# The Realities of Genome Assembly

1

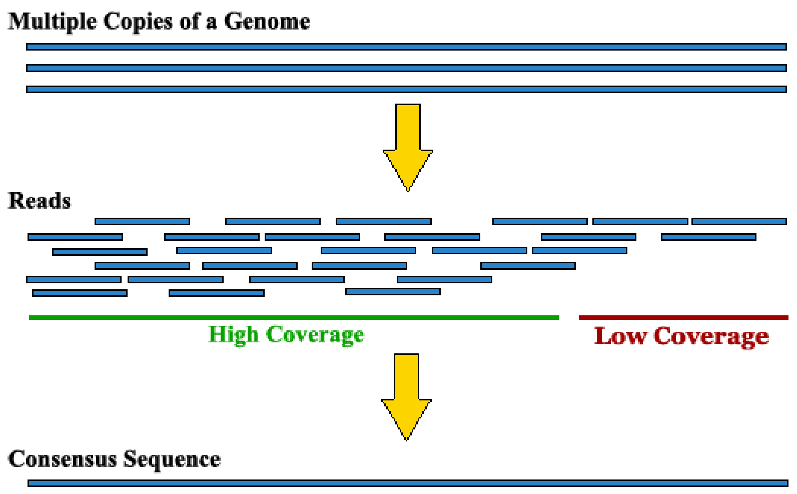
## From Last Time

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
    - 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

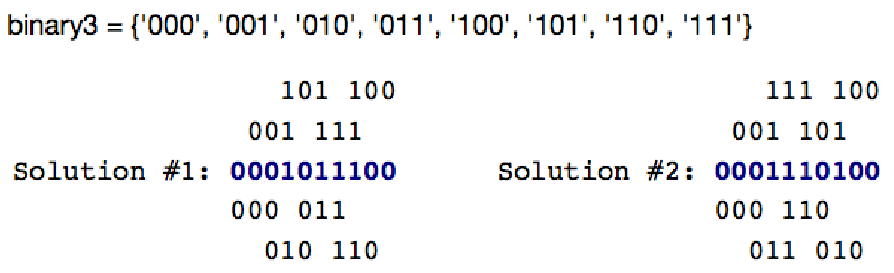
- Can be constructed by finding a *Eulerian path* of a (k−1)-dimensional De Bruijn graph where k-mers are edges.
    - Euler's method finds a path using all edges in $O(E) \equiv O(V^2)$ steps
    - Graph must statisfy contraints to be sure that a solution exists
        - All but two vertices must be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



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
```

- Mininmal substring problem
  - Every k-mer is 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

```
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 *k*× coverage, ignoring boundaries
- 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', 'GGCGC_
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
        edges from the graph. """
        if (len(verticesLeft) == 0):
```

# Our Graph class from last lecture

```
    def SearchTree(self, path, verticesLeft):
        """ A recursive Branch-and-Bound Hamiltonian Path search.
        Paths are extended one node at a time using only available
        edges from the graph. """
        if (len(verticesLeft) == 0):
            self.PathV2result = [self.vertex[i] for i in path]
            return True
        for v in verticesLeft:
            if (len(path) == 0) or ((path[-1],v) in self.edge):
                if self.SearchTree(path+[v], [r for r in verticesLeft if r != v]):
                    return True
        return False
    def hamiltonianPathV2(self):
        """ A wrapper function for invoking the Branch-and-Bound
        Hamiltonian Path search. """
        self.PathV2result = []
        self.SearchTree([],sorted(self.index.values()))
        return self.PathV2result
    def degrees(self):
        """ Returns two dictionaries with the inDegree and outDegree
        of each node from the graph. """
        inDegree = {}
        outDegree = {}
        for src, dst in self.edge:
            outDegree[src] = outDegree.get(src, 0) + 1
            inDegree[dst] = inDegree.get(dst, 0) + 1
        return inDegree, outDegree
    def verifyAndGetStart(self):
        inDegree, outDegree = self.degrees()
        start = 0
        end = 0
        for vert in self.vertex.iterkeys():
            ins = inDegree.get(vert,0)
            outs = outDegree.get(vert,0)
            if (ins == outs):
                continue
            elif (ins - outs == 1):
                end = vert
```

# Our Graph class from last lecture

```
            elif (ins - outs == 1):
                end = vert
            elif (outs - ins == 1):
                start = vert
            else:
                start, end = -1, -1
                break
        if (start >= 0) and (end >= 0):
            return start
        else:
            return -1
    def eulerianPath(self):
        graph = [(src,dst) for src,dst in self.edge]
        currentVertex = self.verifyAndGetStart()
        path = [currentVertex]
        # "next" is where vertices get inserted into our tour
        # it starts at the end (i.e. it is the same as appending),
        # but later "side-trips" will insert in the middle
        next = 1
        while len(graph) > 0:
            for edge in graph:
                if (edge[0] == currentVertex):
                    currentVertex = edge[1]
                    graph.remove(edge)
                    path.insert(next, currentVertex)
                    next += 1
                    break
            else:
                for edge in graph:
                    try:
                        next = path.index(edge[0]) + 1
                        currentVertex = edge[0]
                        break
                    except ValueError:
                        continue
                else:
                    print "There is no path!"
                    return False
        return path
```

# Our Graph class from last lecture

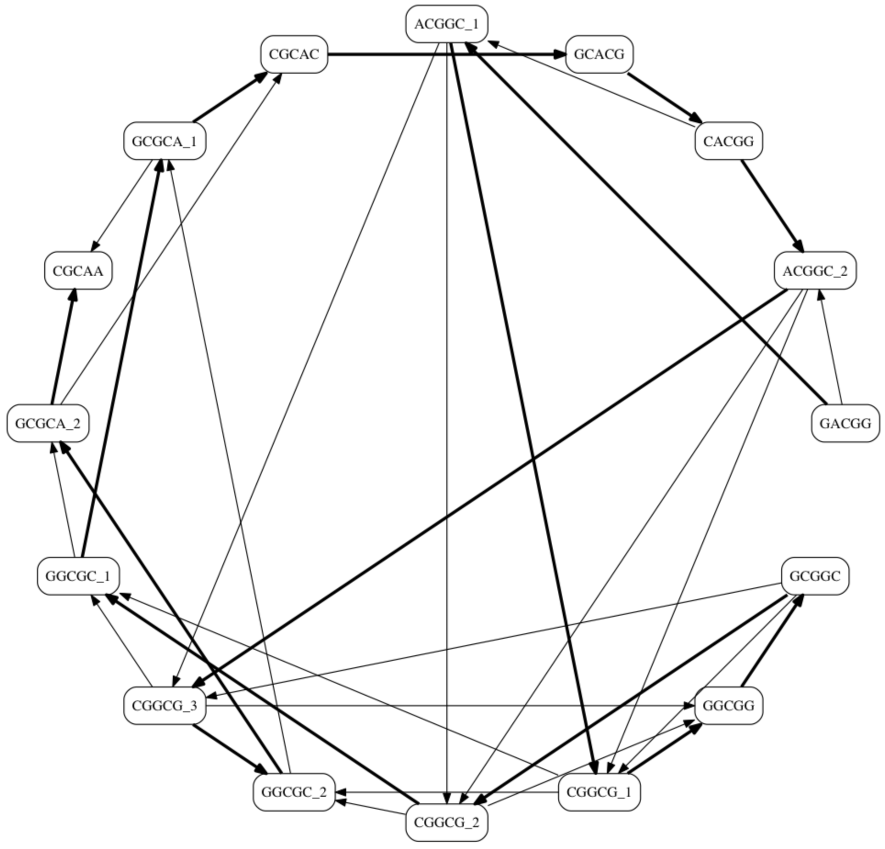
```
    def eulerEdges(self, path):
        edgeId = {}
        for i in xrange(len(self.edge)):
            edgeId[self.edge[i]] = edgeId.get(self.edge[i], []) + [i]
        edgeList = []
        for i in xrange(len(path)-1):
            edgeList.append(self.edgelabel[edgeId[path[i],path[i+1]].pop()])
        return edgeList
    def render(self, highlightPath=[]):
        """ Outputs a version of the graph that can be rendered
        using graphviz tools (http://www.graphviz.org/)."""
        edgeId = {}
        for i in xrange(len(self.edge)):
            edgeId[self.edge[i]] = edgeId.get(self.edge[i], []) + [i]
        edgeSet = set()
        for i in xrange(len(highlightPath)-1):
            src = self.index[highlightPath[i]]
            dst = self.index[highlightPath[i+1]]
            edgeSet.add(edgeId[src,dst].pop())
        result = ''
        result += 'digraph {\n'
        result += '   graph [nodesep=2, size="10,10"];\n'
        for index, label in self.vertex.iteritems():
            result += '    N%d [shape="box", style="rounded", label="%s"];\n' % (index, label)
        for i, e in enumerate(self.edge):
            src, dst = e
            result += '    N%d -> N%d' % (src, dst)
            label = self.edgelabel[i]
            if (len(label) > 0):
                if (i in edgeSet):
                    result += ' [label="%s", penwidth=3.0]' % (label)
                else:
                    result += ' [label="%s"]' % (label)
            elif (i in edgeSet):
                result += ' [penwidth=3.0]'
            result += ';\n'
        result += '    overlap=false;\n'
        result += '}\n'
```

## Finding Paths in our K-mer De Bruijn Graphs

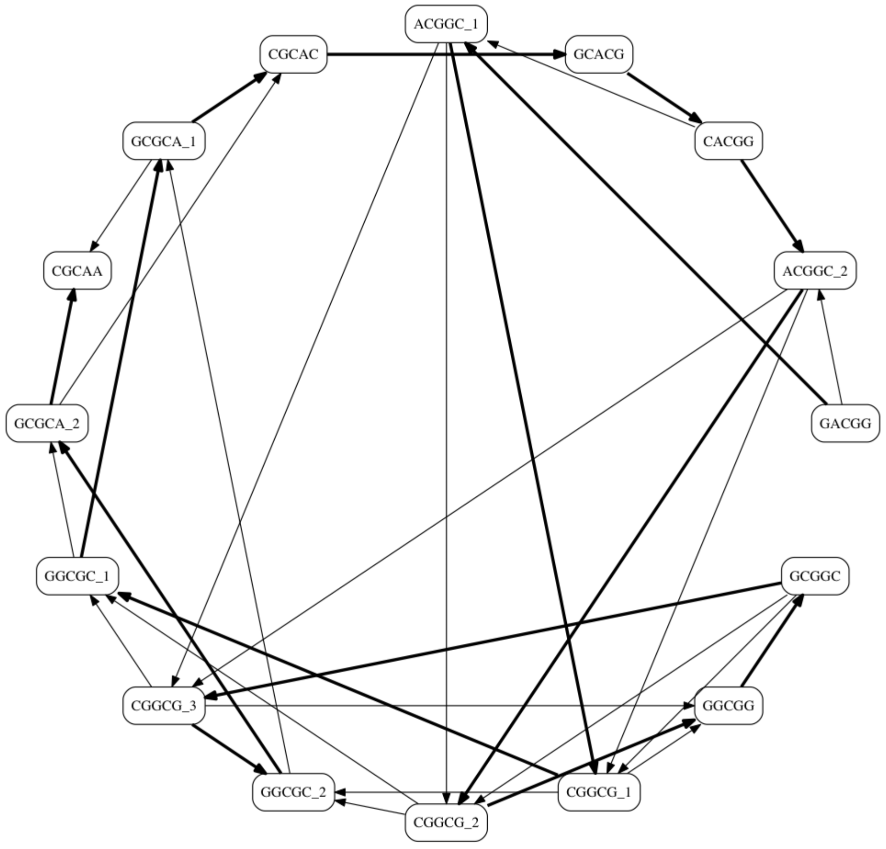
```
k = 5
target = "GACGGCGGCGCACGGCGCAA"
kmers = kmersUnique(target, k)
G1 = Graph(kmers)
for vsrc in kmers:
    for vdst in kmers:
        if (vsrc[1:k] == vdst[0:k-1]):
            G1.addEdge(vsrc,vdst)
path = G1.hamiltonianPathV2()

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

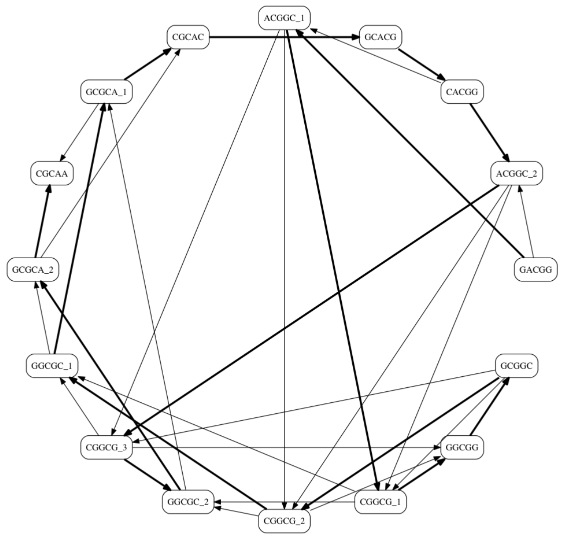


The one we hoped for. Visits $CGGCG_3$ before $CGGCG_2$



The one we found Visits $CGGCG_2$ before $CGGCG_3$

## What's the Problem?



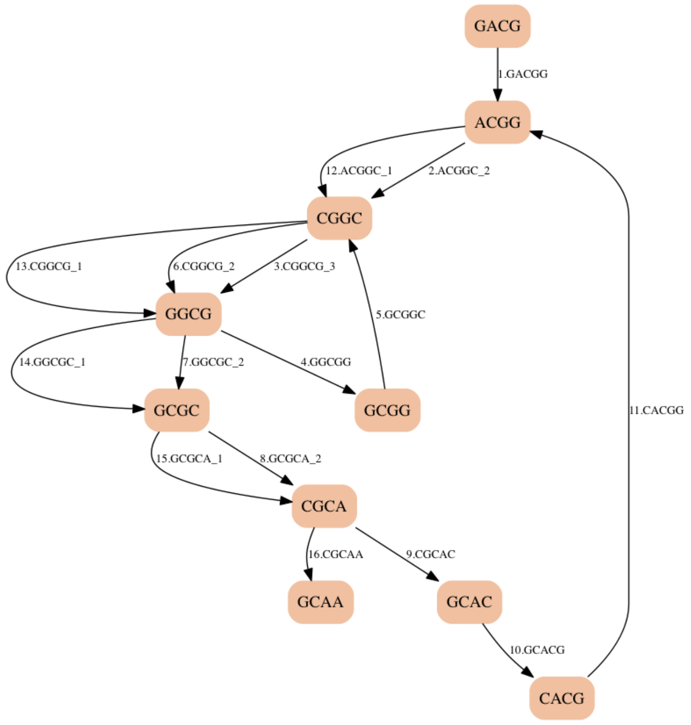
- There are many possible Hamiltonian Paths
- How do they differ?
  - There were two possible paths leaving any [CGGCG] node
    - [CGGCG] → [GGCGC]
    - [CGGCG] → [GGCGG]
  - A valid solution can be found down either path
- There might be even more solutions
- Genome assembly appears ambiguous like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

## How about an Euler Path?

```
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
```

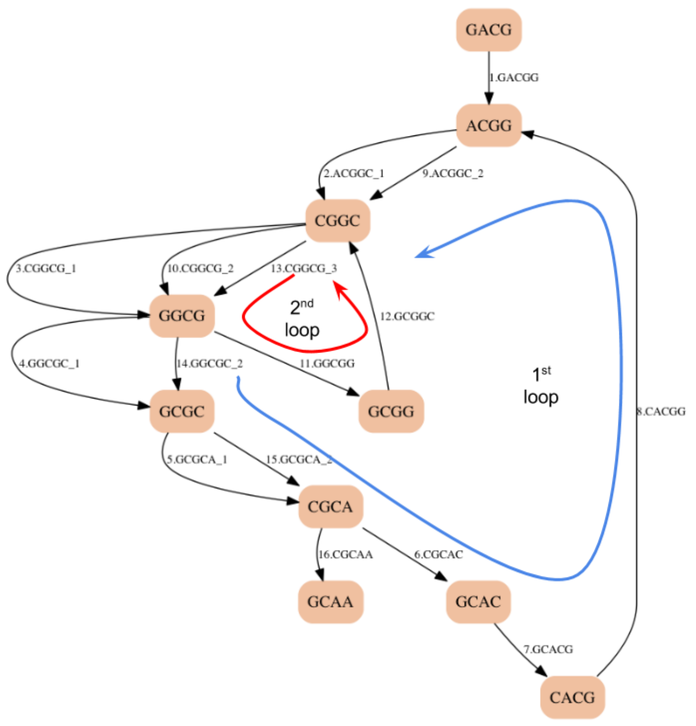
```
['ACGGC_1', 'ACGGC_2', 'CACGG', 'CGCAA', 'CGCAC', 'CGGCG_1', 'CGGCG_2', 'CGGCG_3', 'GACGG', 'GCACG', 'GCGCA_1', 'GCGCA_2', 'GCGGC', 'GGCGC_
1', 'GGCGC_2', 'GGCGG']
['ACGG', 'CACG', 'CGCA', 'CGGC', 'GACG', 'GCAA', 'GCAC', 'GCGC', 'GCGG', 'GGCG']
[4, 0, 3, 9, 8, 3, 9, 7, 2, 6, 1, 0, 3, 9, 7, 2, 5]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

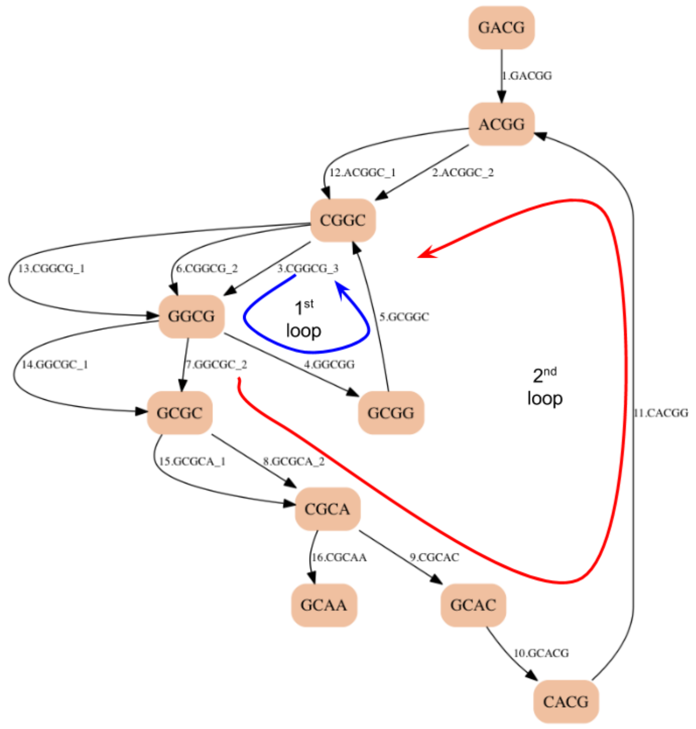
## The k-1 De Bruijn Graph with k-mer edges



- 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?

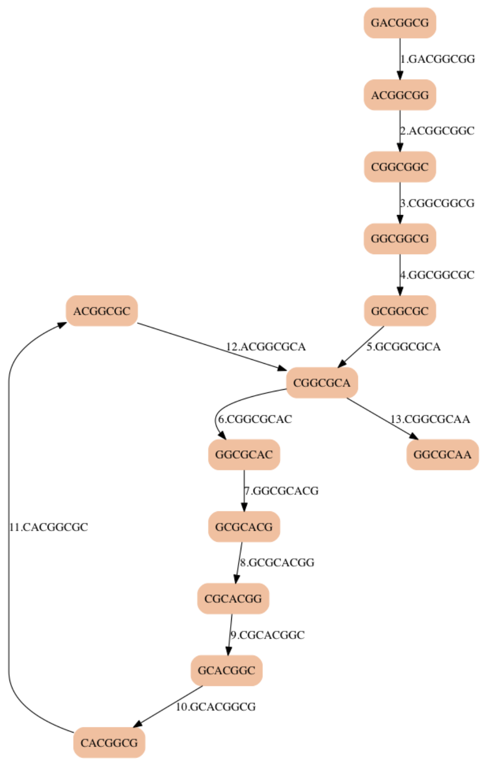
## Choose a bigger k-mer

```
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

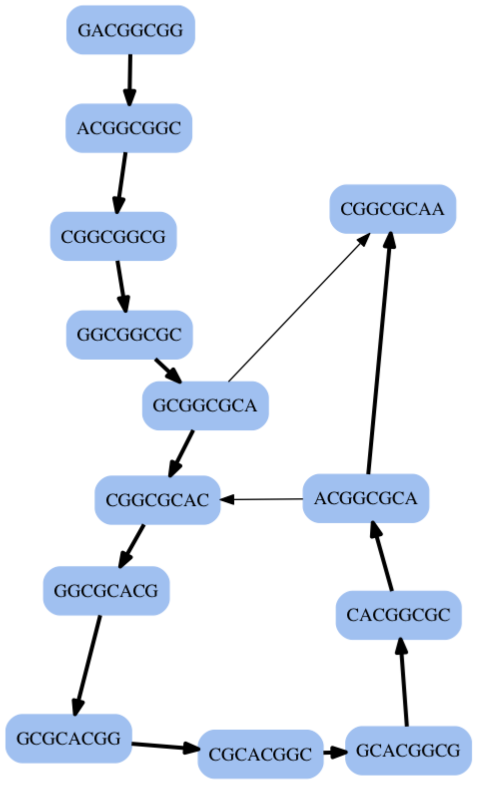


## Applied to the Hamiltonian Solution

```
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

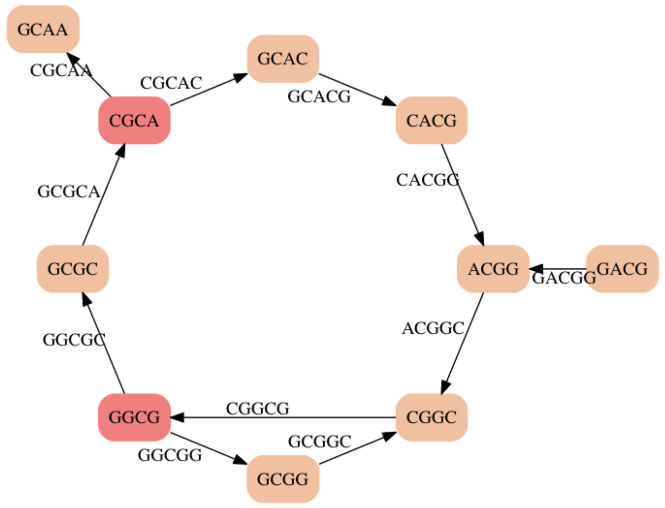
## Assembly in Reality

- 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)]
```

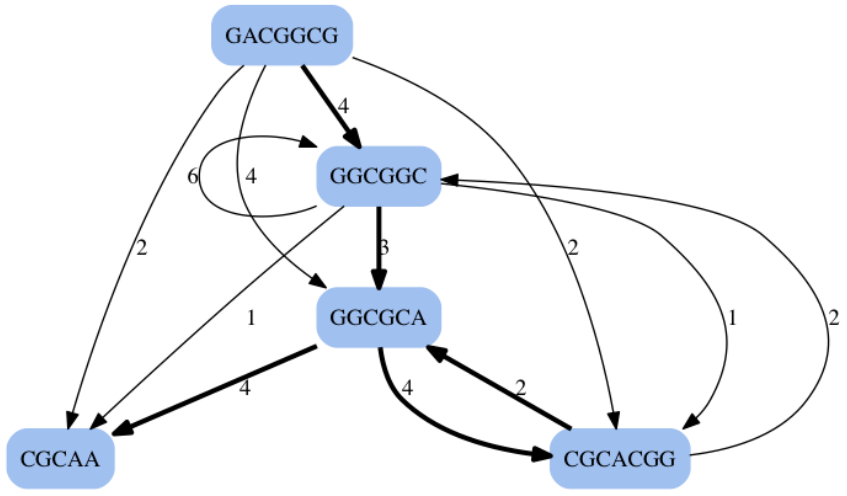
# Resulting Graph with "distinct" 5-mers as edges



- 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.