Georgia Tech CS7637 KBAI

Mini-Project 2: Block World

Viktor Kis

vkis3@gatech.edu

1 INTRODUCTION

This project explores how an Artificial Intelligence (AI) agent could be constructed to solve a stacks constrain satisfaction problem. The goal of this constrain satisfaction problem is to re-stack blocks from some given initial state to some goal state with the following rules:

- There may be up to 26 unique blocks (each block is a unique block that is represented by a character such as "A", "B" and so on)
- These blocks can be in any initial configuration. Fully stacking on top of each other in one single stack. Completely flat on the table (1 block per stack) or any combination in between
- Only 1 block may be moved at a time, and only from the top of a stack
- There is no limitation on the number of stacks during any point of the problem. For example, an initial single stack of 10 block may be completely unstacked to 10 single block stacks
- The solution must be optimal (least number of moves to reach the goal state)

An example of an initial state and goal state is shown in Figure 1 below.

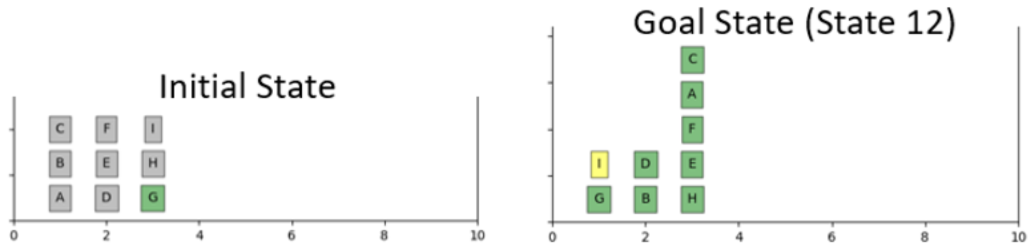


Figure 1— Example Problem Initial State vs Goal State

2 AGENT DESIGN

In simple terms the agent design follows an information driven approach that assesses available options (all tops of the stacks) and makes an informed decision on whether there is a goal productive step or an unstacking is required. Although there was no limitation on number of stacks generated in process of solving the problem, if a goal productive step was available, it was prioritized over unstacking (there by limiting the maximum number of stacks used). After further investigation there could have been further improvements made with a heuristic approach to further limit the maximum number stacks used for a solution; essentially prioritize unstacking that would lead to finishing a goal stack earlier.

2.1 Agent Solver Logic

As mentioned above, the agent uses an information driven approach that is best described by KBAI Production System Frames, where each block has attributes such as:

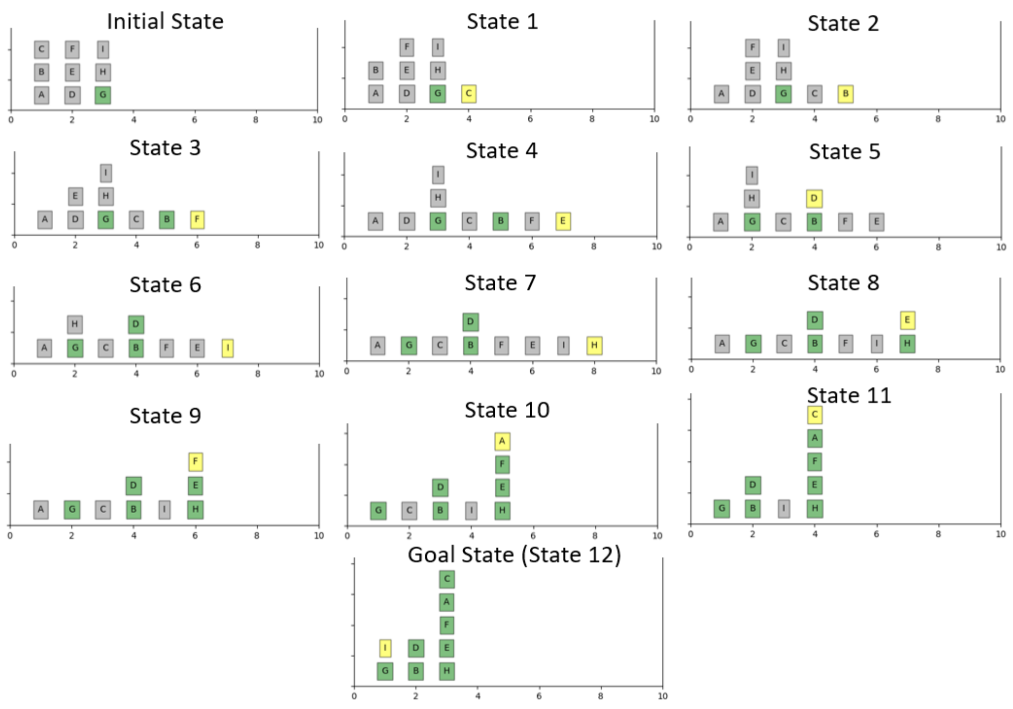
- stack index (which stack the block belongs to)
- block index (within the stack what is the index of the block)
- what is below the block
- what is above the block
- is the block (and its stack up to this point) correct

With this approach of Frame description, the looping through information was made much more efficient as opposed to looping through the stack with each state configuration. All of the above attributes were straight forward to gather in a single loop of the initial state and the goal state, with the exception of the "correct" block attribute. The correct block attribute was done with 2 pieces of logic:

- cross validation of block below between current state and goal state (i.e.: is the current block stacked on the same block as the goal state)
- and did the block below also validate to True (i.e.: the stack is correct up to this point). This was an important addition to the logic mentioned above

because the evaluation of a stack of “A”, “B”, “C” in current state may evaluate to true when assessing block “C” against a goal state of “D”, “B”, “C” however as we can see we would have to unstack all to fix the incorrect bottom block of “A”

After the initial loop through the current state and the goal state to gather the above attribute information, the rest of the logic was quite simple. A block that was goal productive was selected first, if there were none, then the 1st in queue was unstacked onto the table. This is best illustrated in the agents graphical output shown in Figure 2.



[('C', 'Table'), ('B', 'Table'), ('F', 'Table'), ('E', 'Table'), ('D', 'B'), ('I', 'Table'), ('H', 'Table'), ('E', 'H'), ('F', 'E'), ('A', 'F'), ('C', 'A'), ('I', 'G')]

Figure 2— Example Problem (Gray: incorrect, Green: correct, Yellow: last moved)

2.2 Agent Performance

The agent’s design was optimal, meaning it always solved the problem with the least number of moves, because it never repeated a move (such as placing the same block on the table more than once) or made a counter-productive move (such as placing a block on a stack that would require unstacking again). There were no known configurations that presented the agent design any issues solving and the Production System Frame attributes approach did not require any edge case programming.

2.3 Agent Efficiency and Cleaver Improvements

The agent's efficiency in time complexity is O(4N) in worst case scenario where the initial state is completely reversed from the goal state. The agent would have to loop through the initial state (1xN), the goal state (1xN), then unstack (1xN) and restack (1xN-1).

The Frame attributes agent design is pretty cleaver in that it prevents unnecessary looping during the state update / solver steps since all of the information about what is above/below a block is already determined on a single initial looping through the data. It is important to note that space complexity is much more costly with a Frame agent design since there is a lot of "unnecessary" information stored in memory that is not used in a state update. For such a small problem, this is of course not an constrain on the machine, however a large scale application of this approach may warrant re-evaluation.

3 AGENT LOGIC VERSUS HUMAN LOGIC

The agent design was originally written up as a Greedy Best First Search algorithm using a heuristic function that gave a score to each block, then tallied up a stack's score to determine which stack was most nearly complete. This approach was sound but seemed way too complicated with the management of all the edge cases. The reason why I went down this path originally was to challenge myself and try to use the least amount of stack while maintaining optimal minimum number of moves. As mentioned above this approach quickly became complicated to manage and was far away from how a human would solve this problem. Then I took a look at how I was solving these problems by hand again and realized that my human approach was available information based. See what moves I can make, and see if any of them are productive. If I have a productive move, then make it, else place it on the table. This approach was translated to my agent logic which was surprisingly simple to implement.