Maxime Kotur

CSCI 6511 Project 1

The Water Jug Problem is defined as having an amount N buckets with each having an amount C capacity. A target capacity is given and the goal is the find the minimum amount of actions to reach the target capacity. Therefore, I defined that we have 4 main actions possible:

Fill: Fills the bucket to its full capacity

Drain: Empties the bucket to 0 and throws the water on the floor

Cap: Empties the bucket into to the target capacity infinite bucket

Pour: Pours the bucket into the selected other bucket

These actions would be translated into a Bucket class which would also have the bucket's capacity available and how full it is currently.

I also created a Node Class which contains the bucket array and the current state of the target capacity. Each node will have a parent node and their children. The children are defined as the possible and legal actions the node can perform. The children are therefore each possible state from the current state. Each node also carries a heuristic value which would be initially set as inf.

Finally, we have a Player class which allows us to run the algorithm. The Player class helps us set up the construction of the state space tree. We first define our starting node and create the state space tree until the node with the target capacity is found. This is done in the construct graph method. This will also return -1 if the goal state hasn't been found and we have no more legal actions to perform that haven't been done yet. Once the method is done, we can apply the heuristic value for each node using apply_heurisitc(). The heuristic value is an estimation of the real cost from node to end node.

Heuristic value lower bound: the lower bound is the number of edges between the node and the end node. First the h(end node) = 0. Then, I can get this value by starting from the end node and taking its parent while setting the parents cost to 1 and its children excluding the current node to 2. We would keep on doing this until we eventually reach the start node which would be the total cost to get to the end node.

Once the heuristic values have been applied, I run the A* search algorithm. I use a priority queue and add each child node with their heuristic values + 1 as the distance from one space to the other is 1. The child nodes are added if they haven't been visited yet and we loop until we either have no solution (priority queue is empty) in which case we return -1 or until we have our goal state where we return the number of steps. The number of steps is represented as the minimum cost path as we are using a priority queue. Thus, we return the steps to get the minimum and the algorithm is done.

For the **unit testing**, I created a new python file and used the unittest import. I created a few examples with the test cases given in the project documentation and others that I thought of. It uses all the edge cases where there is no solution (returns -1) and even when the target is 0 (returns 0). The other examples are working solutions which I have calculated by hand. The way the tests are passed is if the assertion that the Player.run() method returns the correct number of steps.