

**Name: Hafiz Haider Ali**

**Roll No: BSAI-130-4C**

**Date: 27 Feb 2025**

**Subject: PAI LAB**

#### NWater Jug Problem Solver

This Python program solves the classic Water Jug Problem, where you have two jugs with given capacities (e.g., 4 liters and 3 liters) and need to measure a specific amount of water (e.g., 2 liters) using operations like filling, emptying, and pouring water between them. The program uses Breadth-First Search (BFS) to find the shortest sequence of steps to achieve the goal. Below is a detailed explanation of the code.

#### 1. Importing Necessary Libraries

* **Library Used:**
  1. from collections import deque: Imports the deque class, which implements a double-ended queue.
* **How It Works:** deque is used as a queue to store states of the jugs during BFS.
* **Why Used:** BFS requires processing states in the order they are discovered (first-in, first-out), and deque provides efficient append and pop operations from the left (front) of the queue.

#### 2. Defining the Solver Function: water\_jug\_solver(jug1, jug2, goal)

* **Parameters:**
  + jug1: Capacity of the first jug.
  + jug2: Capacity of the second jug.
  + goal: Target amount of water to measure.
* **How It Works:** This function uses BFS to explore all possible states of water in the jugs until the goal is reached or no solution is found.
* **Why Used:** The function encapsulates the logic to systematically solve the problem, making it reusable and modular.

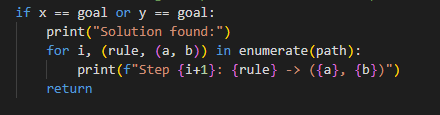
#### 3. Initializing the BFS

* **Code:**
  + queue = deque([(0, 0, [])]): Starts with both jugs empty (0, 0) and an empty list [] for tracking steps.
  + visited = set(): An empty set to track visited states (jug water combinations).
* **How It Works:**
  + The queue holds tuples of (x, y, path), where x is water in jug1, y is water in jug2, and path is the list of steps taken.
  + visited prevents revisiting the same state, avoiding infinite loops.
* **Why Used:** BFS needs an initial state and a way to avoid cycles; this setup ensures we start from empty jugs and explore efficiently.

#### 4. BFS Loop

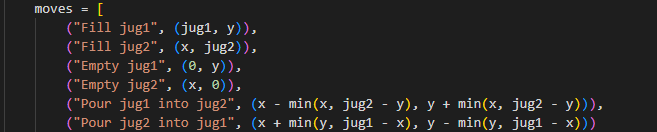
* **Code:** while queue:
* **How It Works:** Continues until the queue is empty (no solution) or a solution is found. Each iteration processes the next state from the queue using queue.popleft().
* **Why Used:** BFS explores states level-by-level, ensuring the shortest path to the goal is found first.

#### Checking for the Goal



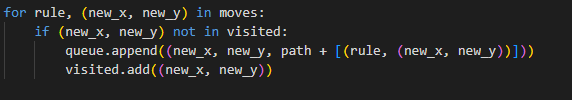
* **How It Works:** If the amount in either jug (x or y) equals the goal, it prints the sequence of steps from path and exits.
* **Why Used:** This is the termination condition; once the goal is reached, we want to show the solution and stop searching.

#### 6. Defining Possible Moves



* **How It Works:** Lists six possible actions with their resulting states:
  1. Fill jug1 to its capacity.
  2. Fill jug2 to its capacity.
  3. Empty jug1.
  4. Empty jug2.
  5. Pour from jug1 to jug2 (limited by jug2’s remaining capacity).
  6. Pour from jug2 to jug1 (limited by jug1’s remaining capacity).
  + min() ensures pouring doesn’t exceed available water or jug capacity.
* **Why Used:** These are the standard operations allowed in the water jug problem, defining the state space to explore.

#### 7. Exploring New States

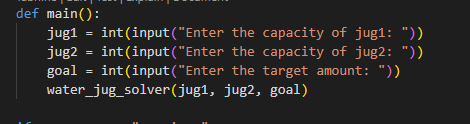


* **How It Works:** For each move, if the new state (new\_x, new\_y) hasn’t been visited, it’s added to the queue with the updated path, and marked as visited.
* **Why Used:** This ensures all possible moves are explored without redundancy, a key part of BFS.

#### 8. Handling No Solution

* **Code:** print("No solution found.")
* **How It Works:** If the queue empties without finding the goal, it indicates no solution exists.
* **Why Used:** Not all combinations of jug capacities and goals are solvable (e.g., goal must be a multiple of the GCD of jug capacities), so this handles impossible cases.

#### 9. User Interaction: main()



* **How It Works:** Prompts the user for jug capacities and the target, then calls the solver.
* **Why Used:** Makes the program interactive, allowing users to input custom problem instances.

#### 10. Program Entry Point

* **Code:** if \_\_name\_\_ == "\_\_main\_\_": main()
* **How It Works:** Runs main() when the script is executed directly.
* **Why Used:** Standard Python practice to ensure the code runs only if intended, not when imported as a module.

### How and Why We Use These Functions

1. deque:
   * **How:** Implements the BFS queue with append and popleft.
   * **Why:** Provides O(1) time complexity for adding and removing elements from the front, ideal for BFS.
2. set (for visited):
   * **How:** Stores visited states as tuples (x, y).
   * **Why:** Offers O(1) lookup time to check if a state has been seen, preventing cycles.
3. min():
   * **How:** Limits water poured to the smaller of available water or remaining capacity.
   * **Why:** Ensures realistic pouring operations within jug constraints.
4. enumerate:
   * **How:** Adds step numbers to the solution output.
   * **Why:** Improves readability by numbering each step in the solution path.
5. water\_jug\_solver:
   * **How:** Uses BFS to explore states systematically.
   * **Why:** BFS guarantees the shortest solution, which is desirable for this problem.