# Experiment No. 02

# Aim: Implementation of BFS, DFS, DFID

# Theory:

**BFS** is an algorithm for traversing or searching tree or graph data structures. It explores all the vertices of the graph in breadth-first manner, i.e., it visits all the neighbours of a vertex before moving to the next level. This ensures that it visits all the nodes at distance *d* from the source node before visiting nodes at distance of *d*+1. Application: BFS is used in various applications like finding the shortest path, solving puzzles with multiple solutions, analysing network broadcasting, and in general graph algorithms.

**DFS** is another graph traversal algorithm that explores as far as possible along each branch before backtracking. It uses a stack to remember which vertices to visit next. The algorithm starts at the root node and explores as far as possible along each branch before backtracking. Application: DFS is used in topological sorting, solving mazes, finding connected components in a graph, and in various applications related to graph and tree structures.

**DFID** is an algorithm that combines the benefits of both BFS and DFS. It repeatedly applies DFS with a gradually increasing depth limit until the goal is found. It starts with a depth limit of 0, then 1, then 2, and so on until the solution is found. DFID guarantees to find the shallowest goal in an unweighted graph, making it memory efficient like BFS while maintaining completeness like DFS. Application: DFID is particularly useful in scenarios where memory usage is a concern, and you want to find the shallowest goal in a graph without using excessive memory.

# Code:

**BFS**

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = [] queue = []

closed\_list=[] path=[]

def bfs(visited, graph, node,goal): visited.append(node)

path={}

path[node]=node root=[] queue.append(node)

print(f"Open List: {queue}\nClosed list: {closed\_list} ")

while queue:

m = queue.pop(0) closed\_list.append(m)

print(f"Open List: {queue}\nClosed list: {closed\_list} ")

if m==goal:

while path[m]!=m: root.append(m) m=path[m]

root.append(m) root.reverse() print(f'Path:{root}') return

for neighbour in graph[m]:

if neighbour not in visited: visited.append(neighbour) queue.append(neighbour) path[neighbour]=m

print(f'Path :{path}')

print("Following is the Breadth-First Search") bfs(visited, graph, '5','4')

**DFS**

graph={ '5':['3','7'],

'3': ['2', '4'],

'7': ['8'],

'2': [],

'4': ['8'],

'8': [],

}

visited=[] queue=[]

closed\_list=[]

def dfs(visited, graph,node,goal): path={}

path[node]=node root=[] queue.append(node)

print(f'Open List: {queue}\nClosed List: {closed\_list}\n')

while queue:

m=queue.pop() closed\_list.append(m)

print(f'Open List: {queue}\nClosed List: {closed\_list}\n')

if(m==goal):

while path[m]!=m:

root.append(m) m=path[m] root.append(m) root.reverse() print(f"\nPath: {root}") return

for neighbour in graph[m]:

if neighbour not in visited: visited.append(neighbour) queue.append(neighbour) path[neighbour]=m

print(f'Open List: {queue}\nClosed List: {closed\_list}\n') print("Path does not found")

dfs(visited,graph,'5','4')

# DFID

from collections import defaultdict

class Graph:

def init (self,vertices):

self.V = vertices # Number of 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

# Create a graph # 0

# / \

# 1 2

# / \ | \

# 3 4 5 6

g = Graph (7) g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 3)

g.addEdge(1, 4)

g.addEdge(2, 5)

g.addEdge(2, 6)

for i in range(2):

target = [5, 6]; maxDepth = [3,2]; src = 0

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

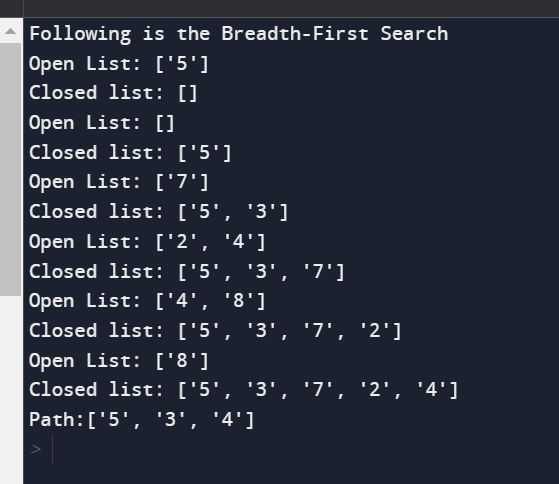
print ("Target "+str(target[i]) +" is reachable from source within max depth of "+str(maxDepth[i]))

else :

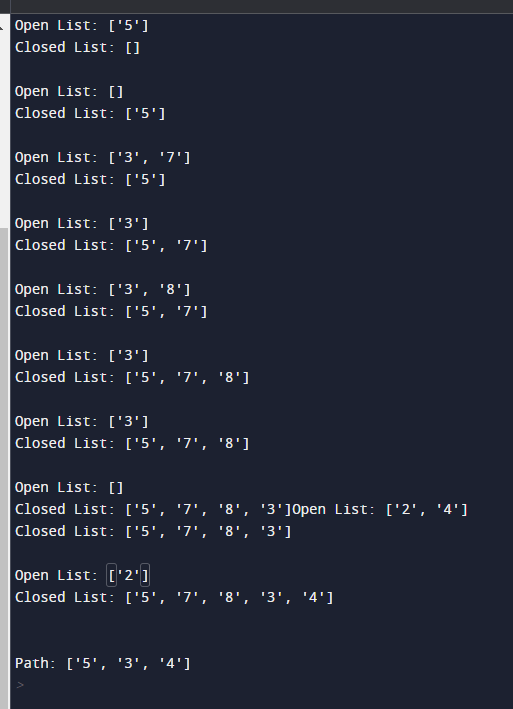
print ("Target "+str(target[i]) +" is NOT reachable from source within max depth of "+str(maxDepth[i]))

# Output:

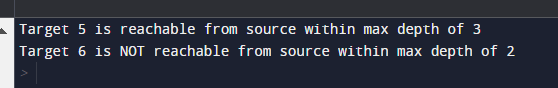
* 1. **BFS**



# DFS



* 1. **DFID**



# Conclusion:

In summary, the choice of algorithm depends on the specific requirements of the problem at hand. If finding the shortest path is crucial and memory is not a significant concern, BFS is a reliable choice. If memory efficiency is vital, especially in large graphs, DFS might be preferred, although it might not always yield the optimal solution. DFID, on the other hand, strikes a balance between optimality and memory efficiency, making it a practical choice in scenarios where both factors are important considerations. It's important to note that the performance of these algorithms can vary based on the characteristics of the graph or tree being traversed, such as its size, structure, and the location of the goal node. Therefore, it's essential to analyse the specific problem requirements and choose the appropriate algorithm accordingly.