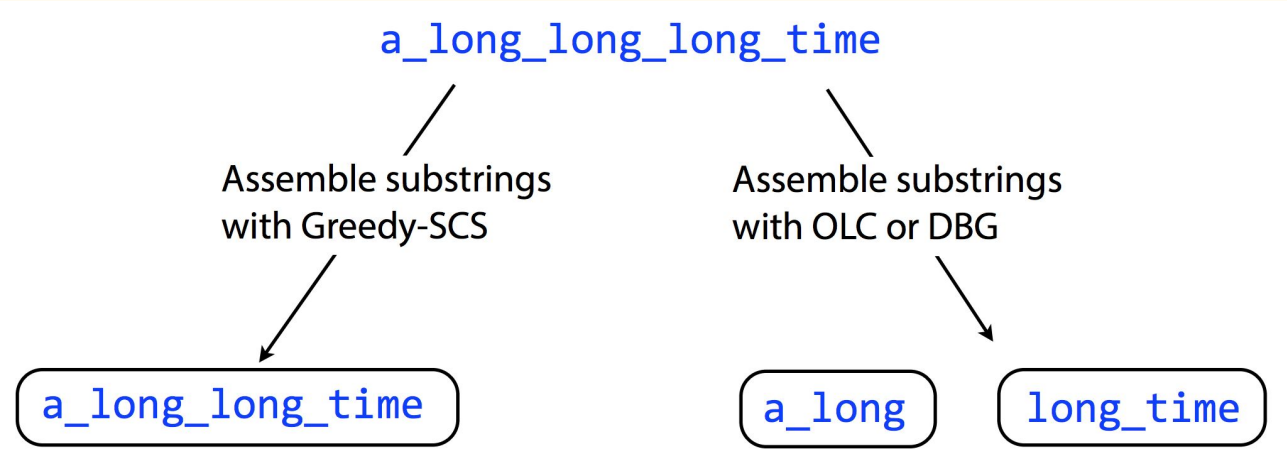
De Bruijn graph assembly

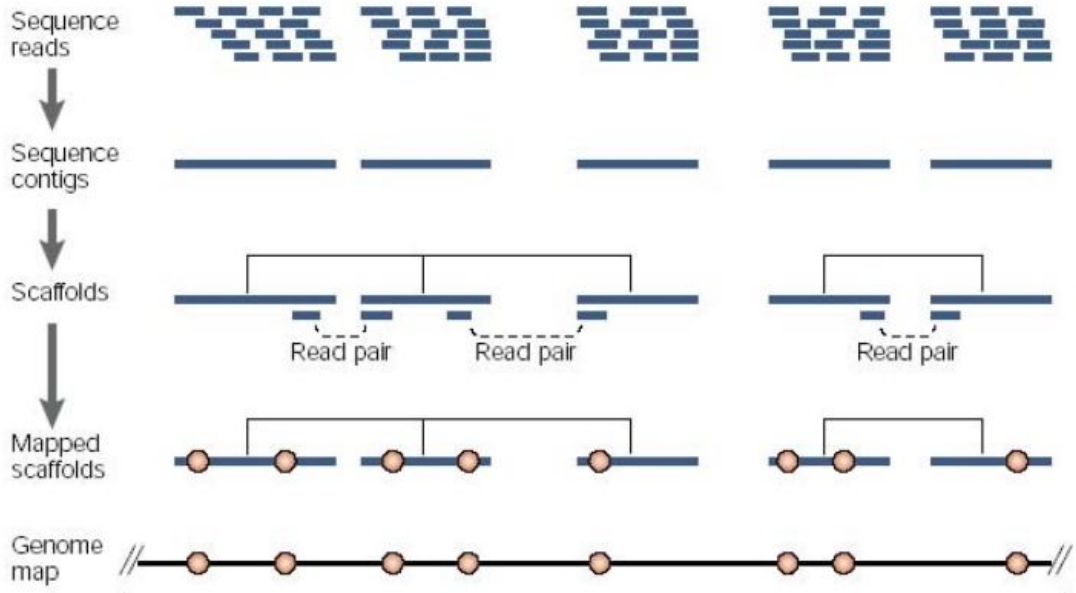
Real-world assembly methods

OLC: Overlap-Layout-Consensus assembly
DBG: De Bruijn graph assembly

Both handle unresolvable repeats by essentially leaving them out
Unresolvable repeats break the assembly into fragments
Fragments are contigs (short for contiguous)



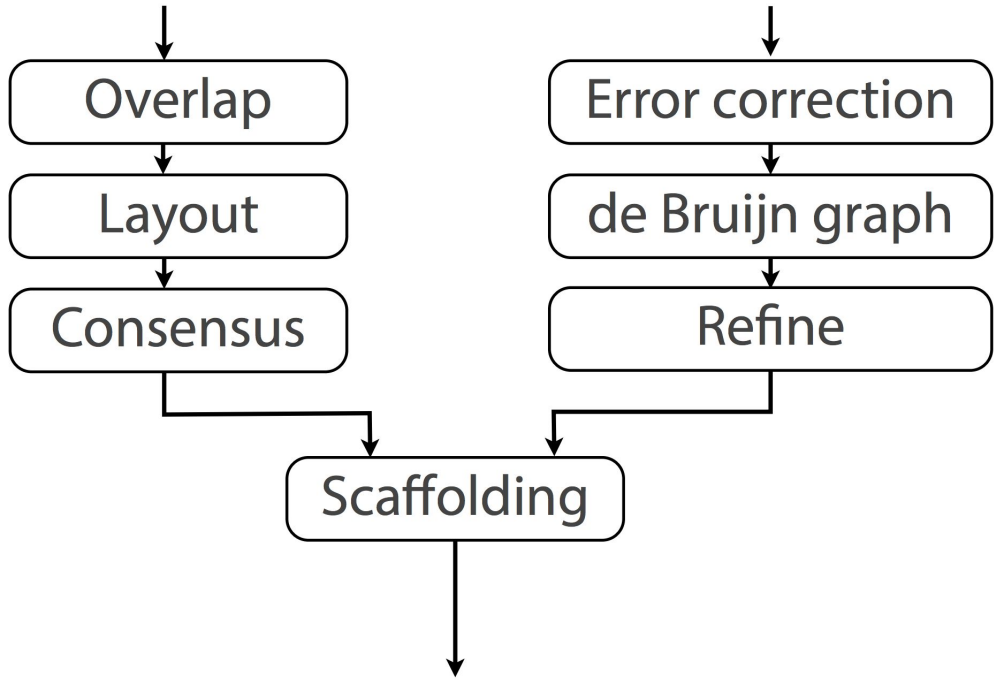
de novo whole-genome shotgun assembly



Assembly process approaches

Alternative 1: Overlap-Layout-Consensus (OLC) assembly

Alternative 2: de Bruijn graph (DBG) assembly



De Bruijn graph assembly

A formulation conceptually similar to overlapping/SCS, but has some potentially helpful properties not shared by SCS.

k-mer

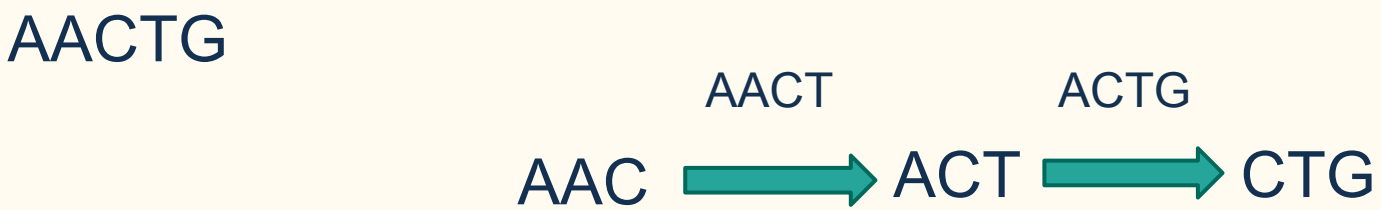
"k-mer" is a substring of length k; "*mer*" from Greek part

I'll use "k-1-mer" to refer to a substring of length k - 1

De Bruijn graphs

- For a fixed integer **k**:
  - **nodes** - all **k-1**-mers present in reads
  - **edges** - for each **k**-mer **x** present in reads there is an edge between **k-1**-mer **prefix** of **x**, and **k-1**-mer **suffix** of **x**

- Example for a single read and k = 4:



De Bruijn graphs: k-mer oddity

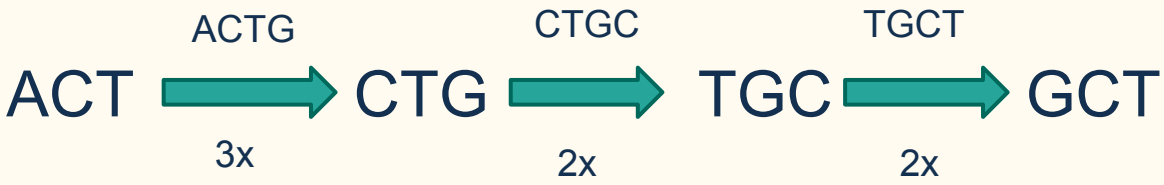
- In practical implementation, we always identify a read with its reverse complement
- **k** is practically exclusively an **odd** integer as a result (otherwise that causes ambiguities in the strand-specificness of the graph)

Assembler who utilize this in order to avoid palindromic sequences: SOAPdenovo2, Velvet

De Bruijn graphs: edge weight

- What happens if we add redundancy?
- Previous example for multiple reads and k = 4:

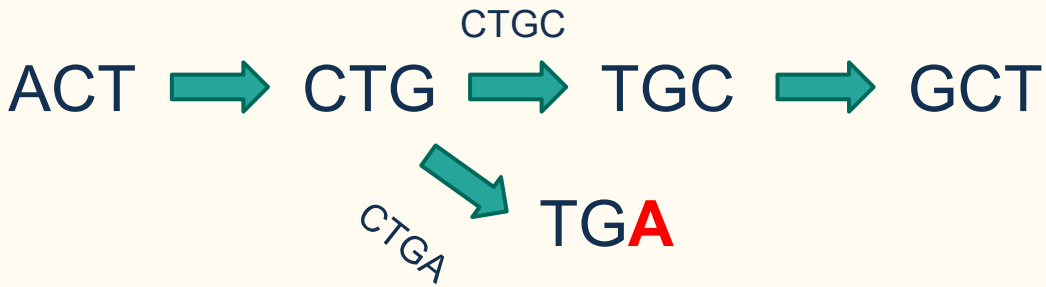
ACTG
ACTG
ACTG
ACTGC
ACTGC
ACTGCT
ACTGCT



De Bruijn graphs: sequencing error impact

- How does a sequencing error impact de Bruijn graph?
- Previous example for multiple reads and k = 4:

ACTG
ACTG
ACTG
ACTGC
ACTG**A**
ACTGCT
ACTGCT

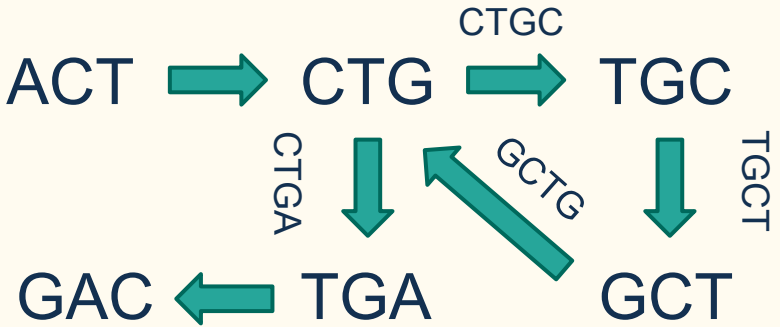


De Bruijn graphs

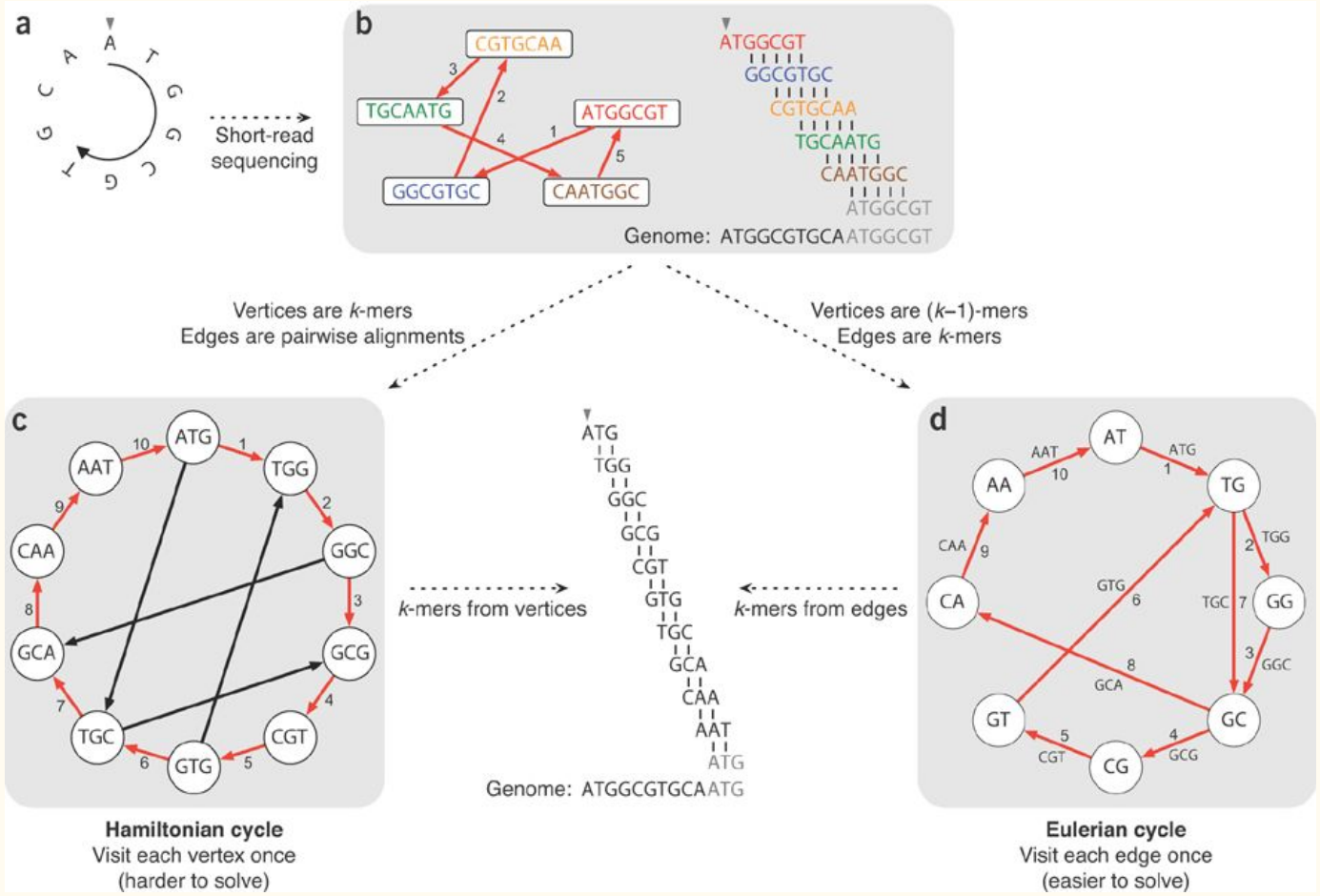
- What is the effect of a small repeat on the graph?
- Example for multiple reads and k = 4:

ACTG
CTGC
TGCT
G**CTG**
CTGA
TGAC

***ACTGCTG**AC*
***ACTGCTGCTGCTG**AC*



Eulerian vs Hamiltonian cycle



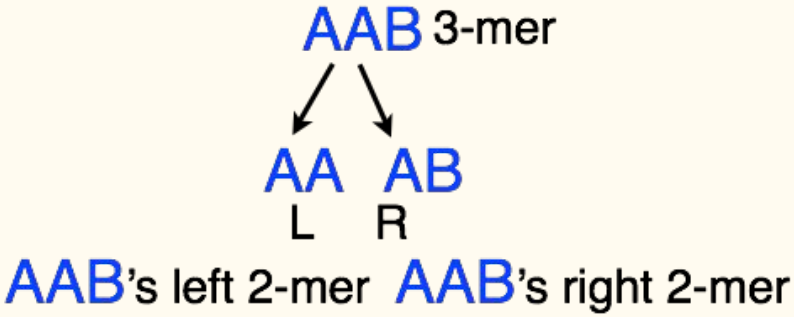
P. E. Compeau, et al: How to apply de Bruijn graphs to genome assembly, 2011, Nat. biotechnology, doi:10.1038/nbt.2023

De Bruijn graph

As usual, we start with a collection of reads, which are substrings of the reference genome.

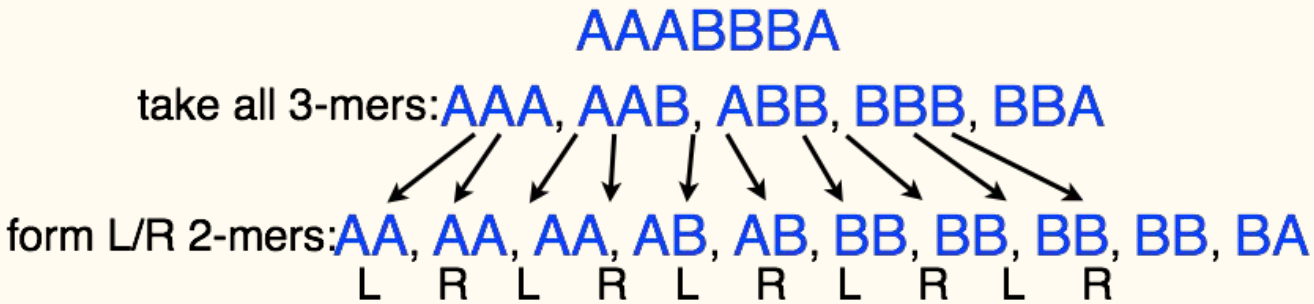
AAA, AAB, ABB, BBB, BBA

AAB is a k-mer (k = 3). AA is its left k-1-mer, and AB is its right k-1-mer.

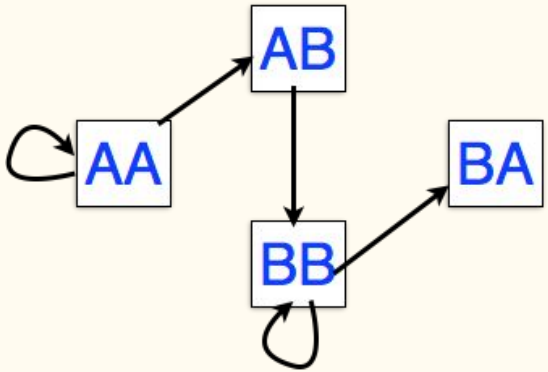


De Bruijn graph

Take each length-3 input string and split it into two overlapping substrings of length 2. Call these the left and right 2-mers.

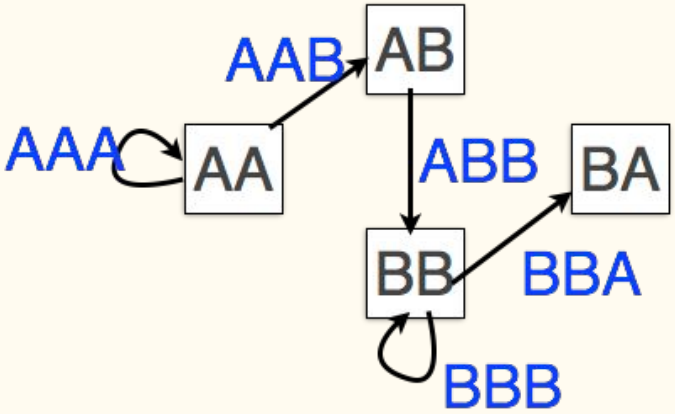


Let 2-mers be nodes in a new graph. Draw a directed edge from each left 2-mer to corresponding right 2-mer:



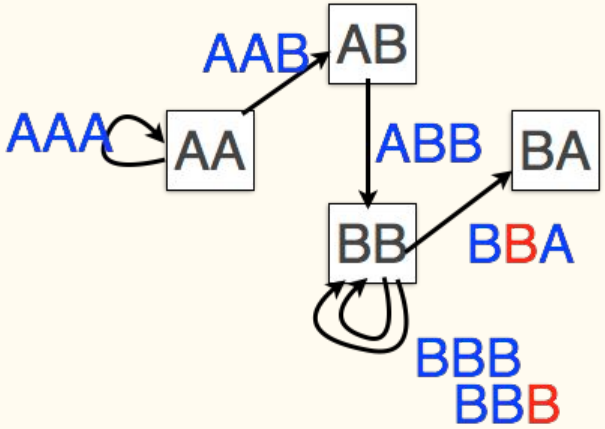
Each edge in this graph corresponds to a length-3 input string

De Bruijn graph



An edge corresponds to an overlap (of length k-2) between two k-1 mers. More precisely, it corresponds to a k-mer from the input.

De Bruijn graph



If we add one more B to our input string: AAABBBBA, and rebuild the De Bruijn graph accordingly, we get a *multiedge*.

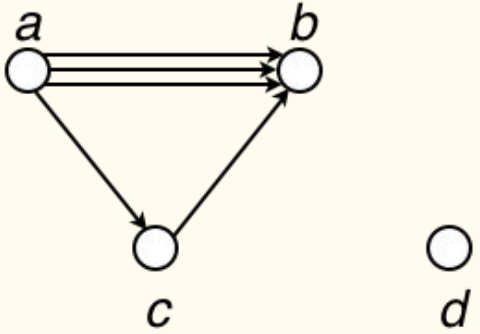
Graphs: Directed multigraph

Directed multigraph G(V, E) consists of set of vertices, V and **multiset** of directed edges, E.

Otherwise, like a directed graph.

Node's indegree = # incoming edges.
Node's outdegree = # outgoing edges.
(Note: loops are counted twice)

De Bruijn graph is a directed multigraph.



V = { a, b, c, d }
E = { (a, b), (a, b), (a, b), (a, c), (c, b) }
Repeated

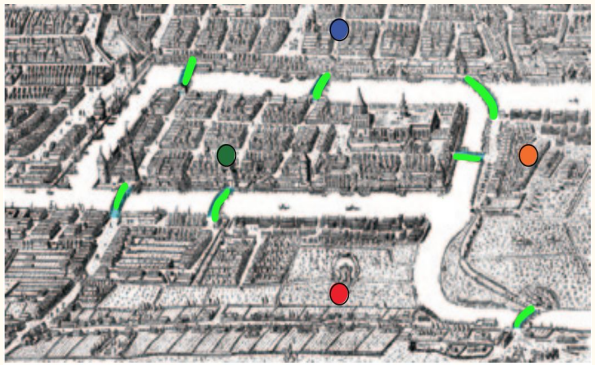
Graphs: Seven Bridges of Königsberg

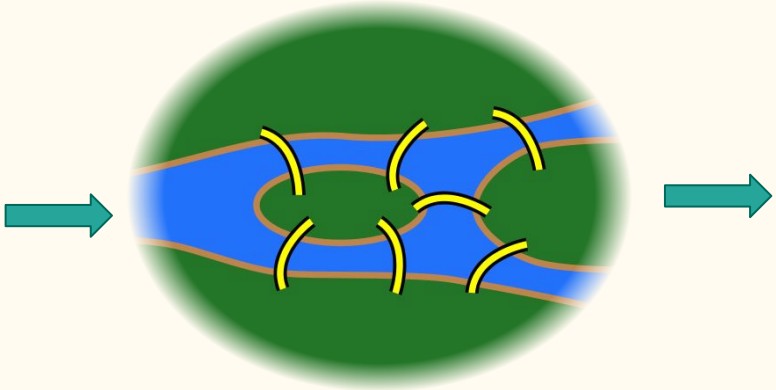
Seven Bridges Problem: Can we have a walk through the city that would cross each of those bridges once and only once. (*Eulerian walk*)

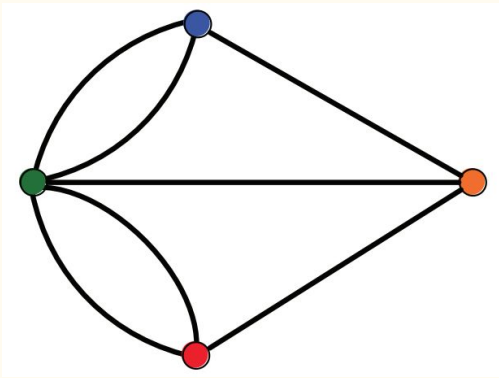
- Graph is connected
- Zero or two nodes with odd degree

Euler cycle (tour): Find a cycle in a graph that visits every edge exactly once.

- Graph is connected
- No nodes of odd degree







Graphs: Eulerian graph

- Node is *balanced* if indegree equals outdegree.
- Node is *semi-balanced* if indegree differs from outdegree by 1.
- Graph is *connected* if each node can be reached by some other node.

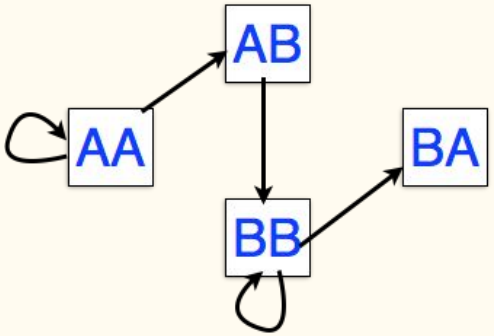
A directed graph G is Eulerian if it contains an *Eulerian cycle (tour)*.
Def:A connected graph has Eulerian cycle if and only if **each** of its vertices is balanced.

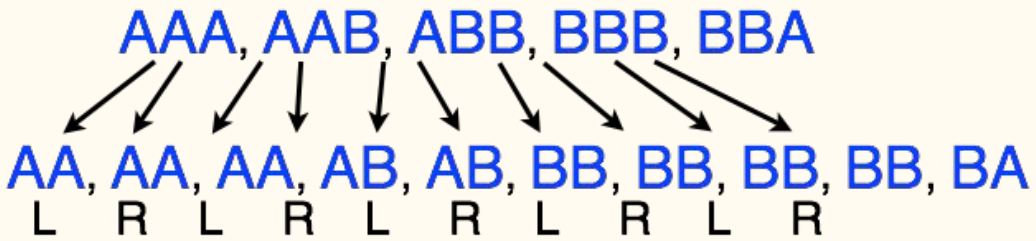
Eulerian walk (*path*) visits each edge exactly once. Not all graphs have Eulerian walks. Graphs that do are Eulerian (semi-Eulerian actually).
(**For simplicity, in this course we won't distinguish Eulerian from semi-Eulerian**.)

Def: A directed, connected graph has an *Eulerian path* if and only if it has **at most two** semi-balanced nodes and all other nodes are balanced.

De Bruijn graph

Back to our De Bruijn graph





Is it Eulerian? Yes

Argument 1: AA → AA → AB → BB → BB → BA

Argument 2: AA and BA are semi-balanced, AB and BB are balanced.

De Bruijn graph: building the 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 (k-mer) length k: 5

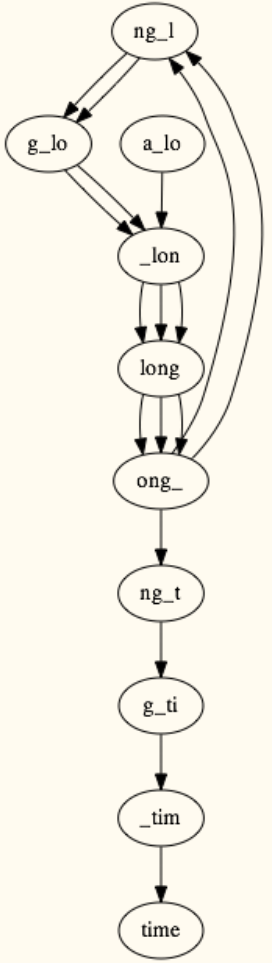
-Start with an input string:

```
a_long_long_long_time
```

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

```
      long_
   ↙      ↘
long    ong_
```

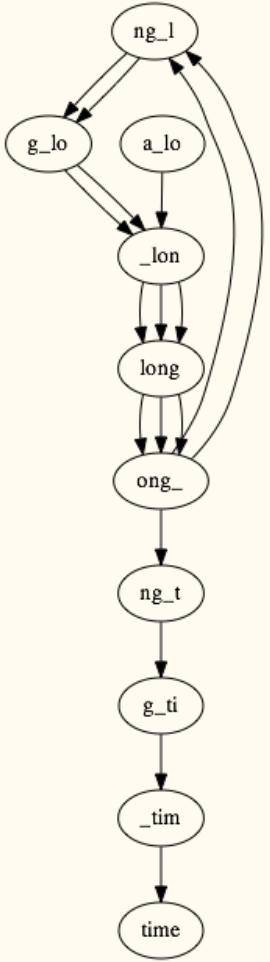
-Add k-1 mers as nodes to De Bruijn graph (if not already there), add edge from left k-1 mer to right k-1 mer.



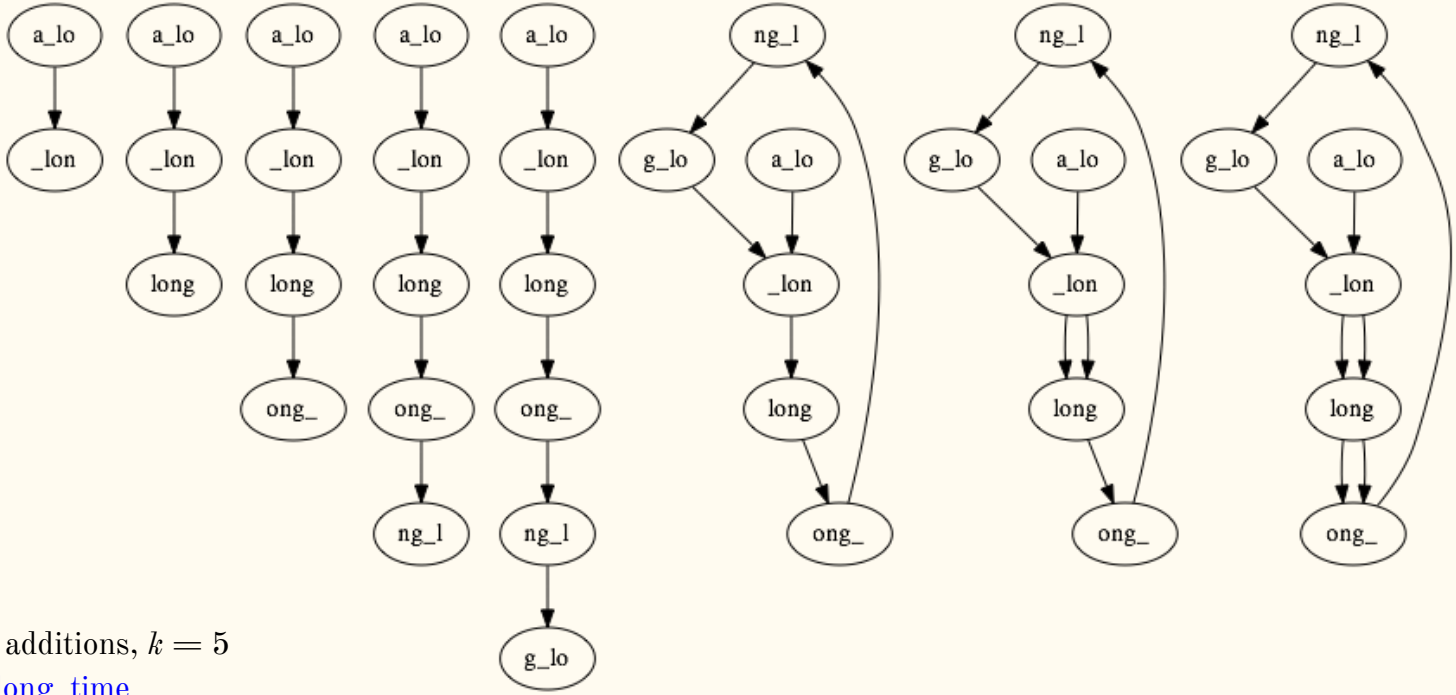
De Bruijn graph: adding edges

Connect one k-1-mer to another using a directed edge if the suffix of the former equals the prefix of the latter—that is, if the two k-1-mers completely overlap except for one nucleotide at each end. Label the edge with this k-mer.

Example: If some k-mer (e.g., ATG) has prefix x (e.g., AT) and suffix y (e.g., TG), connect node x to node y with a directed edge, and label the edge with this k-mer.

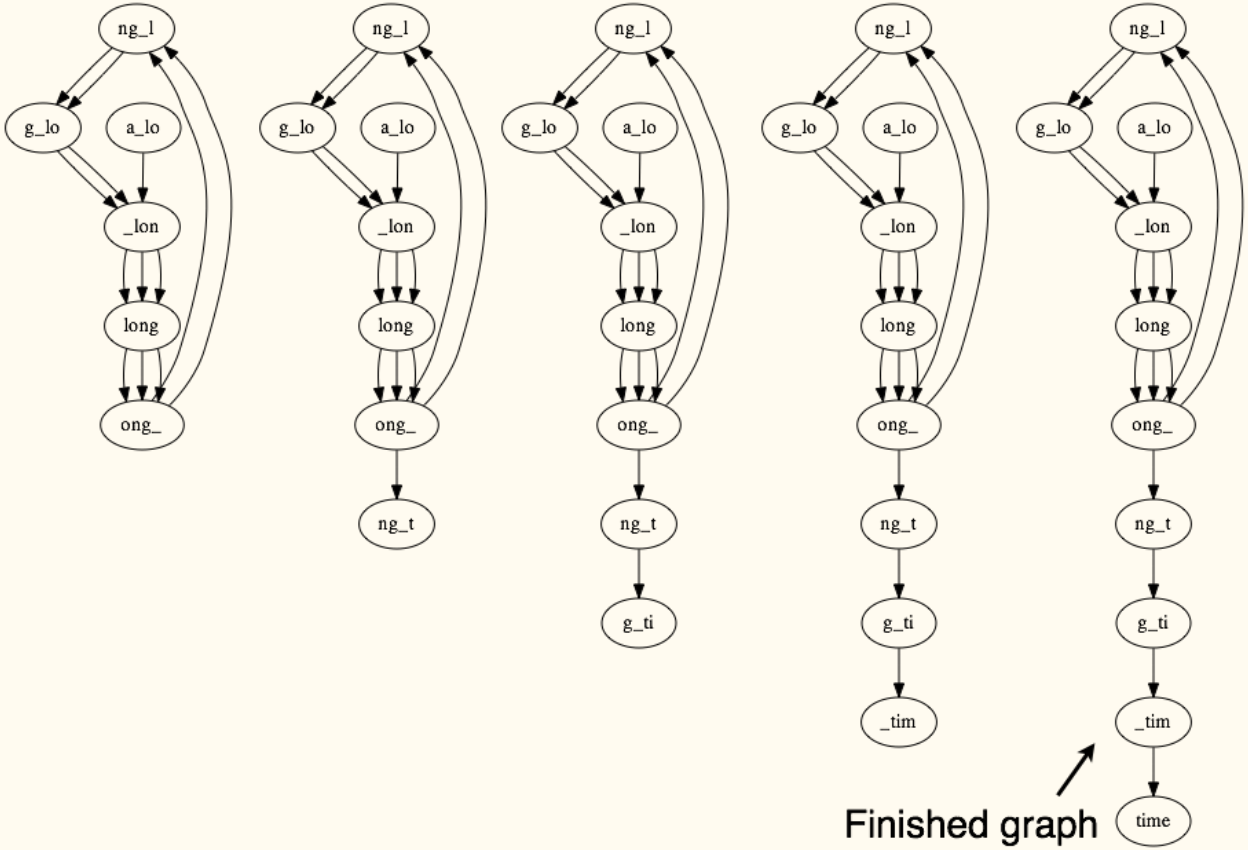


De Bruijn graph: building the graph



First 8 k-mer additions, $k = 5$
a_long_long_long_time

De Bruijn graph: building the graph



Last 5 k-mer additions, $k = 5$

a_long_long_long_time

De Bruijn graphs: graph traversal

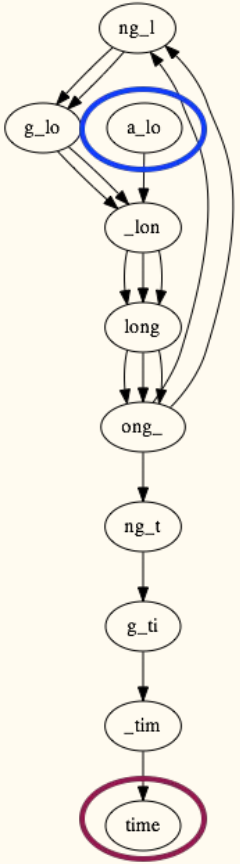
With perfect sequencing, this procedure always yields an Eulerian graph. Why?

Node for k-1-mer from left end is semi-balanced with one more outgoing edge than incoming *

Node for k-1-mer at right end is semi-balanced with one more incoming than outgoing *

Other nodes are balanced since # times k-1-mer occurs as a left k-1-mer = # times it occurs as a right k-1-mer

* Unless genome is circular

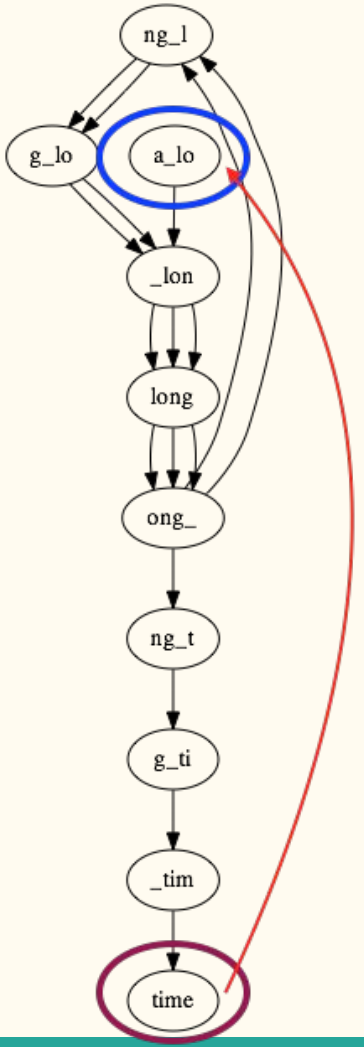


De Bruijn graphs: graph traversal

In order to be able to traverse graph it should contain Eulerian cycle or walk(path)*.

<u>First</u>, convert Eulerian walk to Eulerian cycle by adding edge between the two semi balanced nodes.

This is why (in this lecture) we don't distinguish Eulerian from semi-eulerian graph. Because semi-Eulerian can easily be converted to Eulerian.



De Bruijn graphs: graph traversal

For Eulerian graph, Eulerian walk can be found in O(| E |) time. | E | is # edges.

Several algorithms:

- Fleury's algorithm
- Cycle finding algorithm

De Bruijn graphs: graph traversal

- If graph is Eulerian by randomly picking any start edge and visiting all edges until we reach the node with no outgoing unvisited edges. (For Eulerian graph, this is a starting node)
- Important: **If C is a cycle in an Eulerian graph, then after removing all edges of C, remaining connected components are also Eulerian.**
- So, now we pick one of unvisited nodes, and visit remaining edges until we go back to starting edge.
- Repeat this procedure until there are no unvisited edges.

Euler showed how to connect two (or more) cycles into a single cycle.

De Bruijn graphs: graph traversal (procedure)

-Pick any start edge and visiting all edges until we reach the node with no outgoing unvisited edges (which is start node)

While not all edges are visited (traversed cycle is eulerian):

-From traversed cycle, pick any node with unvisited outgoing edges

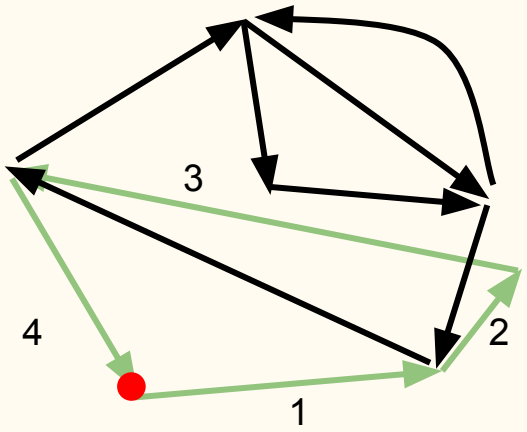
-Start traversing graph from this node*:

- First traverse all already traversed cycle(s)
- After that, from (newly chosen) start node, and traverse graph randomly

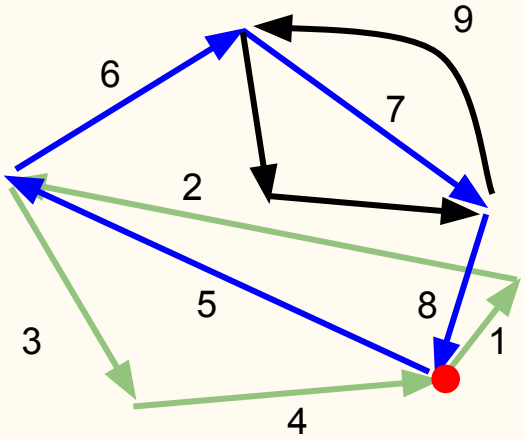
Note: If we've added edge to a graph, don't forget to remove this edge from path when we finish traversal.

*In practice there are multiple ways of doing this, one of the way is just shifting (cyclic rotation) the list of already visited nodes so that this node is the beginning of the list. Then we just traverse untraversed edges starting from this node, and append it to list. Repeat until all nodes are traversed.

De Bruijn graphs: graph traversal

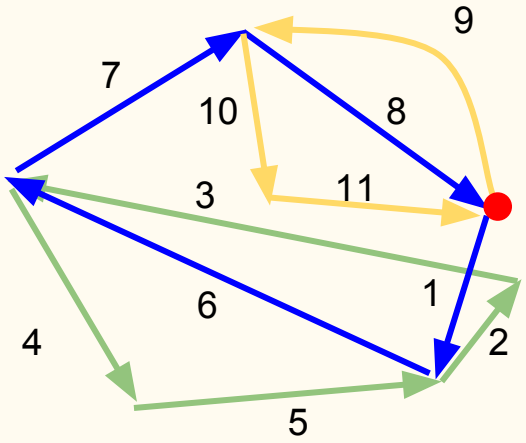


First try



Second try.

We move along previously traversed path to find a new starting point without previously traversed edges.
Then we first traverse first path again, and after that we randomly traverse unvisited edges.



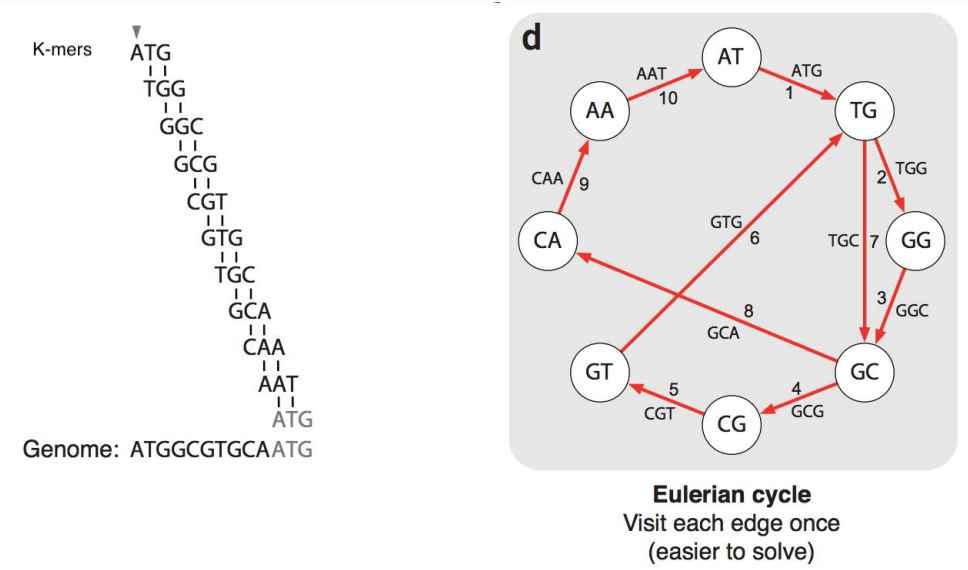
Third try

Again, we move along the previously traversed cycle to find a new starting node.

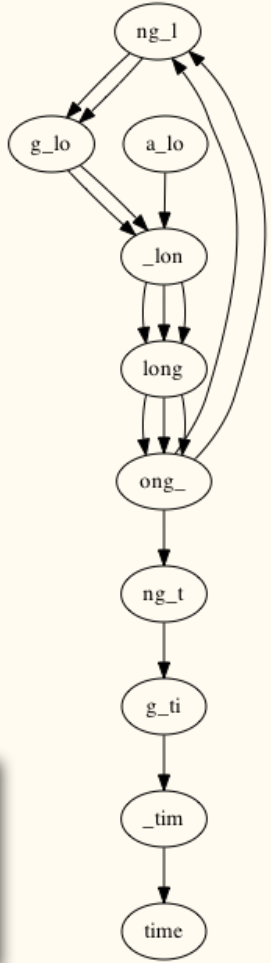
How do we reconstruct the sequence?

- Sequence (S) begins with the label of first **node** on the path, and follows concatenation (in visiting order) last character of each visited edge (or node).

Example: When walking thorough graph on our example:
AT+G+G+C+G+T+G+C+A+A+T



De Bruijn graphs: graph traversal



Example where Eulerian walk gives correct answer for small k whereas Greedy-SCS could spuriously collapse repeat:

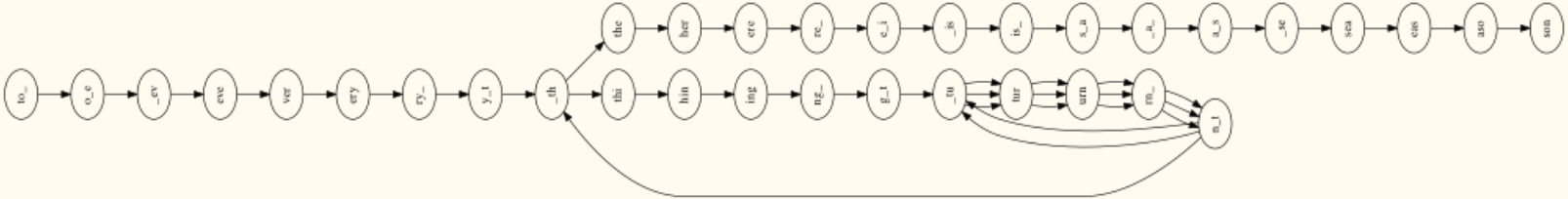
```
>>> G = DeBruijnGraph(["a_long_long_long_time"], 5)
>>> print G.eulerianWalkOrCycle()
['a_lo', '_lon', 'long', 'ong_', 'ng_l', 'g_lo', '_lon', 'long', 'ong_', 'ng_l',
'g_lo', '_lon', 'long', 'ong_', 'ng_t', 'g_ti', '_tim', 'time']
```

Note: this is for idealized case where we now multiplicity of k-mers. In practice, we don't know this and give up on repeats

De Bruijn graphs: graph traversal

Another example Eulerian walk:

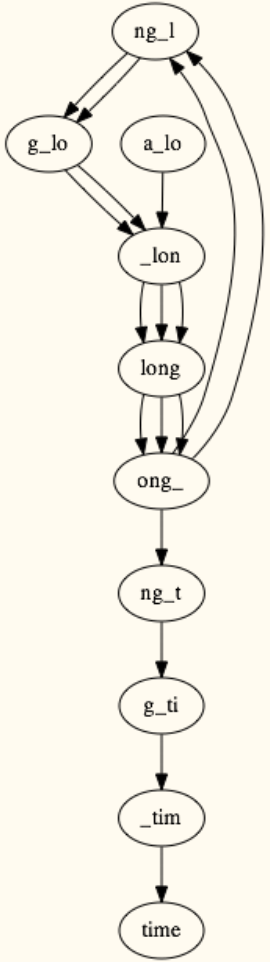
```
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 4)
>>> path = G.eulerianWalkOrCycle()
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
to_every_thing_turn_turn_turn_there_is_a_season
```



De Bruijn graph: practical limitations

Assuming perfect sequencing, procedure yields graph with Eulerian walk that can be found efficiently.

We saw cases where Eulerian walk corresponds to the original superstring. Is this always the case?



De Bruijn graph: practical limitations

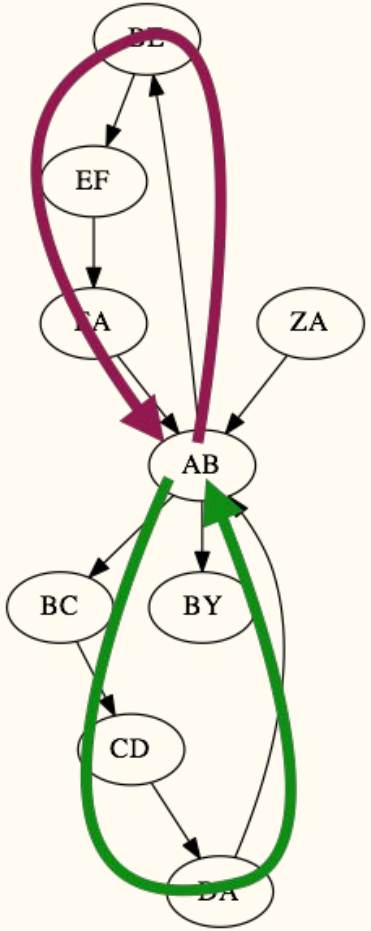
No: graph can have multiple Eulerian walks, only one of which corresponds to original superstring.

Right: graph for ZABCDABEFABY, $k = 3$

Alternative Eulerian walks:
ZA → AB → BE → EF → FA → AB → BC → CD → DA → AB → BY
ZA → AB → BC → CD → DA → AB → BE → EF → FA → AB → BY

These correspond to two edge-disjoint directed cycles joined by node AB

AB is a repeat: ZABCDABEFABY



De Bruijn graph: practical limitations

For k=4:

```
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 4)
>>> path = G.eulerianWalkOrCycle()
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
to_every_thing_turn_turn_turn_there_is_a_season
```

For k=3:

```
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 3)
>>> path = G.eulerianWalkOrCycle()
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
to_every_turn_turn_thing_turn_there_is_a_season
```

De Bruijn graph: practical limitations

This is the first sign that Eulerian walks can't solve all our problems.

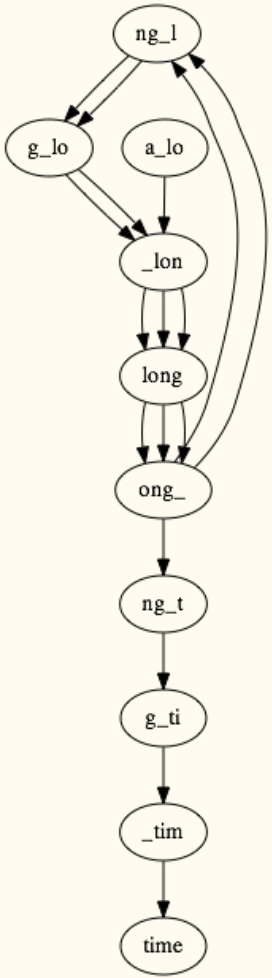
Other signs emerge when we think about how actual sequencing differs from our idealized construction.

- Imperfect coverage (generating all k-mers)
- Reads are error prone
- Single/multiple chromosomes
- Known multiplicity of k-mers
- Repeats (multiplicities of k-mers are unknown ATGCATGC)
- Zygosity

De Bruijn graph: practical limitations

- Gaps in coverage can lead to *disconnected* graph

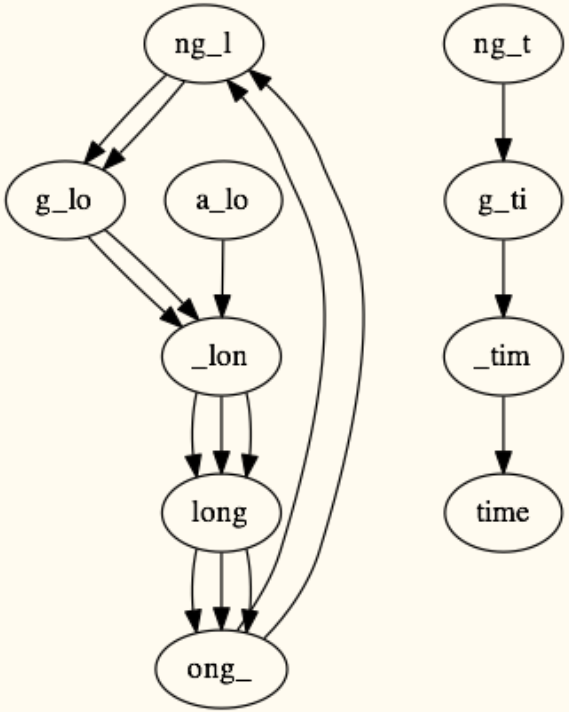
Graph for a_long_long_long_time, $k = 5$:



De Bruijn graph: practical limitations

- Gaps in coverage can lead to *disconnected* graph

Graph for a_long_long_long_time, $k = 5$ but *omitting* ong_t :



Connected components are individually Eulerian, overall graph is not.

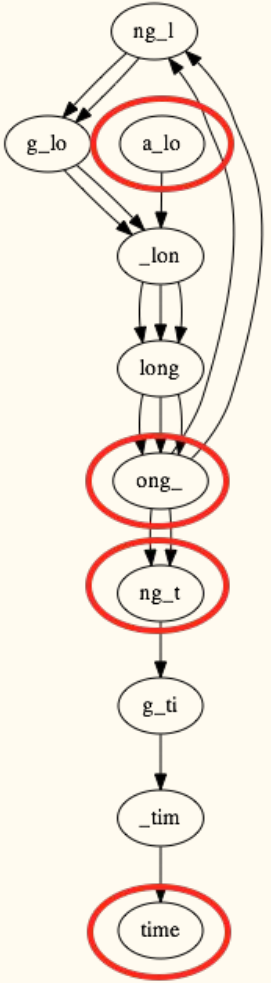
Luckily even if read itself is not present in genome, it's k-mers might be.

De Bruijn graph: practical limitations

- *Differences* in coverage also lead to non-Eulerian graph

Graph for a_long_long_long_time, $k = 5$ but with *extra copy* of ong_t :

Graph has 4 semi-balanced nodes, isn't Eulerian.

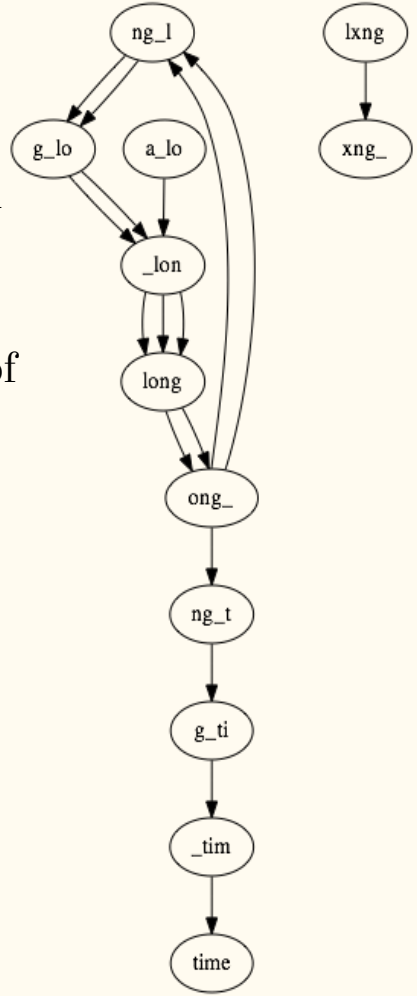


De Bruijn graph: practical limitations

- Errors and *differences between chromosomes* also lead to non-Eulerian graphs

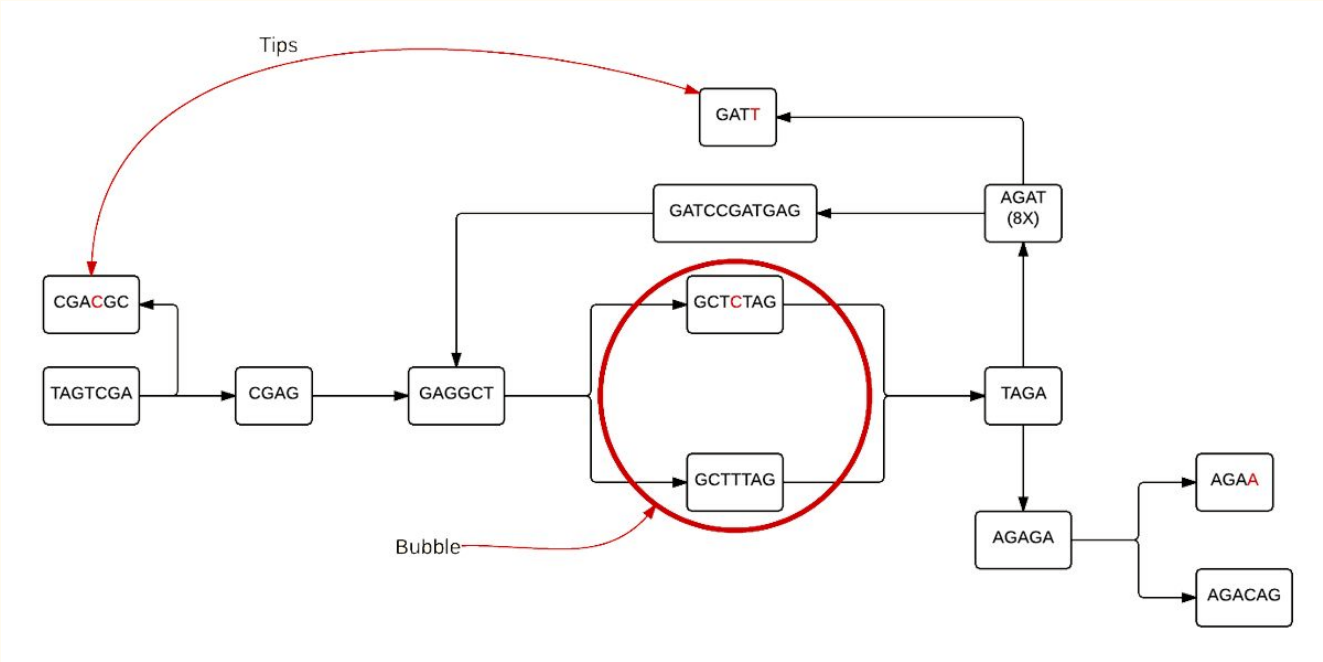
Graph for a_long_long_long_time, $k = 5$ but with error that turns a copy of long_ into lxng_

Graph is not connected;
largest component is not Eulerian



De Bruijn graph: practical limitations

- Errors also lead to bubbles and tips in graph



De Bruijn graph

Casting assembly as Eulerian walk is appealing, but not practical:

- Uneven coverage, sequencing errors, etc. make graph non-Eulerian
- Even if graph were Eulerian, repeats yield many possible walks

Kingsford, Carl, Michael C. Schatz, and Mihai Pop. "Assembly complexity of prokaryotic genomes using short reads." BMC bioinformatics 11.1 (2010): 21.

De Bruijn Superwalk Problem (DBSP) is an improved formulation where we seek a walk over the De Bruijn graph, where walk contains each read as a subwalk

Proven NP-hard!

Medvedev, Paul, et al. "Computability of models for sequence assembly." Algorithms in Bioinformatics. Springer Berlin Heidelberg, 2007. 289-301.

De Bruijn graph

In practice, De Bruijn graph-based tools give up on unresolvable repeats and yield fragmented assemblies, just like OLC tools.

But first we note that using the De Bruijn graph representation has other advantages...

De Bruijn graph

Say a sequencer produces d reads of length n from a genome of length m

$d = 6 \times 10^9$
n = 100 nt
$m = 3 \times 10^9$ nt ≈ human

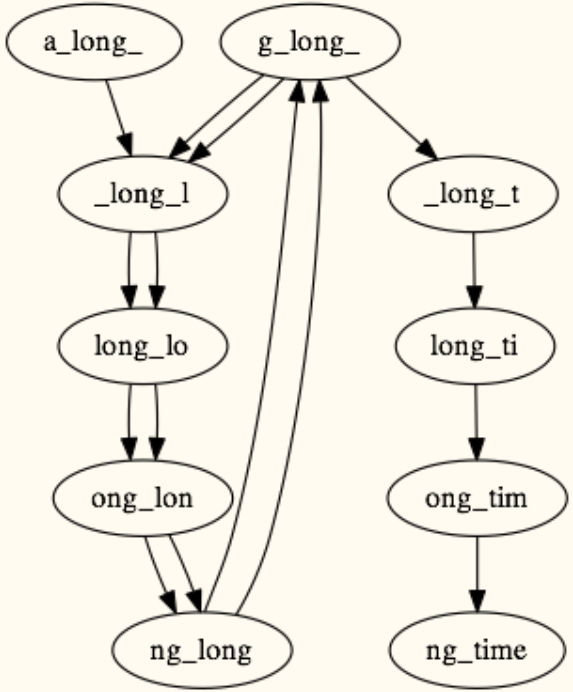
} ≈ 1 sequencing run

To build a De Bruijn graph in practice:

Pick k. Assume k $\leq$ shortest read length (k = 30 to 50 is common).
For each read:
    For each k-mer:
        Add k-mer's left and right k-1-mers to graph if not there already.
        Draw an edge from left to right k-1-mer.

De Bruijn graph

Pick k = 8



Genome: a_long_long_long_time

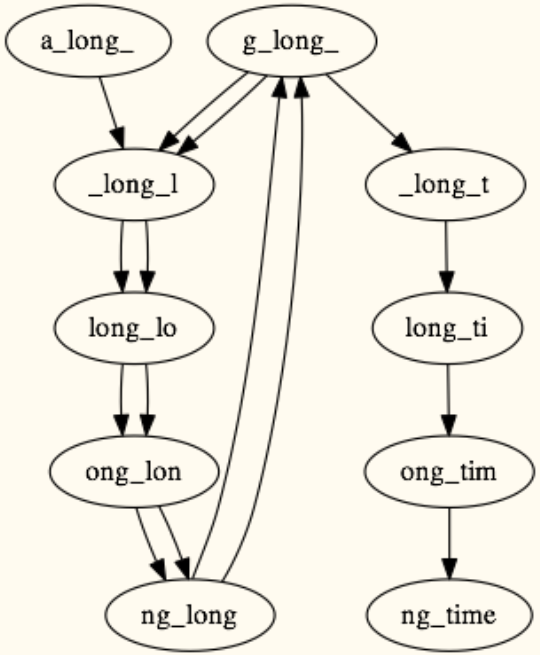
Reads: a_long_long_long, ng_long_l, g_long_time

K-mers: a_long_l
_long_lo
long_lon
ong_long
ng_long_
g_long_l
_long_lo
long_lon
ong_long
ng_long_
g_long_l
_long_ti
long_tim
ong_time
g_long_t

Given n (# reads), N (total length of all reads) and k, and assuming k < length of shortest read:

- Exact number of k-mers: N - n (k -1) O(N)
- This is also the number of edges, | E |
- Number of nodes | V | is at most 2 • | E |, but typically much smaller due to repeated k-1-mers

De Bruijn graph



How much work to build graph?

For each k-mer, add 1 edge and up to 2 nodes

Reasonable to say this is O(1) expected work

Assume hash map encodes nodes & edges

Assume k-1-mers fit in O(1) machine words, and hashing O(1) machine words is O(1) work

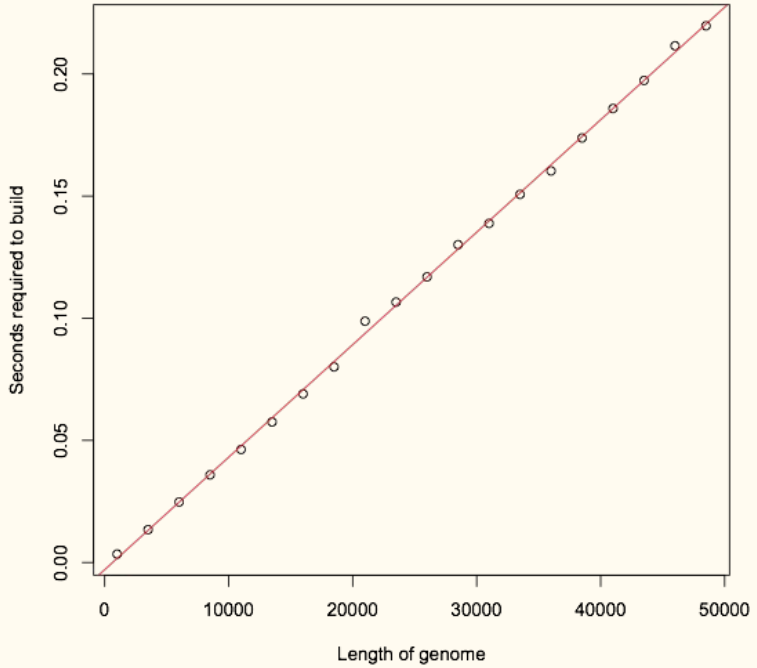
Querying / adding a key is O(1) expected work

O(1) expected work for 1 k-mer, O(N) overall

De Bruijn graph

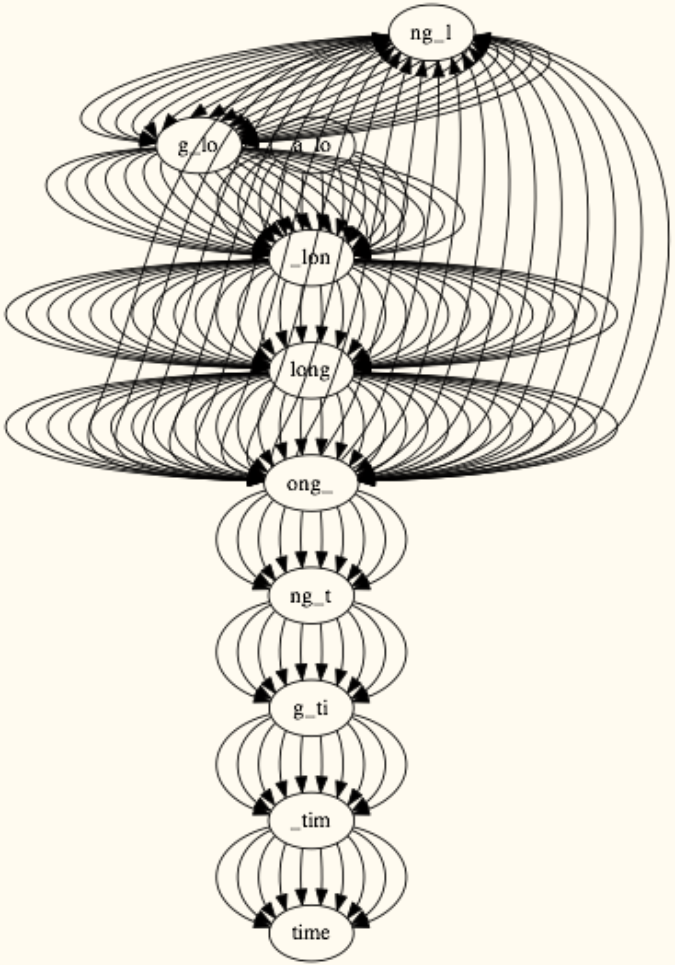
Timed De Bruijn graph construction applied to progressively longer prefixes of lambda phage genome (48k bp), k = 14

O(N) expectation appears to work in practice, at least for this small example.



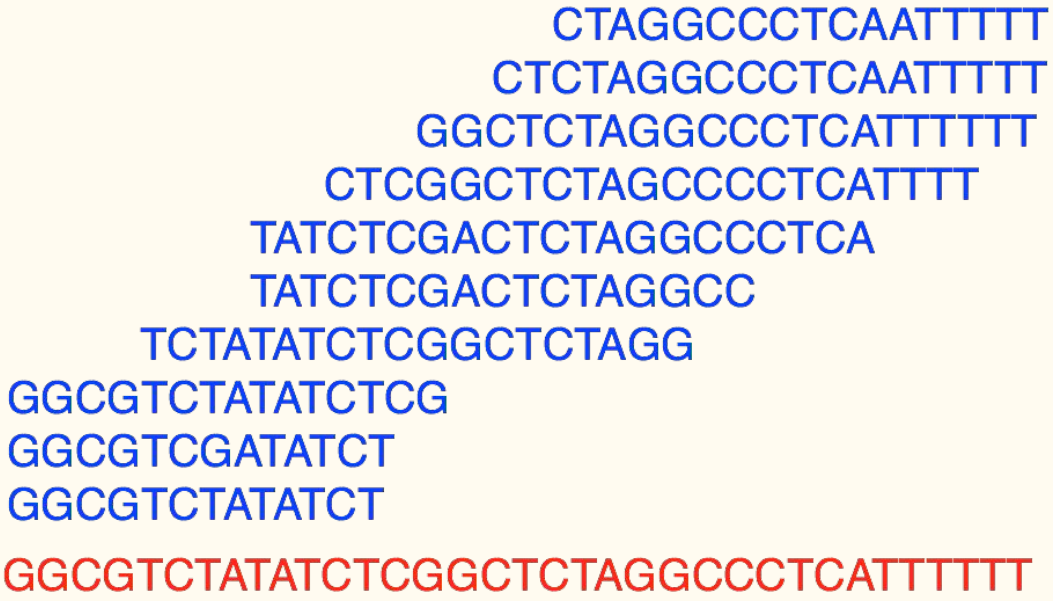
De Bruijn graph

In typical assembly projects, average coverage is ~ 30 - 50.



De Bruijn graph

Recall average coverage: average # reads covering a genome position.



177 nucleotides

35 nucleotides

Average coverage = 177 / 35 ≈ 5x

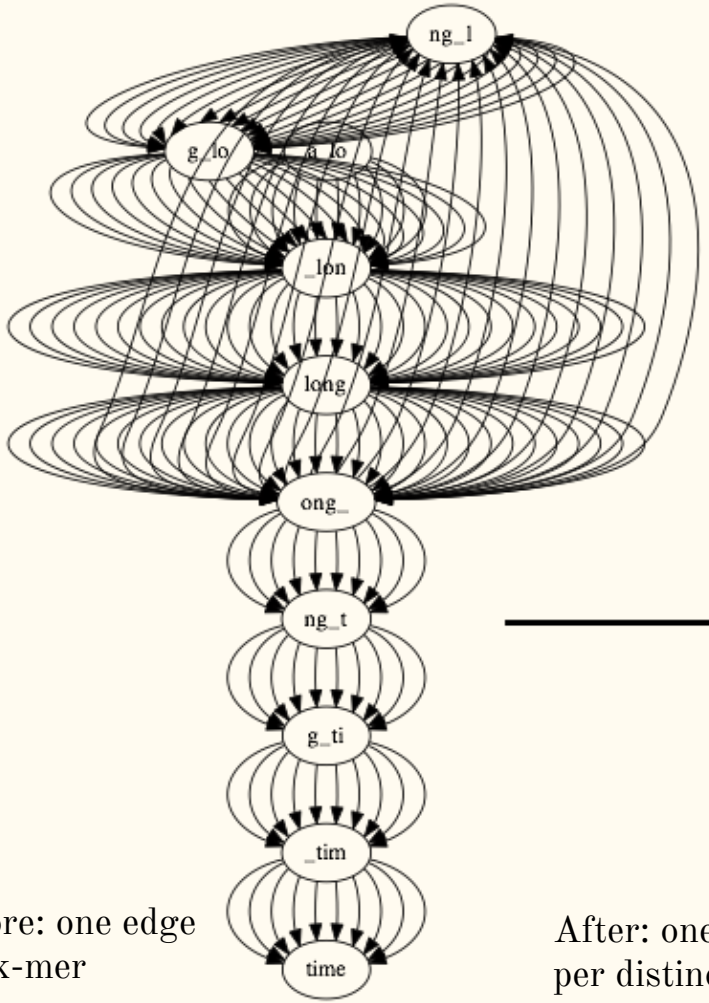
De Bruijn graph

In typical assembly projects, average coverage is ~ 30 - 50

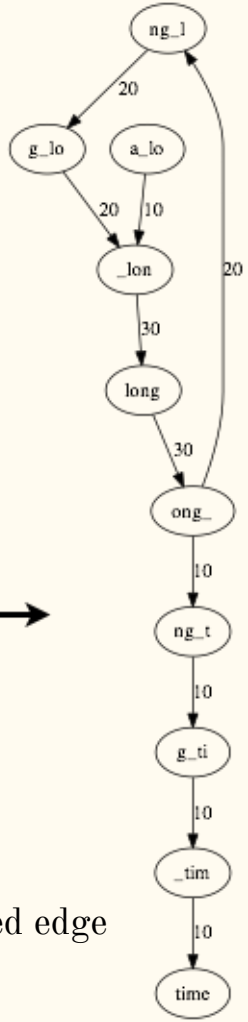
Same edge might appear in dozens of copies; let's use edge *weights* instead

Weight = # times *k*-mer occurs

Using weights, there's one *weighted* edge for each *distinct k*-mer.



Before: one edge per k-mer



After: one weighted edge per distinct k-mer

De Bruijn graph

of nodes and edges both O(N); N is total length of all reads

Say (a) reads are error-free, (b) we have one weighted edge for each distinct k-mer, and (c) length of genome is G

There's one node for each distinct k-1-mer, one edge for each distinct k-mer

Can't be more distinct k-mers than there are k-mers in the genome; likewise for k-1-mers

So # of nodes and edges are also both O(G)

Combine with the O(N) bound and the # of nodes and edges are both O(min(N, G))

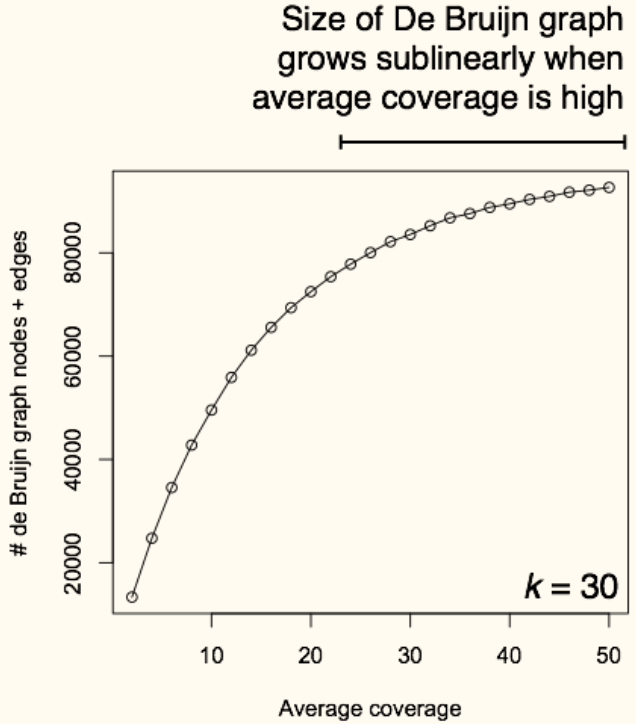
De Bruijn graph

With high average coverage, O(G) size bound is advantageous

Genome = lambda phage (~ 48.5 K nt)

Draw random k-mers until target average coverage is reached (x axis)

Build De Bruijn graph and total the # of nodes and edges (y axis)



De Bruijn graph

What De Bruijn graph advantages have we discovered?

Can be built in O(N) expected time, N = total length of reads

With perfect data, graph is O(min(N, G)) space; G = genome length

Note: when average coverage is high, $G \ll N$

Compares favorably with overlap graph.

Space is O(N + a).

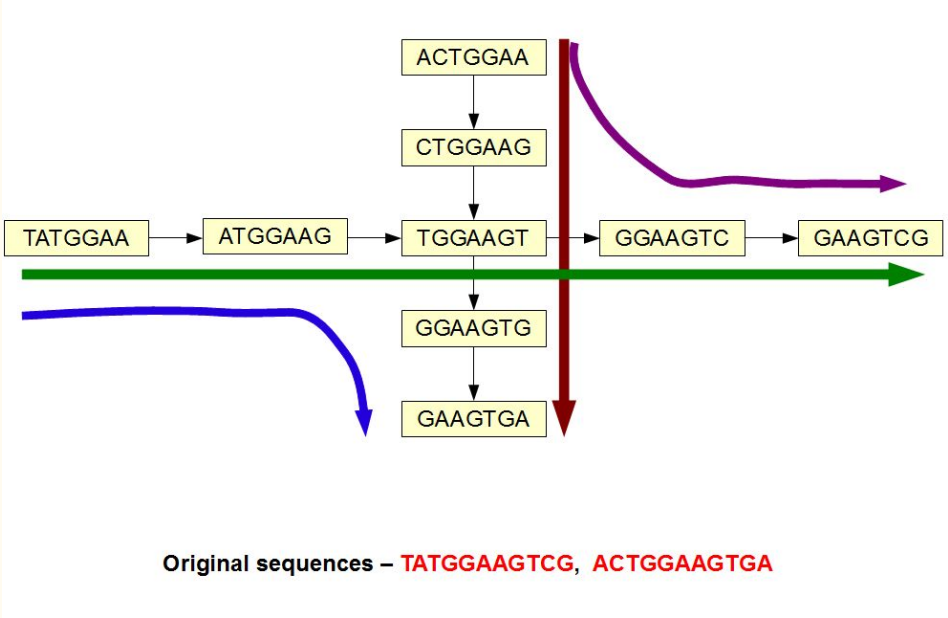
Fast overlap graph construction (suffix tree) is O(N + a) time a is $O(n^2)$

De Bruijn graph

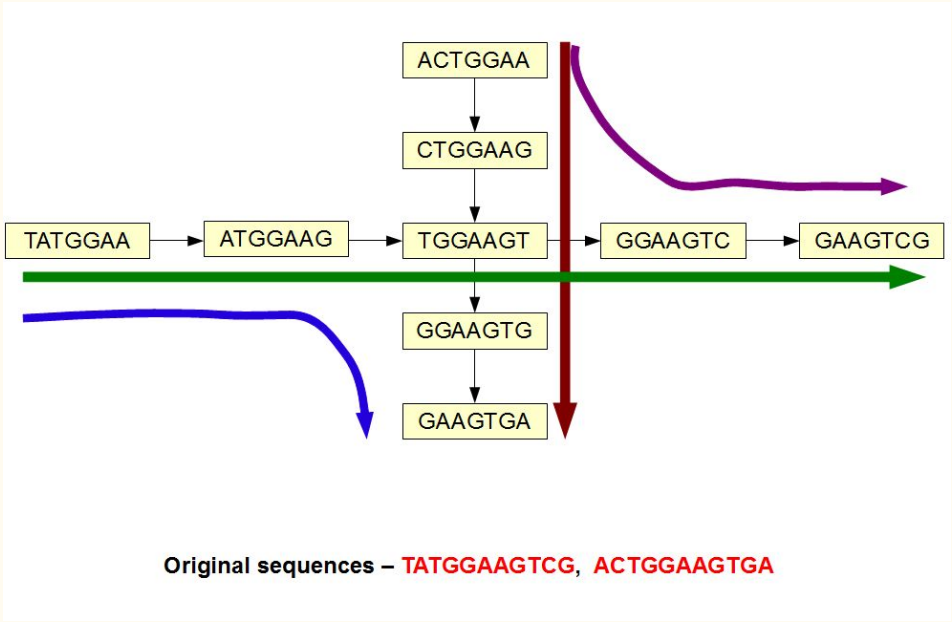
- How do we choose **k**?
- Large **k**:
  - sequencing errors bigger problem
  - graph less connected

- Small **k**:
  - ambiguous paths

- Balance, try different values for **k**



De Bruijn graph



k=7

k=9

String vs De Bruijn graph

- Which is better?
  - String graphs capture whole read information
  - de Bruijn graphs are conceptually simpler:
    - single node length
    - single overlap definition

- Historically, **string graphs** were used for **long reads**
- and **De Bruijn** graphs for **short reads**

DBG-OLC tradeoff

What did we give up?

Reads are immediately split into shorter k-mers; can't resolve repeats as well as overlap graph.
Read coherence is lost. Some paths through De Bruijn graph are inconsistent with respect to input reads.
Only a very specific type of "overlap" is considered, which makes dealing with errors more complicated, as we'll see.

This is the OLC ↔ DBG tradeoff

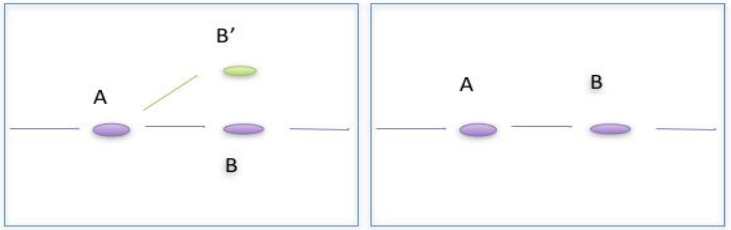
Single most important benefit of De Bruijn graph is the O(min(G, N)) space bound, though we'll see this comes with large caveats.

De Bruijn graph summary

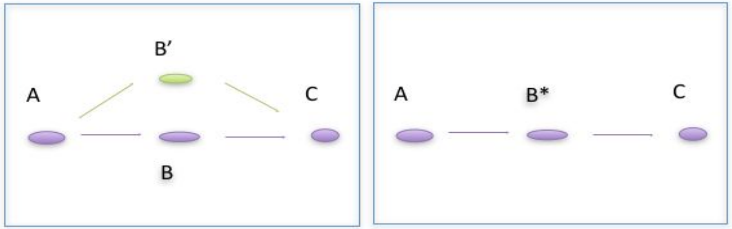
- Given any sequence and k-mer size, we can create a De Bruijn graph in an unique manner.
- The other direction is not true. All De Bruijn graphs cannot be resolved into unique sequences. Unless the De Bruijn graph is in its simplest form, it usually resolves into many possible sequences.
- Larger the k-mers, more easy it is to convert the De Bruijn graph into an unique sequence.

Graph topology based error eorrection

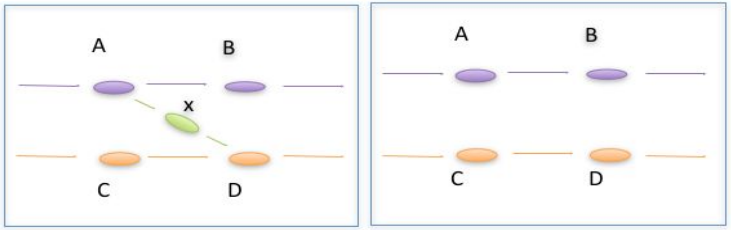
- Errors at end of read
  - Trim off ‘dead-end’ tips



- Errors in middle of read
  - Pop Bubbles



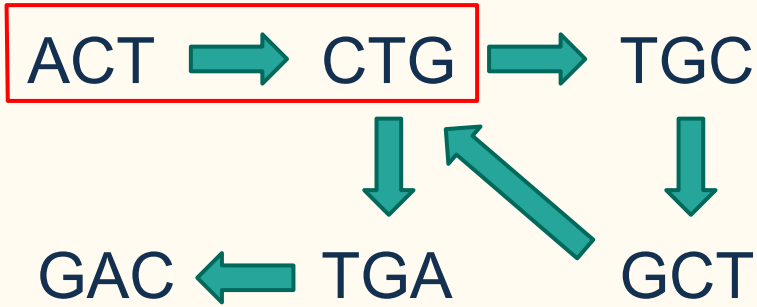
- Chimeric Edges
  - Clip short, low coverage nodes



Contigs construction

Contigs construction in practice: return a set of paths covering the graph, such that all possible assemblies contain these paths.

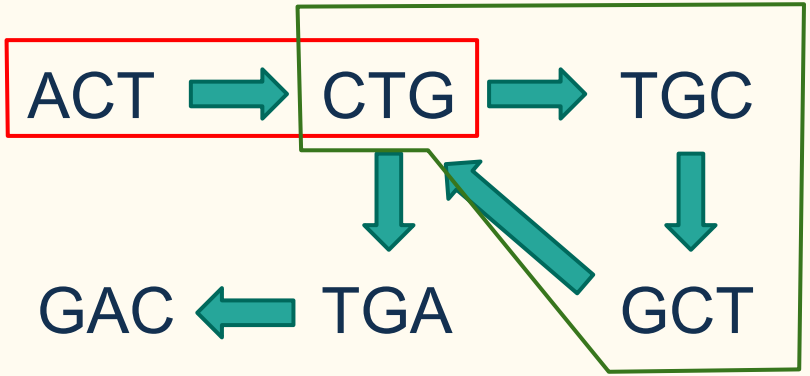
ACTG



Contigs construction

Contigs construction in practice: return a set of paths covering the graph, such that all possible assemblies contain these paths.

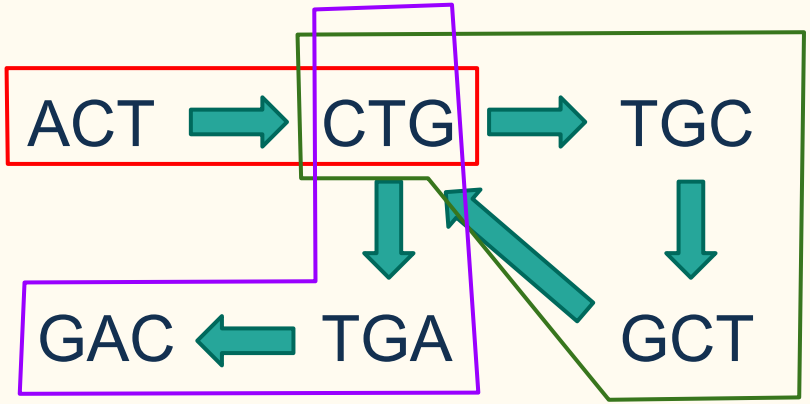
ACTG
CTGCT



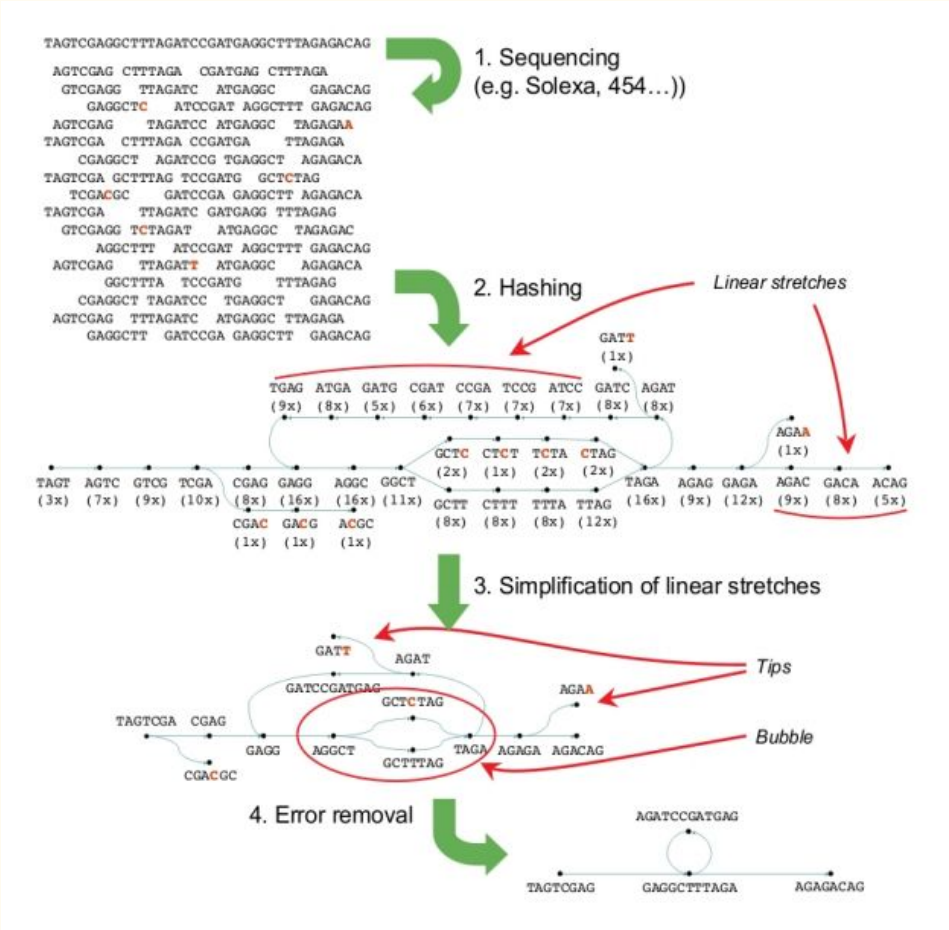
Contigs construction

Contigs construction in practice: return a set of paths covering the graph, such that all possible assemblies contain these paths.

ACTG
CTGCT
CTGAC



Contig construction in practice

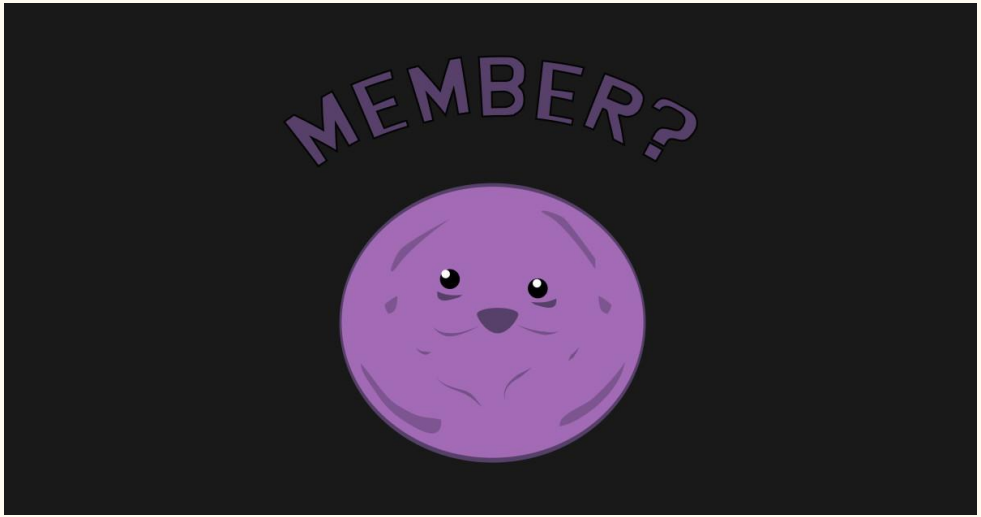


Contig construction

Potential assemblies:

TAGTCGAGGAGGCTTTAGAAGATCCGATGAGGAGGCTTTAGAAGAGACAG
TAGTCGAGGAGGCTTTAGAAGATCCGATGAGGAGGCTTTAGAAGATCCGATGAGGAGGCTTTAGAAG
AGACAG
…

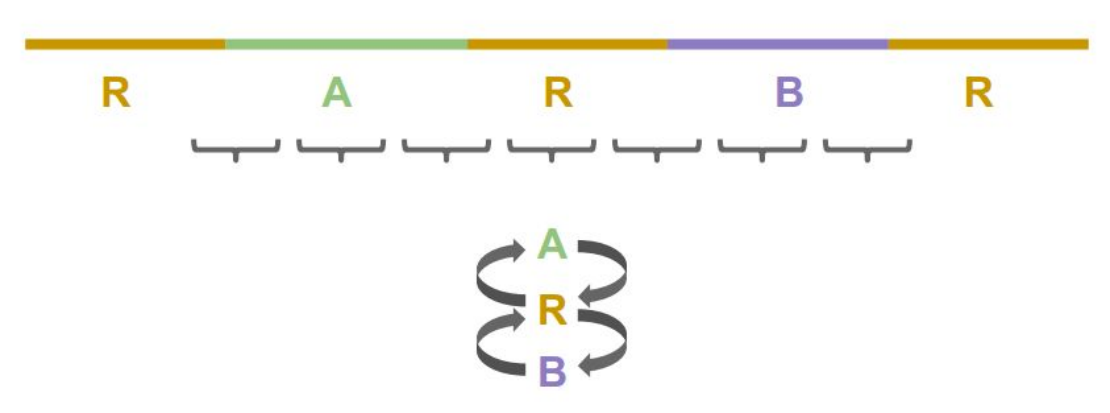
De novo assembly



Repeats foil assembly!

De novo assembly

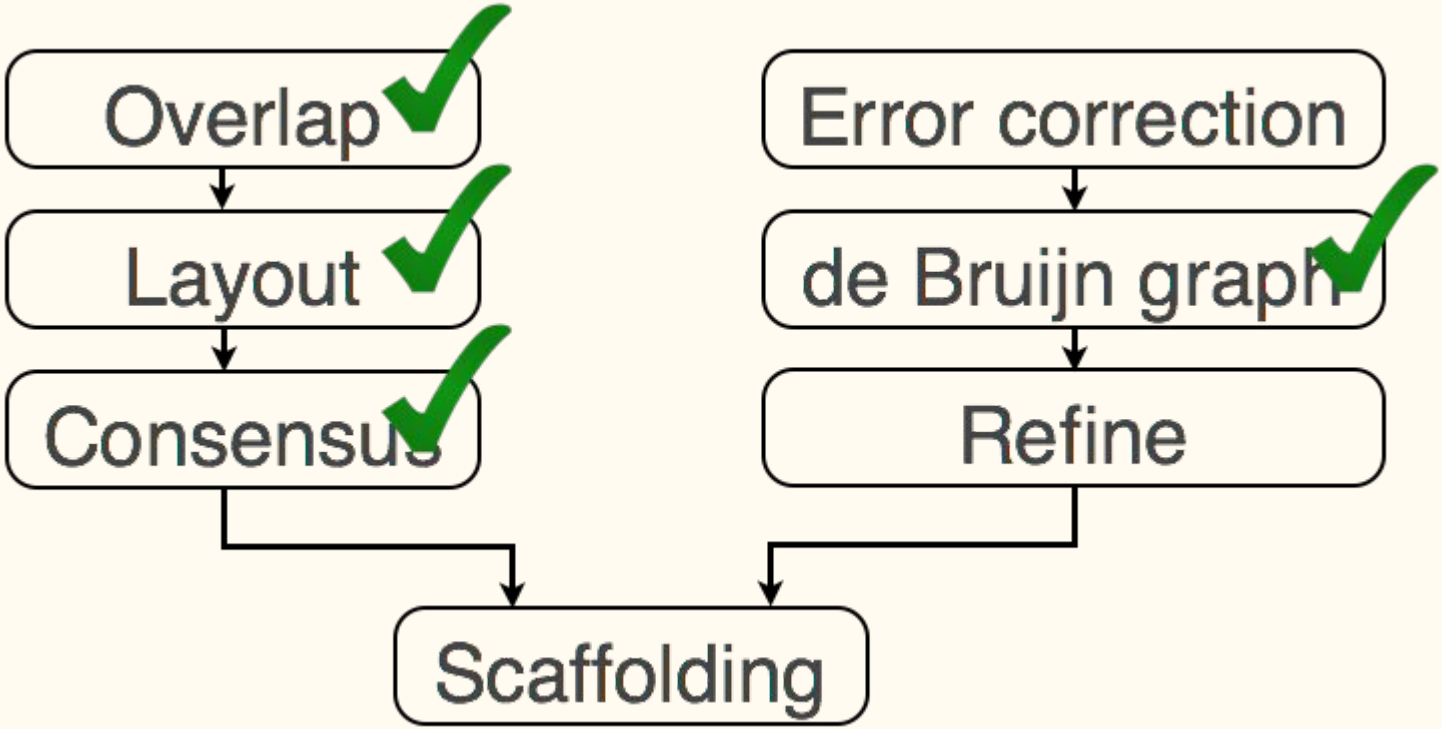
- “[...] **repeats** are the single biggest impediment to all assembly algorithms and sequencing technologies” - Koren 2012 Nature Biotechnology



Error correction

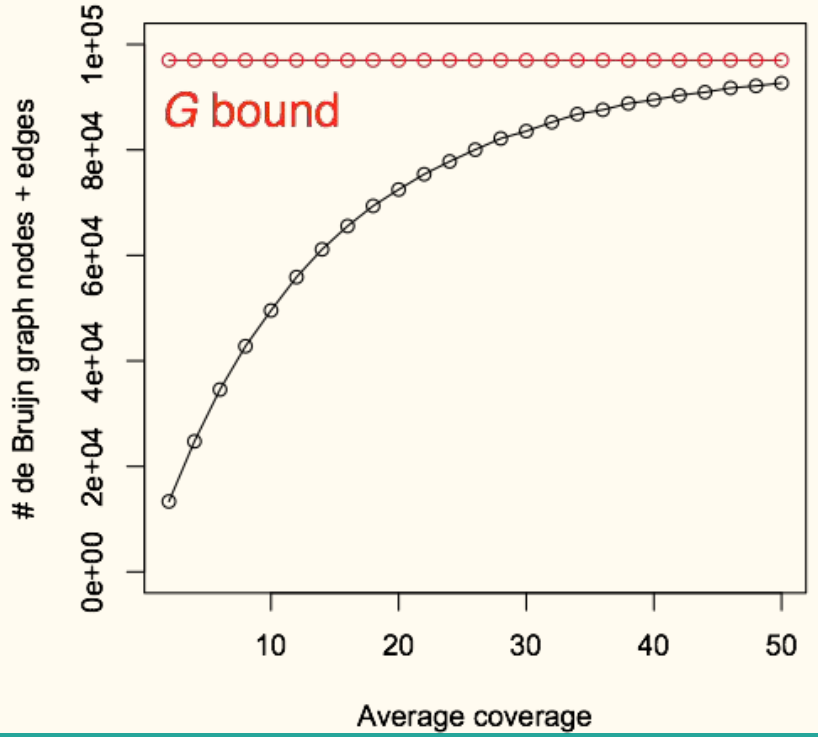
Assembly paradigms

1: Overlap-Layout-Consensus (OLC) assembly 2: de Bruijn graph (DBG) assembly



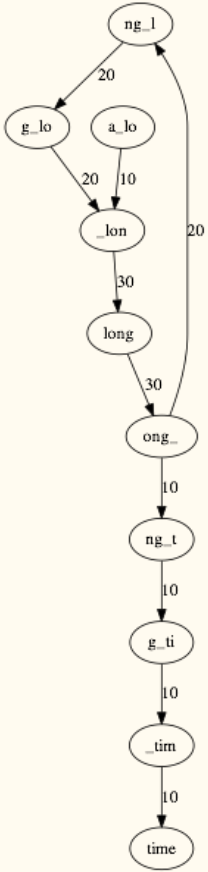
Error correction

When data is error-free, # nodes, edges in de Bruijn graph is O(min(G, N))



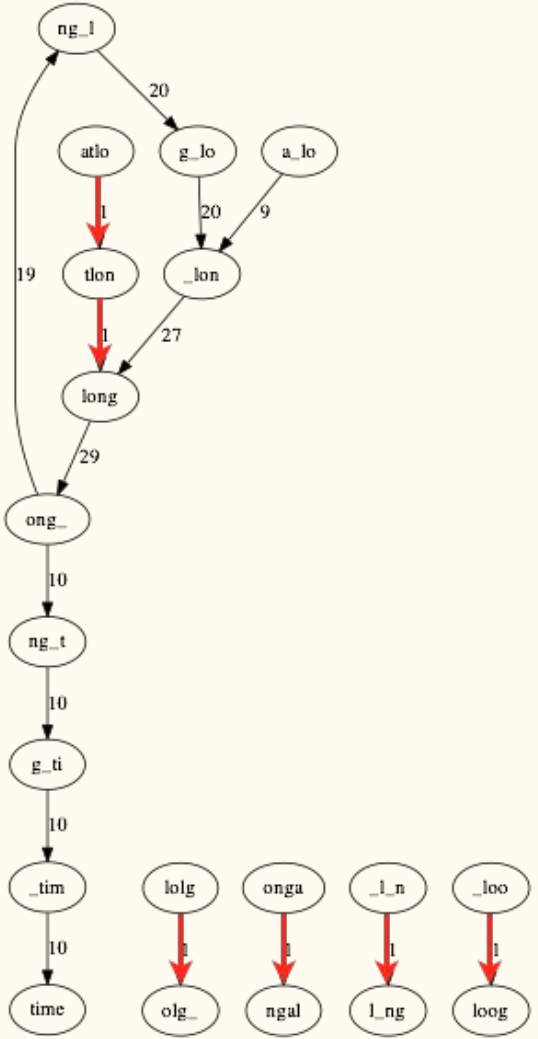
What about data with sequencing errors?

Error correction



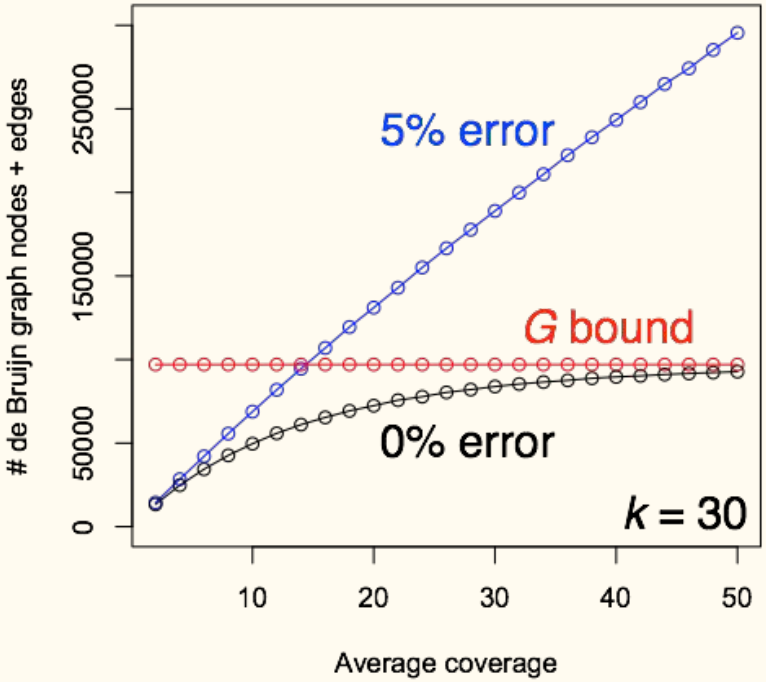
Take an example we saw (left) and mutate a k-mer character to a random other character with probability 1% (right)

6 errors result in 10 new nodes and 6 new *weighted* edges, all with weight 1



Error correction

As more k-mers overlap errors, # nodes, edges approach N

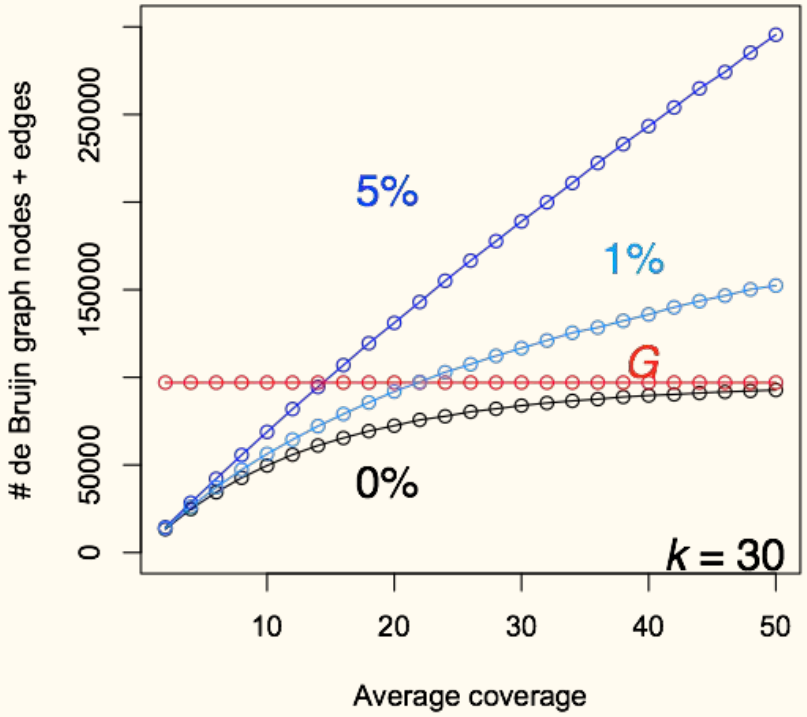


Same experiment as before but with 5% error added

Errors wipe out much of the benefit of the G bound

Instead of $O(\min(G, N))$, we have something more like $O(N)$

Error correction



Error correction

If we can correct sequencing errors up-front, we can prevent De Bruijn graph from growing much beyond the G bound.

How do we correct errors?

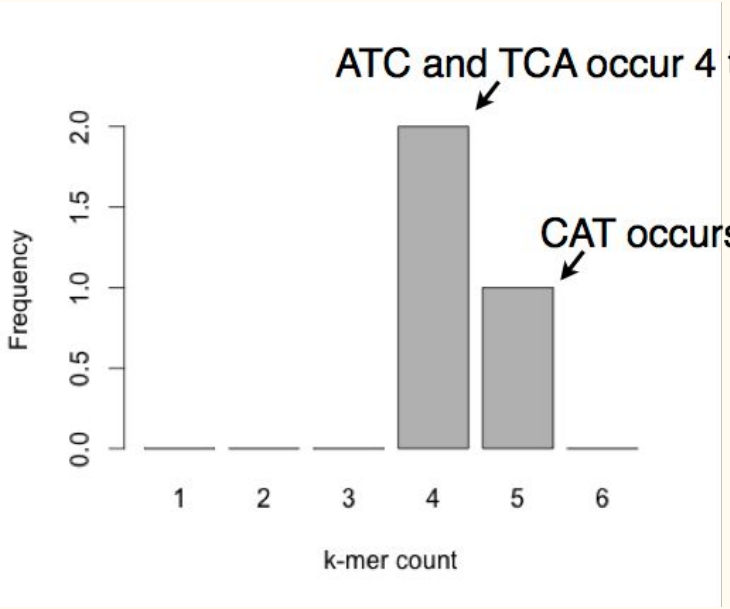
> Analogy: design a spell checker for a language you've never seen before. How do you come up with suggestions?

Error correction

k-mer count histogram:

x axis is an integer k-mer count, y axis is # distinct k-mers with that count

Right: such a histogram
for 3-mers of CATCATCATCATCAT:

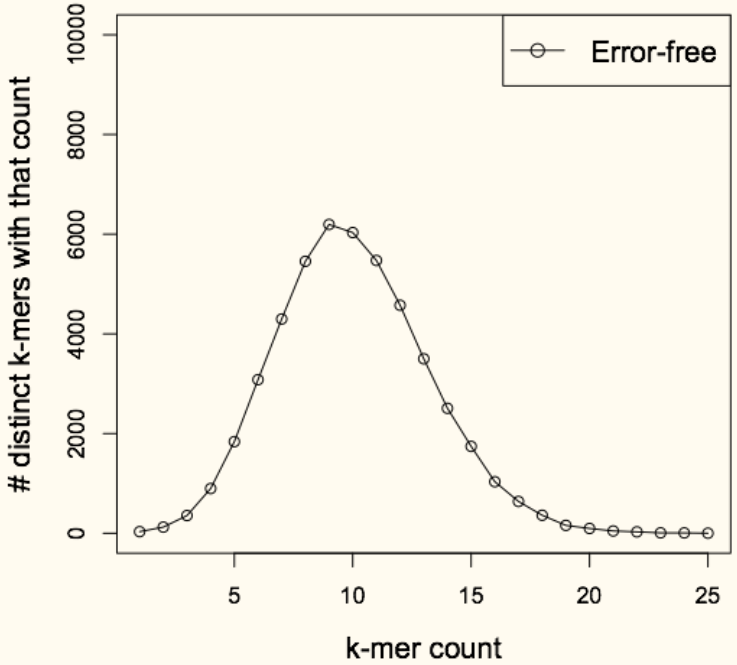


Error correction

Say we have error-free sequencing reads drawn from a genome. The amount of sequencing is such that average coverage = 200. Let $k = 20$.

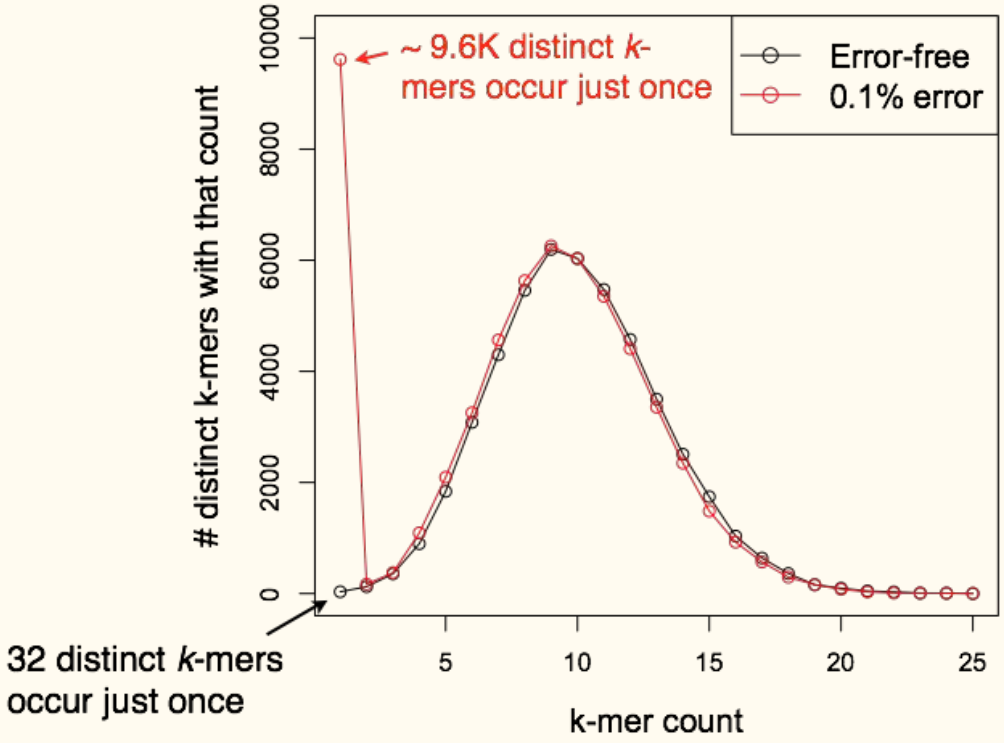
How would the picture change for data with 1% error rate?

Hint: errors usually change high-count k-mer into low-count k-mer



Error correction

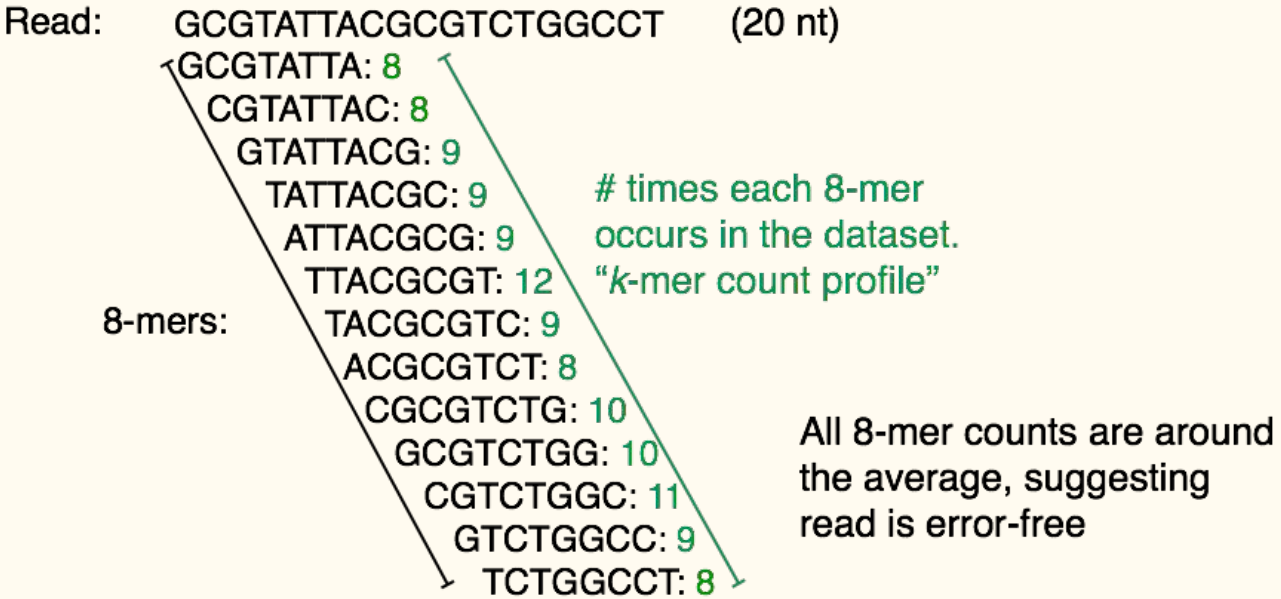
k-mers with errors usually occur fewer times than error-free *k*-mers



Error correction

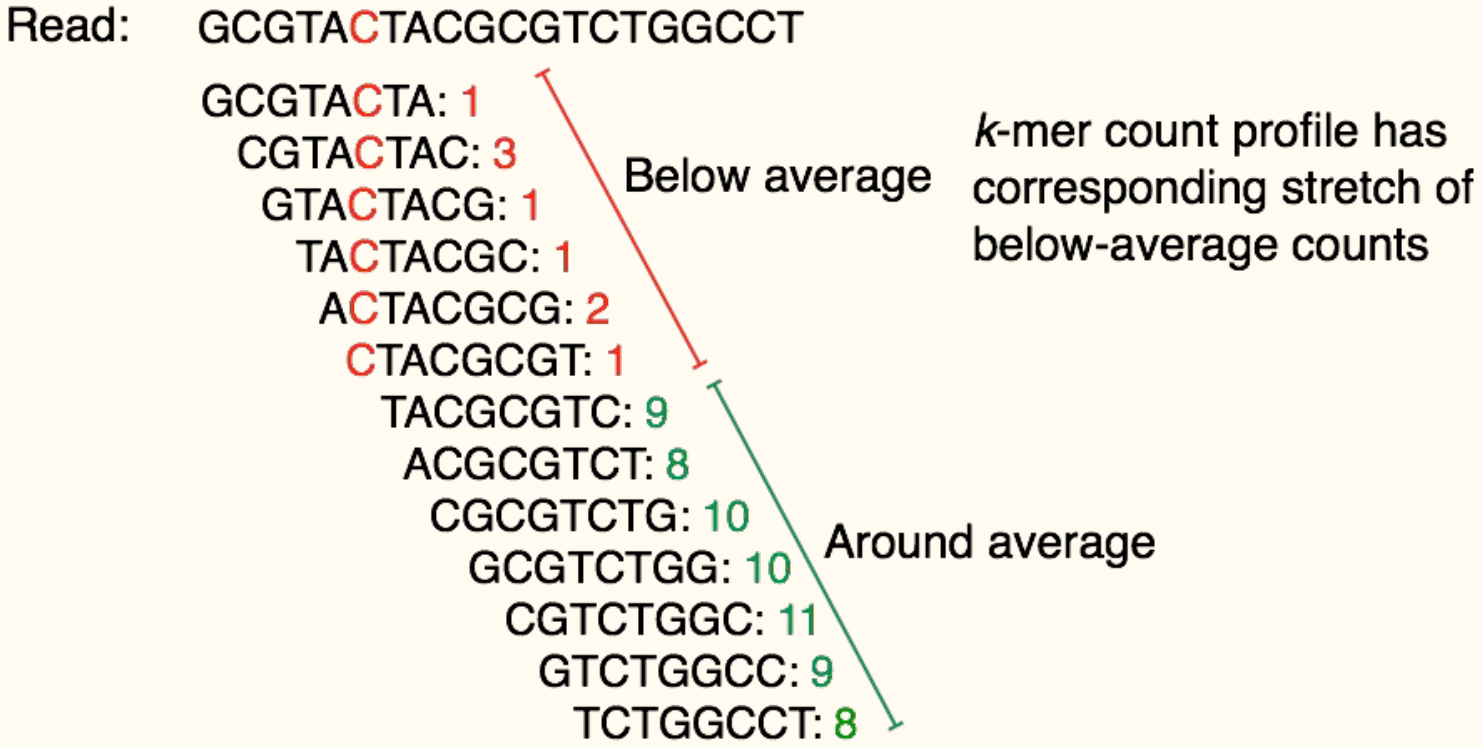
Idea: errors tend to turn frequent k-mers to infrequent k-mers, so corrections should do the reverse

Say we have a collection of reads where each distinct 8-mer occurs an average of ~10 times, and we have the following read:



Error correction

Suppose there's an error

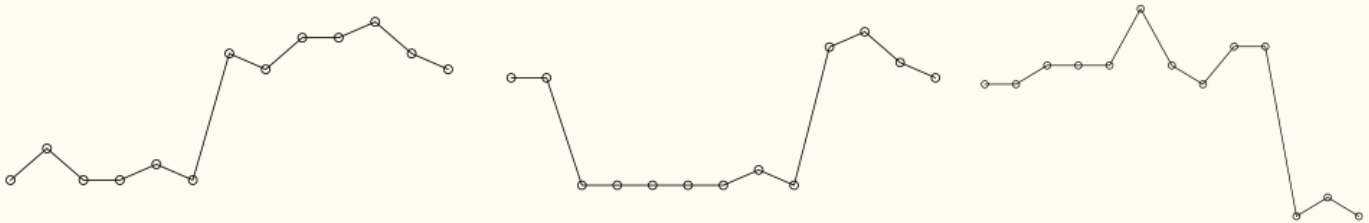


Error correction

k-mer count profiles when errors are in different parts of the read:

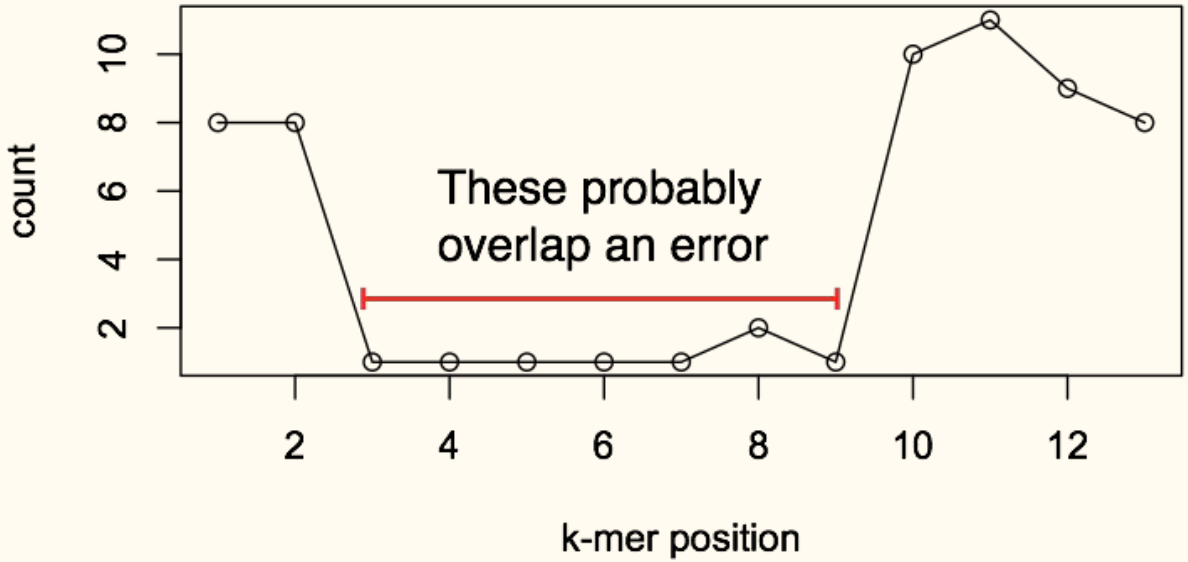
GCGTACTACGCGTCTGGCCTGCGTATTACACGTCTGGCCT GCGTATTACGCGTCTGGTCT

GCGTACTA: 1	GCGTATTA: 8	GCGTATTA: 8
CGTACTAC: 3	CGTATTAC: 8	CGTATTAC: 8
GTACTACG: 1	GTATTACA: 1	GTATTACG: 9
TACTACGC: 1	TATTACAC: 1	TATTACGC: 9
ACTACGCG: 2	ATTACACG: 1	ATTACGCG: 9
CTACGCGT: 1	TTACACGT: 1	TTACGCGT: 12
TACGCGTC: 9	TACACGTC: 1	TACGCGTC: 9
ACGCGTCT: 8	ACACGTCT: 2	ACGCGTCT: 8
CGCGTCTG: 10	CACGTCTG: 1	CGCGTCTG: 10
GCGTCTGG: 10	GCGTCTGG: 10	GCGTCTGG: 10
CGTCTGGC: 11	CGTCTGGC: 11	CGTCTGGT: 1
GTCTGGCC: 9	GTCTGGCC: 9	GTCTGGTC: 2
TCTGGCCT: 8	TCTGGCCT: 8	TCTGGTCT: 1



Error correction

k-mer count profile indicates where errors are



Error correction

Simple algorithm: given a count threshold t:

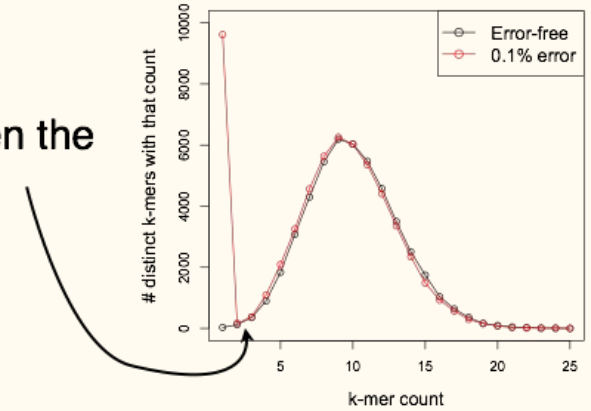
For each read:

For each k-mer:

If k-mer count $< t$:

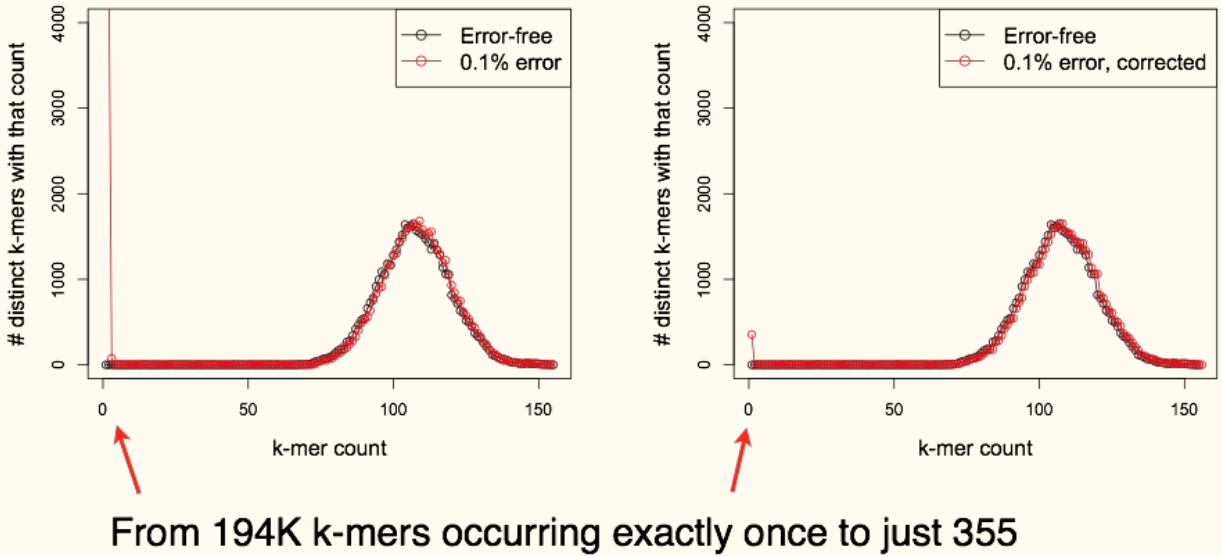
Examine k-mer's neighbors within certain Hamming/edit distance. If neighbor has count $\geq t$, replace old k-mer with neighbor.

Pick a *t* that lies in the trough (the dip) between the peaks



Error correction

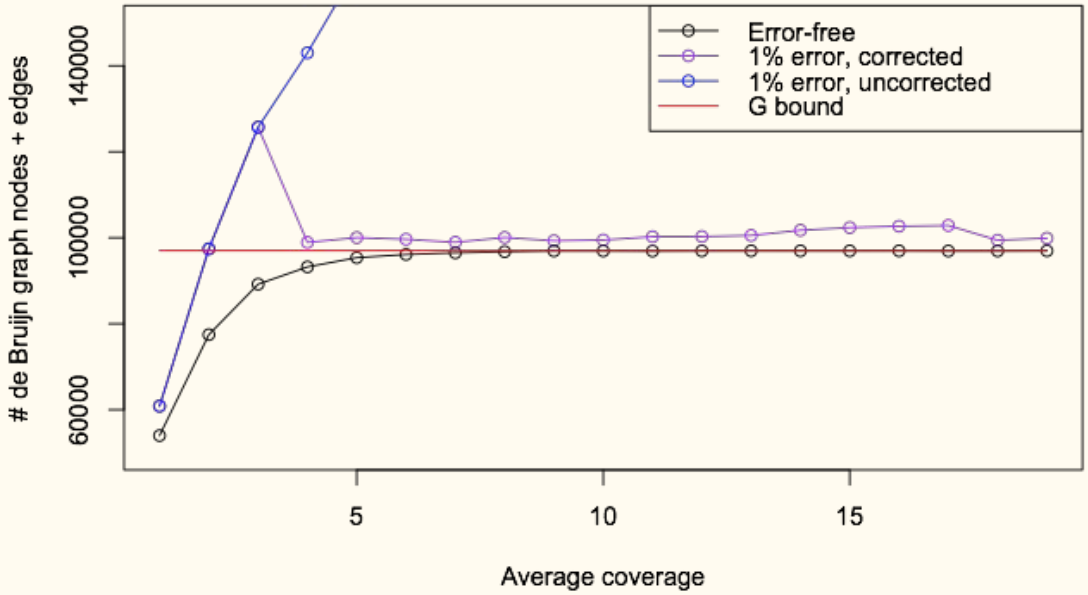
Corrects 99.2% of the errors in the example 0.1% error dataset



Error correction: results

For uncorrected reads, De Bruijn graph size is off the chart.

For corrected reads, De Bruijn graph size is near G bound.



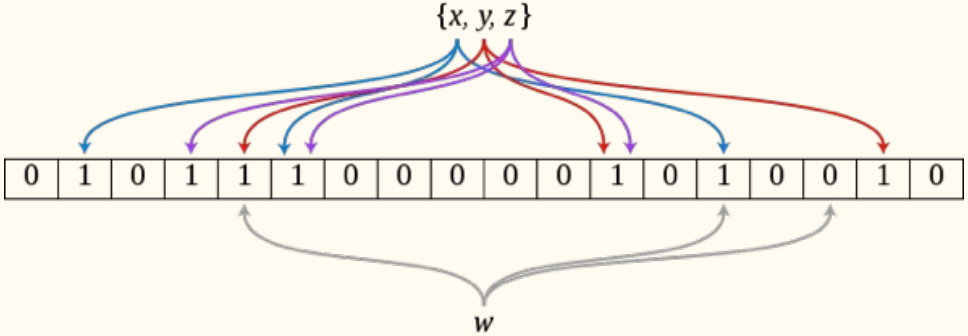
Error correction

For error correction to work well:

- Average coverage should be high enough and k should be set so we can distinguish infrequent from frequent k-mers
- k-mer neighborhood we explore must be broad enough to find frequent neighbors. Depends on error rate and k.
- Data structure for storing k-mer counts should be substantially smaller than the De Bruijn graph
    - Otherwise there's no point doing error correction separately
    - Counts don't have to be 100% accurate; just have to distinguish frequent and infrequent

Error correction: sketches

Sketch data structures are *extremely* compact, but *fail* sometimes
E.g. a Bloom Filter is like a hash set, but far smaller, and will sometimes say an object is in the set when it's not



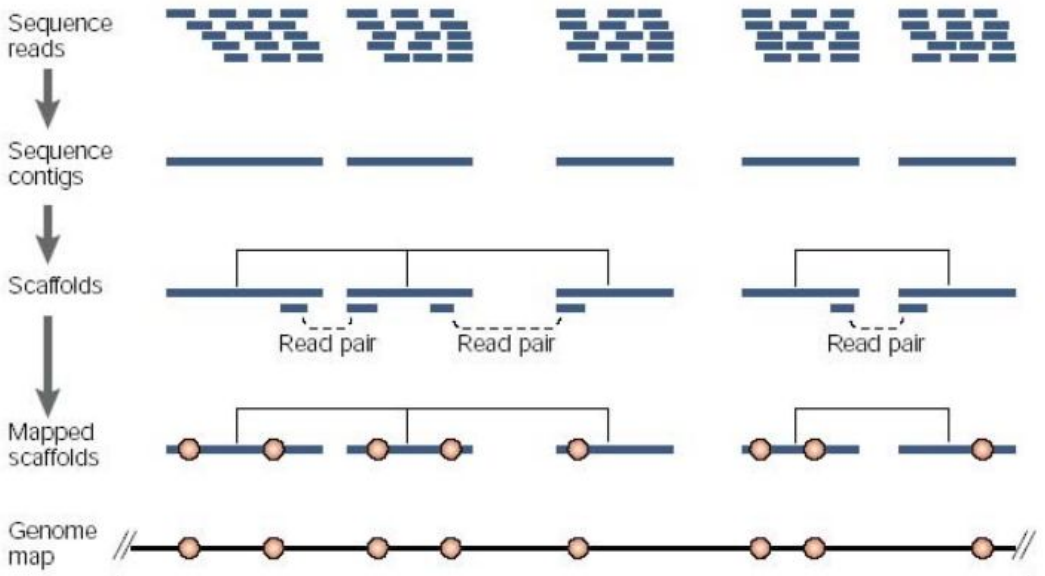
CountMin sketches generalize Bloom Filters for histograms (sets where elements have associated counts); reported counts might be too high

These are candidates for compactly storing k-mer counts:

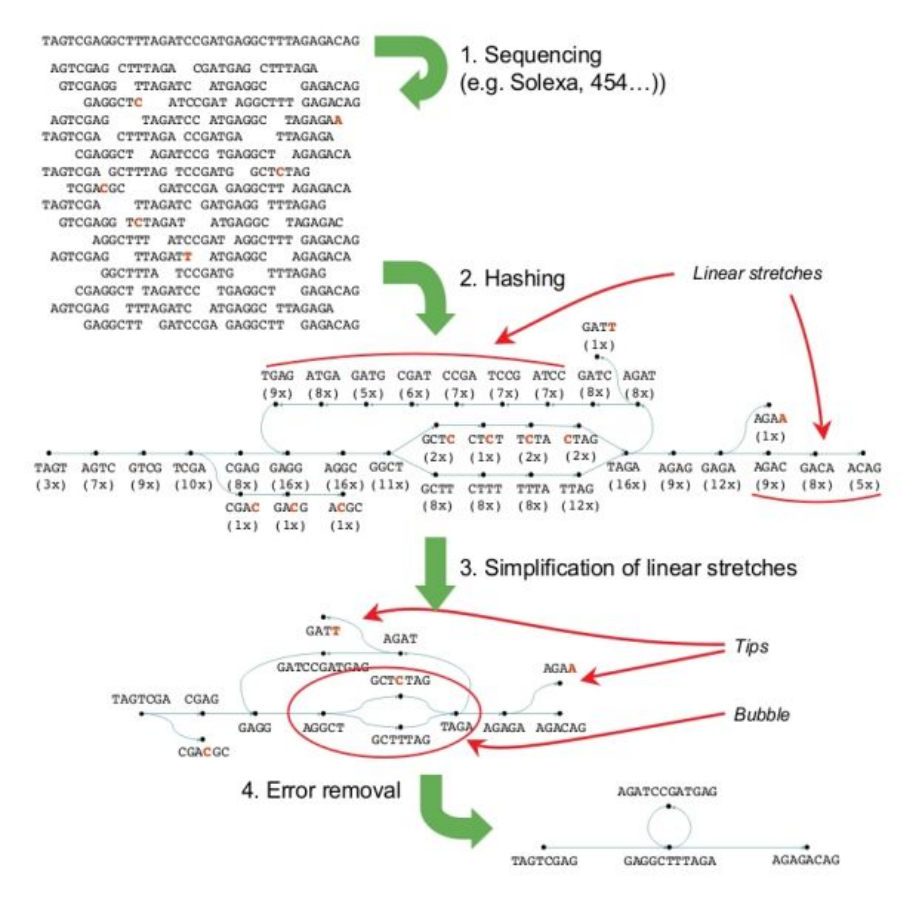
- http://en.wikipedia.org/wiki/Bloom_filter
- http://en.wikipedia.org/wiki/Count-Min_sketch

Scaffolding

de novo whole-genome shotgun assembly



Contig construction



Scaffolding

Both OLC and DBG are concerned with constructing the longest, most accurate *contigs* possible

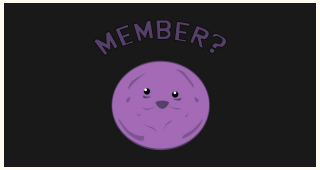
Scaffolding orders and orients *contigs* with respect to each other

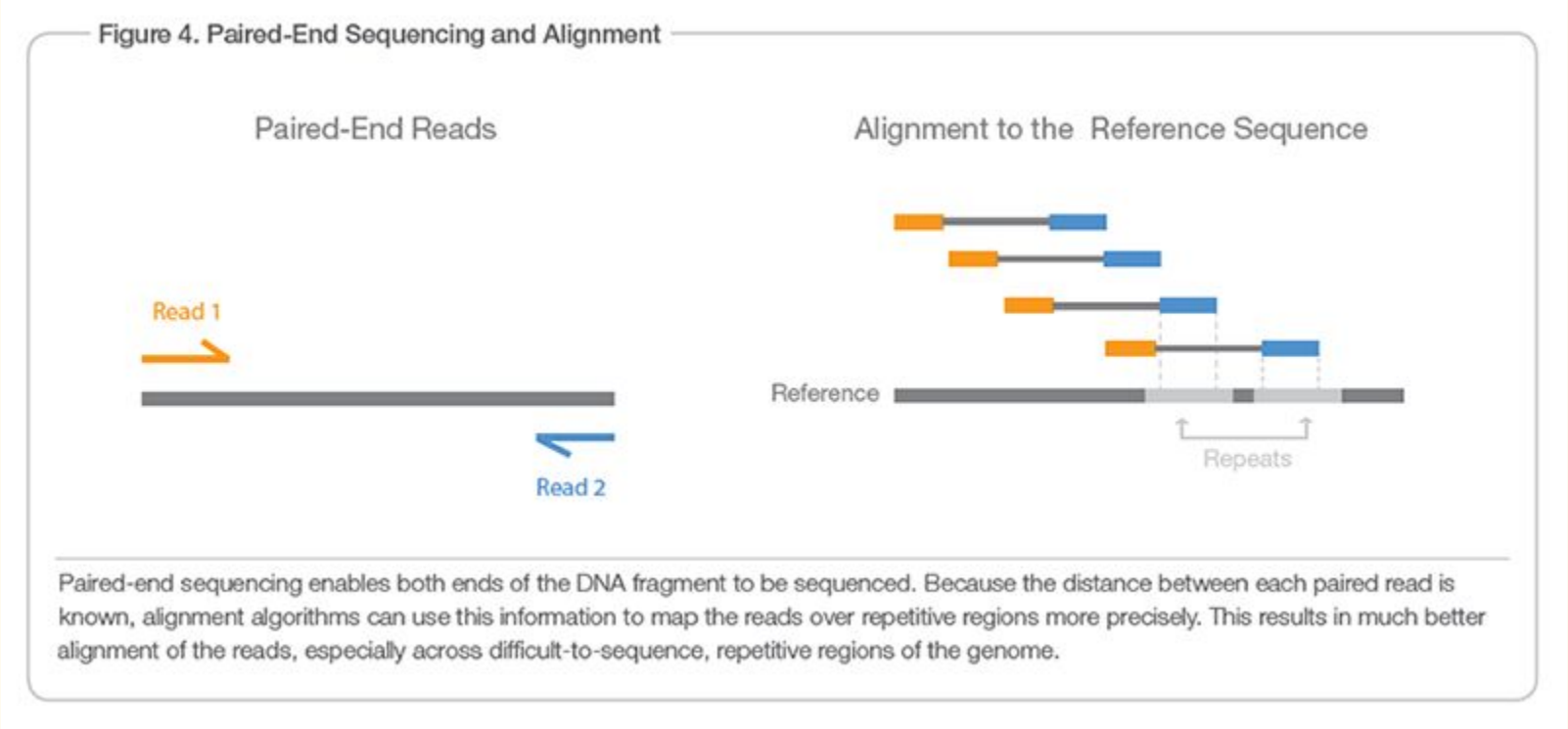
For this we can use data from various sources, especially paired ends.

Contig: is a stretch of unambiguously assembled sequence.

Scaffold: may contain gaps.

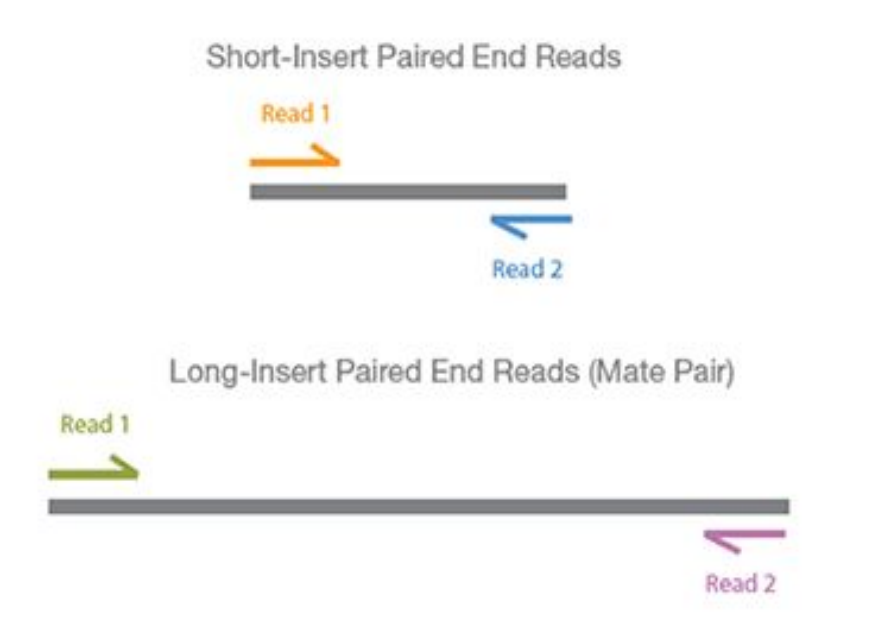
Paired-end sequencing





Vocabulary

- **Paired end reads:** read1, insert $<$ 500 bp, read2
- **Mate pair reads:** read1, insert $>$ 1 kbp, read2
- **k-mer:** any sequence of length k
- **Contig:** gap-less assembled sequence
- **Scaffold:** sequence which may contain gaps



Scaffolding: paired-end sequencing

Alternative protocol produces a *pair* of reads taken from either end of a longer fragment

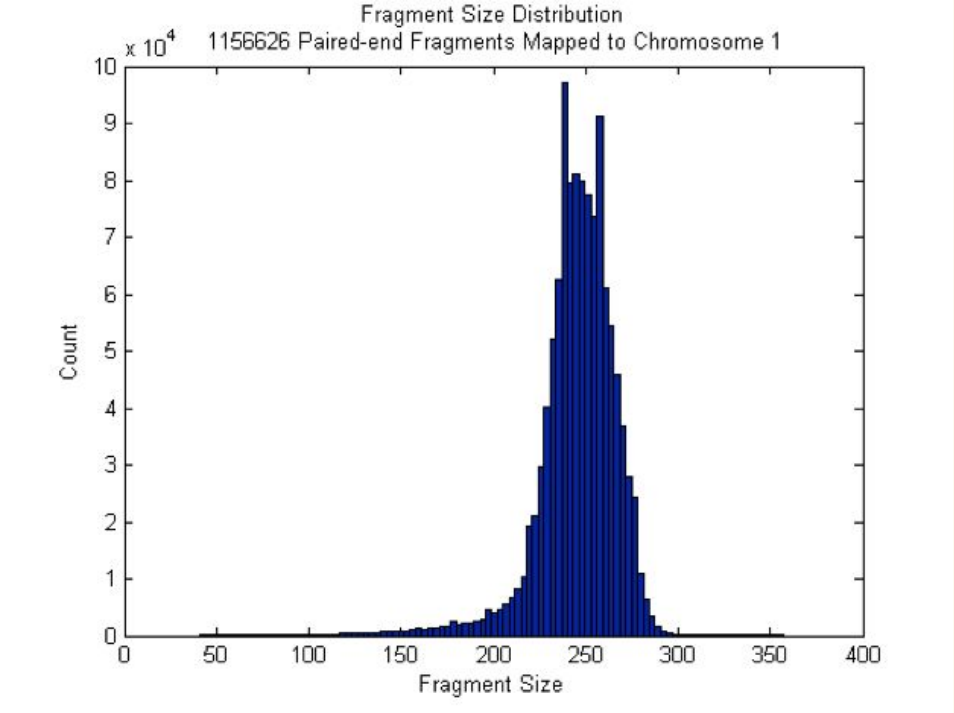
Paired reads are also called *mates* to distinguish them from the *unpaired* reads we've been discussing

Depending on lengths, mates might overlap in the middle of the fragment

Scaffolding: paired-end sequencing

Example fragment length distribution

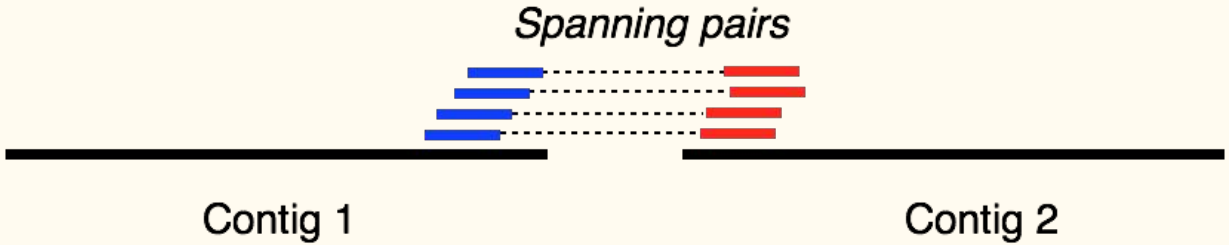
Fragments are not exactly the same length, but there's a clear peak around 250 nt, very few < 150 nt or > 300 nt



Scaffolding: paired-end sequencing

Say we have a collection of pairs and we assemble them as usual.

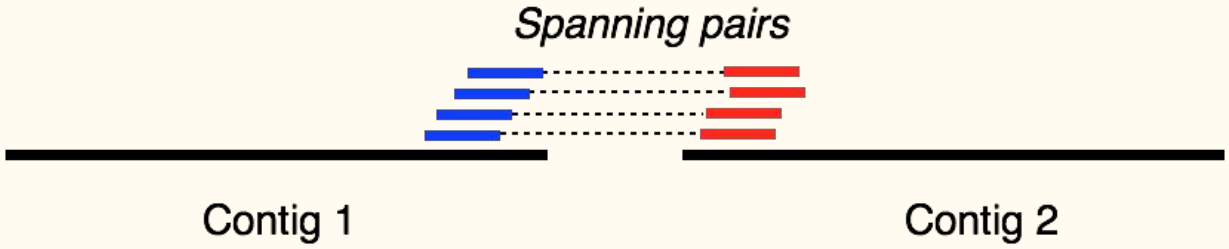
Assembly yields two contigs:



...and we discover that some of the mates at one edge of contig 1 are paired with mates in contig 2

Call these *spanning pairs*

Scaffolding: paired-end sequencing

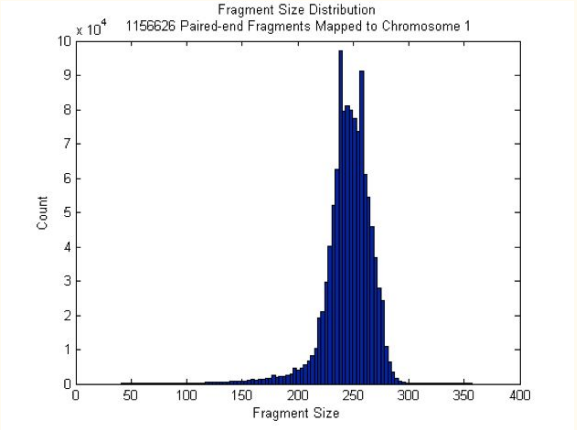


What does this tell us?

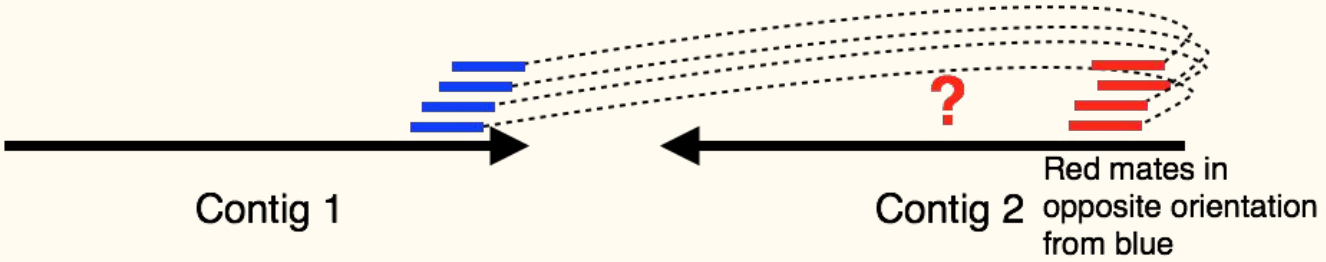
Contig 1 is close to contig 2 in the genome.

In fact, we can *estimate distance between contigs* using what we know about fragment length distribution.

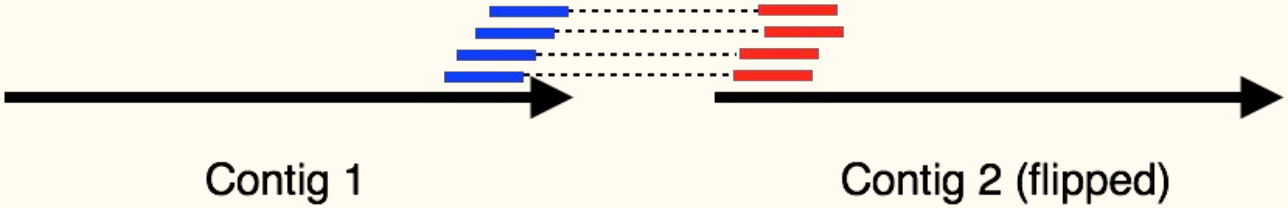
The more spanning pairs we have, the better our estimate.



Scaffolding: paired-end sequencing



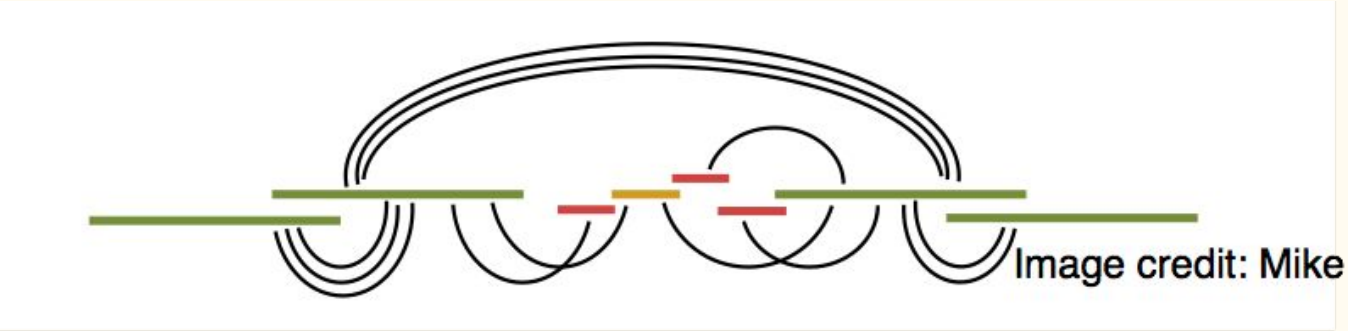
What does the picture look like if contigs 1 and 2 are close, but we assembled contig 2 “backwards” (i.e. reverse complemented)



Pairs also tell us about contigs’ relative *orientation*

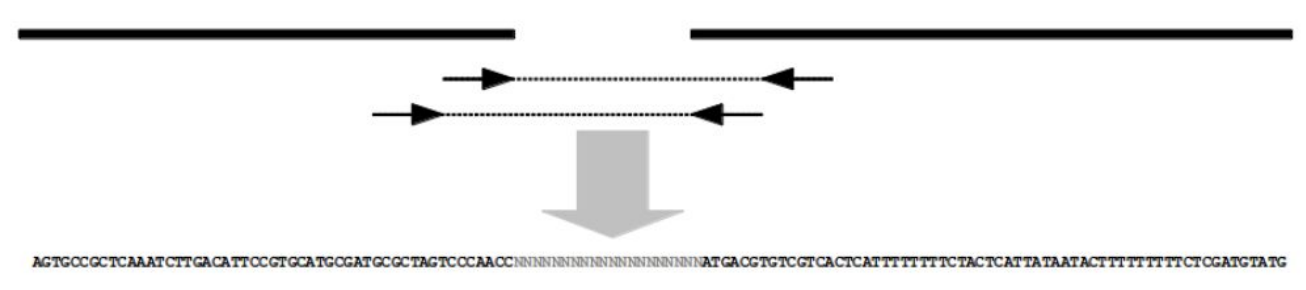
Scaffolding

Scaffolding output: collection of *scaffolds*, where a scaffold is a collection of contigs related to each other with high confidence using pairs.



Scaffolds construction in practice (lab)

- Scaffolding using pairing information
- This is why libraries containing **multiple insert sizes** are used:
  - Gnerre 2011:
    - 45x overlapping PE reads (insert size: 180 bp, reads: >100 bp)
    - 45x short jump MP reads (insert size: 3 kb)
    - 5x (optional) long jump MP reads (insert size: 6 kb)
    - 1x (optional) fosmid jump MP reads (insert size: 40 kb)
  - Ribeiro 2012:
    - 50x overlapping PE reads (insert size: 180 bp, reads: >100 bp)
    - 50x PacBio reads (reads: 1-3 kb)



Scaffolds construction

- Scaffolding using pairing information
- Techniques:
  - Jumping libraries
  - Linked reads (10x Genomics, Dovetail Genomics - Chicago libraries)
  - Long reads (PacBio, Oxford nanopore)
  - Structural maps (BioNano)

Gap closing / contig extension