**UIT 2402 – ADVANCED DATA STRUCTURES AND ALGORITHM ANALYSIS**

**EX 3: State, Space, Search**

**Name:** B.VASUNDHARA

**Roll no:** 3122 21 5002 119

**ALGORITHM:**

**BFS:**

**Step 1:** Consider the graph you want to navigate.

**Step 2:** Select any vertex in your graph (say **v1**), from which you want to traverse the graph.

**Step 3:** Utilize the following two data structures for traversing the graph.

Visited array(size of the graph)

Queue data structure

**Step 4:** Add the starting vertex to the visited array, and afterward, you add v1’s adjacent vertices to the queue data structure.

**Step 5:** Now using the FIFO concept, remove the first element from the queue, put it into the visited array, and then add the adjacent vertices of the removed element to the queue.

**Step 6:** Repeat step 5 until the queue is not empty and no vertex is left to be visited.

**DFS:**

**Step 1:** Create a set or array to keep track of visited nodes.

**Step 2:** Choose a starting node.

**Step 3:**Create an empty stack and push the starting node onto the stack.

**Step 4:**Mark the starting node as visited.

**Step 5:**While the stack is not empty, do the following:

* + Pop a node from the stack.
  + Process or perform any necessary operations on the popped node.
  + Get all the adjacent neighbors of the popped node.
  + For each adjacent neighbor, if it has not been visited, do the following:
    - Mark the neighbor as visited.
    - Push the neighbor onto the stack.

**Step 6:** Repeat step 5 until the stack is empty.

**IDDFS:**

// Returns true if target is reachable from

// src within max\_depth

**bool** IDDFS(src, target, max\_depth)

**for** limit **from** 0 **to** max\_depth

**if** DLS(src, target, limit) == **true**

**return** true

**return** **false**

**bool** DLS(src, target, limit)

**if** (src == target)

**return** **true**;

// If reached the maximum depth,

// stop recursing.

**if** (limit <= 0)

**return** **false**;

**foreach** adjacent i of src

**if** DLS(i, target, limit?1)

**return** **true**

**return** **false**

**Program Code:**

**BFS:**

# Python3 Program to print BFS traversal

# from a given source vertex. BFS(int s)

# traverses vertices reachable from s.

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

g = Graph()

n=int(input("Enter the no.of edges: "))

for i in range(n):

    print("Edge ",(i+1))

    u=int(input("Enter start node: "))

    v=int(input("Enter end node: "))

    print()

    g.addEdge(u, v)

s=int(input("Enter the starting node: "))

g.BFS(s)

**DFS:**

# Python3 program to print DFS traversal

# 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

g = Graph()

n=int(input("Enter the no.of edges: "))

for i in range(n):

    print("Edge ",(i+1))

    u=int(input("Enter start node: "))

    v=int(input("Enter end node: "))

    print()

    g.addEdge(u, v)

s=int(input("Enter the starting node: "))

g.DFS(s)

**IDS:**

# Python program to print DFS traversal from a given

# given graph

from collections import defaultdict

# This class represents a directed graph using adjacency

# list representation

class Graph:

    def \_\_init\_\_(self,vertices):

        # No. of vertices

        self.V = vertices

        # 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 to perform a Depth-Limited search

    # from given source 'src'

    def DLS(self,src,target,maxDepth):

        if src == target : return True

        # If reached the maximum depth, stop recursing.

        if maxDepth <= 0 : return False

        # Recur for all the vertices adjacent to this vertex

        for i in self.graph[src]:

                if(self.DLS(i,target,maxDepth-1)):

                    return True

        return False

    # IDDFS to search if target is reachable from v.

    # It uses recursive DLS()

    def IDDFS(self,src, target, maxDepth):

        # Repeatedly depth-limit search till the

        # maximum depth

        for i in range(maxDepth):

            if (self.DLS(src, target, i)):

                return True

        return False

N=int(input("Enter the no.of vertices: "))

n=int(input("Enter the no.of edges: "))

g = Graph(N)

for i in range(n):

    print("Edge ",(i+1))

    u=int(input("Enter start node: "))

    v=int(input("Enter end node: "))

    print()

    g.addEdge(u, v)

target=int(input("Enter the target: "))

maxDepth=int(input("Enter the maximum Depth: "))

src=int(input("Enter the source node: "))

if g.IDDFS(src, target, maxDepth) == True:

    print ("Target is reachable from source " +

        "within max depth")

else :

    print ("Target is NOT reachable from source " +

        "within max depth")

**OUTPUT:**

**BFS:**

A screenshot of a computer

Description automatically generated

**DFS:**

A screenshot of a computer

Description automatically generated

**IDS:**

A screenshot of a computer

Description automatically generated