The de Bruijn graph and genome assembly

Lecture slides adapted from the dBG lecture slides of Ben Langmead.
All slides in this lecture marked with “*” courtesy of Ben Langmead.

Different kind of graph

“tomorrow and tomorrow and tomorrow”

Lecture slides adapted from the dBG lecture slides of Ben Langmead.
All slides in this lecture marked with “*” courtesy of Ben Langmead.

*

Different kind of graph

“tomorrow and tomorrow and tomorrow”

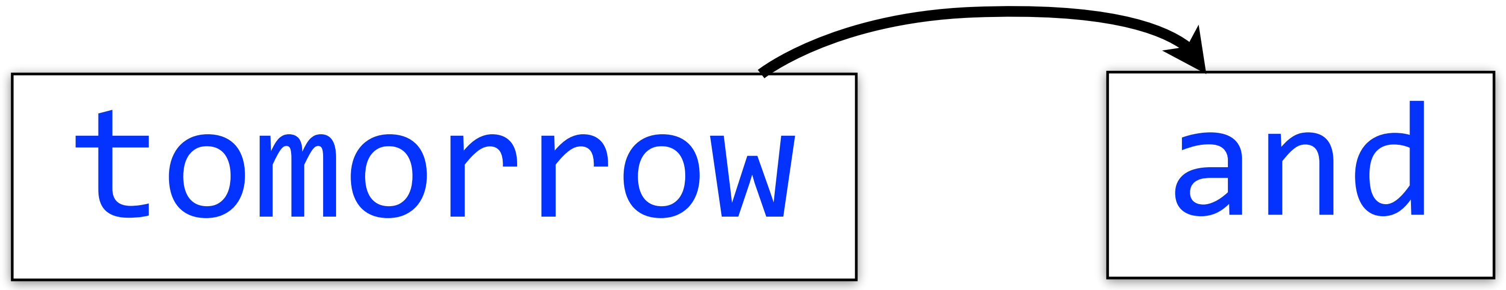
`tomorrow` `and`

Lecture slides adapted from the dBG lecture slides of Ben Langmead.
All slides in this lecture marked with “*” courtesy of Ben Langmead.

*

Different kind of graph

"tomorrow and tomorrow and tomorrow"

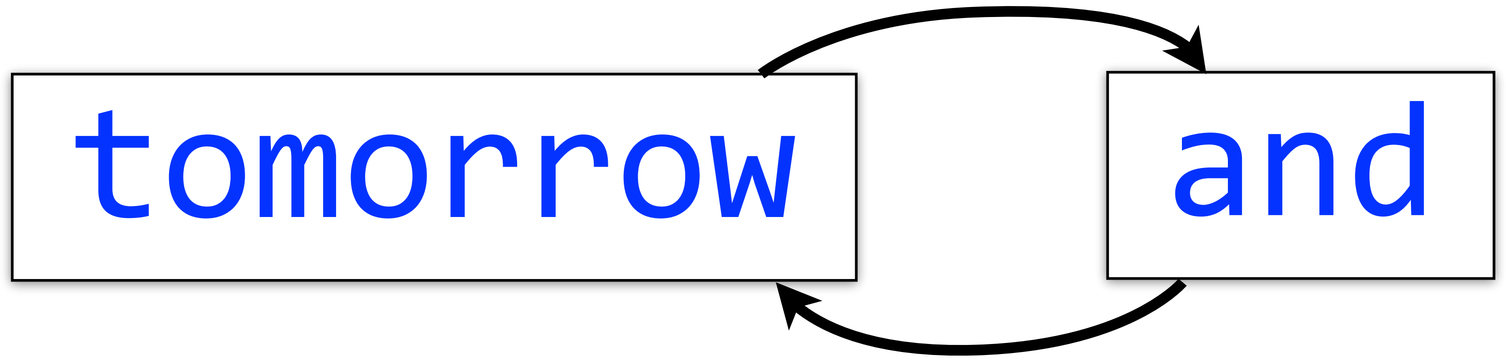


Lecture slides adapted from the dBG lecture slides of Ben Langmead.
All slides in this lecture marked with "*" courtesy of Ben Langmead.

*

Different kind of graph

“tomorrow and tomorrow and tomorrow”

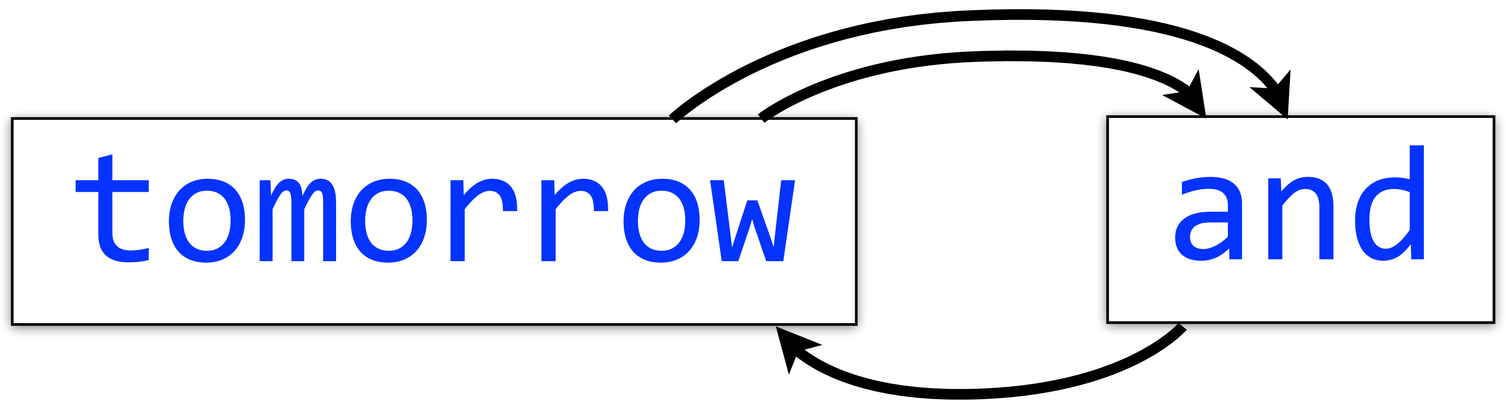


Lecture slides adapted from the dBG lecture slides of Ben Langmead.
All slides in this lecture marked with “*” courtesy of Ben Langmead.

*

Different kind of graph

“tomorrow and tomorrow and tomorrow”

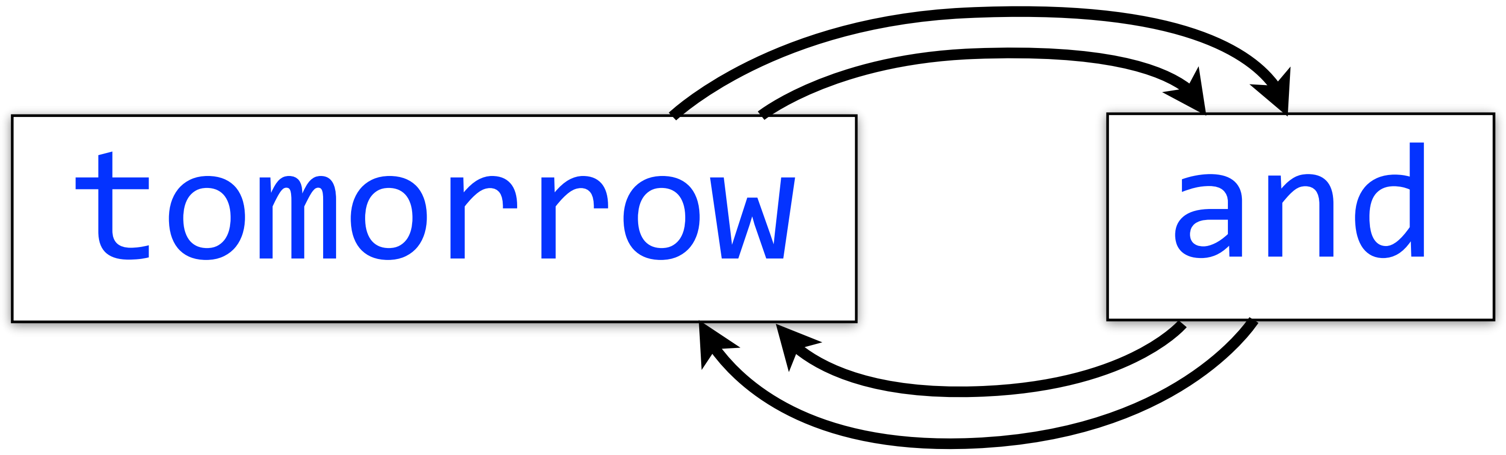


Lecture slides adapted from the dBG lecture slides of Ben Langmead.
All slides in this lecture marked with “*” courtesy of Ben Langmead.

*

Different kind of graph

"tomorrow and tomorrow and tomorrow"

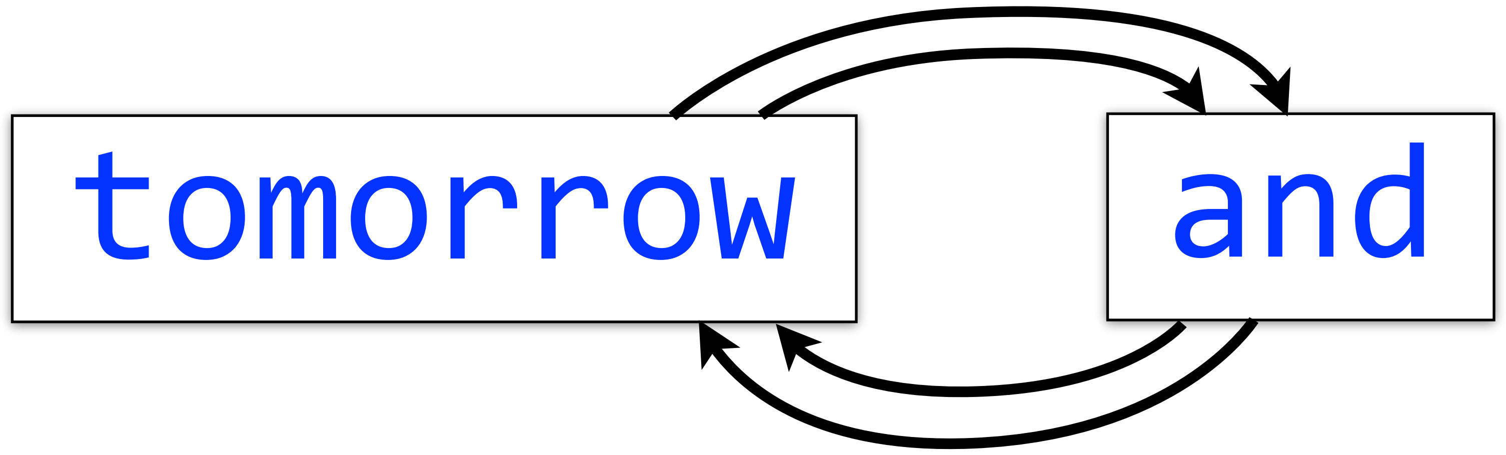


Lecture slides adapted from the dBG lecture slides of Ben Langmead.
All slides in this lecture marked with "*" courtesy of Ben Langmead.

*

Different kind of graph

"tomorrow and tomorrow and tomorrow"



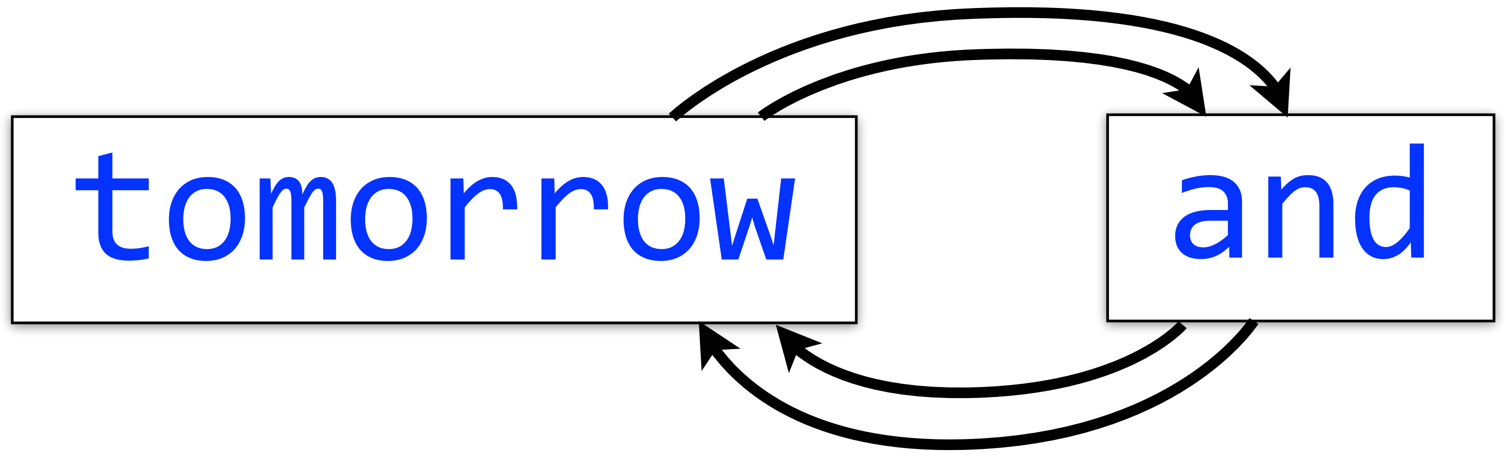
An edge represents an ordered pair of adjacent words in the input

Lecture slides adapted from the dBG lecture slides of Ben Langmead.
All slides in this lecture marked with "*" courtesy of Ben Langmead.

*

Different kind of graph

"tomorrow and tomorrow and tomorrow"



An edge represents an ordered pair of adjacent words in the input

Multigraph: there can be more than one edge from node A to node B

Lecture slides adapted from the dBG lecture slides of Ben Langmead.
All slides in this lecture marked with "*" courtesy of Ben Langmead.

*

De Bruijn graph

genome: AAABBBBA

*

De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

*

De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA

De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA

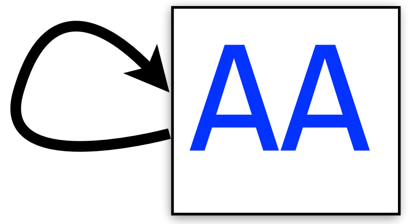
AA

De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA

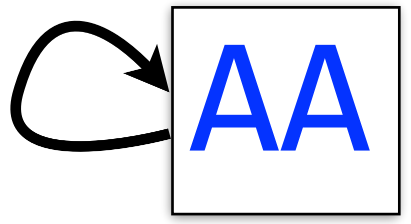


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA AA, AB

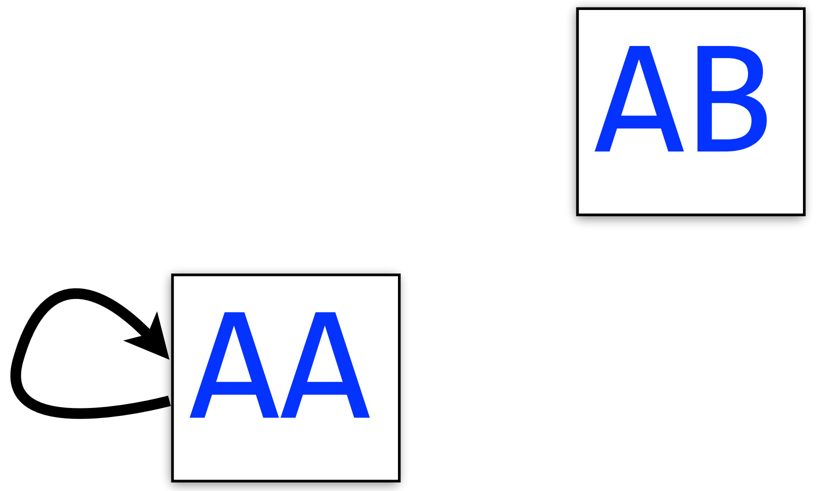


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA AA, AB

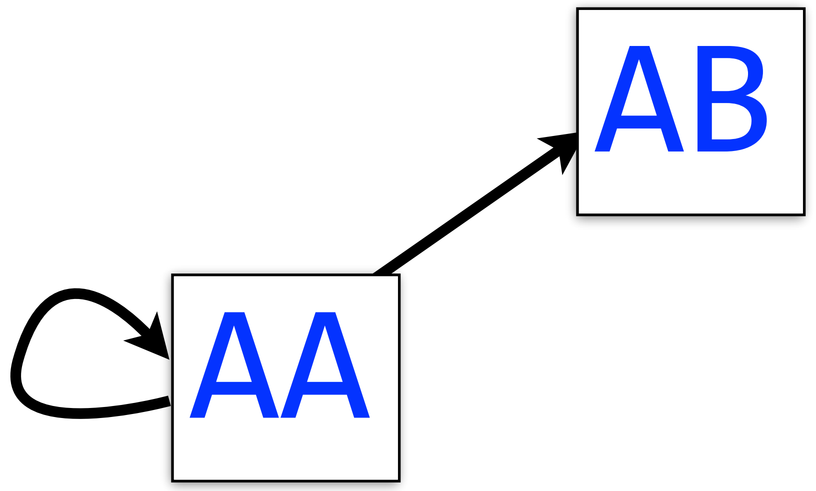


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA AA, AB

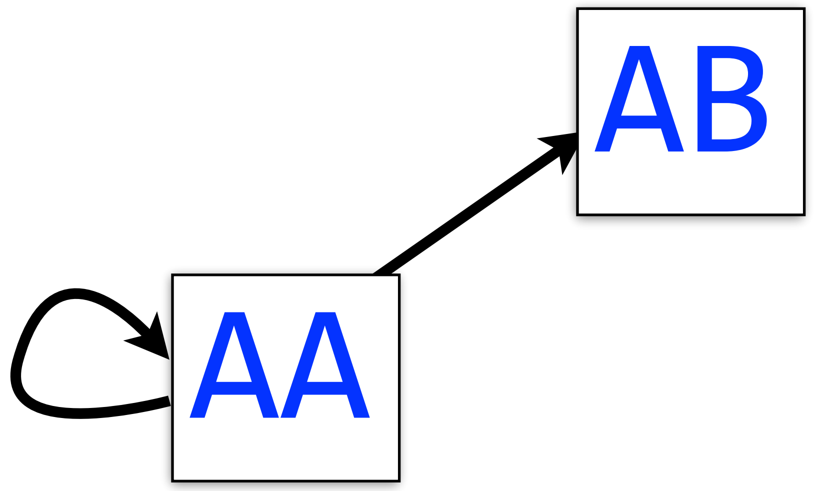


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA AA, AB AB, BB

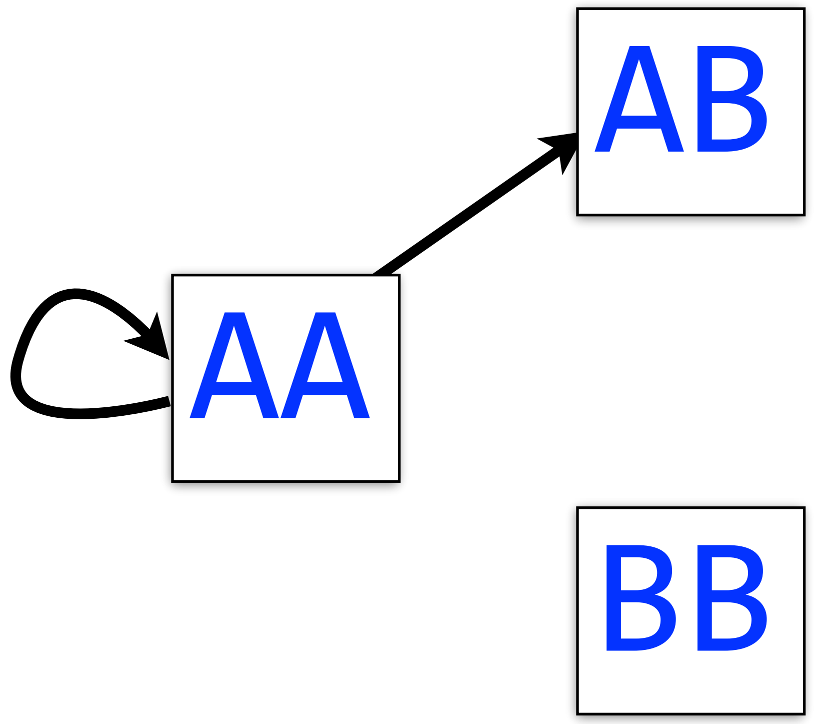


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA AA, AB AB, BB

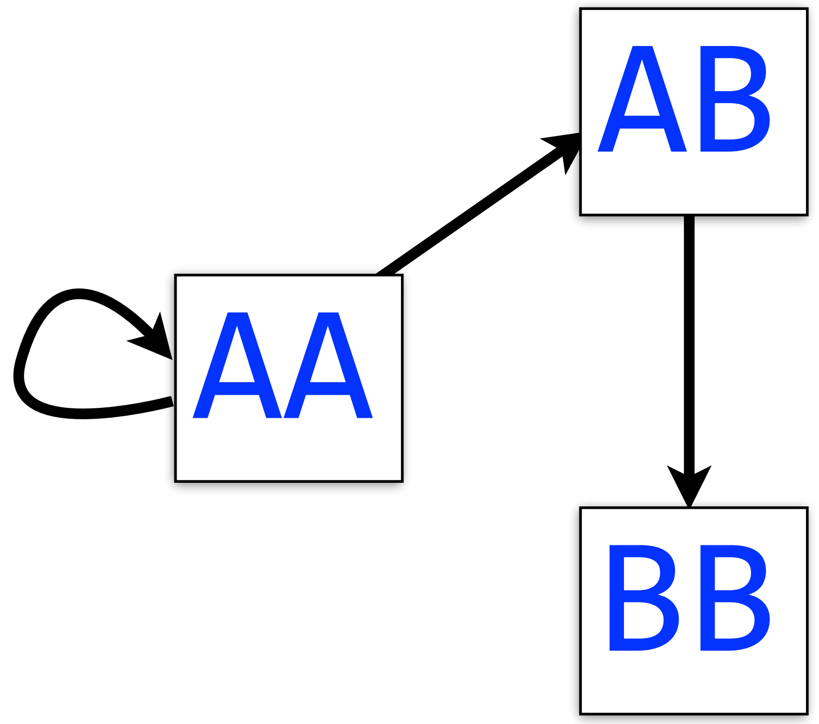


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA AA, AB AB, BB

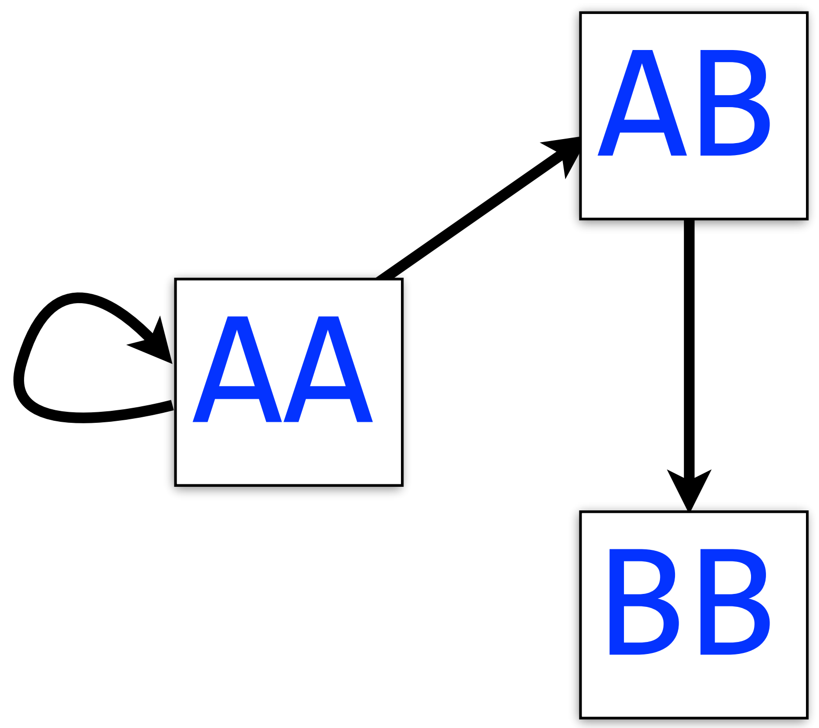


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA AA, AB AB, BB BB, BB

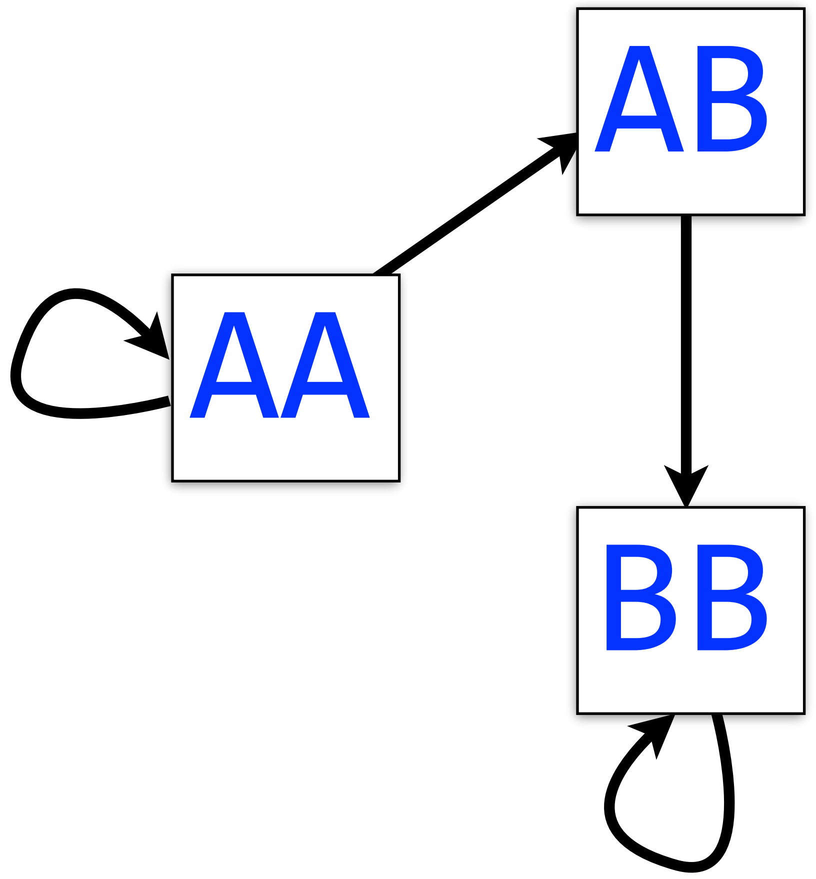


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA   AA, AB   AB, BB   BB, BB

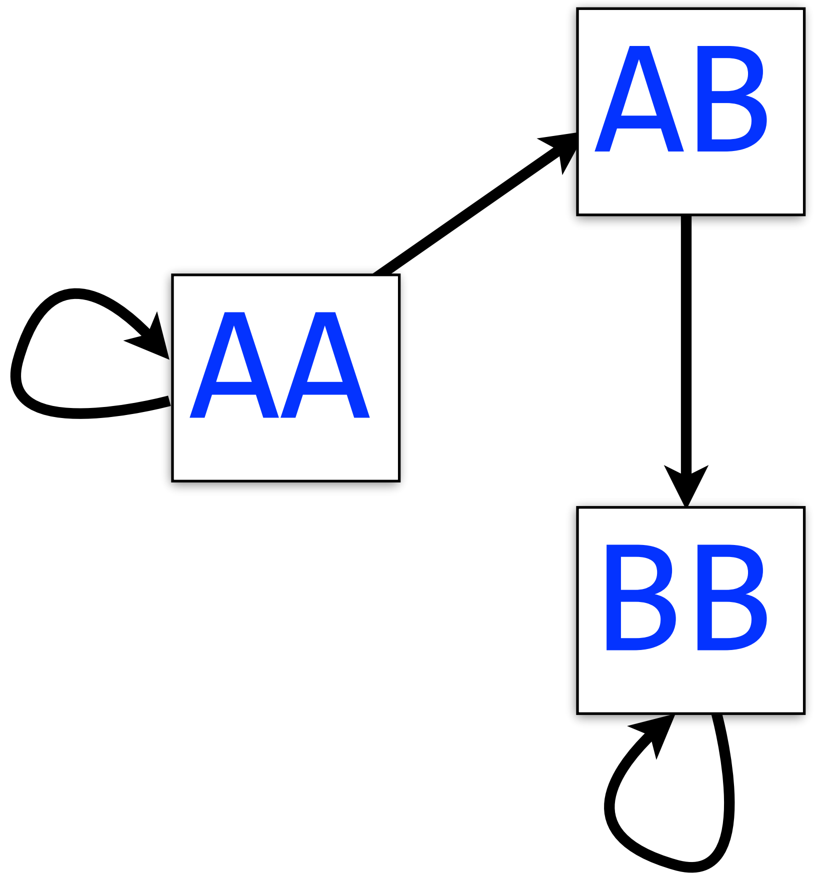


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA AA, AB AB, BB BB, BB BB, BB

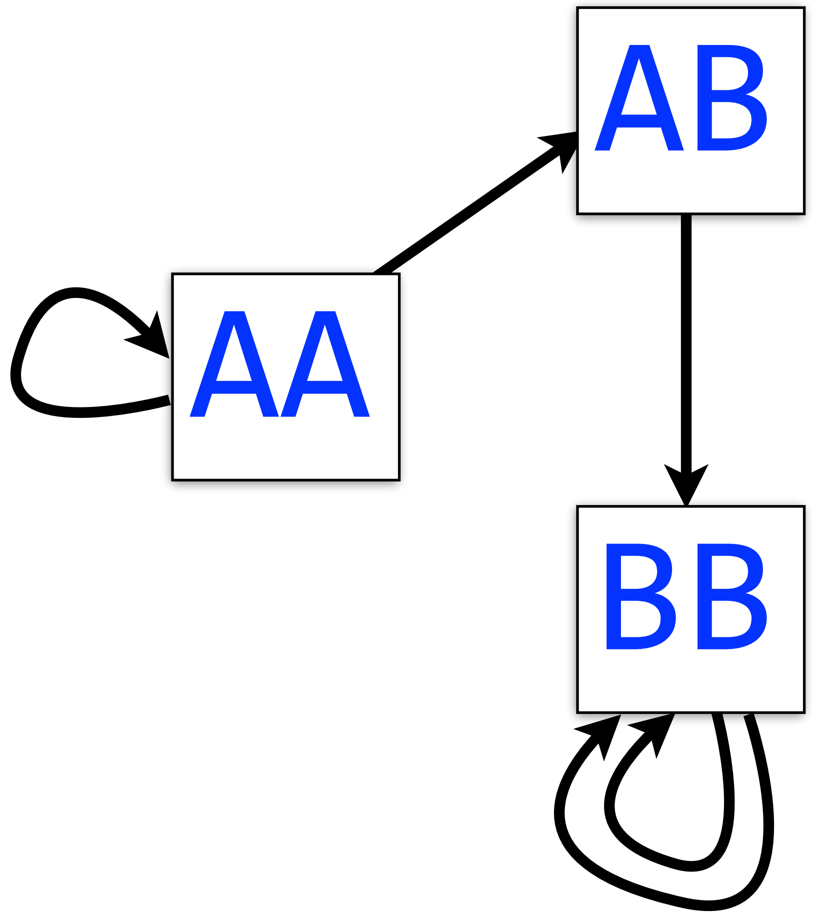


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA AA, AB AB, BB BB, BB BB, BB

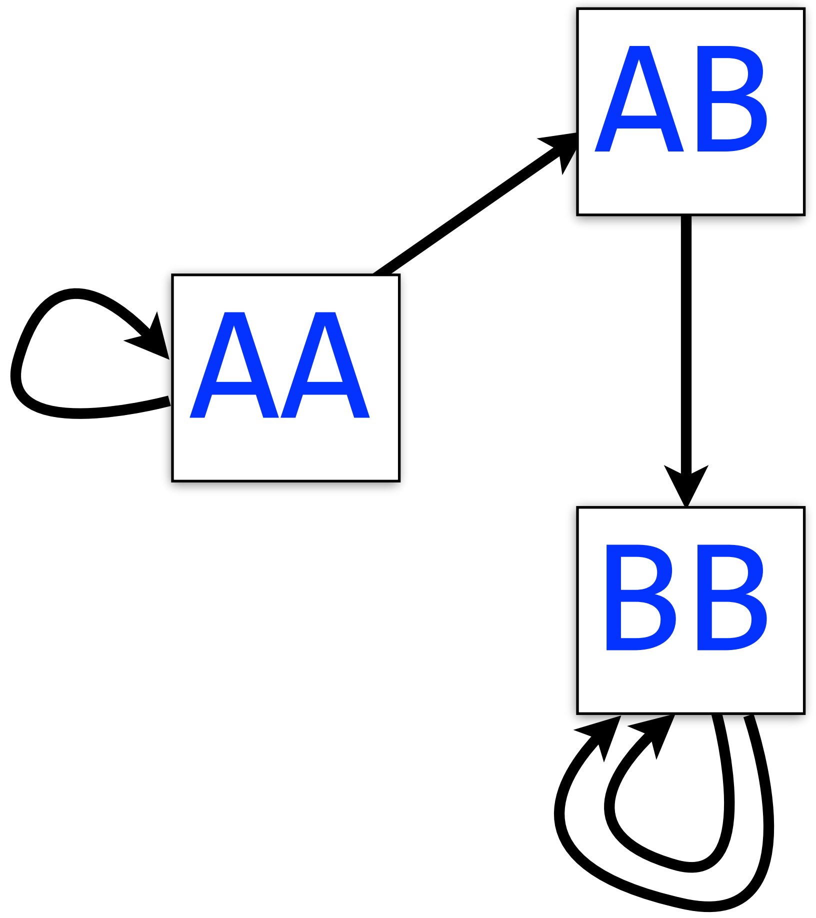


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA   AA, AB   AB, BB   BB, BB   BB, BB   BB, BA

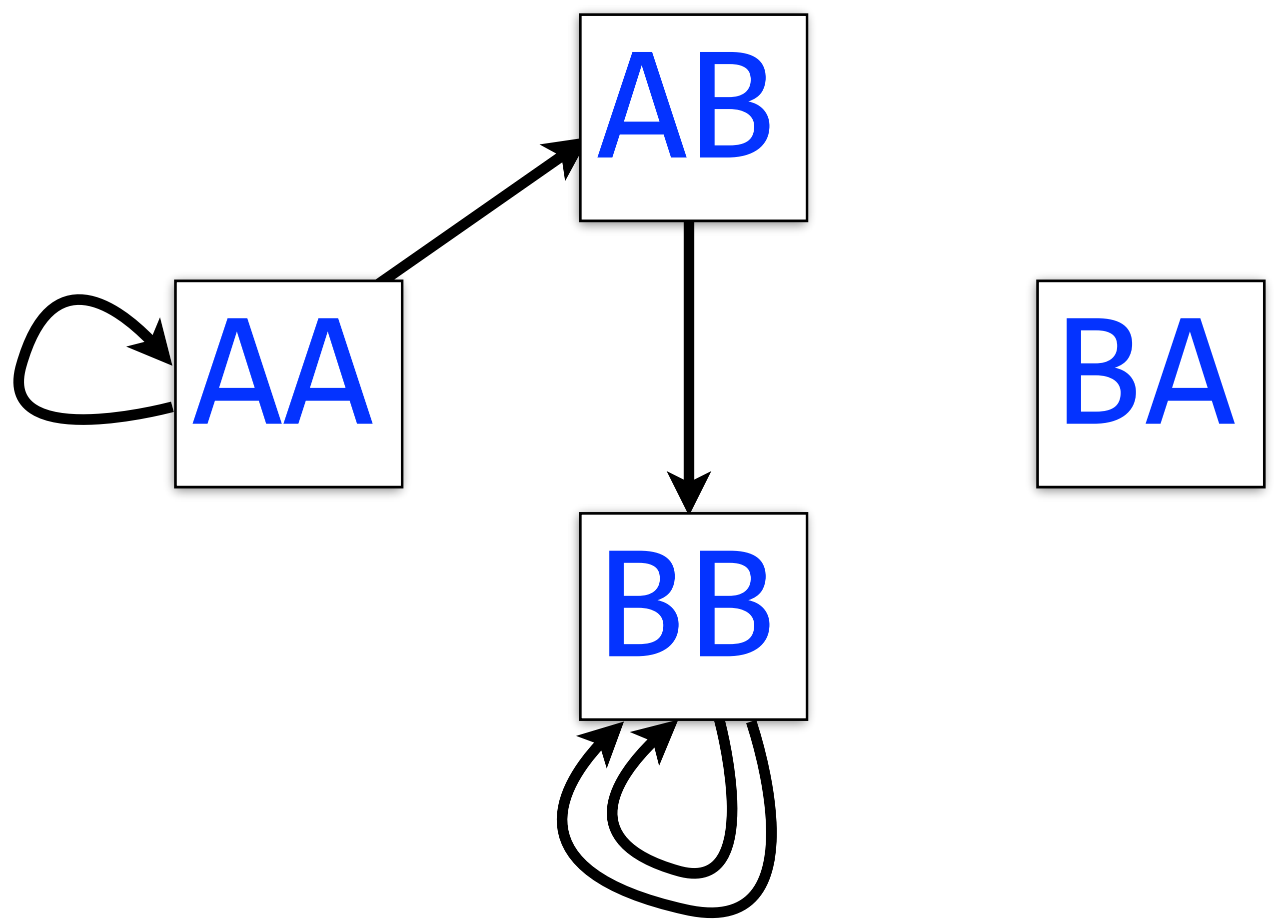


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA AA, AB AB, BB BB, BB BB, BB BB, BA

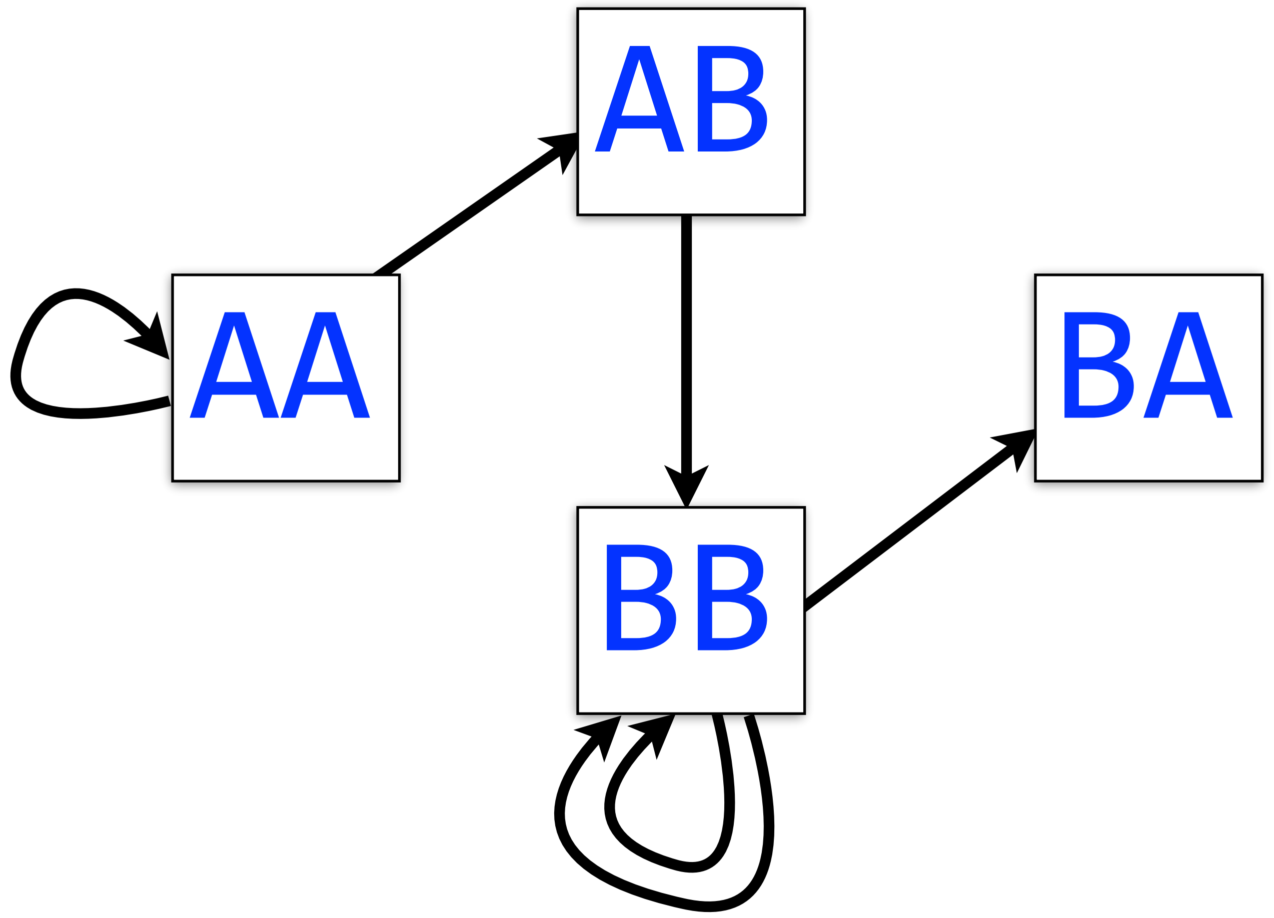


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA AA, AB AB, BB BB, BB BB, BB BB, BA

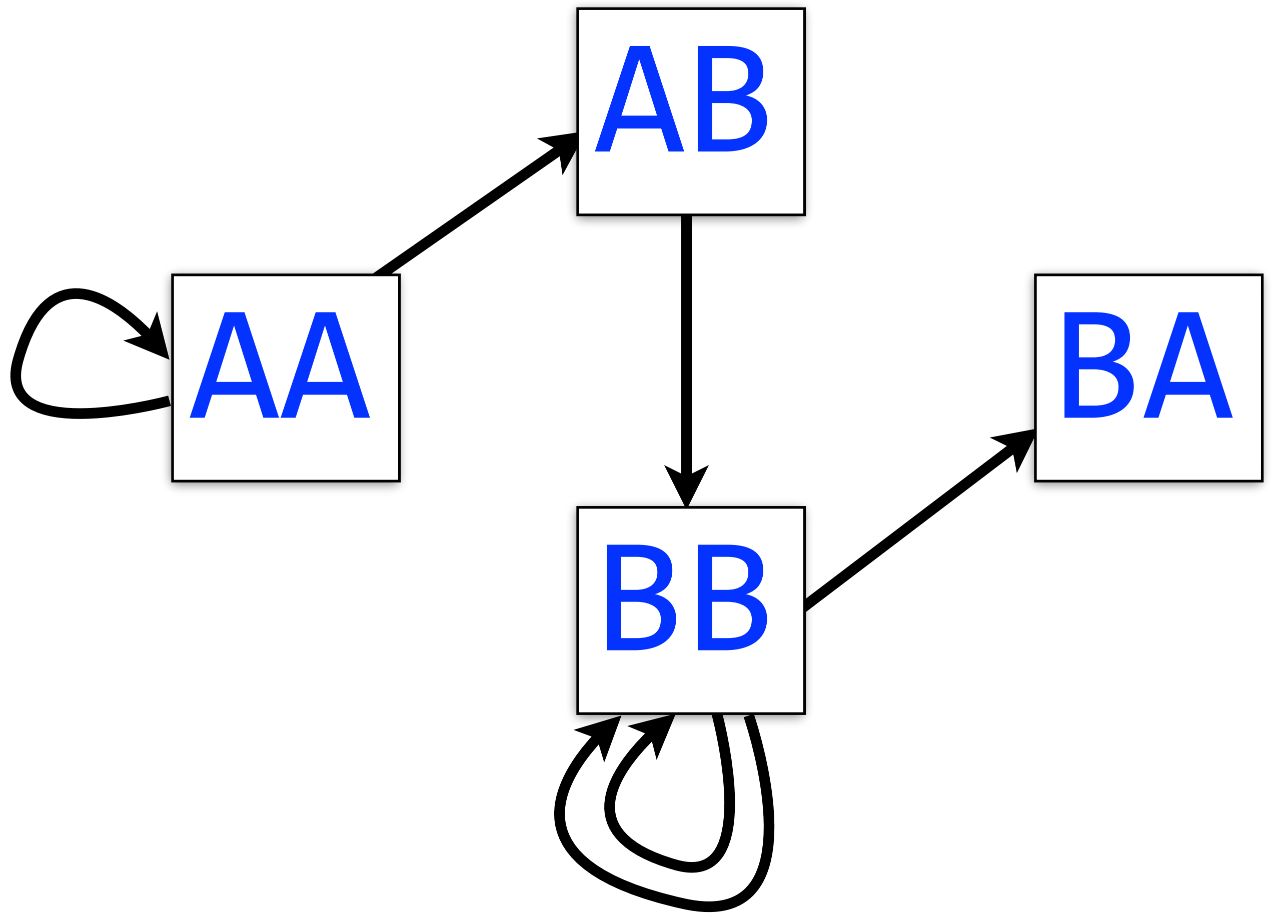


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA AA, AB AB, BB BB, BB BB, BB BB, BA



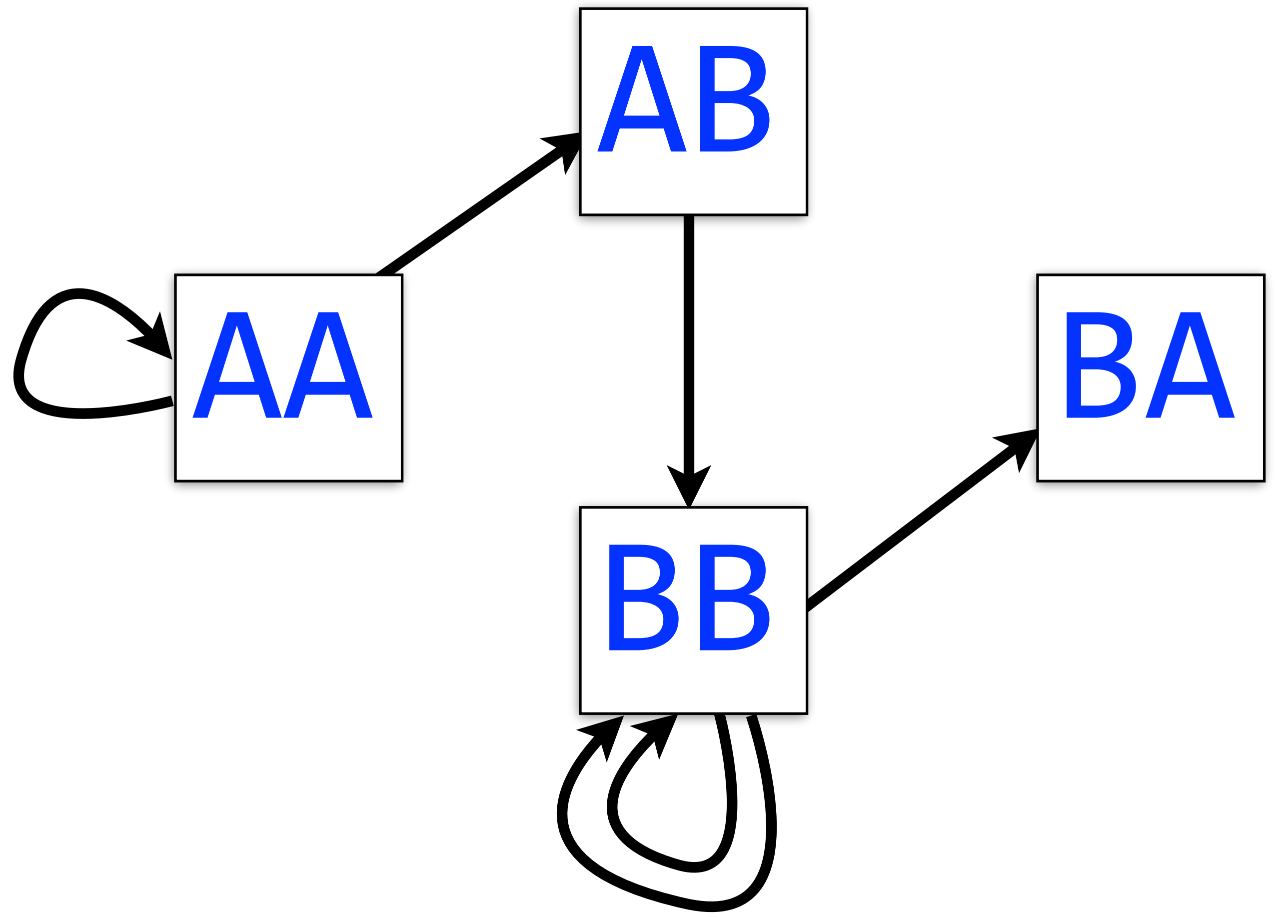
One edge per k-mer

De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

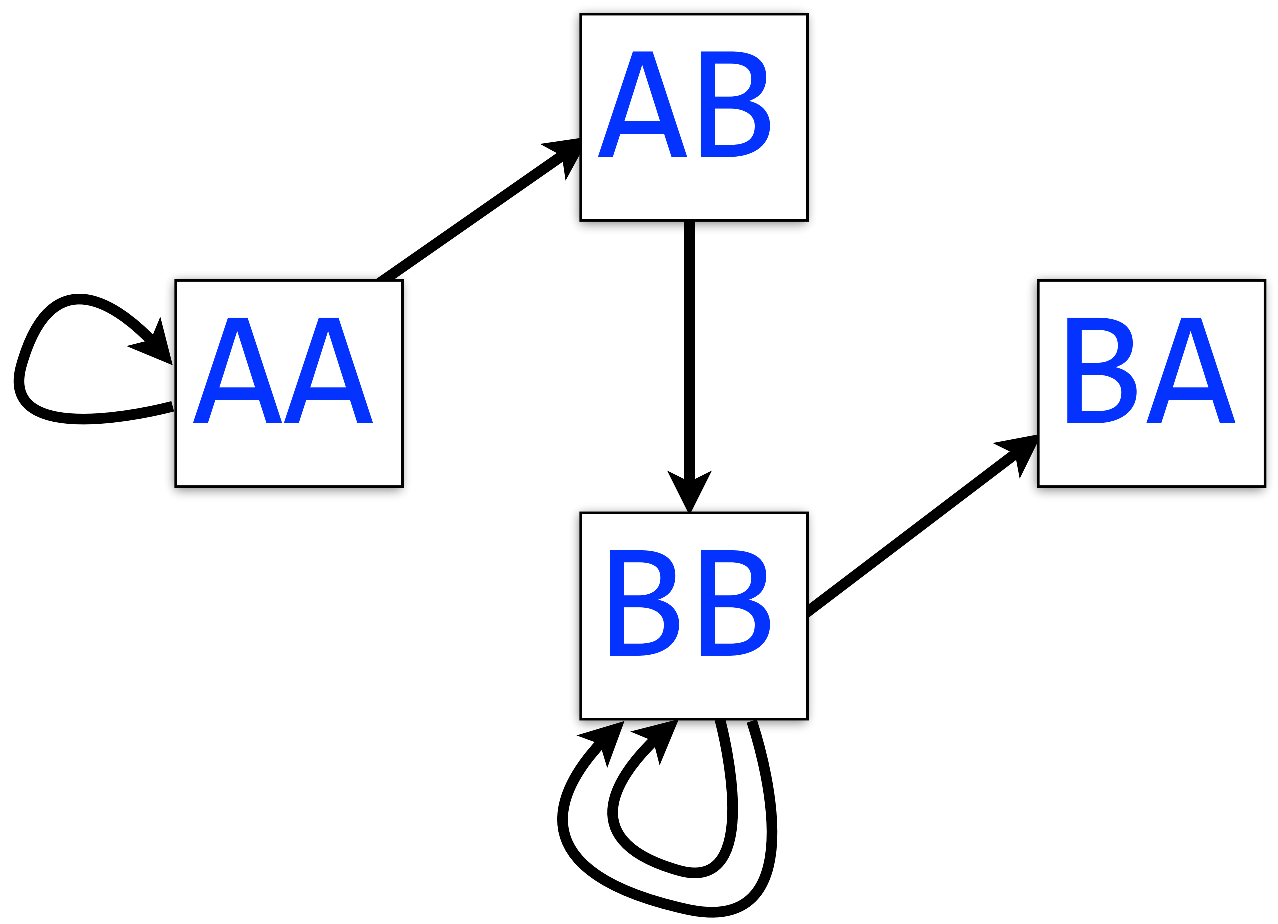
L/R 2-mers: AA, AA   AA, AB   AB, BB   BB, BB   BB, BB   BB, BA



One edge per k-mer

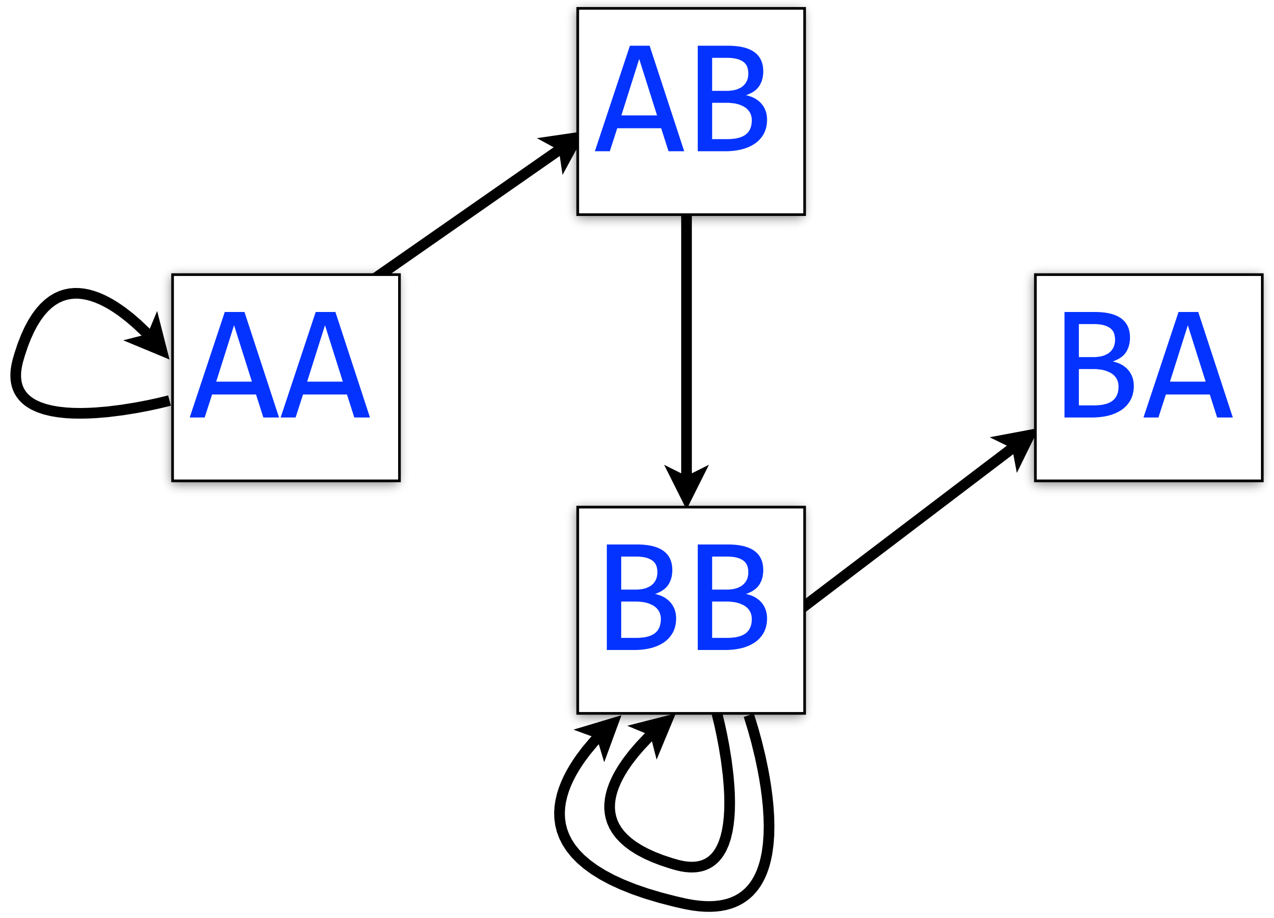
One node per distinct k-1-mer

De Bruijn graph



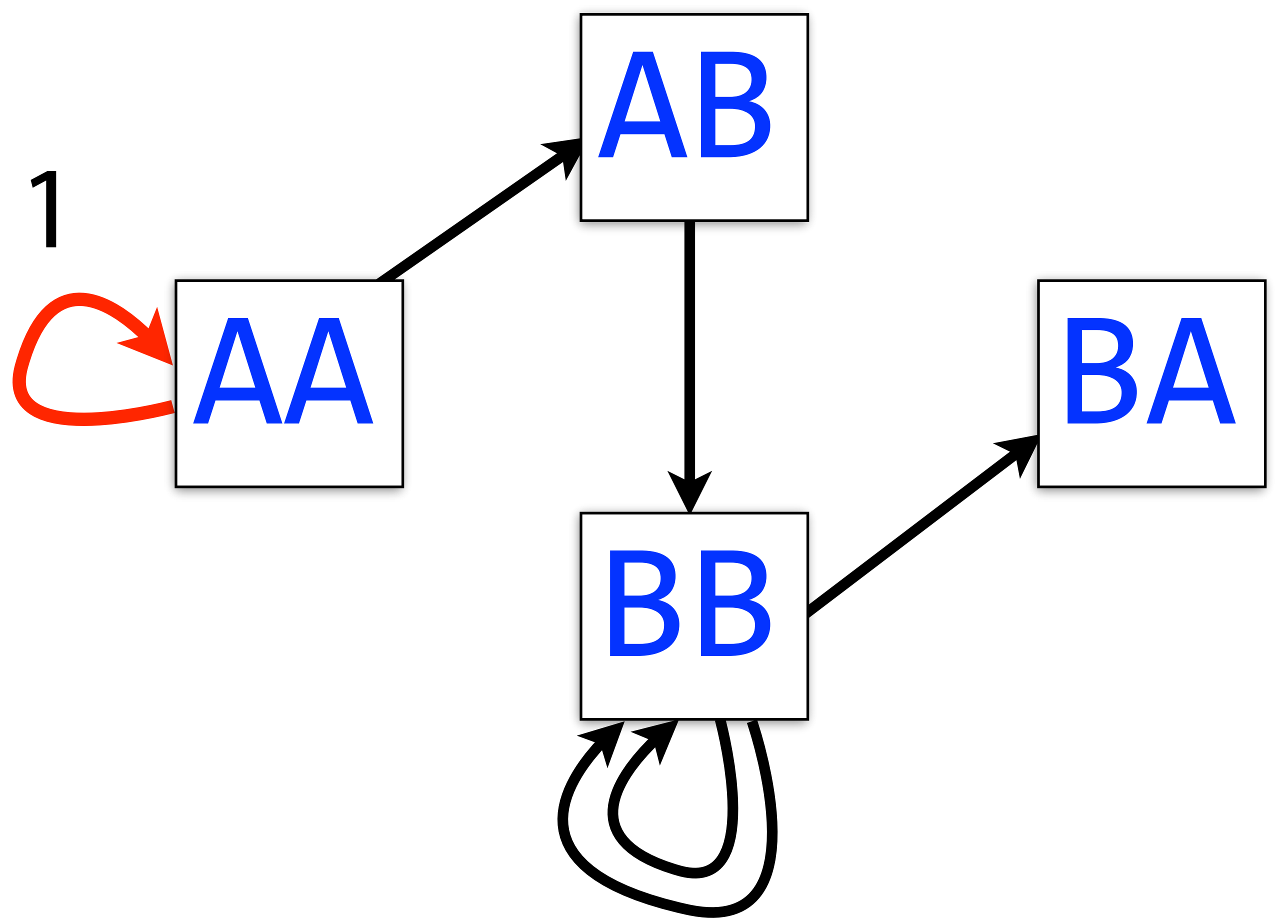
*

De Bruijn graph



Walk crossing each edge exactly once gives a reconstruction of the genome

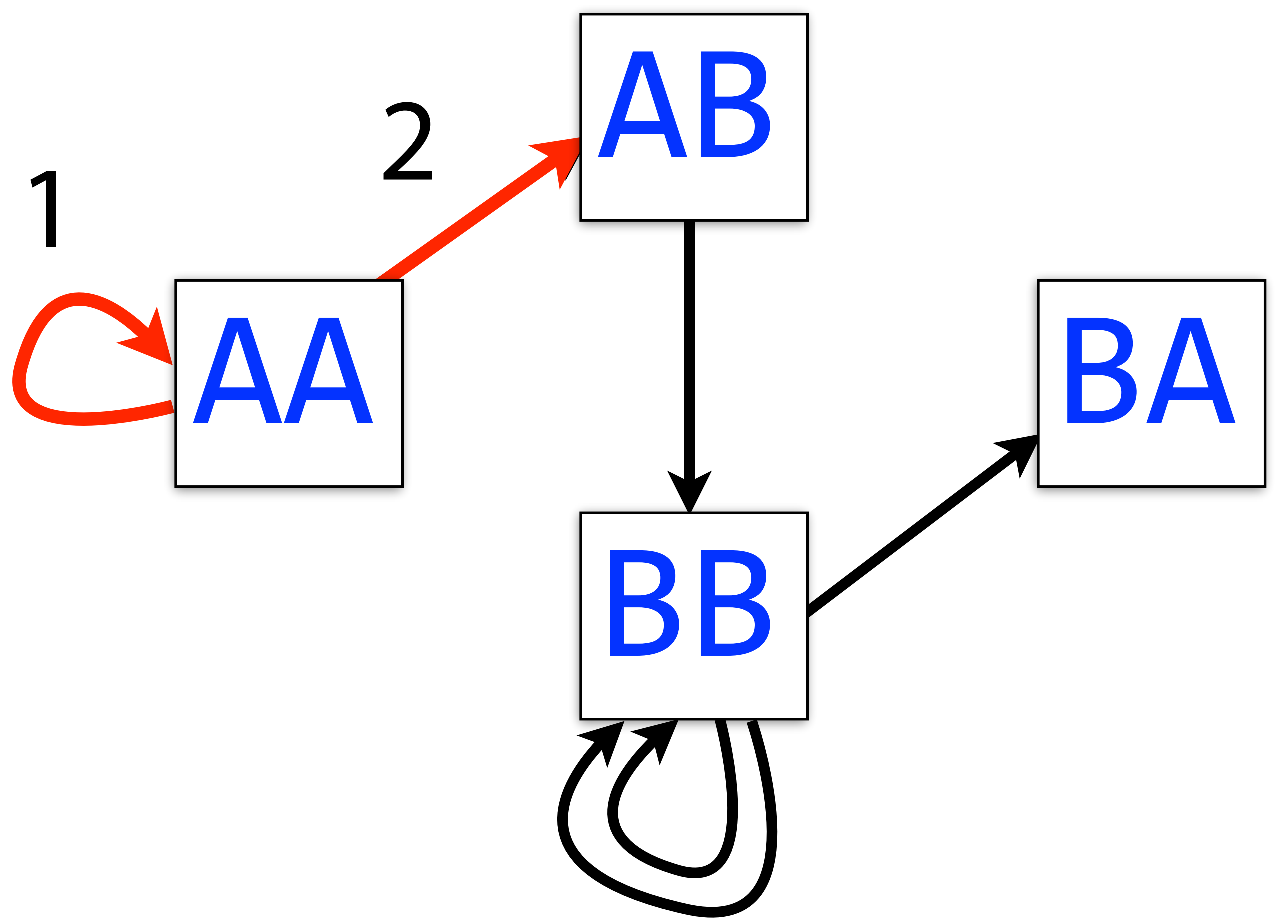
De Bruijn graph



AAA

Walk crossing each edge exactly once gives a reconstruction of the genome

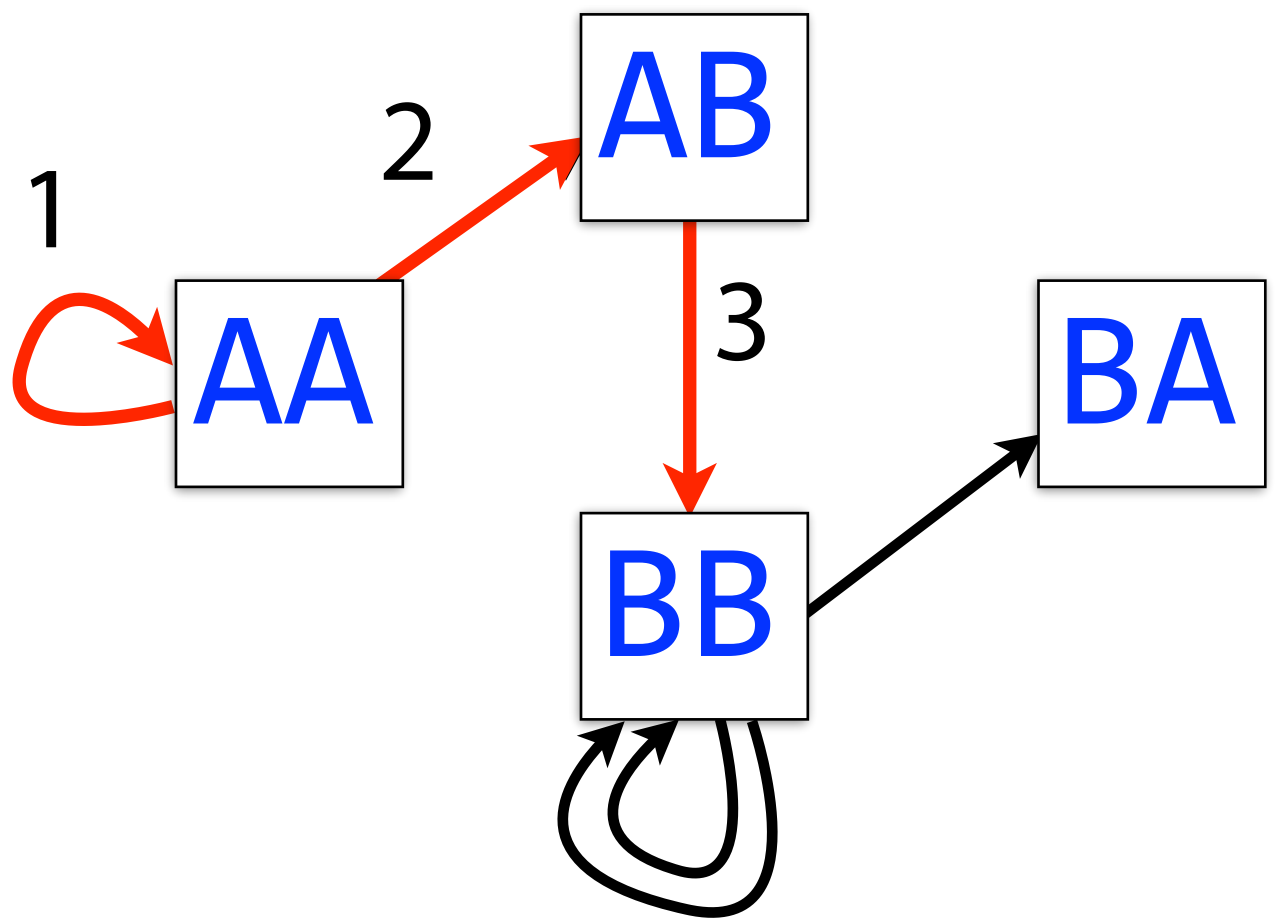
De Bruijn graph



AAA B

Walk crossing each edge exactly once gives a reconstruction of the genome

De Bruijn graph

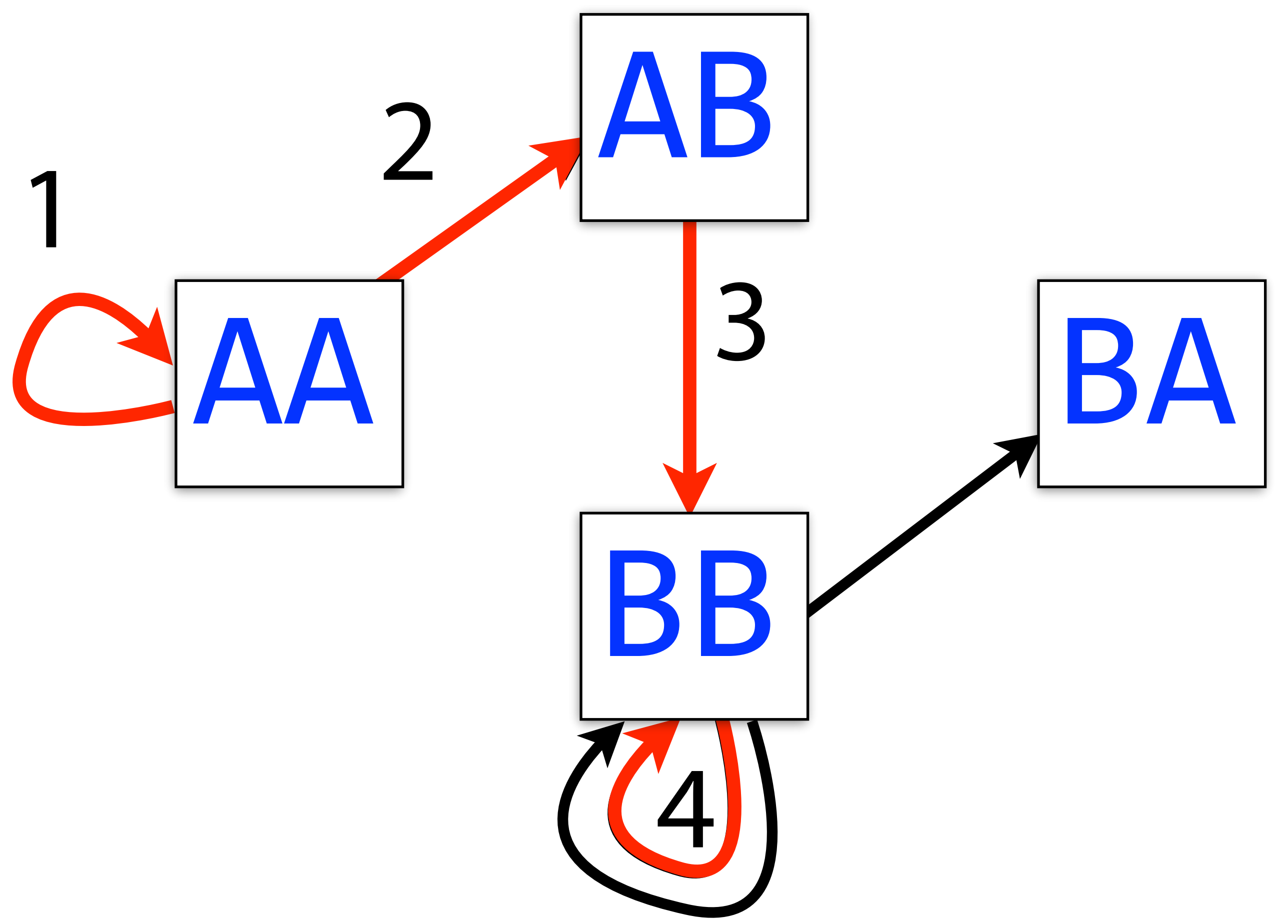


AAA BB

Walk crossing each edge exactly once gives a reconstruction of the genome

*

De Bruijn graph

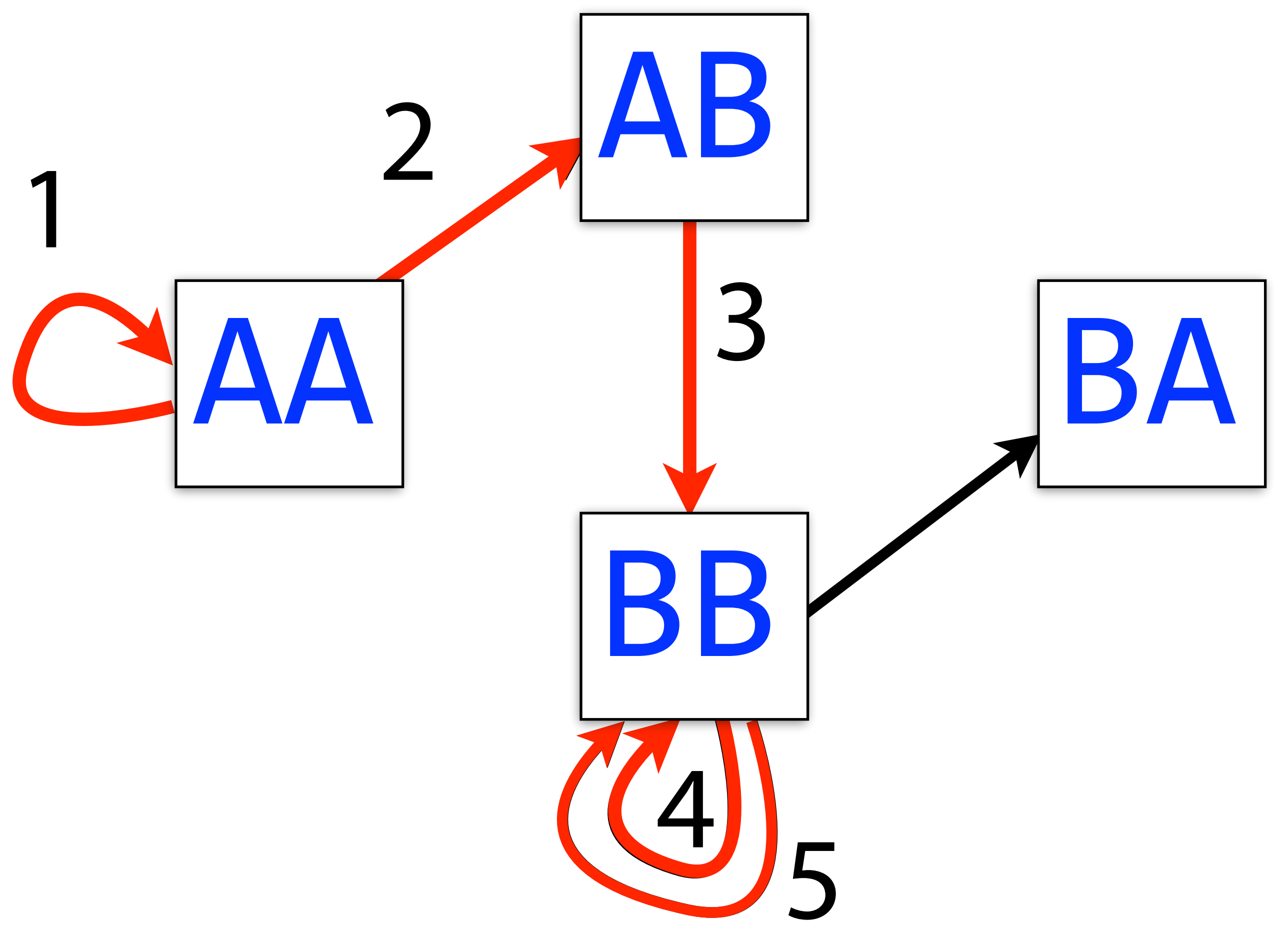


AAA BBB

Walk crossing each edge exactly once gives a reconstruction of the genome

*

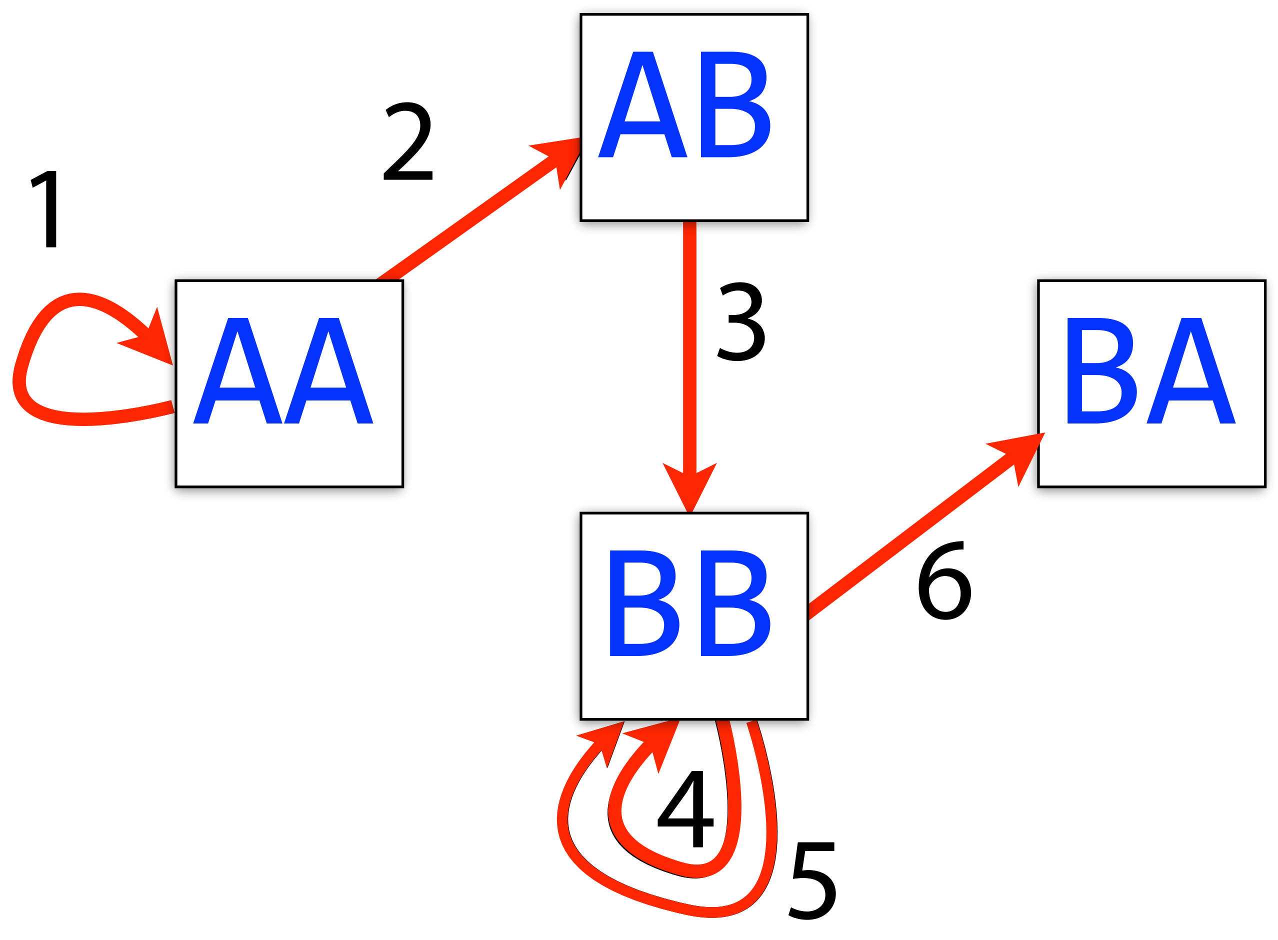
De Bruijn graph



AAA BBBB

Walk crossing each edge exactly once gives a reconstruction of the genome

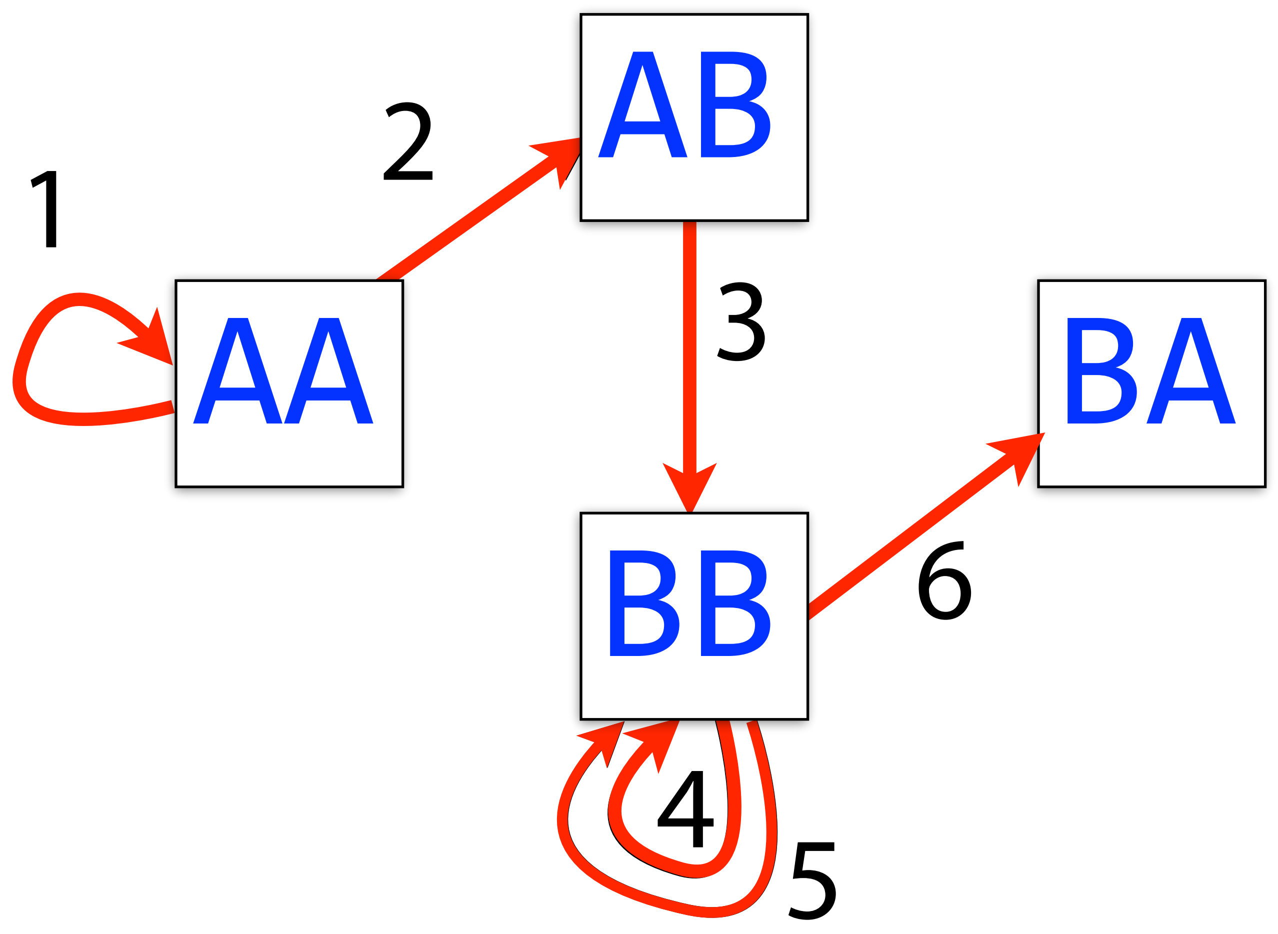
De Bruijn graph



AAA BBBBA

Walk crossing each edge exactly once gives a reconstruction of the genome

De Bruijn graph



AAA BBBBA

Walk crossing each edge exactly once gives a reconstruction of the genome . This is an Eulerian walk.

De Bruijn graph

Aside: how do you pronounce "De Bruijn"?

There is debate:

https://www.biostars.org/p/7186/

Nicolaas Govert
de Bruijn
1918 -- 2012

Directed multigraph

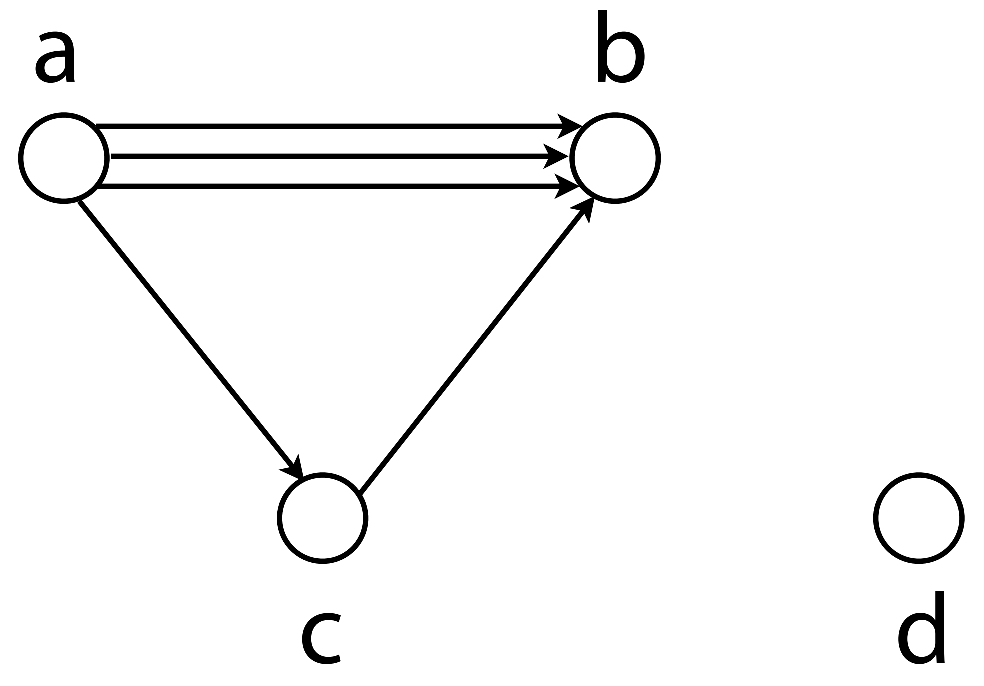
Directed multigraph G(V, E) consists of set of vertices, V and multiset of directed edges, E

Otherwise, like a directed graph

Node's indegree = # incoming edges

Node's outdegree = # outgoing edges

De Bruijn graph is a directed multigraph



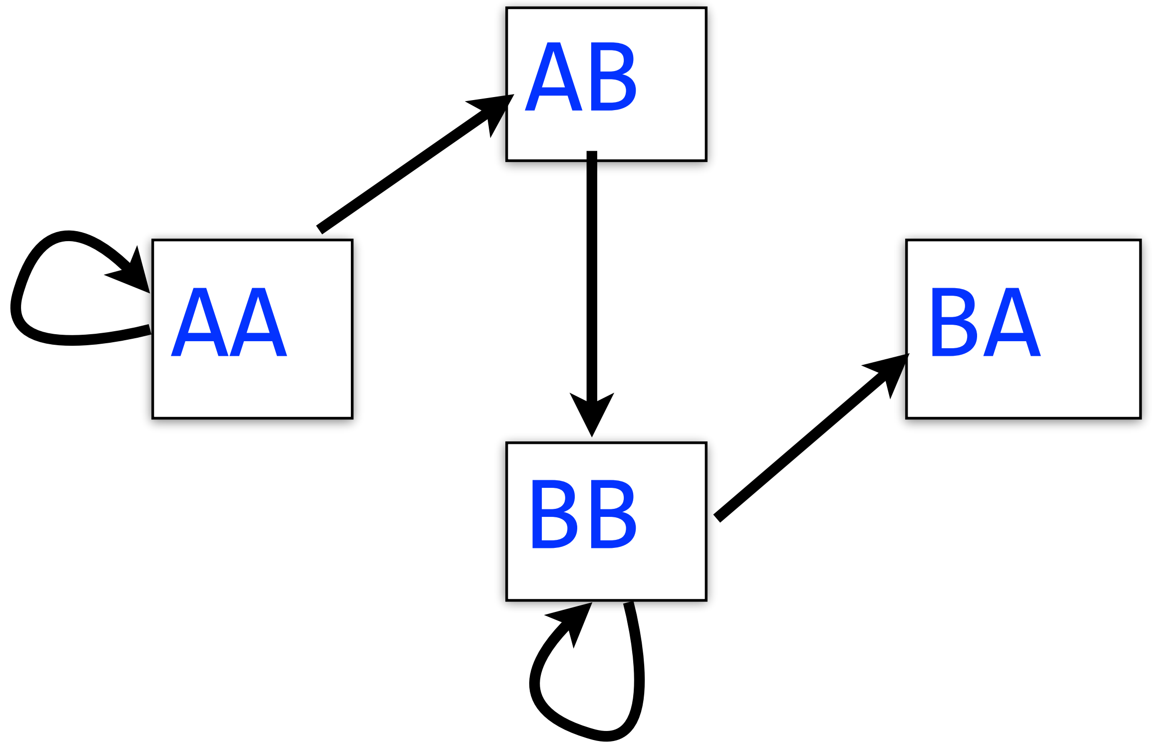
V = { a, b, c, d }

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

Repeated

Eulerian walk definitions and statements

Node is balanced if indegree equals outdegree

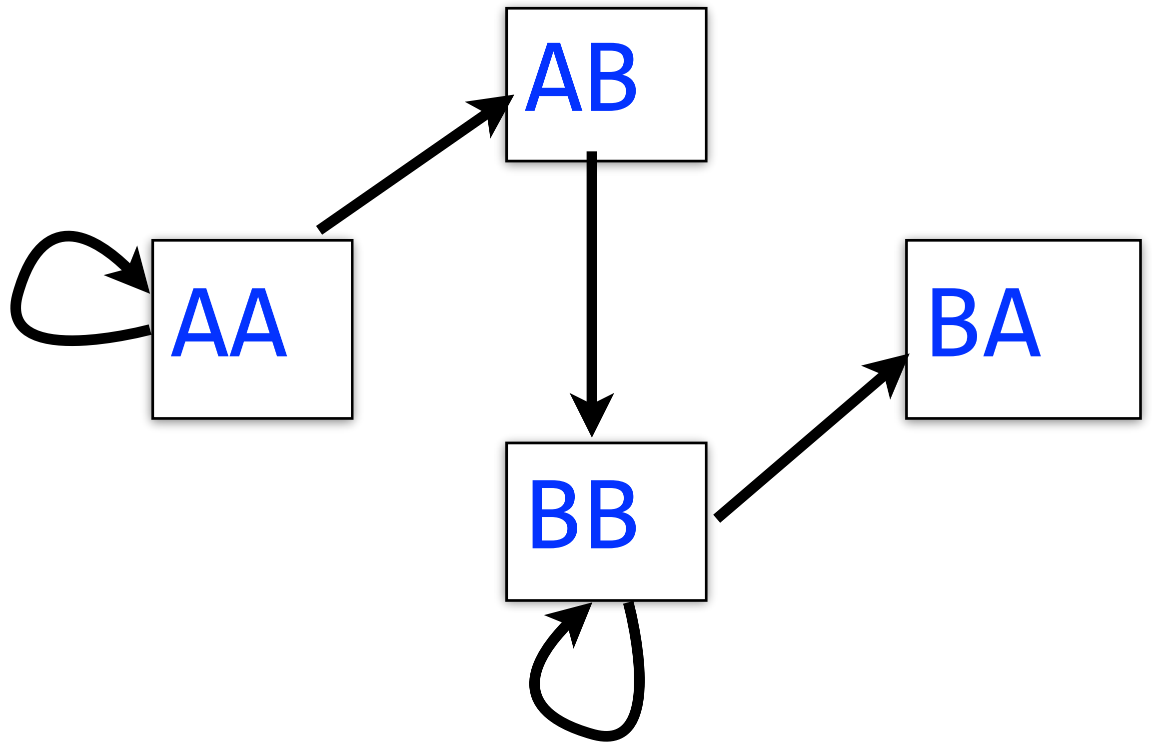


*

Eulerian walk definitions and statements

Node is balanced if indegree equals outdegree

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



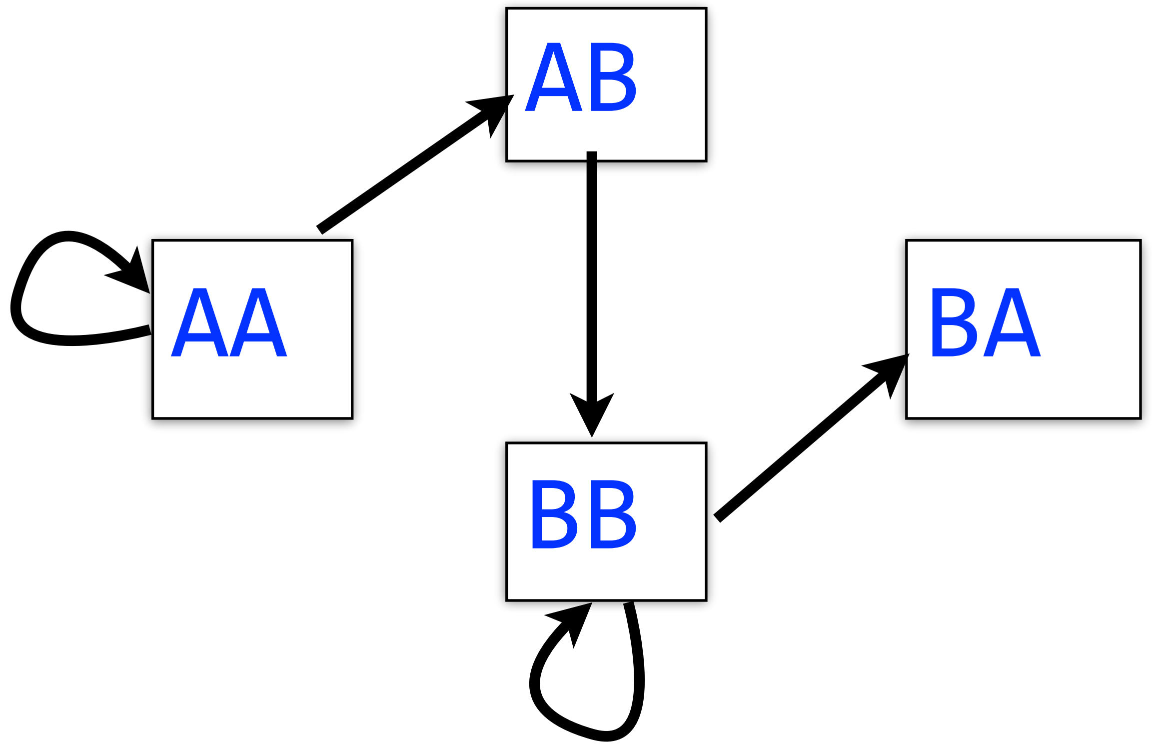
*

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



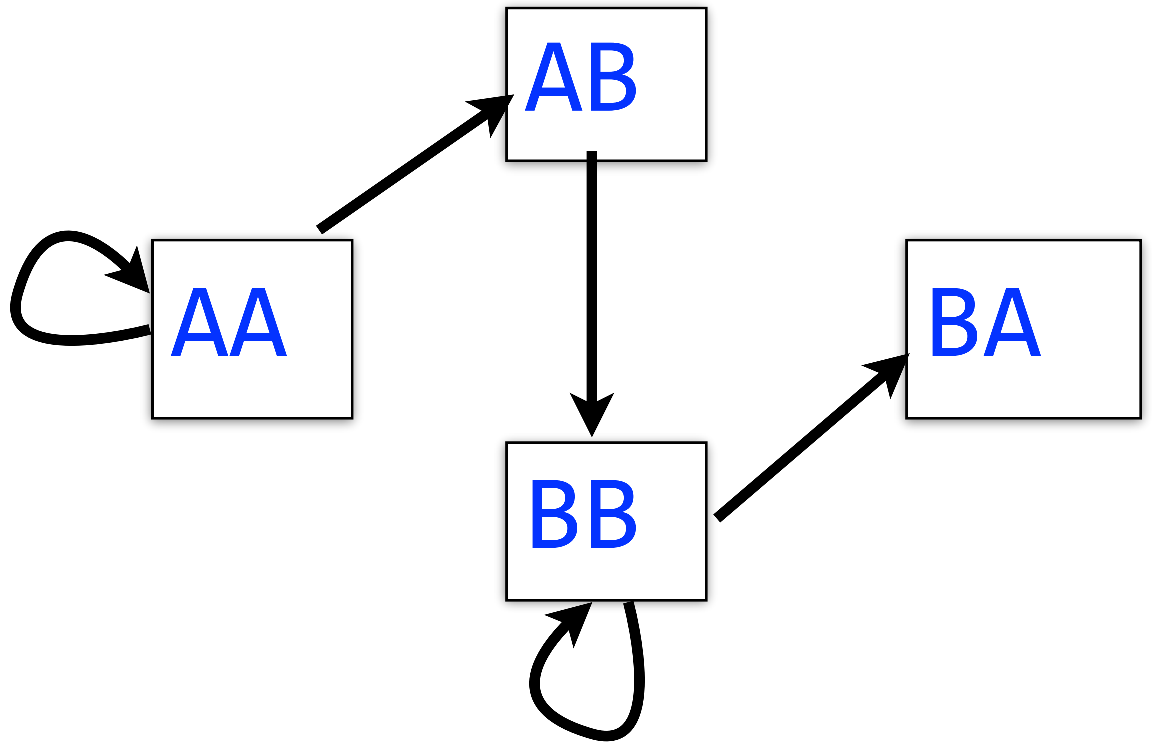
Eulerian walk definitions and statements

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



*

Eulerian walk definitions and statements

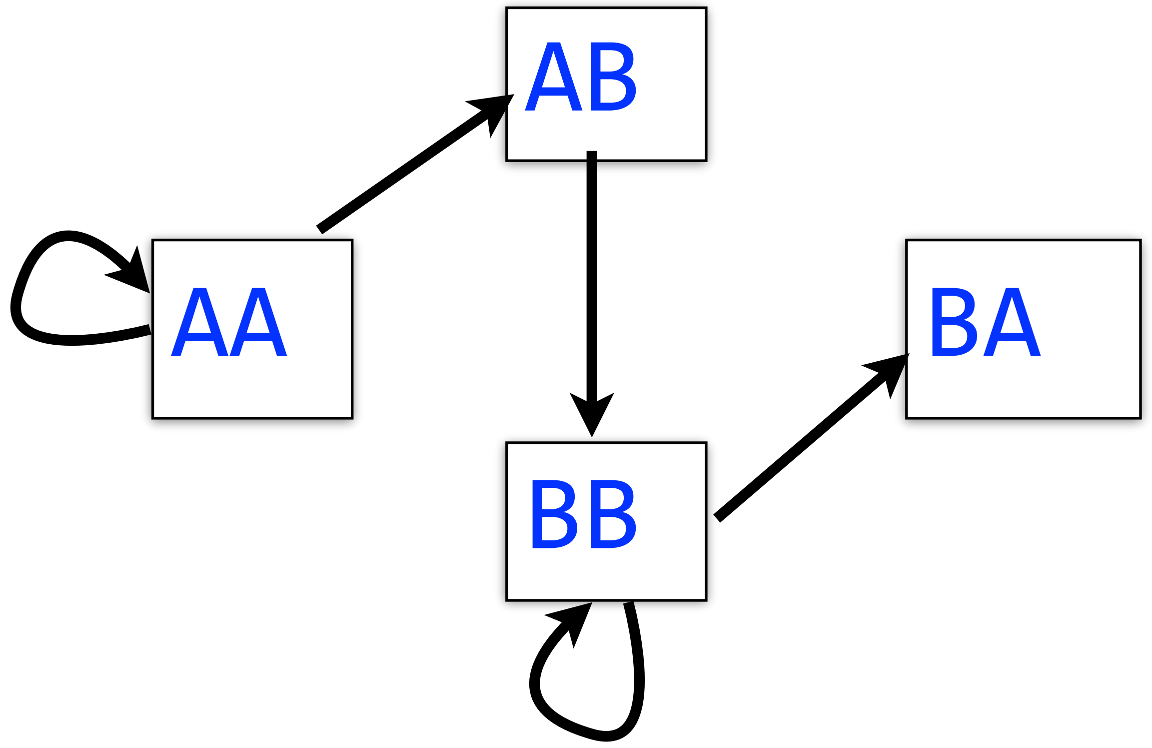
Node is balanced if indegree equals outdegree

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. (For simplicity, we won't distinguish Eulerian from semi-Eulerian.)



Eulerian walk definitions and statements

Node is balanced if indegree equals outdegree

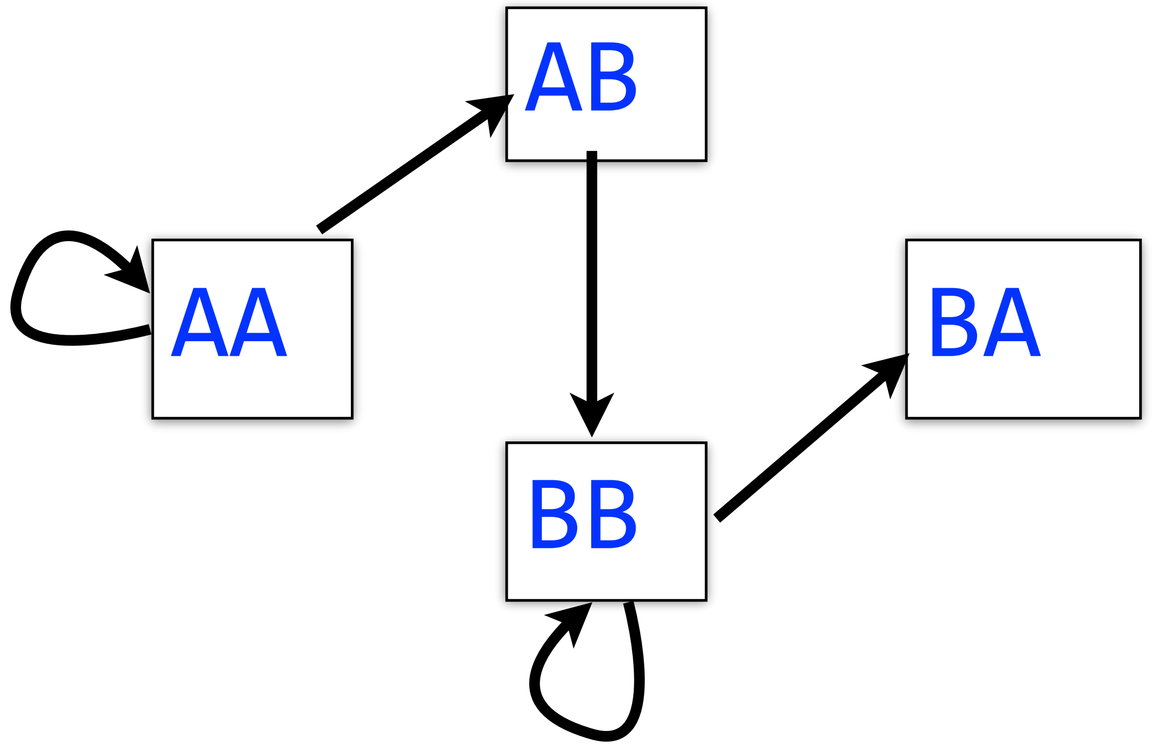
Node is semi-balanced if indegree differs from outdegree by 1

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. (For simplicity, we won't distinguish Eulerian from semi-Eulerian.)

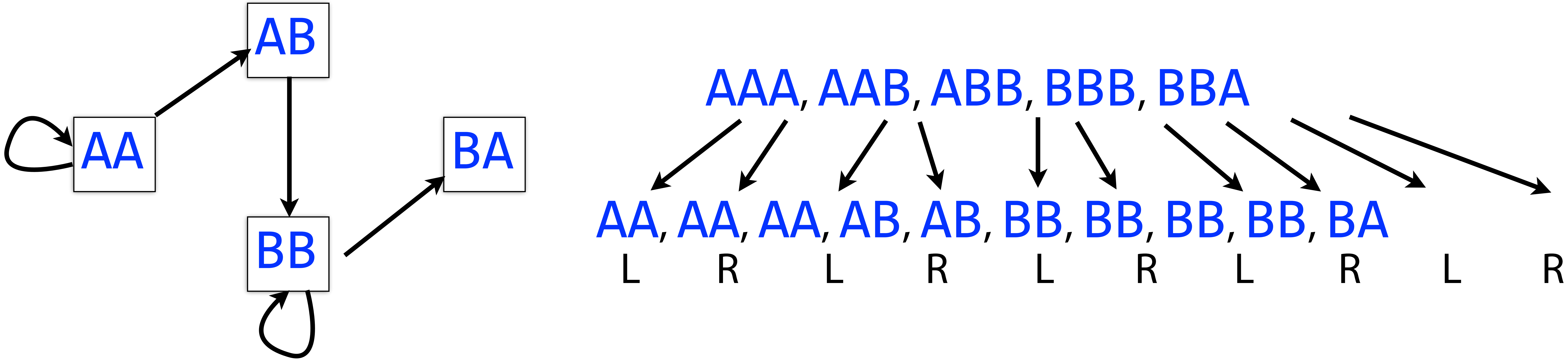
A directed, connected graph is Eulerian if and only if it has at most 2 semi-balanced nodes and all other nodes are balanced



Jones and Pevzner section 8.8

De Bruijn graph

Back to de Bruijn graph



AAA, AAB, ABB, BBB, BBA

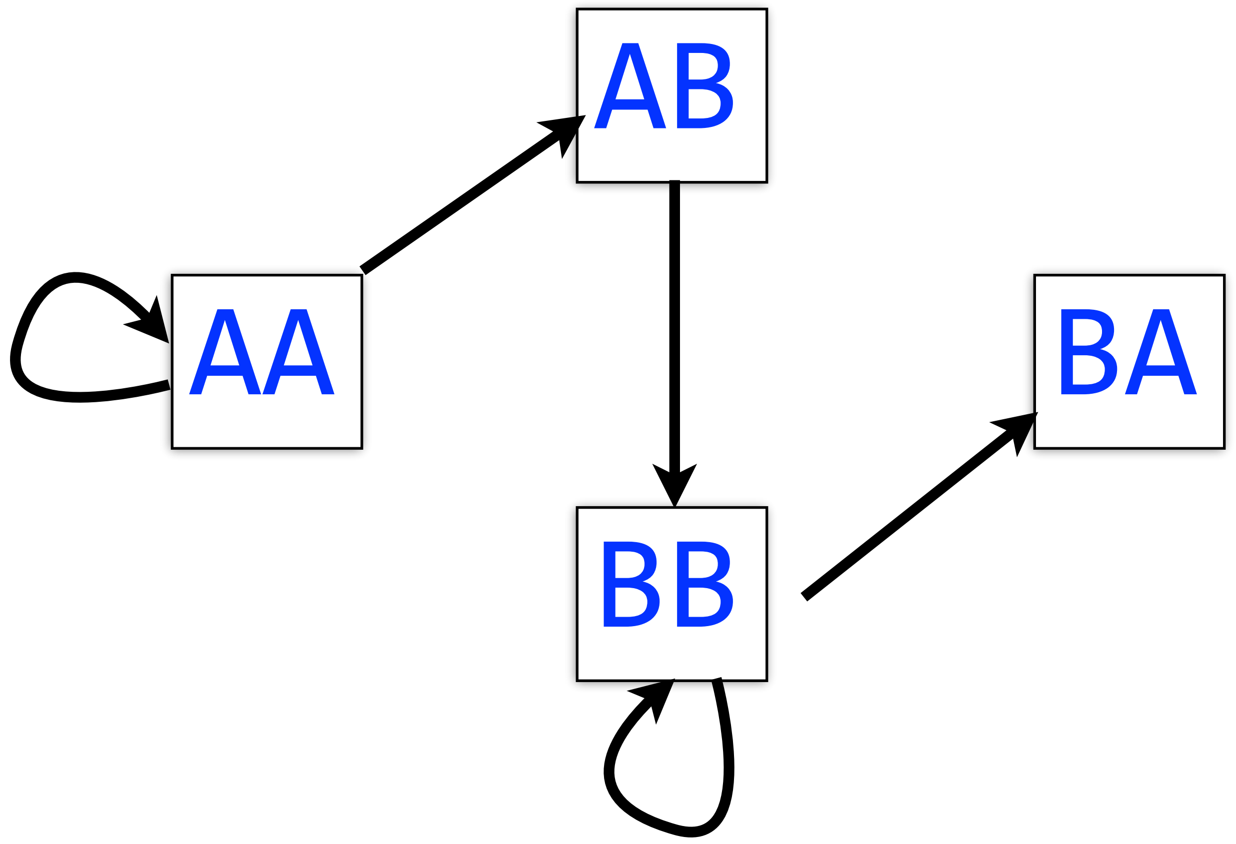
AA, AA, AA, AB, AB, BB, BB, BB, BB, BA

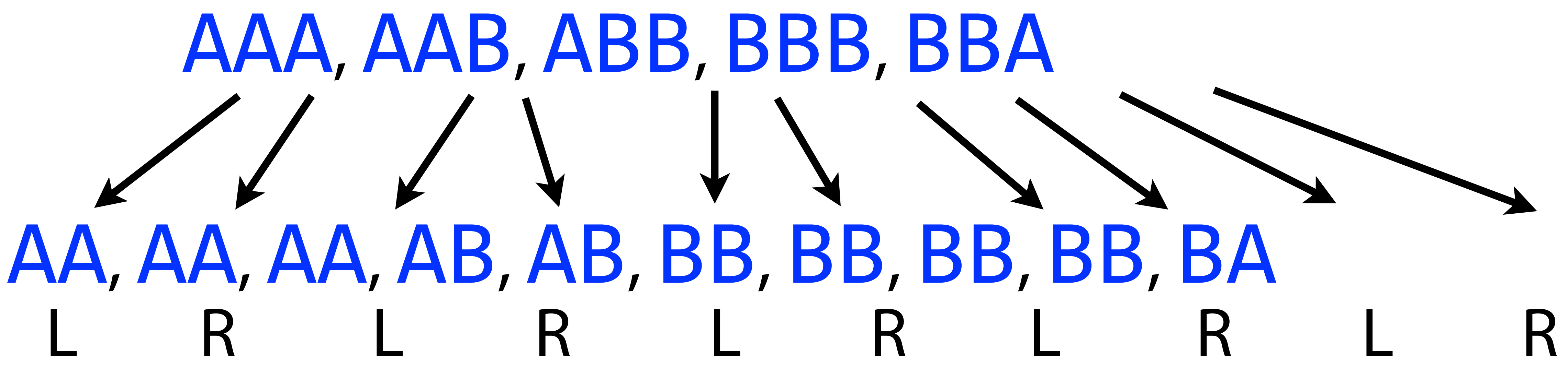
L R L R L R L R L R

Is it Eulerian?

De Bruijn graph

Back to de Bruijn graph




AAA, AAB, ABB, BBB, BBA

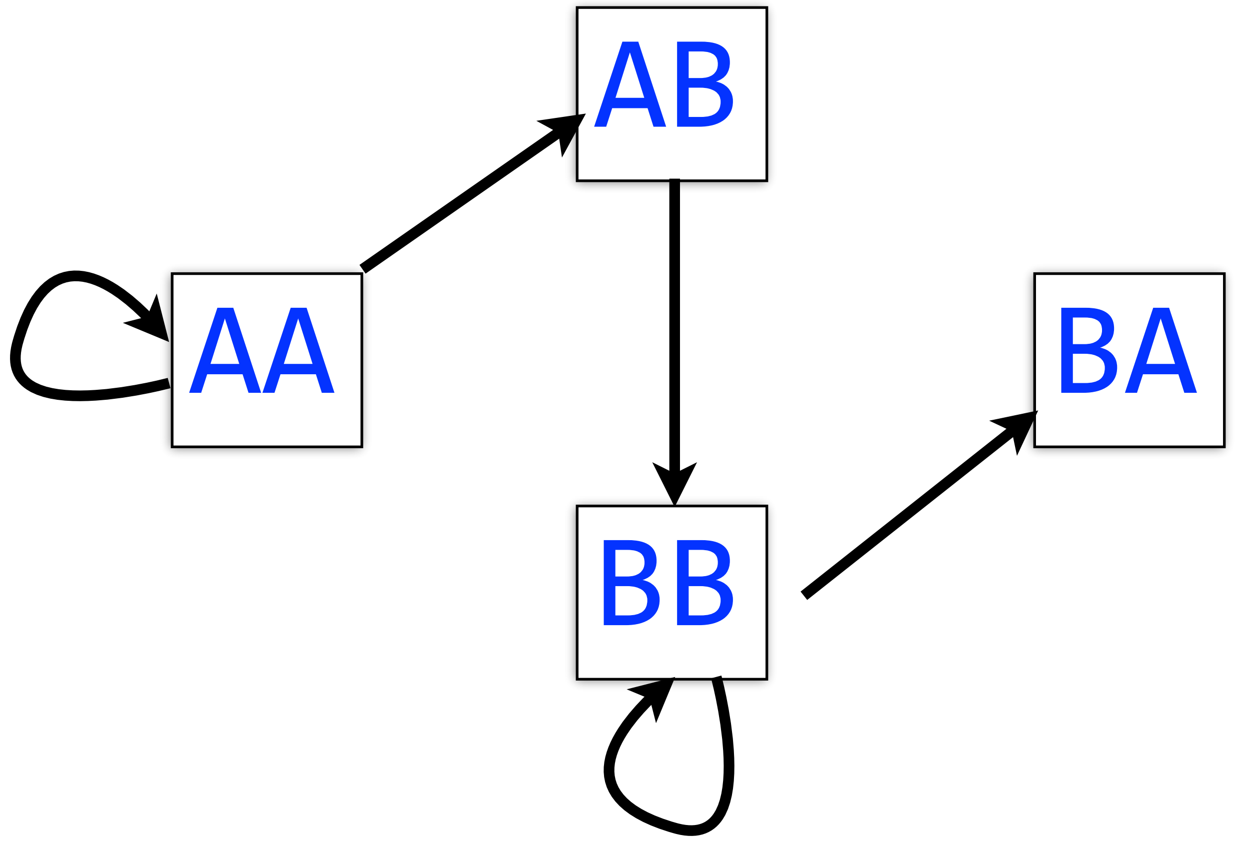
AA, AA, AA, AB, AB, BB, BB, BB, BB, BA

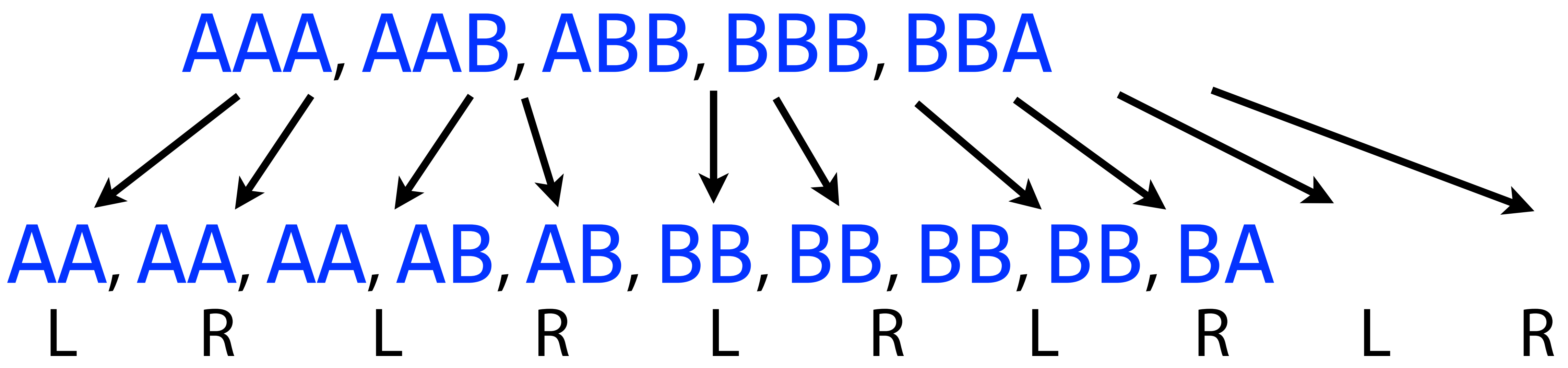
L R L R L R L R L R

Is it Eulerian? Yes

De Bruijn graph

Back to de Bruijn graph





AAA, AAB, ABB, BBB, BBA

AA, AA, AA, AB, AB, BB, BB, BB, BB, BA

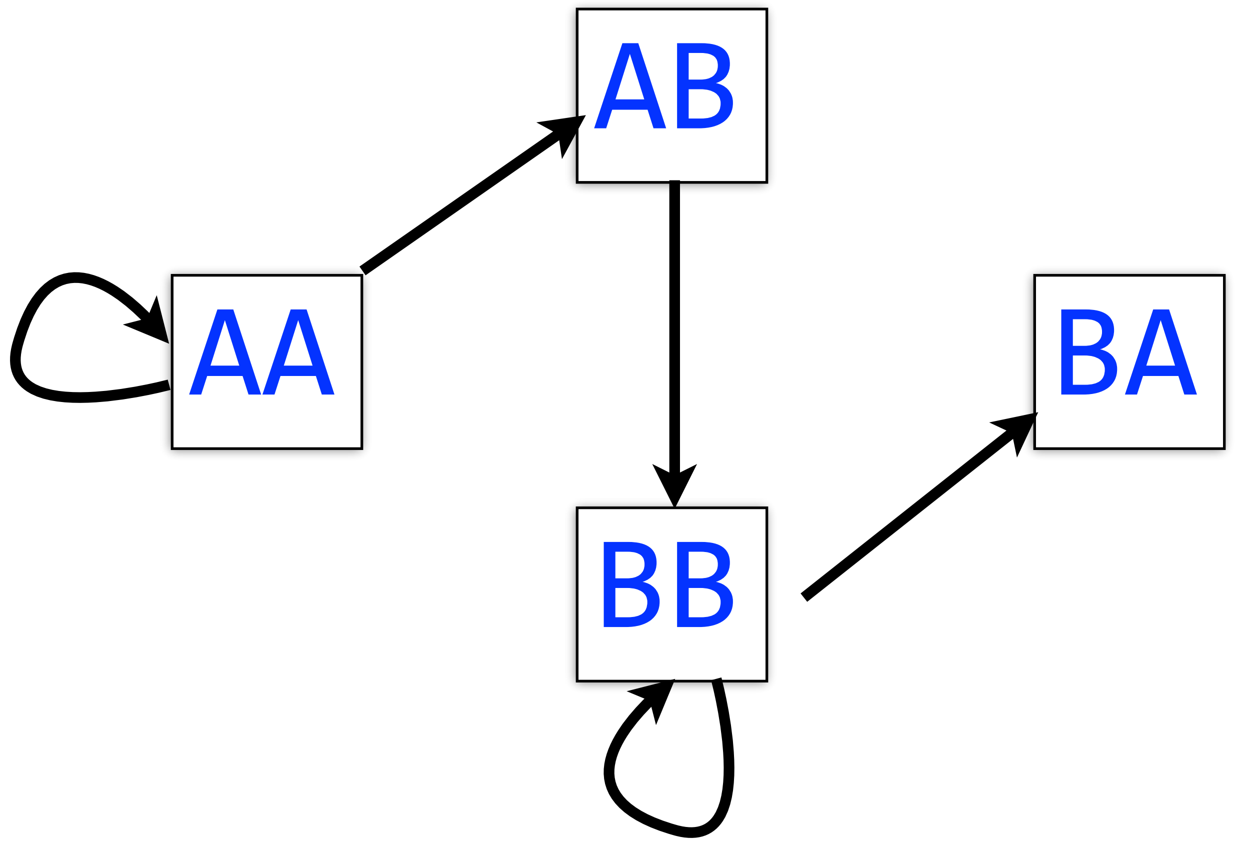
L R L R L R L R L R

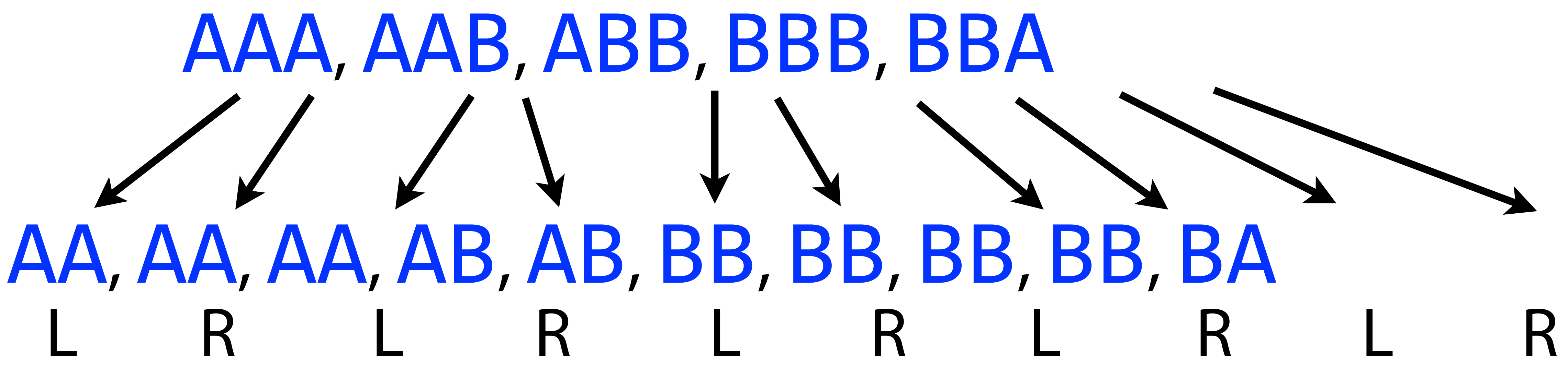
Is it Eulerian? Yes

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

De Bruijn graph

Back to de Bruijn graph





AAA, AAB, ABB, BBB, BBA

AA, AA, AA, AB, AB, BB, BB, BB, BB, BA

L R L R L R L R L R

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

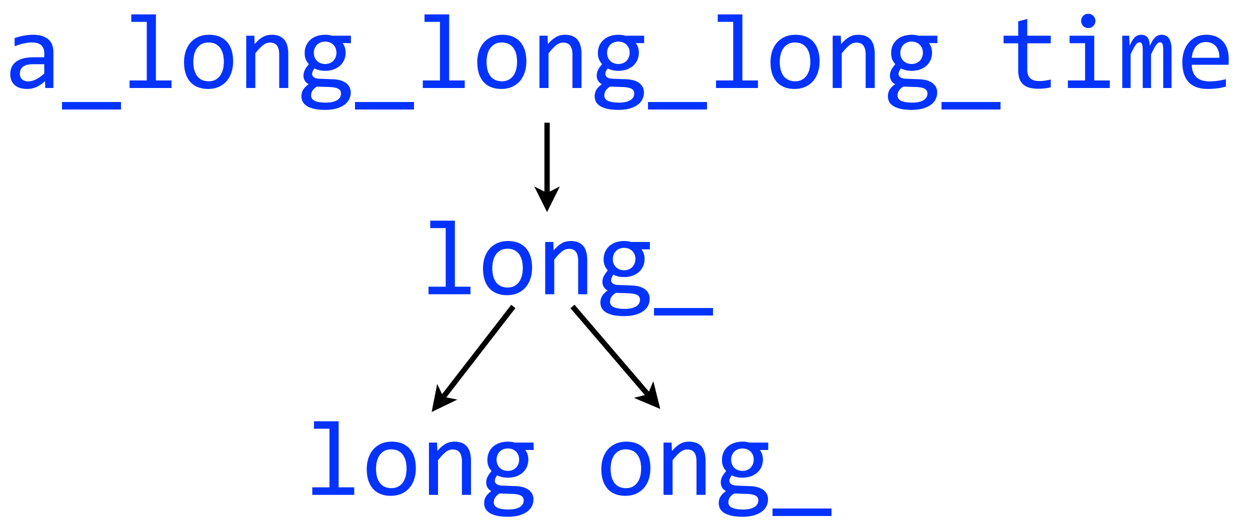
De Bruijn graph

A procedure for making a de Bruijn graph
for a genome

Assume "perfect sequencing": each genome k-mer is sequenced exactly once with no errors

Pick a substring length k: 5

Start with an input string:



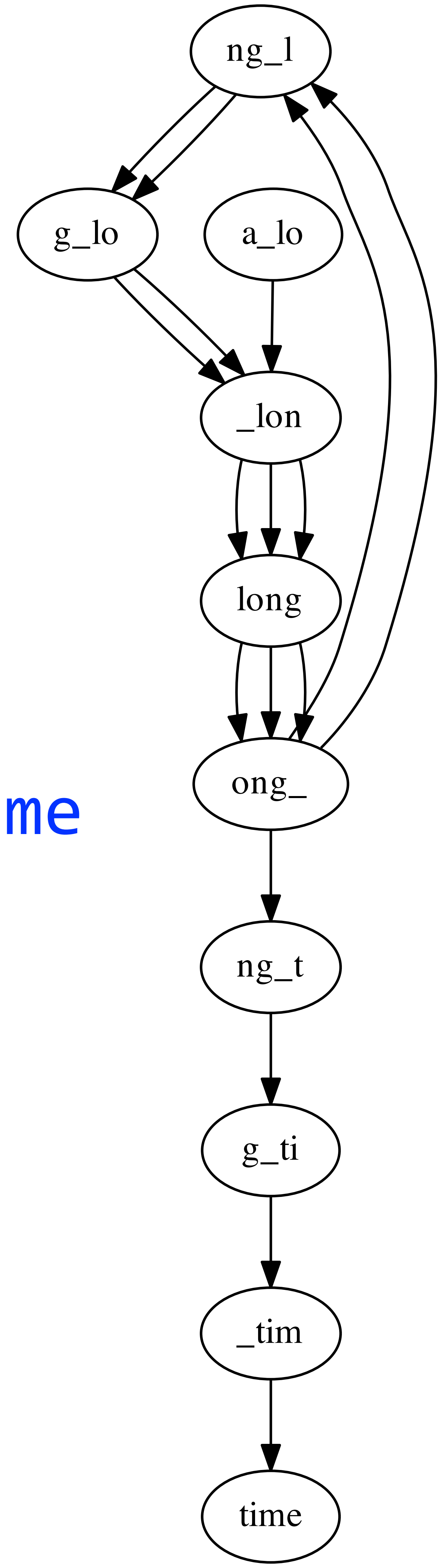
a_long_long_long_time

long_

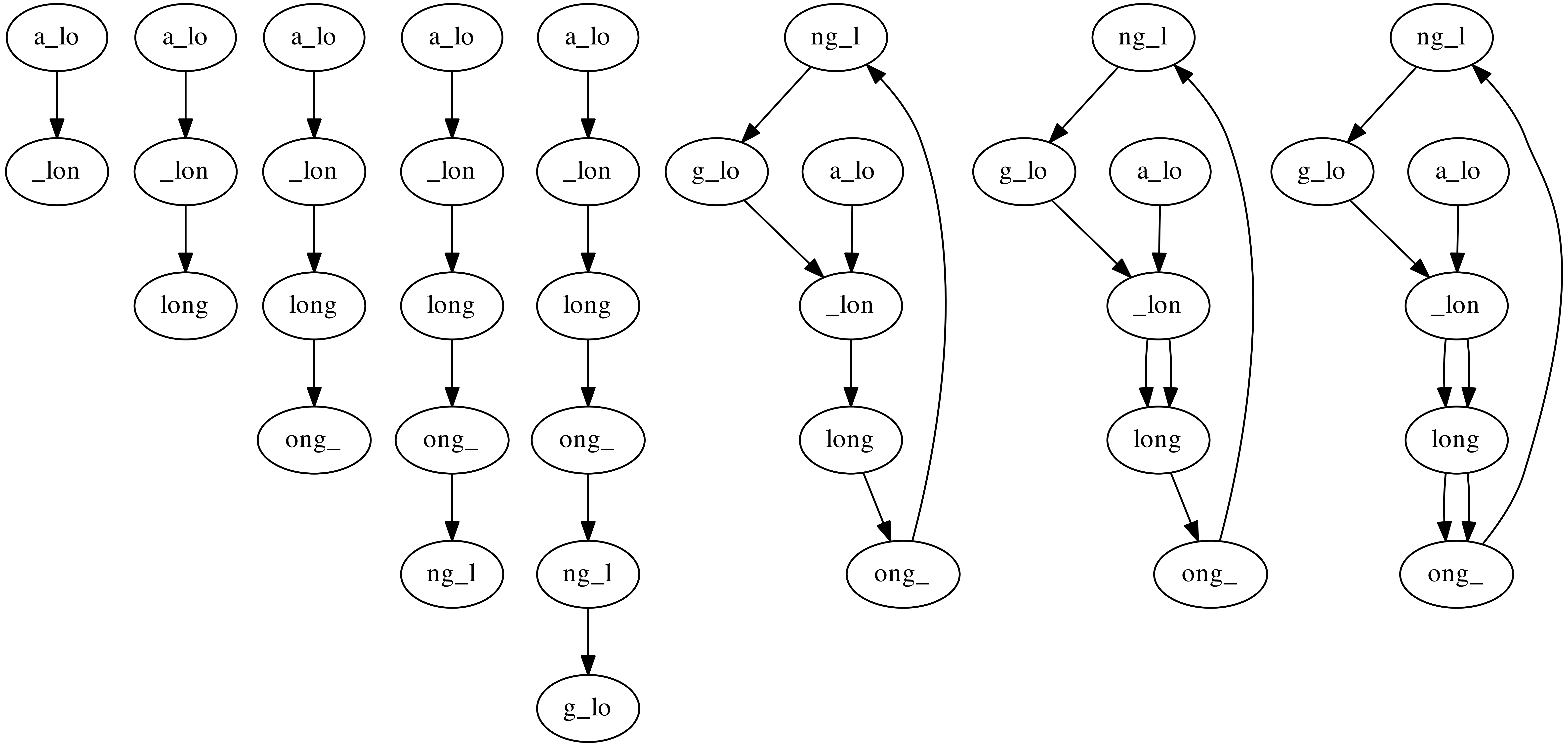
long ong_

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

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



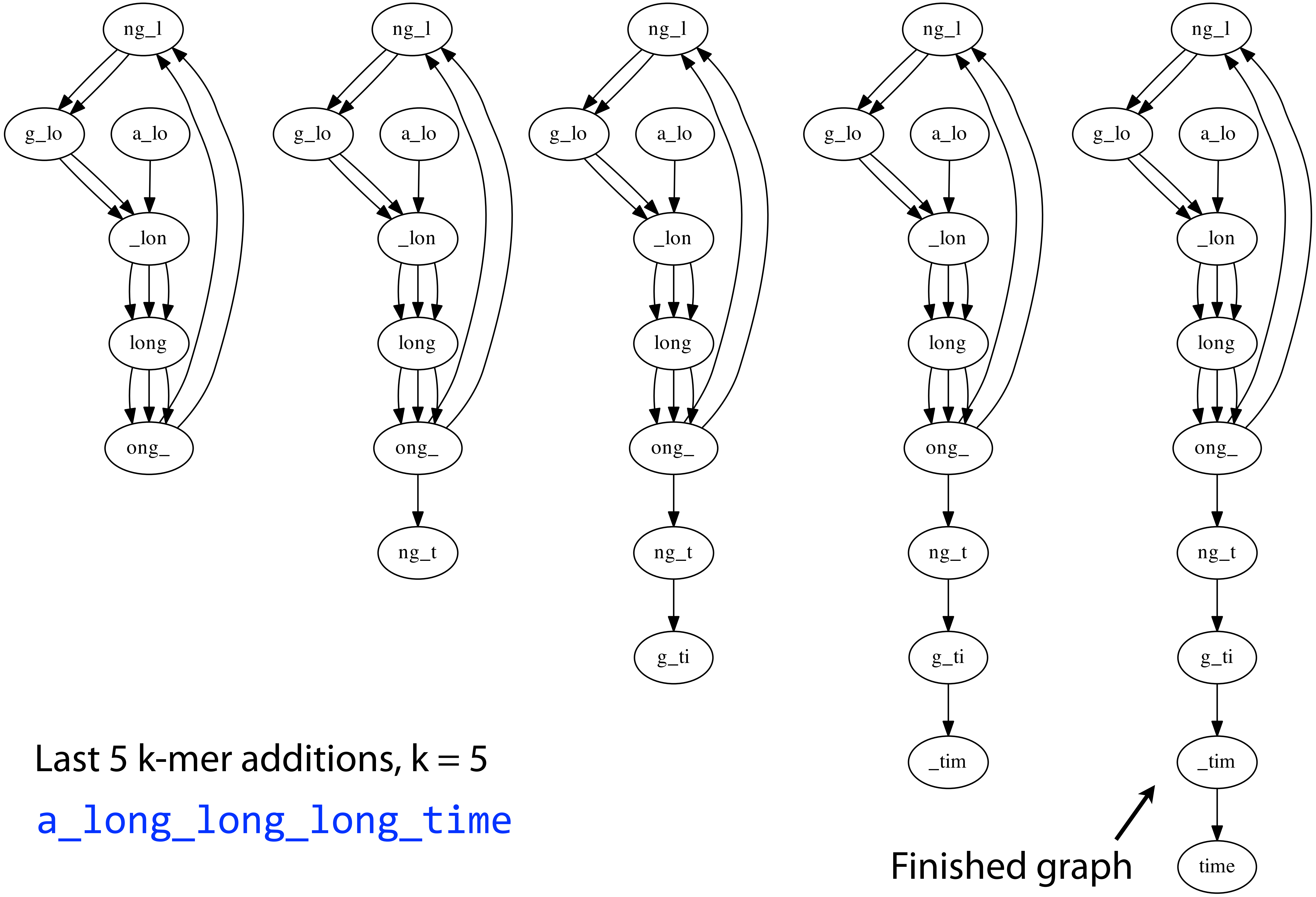
De Bruijn graph



First 8 k-mer additions, k = 5

`a_long_long_long_time`

De Bruijn graph



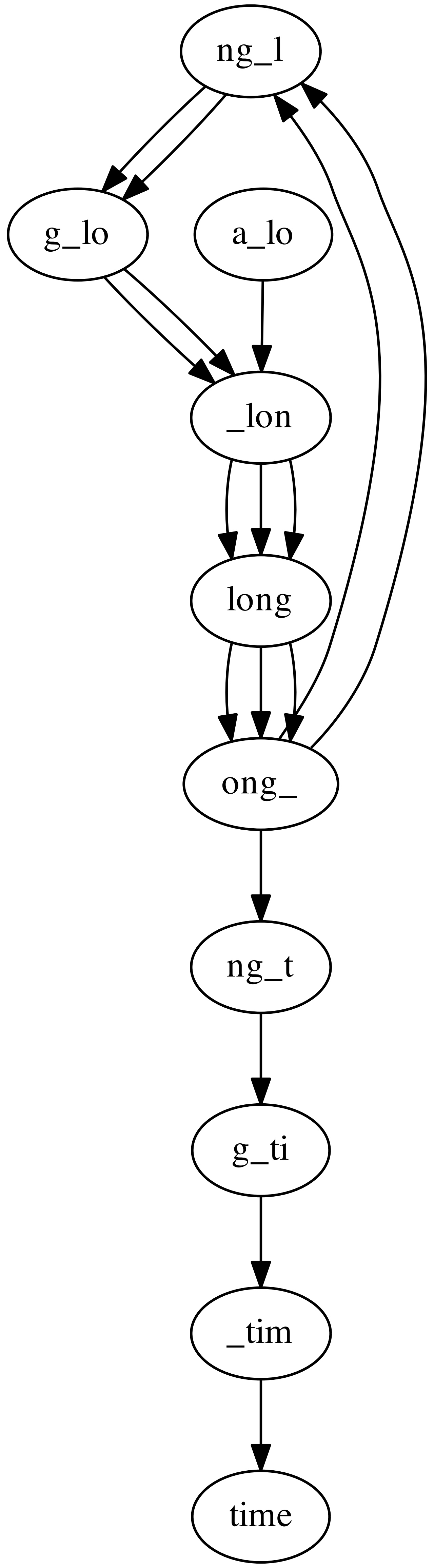
Last 5 k-mer additions, k = 5

a_long_long_long_time

Finished graph

De Bruijn graph

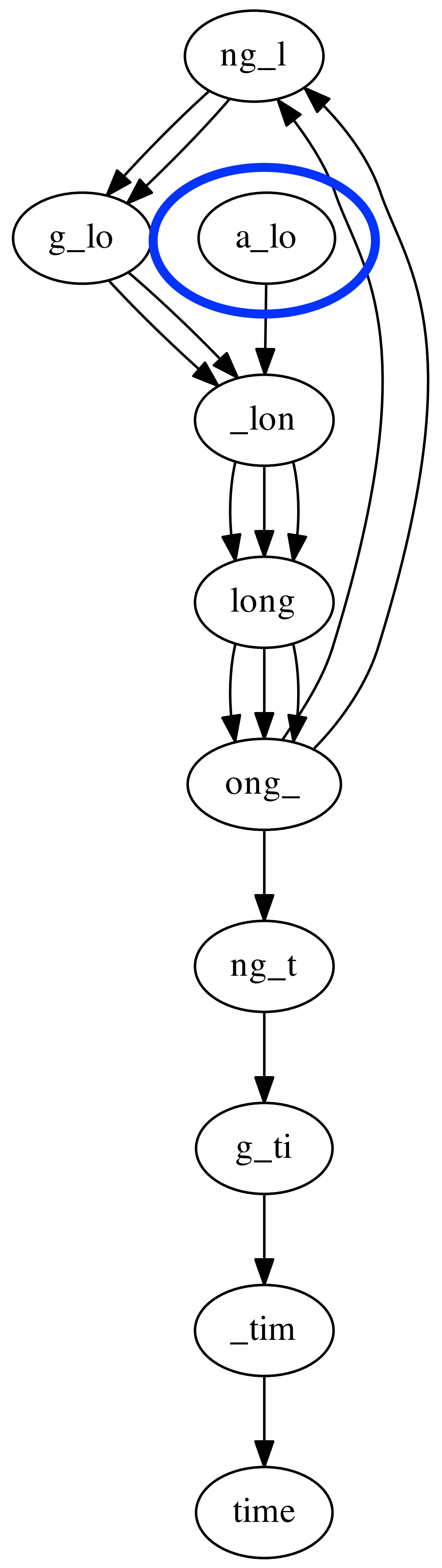
Procedure yields Eulerian graph. Why?



De Bruijn graph

Procedure yields Eulerian graph. Why?

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



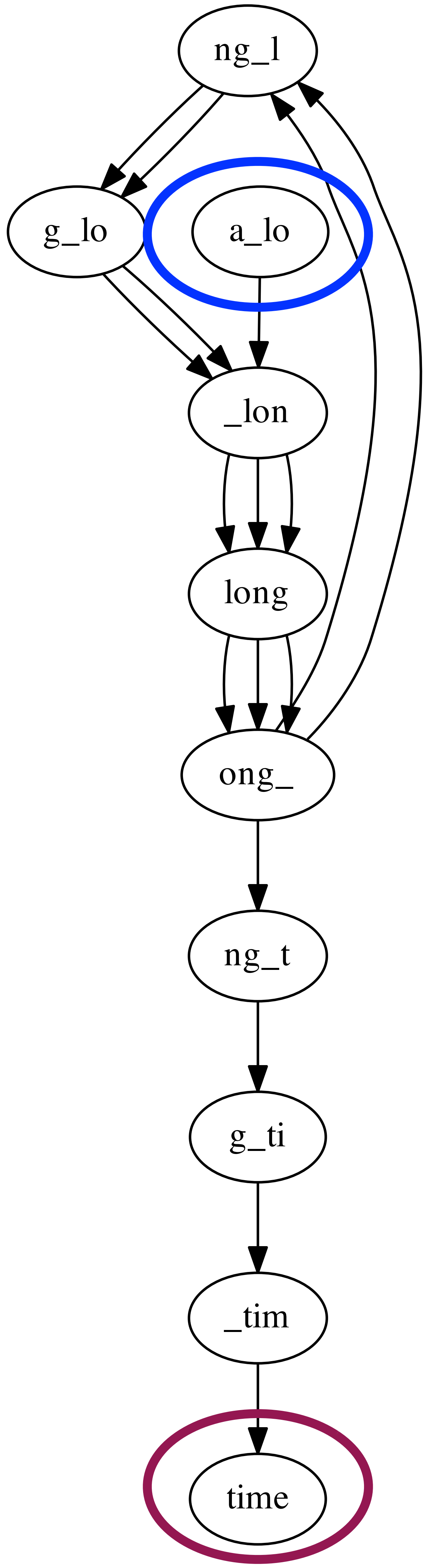
* Unless left- and right-most k-1-mers are equal

De Bruijn graph

Procedure yields 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 *



* Unless left- and right-most k-1-mers are equal

*

De Bruijn graph

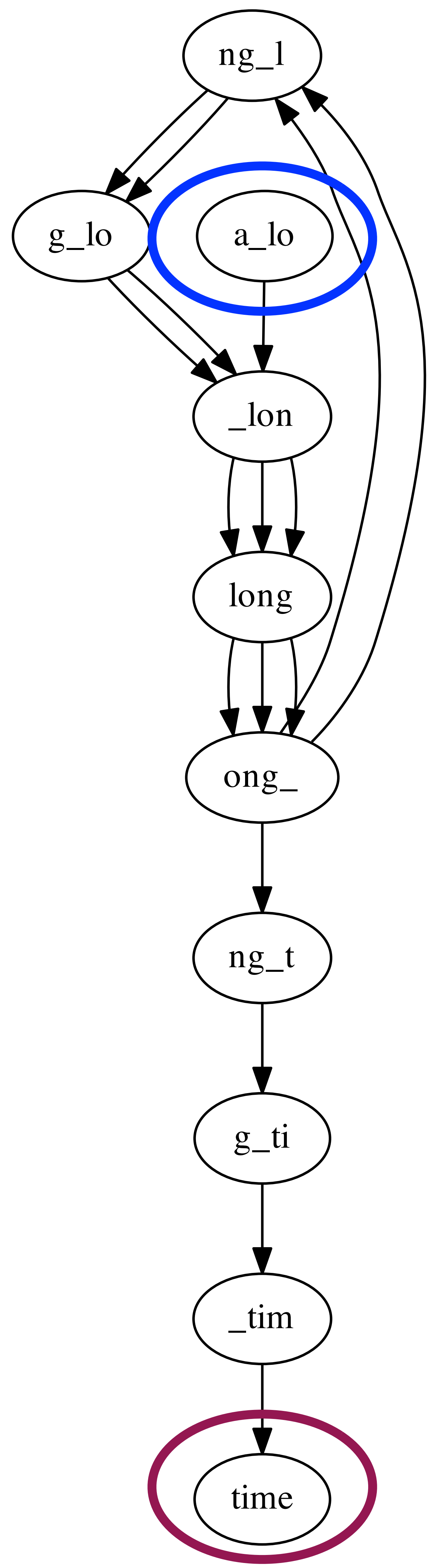
Procedure yields Eulerian graph. Why?

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 *

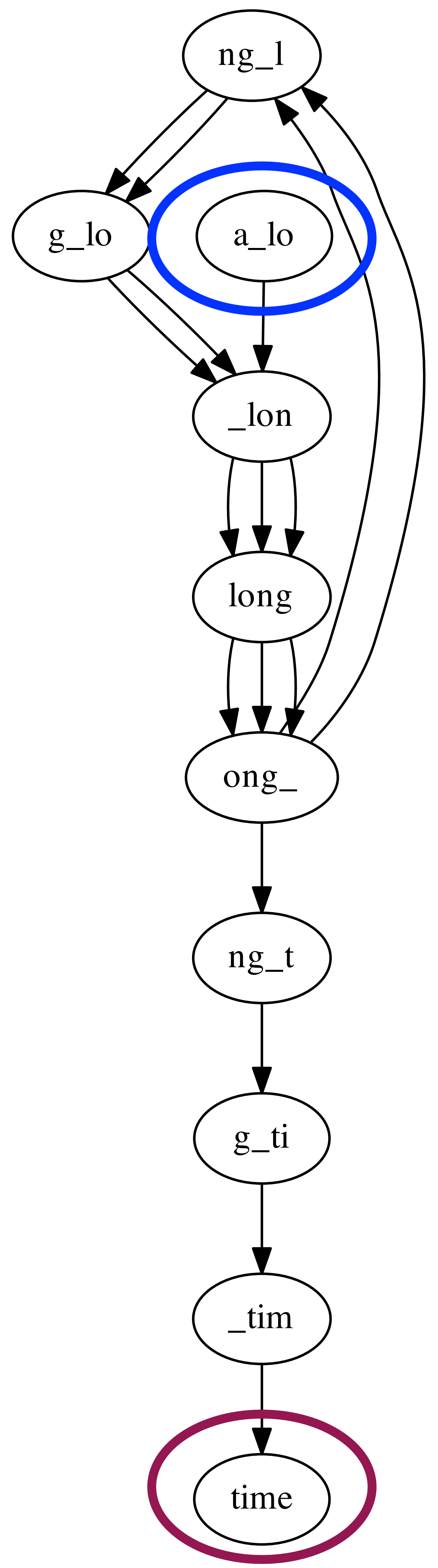
Other nodes are balanced since # times k-1-mer occurs as a left k-1-mer = # times it occurs as a right k-1-mer

* Unless left- and right-most k-1-mers are equal



De Bruijn graph

What string does the Eulerian path spell out?

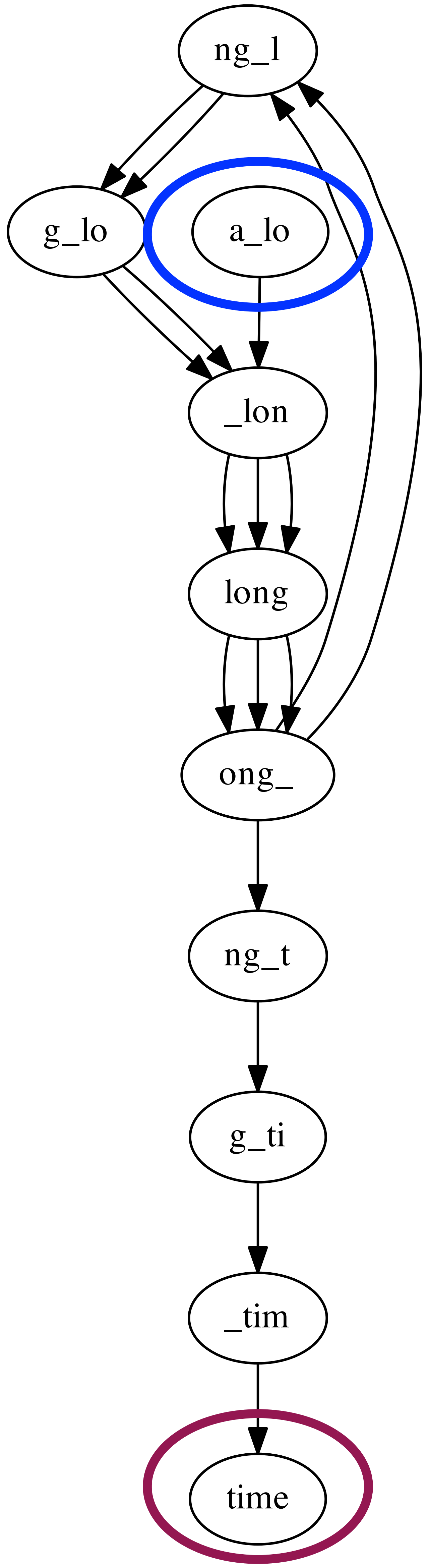


*

De Bruijn graph

What string does the Eulerian path spell out?

`a_long_long_long_time`

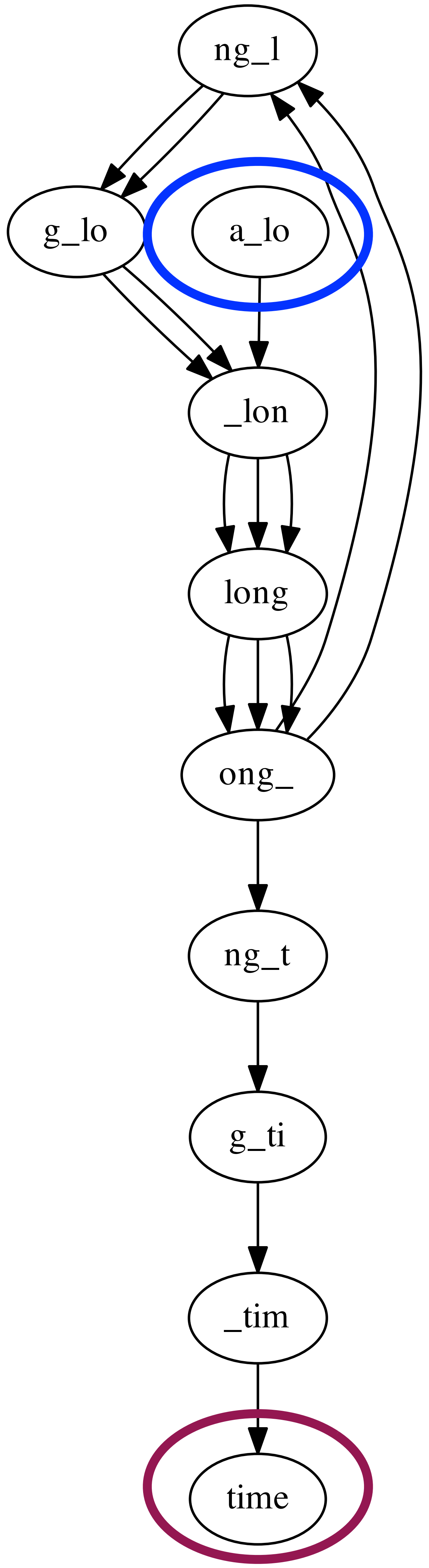


De Bruijn graph

What string does the Eulerian path spell out?

`a_long_long_long_time`

The original string! No collapsing!



De Bruijn graph builder implementation

```
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

De Bruijn graph

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

Convert graph into one with Eulerian cycle (add an edge to make all nodes balanced), then use this recursive procedure

Insight: If C is a cycle in an Eulerian graph, then after removing edges of C, remaining connected components are also Eulerian

```
# Make all nodes balanced, if not already

tour = []
# Pick arbitrary node
src = g.iterkeys().next()

def __visit(n):
 while len(g[n]) > 0:
  dst = g[n].pop()
  __visit(dst)
   tour.append(n)

__visit(src)
# Reverse order, omit repeated node
tour = tour[::-1][:-1]

# Turn tour into walk, if necessary
```

http://www.algorithmist.com/index.php/Eulerian_tour

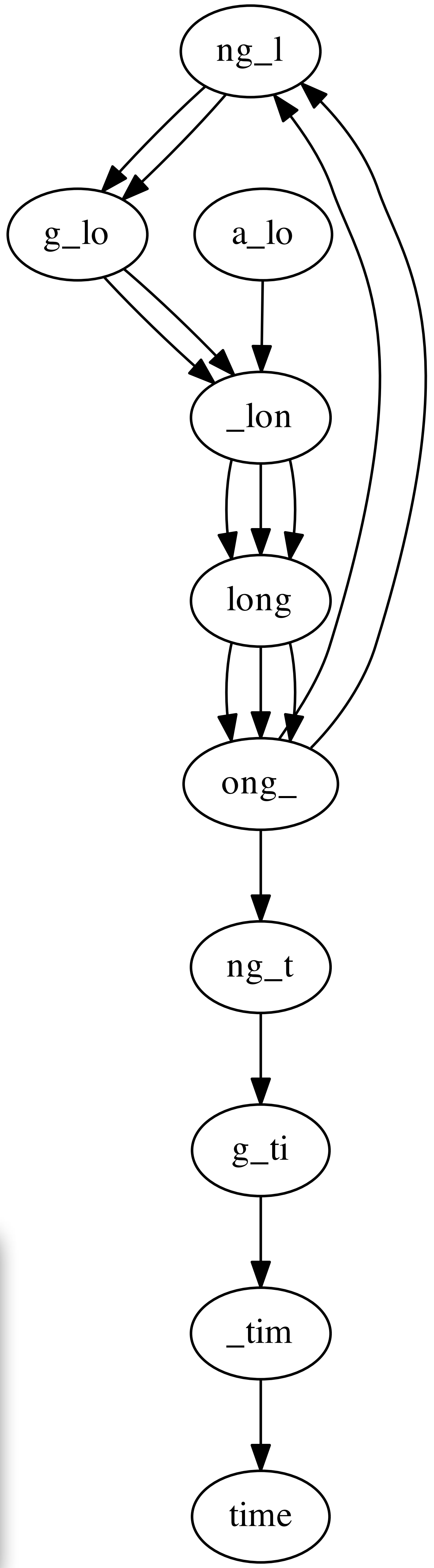
De Bruijn graph

Full illustrative de Bruijn graph and Eulerian walk implementation:

http://bit.ly/CG_DeBruijn

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

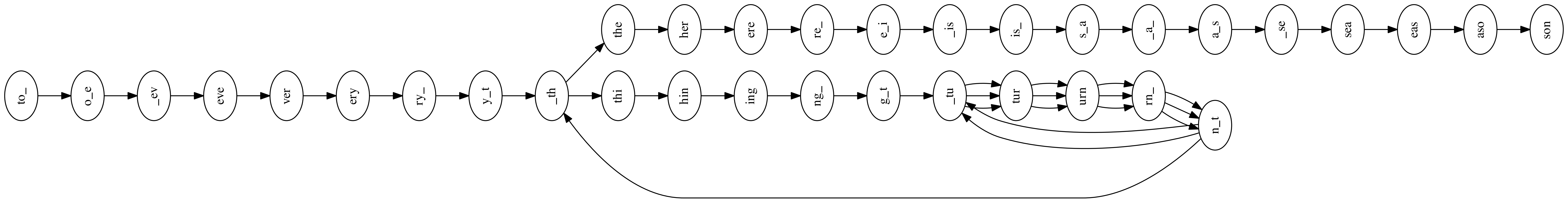
```
>>> G = DeBruijnGraph(["a_long_long_long_time"], 5)
>>> print G.eulerianWalkOrCycle()
['a_lo', '_lon', 'long', 'ong_', 'ng_l', 'g_lo',
'_lon', 'long', 'ong_', 'ng_l', 'g_lo', '_lon', 'long',
'ong_', 'ng_t', 'g_ti', '_tim', 'time']
```



De Bruijn graph

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

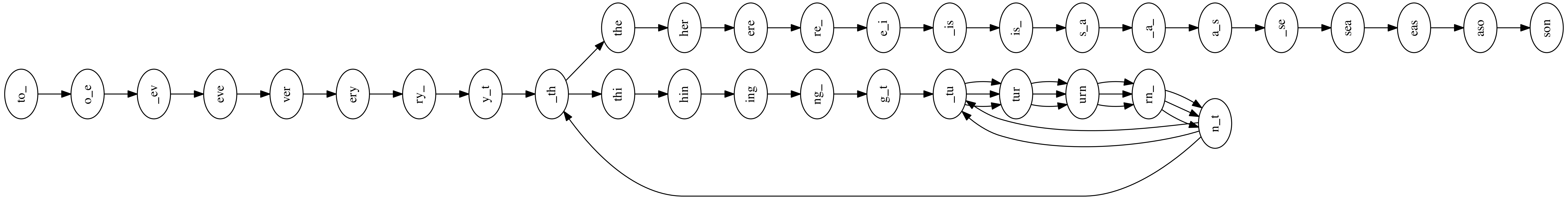
http://bit.ly/CG_DeBruijn



De Bruijn graph

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

http://bit.ly/CG_DeBruijn

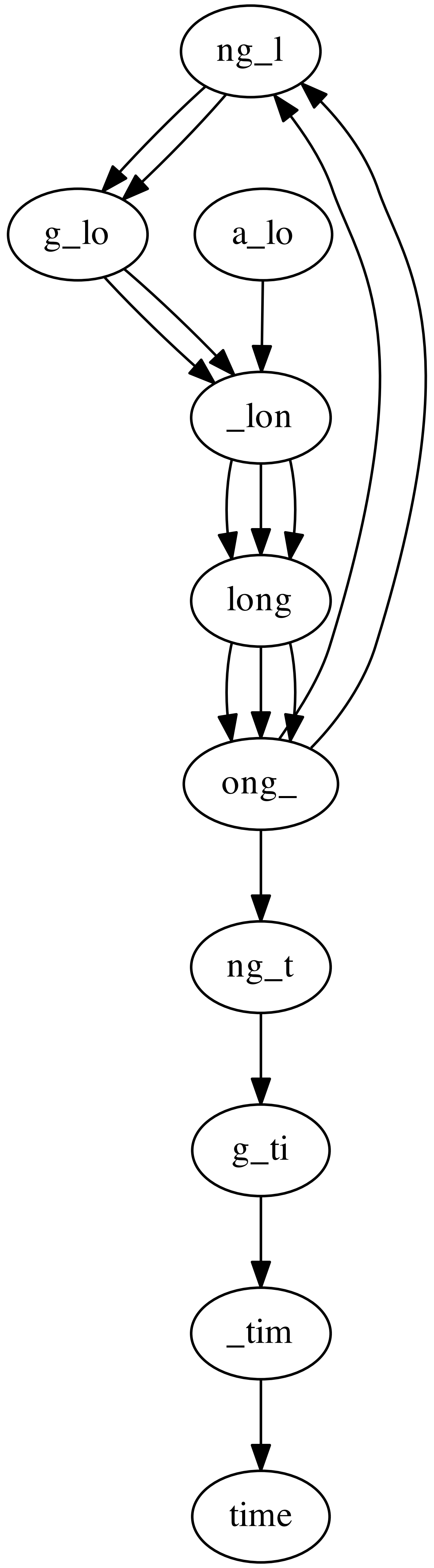


Recall: This is not generally possible or tractable in the overlap/SCS formulation

De Bruijn graph

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

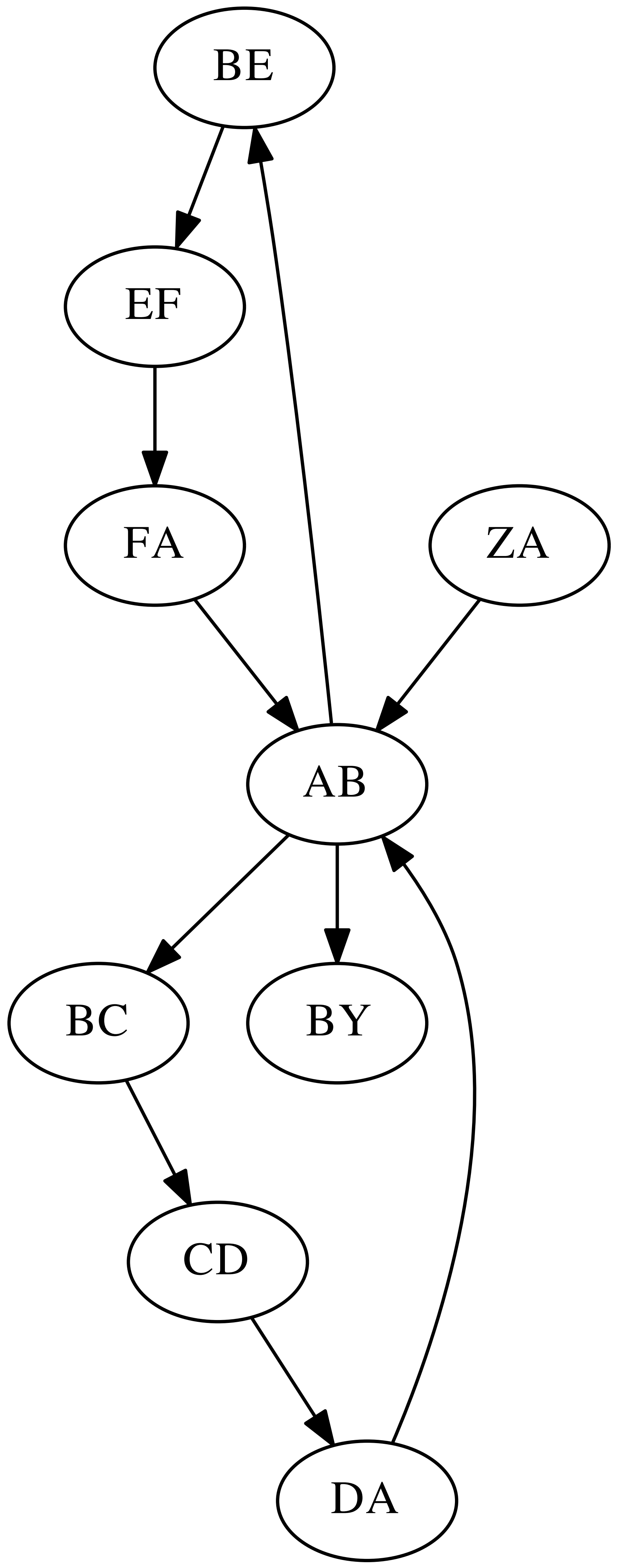
We saw cases where Eulerian walk corresponds to the original superstring. Is this always the case?



*

De Bruijn graph

Problem 1: Repeats still cause misassembles

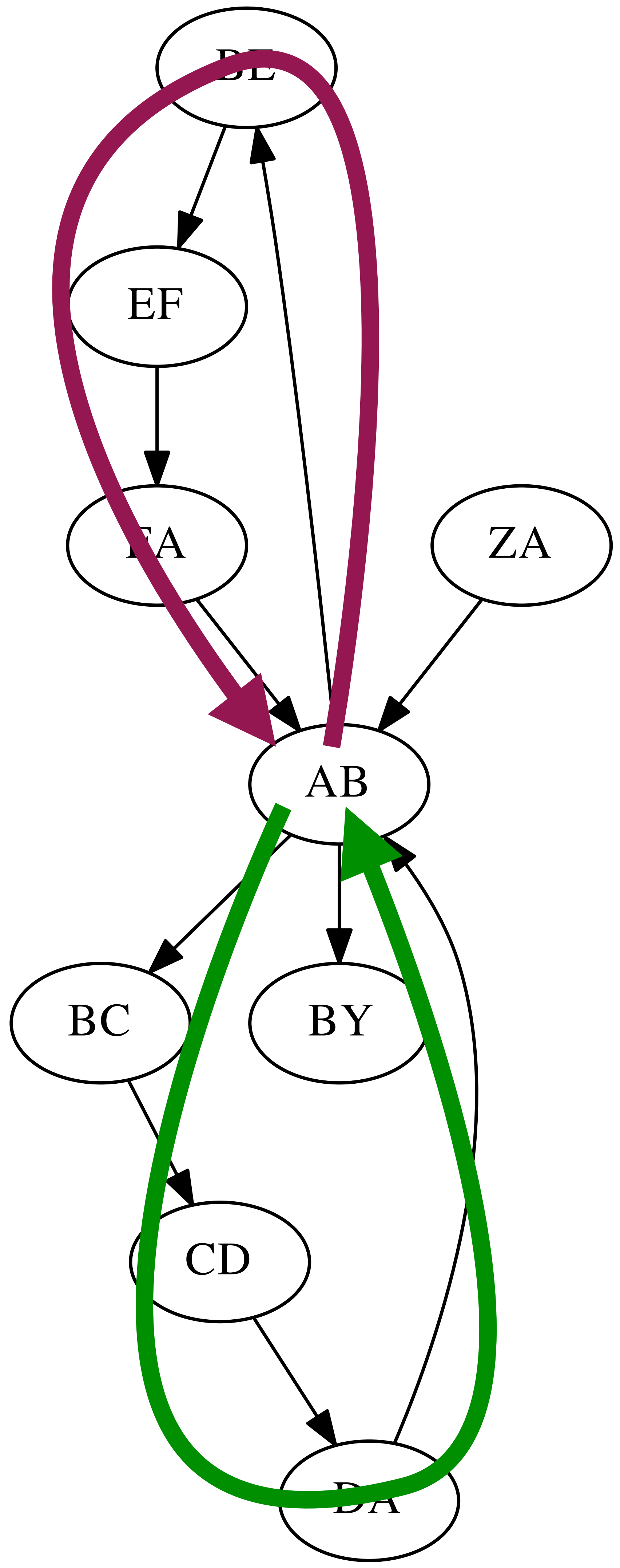


De Bruijn graph

Problem 1: Repeats still cause misassembles

ZA → AB → BE → EF → FA → AB → BC → CD → DA → AB → BY

ZA → AB → BC → CD → DA → AB → BE → EF → FA → AB → BY



De Bruijn graph

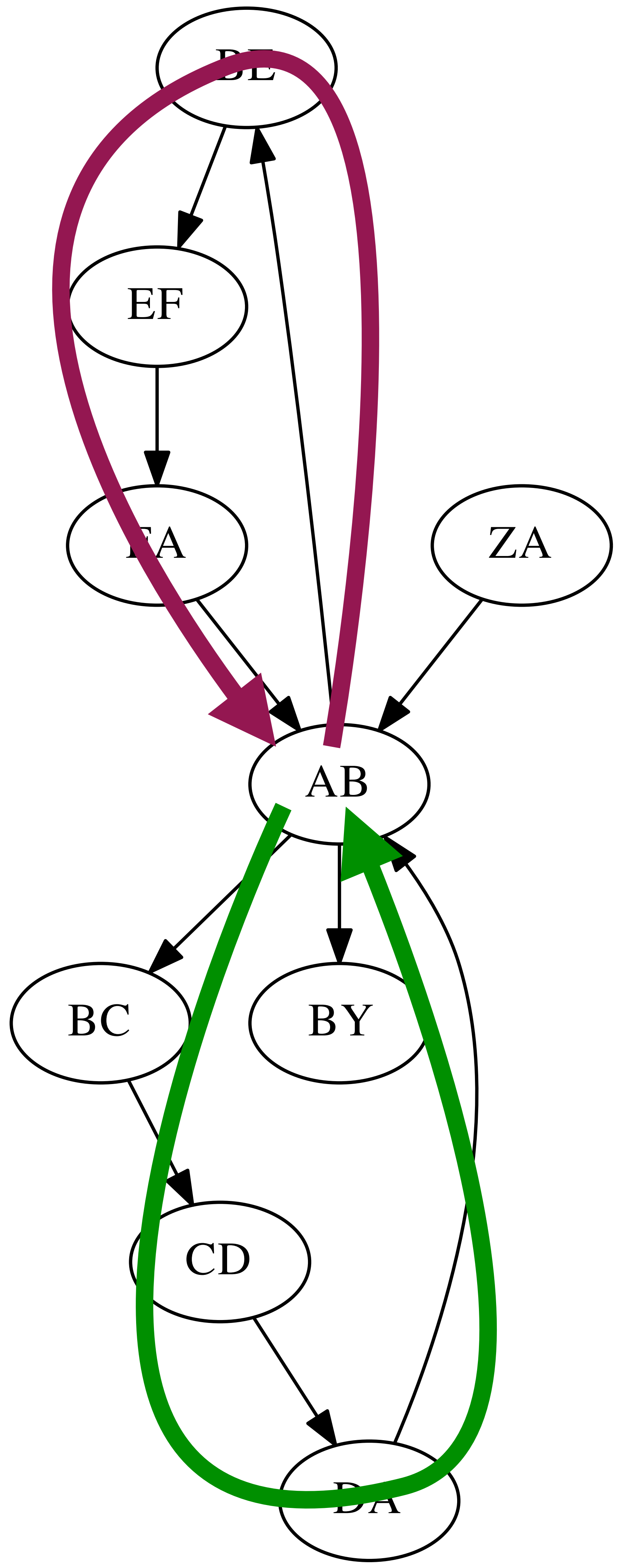
Problem 1: Repeats still cause misassembles

ZA → AB → BE → EF → FA → AB → BC → CD → DA → AB → BY

ZA → AB → BC → CD → DA → AB → BE → EF → FA → AB → BY

Problem 2:

We've been building DBGs assuming "perfect" sequencing: each k-mer reported exactly once, no mistakes. Real datasets aren't like that.



The Problem with Eulerian Paths

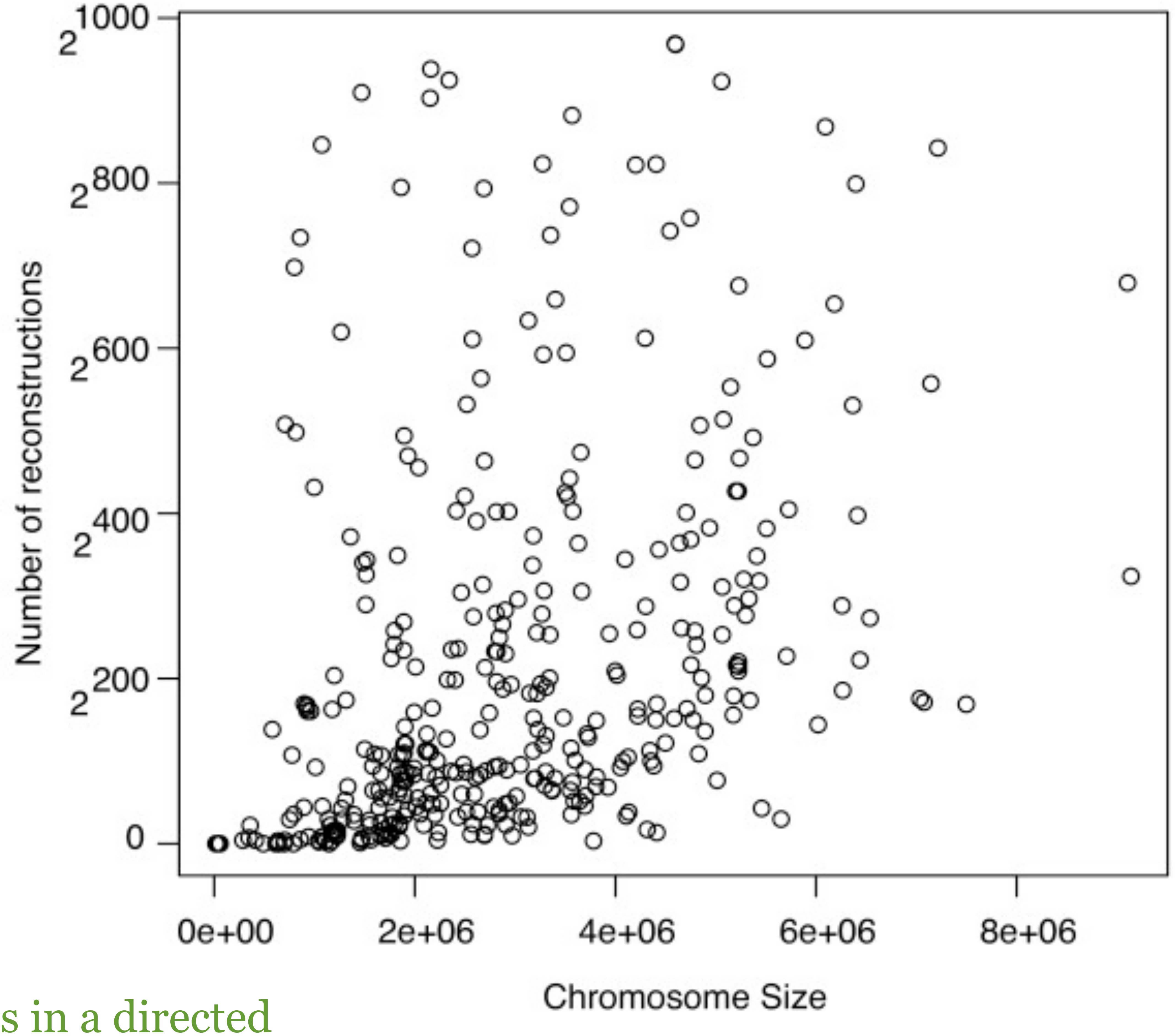
There are typically an astronomical number of possible Eulerian tours with perfect data.

Adding back constraints to limit # of tours leads to a NP-hard problem.

With imperfect data, there are usually NO Eulerian tours

Estimating # of parallel edges is usually tricky.

k = 50



(Kingsford, Schatz, Pop, 2010)

Aside: counting # of Eulerian tours in a directed graph is easy, but in an undirected graph is #P-complete (hard).

* slide courtesy of Carl Kingsford

Third law of assembly

Repeats make assembly difficult; whether we can assemble without mistakes depends on length of reads and repetitive patterns in genome

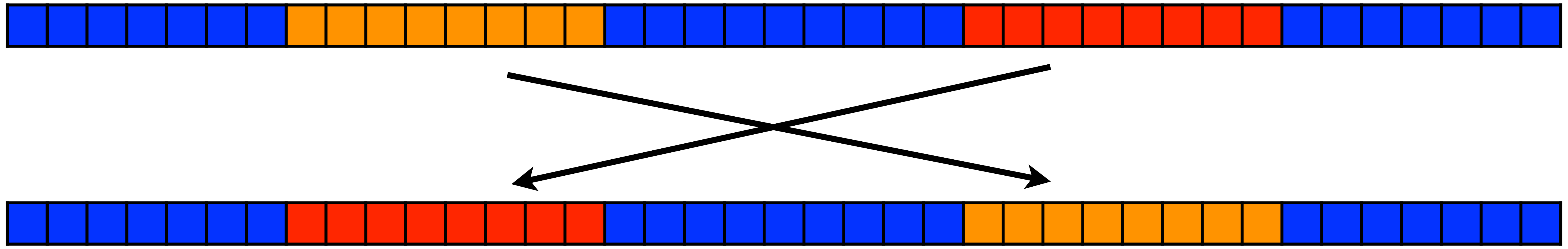
Collapsing:

a_long_long_long_time

↓

a_long_long_time

Shuffling:

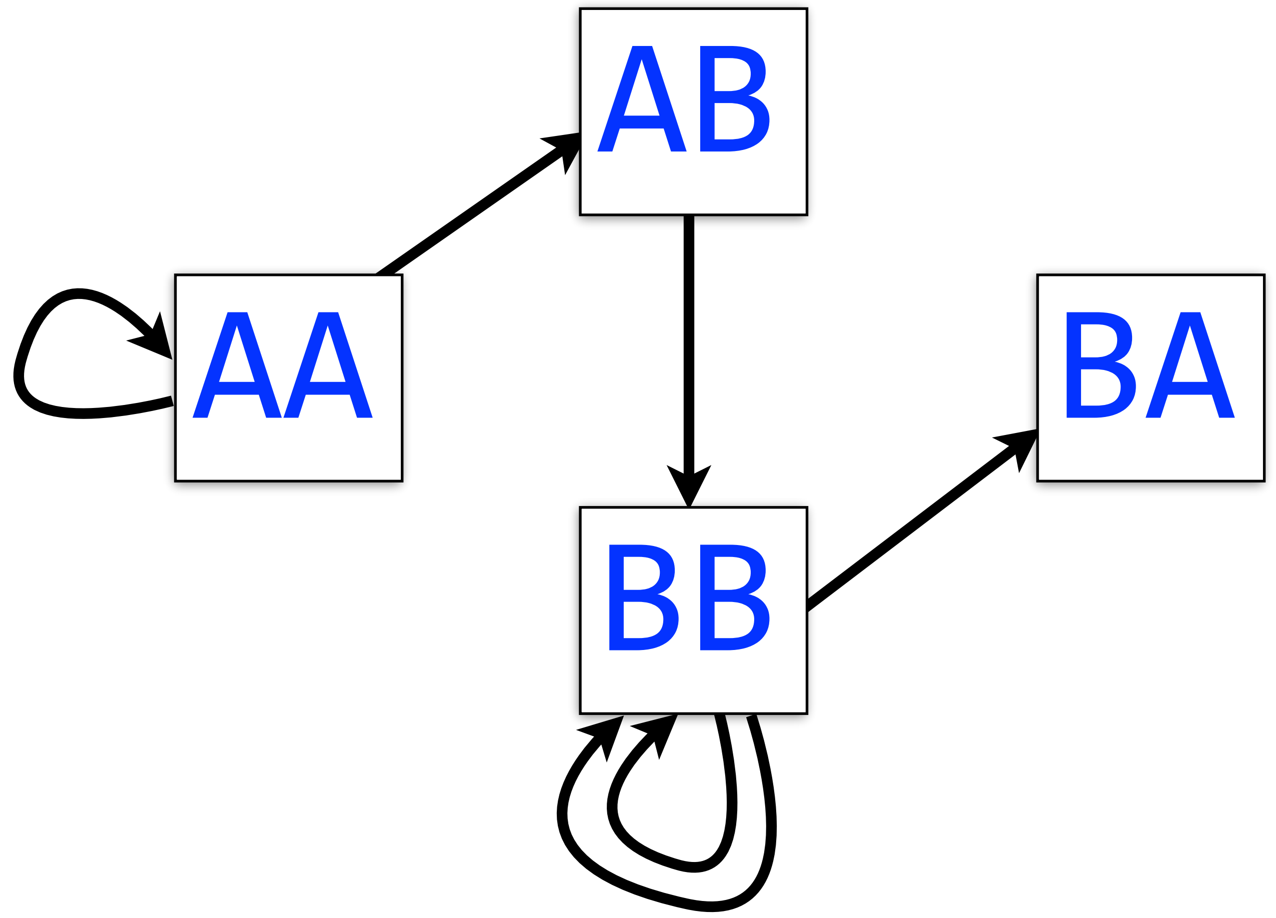


De Bruijn graph

genome: AAABBBBA

3-mers: AAA, AAB, ABB, BBB, BBB, BBA

L/R 2-mers: AA, AA   AA, AB   AB, BB   BB, BB   BB, BB   BB, BA



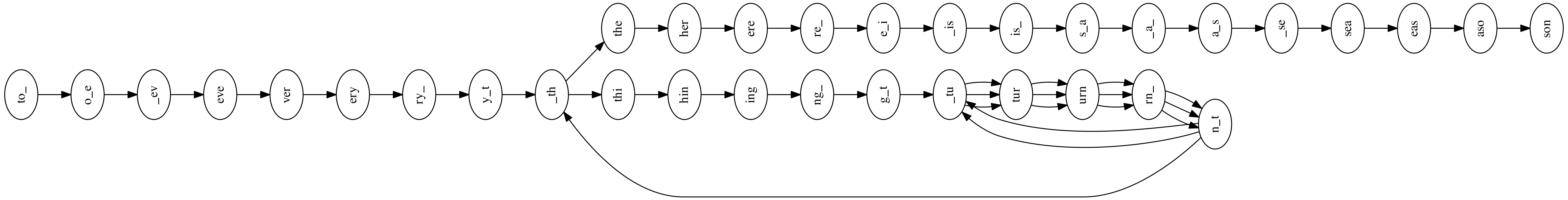
One edge per k-mer

One node per distinct k-1-mer

De Bruijn graph

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

http://bit.ly/CG_DeBruijn



De Bruijn graph

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
```

De Bruijn graph

Case where k = 4 works:

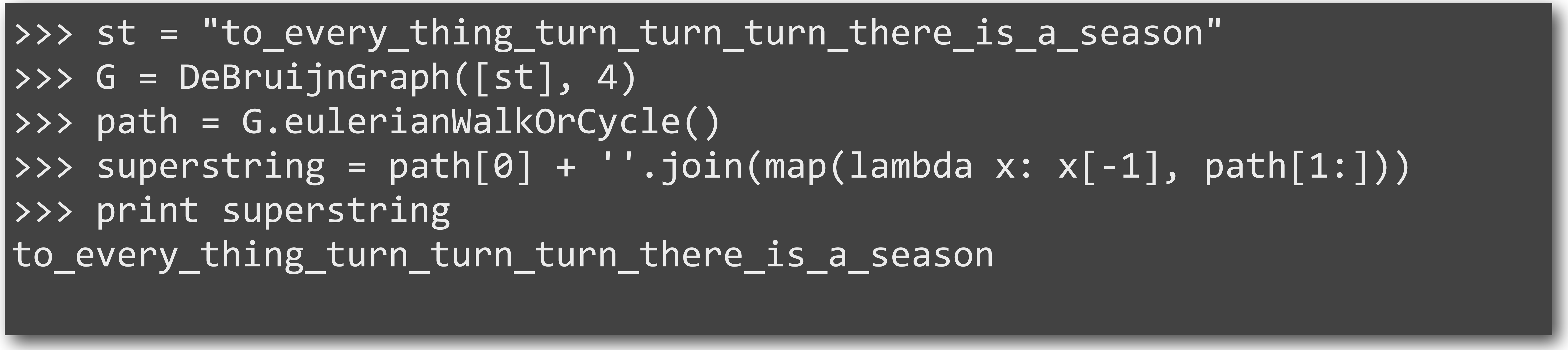
```
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 4)
>>> path = G.eulerianWalkOrCycle()
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
to_every_thing_turn_turn_turn_there_is_a_season
```

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
```

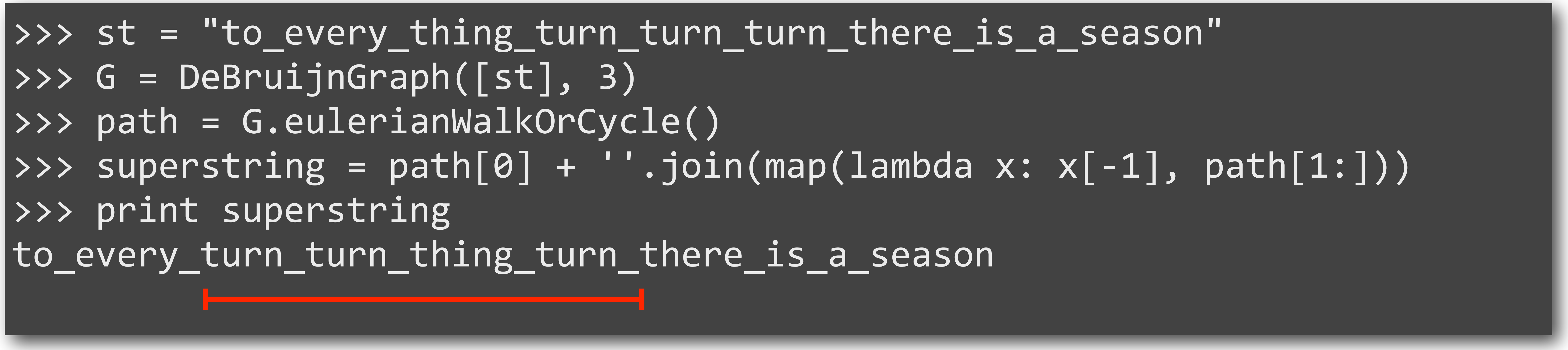
De Bruijn graph

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:



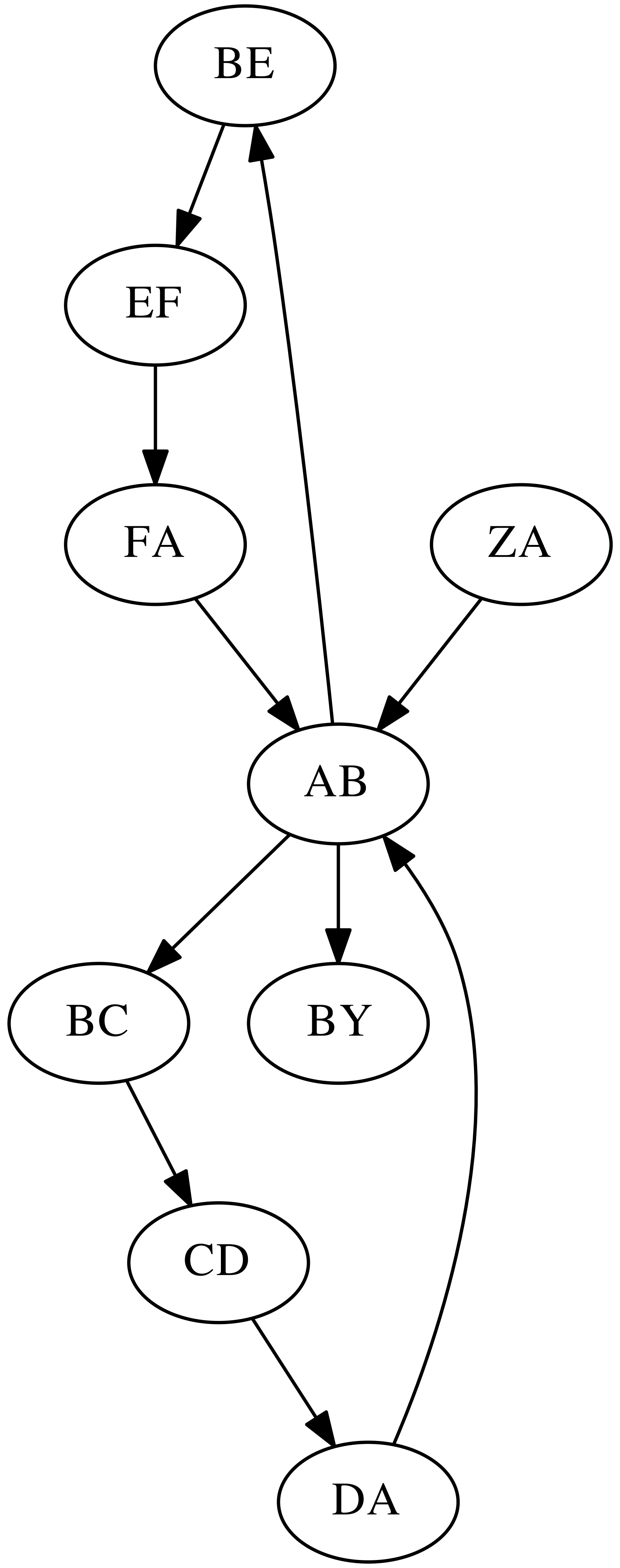
```
>>> st = "to_every_thing_turn_turn_turn_there_is_a_season"
>>> G = DeBruijnGraph([st], 3)
>>> path = G.eulerianWalkOrCycle()
>>> superstring = path[0] + ''.join(map(lambda x: x[-1], path[1:]))
>>> print superstring
to_every_turn_turn_thing_turn_there_is_a_season
```

Due to repeats that are unresolvable at k = 3

*

De Bruijn graph

Problem 1: Repeats still cause misassembles



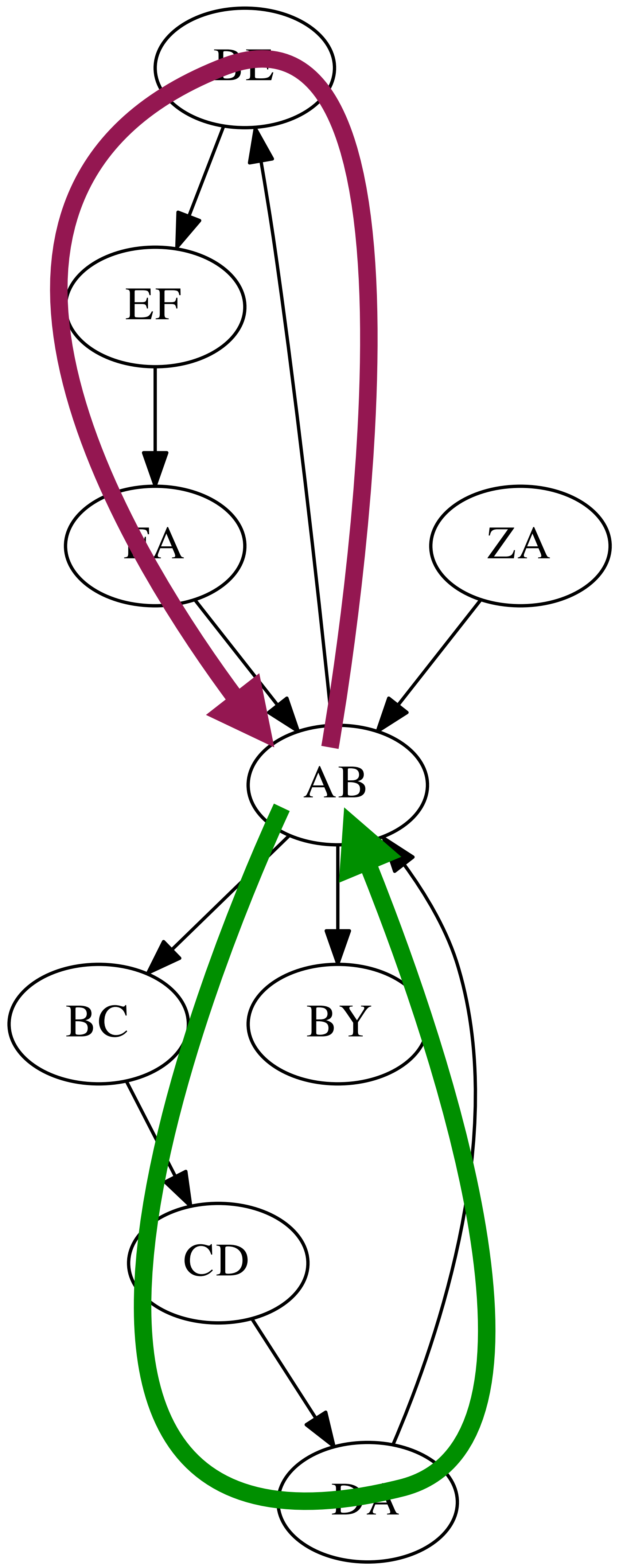
*

De Bruijn graph

Problem 1: Repeats still cause misassembles

ZA → AB → BE → EF → FA → AB → BC → CD → DA → AB → BY

ZA → AB → BC → CD → DA → AB → BE → EF → FA → AB → BY



De Bruijn graph

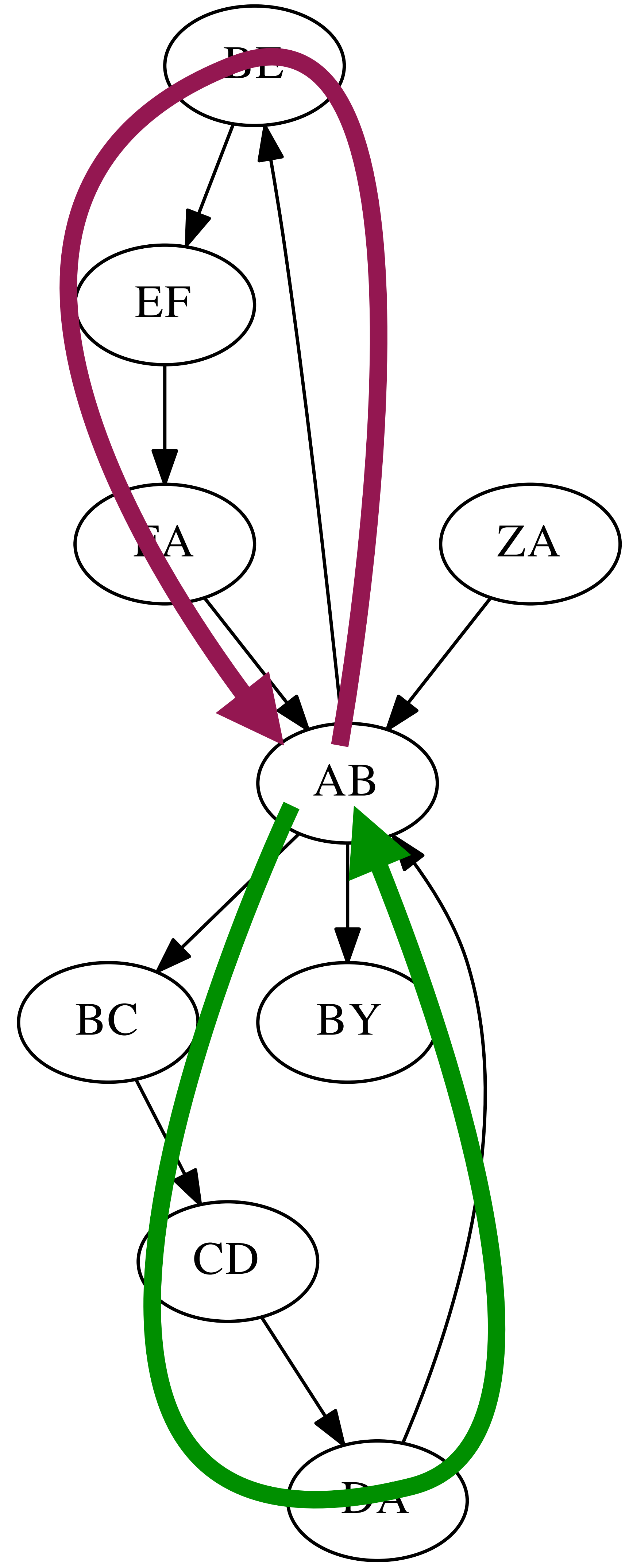
Problem 1: Repeats still cause misassembles

ZA → AB → BE → EF → FA → AB → BC → CD → DA → AB → BY

ZA → AB → BC → CD → DA → AB → BE → EF → FA → AB → BY

Problem 2:

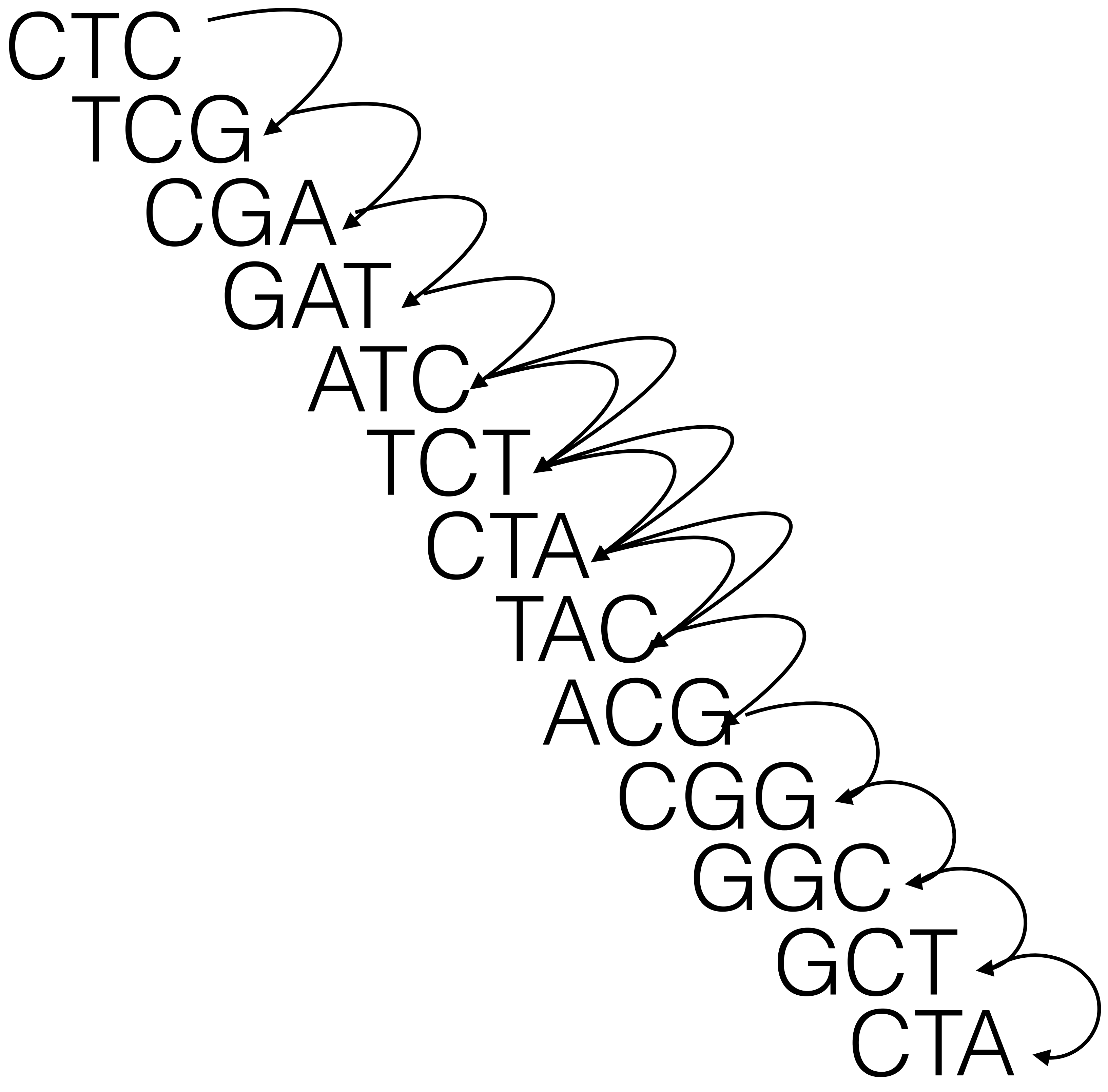
We've been building DBGs assuming "perfect" sequencing: each k-mer reported exactly once, no mistakes. Real datasets aren't like that.



Uneven coverage foils Eulerian Paths

r1: CTCGATCTAC
r2: ATCTACGGCTA

k=4

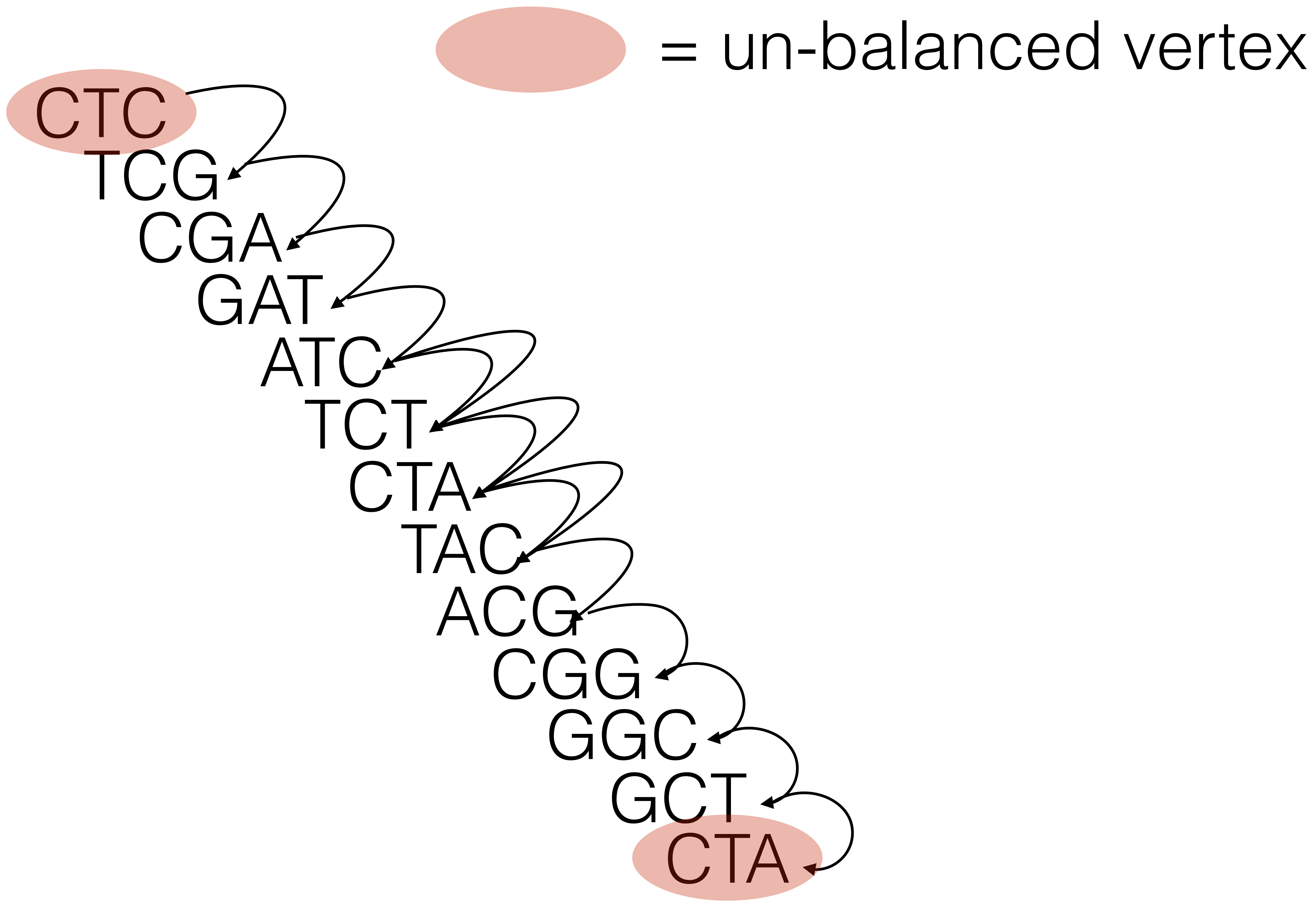


CTC
TCG
CGA
GAT
ATC
TCT
CTA
TAC
ACG
CGG
GGC
GCT
CTA

Uneven coverage foils Eulerian Paths

r1: CTCGATCTAC
r2: ATCTACGGCTA

k=4

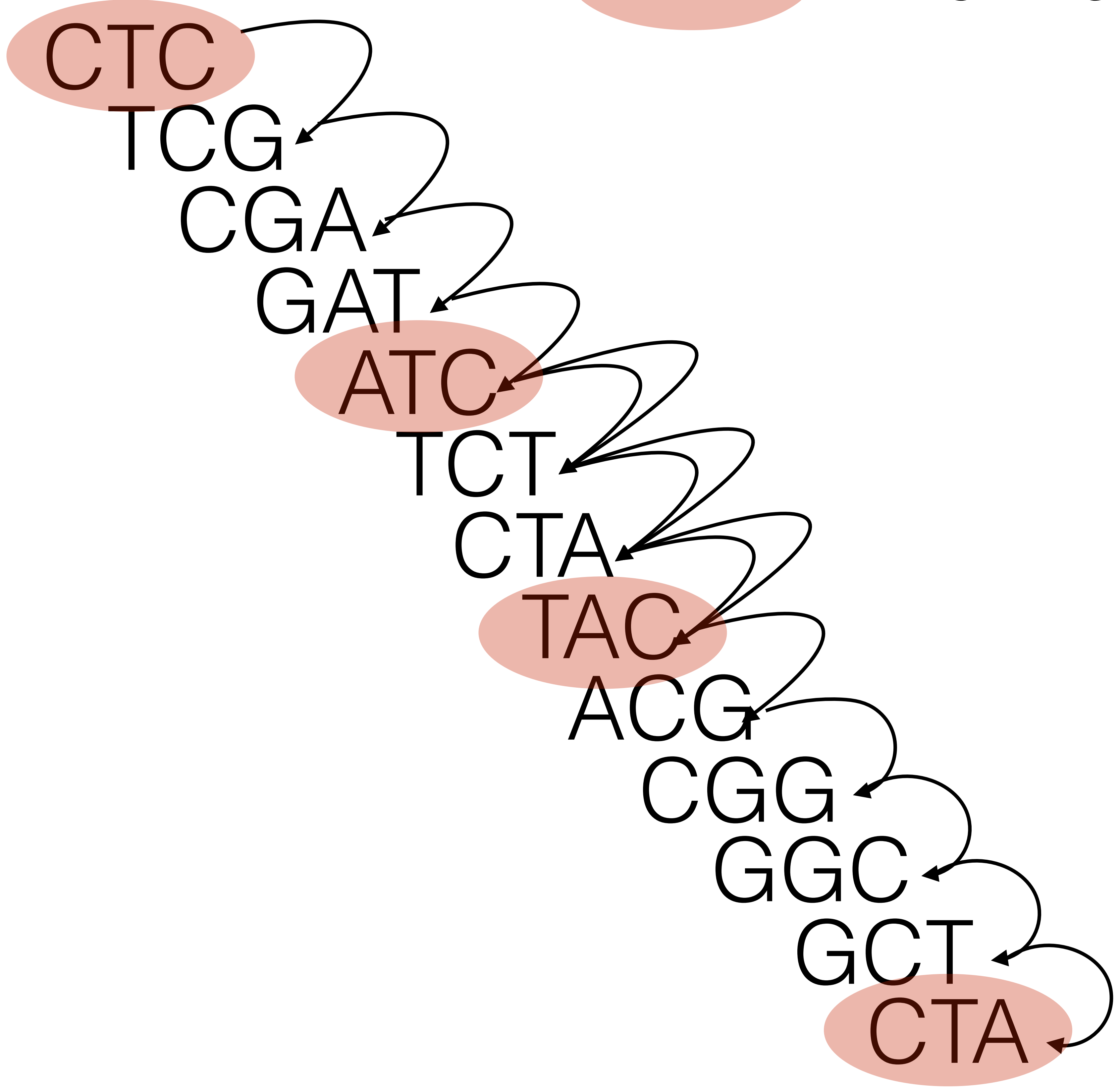


Uneven coverage foils Eulerian Paths

r1: CTCGATCTAC
r2: ATCTACGGCTA

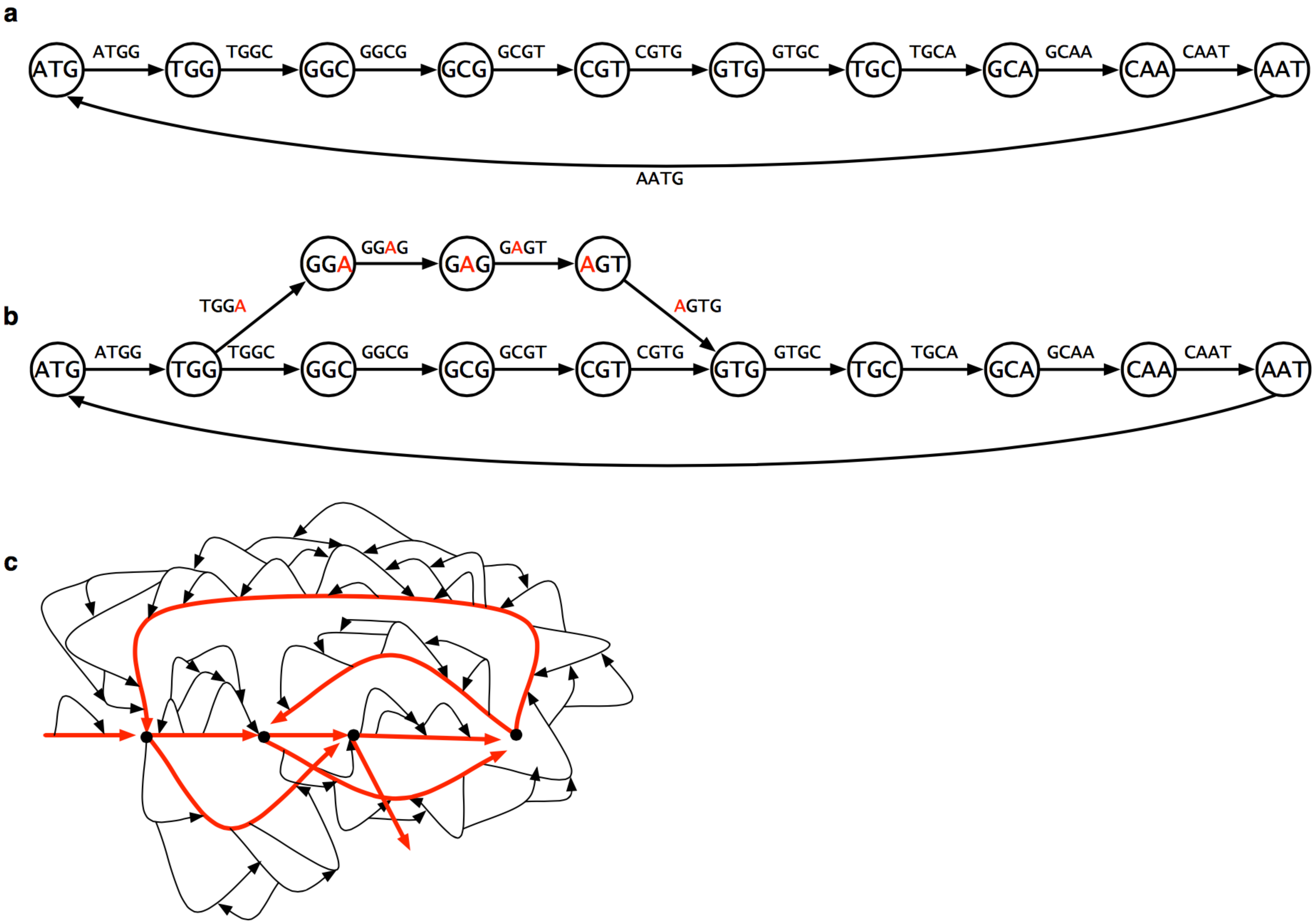
k=4

● = un-balanced vertex



CTC
TCG
CGA
GAT
ATC
TCT
CTA
TAC
ACG
CGG
GGC
GCT
CTA

Bursting bubbles



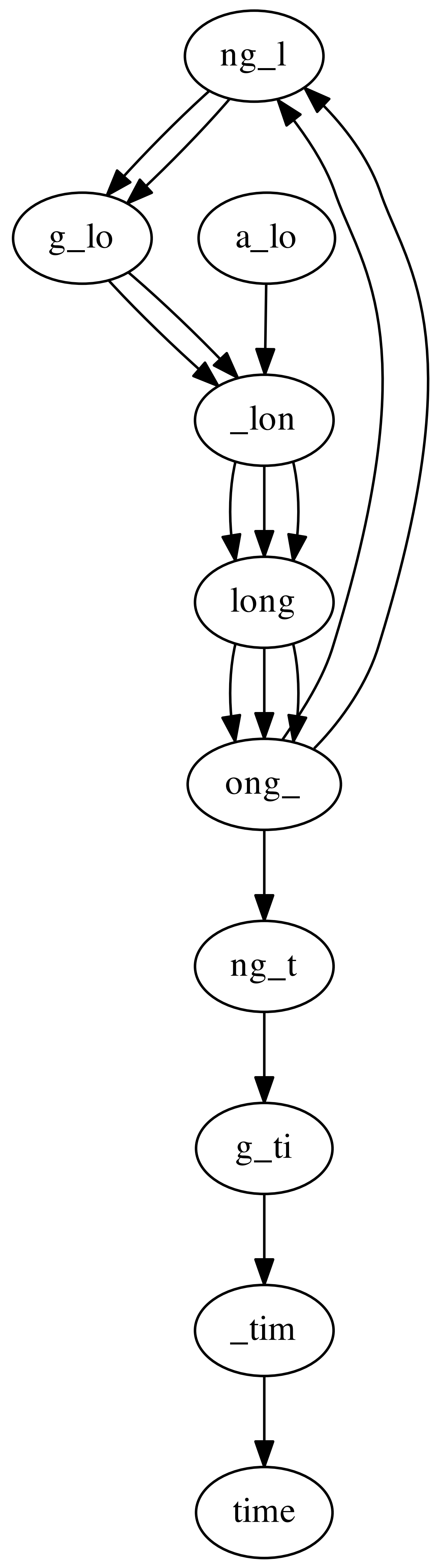
Compeau, Phillip EC, Pavel A. Pevzner, and Glenn Tesler. "How to apply de Bruijn graphs to genome assembly." *Nature biotechnology* 29.11 (2011): 987-991.

*

De Bruijn graph

Gaps in coverage (missing k-mers) lead to disconnected or non-Eulerian graph

Graph for `a_long_long_long_time`, k = 5:

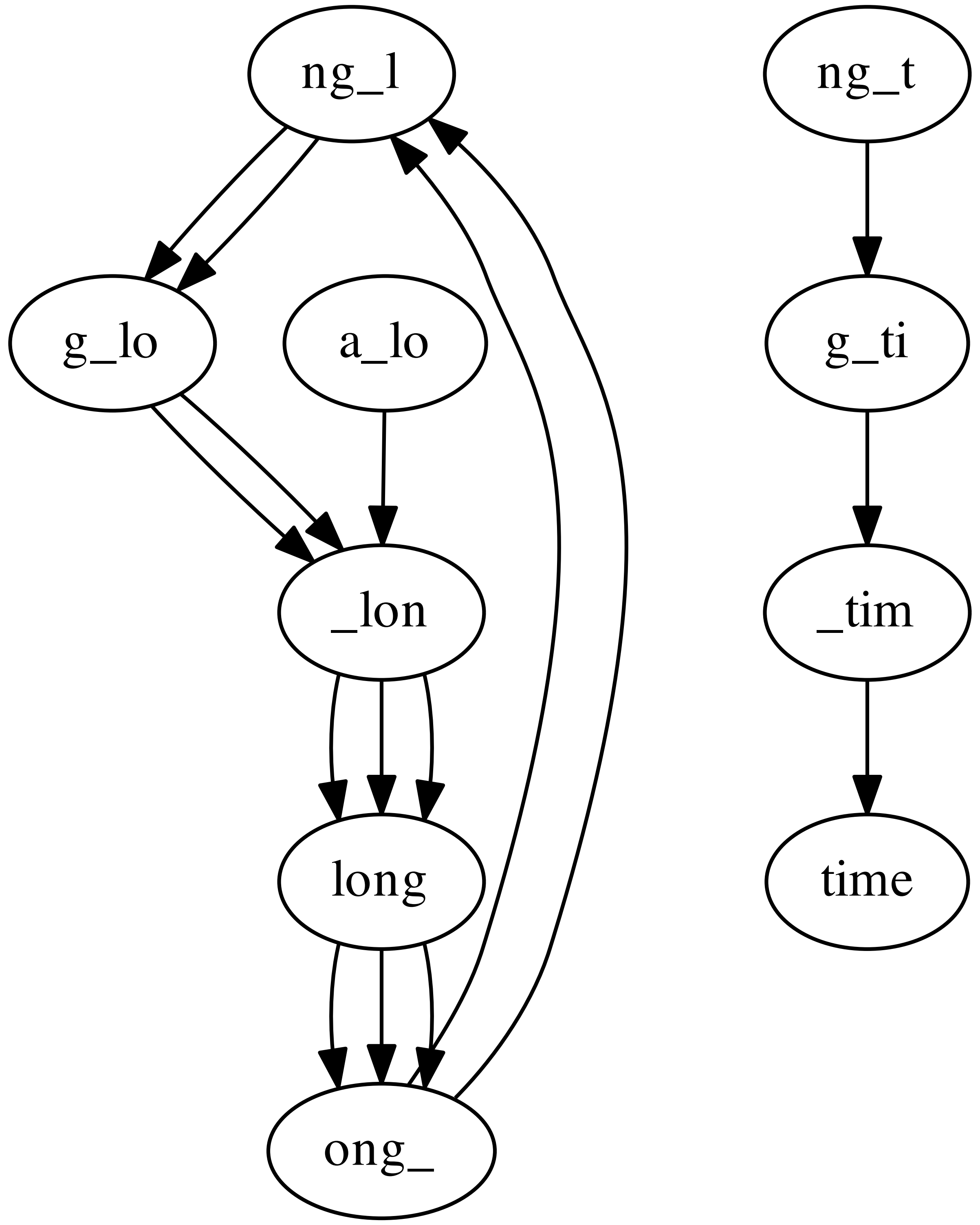


*

De Bruijn graph

Gaps in coverage (missing k-mers) lead to disconnected or non-Eulerian graph

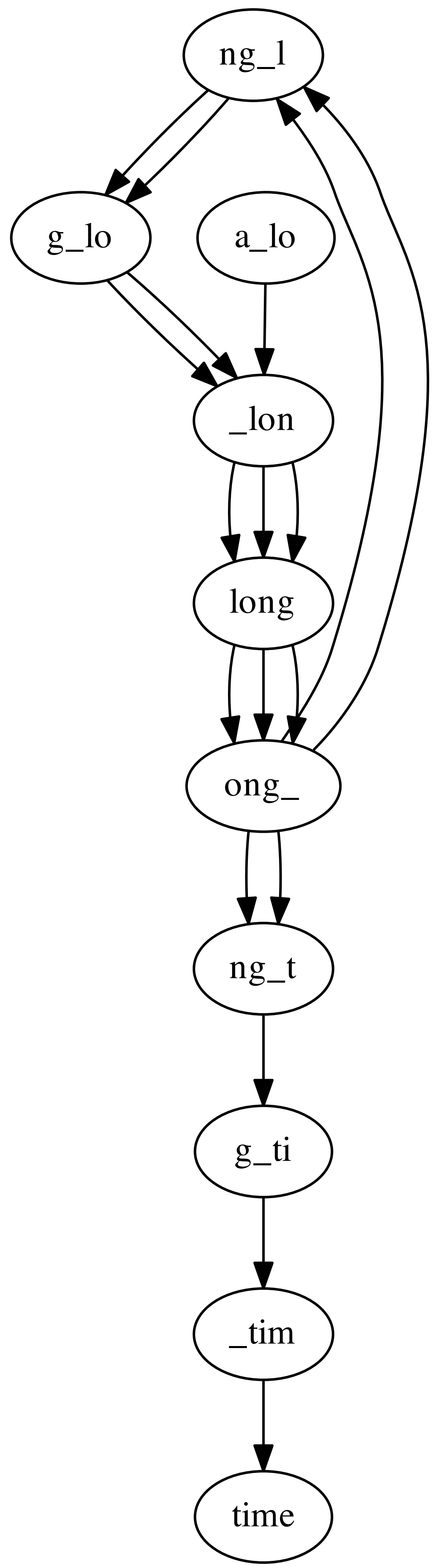
Graph for `a_long_long_long_time`, k = 5 but omitting `ong_t` :



De Bruijn graph

Coverage differences make graph non-Eulerian

Graph for `a_long_long_long_time`, k = 5, with extra copy of `ong_t` :



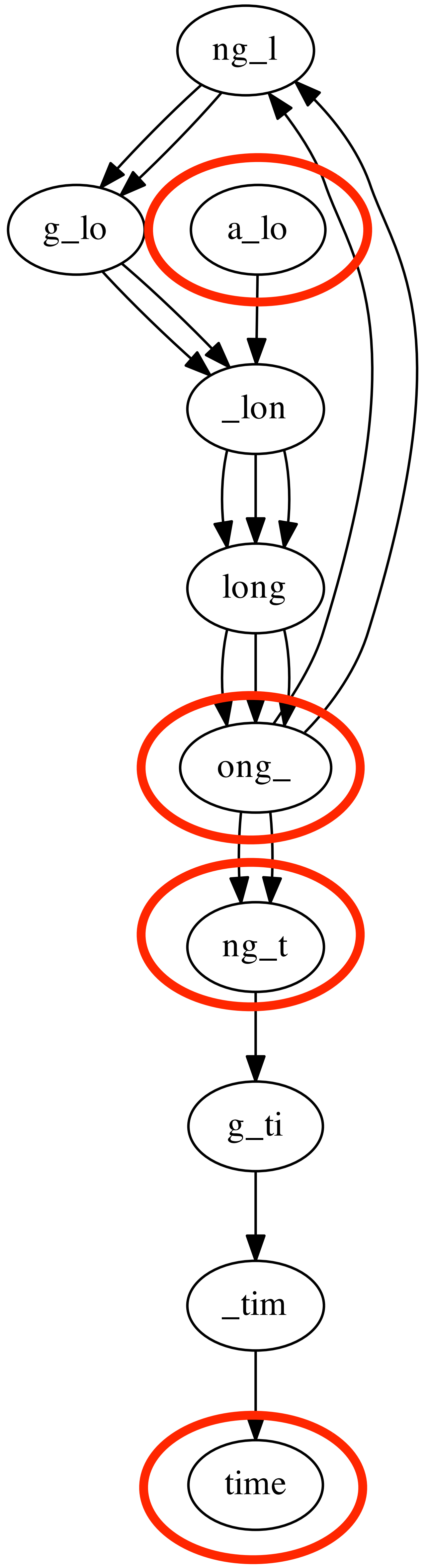
*

De Bruijn graph

Coverage differences make graph non-Eulerian

Graph for `a_long_long_long_time`, k = 5, with extra copy of `ong_t` :

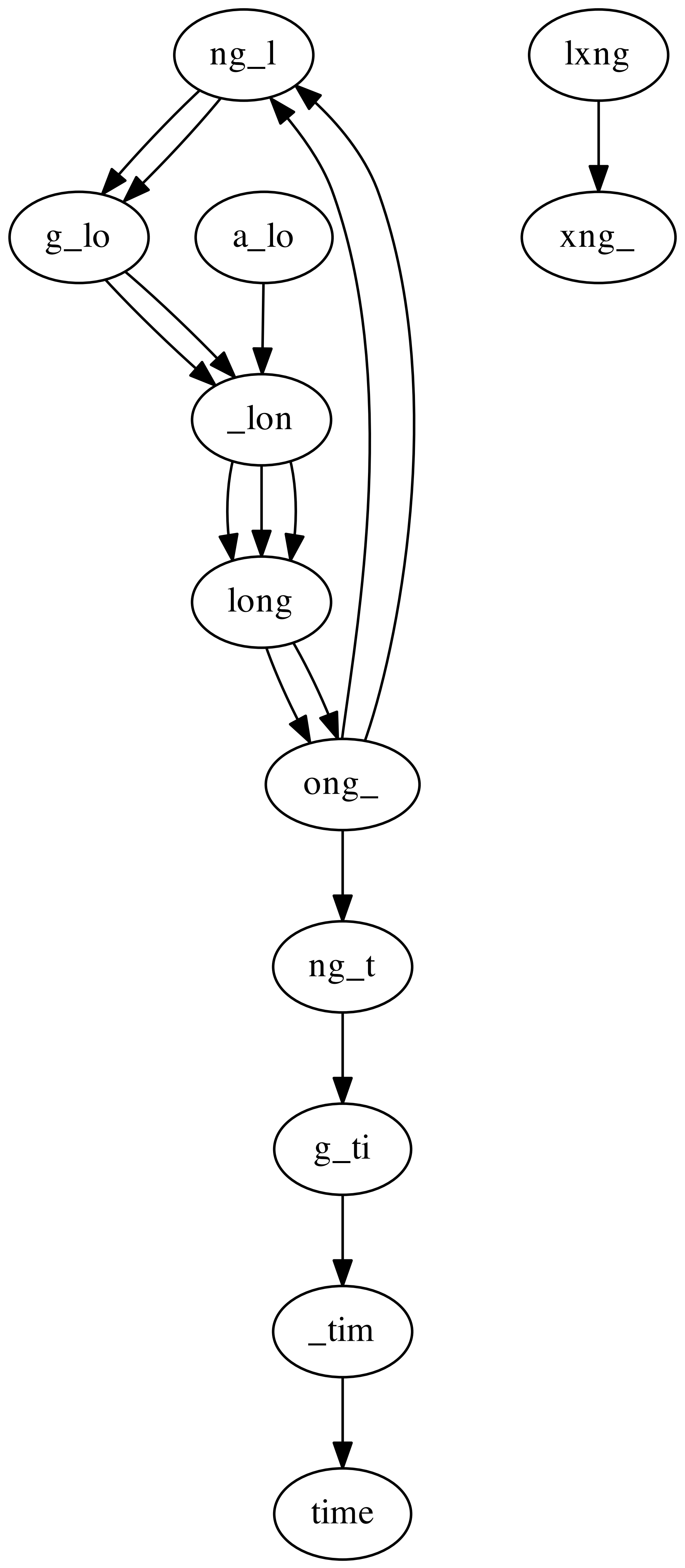
4 semi-balanced nodes



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 one copy of `long_` into `lxng_`



*

De Bruijn graph

Casting assembly as Eulerian walk is appealing, but not practical

*

De Bruijn graph

Casting assembly as Eulerian walk is appealing, but not practical

- Uneven coverage, sequencing errors, etc make graph non-Eulerian

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 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) seeks a walk over the De Bruijn graph, where walk contains each read as a subwalk

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) seeks 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**

$$\left.\begin{array}{l}\mathbf{d} = 6 \text{ x } 10^9 \text{ reads}\\ \mathbf{n} = 100 \text{ nt}\end{array}\right\} \approx 1 \text{ sequencing run}$$

$$\mathbf{m} = 3 \text{ x } 10^9 \text{ nt} \approx \text{human}$$

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.
```

*

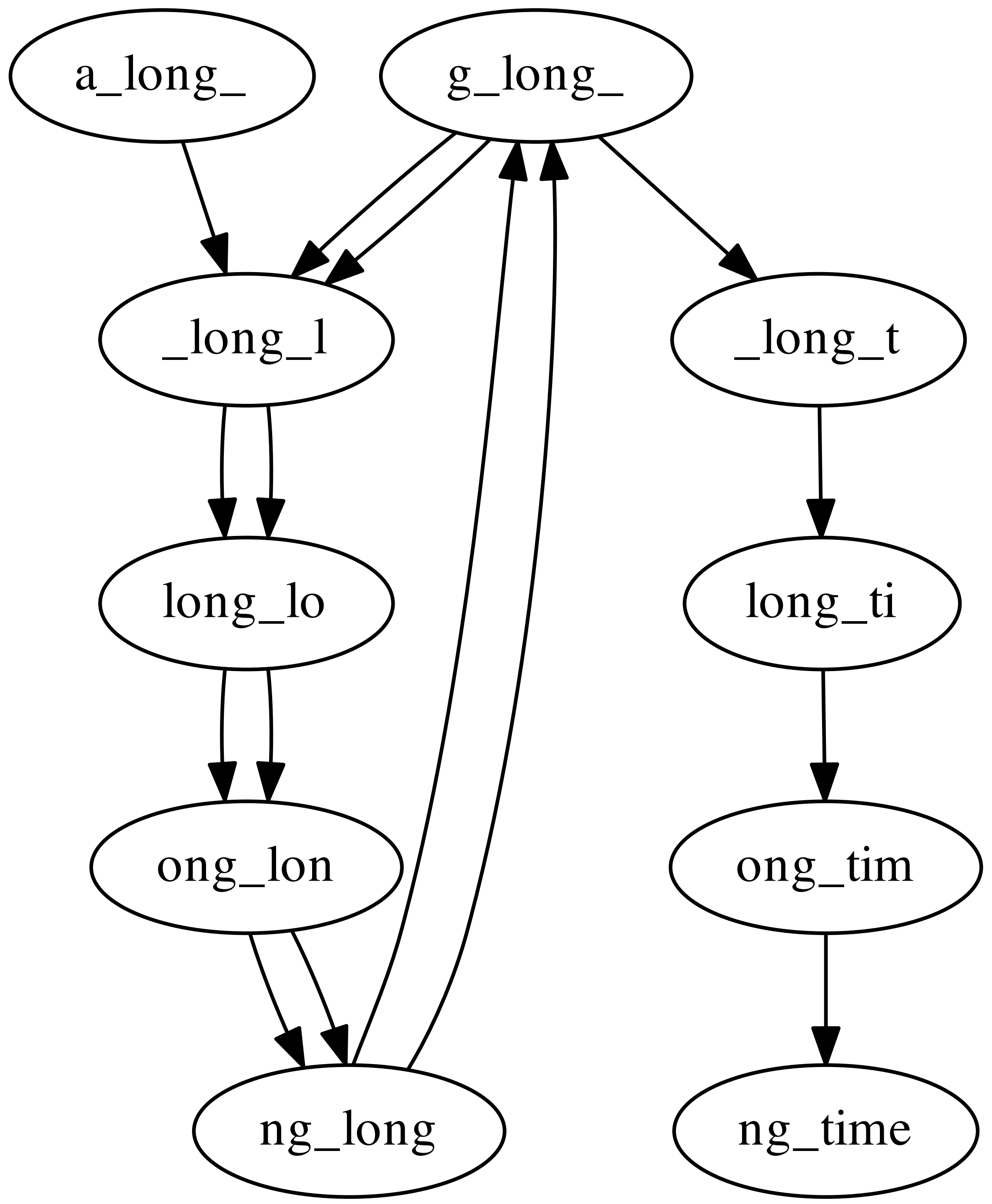
De Bruijn graph

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          ng_long_  g_long_t
 _long_lo          g_long_l  _long_ti
  long_lon                    long_tim
   ong_long                    ong_time
    ng_long_
     g_long_l
      _long_lo
       long_lon
        ong_long
```



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

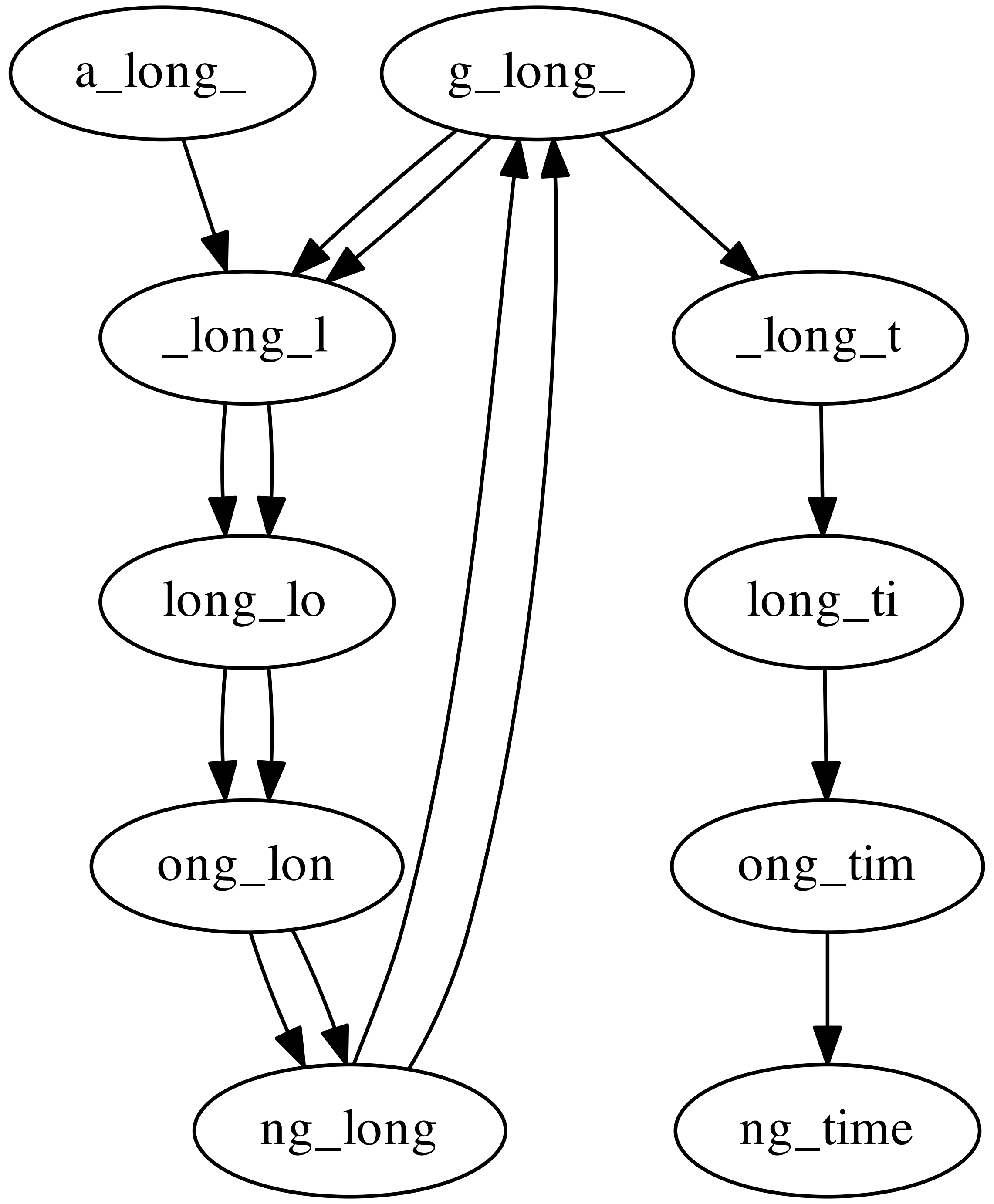
Exact number of k-mers: $N - n(k-1)$ $O(N)$

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

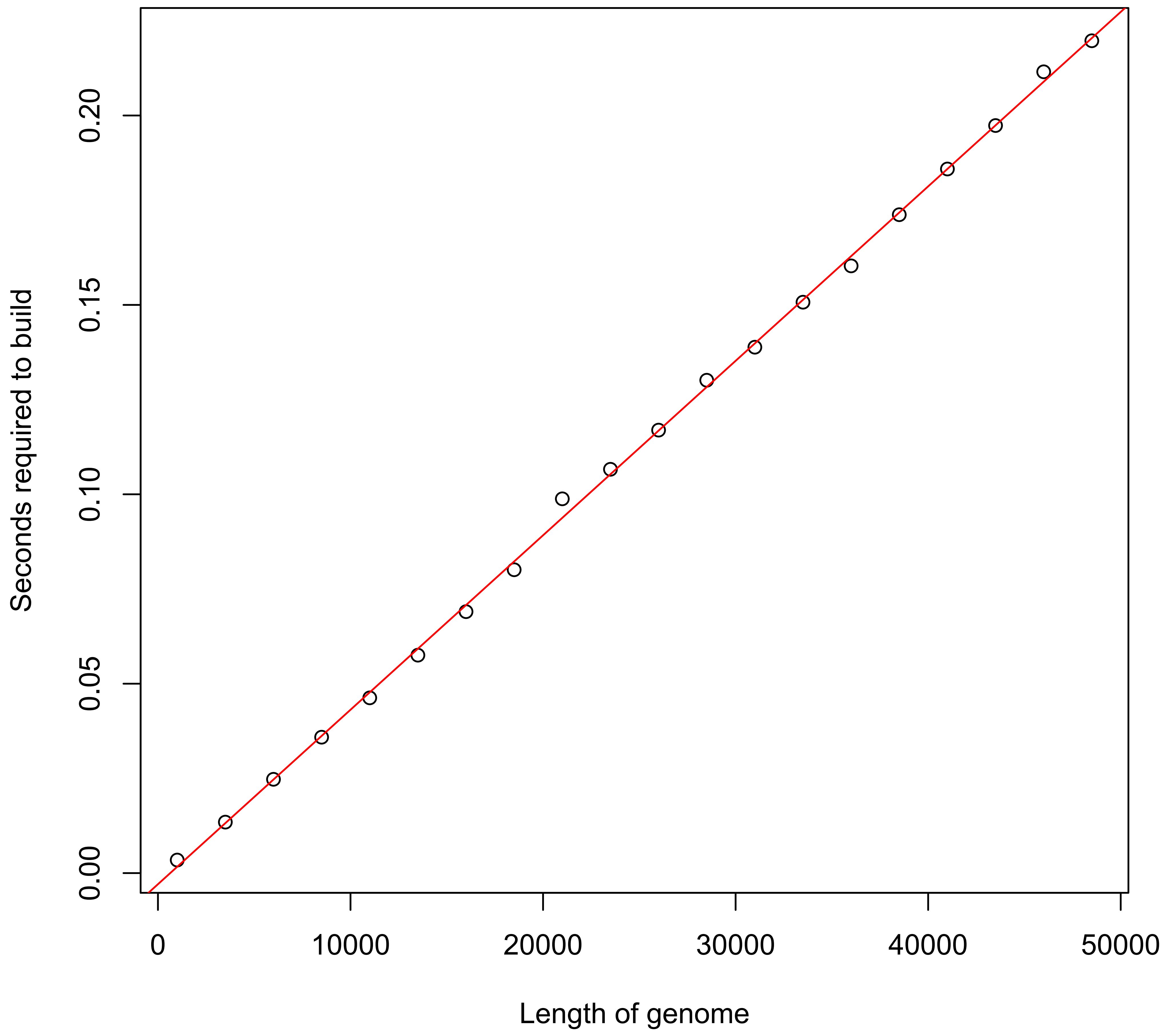
Querying / adding a key is O(1) expected work

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

De Bruijn graph

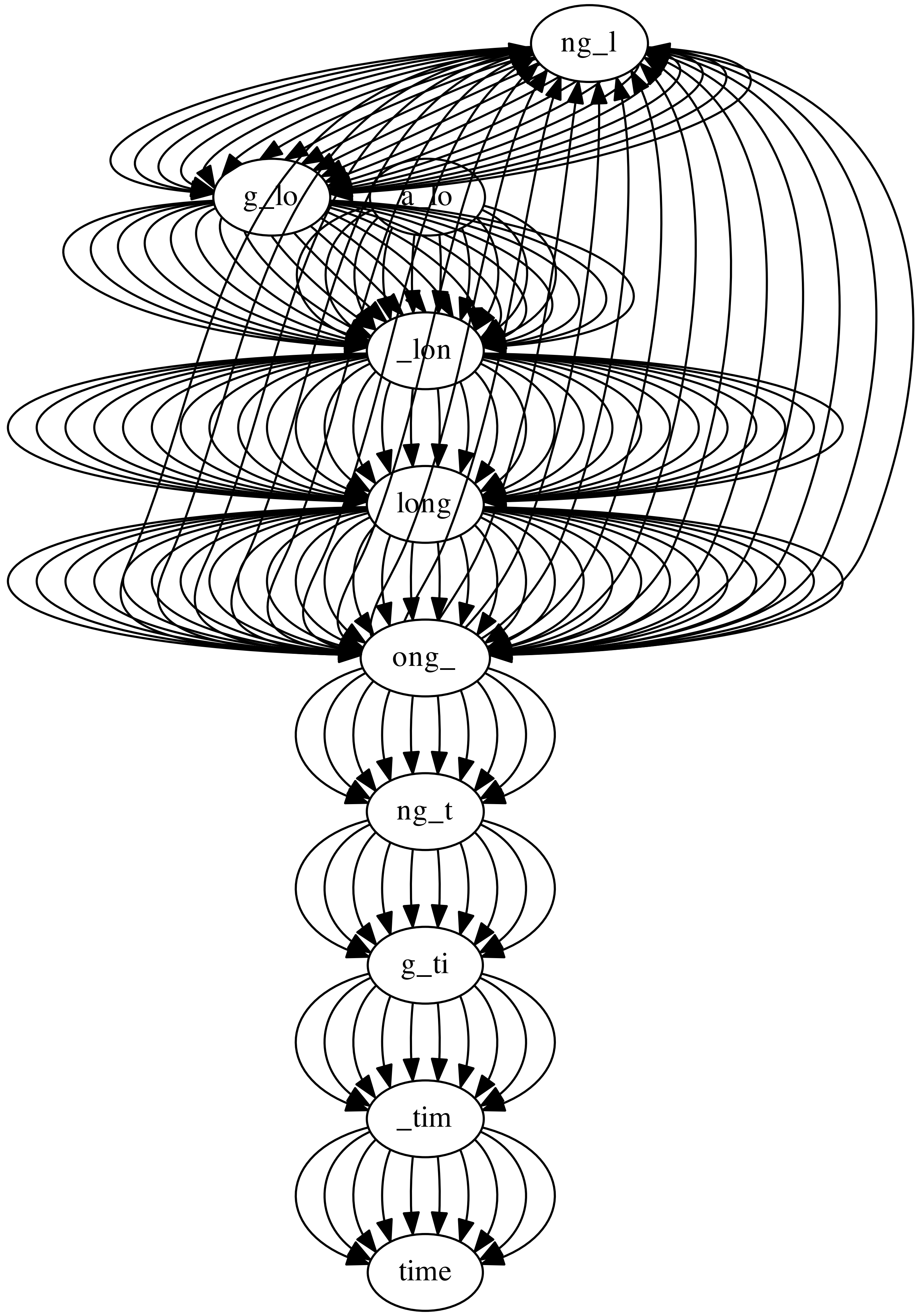
Timed De Bruijn graph construction applied to progressively longer prefixes of lambda phage genome, $k = 14$

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

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

```
                  CTAGGCCCTCAATTTTT
                CTCTAGGCCCTCAATTTTT
              GGCTCTAGGCCCTCATTTTTT
           CTCGGCTCTAGCCCCTCATTTT
        TATCTCGACTCTAGGCCCTCA
        TATCTCGACTCTAGGCC
    TCTATATCTCGGCTCTAGG
GGCGTCTATATCTCG
GGCGTCGATATCT
GGCGTCTATATCT
GGCGTCTATATCTCGGCTCTAGGCCCTCATTTTTT
```

177 nucleotides

35 nucleotides

Average coverage = 177 / 35 ≈ 7x

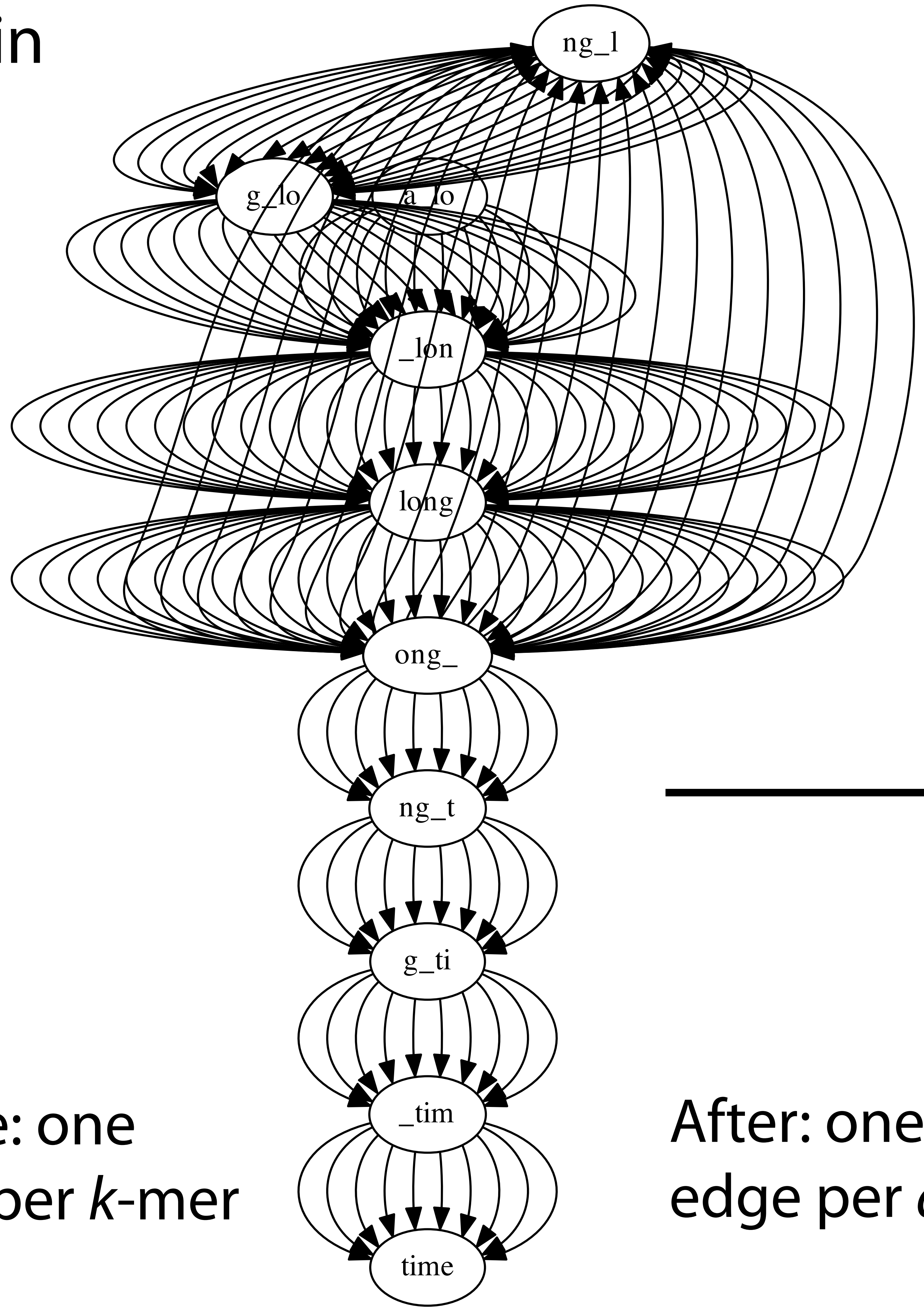
De Bruijn graph

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

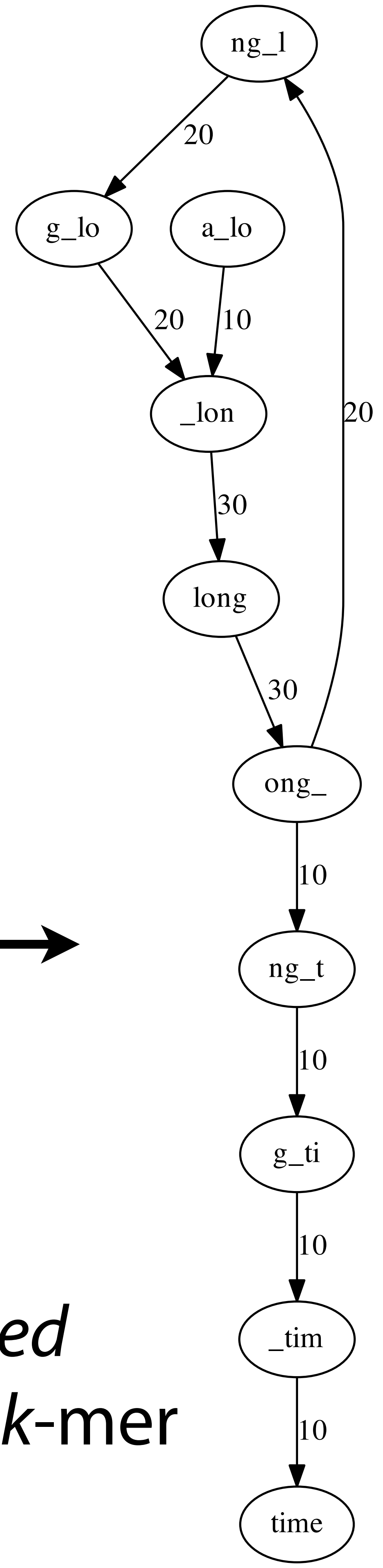
Same edge might appear in dozens of copies; let's use edge *weights* instead

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

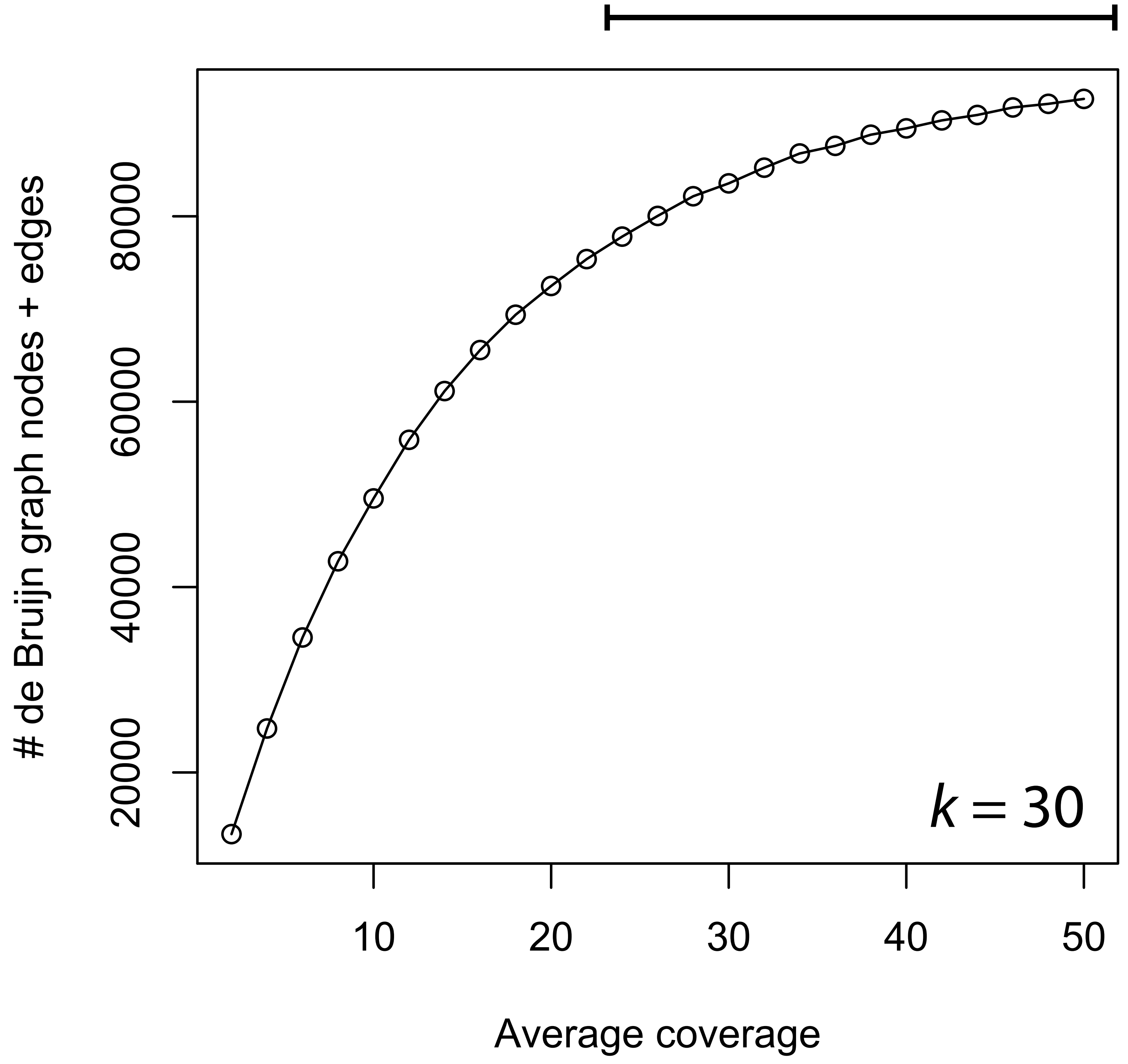
With high average coverage, O(*G*) size bound is advantageous

Genome = lambda phage (~ 48.5 K nt)

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

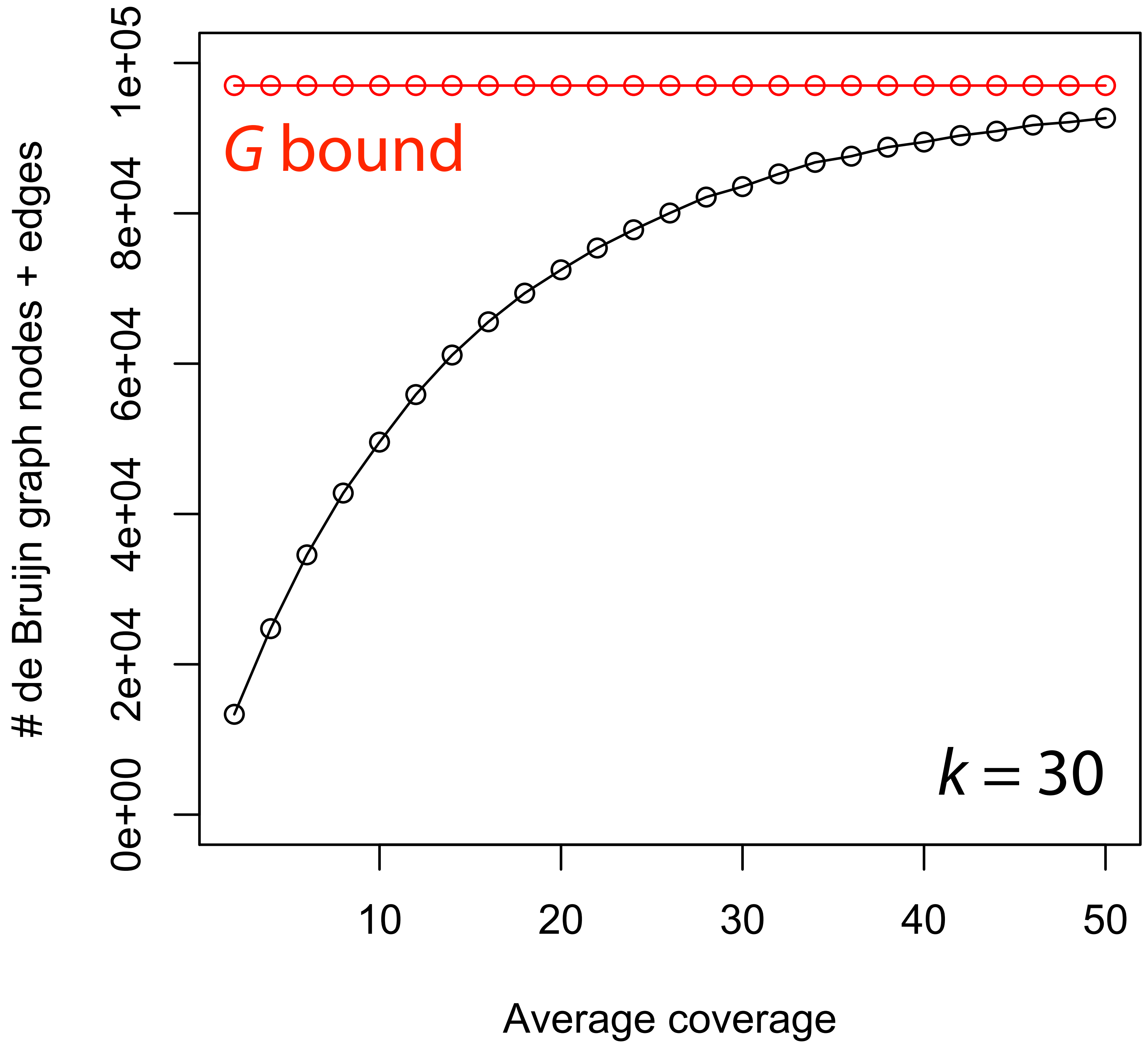
Build De Bruijn graph and total the # of nodes and edges (y axis)

Size of De Bruijn graph grows sublinearly when average coverage is high



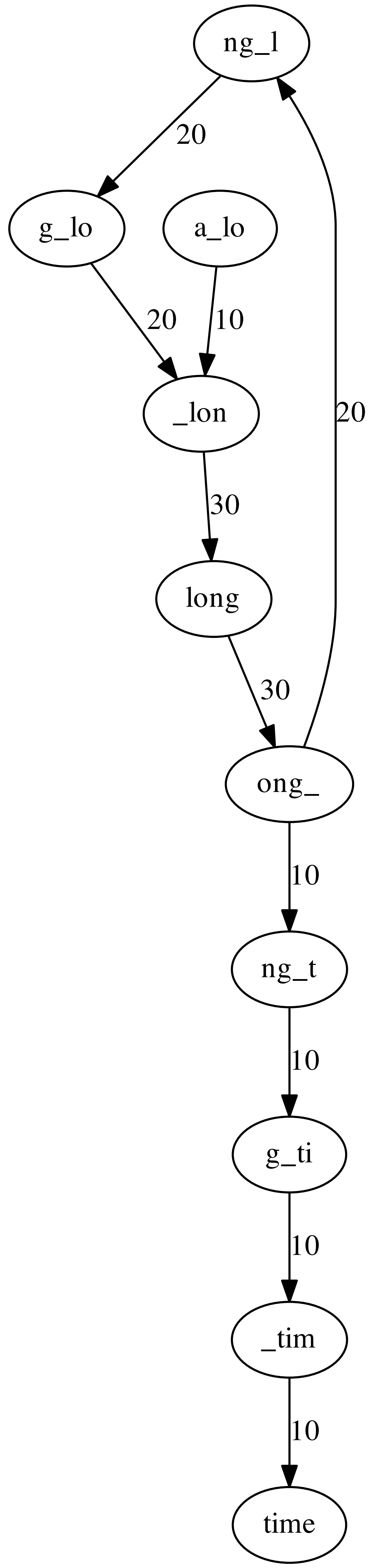
Error correction

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



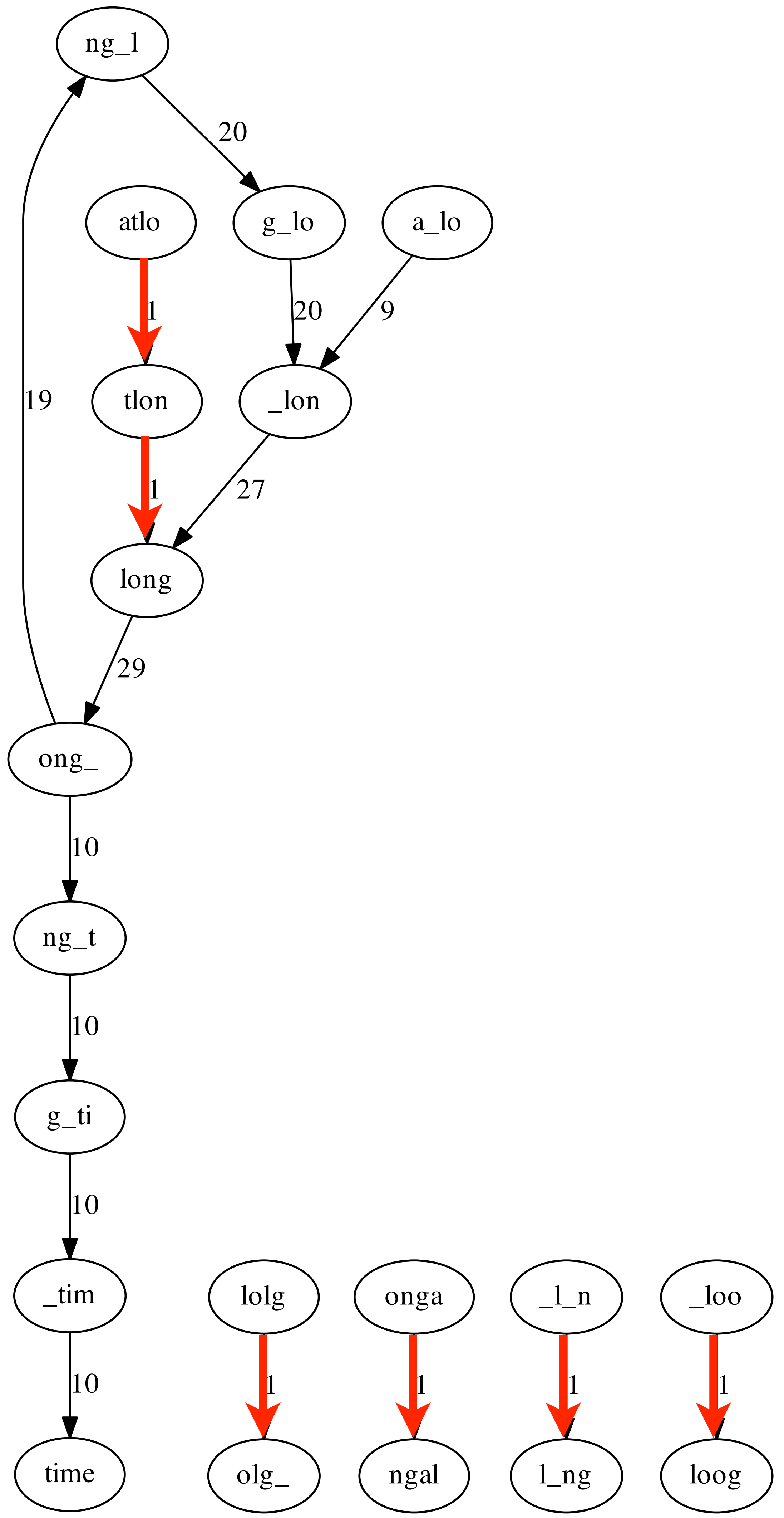
What about data with sequencing errors?

Error correction



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

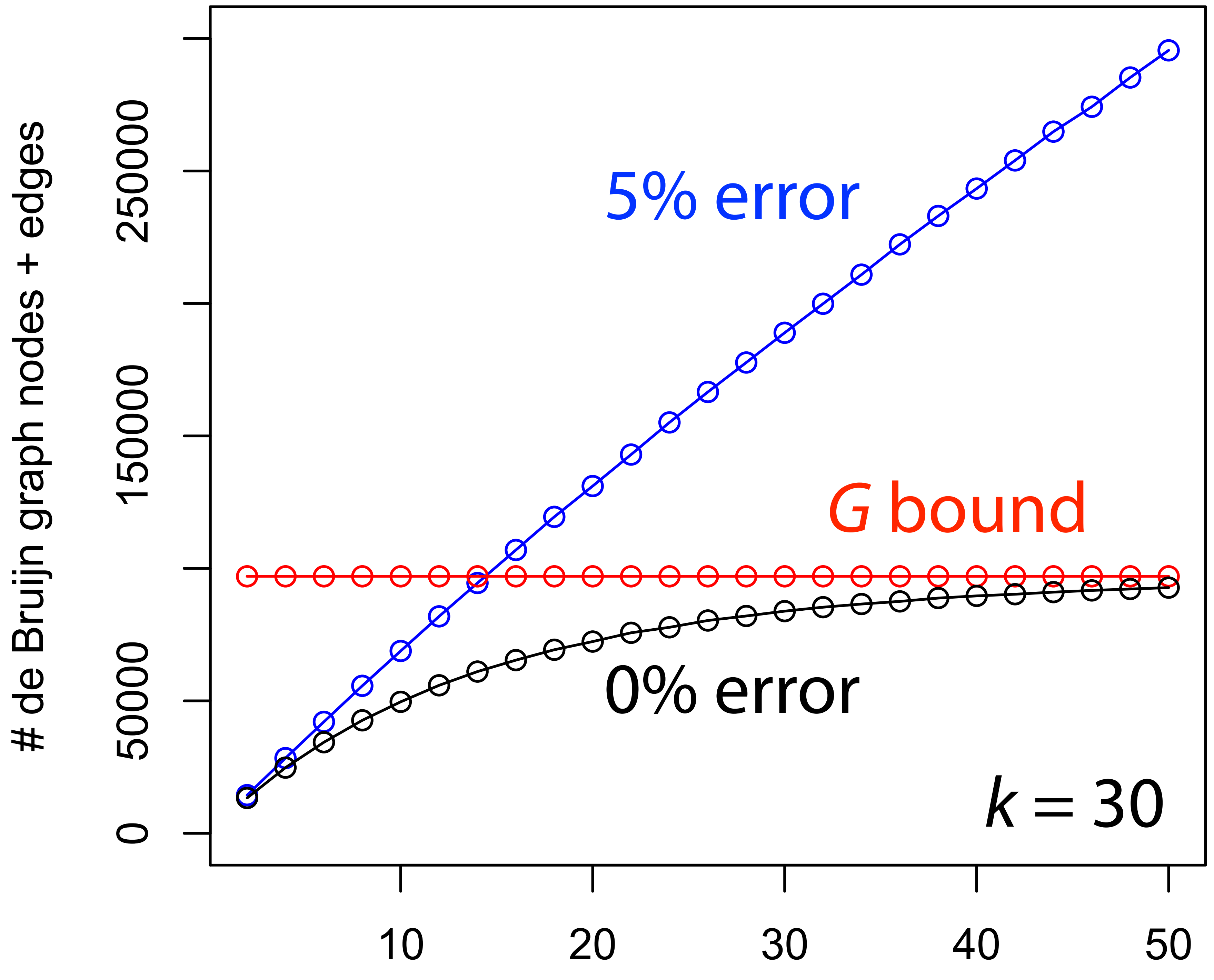
6 errors result in 10 new nodes and 6 new weighted edges, all with weight 1



*

Error correction

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



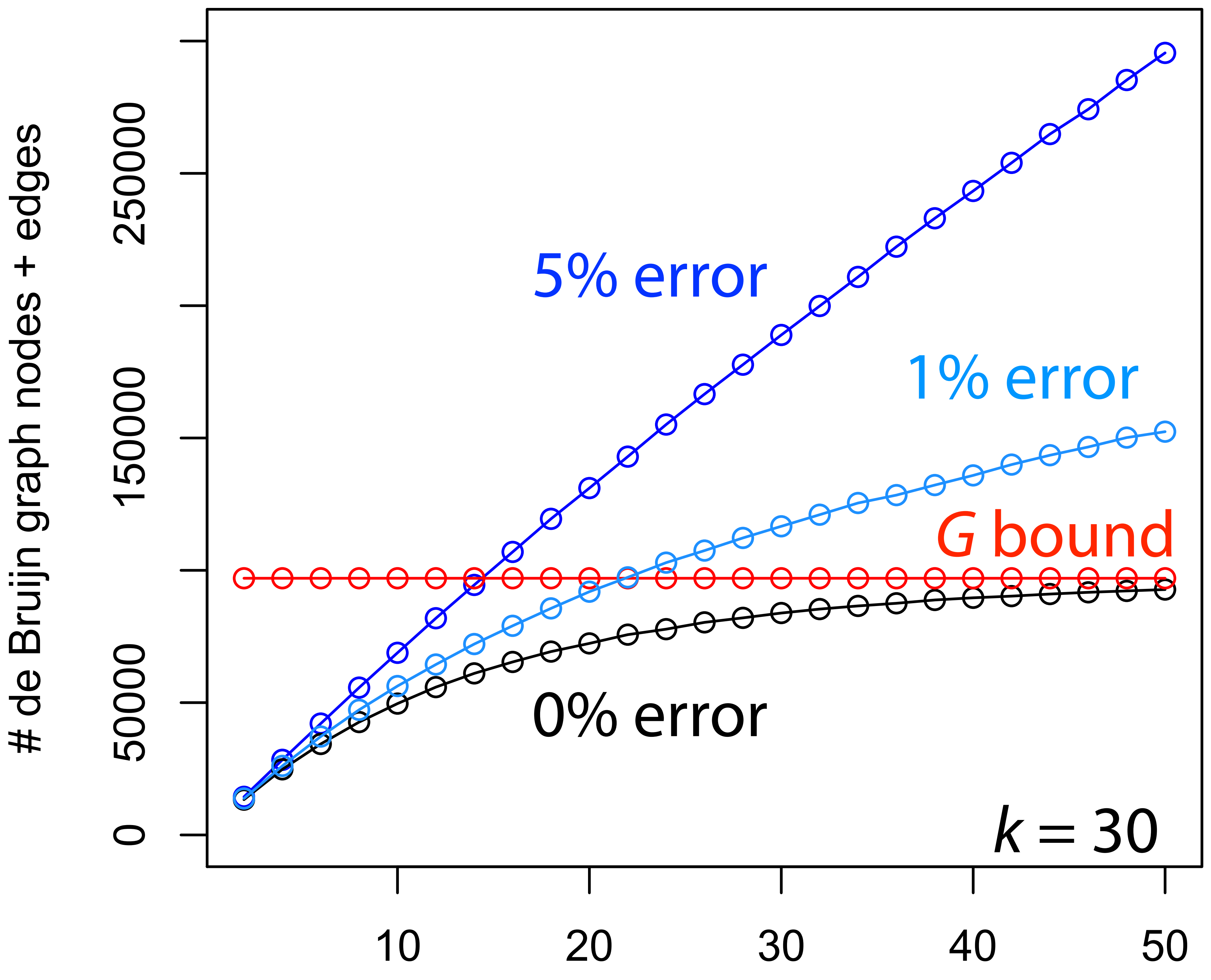
Average coverage
Lambda phage genome

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



Average coverage

Lambda phage genome

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?

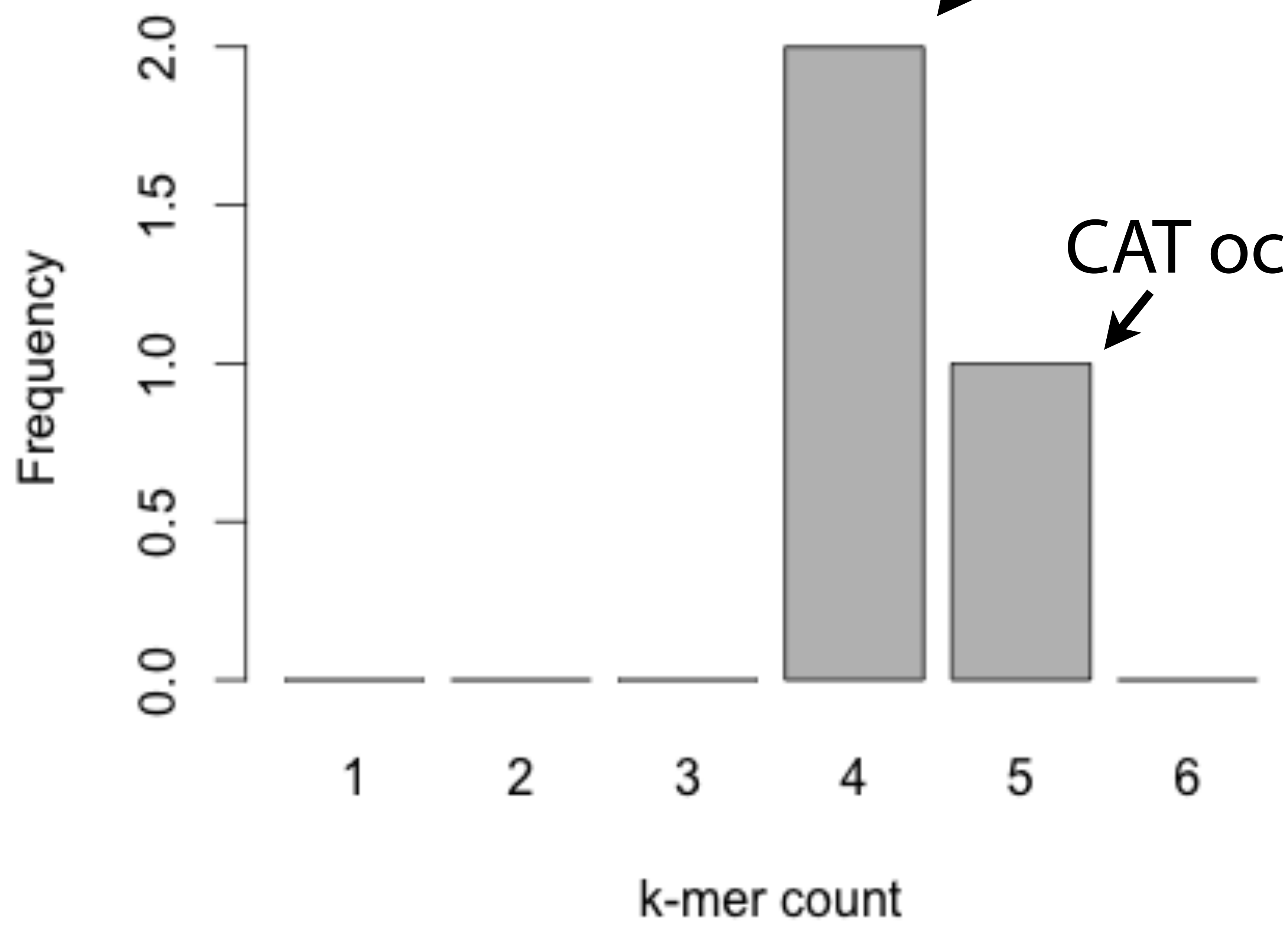
- Analogy: design a spell checker for a language you've never seen before. How do you come up with suggestions?

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:



ATC and TCA occur 4 times

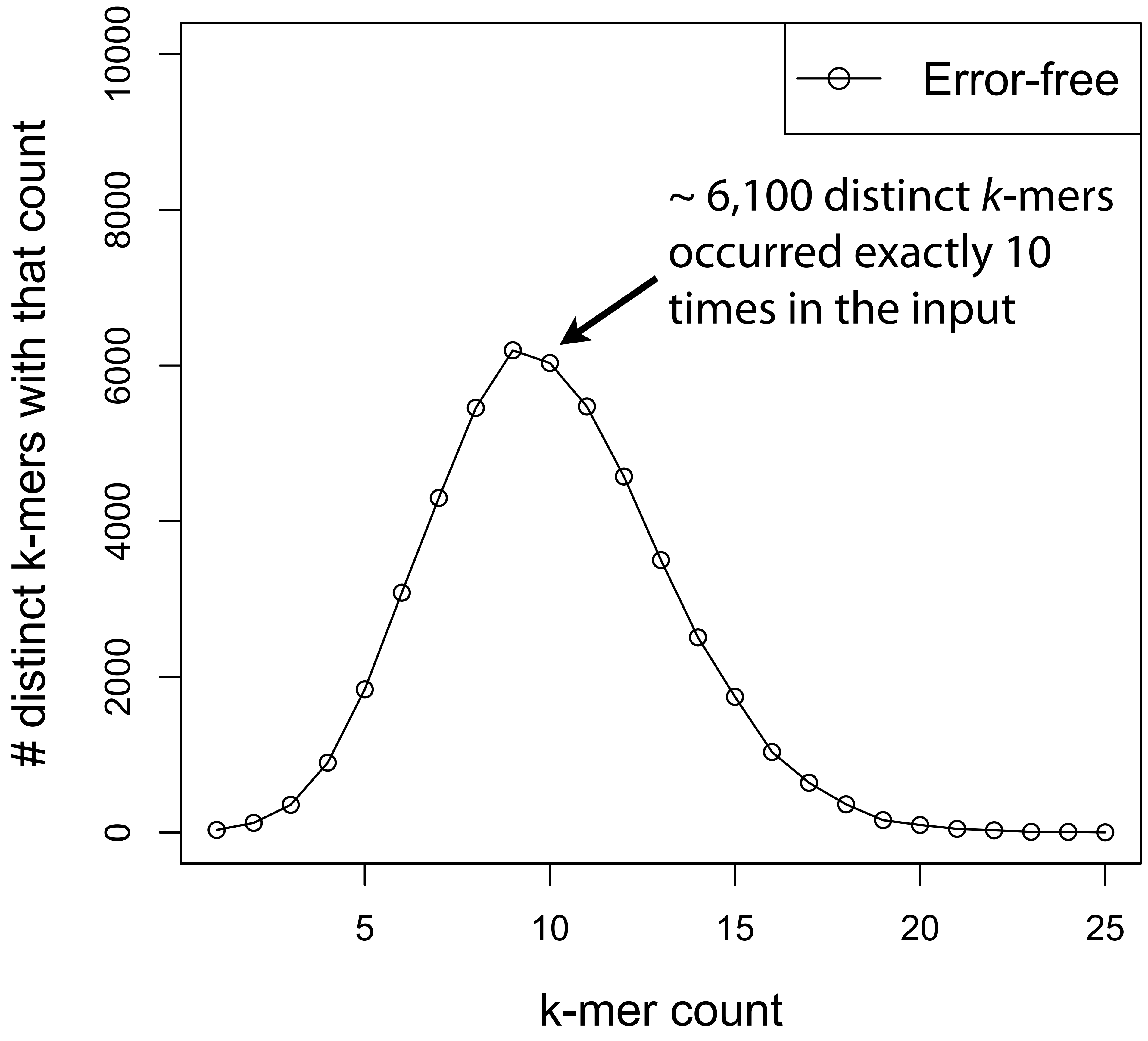
CAT occurs 5 times

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

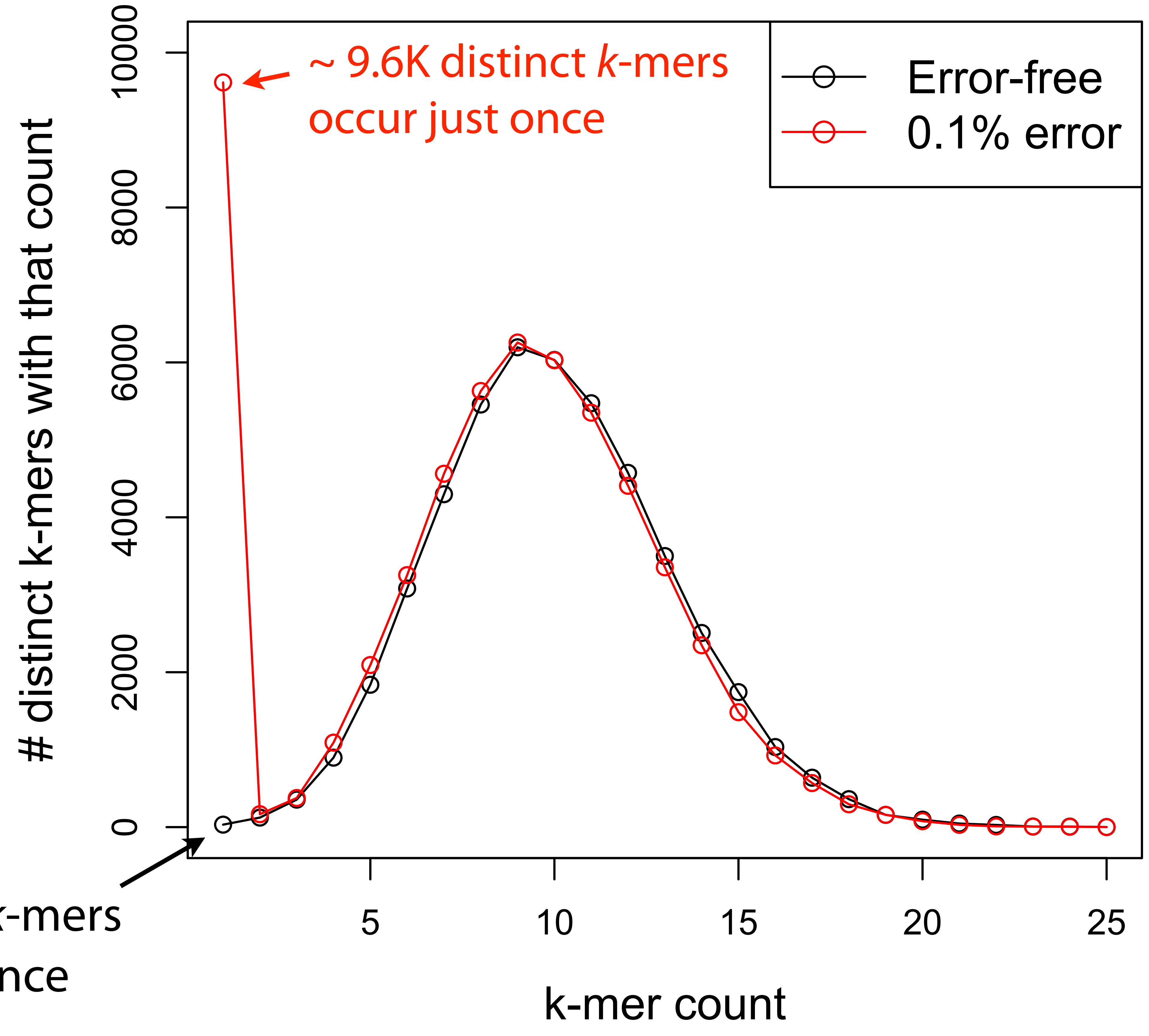


~ 6,100 distinct *k*-mers occurred exactly 10 times in the input

*

Error correction

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



32 distinct *k*-mers occur just once

*

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)

8-mers:

```
GCGTATTA: 8
 CGTATTAC: 8
  GTATTACG: 9
   TATTACGC: 9
    ATTACGCG: 9
     TTACGCGT: 12
      TACGCGTC: 9
       ACGCGTCT: 8
        CGCGTCTG: 10
         GCGTCTGG: 10
          CGTCTGGC: 11
           GTCTGGCC: 9
            TCTGGCCT: 8
```

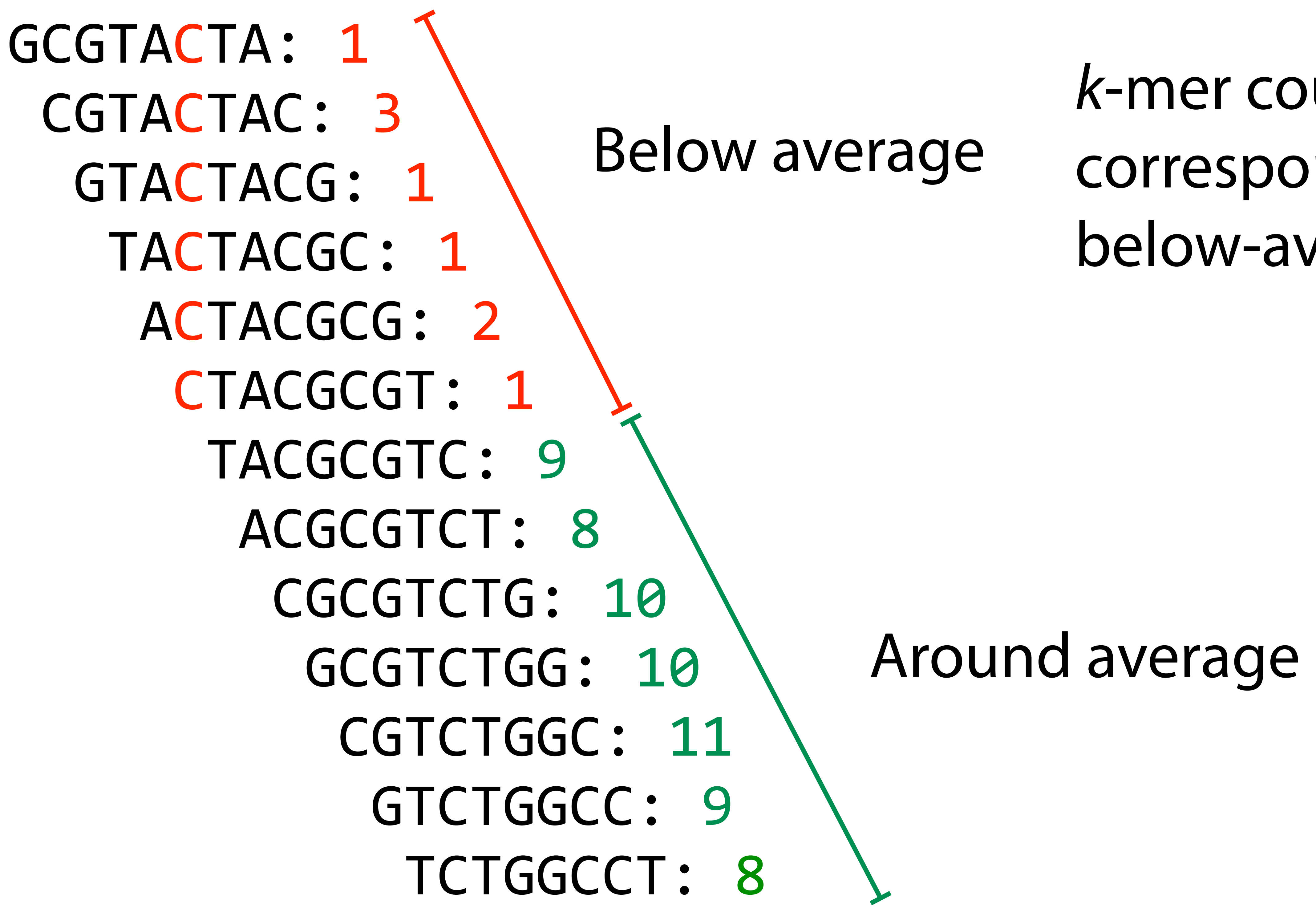
times each 8-mer occurs in the dataset. "*k*-mer count profile"

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

Error correction

Suppose there's an error

Read: GCGTACTACGCGTCTGGCCT



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

Below average

Around average

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

*

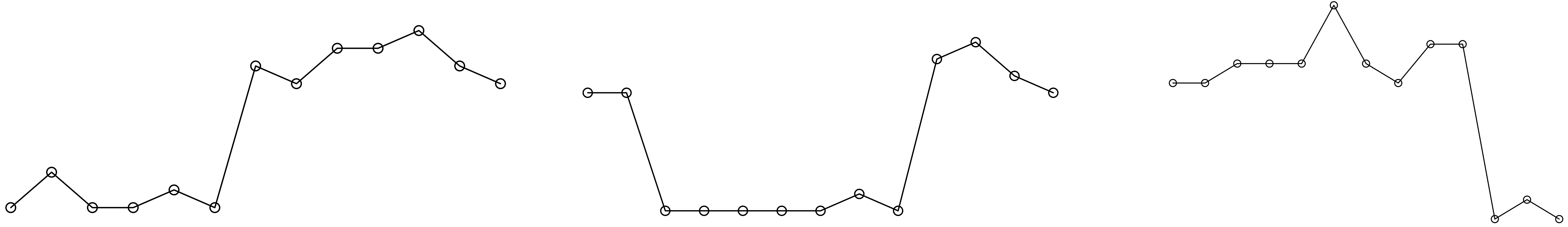
Error correction

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
```

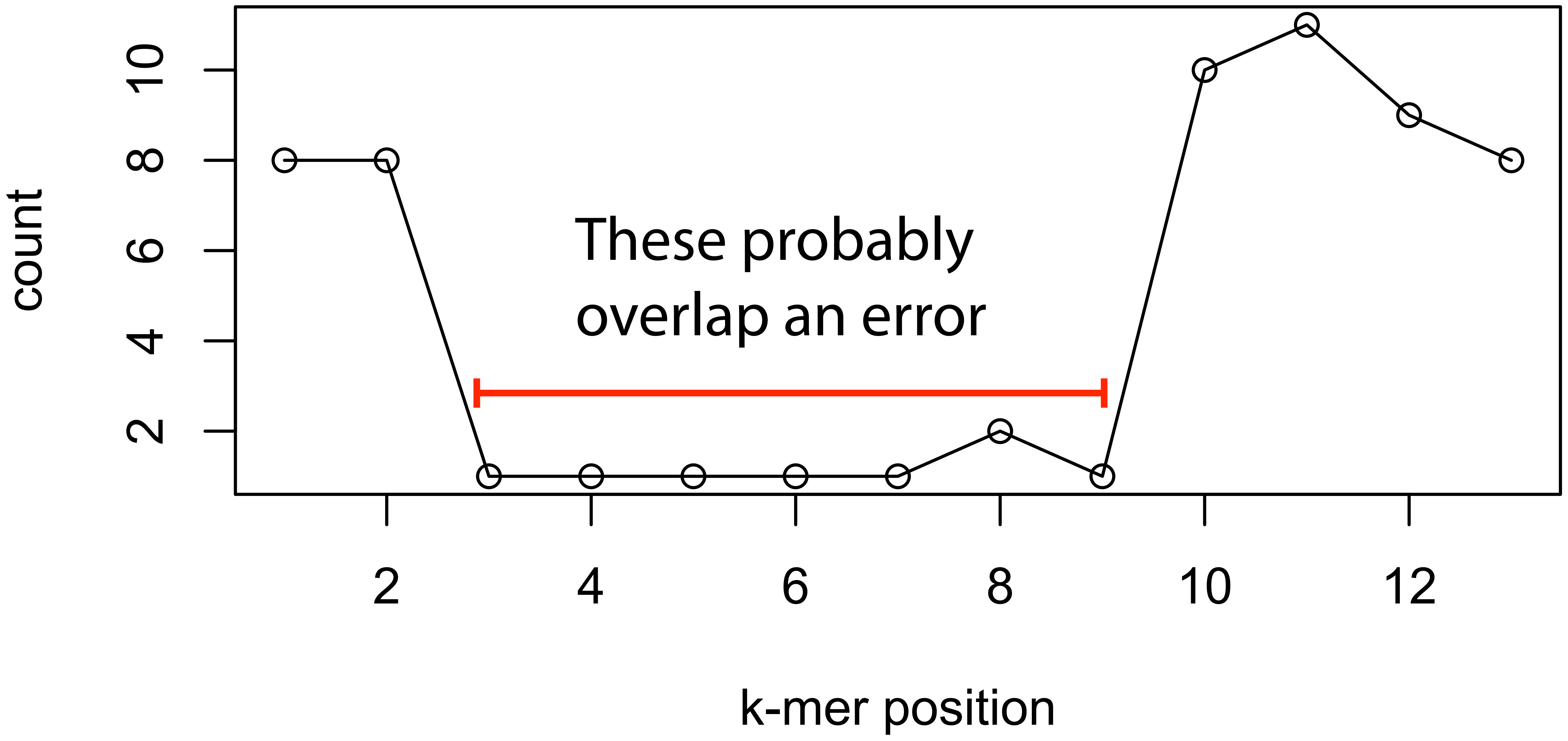
```
GCGTATTACGCGTCTGGTCT
GCGTATTA: 8
 CGTATTAC: 8
  GTATTACG: 9
   TATTACGC: 9
    ATTACGCG: 9
     TTACGCGT: 12
      TACGCGTC: 9
       ACGCGTCT: 8
        CGCGTCTG: 10
         GCGTCTGG: 10
          CGTCTGGT: 1
           GTCTGGTC: 2
            TCTGGTCT: 1
```



*

Error correction

k-mer count profile indicates where errors are

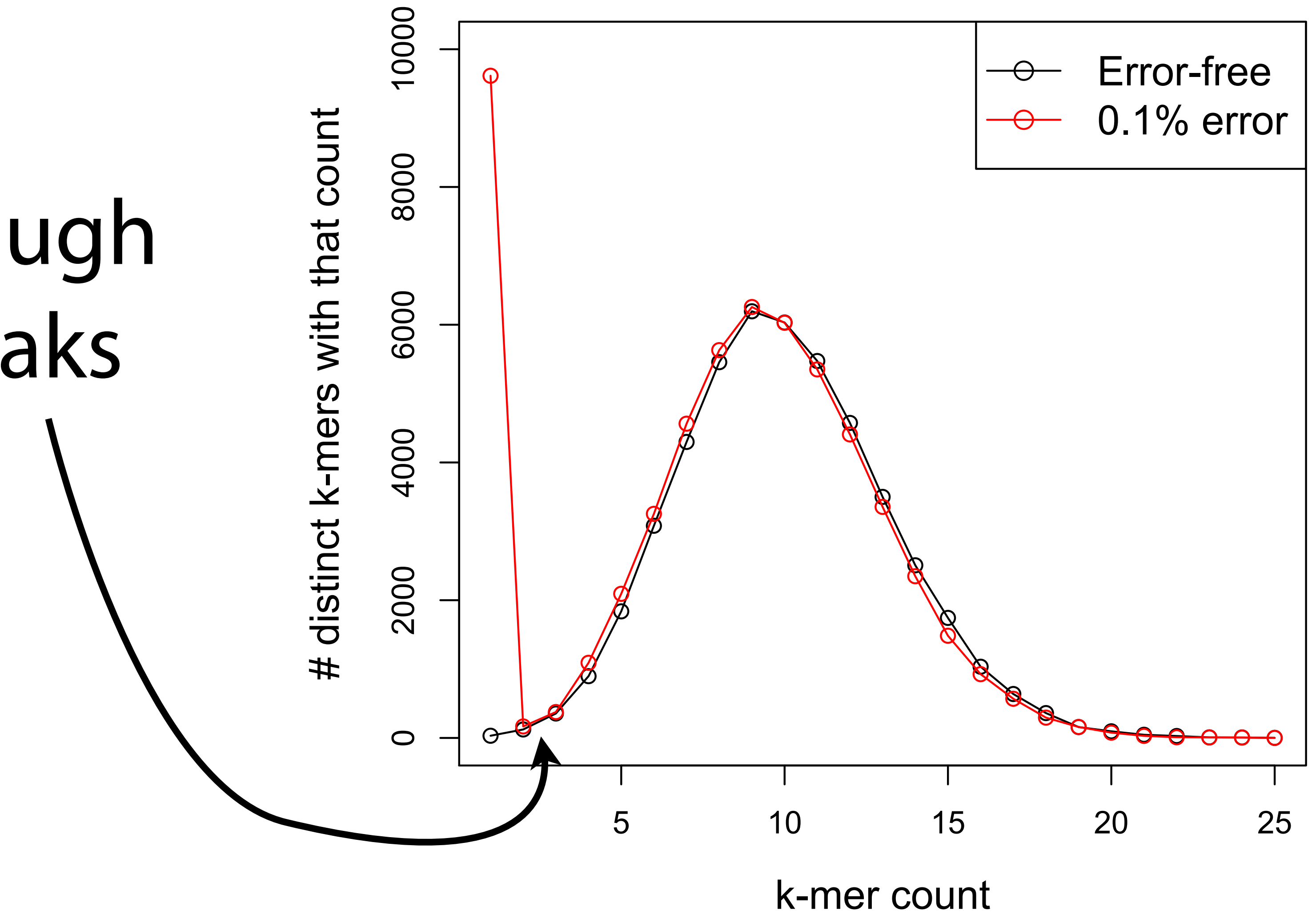


Error correction

Simple algorithm: given a count threshold *t*:

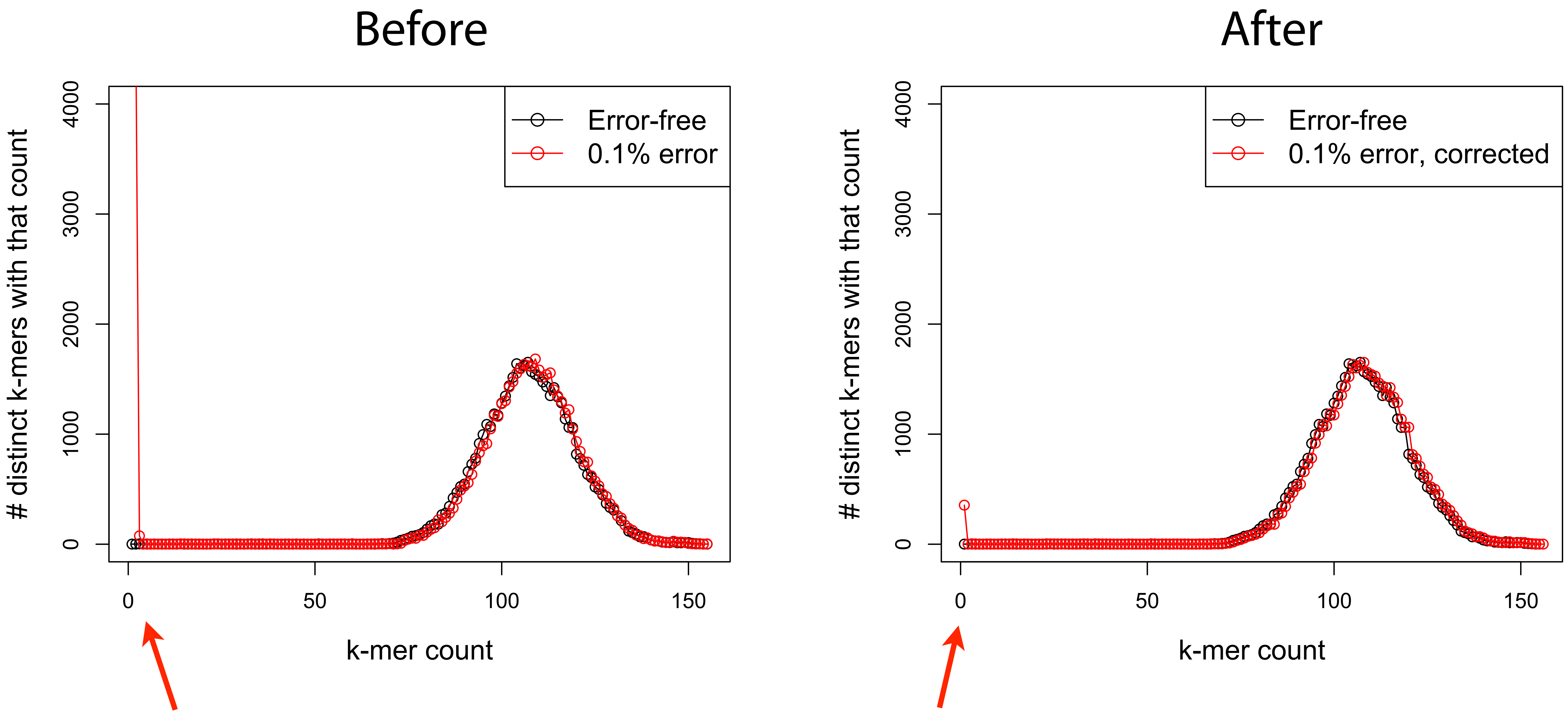
```
For each read:
    For each k-mer:
        If $k$-mer count < $t$:
            Examine $k$-mer's neighbors within certain Hamming/edit distance.
            If neighbor has count ≥ $t$, replace old $k$-mer with neighbor.
```

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

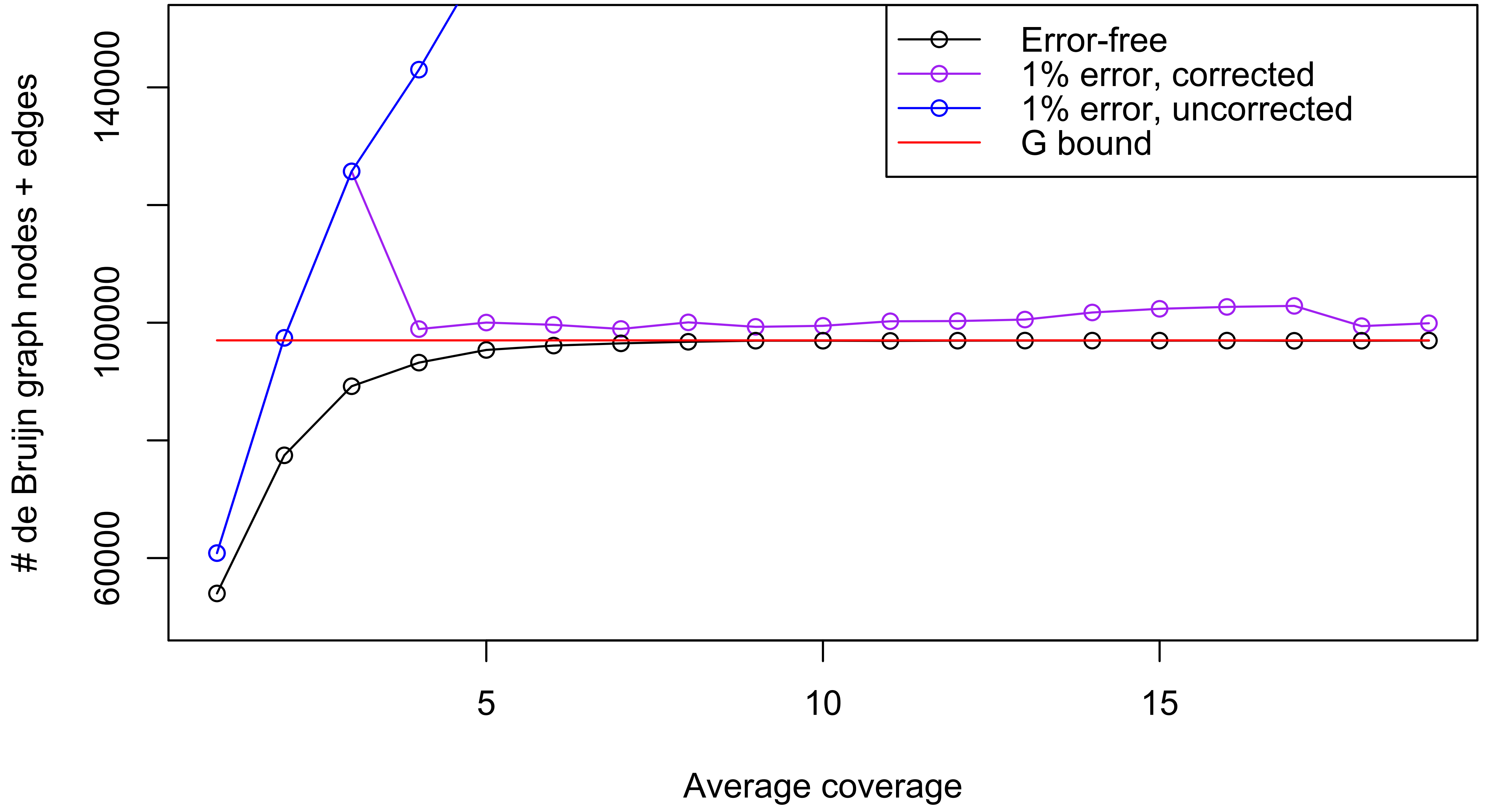


From 194K k-mers occurring exactly once to just 355

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