Mini-Project 2: Block World

David Adamashvili
dadamashvili3gatech.edu

***Abstract*— In this paper I will outline my solution for the Block World problem. We are given two configurations of blocks, a starting configuration and a goal configuration. There are several stacks of at most 26 unique blocks. We want to get to the goal configuration using as few moves as possible. All moves consist of taking a block from the top of any stack and either putting it on the table, or onto another stack.**

1 AGENT DESCRIPTION

1.1 A* (Failed attempt)

At first, I tried to solve this problem using a **Means-Ends Analysis** approach. However, a normal **BFS** algorithm would have been too slow, as the number of nodes in the state graph is astronomical. I hoped to remedy this by using the **A*** algorithm along with the heuristic provided in the lectures, and while it solved some of the smaller problems in contrast to **BFS**, it still failed to solve anything with more than a few blocks.

1.2 Problem reduction

My next approach was to look at the problem differently and solve it in a more step-by-step algorithmic manner. My first observation was that, if the bottom few blocks of a stack were already correctly placed, they should not be touched. Every such correct sub-stack either has no block on top of it, or at least one incorrectly placed block. We want to replace this incorrectly placed block with a correct one. To find the block that's supposed to be there, we look at the corresponding stack in the goal configuration to see which block is on top of this sub-stack, and then look for the position of this same block in the starting configuration.

So, at some point in our algorithm, this block needs to move on top of the correct sub-stack. This is a **sub-problem** for our final goal. But, there must not be any blocks on top of these two to move one onto another. This means that, to solve this **sub-problem**, we in turn need to solve another **sub-problem** of freeing the top of a block. We do this by sequentially removing all of these blocks. Where should we place the blocks we remove? Well, if one of them can be placed on an already correct stack so that it still remains correct, then do so. Otherwise, move it onto the table.

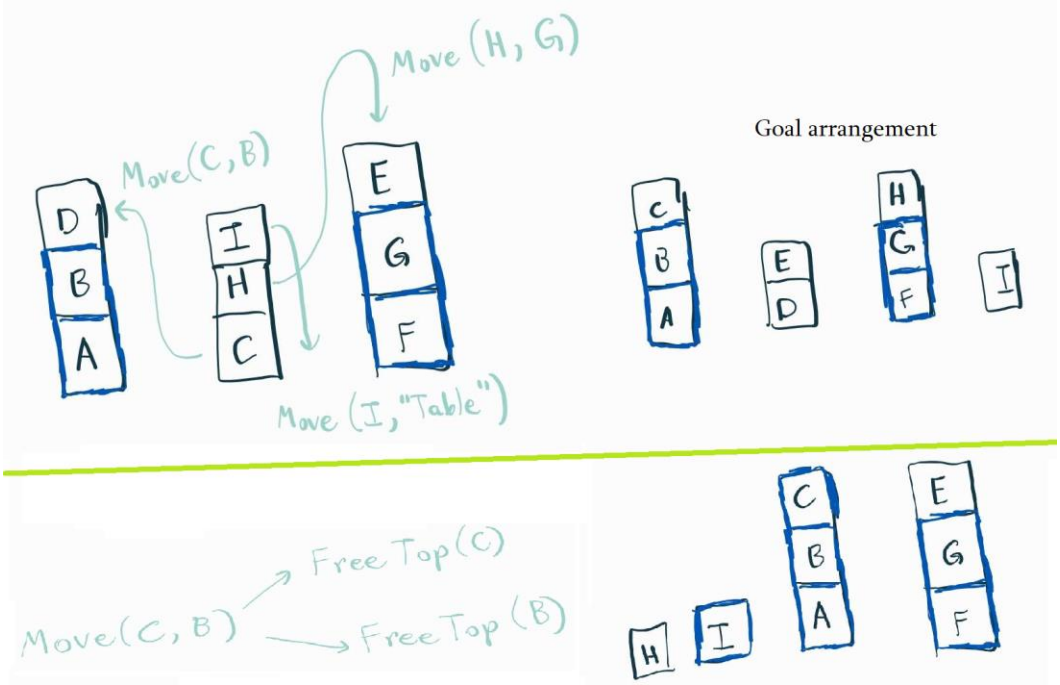

Figure 1 - **Generating subproblems**

So the step-by-step approach can be described as follows:

1. Find correct sub-stacks
2. Find blocks that need to be moved onto these sub-stacks to generate **Move(Block_1,Block_2)** sub-problems, where **Block_1** needs to be moved on top of **Block_2**.
3. For each **Move(Block_1,Block_2)** sub-problem, we need to solve **FreeTop(Block_1)** and **FreeTop(Block_2)** sub-problems as described above before **Move** can be executed.

4. Repeat until we reach the goal state

1.3 Algorithm correctness and optimality proofs

I will only provide a proof sketch. It's obvious that the algorithm will terminate, because after each iteration the number of correctly placed blocks increases by at least one. This also shows that the algorithm will always arrive at the correct answer.

To prove optimality, first we need to notice that there is no point to moving an incorrectly placed block onto another incorrectly placed block, because they will need to be moved eventually. It only makes sense to "lengthen" correct sub-stacks. The blocks above these sub-stacks will eventually need to be removed. It shouldn't be too hard to show that the order of these operations does not matter. After all of this is proven, if the order does not matter, if we show the optimality of an isolated **Move** operation, it will be sufficient to prove the optimality of the whole algorithm. This should be pretty easy.

2 PERFORMANCE AND EFFICIENCY

2.1 Performance

My current implementation of the algorithm gives a 37/40 score. It returns sub-optimal answers for 3 test cases. This is because the abovementioned algorithm is not fully implemented. In my code, the **FreeTop** function puts the removed blocks only on the floor, which is sub-optimal for those three test cases, as those blocks can go on top of a correct sub-stack.

This means that, the algorithm only returns optimal solutions for those kinds of problems where once we start freeing the tops of correct sub-stacks, the removed blocks would need to go onto the table. In this case, I honestly got lucky with how well the algorithm did without this additional check, however at this point in time I do not have enough free time on my hands to add another check to my **FreeTop** function.

2.2 Efficiency (Big O)

Note: I will not do an amortized analysis, all of the following claims are about rough upper bounds. Also, I am sure that a more efficient implementation of my

algorithm could exist, but at the moment I am short on time and cannot pursue these thoughts further.

All of the sub-problem functions such as **FreeTop**, **Move** and helper functions that they invoke (such as transforming arrangements to various other representations) contain doubly nested loops. Move contains two calls to **FreeTop**, but this doesn't change its complexity. Hence, all of them have O(n^2) time complexity where n is the number of blocks. Since there are at most linear amount of **Move** sub-problems, the total asymptotic time complexity of the algorithm is O(n^3).

As for space complexity, my program keeps a single deep copy of the initial arrangement, meaning the space complexity is O(n).

3 HUMAN COMPARISON

Even though my algorithm is intuitive, it's not something that most humans would use in my opinion. What most people would do would be to move every block onto the table, and then assemble the goal arrangement from there. This, of course, is not the optimal solution in contrast to the algorithm using problem reduction. However, as we know from the course, human cognition is not always optimal in its nature.

The human approach is similar to the algorithm I gave in the sense that it's not an exhaustive search. Both do something clever (though one of them is more clever than the other!). One advantage of the human algorithm is the ease of implementation and asymptotic performance, which will be linear. This is because every block will be moved at most twice, and we know that **O(2*n) = O(n)**.

But, on the other hand, every human is unique and I'm sure a some smart people (not me) would come up with the problem reduction idea on the spot. This is the beauty of being a human – we all approach problems differently and can generalize our thinking patterns. However, an agent can only do what it has be taught to do. Unless we teach it to "learn" problem solving strategies, it will always have a very narrow use case.