

Shraddha Jamadade  
Student Id: 110287963  
CSci 174 Assignment 3  
Graphs  
Date: 11/05/2017

---

## **PROBLEM STATEMENT:**

### **Graphs Problem**

In this assignment, we are dealing with graphs and executing BFS and DFS over three graph examples and generating necessary graph data:

1. In BFS:
  - Calculate the shortest path from 1 to each node
  - Find the sequence of nodes followed for the path
2. In DFS:
  - Generate Discover and Finish time
  - Topological ordering
  - Edge Types

### **Input:**

The input given to the program is a graph with 10 vertices numbered 1 through 10 and directed edges. There are three inputs given with three different graphs.

The three inputs graphs are stored in 3 input files namely:

1. data\_1 with the given input graph 1
2. data\_2 with the given input graph 2
3. data\_3 with the given input graph 3

Example of the given 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
```

### Output:

Firstly, the program should run the **Breadth-First-Search** and calculate the shortest path from node 1 to every other node and also the sequence of nodes followed for the path.

The three BFS Outputs are stored in 3 output files namely:

1. BFS\_output\_graph1 for BFS Calculation for graph 1
2. BFS\_output\_graph2 for BFS Calculation for graph 2
3. BFS\_output\_graph3 for BFS Calculation for graph 3

Expected output for BFS (as given):

```
Vertex: Distance [Path]
1 : 0 [1]
2 : 5 [1, 10, 3, 4, 8, 2]
3 : 2 [1, 10, 3]
4 : 3 [1, 10, 3, 4]
5 : 2 [1, 10, 5]
6 : 6 [1, 10, 3, 4, 8, 2, 6]
7 : 2 [1, 9, 7]
8 : 4 [1, 10, 3, 4, 8]
9 : 1 [1, 9]
10 : 1 [1, 10]
```

Secondly, the program must run the **Depth-First-Search** and calculate the Discover/Finish time, Topological Ordering and Edge Types.

The three DFS Outputs are stored in 3 output files namely:

1. DFS\_output\_graph1 for DFS Calculation for graph 1
2. DFS\_output\_graph2 for DFS Calculation for graph 2
3. DFS\_output\_graph3 for DFS Calculation for graph 3

Expected output for BFS (as given):

```
Discover/Finish: 1 : 1 20
Discover/Finish: 2 : 12 15
Discover/Finish: 3 : 7 18
Discover/Finish: 4 : 8 17
Discover/Finish: 5 : 9 10
Discover/Finish: 6 : 13 14
Discover/Finish: 7 : 3 4
Discover/Finish: 8 : 11 16
Discover/Finish: 9 : 2 5
Discover/Finish: 10 : 6 19
Tree: [(1, 9), (1, 10), (2, 6), (3, 4), (4, 5), (4, 8), (8, 2), (9, 7), (10, 3)]
Back: [(5, 1), (6, 10), (7, 9), (8, 1)]
Forward: [(3, 5), (10, 5)]
Cross: [(2, 5), (4, 9), (10, 7)]
Vertices in Topological Order: [1, 10, 3, 4, 8, 2, 6, 5, 9, 7]
```

Execute Test Graph 2 and Test Graph 3 using the same process and generate the required output with BFS and DFS.

**Program Name: 174graphs.py**

Executed on Windows Command Prompt as: **python 174graphs.py**

Editor: Notepad++

Installed Python 2.7.6

---

**SOLUTION DESIGN:**

To solve problems on graphs, we need a mechanism for traversing the graphs. Also called as graph search algorithms which can be thought of as starting at some source vertex in a graph and “searching” the graph by going through the edges and marking the vertices. The two such algorithms for traversing the graphs are:

1. **BFS** (Breadth First Search)

This works similar to the level-order traversal in trees. BFS uses queues. BFS works level by level. Initially BFS starts at a given vertex, which is at level 0.

In the first stage, it visits all the vertices at level 1.

In the second stage, it visits all the vertices in the second level.

BFS continues this process until all the levels of the graph are completed

2. **DFS** (Depth First Search)

DFS algorithms work in a manner similar to preorder traversal of trees. Initially, all the vertices are marked as unvisited (false).

The DFS algorithm starts at a vertex  $n_1$  in the graph. It considers the edges from  $n_1$  to other vertices. If the edge leads to an already visited vertex, then backtrack to current vertex  $n_1$ .

If an edge leads to an unvisited vertex, then go to that vertex and start processing from that vertex.

That means the new vertex becomes the current vertex. Follow this process until we reach the dead-end. At this point start backtracking.

The process terminates when backtracking leads back to the start vertex.

In our program, information about three given graphs is stored in data files (input files).

These files are given as input to the program.

The program outputs the result in a separate text files.

For example for test graph 1, it takes input from file data\_1 and gives result of the BFS output data and DFS output data in files BFS\_output\_graph1 and DFS\_output\_graph2 respectively.

And same for test graph 2 and test graph 3.

In the input files, each line is an edge in a directed graph.

For example in the first input graph,

```
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
```

We can read the graphs as key value pairs. Each key corresponds to one of the graph nodes. Each value is a set of nodes (nodes which the current node directs to). This can also be represented as:

```
{
1: {9, 10},
2: {5, 6},
3: {4, 5},
4: {5, 8, 9},
5: {1},
6: {10},
7: {9},
8: {1, 2},
9: {7},
10: {3, 5, 7}
}
```

Thus the program initially reads the input files. It removes the new line character by using `rstrip()` function. Split the resultant string using whitespaces as a separator.

Convert the two separate strings into integers. All these operations are performed in the **ReadInputGraph** Function.

In this program we are using the default dictionary. A new dictionary entry is created for a key.

To keep track of the visited nodes, we keep a set data structure named `visitedNodes`. Thus we can check if the node was visited or not. Depending on whether the value is present in `visitedNode` set.

A queue of nodes is also initialized. It is a list data structure. It is used to keep track of the nodes which are discovered and we need to visit. This is used to maintain the order of nodes to be visited.

The pop(0) function is used to remove, extract the first element of the queue. This is done in the **getBFS** Function.

**getBFSedges** function is used to find the shortest path from the source node to the destination node. The queue stores values in the required format. The queue is traversed to check if it contains values. While traversing, if destination node is achieved, then return path else add current node to queue. If queue contains only one node, return its value.

The **timeDFS** Function is used to calculate the traverse time. Initializes the timeToTraverse using default dictionary, and count.

The **getDFSedges** function is used to get the nodes of the given input graph in DFS format. Descends object is initialized using default dictionary. The back, cross and forward edges for graphs are computed in this function. The graph is traversed. The result is obtained in the expected output format mentioned in the problem.

The **orderInDFS** function is used to compute the topological order in DFS for the given graphs. The input files are read and traversed.

The functions **getBFSOutput** and **getDFSOutput** is used to generate output files containing the required BFS and DFS output data of the given graphs. And the required data is returned.

The program file (174graphs.py) contains more documentation of the program.

---

## CODE:

Code file name is **174graphs.py** is attached with is submission  
It contains the code and documentation of the code

---

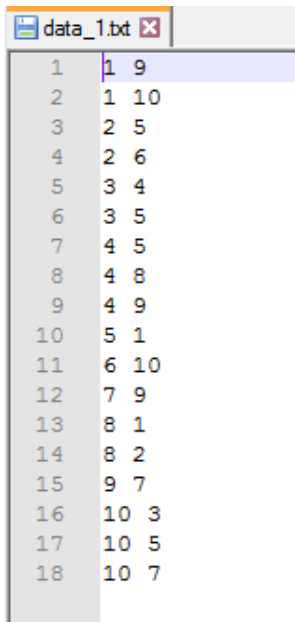
## INPUTS:

### INPUT FILE 1:

It is a text file containing the TEST GRAPH 1.

File name: data\_1

File contents:



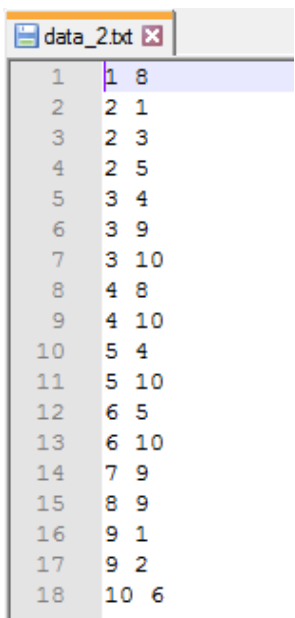
1	1	9
2	1	10
3	2	5
4	2	6
5	3	4
6	3	5
7	4	5
8	4	8
9	4	9
10	5	1
11	6	10
12	7	9
13	8	1
14	8	2
15	9	7
16	10	3
17	10	5
18	10	7

### INPUT FILE 2:

It is a text file containing the TEST GRAPH 2.

File name: data\_2

File contents:



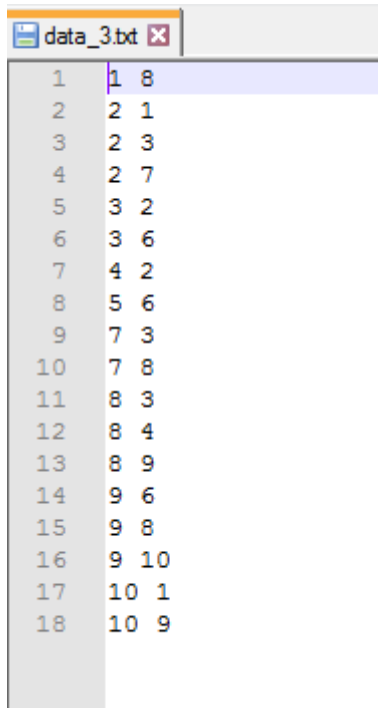
1	1	8
2	2	1
3	2	3
4	2	5
5	3	4
6	3	9
7	3	10
8	4	8
9	4	10
10	5	4
11	5	10
12	6	5
13	6	10
14	7	9
15	8	9
16	9	1
17	9	2
18	10	6

### INPUT FILE 3:

It is a text file containing the TEST GRAPH 3.

File name: data\_3

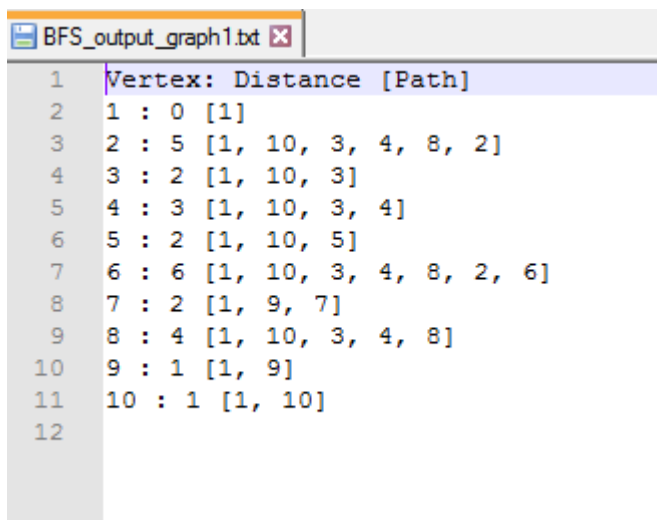
File contents:



1	1	8
2	2	1
3	2	3
4	2	7
5	3	2
6	3	6
7	4	2
8	5	6
9	7	3
10	7	8
11	8	3
12	8	4
13	8	9
14	9	6
15	9	8
16	9	10
17	10	1
18	10	9

### OUTPUT FILES:

BFS Output for TEST GRAPH 1:



1	Vertex: Distance [Path]
2	1 : 0 [1]
3	2 : 5 [1, 10, 3, 4, 8, 2]
4	3 : 2 [1, 10, 3]
5	4 : 3 [1, 10, 3, 4]
6	5 : 2 [1, 10, 5]
7	6 : 6 [1, 10, 3, 4, 8, 2, 6]
8	7 : 2 [1, 9, 7]
9	8 : 4 [1, 10, 3, 4, 8]
10	9 : 1 [1, 9]
11	10 : 1 [1, 10]
12	

BFS Output for TEST GRAPH 2:

```
BFS_output_graph2.txt x
1 Vertex: Distance [Path]
2 1 : 0 [1]
3 2 : 3 [1, 8, 9, 2]
4 3 : 4 [1, 8, 9, 2, 3]
5 4 : 5 [1, 8, 9, 2, 3, 4]
6 5 : 4 [1, 8, 9, 2, 5]
7 6 : 6 [1, 8, 9, 2, 3, 10, 6]
8 7 : 0 [1]
9 8 : 1 [1, 8]
10 9 : 2 [1, 8, 9]
11 10 : 5 [1, 8, 9, 2, 3, 10]
12
```

BFS Output for TEST GRAPH 3:

```
BFS_output_graph3.txt x
1 Vertex: Distance [Path]
2 1 : 0 [1]
3 2 : 3 [1, 8, 3, 2]
4 3 : 2 [1, 8, 3]
5 4 : 2 [1, 8, 4]
6 5 : 0 [1]
7 6 : 3 [1, 8, 9, 6]
8 7 : 4 [1, 8, 3, 2, 7]
9 8 : 1 [1, 8]
10 9 : 2 [1, 8, 9]
11 10 : 3 [1, 8, 9, 10]
12
```



### DFS Output for TEST GRAPH 1:

```
DFS_output_graph1.txt
1 Discover/Finish: 1 : 1 20
2 Discover/Finish: 2 : 10 15
3 Discover/Finish: 3 : 7 18
4 Discover/Finish: 4 : 8 17
5 Discover/Finish: 5 : 11 12
6 Discover/Finish: 6 : 13 14
7 Discover/Finish: 7 : 3 4
8 Discover/Finish: 8 : 9 16
9 Discover/Finish: 9 : 2 5
10 Discover/Finish: 10 : 6 19
11 Tree: [(1, 9), (9, 7), (1, 10), (10, 3), (3, 4), (4, 8), (8, 2), (2, 5), (2, 6)]
12 Back: [(5, 1), (8, 1), (7, 9), (6, 10)]
13 Forward: [(3, 5), (4, 5), (10, 5)]
14 Cross: [(4, 9), (10, 7)]
15 Vertices in topological order: [1, 9, 7, 10, 3, 4, 8, 2, 5, 6]
16
```

### DFS Output for TEST GRAPH 2:

```
DFS_output_graph2.txt
1 Discover/Finish: 1 : 1 18
2 Discover/Finish: 2 : 4 15
3 Discover/Finish: 3 : 5 14
4 Discover/Finish: 4 : 9 10
5 Discover/Finish: 5 : 8 11
6 Discover/Finish: 6 : 7 12
7 Discover/Finish: 7 : None None
8 Discover/Finish: 8 : 2 17
9 Discover/Finish: 9 : 3 16
10 Discover/Finish: 10 : 6 13
11 Tree: [(1, 8), (8, 9), (9, 2), (2, 3), (3, 10), (10, 6), (6, 5), (5, 4)]
12 Back: [(2, 1), (9, 1), (4, 8), (3, 9), (4, 10), (5, 10), (6, 10)]
13 Forward: [(2, 5), (3, 4)]
14 Cross: [(7, 9)]
15 Vertices in topological order: [1, 8, 9, 2, 3, 10, 6, 5, 4]
16
```

### DFS Output for TEST GRAPH 3

```
DFS_output_graph3.txt
1 Discover/Finish: 1 : 1 18
2 Discover/Finish: 2 : 10 13
3 Discover/Finish: 3 : 9 14
4 Discover/Finish: 4 : 15 16
5 Discover/Finish: 5 : None None
6 Discover/Finish: 6 : 6 7
7 Discover/Finish: 7 : 11 12
8 Discover/Finish: 8 : 2 17
9 Discover/Finish: 9 : 3 8
10 Discover/Finish: 10 : 4 5
11 Tree: [(1, 8), (8, 9), (9, 10), (9, 6), (8, 3), (3, 2), (2, 7), (8, 4)]
12 Back: [(2, 1), (10, 1), (2, 3), (7, 3), (7, 8), (9, 8), (10, 9)]
13 Forward: []
14 Cross: [(3, 6), (4, 2), (5, 6)]
15 Vertices in topological order: [1, 8, 9, 10, 6, 3, 2, 7, 4]
16
```