Mini-Project 2: Block World

Kevin Ger
kger3@gatech.edu

Abstract—In this mini-project, we are presented with the block world problem where we need to move an initial arrangement of blocks into a designated goal arrangement. I will begin with providing my approach in designing the agent to solve the problem. I will then go over the performance of my agent and end with a comparison between humans and my agent.

1 OVERVIEW OF AGENT'S DESIGN

1.1 Approach

Similar to mini-project 1, I chose to solve this problem using the BFS algorithm with a generate and test method. However, I will be using a smart generator which only generates necessary states, therefore it also contains means-ends analysis.

1.1.1 *Generating new states*

Unlike mini-project 1, there are many possible moves depending on the initial block count. In the worst case scenario, you can start with all 26 letter blocks that are available for movement, generating 26 possible next states. To combat this, I decided to go with a smart generator.

To begin with, I decided to create a state for each block. Each block will remember what block is under it and what block is on top of it. If it's the bottom block of the stack, its "bottom block" would be the table; if it's the top block of the stack, its "top block" would be empty. All the block states will then be included in a dictionary structure to form the current state of the arrangement.

While a block is valid for movement, it doesn't necessarily mean the block should be moved. Generally, we don't want to disturb the stacks that are already in their goal state. To start, I've identified two major states: block is on the table and can be moved onto a stack and block is on a stack and can be moved onto the table or another stack. From here I go deeper into my restrictions in generating the next state. I've made it so that the block doesn't move unless necessary.

For blocks on the table, if their end state is on the table, they don't move anymore, if not then it'll only be moved onto the block that is its bottom block in the end state and if the stack it's being moved on is in the end state from its current top block. For example, if there is an end state stack [A, B, C, D] with A on the table, and currently we are trying to move D while a stack of [A, B, C] exists, then D can be moved on to C as C is D's bottom block in the end state and stack [A, B, C] is in the end state from C to A.

For blocks on the top of the stacks, if the entire stack from it to the bottom block on the table is in the end state, then they don't move anymore. If not, then I first check if it can be moved from its current stack to another stack. The new stack will have to be in its end state for its current blocks and the moving block will have to be part of the stack's end state for it to be moved onto the stack. If no such stack exists for the moving block, then it will be moved onto the table as its current stack is not part of its end state and its end state stack is not yet formed. Lastly, if the state is visited, then it won't be added to the list for processing to avoid loops.

1.1.2 *Testing new states*

The testing is relatively simple as I have chosen the generator to be smart. Since only necessary moves are made, only necessary states are generated. The only testing done is to check if the current state is equal to the end state.

1.1.3 *Wrap up*

Using the BFS algorithm, I had a structure to keep track of the visited states, which I backtrack through to find the path from start to end. Then, I use two pointers to look at the current state and the next state to find the move being performed and add it to the list of moves to return.

2 PERFORMANCE AND EFFICIENCY OF AGENT

2.1 Performance

My agent was able to solve all 40 problems on Gradescope, providing an optimal path for each question.

2.2 Efficiency

I believe my agent is efficient as it only generates necessary states. Stacks in their end states or close are not touched. Blocks only move onto stacks if it's part of the stack's end state and moved to the table to make way for stacks to be created towards their end states.

I believe including the means-ends analysis to the BFS implementation is the clever part of my agent. Normal BFS would require me to generate all possible states for traversal. If I had infinite computational time and space, it wouldn't be a problem. However, I did run into long computation times by generating all possible next states. Having my agent detect subgoals where end state stacks are not to be touched and blocks only require specific movements allowed my agent to work faster than before.

2.3 Increasing the numbers of blocks

I was able to perform random testing using the code provided by our classmate Kiran Gopala Iyer. We can see below how each variable contributes to the calculation time.

Table 1—A snapshot of how my agent performed with an increase in block count and differences in initial and goal arrangements.

Block Count	Initial Stack Count	Goal Stack Count	Calculation Time (sec)	Initial Arrangement	Goal Arrangement
14	4	4	0.0524795	[['A', 'B', 'C', 'D'], ['E', 'F', 'G', 'H'], ['I', 'J', 'K', 'L'], ['M', 'N']]	[['D', 'B', 'C', 'A'], ['F', 'G', 'H', 'E'], ['L', 'I', 'J', 'K'], ['M', 'N']]
14	2	4	0.2765398	[['A', 'B', 'C', 'D', 'E', 'F', 'G'], ['H', 'I', 'J', 'K', 'L', 'M', 'N']]	[['B', 'D', 'A', 'C'], ['F', 'G', 'E', 'H'], ['K', 'L', 'J', 'I'], ['M', 'N']]
15	3	3	0.1547872	[['A', 'B', 'C', 'D', 'E'], ['F', 'G', 'H', 'I', 'J'], ['K', 'L', 'M', 'N', 'O']]	[['A', 'D', 'E', 'C', 'B'], ['H', 'J', 'F', 'G', 'I'], ['L', 'M', 'O', 'K', 'N']]
15	3	5	0.3413254	[['A', 'B', 'C', 'D', 'E'], ['F', 'G', 'H', 'I', 'J'], ['K', 'L', 'M', 'N', 'O']]	[['B', 'C', 'A'], ['F', 'D', 'E'], ['G', 'H', 'I'], ['J', 'K', 'L'], ['N', 'M', 'O']]
15	5	5	0.1399548	[['A', 'B', 'C'], ['D', 'E', 'F'], ['G', 'H', 'I'], ['J', 'K', 'L'], ['M', 'N', 'O']]	[['A', 'B', 'C'], ['F', 'E', 'D'], ['G', 'I', 'H'], ['J', 'L', 'K'], ['O', 'M', 'N']]
15	5	5	2.3833443	[['A', 'B', 'C'], ['D', 'E', 'F'], ['G', 'H', 'I'], ['J', 'K', 'L'], ['M', 'N', 'O']]	[['C', 'A', 'B'], ['F', 'D', 'E'], ['H', 'I', 'G'], ['K', 'L', 'J'], ['N', 'O', 'M']]

I was able to run the test for more than two hundred variations, and using the above snippet we can observe that while the increase in block count doesn't necessarily contribute to the increase in calculation time, what really makes my agent compute longer is the complexity between the initial arrangement and the goal arrangement. I believe I can safely conclude that while increasing the block count isn't correlated to increase in computation time, it does give the possibility of more complex initial and goal arrangement difference, which will then increase computation time.

3 COMPARISON BETWEEN HUMAN AND AGENT

3.1 Human approach

While I've only done the block world problem on paper, I feel like I'd have a good grasp of what I would do if I had physical blocks in front of me. If I had physical blocks in front of me and were tasked to arrange them from one arrangement to another arrangement, I don't believe I would use any sort of algorithm or tricks to do so. The only restrictions we have are that we can only move one block at a time and only the blocks that have no blocks on them, therefore the easiest way to go about rearranging is to simply choose an end state stack to create at a time and arrange them until all stacks are arranged. Many blocks will be moved onto the table in the process as I believe it'll take me much longer to think if moving them is optimal than to simply put them on the table and worry about them later. As a human, I will create numerous nonoptimal next states, but will drastically decrease my computation time by letting my hand move continuously instead of letting my brain think.

3.2 Comparison

Generating states is costly for time and space, so an agent should never be designed to just move any movable blocks onto the table to rearrange the stacks. Some similarities humans and my agent share is that they will both not touch stacks that are in their end states. The major difference, as mentioned above, is that humans can generate numerous nonoptimal states and have the possibility of finishing faster than if they tried to move optimally while my agent will try to generate as few states as possible.