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

type your answer here:

The DFS method explores paths by traversing as far as possible along each branch before backtracking. Upon reaching the destination node, it backtracks to explore other unvisited branches.

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:

BFS examines all nodes at the current level before moving on to nodes at deeper levels. If the destination node is encountered during this traversal, the algorithm terminates, returning the shortest path found up to that point.

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

```
class Node(object):
    def __init__(self, name): # initialize a node with a name
        self.name = name
    def getName(self): # return name of node
        return self.name
    def __str__(self): # return the string of node (routes)
        return self.name

class Edge(object): # initialize an edge with source and destination node
    def __init__(self, src, dest):
        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):
    def __init__(self):
        self.edges = {}
    def addNode(self, node): # add a node to the graph
        if node in self.edges:
            raise ValueError('Duplicate node')
        else:
            self.edges[node] = []
    def addEdge(self, edge): # add an edge to the graph
        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): # validate if node is in graph
        return node in self.edges
    def getNode(self, name): # return the node with the given route
        for n in self.edges:
            if n.getName() == name:
                return n
        raise NameError(name)
    def __str__(self): # return the strings of the graph
        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

class Graph(Digraph): # add an edge to the graph and its reverse
    def addEdge(self, edge):
        Digraph.addEdge(self, edge)
        rev = Edge(edge.getDestination(), edge.getSource())
        Digraph.addEdge(self, rev)

def buildJeepneyRouteGraph(graphType): # create a graph representing a commuting route of a jeepney from fortune to cuba
    g = graphType()
    for name in ('Fortune', 'Parang', 'Tumana', 'Lamuan', 'Concepcion', 'Katipunan', 'Aurora', 'Cubao'):
        g.addNode(Node(name))

    g.addEdge(Edge(g.getNode('Fortune'), g.getNode('Parang')))
    g.addEdge(Edge(g.getNode('Parang'), g.getNode('Tumana')))
    g.addEdge(Edge(g.getNode('Tumana'), g.getNode('Lamuan')))
    g.addEdge(Edge(g.getNode('Lamuan'), g.getNode('Concepcion')))
    g.addEdge(Edge(g.getNode('Concepcion'), g.getNode('Katipunan')))
    g.addEdge(Edge(g.getNode('Katipunan'), g.getNode('Aurora')))
    g.addEdge(Edge(g.getNode('Aurora'), g.getNode('Cubao')))

    return g

def printRoute(route): # print the route from start to end
    result = ''
```

```
    for i in range(len(route)):
        result = result + str(route[i])
        if i != len(route) - 1:
            result = result + ' to '
    return result

def DFS(graph, start, end, route, shortest, toPrint = False): # find the shortest route method function
    route = route + [start]
    if toPrint:
        print('Current DFS route:', printRoute(route))
    if start == end:
        return route
    for node in graph.childrenOf(start):
        if node not in route: # avoid cycles
            if shortest == None or len(route) < len(shortest):
                newRoute = DFS(graph, node, end, route, shortest, toPrint)
                if newRoute != None:
                    shortest = newRoute
        elif toPrint:
            print('Already visited', node)
    return shortest

def shortestRoute(graph, start, end, toPrint = False): # find the shortest route from start to end
    return DFS(graph, start, end, [], None, toPrint)

def testSR(source, destination): # test the shortest route function
    g = buildJeepneyRouteGraph(Digraph)
    sp = shortestRoute(g, g.getNode(source), g.getNode(destination), toPrint = True)
    if sp != None:
        print('Shortest route from', source, 'to', destination, 'is:', printRoute(sp))
    else:
        print('There is no route from', source, 'to', destination)

testSR('Parang', 'Aurora')
```

```
 Current DFS route: Parang
 Current DFS route: Parang to Tumana
 Current DFS route: Parang to Tumana to Lamuan
 Current DFS route: Parang to Tumana to Lamuan to Concepcion
 Current DFS route: Parang to Tumana to Lamuan to Concepcion to Katipunan
 Current DFS route: Parang to Tumana to Lamuan to Concepcion to Katipunan to Aurora
 Shortest route from Parang to Aurora is: Parang to Tumana to Lamuan to Concepcion to Katipunan to Aurora
```

```
def BFS(graph, start, end, toPrint = False):

    initRoute = [start] # initialize the initial route with the starting node
    routeQueue = [initRoute] # initialize a queue to store routes

    while len(routeQueue) != 0: # iterate until the routeQueue is empty
        tmpRoute = routeQueue.pop(0) # get and remove the oldest element in routeQueue

        if toPrint: # print current route if toPrint is True
            print('Current BFS route:', printRoute(tmpRoute))

        lastNode = tmpRoute[-1] # get the last node in the current route

        if lastNode == end: # check if the last node is the destination
            return tmpRoute

        for nextNode in graph.childrenOf(lastNode): # explore the children of the last node
            if nextNode not in tmpRoute: # validate if the child node is not in the current route
                newRoute = tmpRoute + [nextNode] # create a new route by extending the current route with the child node
                routeQueue.append(newRoute) # add the new route to the route queue
    return None

def shortestRoute(graph, start, end, toPrint = False):
    return BFS(graph, start, end, toPrint)

testSR('Parang', 'Aurora') # call the testSR function to find and print the shortest route from parang to aurora
```

```
Current BFS route: Parang
Current BFS route: Parang to Tumana
Current BFS route: Parang to Tumana to Lamuan
Current BFS route: Parang to Tumana to Lamuan to Concepcion
Current BFS route: Parang to Tumana to Lamuan to Concepcion to Katipunan
Current BFS route: Parang to Tumana to Lamuan to Concepcion to Katipunan to Aurora
Shortest route from Parang to Aurora is: Parang to Tumana to Lamuan to Concepcion to Katipunan to Aurora
```

Type your evaluation about the performance of DFS and BFS:

BFS explores paths level by level, systematically visiting all nodes at the current depth before moving to nodes at the next depth. This approach ensures that the shortest path is found first because BFS explores all possible paths of length 1 before exploring paths of length 2, and so on. Therefore, BFS guarantees finding the shortest path between two nodes in a graph, making it more suitable for computing shortest paths compared to DFS.

Conclusion

In conclusion, BFS systematically explores all paths level by level, visiting all neighbors of a node before moving on to the next level. The DFS method uses a recursive approach, while BFS employs a queue to maintain and explore paths layer by layer.

BFS guarantees finding the shortest path between two nodes in a graph, making it more suitable for computing shortest paths compared to DFS