Name: Vinay Mamidi
ID: 109863968

CSCI-174
Assignment 3

Aim: The Aim of this assignment is to get a better understanding of graphs by exploring graphs by executing Breadth First Search(BFS) and Depth First Search(DFS) over a few graphs and generating useful graph data.

Input: As input we take 3 test graphs each with ten vertices numbered 1 through 10 with the following directed edges:

TEST GRAPH 1:

```
1 9
1 10
2 5
2 6
3 4
3 5
4 5
4 8
4 9
5 1
6 10
7 9
8 1
8 2
9 7
10 3
10 5
10 7
```

Test graph 2:

```
1 8
2 1
2 3
2 5
3 4
3 9
3 10
4 8
4 10
5 4
5 10
6 5
6 10
7 9
8 9
9 1
9 2
10 6
```

Name: Vinay Mamidi
ID: 109863968

Test graph 3:

```
1 8
2 1
2 3
2 7
3 2
3 6
4 2
5 6
7 3
7 8
8 3
8 4
8 9
9 6
9 8
9 10
10 1
10 9
```

Solution:

1. Read all the input vertices of graphs by using dictionary data structure.
2. Execute BFS and calculate the shortest path from 1 to each node, along with the sequence of nodes followed for the path.
3. Then the shortest path for each vertex is displayed along with the distance.
4. Execute DFS and generate Discover/Finish time, topological ordering, and edge types.

Code Folder Contains:

The Assignment folder consists of 3 files named init.py, dfs.py and vertex.py.

> init.py contains BFS algorithm and is the main file which is to be run in order to execute the program. It imports the vertex and dfs files.

> vertex.py converts vertex 1, 2, 3, etc. to a, b, c, etc. for storing name, discover time, and finish time for ea ch node

> dfs.py contains DFS algorithm for calculating discover/finish time, and functions for classifying the edge types and topological order.

Solution Code:

Init.py

```
from collections import defaultdict
from collections import OrderedDict
from dfs import Graph
from vertex import Vertex
```

```
# finds shortest path between 2 nodes of a graph using BFS
def bfs_shortest_path(graph, start, goal):
    # keep track of explored nodes
    explored = []
    # keep track of all the paths to be checked
    queue = [[start]]
    # keeps looping until all possible paths have been checked
    while queue:
        # pop the first path from the queue
        path = queue.pop(0)
        # get the last node from the path
        node = path[-1]
        if node not in explored:
            neighbours = graph[node]
            # go through all neighbour nodes, construct a new path and
            # push it into the queue
            for neighbour in neighbours:
                new_path = list(path)
                new_path.append(neighbour)
                queue.append(new_path)
                # return path if neighbour is goal
                if neighbour == goal:
                    return new_path

            # mark node as explored
            explored.append(node)

    # in case there's no path between the 2 nodes
    return "Connecting path doesn't exist"


def implement_bfs(nodeset):
    starting_node = 1;
    path_for_n_node = [];
    for i in range(1,11):
        if i == starting_node:
            path_for_n_node.append(i)
            print i , '\t  :\t\t' , path_for_n_node.__len__()-1, '\t\t'
,path_for_n_node
        else:
            path_for_n_node = bfs_shortest_path(nodeset,starting_node,i)
            print i, '\t  :\t\t', path_for_n_node.__len__() - 1, '\t\t',
path_for_n_node
        path_for_n_node = []

def graphfunction():
    print "*********** Output for graph 1 ***************\n"
    f = open('input.txt', 'r')
    node = []
    connecting_node = []
    for line in f:
        temp = line.split(" ");
        node.append(int(temp[0].strip()))
        connecting_node.append(int(temp[1].strip()))
    counter = 1
    temp = []
    nodeset = {1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: [], 8: [], 9: [], 10: []}
    while (counter <= 10):
        temp = [];
        for i in range(0, node.__len__()):
            if (node[i] == counter):
                temp.append(connecting_node[i])
        nodeset[counter] = temp
```

```
        counter += 1;
    print 'Vertex  Distance  [Path]'
    implement_bfs(nodeset)

    a = Vertex('1')
    b = Vertex('2')
    c = Vertex('3')
    d = Vertex('4')
    e = Vertex('5')
    f = Vertex('6')
    g = Vertex('7')
    h = Vertex('8')
    i = Vertex('9')
    j = Vertex('10')

    # directed graph in form of vertices for keeping track of discovery and finish
time of a node.
    G = Graph(OrderedDict(
        [(a, [i, j]), (b, [e, f]), (c, [d, e]), (d, [e, h, i]), (e, [a]), (f, [j]),
(g, [i]), (h, [a, b]), (i, [g]),
         (j, [c, e, g])]))
    G.DFS()
    G.classifyedges()
    G.toposort()
    print
"*********************************************************************************\n\
n\n\n"


def graphfunction2():
    print "************* Output for graph 2 *******************\n"
    f = open('input2.txt', 'r')
    node = []
    connecting_node = []
    for line in f:
        temp = line.split(" ");
        node.append(int(temp[0].strip()))
        connecting_node.append(int(temp[1].strip()))
    counter = 1
    temp = []
    nodeset = {1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: [], 8: [], 9: [], 10: []}
    while (counter <= 10):
        temp = [];
        for i in range(0, node.__len__()):
            if (node[i] == counter):
                temp.append(connecting_node[i])
        nodeset[counter] = temp
        counter += 1;
    print 'Vertex  Distance  [Path]'
    implement_bfs(nodeset)
    print nodeset
    a = Vertex('1')
    b = Vertex('2')
    c = Vertex('3')
    d = Vertex('4')
    e = Vertex('5')
    f = Vertex('6')
    g = Vertex('7')
    h = Vertex('8')
    i = Vertex('9')
    j = Vertex('10')

    # directed graph in form of vertices for keeping track of discovery and finish
```

```
time of a node.
    G = Graph(OrderedDict(
        [(a, [h]), (b, [a, c, e]), (c, [d, i, j]), (d, [h, j]), (e, [d, j]), (f, [e,
j]), (g, [i]), (h, [i]), (i, [a, b]),
         (j, [f])]))


    G.DFS()
    G.classifyedges()
    G.toposort()
    print
"*********************************************************************************\n\
n\n\n"


def graphfunction3():
    print "************* Output for graph 3 ***************\n"
    f = open('input3.txt', 'r')
    node = []
    connecting_node = []
    for line in f:
        temp = line.split(" ");
        node.append(int(temp[0].strip()))
        connecting_node.append(int(temp[1].strip()))
    counter = 1
    temp = []
    nodeset = {1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: [], 8: [], 9: [], 10: []}
    while (counter <= 10):
        temp = [];
        for i in range(0, node.__len__()):
            if (node[i] == counter):
                temp.append(connecting_node[i])
        nodeset[counter] = temp
        counter += 1;
    print 'Vertex  Distance  [Path]'
    implement_bfs(nodeset)
    print nodeset

    a = Vertex('1')
    b = Vertex('2')
    c = Vertex('3')
    d = Vertex('4')
    e = Vertex('5')
    f = Vertex('6')
    g = Vertex('7')
    h = Vertex('8')
    i = Vertex('9')
    j = Vertex('10')

    # directed graph in form of vertices for keeping track of discovery and finish
time of a node.
    G = Graph(OrderedDict(
        [(a, [h]), (b, [a, c, g]), (c, [b, f]), (d, [ b]), (e, [f]), (f, []), (g, [c,
h]), (h, [c, d, i]), (i, [f, h, j]),
         (j, [a, i])]))
    G.DFS()
    G.classifyedges()
    G.toposort()
    print
"*********************************************************************************\n\
n\n\n"
```

```
def main():
    graphfunction();
    graphfunction2();
    graphfunction3();

if __name__ == "__main__":
    main();
```

dfs.py

```
# Depth-First Search (DFS)

from collections import OrderedDict
from vertex import Vertex

class Graph(object):

    def __init__(self, G):
        self.G = G
        self.timestamp = 0
        self.finished = []

    def reset(self):
        self.finished = []
        self.timestamp = 0
        for v in self.G.keys():
            v.reset()

    def DFSvisit(self, s):
        self.timestamp += 1
        s.start = self.timestamp

        for v in self.G[s]:
            if v.start == None:
                v.level  = s.level + 1
                v.parent = s
                self.DFSvisit(v)

        self.timestamp += 1
        self.finished.append(s)
        s.finish = self.timestamp


    def DFS(self):
        self.reset()
        for v in self.G.keys():
            if v.start == None:
                v.level = 0
                self.DFSvisit(v)

    def classifyedges(self):
        tree = []
        cross = []
        forward = []
        backward = []
        if self.timestamp == 0:
            print "Error: need to run DFS first!"
            return
        for v in self.G.keys():
            print 'discover/finish', v, ':  ', v.start, ' ', v.finish;
        for v in self.G.keys():
            for u in self.G[v]:
                if u.parent == v:
```

```
                    tree.append([int(v.name), int(u.name)])
                elif v.start < u.start and v.finish > u.finish:
                    forward.append([int(v.name), int(u.name)])
                elif v.start > u.start and v.finish < u.finish:
                    backward.append([int(v.name), int(u.name)])
                else:
                    cross.append([int(v.name), int(u.name)])
        print 'tree edge :' , tree
        print 'forward edge :', forward
        print 'backward edge :', backward
        print 'cross edge :', cross

    def toposort(self):
        print "toposort:", " ".join([str(v) for v in reversed(self.finished)])
```

Vertex.py

```
# vertex class

class Vertex(object):

    def __init__(self, name):
        self.level  = None
        self.parent = None
        self.start  = None
        self.finish = None
        self.name   = name

    def reset(self):
        self.level  = None
        self.parent = None
        self.start  = None
        self.finish = None

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

Output Screenshots:

Output for Input.txt:

```
Vertex  Distance  [Path]
1     :     0       [1]
2     :     3       [1, 8, 3, 2]
3     :     2       [1, 8, 3]
4     :     2       [1, 8, 4]
5     :     28      Connecting path doesn't exist
6     :     3       [1, 8, 3, 6]
7     :     4       [1, 8, 3, 2, 7]
8     :     1       [1, 8]
9     :     2       [1, 8, 9]
10    :     3       [1, 8, 9, 10]
{1: [8], 2: [1, 3, 7], 3: [2, 6], 4: [2], 5: [6], 6: [], 7: [3, 8], 8: [3, 4, 9], 9: [6, 8, 10], 10: [1, 9]}
discover/finish 1 :   1   18
discover/finish 2 :   4   7
discover/finish 3 :   3   10
discover/finish 4 :   11   12
discover/finish 5 :   19   20
discover/finish 6 :   8   9
discover/finish 7 :   5   6
discover/finish 8 :   2   17
discover/finish 9 :   13   16
discover/finish 10 :   14   15
tree edge : [[1, 8], [2, 7], [3, 2], [3, 6], [8, 3], [8, 4], [8, 9], [9, 10]]
forward edge : []
backward edge : [[2, 1], [2, 3], [7, 3], [7, 8], [9, 8], [10, 1], [10, 9]]
cross edge : [[4, 2], [5, 6], [9, 6]]
toposort: 5 1 8 9 10 4 3 6 2 7
```

Output for Input2.txt:-

```
Vertex  Distance  [Path]
1     :     0       [1]
2     :     3       [1, 8, 9, 2]
3     :     4       [1, 8, 9, 2, 3]
4     :     5       [1, 8, 9, 2, 3, 4]
5     :     4       [1, 8, 9, 2, 5]
6     :     6       [1, 8, 9, 2, 3, 10, 6]
7     :     28      Connecting path doesn't exist
8     :     1       [1, 8]
9     :     2       [1, 8, 9]
10    :     5       [1, 8, 9, 2, 3, 10]
{1: [8], 2: [1, 3, 5], 3: [4, 9, 10], 4: [8, 10], 5: [4, 10], 6: [5, 10], 7: [9], 8: [9], 9: [1, 2], 10: [6]}
discover/finish 1 :   1   18
discover/finish 2 :   4   15
discover/finish 3 :   5   14
discover/finish 4 :   6   13
discover/finish 5 :   9   10
discover/finish 6 :   8   11
discover/finish 7 :   19   20
discover/finish 8 :   2   17
discover/finish 9 :   3   16
discover/finish 10 :   7   12
tree edge : [[1, 8], [2, 3], [3, 4], [4, 10], [6, 5], [8, 9], [9, 2], [10, 6]]
forward edge : [[2, 5], [3, 10]]
backward edge : [[2, 1], [3, 9], [4, 8], [5, 4], [5, 10], [6, 10], [9, 1]]
cross edge : [[7, 9]]
toposort: 7 1 8 9 2 3 4 10 6 5
```

Output for Input3.txt:-

```
Vertex  Distance  [Path]
1     :     0       [1]
2     :     3       [1, 8, 3, 2]
3     :     2       [1, 8, 3]
4     :     2       [1, 8, 4]
5     :     28      Connecting path doesn't exist
6     :     3       [1, 8, 3, 6]
7     :     4       [1, 8, 3, 2, 7]
8     :     1       [1, 8]
9     :     2       [1, 8, 9]
10    :     3       [1, 8, 9, 10]
{1: [8], 2: [1, 3, 7], 3: [2, 6], 4: [2], 5: [6], 6: [], 7: [3, 8], 8: [3, 4, 9], 9: [6, 8, 10], 10: [1, 9]}
discover/finish 1 :   1   18
discover/finish 2 :   4   7
discover/finish 3 :   3   10
discover/finish 4 :   11   12
discover/finish 5 :   19   20
discover/finish 6 :   8   9
discover/finish 7 :   5   6
discover/finish 8 :   2   17
discover/finish 9 :   13   16
discover/finish 10 :   14   15
tree edge : [[1, 8], [2, 7], [3, 2], [3, 6], [8, 3], [8, 4], [8, 9], [9, 10]]
forward edge : []
backward edge : [[2, 1], [2, 3], [7, 3], [7, 8], [9, 8], [10, 1], [10, 9]]
cross edge : [[4, 2], [5, 6], [9, 6]]
toposort: 5 1 8 9 10 4 3 6 2 7
```