Hands-on Activity 1.3 | Transportation using Graphs

Objective(s):

This activity aims to demonstrate how to solve transportation related problem using Graphs

Intended Learning Outcomes (ILOs):

- Demonstrate how to compute the shortest path from source to destination using graphs
- Apply DFS and BFS to compute the shortest path

Resources:

- Jupyter Notebook

Procedures:

1. Create a Node class

```
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
```

2. Create an Edge class

```
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()
```

3. Create Digraph class that add nodes and edges

```
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)
    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
```

4. Create a Graph class from Digraph class that deifnes the destination and Source

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

5. Create a buildCityGraph method to add nodes (City) and edges (source to destination)

```
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('Denver'), g.getNode('Phoenix')))
    g.addEdge(Edge(g.getNode('Denver'), g.getNode('New York')))
    g.addEdge(Edge(g.getNode('Los Angeles'), g.getNode('Boston')))
    return g
```

```
def printPath(path):
    """Assumes path is a list of nodes"""
    result = ''
    for i in range(len(path)):
        result = result + str(path[i])
        if i != len(path) - 1:
            result = result + '->'
    return result
```

6. Create a method to define DFS technique

```
def DFS(graph, start, end, path, shortest, toPrint = False):
    """Assumes graph is a Digraph; start and end are nodes;
          path and shortest are lists of nodes
       Returns a shortest path from start to end in graph"""
    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
```

7. Define a shortestPath method to return the shortest path from source to destination using DFS

```
def shortestPath(graph, start, end, toPrint = False):
    """Assumes graph is a Digraph; start and end are nodes
       Returns a shortest path from start to end in graph"""
    return DFS(graph, start, end, [], None, toPrint)
```

8. Create a method to test the shortest path method

```
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)
```

9. Execute the testSP method

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

Question:

Describe the DFS method to compute for the shortest path using the given sample codes

Based on what I can see from the output, it seems like it updates based on which node is currently visited. However, there is a slight error to how the code is implemented? I think the function called here is BFS and not DFS.

10. Create a method to define BFS technique

```
def BFS(graph, start, end, toPrint = False):
    """Assumes graph is a Digraph; start and end are nodes
       Returns a shortest path from start to end in graph"""
    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
```

11. Define a shortestPath method to return the shortest path from source to destination using DFS

```
def shortestPath(graph, start, end, toPrint = False):
    """Assumes graph is a Digraph; start and end are nodes
       Returns a shortest path from start to end in graph"""
    return BFS(graph, start, end, toPrint)
```

12. Execute the testSP method

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

Question:

Describe the BFS method to compute for the shortest path using the given sample codestion:

It visits its neighboring nodes first before traversing again

Supplementary Activitiy

- Use a specific location or city to solve transportation using graph
- Use DFS and BFS methods to compute the shortest path
- Display the shortest path from source to destination using DFS and BFS
- Differentiate the performance of DFS from BFS

```
'''
Supplementary for demonstrating BFS and DFS as methods
'''
#Supplementary

class Node(object):
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name
```

```
class Graph(object):
    def __init__(self):
        self.adj = {}

    def addNode(self, node):
        if node in self.adj:
            raise ValueError("Duplicate Node!")
        self.adj[node] = []

    def findNode(self, node):
        for exsNode in self.adj.keys():
            if exsNode == node:
                return node.getName()
        raise ValueError("Node Does not Exist!")

    def displayNodes(self):
        for node, neighbors in self.adj.items():
            neighbor_names = [neighbor.getName() for neighbor in neighbors]
            print(node.getName(), '--->', neighbor_names)


    def addEdge(self, a, b):
      if a not in self.adj or b not in self.adj:
          raise ValueError("Nodes do not exist!")
      if b not in self.adj[a]:
          self.adj[a].append(b)
      if a not in self.adj[b]:
          self.adj[b].append(a)


    def DFS(self, src, dest):
        stack = [(src, [src])]
        visited = set()
        while stack:
            (node, path) = stack.pop()
            if node.getName() == dest.getName():
                shortPathNames = [node.getName() for node in path]
                return shortPathNames
            visited.add(node)
            for i in self.adj[node]:
                stack.append((i, path + [i]))
        return None

    def BFS(self, src, dest):
        visited = set()
        queue = [(src, [src])]
        while queue:
            (node, path) = queue.pop(0)
            if node.getName() == dest.getName():
                shortest_path_names = [node.getName() for node in path]
                return shortest_path_names
            visited.add(node)
            for neighbor in self.adj[node]:
                if neighbor not in visited:
                    queue.append((neighbor, path + [neighbor]))
        return None


# Create Graph
graph = Graph()

# Create Nodes
node1 = Node("Lobby")
node2 = Node("Seminar Room 1")
node3 = Node("Principal Office")
node4 = Node("Canteen")
node5 = Node("Comfort Room")
node6 = Node("Dean Office")
node7 = Node("Seminar Room 2")

# Add the Nodes to the Graph
graph.addNode(node1)
graph.addNode(node2)
graph.addNode(node3)
graph.addNode(node4)
graph.addNode(node5)
graph.addNode(node6)
```

```
graph.addNode(node7)

# Add Edges
graph.addEdge(node1, node2)
graph.addEdge(node2, node3)
graph.addEdge(node3, node4)
graph.addEdge(node3, node5)
graph.addEdge(node3, node6)
graph.addEdge(node6, node7)

# Display possible Paths
graph.displayNodes()
print("-----------------------------------------------------------------------------------------------")
#Depth First Search
DFS = graph.DFS(node1, node7)
print("Shortest Path Using DFS: ", DFS)

#Breadth First Search
print("-----------------------------------------------------------------------------------------------")
BFS = graph.BFS(node1, node7)
print("Shortest Path Using BFS: ", BFS)
```

```
Lobby ---> ['Seminar Room 1']
Seminar Room 1 ---> ['Lobby', 'Principal Office']
Principal Office ---> ['Seminar Room 1', 'Canteen', 'Comfort Room', 'Dean Office']
Canteen ---> ['Principal Office']
Comfort Room ---> ['Principal Office']
Dean Office ---> ['Principal Office', 'Seminar Room 2']
Seminar Room 2 ---> ['Dean Office']
-----------------------------------------------------------------------------------------------
Shortest Path Using DFS:  ['Lobby', 'Seminar Room 1', 'Principal Office', 'Dean Office', 'Seminar Room 2']
-----------------------------------------------------------------------------------------------
Shortest Path Using BFS:  ['Lobby', 'Seminar Room 1', 'Principal Office', 'Dean Office', 'Seminar Room 2']
```

Type your evaluation about the performance of DFS and BFS

**In terms of which search algorithms is better with regards to its performance, It really depends on how large the graph is. For DFS, its not gauranteed that you find the shortest path. However, it does need minimal space since it only stores start to current node. It is also easy to implement unlike BFS. Moreover, BFS gaurantees that you will find the shortest path in the graph. In my example, both DFS and BFS provided the same output since the graph only consisted of seven nodes and there were no loops within the graph **

Conclusion

type your conclusion here

In conlusion, the activity have shown proper implementation of graphs and how to traverse the graph. Moreover, I was able to implement it by solving a unique problem using the search algorithms.