ASSIGNMENT 3: Graphs

Abhishek Gupta

CSCI 174

5 November 2017

DISCLAIMER

All source, input, and output files along with this documentation / report will be submitted through Blackboard and also made available through GitHub.

I would also consider volunteering to present my code IF my code is deemed correct.

Table of Contents

Abhishek Gupta

CSCI 174

5 November 2017

Assignment 3: Graphs

Problem Description

We need to write a program that takes in a directed graph and outputs useful data for that graph. In the instructions, we are given 3 graphs:

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
```

For the above test graph 1, we must run Breadth-First Search (BFS) in order to calculate the shortest path from 1 to each node. We must also calculate the sequence of nodes followed for the path.

For the given test graph 1, we are given the following output for the BFS operation:

```
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]
```

We will then need to execute the DFS operation and generate Discover/Finish Time, Topological Ordering, and Edge Types as shown below:

```
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]
```

We need to then repeat the above steps for test graph 2 and test graph 3. We aren't given the output results for these graphs so our program has to generate those.

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
```

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
```

After we have finished our program, we must write a report (PDF) containing the results and information about the code that we developed to produce the required results. This report will follow a similar format to the first assignment. Along with the PDF, we must also upload all code that was used to produce the results.

Assignment Assumptions

We were told by Dr. Ruby that our program would only need to run for 3 possible inputs. The manner in which our program takes input and outputs the results was up to our discretion as long as the results are outputted in the given manner. The option of hardcoding the 3 given inputs was made available to us. Additionally, this program is only made for directed graphs with 10 vertices (numbered 1 to 10).

Given the above information, my assumptions are the following:

1. The program can have file arguments as input or have the input data hardcoded.
2. The program can output the results in a text file or into the console.
3. I only need to make my program accommodate 3 possible inputs. I will make a more generalized program after I have submitted the assignment for the sake of time.
4. Inputs are directed graphs with 10 vertices.
5. The instructions never stated whether the BFS and DFS outputs all had to be in the same file. My program outputs the BFS and DFS data for a given graph in separate

files. For example, my program takes in the test graph 1 data from a text file and then outputs a BFS text file and a DFS text file. Given the open-ended problem description, I assumed this was ok.

6. The BFS and DFS answers are given for test graph 1, but not for test graph 2 or test graph 3. My assumption is that the output BFS and DFS files must follow the same format as those for test graph 1.

Solution Design

Reading the Graphs

The program takes in input in the form of text files. These text files are called input_1.txt, input_2.txt, and input_3.txt and they hold the data for test graph 1, test graph 2, and test graph 3, respectively. We first needed to figure out how our program reads / interprets the data in our text files. Each graph is represented by a text file where each line is an edge in a directed graph.

For example, in the following file (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
```

Each line or row represents a directed edge. Additionally, we read this graph as a dictionary or key-value pairs. Each key corresponds to one of the graph nodes (1,2,3, etc.). Each value is a set of nodes which are children of the current nodes. In other words, there is an edge from the current node to that one.

The resulting representation will look like the following:

{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}}

We see that node 1 has two outgoing edges: to node 9 and to node 10. Suppose we need to read the filename input_1.txt. We would do the following:

1. Read each line, which looks like "1 10\n".
2. Remove the newline '\n' character by using rstrip().
3. Split the resulting string (such as "1 10") into two numbers using whitespace (' ') as a separator: split(' ').
4. Now we have two strings "1" and "10", and we can convert them to integers using int().
5. Update the dictionary: add "10" to the set of edges originating from "1"

In the code, this part looks like that:

In [ ]:
```
    # Open the file
    with open(filename) as input_data:
        # Read each line from the file
        for line in input_data:
            # Extract two values separated by whitespace; ignore newline character
            a, b = line.rstrip().split(' ')
            # Convert extracted values into integers
            a, b = int(a), int(b)
            # Add the extracted edge into the graph
            graph[a].add(b)
```

From Python collections module, we are also including “defaultdict.” We use "default dictionaries" to collect information about the graph. We then initialize an empty dictionary which would contain sets as values. The only difference with traditional dictionaries is that we don't need to double check if the dictionary already contains a specific key. If the key is completely new for this dictionary, the corresponding value will be initialized to an empty set.

Our next challenge is to check every node for outgoing connections. If our graph contains an isolated node or a node with no outgoing connections, then it will not be represented as a key of the graph. We want to fix this by explicitly adding it to the graph with an empty set of outgoing connections.

```
    # Review all nodes
    for i in range(1, len(graph.keys()) + 1):
        # if the node is not present in the set og graph keys
        if i not in set(graph.keys()):
            # then initialize it as an empty set
            graph[i] = set()
```

BFS Design

To keep track of visited nodes, we keep the data structure “visited” which is a set (a collection of elements with no specific order). As a result, we can quickly check if a node was visited or not: if node not in visited, check if the value of variable node is present in the set visited.

To add a new element to a set, we use add(). For example, “visited.add(node)”. To find the difference between two sets, we use 'minus' ('-'). For example, “graph[node] – visited” results in the difference between two sets: “graph[node]” and “visited.” By our definition, “graph[node]” is a set of nodes which are directly connected to a given node.

We also need a queue of nodes to visit in the future. To keep track of the queue of discovered nodes which we need to visit, we use the queue data structure which is a list (a collection of elements which have a specific order). This helps us to visit the required nodes in the required order. To extract and delete the first element of the queue, we can use the “pop(0)” method: node = queue.pop(0).

```python
def bfs(graph, start):
    """Breadth-first search in a graph starting from the node 'start'.
    """
    # Initialize the set of visited nodes + queue of the nodes to visit
    visited, queue = set(), [start]
    # While queue is not empty = we still have nodes to visit:
    while queue:
        # Return and remove the first element of the queue
        node = queue.pop(0)
        # If node was not visited yet
        if node not in visited:
            # Mark it as visited
            visited.add(node)
            # Add outgoing connections from this node to the queue (except visited)
            queue.extend(graph[node] - visited)
    # Return the list of visited nodes in the right order
    return visited

def bfs_paths(graph, start, end):
    """Find shortest paths in the graph from node 'start' to node 'finish'.
    """
    # Initialize the queue as a tuple: starting node, starting path
    queue = [(start, [start])]
    # While queue is not empty
    while queue:
        # Return and remove the first element of the queue
        (node, path) = queue.pop(0)
        # For each outgoing connection which wasn't visited yet
        for next in graph[node] - set(path):
            # If we reached our destination node
            if next == end:
                # Return the resulting path
                return path + [next]
            else:
                # Add the current node to the path, and continue search
                queue.append((next, path + [next]))
    # If the queue is empty, return the starting node only
    return [start]
```

Next we need to write the output. Let's say we have a graph:

```python
graph = read_graph("input_1.txt")

graph
```

```
{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}}
```

We can call BFS to return all paths from one node to another one, for example, from 1 to 2:

```python
bfs_paths(graph, 1, 2)
```

```
[1, 10, 3, 4, 8, 2]
```

The goal is to generate paths from node 1 to all existing nodes in the graph. So we need a loop:

```
# For each node in the graph
for node in range(1, len(graph.keys()) + 1):
   # Seach for shortest paths from node #1 to this node
    paths = bfs_paths(graph, 1, node)
    print('Paths from 1 to ', node, ' (length = ', len(paths) - 1, '): ', paths)
```

```
Paths from 1 to  1  (length =  0 ):  [1]
Paths from 1 to  2  (length =  5 ):  [1, 10, 3, 4, 8, 2]
Paths from 1 to  3  (length =  2 ):  [1, 10, 3]
Paths from 1 to  4  (length =  3 ):  [1, 10, 3, 4]
Paths from 1 to  5  (length =  2 ):  [1, 10, 5]
Paths from 1 to  6  (length =  6 ):  [1, 10, 3, 4, 8, 2, 6]
Paths from 1 to  7  (length =  2 ):  [1, 9, 7]
Paths from 1 to  8  (length =  4 ):  [1, 10, 3, 4, 8]
Paths from 1 to  9  (length =  1 ):  [1, 9]
Paths from 1 to  10  (length =  1 ):  [1, 10]
```

We subtract 1 to get the length of the path in edges (not nodes). Now we need to format the output in the correct way in order to write it in the output file.

The next step involves applying the algorithm to our input files which will generate three output files: bfs_1.txt, bfs_2.txt, and bfs_3.txt.

Complete Code

```
def write_bfs_output(input_filename, output_filename):
    """Create an output file with BFS data for the given input file."""
    # Read the graph
    graph = read_graph(input_filename)
    # Write the output file
    with open(output_filename, 'w') as output_file:
        # Write the header
        output_file.write('Vertex: Distance [Path]\n')
        # For each node in the graph
        for node in range(1, len(graph.keys()) + 1):
            # Seach for shortest paths from node #1 to this node
            paths = bfs_paths(graph, 1, node)
            # If paths exist
            if paths:
                # Write the node number
                output_file.write(str(node) + ' : ')
                # Write the number of edges in the path (which is the number of
 nodes - 1)
                output_file.write(str(len(paths) - 1) + ' ')
                # Write the shortest path itself
                output_file.write(str(paths) + '\n')
```

DFS Design

We are working with graphs in the following format: a list of nodes: {outgoing edges}

```
{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}}
```

We can see that node 1 has two outgoing edges: to node 9 and to node 10.

For topological order of vertices, “compute_order” is a recursive function which does the following:

Initialization

- Start with an empty set of visited nodes: visited = set() - in Python, *set* is the data structure where the order of elements is not important.
- Topological order will start with a single-element list which only contains the starting node: order = [starting_vertex] - in Python, *list* is a data structure where the order of elements is very important.

Computing the Order

If we compute the order for a given node:

1. First, we add it to the list of visited nodes.
2. Next, we analyze the list of outgoing edges which start in the given node (graph[vertex]):

   -- for each outgoing node next_vertex:

   ---- if the node is not visited yet, add this node to the list, and recursively call computing the order starting from this node.

```
    def compute_order(vertex):
        visited.add(vertex)
        for next_vertex in graph[vertex]:
            if next_vertex not in visited:
                order.append(next_vertex)
                compute_order(next_vertex)
```

The complete functions which starts with initialization, recursively computes the order, and returns the result:

```
def dfs_order(graph, starting_vertex):
    visited = set()
    order = [starting_vertex]

    def compute_order(vertex):
        visited.add(vertex)
        for next_vertex in graph[vertex]:
            if next_vertex not in visited:
                order.append(next_vertex)
                compute_order(next_vertex)

    # in this case start with just one vertex, but we could equally
    # dfs from all_vertices to product a dfs forest
    compute_order(starting_vertex)
    return order
```

Example: Topological Order of Vertices

For the following graph:

```
graph = {
    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}
}
```

We get the following result:

```
In [5]: starting_vertex = 1

        dfs_order(graph, starting_vertex)

Out[5]: [1, 9, 7, 10, 3, 4, 8, 2, 5, 6]
```

Times of Traversal For Each Node

“traverse” is a recursive function which does the following:

Initialization

- visited, a *set* of visited notes, is initialized as an empty set: visited = set()
- counter for traversal times is initialized as a list which contains a single element, zero: counter = [0]

- traversal_times, a *dictionary* of traversal times for each nodes, is initialized as a default dictionary: traversal_times = defaultdict(dict)

Recursive computation

The steps to traverse a specific node are the following:

- add this node to the list of visited nodes: visited.add(vertex)
- increase the counter: counter[0] += 1. This way, the counter will increase to reflect the number of steps required to reach this vertex.
- discovery traversal time for the given node is equal to the current value of the counter: traversal_times[vertex]['discovery'] = counter[0]
- recursively call the function for each of the next vertices which were not visited yet.
- after we have visited all the nodes, we increase the counter again and write down the 'finish' time: traversal_times[vertex]['finish'] = counter[0]

The complete code:

```
from collections import defaultdict

def dfs_times(graph, starting_vertex):
    visited = set()
    counter = [0]
    traversal_times = defaultdict(dict)

    def traverse(vertex):
        visited.add(vertex)
        counter[0] += 1
        traversal_times[vertex]['discovery'] = counter[0]

        for next_vertex in graph[vertex]:
            if next_vertex not in visited:
                traverse(next_vertex)

        counter[0] += 1
        traversal_times[vertex]['finish'] = counter[0]

    # in this case start with just one vertex, but we could equally
    # dfs from all_vertices to product a dfs forest
    traverse(starting_vertex)
    return traversal_times
```

Example: Times of traversal for each node

For the previously defined graph:

```
dfs_times(graph, starting_vertex)
```

```
defaultdict(dict,
            {1: {'discovery': 1, 'finish': 20},
             2: {'discovery': 10, 'finish': 15},
             3: {'discovery': 7, 'finish': 18},
             4: {'discovery': 8, 'finish': 17},
             5: {'discovery': 11, 'finish': 12},
             6: {'discovery': 13, 'finish': 14},
             7: {'discovery': 3, 'finish': 4},
             8: {'discovery': 9, 'finish': 16},
             9: {'discovery': 2, 'finish': 5},
             10: {'discovery': 6, 'finish': 19}})
```

Compute the tree

Initialization:

- visited, a *set* of visited notes, is initialized as an empty set: visited = set()
- tree, the tree which we are building, is initialized as an empty list: tree = []

Recursive computation:

to compute a tree which starts from a given node:

- add it to the set of visited nodes: visited.add(vertex)
- for each of the outgoing edges, if the destination is not visited yet, add this outgoing edge to the tree: tree.append((vertex, next_vertex))
- recursively compute the tree starting from the next edge.

The complete code:

```
def dfs_tree(graph, starting_vertex):
    visited = set()
    tree = []

    def compute_tree(vertex):
        visited.add(vertex)
        for next_vertex in graph[vertex]:
            if next_vertex not in visited:
                tree.append((vertex, next_vertex))
                compute_tree(next_vertex)

    # in this case start with just one vertex, but we could equally
    # dfs from all_vertices to product a dfs forest
    compute_tree(starting_vertex)
    return tree
```

Example: Tree Computed For a Graph

For the previously defined graph:

```
dfs_tree(graph, starting_vertex)
```

```
[(1, 9), (9, 7), (1, 10), (10, 3), (3, 4), (4, 8), (8, 2), (2, 5), (2, 6)]
```

Produce a List of Edges of Different Types

This function works with the data structure tree_graph which is constructed as a graph of all edges which are included in the tree:

```
    tree_graph = defaultdict(set)
    for (u, v) in tree:
        tree_graph[u].add(v)
```

This represents the same information as dfs_tree(), but in a slightly different format.

Example of usage:

```
# Compute the tree as previously
tree = dfs_tree(graph, starting_vertex)

# Compute the tree graph
tree_graph = defaultdict(set)
for (u, v) in tree:
    tree_graph[u].add(v)

tree_graph
```

```
defaultdict(set,
            {1: {9, 10}, 2: {5, 6}, 3: {4}, 4: {8}, 8: {2}, 9: {7}, 10: {3}})
```

Looking for back-, forward-, and cross-edges

Initialization:

1) all required lists are initialized as empty lists:

back = []
forward = []
cross = []

2) “get_children” is a helper function to get successors of the current node:

```
    def get_children(node):
        for child in tree_graph[node]:
            yield child
            for grandchild in get_children(child):
                yield grandchild
```

Graph traversal:

Each of the lists is filled during the graph traversal.

This is how it is implemented:

1) start with initialization:

```
    back = []
    forward = []
    cross = []
    descents_ok = defaultdict(set)
```

2) process each node from `graph_keys()`:

```
graph.keys()
```

```
dict_keys([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

The first node from this list is 1:

```
u = 1
```

We get children of this node:

```
descents = get_children(u)

list(descents)
```

```
[9, 7, 10, 3, 4, 8, 2, 5, 6]
```

We change the structure of this data to record it as a dictionary element in `descents_ok`:

```
descents = get_children(u)

for i in descents:
    descents_ok[u].add(i)

descents_ok
```

```
defaultdict(set, {1: {2, 3, 4, 5, 6, 7, 8, 9, 10}})
```

Now, we process each element v of this set:

- if there is an edge which leads from v back to the current node u, then we add it to the list of "back edges".
- if there is an edge which goes from the current node u to the next node v, and this edge is not already included in the tree, then we add it to the list of "forward edges".

Example if the current node u is equal to 1:

```
for v in descents_ok[u]:
            print('Processing the node: ', v)
            if u in graph[v]:
                print('Back edge: from ', v, ' to ', u)
                back.append((v, u))
            if v in graph[u] and (u, v) not in tree:
                print('Forward edge: from ', u, ' to ', v)
                forward.append((u, v))
```

```
Processing the node:  2
Processing the node:  3
Processing the node:  4
Processing the node:  5
Back edge: from  5  to  1
Processing the node:  6
Processing the node:  7
Processing the node:  8
Back edge: from  8  to  1
Processing the node:  9
Processing the node:  10
```

The process is repeated for each node of the graph.

Looking For Cross-Edges

If the current node is still u = 1, then we look for cross-edges in the following way:

- if there is an edge which goes from the current node to v: for v in graph[u]
- if the current node u is not a child of the next node v: if u not in descents_ok[v]
- if the next node v is not a child of the current node u: v not in descents_ok[u]

then the edge (u, v) is added to the list of "cross-edges".

In [42]:
```
    cross = []
    for v in graph[u]:
            if u not in descents_ok[v] and v not in descents_ok[u]:
                cross.append((u, v))
    cross
```

Out[42]: []

Like in the previous case, the process is repeated for each node of the graph.

The Complete Code:

```
def dfs_edges(tree, graph, order):
    tree_graph = defaultdict(set)
    for (u, v) in tree:
        tree_graph[u].add(v)

    def get_children(node):
        for child in tree_graph[node]:
            yield child
            for grandchild in get_children(child):
                yield grandchild

    back = []
    forward = []
    cross = []

    descents_ok = defaultdict(set)
    for u in graph.keys():
        descents = get_children(u)
        for i in descents:
            descents_ok[u].add(i)
        for v in descents_ok[u]:
            if u in graph[v]:
                back.append((v, u))
            if v in graph[u] and (u, v) not in tree:
                forward.append((u, v))


    for u in graph.keys():
        for v in graph[u]:
            if u not in descents_ok[v] and v not in descents_ok[u]:
                cross.append((u, v))

    return back, forward, cross
```

Example: Types of edges

```
order = dfs_order(graph, 1)

back, forward, cross = dfs_edges(tree, graph, order)

print('Back:', back)
print('Forward:', forward)
print('Cross:', cross)
```

```
Back: [(5, 1), (8, 1), (7, 9), (6, 10)]
Forward: [(3, 5), (4, 5), (10, 5)]
Cross: [(4, 9), (10, 7)]
```

Write the Output

This function calls all previous functions to provide results in the required format:

```
def write_dfs_output(input_filename, output_filename):
    """Create an output file with DFS data for the given input file."""
    # Read the graph
    graph = read_graph(input_filename)
    # Write the output file
    with open(output_filename, 'w') as output_file:
        # For each node in the graph
        for node in range(1, len(graph.keys()) + 1):
            # Seach for shortest paths from node #1 to this node
            times = dfs_times(graph, 1)
            output_file.write("Discover/Finish: "+str(node)+" : ")
            if 'discovery' not in times[node]:
                output_file.write('None None\n')
            else:
                output_file.write(str(times[node]['discovery'])+' '+str(times[no
de]['finish'])+'\n')
        tree = dfs_tree(graph, 1)
        output_file.write("Tree: "+str(tree)+'\n')

        order = dfs_order(graph, 1)
        back, forward, cross = dfs_edges(tree, graph, order)
        output_file.write("Back: "+str(back)+'\n')
        output_file.write("Forward: "+str(forward)+'\n')
        output_file.write("Cross: "+str(cross)+'\n')
        output_file.write("Vertices in topological order: "+str(order)+'\n')
```

Produce the Result

When we call the function write_dfs_output, it automatically looks for the input file, processes it, and writes the result in the output file:

```
    write_dfs_output("input_1.txt", "dfs_1.txt")
    write_dfs_output("input_2.txt", "dfs_2.txt")
    write_dfs_output("input_3.txt", "dfs_3.txt")
```

Usage Instructions

You must have Python 3 installed on your machine. Assuming Python installation is correct, cd into the folder that contains the graphs.py program. Make sure this program is in the same folder as input_1.txt, input_2.txt, and input_3.txt files. Make sure these text input files are not in a subfolder. Then type “python graphs.py” and this will generate 6 output files (2 output files per graph; a BFS and DFS file).

NOTE: Some people have Python 2 and 3 installed on the same machine. If Python 2 is your default, you will run my source code using “python3 graphs.py” which will generate the output files.

Sample Input / Output

Input for 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

```

BFS Output for Test Graph 1:

```
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]

```

DFS Output for Test Graph 1:

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

```

Input for 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
```

BFS Output for Test Graph 2:

```
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 : 0 [1]
8 : 1 [1, 8]
9 : 2 [1, 8, 9]
10 : 5 [1, 8, 9, 2, 3, 10]

```

DFS Output for Test Graph 2:

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

```

Input for 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
```

BFS Output for Test Graph 3:

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

```

DFS Output for Test Graph 3:

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

```

All Submitted Content

- graphs.py
  The source code for the program.
- Abhishek Gupta ASG3 Report.pdf
  This document.
- input_1.txt, input_2.txt, input_3.txt
  All possible input for the program. The text files contain test graph 1, test graph 2, and test graph 3, respectively.
- Output
  This folder contains the BFS and DFS output for test graph 1, test graph 2, and test graph 3. This folder exists if you don't want to run the source code and just want to view the output.