Assignment 2

Vlad Bogdan-Tudor, 1st Year, group 917, Computer Science

MAIN ASSIGNED PROBLEM

1. Write a program that, given a directed graph and two vertices, finds a lowest length path between them, by using a forward breadth-first search from the starting vertex.

This function can be used to traverse the given graph in Breadth First Search manner:

```
def bfs(self, start_vertex):
    """
    Performs a modified Breadth First Search from the given starting vertex. Once the search reaches the vertex <end_vertex> (that is, if it reaches it), the algorithm stops.
    :param start_vertex: Integer; the vertex where the Breadth First Search starts from
    :returns: dictionary visited - the keys are all the vertices from the graph, the values are truth
    values denoting whether or not that vertex is accessible from the starting vertex
    :returns: dictionary prev - the keys are all the vertices from the graph, the value of each key is the previous vertex on the path from the starting vertex to the key vertex
    :returns: dictionary dist - the keys are all the vertices from the graph, the values are the lengths of the paths from the starting vertex to that vertex or INFINITY if that vertex is not accessible from the starting vertex
    :except: GraphException - if the given <start_vertex> is not in the graph
    """
    if not self.is_vertex_in_graph(start_vertex):
        raise GraphException(f"Error! The vertex {start_vertex} is not in the graph.")
    # Initialize the visited dictionary with False values, the distance dictionary with infinity values
    # and the previous dictionary with None values
    visited = {node: False for node in self.get_all_vertices()}
    dist = {node: INFINITY for node in self.get_all_vertices()}
```

```
prev = {node: None for node in self.get_all_vertices()}
# We will treat the following list as a queue
queue = [start_vertex]
visited[start_vertex] = True
dist[start_vertex] = 0
while len(queue):
    # Get the vertex from the top of the queue and pop it from the queue
    top_of_queue = queue[0]
    queue = queue[1:]
    for neighbour in self.get_outbound_neighbours(top_of_queue):
        if not visited[neighbour]:
            queue.append(neighbour)
            visited[neighbour] = True
            dist[neighbour] = dist[top_of_queue] + 1
            prev[neighbour] = top_of_queue
            if neighbour == end_vertex:
                return visited, prev, dist
return visited, prev, dist
```

Using the above *bfs()* function, the following function finds the lowest length path between 2 given vertices:

```
    def lowest_length_path(self, start_vertex, end_vertex):
        """
        Finds the lowest length path between <start_vertex> and <end_vertex> using a forward breadth first search starting from <start_vertex>. Note: we are using a modified version of the BFS, stopping it once we get to <end_vertex>.
        :param start_vertex: Integer; The vertex where the path (and the BFS) starts
        :param end_vertex: Integer; The vertex where the path ends
        :return: A list containing the lowest length path, the first element in the list will be <start_vertex>, while the last element in the list will be <end_vertex>
        :except: GraphException - if one of the given vertices are not in the graph, OR if <end_vertex> is not accessible from <start_vertex>
        """
        if not self.is_vertex_in_graph(start_vertex):
            raise GraphException(f"Error! The starting vertex {start_vertex} is not in the graph.")
        if not self.is_vertex_in_graph(end_vertex):
            raise GraphException(f"Error! The ending vertex {end_vertex} is not in the graph.")
        visited, prev, dist = self.bfs(start_vertex)
        if not visited[end_vertex]:
            raise GraphException(f"Error! The node {end_vertex} is not accessible from node {start_vertex}.")
        path = []
        node = end_vertex
        while prev[node] is not None:
            path.append(node)
            node = prev[node]
        path.append(node)
        return path[::-1]
```

<u>BONUS PROBLEM</u>

1B. Write a program that finds the strongly-connected components of a directed graph in O(n+m) (n=no. of vertices, m=no. of arcs)

The following function will be used in the Kosaraju Algorithm as the function with the purpose of traversing the graph for the first time. The traversal will be done in Depth First Search manner:

```
def __dfs1(self, vertex, visited, stack):
    """
    Traverses all the vertices from the graph using the Depth First Search algorithm and pushes all the vertices on a stack such that the last vertex visited will be the first one pushed on the stack (so the first vertex visited will be at the top of the stack after the algorithm finishes). This function is meant to be used as the first DFS in The Kosaraju Algorithm.
    :param vertex: The current vertex that is being traversed; Integer
    :param visited: Keeps track of whether or not a vertex was already visited; List of bool values
    :param stack: The stack where we will push the visited vertices; List
    :except GraphException: If the given vertex <vertex> is not in the graph (This can only happen in the first call of the function, when the function is called by the user)
    """
    if not self.is_vertex_in_graph(vertex):
       raise GraphException(f"ERROR: The vertex {vertex} is not in the graph.")
    visited[vertex] = True
    for neighbour in self.__dict_out[vertex]:
       if not visited[neighbour]:
          self.__dfs1(neighbour, visited, stack)
    stack.append(vertex)
```

The next function will serve as the second graph traversal, the one used after we transpose the graph. It will also use DFS:

```
def __dfs2(self, vertex, visited, strongly_connected_comps):
    """
    Traverses all the vertices from the graph using the Depth First Search algorithm and appends the currently visited vertex in the last list from the <strongly_connected_comps> list. This function is meant to be used as the second DFS in The Kosaraju Algorithm.
    :param vertex: The current vertex that is being traversed; Integer
    :param visited: Keeps track of whether or not a vertex was already visited; List of bool values
    :param strongly_connected_comps: List of lists where the strongly connected components will be stored
    :except GraphException: If the given vertex <vertex> is not in the graph (This can only happen in the first call of the function, when the function is called by the user)
    """
    if not self.is_vertex_in_graph(vertex):
        raise GraphException(f"ERROR: The vertex {vertex} is not in the graph.")
    visited[vertex] = True
    strongly_connected_comps[-1].append(vertex)
    for neighbour in self.__dict_out[vertex]:
        if not visited[neighbour]:
            self.__dfs2(neighbour, visited, strongly_connected_comps)
```

The function *transposed_graph* below will be used to get a transposed version of the graph

```
def transposed_graph(self):
    """
    Transposes the graph (i.e., reverses the orientation of all the edges). This does NOT happen in-place.
    :return: A new graph (instance of the TripleDictGraph class) where each edge was obtained by reversing some edge from this graph.
    """
    new_graph = TripleDictGraph()
    for vertex in self.get_all_vertices():
        new_graph.add_vertex(vertex)
    for _from, _to, _cost in self.get_all_edges():
        new_graph.add_edge(_to, _from, _cost)
    return new_graph
```

Using all of the above functions, namely *__dfs1()*, *__dfs2()*, and *transposed_graph()*, the below function will find all of the strongly connected components of a graph and return them as lists containing the all the vertices in each strongly connected component:

```
def find_all_scc(self):
    """
    Finds all of the strongly connected components of the graph using the Kosaraju algorithm.
    :return: List of lists where each lists contains all the vertices from a strongly connected component
    """
    stack = []
    visited = [False] * self.get_no_vertices()
    for vertex in self.get_all_vertices():
        if not visited[vertex]:
            self.__dfs1(vertex, visited, stack)
    transposed_graph = self.transposed_graph()
    visited = [False] * self.get_no_vertices()
    strongly_connected_comps = []
    while stack:
        top = stack.pop()
        if not visited[top]:
            strongly_connected_comps.append([])
            transposed_graph.__dfs2(top, visited, strongly_connected_comps)
    return strongly_connected_comps
```

The above implementation of the Kosaraju Algorithm works fine for graphs with a small number of vertices. However, once the number of vertices increases the above implementation stops working. In our case, as soon as a graph with ~10k vertices is given, the algorithm produces a stackoverflow, having reached the maximum stack depth. To solve this issue and to be able to find the strongly connected components for graph with a large number of vertices, we will implement a modified version of the Kosaraju Algorithm where we won't rely on recursion anymore (so we don't increase the stack depth anymore) and instead everything will be done iteratively:

```
def kosaraju(self):
    """
    Finds all of the strongly connected components of the graph using the Kosaraju algorithm.
    :return: List of lists where each lists contains all the vertices from a strongly connected
component
    """
    nr_vertices = self.get_no_vertices()
    T, L, U = [[] for _ in range(nr_vertices)], [], [False] * nr_vertices
    for u in range(nr_vertices):
        if not U[u]:
            U[u], S = True, [u]
            while S:
                u, done = S[-1], True
                for v in self.__dict_out[u]:
                    T[v].append(u)
                    if not U[v]:
                        U[v], done = True, False
                        S.append(v)
                        break
                if done:
                    S.pop()
                    L.append(u)
    scc = [None] * nr_vertices
```

```
while L:
    r = L.pop()
    S = [r]
    if U[r]:
        U[r], scc[r] = False, r
    while S:
        u, done = S[-1], True
        for v in T[u]:
            if U[v]:
                U[v] = done = False
                S.append(v)
                scc[v] = r
                break
        if done:
            S.pop()
d = {}
for vertex in sorted(set(scc)):
    d[vertex] = []
for index, vertex in enumerate(scc):
    d[vertex].append(index)
comps = [value for value in d.values()]
return comps
```