Algorithms in Bioinformatics

Graph Algorithms
(partially based on Langmead notes)

Recap

- HW still coming
- Today: graphs

Graphs (again)

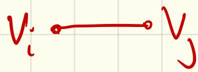
Used to model everything!

$G = (V, E)$ $\quad |V| = n, \ |E| = m$

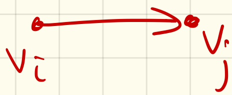
V: vertices $= \{v_1, \dots, v_n\}$

E: edges (directed or undirected)

$= \{ \{v_i, v_j\}, \dots \}$

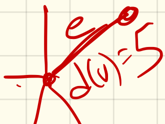


(v_i, v_j)



Useful fact: degree-sum formula

undirected:



$$\sum_v d(v) = 2\,\overset{m}{\overset{\shortparallel}{|E|}}$$

degree, # of edges adjacent to v

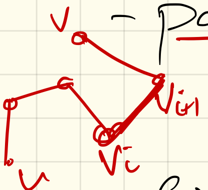
If directed:

$$\sum_v \text{indegree}(v) = \sum_v \text{outdegree}(v)$$

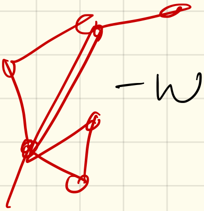
$$= |E|$$

Dfns:

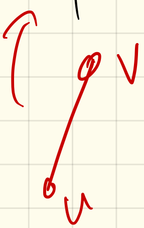
- connected: For every pair u, v of vertices, there is u-v path
- connected components: maximal connected subgraphs

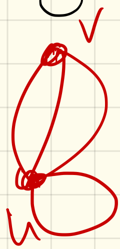


- path: sequence of vertices $v_1, v_2, v_3, \dots v_k$ such that $\forall i, \{v_i v_{i+1}\} \in E$ (no repeats)
- cycle: "path" where $v_1 = v_k$



- walk: allows repeats of edges or vertices
- simple (vs. multigraph)





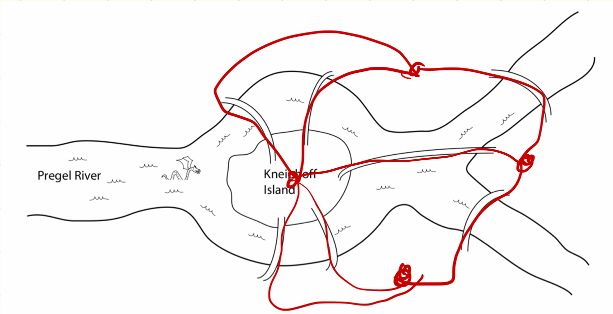
- circuit: cycle, but allows repetition of vertices

First problem: Königsberg bridges

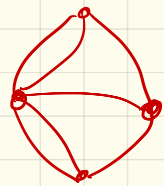
Bridge Obsession Problem:
Find a tour through a city (located on n islands connected by m bridges) that starts on one of the islands, visits every bridge exactly once, and returns to the originating island.

Input: A map of the city with n islands and m bridges.

Output: A tour through the city that visits every bridge exactly once and returns to the starting island.



Graph



This becomes:

Eulerian ~~Cycle~~ Circuit Problem:
Find a ~~cycle~~ circuit in a graph that visits every edge exactly once.

Input: A graph G.

Output: A ~~cycle~~ in G that visits every edge exactly once.

How to solve?

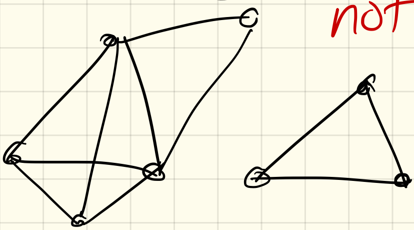
Breaking it down:

What is a necessary condition?

↳ all vertices must have even degree, & ≠0

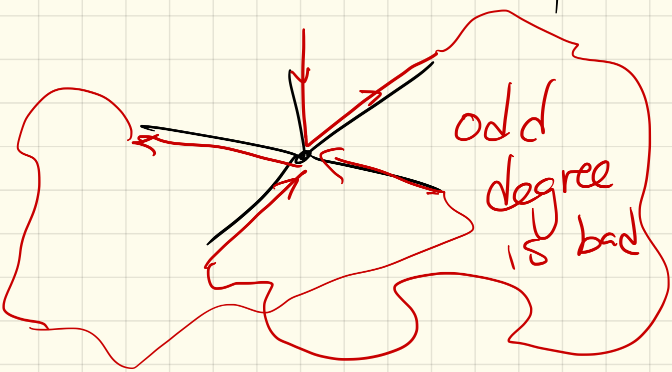
Obvious one: can we tour this graph?

not connected!



Now:

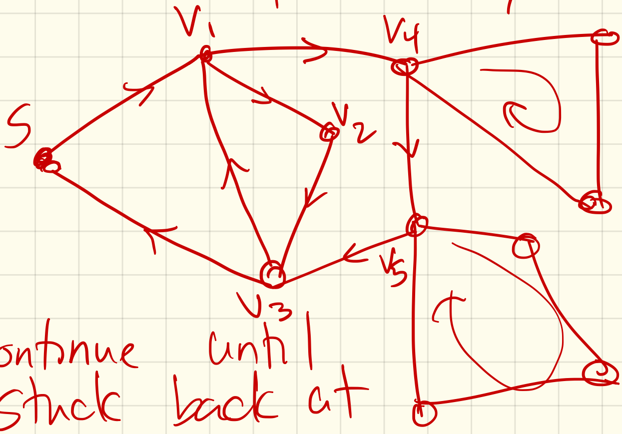
Think about a single vertex: how would a tour proceed?



Is this sufficient?

<u>Yes</u>: Consider a graph w/ all even degrees, & build an Euler tour:

Start at a vertex & walk — pick any edge



Continue until stuck back at S

$S - V_1 - V_2 - V_3 - V_1 - V_4 - V_5 - V_3 - S$

recurse & paste in sub-circuit

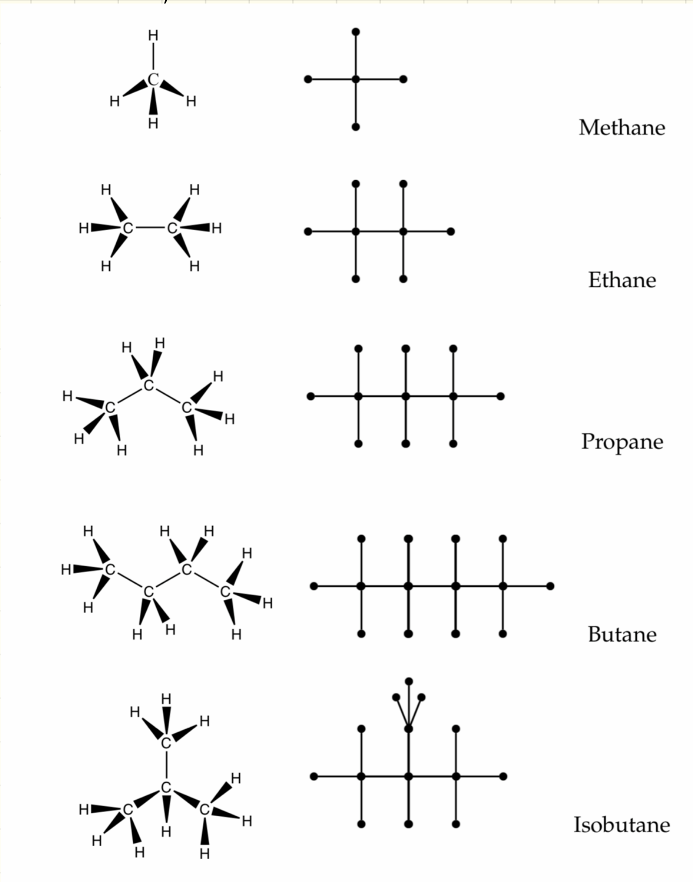
Algorithm:

```
# circuit is a global array
   find_euler_circuit
     circuitpos = 0
     find_circuit(node 1)

# nextnode and visited is a local array
# the path will be found in reverse order
  find_circuit(node i)

    if node i has no neighbors then
      circuit(circuitpos) = node i
      circuitpos = circuitpos + 1
    else
      while (node i has neighbors)
          pick a random neighbor node j of node i
          delete_edges (node j, node i)
          find_circuit (node j)
      circuit(circuitpos) = node i
      circuitpos = circuitpos + 1
```

Runtime: $O(m+n)$

while (V has pos degree)
    visit edges

Next problem: Cayley, studying hydrocarbons



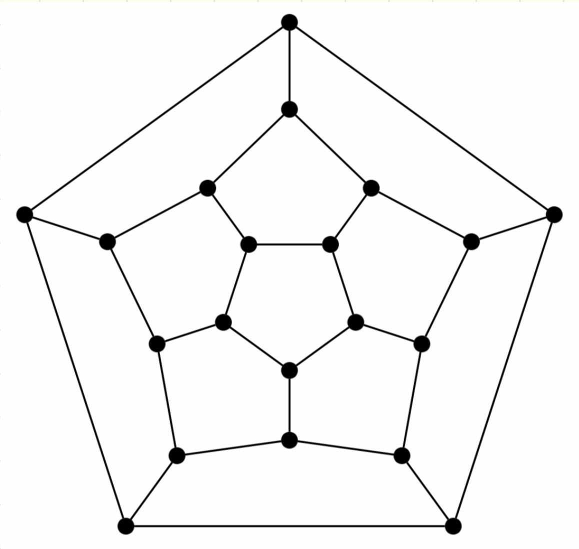
Examples of trees:

Finally, Hamilton created a game:
Visit every vertex in a graph exactly once

Hamiltonian Cycle Problem:
Find a cycle in a graph that visits every vertex exactly once.

Input: A graph G.

Output: A cycle in G that visits every vertex exactly once (if such a cycle exists).



Note: This one is hard.

Weighted graphs:

We've actually talked about these in the last few chapters.

Each edge gets a weight:

Last chapter or 2, we hunted for longest paths.

Can also reverse this:

Shortest Path Problem:
Given a weighted graph and two vertices, find the shortest distance between them.

> **Input:** A weighted graph, $G = (V, E, w)$, and two distinguished vertices s and t.
>
> **Output:** The shortest path between s and t in graph G.

On to some biology:

Back to assembly!

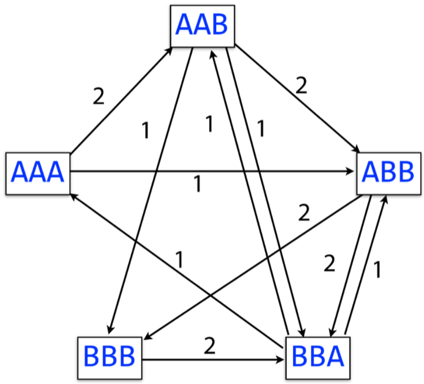
Last time (3 weeks ago) we did the greedy graph algorithm.

Greedy-SCS: in each round, merge pair of strings with maximal overlap. Stop when there's 1 string left. l = minimum overlap.

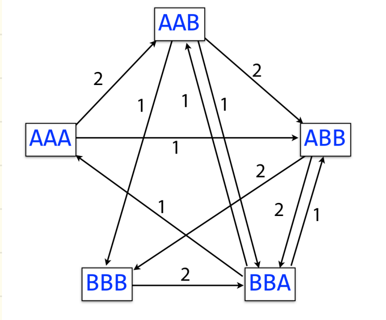
Algorithm in action ($l = 1$):

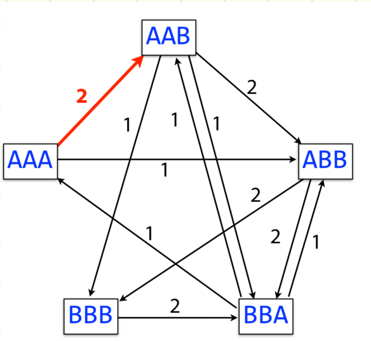
Input strings

AAA AAB ABB BBB BBA

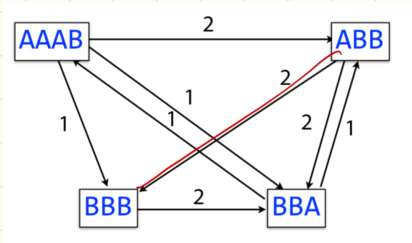


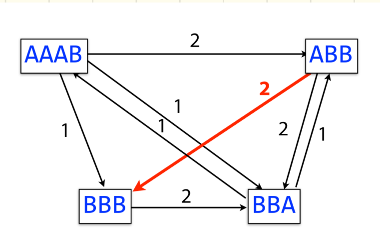
In action:



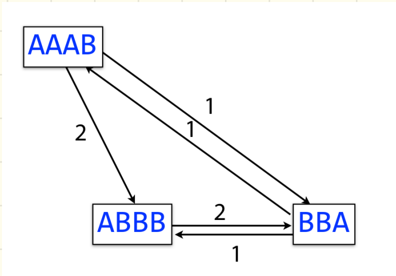


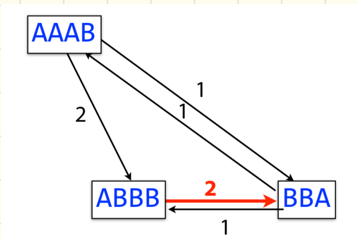
collapse →



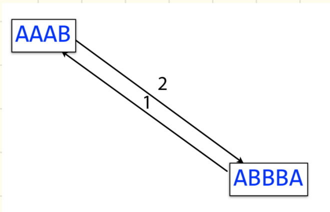


collapse →





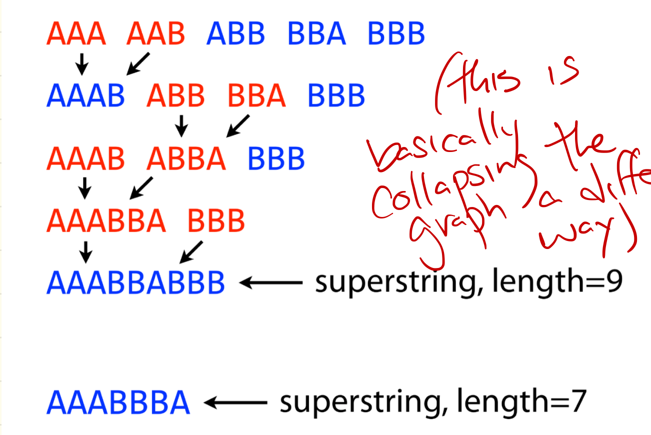
collapse →



final:

AAABBBA ← superstring, length=7

Problem: Greed (usually) doesn't win!



(this is basically collapsing the graph a different way)

Approximation

However, this does give a ~2.5-approximation

length of greedy ≤ ~2.5(length of OPT)

In particular, known issue

Greedy-SCS assembling all substrings of length 6 from:
a_long_long_long_time. $l = 3$.

6 characters

```
ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long g_time ng_tim
ng_time ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long
ng_time g_long_ ng_lon a_long long_l ong_ti ong_lo long_t
ng_time long_ti g_long_ ng_lon a_long long_l ong_lo
ng_time ong_lon long_ti g_long_ a_long long_l
ong_lon long_time g_long_ a_long long_l
long_lon long_time g_long_ a_long
long_lon g_long_time a_long
long_long_time a_long
a_long_long_time
```

Foiled by repeat!

To fix: longer reads!

length 8

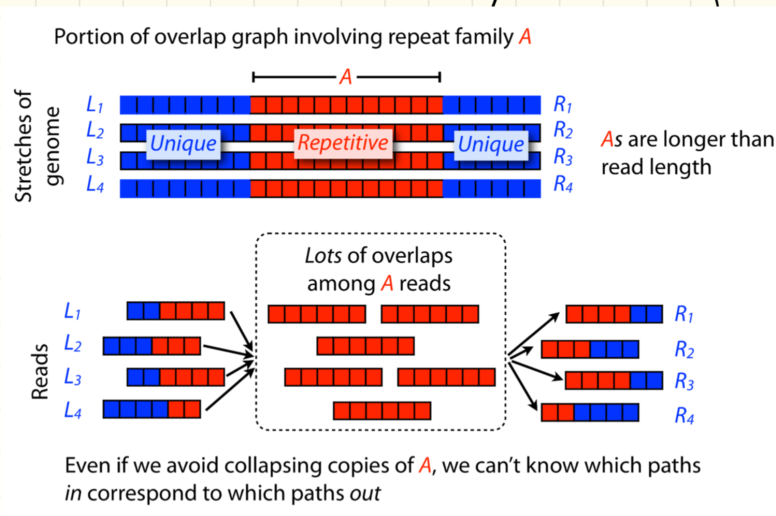
```
long_lon ng_long_ _long_lo g_long_t ong_long g_long_l ong_time a_long_l _long_ti long_tim
long_time long_lon ng_long_ _long_lo g_long_t ong_long g_long_l a_long_l _long_ti
_long_time long_lon ng_long_ _long_lo g_long_t ong_long g_long_l a_long_l
_long_time a_long_lo long_lon ng_long_ g_long_t ong_long g_long_l
_long_time ong_long_ a_long_lo long_lon g_long_t g_long_l
g_long_time ong_long_ a_long_lo long_lon g_long_l
g_long_time ong_long_ a_long_lon g_long_l
g_long_time ong_long_l a_long_lon
g_long_time a_long_long_l
a_long_long_long_time
a_long_long_long_time
```

Repeats

These often foil assembly – certainly SCS, b/c of "shortest"

Need longer reads
↳ catches the repeat

But: algorithms that don't pay attention to repeats will always collapse them

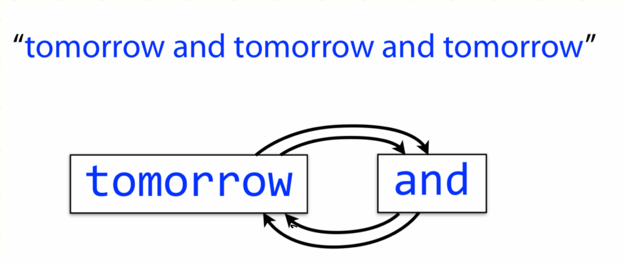


Problem:

Human genome is 50% repetition!

This time: De Bruijn Graph Assembly

Idea: build a different graph

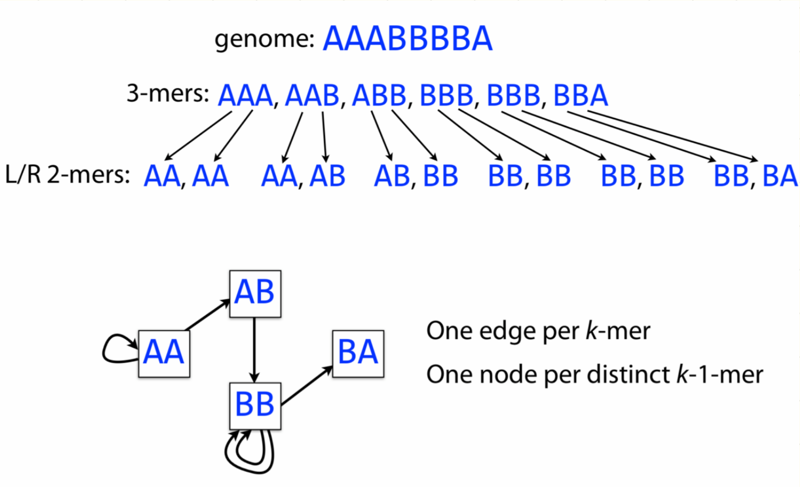


Vertices: "words" (or length k substrings)

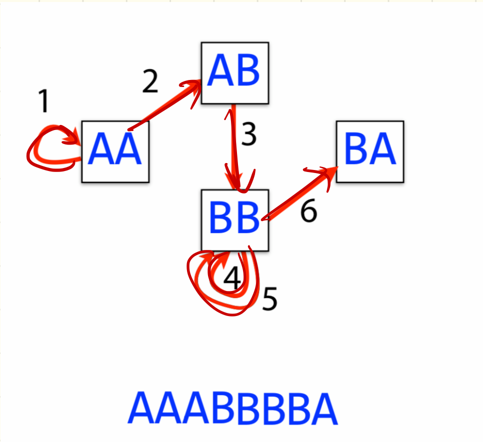
Edges: u → v edge for each time u then v appers in input

Note: • Definitely a multigraph!
• directed, unweighted

A better example:



Key:



AAABBBBA

Note: Path, not a circuit

De Bruijn Graphs:
How to build?

General procedure:

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`

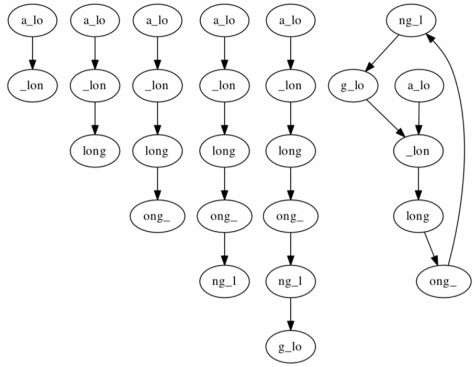
Take each k mer and split into left and right k-1 mers

```
    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

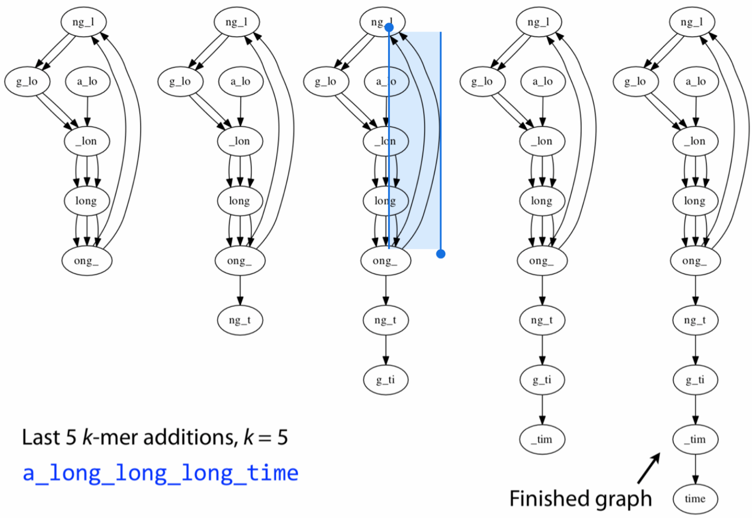
(Obvious problem:)

An example:



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

a_long_long_long_time



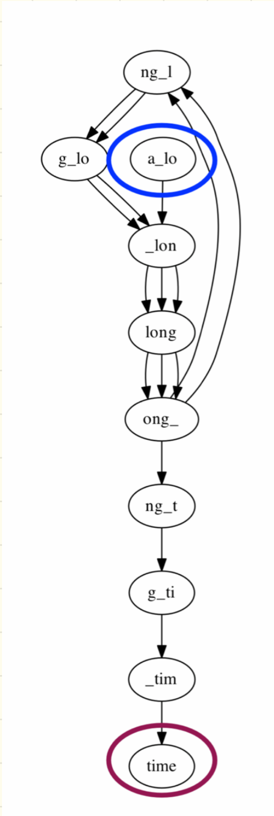
Last 5 k-mer additions, $k = 5$

a_long_long_long_time

Finished graph

Question: Why is this Eulerian?

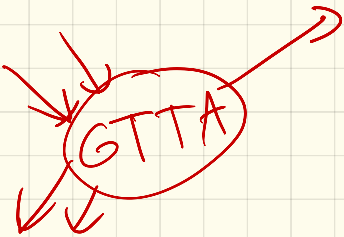
Think about how we built it:



each time a vertex was added (other than 1st + last), had in edge + out edge

2 odd degree edges

start at one, + must get stuck at other



Algorithm: (Python)

```
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

Problems

① Perfect sequencing

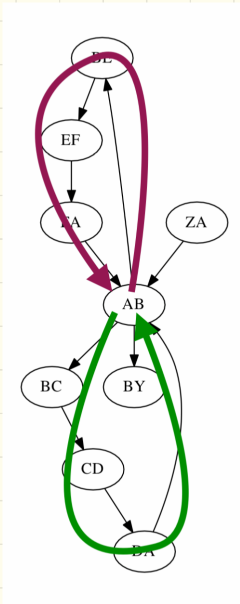
Never (next slide)

② Repeats can still cause issues!

Simple(ish) example of how:

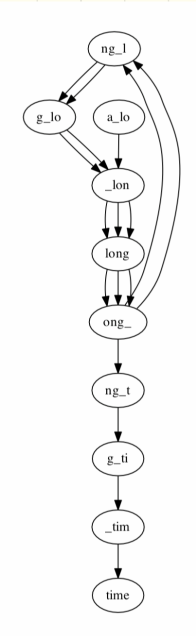
ZA → AB → BE → EF → FA → AB → BC → CD → DA → AB → BY

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

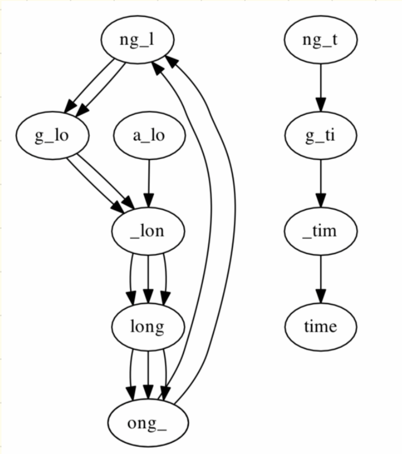


More issues (ie 1 is a big deal!)

Graph for: a-long-long-time, $k=5$:



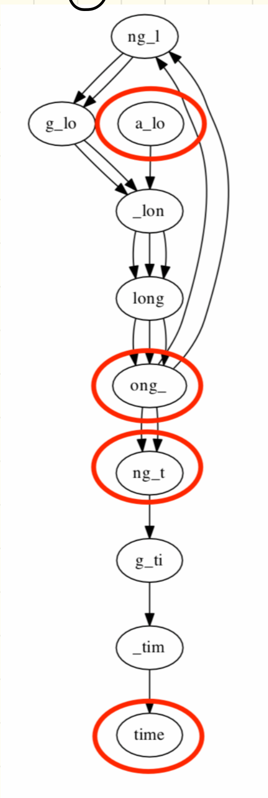
Same, but missing ong-t:



Issue:

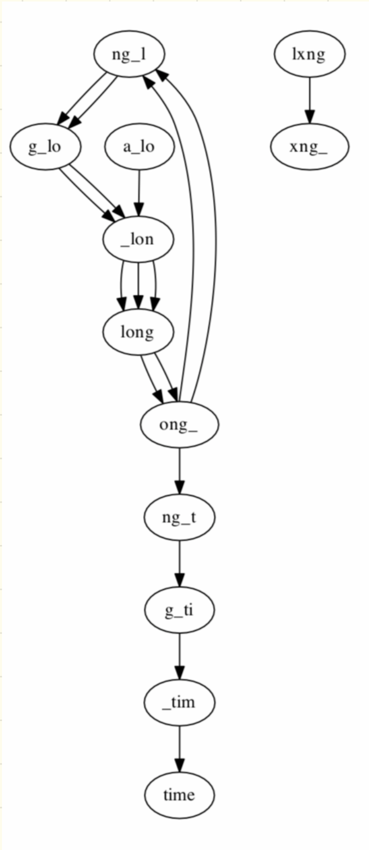
not connected

Same but has extra copy of ong_t



Issue: not Euler

Same, but error: long → lxng



Issue: not connected

Final Conclusions

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.