Water Jug Problem

Problem:

Given two jugs with fixed capacities and a target amount of water, determine whether it is possible to measure exactly the target amount of water using the two jugs. You can perform a series of actions like filling, emptying, or pouring water from one jug to the other.

Description:

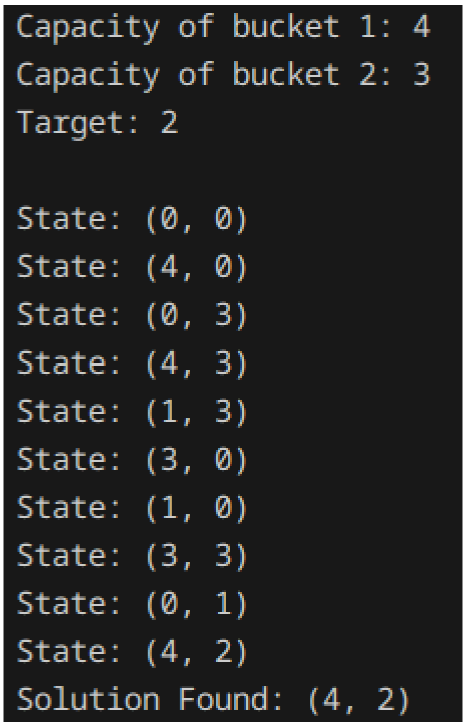
This code solves the Water Jug Problem using a **Breadth-First Search (BFS)** approach. The state of the two jugs is represented as a pair (jug1, jug2) where jug1 is the amount of water in the first jug, and jug2 is the amount of water in the second jug. The BFS explores all possible states of the jugs and tries to reach the target amount of water in either of the jugs.

The program uses a queue to manage the states, with each state representing a possible configuration of the two jugs. The BFS approach ensures that the solution is found in the least number of steps. If the target is found in either jug, the program prints the solution; otherwise, it prints "No solution found."

Code:

```
#include <iostream>
#include <queue>
#include <set>
#include <tuple>

using namespace std;

struct State
{
   int jug1, jug2;

   bool operator<(const State &other) const
   {
       return tie(jug1, jug2) < tie(other.jug1, other.jug2);
   }
};

void solveWaterJug(int capacity1, int capacity2, int target)
{
   set<State> visited;
   queue<State> q;

   q.push({0, 0});
   visited.insert({0, 0});

   while (!q.empty())
   {
       State current = q.front();
       q.pop();

       int jug1 = current.jug1, jug2 = current.jug2;

       // Just print the volume state of the jugs
       cout << "State: (" << jug1 << ", " << jug2 << ")" << endl;

       if (jug1 == target || jug2 == target)
       {
           cout << "Solution Found: (" << jug1 << ", " << jug2 << ")" << endl;
           return;
       }

       // Generate next possible states
       vector<State> nextStates = {
           {capacity1, jug2},                                          // Fill Jug1
           {jug1, capacity2},                                          // Fill Jug2
           {0, jug2},                                                  // Empty Jug1
           {jug1, 0},                                                  // Empty Jug2
           {min(jug1 + jug2, capacity1), max(0, jug1 + jug2 - capacity1)}, // Pour Jug2 into Jug1
           {max(0, jug1 + jug2 - capacity2), min(jug1 + jug2, capacity2)}  // Pour Jug1 into Jug2
       };

       // Push the next states into the queue if they haven't been visited
       for (const State &nextState : nextStates)
       {
           if (visited.find(nextState) == visited.end())
           {
               visited.insert(nextState);
               q.push(nextState);
           }
       }
   }

   cout << "No solution found." << endl;
}

int main()
{
   int capacity1 = 4, capacity2 = 3, target = 2;
   cout << "Capacity of bucket 1: " << capacity1 << endl
        << "Capacity of bucket 2: " << capacity2 << endl
        << "Target: " << target << endl
        << endl;
   solveWaterJug(capacity1, capacity2, target);
   return 0;
}
```

Output:



```
Capacity of bucket 1: 4
Capacity of bucket 2: 3
Target: 2

State: (0, 0)
State: (4, 0)
State: (0, 3)
State: (4, 3)
State: (1, 3)
State: (3, 0)
State: (1, 0)
State: (3, 3)
State: (0, 1)
State: (4, 2)
Solution Found: (4, 2)
```

Approach:

1. **State Representation**: Each state is represented by the amount of water in both jugs (jug1, jug2).
2. **Breadth-First Search (BFS)**: BFS is used to explore all possible states starting from (0, 0). States are explored by performing the following actions:
    - Fill Jug 1
    - Fill Jug 2
    - Empty Jug 1
    - Empty Jug 2
    - Pour water from Jug 2 into Jug 1
    - Pour water from Jug 1 into Jug 2
3. **State Transition**: For each state, all possible transitions to new states are calculated, and if a new state has not been visited, it is added to the queue for further exploration.
4. **Target Condition**: If either jug reaches the target amount of water, the solution is found and printed.

Complexity Analysis:

- **Time Complexity**: The BFS explores each possible state exactly once. Each state represents a pair of integers (jug1, jug2), and there are at most capacity1 * capacity2 possible states. Therefore, the time complexity is O(capacity1 * capacity2), where capacity1 and capacity2 are the capacities of the two jugs.

- **Space Complexity**: The space complexity is O(capacity1 * capacity2) due to the storage required for the visited states and the queue. The queue may hold at most capacity1 * capacity2 states, and the visited set also stores these states to avoid revisiting them.