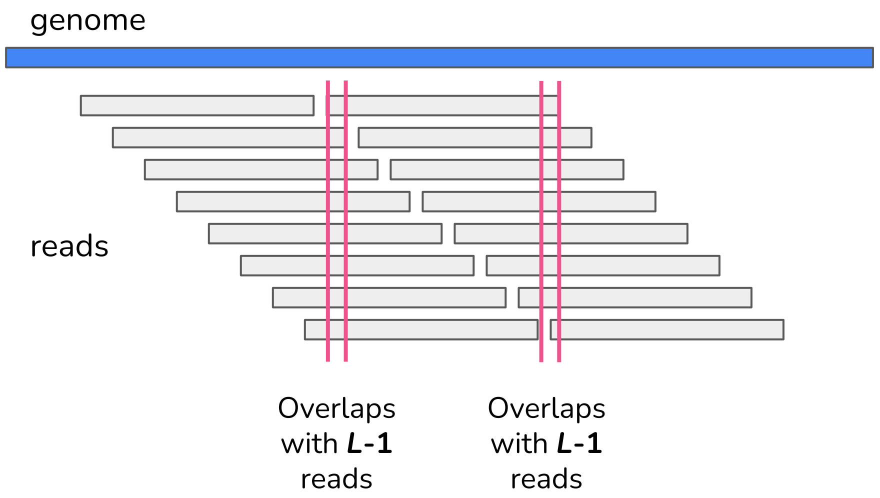
Intrinsic to overlap graphs.

# OLC Specific

## Many edges

Intrinsic to overlap graphs.

- Sequencing depth = 30x
- Average 30 reads cover each position
- Each read overlaps with ~60 other reads
- Not including repetitive sequence
- This is a ***big omission***



Sequencing depth = 30x（测序深度 = 30倍）：
这意味着平均每个基因组位置被测序了30次。测序深度是基因组测序中一个关键的参数，代表了平均每个基因组位置的读取数量
。

Average 30 reads cover each position（平均每个位置有30个 reads 覆盖）：
这是上面测序深度的直接结果。它告诉我们每个基因组的位置大约被30个不同的 reads 覆盖。

Each read overlaps with ~60 other reads（每个 read 与大约60个其他 reads 重叠）：
如果每个位置平均被30个 reads 覆盖，那么考虑到 reads 的两端，一个 read 大致会与其他60个 reads 重叠。这增加了构建重叠图的复杂性。

Not including repetitive sequence（没有包括重复序列）：
重复序列是基因组组装中的一个主要挑战。在这里，您指出的是，上述的计算（每个 read 与大约60个其他 reads 的重叠）没有考虑到基因组中可能存在的重复序列。这意味着实际的重叠可能远远超过这个数字。

This is a big omission（这是一个大的遗漏）：
这句话意味着，不考虑重复序列是一个重大的疏忽。因为重复序列在基因组组装中是一个关键的难点，所以不考虑它可能会导致组装的错误或不准确

# OLC Specific

## Many edges

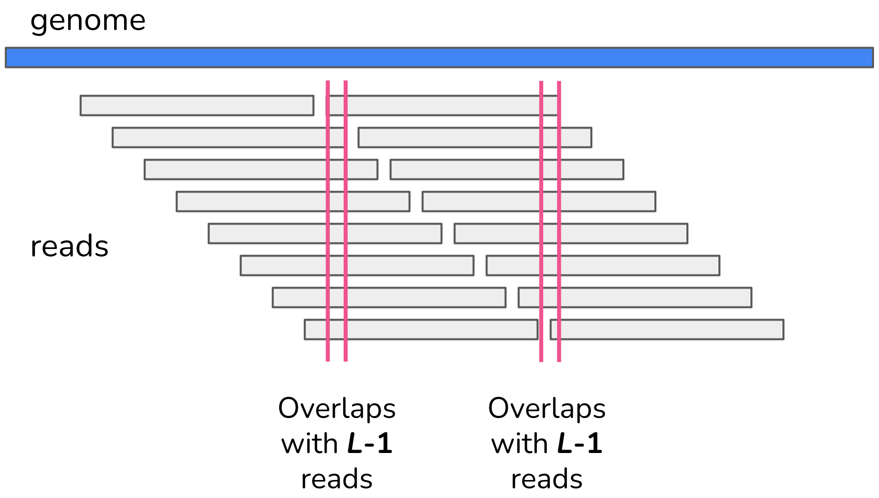
Intrinsic to overlap graphs.

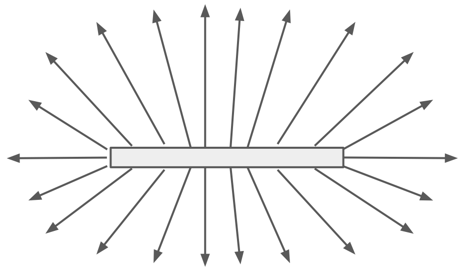
- Sequencing depth = 30x
- Average 30 reads cover each position
- Each read overlaps with ~60 other reads
- Not including repetitive sequence
- This is a ***big omission***

Problematic when calculating hamiltonian path.

Can mitigate by enforcing min overlap len.

Some edges are redundant?





## De Bruijn Specific

### Kmers

Short reads already short!

- Reducing them further is a sin.. but no choice
- Lose repeat resolution power
- Lose read coherence

# De Bruijn Specific

## Kmers

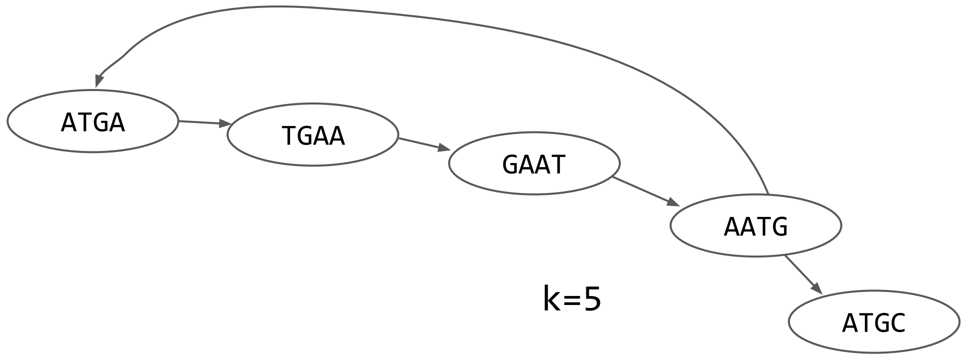
Short reads already short!

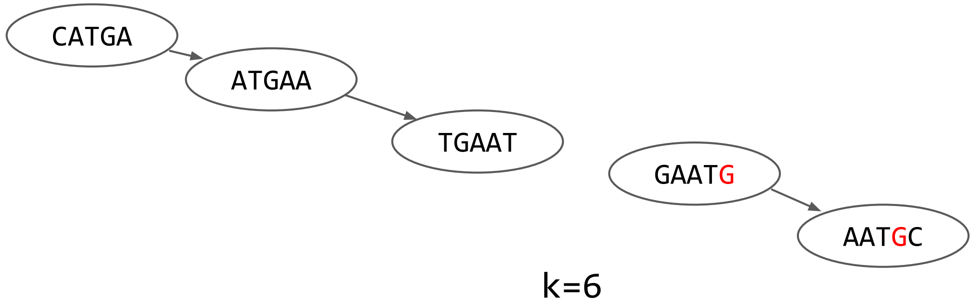
- Reducing them further is a sin.. but no choice
- Lose repeat resolution power
- Lose read coherence

Lower K: Fewer nodes, more edges

Higher K: More nodes, fewer edges

Balance these two forces. See more Lecture 8B.



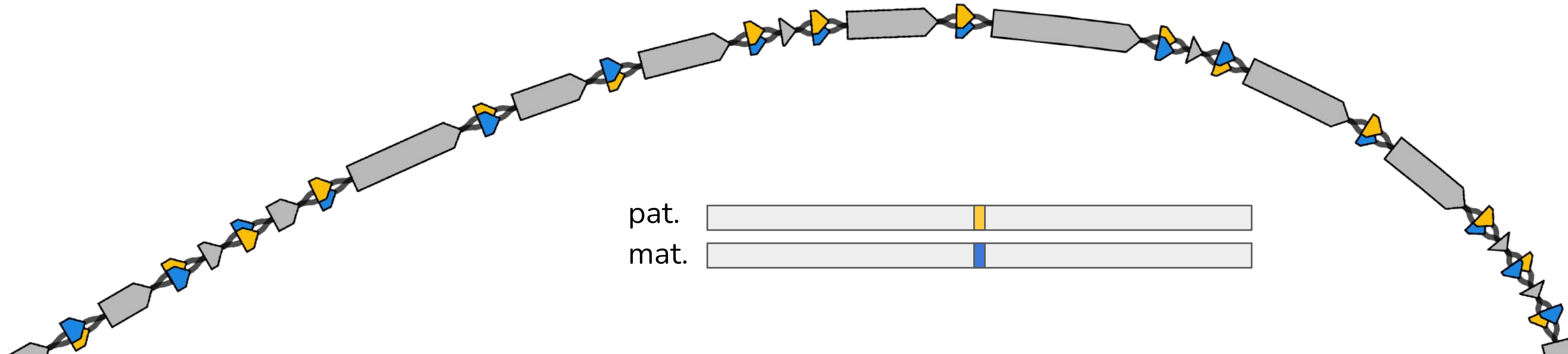


# De Bruijn Specific

Heterozygosity causes many parallel splits and joins（杂合性导致许多并行的分裂和连接）：
杂合性是指一个生物在某个基因座位上具有两个不同的等位基因。在 De Bruijn 图中，杂合性会引起问题，因为这意味着在某些位置，两套染色体上的序列是不同的。在图中，这可能导致多条并行的路径，因为每个等位基因都可能有自己的序列变体。这些并行的路径会在那些杂合的位置处分开（split）并最终在某处再次连接（join），这增加了图的复杂性，并使得从图中解析出一个准确的基因组序列变得更加困难

## Ploidy

Heterozygosity causes many parallel splits and joins.



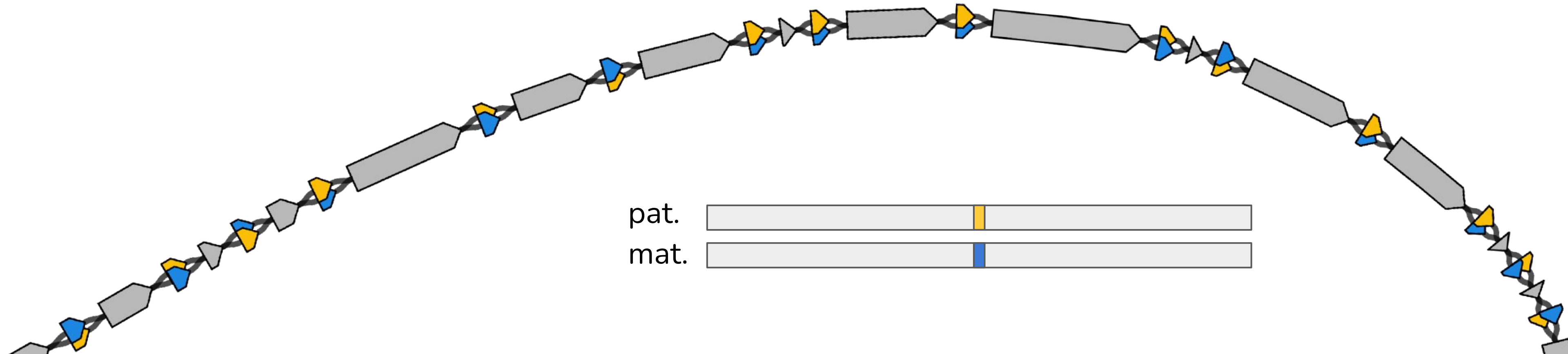
# De Bruijn Specific

## Ploidy

Heterozygosity causes many parallel splits and joins.

OLC: Reads from different strands overlap. Will be merged eventually.

De Bruijn: No notion of overlaps. Will need to correct these at some point...



# De Bruijn Specific

## Read Errors

Mainly affects De Bruijn Graphs

- OLC: alignment allows mismatches, gaps
- De Bruijn: use of kmers & exact matching

# De Bruijn Specific

Read Errors

Mainly affects De Bruijn Graphs

OLC: alignment allows mismatches, gaps

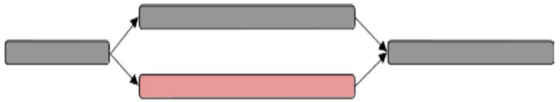
De Bruijn: use of kmers & exact matching

Middle of read: causes bubbles

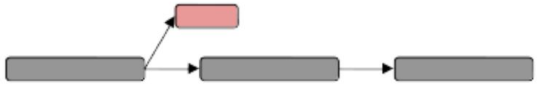
End of read: cause spurs

Use of kmers: causes erroneous edges

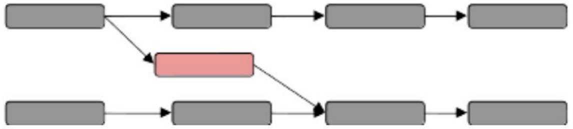
Bubbles



Spurs
(tips / dead ends)



Erroneous
edges



基因组组装时读取错误对两种主要组装策略——OLC (Overlap-Layout-Consensus) 和 De Bruijn 图——的影响

OLC: alignment allows mismatches, gaps（OLC：比对允许错配和缺口）：
在 OLC 策略中，reads 之间的比对允许存在一些错配和缺口。这意味着即使 reads 之间有些差异（可能由于读取错误或其他原因），它们仍然可以被识别为重叠。

De Bruijn: use of kmers & exact matching（De Bruijn：使用 k-mers 和精确匹配）：
De Bruijn 图策略是基于 k-mers 进行的，需要对 k-mers 进行精确匹配。这意味着读取中的任何小错误都会导致一个全新的 k-mer

Middle of read: causes bubbles（读取的中部：导致“气泡”）：
如果读取的中部出现错误，这在 De Bruijn 图中可能会形成一个被称为“气泡”的结构。气泡是由于两个轻微变异的序列路径导致的，它们在开始和结束时合并，但在中间有所不同。

End of read: cause spurs（读取的末端：导致“刺”）：
如果读取的末端出现错误，这可能会在 De Bruijn 图中形成一个“刺”。这是一个短的、通常是错误的序列分支。

Use of kmers: causes erroneous edges（使用 k-mers：导致错误的边）：
由于读取错误导致的错误 k-mers 会在 De Bruijn 图中引入错误的边，这可能会干扰正确的基因组组装。

## De Bruijn Specific

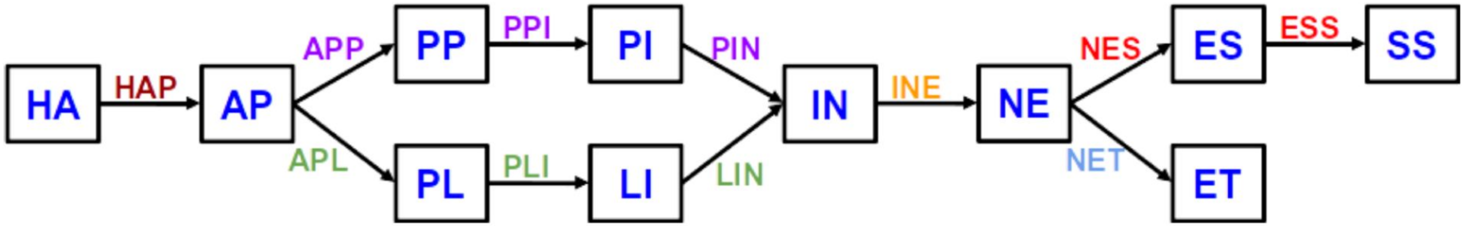
Read Errors

Example

```
HAPPI INESS APLIN PINET
                *     *
```

Unique 3-mers (*k* = 3):

```
HAP APP PPI
 INE NES ESS
  APL PLI LIN
    PIN NET
```


$$\begin{tikzcd}
 & & PP \arrow[r, "PPI"] & PI \arrow[dr, "PIN"] & & & ES \arrow[r, "ESS"] & SS \\
HA \arrow[r, "HAP"] & AP \arrow[ur, "APP"] \arrow[dr, "APL"'] & & & IN \arrow[r, "INE"] & NE \arrow[ur, "NES"] \arrow[dr, "NET"'] & & \\
 & & PL \arrow[r, "PLI"] & LI \arrow[ur, "LIN"'] & & & ET &
\end{tikzcd}$$

## Both - Repeats

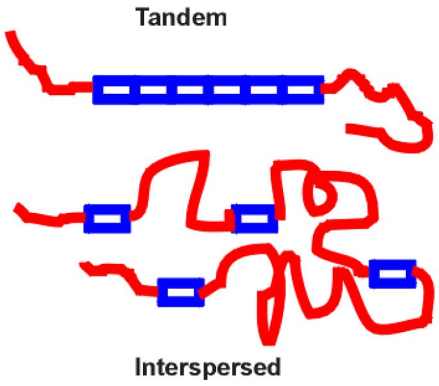
### Repeats

A segment of DNA that occurs more than once in the genome

Tandem repeats: Next to each other

Interspersed repeats: Have other sequence between repeats

Generally can't do much about repeats*



Repeats（重复序列）：
重复序列指的是在基因组中出现多次的 DNA 片段。因为它们的存在，基因组组装变得更加复杂和困难。

Tandem repeats（串联重复）：
串联重复指的是重复的 DNA 序列紧挨着彼此出现。也就是说，同一片段的 DNA 在基因组中一次又一次地连续出现。

Interspersed repeats（散在重复）：
散在重复是指重复的 DNA 片段之间有其他的 DNA 序列。这些重复不是连续的，而是在基因组的不同位置出现，它们之间有其他的、非重复的序列。

*Generally can’t do much about repeats（通常对于重复序列没什么办法）**:
这意味着，由于重复序列的存在，基因组组装变得非常困难。当组装软件遇到重复序列时，很难确定某个重复的片段是属于基因组的哪个位置，因为这个片段在多个地方都出现。这大大增加了组装过程中出错的可能性。因此，尽管科研人员已经开发了许多组装算法和策略，重复序列仍然是基因组组装中的一个主要挑战。

# Both - Repeats

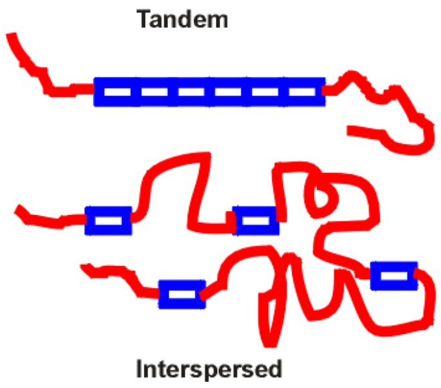
## Repeats

A segment of DNA that occurs more than once in the genome

Tandem repeats: Next to each other

Interspersed repeats: Have other sequence between repeats

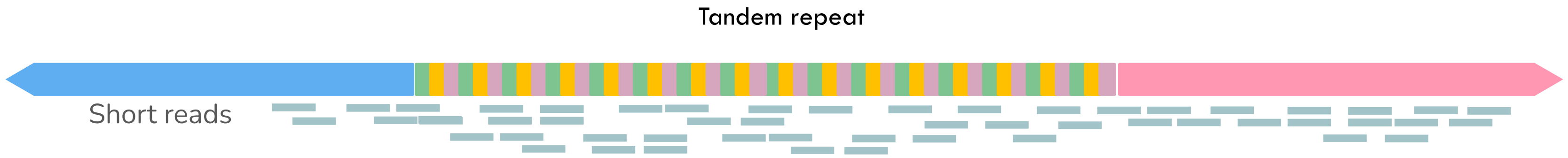
Generally can't do much about repeats*



| Repeat Class | Arrangement | Coverage (Hg) | Length (bp) |
|---|---|---|---|
| **Microsatellite** | Tandem | 3.00% | 2–100 |
| **SINE** | Interspersed | 15.00% | 100–300 |
| **Transposable elements** | Interspersed | 12.00% | 200–5k |
| **LINE** | Interspersed | 21.00% | 500–8k |
| **rDNA** | Tandem | 0.01% | 2k–43k |
| **Segmental Duplications** | Tandem or Interspersed | 0.20% | 1k–100k |

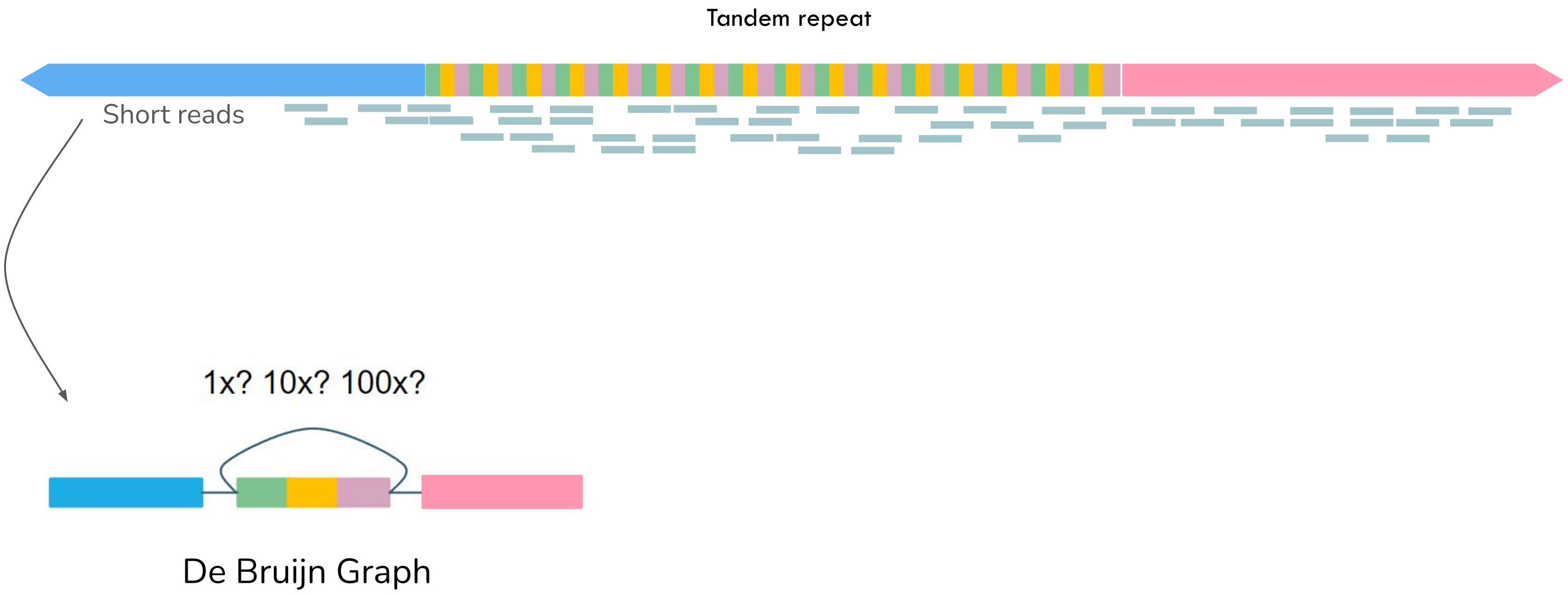
# Both - Repeats

## Tandem repeats



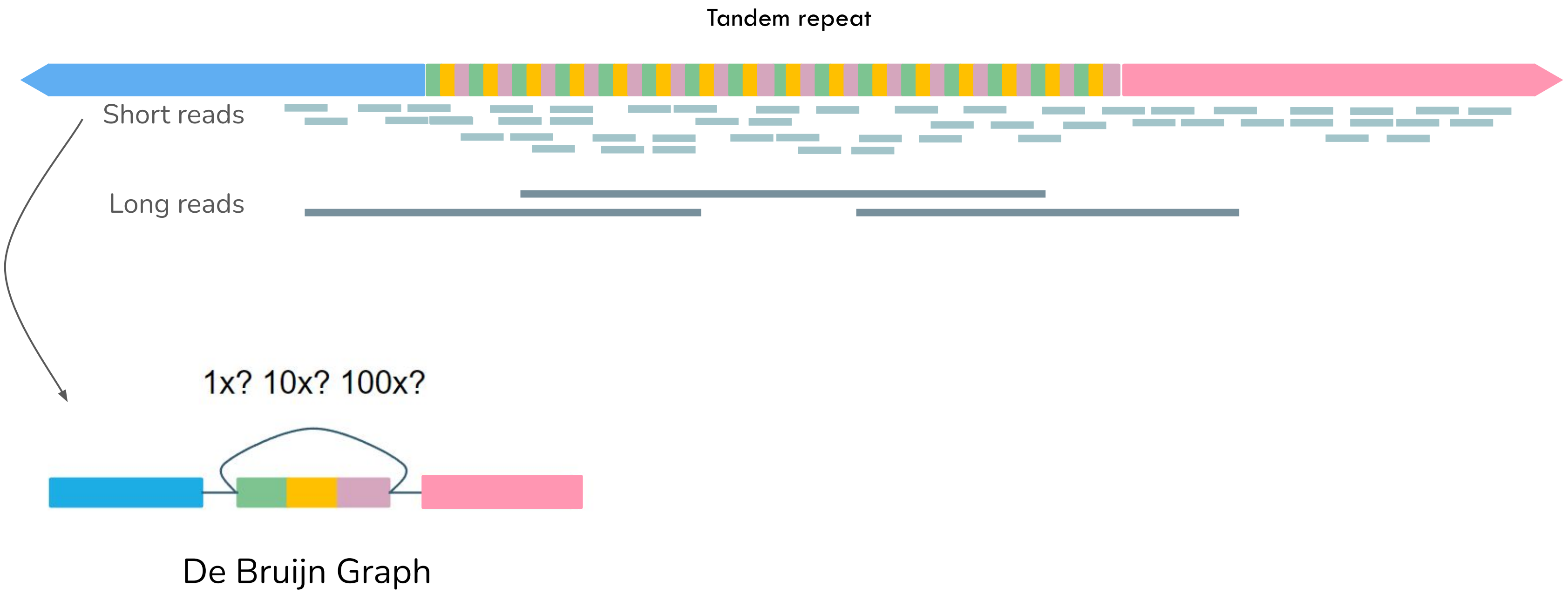
# Both - Repeats

## Tandem repeats



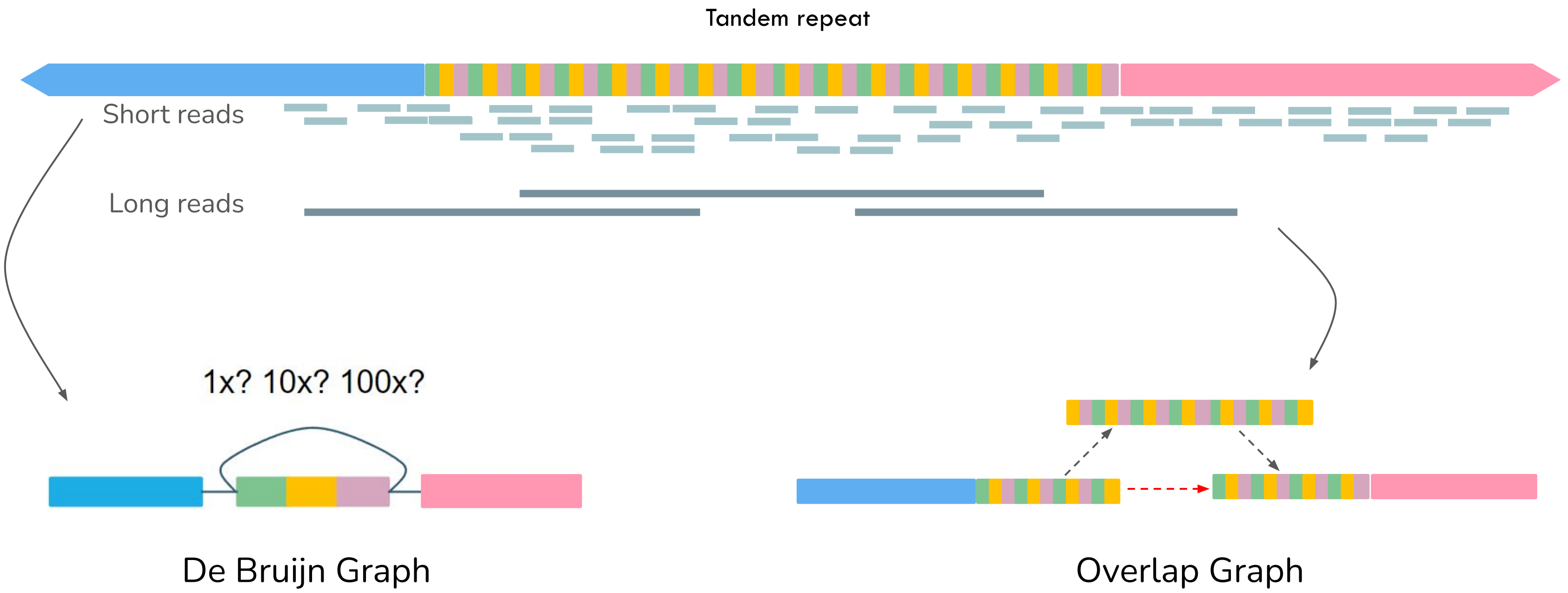
# Both - Repeats

## Tandem repeats



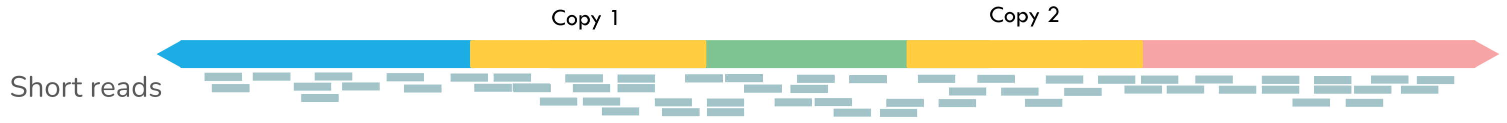
# Both - Repeats

## Tandem repeats



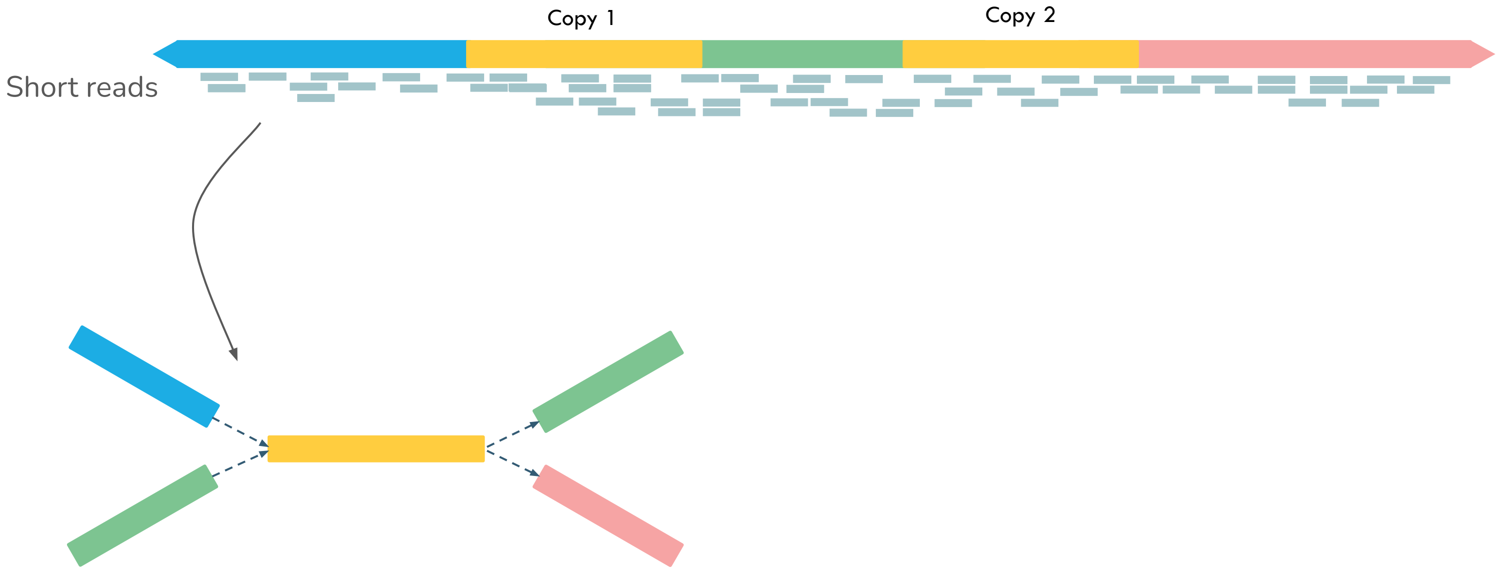
# Both - Repeats

Interspersed repeats



# Both - Repeats

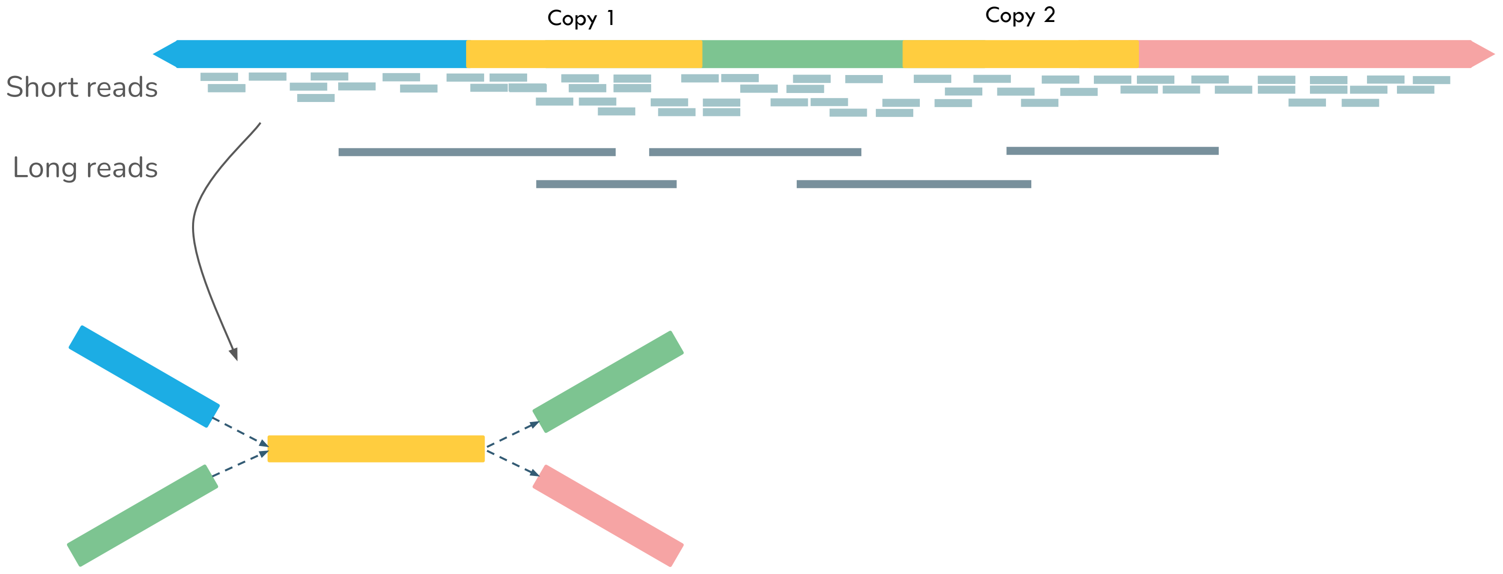
## Interspersed repeats



De Bruijn Graph

# Both - Repeats

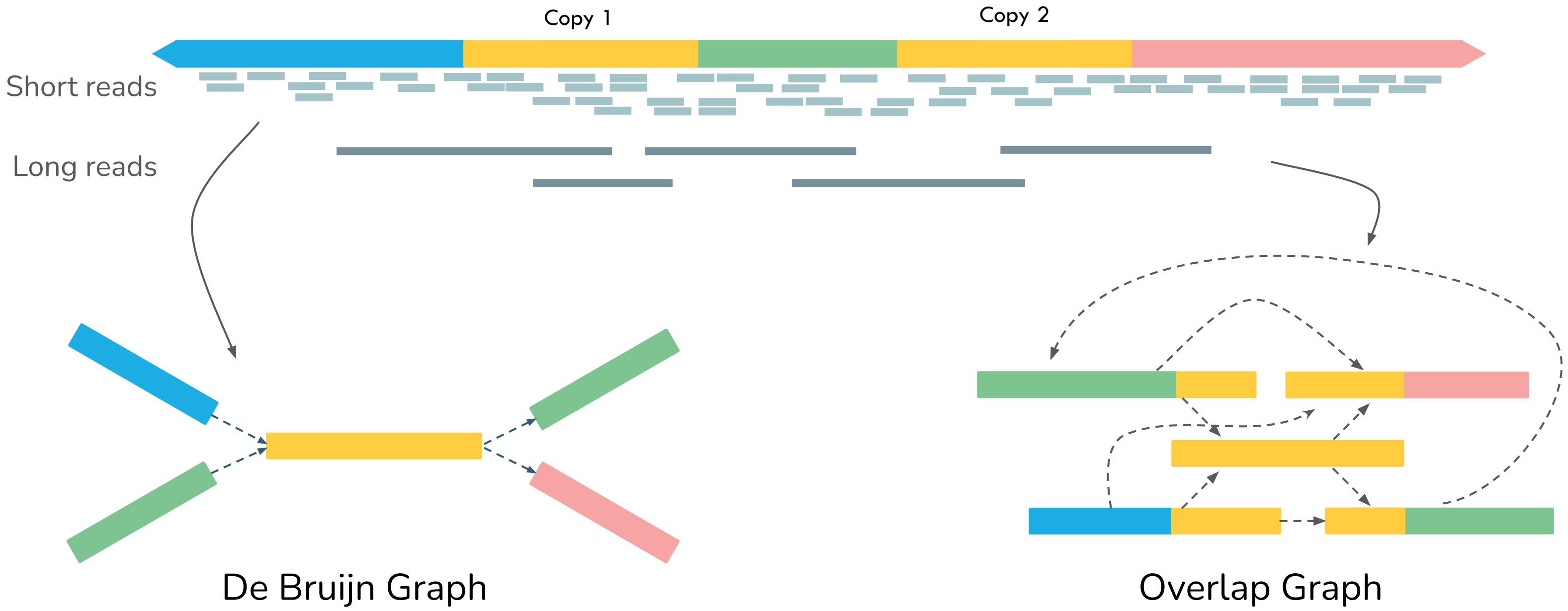
## Interspersed repeats



De Bruijn Graph

# Both - Repeats

## Interspersed repeats



## Both - Strandedness

*DNA and English sentences are just strings of characters*

## Both - Strandedness

*DNA and English sentences are just strings of characters*

5’ ATCGATCGTAAATGGC 3’
3’ TAGCTAGCATTTACCG 5’

DNA is double stranded

- 🍪 minus strand is the ***reverse complement*** of the plus strand

对于 OLC:

在 OLC 组装方法中，由于我们考虑整个 reads 的重叠，因此 reads 可能来自 DNA 的任一链。因此，为了正确地组装，必须考虑到 reads 可能来自的链（正链或反链）。
当考虑 reads 的 strandedness 时，可以更准确地确定 reads 之间的重叠，并减少由于忽略 strandedness 而产生的错误组装。
对于 De Bruijn 图:

在构建 De Bruijn 图时，使用的是 k-mers。由于考虑的是较短的序列片段，通常会同时考虑一个 k-mer 及其互补的反向 k-mer。
这意味着，尽管 De Bruijn 图的策略没有直接处理整个 reads 的方向性，但在处理 k-mers 时，它实际上已经考虑了 strandedness

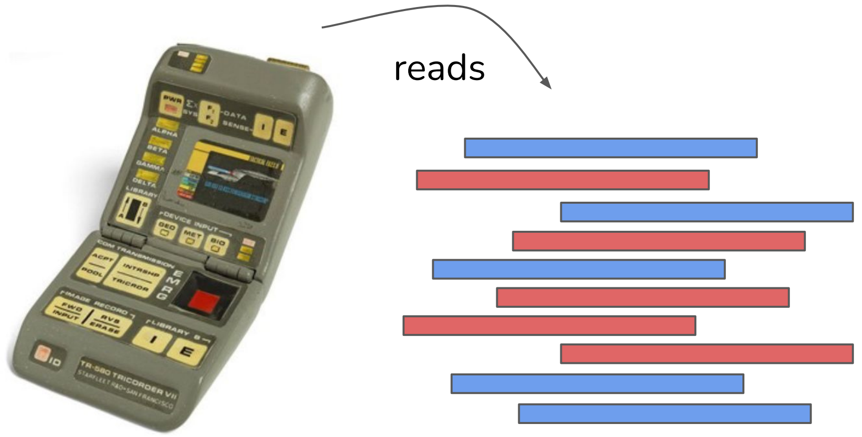
# Both - Strandedness

*DNA and English sentences are just strings of characters*

5’ ATCGATCGTAAATGGC 3’
3’ TAGCTAGCATTTACCG 5’

DNA is double stranded

- minus strand is the ***reverse complement*** of the plus strand
- (most) sequencing is ***NOT STRAND SPECIFIC***
- how do we deal with this in the graph?



# Both - Strandedness - OLC

Overlap graphs (seen in OLC) use additional data on the edges.

# Both - Strandedness - OLC

Overlap graphs (seen in OLC) use additional data on the edges.

Reads:

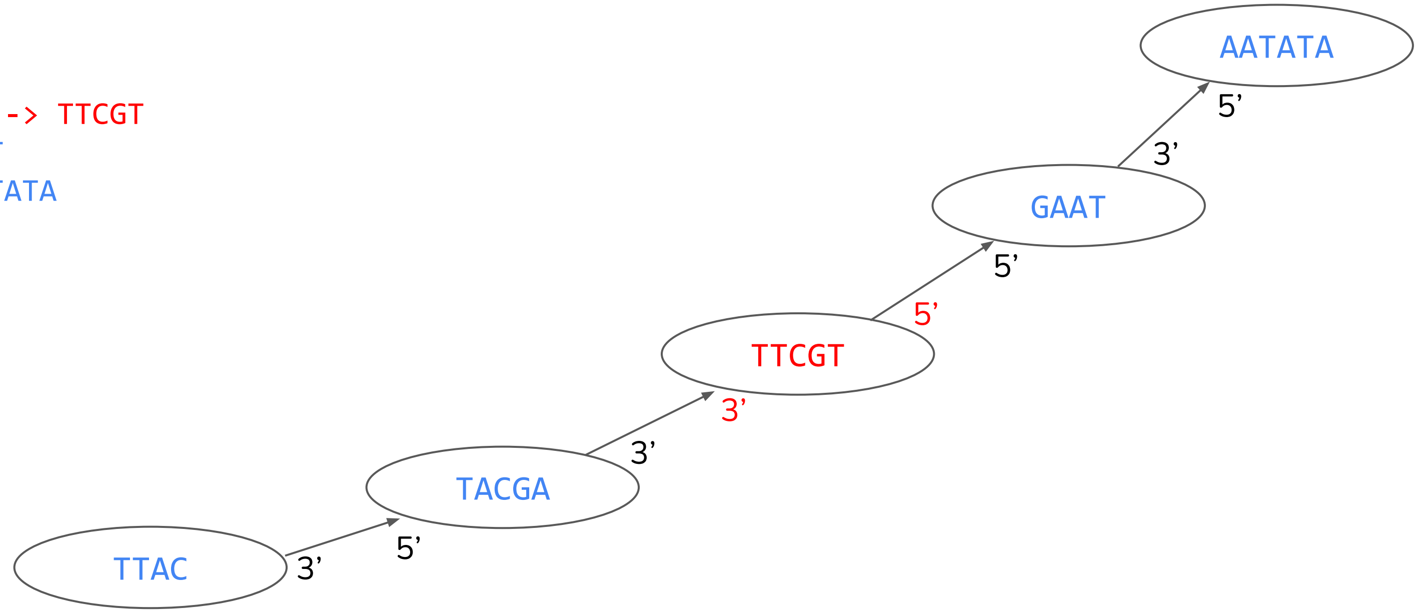
```
(f) TTAC
(f)  TACGA
(r)   ACGAA -> TTCGT
(f)     GAAT
(f)      AATATA
```

# Both - Strandedness - OLC

Overlap graphs (seen in OLC) use additional data on the edges.

Reads:

```
(f) TTAC
(f)  TACGA
(r)   ACGAA -> TTCGT
(f)     GAAT
(f)      AATATA
```

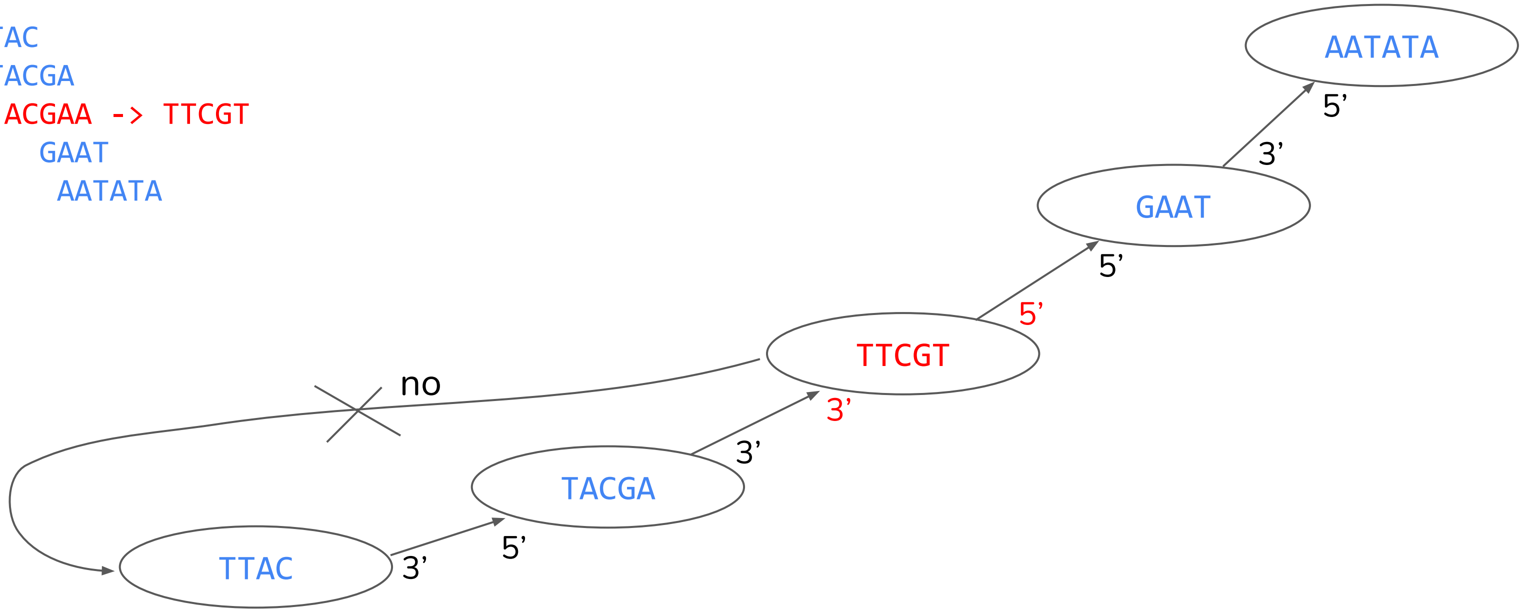


# Both - Strandedness - OLC

Overlap graphs (seen in OLC) use additional data on the edges.

Reads:

```
(f) TTAC
(f)  TACGA
(r)   ACGAA -> TTCGT
(f)     GAAT
(f)      AATATA
```

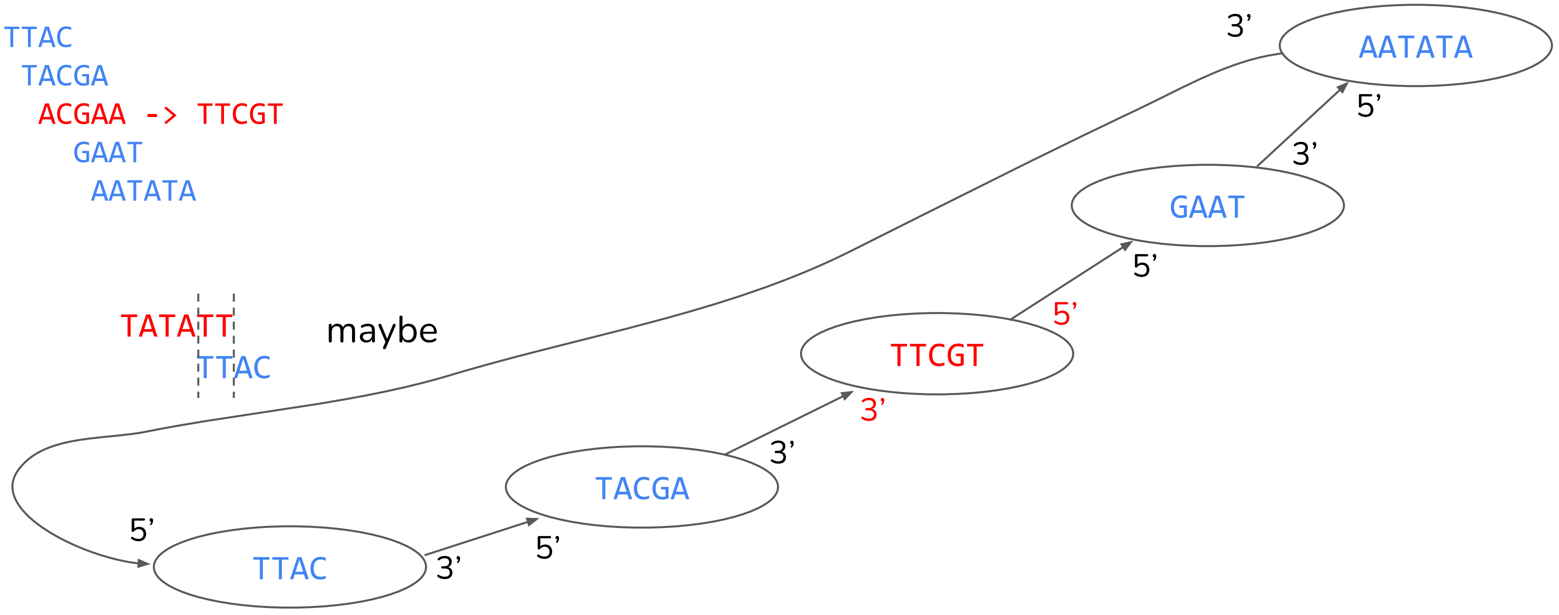


# Both - Strandedness - OLC

Overlap graphs (seen in OLC) use additional data on the edges.

Reads:

```
(f) TTAC
(f)  TACGA
(r)   ACGAA -> TTCGT
(f)     GAAT
(f)      AATATA
```



## Both - Strandedness - De Bruijn

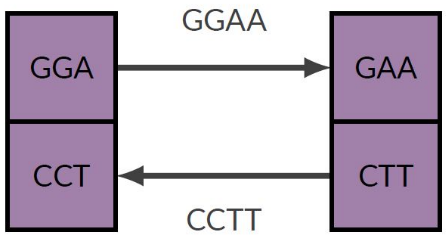
De Bruijn Graphs directly add the reverse-complemented kmers to the graph

# Both - Strandedness - De Bruijn

De Bruijn Graphs directly add the reverse-complemented kmers to the graph

1. Take the first *k*-mer
2. Split into prefix/suffix
3. Add them as nodes
4. Add the reverse complement to nodes
5. Add *k*-mers (forwards and reverse complement) as edges
6. Repeat for the rest of the *k*-mers

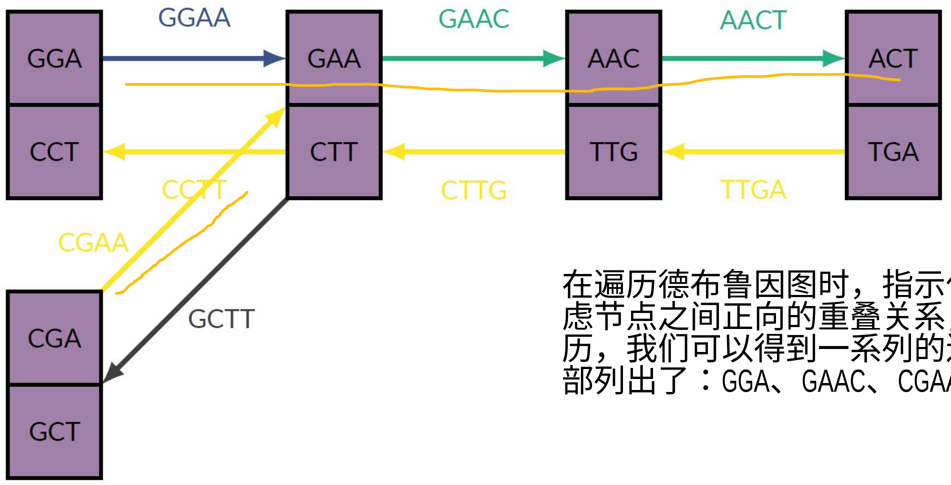
```
GGAA   GAAC   AACT   TTCC   TTCG   GTTC   AGTT
^^^^
```



# Both - Strandedness - De Bruijn

De Bruijn Graphs directly add the reverse-complemented kmers to the graph

```
GGAA   GAAC   AACT   TTCC   TTCG   GTTC   AGTT
```



在遍历德布鲁因图时，指示仅在前向方向移动。这意味着我们只考虑节点之间正向的重叠关系，而不是反向互补的连接。根据图的遍历，我们可以得到一系列的连续序列（contigs），这在幻灯片的底部列出了：GGA、GAAC、CGAA。

Traversing the graph: ***move in the forwards direction only***

Contigs: GGAA  GAACT  CGAA

## Summary

Our graphs get very complex! Need to simplify.

# Summary

Our graphs get very complex! Need to simplify.

Overlap Graphs (seen in OLC)

Unnecessary edges

- Intrinsic to overlap graphs

Graph has spurs

- Due to sequencing errors chimeric sequences (repeats)

## Summary

Our graphs get very complex! Need to simplify.

Overlap Graphs (seen in OLC)

Unnecessary edges

- Intrinsic to overlap graphs

Graph has spurs

- Due to sequencing errors chimeric sequences (repeats)

De Bruijn Graphs

Graph has bubbles

- Read errors (middle of read), heterozygosity

Graph has spurs

- Read errors (end of read)

Graph has erroneous edges

- Use of kmers

# Assembly Algorithms

Recap

OLC vs De Bruijn Assembly

Considerations

OLC Simplification

## Overlap Graph Simplification

Want a hamiltonian path

...but overlap graphs have many edges & dead ends

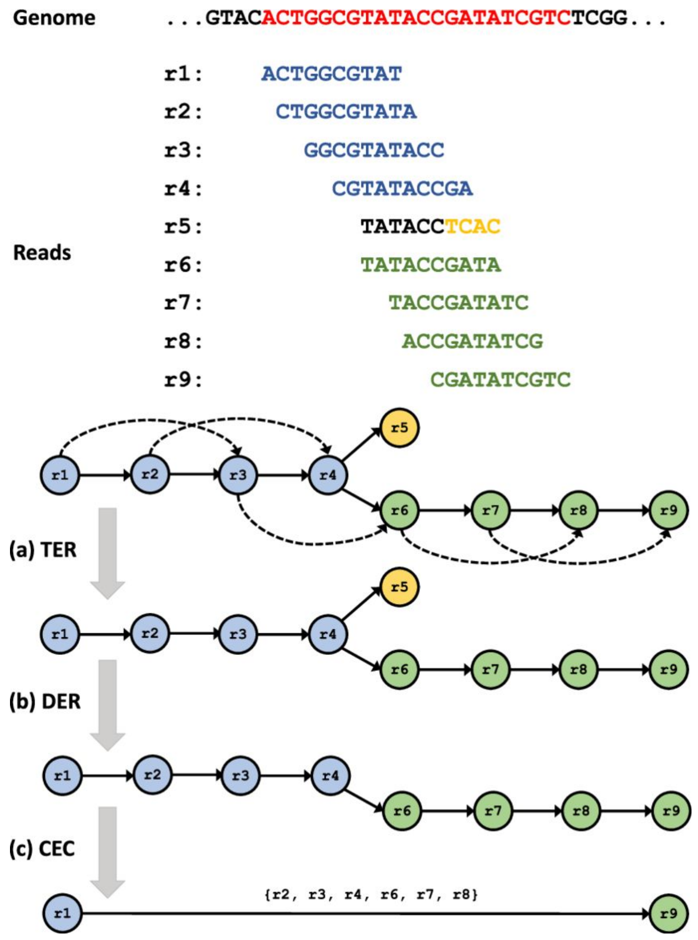
# Overlap Graph Simplification

Want a hamiltonian path

…but overlap graphs have many edges & dead ends

Transitive Edge Reduction (TER)

- Remove unnecessary edges



Paul et al (2019)

# Overlap Graph Simplification

Want a hamiltonian path

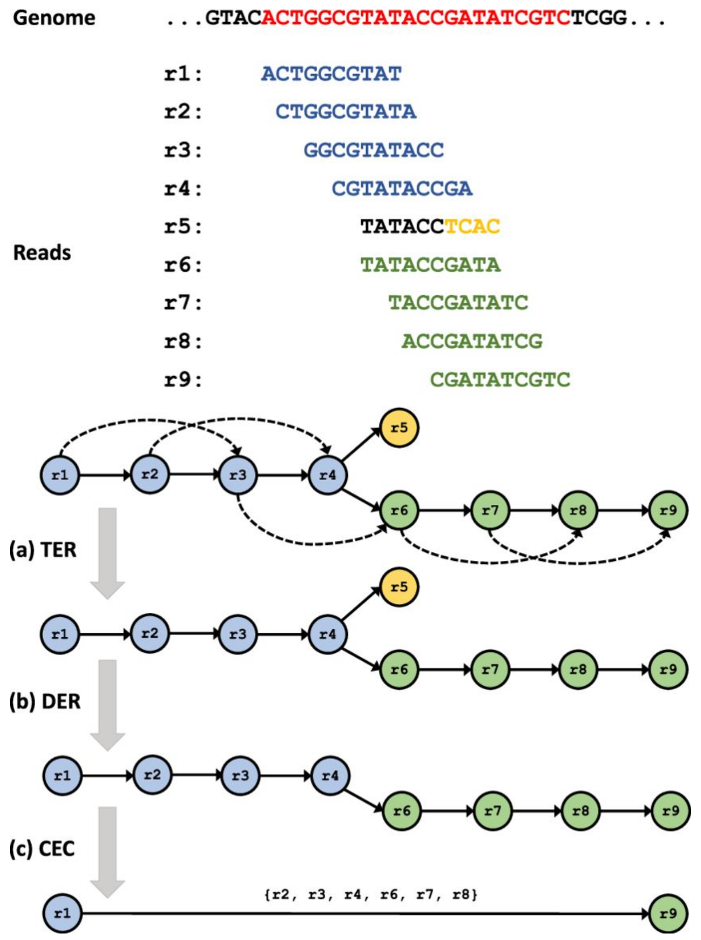
...but overlap graphs have many edges & dead ends

Transitive Edge Reduction (TER)

- Remove unnecessary edges

Dead-End Removal (DER)

- Remove short spurs / dead ends



Paul et al (2019)

# Overlap Graph Simplification

Want a hamiltonian path

...but overlap graphs have many edges & dead ends

Transitive Edge Reduction (TER)

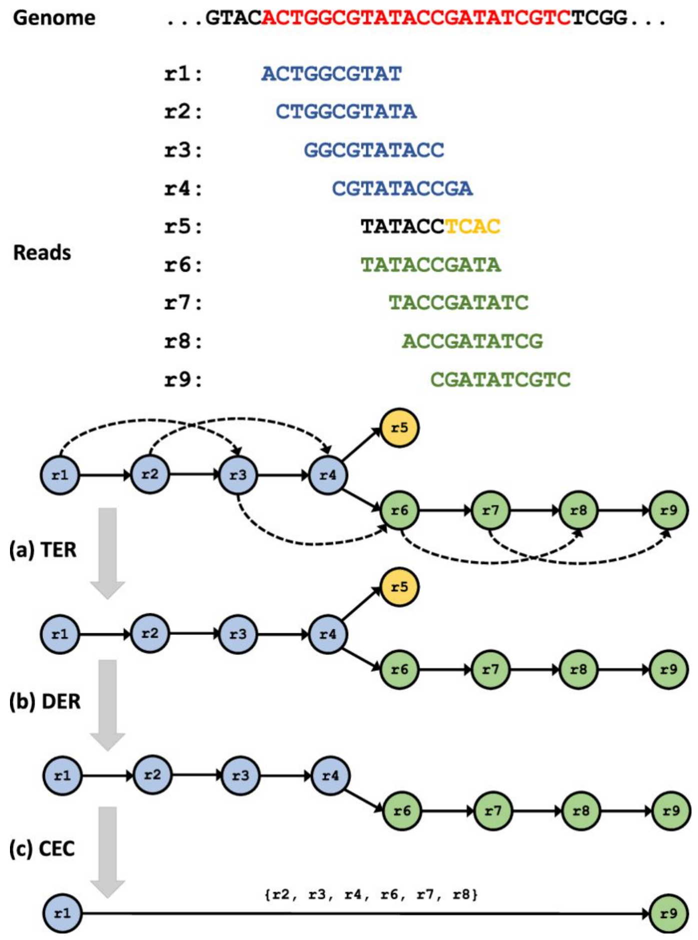
- Remove unnecessary edges

Dead-End Removal (DER)

- Remove short spurs / dead ends

Composite Edge Contraction(CEC)

- Merges nodes in manner which does not lose information
- Quite complex. Not covering this.



Paul et al (2019)

Want a hamiltonian path（想要一个哈密尔顿路径）：
哈密尔顿路径是图论中的概念，它指的是一个在图中经过每个顶点恰好一次的路径。在基因组组装的上下文中，这意味着找到一个表示完整基因组序列的路径，其中每个读取（read）恰好被使用一次。

…but overlap graphs have many edges & dead ends（但重叠图中有很多边和死胡同）：
由于每个读取可能与许多其他读取重叠，因此在基因组组装中，重叠图往往非常复杂，有很多边。这使得找到哈密尔顿路径变得困难。

Transitive Edge Reduction (TER)（传递性边缘简化）：
这是一种策略，旨在删除那些不必要的、可以通过其他边间接表示的边。这样可以简化图并使其更容易理解和处理。

Remove unnecessary edges（移除不必要的边）：
通过 TER 方法，我们可以移除那些不必要的、可以通过其他路径间接表示的边。

Dead-End Removal (DER)（死胡同移除）：
这是另一种策略，旨在删除图中的短刺和死胡同。这些通常是由于读取错误或重复序列产生的，对组装并不有帮助。

Composite Edge Contraction (CEC)（复合边缘缩减）：
这种方法的目的是合并节点，但在这个过程中不失去关于这些节点的重要信息。这是一种将图进一步简化的策略。

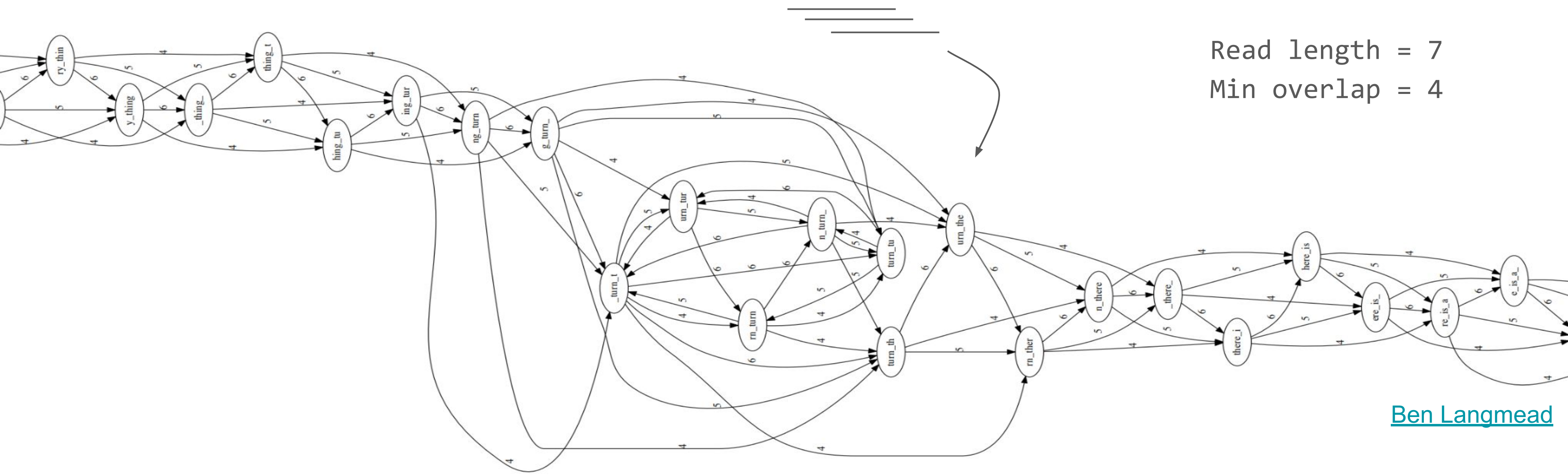
# Transitive Edge Reduction

Overlap graphs inherently messy.

Each read overlaps with many other reads.

```
to_every_thing_turn_turn_turn_there_is_a_season
```

```
Read length = 7
Min overlap = 4
```

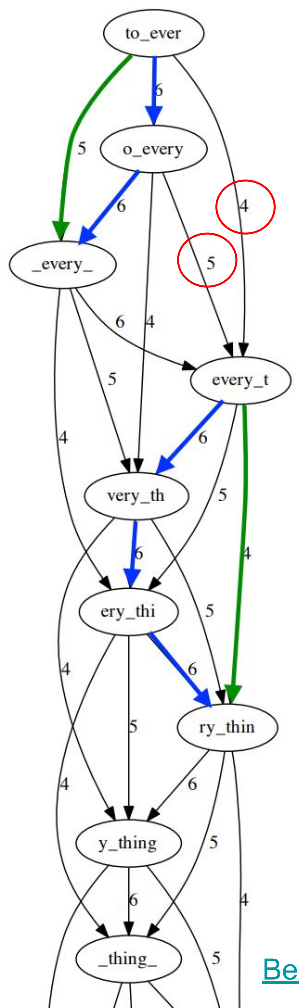


Ben Langmead

# Transitive Edge Reduction

Luckily, many of the overlaps are redundant and can be removed.

Call these edges ***transitive***



[Ben Langmead](#)

# Transitive Edge Reduction

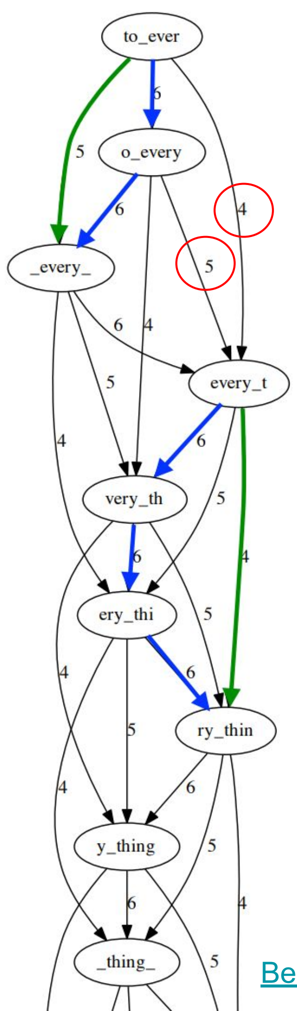
Luckily, many of the overlaps are redundant and can be removed.

Call these edges ***transitive***

Transitive edges can be inferred from other edges.

In the figure, **green** edges can be inferred from **blue** edges.

Know which path to walk when removing these due to the **overlap lengths**



Ben Langmead

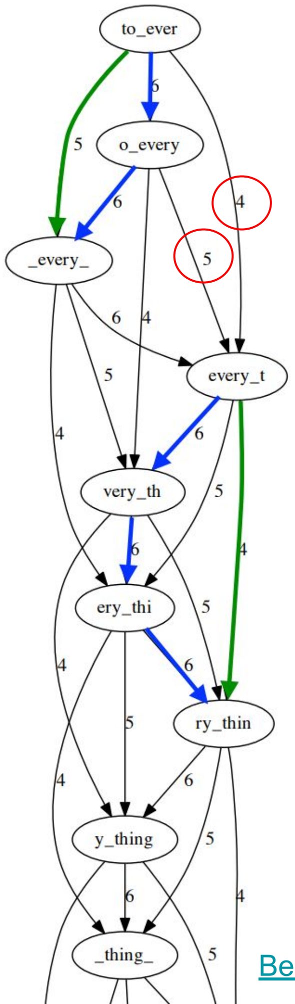
# Transitive Edge Reduction

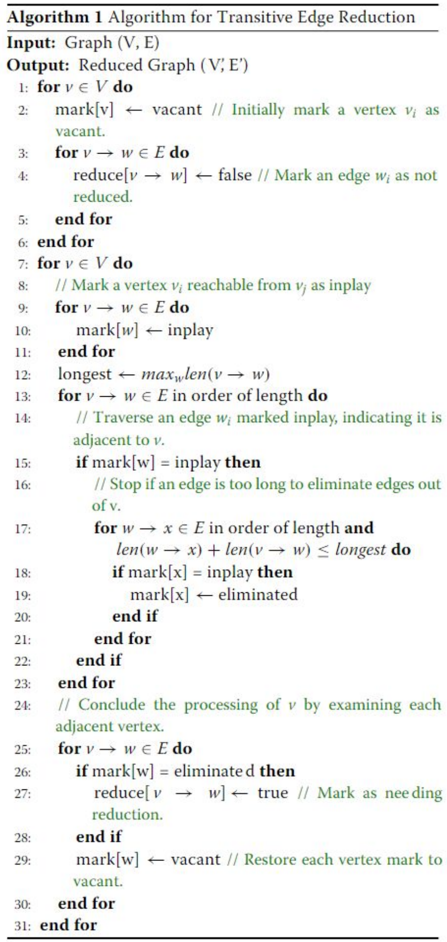
Luckily, many of the overlaps are redundant and can be removed.

- Call these edges ***transitive***

Transitive edges can be inferred from other edges.

- In the figure, **green** edges can be inferred from **blue** edges.
- Know which path to walk when removing these due to the **overlap lengths**





**Algorithm 1** Algorithm for Transitive Edge Reduction

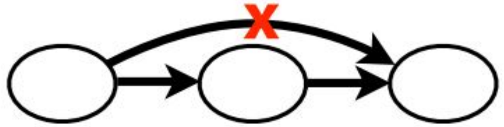
```
Input: Graph (V, E)
Output: Reduced Graph (V', E')
1:  for $v \in V$ do
2:      mark[v] ← vacant // Initially mark a vertex $v_i$ as vacant.
3:      for $v \to w \in E$ do
4:          reduce[$v \to w$] ← false // Mark an edge $w_i$ as not reduced.
5:      end for
6:  end for
7:  for $v \in V$ do
8:      // Mark a vertex $v_i$ reachable from $v_j$ as inplay
9:      for $v \to w \in E$ do
10:         mark[$w$] ← inplay
11:     end for
12:     longest ← $max_w len(v \to w)$
13:     for $v \to w \in E$ in order of length do
14:         // Traverse an edge $w_i$ marked inplay, indicating it is adjacent to $v$.
15:         if mark[w] = inplay then
16:             // Stop if an edge is too long to eliminate edges out of v.
17:             for $w \to x \in E$ in order of length and $len(w \to x) + len(v \to w) \le longest$ do
18:                 if mark[x] = inplay then
19:                     mark[x] ← eliminated
20:                 end if
21:             end for
22:         end if
23:     end for
24:     // Conclude the processing of $v$ by examining each adjacent vertex.
25:     for $v \to w \in E$ do
26:         if mark[w] = eliminate d then
27:             reduce[ $v \to w$] ← true // Mark as nee ding reduction.
28:         end if
29:         mark[w] ← vacant // Restore each vertex mark to vacant.
30:     end for
31: end for
```

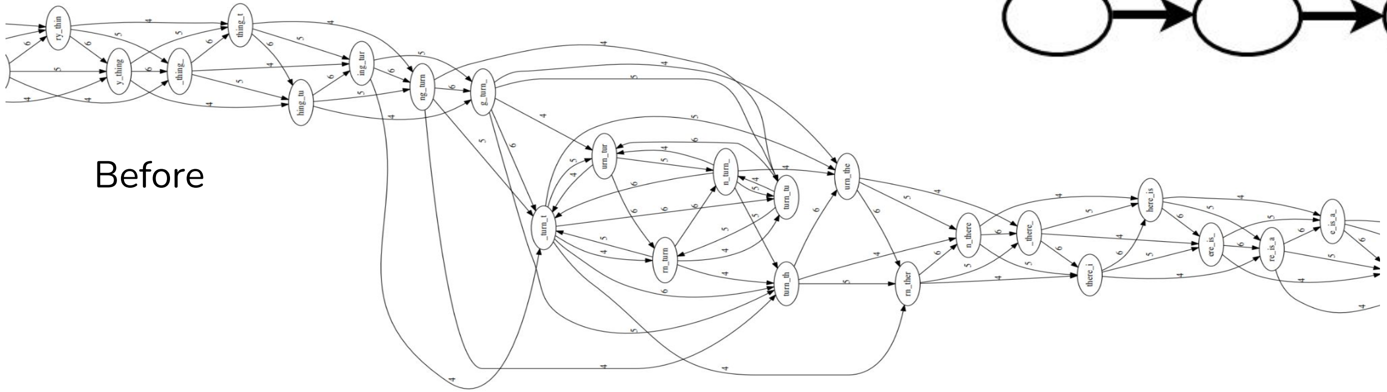
Paul et al (2019)

Ben Langmead

## Transitive Edge Reduction

Process: (1) Remove edges which skip one node



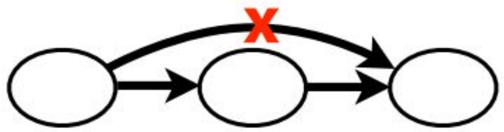


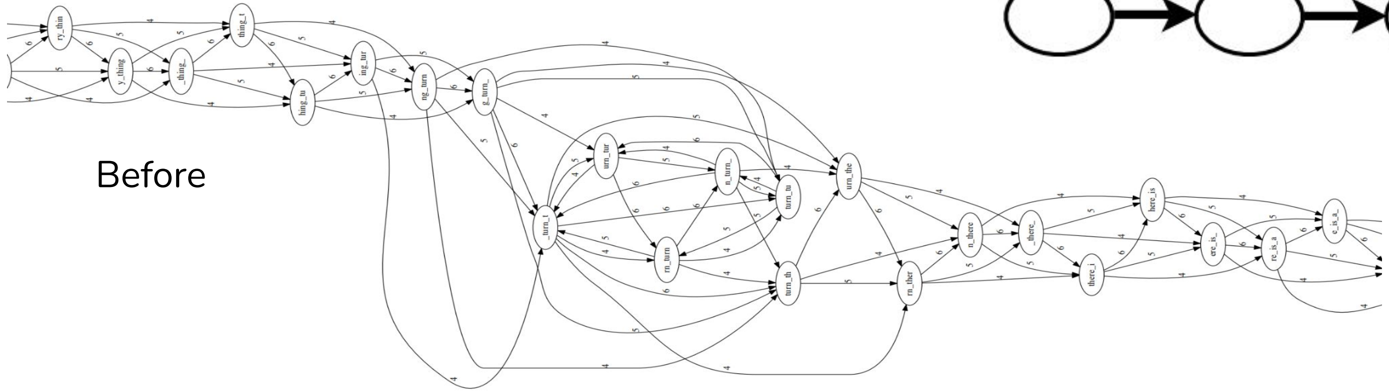
Transitive Edge Reduction (TER)（传递边缘简化）：
为了简化重叠图并使其更容易处理，可以采用传递边缘简化技术。这种方法的核心思想是删除那些不必要的、可以由其他边间接表示的边。

Remove unnecessary edges（移除不必要的边）：
通过 TER 方法，我们可以确定并删除那些不必要的边。例如，如果有三个 reads A、B 和 C，并且 A 重叠于 B，B 重叠于 C，那么如果 A 也直接重叠于 C，这个直接的 A 到 C 的边就可能是多余的，因为可以通过 A 到 B 到 C 的路径来表示

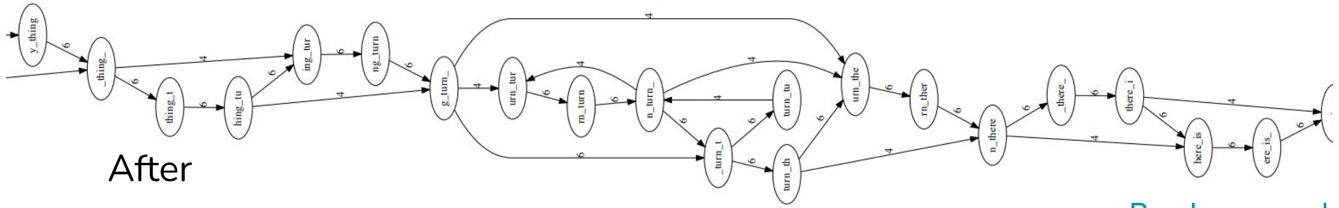
# Transitive Edge Reduction

Process: (1) Remove edges which skip one node





Before

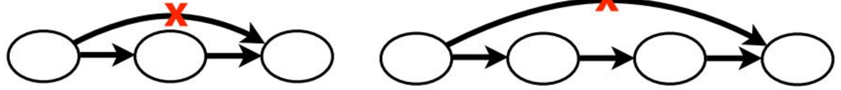


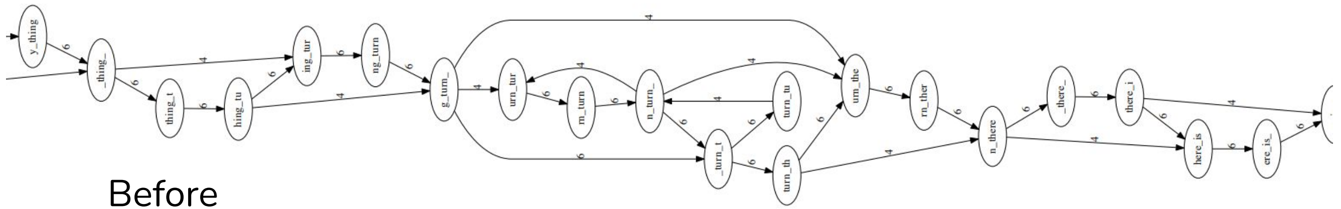
After

Ben Langmead

# Transitive Edge Reduction

Process: (2) Remove edges which skip one or two nodes

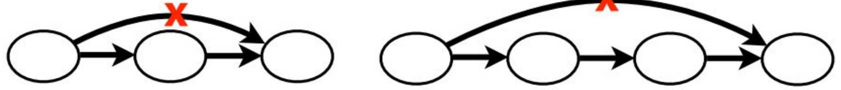


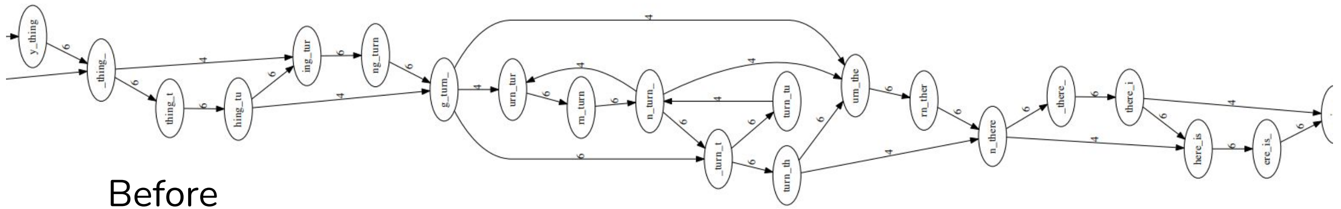


Before

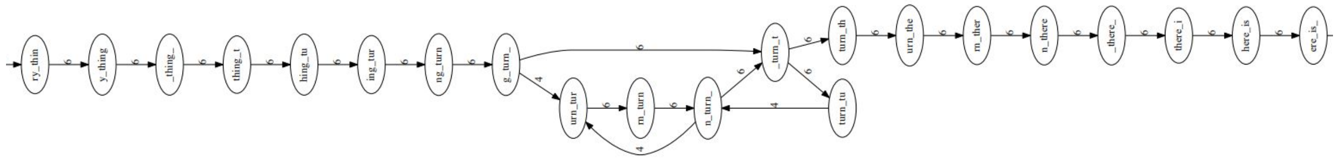
# Transitive Edge Reduction

Process: (2) Remove edges which skip one or two nodes





Before



After

Ben Langmead

# Dead-End Removal (DER)

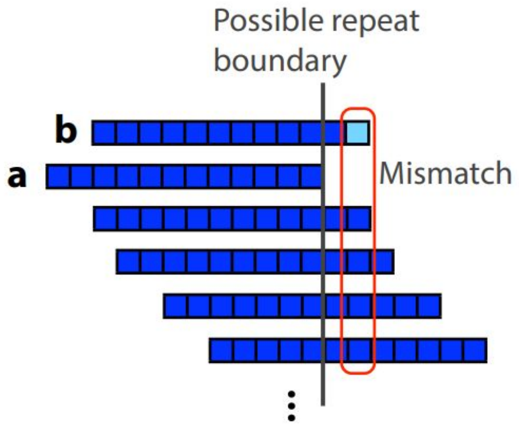
Remove short spurs / dead ends

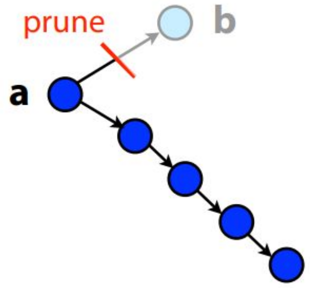
Caused by sequencing errors

Caused by overlapping of chimeric sequences (repeats)

Simple to remove

- Identify, then prune
- Short length edges
- Low coverage (depth)





Ben Langmead

Melbourne Bioinformatics

BIOINFORMATICS + DATA SERVICES + INFRASTRUCTURE, FOR LIFE SCIENCES TODAY

THE UNIVERSITY OF MELBOURNE

## Thank you!

**Today:** Considerations & Graph Simplification

**Next time:** Assembly in Practice