3. Graph-Theoretic Models

Monday, April 20, 2020 8:33 AM

Models:

Purpose: Taking an informal problem (choosing what to eat, Knapsack problem) and runing experiments and trials using programing

Graphs

A set of <u>nodes</u> (vertices) that have a property that they share

<u>Edges</u> (arcs) allows you to build graphs to see connections between nodes

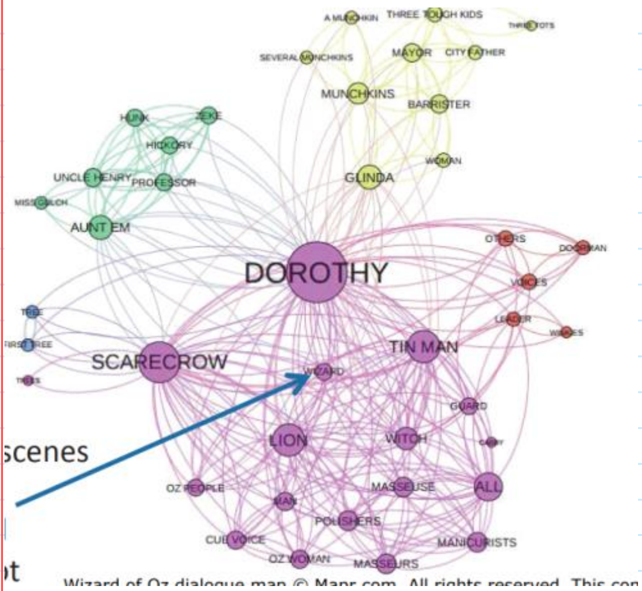
- Undirected graph: No relationship drawn from nodes
- Directed, Digraph: Information flows from source (parent) to destination (child)

Why?

The worlds are based on networks and if there are connections, then there are a use for graphs

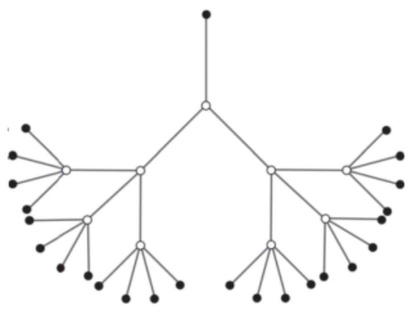
- Transportation networks: best route from one city to the next city, each destination is a node
- Political/Criminal/Social networks: how do I maximize reach?
- Family trees demonstrate directional edges and traverses in one direction which shows distinct relationship

A graph that captures and shows interactions which allows you to make inferences (i.e. Wizard of Oz shared dialogue)



Tree:

- Useful and special kind of directed graph in which any pair of nodes is connected by a single path
- Roots are on top, leaves are on bottom
-



Graph Theory

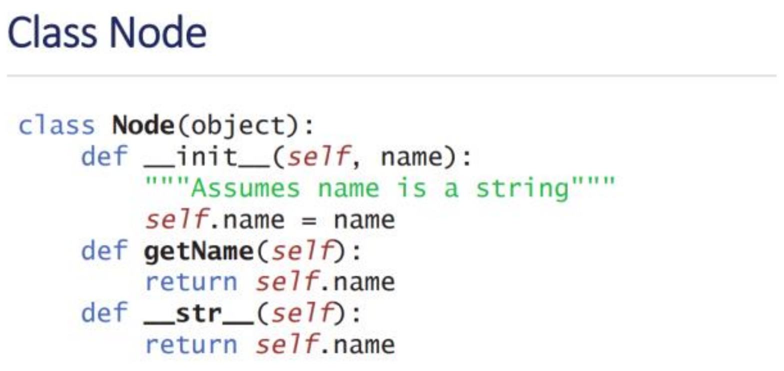
- Finding sequence of links bw elements- is ther a path from A to B?
- Finding the least expensive path bw elements *Shortest Path Problem*
- Partitioning the graph into sets of connected elements *Graph Partitioning Problem*
- Finding the most efficient way to separate sets of connected elements *The min-cut/max-flow problem*

Building a Graph

We like to abstract real problems into graphs, we also like to abstract graphs to nodes/connections, so we can be more creative than a typical latitude-longitude graph.

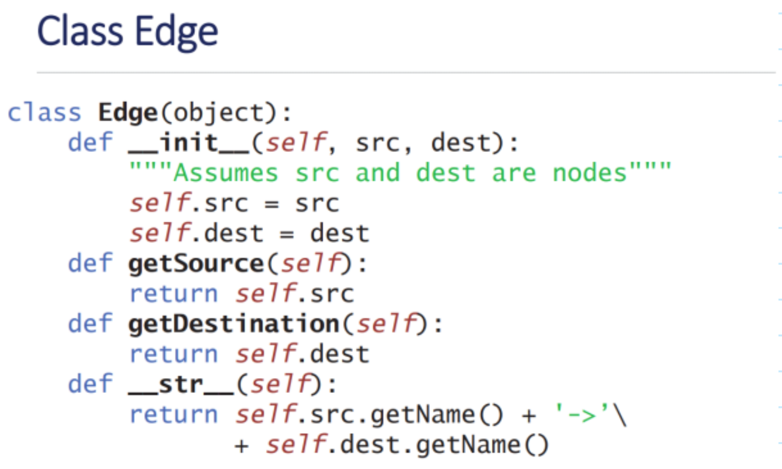
Adjacency List: Simple one-directional Abstraction (using Python and Strings)

1. Initialize a new node and edge class



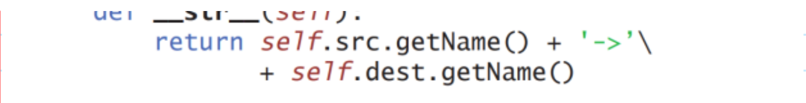
Class Node

```
class Node(object):
    def __init__(self, name):
        """Assumes name is a string"""
        self.name = name
    def getName(self):
        return self.name
    def __str__(self):
        return self.name
```



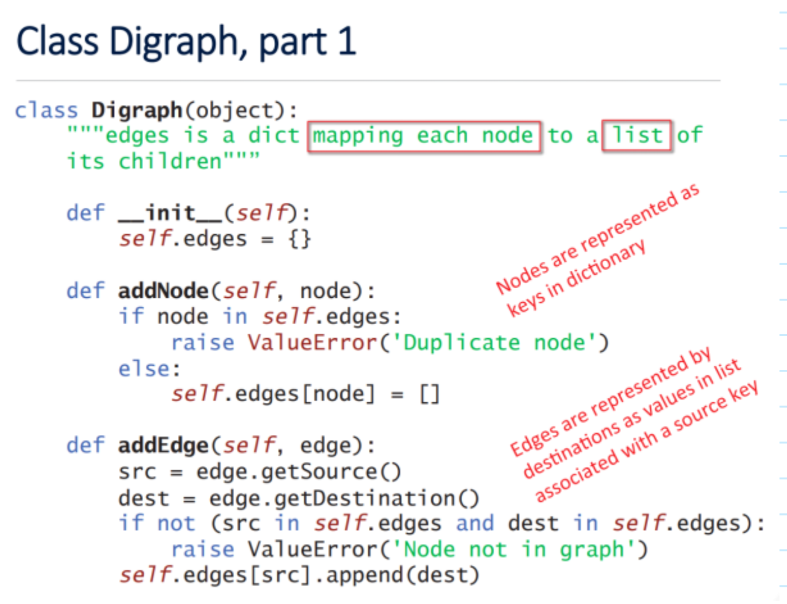
Class Edge

```
class Edge(object):
    def __init__(self, src, dest):
        """Assumes src and dest are nodes"""
        self.src = src
        self.dest = dest
    def getSource(self):
        return self.src
    def getDestination(self):
        return self.dest
    def __str__(self):
        return self.src.getName() + '->'\
               + self.dest.getName()
```



```
    def __str__(self):
        return self.src.getName() + '->'\
               + self.dest.getName()
```

2. Store the node as a key to a library, and associate it with a list of the nodes I can reach with that node

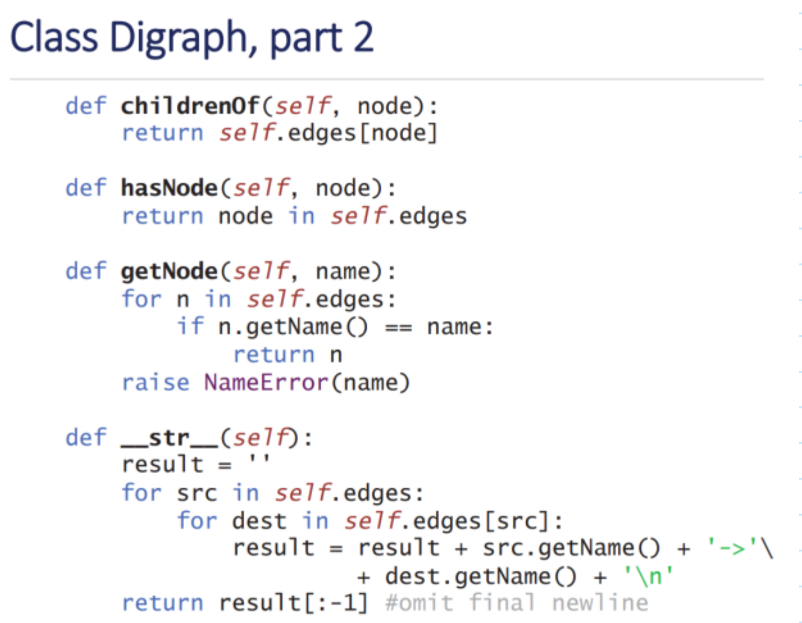


Class Digraph, part 1

```
class Digraph(object):
    """edges is a dict mapping each node to a list of
    its children"""

    def __init__(self):
        self.edges = {}

    def addNode(self, node):
        if node in self.edges:
            raise ValueError('Duplicate node')
        else:
            self.edges[node] = []

    def addEdge(self, edge):
        src = edge.getSource()
        dest = edge.getDestination()
        if not (src in self.edges and dest in self.edges):
            raise ValueError('Node not in graph')
        self.edges[src].append(dest)
```

Nodes are represented as keys in dictionary

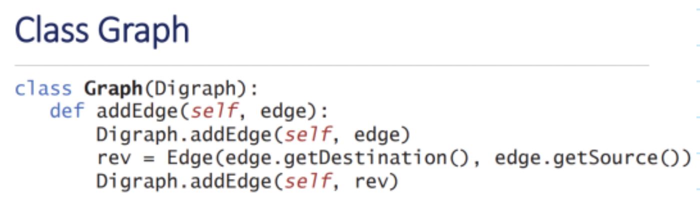
Edges are represented by destinations as values in list associated with a source key



Class Digraph, part 2

```
    def childrenOf(self, node):
        return self.edges[node]

    def hasNode(self, node):
        return node in self.edges

    def getNode(self, name):
        for n in self.edges:
            if n.getName() == name:
                return n
        raise NameError(name)

    def __str__(self):
        result = ''
        for src in self.edges:
            for dest in self.edges[src]:
                result = result + src.getName() + '->'\
                         + dest.getName() + '\n'
        return result[:-1] #omit final newline
```

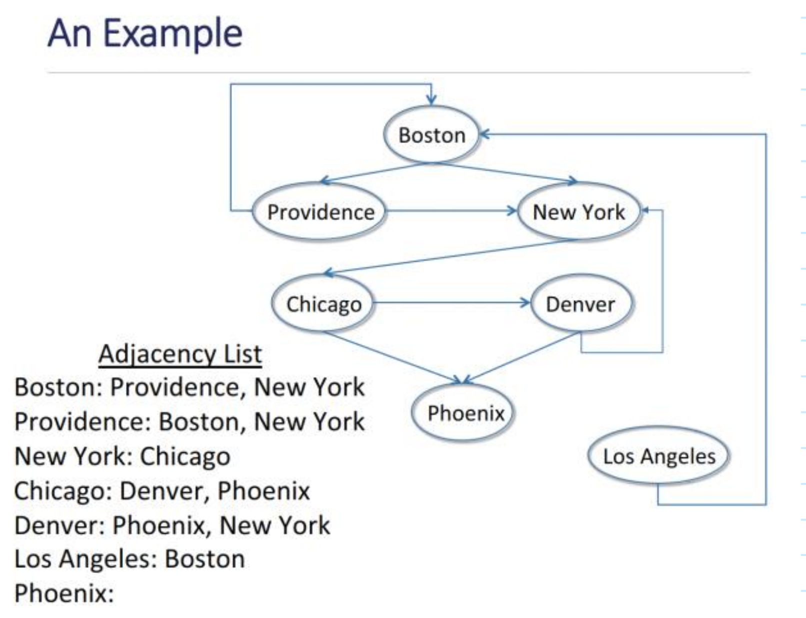
Making a Graph
Adjust the digraph, because anything that works for diagraph works for a graph and not the other way around (digraph is the bigger instance, graph is the smaller instance):



Class Graph

```
class Graph(Digraph):
    def addEdge(self, edge):
        Digraph.addEdge(self, edge)
        rev = Edge(edge.getDestination(), edge.getSource())
        Digraph.addEdge(self, rev)
```

Example Representation



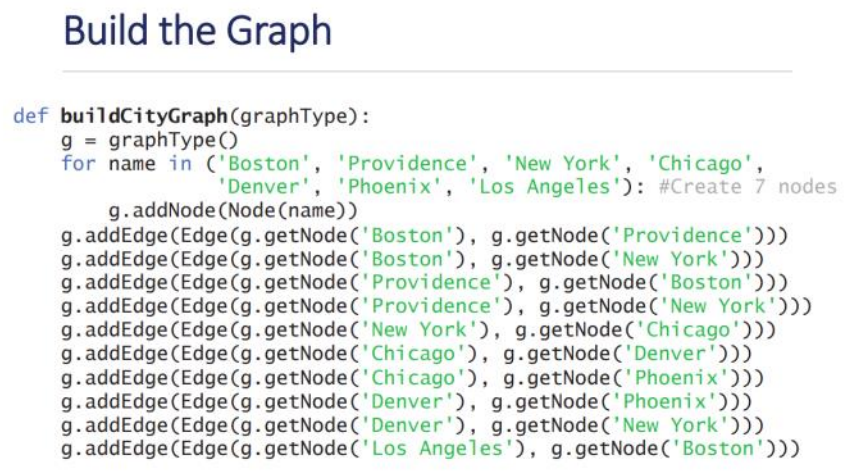
An Example

Adjacency List

Boston: Providence, New York
Providence: Boston, New York
New York: Chicago
Chicago: Denver, Phoenix
Denver: Phoenix, New York
Los Angeles: Boston
Phoenix:

Implementation

Asking the program what's the shortest number of step that will get me from source to destination



Build the Graph

```
def buildCityGraph(graphType):
    g = graphType()
    for name in ('Boston', 'Providence', 'New York', 'Chicago',
                 'Denver', 'Phoenix', 'Los Angeles'): #Create 7 nodes
        g.addNode(Node(name))
    g.addEdge(Edge(g.getNode('Boston'), g.getNode('Providence')))
    g.addEdge(Edge(g.getNode('Boston'), g.getNode('New York')))
    g.addEdge(Edge(g.getNode('Providence'), g.getNode('Boston')))
    g.addEdge(Edge(g.getNode('Providence'), g.getNode('New York')))
    g.addEdge(Edge(g.getNode('New York'), g.getNode('Chicago')))
    g.addEdge(Edge(g.getNode('Chicago'), g.getNode('Denver')))
    g.addEdge(Edge(g.getNode('Chicago'), g.getNode('Phoenix')))
    g.addEdge(Edge(g.getNode('Denver'), g.getNode('Phoenix')))
    g.addEdge(Edge(g.getNode('Denver'), g.getNode('New York')))
    g.addEdge(Edge(g.getNode('Los Angeles'), g.getNode('Boston')))
```

Finding the Shortest Path

Depth First Search (DFS): since it's a graph, not a tree, there's a chance of loops. You take the first edge out of a node and go as "deep as you can" checking each step of the way to see if you've made it to your destination or run out of options

Depth First Search (DFS)

```python
def DFS(graph, start, end, path, shortest, toPrint = False):
    path = path + [start]
    if toPrint:
        print('Current DFS path:', printPath(path))
    if start == end:
        return path
    for node in graph.childrenOf(start):
        if node not in path: #avoid cycles
            if shortest == None or len(path) < len(shortest):
                newPath = DFS(graph, node, end, path, shortest, toPrint)
                if newPath != None:
                    shortest = newPath
        elif toPrint:
            print('Already visited', node)
    return shortest

def shortestPath(graph, start, end, toPrint = False):
    return DFS(graph, start, end, [], None, toPrint)
```

Note how will explore all paths through first node, before ...

... returning to this point in the recursion to try next node

DFS called from a wrapper function: `shortestPath`

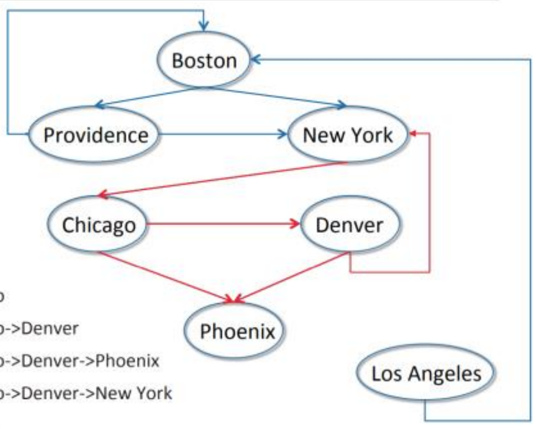
Gets recursion started properly

Provides appropriate abstraction

Test DFS

```python
def testSP(source, destination):
    g = buildCityGraph(DiGraph)
    sp = shortestPath(g, g.getNode(source), g.getNode(destination)
                      toPrint = True)
    if sp != None:
        print('Shortest path from', source, 'to',
              destination, 'is', printPath(sp))
    else:
        print('There is no path from', source, 'to', destination)

testSP('Boston', 'Chicago')
```

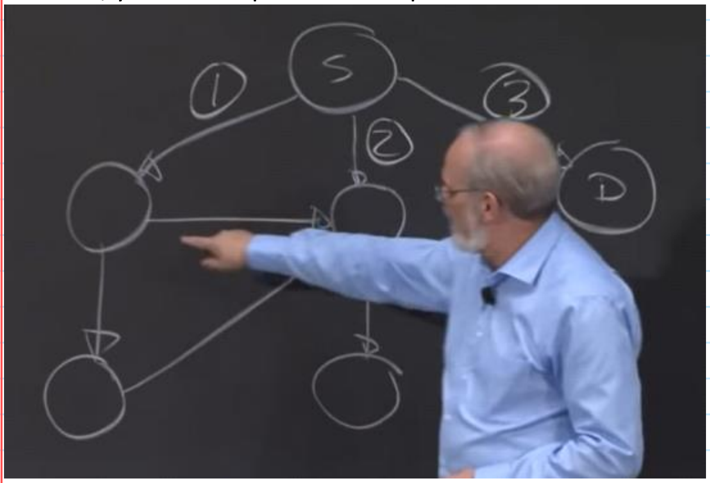
Output (Chicago to Boston)

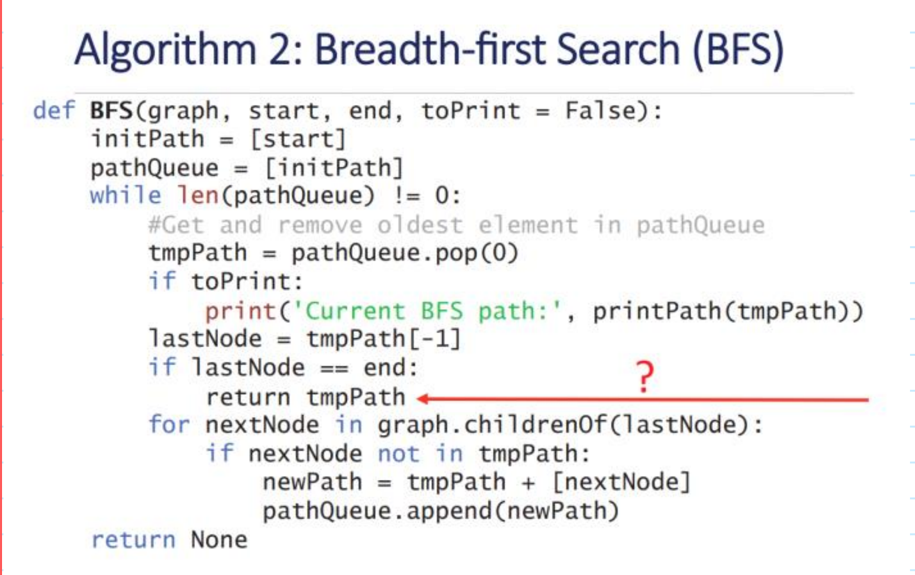


```
Current DFS path: Chicago
Current DFS path: Chicago->Denver
Current DFS path: Chicago->Denver->Phoenix
Current DFS path: Chicago->Denver->New York
Already visited Chicago
Current DFS path: Chicago->Phoenix
There is no path from Chicago to Boston
```

Breadth First Search

Finding the shortest solution by exploring all the paths of length one before you get to the path of length two. Consider all the edges that leave the node in some order and follow the first edges leaving from the starting nodes. It will keep track of all open nodes worth exploring, and once you find a solution, you can stop. Once it stops it knows that it's reached the shortest path.

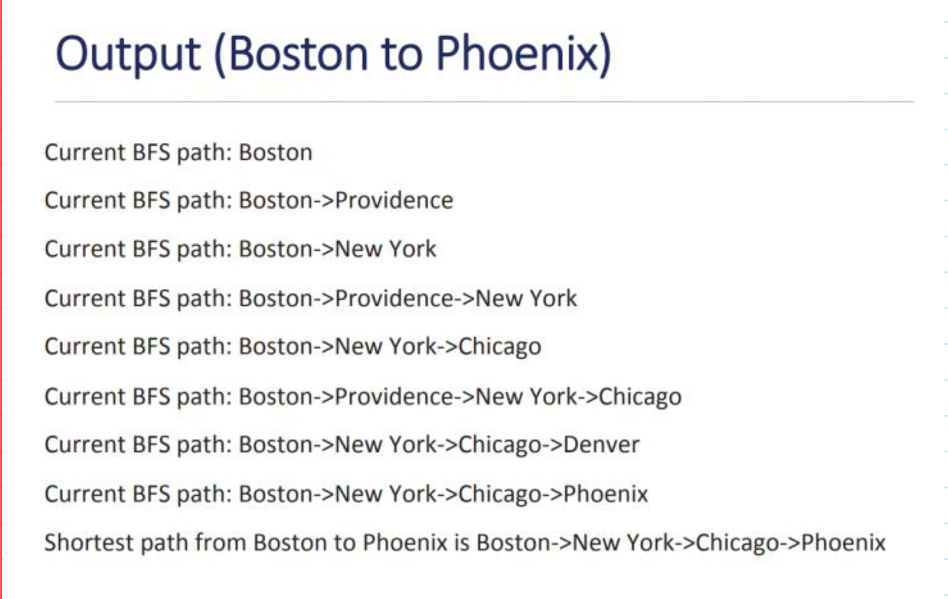




Algorithm 2: Breadth-first Search (BFS)

```
def BFS(graph, start, end, toPrint = False):
    initPath = [start]
    pathQueue = [initPath]
    while len(pathQueue) != 0:
        #Get and remove oldest element in pathQueue
        tmpPath = pathQueue.pop(0)
        if toPrint:
            print('Current BFS path:', printPath(tmpPath))
        lastNode = tmpPath[-1]
        if lastNode == end:
            return tmpPath
        for nextNode in graph.childrenOf(lastNode):
            if nextNode not in tmpPath:
                newPath = tmpPath + [nextNode]
                pathQueue.append(newPath)
    return None
```

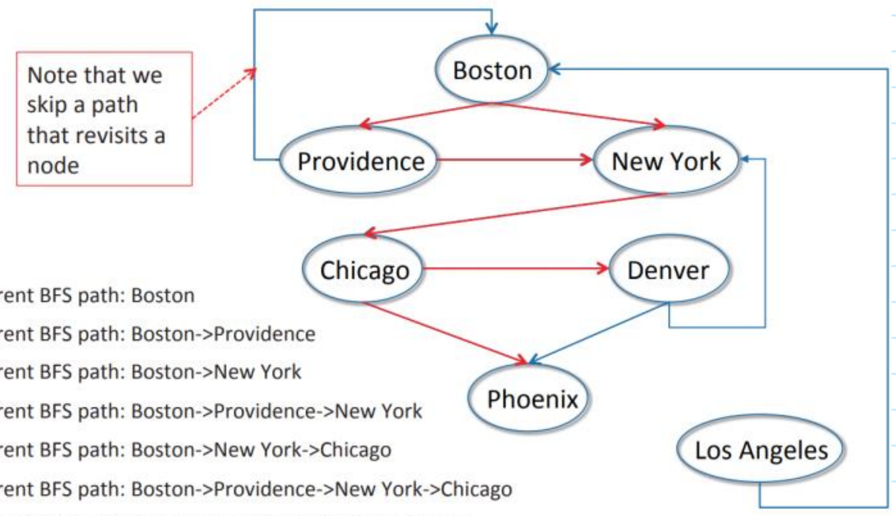
?



Output (Boston to Phoenix)

```
Current BFS path: Boston
Current BFS path: Boston->Providence
Current BFS path: Boston->New York
Current BFS path: Boston->Providence->New York
Current BFS path: Boston->New York->Chicago
Current BFS path: Boston->Providence->New York->Chicago
Current BFS path: Boston->New York->Chicago->Denver
Current BFS path: Boston->New York->Chicago->Phoenix
Shortest path from Boston to Phoenix is Boston->New York->Chicago->Phoenix
```

Output (Boston to Pheonix)



```
Current BFS path: Boston
Current BFS path: Boston->Providence
Current BFS path: Boston->New York
Current BFS path: Boston->Providence->New York
Current BFS path: Boston->New York->Chicago
Current BFS path: Boston->Providence->New York->Chicago
Current BFS path: Boston->New York->Chicago->Denver
Current BFS path: Boston->New York->Chicago->Phoenix
Shortest path from Boston to Phoenix is Boston->New York->Chicago->Phoenix
```

Extension

To incorporate the flight time/distance to ask a dif question of the shortest distance, you can add a weight to the edges so longer paths (Boston-LA) has a greater weight than shorter paths (Boston-NY). You'd modify the same procedure except now, the cost that you want to minimize is the sum of the weights with each paths

- To implement this, you can only use a depth first search because it allows you to collect all the paths that will take you from a source to the destination and then allow you to compare the weights of the nodes whereas the breadth first path doesn't accumulate solutions, because it stops at the minimum number of *loops* regardless of the weights