Comp 555 - BioAlgorithms - Spring 2022

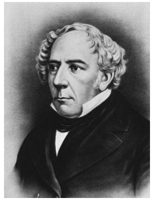
- Graph representations
- Hamiltonian Paths
- De Bruijn Sequences
- Eulerian Tours

- PS#1 is due Tonight before midnight
- Extra Office Hour 3:30-4:30 today
- PS#2 will be posted Today

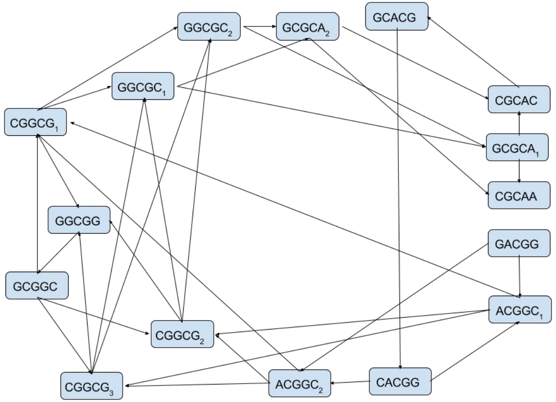
Finding Paths in Graphs

Assembling sequences is a graph problem

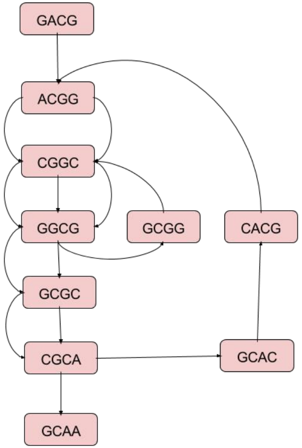
Two graphs representing 5-mers from the sequence "GACGGCGGCGCACGGCGCAA"



Hamiltonian Path:



Eulerian Tour:

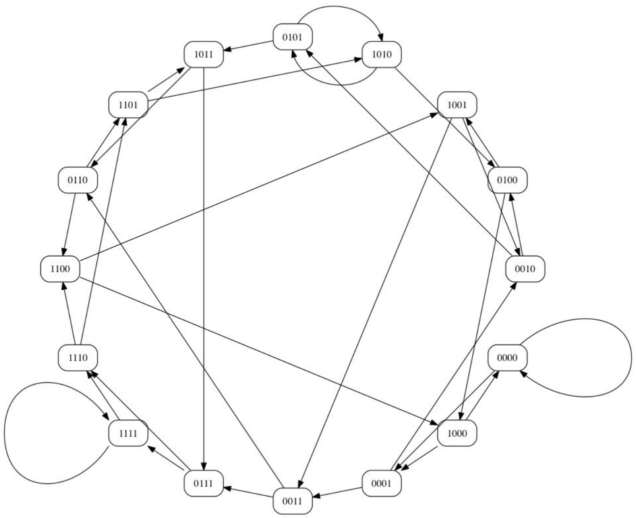


Each k-mer is a vertex. Find a path that passes through every *vertex* of this graph exactly once.

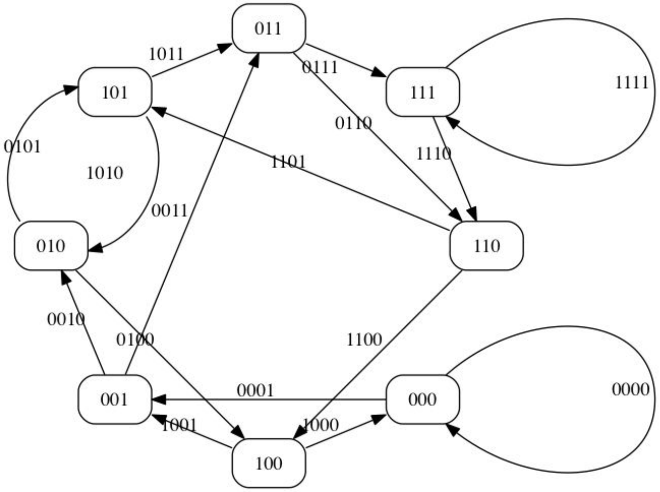
Each k-mer is an edge. Find a path that passes through every *edge* of this graph exactly once.

De Bruijn's Minimal Superstring Problem

Minimal Superstrings can be constructed by finding a Hamiltonian path of an k-dimensional De Bruijn graph. Defined as a graph with $|\Sigma|^k$ knodes and edges from nodes whose k−1 suffix matches a node's k−1 prefix



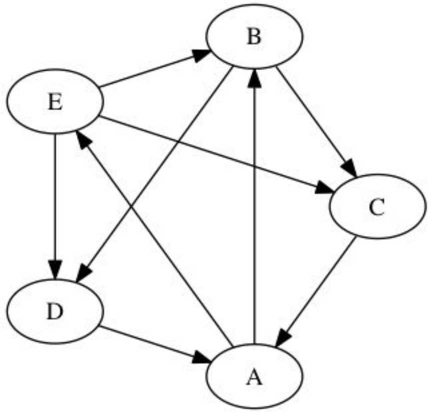
Or, equivalently, a Eulerian cycle of in a (k−1)-dimensional De Bruijn graph. Here edges represent the k-length substrings.



Solving Graph Problems on a Computer

Graph Representations

An example graph:



An Adjacency Matrix:

	A	B	C	D	E
A	0	1	0	0	1
B	0	0	1	1	0
C	1	0	0	0	0
D	1	0	0	0	0
E	0	1	1	1	0

An $n \times n$ matrix where A_{ij} is 1 if there is an edge connecting the ith vertex to the j^{th} vertex and 0 otherwise.

Adjacency Lists:

NodeName = ['A', 'B', 'C', 'D', 'E']

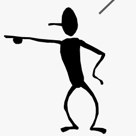
Edge = [(0,1), (0,4),
(1,2), (1,3),
(2,0),
(3,0),
(4,1), (4,2), (4,3)]

An array or list of vertex pairs *(i,j)* indicating an edge from the i^{th} vertex to the j^{th} vertex.

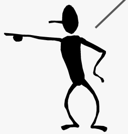
An adjacency list graph object

In [1]:

```
class BasicGraph:
    def __init__(self, vlist=[]):
        """ Initialize a Graph with an optional vertex list """
        self.index = {v:i for i,v in enumerate(vlist)}    # looks up index given name
        self.vertex = {i:v for i,v in enumerate(vlist)}   # looks up name given index
        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)
```



This graph class implements an adjacency list. Each vertex has a "Name" and an "index" that is used internally. Notice the trick here of how each node's index is created. This wouldn't be ideal for graphs that need to remove vertices.



Edges connect vertices by "name", but can have their own "label"

Usage example

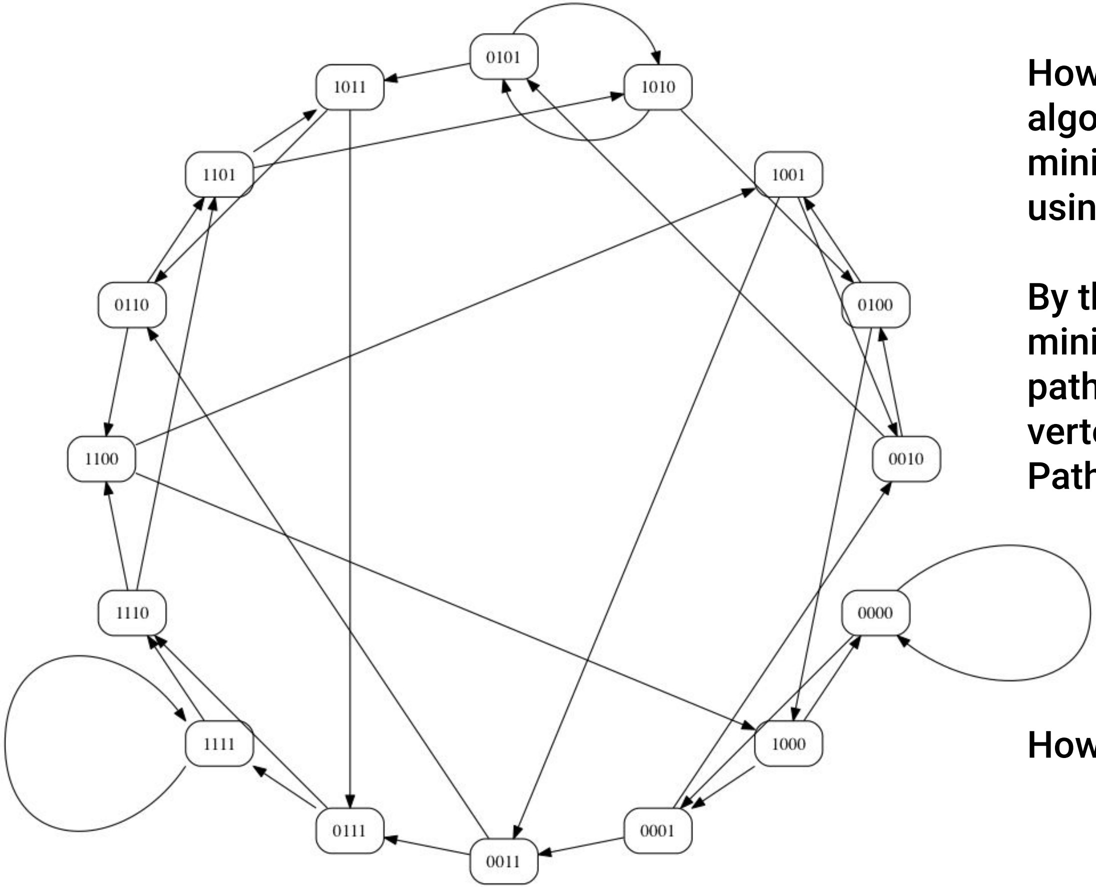
Let's generate the vertices needed to find De Bruijn's superstring of 4-bit binary strings...
and create a graph object using them.

In [17]:

```python
import itertools

# build a list of binary number "strings"
binary = [''.join(t) for t in itertools.product('01', repeat=4)]

print(binary)

# build a graph with edges connecting binary strings where
# the k-1 suffix of the source vertex matches the k-1 prefix
# of the destination vertex
G1 = BasicGraph(binary)
for vsrc in binary:
    G1.addEdge(vsrc, vsrc[1:]+'0')
    G1.addEdge(vsrc, vsrc[1:]+'1')

print()
print("Vertex indices = ", G1.index)
print()
print("Index to Vertex = ", G1.vertex)
print()
print("Edges =", G1.edge)

for i, (src, dst) in enumerate(G1.edge):
    print("%2d: %s --> %s" % (i, G1.vertex[src], G1.vertex[dst]), end = "  ")
    if (i % 4 == 3):
        print()
```

```
['0000', '0001', '0010', '0011', '0100', '0101', '0110', '0111', '1000', '1001', '1010', '1011', '1100', '1101', '1110',
'1111']

Vertex indices =  {'0000': 0, '0001': 1, '0010': 2, '0011': 3, '0100': 4, '0101': 5, '0110': 6, '0111': 7, '1000': 8, '100
1': 9, '1010': 10, '1011': 11, '1100': 12, '1101': 13, '1110': 14, '1111': 15}

Index to Vertex =  {0: '0000', 1: '0001', 2: '0010', 3: '0011', 4: '0100', 5: '0101', 6: '0110', 7: '0111', 8: '1000', 9:
'1001', 10: '1010', 11: '1011', 12: '1100', 13: '1101', 14: '1110', 15: '1111'}

Edges = [(0, 0), (0, 1), (1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7), (4, 8), (4, 9), (5, 10), (5, 11), (6, 12), (6, 1
3), (7, 14), (7, 15), (8, 0), (8, 1), (9, 2), (9, 3), (10, 4), (10, 5), (11, 6), (11, 7), (12, 8), (12, 9), (13, 10), (13,
11), (14, 12), (14, 13), (15, 14), (15, 15)]
 0: 0000 --> 0000   1: 0000 --> 0001   2: 0001 --> 0010   3: 0001 --> 0011
 4: 0010 --> 0100   5: 0010 --> 0101   6: 0011 --> 0110   7: 0011 --> 0111
 8: 0100 --> 1000   9: 0100 --> 1001  10: 0101 --> 1010  11: 0101 --> 1011
12: 0110 --> 1100  13: 0110 --> 1101  14: 0111 --> 1110  15: 0111 --> 1111
16: 1000 --> 0000  17: 1000 --> 0001  18: 1001 --> 0010  19: 1001 --> 0011
20: 1010 --> 0100  21: 1010 --> 0101  22: 1011 --> 0110  23: 1011 --> 0111
24: 1100 --> 1000  25: 1100 --> 1001  26: 1101 --> 1010  27: 1101 --> 1011
28: 1110 --> 1100  29: 1110 --> 1101  30: 1111 --> 1110  31: 1111 --> 1111
```

The resulting graph



How would we write an algorithm to find the minimal superstring using this graph?

By this construction, the minimal superstring is a path through every vertex, a "Hamiltonian Path"

How do we find one?

The Hamiltonian Path Problem

Next, we need an algorithm to find a path in a graph that visits every node exactly once, if such a path exists.

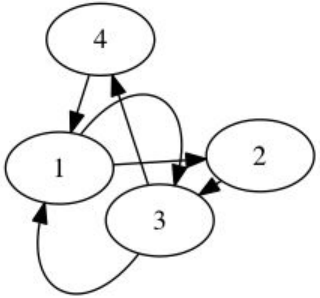
How?

Approach:

- Enumerate every possible path (all permutations of N vertices).
  Python's `itertools.permutations()` does this.
- Verify that there is an edge connecting all N-1 pairs of adjacent vertices

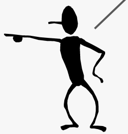
All vertex permutations = *every* possible path

A simple graph with 4 vertices




```
In [5]:  import itertools

         start = 1
         for path in itertools.permutations([1,2,3,4]):
             if (path[0] != start):
                 print()
                 start = path[0]
             print(path, end=', ')
```

```
(1, 2, 3, 4), (1, 2, 4, 3), (1, 3, 2, 4), (1, 3, 4, 2), (1, 4, 2, 3), (1, 4, 3, 2),
(2, 1, 3, 4), (2, 1, 4, 3), (2, 3, 1, 4), (2, 3, 4, 1), (2, 4, 1, 3), (2, 4, 3, 1),
(3, 1, 2, 4), (3, 1, 4, 2), (3, 2, 1, 4), (3, 2, 4, 1), (3, 4, 1, 2), (3, 4, 2, 1),
(4, 1, 2, 3), (4, 1, 3, 2), (4, 2, 1, 3), (4, 2, 3, 1), (4, 3, 1, 2), (4, 3, 2, 1),
```



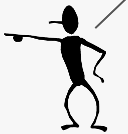
Permutations is another iterator in our frienemy, itertools package. It generates all N! permutations of a given list or set.

A Hamiltonian Path Algorithm

- Test each vertex permutation to see if it is a valid path
- Let's extend our BasicGraph into an EnhancedGraph class
- Create the superstring graph and find a Hamiltonian Path

In [10]:

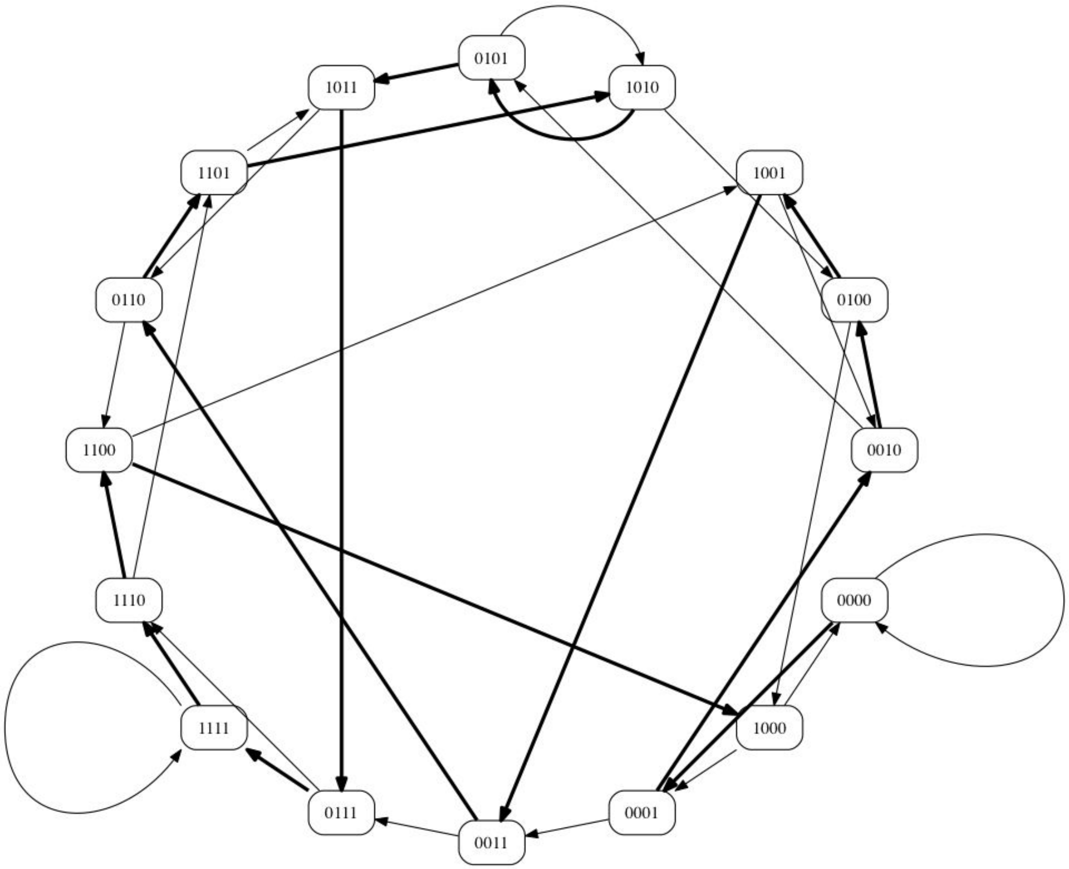
```python
import itertools

class EnhancedGraph(BasicGraph):
    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 range(len(path)-1):
                if ((path[i],path[i+1]) not in self.edge):
                    break
            else:
                return [self.vertex[i] for i in path]
        return []

G1 = EnhancedGraph(binary)
for vsrc in binary:
    G1.addEdge(vsrc,vsrc[1:]+'0')
    G1.addEdge(vsrc,vsrc[1:]+'1')

# WARNING: takes about 20 mins
%time path = G1.hamiltonianPath()
print(path)
superstring = path[0] + ''.join([path[i][3] for i in range(1,len(path))])
print(superstring)
```



Note that this code exits once it finds *any* Hamiltonian path.. It makes no attempt to find every one possible.

```
CPU times: user 18min 11s, sys: 52 ms, total: 18min 11s
Wall time: 18min 11s
['0000', '0001', '0010', '0100', '1001', '0011', '0110', '1101', '1010', '0101', '1011', '0111', '1111', '1
110', '1100', '1000']
0000100110101111000
```

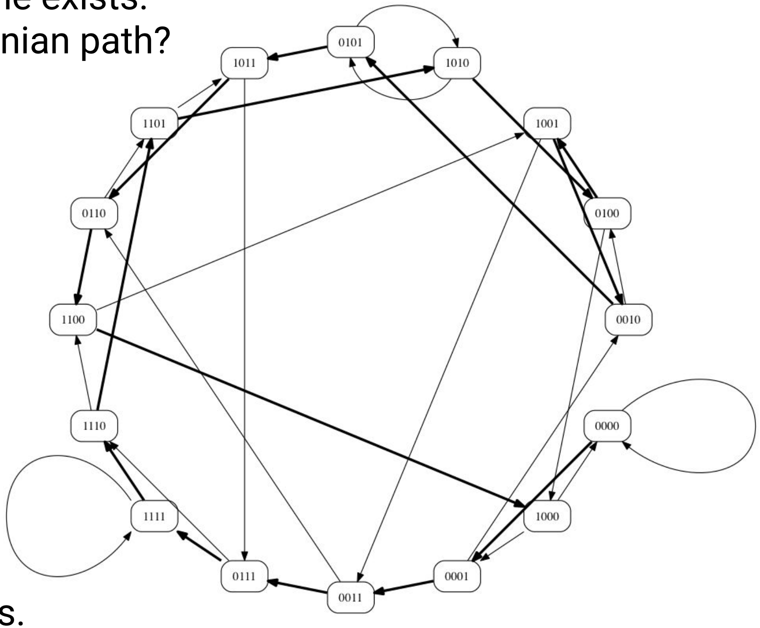
Visualizing the result



Is this solution unique?

How about the path = "0000111101001011000"

- Our Hamiltonian path finder produces a single path, if one exists.
- How would you modify it to produce every valid Hamiltonian path?
- How long would that take?



One of De Bruijn's contributions is that there are:

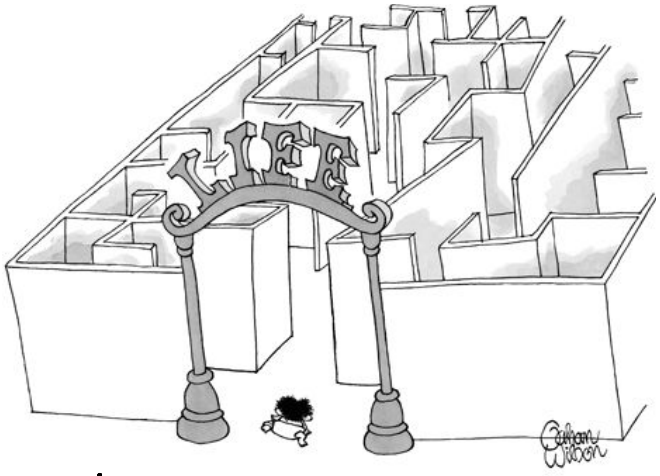
$$\frac{(\sigma!)^{\sigma^{k-1}}}{\sigma^k}$$

paths leading to superstrings where $\sigma=|\Sigma|$.

In our case $\sigma=2$ and $k = 4$, so there should be $2^8 / 2^4 = 16$ paths.
(ignoring those that are just different starting points on the same cycle)

Brute Force is slow!

- There are N! possible paths for N vertices.
- Our 16 vertices give 20,922,789,888,000 possible paths
- There is a fairly simple *Branch-and-Bound* evaluation strategy
  - Extend paths using only valid edges
  - Use recursion to extend paths along graph edges
  - Trick is to maintain two lists:
    - The path so far, where each adjacent pair of vertices is connected by an edge
    - Unused vertices. When the unused list becomes empty we've found a path



A Branch-and-Bound Hamiltonian Path Finder

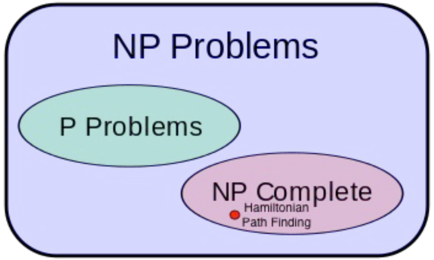
In [9]:

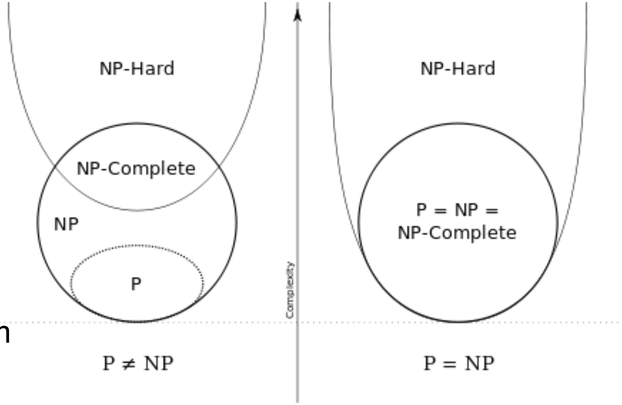
```
import itertools

class ImprovedGraph(BasicGraph):

    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 hamiltonianPath(self):
        """ A wrapper function for invoking the Branch-and-Bound
        Hamiltonian Path search. """
        self.PathV2result = []
        self.SearchTree([],sorted(self.index.values()))
        return self.PathV2result

G1 = ImprovedGraph(binary)
for vsrc in binary:
    G1.addEdge(vsrc,vsrc[1:]+'0')
    G1.addEdge(vsrc,vsrc[1:]+'1')
%timeit path = G1.hamiltonianPath()
path = G1.hamiltonianPath()
print(path)
superstring = path[0] + ''.join([path[i][3] for i in range(1,len(path))])
print(superstring)
```

```
81 µs ± 684 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
['0000', '0001', '0010', '0100', '1001', '0011', '0110', '1101', '1010', '0101', '1011', '0111', '1111', '1
110', '1100', '1000']
0000100110101111000
```

Is there a better Hamiltonian Path Algorithm?

- Better in what sense?
- Better = number of steps to find a solution that is polynomial in either the number of edges or vertices
    - Polynomial: $variable^{constant}$
    - Exponential: $constant^{variable}$ or worse, $variable^{variable}$
    - For example our Brute-Force algorithm was $O(k^V)$ < **O(V!)** < $O(V^V)$ where *V* is the number of vertices in our graph, a problem variable
- We can only practically solve only small problems if the algorithm for solving them takes a number of steps that grows exponentially with a problem variable (i.e. the number of vertices), or else be satisfied with heuristic or *approximate* solutions
- Can we *prove there is no algorithm* to find a Hamiltonian Path in a time that is polynomial in the number of vertices or edges in the graph?
    - No one has, and here is a [million-dollar reward]() if you can!
    - If instead of a *brute* who just enumerates all possible answers we knew an *oracle* could just tell us the right answer (i.e. *Nondeterministically*)
    - It's easy to verify that an answer is correct in *Polynomial* time.
    - A lot of known problems will suddenly become solvable using your algorithm

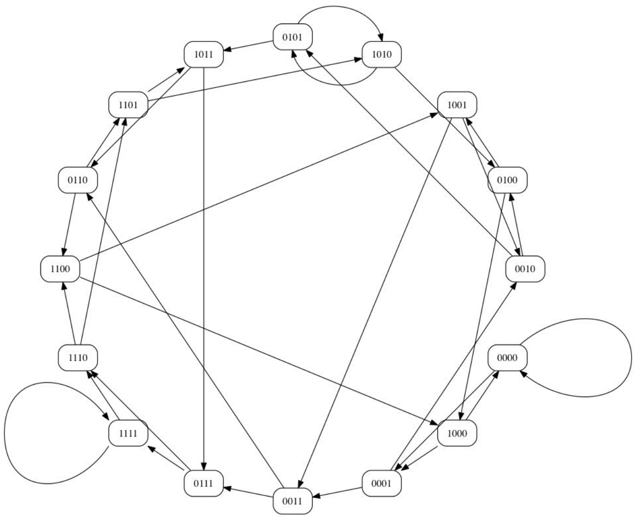




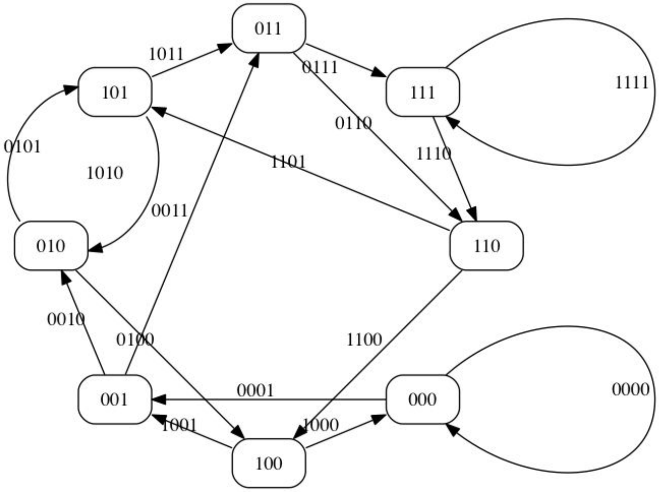
Recall De Bruijn's Problem

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

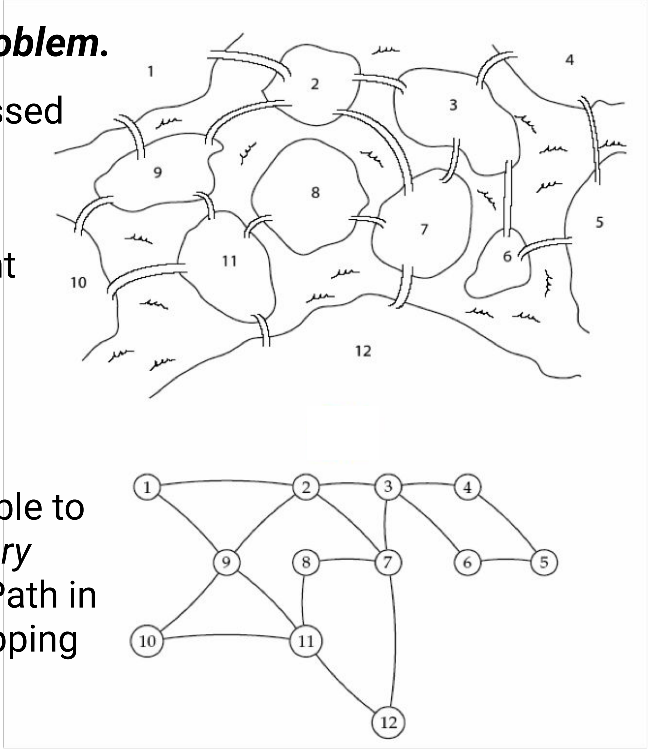
De Bruijn knew that Euler had an ingenious way to solve this problem.

Recall Euler's desire to construct 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 formulated 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.

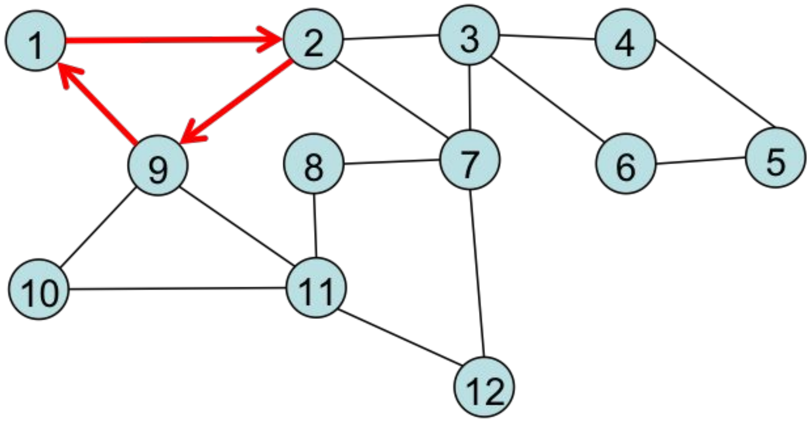
Let’s demonstrate using the islands and bridges shown.



A more complicated Königsberg

An algorithm for finding an Eulerian cycle

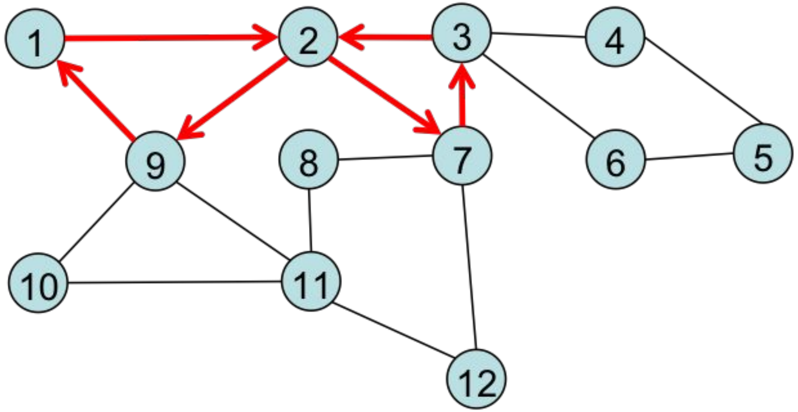
Our first path:



A. 1 → 2 → 9

Take a side-trip

and merge it into our previous path:



B. 2 → 7 → 3 → 2

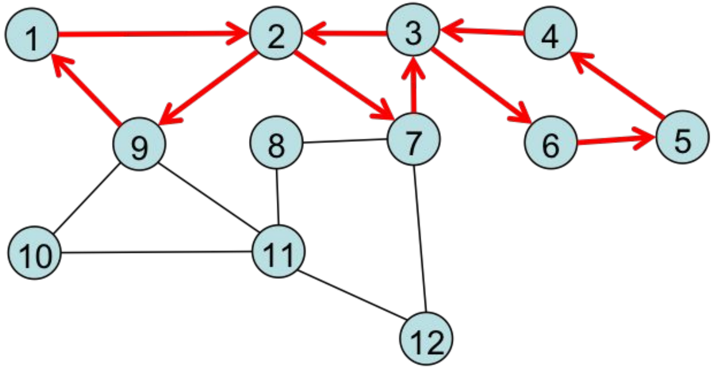
1 → 2 → 9 → 1

2 → 7 → 3 → 2

1 → 2 → 7 → 3 → 2 → 9 → 1

Continue making side trips

merging in a second side-trip:



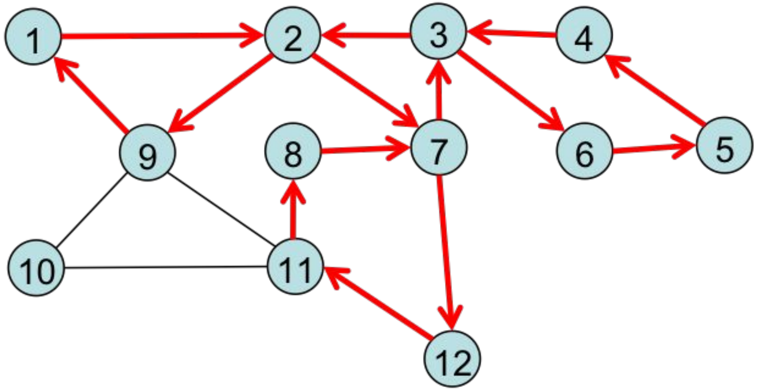
C. 3 → 6 → 5 → 4 → 3

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:



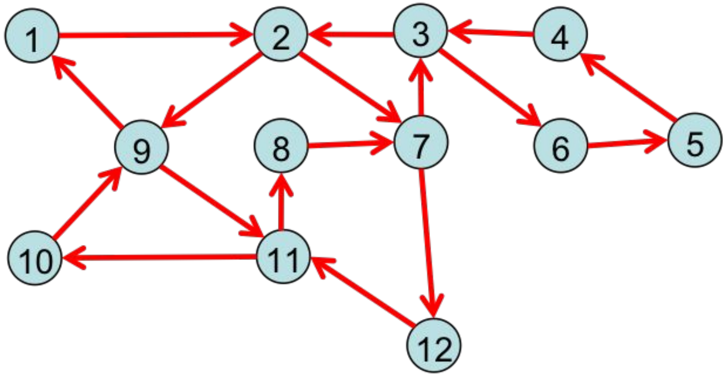
D. 7 → 12 → 11 → 8 → 7

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

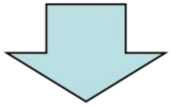
Repeat until there are no more side trips to take

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
```

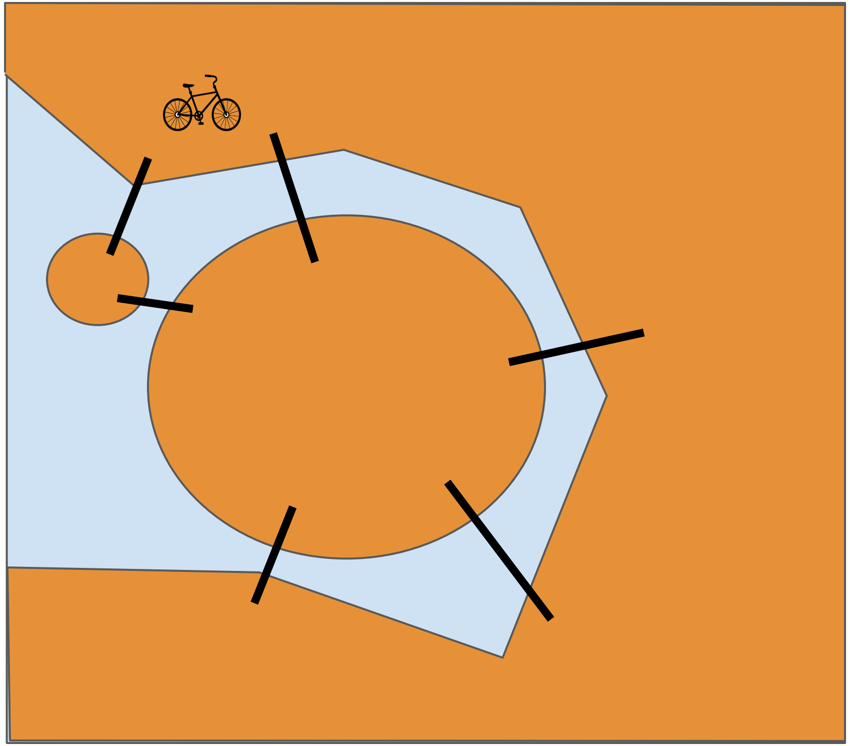
Some issues with our code

- 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 succeed?

Is 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 all 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 *exactly two semi-balanced vertices* and all others are balanced.

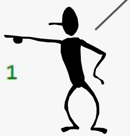
- Exceptions are allowed for the start and end of the tour
- A single start vertex can have one more outgoing path than incoming paths
- A single end vertex can have one more incoming path than outgoing paths

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) and (end == 0):
            end = vert
        elif (outs - ins == 1) and (start == 0):
            start = vert
        else:
            start, end = -1, -1
            break
    if (start >= 0) and (end >= 0):
        return start
    else:
        return -1
```



The "degree" of a vertex is a measure of how many edges it has. For a directed graph it makes sense to consider how many edges enter the node (its in-degree) and how many leave the node (its out-degree).

There's something subtle going on here that might make a good problem set or exam question

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

```python
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]:

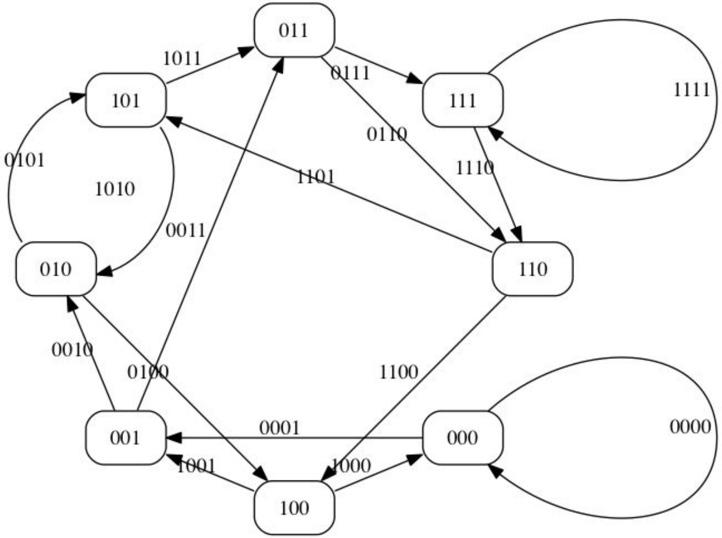
```
binary = [''.join(t) for t in itertools.product('01', repeat=4)]

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
```

Recall this took over 18 mins using the Hamiltonian path approach, and 81 µs with branch-and-bound

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 arbitrarily 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."