Name: Quibral, Juliann Vincent B.

Section: BSCPE22S3

```
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
    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


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 ('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('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


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


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)


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
```

Based on the ouput given by the DFS method:

The node "La Vista" is where the DFS algorithm begins. It adds each node to the current route by recursively exploring each surrounding node.

This demonstrates how the Depth-First Search algorithm explores paths in a graph by traversing as far as possible along each branch before backtracking.

```
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


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)


def testSP(source, destination):
    #Tests the shortestPath function by finding and printing the shortest path between two cities.

    # Build the city graph
    g = buildCityGraph(Digraph)
    # Find the shortest path
    sp = shortestPath(g, g.getNode(source), g.getNode(destination), toPrint=True)
    if sp is not None:  # If a path exists
        print('Shortest path from', source, 'to', destination, 'is', printPath(sp))
    else:
        print('There is no path from', source, 'to', destination)



testSP('Boston', 'Phoenix')
```

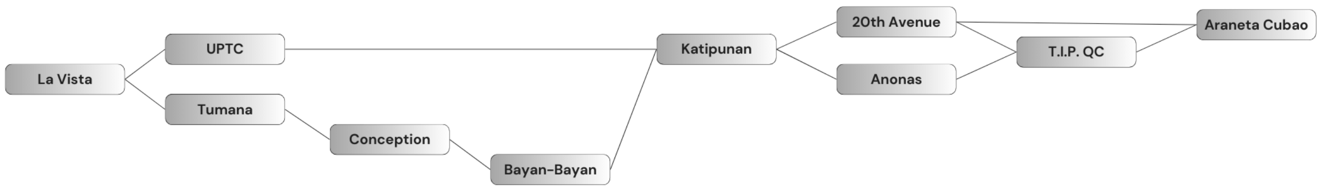
```
Current BFS path: Boston
Current BFS path: Boston->Providence
Current BFS path: Boston->New York
Current BFS path: Boston->Providence->New York
Current BFS path: Boston->New York->Chicago
Current BFS path: Boston->Providence->New York->Chicago
Current BFS path: Boston->New York->Chicago->Denver
Current 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
```

As for the Breadth-First Search (BFS), in contrast to Depth-First Search (DFS), investigates every neighbor of a node. The BFS output is created as follows.

This demonstrates how Breadth-First Search systematically explores nodes level by level, ensuring that it finds the shortest path to the destination node in terms of the number of edges traversed.

Map Graph

Marikina City to Quezon City



```
def buildCityGraph(graphType):
    g = graphType()
    for name in ('La Vista', 'UPTC', 'Tumana', 'Conception', 'Bayan-Bayan', 'Katipunan', 'Anonas', '20th Avenue', 'T.I.P. QC', 'Araneta Cuba
        #Create 10 nodes
        g.addNode(Node(name))
    g.addEdge(Edge(g.getNode('La Vista'), g.getNode('UPTC')))
    g.addEdge(Edge(g.getNode('La Vista'), g.getNode('Tumana')))
    g.addEdge(Edge(g.getNode('Tumana'), g.getNode('La Vista')))
    g.addEdge(Edge(g.getNode('Tumana'), g.getNode('Conception')))
    g.addEdge(Edge(g.getNode('Conception'), g.getNode('Bayan-Bayan')))
    g.addEdge(Edge(g.getNode('Bayan-Bayan'), g.getNode('Katipunan')))
    g.addEdge(Edge(g.getNode('UPTC'), g.getNode('Katipunan')))
    g.addEdge(Edge(g.getNode('Katipunan'), g.getNode('Anonas')))
    g.addEdge(Edge(g.getNode('Katipunan'), g.getNode('20th Avenue')))
    g.addEdge(Edge(g.getNode('Anonas'), g.getNode('T.I.P. QC')))
    g.addEdge(Edge(g.getNode('20th Avenue'), g.getNode('T.I.P. QC')))
    g.addEdge(Edge(g.getNode('T.I.P. QC'), g.getNode('Araneta Cubao')))
    g.addEdge(Edge(g.getNode('20th Avenue'), g.getNode('Araneta Cubao')))
    return g


def DFS(graph, start, end, path=[], shortest=None, toPrint=False):
    #Performs Depth-First Search (DFS) algorithm to find the shortest path from start to end in a graph.

    path = path + [start]  # Add the current node to the path
    if toPrint:
        print('Current DFS path:', printPath(path))  # Print the current path if required
    if start == end:  # If the destination is reached, return the path
        return path
    for node in graph.childrenOf(start):  # Explore each child node
        if node not in path:  # Avoid cycles
            newPath = DFS(graph, node, end, path, shortest, toPrint)  # Recursively search for a path from the child node to the destinatior
            if newPath and (not shortest or len(newPath) < len(shortest)):  # If a path is found and it is shorter than the current shortest
                shortest = newPath  # Update the shortest path
        elif toPrint:
            print('Already visited', node)  # Print a message if the node has already been visited
    return shortest  # Return the shortest path found


testSP('La Vista', 'Araneta Cubao')
```

```
Current DFS path: La Vista
Current DFS path: La Vista->UPTC
Current DFS path: La Vista->UPTC->Katipunan
Current DFS path: La Vista->UPTC->Katipunan->Anonas
Current DFS path: La Vista->UPTC->Katipunan->Anonas->T.I.P. QC
Current DFS path: La Vista->UPTC->Katipunan->Anonas->T.I.P. QC->Araneta Cubao
Current DFS path: La Vista->UPTC->Katipunan->20th Avenue
Current DFS path: La Vista->UPTC->Katipunan->20th Avenue->T.I.P. QC
Current DFS path: La Vista->UPTC->Katipunan->20th Avenue->T.I.P. QC->Araneta Cubao
Current DFS path: La Vista->UPTC->Katipunan->20th Avenue->Araneta Cubao
Current DFS path: La Vista->Tumana
Already visited La Vista
Current DFS path: La Vista->Tumana->Conception
Current DFS path: La Vista->Tumana->Conception->Bayan-Bayan
Current DFS path: La Vista->Tumana->Conception->Bayan-Bayan->Katipunan
Current DFS path: La Vista->Tumana->Conception->Bayan-Bayan->Katipunan->Anonas
Current DFS path: La Vista->Tumana->Conception->Bayan-Bayan->Katipunan->Anonas->T.I.P. QC
Current DFS path: La Vista->Tumana->Conception->Bayan-Bayan->Katipunan->Anonas->T.I.P. QC->Araneta Cubao
Current DFS path: La Vista->Tumana->Conception->Bayan-Bayan->Katipunan->20th Avenue
Current DFS path: La Vista->Tumana->Conception->Bayan-Bayan->Katipunan->20th Avenue->T.I.P. QC
Current DFS path: La Vista->Tumana->Conception->Bayan-Bayan->Katipunan->20th Avenue->T.I.P. QC->Araneta Cubao
Current DFS path: La Vista->Tumana->Conception->Bayan-Bayan->Katipunan->20th Avenue->Araneta Cubao
Shortest path from La Vista to Araneta Cubao is La Vista->UPTC->Katipunan->20th Avenue->Araneta Cubao
```

```python
def buildCityGraph(graphType):
    g = graphType()
    for name in ('La Vista', 'UPTC', 'Tumana', 'Conception', 'Bayan-Bayan', 'Katipunan', 'Anonas', '20th Avenue', 'T.I.P. QC', 'Araneta Cuba
        #Create 10 nodes
        g.addNode(Node(name))
    g.addEdge(Edge(g.getNode('La Vista'), g.getNode('UPTC')))
    g.addEdge(Edge(g.getNode('La Vista'), g.getNode('Tumana')))
    g.addEdge(Edge(g.getNode('Tumana'), g.getNode('La Vista')))
    g.addEdge(Edge(g.getNode('Tumana'), g.getNode('Conception')))
    g.addEdge(Edge(g.getNode('Conception'), g.getNode('Bayan-Bayan')))
    g.addEdge(Edge(g.getNode('Bayan-Bayan'), g.getNode('Katipunan')))
    g.addEdge(Edge(g.getNode('UPTC'), g.getNode('Katipunan')))
    g.addEdge(Edge(g.getNode('Katipunan'), g.getNode('Anonas')))
    g.addEdge(Edge(g.getNode('Katipunan'), g.getNode('20th Avenue')))
    g.addEdge(Edge(g.getNode('Anonas'), g.getNode('T.I.P. QC')))
    g.addEdge(Edge(g.getNode('20th Avenue'), g.getNode('T.I.P. QC')))
    g.addEdge(Edge(g.getNode('T.I.P. QC'), g.getNode('Araneta Cubao')))
    g.addEdge(Edge(g.getNode('20th Avenue'), g.getNode('Araneta Cubao')))
    return g


def BFS(graph, start, end, toPrint=False):
    #Performs Breadth-First Search (BFS) algorithm to find the shortest path from start to end in a graph.
    # Initialize the path with the starting node
    initPath = [start]
    # Initialize the queue with the initial path
    pathQueue = [initPath]
    while len(pathQueue) != 0:
        # Get and remove the oldest element in the queue
        tmpPath = pathQueue.pop(0)
        if toPrint:
            print('Current BFS path:', printPath(tmpPath))
        lastNode = tmpPath[-1]
        if lastNode == end:  # If the destination node is reached, return the path
            return tmpPath
        for nextNode in graph.childrenOf(lastNode):  # Explore each child node
            if nextNode not in tmpPath:  # Avoid cycles
                # Create a new path by adding the next node to the current path
                newPath = tmpPath + [nextNode]
                # Add the new path to the queue
                pathQueue.append(newPath)
    return None  # If no path is found, return None



testSP('La Vista', 'Araneta Cubao')
```

```
Current BFS path: La Vista
Current BFS path: La Vista->UPTC
Current BFS path: La Vista->Tumana
Current BFS path: La Vista->UPTC->Katipunan
Current BFS path: La Vista->Tumana->Conception
Current BFS path: La Vista->UPTC->Katipunan->Anonas
Current BFS path: La Vista->UPTC->Katipunan->20th Avenue
Current BFS path: La Vista->Tumana->Conception->Bayan-Bayan
Current BFS path: La Vista->UPTC->Katipunan->Anonas->T.I.P. QC
Current BFS path: La Vista->UPTC->Katipunan->20th Avenue->T.I.P. QC
Current BFS path: La Vista->UPTC->Katipunan->20th Avenue->Araneta Cubao
Shortest path from La Vista to Araneta Cubao is La Vista->UPTC->Katipunan->20th Avenue->Araneta Cubao
```

Evaluation

DFS Evaluation

When examining graphs, Depth-First Search (DFS) provides speed and simplicity. Because it is recursive, it is simple to use and comprehend. Because DFS can navigate deeply into graphs, it may be used for tasks such as cycle identification and topological sorting. However, when DFS traverses cyclic graphs improperly, it might become stuck in an endless cycle. Furthermore, it does not always discover the shortest path, even if it might do it rapidly in some circumstances. With big graphs, its stack-based approach can cause stack overflow, which needs to be carefully managed. DFS is an effective technique for graph exploration overall, but it has several drawbacks that should be taken into account.

BFS Evaluation

The most useful method for locating the shortest path in unweighted graphs is called Breadth-First Search, or BFS. By examining nodes one at a time, its methodical methodology guarantees that the shortest path is discovered quickly and effectively. BFS is appropriate for a variety of applications, such as network routing protocols and navigation systems, because to its robustness and ease of implementation. But because BFS needs to store every node at every level, it takes more memory than DFS. Furthermore, BFS could not function effectively on graphs with different edge weights. BFS is still a vital technique for graph exploration and pathfinding in spite of these drawbacks.

Double-click (or enter) to edit

Conclusion

Basic graph traversal algorithms, such as Depth-First Search (DFS) and Breadth-First Search (BFS), have a variety of uses. The shortest path between two nodes in an unweighted graph is found using BFS, which is essential for network routing protocols and navigation systems. Its methodical investigation guarantees an effective search for the shortest path. On the other hand, DFS is essential for maze solution, cycle detection, and topological sorting because to its ease of use and adaptability. It is perfect for jobs like detecting linked components in a graph because of its depth-first exploration. In a variety of domains, including computer science, data analysis, artificial intelligence, and game creation, both methods are essential.
