The Realities of Genome Assembly

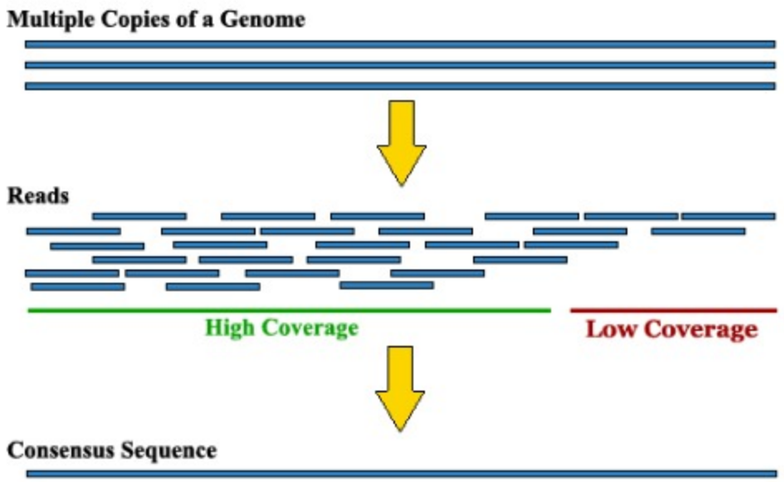
- Problem Set #2 is posted

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 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

- Mininmal substring problem
  - Every k-mer are known and used as a vertex, (all σ^k)
  - Paths, and there may be multiple, are solutions
- Read fragments
  - No guarentee that we will see every k-mer
  - Can't disambiguate repeats

A small "Toy" example

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

- All *k-mers* is equivalent to *kx* coverage
- Four repeated k-mers {ACGGC, CGGCG, GCGCA, GGCGC}

Some Code

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

```
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("GACGGCGGCGCACGGCGCAA", 5)
print kmers
```

```
['ACGGC_1', 'ACGGC_2', 'CACGG', 'CGCAA', 'CGCAC', 'CGGCG_1', 'CGGCG_2', 'CGGCG_3', 'GACGG', 'GCACG', 'GCGCA_1', 'GCGCA_2', 'GCGGC', 'G
GCGC_1', 'GGCGC_2', 'GGCGG']
```

Our Graph class from last lecture

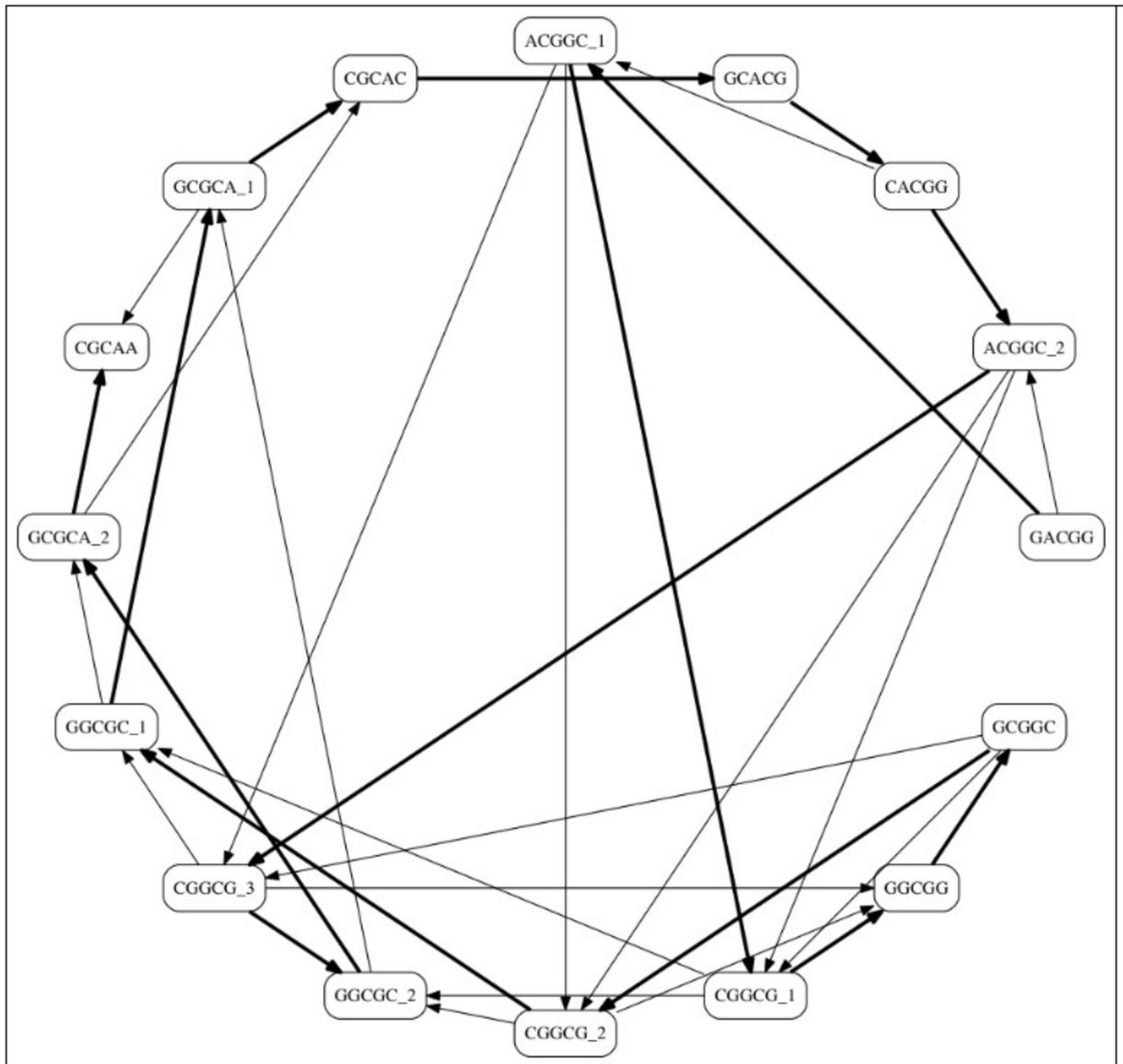
```
import itertools

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 distictions
        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

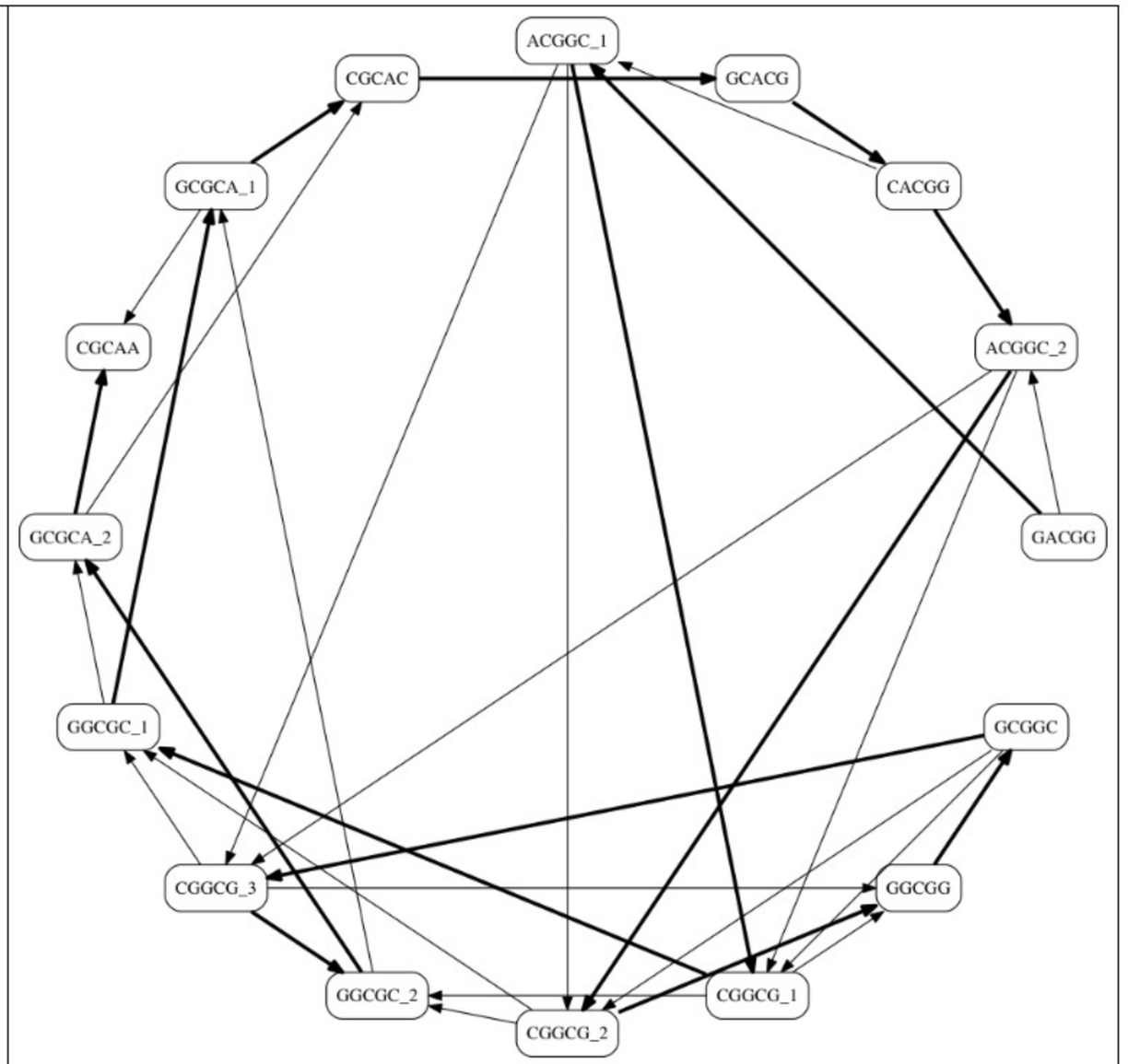
```
k = 5
target = "GACGGCGGCGCACGGCGCAA"
kmers = kmersUnique(target, k)
G1 = Graph(kmers)
for vsrc in kmers:
    for vdst in kmers:
        if (vsrc[1:k] == vdst[0:k-1]):
            G1.addEdge(vsrc,vdst)
path = G1.hamiltonianPathV2()

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

```
['GACGG', 'ACGGC_1', 'CGGCG_1', 'GGCGC_1', 'GCGCA_1', 'CGCAC', 'GCACG', 'CACGG', 'ACGGC_2', 'CGGCG_2', 'GGCGG', 'GCGGC', 'CGGCG_3', 'GGCGC_2',
'GCGCA_2', 'CGCAA']
GACGGCGCACGGCGGCGCAA
False
```

Not what we Expected



The one started with



The one we found

What's the Problem?

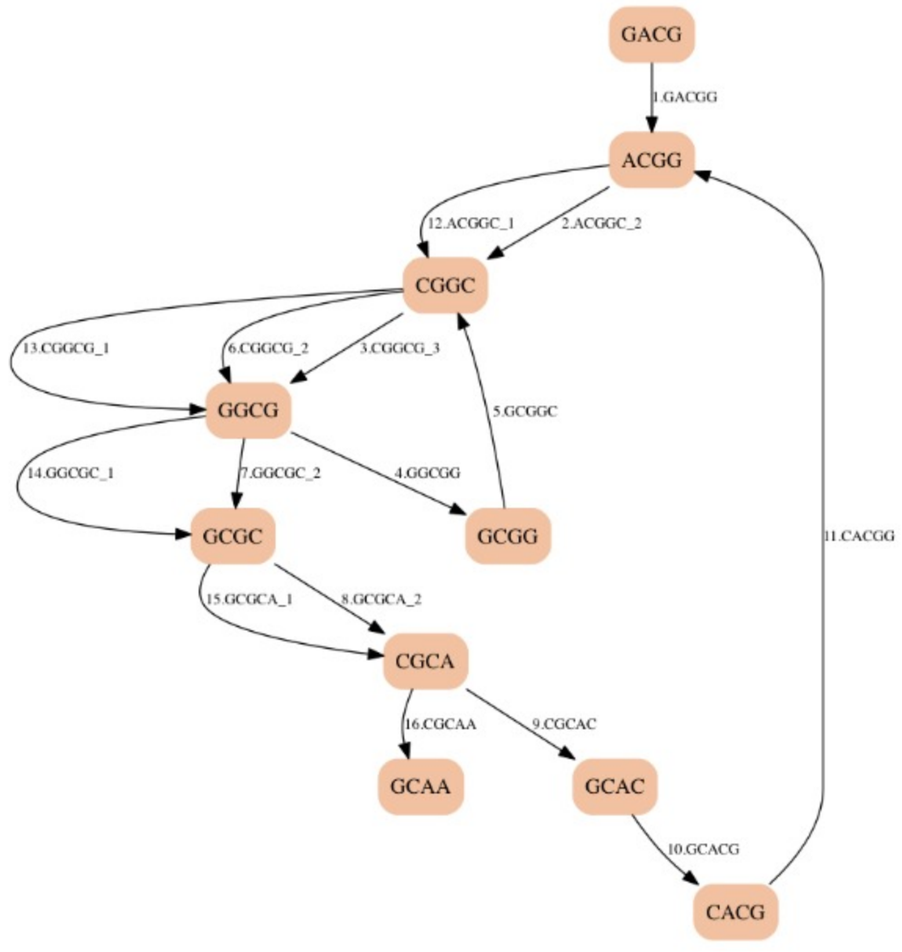
- There are many possible Hamiltonian Paths
- How do they differ?
  - There were two possible paths leaving any [CGGCG] node
    - [CGGCG] → [GGCGC]
    - [CGGCG] → [GGCGG]
  - A valid solution can be found down either path
- There might be even more solutions
- Genome assembly is not as ambiguous as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

How about an Euler Path?

```
k = 5
target = "GACGGCGGCGCACGGCGCAA"
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
```

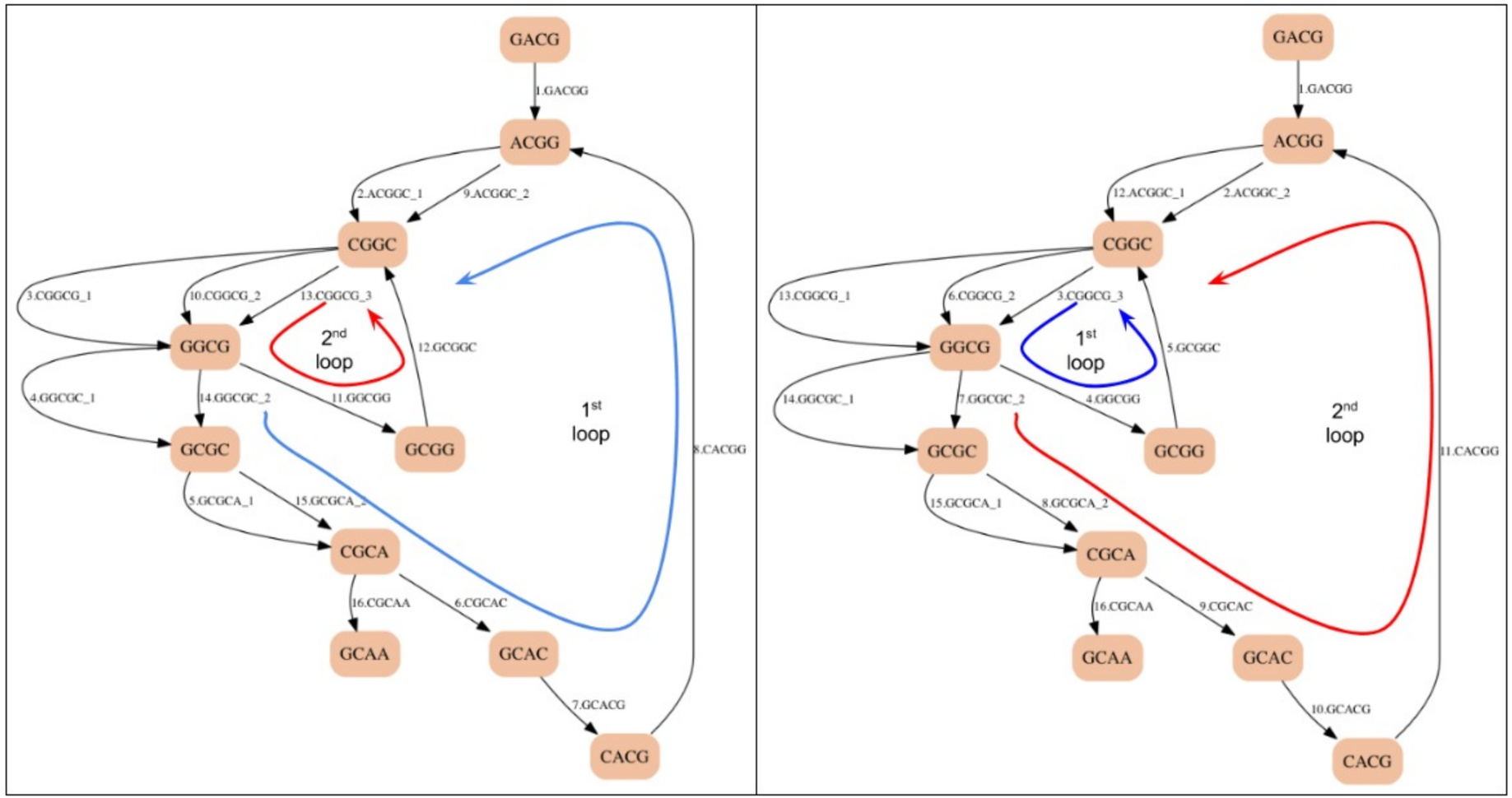
```
['ACGGC_1', 'ACGGC_2', 'CACGG', 'CGCAA', 'CGCAC', 'CGGCG_1', 'CGGCG_2', 'CGGCG_3', 'GACGG', 'GCACG', 'GCGCA_1', 'GCGCA_2', 'GCGGC', 'GGCGC_1',
'GGCGC_2', 'GGCGG']
['ACGG', 'CACG', 'CGCA', 'CGGC', 'GACG', 'GCAA', 'GCAC', 'GCGC', 'GCGG', 'GGCG']
[4, 0, 3, 9, 8, 3, 9, 7, 2, 6, 1, 0, 3, 9, 7, 2, 5]
['GACGG', 'ACGGC_2', 'CGGCG_3', 'GGCGG', 'GCGGC', 'CGGCG_2', 'GGCGC_2', 'GCGCA_2', 'CGCAC', 'GCACG', 'CACGG', 'ACGGC_1', 'CGGCG_1', 'GGCGC_1',
'GCGCA_1', 'CGCAA']
GACGGCGGCGCACGGCGCAA
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?

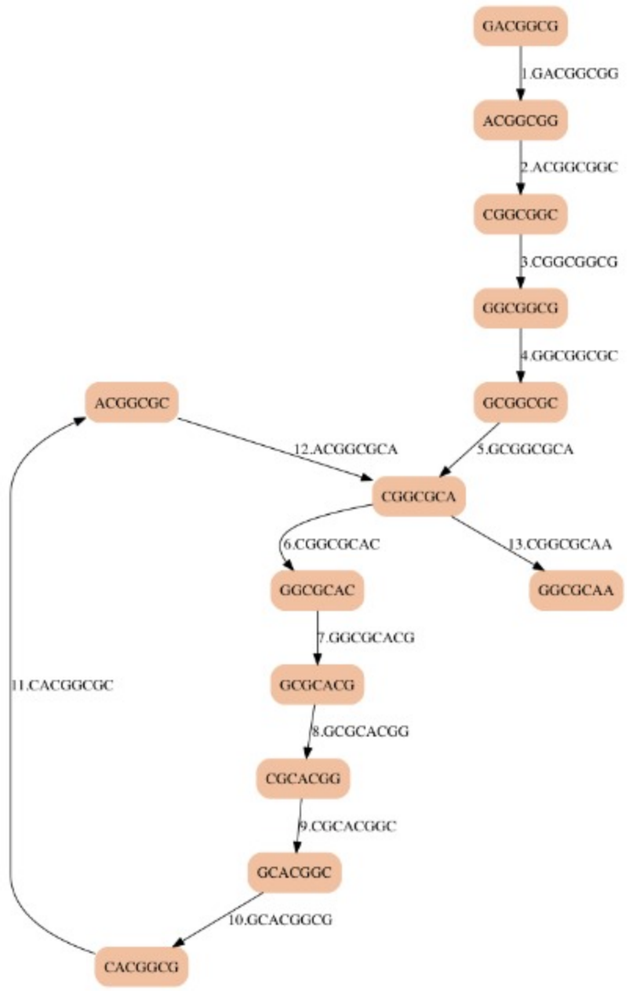
Choose a bigger k-mer

```
k = 8
target = "GACGGCGGCGCACGGCGCAA"
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
```

```
['ACGGCGCA', 'ACGGCGGC', 'CACGGCGC', 'CGCACGGC', 'CGGCGCAA', 'CGGCGCAC', 'CGGCGGCG', 'GACGGCGG', 'GCACGGCG', 'GCGCACGG', 'GCGGCGCA', 'GGCGCAC
G', 'GGCGGCGC']
['ACGGCGC', 'ACGGCGG', 'CACGGCG', 'CGCACGG', 'CGGCGCA', 'CGGCGGC', 'GACGGCG', 'GCACGGC', 'GCGCACG', 'GCGGCGC', 'GGCGCAA', 'GGCGCAC', 'GGCGGC
G']
[6, 1, 5, 12, 9, 4, 11, 8, 3, 7, 2, 0, 4, 10]
['GACGGCGG', 'ACGGCGGC', 'CGGCGGCG', 'GGCGGCGC', 'GCGGCGCA', 'CGGCGCAC', 'GGCGCACG', 'GCGCACGG', 'CGCACGGC', 'GCACGGCG', 'CACGGCGC', 'ACGGCGC
A', 'CGGCGCAA']
GACGGCGGCGCACGGCGCAA
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

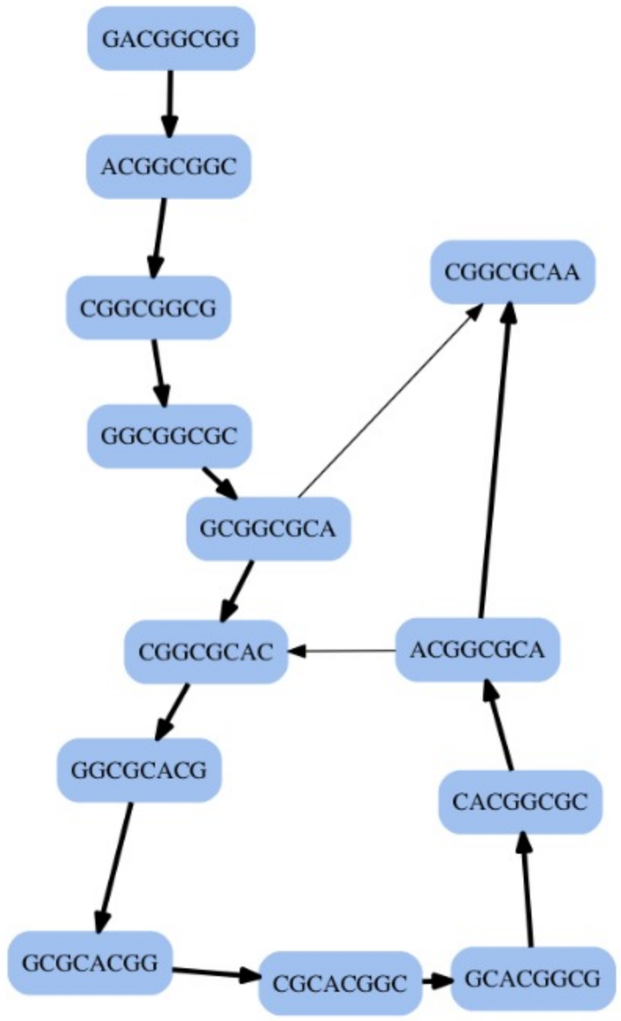


Applied to the Hamiltonian Solution

```
k = 8
target = "GACGGCGGCGCACGGCGCAA"
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
```

```
['GACGGCGG', 'ACGGCGGC', 'CGGCGGCG', 'GGCGGCGC', 'GCGGCGCA', 'CGGCGCAC', 'GGCGCACG', 'GCGCACGG', 'CGCACGGC', 'GCACGGCG', 'CACGGCGC', 'ACGGCGC
A', 'CGGCGCAA']
GACGGCGGCGCACGGCGCAA
True
```

Graph with 8-mers as vertices



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

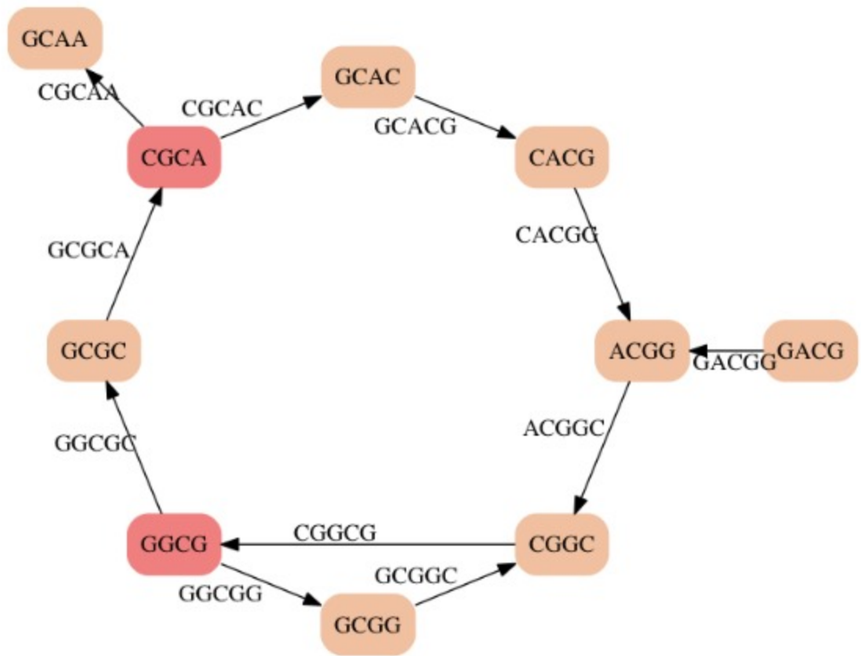
Assembly in Reality

- 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:

```
k = 5
target = "GACGGCGGCGCACGGCGCAA"
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, 'CACG'), (2, 'CGCA'), (3, 'CGGC'), (4, 'GACG'), (5, 'GCAA'), (6, 'GCAC'), (7, 'GCGC'), (8, 'GCGG'), (9, 'GGCG')]
[(7, 2), (1, 0), (2, 6), (9, 8), (4, 0), (3, 9), (0, 3), (9, 7), (6, 1), (2, 5), (8, 3)]
```

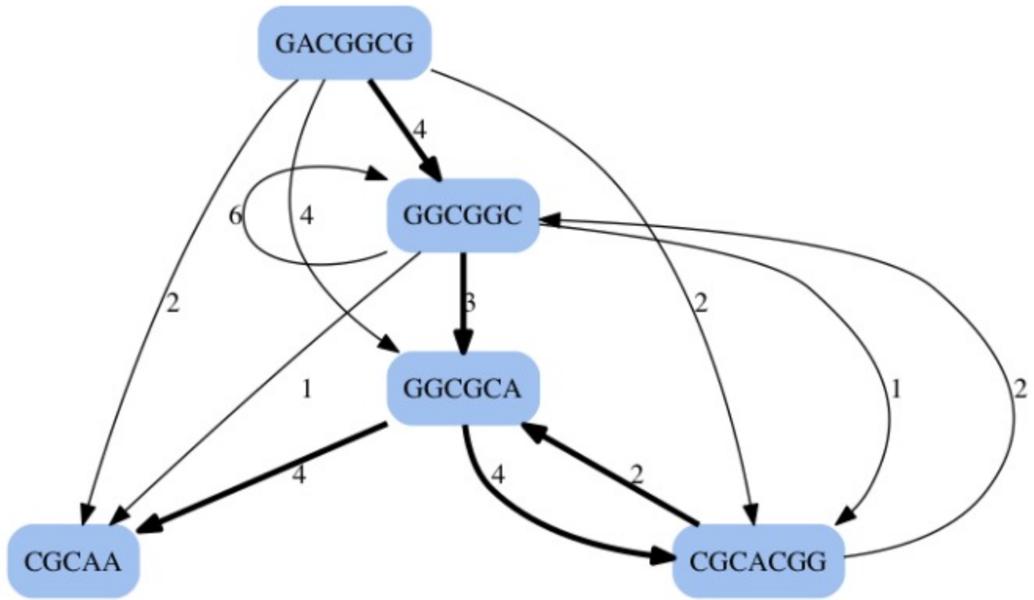
Resulting Graph with "distinct" 5-mers as edges



- 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 suffciently, 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.