Fr. Conceicao Rodrigues College of Engineering
Fr. Agnel Ashram, Bandstand, Bandra (W), Mumbai - 400050

Department of Computer Engineering
Academic Term II: 23-24

Class: B.E (Computer), Sem – VI **Subject Name: Artificial Intelligence**

Student Name: Siddhesh Pradhan **Roll No: 9632**

Practical No:	**3**
Title:	Use DFS problem solving method for a) Water Jug Problem b) Missionaries & Cannibals
Date of Performance:	12/02/2024
Date of Submission:	12/02/2024

Rubrics for Evaluation:

Sr. No	Performance Indicator	Excellent	Good	Below Average	Marks
1	On time Completion & Submission (01)	01 (On Time)	NA	00 (Not on Time)	
2	Logic/Algorithm Complexity analysis (03)	03(Correct)	02(Partial)	01 (Tried)	
3	Coding Standards (03): Comments/indention/Naming conventions Test Cases /Output	03(All used)	02 (Partial)	01 (rarely followed)	
4	Post Lab Assignment (03)	03(done well)	2 (Partially Correct)	1(submitted)	
Total					

Signature of the Teacher:

Experiment No: 3

Title: Use DFS problem solving method for

a) Water Jug Problem
b) Missionaries & Cannibals

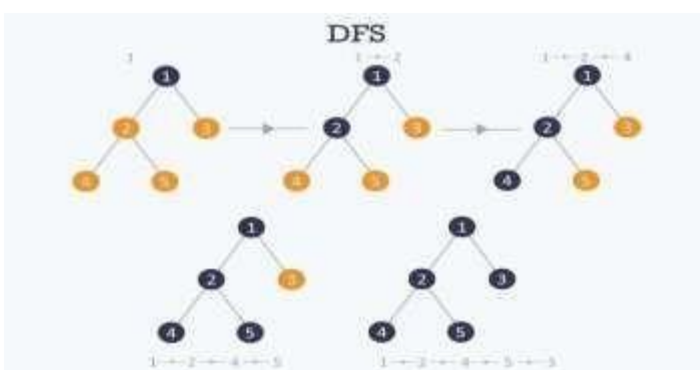
Objective: To write programs which solve the water jug problem and Missionaries & Cannibals problem in an efficient manner using Depth First Search.

Theory:

Depth-first search (DFS) is an algorithm for searching a graph or tree data structure. The algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it. The algorithm does this until the entire graph has been explored.

Depth first search is another way of traversing graphs, which is closely related to preorder traversal of a tree. Recall that preorder traversal simply visits each node before its children. It is most easy to program as a recursive routine:

```
preorder (node v)
{ visit(v); for each
child w of v
preorder(w);
}
```



a) WATER JUG PROBLEM

Given a 'm' liter jug and a 'n' liter jug, both the jugs are initially empty. The jugs don't have markings to allow measuring smaller quantities. You have to use the jugs to measure d liters of water where d is less than n.
(X, Y) corresponds to a state where X refers to amount of water in Jug1 and Y refers to amount of water in Jug2
Determine the path from initial state (xi, yi) to final state (xf, yf), where (xi, yi) is (0, 0) which indicates both Jugs are initially empty and (xf, yf) indicates a state which could be (0, d) or (d, 0).

The operations you can perform are:

1. Empty a Jug, (X, Y)-> (0, Y) Empty Jug 1
2. Fill a Jug, (0, 0)-> (X, 0) Fill Jug 1
3. Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

Just like we did for BFS, we can use DFS to classify the edges of G into types. Either an edge vw is in the DFS tree itself, v is an ancestor of w, or w is an ancestor of v. (These last two cases should be thought of as a single type, since they only differ by what order we look at the vertices in.) What this means is that if v and w are in different subtrees of v, we can't have an edge from v to w. This is because if such an edge existed and (say) v were visited first, then the only way we would avoid adding vw to the DFS tree would be if w were visited during one of the recursive calls from v, but then v would be an ancestor of w.

OUTPUT:-

expt3_WaterJugDFS.py

```
class State:
    def __init__(self, jug1, jug2):
        self.jug1 = jug1
        self.jug2 = jug2

    def __eq__(self, other):
        return self.jug1 == other.jug1 and self.jug2 == other.jug2

    def __hash__(self):
        return hash((self.jug1, self.jug2))


def dfs(current, visited, target, jug1_capacity, jug2_capacity):
    if current == target:
        return True

    visited.add(current)

    next_states = []

    # Fill jug1
    next_states.append(State(jug1_capacity, current.jug2))

    # Fill jug2
    next_states.append(State(current.jug1, jug2_capacity))

    # Empty jug1
    next_states.append(State(0, current.jug2))

    # Empty jug2
    next_states.append(State(current.jug1, 0))

    # Pour jug1 to jug2
    pour_amount = min(current.jug1, jug2_capacity - current.jug2)
    next_states.append(State(current.jug1 - pour_amount, current.jug2 + pour_amount))

    # Pour jug2 to jug1
```

```
        pour_amount = min(current.jug2, jug1_capacity - current.jug1)
        next_states.append(State(current.jug1 + pour_amount, current.jug2 - pour_amount))

    for next_state in next_states:
        if next_state not in visited:
            if dfs(next_state, visited, target, jug1_capacity, jug2_capacity):
                return True

    return False


def water_jug_problem():
    jug1_capacity = int(input("Enter the capacity of jug 1: "))
    jug2_capacity = int(input("Enter the capacity of jug 2: "))
    target_jug1 = int(input("Enter the desired amount of water in jug 1: "))
    target_jug2 = int(input("Enter the desired amount of water in jug 2: "))
    target = State(target_jug1, target_jug2)
    initial_state = State(0, 0)
    visited = set()
    return dfs(initial_state, visited, target, jug1_capacity, jug2_capacity)


if __name__ == "__main__":
    if water_jug_problem():
        print("Goal state is reachable.")
    else:
        print("Goal state is not reachable.")
```

```
PS C:\Local Disk D\6thSem\AI pracs> python3 expt3_WaterJugDFS.py
Enter the capacity of jug 1: 4
Enter the capacity of jug 2: 3
Enter the desired amount of water in jug 1: 2
Enter the desired amount of water in jug 2: 0
Goal state is reachable.
```

expt3_MisCanDFS.py

```
class State:
    def __init__(self, left_m, left_c, boat, right_m, right_c):
        self.left_m = left_m
        self.left_c = left_c
        self.boat = boat
        self.right_m = right_m
        self.right_c = right_c

    def __eq__(self, other):
        return (self.left_m, self.left_c, self.boat, self.right_m, self.right_c) == (
            other.left_m,
            other.left_c,
            other.boat,
            other.right_m,
            other.right_c,
        )

    def __hash__(self):
        return hash((self.left_m, self.left_c, self.boat, self.right_m,
self.right_c))


def is_valid(state):
    if state.left_m < 0 or state.left_c < 0 or state.right_m < 0 or state.right_c <
0:
        return False
    if state.left_m > 3 or state.left_c > 3 or state.right_m > 3 or state.right_c >
3:
        return False
    if (state.left_c > state.left_m > 0) or (state.right_c > state.right_m > 0):
        return False
    return True


def dfs(current, visited, target):
    if current == target:
        return True

    visited.add(current)
```

```
    next_states = []

    if current.boat == "left":
        # Send 1 missionary
        next_states.append(State(current.left_m - 1, current.left_c, "right", 
current.right_m + 1, current.right_c))
        # Send 1 cannibal
        next_states.append(State(current.left_m, current.left_c - 1, "right", 
current.right_m, current.right_c + 1))
        # Send 1 missionary and 1 cannibal
        next_states.append(State(current.left_m - 1, current.left_c - 1, "right", 
current.right_m + 1, current.right_c + 1))
        # Send 2 missionaries
        next_states.append(State(current.left_m - 2, current.left_c, "right", 
current.right_m + 2, current.right_c))
        # Send 2 cannibals
        next_states.append(State(current.left_m, current.left_c - 2, "right", 
current.right_m, current.right_c + 2))
    else:
        # Bring 1 missionary back
        next_states.append(State(current.left_m + 1, current.left_c, "left", 
current.right_m - 1, current.right_c))
        # Bring 1 cannibal back
        next_states.append(State(current.left_m, current.left_c + 1, "left", 
current.right_m, current.right_c - 1))
        # Bring 1 missionary and 1 cannibal back
        next_states.append(State(current.left_m + 1, current.left_c + 1, "left", 
current.right_m - 1, current.right_c - 1))
        # Bring 2 missionaries back
        next_states.append(State(current.left_m + 2, current.left_c, "left", 
current.right_m - 2, current.right_c))
        # Bring 2 cannibals back
        next_states.append(State(current.left_m, current.left_c + 2, "left", 
current.right_m, current.right_c - 2))

    for next_state in next_states:
        if is_valid(next_state) and next_state not in visited:
            if dfs(next_state, visited, target):
                return True
```

```
        return False


def missionaries_cannibals_problem():
    left_m = int(input("Enter the number of missionaries on the left bank: "))
    left_c = int(input("Enter the number of cannibals on the left bank: "))
    boat = "left"
    right_m = 0
    right_c = 0
    initial_state = State(left_m, left_c, boat, right_m, right_c)

    right_m = int(input("Enter the number of missionaries on the right bank: "))
    right_c = int(input("Enter the number of cannibals on the right bank: "))
    boat = "right"
    target_state = State(0, 0, boat, right_m, right_c)

    visited = set()
    return dfs(initial_state, visited, target_state)


if __name__ == "__main__":
    if missionaries_cannibals_problem():
        print("Solution found.")
    else:
        print("No solution found.")
```

```
PS C:\Local Disk D\6thSem\AI pracs> python3 expt3_MisCanDFS.py
Enter the number of missionaries on the left bank: 3
Enter the number of cannibals on the left bank: 3
Enter the number of missionaries on the right bank: 3
Enter the number of cannibals on the right bank: 3
Solution found.
```

Post Lab Assignment:

1. What is the time complexity of the Water Jug problem?

Ans:- The time complexity of the Water Jug problem using a Depth-First Search (DFS) approach is O(J×K), where J and K are the capacities of the two jugs, respectively. This is because each state can have up to O(J×K) successor states, and the DFS explores these states until a solution is found.

2. Why is DFS not used for solving a water jug problem?

Ans:- DFS is not typically used to solve the Water Jug problem because it may lead to an exponential number of states being explored, especially in the worst case where no solution exists. Additionally, DFS does not guarantee finding the shortest path to the solution, which is often desired in optimization problems like the Water Jug problem