**Exercise - 3**

**State, Space, Search Algorithms**

**Aim:** To write a python code to implement State, Space and Search algorithms

**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.

1. Visited array(size of the graph)
2. 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 neighbours of the popped node.
  + For each adjacent neighbour, if it has not been visited, do the following:
    - Mark the neighbour as visited.
    - Push the neighbour onto the stack.

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

**IDDFS:**

1. Start with a depth limit of 0.
2. Repeat the following steps:

* Initialize an empty set called "visited" to keep track of visited nodes.
* Perform a Depth-First Search (DFS) with the current depth limit by calling the DLS (Depth-Limited Search) function.
* If the DLS function returns a non-null result, it means the target node has been found. Return the result.
* Increase the depth limit by 1 and repeat the process.

1. If the entire search space has been explored and the target node is not found, return null.

The DLS (Depth-Limited Search) function is used to perform DFS:

1. If the current node is null, return null.
2. If the value of the current node matches the target value, return the current node.
3. If the depth limit reaches 0, return null.
4. If the current node is already in the "visited" set, skip it to avoid revisiting.
5. Add the current node to the "visited" set.
6. Recursively call the DLS function on each child of the current node with a decreased depth limit.
7. If any of the child searches return a non-null result, return that result.
8. If no match is found at the given depth, return null.

**Source Code:**

**BFS:**

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(list)

    def addEdge(self, u, v):

        self.graph[u].append(v)

    def BFS(self, s):

        visited = [False] \* (max(self.graph) + 1)

        queue = []

        queue.append(s)

        visited[s] = True

        while queue:

            s = queue.pop(0)

            print(s, end=" ")

            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:**

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(list)

    def addEdge(self, u, v):

        self.graph[u].append(v)

    def DFSUtil(self, v, visited):

        visited.add(v)

        print(v, end=' ')

        for neighbour in self.graph[v]:

            if neighbour not in visited:

                self.DFSUtil(neighbour, visited)

    def DFS(self, v):

        visited = set()

        self.DFSUtil(v, visited)

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)

**IDDFS:**

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self,vertices):

        self.V = vertices

        self.graph = defaultdict(list)

    def addEdge(self,u,v):

        self.graph[u].append(v)

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

        if src == target : return True

        if maxDepth <= 0 : return False

        for i in self.graph[src]:

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

                    return True

        return False

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

        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")

**Sample Input and Output:**

**BFS:**

**A screenshot of a computer program

Description automatically generated with medium confidence**

**DFS:**

**A screenshot of a computer program

Description automatically generated with medium confidence**

**IDDFS:**

**A screenshot of a computer program

Description automatically generated with medium confidence**

**Result:**

Thus, the State, Space and Search algorithms have been successfully implemented using Python code and the output is verified.