Problem set #2 is on-line

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 $\sigma$ 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 solved 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 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 is present and used as a vertex, \((\text{all } \sigma^k)\)
  - Paths, and there may be multiple, are solutions

- Read fragments
  - No guarantee that we will ever 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              000 110
  010 110              011 010
```
Recall our “Toy” example

GACGCGGCGCAGCGGA
GACGG CGCAC
ACGCG GCACG
CGGCG CACGG
GGCGG ACGGC
GCGGC CGGCG
CGGCG CGGCG
GGCGC GCGCA
GGCGA CGCAA

- Our toy 20 base sequence from 2 lectures ago

GACGCGGCGCAGCGGA
GACGG CGCAC
ACGCG GCACG
CGGCG CACGG
GGCGG ACGGC
GCGGC CGGCG
CGGCG CGGCG
GGCGC GCGCA
GGCGA CGCAA

- The complete set of 16 5-mers

Issues:

- All k-mers is equivalent to $k \times$ coverage, ignoring boundaries
- Four repeated k-mers \{ACGCG, CGGCG, GCGCA, GGCGC\}
Some Code

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

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

kmer = kmerUnique("GACGGCGCGCAGCGCGCAG", 5)
print(kmer)

['ACGGC_1', 'ACGGC_2', 'CACGG', 'CGCAA', 'CGGAC', 'CGGGG_1', 'CGGGG_2', 'CGGGG_3', 'GACGG', 'GCACG', 'GCCGA_1', 'GCCGA_2', 'GCCGC', 'GGCGC_1', 'GGCGC_2', 'GGCGG']
```
Our Graph class from last lecture

```python
def hamiltonianPath2(self):
    """ A wrapper function
    Hamiltonian Path is self.Path2Result = self.SearchTreeNode([],
    return self.Path2Result

def degrees(self):
    """ Returns two dic of each node from inDegree = {} and
    outDegree = {} for src, dst in self.outDegreeList:
        return self.inDegree, outDegree

    def verifyAndSetStart(self):
        inDegree, outDegree = 0, 0
        for vert in self.vertices:
            if (vert in self.inDegree):
                continue
            elif (vert in self.outDegree):
                if (src == dst):
                    continue
                else:
                    start, end = vert
                    self.outDegree[vert] = inDegree,
                    self.inDegree[end] = outDegree
                    return

    def euclidianPath(self):
        graph = [(src, dst) for src, dst in currentVertex = self.verifyAndGetPath
            if currentVert in neighbors[0]:
                path = [currentVertex]
                if (len(path) > 0):
                    if (1 in edgeSet):
                        result += ' [label="%s", penwidth=3.0"]' % (label)
            else:
                result += ' [label="%s"]' % (label)
            result += '
            result = result + overlap=false
            return result

for i, e in enumerate((self.edge):
    src, dst = e
    result += ' NMod >> NMod1 % (src, dst)
    label = self.edge_label[i]
    if (len(label) > 0):
        if (1 in edgeSet):
            result += ' [label="%s", penwidth=3.0"]' % (label)
        else:
            result += ' [label="%s"]' % (label)
    result += '
    result = result + overlap=false
    return result

result += ' [label="%s", penwidth=3.0"]' % (label)
result += '
    result = result + overlap=false
    return result

for (path) in itertools:
    if (path[0] ==
        result = result + overlap=false
        return result
```

Finding Paths in our K-mer De Bruijn Graphs

In [8]:

```python
k = 5
target = "GACGCGCGCGACGCGCGCAA"
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)
```

```
GACGCGACGCGCGCGCAA
False
```

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

The one we hoped for. Visits CGGCG$_3$ before CGGCG$_2$

The one we found visits CGGCG$_2$ before CGGCG$_3$
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?

In [20]:

k = 5

target = "GACGGCGCGCAGCGCAGCAA"

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)

seq = path[0][0:k]

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

print(seq)

print(seq == target)

['AGCGCC', 'ACCGG_2', 'CACG', 'CGCCA', 'CGCAC', 'CGGCG_1', 'CGGC_2', 'CGGC_3', 'GACGG', 'GACCG', 'GCACG', 'GCC_G1', 'GCC_G2', 'GCCG']

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

['ACGG', 'CACG', 'CGCA', 'CGGC', 'GACG', 'GCCA', 'GCG', 'GCAC', 'GCCG', 'GCGG', 'GCCG']

['GACGG', 'ACCGG_2', 'CGGGC_3', 'GCCG', 'GCCG', 'GCCG_2', 'GCG_2', 'GCC_2', 'GCACAC', 'GCACG', 'CAGG', 'ACGG_1', 'CAGG_2']

GACGGCGCGCAGCGCAGCAA

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 = "GACGCGCGCGCAACGCGCGCA"
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[1:1]:
    seq += kmer[k-1]
print(seq)
print(seq == target)
```

["ACACGCACA", "ACCGGCGC", "CACGGCAGC", "CGGCGCGA", "CGGGCGAC", "CGGCGGAC", "GCACGCGG", "GCAGCGGC", "GGCGCGA"]

[6, 1, 5, 12, 9, 4, 11, 8, 3, 7, 2, 0, 4, 10]


GACGCGCGCGACGCGCGCA

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
```
In [23]:

k = 8
target = "GACGCGGCGCACGGCGCAA"
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)
print(seq == target)

GACGCGGCGCACGGCGCAA
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[1:i+k] for i in range(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)
```

```
[[('ACGG'), (1, 'ACGC'), (2, 'CGCA'), (3, 'CGGC'), (4, 'GACG'), (5, 'GCCA'), (6, 'GCAC'), (7, 'GCCG'), (8, 'GCCG'), (9, 'GC6C')]]
[[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

\[
\text{[ 'GACGGCG', 'GGCGGC', 'GGCGCA', 'CGCAA', 'CGCAGC' ]}
\]

- 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

GACGGCGGCGCAGCGCGCAA
GACGGCG
GGCGGC
GGCGCA
CGCAGGG
GGCGCA
CGCAA

GACGGCG
GGCGGC
GGCGCA
CGCAGGG
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.