

Name: Ronit Khalate.

Roll No. 2231028

Assignment No.01

Title: Implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure

DFS – Depth First Search

Input :

```
# from a given graph
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation

class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self, v, visited):

        # Mark the current node as visited
        # and print it
        visited.add(v)
        print(v, end=' ')

        # Recur for all the vertices
        # adjacent to this vertex
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)
```

```

# The function to do DFS traversal. It uses
# recursive DFSUtil()
def DFS(self, v):

    # Create a set to store visited vertices
    visited = set()

    # Call the recursive helper function
    # to print DFS traversal
    self.DFSUtil(v, visited)

# Driver's code

# Create a graph given
# in the above diagram
if __name__ == "__main__":
    g = Graph()
    g.addEdge(0, 1)
    g.addEdge(0, 2)
    g.addEdge(1, 2)
    g.addEdge(2, 0)
    g.addEdge(2, 3)
    g.addEdge(3, 3)

    print("Following is DFS from (starting from vertex 2)")
    # Function call
    g.DFS(2)

```

Output :

Following is DFS from (starting from vertex 2)

2 0 1 3

BFS – Breadth-First Search

Input:

```
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation

class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (max(self.graph) + 1)

        # Create a queue for BFS
        queue = []

        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

            # Dequeue a vertex from
            # queue and print it
            s = queue.pop(0)
            print(s, end=" ")

            # Get all adjacent vertices of the
            # dequeued vertex s. If a adjacent
            # has not been visited, then mark it
            # visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
```

```
        queue.append(i)
        visited[i] = True

# Driver code

# Create a graph given in
# the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print("Following is Breadth First Traversal"
      " (starting from vertex 2)")
g.BFS(2)
```

Output:

Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1