Department of Computer Engineering

Academic Term II: 23-24

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

Student Name: Jatin Jaywant Kadu **Roll No: 9548**

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

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(submitte d)	
Total					

Signature of the Teacher:

Experiment No: 3

Title: Use DFS problem solving method for

a) Water Jug Problem
b) Missionaries & Cannibals

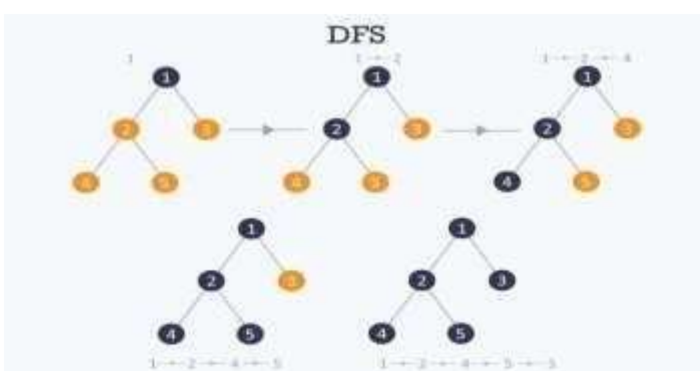
Objective: To write programs which solve the water jug problem and Missionaries & Cannibals problem in an efficient manner using Depth First Search.

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.

Post Lab Assignment:

1. What is the time complexity of the Water Jug problem?
2. Why is DFS not used for solving a water jug problem?

Code:

```
def pour_water(state, action):
    x, y = state
    if action == 'fill_4':
        return (4, y)
    elif action == 'fill_3':
        return (x, 3)
    elif action == 'empty_4':
        return (0, y)
    elif action == 'empty_3':
        return (x, 0)
    elif action == 'pour_4_to_3':
        amount = min(x, 3 - y)
        return (x - amount, y + amount)
    elif action == 'pour_3_to_4':
        amount = min(y, 4 - x)
        return (x + amount, y - amount)
    else:
        return state

def dfs(state, visited):
    if state[0] == 2:
        return [state]
    visited.add(state)
    for action in ['fill_4', 'fill_3', 'empty_4', 'empty_3', 'pour_4_to_3', 'pour_3_to_4']:
        new_state = pour_water(state, action)
        if new_state not in visited:
            path = dfs(new_state, visited)
            if path:
                return [state] + path
    return None

def print_steps(path):
    for i, state in enumerate(path):
        jug_4, jug_3 = state
        if i == 0:
            print(f"Initial State: {jug_4} | {jug_3}")
        else:
            prev_jug_4, prev_jug_3 = path[i - 1]
            if jug_4 > prev_jug_4:
```

```python
            print(f"Fill Jug1: {jug_4} | {jug_3}")
        elif jug_3 > prev_jug_3:
            print(f"Fill Jug2: {jug_4} | {jug_3}")
        elif jug_4 < prev_jug_4:
            print(f"Empty Jug1: {jug_4} | {jug_3}")
        elif jug_3 < prev_jug_3:
            print(f"Empty Jug2: {jug_4} | {jug_3}")
        elif jug_4 != prev_jug_4 and jug_3 != prev_jug_3:
            if jug_4 == 0:
                print(f"Pour Jug2 to Jug1: {jug_4} | {jug_3}")
            elif jug_3 == 0:
                print(f"Pour Jug1 to Jug2: {jug_4} | {jug_3}")

initial_state = (0, 0)
visited = set()
path = dfs(initial_state, visited)

if path:
    print("Steps to measure 2 gallons:")
    print_steps(path)
else:
    print("No solution found.")
```

Output:

```
thon.exe "c:/Users/hacke/OneDrive/Desktop/SEM VI/AI/dfs_waterjug.py"
Steps to measure 2 gallons:
Initial State: 0 | 0
Fill Jug1: 4 | 0
Fill Jug2: 4 | 3
Empty Jug1: 0 | 3
Fill Jug1: 3 | 0
Fill Jug2: 3 | 3
Fill Jug1: 4 | 2
Empty Jug1: 0 | 2
Fill Jug1: 2 | 0
```

Code:

```
class State:
    def __init__(self, missionaries, cannibals, boat_position):
        self.missionaries = missionaries
        self.cannibals = cannibals
        self.boat_position = boat_position

    def is_valid(self):
        if (
            0 <= self.missionaries <= 3
            and 0 <= self.cannibals <= 3
            and 0 <= self.boat_position <= 1
        ):
            if (
                self.missionaries == 0
                or self.missionaries == 3
                or self.missionaries >= self.cannibals
            ):
                return True
        return False

    def is_goal(self):
        return self.missionaries == 0 and self.cannibals == 0 and self.boat_position == 0

    def __eq__(self, other):
        return (
            self.missionaries == other.missionaries
            and self.cannibals == other.cannibals
            and self.boat_position == other.boat_position
        )

    def __hash__(self):
        return hash((self.missionaries, self.cannibals, self.boat_position))


def generate_next_states(current_state):
    next_states = []
    moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]

    for m, c in moves:
        if current_state.boat_position == 1:
```

```python
            new_state = State(
                current_state.missionaries - m,
                current_state.cannibals - c,
                0,
            )
        else:
            new_state = State(
                current_state.missionaries + m,
                current_state.cannibals + c,
                1,
            )

        if new_state.is_valid():
            next_states.append(new_state)

    return next_states


def dfs_search():
    start_state = State(3, 3, 1)
    goal_state = State(0, 0, 0)

    stack = [(start_state, [])]
    visited = set()

    while stack:
        current_state, path = stack.pop()

        if current_state.is_goal():
            return path

        if current_state not in visited:
            visited.add(current_state)

            next_states = generate_next_states(current_state)
            for next_state in next_states:
                if next_state not in visited:
                    stack.append((next_state, path + [current_state]))

    return None
```

```python
def print_state_description(state):
    left_shore = f"{state.missionaries} Missionaries and {state.cannibals} Cannibals on the Left Shore"
    right_shore = f"{3 - state.missionaries} Missionaries and {3 - state.cannibals} Cannibals on the Right
Shore"

    print(f"{left_shore}, {right_shore}\n")



if __name__ == "__main__":
    solution_path = dfs_search()

    if solution_path:
        print("Solution Path:")
        for i, state in enumerate(solution_path):
            print(f"Step {i + 1}:")
            print_state_description(state)

    else:
        print("No solution found.")
```

Output:

```
PS C:\Users\hacke\OneDrive\Desktop\SEM VI\AI> & C:/Users/hacke/AppData/Local/Programs/Python/Python311/py
thon.exe "c:/Users/hacke/OneDrive/Desktop/SEM VI/AI/dfs_waterjug.py"
Solution Path:
Step 1:
3 Missionaries and 3 Cannibals on the Left Shore, 0 Missionaries and 0 Cannibals on the Right Shore

Step 2:
2 Missionaries and 2 Cannibals on the Left Shore, 1 Missionaries and 1 Cannibals on the Right Shore

Step 3:
3 Missionaries and 2 Cannibals on the Left Shore, 0 Missionaries and 1 Cannibals on the Right Shore

Step 4:
2 Missionaries and 1 Cannibals on the Left Shore, 1 Missionaries and 2 Cannibals on the Right Shore

Step 5:
2 Missionaries and 2 Cannibals on the Left Shore, 1 Missionaries and 1 Cannibals on the Right Shore

Step 6:
1 Missionaries and 1 Cannibals on the Left Shore, 2 Missionaries and 2 Cannibals on the Right Shore

Step 7:
3 Missionaries and 1 Cannibals on the Left Shore, 0 Missionaries and 2 Cannibals on the Right Shore

Step 8:
2 Missionaries and 0 Cannibals on the Left Shore, 1 Missionaries and 3 Cannibals on the Right Shore

Step 9:
2 Missionaries and 1 Cannibals on the Left Shore, 1 Missionaries and 2 Cannibals on the Right Shore
```

```
Step 10:
1 Missionaries and 0 Cannibals on the Left Shore, 2 Missionaries and 3 Cannibals on the Right Shore

Step 11:
1 Missionaries and 1 Cannibals on the Left Shore, 2 Missionaries and 2 Cannibals on the Right Shore
```