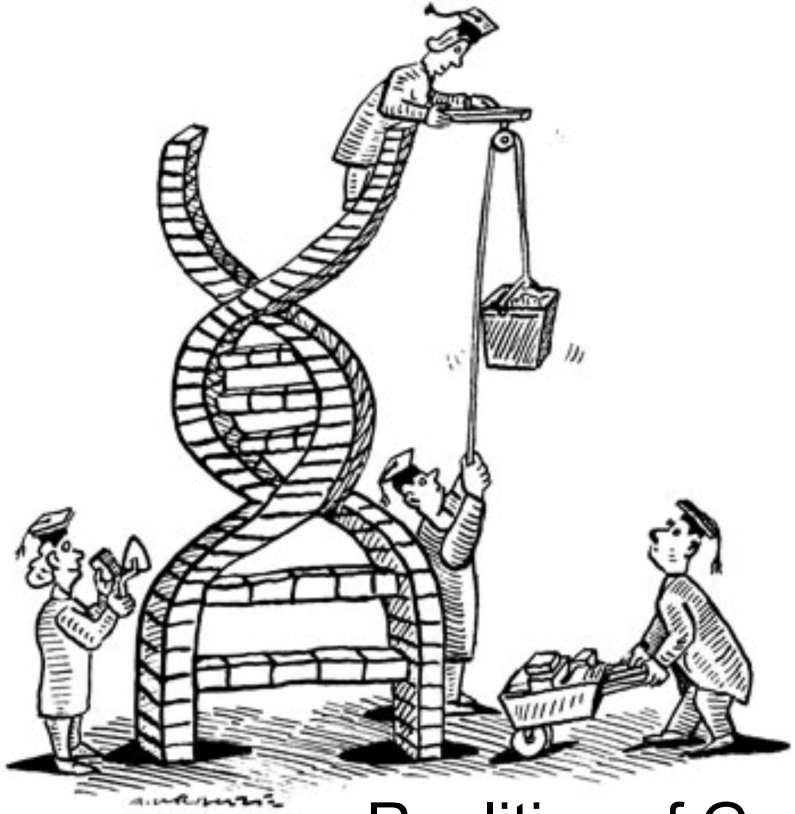
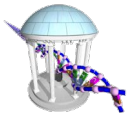
Comp 555 - BioAlgorithms - Spring 2022



Realities of Genome Assembly

- Problem set #2 is online and due Thursday 2/17 (not hard, but takes time)

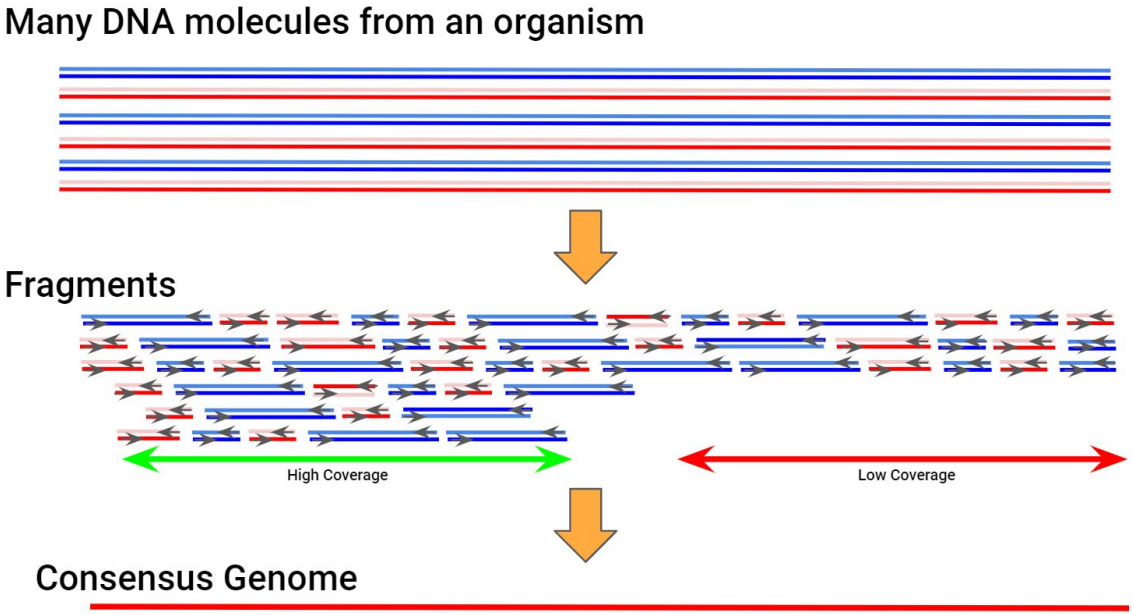
From Last Time



What we learned from a related "Minimal Superstring" problem

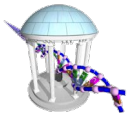
- Can be constructed by finding a Hamiltonian path of an k-dimensional De Bruijn graph over σ symbols
  - Brute-force method is explores all $V!$ paths through V vertices
  - Branch-and-Bound method considers only paths composed of edges in the graph
  - Finding a Hamiltonian path is an NP-complete problem
  - There is no known method that can solve it efficiently as the number of vertices grows
- Can be 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 constraints to be sure that a solution exists
- All but two vertices must be balanced to have an Euler "tour/cycle"
- At most two can be semi-balanced, one with 1 more outgoing edge than incoming the other with one more incoming that outgoing to find a Euler "path"

Returning to the problem of Assembling Genomes



- Extracted DNA is fractured/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

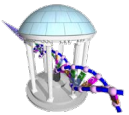


binary3 = {'000', '001', '010', '011', '100', '101', '110', '111'}

```
                 101 100                          111 100
               001 111                          001 101
Solution #1: 0001011100          Solution #2: 0001110100
              000 011                          000 110
                010 110                          011 010
```

- Minimal substring problem
  - Every k-mer is present, (all σ^k)
  - Paths, and there may be multiple, all are solutions
- Read fragments
  - No guarantee that we will ever see every k-mer
  - Can't technically disambiguate repeats except by using heuristics

Recall our “Toy” 20-base genome example

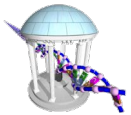


```
GACGGCGGCGCACGGCGCAA    - Our toy 20 base sequence from 2 lectures ago
GACGG   CGCAC
 ACGGC   GCACG
  CGGCG   CACGG         - The complete set of 16 (20-5+1) 5-mers
   GGCGG   ACGGC
    GCGGC   CGGCG
     CGGCG   GGCGC
      GGCGC   GCGCA
       GGCGA   CGCAA
```

Issues:

- Having every *k-mers* is equivalent to *k*× coverage, ignoring boundaries
- Four repeated k-mers {ACGGC, CGGCG, GCGCA, GGCGC}

Some Code

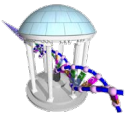


First let's add a function to uniquely label repeated k-mers

In [7]:
```
def kmersUnique(seq, k):
    """ extracts all *k*-mers from *seq* string, while appending a
        unique subscript to each repeated k-mer """
    kmers = sorted([seq[i:i+k] for i in range(len(seq)-k+1)])  # trick is to sort them first making repeats adjacent
    for i in range(1,len(kmers)):
        if (kmers[i] == kmers[i-1][0:k]):           # check adjacent k-mers
            t = kmers[i-1].find('_')
            if (t >= 0):                            # more than 2 repeats
                n = int(kmers[i-1][t+1:]) + 1
                kmers[i] = kmers[i] + "_" + str(n)
            else:                                   # first repeat
                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', 'GCG
GC', 'GGCGC_1', 'GGCGC_2', 'GGCGG']
```

Our Graph class (renamed) from last lecture



In [25]:

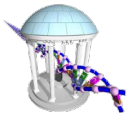
```
import itertools

class Graph:
    def __init__(self, vlist=[
        """ Initialize a Graph
        self.index = {v:i for
        self.vertex = {i:v for
        self.edge = []
        self.edgelabel = []
    def addVertex(self, label)
        """ Add a labeled vert
        index = len(self.index
        self.index[label] = in
        self.vertex[index] = l
    def addEdge(self, vsrc, vd
        """ Add a directed edg
        Repeated edges are dis
        e = (self.index[vsrc],
        if (repeats) or (e not
            self.edge.append(e
            self.edgelabel.app
    def hamiltonianPath(self):
        """ A Brute-force meth
        Basically, all possibl
        for edges. Since edges
        made for *which* versi
        for path in itertools.
            for i in xrange(le
                if ((path[i],p
                    break
            else:
                return [self.v
        return []

    def SearchTree(self, path,
        """ A recursive Branch
        Paths are extended one
        edges from the graph.
        if (len(verticesLeft)
            self.PathV2result
            return True
        for v in verticesLeft:
            if (len(path) == 0
                if self.Search
                    return Tru
        return False
```

```
    def hamiltonianPathV2(s
        """ A wrapper funct
        Hamiltonian Path se
        self.PathV2result =
        self.SearchTree([],
        return self.PathV2r

    def degrees(self):
        """ Returns two dic
        of each node from t
        inDegree = {}
        outDegree = {}
        for src, dst in sel
            outDegree[src]
            inDegree[dst] =
        return inDegree, ou

    def verifyAndGetStart(s
        inDegree, outDegree
        start = 0
        end = 0
        for vert in self.ve
            ins = inDegree.
            outs = outDegre
            if (ins == outs
                continue
            elif (ins - out
                end = vert
            elif (outs - in
                start = ver
            else:
                start, end
                break
        if (start >= 0) and
            return start
        else:
            return -1

    def eulerEdges(self, pa
        edgeId = {}
        for i in xrange(len
            edgeId[self.edg
        edgeList = []
        for i in xrange(len
            edgeList.append
        return edgeList
```

```
    def eulerianPath(self):
        graph = [(src,dst) for src,dst in
        currentVertex = self.verifyAndGet
        path = [currentVertex]
        # "next" is where vertices get in
        # it starts at the end (i.e. it i
        # but later "side-trips" will ins
        next = 1
        while len(graph) > 0:
            for edge in graph:
                if (edge[0] == currentVer
                    currentVertex = edge[
                    graph.remove(edge)
                    path.insert(next, cur
                    next += 1
                    break
            else:
                for edge in graph:
                    try:
                        next = path.index
                        currentVertex = e
                        break
                    except ValueError:
                        continue
                else:
                    print "There is no pa
                    return False
        return path

    def render(self, highlightPath=[]):
        """ Outputs a version of the grap
        using graphviz tools (http://www.
        edgeId = {}
        for i in xrange(len(self.edge)):
            edgeId[self.edge[i]] = edgeId
        edgeSet = set()
        for i in xrange(len(highlightPath
            src = self.index[highlightPat
            dst = self.index[highlightPat
            edgeSet.add(edgeId[src,dst].p
        result = ''
        result += 'digraph {\n'
        result += '   graph [nodesep=2, s
        for index, label in self.vertex.i
            result += '    N%d [shape="bo
```

```
        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'
        return result
```

Finding Paths in our K-mer De Bruijn Graphs



In [8]:

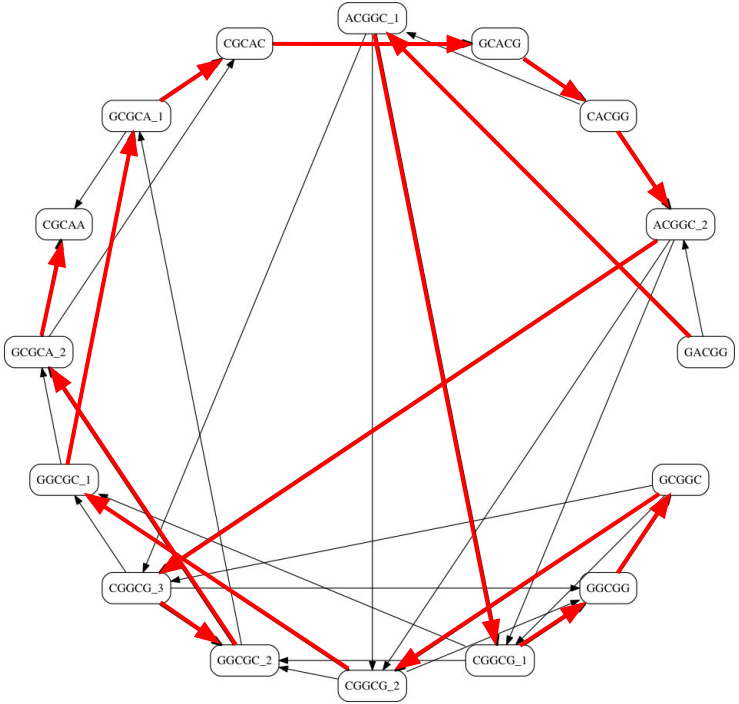
```
k = 5
target = "GACGGCGGCGCACGGCGCAA"
kmers = kmersUnique(target, k)
G1 = Graph(kmers)
for vsrc in kmers:
    for vdst in kmers:
        if (vsrc[1:k] == vdst[0:k-1]):
            G1.addEdge(vsrc,vdst)
path = G1.hamiltonianPathV2()

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

Target: GACGGCGGCGCACGGCGCAA
Result: GACGGCGCACGGCGGCGCAA

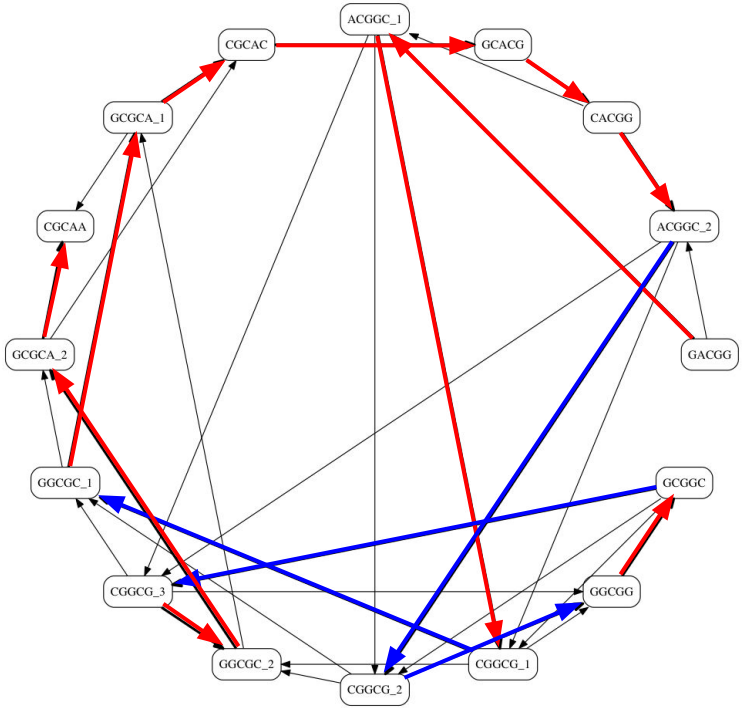
```
['GACGG', 'ACGGC_1', 'CGGCG_1', 'GGCGC_1', 'GCGCA_1', 'CGCAC', 'GCACG', 'CACGG', 'ACGGC_2', 'CGGCG_2', 'GGCGG', 'GCGGC', 'CG
GCG_3', 'GGCGC_2', 'GCGCA_2', 'CGCAA']
GACGGCGCACGGCGGCGCAA
False
```

Not the sequence we expected ...

Let’s look at the resulting graphs

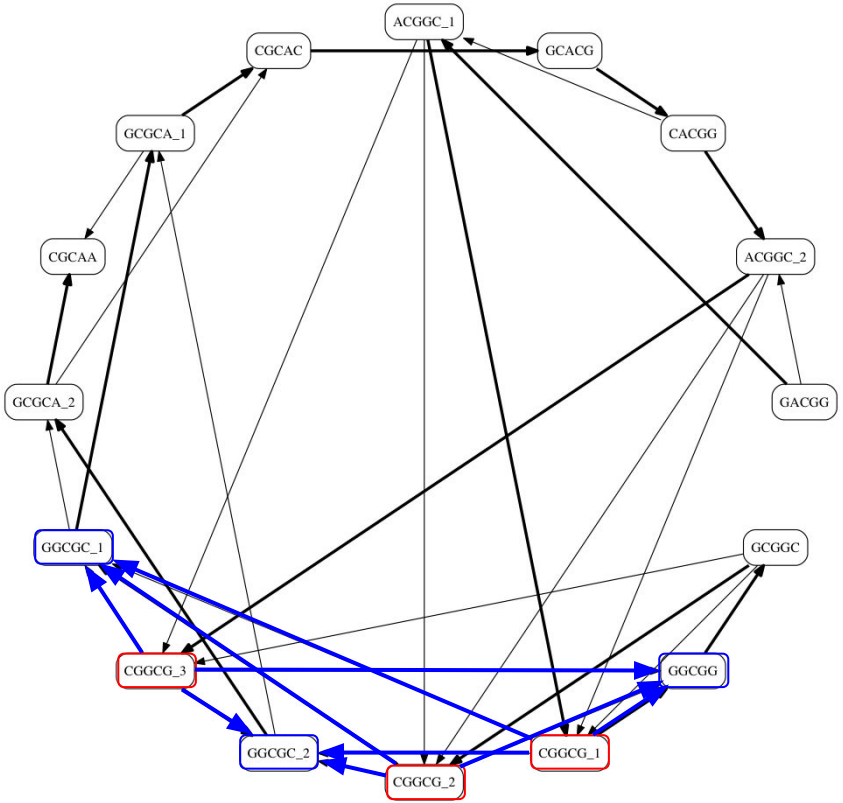


The one we hoped for. Visits $CGGCG_3$ before $CGGCG_2$



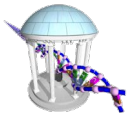
The one we found visits $CGGCG_2$ before $CGGCG_3$

What's the Problem?



- There are many possible Hamiltonian Paths
- How do they differ?
  - There were two possible paths leaving any [CGGCG] node
    - 3 × [CGGCG] → [GGCGC] × 2
    - 3 × [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?

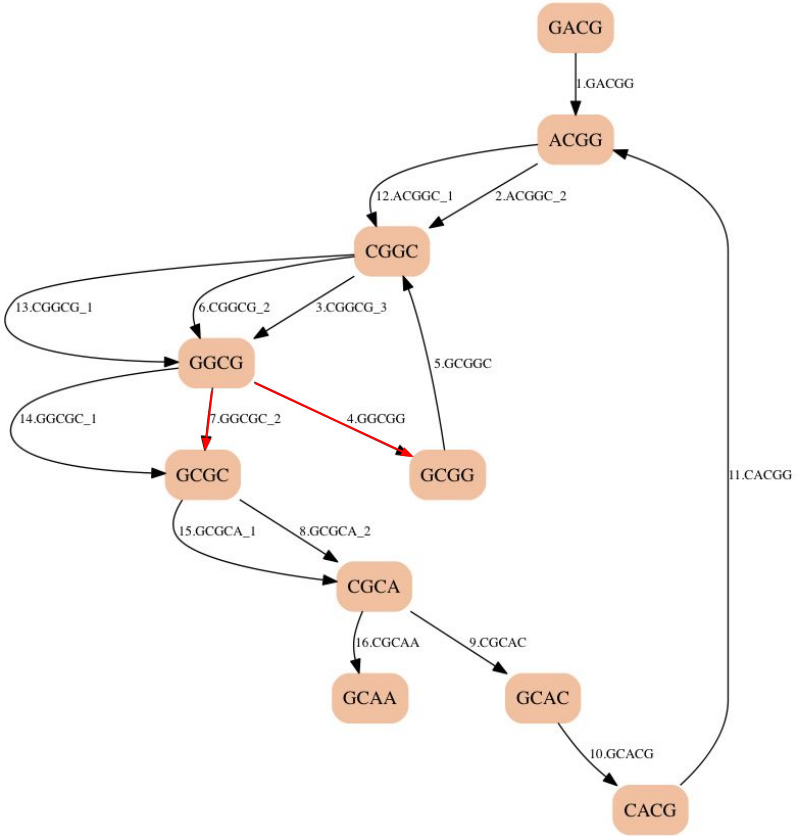


In [20]:

```
k = 5
target = "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', 'CG
GCG_1', 'GGCGC_1', 'GCGCA_1', 'CGCAA']
GACGGCGGCGCACGGCGCAA
True
```

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

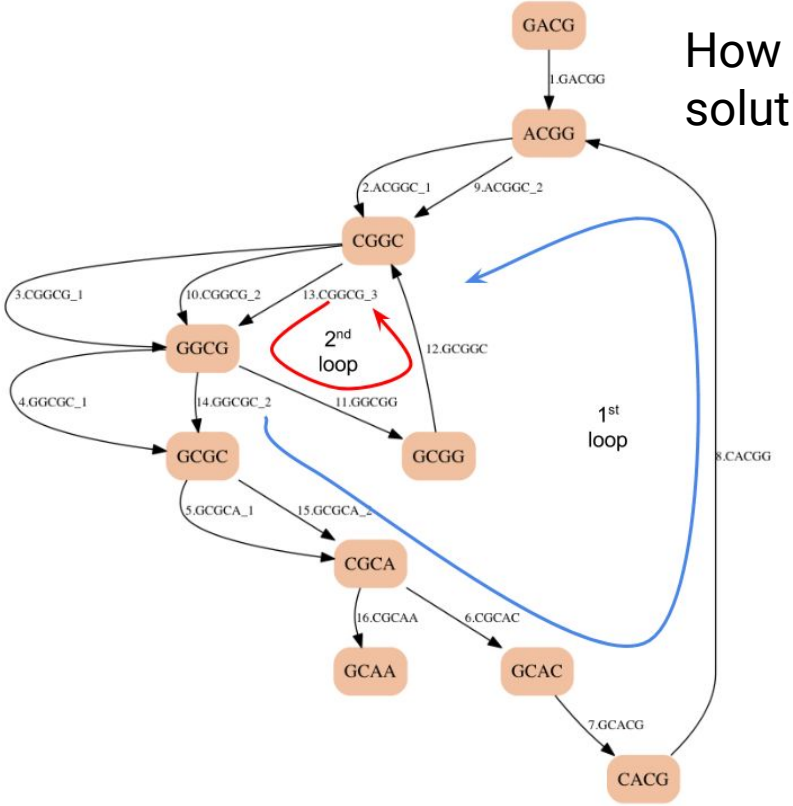


- We got the right answer, but we were lucky.
- There is a path in this graph that matches the Hamiltonian path that we found before

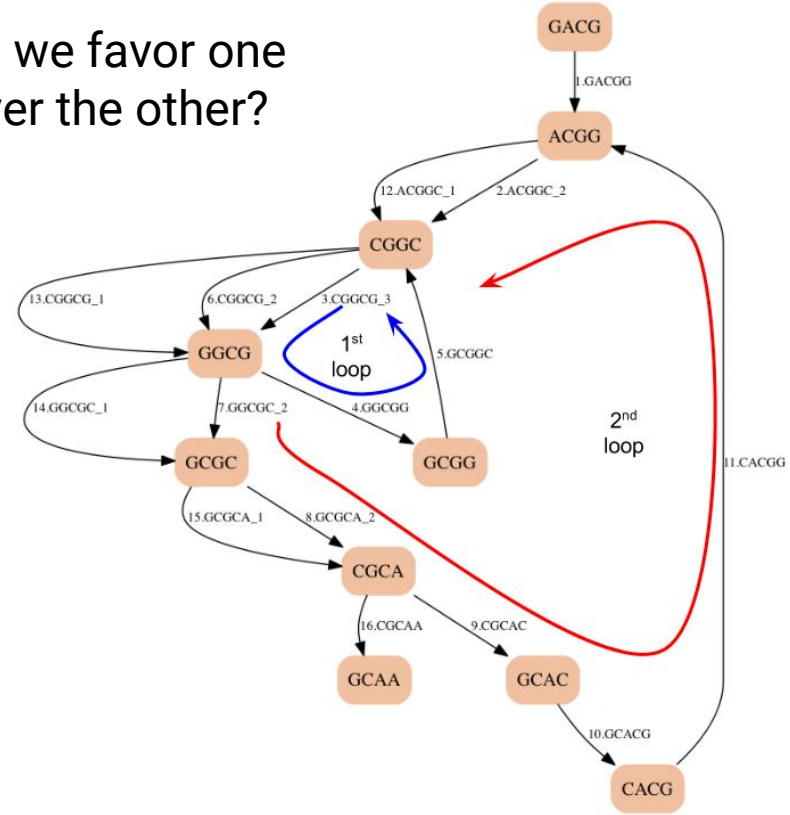
Only when leaving the island "GGCG" do you have a real choice of next islands to visit.

Target: GACGGCGGCGCACGGCGCAA
Result: GACGGCGCACGGCGGCGCAA

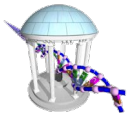
What are the Differences?



How might we favor one solution over the other?



Choose a bigger k-mer



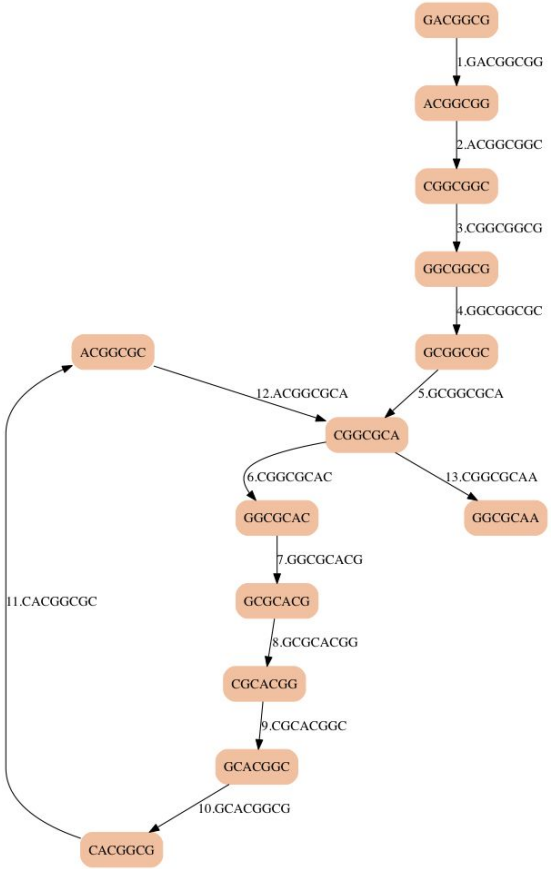
In [22]:

```
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)
```

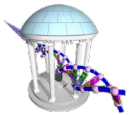
```
['ACGGCGCA', 'ACGGCGGC', 'CACGGCGC', 'CGCACGGC', 'CGGCGCAA', 'CGGCGCAC', 'CGGCGGCG', 'GACGGCGG', 'GCACGGCG', 'GCGCACGG', 'GC
GGCGCA', 'GGCGCACG', 'GGCGGCGC']
['ACGGCGC', 'ACGGCGG', 'CACGGCG', 'CGCACGG', 'CGGCGCA', 'CGGCGGC', 'GACGGCG', 'GCACGGC', 'GCGCACG', 'GCGGCGC', 'GGCGCAA', 'G
GCGCAC', 'GGCGGCG']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



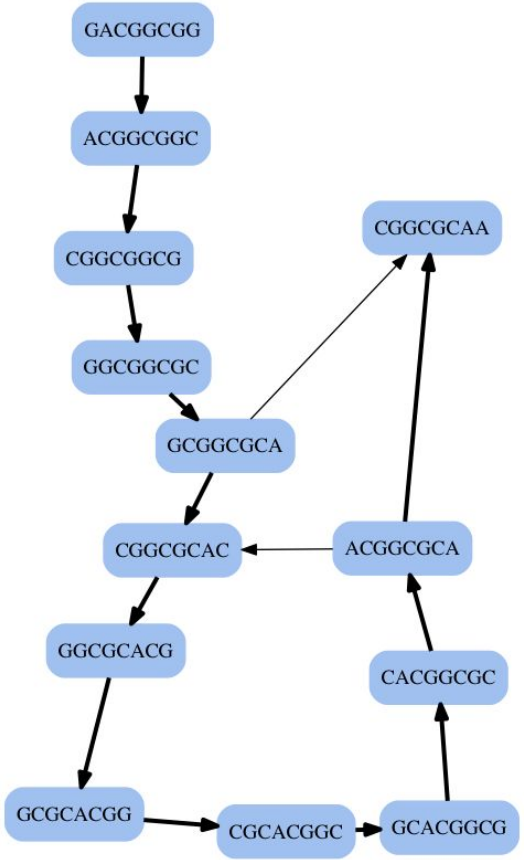
In [23]:

```
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)
```

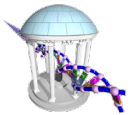
```
['GACGGCGG', 'ACGGCGGC', 'CGGCGGCG', 'GGCGGCGC', 'GCGGCGCA', 'CGGCGCAC', 'GGCGCACG', 'GCGCACGG', 'CGCACGGC', 'GCACGGCG', 'CA
CGGCGC', 'ACGGCGCA', 'CGGCGCAA']
GACGGCGGCGCACGGCGCAA
True
```

Graph with 8-mers as vertices

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



Assembly in Reality



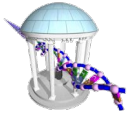
- Problems with repeated k-mers
- We can't distinguish between repeated k-mers
- Recall we knew from our example that were {2:ACGGC, 3:CGGCG, 2:GCGCA, 2:GGCGC}
- Assembling path without repeats:

In [26]:

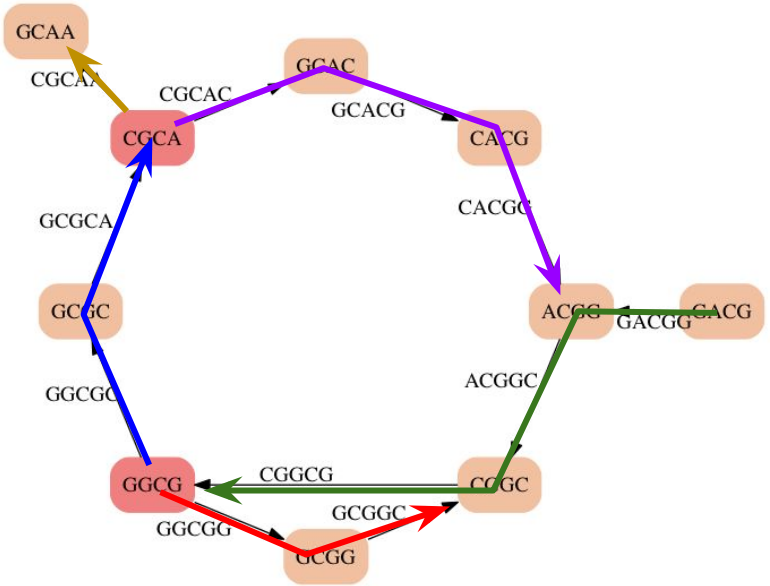
```
k = 5
target = "GACGGCGGCGCACGGCGCAA"
kmers = set([target[i:i+k] for i in range(len(target)-k+1)])
nodes = sorted(set([code[:k-1] for code in kmers] + [code[1:k] for code in kmers]))
G5 = Graph(nodes)
for code in kmers:
   G5.addEdge(code[:k-1],code[1:k],code)

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, 'G
GCG')]
[(9, 8), (3, 9), (1, 0), (4, 0), (6, 1), (8, 3), (0, 3), (2, 5), (7, 2), (2, 6), (9, 7)]
```

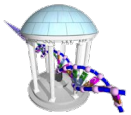
Resulting Graph with "unique" 5-mers as edges



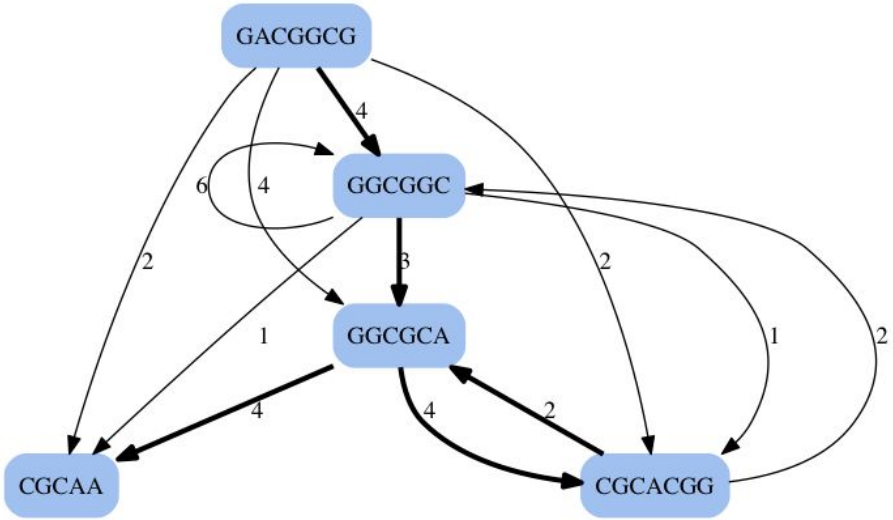
- There is no single Euler Path
- But there are is a set of paths that covers all edges
  ['GACGGCG', 'GGCGGC', 'GGCGCA',
  'CGCAA', 'CGCACGG']
  - Extend a sequence from a node until you reach a node with an out-degree > in-degree
  - Save these partially assembled subsequences, call them *contigs*
  - 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



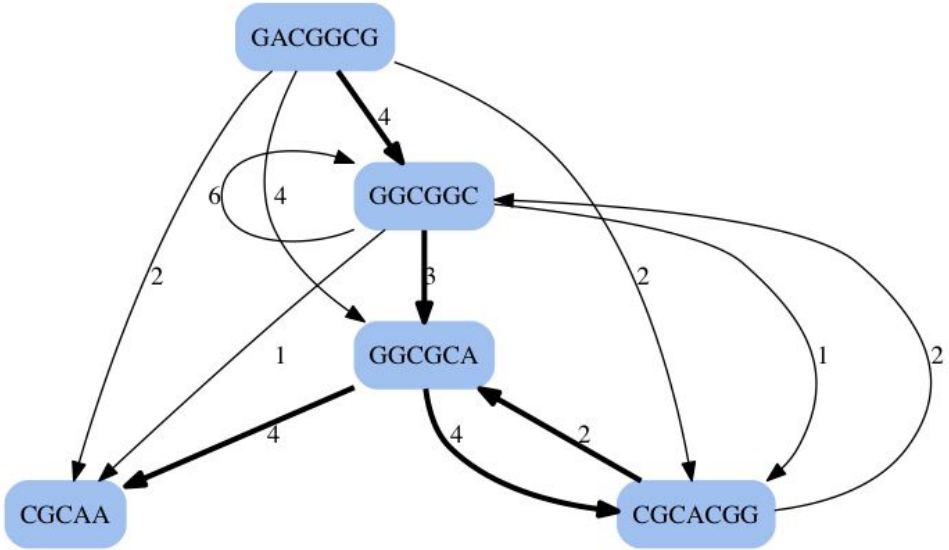
- Usually much smaller than the graph made from k-mers
- Sometimes you can add extra edges to the de Bruijn graph based on coverage

A Heavy Path

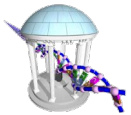
Find the heaviest path touching all vertices in this smaller graph

GACGGCGGCGCACGGCGCAA

GACGGCG	
GGCGGC	4
GGCGCA	3
CGCACGG	4
GGCGCA	2
CGCAA	4
	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.