

**TASK:1****Implementation of Graph search algorithms  
(Breadth first search and Depth First Search)**

---

Implementation of Graph search algorithms (Breadth first search and Depth First Search) using following constraints

**BFS:** Pick any node, visit the adjacent unvisited vertex, mark it as visited, display it, and insert it in a queue. If there are no remaining adjacent vertices left, remove the first vertex from the queue. Apply recursion concept to follow the above steps until the queue is empty or the desired node is found.

**DFS:** Pick any node. If it is unvisited, mark it as visited and recur on all its adjacent nodes. Repeat until all the nodes are visited, or the node to be searched is found.

**Tools- Python**

**PROBLEM STATEMENT:**  
**S3**

**CO1**

In a social networking platform, users are connected through friendships, forming a network of relationships. Each user can be represented as a node in a graph, and a friendship between two users is represented as an edge. To build a friend recommendation system, we can use graph search algorithms like Breadth-First Search (BFS) and Depth-First Search (DFS). BFS is useful for exploring the network level by level, helping to identify "friends of friends" who can be suggested as potential new connections. For example, starting from a user, BFS first finds their immediate friends, and then their friends' friends. DFS, on the other hand, explores each connection path deeply before backtracking, which is helpful for analyzing how far and in what ways users are connected throughout the network. By using both BFS and DFS, we can effectively explore the structure of the social network for various purposes like friend suggestions, connection analysis, and understanding the reach of each user.

## **BREADTH FIRST SEARCH AND DEPTH FIRST SEARCH**

### **AIM**

To Implement of Graph search algorithms (Breadth first search and Depth First Search) using Python.

### **BFS – Shortest Path in a Social Network Graph**

#### **ALGORITHM**

Step 1:

Start with an empty list visited to keep track of the nodes already visited.

Step 2:

Create an empty list queue which will be used to explore the graph level by level.

Step 3:

Insert the starting node into both visited and queue.

Step 4:

Repeat the following steps until the queue is empty.

Step 5:

Remove the front element from the queue and call it the current\_node.

Step 6:

For each neighbor of the current\_node, if it is not already in visited, then:

→ Add it to visited

→ Add it to the queue

Step 7:

Continue the process until the queue becomes empty. The visited list now contains nodes in BFS traversal order.

### **DFS**

#### **ALGORITHM**

Step 1:

Start with an empty set visited to keep track of the nodes already visited.

Step 2:

Initialize a stack and push the starting node onto it.

Step 3:

Repeat the following steps until the stack becomes empty.

Step 4:

Pop the top element from the stack and call it `current_node`.

Step 5:

If `current_node` is not in visited, then:

→ Print or process the node.

→ Add `current_node` to the visited set.

Step 6:

For each neighbor of `current_node` in the graph:

→ If the neighbor is not in visited, push it onto the stack.

Step 7:

Continue the process until the stack becomes empty. The nodes will be visited in DFS order.

## PROGRAM

### Graph Representation (Friend Network)

```
from collections import deque
```

```
# Breadth First Search using recursion and queue
```

```
def bfs_recursive(graph, queue, visited, target=None):
```

```
    if not queue:
```

```
        return
```

```
    node = queue.popleft()
```

```
    print(f'BFS visited: {node}')
```

```
    if node == target:
```

```
        print(f'Target {target} found in BFS!')
```

```
        return
```

```
    for neighbor in graph[node]:
```

```
        if neighbor not in visited:
```

```
            visited.add(neighbor)
```

```
            queue.append(neighbor)
```

```

    bfs_recursive(graph, queue, visited, target)

# Depth First Search using recursion
def dfs_recursive(graph, node, visited, target=None):
    if node not in visited:
        print(f"DFS visited: {node}")
        visited.add(node)
        if node == target:
            print(f"Target {target} found in DFS!")
            return
        for neighbor in graph[node]:
            dfs_recursive(graph, neighbor, visited, target)

# -----
# Graph data
friend_graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A'],
    'D': ['B'],
    'E': ['B']
}

start_node = 'A'
target_node = 'E'

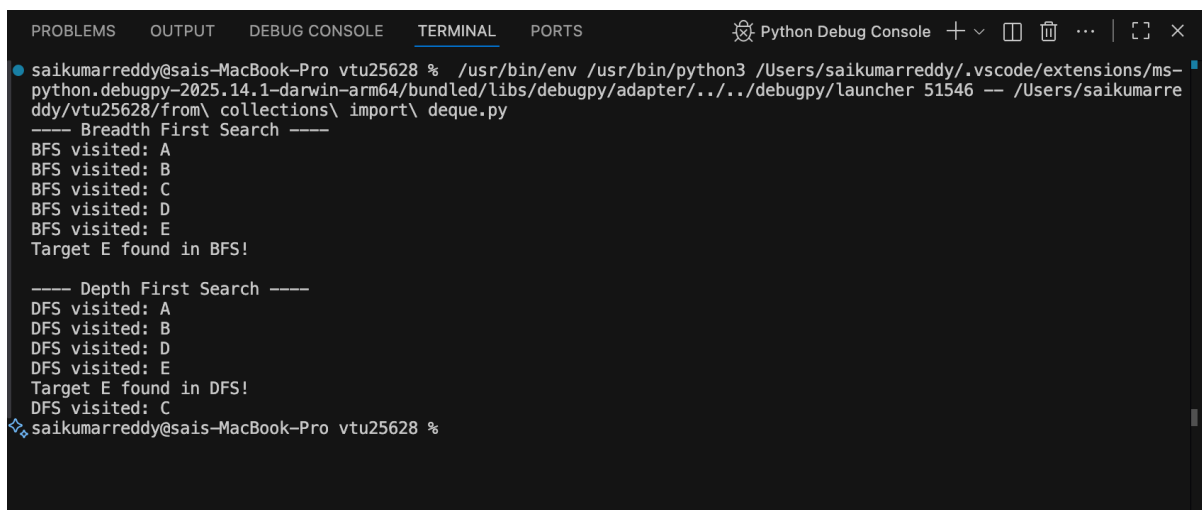
# Run BFS
print("---- Breadth First Search ----")
visited_bfs = set([start_node])
queue_bfs = deque([start_node])
bfs_recursive(friend_graph, queue_bfs, visited_bfs, target_node)

# Run DFS

```

```
print("\n---- Depth First Search ----")
visited_dfs = set()
dfs_recursive(friend_graph, start_node, visited_dfs, target_node)
```

## OUTPUT



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python Debug Console + - [ ] [x] ... | [ ] [x]
saikumarreddy@saish-MacBook-Pro vtu25628 % /usr/bin/env /usr/bin/python3 /Users/saikumarreddy/.vscode/extensions/ms-
python.debugpy-2025.14.1-darwin-arm64/bundled/libs/debugpy/adapter/../../debugpy/launcher 51546 -- /Users/saikumarre
ddy/vtu25628/from\ collections\ import\ deque.py
---- Breadth First Search ----
BFS visited: A
BFS visited: B
BFS visited: C
BFS visited: D
BFS visited: E
Target E found in BFS!

---- Depth First Search ----
DFS visited: A
DFS visited: B
DFS visited: D
DFS visited: E
Target E found in DFS!
DFS visited: C
saikumarreddy@saish-MacBook-Pro vtu25628 %
```

## **RESULT**

Thus the Implementation of Graph search algorithms (Breadth first search and Depth First Search) using Python was successfully executed and output was verified.