Lab Week 2

Session: 2022 – 2026

Submitted by:

Muhammad Saad Akmal 2022-CS-148

Submitted To:

Sir Nauman Shafi

Department of Computer Science

University of Engineering and Technology

Lahore Pakistan

Problem 1:

```
class Graph:
    def __init__(self):
        self.vertices = []          # List to store vertex names
        self.edges = []             # List to store edges as tuples (start, end)
        self.is_directed = False    # To store whether the graph is directed
        self.adjacency_list = {}    # Dictionary to store adjacency list

    def read_graph_from_file(self, filename):
        """
        Reads the graph from a file with the specified format.
        Input: filename - name of the file containing the graph.
        """
        with open(filename, 'r') as file:

            lines = file.readlines()
            #SPLITING THE DATA GOT FROM FILE
            first = lines[0].strip().split('_')
            self.vertices = lines[2].strip().split()   #VERTICES NAME
            self.is_directed = True if first[1] == '1' else False #DIRECTED OR NOT
            self.edges = [line.strip() for line in lines[6::2] if line.strip()] #EDGES LIST
            for edge in self.edges:
                #SEPERATING EDGES NODES LIKE FOR AB u=A and v=B
                u, v = edge[0], edge[1]

                #MAKING A ROW FOR EACH VERTEX
                if u not in self.adjacency_list:
                    self.adjacency_list[u] = []
                if v not in self.adjacency_list:
                    self.adjacency_list[v] = []

                #ADDING SECOND VERTEX TO FIRST VERTEX NEIUGHBOUR
                self.adjacency_list[u].append(v)

                #IF GRAPGH IS UNDIRECTED THEN ADD FIRST VERTEX TO SECOND VERTEX NEIUGHBOUR TOO
                if self.is_directed == False:
                    self.adjacency_list[v].append(u)


    def get_vertex_count(self):
        """
        Returns the total number of vertices in the graph.
        Output: int - number of vertices.
        """

        return len(self.vertices)

    def get_edge_count(self):
        """
        Returns the total number of edges in the graph.
        Output: int - number of edges.
        """
        return len(self.edges)

    def is_graph_directed(self):
        """
        Returns whether the graph is directed or not.
        Output: bool - True if the graph is directed, False otherwise.
        """
        return self.is_directed

    def get_neighbors(self, vertex):
        """
        Returns the neighbors of the given vertex.
        Input: vertex - the vertex whose neighbors are to be returned.
        Output: list - list of neighboring vertices.
        """
        return self.adjacency_list.get(vertex, [])


g = Graph()


file_name = input("Enter file name: ")

g.read_graph_from_file(file_name)

v = g.get_vertex_count()
print(f'The vertices are {v}')

edges = g.get_edge_count()
print(f'The number of edges are {edges}')


directed = g.is_graph_directed()
print(f'The graph is {"Directed" if directed else "Undirected"}')



while True:
    n = input("Enter vertex to get neighbours/ Press q to exit: ")
    if n =='q':
        break
    print(f'The neighbour of {n} are {g.get_neighbors(n)}')
```

Output:

```
Enter file name: graph.txt
The vertices are 4
The number of edges are 4
The graph is Undirected
Enter vertex to get neighbours/ Press q to exit: A
The neighbour of A are ['B', 'D']
Enter vertex to get neighbours/ Press q to exit: q
```

Problem 2:

```
track = []
cycle = []

def dfs(graph,node):
    visited = []


    dfs_rec(graph , node , visited)
    return visited


def dfs_rec(graph,node,visited):
    if node in visited:
        return

    # OTHER WISE ADD IN VISTED LIST
    visited.append(node)
    track.append(node)

    #CHECKING ITS neighbors
    neighbors = graph.get_neighbors(node)

    #APPLYING SAME FUNCTION FOR ALL neighbors OF A NODE
    for i in neighbors:
        if i not in visited:
            dfs_rec(graph , i , visited)
        if i not in track and i!=node:
            cycle.extend(track)
```

Output:

```
['A', 'B', 'C', 'D']
```

Problem 3:

```
#BFS
def bfs(graph, start_vertex):

    running = []
    visited = []
    rec_bfs(visited , graph , start_vertex , running)
    return visited


#CALCULATING THE DISTANCE
def bfs_distance(graph, start_vertex,end_vertex)->int:

    distance = 0
    visited = bfs(graph , start_vertex)

   #LOOPING THROUGH NODES AND GETTING DISTANCE
    for i in visited:
      if i == end_vertex:
          break
      distance = distance + 1

    return distance


def bfs_number_of_levels(graph, start_vertex,end_vertex)->int:
    level = 0
    visited = bfs(graph , start_vertex)

   #LOOPING THROUGH NODES AND GETTING LEVEL
    for i in visited:
      if i == end_vertex:
          break
      level = level + 1

    return level
```

```python
#HELPING FUNCTION
def rec_bfs(visited , graph , node , running):

    #IF NOD IS ALREADY VISTED RETIRN I.E BASE CASE
    if node in visited:
        return

    # OTHER WISE ADD IN VISTED LIST
    visited.append(node)

    #CHECKING ITS NEIUGHBOURS
    neiughbours = graph.get_neighbors(node)


    for i in neiughbours:

        #IF ALREADY VISITED THEN CONTINUE
        if i in visited:
            continue

        #APPEND TO RUNNING QUEUE SO WE APPLY RECURSION IN FIFO ORDER NOT IN LIFO
        running.append(i)

    #LOOPING THROUGH THE QUEUE
    for i in running:
        rec_bfs(visited , graph , i , running)
```

Output:

```
['A', 'B', 'D', 'C']
2
2
```

Problem 4:

```python
visited = []
path = []
all_cycles = []


def is_Acyclic(graph , start_vertex):

    util(graph , start_vertex , None , start_vertex)

    #CHECKING IF THEIR WASD A CYCLE
    if all_cycles:
        return True , len(all_cycles) , all_cycles   #RETURN THE PATHS AND NUMBER OF CYCLES
    else:
        return False

def util(graph , node , parent , start_vertex):

        #IF NODE IS PRESENT RETURN IT
        if node in visited:
            return

        #ADDING THE NODE TO MARK AS VISITED AND ADDING IN PATH TOO
        visited.append(node)
        path.append(node)

        #GETTING NEIGHBORS
        neighbors = graph.get_neighbors(node)

        for i in neighbors: #LOOPING NEIGHBORS
            if i == start_vertex and i != parent: #CECKING IF THE NODE IS REPEATED AND IT IS NOT THE PREVIOUS NODE IN CASE OF UNDIRECTED GRAPH

                cycle = path.copy() # COPING THE PATH
                cycle.append(start_vertex) #ADD THE NODE TO SHOW A CYCLE
                all_cycles.append(cycle) #ADDING N CYCLES TO KEEP RECORD OF ALL CYCLES THAT OCCURED

            elif i not in visited: #IF NOT CYCLE MEANS NEW NODE SO CHECK IF ALREADY VISITED

                util(graph , i , node , start_vertex)  #CALLING THE FUNCTION AGAIN

        #CLEARING PATH AND VISITED WHEN ONE SIDE IS COMPLETELY VISITED
        path.pop()
        visited.remove(node)
        return None
```

Output:

```
(True, 2, [['A', 'B', 'C', 'D', 'A'], ['A', 'D', 'C', 'B', 'A']])
```