SUPERIOR UNIVERSITY

NAME: MINAHIL IQBAL

ROLL NO: 062

SECTION: BSAI-4B

SUBJECT: PAI (LAB)

SUBMITTED TO: SIR RASIKH

Report on Water Jug Problem Implementation

Introduction

The Water Jug Problem is a classic problem in artificial intelligence and algorithm design, commonly solved using search techniques such as depth-first search (DFS) or breadth-first search (BFS). This report analyzes the given Python program that implements a DFS-based approach to solving the Water Jug Problem.

Problem Definition

The Water Jug Problem involves two jugs with fixed capacities and an unlimited water source. The goal is to measure a specific target amount of water using the two jugs. The permissible operations are:

1. Fill either jug to its full capacity.
2. Empty either jug completely.
3. Transfer water from one jug to the other until one is full or the other is empty.

Program Overview

The program is designed to explore different possible states of the two jugs using a depth-first search approach. The key components of the program are:

1. Data Structures

1. stack: A list used as a stack to store states during exploration.
2. visited: A set to keep track of visited states to prevent redundant calculations.
3. actions: A list to store the sequence of steps leading to the solution.

2. State Representation

- Each state in the problem is represented as a tuple (jug1, jug2), where:
- jug1 represents the amount of water in the first jug.
- jug2 represents the amount of water in the second jug.

3. Solution Approach

The program starts with both jugs empty (0,0). It then explores possible actions and transitions through different states using the following rules:

- Fill Jug 1 to capacity.
- Fill Jug 2 to capacity.
- Empty Jug 1.
- Empty Jug 2.
- Transfer water from Jug 1 to Jug 2.
- Transfer water from Jug 2 to Jug 1.
- Pour all water from Jug 1 into Jug 2 if it fits within Jug 2's capacity.
- Pour all water from Jug 2 into Jug 1 if it fits within Jug 1's capacity.

The algorithm iterates through these possible states, checking if either jug reaches the target capacity. If a solution is found, it prints the sequence of actions taken.

4. Termination Conditions

If a state where either jug1 == goal or jug2 == goal is found, the program prints the solution sequence and terminates.

If all possible states have been visited and no solution is found, the program prints "No Solution Found" and terminates.

Example Execution

For an input of jug1Capacity = 7, jug2Capacity = 5, and target = 2, the program searches for a sequence of actions leading to exactly 2 liters of water in either jug. The output consists of the sequence of jug states leading to the solution or an indication that no solution exists.

Strengths of the Implementation

- Depth-First Search (DFS) Approach: Uses an iterative DFS strategy to explore possible states efficiently.
- State Tracking: Prevents infinite loops by storing visited states.
- Comprehensive State Transition Rules: Covers all possible operations.

Limitations

- DFS May Not Always Find the Shortest Path: The solution found might not be the most optimal (minimum steps required).
- Stack-Based Implementation: Can lead to high memory usage in case of large search spaces.
- No Backtracking Optimization: The algorithm does not backtrack to find alternative shorter paths.

Possible Improvements

- Use BFS Instead of DFS: A breadth-first search would guarantee the shortest sequence of actions.
- Optimize State Transition Rules: Some redundant checks could be eliminated for efficiency.
- Graph Representation: Representing the problem as a graph and using standard shortest-path algorithms like Dijkstra's algorithm could improve performance.

Conclusion

This implementation provides a functional approach to solving the Water Jug Problem using DFS. It efficiently explores possible states while preventing unnecessary revisits, ensuring a correct solution if one exists. However, using BFS or heuristic-based search could optimize the solution further.

OUTPUT

```
Solution Found
(0, 0)
(0, 5)
(5, 0)
(5, 5)
(7, 3)
(0, 3)
(3, 0)
(3, 5)
(7, 1)
(0, 1)
(1, 0)
(1, 5)
(6, 0)
(6, 5)
(7, 4)
(0, 4)
(4, 0)
(4, 5)
(7, 2)
: True
```