Comp 555 - BioAlgorithms - Spring 2020

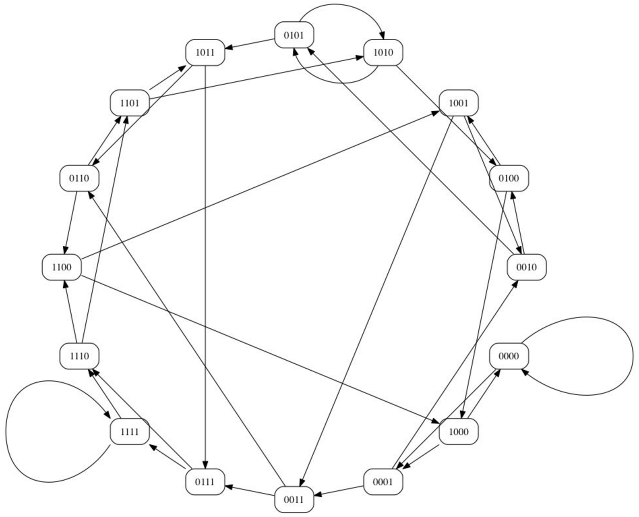
- PROBLEM SET #1 IS DUE TONIGHT JUST BEFORE MIDNIGHT.
- PROBLEM SET #2 SHOULD BE UP BY THEN

Finding Eulerian Paths

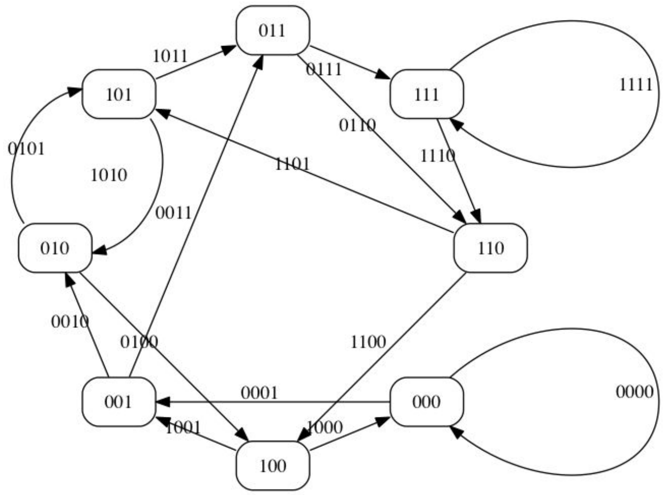
Recall De Bruijn's Problem

Find the shortest string that includes all possible k-mers, from a given alphabet, ∑.

Such "Minimal Superstrings" can be constructed by finding a Hamiltonian path of an *k-dimensional* De Bruijn graph. Defined as a graph with $|\Sigma|^k$ nodes with edges between nodes whose k−1 suffix match another node's k−1 prefix



Or, equivalently, a Eulerian (Edge) cycle of in a *(k−1)-dimensional* De Bruijn graph. Here edges represent the k-length substrings.



De Bruijn's Insight

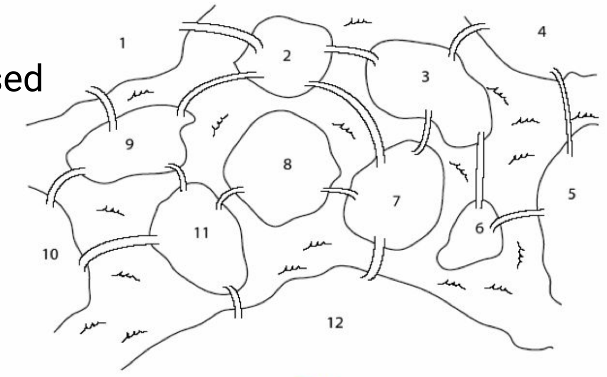
De Bruijn knew that Euler had an ingenous way to solve this problem.

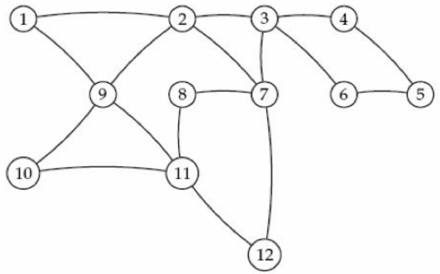
Recall Euler's desire to counstuct a tour where each bridge was crossed only once.

- Start at any vertex v, and follow edges until you return to v
- As long as there exists any vertex u that belongs to the current tour, but has adjacent edges that are not part of the tour
    - Start a new path from u
    - Following unused edges until you return to u
    - Join the new trail to the original tour

He didn't solve the general Hamiltonian Path problem, but he was able to remap Minimal Superstring problem to a simpler problem. Note *every* Minimal Superstring Problem can be fomulated as a Hamiltonian Path in a graph, but the converse is not true. Instead, he found a clever mapping of every Minimal Superstring Problem to a Eulerian Path problem.

Let’s demonstrate using the islands and bridges shown.

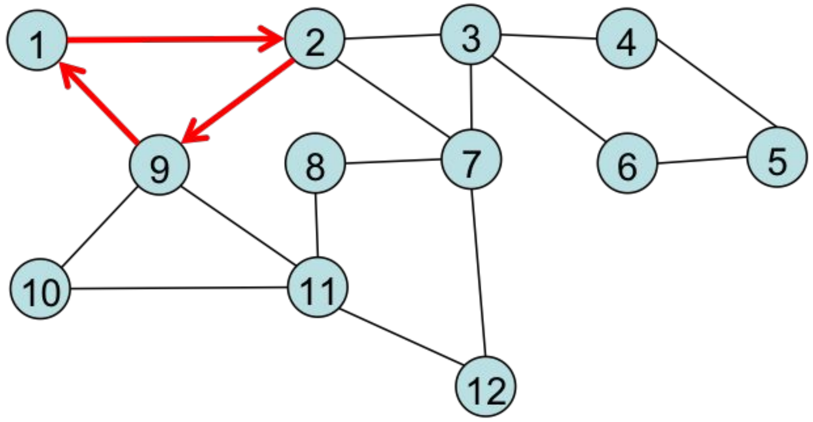




A more complicated Königsberg

An algorithm for finding an Eulerian cycle

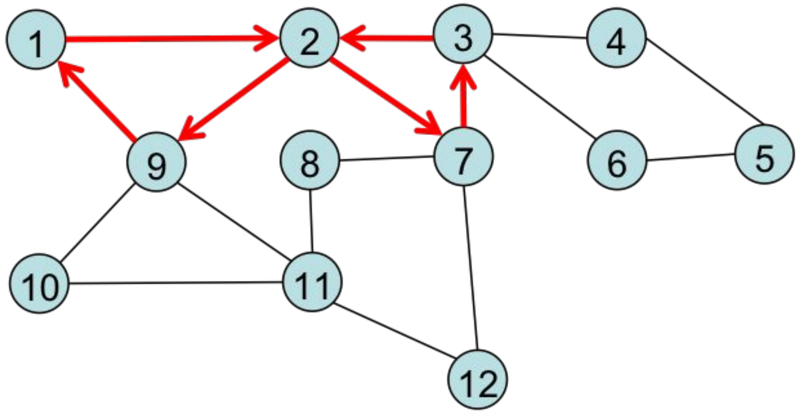
Our first path:



A. 1 → 2 → 9

Take a side-trip

and merge it into our previous path:



B. 2 → 7 → 3 → 2

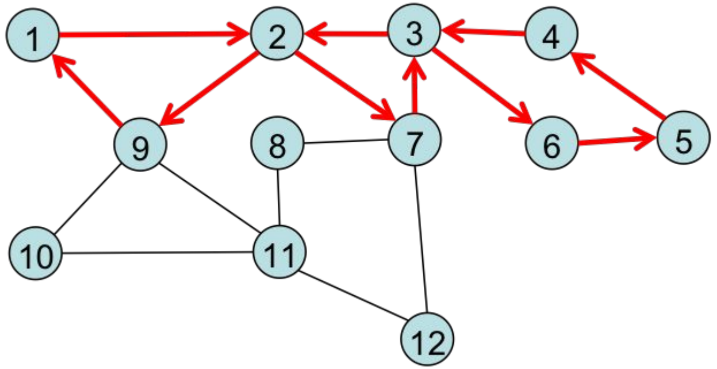
1 → 2 → 9 → 1

2 → 7 → 3 → 2

1 → 2 → 7 → 3 → 2 → 9 → 1

Continue making side trips

merging in a second side-trip:



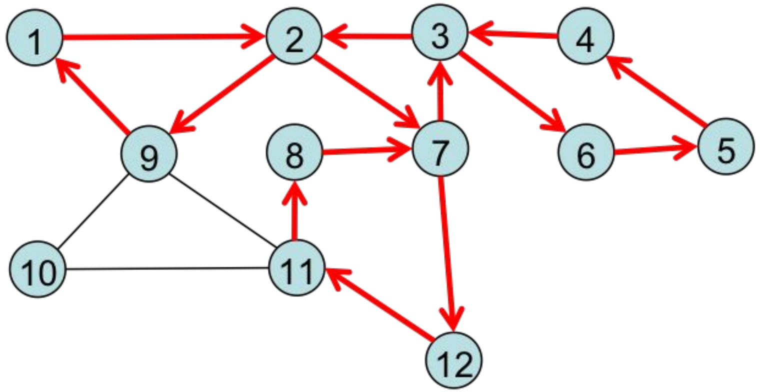
C. 3 → 6 → 5 → 4 → 3

1 → 2 → 7 → 3 → 2 → 9 → 1

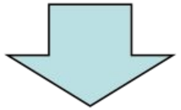
1 → 2 → 7 → 3 → 6 → 5 → 4 → 3 → 2 → 9 → 1

Continue making side trips

merging in a third side-trip:



D. 7 → 12 → 11 → 8 → 7

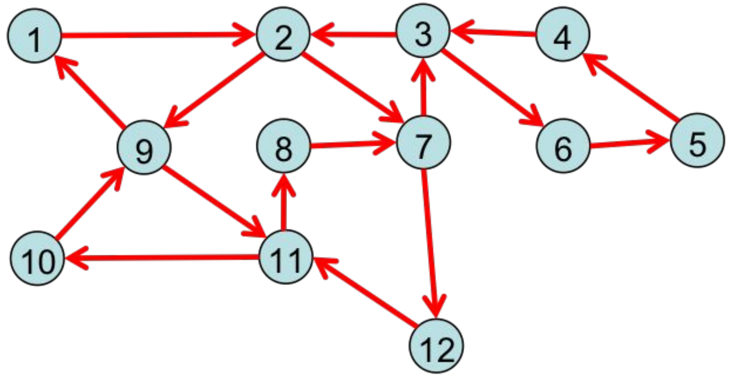


1 → 2 → 7 → 3 → 6 → 5 → 4 → 3 → 2 → 9 → 1

1 → 2 → 7 → 12 → 11 → 8 → 7 → 3 → 6 → 5 → 4 → 3 → 2 → 9 → 1

Repeat until there are no more side trips to take

merging in a final side-trip:



D. 9 → 11 → 10 → 9

This algorithm requires a number of steps that is linear in the number of graph edges, $O(E)$.
The number of edges in a general graph is $E=O(V^2)$ (the adjacency matrix tells us this).

1 → 2 → 7 →12 →11 → 8 → 7 → 3 → 6 → 5 → 4 → 3 → 2 → 9 → 1

1 → 2 → 7 →12 →11 → 8 → 7 → 3 → 6 → 5 → 4 → 3 → 2 →
9 → 11 → 10 → 9 → 1

Converting to code

```python
def eulerianPath(self):
    graph = [(src,dst) for src,dst in self.edge]
    currentVertex = self.verifyAndGetStart()
    path = [currentVertex]
    # "next" is the list index where vertices get inserted into our tour
    # it starts at the end (i.e. same as appending), but later "side-trips" will insert in the middle
    next = 1
    while (len(graph) > 0):                   # when all edges are used, len(graph) == 0
        # follows a path until it ends
        for edge in graph:
            if (edge[0] == currentVertex):
                currentVertex = edge[1]
                graph.remove(edge)
                path.insert(next, currentVertex)  # inserts vertex in path
                next += 1
                break
        else:
            # Look for side-trips along the current path
            for edge in graph:
                try:
                    # insert our side-trip after the "u" vertex that is starts from
                    next = path.index(edge[0]) + 1
                    currentVertex = edge[0]
                    break
                except ValueError:
                    continue
            else:
                print("There is no path!")
                return False
    return path
```

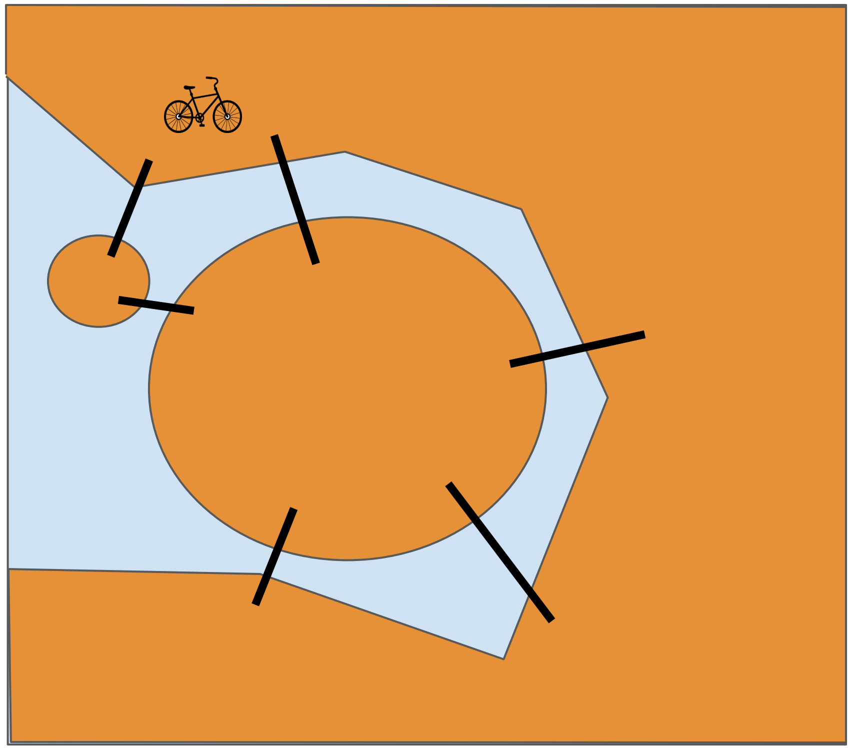
Some issues with our code

- Where do we start our tour?
  (The mysterious VerifyandGetStart() method)
- Where will it end?
- How do we know that each side-trip will rejoin the graph at the same point where it began?
- Will this approach always work?
  If no, when will it fail?
  What conditions are necessary for it to suceed?

It there always a solution?

In our bridge tour example, we mentioned parking our bike, taking a walking tour, (blowing up bridges as we cross them), and then getting back on our bike once the tour is over.

Is there any way to visit all bridges in this example, and still get back to our bike?



Euler's Theorems

A graph is balanced if, for every vertex, the number of incoming edges equals to the number of outgoing edges:

$$in(v)=out(v)$$

Theorem 1: A connected graph has a ***Eulerian Cycle*** if and only if each of its vertices are balanced.

- Sketch of Proof:
- In mid-tour of a valid Euler cycle, there must be a path onto an island and another path off
- This is true until no paths exist
- Thus every vertex must be balanced

Theorem 2: A connected graph has an ***Eulerian Path*** if and only if it contains at exacty two semi-balanced vertices and all others are balanced.

- Exceptions are allowed for the start and end of the tour
- A single start vertex can have one more outgoing path than incoming paths
- A single end vertex can have one more incoming path than outgoing paths

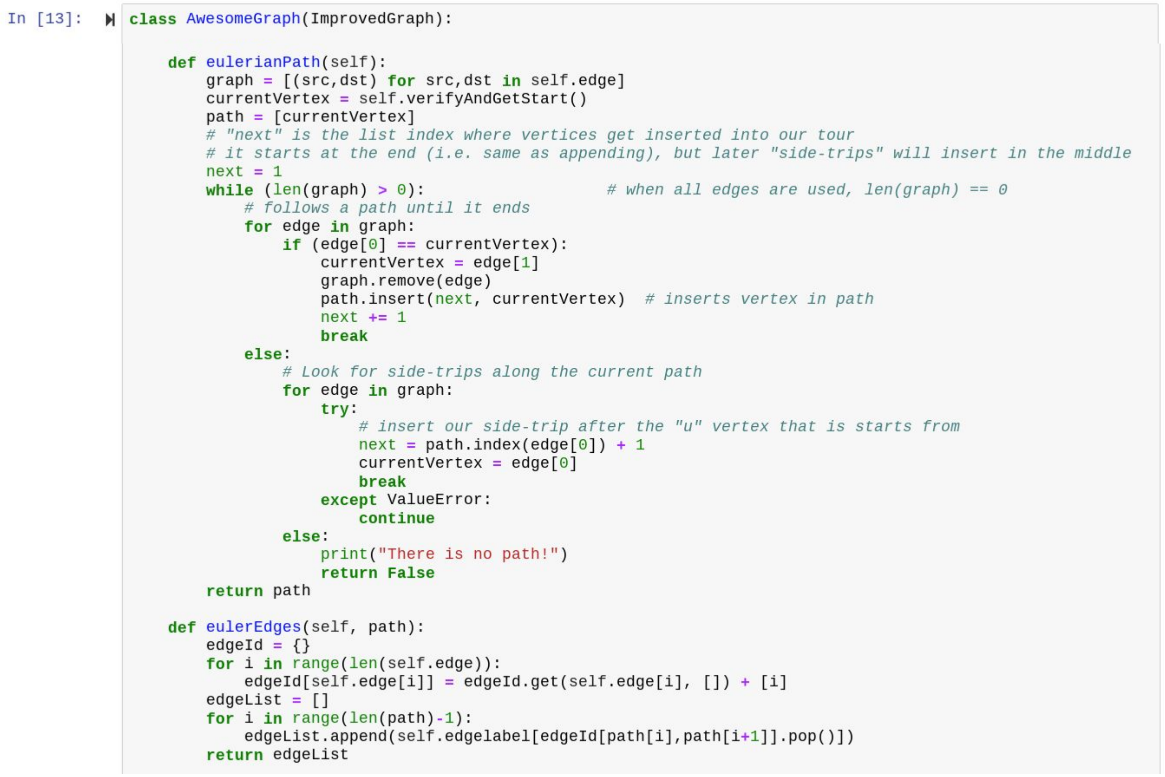
Semi-balanced vertex: $|in(v)-out(v)|=1$

One of the semi-balanced vertices, with $out(v) = in(v)+1$ is the start of the tour.
The other semi-balanced vertex, with $in(v) = out(v)+1$ is the end of the tour

VerifyAndGetStart Code

```python
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, end = 0, 0
    # node 0 will be the starting node is a Euler cycle is found
    for vert in self.vertex:
        ins = inDegree.get(vert,0)
        outs = outDegree.get(vert,0)
        if (ins == outs):
            continue
        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
```

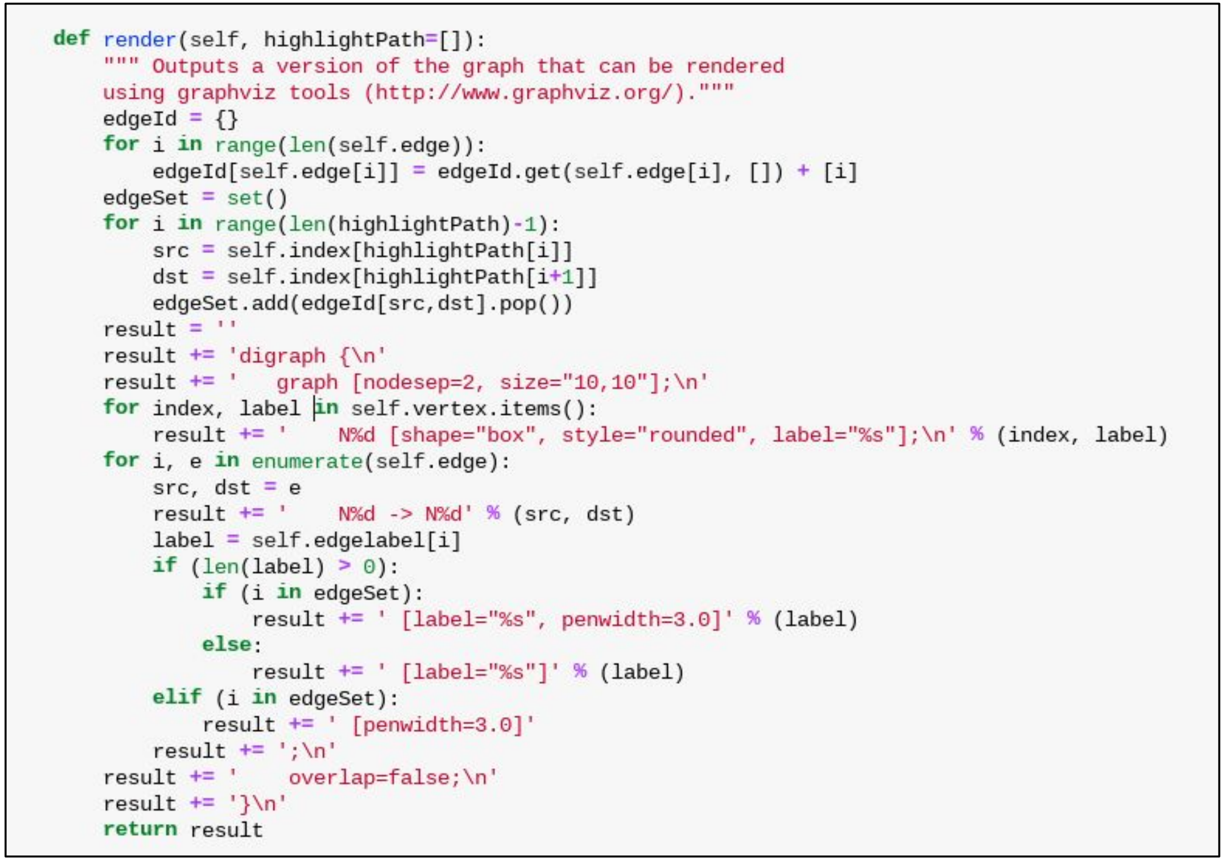
A New Graph Class



```
In [13]:  class AwesomeGraph(ImprovedGraph):

               def eulerianPath(self):
                   graph = [(src,dst) for src,dst in self.edge]
                   currentVertex = self.verifyAndGetStart()
                   path = [currentVertex]
                   # "next" is the list index where vertices get inserted into our tour
                   # it starts at the end (i.e. same as appending), but later "side-trips" will insert in the middle
                   next = 1
                   while (len(graph) > 0):                   # when all edges are used, len(graph) == 0
                       # follows a path until it ends
                       for edge in graph:
                           if (edge[0] == currentVertex):
                               currentVertex = edge[1]
                               graph.remove(edge)
                               path.insert(next, currentVertex)  # inserts vertex in path
                               next += 1
                               break
                       else:
                           # Look for side-trips along the current path
                           for edge in graph:
                               try:
                                   # insert our side-trip after the "u" vertex that is starts from
                                   next = path.index(edge[0]) + 1
                                   currentVertex = edge[0]
                                   break
                               except ValueError:
                                   continue
                           else:
                               print("There is no path!")
                               return False
                   return path

               def eulerEdges(self, path):
                   edgeId = {}
                   for i in range(len(self.edge)):
                       edgeId[self.edge[i]] = edgeId.get(self.edge[i], []) + [i]
                   edgeList = []
                   for i in range(len(path)-1):
                       edgeList.append(self.edgelabel[edgeId[path[i],path[i+1]].pop()])
                   return edgeList
```

Note: I also added an eulerEdges() method to the class. The Eulerian Path algorithm returns a list of vertices along the path, which is consistent with the Hamiltonian Path algorithm. However, in our case, we are less interested in the series of vertices visited than we are the series of edges. Thus, eulerEdges(), returns the edge labels along a path.

A visualization method for the graph



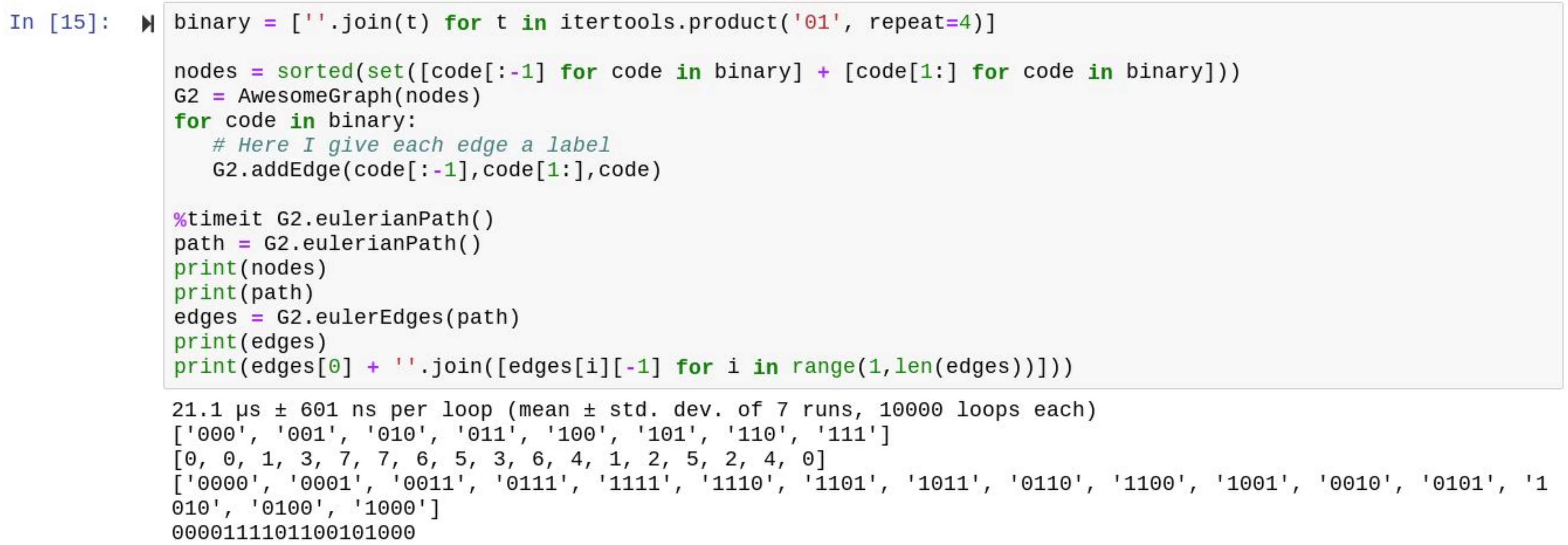
```
def render(self, highlightPath=[]):
    """ Outputs a version of the graph that can be rendered
    using graphviz tools (http://www.graphviz.org/)."""
    edgeId = {}
    for i in range(len(self.edge)):
        edgeId[self.edge[i]] = edgeId.get(self.edge[i], []) + [i]
    edgeSet = set()
    for i in range(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.items():
        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'
    return result
```

Creates a graph description
That can be rendered using
A package called "graphvis"

Available at:

https://www.graphviz.org

Finding Minimal Superstrings with an Euler Path



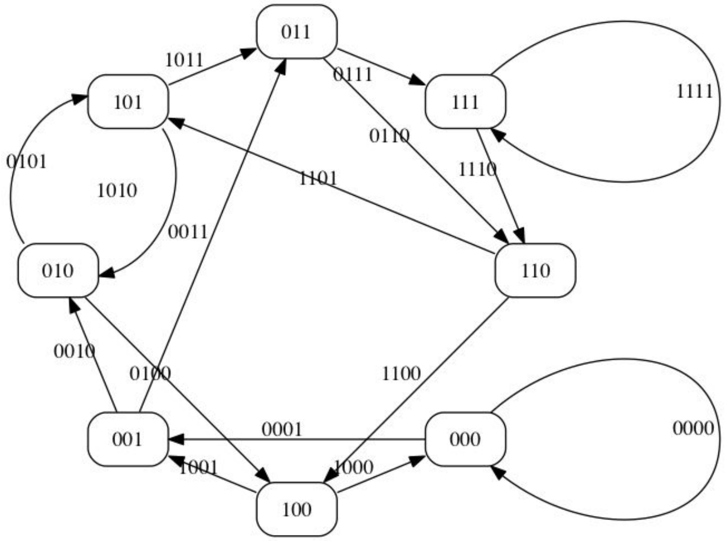
In [15]:

```
binary = [''.join(t) for t in itertools.product('01', repeat=4)]

nodes = sorted(set([code[:-1] for code in binary] + [code[1:] for code in binary]))
G2 = AwesomeGraph(nodes)
for code in binary:
   # Here I give each edge a label
   G2.addEdge(code[:-1],code[1:],code)

%timeit G2.eulerianPath()
path = G2.eulerianPath()
print(nodes)
print(path)
edges = G2.eulerEdges(path)
print(edges)
print(edges[0] + ''.join([edges[i][-1] for i in range(1,len(edges))]))
```

```
21.1 µs ± 601 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
['000', '001', '010', '011', '100', '101', '110', '111']
[0, 0, 1, 3, 7, 7, 6, 5, 3, 6, 4, 1, 2, 5, 2, 4, 0]
['0000', '0001', '0011', '0111', '1111', '1110', '1101', '1011', '0110', '1100', '1001', '0010', '0101', '1
010', '0100', '1000']
0000111101100101000
```

Our graph and its Euler path

- In this case our the graph was fully balanced. So the Euler Path is a cycle.
- Our tour starts arbitarily with the first vertex, '000'



000 → 000 → 001 → 011 → 111 → 111 → 110 → 101 → 011 → 110 → 100 → 001 → 010 → 101 → 010 → 100 → 000

superstring = "0000111101100101000"

Next Time

Back to genome assembly

"We encourage our employees to take a bath here."