Comp 555 - BioAlgorithms - Spring 2020

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

- Problem set #1 is due next Tuesday.

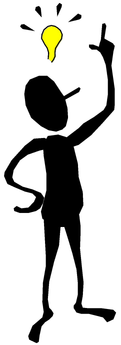
Finding Paths in Graphs

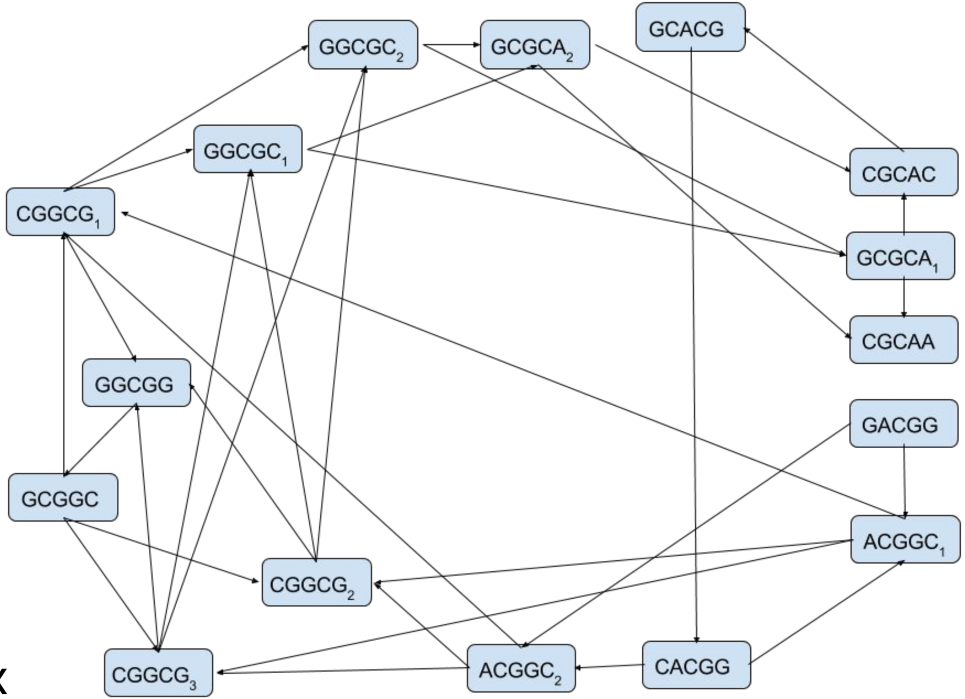
From Last Time

We discussed how to turn a sequence into a graph

```
GACGGCGGCGCACGGCGCAA
GACGG
 ACGGC
  CGGCG
   GGCGG
    GCGGC
     CGGCG
      GGCGC
       GCGCA
        CGCAC
         GCACG
          CACGG
           ACGGC
            CGGCG
             GGCGC
              GCGCA
               CGCAA
```

Our original sequence is just a path in this graph. How would you find it?





By placing edges connecting k-mers whose k-1 suffix matches a k-1prefix

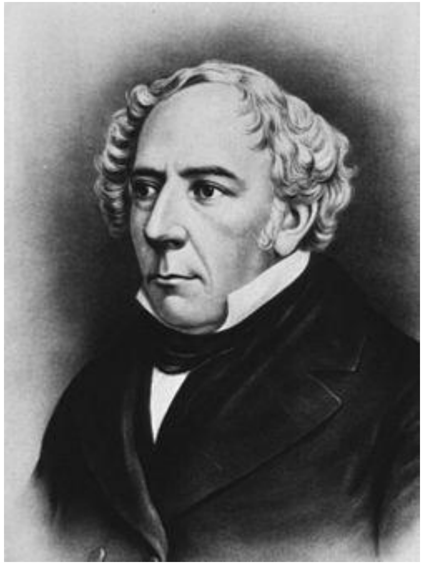
Parlor games

Once finding paths in graphs was a popular form of entertainment…

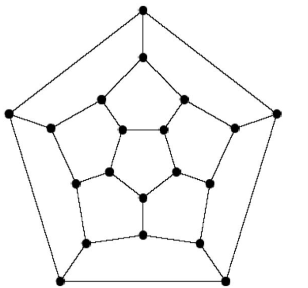
Graphs would be printed in newspapers, and people would try to find paths in them as a game.

The rules of our game

- Every node, k-mer, can be used exactly once
- The object is to find a path along edges that visits every node one time
- This game was invented in the mid 1800's by a mathematician called ***Sir William Hamilton***



An example of Hamilton's game:



Finding a Hamiltonian Path in our graph

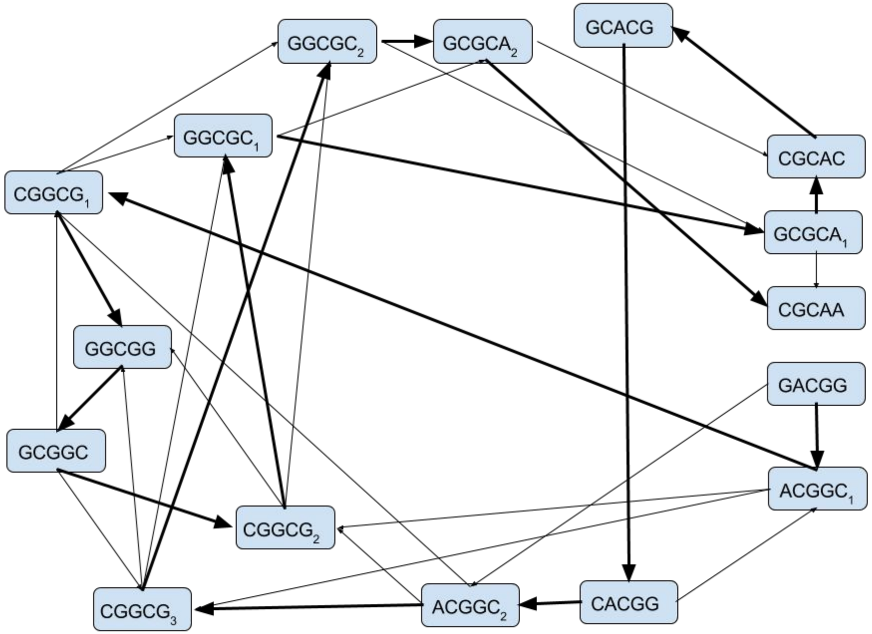
For our desired sequence:

GACGGCGGCGCACGGCGCAA

is indeed a path in this graph.

How would you write a program
To solve Hamilton's puzzles?

Is the solution unique?



De Bruijn's Problem

Nicolaas de Bruijn
(1918-2012)

A dutch mathematician noted for his many contributions in the fields of graph theory, number theory, combinatorics and logic.

Minimal Superstring Problem:

Find the shortest sequence that contains all $|\Sigma|^k$ strings of length k from the alphabet Σ as a substring.

Example: All strings of length 3 from the alphabet {'0','1'}.

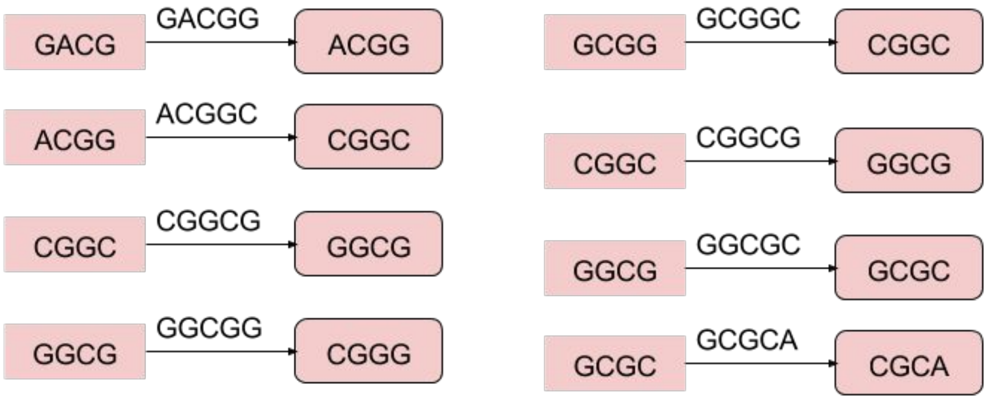
binary3 = {'000', '001', '010', '011', '100', '101', '110', '111'}

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

He solved this problem by mapping it to a graph. Note, this particular problem leads to cyclic sequence.

Another represention of k-mers in a graph

- Rather than making each k-mer a node, let's try making them an edge
- That seems odd, but it is related to the overlap idea
  - The 5-mer GACGG has a prefix GACG and a suffix ACGG
  - Think of the k-mer as the edge connecting a prefix to a suffix
  - This leads to a series of simple graphs



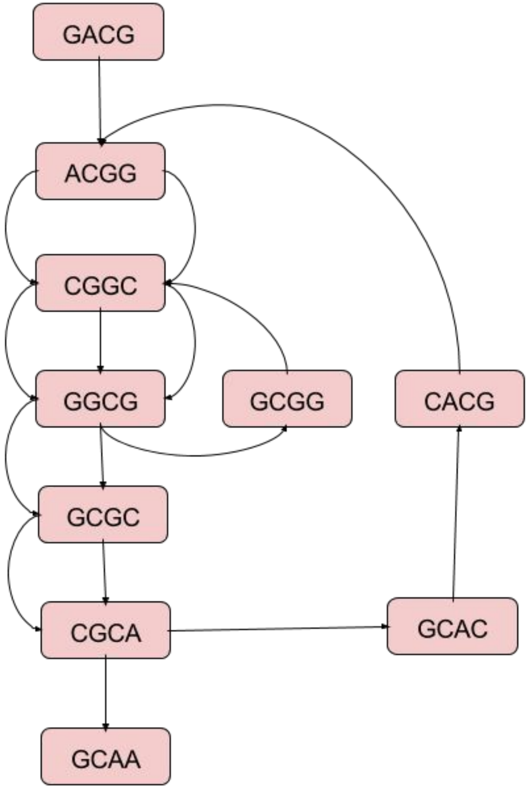
  - Then combine all nodes with the same label

A De Bruijn Graph

This graph, like the previous one has the property that edges connect nodes where a k-1 suffix matches a k-1 prefix. Graphs of this type are called "De Bruijn" graphs, after a famous mathematician.

Recall that our original 5-mers are edges in this graph, whereas they were nodes in the previous one.

Now, how might you infer the original sequence using this graph?



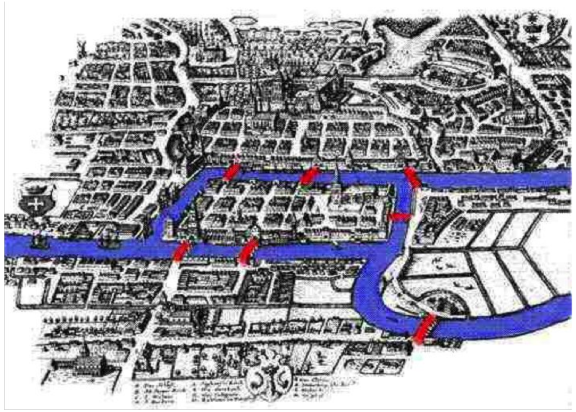
This leads to a new game

The rules of our new game

- Every *edge*, k-mer, can be used exactly once
- The object is to find a path in the graph that uses each *edge* only one time
- This game was invented in the late 1700's by a mathematician called Leonhard Euler

Leonhard Euler

A version of Euler's game:

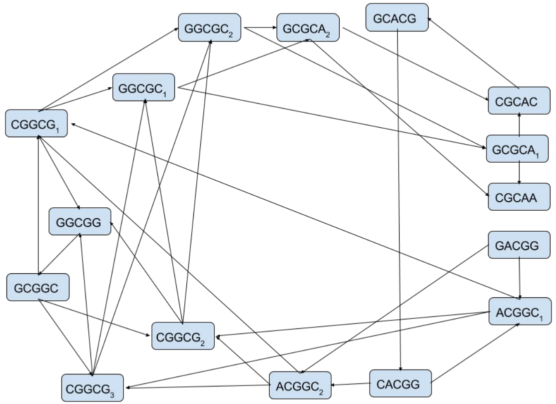


Bridges of Königsberg: Find a city tour that crosses every bridge just once

Two graphs, same problem

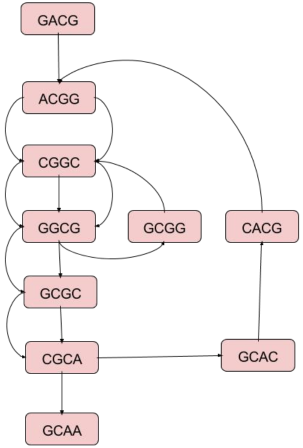
Two graphs representing 5-mers from the sequence "GACGGCGGCGCACGGCGCAA"

Hamiltonian Path:



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

Eulerian Path:

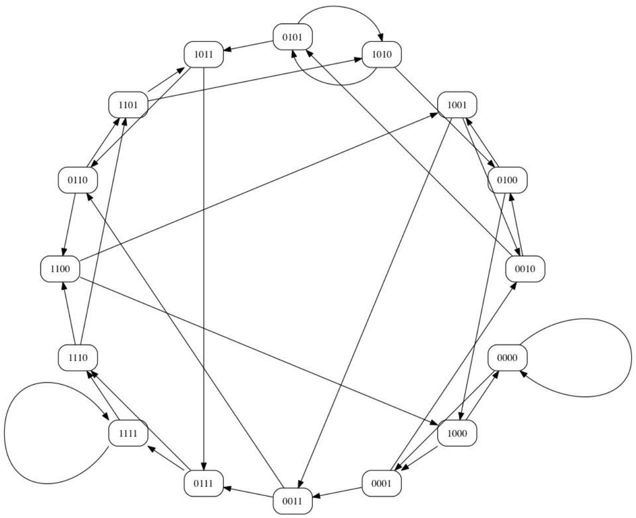


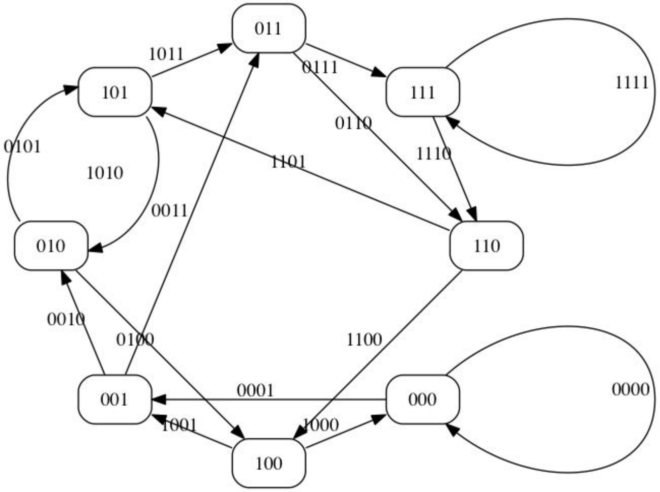
Each k-mer is an edge. Find a path that passes through every *edge* of this graph exactly once.

De Bruijn's Graphs

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.

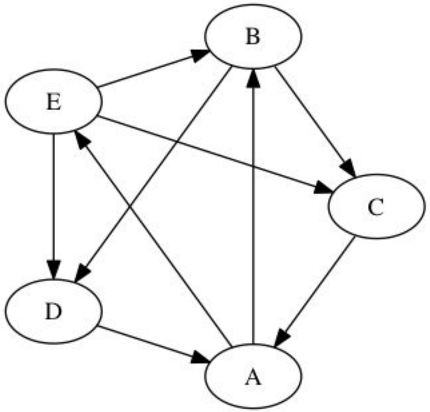




Solving Graph Problems on a Computer

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:

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 ith 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)
```

Usage example

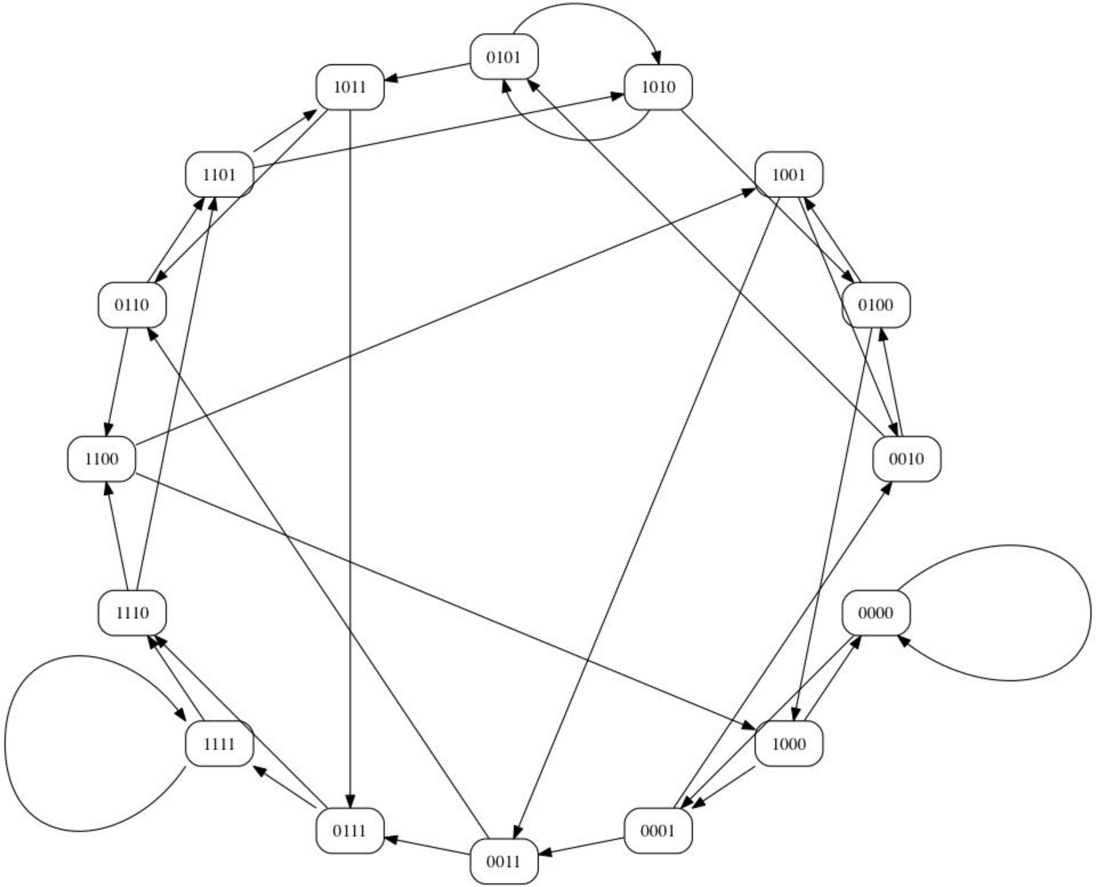
Let's generate the vertices needed to find De Bruijn's superstring of 4-bit binary strings...
and create a graph object using them.

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



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.

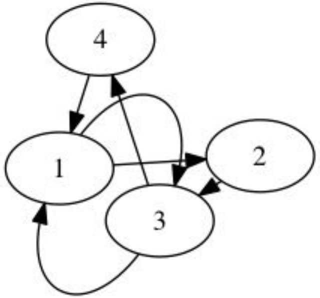
How?

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),
```

A Hamiltonian Path Algorithm

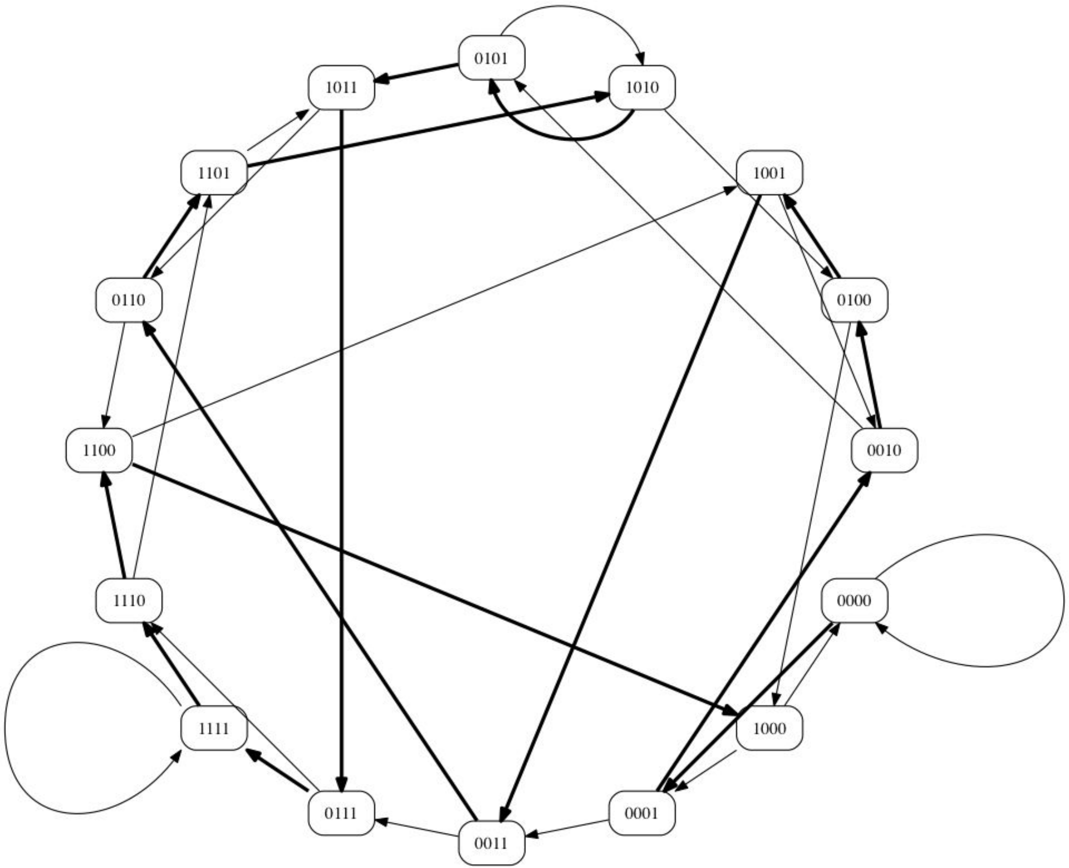
- Test each vertex permutation to see if it is a valid path
- Let's extend our BasicGraph into an EnhancedGraph class
- Create the superstring graph and find a Hamiltonian Path

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)
```

```
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
```

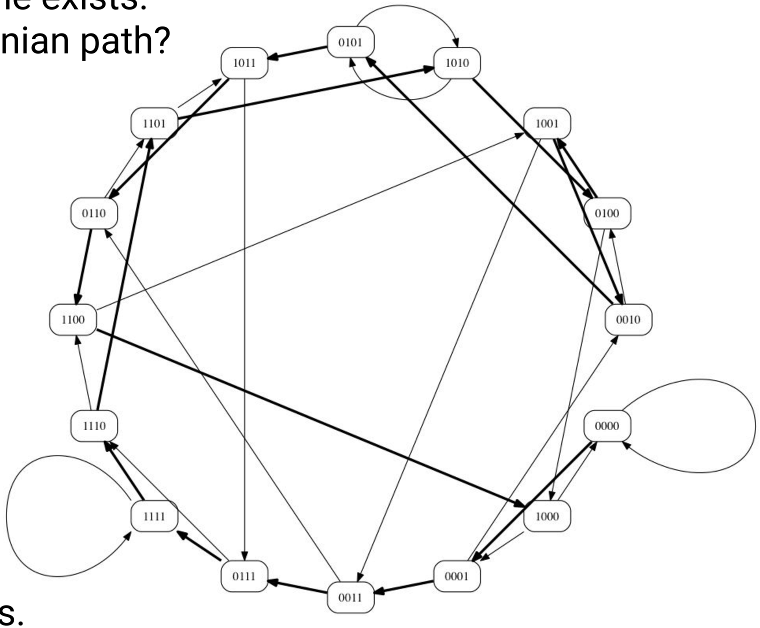
Visualizing the result



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:

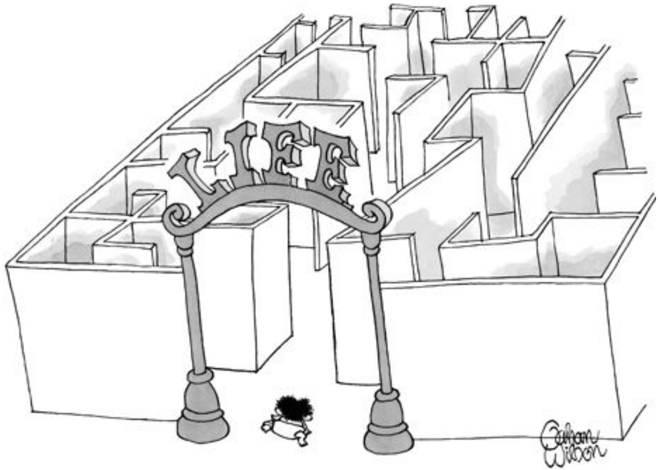
$$\frac{(\sigma!)^{\sigma^{k-1}}}{\sigma^k}$$

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

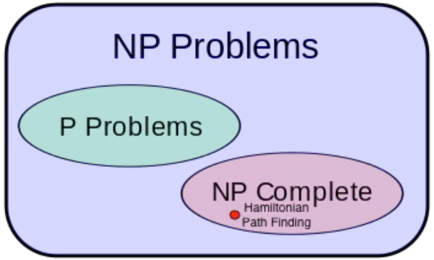
In [9]:

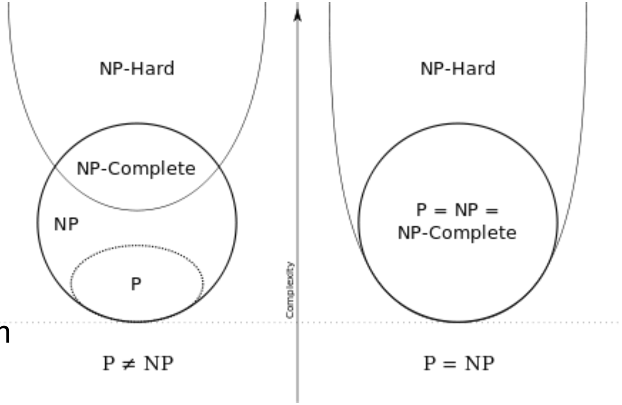
```
import itertools

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?

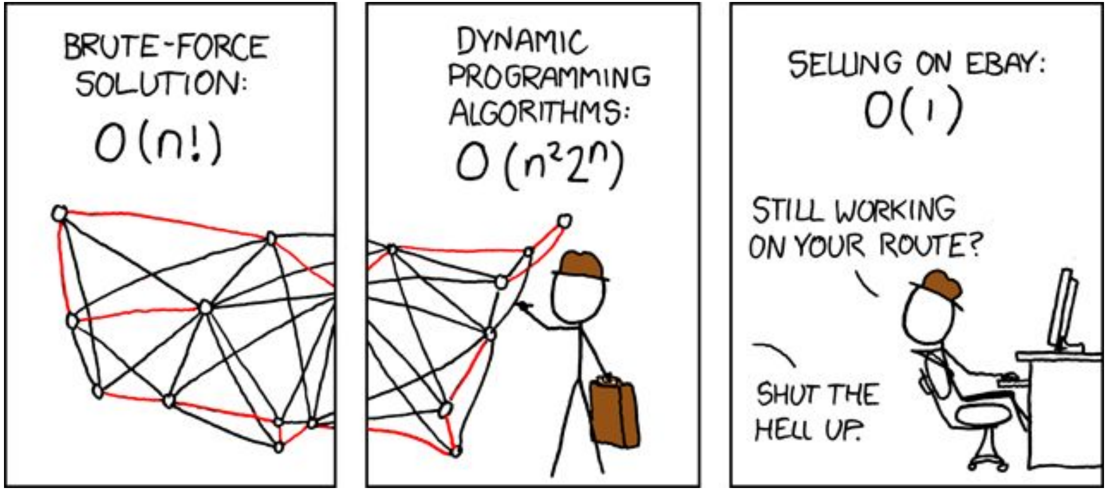
- Better in what sense?
- Better = number of steps to find a solution that is polynomial in either the number of edges or vertices
  - Polynomial: variable$^{\text{constant}}$
  - Exponential: constant$^{\text{variable}}$ or worse, variable$^{\text{variable}}$
  - For example our Brute-Force algorithm 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





What next?

Is there hope?



What if our k-mers are edges?