SUPERIOR UNIVERSITY

Name : Huzaifa Rehan

Roll No : SU92-BSAIM-F23-071

Section : AI(4-B)

Lab Task : 03

Submitted to : Sir Rasikh Ali

Lab Task -3 (Water-Jug Problem)

Water Jug Problem Report

Introduction

The Water Jug problem is a classical problem in artificial intelligence and algorithm design. It involves two jugs of different capacities and aims to measure an exact quantity of water using only these two jugs. The problem is typically solved using search algorithms, making it a fundamental example of problem-solving techniques in AI.

Problem Definition

Given two jugs with fixed capacities, say X liters and Y liters, and an unlimited water supply, the objective is to measure exactly Z liters using the following operations:

1. **Fill a jug** completely from the water source.
2. **Empty a jug** completely.
3. **Pour water** from one jug to another until either the first jug is empty or the second jug is full.
4. **Check if the required amount (Z) can be measured.**

Applications of the Water Jug Problem

- **Artificial Intelligence:** Used in AI search algorithms and problem-solving strategies.
- **Robotics:** Helps in planning and constraint-based problem-solving.
- **Computer Science:** Useful in algorithm design, especially for state-space search problems.
- **Mathematics:** Forms the basis for problems involving Diophantine equations and number theory.

Solution Approaches

Several approaches exist to solve the Water Jug problem efficiently:

1. Breadth-First Search (BFS)

- The problem can be represented as a graph where each state represents a unique combination of water levels in both jugs.

- BFS explores all possible moves level by level until the desired state is reached.
- Guarantees the shortest sequence of steps to reach the goal.
- Time Complexity: O(X * Y) (since there are at most X * Y possible states).

2. Depth-First Search (BFS)

- Similar to BFS but explores one branch of the state space deeply before backtracking.
- May not always find the shortest path.
- Suitable for cases where space constraints exist.
- Time Complexity: O(X * Y) in the worst case.

3. Using Mathematical Approach (Diophantine Equations)

- The problem can be solved if and only if Z is a multiple of the greatest common divisor (GCD) of X and Y.
- Uses the equation: ax + by = Z, where a and b are integers, and gcd(X, Y) divides Z.
- Efficient for determining whether a solution exists.

4. A Search Algorithm*

- Uses heuristics to find the optimal solution efficiently.
- Combines BFS with cost estimation to prioritize moves that lead to the goal faster.
- Best suited for complex variations of the problem.

Complexity Analysis

- **BFS and DFS:** O(X * Y) in worst cases.
- **Mathematical Approach:** O(log(min(X, Y))) due to GCD computation.
- *A Algorithm:** Optimized but depends on the heuristic used.

Challenges in the Water Jug Problem

1. **Unsolvable Cases:** If Z is not a multiple of gcd(X, Y), no solution exists.
2. **State Explosion:** Large jug capacities lead to a large search space, increasing computational complexity.

3. **Memory Constraints:** BFS may require significant memory when dealing with large search spaces.

Conclusion

The Water Jug problem is a fundamental example of state-space search problems in artificial intelligence and computer science. While brute-force approaches like DFS and BFS provide solutions, optimized methods such as the mathematical approach and A* search make the problem more efficient. Understanding this problem helps in mastering problem-solving techniques applicable to a wide range of real-world scenarios.

Code:

```
class waterjug:
    def __init__(self, jug1, jug2, target):
        self.jug1 = jug1
        self.jug2 = jug2
        self.target = target
        self.visited = set()

    def valid_state(self, state):
        return state not in self.visited

    def dfs(self, state, path):
        if state in self.visited:
            return False

        jug1, jug2 = state
        print(f"Current State: {state}")
        self.visited.add(state)
        path.append(state)

        if jug1 == self.target or jug2 == self.target:
            print("reached")
            return True
```

```
        next_states = [
            (self.jug1, jug2),
            (jug1, self.jug2),
            (0, jug2),
            (jug1, 0),
            (jug1 - min(jug1, self.jug2 - jug2), jug2 + min(jug1, self.jug2 - jug2)),
            (jug1 + min(jug2, self.jug1 - jug1), jug2 - min(jug2, self.jug1 - jug1))
        ]

        for next_state in next_states:
            if self.valid_state(next_state) and self.dfs(next_state, path):
                return True

        path.pop()
        return False
```

```
    def solve(self):
        print("Rules:")
        print("1. Fill Jug 1 completely.")
        print("2. Fill Jug 2 completely.")
        print("3. Empty Jug 1.")
        print("4. Empty Jug 2.")
        print("5. Pour water from Jug 1 to Jug 2 until Jug 2 is full or Jug 1 is empty.")
        print("6. Pour water from Jug 2 to Jug 1 until Jug 1 is full or Jug 2 is empty.")

        path = []
        if self.dfs((0, 0), path):
            print("Solution Path:")
            for step in path:
                print(step)
        else:
            print("No solution found.")

if __name__ == "__main__":
    jug1_capacity = 4
    jug2_capacity = 3
    target_amount = 2

    puzzle = waterjug(jug1_capacity, jug2_capacity, target_amount)
    puzzle.solve()

```

Output

```
Rules:
1. Fill Jug 1 completely.
2. Fill Jug 2 completely.
3. Empty Jug 1.
4. Empty Jug 2.
5. Pour water from Jug 1 to Jug 2 until Jug 2 is full or Jug 1 is empty.
6. Pour water from Jug 2 to Jug 1 until Jug 1 is full or Jug 2 is empty.
Current State: (0, 0)
Current State: (4, 0)
Current State: (4, 3)
Current State: (0, 3)
Current State: (3, 0)
Current State: (3, 3)
Current State: (4, 2)
reached
Solution Path:
(0, 0)
(4, 0)
(4, 3)
(0, 3)
(3, 0)
(3, 3)
(4, 2)
```