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 DFS path: Boston
Current DFS path: Boston->Providence
Already visited Boston
Current DFS path: Boston->Providence->New York
Current DFS path: Boston->Providence->New York->Chicago
Current DFS path: Boston->Providence->New York->Chicago->Denver
Current DFS path: Boston->Providence->New York->Chicago->Denver->Phoenix
Already visited New York
Current DFS path: Boston->New York
Current DFS path: Boston->New York->Chicago
Current DFS path: Boston->New York->Chicago->Denver
Current DFS path: Boston->New York->Chicago->Denver->Phoenix
Already visited New York
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

The Dfs method prints the current path one by one depending which node the path is going through, it also shows the shortest path by recursively checking if the current

path is shorter than the current shortest path or if no shortest path has been discovered yet, it will recursively call itself for each child node of the current node.

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
```

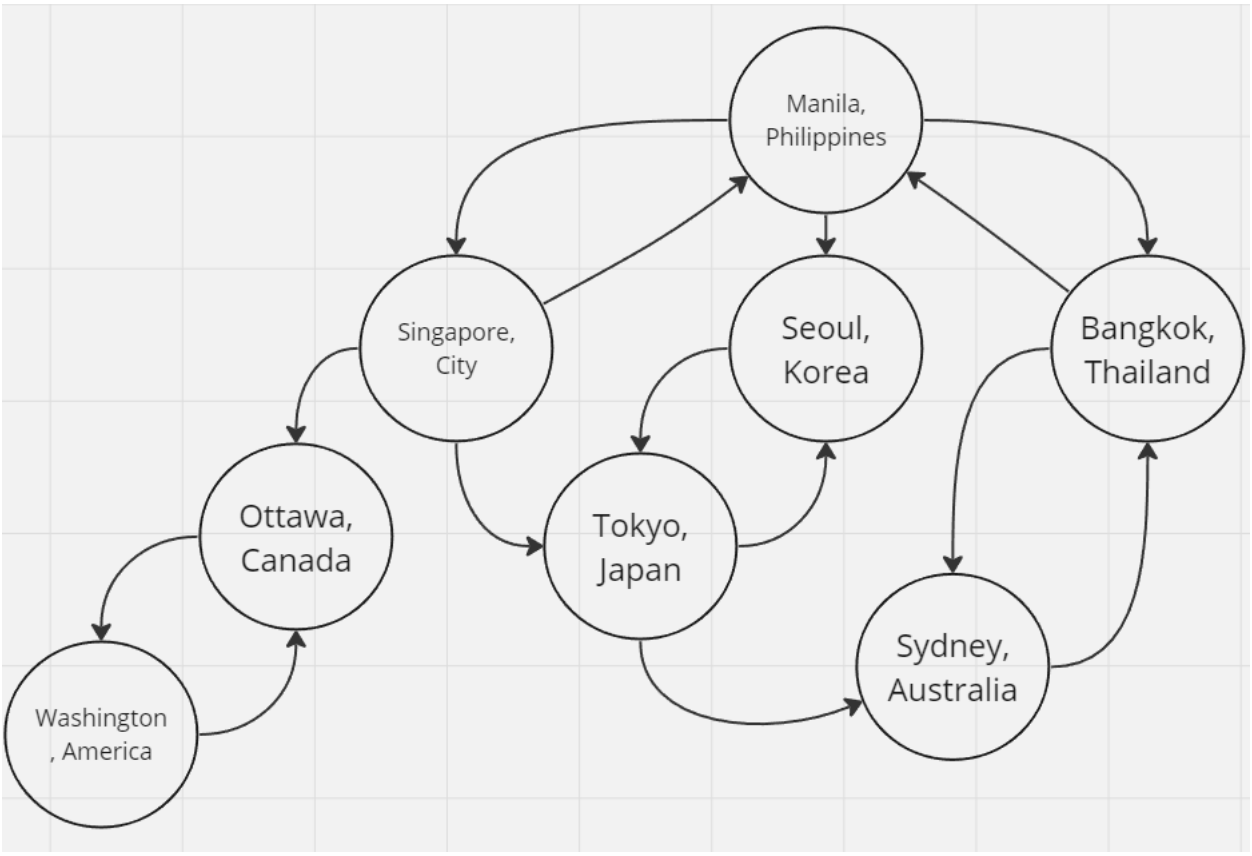
Question:

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

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

I created a graph for countries with the routes that they are connected to, here it is Below:




```
# type your code here using DFS
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(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()

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
```

```
            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]

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

    def buildCityGraph(graphType):
        g = graphType()
        for name in ('Philippines', 'Singapore', 'Korea', 'Thailand', 'Canada', 'America', 'Japan', 'Australia'):
            g.addNode(Node(name))

        #Nodes that Philippines are connected to
        g.addEdge(Edge(g.getNode('Philippines'), g.getNode('Singapore')))
        g.addEdge(Edge(g.getNode('Philippines'), g.getNode('Korea')))
        g.addEdge(Edge(g.getNode('Philippines'), g.getNode('Thailand')))\
        #Nodes that Singapore are connected to
        g.addEdge(Edge(g.getNode('Singapore'), g.getNode('Philippines')))
        g.addEdge(Edge(g.getNode('Singapore'), g.getNode('Canada')))
        g.addEdge(Edge(g.getNode('Singapore'), g.getNode('Japan')))
        #Nodes that Korea are connected to
        g.addEdge(Edge(g.getNode('Korea'), g.getNode('Japan')))
        #Nodes that Thailand are connected to
        g.addEdge(Edge(g.getNode('Thailand'), g.getNode('Australia')))
        g.addEdge(Edge(g.getNode('Thailand'), g.getNode('Philippines')))
        #Nodes that Canada are connected to
        g.addEdge(Edge(g.getNode('Canada'), g.getNode('America')))
        #Nodes that Japan are connected to
        g.addEdge(Edge(g.getNode('Japan'), g.getNode('Korea')))
        g.addEdge(Edge(g.getNode('Japan'), g.getNode('Australia')))
        #Nodes that Australia are connected to
        g.addEdge(Edge(g.getNode('Australia'), g.getNode('Thailand')))
        #Nodes that America are connected to
        g.addEdge(Edge(g.getNode('America'), g.getNode('Canada')))
        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

    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:
                if shortest == None or len(path) < len(shortest):
                    newPath = DFS(graph, node, end, path, shortest,
                                  toPrint)
                    if newPath != None:
                        shortest = newPath
            elif toPrint:
                print('\nAlready visited', node,'\n')
        return shortest
```

```
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)

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)

print('Output: ')
testSP('Philippines', 'Australia')
```

```
Output:
Current DFS path: Philippines
Current DFS path: Philippines->Singapore

Already visited Philippines

Current DFS path: Philippines->Singapore->Canada
Current DFS path: Philippines->Singapore->Canada->America

Already visited Canada

Current DFS path: Philippines->Singapore->Japan
Current DFS path: Philippines->Singapore->Japan->Korea

Already visited Japan

Current DFS path: Philippines->Singapore->Japan->Australia
Current DFS path: Philippines->Korea
Current DFS path: Philippines->Korea->Japan

Already visited Korea

Current DFS path: Philippines->Korea->Japan->Australia
Current DFS path: Philippines->Thailand
Current DFS path: Philippines->Thailand->Australia

Already visited Philippines

Shortest path from Philippines to Australia is Philippines->Thailand->Australia
```

Since I wanted to take the short way from Philippines to Australia, the code shows that the shortest way of it is by going from Philippines, Thailand, and Australia.

```
# type your code here using BFS
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:
        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

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)
```

```
print('Output: \n')
testSP('Philippines', 'Australia')
```

```
Output:

    Current BFS path: Philippines
    Current BFS path: Philippines->Singapore
    Current BFS path: Philippines->Korea
    Current BFS path: Philippines->Thailand
    Current BFS path: Philippines->Singapore->Canada
    Current BFS path: Philippines->Singapore->Japan
    Current BFS path: Philippines->Korea->Japan
    Current BFS path: Philippines->Thailand->Australia
    Shortest path from Philippines to Australia is Philippines->Thailand->Australia
```

Type your evaluation about the performance of DFS and BFS

The DFS and BFS is a very good way for determining the shortest way for graphs.

Both DFS and BFS provide a clear and straightforward way to explore the structure of a graph. DFS and BFS are also valuable for analyzing the connectivity of a graph, identifying connected components, and determining if a graph is connected or not.

Conclusion

As I noticed for the difference of DFS and BFS, they are both a straighforward way to explore the structure of graph but the DFS always decides to check the connectivity of a node one by one starting from a specific node. While on the other hand, the BFS always