Mini-Project 2:
Block World

Lukas Olson
lolson6@gatech.edu

1 INTRODUCTION

The purpose of this assignment is to design an agent capable of solving the Block World problem. The problem involves stacking and unstacking blocks to move from an initial configuration to some goal arrangement. Only one block may be moved at a time, and only the top block in each stack can be moved. Blocks may be moved to the top of another stack or to the table to start a new stack.

Figure 1 depicts a simple example of one possible problem. In this example, the goal state can be reached in a single move: unstack C from the left-most stack and stack it onto D in the center stack.

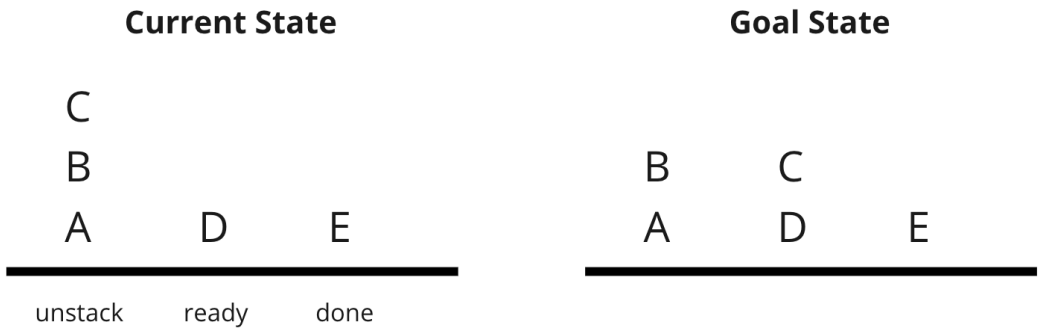

Figure 1—An example problem representation.

2 AGENT DESIGN

I went through several designs to solve this problem. Initially I tried breadth-first search with generate and test, but immediately found that the search space grew too quickly without severe pruning. I transitioned to uniform cost search and tested various heuristic functions to sort the search frontier. While designing my heuristics, I realized that at each state of the problem it is possible to identify precisely which moves will progress towards the goal state. This simplifies the implementation because searching over multiple candidate moves is no longer necessary. Ultimately, the method I arrived at is means-ends analysis applied to a semantic network.

2.1 Problem Representation

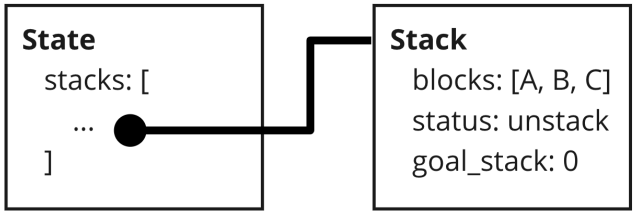

Figure 2—An example of the frames used to store information about each state.

The semantic network represents each possible state (block arrangement) as a node in the graph and each move as an edge. The information about each state is stored as a collection of stacks. Each stack has several slots: the blocks in that stack, the status of the stack, and the index of the corresponding stack in the goal state, if any. Together this information comprises the frames exemplified in Figure 2.

The status slot of each stack takes on one of 3 fillers:

- Unstack: In order to progress toward the goal state, blocks must be removed from the top of this stack.
- Ready: This stack matches the bottom portion of one of the goal stacks, and blocks must be added to it to progress.
- Done: This stack perfectly matches one of the goal stacks, so no more moves should be applied to it.

Finally, the goal_stack slot identifies the stack in the goal state that matches the bottom blocks in the current stack, if any blocks match. This attribute is used to determine the stack's status and thereby the next sub-goal.

2.2 Solution Algorithm

In each iteration of the algorithm, the agent relabels the stacks in the current state to determine the next sub-goal to achieve. The agent checks each pair of stacks labeled 'ready' and 'unstack' to determine whether moving the top block from a stack labeled 'unstack' to one labeled 'ready' will make progress towards the 'ready' stack's goal_stack. If so, the agent executes that move and proceeds to the next iteration. If not, the agent finds an 'unstack' stack with more than 1 block and moves its top block to the table. This process repeats until all stacks

are labeled as 'done', and the agent returns the moves list.

The sub-goal labeling process is the key component of this algorithm and what makes the means-ends analysis approach so efficient. To find the corresponding goal_stack, the labeler compares the bottom block in the current stack to the bottom blocks of each stack in the goal state. If there is no match, the stack must be fully unstacked to progress, so it is labeled as 'unstack'. If there is a match for one or more blocks from the bottom up, but the top blocks do not match, the top blocks must also be unstacked, so it is labeled as 'unstack'. If the stack perfectly matches the goal stack it is labeled as 'done'; otherwise it is labeled as 'ready'. These labels allow the agent to reduce the number of possible moves to evaluate and to test whether any blocks can be placed on a 'ready' stack.

3 EVALUATION

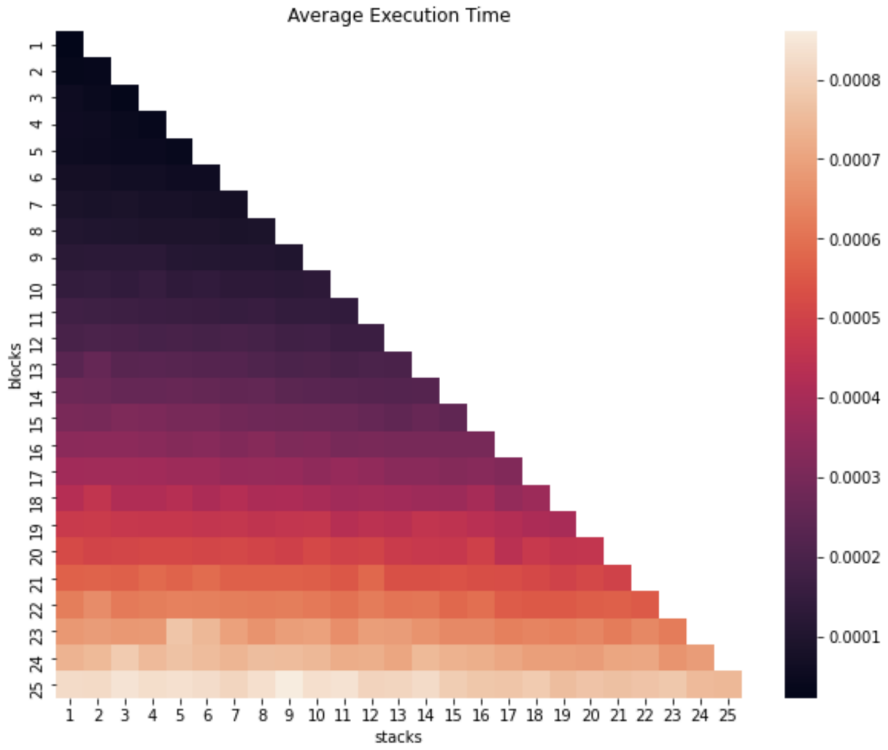

Figure 3—Execution time for each combination of numbers of blocks and stacks, averaged over 100 runs.

This approach met all of the auto-grader's performance criteria, finding both valid and optimal solutions every time. Figure 3 summarizes its efficiency in terms of time. I randomly generated problems with a given number of blocks

and stacks and averaged the agent's efficiency over 100 problems for each combination of parameters. The average execution time increased approximately quadratically with respect to the number of blocks. The execution time is inversely related to the number of stacks because when more blocks are consolidated in very few stacks, those stacks will need to be unstacked in order to rearrange the blocks. With more stacks, fewer blocks need to be unstacked to reach the block that must be stacked next.

Although a less meaningful measure, the number of iterations of the main loop of the algorithm also provides information about the efficiency of the means-ends analysis. The number of iterations is perfectly correlated with the number of moves in the generated solution—meaning the algorithm only ever makes moves that contribute to the final answer. This is in contrast to an approach like breadth-first search where multiple moves will be attempted before finding one that progresses towards the goal. Although not a guarantee, empirical results from test cases I designed and the auto-grader's output indicate the algorithm always returns the most efficient set of moves possible.

4 CONCLUSION

The combination of means-ends analysis and a clever state representation using frames allowed my agent to perform efficiently on this task. Although my first attempts did not perform so well, they helped me recognize where my agent's approach deviated from my own when solving the problems by hand. This insight lead me towards means-ends analysis and the labels I used to identify sub-goals. The biggest difference between the algorithm's and my own approach is that I create more general sub-goals so that I do not need to reevaluate them after every move. For example, I can recognize that multiple blocks need be unstacked before I will reach the block I need to stack next, and therefore I can execute those moves all in quick succession. The algorithm, on the other hand, will unstack one block at a time, recalculating the sub-goal after every move. I could also improve my algorithm's efficiency by taking advantage of its working memory—I do not need to recalculate the sub-goal for stacks that do not change after an iteration.