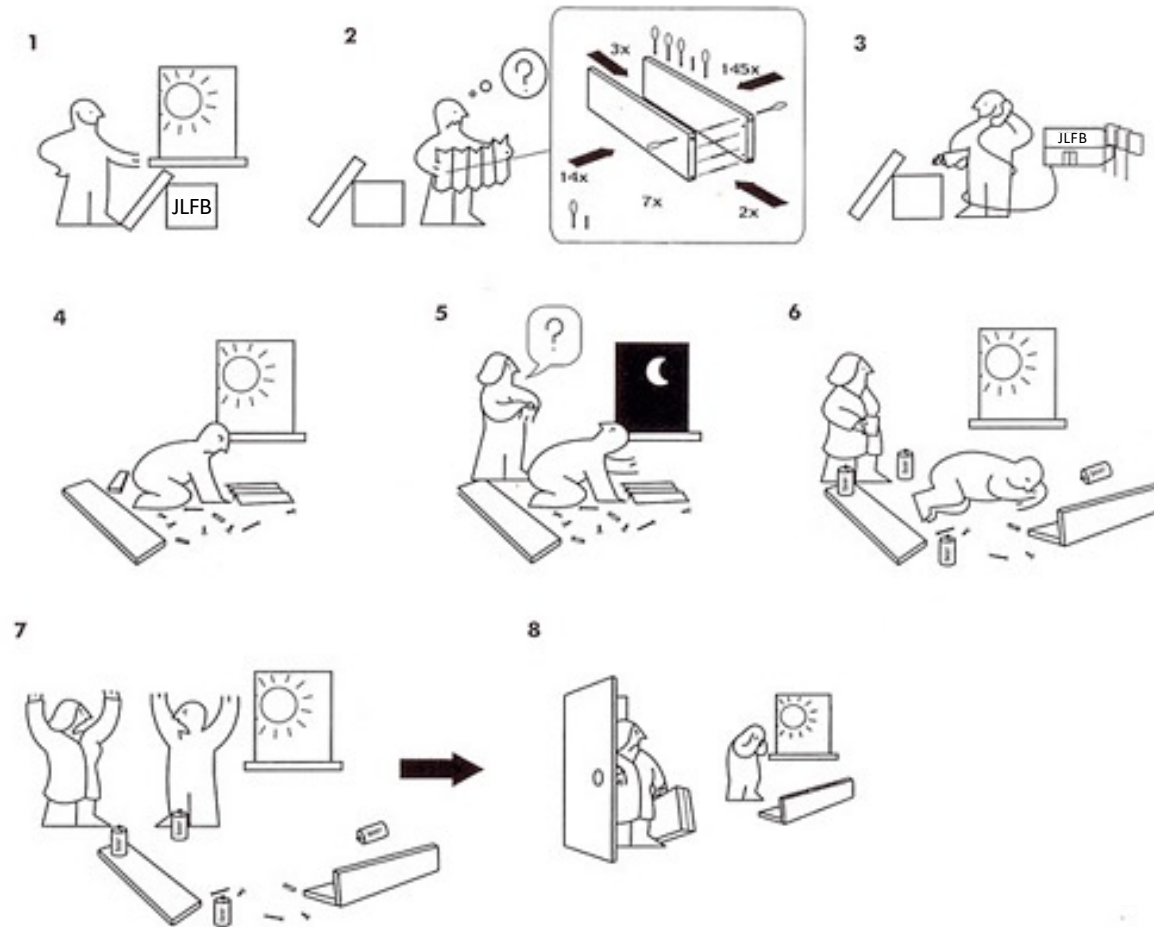


Algorithms in Sequence Analysis 4

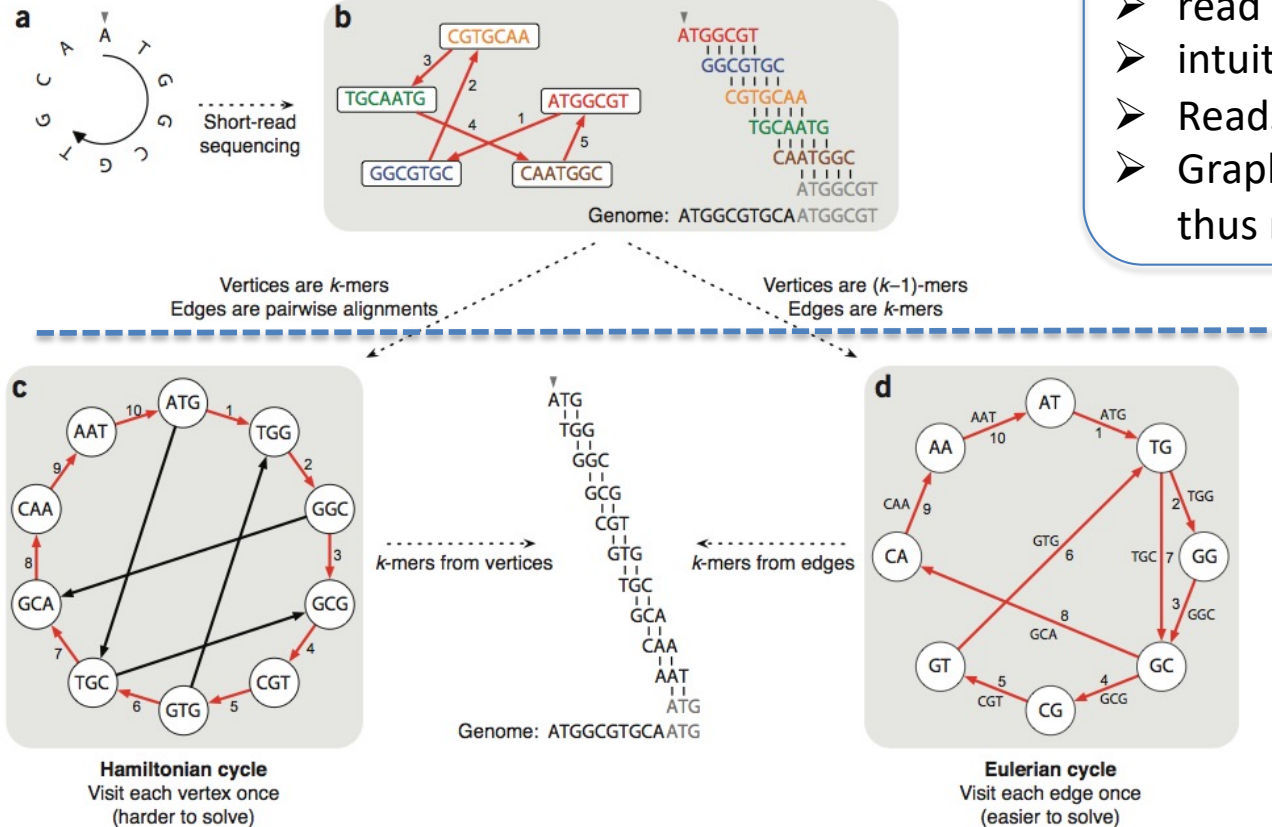


The assembly problem...

Different approaches to the sequence assembly problem

Overlap based assembly

- read identity is maintained
- intuitive
- Reads can be organized in an overlap graph
- Graph complexity increases with coverage, thus read redundancy inflates the graph



Kmer approaches

- read identity is (temporarily) lost...
- Reads are organized in deBruijn graphs
- Graph complexity depends on Kmer size
- Graph complexity is (by and large) independent from coverage, read redundancy is naturally handled
- repeats are represented only once in the graph with explicit links to the different start and end points

Genome assembly abstracted to the problem of finding a shortest superstring

0000110010111101

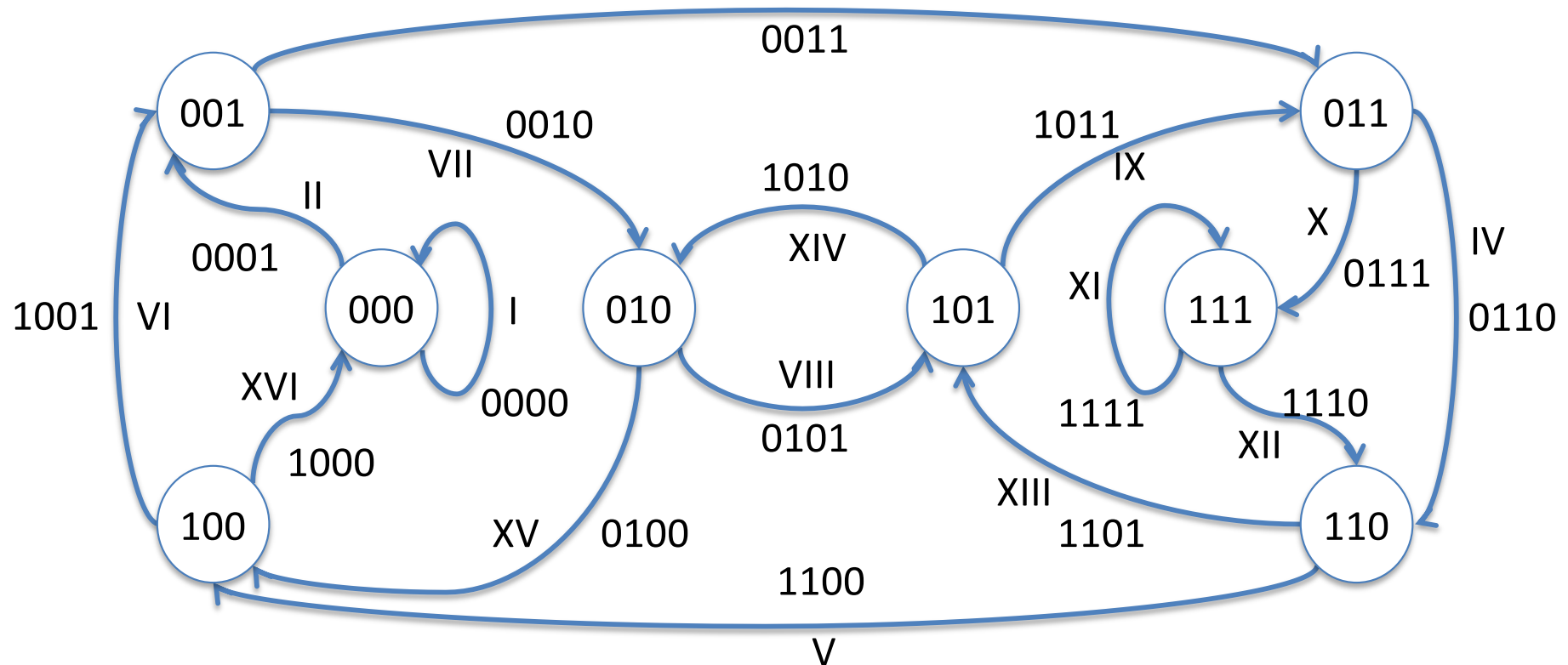
↓ Extract all words

0000, 0001, 0011, 0110, 1100, 1001, 0010, 0101, 1011, 0111, 1111, 1110, 1101, 1010, 0100, 1000

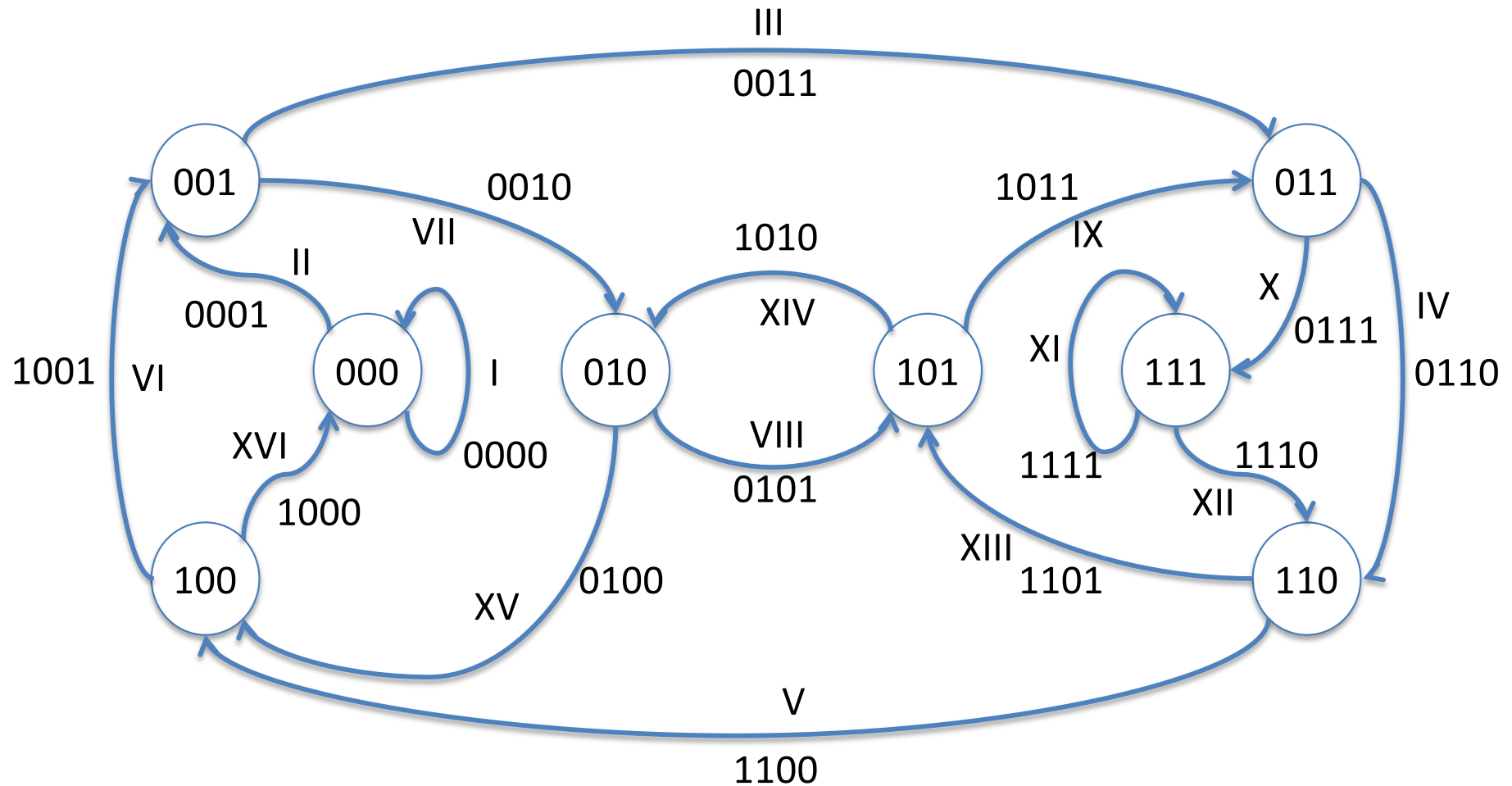
↓ Extract all k-1 words and use as nodes

↓ Connect two nodes if they form an observed kmer

III Find the Eulerian cycle/path through the graph



Passing through the edges by following the roman numbers reconstructs the superstring using each word exactly once!



I: 0000, II: 0001, III: 0011; IV: 0110; V: 1100; VI: 1001; VII: 0010; VIII: 0101; IX: 1011; X: 0111; XI: 1111;
 XII: 1110; XIII: 1101; XIV: 1010; XV: 0100; XVI: 1000

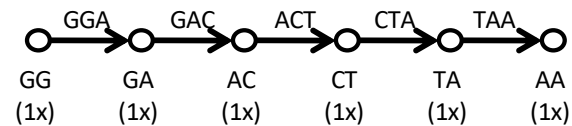
0000110010111101

De Bruijn Graph Example

Shred reads into k-mers (k = 3)

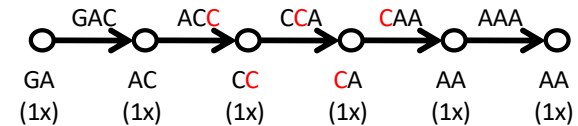
Read 1

G G A C T A A
G G A
G A C
A C T
C T A
T A A



Read 2

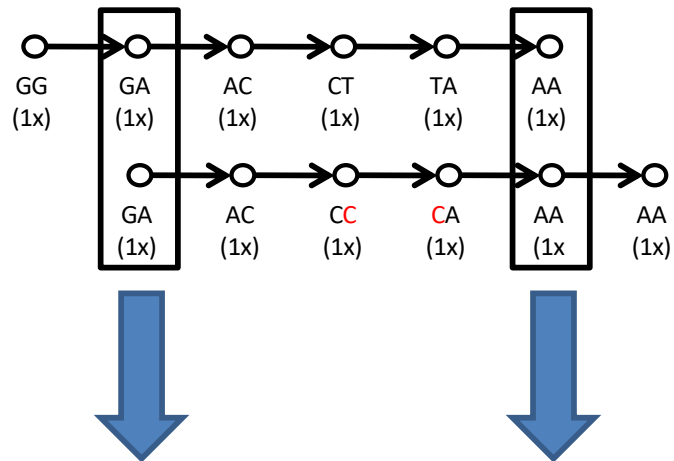
G A C C A A A
G A C
A C C
C C A
C A A
A A A



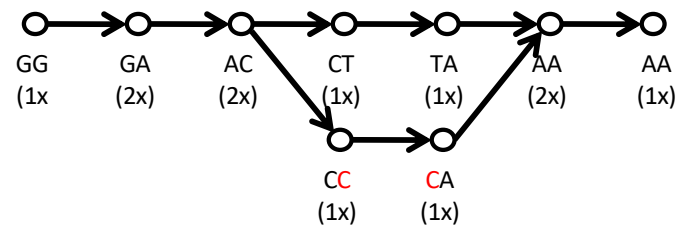
De Bruijn Graph Example

Merge vertices labeled by identical (k-1)-mers

Read 1:



Resulting Graph:



Basic concepts of de Bruijn graph based assemblers

- The sequence is treated as a consecutive string of words of length K
- Sequence reads are no longer considered to represent a consecutive string of nucleotides. Thus read length as well as read overlap become, in principle, irrelevant¹
- Sequence reads are only used to identify words of length K occurring in the sequence²
- Given perfect data – error-free K -mers providing full coverage³ and spanning every repeat – the K -mer graph would be a de Bruijn graph and it would contain an Eulerian path, that is, a path that traverses each edge exactly once

¹ This is of course not entirely true. Make sure to understand where the overlap matters

² This is only true for the graph construction. When it comes to finding the path through the graph, read identity does matter

³ every K -mer that occurs in the original sequence is represented in the K -mer list extracted from the reads

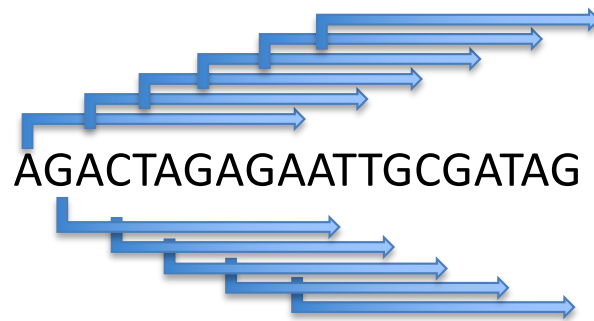
The magic '*Kmer*' gives most users of graph based assembly algorithms a very hard time as they have to decide on the size of K .



To give an informed statement we need to make sure to understand what K should represent and what the algorithmic requirements of de Bruijn graph assemblers are



Considering the size of K



A sequence of length 20 contains 11 different words of length 10!

Now, consider the sequence is spanned by 2 reads of length 13:

T: AGACTAGAGAATTGCGATAG

R1: AGACTAGAGAATT

R2: AGAATTGCGATAG

It is easy to see that not all 11 words of length 10 can be reconstructed with the two reads.

This violates the key assumption of the de Bruijn graphs

It is also easy to see that reducing K ameliorates the problem and eventually gets rid of it (just consider $K=1$...)

Kmer coverage depends on K and read coverage **and** read start point distribution

AGACCGTAACTTTAAAGGGCGAC

AGAC

AGACCGTAACT

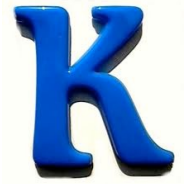
CGTAACTTTAAA

TTAAAGGGCGAC

GGGCGAC

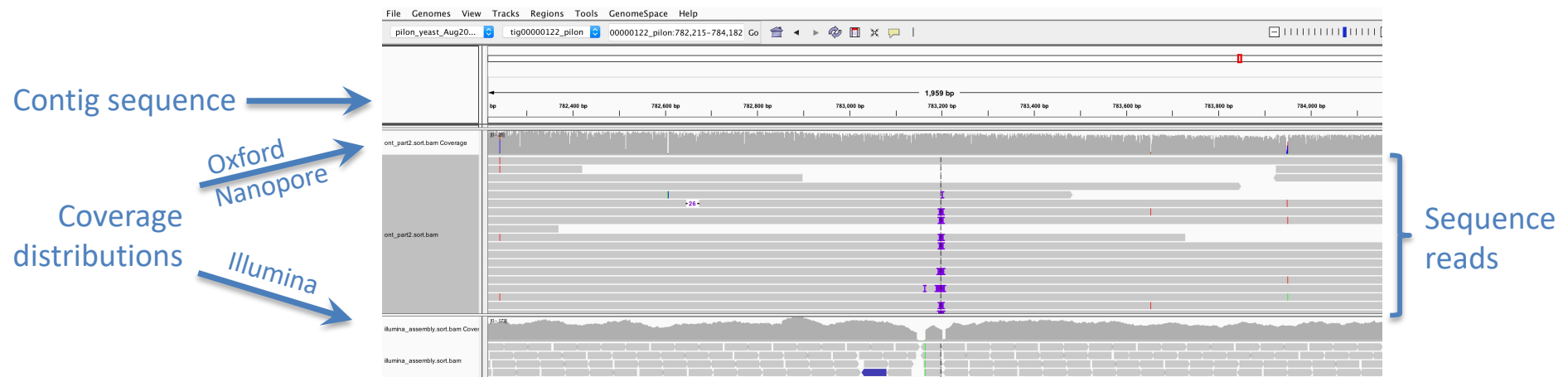
average read coverage (per position): 2

K=4	Kmer coverage	K=8	Kmer coverage
AGAC	2	AGACCGTA	1
GACC	1	GACCGTAA	1
ACCG	1	ACCGTAAC	1
CCGT	1	CCGTAACT	1
CGTA	2	CGTAACTT	1
GTAA	2	GTAAC TTT	1
TAAC	2	TAACTTTA	1
AACT	2	AACTTTAA	1
ACTT	1	ACTTTAAA	1
CTTT	1	CTTTAAAG	0
TTTA	1	TTTAAAGG	0
TTAA	2	TTAAAGGG	1
TAAA	2	TAAAGGGC	1
AAAG	1	AAAGGGCG	1
AAGG	1	AAGGGCGA	1
AGGG	1	AGGGCGAC	1
GGGC	2		
GGCG	2		
GCGA	2		
CGAC	2		

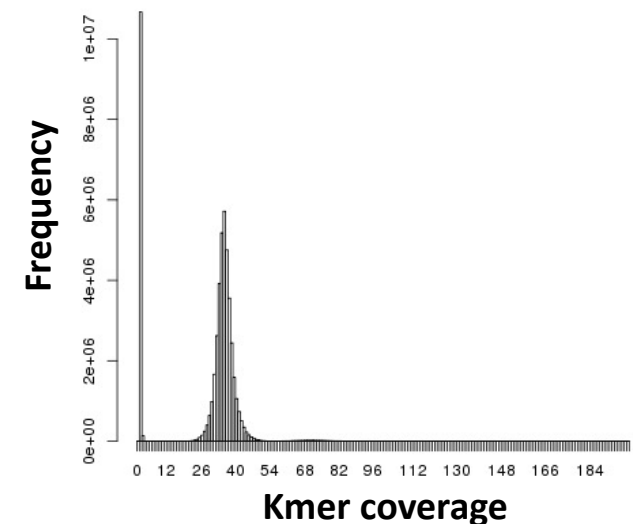


Coverage and Kmer coverage

- **Coverage**, a.k.a. read depth is the number of reads covering on average a position in the sequenced template. This value is invariant for a given experiment

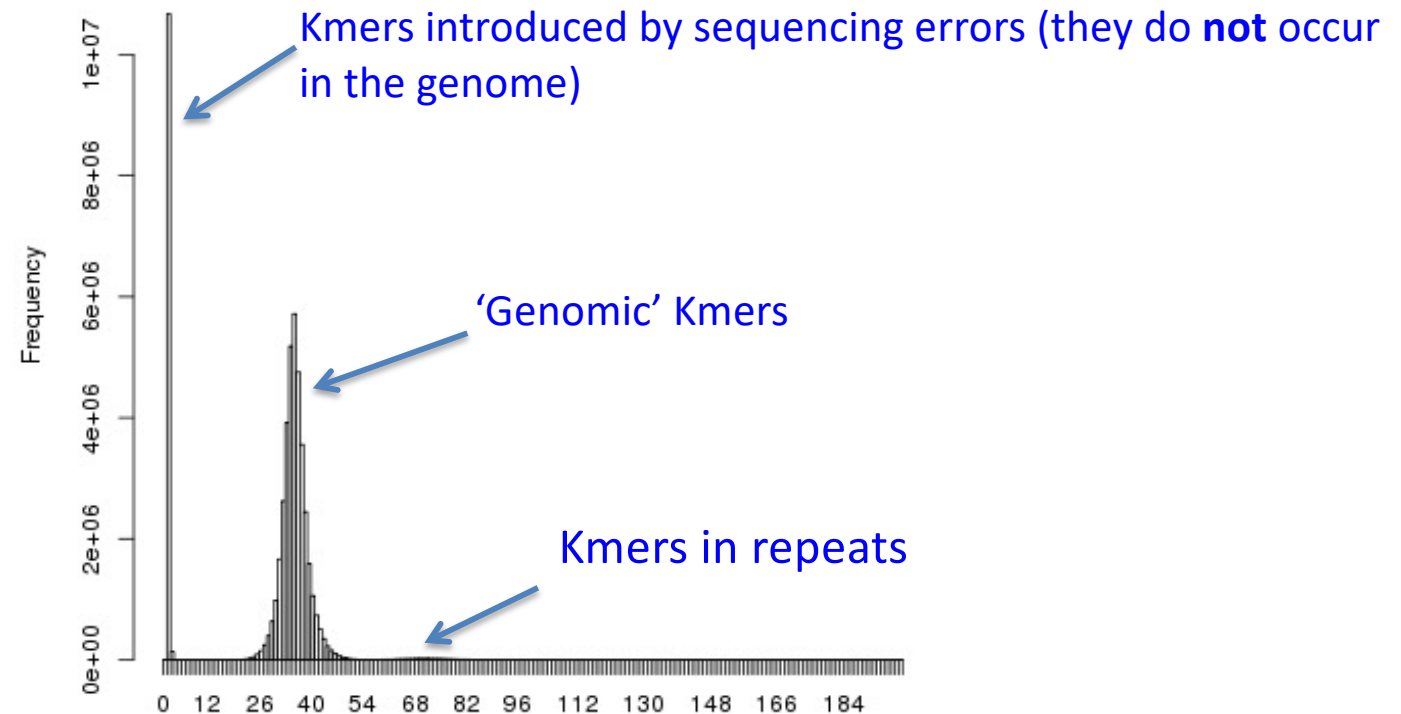


- **Kmer Coverage** is the number of reads a given Kmer is represented in. The value ranges from a minimum of 1 to a maximum of the number of sequence reads and depends on the Kmer size.

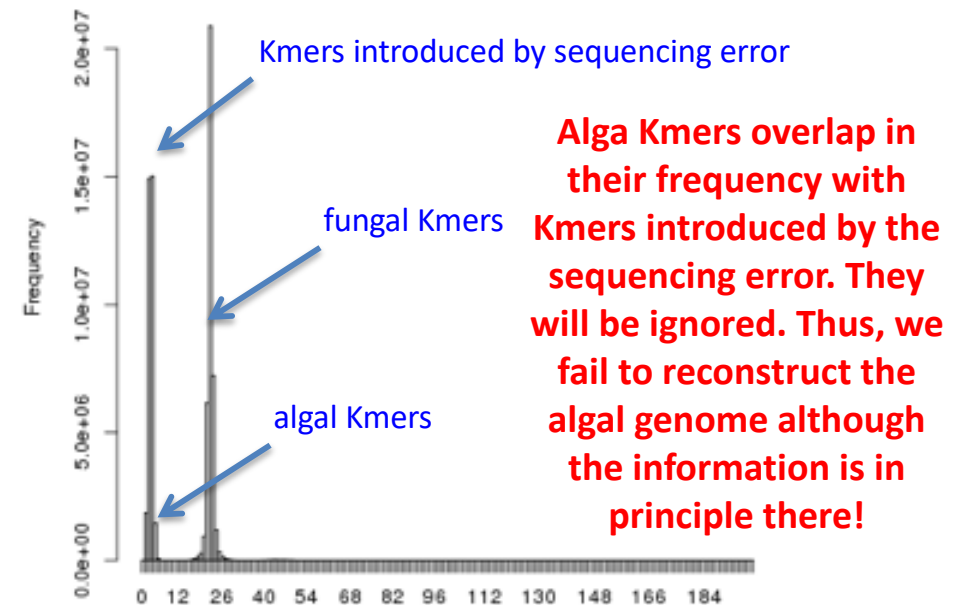
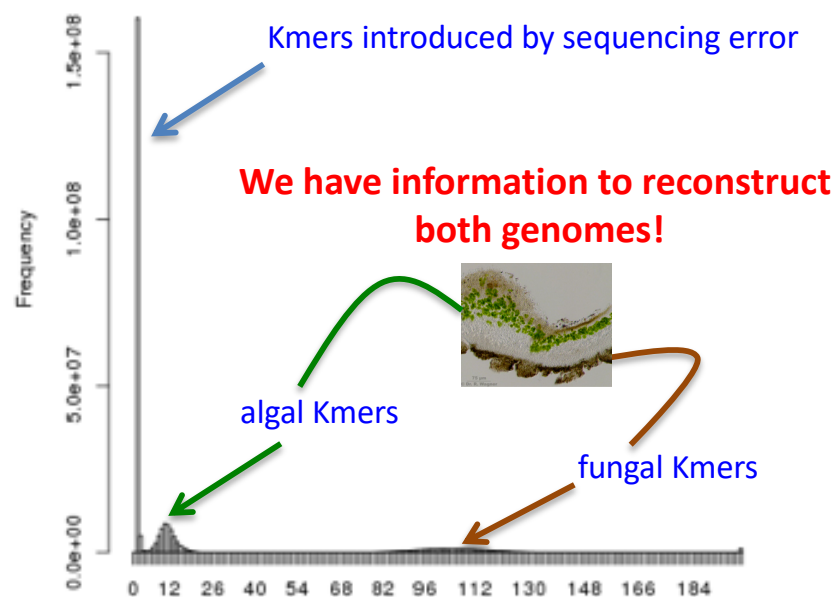




Kmer coverage distributions tend to be
at least bi-modal



Kmer coverage plots for two different values of K using the same (metagenomic) data sets



K=51 what happens if we increase K? K=151

The Kmer coverage is a function of the read coverage and the Kmer length. Thus, even when the read coverage is way above 1, long Kmers may occur in only one or two reads, having thus the same occurrence frequency as Kmers introduced due to sequencing errors.

Short read assembly with Velvet



CSHL Press | Journal Home | Subscriptions | eTOC Alerts | BioSupplyNet

[Genome Res.](#) 2008 May; 18(5): 821–829.

PMCID: PMC2336801

doi: [10.1101/gr.074492.107](https://doi.org/10.1101/gr.074492.107)

Velvet: Algorithms for de novo short read assembly using de Bruijn graphs

[Daniel R. Zerbino](#) and [Ewan Birney](#)¹

[Author information](#) ► [Article notes](#) ► [Copyright and License information](#) ►

Velvet: Graph construction

1. Read hashing with Kmer size of 5.
Record for each Kmer also its reverse complement.

Read 1: TAGACTGATTG
 Read 2: TAGACTG
 Read 3: ACTGATTG
 Read 4: ATTGACCA
 Read 5: ATTGCC...

Kmer Rev. complement	Read-Id	Offset
TAGAC GTCTA	Read1	0 6
AGACT AGTCT	Read1	1 5
GACTG CAGTC	Read1	2 4
ACTGA TCAGT	Read1	3 3
CTGAT ATCAG	Read1	4 2
TGATT AATCA	Read1	5 1
GATTG CAATG	Read1	6 0
ATTGA TCAAT	Read4	0 3
TTGAC GTCAA	Read4	1 2
TGACC GGTCA	Read4	2 1
GACCA TGGTC	Read4	3 0
ATTGC GCAAT	Read5	0 0
TTGCC GGCAA	Read5	1 ? ¹

¹ Note, read 5 is not given in full length, hence the '?'

Velvet: Graph construction

1. Read hashing with Kmer size of 5.
Record for each Kmer also its reverse complement.

Read 1: T A G A C T G A T T G
 Read 2: T A G A C T G
 Read 3: A C T G A T T G
 Read 4: A T T G A C C A
 Read 5: A T T G C C...

Kmer Rev. complement	Read-Id	Offset
TAGAC GTCTA	Read1	0 6
AGACT AGTCT	Read1	1 5
GACTG CAGTC	Read1	2 4
ACTGA TCAGT	Read1	3 3
CTGAT ATCAG	Read1	4 2
TGATT AATCA	Read1	5 1
GATTG CAATG	Read1	6 0
ATTGA TCAAT	Read4	0 3
TTGAC GTCAA	Read4	1 2
TGACC GGTCA	Read4	2 1
GACCA TGGTC	Read4	3 0
ATTGC GCAAT	Read5	0 0
TTGCC GGCAA	Read5	1 ?

Velvet: Graph construction

1. Read hashing with Kmer size of 5.
Record for each Kmer also its reverse complement.

Read 1: T A G A C T G A T T G

Read 2: T A G A C T G

Read 3: A C T G A T T G

Read 4: A T T G A C C A

Read 5: A T T G C C...

Kmer Rev. complement	Read-Id	Offset
TAGAC GTCTA	Read1	0 6
AGACT AGTCT	Read1	1 5
GACTG CAGTC	Read1	2 4
ACTGA TCAGT	Read1	3 3
CTGAT ATCAG	Read1	4 2
TGATT AATCA	Read1	5 1
GATTG CAATG	Read1	6 0
ATTGA TCAAT	Read4	0 3
TTGAC GTCAA	Read4	1 2
TGACC GGTCA	Read4	2 1
GACCA TGGTC	Read4	3 0
ATTGC GCAAT	Read5	0 0
TTGCC GGCAA	Read5	1 ?

Velvet: Graph construction

1. Read hashing with Kmer size of 5.
Record for each Kmer also its reverse complement.

Read 1: T A G A C T G A T T G

Read 2: T A G A C T G

Read 3: A C T G A T T G

Read 4: A T T G A C C A

Read 5: A T T G C C...

Kmer Rev. complement	Read-Id	Offset
TAGAC GTCTA	Read1	0 6
AGACT AGTCT	Read1	1 5
GACTG CAGTC	Read1	2 4
ACTGA TCAGT	Read1	3 3
CTGAT ATCAG	Read1	4 2
TGATT AATCA	Read1	5 1
GATTG CAATG	Read1	6 0
ATTGA TCAAT	Read4	0 3
TTGAC GTCAA	Read4	1 2
TGACC GGTCA	Read4	2 1
GACCA TGGTC	Read4	3 0
ATTGC GCAAT	Read5	0 0
TTGCC GGCAA	Read5	1 ?

Velvet: Graph construction

2. Rewrite reads with original Kmers

Read 1: TAGACTGATTG

1.0|1.1|1.2|1.3|1.4|1.5|1.6

Read 2: TAGACTG

1.0|1.1|1.2

Read 3: ACTGATTG

1.3|1.4|1.5|1.6

Read 4: ATTGACCA

4.0|4.1|4.2|4.3

Read 5: ATTGCC...

5.0|5.1...

Kmer Rev. complement	Read-Id	Offset
TAGAC GTCTA	Read1	0 6
AGACT AGTCT	Read1	1 5
GACTG CAGTC	Read1	2 4
ACTGA TCAGT	Read1	3 3
CTGAT ATCAG	Read1	4 2
TGATT AATCA	Read1	5 1
GATTG CAATG	Read1	6 0
ATTGA TCAAT	Read4	0 3
TTGAC GTCAA	Read4	1 2
TGACC GGTCA	Read4	2 1
GACCA TGGTC	Read4	3 0
ATTGC GCAAT	Read5	0 0
TTGCC GGCAA	Read5	1 ? ¹

¹ Note, read 5 is not given in full length, hence the '?'

Velvet: Graph construction

3. Record for each read which of its original Kmers is overlapped by subsequent reads (Point 5)

Read 1: TAGACTGATTG

1.0|1.1|1.2|1.3|1.4|1.5|1.6

Read 2: TAGACTG

1.0|1.1|1.2

Read 3: ACTGATTG

1.3|1.4|1.5|1.6

Read 4: ATTGACCA

4.0|4.1|4.2|4.3

Read 5: ATTGCC...

5.0|5.1...

Cut the chain of ordered original Kmers each time an overlap with another read starts or ends. For each **uninterrupted sequence of original Kmers** a node is created.

Original Kmer		
Read-Id	Position	Occurrence
Read1	0	Read2
Read1	1	Read2
Read1	2	Read2
Read1	3	Read3
Read1	4	Read3
Read1	5	Read3
Read1	6	Read3
Read4	0	-
Read4	1	-
Read4	2	-
Read4	3	-
Read5	0	-
Read5	1	-

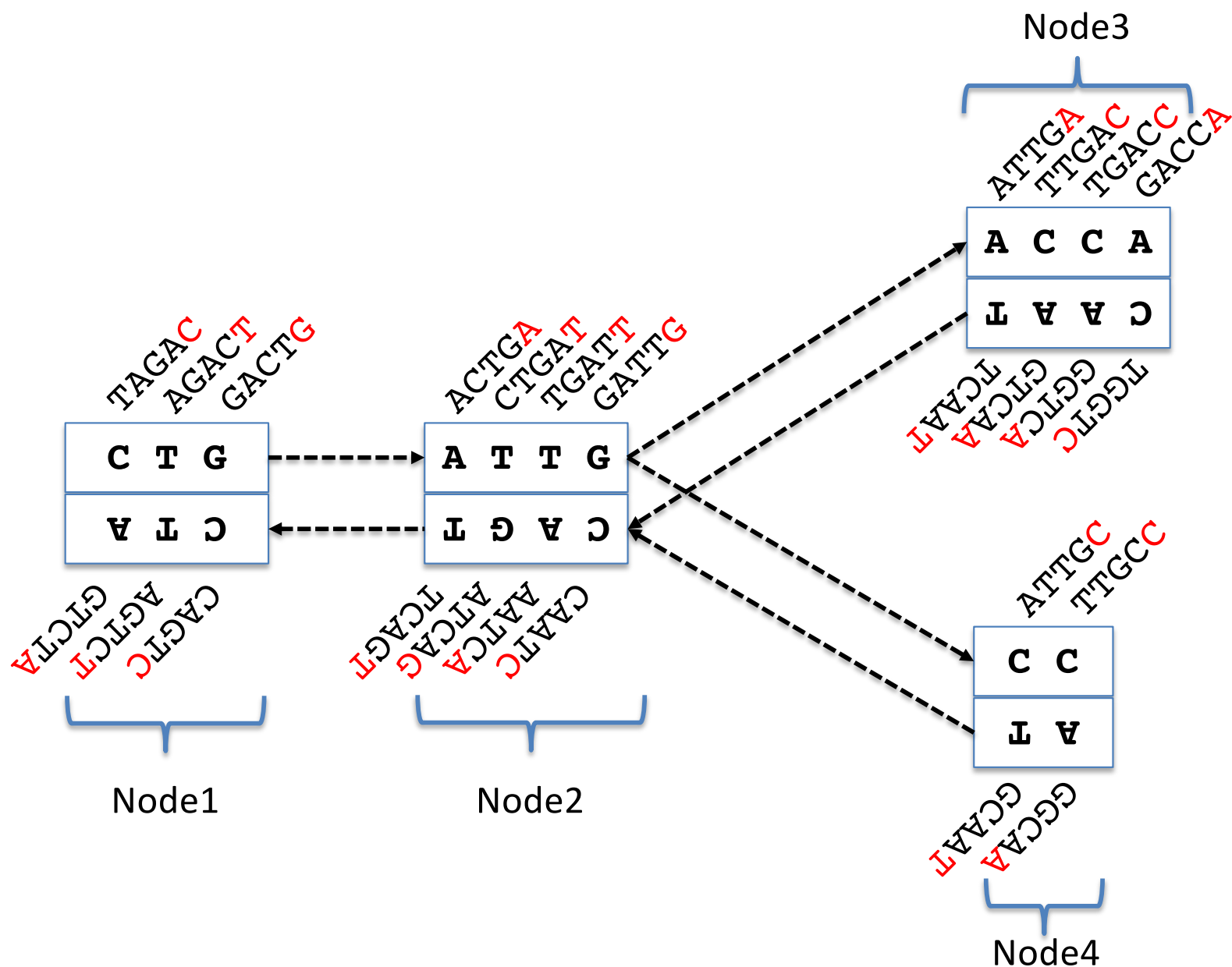
Node1

Node2

Node3

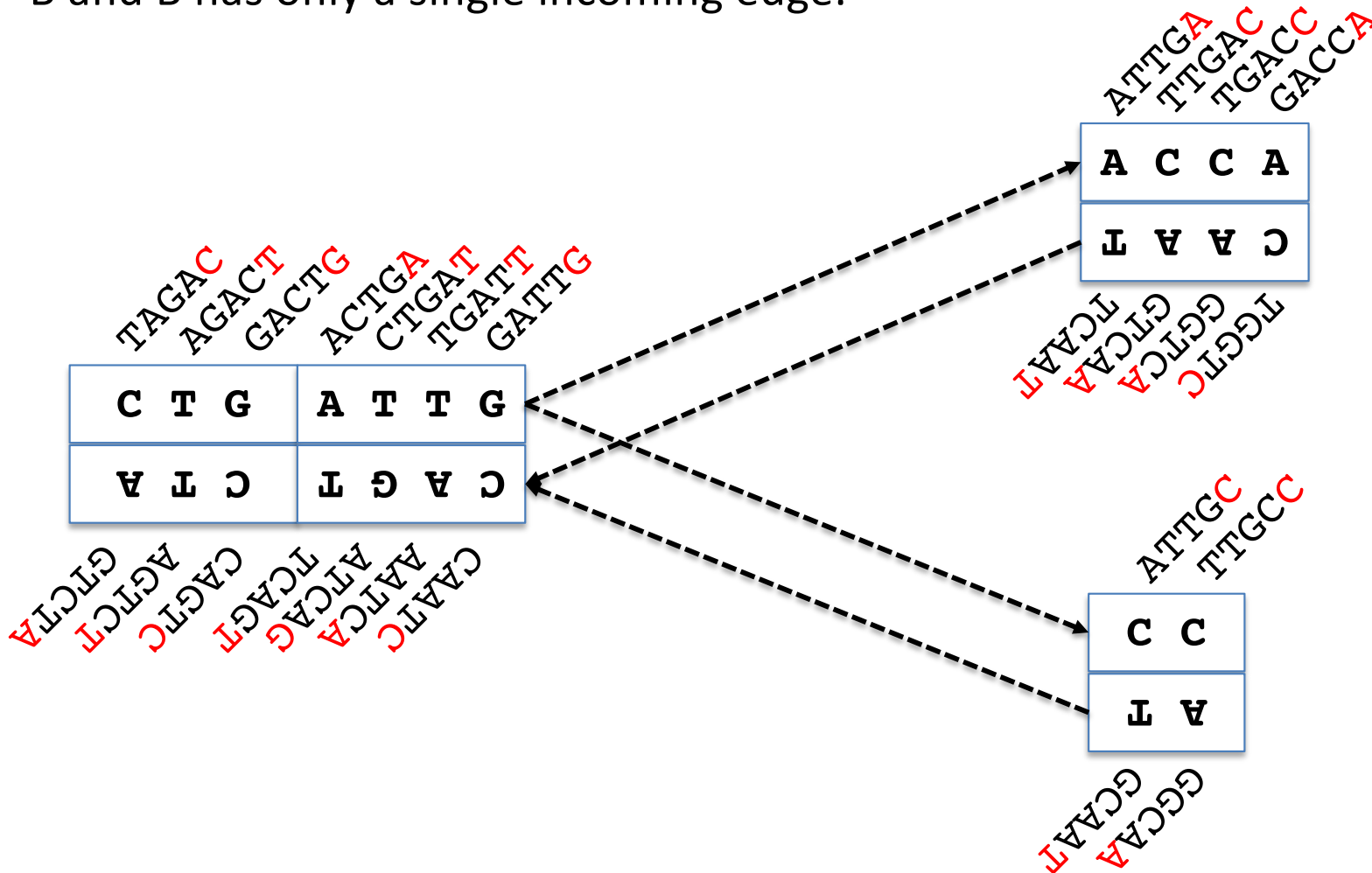
Node4

Constructing the graph



Graph simplification

Simplification: The algorithm so far results in blocks (uninterrupted runs of original Kmers) that occur in a linear arrangement (chain). Thus, connect adjacent nodes A and B whenever A has only one outgoing edge that leads to B and B has only a single incoming edge.



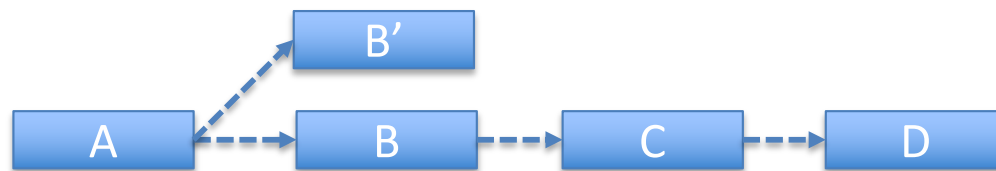
Velvet: Initial read processing and graph construction

1. hash reads using the pre-defined kmer size: smaller Kmers increase connectivity of the graph but also the number of ambiguous repeats
2. record for each observed Kmer the id of the first read containing this Kmer and the position of the Kmer.
3. record each Kmer together with its reverse complement (typically odd Kmer sizes are used to avoid that palindromes (e.g. GAATTC) confuse graph construction).
4. re-write each read as a set of original Kmers -> *Roadmap*
5. build 2nd database with the information for each read which of its original Kmers are overlapped by subsequent reads. Cut the chain of ordered original Kmers each time an overlap with another read starts or ends. For each uninterrupted sequence of original Kmers a node is created.
6. Trace reads through the graph using the *roadmap*.

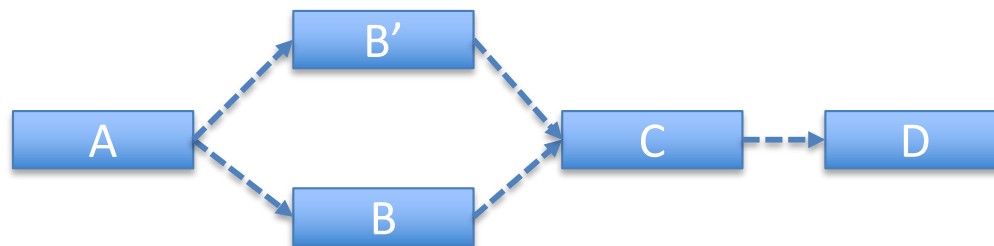
In brief: Information about Kmer-read association is maintained throughout the de Bruijn graph construction

Velvet: Error removal

- Naïve approach: use deviation from [expected coverage](#)¹ to identify and remove 'errors' (sequencing errors and polymorphisms).
- Velvet focuses on topological features:
 - tips due to errors at the edge of reads



- bulges due to internal read errors



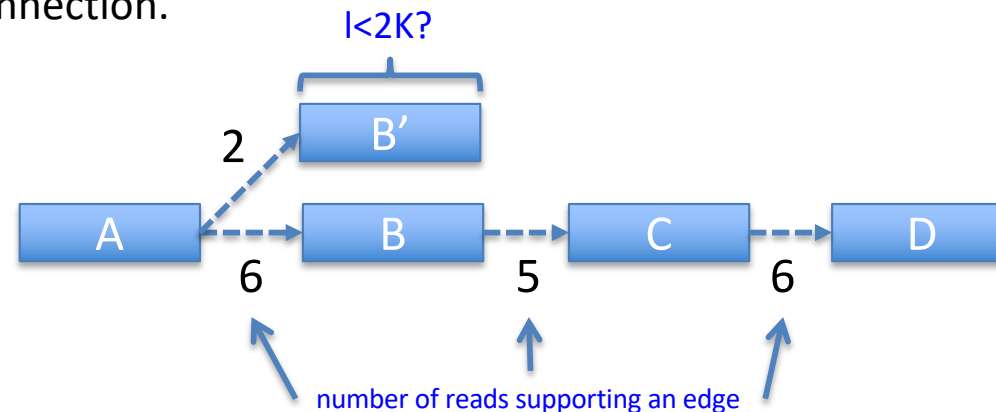
- erroneous connections due to errors in the library preparation (chimera)



¹ is it clear to you how to generate this expectation?

Removal of Tips

- A tip is a chain of nodes that is disconnected on one end
- A tip will be removed if its length l is $< 2K$. Thus, long tips will be retained as they, most likely, represent genuine sequence.
- a minority count criterion is applied. Thus, from two possible paths the more common one is retained.
- Tips are removed in an iterative manner until no more tips fulfill the removal criterion. Subsequent to tip removal the graph is simplified again (joining of adjacent nodes with unambiguous connection).

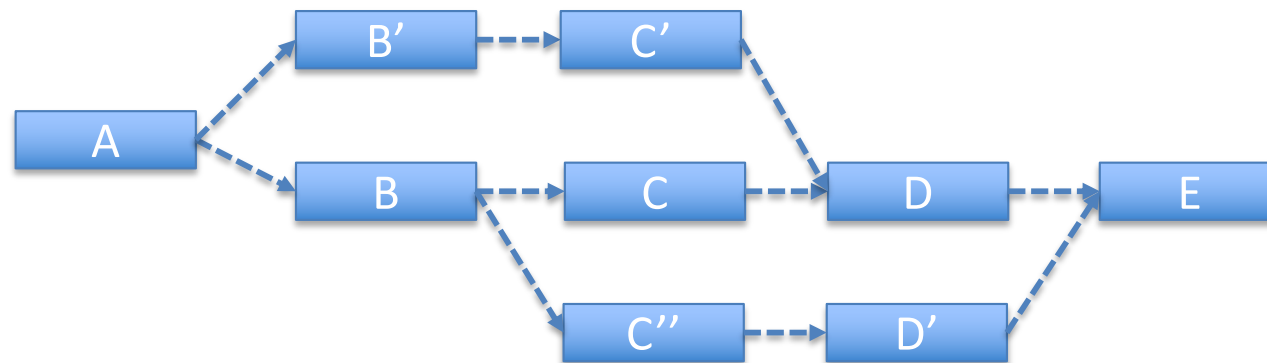


-> remove B' if its length is smaller than $2K$

-> remove B' if less reads support the A->B' connection than the A->B connection.
Remember that Velvet traces the reads in the graph!

Removal of Bubbles

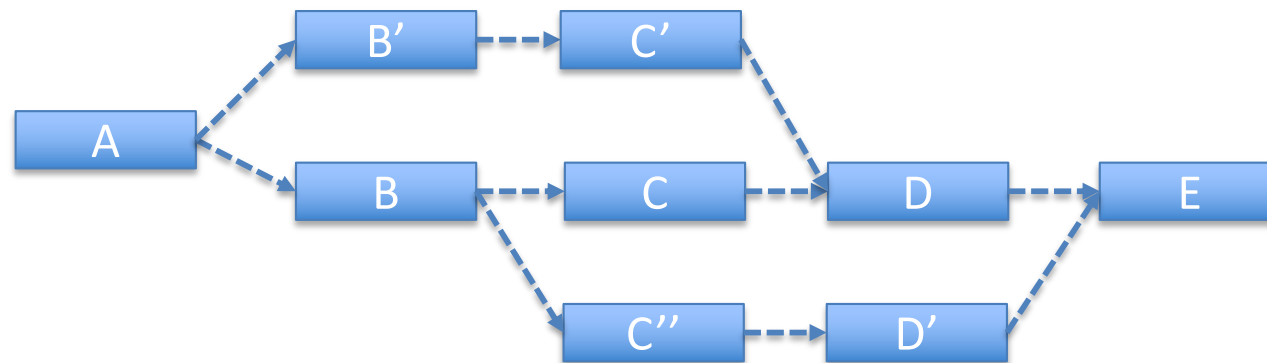
- Two paths are redundant if they start and end at the same nodes and contain similar sequences.
- Velvet removes such bubbles with the 'tour bus' algorithm
- a simple sequence identity and length threshold is applied to select paths for simplification
 - start from an arbitrary node with out-degree larger 1 and progress along the graph visiting nodes in order of increasing distance from the start.
 - The distance between two consecutive nodes X and Y is given by the length of $s(Y)$ divided by the number of reads mapped to the edge between X and Y. Thus, paths represented in many reads result in shorter distances.



Path1: A-B'-C'-D-E with distance PL1

Removal of Bubbles

- Two paths are redundant if they start and end at the same nodes and contain similar sequences.
- Velvet removes such bubbles with the 'tour bus' algorithm
- a simple sequence identity and length threshold is applied to select paths for simplification
 - start from an arbitrary node **with outdegree larger 1** and progress along the graph visiting nodes in order of increasing distance from the start.
 - The traversal through the graph is repeated from the same starting node but now following an alternative path and stopping when an already visited node is reached.

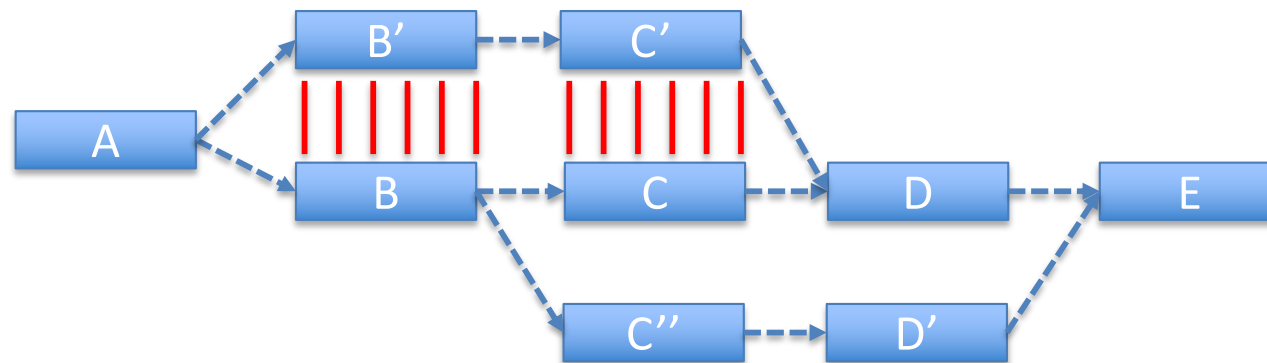


Path1: A-B'-C'-D-E with distance PL1

Path2: A-B-C-D with distance PL2. Note that path stops at D, as this node has been visited before

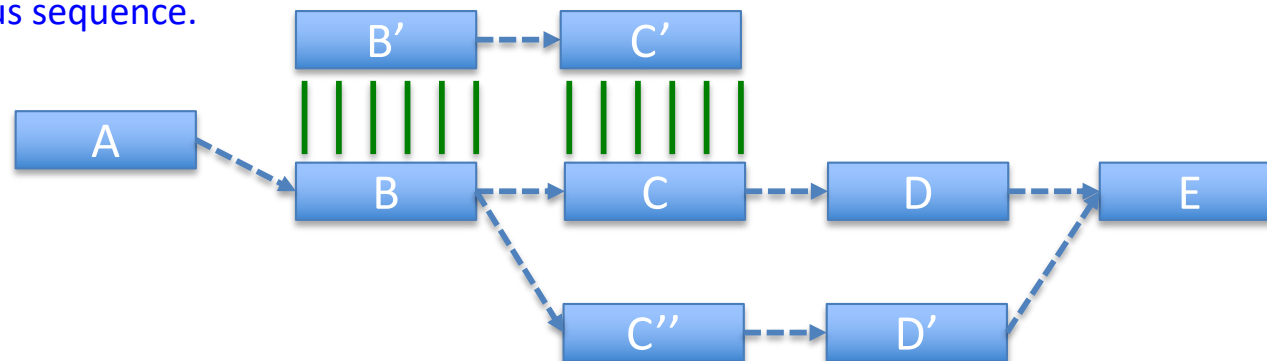
Removal of Bubbles

- Two paths are redundant if they start and end at the same nodes and contain similar sequences.
- Velvet removes such bubbles with the 'tour bus' algorithm
- a simple sequence identity and length threshold is applied to select paths for simplification
 - start from an arbitrary node and progress along the graph visiting nodes in order of increasing distance from the start.
 - The traversal through the path stops as soon as a node 'D' is visited that has been passed in a previous path.
 - A backtrace starts for both paths ending at 'D' to find their closest common ancestor 'A'.
 - The sequences represented by B' – C' and by B – C, respectively, are extracted, aligned and their similarity is assessed.



Removal of Bubbles

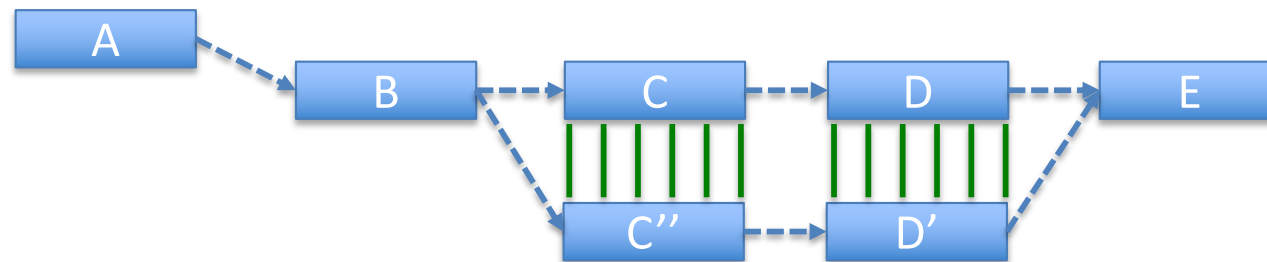
- Two paths are redundant if they start and end at the same nodes and contain similar sequences.
- Velvet removes such bubbles with the 'tour bus' algorithm
- a simple sequence identity and length threshold is applied to select paths for simplification
 - start from an arbitrary node and progress along the graph visiting nodes in order of increasing distance from the start.
 - The traversal through the path stops as soon as a node 'D' is visited that has been passed in a previous path.
 - A backtrace starts for both paths ending at 'D' to find their closest common ancestor 'A'.
 - The sequences represented by B' – C' and by B – C, respectively, are extracted, aligned and their similarity is assessed.
 - If the similarity exceeds a predefined threshold the shorter path is chosen (A-B-C-D) and the alternative path is discarded. *Note, this metric implicitly imposes a majority vote in choosing the consensus sequence.*



Sequence similarity exceeds the threshold

Removal of Bubbles

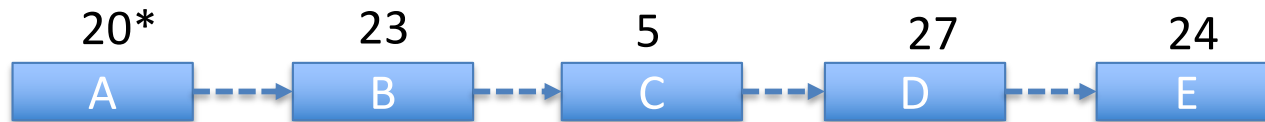
- Two paths are redundant if they start and end at the same nodes and contain similar sequences.
- Velvet removes such bubbles with the 'tour bus' algorithm
- a simple sequence identity and length threshold is applied to select paths for simplification
 - start from an arbitrary node and progress along the graph visiting nodes in order of increasing distance from the start.
 - The traversal through the path stops as soon as a node 'D' is visited that has been passed in a previous path.
 - A backtrace starts for both paths ending at 'D' to find their closest common ancestor 'A'.
 - The sequences represented by B' – C' and by B – C, respectively, are extracted, aligned and their similarity is assessed.
 - If the similarity exceeds a predefined threshold the shorter path is chosen (A-B-C-D) and the alternative path is discarded. Note, this metric implicitly imposes a majority vote in choosing the consensus sequence.



Sequence similarity exceeds the threshold

Removing Erroneous Connections (Chimera problem)

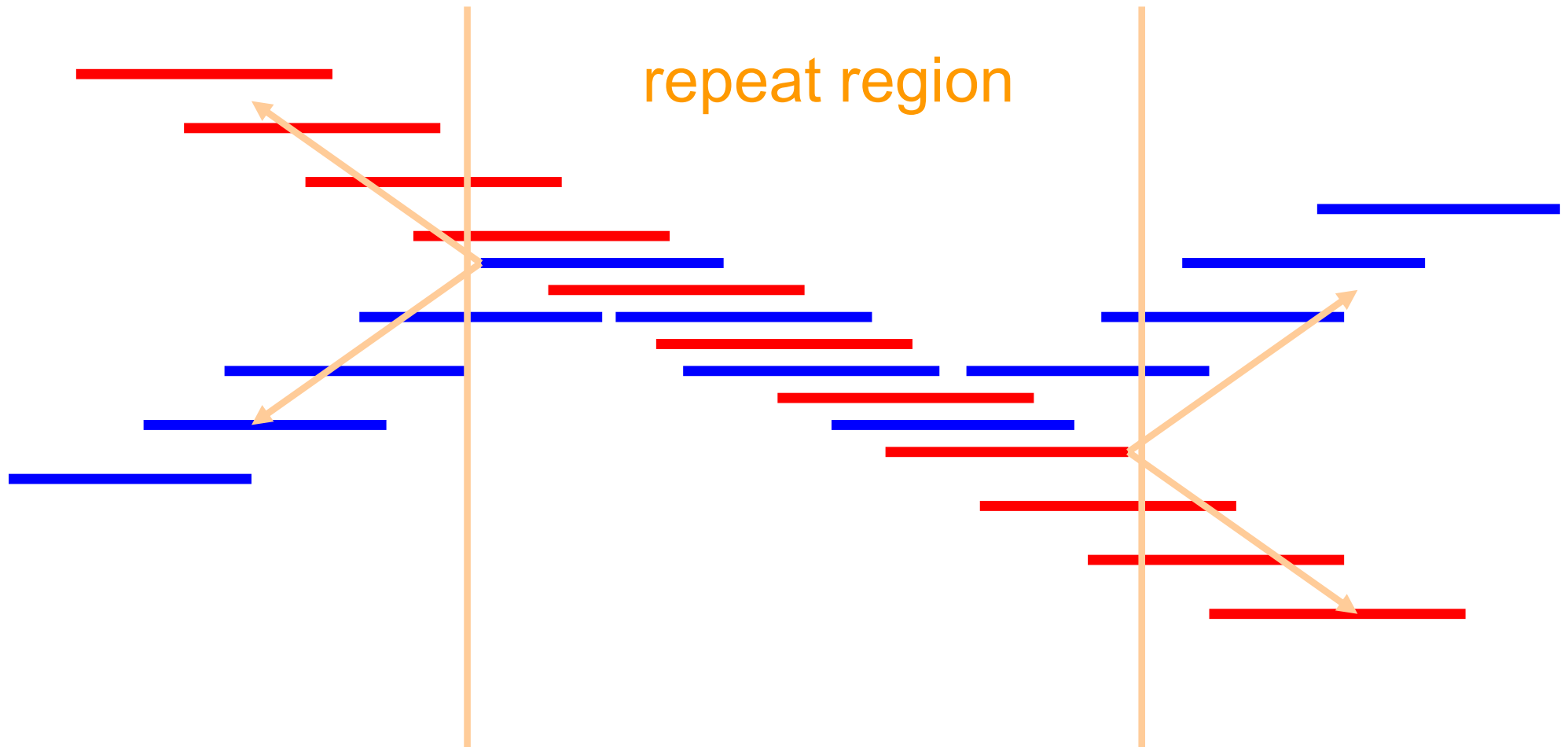
- After completion of the tour bus algorithm the graph is checked for low coverage connections
- Given the law of large numbers any genuine node is expected to be covered by the expected number of reads
- remaining low coverage nodes are likely to flag artificial overlaps due to chimeric reads
- such low-coverage nodes are simply removed and the graph is disrupted



Done with the first step!

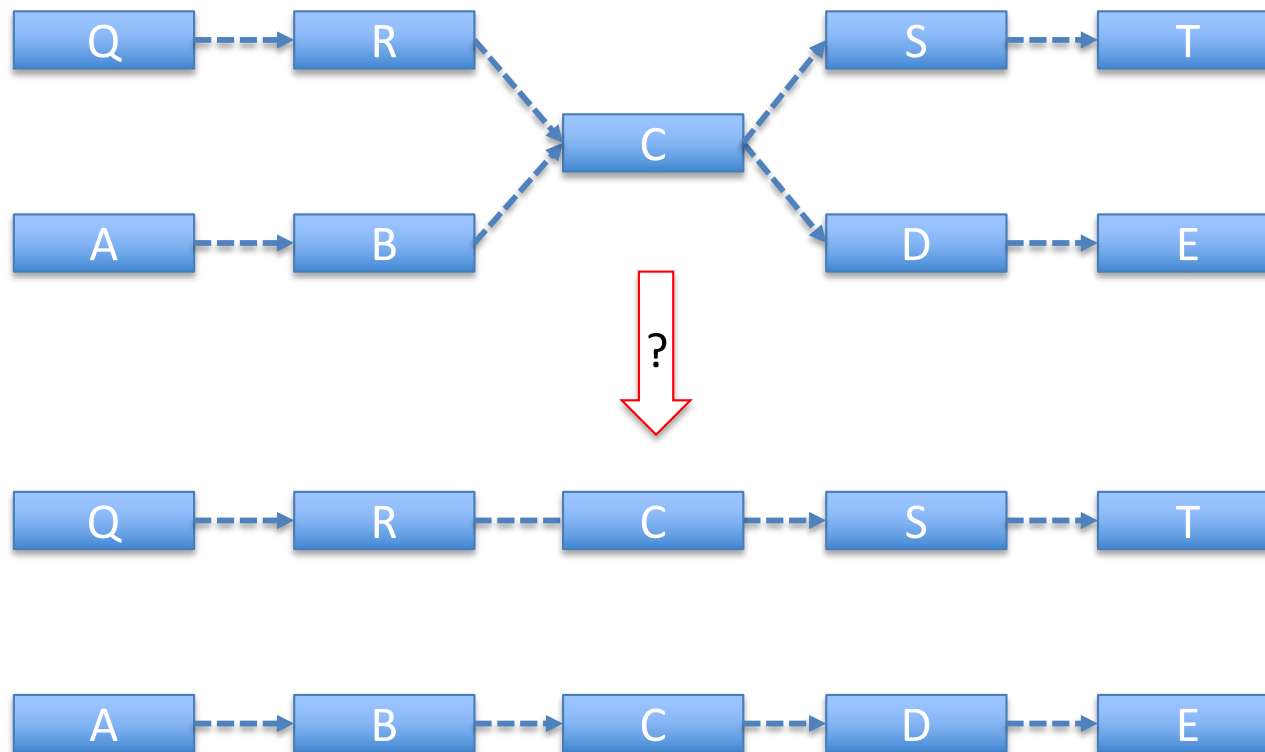
*Numbers denote number of reads covering a given node

The problem with repeats



Merge reads up to potential repeat boundaries

Dealing with repeats

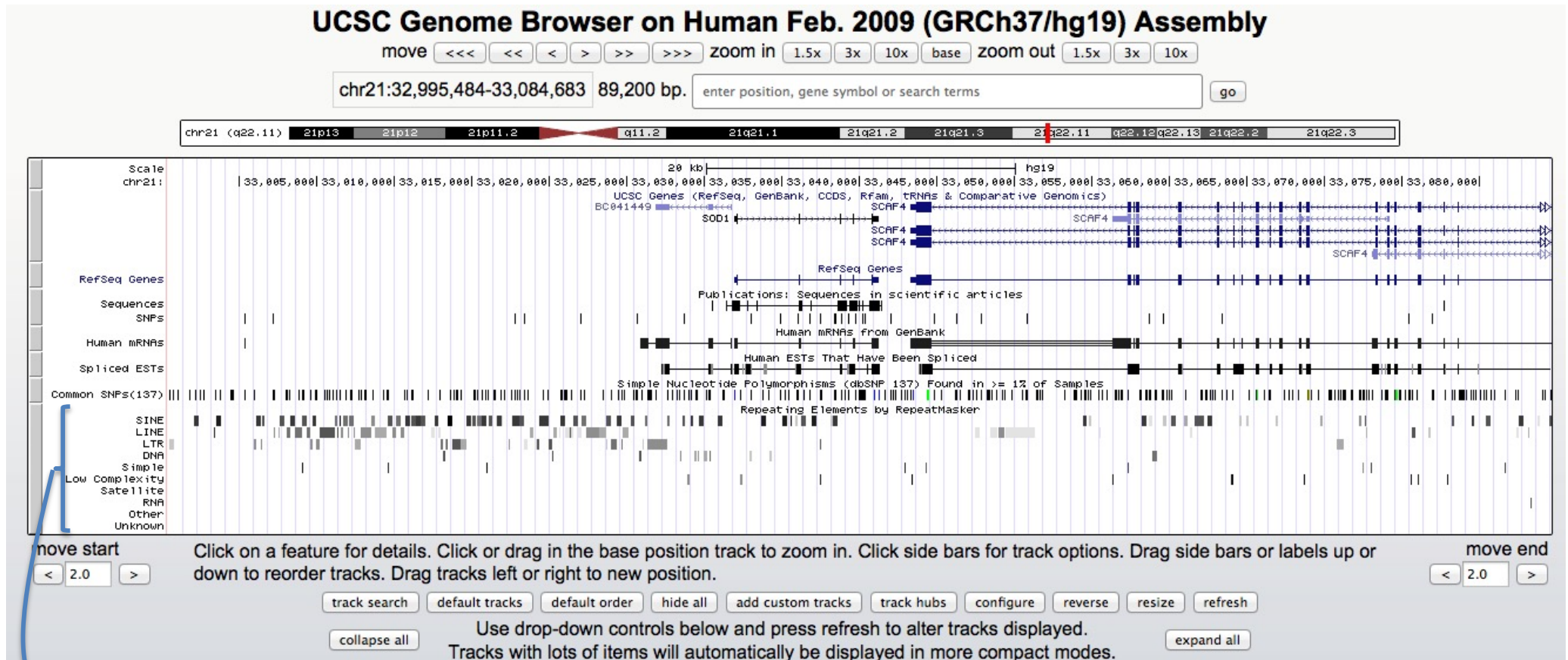


Repeat Types

- **Low-Complexity DNA** (e.g. ATATATATACATA...)
- **Microsatellite repeats** $(a_1...a_k)^N$ where $k \sim 3-6$
(e.g. CAGCAGTAGCAGCACCAG)
- **Transposons/retrotransposons**
 - **SINE** Short Interspersed Nuclear Elements
(e.g., *Alu*: ~300 bp long, 10^6 copies)
 - **LINE** Long Interspersed Nuclear Elements
~500 - 5,000 bp long, 200,000 copies
 - **LTR retroposons** Long Terminal Repeats (~700 bp) at each end
- **Gene Families** genes duplicate & then diverge
- **Segmental duplications** ~very long, very similar copies

How big is the problem?

A human example



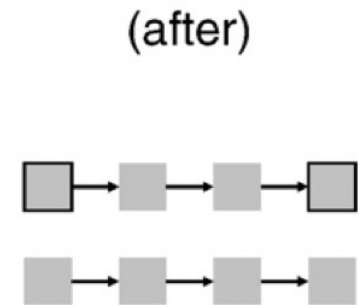
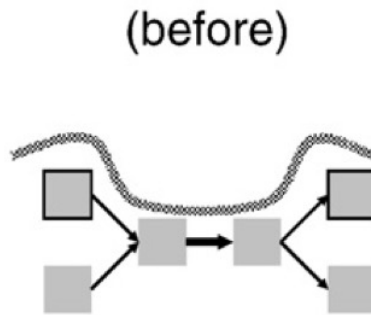
All these are annotated repeats in the human genome

Repeats, Errors, and Contig Lengths

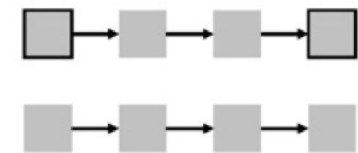
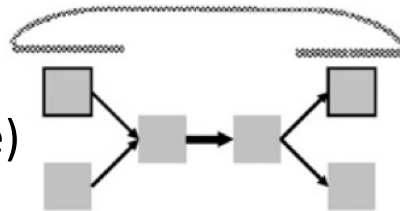
- Repeats shorter than read length are OK
- Repeats with more base pair differences than sequencing error rate are OK
- To make a smaller portion of the genome **appear** repetitive, try to:
 - Increase read length
 - Decrease sequencing error rate

Different strategies of repeat resolution

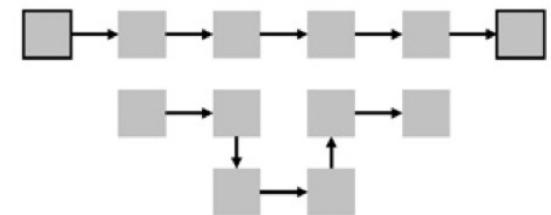
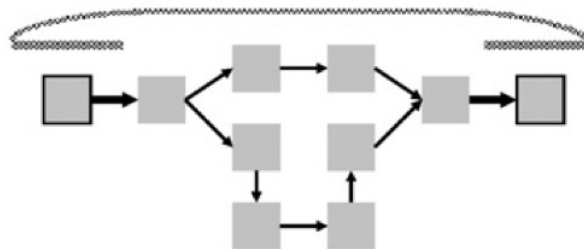
Read threading
(repeat length < read length)



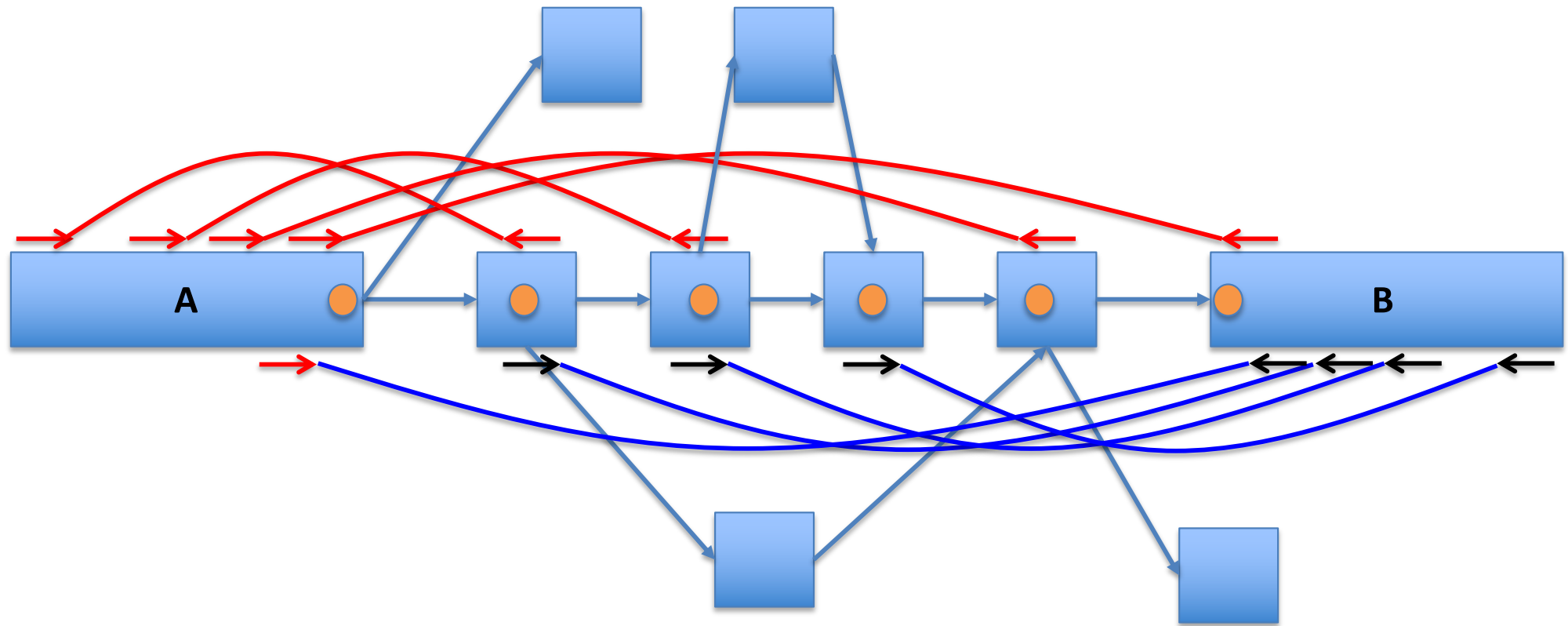
Mate threading
(repeat length < paired end distance)



Path exclusion
(paired end distance is used to exclude longer paths)

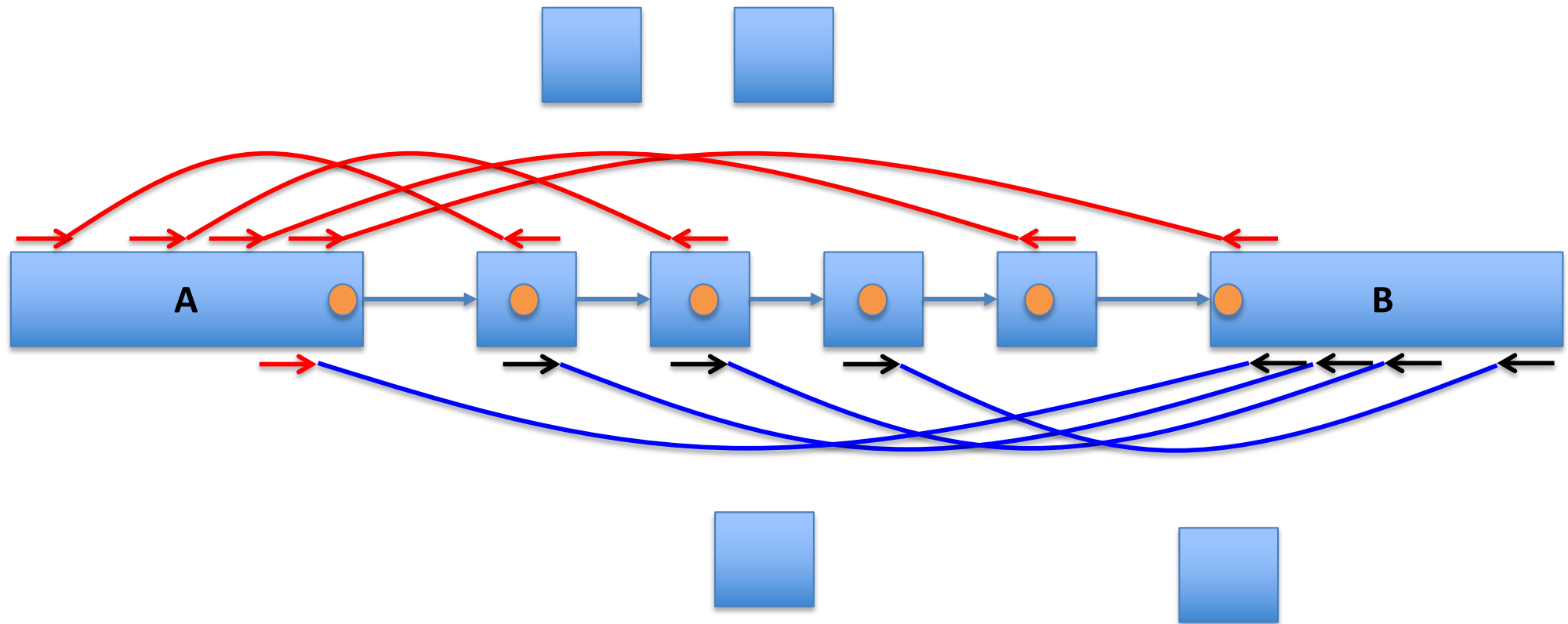



Velvet deals with repeats using its 'Breadcrumb' algorithm



- Determine insert length distribution and decide on a cutoff length longer than practically all inserts
- Identify 'long nodes' **A** and **B** which lengths exceed length cutoff
- use read pairs to identify neighboring long nodes (make sure to understand relevance of length cutoff)
- Identify reads mapping to the long node **A**
- Flag nodes containing the corresponding read pairs
- Perform read mapping for long node **B**
- Identify the consistently paired nodes ●
- Simplify the graph to obtain in the best case a linear path from **A** to **B** (error correction can be applied)

The Breadcrumb algorithm (mate threading)



- Determine insert length distribution and decide on a cutoff length longer than practically all inserts
- Identify 'long nodes' **A** and **B** which lengths exceed length cutoff
- use read pairs to identify neighboring long nodes (make sure to understand relevance of length cutoff)
- Identify reads mapping to the long node **A**
- Flag nodes containing the corresponding read pairs
- Perform read mapping for long node **B**
- Identify the consistently paired nodes 
- Simplify the graph to obtain in the best case a linear path from **A** to **B** (error correction can be applied)

Resolution of repeats with short read pairs

Breadcrumb algorithm (BC)

- determine insert length distribution
 - determine cutoff length longer than practically all inserts and designate as 'long nodes' such nodes in the graph that are longer than the cutoff length. Thus, very few read pairs span over such long nodes!
 - Read pairs are used to pair up neighboring 'long nodes' **A** and **B**. The previous cutoff ensures that no intervening 'long node' can be missed.
 - So far, no test for consistency is applied. Thus, the pairing of 'long nodes' can be ambiguous. However, ambiguously paired nodes are ignored in the next step.
 - For each read starting in a paired 'long node' *BC* identifies and flags the node (now considering all nodes!) containing its mate. If a unique neighboring long-node exists, then this procedure is repeated for the other long node.
 - If the flags results in a linear path from **A** to **B** all flagged nodes can be merged. Otherwise a Tour-Bus like process can be applied for correcting remaining errors. Likewise, long nodes connected by <5 read pairs are not joined.
- It may not be entirely obvious by now, but INSERT SIZE of the sequencing library is a crucial parameter for sequence assembly!