The Realities of Genome Assembly

• Problem Set #2 is posted
From Last Time

What we learned from a related "Minimal Superstring" problem

- Can be constructed by finding a Hamiltonian path of an n-dimensional De Bruijn graph over k symbols
  - Brute-force method is explores all V! paths through V vertices
  - Branch-and-Bound method considers only paths composed of edges
  - 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 (n−1)-dimensional De Bruijn graph.
  - Euler's method finds a path using all edges in $O(E) \equiv O(V^2)$ steps
  - Graph must satisfy 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 are known and used as a vertex, (all $\sigma^k$)
  - Paths, and there may be multiple, are solutions
- Read fragments
  - No guarantee that we will see every k-mer
  - Can't disambiguate repeats
A small "Toy" example

- Our toy sequence from 2 lectures ago
- The complete set of 16 5-mers

- All $k$-mers is equivalent to $kx$ coverage
- Four repeated $k$-mers \{ACGGC, CGCGC, GCGCA, GGCGC\}
Some Code

- First let's add a function to uniquely label repeated k-mers

```python
def kmersUnique(seq, k):
    kmers = sorted([seq[i:i+k] for i in xrange(len(seq)-k+1)])
    for i in xrange(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("GACGCGCCGCACGCCGCAA", 5)
print kmers

['ACGCG_1', 'ACGCG_2', 'CACGG', 'CGCAA', 'CGCAC', 'CGCG_1', 'CGCG_2', 'CGCG_3', 'GACGG', 'GCACG', 'GCACA_1', 'GCGCA_2', 'GCACC', 'GCGC_1', 'GCGGC_2', 'GCGGG']
```
import itertools

class Graph:
    def __init__(self, vlist=[]):
        """ Initialize a Graph with an optional vertex list """
        self.index = {v:i for i,v in enumerate(vlist)}
        self.vertex = {i:v for i,v in enumerate(vlist)}
        self.edge = []
        self.edgelabel = []
    def addVertex(self, label):
        """ Add a labeled vertex to the graph """
        index = len(self.index)
        self.index[label] = index
        self.vertex[index] = label
    def addEdge(self, vsrc, vdst, label='', repeats=True):
        """ Add a directed edge to the graph, with an optional label. Repeated edges are distinct, unless repeats is set to False. """
        e = (self.index[vsrc], self.index[vdst])
        if (repeats) or (e not in self.edge):
            self.edge.append(e)
            self.edgelabel.append(label)
    def hamiltonianPath(self):
        """ A Brute-force method for finding a Hamiltonian Path. Basically, all possible N! paths are enumerated and checked for edges. Since edges can be reused there are no distinctions made for "which" version of a repeated edge. """
        for path in itertools.permutations(sorted(self.index.values())):
            for i in xrange(len(path)-1):
                if ((path[i],path[i+1]) not in self.edge):
                    break
            else:
                return [self.vertex[i] for i in path]
        return []
    def SearchTree(self, path, verticesLeft):
        """ A recursive Branch-and-Bound Hamiltonian Path search. Paths are extended one node at a time using only available
Finding Paths in our K-mer De Bruijn Graphs

```
k = 5
target = "GACGGCAGCGCAACAGCGCAGCAGCAA"
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

['GACGG', 'ACGGC_1', 'CGGGC_1', 'GGCG_1', 'CGGCA_1', 'CGCAG', 'GCGGG', 'ACGGC_2', 'CGGGG_2', 'GGGCG', 'GGGCC', 'CGGCG_3', 'GGGCG_2', 'GGCGA_2', 'CGCAA']
GACGGCGCAGCGCAGCAGCAA
False
```
Not what we Expected

The one started with

The one we found
What's the Problem?

• There are many possible Hamiltonian Paths
• How do they differ?
  • There were two possible paths leaving any [CGGCG] node
    ○ [CGGCG] → [GCGG]
    ○ [CGGCG] → [GGCGG]
  • A valid solution can be found down either path
• There might be even more solutions
• Genome assembly is not as ambiguous as the Minimal Substring problem
How about an Euler Path?

```
k = 5
target = "GACGCGCGCACGCGCAAA"
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
```

```
['ACGCG_1', 'ACGCG_2', 'CACCG', 'CGCAA', 'CGCAG', 'CGCG_1', 'CGCG_2', 'CGCG_3', 'GACGG', 'GCACG', 'GCACA_1', 'GCACA_2', 'GCCGG', 'GCCGA_1', 'GCCGA_2', 'GGCGG']
['ACGG', 'CACG', 'CGCA', 'CGGC', 'GACG', 'GCAG', 'GCAC', 'GCCG', 'CGCC', 'GGCG']

[4, 0, 3, 9, 8, 3, 9, 7, 2, 6, 1, 0, 3, 9, 7, 2, 5]

['GACGG', 'ACGCG_2', 'CGCG_3', 'GCGGG', 'GCCGG', 'CGCG_2', 'CGCG_2', 'GCACA_2', 'GCACA', 'GCCG', 'GCAG', 'CACCG', 'ACGCG', 'ACGCG_1', 'CGCG_1', 'GCCG_1', 'GCCG_1', 'CGCGA_1', 'CGCGA']
GACGCGCGACGCGCAAA
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
k = 8
target = "GACGCGCCGCACGGGCAAA"
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.eulerEdges(path)
print path

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

['ACGCGCGG', 'ACGCGGCG', 'CGACCGGC', 'CGGCGCAA', 'CGGGCGAC', 'CGGGCGG', 'GACCACGG', 'GACGCGG', 'GCGACCC', 'GCACGCG', 'GGCGCGA', 'GGCGGCA', 'GGGCCAC', 'GGGCGGC']
['ACGCGCGG', 'ACGCGGCG', 'CGACCGGC', 'CGGCGCAA', 'CGGGCGAC', 'CGGGCGG', 'GACCACGG', 'GACGCGG', 'GCGACCC', 'GCACGCG', 'GGCGCGA', 'GGCGGCA', 'GGGCCAC', 'GGGCGGC']
[5, 1, 5, 12, 9, 4, 3, 7, 2, 0, 4, 19]
['ACGCGCGG', 'ACGCGGCG', 'CGACCGGC', 'CGGCGCAA', 'CGGGCGAC', 'CGGGCGG', 'GACCACGG', 'GACGCGG', 'GCGACCC', 'GCACGCG', 'GGCGCGA', 'GGCGGCA', 'GGGCCAC', 'GGGCGGC']
GACGCGCGGCGACGGGCAAA
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
k = 8
target = "GACGCGCGCGCACGCGC"
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
seq = path[0][0:k]
for kmer in path[1:]:
    seq += kmer[k-1]
print seq == target

[['GACGCGCG', 'ACGCGCGC', 'CGCGCGCG', 'GGCGCGCA', 'CGCGCGAC', 'GCGCGCGA', 'CGCGCGCA', 'CGACCGCG', 'GACCGCGC', 'ACGGCGC', 'ACGGCGCA', 'CGCGCGCA']
GACGCGCGCAGCGCGCA
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:GGGCG}
    - Assembling path without repeats:

```python
k = 5
target = "GACGGCGCAGCGCGGCA"
kmers = set([target[i:i+k] for i in xrange(len(target)-k+1)])
nodes = sorted(set([code[:k-1] for code in kmers] + [code[1: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, 'ACCG'), (2, 'CGCG'), (3, 'CGGCG'), (4, 'GACC'), (5, 'GCGCA'), (6, 'GCGCA'), (7, 'GCGCA'), (8, 'GCGCA'), (9, 'GCGCA'))
((7, 2), (1, 0), (2, 6), (9, 8), (4, 0), (3, 9), (6, 3), (9, 7), (6, 1), (2, 5), (8, 3))]
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
- Find Hamiltonian paths in this smaller graph
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
- Truely 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.