Mini Project 2

Skyler Dale
sdale31@gatech.edu

1 AGENT DESIGN

In order to solve the block world problem, I built an agent that uses a combination of means-end analysis and production rules. By imparting the agent with strong heuristics around what the "next best" move is, I was able to create an efficient and performant solution.

1.1 High Level Approach

Means-ends analysis is an approach to sequential problem-solving where the next move selected is the one that most effectively reduces the distance between the current and target states. Because means-ends analysis is a universal AI method, it is applicable to many different problems, but it does not guarantee success for specific problems.

In order to get around this, I designed an AI agent that is intelligent about *how* it selects its next move. My solution reflects a hybrid between means-ends analysis and production rules, whereby the production rules help establish heuristics that ultimately move the agent closer to its target state. Below is the high-level overview of the rules:

1. A move always brings the current state closer to the target state if it is a move that *is required* for a target state to be reached. This is captured by the following rules:
    a. If Block A is "on top of" block B when block B should be on top of block A, move block A to the table
    b. If Block A is "on top of" block B when it should be, *but* the blocks are part of a stack that is out of order, move block A to the table
    c. If Block A should be on the table, move block A to the table
    d. If Block B is on top of an "ordered stack" and block A should be on block B, move block A on top of Block B
2. If no moves in (1) are possible at the current state, then move the block that is intended to be "lowest" in the target state to the table.

a. Here, we deprioritize blocks that are "higher" in the target state because we have more "time" to find a "required" move for this block (ie. one that is guaranteed not to be a wasteful move)

1.2 Class Design

In order to implement this solution, I created three python classes: a Block class, a Stack class and a State class. State objects store a list of stacks in their "stacks" attribute and stacks store a list of blocks in their "blocks" attribute. This design allows us to represent the current state at all times in a frame-like structure.

I also designed other attributes and methods shown in figure 1 for each of these classes. For example, blocks are aware of their targets (target_name), their height in the optimal solution (height) as well as the blocks they are currently on (is_on_name). Stacks are aware of whether all of their blocks are on the right target or not (is_ordered) and which block object is on top (block_on_top). Finally, states are aware of which stacks are ordered (ordered_stacks, unordered_stacks, is_optimal) and have the ability to "move" blocks and record moves (move_blocks, move_block_to_new_stack, moves).

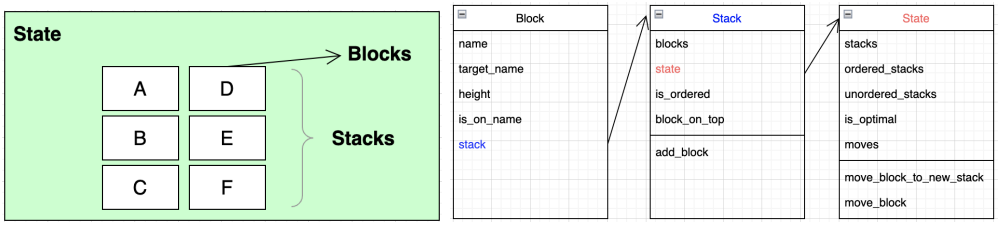

Figure 1—A semantic representation of the classes (left) and a class diagram depicting attributes and methods (right).

1.3 Solution

In order to solve each problem, the agent starts by initializing the stack, state, and block objects using the arrangements provided. Next a while loop is executed that checks whether the state is optimal. While the state remains non-optimal, the agent loops through the top of each unordered stack searching for the move that takes it closest to the target state. If the agent finds any "required moves" it executes them. If no required moves are found, the agent moves the block that has the lowest height in the target solution to the table.

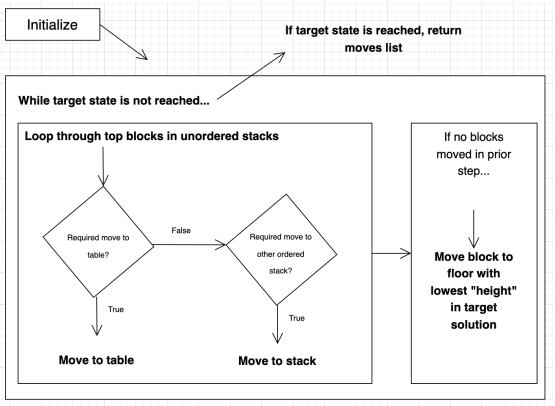

Figure **2**—The agent makes required moves when possible, in order to minimize the likelihood of making an unnecessary move.

2 AGENT PERFORMANCE AND EFFICIENCY

2.1 Performance

I define performance as the ability to accurately get from the current state to the target state. Because my agent is always able to make a "next move," it can solve any block problem. In the worst case scenario, it would be forced to move all blocks to the table and then it would be guaranteed to find the solution from there because there will always be "required moves" at that point. This is further evidenced by the ability of the agent to solve all of the test problems correctly.

2.2 Efficiency

I define efficiency as the ability of the agent to solve the problem with the least amount of computation. One way to measure "computation" is to evaluate the number of "moves" the agent makes.

My agent has a good chance of getting to the target state in the fewest moves. This is because my agent "knows" when moves are "required." Whenever it finds one of these scenarios it is guaranteed to be making the least expensive choice. On the other hand, when the agent is faced with a scenario in which no "required" moves are available, it falls back on the heuristic described in 1.1. Because these moves are not "required" there may be scenarios where they are not the most efficient.

As the number of blocks increases, the agent has to make more total moves on average and that increases the likelihood that it will face scenarios where it has to

fall back on the height-based heuristic. As a result, the agent might make more unnecessary moves when the number of blocks increases, which ultimately decreases its efficiency as defined above.

2.3 Clever Design Choices

The most clever design choice that my agent employs is the ability to recognize scenarios that "require" moves in less obvious ways (figure 3). For example, the fact that we move a block off the top of a stack when that stack is unordered and the block is on its target, is an observation that requires a deep level of reasoning. This design choice makes the agent more efficient because it reduces the amount of moves the agent makes based only on block height.

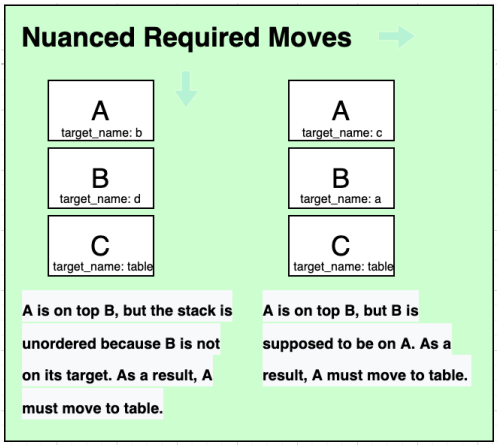

Figure 3—The agent finds "required moves" that are not immediately obvious.

3 COMPARING AI AND HUMAN REASONING

This approach is similar to how a human would solve the problem. As discussed in lectures, humans use weaker methods like means-ends analysis when they are not experts. However, when they have good knowledge about the world, they tend to use more sophisticated methods. In this case, I've given my agent explicit knowledge in the form of rules that help it solve the problem. Because I came up with these rules myself, this is an implicit similarity with how I would solve the problem. One difference, however, is that I can learn and adapt new rules when I'm solving the problem, whereas my agent can only use the rules I've endowed it with up front and cannot learn anything new.