Reeya Gupta

CSCI 6511 Project 1

Description Document

Water Jug Problem:
Having an amount N of buckets, each with an amount C of capacity, is the "Water Jug Problem."The objective is to determine the smallest number of actions necessary to take in order to reach the given target capacity. 4 actions have been defined:
Fill: Completely fill the bucket with liquid
Drain: Throws the water on the ground after emptying the bucket to zero.
Cap: Empties the bucket into an infinite bucket with the desired capacity.
Pour: Empties the bucket into the other one you've chosen.

These operations would be translated into a class called WaterBucket, which would also contain information about the bucket's capacity and current level of filling.
Additionally, I developed a BucketNode Class that houses the bucket array and the target capacity's present condition.
There will be a parent node and its child for every node. The possible and legitimate actions that the node can take are referred to as the children. Each potential state from the current state is therefore represented by the children. Additionally, each node has a heuristic value that is initially set to inf.
The MyPlayer class allows us to execute the algorithm, which is the last step. We can set up the construction of the state space tree with the aid of the MyPlayer class.
Prior to finding the node with the desired capacity, we first define our starting node and build the state space tree. The construct_search_tree method performs this. If the goal state cannot be reached and there are no more legal actions left to take, this will also return -1.
Once the method is complete, we can use heurisitc to apply the heuristic value to each node().
An estimation of the true cost from node to end node is the heuristic value.

The number of edges between the starting node and the ending node serves as the lower bound for the heuristic value. Initially, h(end node) = 0. Then, starting from the end node, I can take its parent while setting the parents cost to 1 and its children, excluding the current node, to 2. This will give me the value that I need. This process would be repeated until we finally arrived at the start node, which would represent the total cost of traveling to the end node.
I run the A* search algorithm after applying the heuristic values. I employ a priority queue and multiply the heuristic values of each child node by 1 to add them all together.
We loop until either we have no solution (the priority queue is empty), in which case we return -1, or until we reach our goal state, in which case we return the number of steps. The child nodes are added if they haven't already been visited. Since we are using a priority queue, the number of steps is represented as the minimum cost path.
Thus, the algorithm is finished after we return the steps taken to obtain the minimum.