AI LAB 3 – 9763-Harsh Parmar– Batch D

Water Jug problem using DFS :

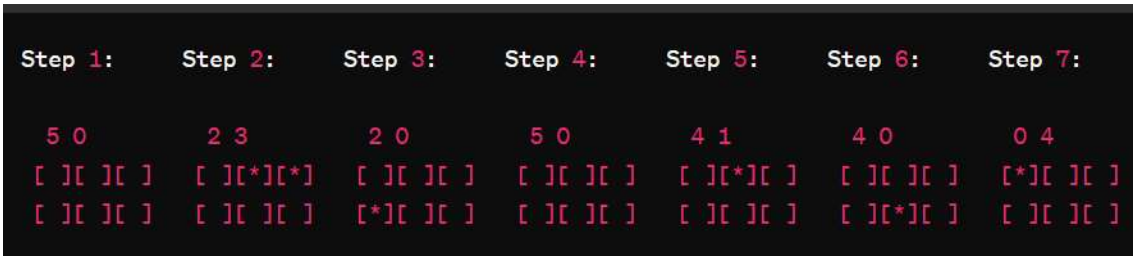
Code:

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

    def __eq__(self, other):
        return self.jugs == other.jugs

    def __hash__(self):
        return hash(tuple(self.jugs))

def successors(state, jug_sizes):
    successors = []
    for i in range(len(state.jugs)):
        for j in range(len(state.jugs)):
            if i != j:
                pour_amount = min(state.jugs[i], jug_sizes[j] - state.jugs[j])
                if pour_amount > 0:
                    new_jugs = list(state.jugs)
                    new_jugs[i] -= pour_amount
                    new_jugs[j] += pour_amount
                    successors.append(State(tuple(new_jugs)))
```

```
    return successors

def dfs(state, goal, jug_sizes, visited):
    if state == goal:
        return []
    visited.add(state)
    for succ_state in successors(state, jug_sizes):
        if succ_state not in visited:
            result = dfs(succ_state, goal, jug_sizes, visited)
            if result is not None:
                pour_amount = [state.jugs[i] - succ_state.jugs[i] for i in
range(len(state.jugs))]
                return [f"Pour {pour_amount[i]} from jug {i+1} to jug {j+1}" for i, j in
enumerate(range(len(state.jugs)))]

    return None

def main():
    jug_sizes = (5, 3)  # Jug sizes (e.g., (5, 3) represents jugs of size 5 and 3)
    initial_state = State((0, 0))  # Initial state of the jugs
    goal_state = State((4, 0))  # Goal state to reach

    visited = set()
    solution = dfs(initial_state, goal_state, jug_sizes, visited)
    if solution:
        print("Solution:", solution)
    else:
```

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

if__name__== "__main__":
    main()
```

OUTPUT:



Missionaries and Cannibals :

Code:

```
# Missionaries & Cannibals Problem using Depth-First Search

class State:
    def__init__(self, missionaries, cannibals, boat):
        self.missionaries = missionaries
        self.cannibals = cannibals
        self.boat = boat
        self.parent = None
        self.action = None

    def__eq__(self, other):
```

```
        return self.missionaries == other.missionaries and self.cannibals ==
other.cannibals and self.boat == other.boat


    def__hash__(self):
        return hash((self.missionaries, self.cannibals, self.boat))


def successors(state):
    successors = []
    if state.boat == 'left':
        for m in range(3):
            for c in range(3):
                if 1 <= m + c <= 2:
                    new_state = State(state.missionaries - m, state.cannibals - c, 'right')
                    if 0 <= new_state.missionaries <= 3 and 0 <= new_state.cannibals <=
3 and (new_state.missionaries >= new_state.cannibals or
new_state.missionaries == 0) and ((3 - new_state.missionaries) >= (3 -
new_state.cannibals) or new_state.missionaries == 3):
                        successors.append((new_state, f"Move {m} missionaries and {c}
cannibals to the right"))
    else:
        for m in range(3):
            for c in range(3):
                if 1 <= m + c <= 2:
                    new_state = State(state.missionaries + m, state.cannibals + c, 'left')
                    if 0 <= new_state.missionaries <= 3 and 0 <= new_state.cannibals <=
3 and (new_state.missionaries >= new_state.cannibals or
new_state.missionaries == 0) and ((3 - new_state.missionaries) >= (3 -
new_state.cannibals) or new_state.missionaries == 3):
```

```
                    successors.append((new_state, f"Move {m} missionaries and {c}
cannibals to the left"))
    return successors

def dfs(initial_state, goal):
    stack = [(initial_state, "Initial state")]
    visited = set()

    while stack:
        state, action = stack.pop()
        if state == goal:
            return action
        if state not in visited:
            visited.add(state)
            stack.extend((s, a) for s, a in successors(state))

    return "No solution found."

def main():
    initial_state = State(3, 3, 'left')  # Initial state of the missionaries and
cannibals
    goal_state = State(0, 0, 'right')  # Goal state to reach

    solution = dfs(initial_state, goal_state)
    print("Solution:", solution)

if__name__== "__main__":
```

main()

POSTLAB:

FR. CONCEICAO RODRIGUES COLLEGE OF ENGINEERING

AI Lab-3 Hausch Parmar 9763

The time complexity of Water Jug Problem depends on the algorithm used to solve it. Generally when solving the Water Jug Problem using a brute force approach, the time complexity can be exponential in worst case. This is because the search space grown expoentially with the number of steps required to reach the goal state. Therefore the time complexity can be expressed as $O(b^d)$ b is braching factor & d is the depth of a search tree.

DFS is not usually used for solving the Water Jug Problem because it does not guarantee finding the shortest Path to the goal state. DFS explores the search space depth first meaning it explores one branch of the search tree has far as possible before backtracking. While DFS may find a solution for the Water Jug Problem, it may not find the optimal solution. Addition DFS may get stuck in infinite loops if the search space contains cycles. Therefore BFS or A* are used for Water Jug Problem.