Comp 555 - BioAlgorithms - Spring 2018

- Problem set #2 is late, but still coming

The Realities of Genome Assembly
From Last Time

What we learned from a related "Minimal Superstring" problem

● Can be constructed by finding a Hamiltonian path of an k-dimensional De Bruijn graph over σ symbols
  ○ Brute-force method is explores all V! paths through V vertices
  ○ Branch-and-Bound method considers only paths composed of edges in the graph
  ○ Finding a Hamiltonian path is an NP-complete problem
  ○ There is no known method that can solve it efficiently as the number of vertices grows

● Can be constructed by finding a Eulerian path of a (k−1)-dimensional De Bruijn graph where k-mers are edges.
  ○ Euler's method finds a path using all edges in \( O(E) \leq O(V^2) \) steps
  ○ Graph must statisfy contraints to be sure that a solution exists

● All but two vertices must be balanced
● The other two must be semi-balanced
Applications to Assembling Genomes

- Extracted DNA is broken into random small fragments
- 100-200 bases are read from one or both ends of the fragment
- Typically, each base of the genome is covered by 10x - 30x fragments
Genome Assembly vs Minimal Superstring

- **Minimal substring problem**
  - Every k-mer is known and used as a vertex, (all $o^k$)
  - Paths, and there may be multiple, are solutions

- **Read fragments**
  - No guarantee that we will see every k-mer
  - Can't disambiguate repeats

```
binary3 = {'000', '001', '010', '011', '100', '101', '110', '111'}

    101 100        111 100
  001 111        001 101
Solution #1: 0001011100  Solution #2: 0001110100
  000 011
    010 110
  011 010
```
Recall our “Toy” example

GACGGCGGCGCACGGCGCAA - Our toy 20 base sequence from 2 lectures ago
GACGG CGCAC
ACGCG GCACG
CGGCG CACGG - The complete set of 16 5-mers
GGCGG ACGGC
GCGGC CGGCG
CGGCG GGCGC
CGGCG GGCGC
GCGCA GGCGA
GGCGA CGCAA

● All $k$-mers is equivalent to $k \times$ coverage, ignoring boundaries
● Four repeated k-mers \{ACGGC, CGGCG, GCGCA, GGCGC\}
First let's add a function to uniquely label repeated k-mers

```python
In [4]: def kmersUnique(seq, k):
    kmers = sorted([seq[i:i+k] for i in range(len(seq)-k+1)])
    for i in range(1, len(kmers)):
        if (kmers[i] == kmers[i-1][0:k]):
            t = kmers[i-1].find('_')
            if (t >= 0):
                n = int(kmers[i-1][t+1:]) + 1
                kmers[i] = kmers[i] + '_' + str(n)
            else:
                kmers[i-1] = kmers[i-1] + '_1'
                kmers[i] = kmers[i] + '_2'
    return kmers

kmers = kmersUnique("GACGCGCGCGACGCGCGCA", 5)
print(kmers)

['AGCGC_1', 'ACGCG_2', 'CACGG', 'CGCAA', 'CGCAG', 'CGGCG_1', 'CGGCG_2', 'CGGCG_3', 'GACGG', 'GCACG', 'GCACA_1', 'GCGCA_2', 'GCACG', 'GCGC_1', 'GCGC_2', 'GCGCA']
```
import itertools

class Graph:
    def __init__(self, vlist):
        self.vertex = list(vlist)
        self.edgeLabel = []

    def addVertex(self, v):
        """ Add a labeled vertex """
        if v not in self.vertex:
            self.vertex.append(v)
            self.edgeLabel.append(v)

    def addEdge(self, v1, v2):
        """ Add a directed edge """
        e = (v1, v2)
        if e not in self.edge:
            self.edge.append(e)

    def degrees(self):
        """ Returns the number of vertices connected to a vertex """
        return len(self.edge)

    def hamiltonianPath(self):
        """ A brute-force method to find a Hamiltonian Path """
        if len(self.vertex) < 2:
            return None
        for i in range(len(self.vertex)):
            visited = [False] * len(self.vertex)
            visited[i] = True
            path = [self.vertex[i]]
            for j in range(i + 1, len(self.vertex)):
                if self.edge[j, i]:
                    visited[j] = True
                    path.append(self.vertex[j])
                    if len(path) == len(self.vertex):
                        return path

    def eulerianPath(self):
        """ Generates an Eulerian Path """
        path = [(src, dst)]
        for src, dst in self.edge:
            if src in path:
                path.append(dst)
            else:
                path.append((src, dst))
        return path

    def render(self):
        """ Renders the graph using NetworkX """
        graph = nx.Graph()
        graph.add_nodes_from(self.vertex)
        graph.add_edges_from(self.edge)
        nx.draw(graph, node_size=100, node_color='blue', alpha=0.5)
        plt.show()
Finding Paths in our K-mer De Bruijn Graphs

In [8]:

```python
k = 5
target = "GACGCGCGCGACGCGCCAA"
kmers = kmersUnique(target, k)
G1 = Graph(kmers)
for vsrc in kmers:
    for vdst in kmers:
        if (vsrc[1:k] == vdst[0:k-1]):
            G1.addEdge(vsrc,vdst)
path = G1.hamiltonianPathV2()

print(path)
seq = path[0][0:k]
for kmer in path[1:]:
    seq += kmer[k-1]
print(seq)
print(seq == target)
```

```
['GACGCG', 'ACGCC_1', 'CGGCC_1', 'GCGCA_1', 'CGCAC', 'GCACG', 'CACGG', 'ACGCG_2', 'CGCGG', 'GCAGC', 'CGGCC_3', 'GCGCC_2', 'GCCCA_2', 'CGCAA']
GACGCGACGCGCGCGCGAA
False
```

Not the sequence we expected ...
Let’s look at the resulting graphs

The one we hoped for. Visits CGGCG₃ before CGGCG₂

The one we found visits CGGCG₂ before CGGCG₃
What's the Problem?

- There are many possible Hamiltonian Paths
- How do they differ?
  - There were two possible paths leaving any [CGGCG] node
    - [CGGCG] → [GGCGC]
    - [CGGCG] → [GGCGG]
  - A valid solution can be found down either path
- There might be even more solutions
- Genome assembly appears ambiguous like the Minimal Substring problem, but is it?
How about an Euler Path?

```python
In [20]:
k = 5
target = "GACGCGCGCGACGCGCA"
kmers = kmersUnique(target, k)
print(kmers)

nodes = sorted(set([code[:k-1] for code in kmers] + [code[1:k] for code in kmers]))
print(nodes)
G2 = Graph(nodes)
for code in kmers:
    G2.addEdge(code[:k-1],code[1:k],code)
path = G2.eulerianPath()
print(path)
path = G2.eulerEdges(path)
print(path)

seq = path[0][0:k]
for kmer in path[1:]:
    seq += kmer[k-1]
print(seq)
print(seq == target)
```

```
['ACGGC_1', 'ACGGC_2', 'CACGG', 'CGCAA', 'CGCAC', 'CGGCG_1', 'CGGCG_2', 'CGGCG_3', 'GACG6', 'GCACG', 'GCGCA_1', 'GCGCA_2',
 'GCCGG', 'GCCGG_1', 'GCCGG_2', 'GCCGG_3']
['ACGG', 'CACG', 'CGCA', 'CGGC', 'GACG', 'GAAA', 'GCAC', 'GCAC', 'GCAC', 'GCCG', 'GCCG', 'GCCG']
[4, 0, 3, 9, 8, 3, 9, 7, 2, 8, 1, 0, 3, 9, 7, 2, 5]
[2GAC6', 'ACGGC_2', 'CGGCG_3', 'GCCCG', 'GCACG', 'GCGCA_2', 'GCCGA_2', 'CGCA', 'CACGG', 'ACACG', 'ACCG_1', 'CG
CGC_1', 'GCCCG_1', 'GCCCA_2', 'CGCAA']
GACGCGCGCGACGCGCA
```

True
The k-1 De Bruijn Graph with k-mer edges

- We got the right answer, but we were lucky.
- There is a path in this graph that matches the Hamiltonian path that we found before.
What are the Differences?

How might we favor one solution over the other?
Choose a bigger k-mer

```python
In [22]:
k = 8
target = "GACGCGCGCAACGCGCGCAAA"
kmers = kmersUnique(target, k)
print(kmers)

nodes = sorted(set([code[:k-1] for code in kmers] + [code[1:k] for code in kmers]))
print(nodes)
G3 = Graph(nodes)
for code in kmers:
    G3.addEdge(code[:k-1], code[1:k], code)
path = G3.eulerianPath()
print(path)
path = G3.eulerianEdges(path)
print(path)

seq = path[0][0:k]
for kmer in path[1:]:
    seq += kmer[k-1]
print(seq)
print(seq == target)
```

```
['ACGCGCGCA', 'ACGCGCGC', 'CAGCGCGC', 'CAGCGCAG', 'CGACGCGC', 'CGACGCGC', 'CGACGCGC', 'GCACGCGC', 'GCCACGCGC', 'GCCACGCGC']
['ACGCGCGC', 'ACGCGCGC', 'CAGCGCGC', 'CAGCGCAG', 'CGACGCGC', 'CGACGCGC', 'CGACGCGC', 'GCACGCGC', 'GCCACGCGC', 'GCCACGCGC']
[6, 1, 5, 12, 9, 4, 11, 8, 3, 7, 2, 0, 4, 10]
["GACGCGCGC", "ACGCGCGCG", "CGCGCGCG", "CGCGCGCG", "CGCGCGCG", "CGCGCGCG", "CGCGCGCG", "CGCGCGCG", "CGCGCGCG", "CGCGCGCG"]
GACGCGCGCGCAGCGCGCAAA
True
```
Advantage of larger k-mers

- Making k larger (8) eliminates the second choice of loops
- There are edges to choose from, but they all lead to the same path of vertices
Applied to the Hamiltonian Solution

```python
In [23]:
k = 8
target = "GACGCGG6CGCAGCGCGCA"
kmers = kmersUnique(target, k)
G4 = Graph(kmers)
for vsrc in kmers:
    for vdst in kmers:
        if (vsrc[1:k] == vdst[0:k-1]):
            G4.addEdge(vsrc, vdst)
path = G4.hamiltonianPathV2()

print(path)
s3q = path[0][0:k]
for kmer in path[1:]:
    s3q += kmer[k-1]
print(s3q)
print(s3q == target)

['GACGCGCGG', 'ACG6CGCG', 'G6CG6CGC', 'GCG6CGCA', 'CG6CGCA6C', 'G6CG6CA6G', 'GC6AC66C', 'GC6ACG6C', 'CA
G6CGC', 'AC66CGCA', '6GCG6CGA']

GACGCGCGGCGCAGCGCGCA

True
```
Graph with 8-mers as vertices

- There is only one Hamiltonian path
- There are no repeated k-mers
Assembly in Reality

- Problems with repeated k-mers
- We can't distinguish between repeated k-mers
- Recall we knew from our example that were:
  \{2:ACGGC, 3:CGGCG, 2:GCGCA, 2:GGCGC\}
- Assembling path without repeats:

```python
In [26]:

k = 5
target = "GACGCGCGCGACGCGCGCA"
kmers = set([target[i:i+k] for i in range(len(target)-k+1)])
nodes = sorted(set([code[:k-1] for code in kmers]) + [code[i:k] for code in kmers])
G5 = Graph(nodes)
for code in kmers:
    G5.addEdge(code[:k-1], code[1:k], code)

print(sorted(G5.vertex.items()))
print(G5.edge)
```

```
[(0, 'ACGG'), (1, 'CAGC'), (2, 'CGCA'), (3, 'CGGC'), (4, 'GACG'), (5, 'GCAA'), (6, 'GCAC'), (7, 'GCGC'), (8, 'GCGG'), (9, 'GCGC')]
[(9, 8), (3, 9), (1, 0), (4, 0), (5, 1), (8, 3), (0, 3), (2, 5), (7, 2), (2, 6), (9, 7)]
```
There is no single Euler Path
But there are is a set of paths that covers all edges
['GACGGCG', 'GGCGGC', 'GGCGCA', 'CGCAA', 'CGCACGG']
○ Extend a sequence from a node until you reach a node with an out-degree → in-degree
○ Save these partially assembled subsequences, call them contigs
○ Start new contigs following each out-going edge at these branching nodes
Next assemble contigs

- Use a modified read-overlap graph to assemble these contigs
- Add edge-weights that indicate the amount of overlap

- Usually much smaller than the graph made from k-mers
A Heavy Path

Find the heaviest path touching all vertices in this smaller graph

GACGGCGGCGCAGGCGCAA
GACGGCG
GGCGGC
GGCGCA
CGCAGG
GGCGCA
CGCAA

GACGGCG

GGCGGC

GGCGCA

CGCAA

17
Discussion

- No simple single algorithm for assembling a real genome sequences
- Generally, an iterative task
  - Choose a k-mer size, ideally such that no or few k-mers are repeated
  - Assemble long paths (contigs) in the resulting graph
  - Use these contigs, if they overlap sufficiently, to assemble longer sequences
- Truly repetitive subsequences are a challenge
  - Leads to repeated k-mers and loops in graphs in the problem areas
  - Often we assemble the "shortest" version of a genome consistent with our k-mer set
- Things we've ignored
  - Our k-mers are extracted from short read sequences that may contain errors
  - Our short read set could be missing entire segments from the actual genome
  - Our data actually supports 2 paths, one through the primary sequence, and a second through it again in reverse complement order.