Experiment 4

Aim: Implementation of Depth first search and its depth limited version (IDDFS).

Objective: To study the uninformed searching techniques and its implementation for problem solving.

Theory:

Artificial Intelligence is the study of building agents that act rationally. Most of the time, theseagents perform some kind of search algorithm in the background in order to achieve their tasks. A search problem consists of:

- **A State Space.** Set of all possible states where you can be.
- **A Start State.** The state from where the search begins.
- **A Goal Test.** A function that looks at the current state returns whether or not itis the goal state.
- The **Solution** to a search problem is a sequence of actions, called the **plan** that transformsthe start state to the goal state.
- This plan is achieved through search algorithms.

Depth First Search: DFS is an uninformed search method. It is also called blind search. Uninformed search strategies use only the information available in the problem definition. A search strategy is defined by picking the order of node expansion. Depth First Search (DFS) searches deeper into the problem space. It is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, elseby backtracking.

The basic idea is as follows:
Pick a starting node and push all its adjacent nodes into a stack.
Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
Repeat this process until the stack is empty.
However, ensure that the nodes that are visited are marked. This will prevent you from visitingthe same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Algorithm:
A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles. TheDFS algorithm works as follows:

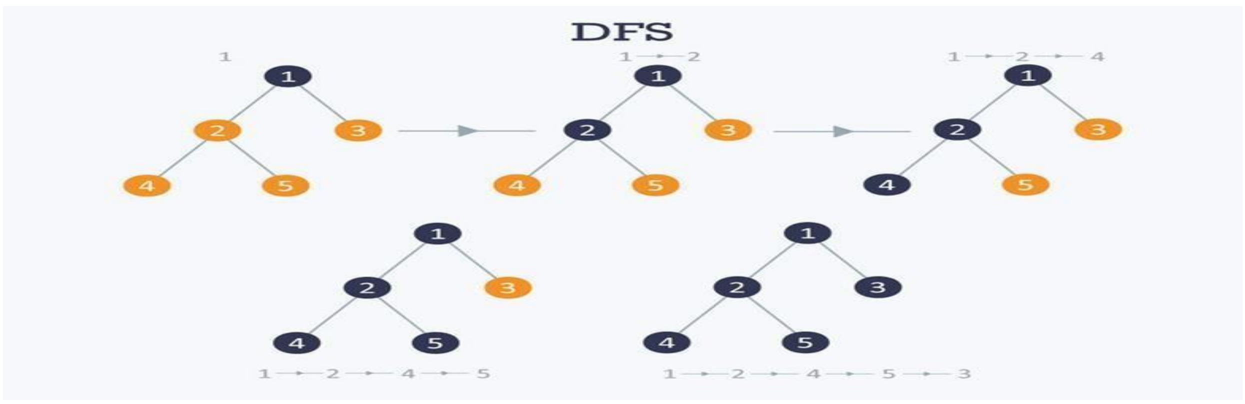
1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in thevisited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Pseudocode:

```
   DFS-iterative (G, s):     //Where G is graph and s is source vertex let S be stack
       S.push( s ) //Inserting s in stackmark s as visited.
       while ( S is not empty):
            //Pop a vertex from stack to visit next v =
   S.top( )
            S.pop( )
          //Push all the neighbours of v in stack that are notvisited for all neighbours
  w of v in Graph G:
                   if wis not visited
          : S.push( w )
                       mark w as visited


       DFS-
  recursive(G, s):
  mark s as visited
            for all neighbours w of s in
```

DFS Working: Examp



Path: 1 □ 2□ 4□ 5□ 3

Searching Strategies are evaluated along the following dimensions:

1. **Completeness:** does it always find a solution if one exists?
2. **Time complexity:** number of nodes generated
3. **Space complexity:** maximum number of nodes in memory
4. **Optimality:** does it always find a least-cost solution?

Properties of depth-first search:

1. Complete:- No: fails in infinite-depth spaces, spaces with loops.
2. Time Complexity: O(b^m)

3. Space Complexity: O(bm), i.e., linear space!

4. Optimal: No

Advantages of Depth-First Search:

1. Memory requirement is only linear with respect to the search graph.

2. The time complexity of a depth-first Search to depth d is O(b^d)

3. If depth-first search finds solution without exploring much in a path then the time andspace it takes will be very less.

Disadvantages of Depth-First Search:

1. There is a possibility that it may go down the left-most path forever. Even a finitegraph can generate an infinite tree.

2. Depth-First Search is not guaranteed to find the solution.

3. No guarantee to find a optimum solution, if more than one solution exists.

Applications:

How to find connected components using DFS?

A graph is said to be disconnected if it is not connected, i.e. if two nodes exist in the graphsuch that there is no edge in between those nodes. In an undirected graph, a connected component is a set of vertices in a graph that are linked to each other by paths.

Consider the example given in the diagram. Graph G is a disconnected graph and has the following 3 connected components.

Code:

```
def iterative_dfs(graph, start):
    stack = [start]
    visited = set()

    while stack:
        current = stack.pop()
        if current not in visited:
            visited.add(current)
            for neighbor in graph[current]:
                stack.append(neighbor)

    return visited

def recursive_dfs(graph, current, visited=set()):
    if current not in visited:
        visited.add(current)
        for neighbor in graph[current]:
            recursive_dfs(graph, neighbor, visited)
    return visited

graph = {'A': ['B', 'C'],
         'B': ['A', 'D', 'E'],
         'C': ['A', 'F'],
         'D': ['B'],
         'E': ['B', 'F'],
         'F': ['C', 'E']}

print(iterative_dfs(graph, 'A'))
print(recursive_dfs(graph, 'A'))
```

Output:

```
PS D:\Vartak college\SEM 5\AI EXP\New folder> py .\dfs.py
{'F', 'B', 'A', 'E', 'C', 'D'}
{'B', 'F', 'A', 'E', 'C', 'D'}
```

Conclusion:

Thus, we have learned to implement uninformed searching techniques. Depth-first search is an uninformed algorithm that uses recursion and backtracking to traverse nodes along entire branches using a stack before moving on, guaranteeing a solution if one exists.