Genome Assembly Algorithms: de Bruijn graphs

Dr. Jared Simpson
Ontario Institute for Cancer Research
&
Department of Computer Science
University of Toronto

Real-world assembly methods

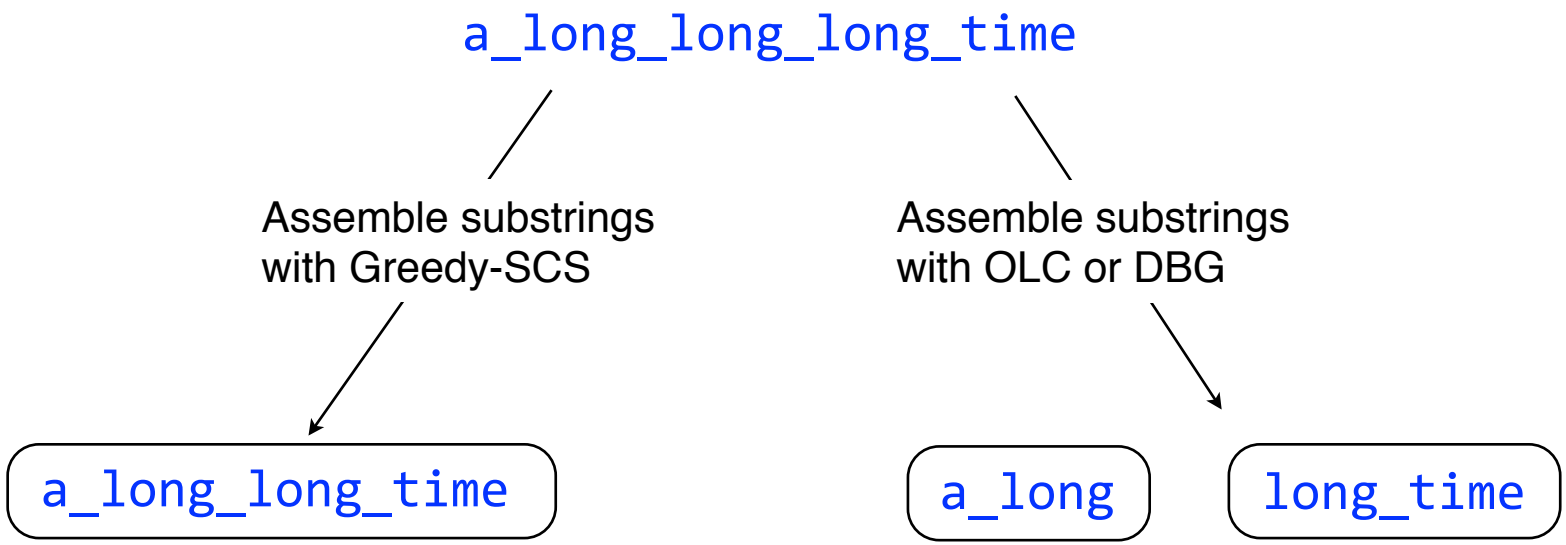
OLC: Overlap-Layout-Consensus assembly

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 Bruijn graph assembly

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

k-mer

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

"*k*-mer" is a substring of length *k*

S: GGCGATTCATCG

A 4-mer of *S*: ATTC

All 3-mers of *S*:

```
GGC
 GCG
  CGA
   GAT
    ATT
     TTC
      TCA
       CAT
        ATC
         TCG
```

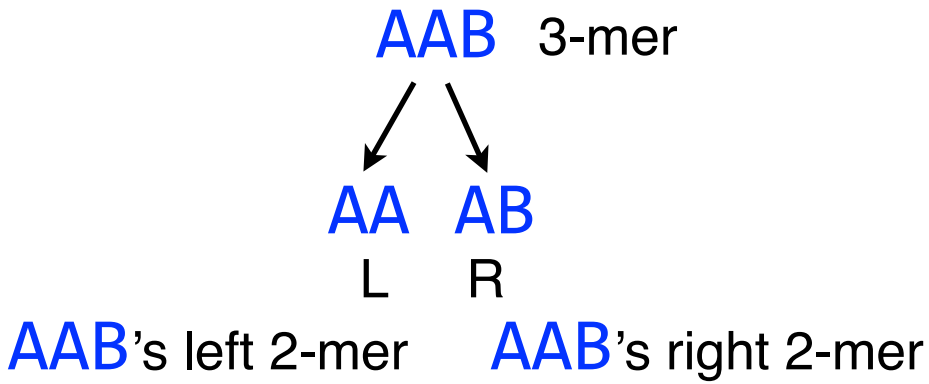
I'll use "*k*-1-mer" to refer to a substring of length *k* - 1

De Bruijn graph

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

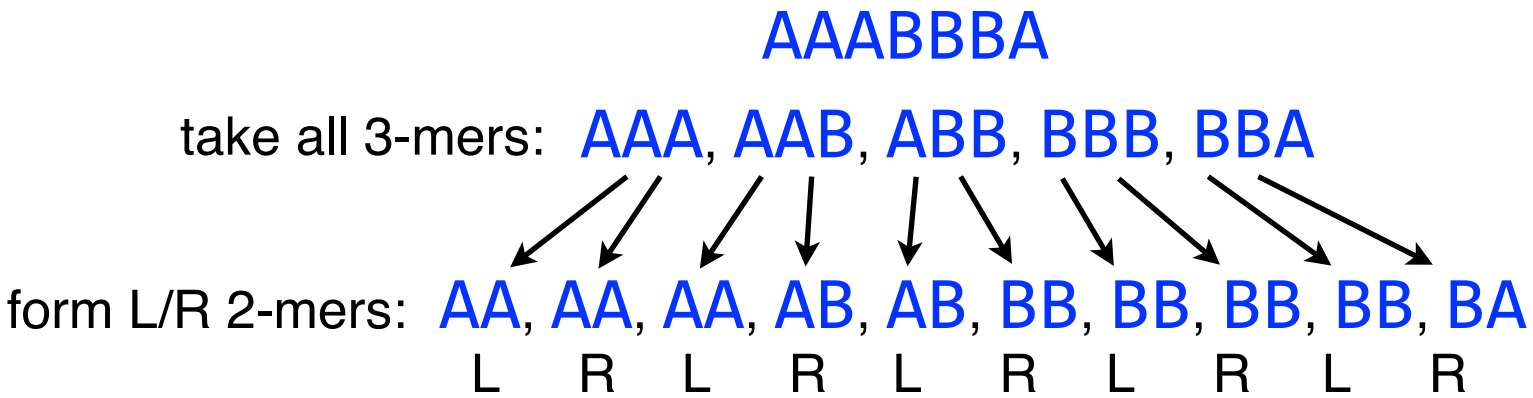
AAA, AAB, ABB, BBB, BBA

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

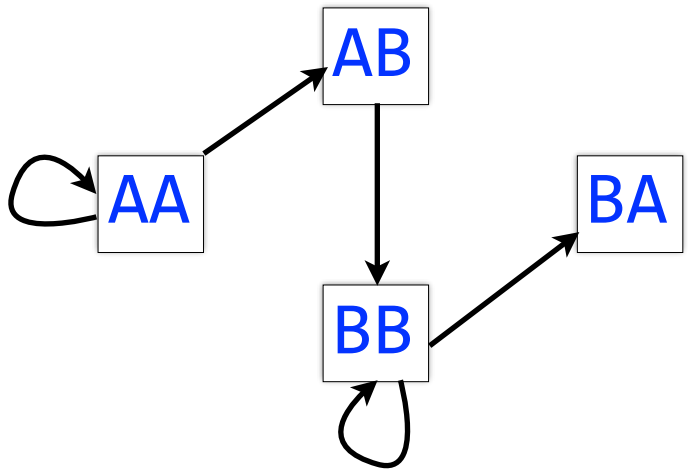


De Bruijn graph

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

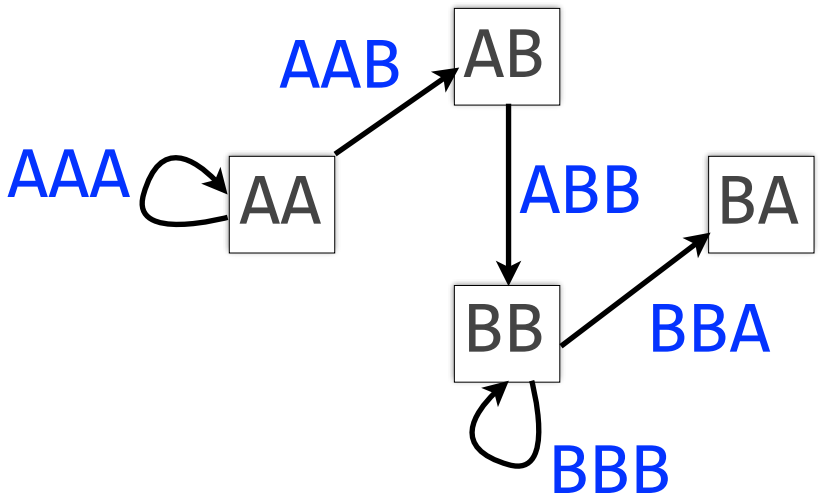


Let 2-mers be nodes in a new graph. Draw a directed edge from each left 2-mer to corresponding right 2-mer:



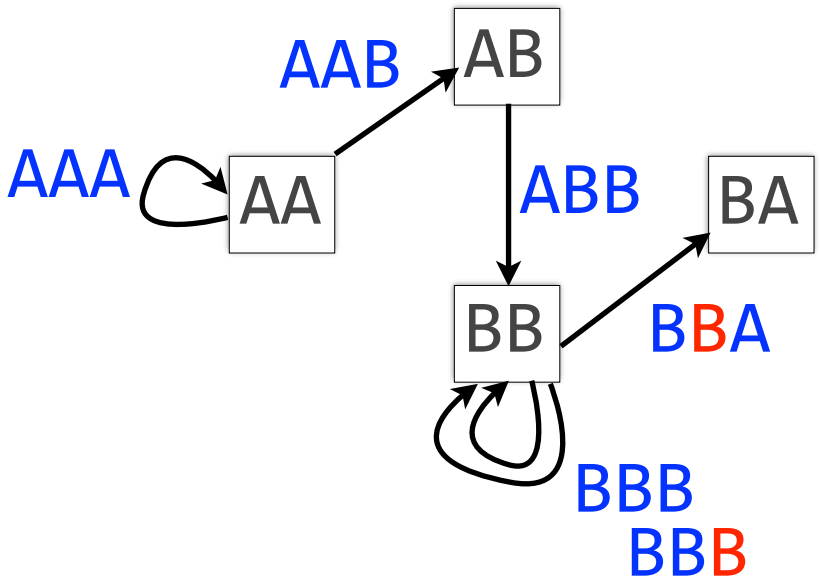
Each *edge* in this graph corresponds to a length-3 input string

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*.

Directed multigraph

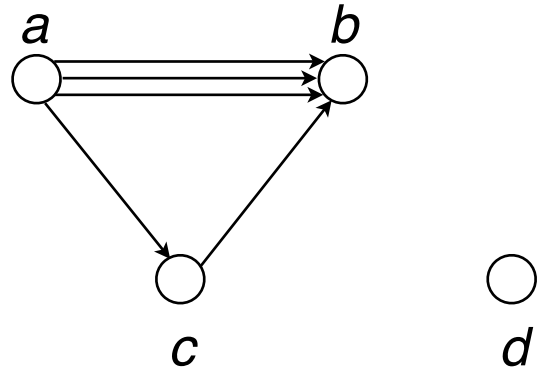
Directed **multigraph** $G(V, E)$ consists of set of *vertices, V* and **multiset** of *directed edges, E*

Otherwise, like a directed graph

Node's *indegree* = # incoming edges

Node's *outdegree* = # outgoing edges

De Bruijn graph is a directed multigraph



$V = \{ a, b, c, d \}$

$E = \{ (a, b), (a, b), (a, b), (a, c), (c, b) \}$

Repeated

Eulerian walk definitions and statements

Node is *balanced* if indegree equals outdegree

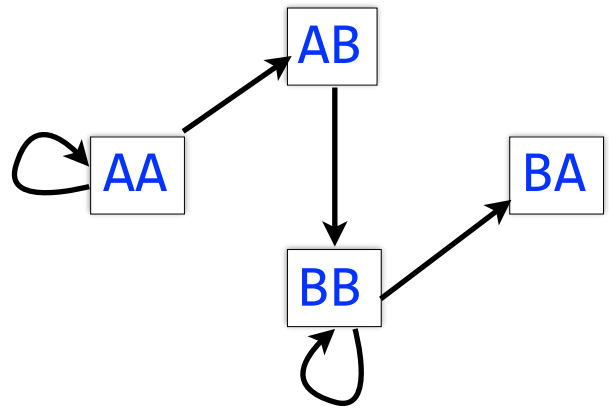
Node is *semi-balanced* if indegree differs from outdegree by 1

Graph is *connected* if each node can be reached by some other node

Eulerian walk visits each edge exactly once

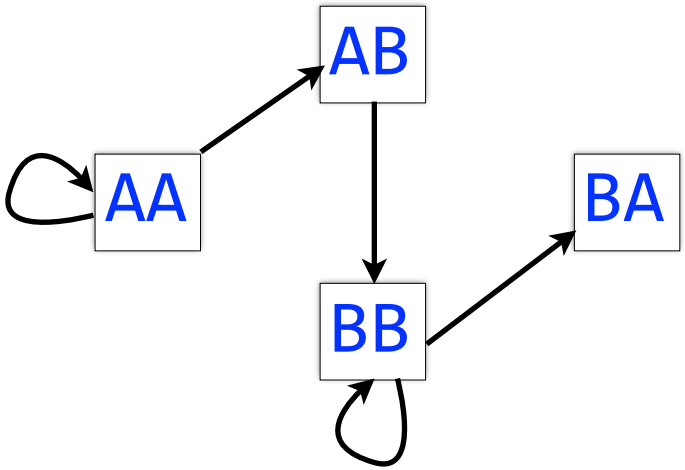
Not all graphs have Eulerian walks. Graphs that do are *Eulerian.*

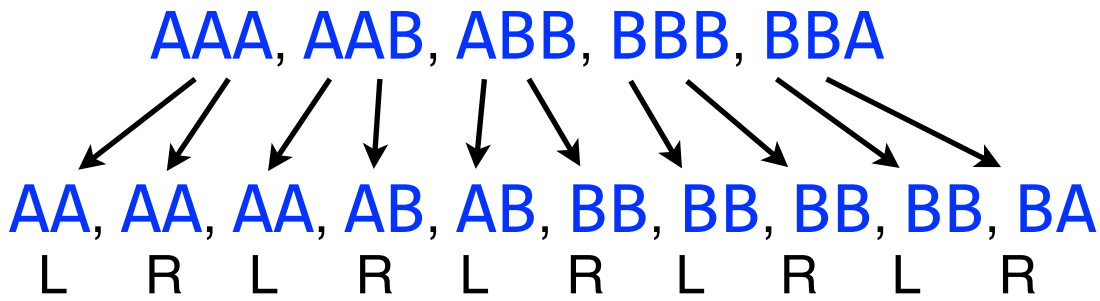
A directed, connected graph is Eulerian if and only if it has at most 2 semi-balanced nodes and all other nodes are balanced



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

A procedure for making a De Bruijn graph for a genome

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

Pick a substring length *k*: 5

Start with an input string:

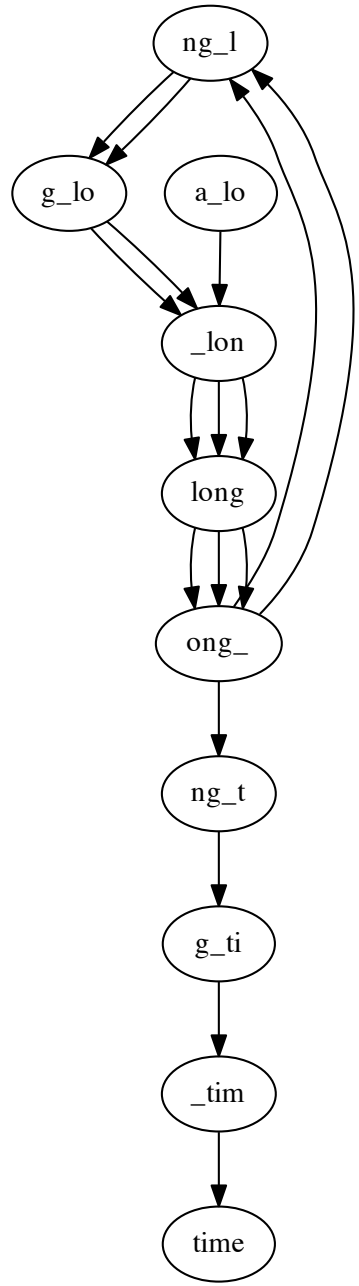
```
a_long_long_long_time
```

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

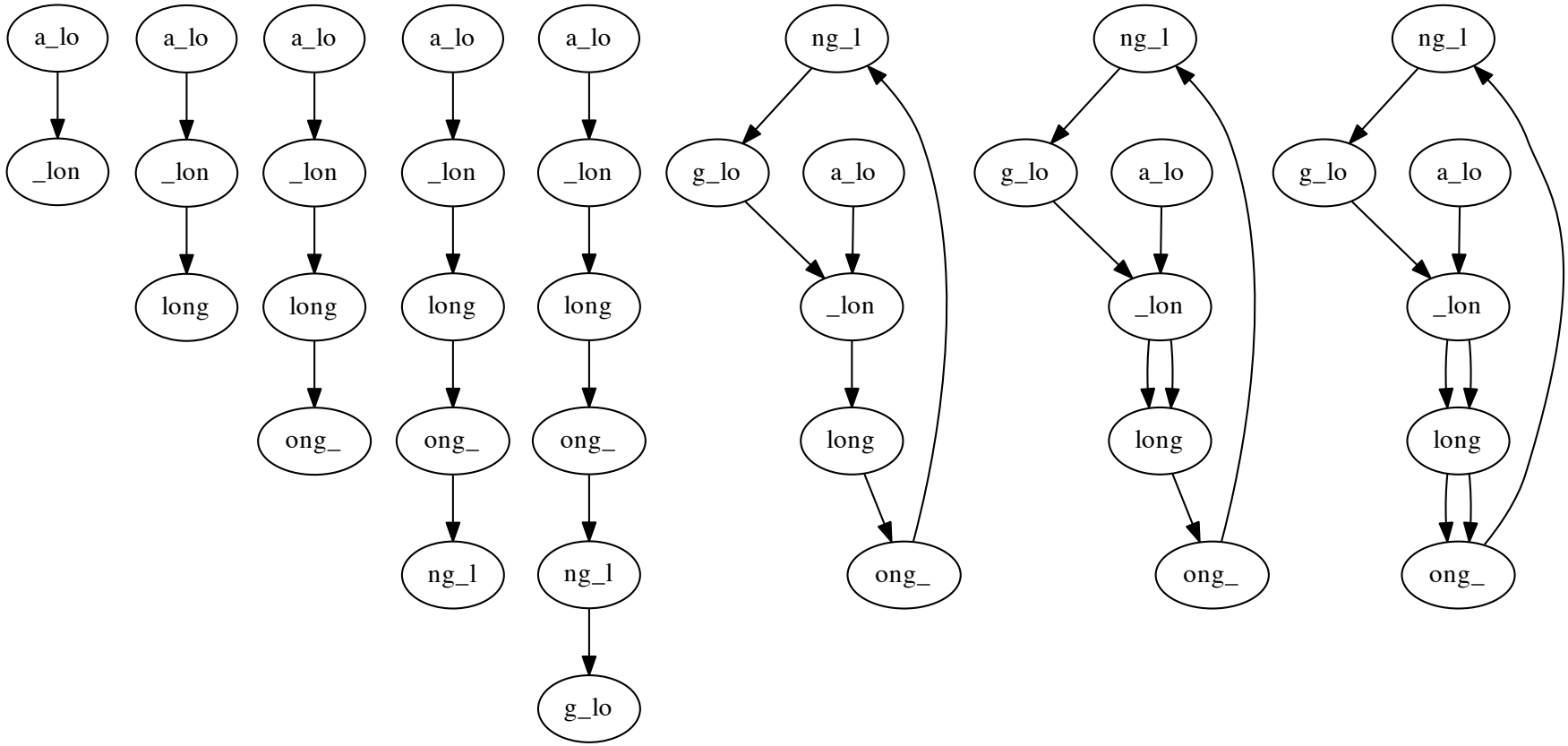
```
long_
```

```
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



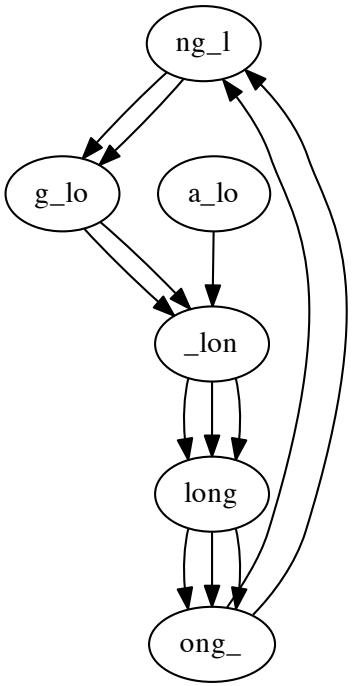
De Bruijn

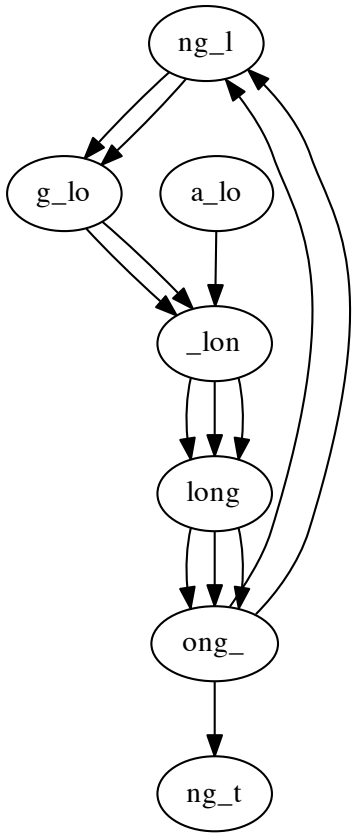


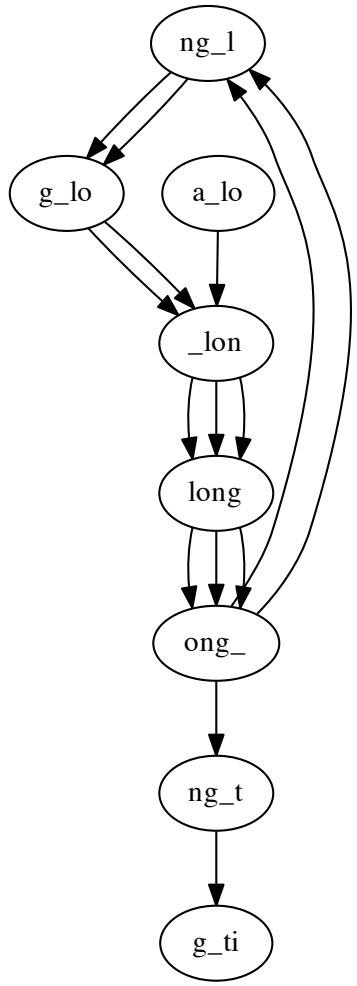
First 8 *k*-mer additions, *k* = 5

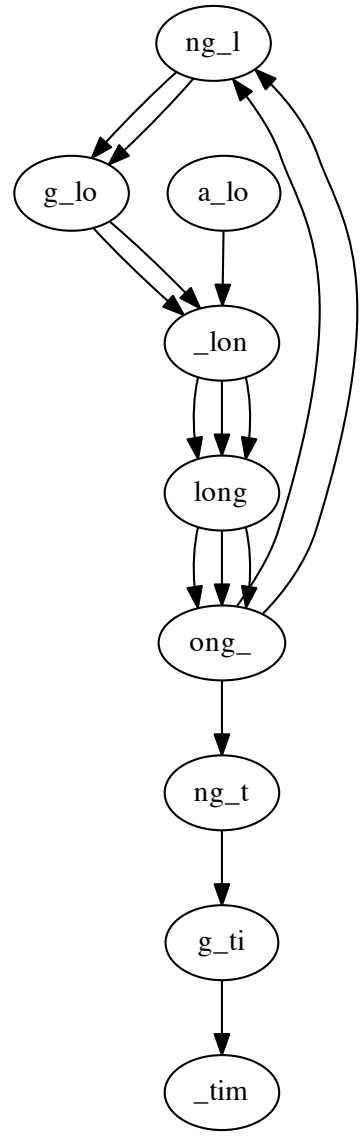
`a_long_long_long_time`

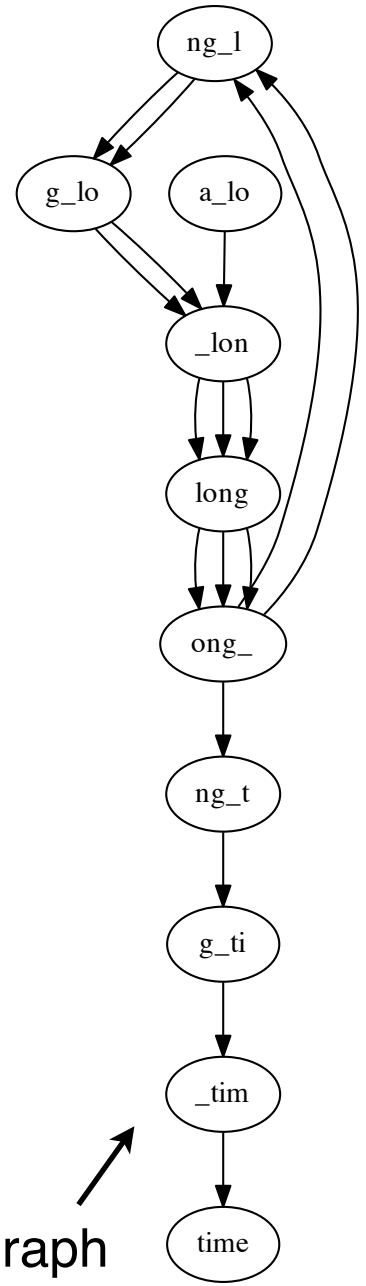
De Bruijn











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

`a_long_long_long_time`

Finished graph

De Bruijn graph

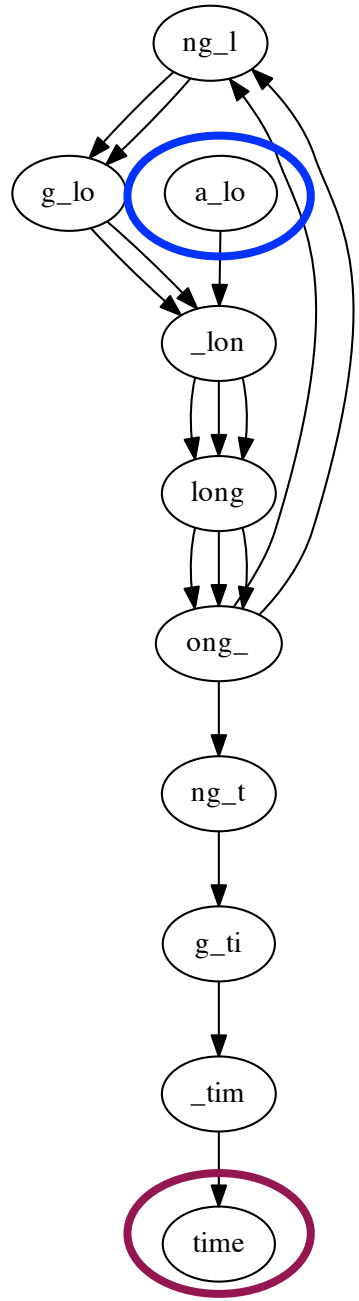
With perfect sequencing, this procedure always yields an Eulerian graph. Why?

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 graph implementation

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

```
class DeBruijnGraph:
    """ A De Bruijn multigraph built from a collection of strings.
        User supplies strings and k-mer length k.  Nodes of the De
        Bruijn graph are k-1-mers and edges join a left k-1-mer to a
        right k-1-mer. """

    @staticmethod
    def chop(st, k):
        """ Chop a string up into k mers of given length """
        for i in xrange(0, len(st)-(k-1)): yield st[i:i+k]

    class Node:
        """ Node in a De Bruijn graph, representing a k-1 mer """
        def __init__(self, km1mer):
            self.km1mer = km1mer

        def __hash__(self):
            return hash(self.km1mer)

    def __init__(self, strIter, k):
        """ Build De Bruijn multigraph given strings and k-mer length k """
        self.G = {}     # multimap from nodes to neighbors
        self.nodes = {} # maps k-1-mers to Node objects
        self.k = k
        for st in strIter:
            for kmer in self.chop(st, k):
                km1L, km1R = kmer[:-1], kmer[1:]
                nodeL, nodeR = None, None
                if km1L in self.nodes:
                    nodeL = self.nodes[km1L]
                else:
                    nodeL = self.nodes[km1L] = self.Node(km1L)
                if km1R in self.nodes:
                    nodeR = self.nodes[km1R]
                else:
                    nodeR = self.nodes[km1R] = self.Node(km1R)
                self.G.setdefault(nodeL, []).append(nodeR)
```

Chop string into *k*-mers

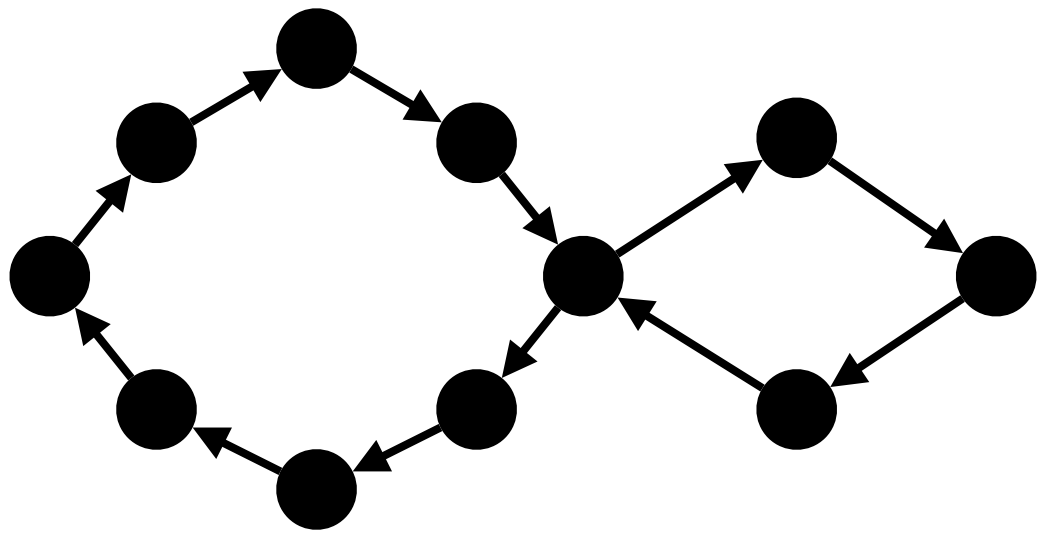
For each *k*-mer, find left and right *k*-1-mers

Create corresponding nodes (if necessary) and add edge

Finding Eulerian cycles

Hierholzer's O(|E|) algorithm:

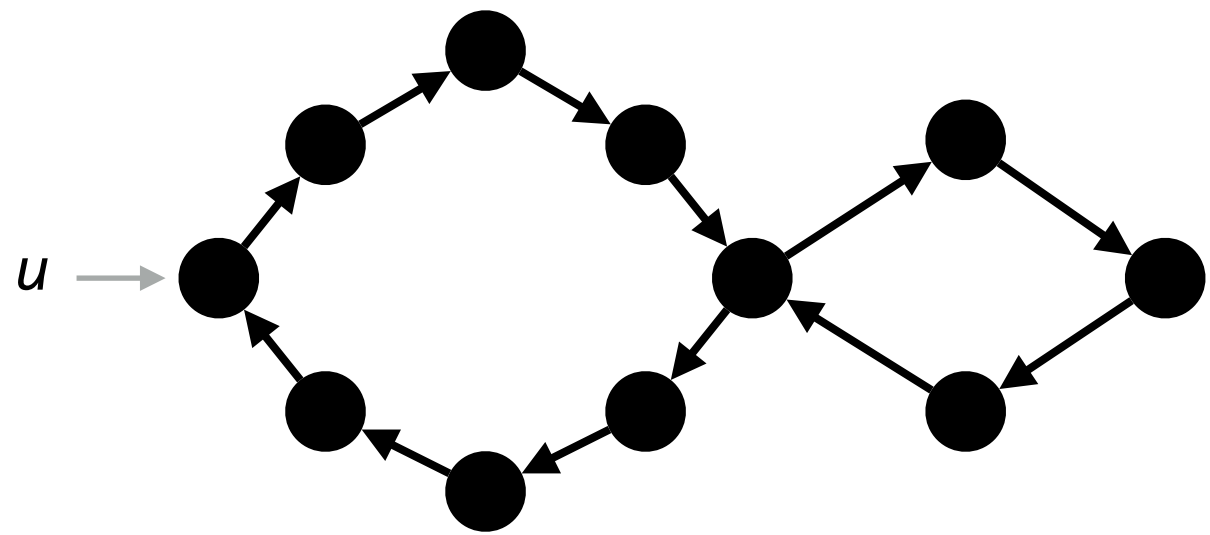
1. *If the graph has 2 semi-balanced vertices, connect them*
2. *Select an arbitrary start point u*
3. *Follow edges until we reach u again*
4. *If there is some vertex v on the path that has an edge not on the path, repeat starting from v with unused edges*
5. *Repeat until all edges are used*
6. *Splice walks from v into the walk from u*



Finding Eulerian cycles

Hierholzer's O(|E|) algorithm:

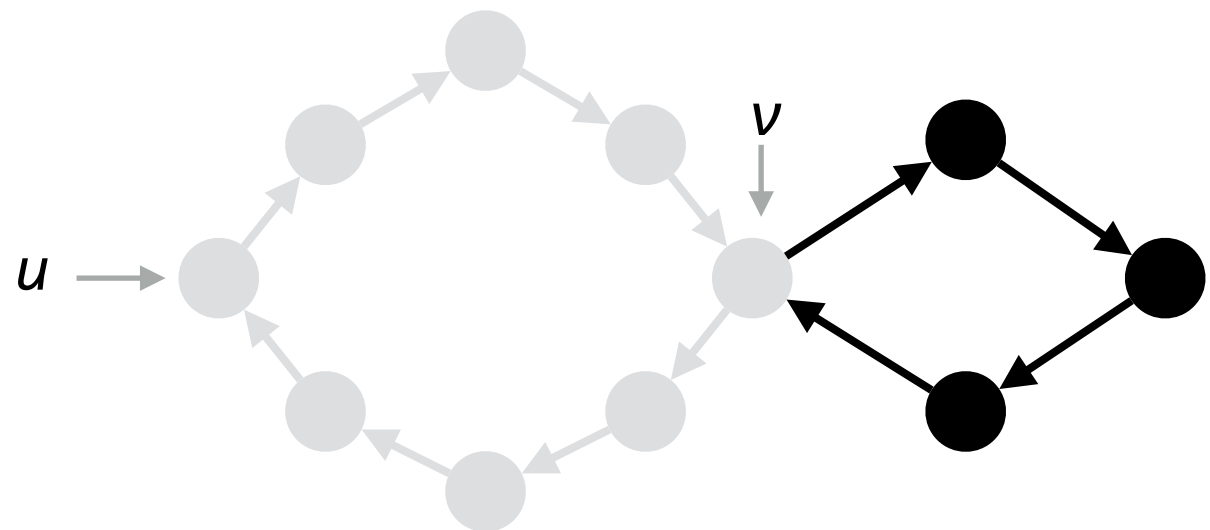
1. *If the graph has 2 semi-balanced vertices, connect them*
2. *Select an arbitrary start point u*
3. *Follow edges until we reach u again*
4. *If there is some vertex v on the path that has an edge not on the path, repeat starting from v with unused edges*
5. *Repeat until all edges are used*
6. *Splice walks from v into the walk from u*



Finding Eulerian cycles

Hierholzer's O(|E|) algorithm:

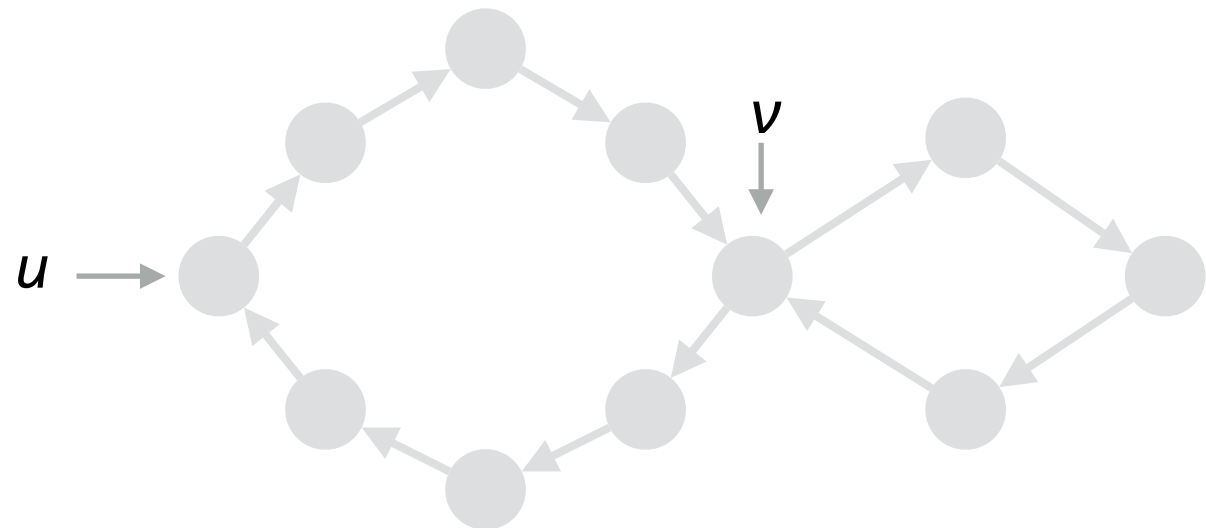
1. *If the graph has 2 semi-balanced vertices, connect them*
2. *Select an arbitrary start point u*
3. *Follow edges until we reach u again*
4. *If there is some vertex v on the path that has an edge not on the path, repeat starting from v with unused edges*
5. *Repeat until all edges are used*
6. *Splice walks from v into the walk from u*



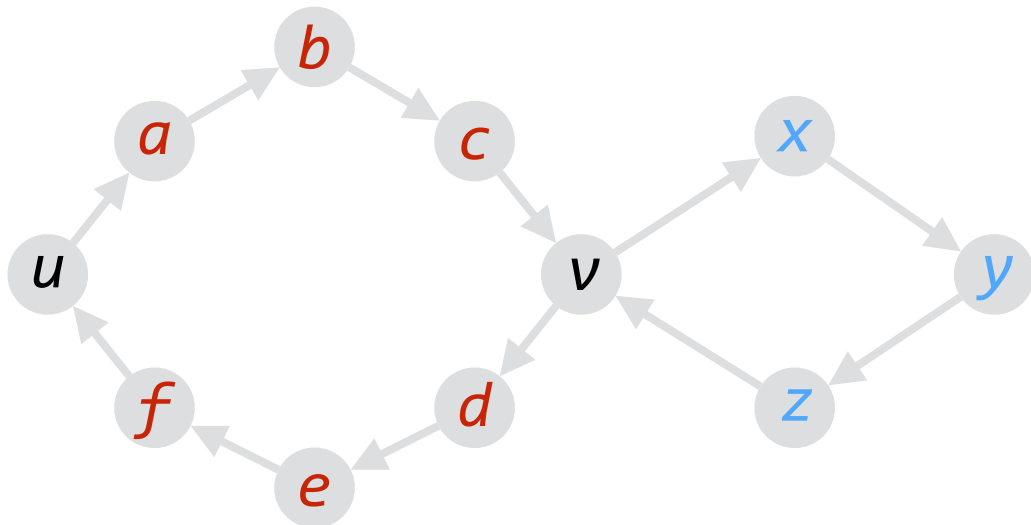
Finding Eulerian cycles

Hierholzer's O(|E|) algorithm:

1. *If the graph has 2 semi-balanced vertices, connect them*
2. *Select an arbitrary start point u*
3. *Follow edges until we reach u again*
4. *If there is some vertex v on the path that has an edge not on the path, repeat starting from v with unused edges*
5. *Repeat until all edges are used*
6. *Splice walks from v into the walk from u*



Finding Eulerian cycles



first cycle: *u a b c v d e f u*

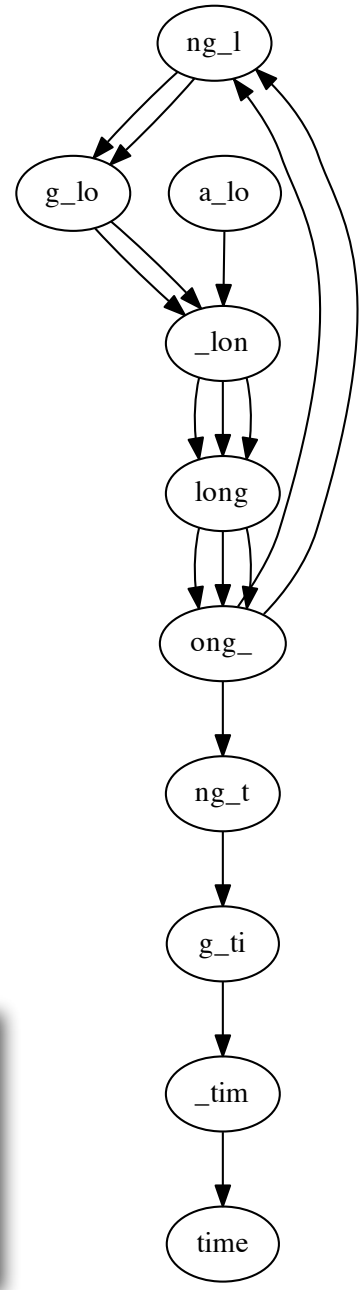
second cycle: *v x y z v*

Eulerian cycle: *u a b c v x y z v d e f u*

De Bruijn

Full illustrative De Bruijn graph and Eulerian walk:

http://nbviewer.ipython.org/7237207



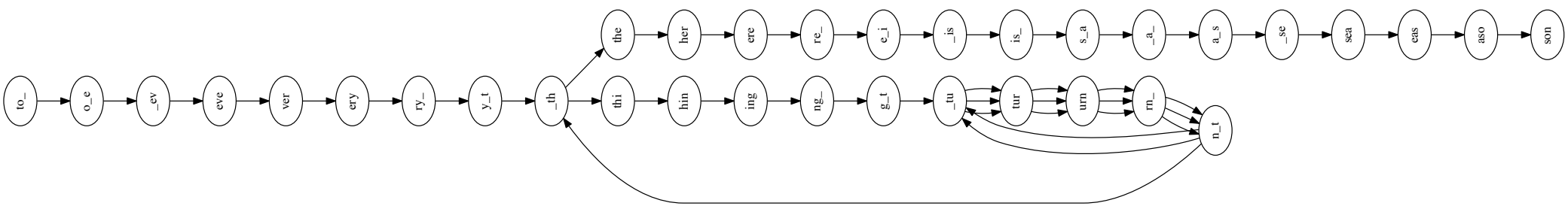
Example where Eulerian walk gives correct answer for small *k* whereas Greedy-SCS could spuriously collapse repeat:

```
>>> 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']
```

De Bruijn

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
```

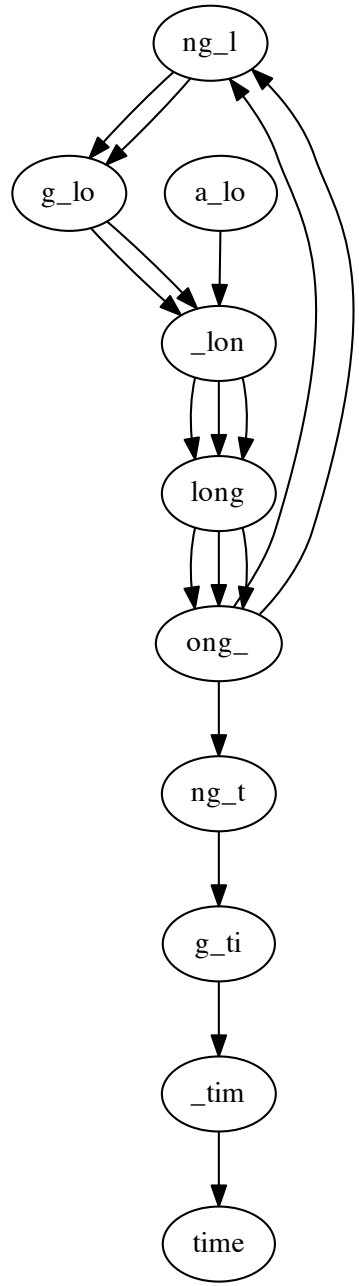


Recall: This is not generally possible or tractable in the overlap/SCS formulation

De Bruijn graph

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

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

Right: graph for ZABCDABEFABY, $k = 3$

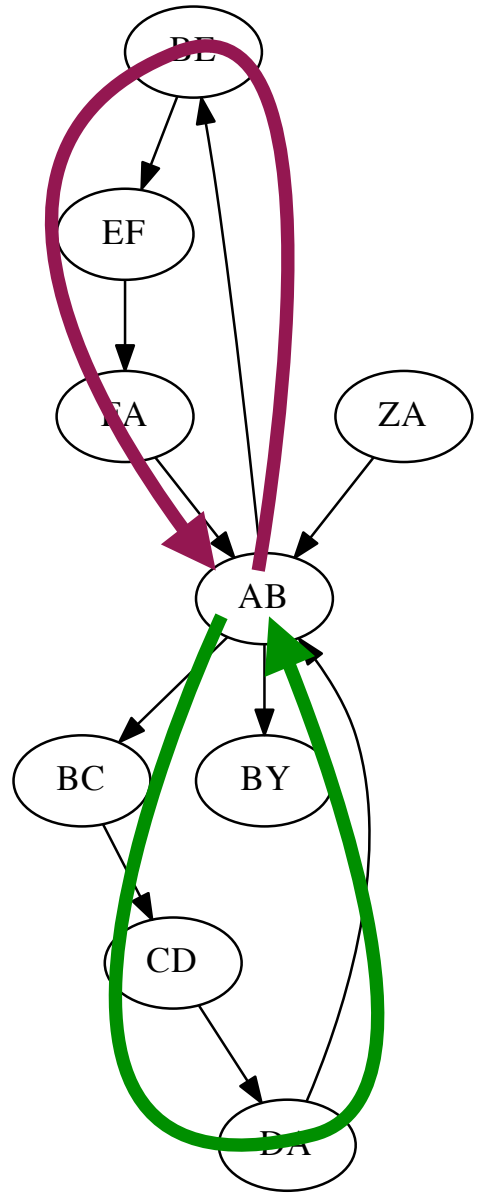
Alternative Eulerian walks:

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

Case where $k = 4$ works:

```
>>> 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
```

But $k = 3$ does not:

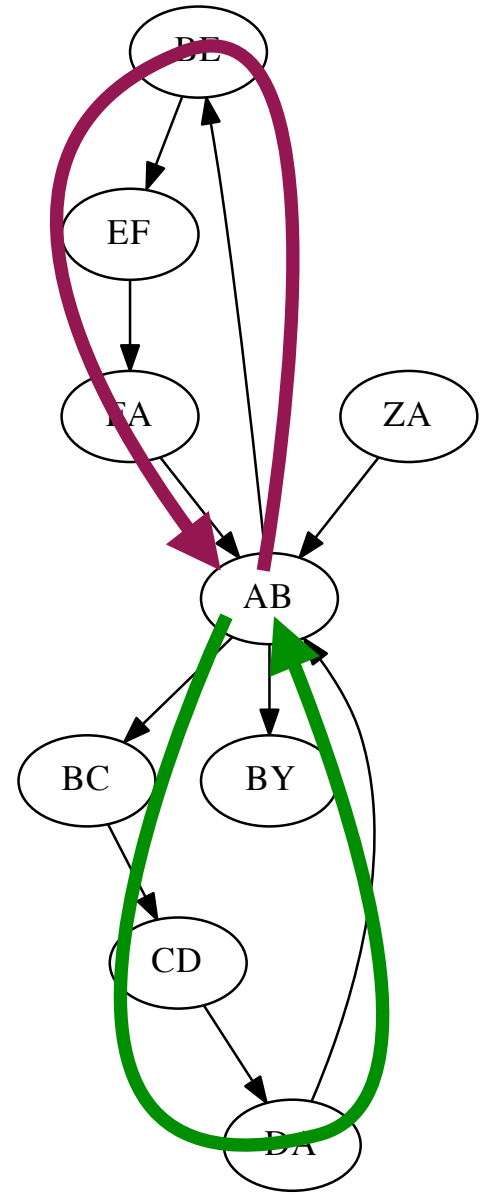
```
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 3)
>>> path = G.eulerianWalkOrCycle()
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
to_every_turn_turn_thing_turn_there_is_a_season
          |-------------------------|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

Due to repeats that are unresolvable at $k = 3$

De Bruijn

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

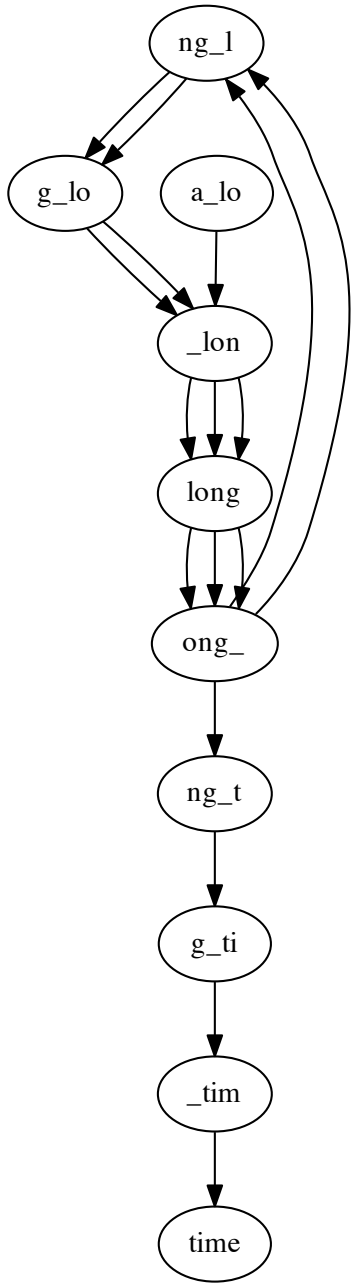
Other signs emerge when we think about how actual sequencing differs from our idealized construction



De Bruijn

Gaps in coverage can lead to *disconnected* graph

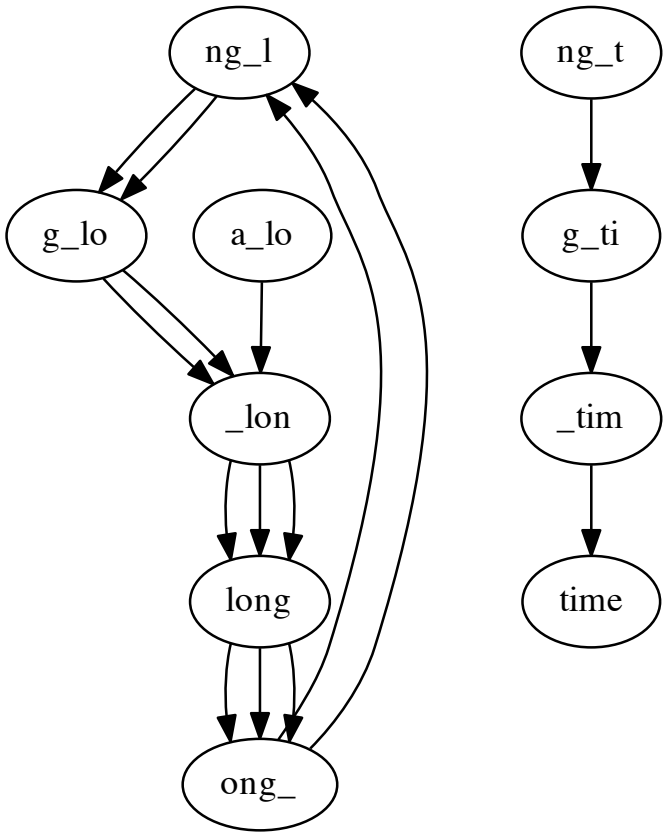
Graph for `a_long_long_long_time`, $k = 5$:



De Bruijn

Gaps in coverage can lead to *disconnected* graph

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



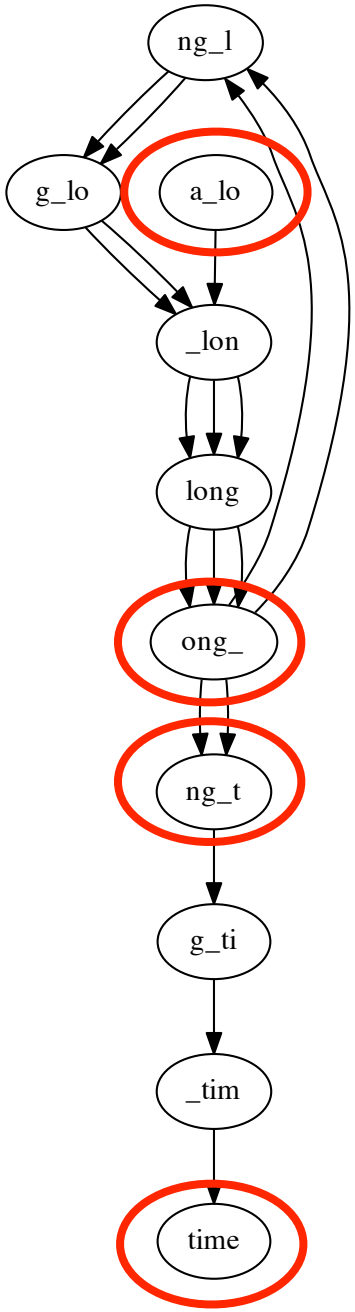
Connected components are individually Eulerian, overall graph is not

De Bruijn

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

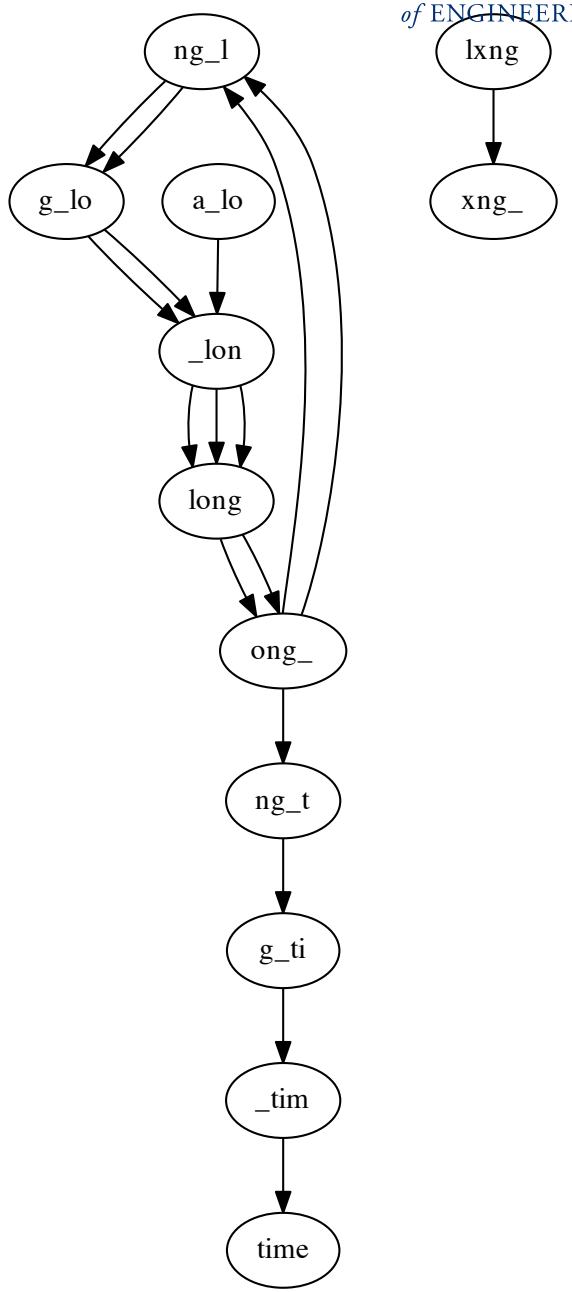


De Bruijn graph

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

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

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

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

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 = 1 x 10^9 reads
n = 100 nt
} ≈ 1 sequencing run
m = 3 x 10^9 nt ≈ human

To build a De Bruijn graph in practice:

Pick *k*. Assume *k* ≤ shortest read length (*k* = 30 to 60 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

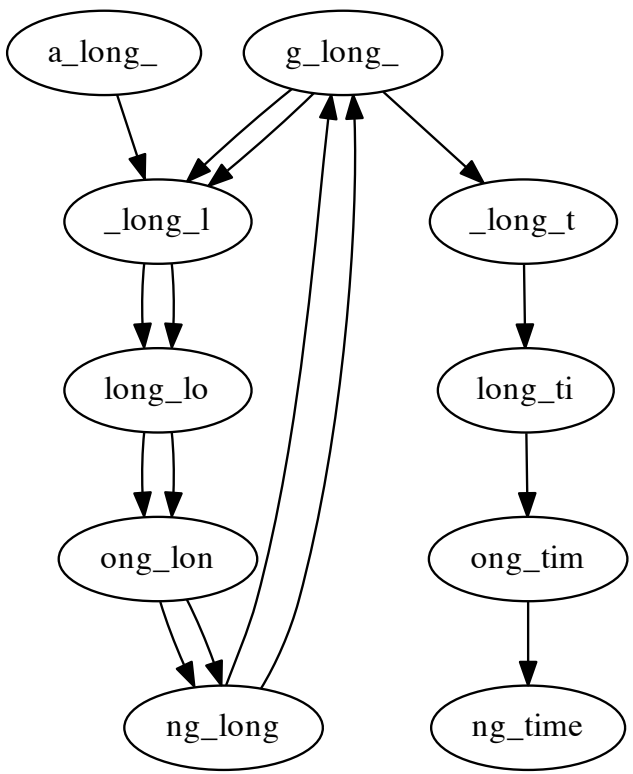
Reads: a_long_long_long, ng_long_l, g_long_time

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

g_long_t
 _long_ti
  long_tim
   ong_time
```



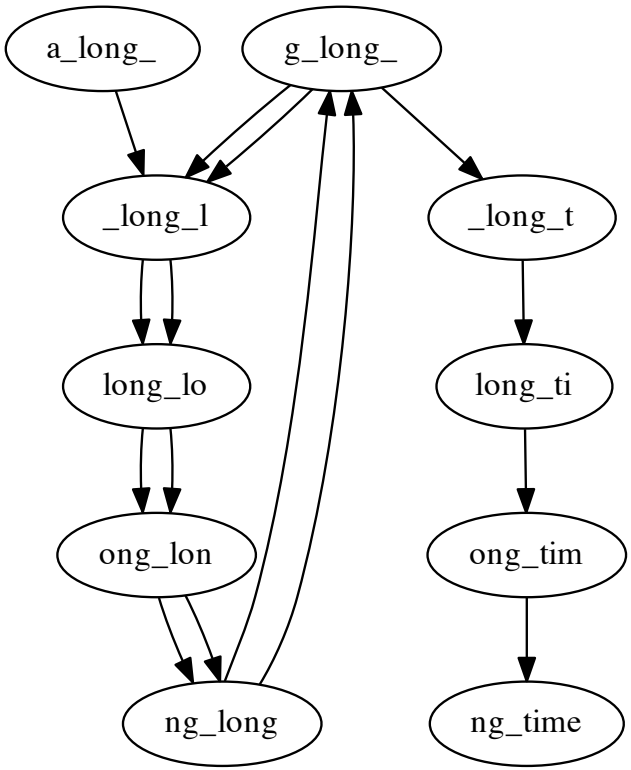
Given n (# reads), N (total length of all reads) and k, and assuming $k <$ length of shortest read:

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 $2 \cdot |E|$, 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

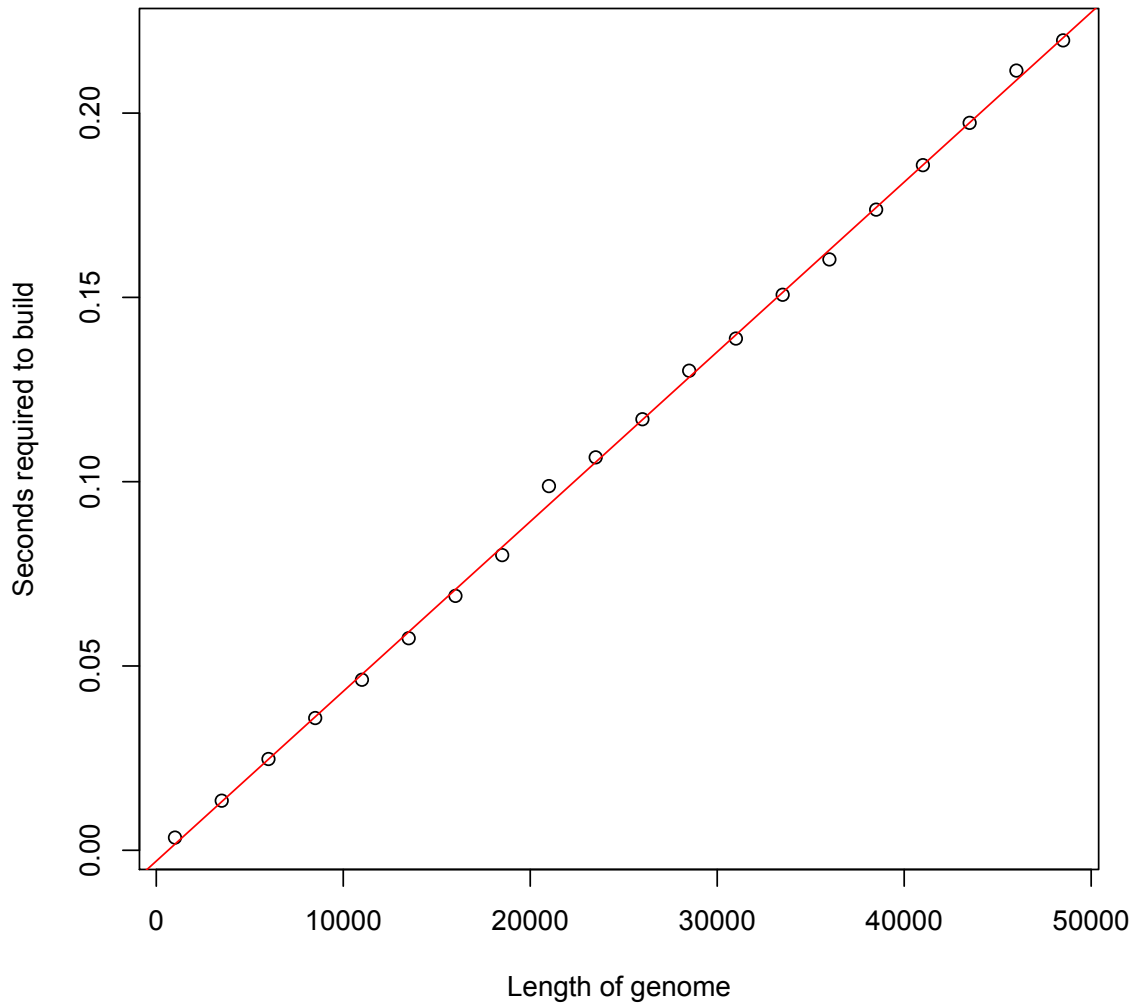
Querying / adding a key is O(1) expected work

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

De Bruijn graph

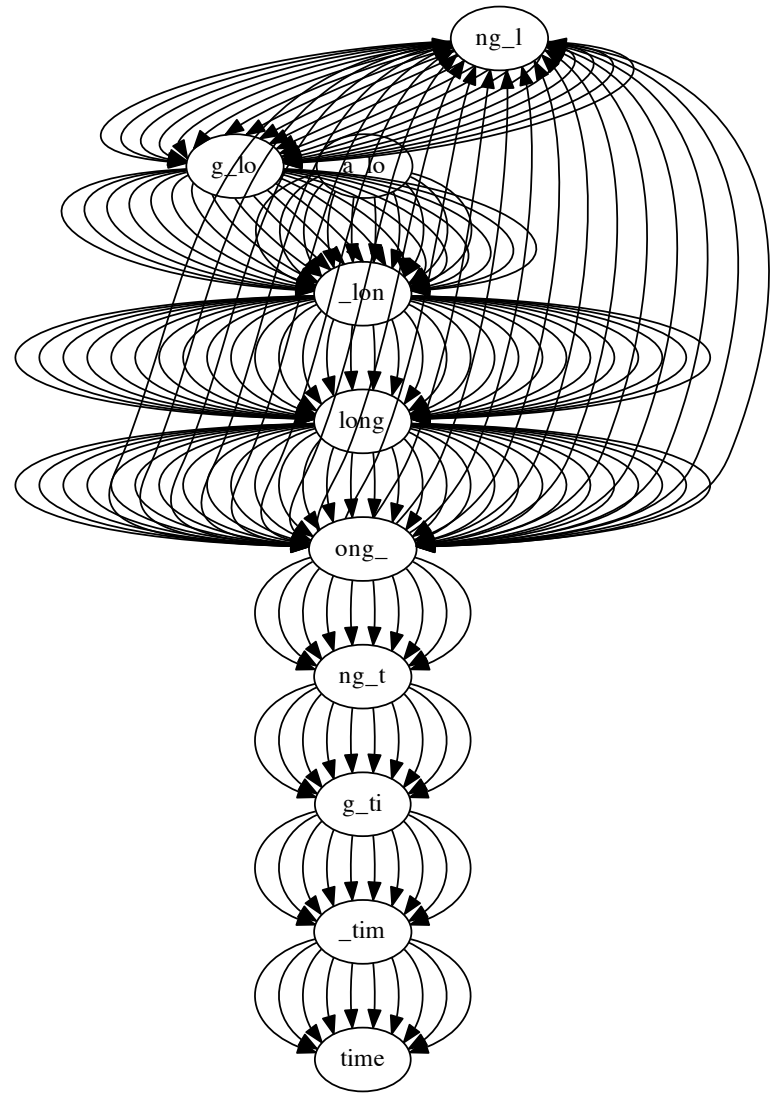
Timed De Bruijn graph construction applied to progressively longer prefixes of lambda phage genome, $k = 14$

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



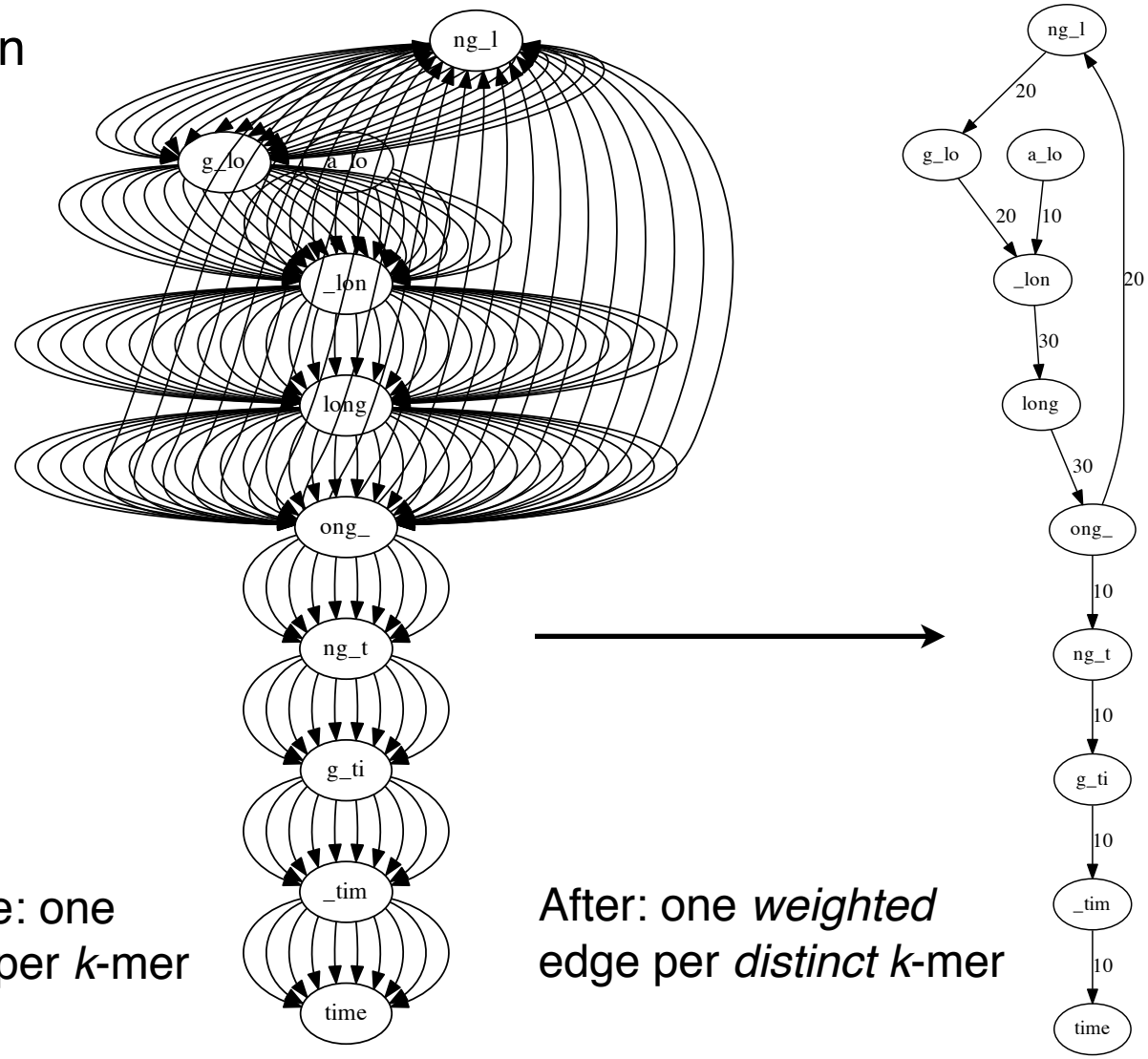
De Bruijn graph

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

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

# 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*))

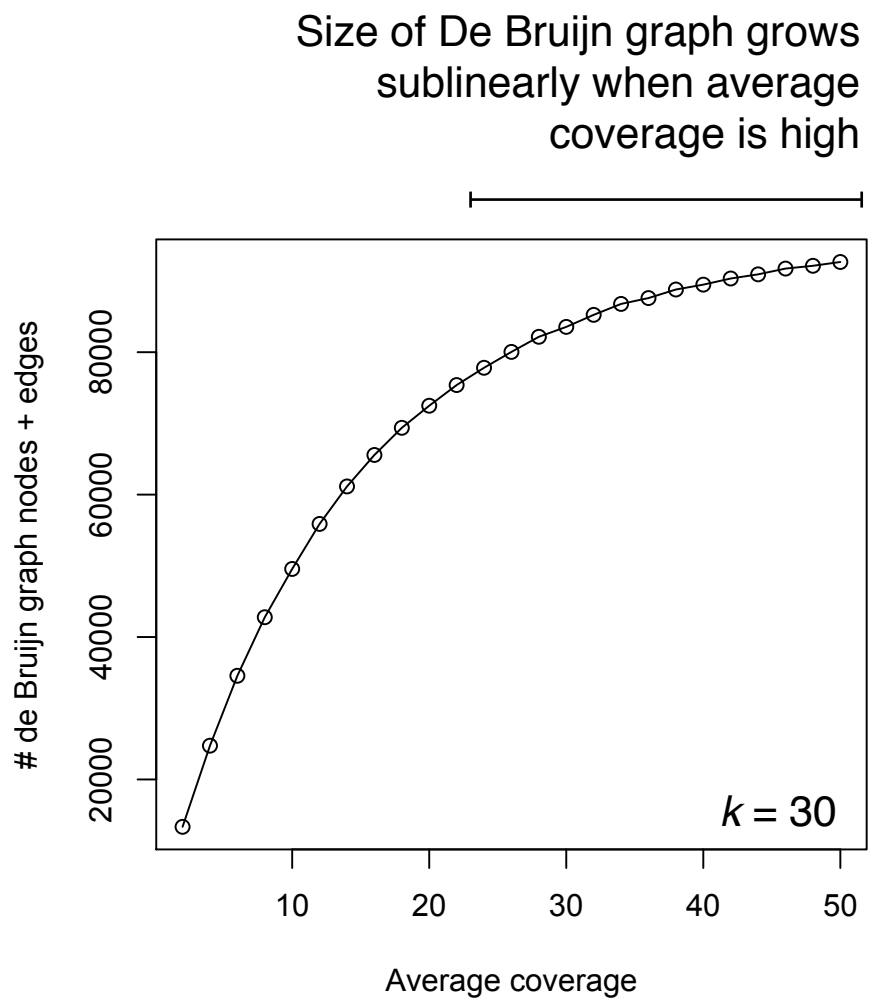
De Bruijn graph

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

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

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

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

What did we give up?

Reads are immediately split into shorter *k*-mers; can't resolve repeats as well as overlap graph

Only a very specific type of "overlap" is considered, which makes dealing with errors more complicated, as we'll see

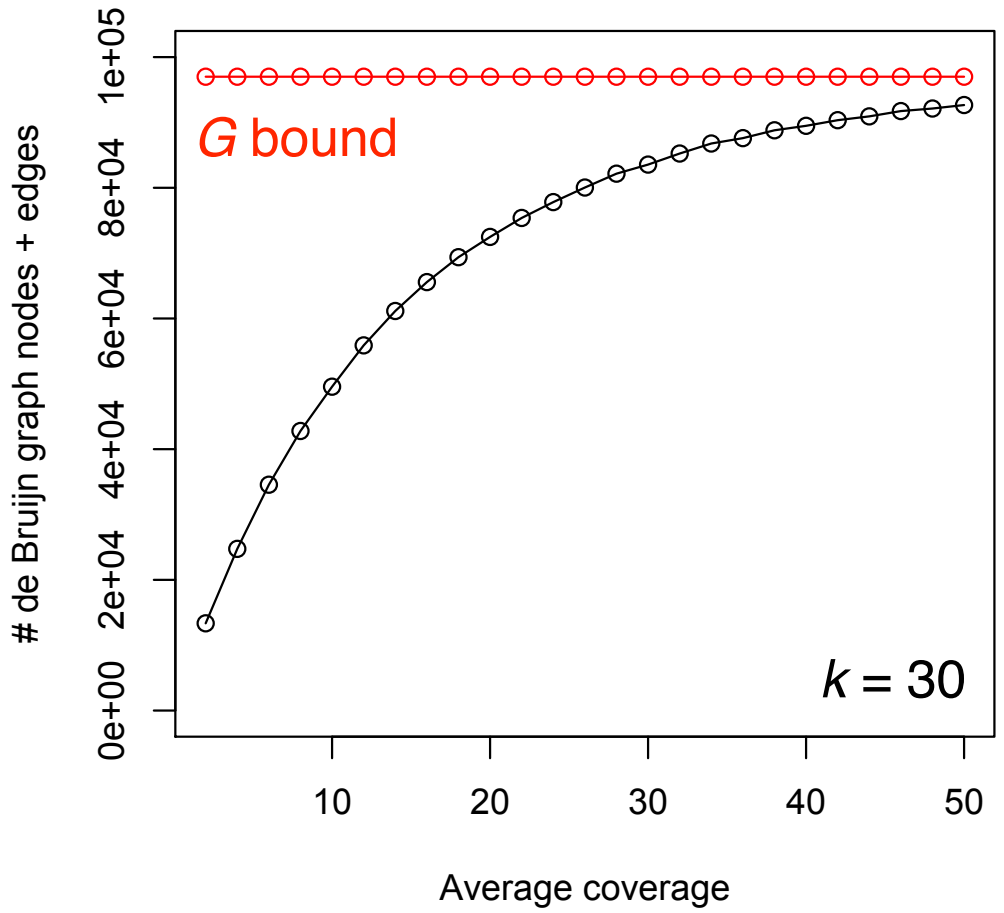
Read coherence is lost. Some paths through De Bruijn graph are inconsistent with respect to input reads.

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

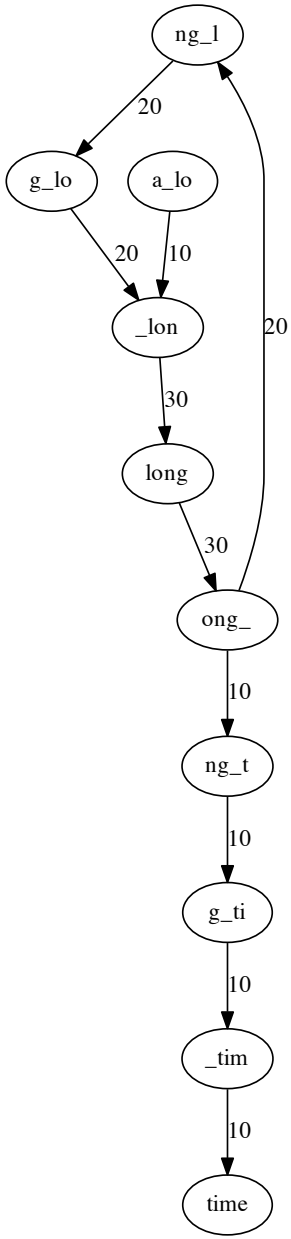
Sequencing errors

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



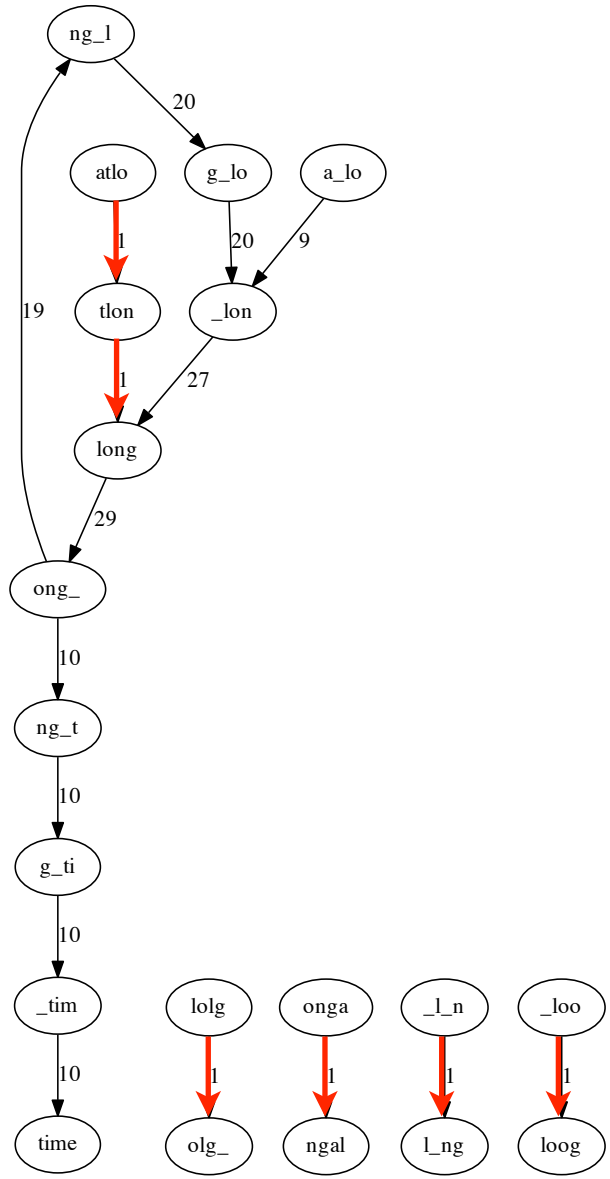
What about data with sequencing errors?

Sequencing Errors



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

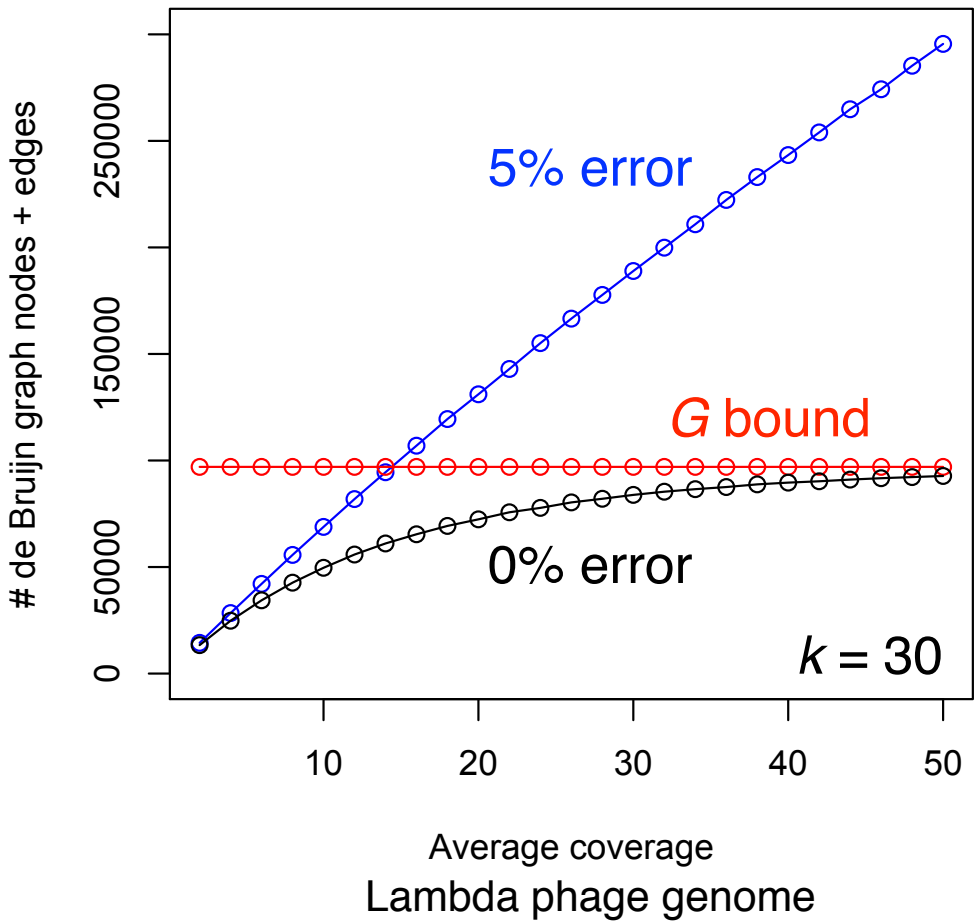
6 errors result in 10 new nodes and 6 new weighted edges, all with weight 1



JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Sequencing Errors

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

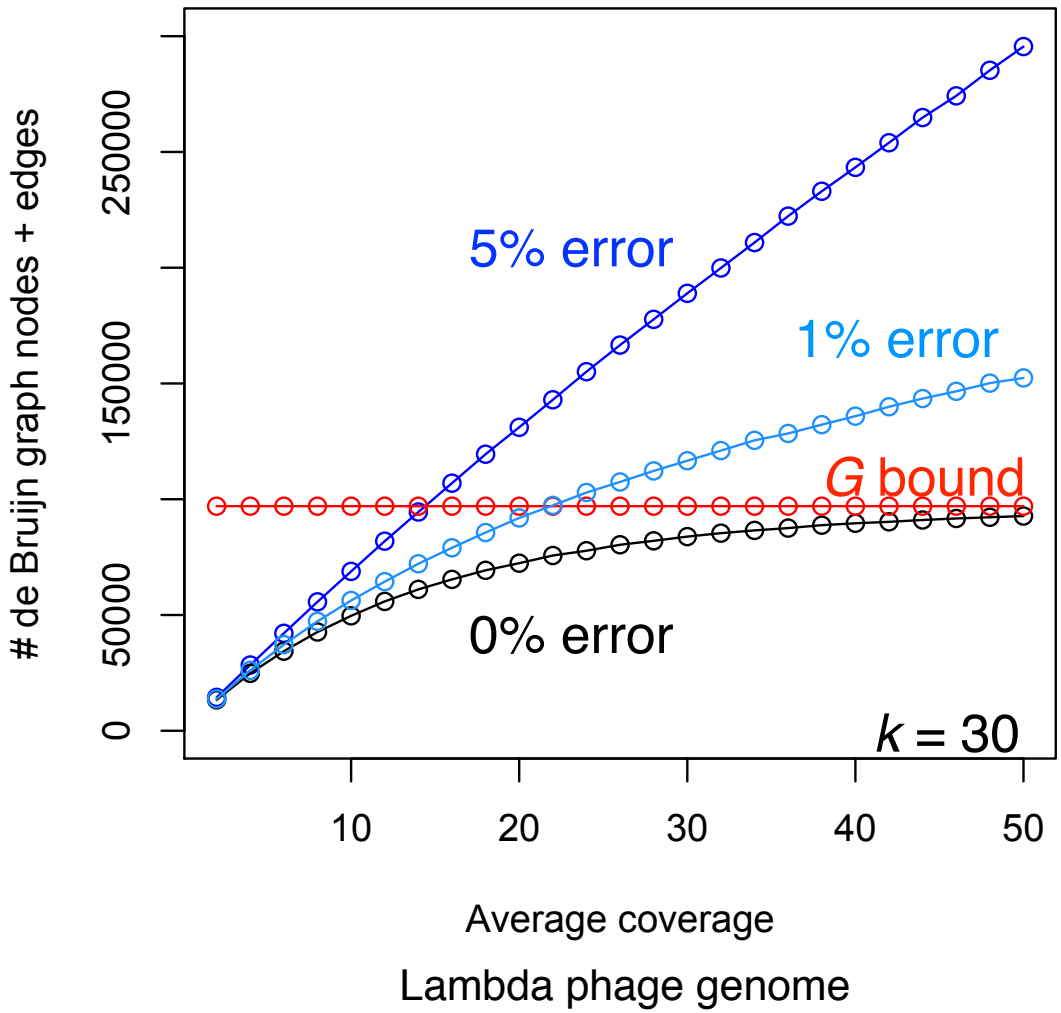


Same experiment as before but with 5% error added

Errors wipe out much of the benefit of the *G* bound

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

Sequencing Errors



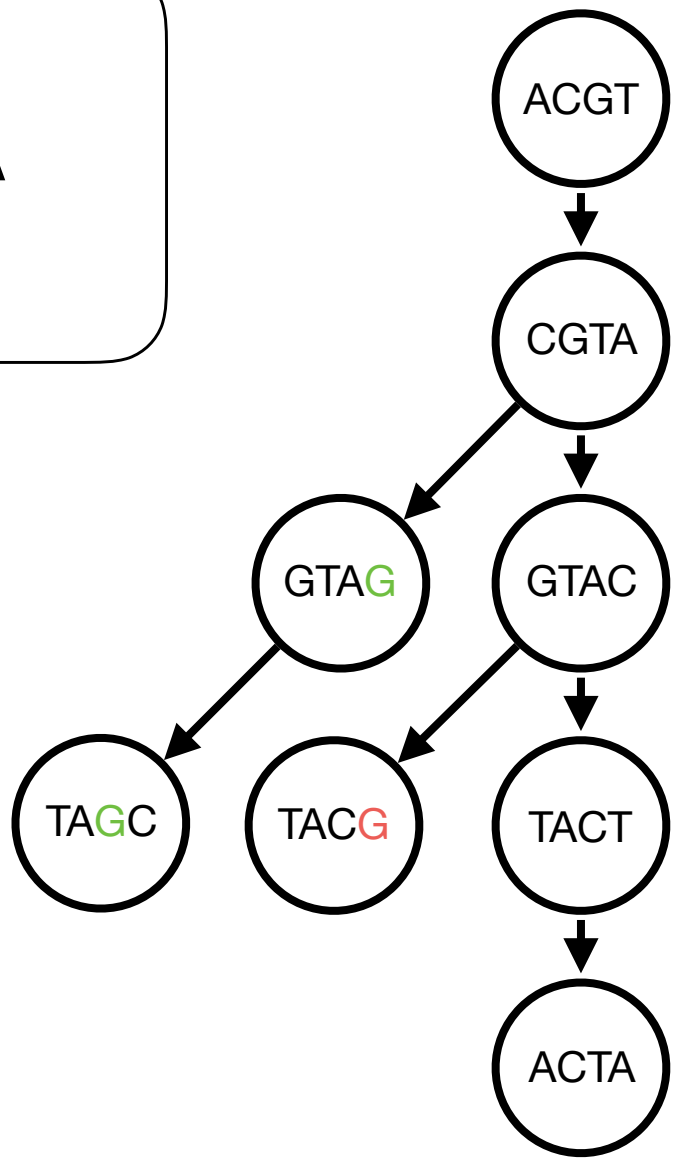
Error cleaning

Errors cause short branches

Branches make contigs stop

We need to “clean” the graph of errors

Reads:
ACGTAC
GTACTA
CGTACG
CGTAGC



Error cleaning

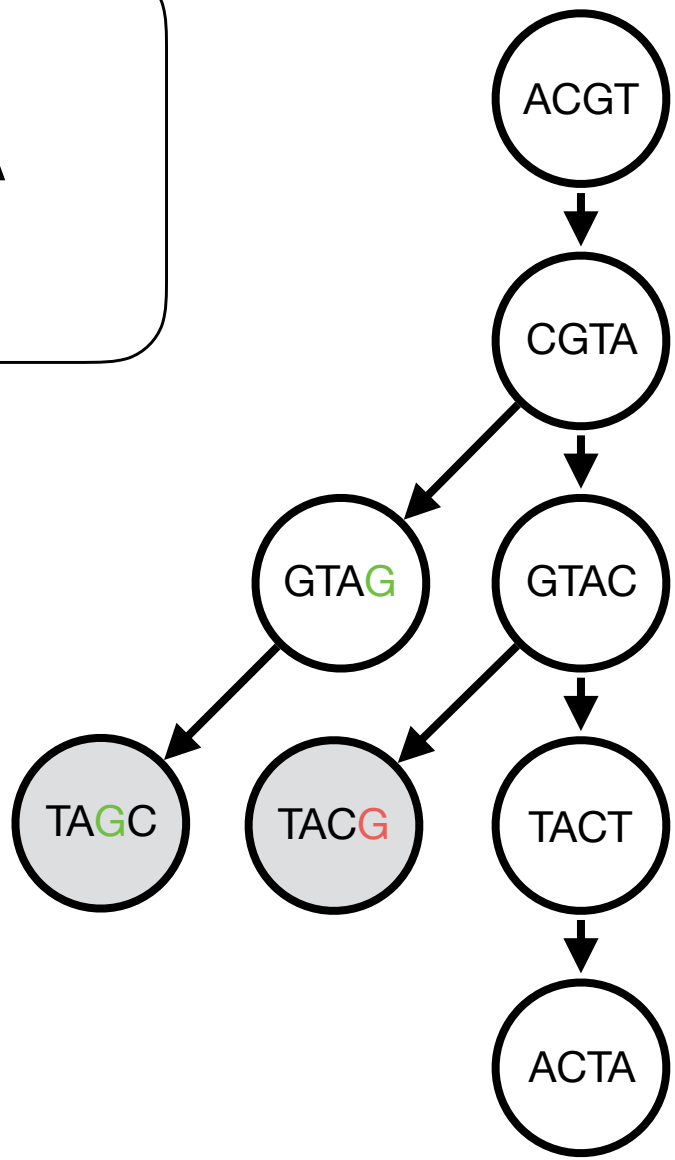
Errors cause short branches

Branches make contigs stop

We need to “clean” the graph of errors

Iteratively remove the semi-connected vertices

Reads:
ACGTAC
GTACTA
CGTACG
CGTAGC



Error cleaning

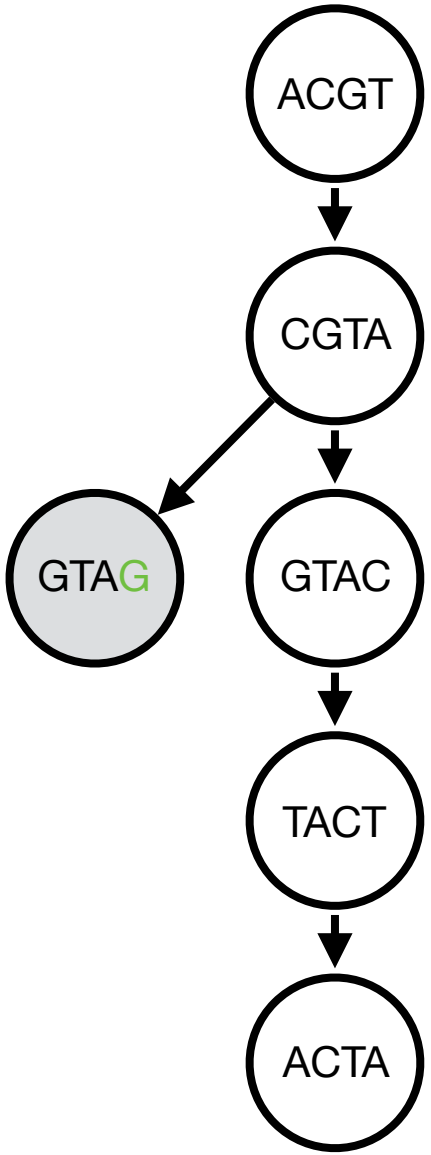
Errors cause short branches

Branches make contigs stop

We need to “clean” the graph of errors

Iteratively remove the semi-connected vertices

Reads:
ACGTAC
GTACTA
CGTACG
CGTAGC



Error cleaning

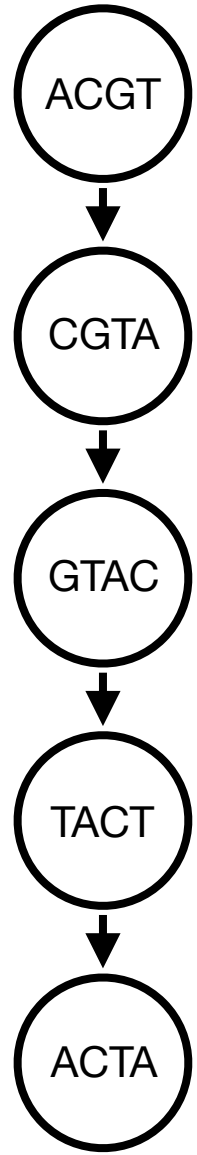
Errors cause short branches

Branches make contigs stop

We need to “clean” the graph of errors

Iteratively remove the semi-connected vertices

Reads:
ACGTAC
GTACTA
CGTACG
CGTAGC



de Bruijn graph assembly

- Construction:
  - Extract *k*-mers from reads, insert into graph
- Refinement:
  - Traverse graph, cleaning errors
- Contig Assembly:
  - Find unambiguous paths, compact into contigs

The size of the graph

Issue with error cleaning approach: the input graph can be huge

First human genome de Bruijn graph from 36bp reads (k=29) had 10 billion vertices

Need either a) approach that can identify errors before loading graph
b) extremely memory efficient representation

Break now, then we'll address this

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?

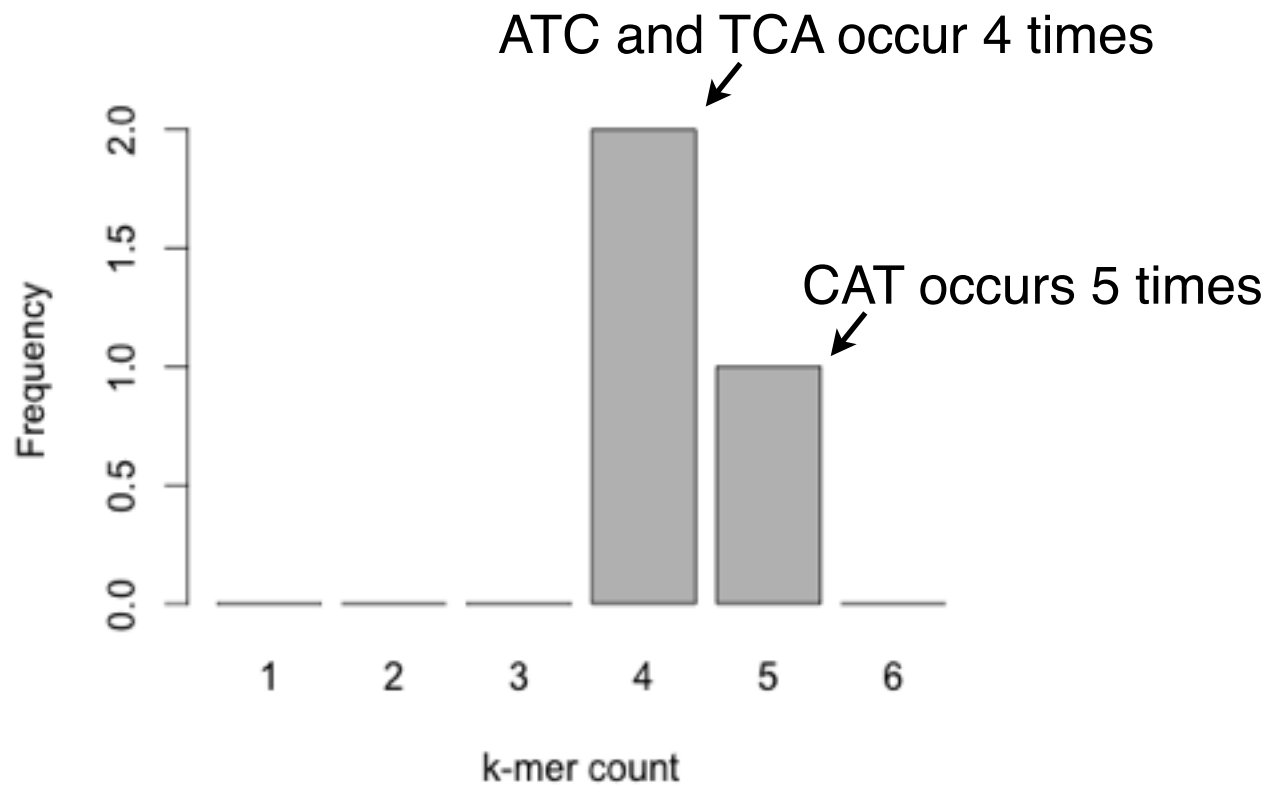
Analogy: design a spell checker for a language you've never seen before. How do you come up with suggestions?

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:

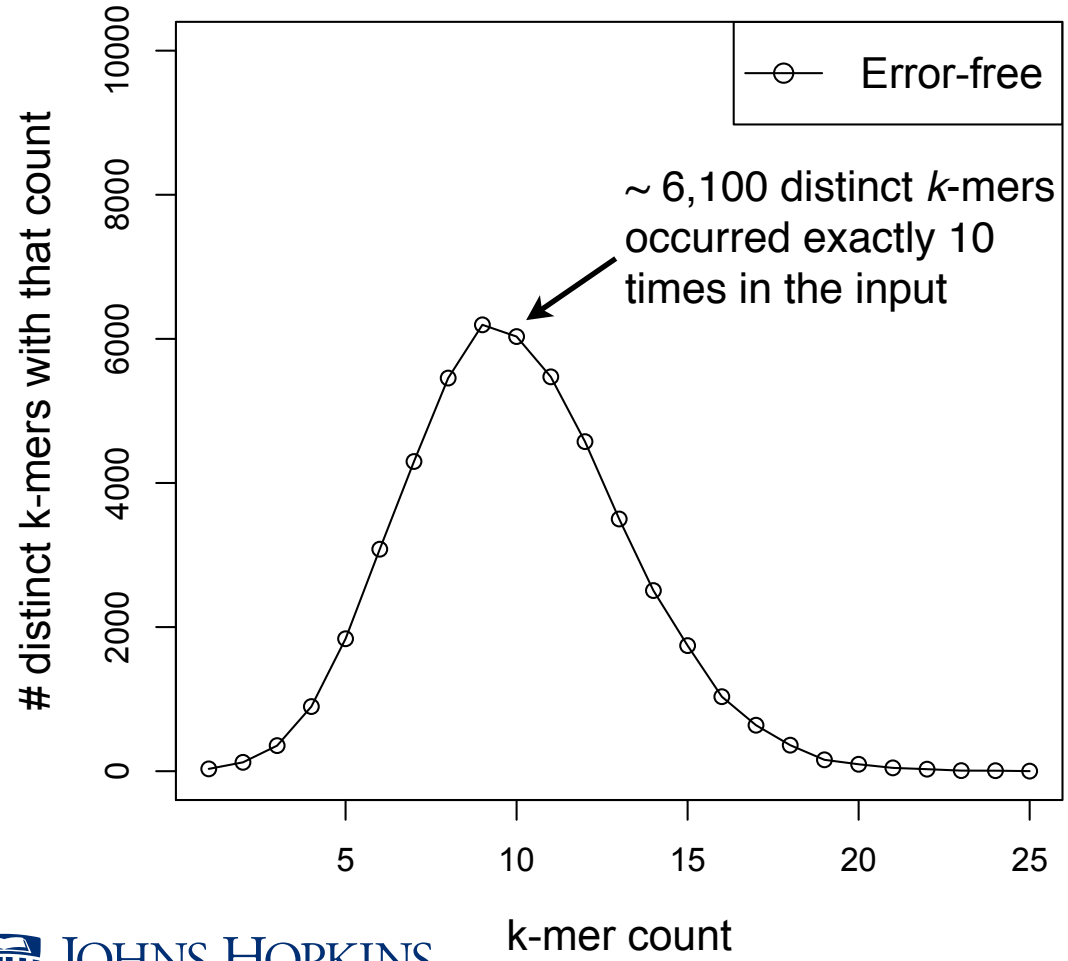


Error correction

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

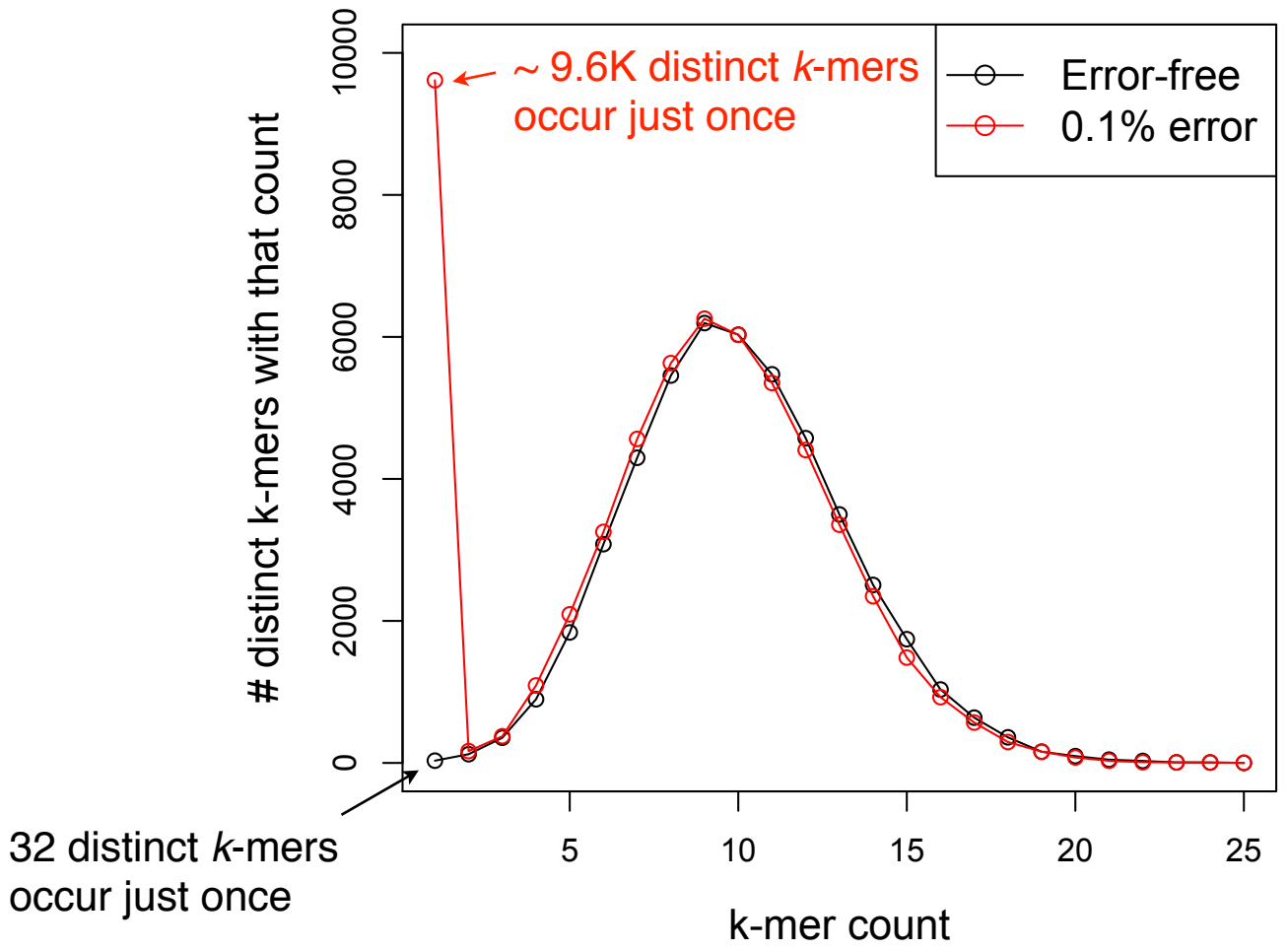
How would the picture change for data with 1% error rate?

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

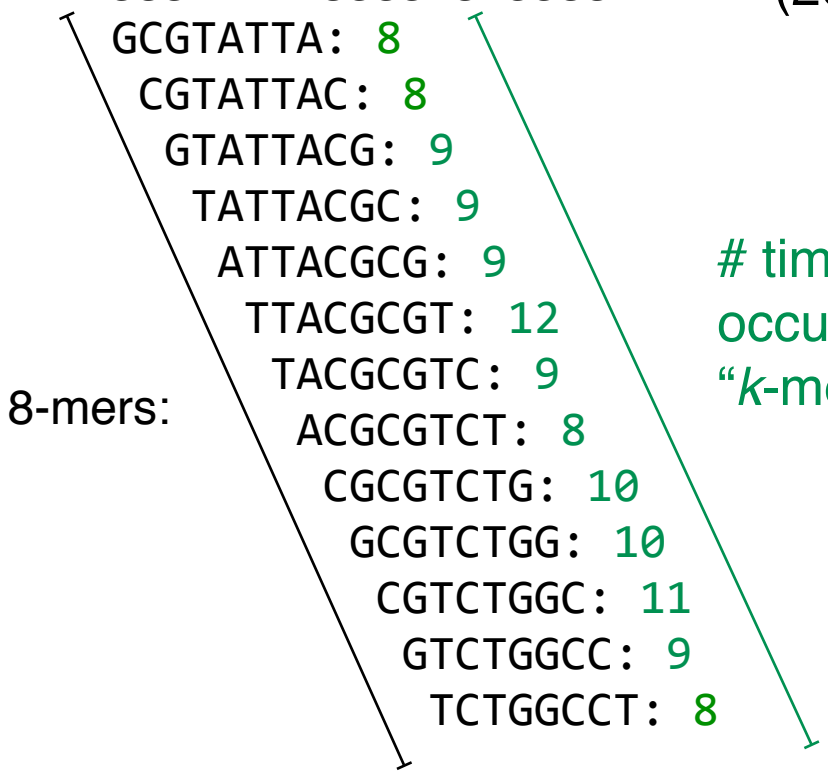


Error correction

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:

Read: GCGTATTACGCGTCTGGCCT (20 nt)



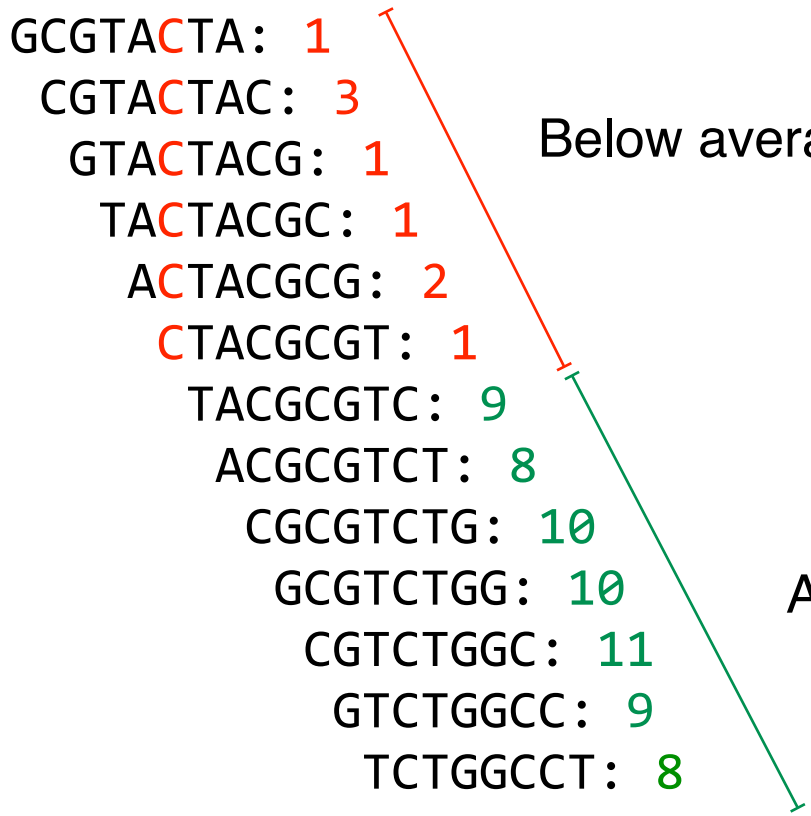
times each 8-mer occurs in the dataset.
"*k*-mer count profile"

All 8-mer counts are around the average, suggesting read is error-free

Error correction

Suppose there's an error

Read: GCGTACTACGCGTCTGGCCT



Below average

Around average

k-mer count profile has corresponding stretch of below-average counts

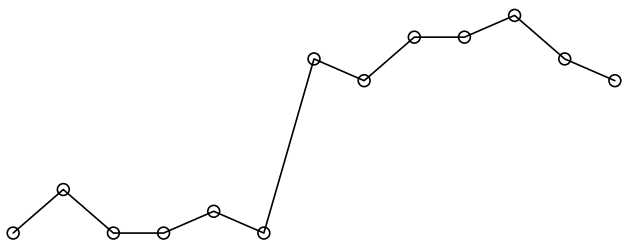
Error correction

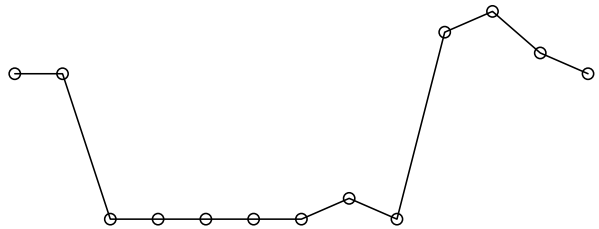
k-mer count profiles when errors are in different parts of the read:

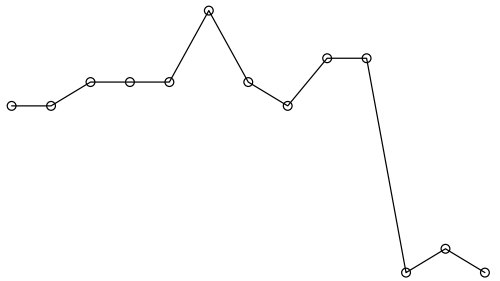
```
GCGTACTACGCGTCTGGCCT
GCGTACTA: 1
 CGTACTAC: 3
  GTACTACG: 1
   TACTACGC: 1
    ACTACGCG: 2
     CTACGCGT: 1
      TACGCGTC: 9
       ACGCGTCT: 8
        CGCGTCTG: 10
         GCGTCTGG: 10
          CGTCTGGC: 11
           GTCTGGCC: 9
            TCTGGCCT: 8
```

```
GCGTATTACACGTCTGGCCT
GCGTATTA: 8
 CGTATTAC: 8
  GTATTACA: 1
   TATTACAC: 1
    ATTACACG: 1
     TTACACGT: 1
      TACACGTC: 1
       ACACGTCT: 2
        CACGTCTG: 1
         GCGTCTGG: 10
          CGTCTGGC: 11
           GTCTGGCC: 9
            TCTGGCCT: 8
```

```
GCGTATTACGCGTCTGGTCT
GCGTATTA: 8
 CGTATTAC: 8
  GTATTACG: 9
   TATTACGC: 9
    ATTACGCG: 9
     TTACGCGT: 12
      TACGCGTC: 9
       ACGCGTCT: 8
        CGCGTCTG: 10
         GCGTCTGG: 10
          CGTCTGGT: 1
           GTCTGGTC: 2
            TCTGGTCT: 1
```

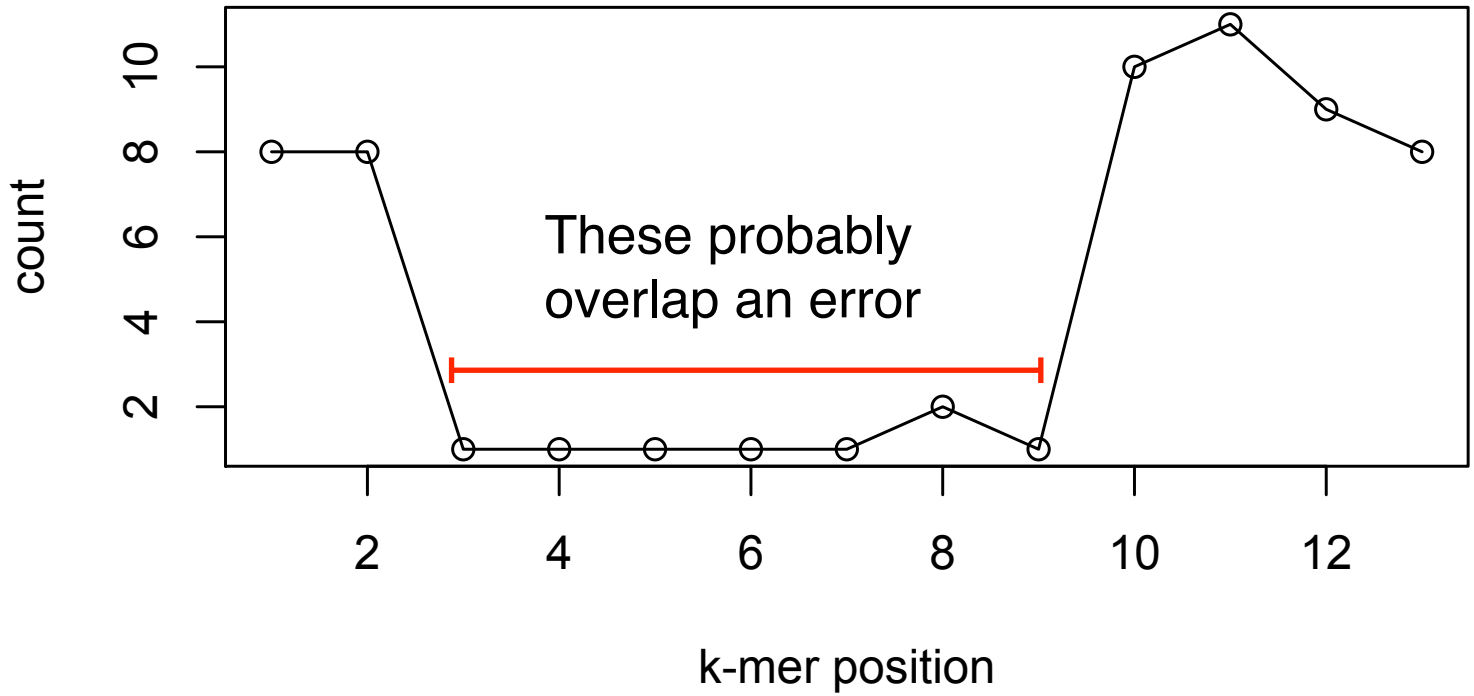






Error correction

k-mer count profile indicates where errors are



Error correction

Simple algorithm: given a count threshold *t*:

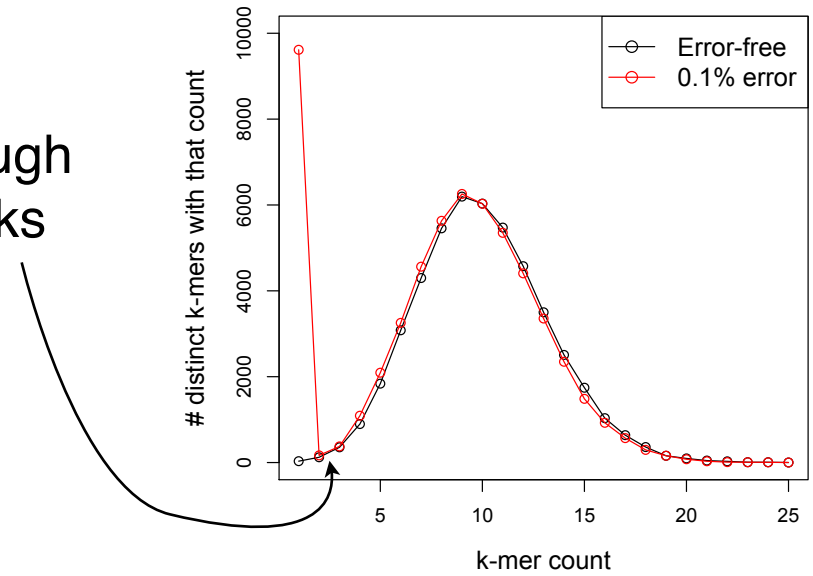
For each read:

For each k-mer:

If *k*-mer count < *t*:

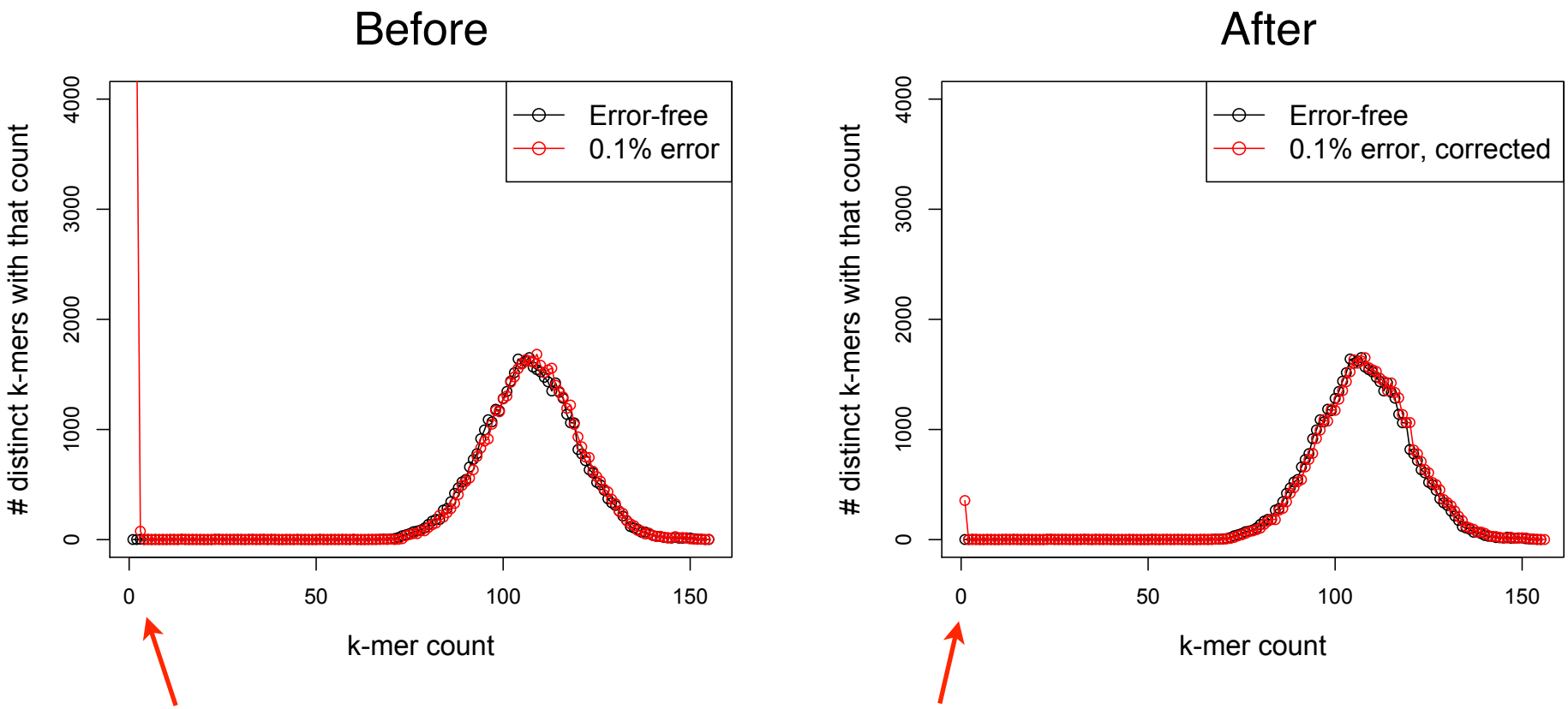
Examine *k*-mer's neighbors within certain Hamming/edit distance.
If neighbor has count ≥ *t*, replace old *k*-mer with neighbor.

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



Error correction: results

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



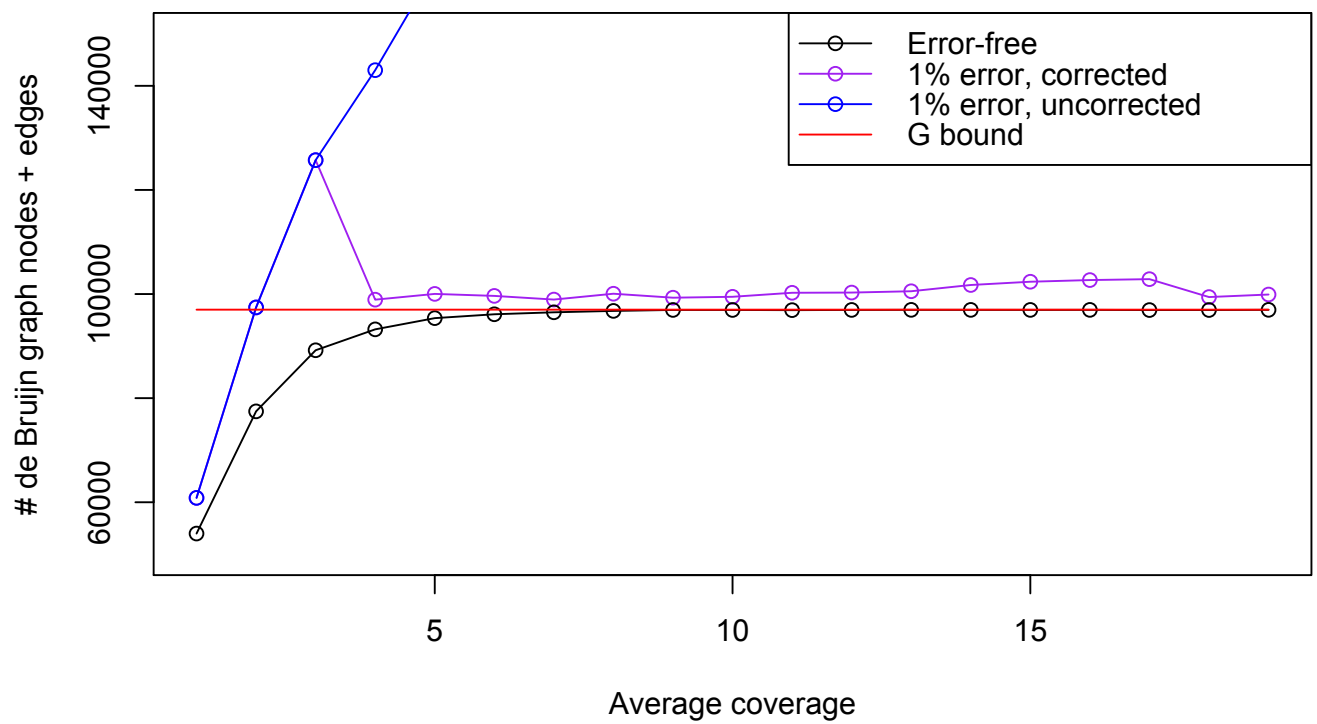
From 194K k-mers occurring exactly once to just 355

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

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



JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Error correction

For error correction to work well:

Average coverage should be high enough and *t* 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

de Bruijn graph assembly

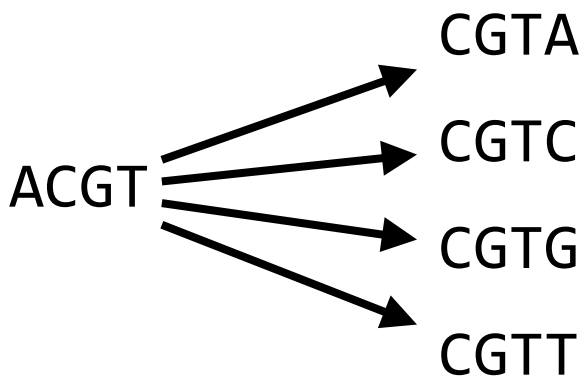
- Two options for reducing memory:
  - ✓ Error correct the reads before entering them into the graph
  - Use memory efficient data structures to represent graph

Representing de Bruijn graphs

- Early dBG assemblers for Illumina reads:
  - no error correction
  - represent edges as list of pointers (64bit/edge)
  - >100 bits per vertex
- Human assembly requires impractical amount of memory (500-1000 GB)

Vertex	Edges
ACGT	2
CCGT	2
CGTT	3,4
GTTA	7
GTTC	8
TACG	0
TCGA	-
TTAC	5
TTCG	6

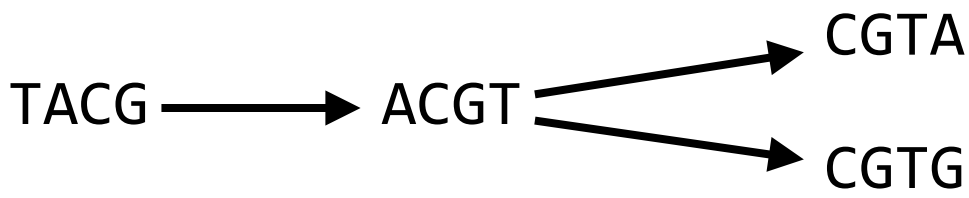
Representing de Bruijn graphs



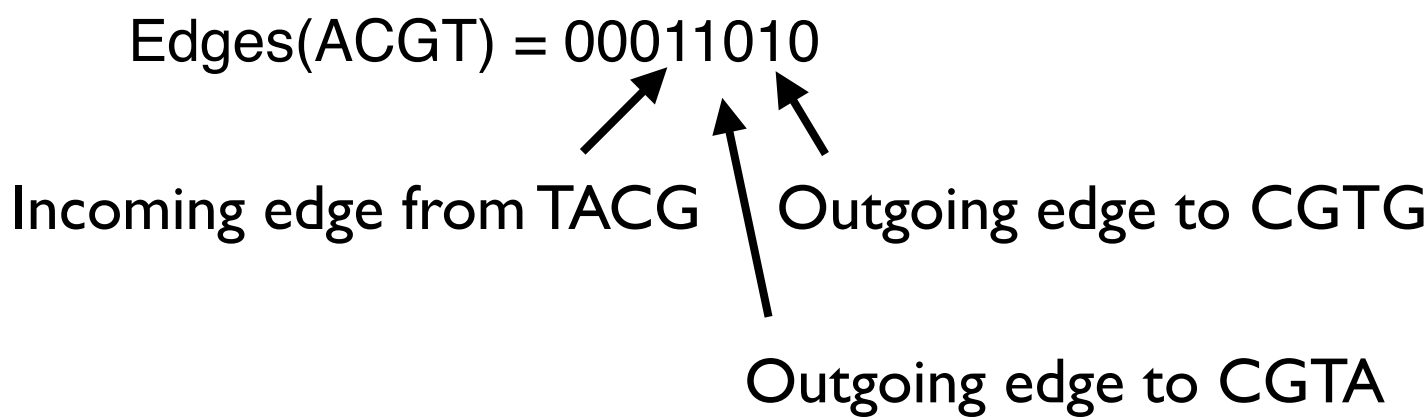
Maximum out degree of any vertex is 4

Representing de Bruijn graphs

How can we compactly represent the edge set for ACGT?

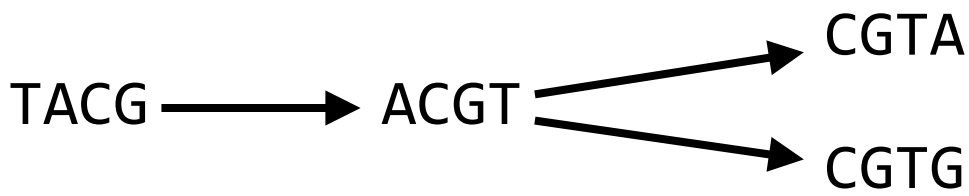


One byte can represent the 4 incoming edges and the 4 outgoing edges:



Representing de Bruijn graphs

How can we compactly represent the edge set for ACGT?



Alternatively, only store *k*-mers and perform set membership queries

S = { TACGT, ACGTA, ACGTG }

outgoing_edges(S, ACGT) = { exists(S, ACGTA), // yes
exists(S, ACGTC), // no
exists(S, ACGTG), // yes
exists(S, ACGTT) } // no

Requires O(|Σ|) queries for each vertex to traverse graph

Set-based dBG

de Bruijn graph representation problem: how much memory is required to represent a set of *k*-mers, *S*?

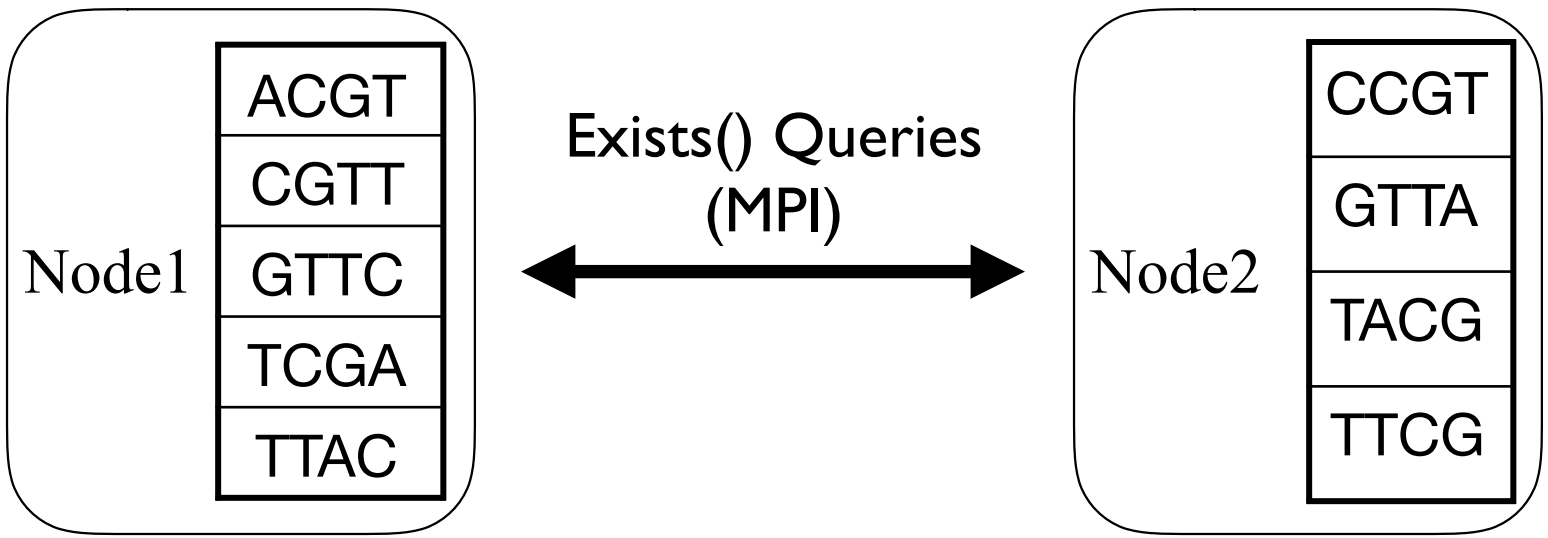
S = { ACGT, CCGT, CGTT, GTTA, GTTC, TACG, TCGA, TTAC, TTCG }

Distributed hash table dBG

One approach: represent *S* in a distributed hash table, use MPI to implement edge queries

330GB of memory distributed to assemble a human genome

S = { ACGT, CCGT, CGTT, GTTA, GTTC, TACG TCGA, TTAC, TTCG }

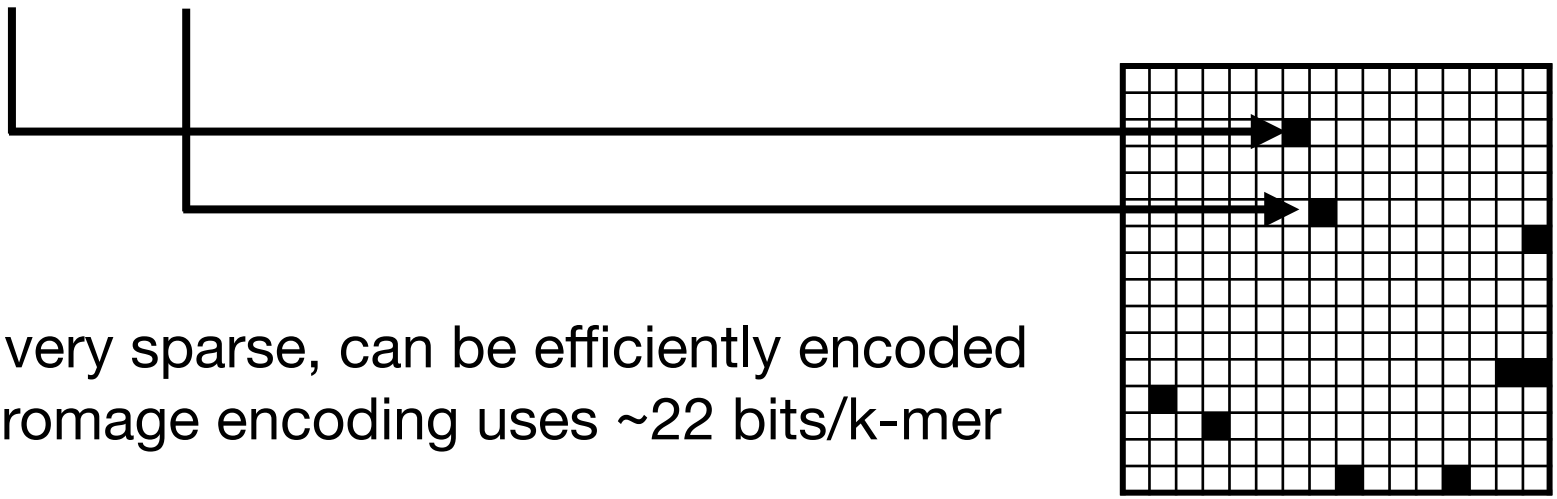


Sparse Bit Vector dBG

Conway and Bromage (2011) represent *S* using a sparse bit vector

Map each *k*-mer to a number in [0, 4^k), set the corresponding bit

S = { ACGT, CCGT, CGTT, GTTA, GTTC, TACG TCGA, TTAC, TTCG }



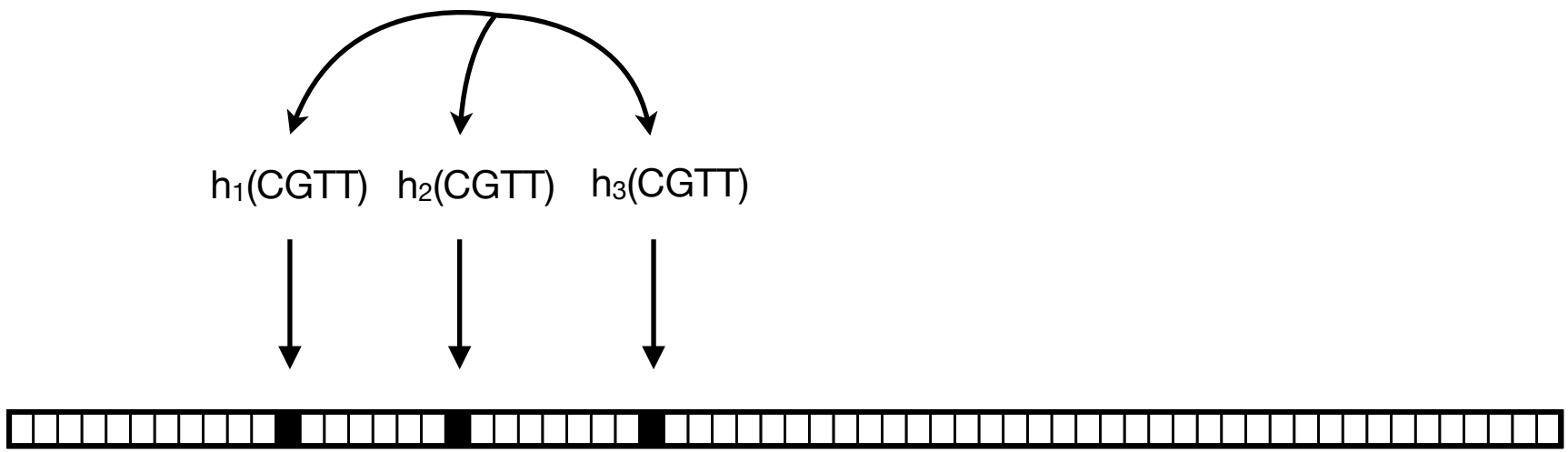
Bit vector is very sparse, can be efficiently encoded
Conway & Bromage encoding uses ~22 bits/k-mer

Bloom Filter dBG

Alternative: encode S using a bloom filter (Melsted and Pritchard 2011, Pell et all 2011)

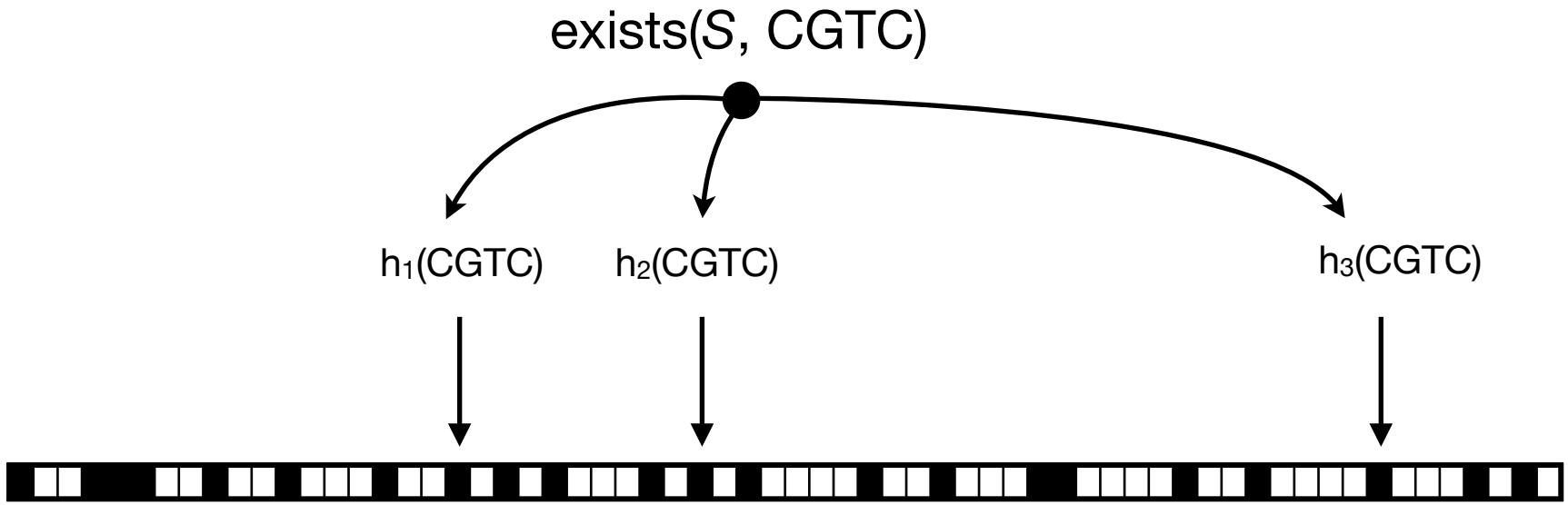
Hash each *k*-mer *h* times, set *h* bits in a bit vector of size *n*

S = { ACGT, CCGT, CGTT, GTTA, GTTC, TACG TCGA, TTAC, TTCG }



Bloom Filter dBG

To query whether a *k*-mer is in *S,* use the same hash functions and check whether all bits are set.



Bloom filter may return *false positives* - reports that an item is present when it was never added. It won't report false negatives however.

False positive rate is tuneable by *n* and the number of hash functions *h*

Bloom Filter dBG

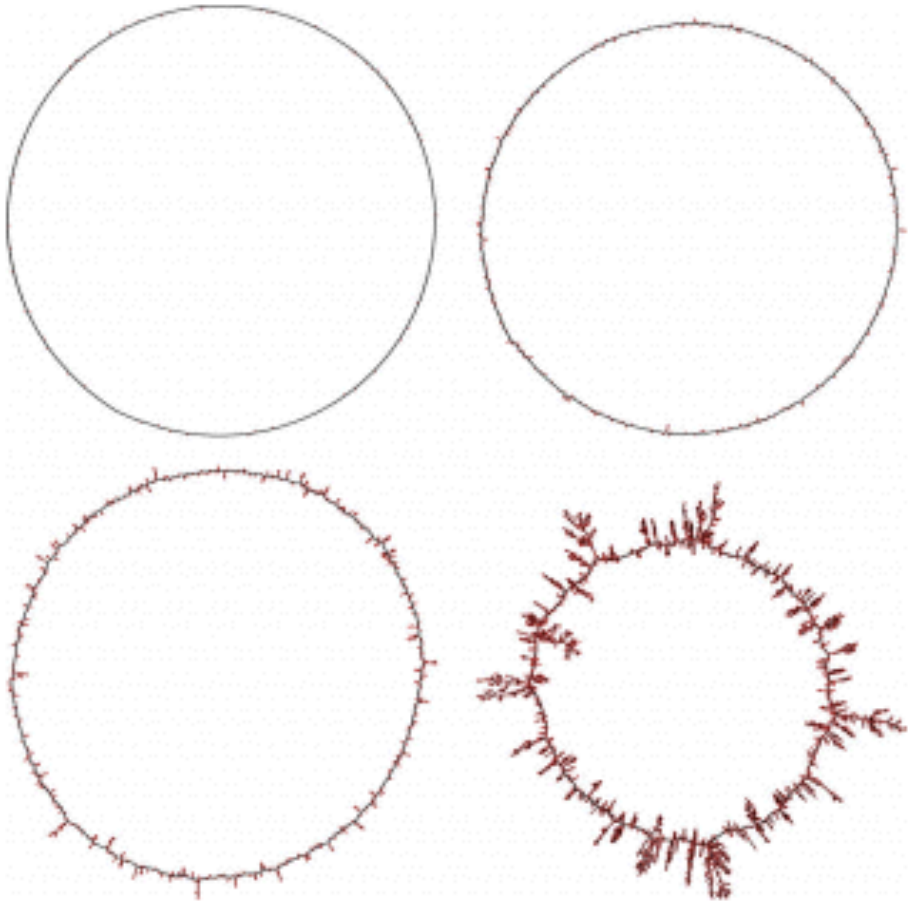


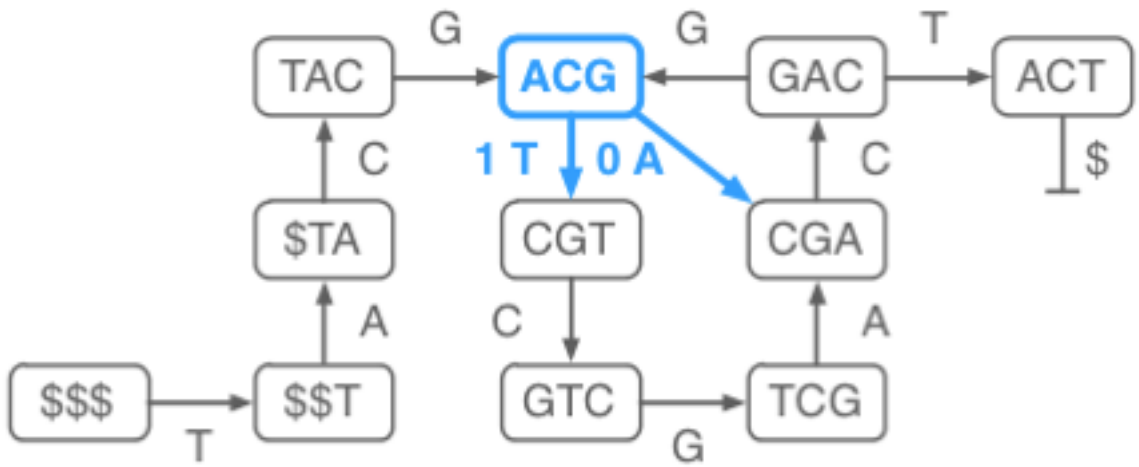
Table 1. Bits per *k*-mer for various false positive rates

False positive rate	Bits/*k*-mer
0.1%	14.35
1%	9.54
5%	6.22
10%	4.78
15%	3.94
20%	3.34

Figures from Pell et al, PNAS 2012

Representing de Bruijn graphs

The FM-index can be adapted to efficiently representing sets of fixed length strings. Two methods:
Bowe et al: “Succinct de Bruijn graphs” (5 bits per k-mer)
Rødland: “kFM-index” (6 bits per *k*-mer)



L	Node	W
1	$ $ $	T
1	C G A	C
1	$ T A	C
0	G A C	G
1	G A C	T
1	T A C	G-
1	G T C	G
0	A C G	A
1	A C G	T
1	T C G	A-
1	$ $ T	A
1	A C T	$
1	C G T	C

Recommended: http://alexbowe.com/succinct-debruijn-graphs/

de Bruijn Graph Representation Summary

- Pointer based: >100 bits per *k*-mer (2007)
- (distributed) hash set: 100 bits (2008)
- Sparse bit vector: 22 bits (2010)
- Bloom filter: 5-10 bits (2011)
- FM-index of reads: 25 bits (2012)
- Succinct dBG/kFM-index: 5 or 6 bits (2013)

>20X improvement in memory use in 5 years of development
Most new assemblers are based on bloom filters - easy to code, fast queries, adaptable performance (see ABySS 2.0 Biorxiv)

Summary

- Fast to construct de Bruijn graph
- Memory depends on error rate and sequence depth
  - Can error correct to reduce # vertices
  - Can represent as set in small memory
- Use cleaning algorithms to get rid of errors
- Dominant assembly paradigm for Illumina reads
- Overlap-based methods are popular again for long reads (PacBio, Oxford Nanopore)