# CS7637 Mini-Project 2:
# Block World

Clement Lu
clu328@gatech.edu

## 1 DESIGN

### 1.1 Overview

The agent works using a hybrid method which utilizes both generate and test and means-end analysis techniques. In each iteration, the agent generates possible following states and tests for the productivity of each state. The definition of what a productive state means will be explored further below. Following, the agent calculates the cost of making each candidate move and assigns it a numerical value based on how close it moves the state towards the goal. Utilizing a mean-end analysis technique, the agent then chooses the next state to explore by choosing the state with the smallest cost from the queue. This process is repeated until the goal state is reached. The agent backtracks from the final state to obtain the list of moves from the initial state and returns.

### 1.2 Environment

The state itself is represented by a N-tuple of 2-tuples, e.g., (("A", "B"), ("B", "Table")). Each tuple represents an "On" relationship where the first element in the tuple is the block that is on top of the second element in the tuple. This makes it easier to identify differences between candidate states and the goal state since each individual relationship can be compared. Each state is contained in a custom node class which also holds references to its parent node, the move taken to arrive at that node, and the costs associated with making that move with respect to the goal state.

### 1.3 Generator

Given a state, the generator returns a list of possible next states. The generator itself is a "smart" generator, i.e., it only generates moves with blocks that can be moved and only to locations that are unobstructed. To determine the blocks that can be moved, the agent collects every second element of the 2-tuples in the current state into a set. The values of this set represent all blocks in which there

is another block on top and thus not available to be moved. The agent subtracts the set of obstructed blocks from the set of all the blocks in the problem to obtain the blocks which are free to be moved in the current state. This also lets the agent know which blocks are free to move another block on top of. The possible moves are obtained by generating permutations of size 2 for the unobstructed nodes list. By default, a move to the table for any unobstructed block is added since this move is always available.

### 1.4 Tester

The job of the tester is to eliminate any unproductive moves generated by the generator. For any move that moves a block on top of another block, if the blocks below are not already in the goal state, then this is considered unproductive since to move the blocks below to the goal state would involve moving the same block again. Notably, the tester does not eliminate moves of any block to the table since this step is the only intermediate step available (if required) to move a block to its goal position. The use of an additional cost heuristic, discussed later, will discourage the move of a block to the table if it is already in its goal position and so the tester does not check for this condition.

### 1.5 Search Algorithm

The agent utilizes the A* search algorithm for exploring the possible state space. A* is like BFS or DFS approaches in which the state space is explored in a branching tree structure; however, A* attempts to optimize this search by choosing the next best available state to explore. The best option is the state with the lowest total cost, `f`, as determined by the function `f = g + h`. Here, `g` is the cost from the initial state to the current state, and `h` is the estimated cost from the current state to the goal state. The estimated cost is determined by a heuristic which effectively is performing means-end analysis. An additional benefit of A* is that if the heuristic is admissible, i.e., it never overestimates the cost from the current state to the goal state, then A* is guaranteed to find the optimal path from the start state to the goal state ("A* search algorithm - Wikipedia", 2022).

For the Block World problem, generally the farther down a block is in a stack in the goal state, the more costly it would be to move a block to that position if it is already not in that position. To implement this logic, the agent first determines the number of blocks above each unique block in the goal state. This is done once

at the start of the problem and is stored in a dictionary with key-value pairs of block letter and cost. As a lower bound, this number can be used as a block's cost for being in the wrong position. The agent takes the difference between the set of block relationships in the goal state and the current state to determine any blocks in the wrong position. The final `h` value can then be determined by looking up the associated cost of each misplaced block and summing them up. This effectively encourages the agent to make moves that place the blocks in the correct order starting from the bottom of the stacks in the goal state.

For the `g` value, the agent simply increments the value by 0.1 each successive level down the tree that is the search space. This small incremental value helps the agent choose shorter paths and avoids potential ties. The total cost for a move is then determined by adding the `g` value with the `h` value.

### 1.6 Performance

In terms of finding a solution (if possible), the agent performs well in that it can find a solution for any number of non-trivial blocks (2 – 26) and does not appear to struggle in any particular cases. Figure 1, below, shows the average runtime of the agent with respect to the number of blocks in the problem.

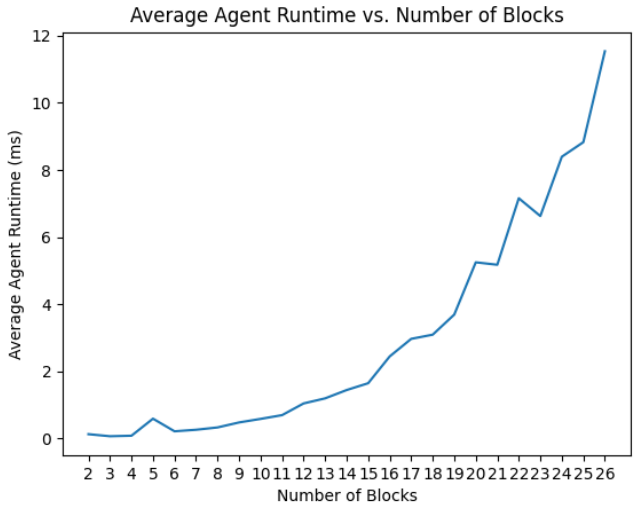


***Figure 1*** — Average agent runtime in milliseconds vs. number of blocks in the problem

We can see that the average runtime of the agent increases exponentially as the number of blocks increase. This makes sense as the search space also increases

exponentially as the number of blocks increase. However, at 26 blocks, the runtime is still relatively fast at just under 12ms. To test timed performance of the agent, every single configuration possible was tested with a range of 2 to 26 blocks. A unique configuration takes into consideration the number and size of the stacks in the input and goal configurations where each stack is adjusted individually, i.e., stacks do not have to be the same size in the input or goal configurations. The times for each of these input and goal configurations are then grouped by the number of blocks and averaged. Additionally, since the agent uses A* with an admissible heuristic, the agent is guaranteed to always find an optimal solution, i.e., a solution with the least possible number of steps possible.

### 1.7 Optimizations

The following is a list of optimizations that were made to help improve agent runtime:

1. Used set to store/retrieve visited states to reduce lookup time to O(1).
2. Used priority queue (heapq) to maintain candidate states. This reduces sort time to O(log(n)) and retrieving the lowest cost candidate state can be done in O(1) time.
3. Removed unproductive states to reduce overall search space.
4. Used heuristic to prioritize placing lower blocks in the goal state first.
5. Used A* to optimize state space exploration (compared to BFS/DFS).

## 2 HUMAN COGNITION COMPARISON

Compared to a human, this agent follows a similar approach that a reasonable person might take, albeit most likely with faster speed. The logic used for eliminating unproductive states and prioritizing placing lower blocks were both derived from domain knowledge I gained from manually stepping through sample problems and so the approach the agent takes is similar to my own approach. What the agent excels at is the ability to find an optimal solution versus just any solution. As a human, it is easy and intuitive to find a valid solution that works for any number of blocks, e.g., move all blocks to the table and then reorder as necessary, but it is not always easy to find the optimal solution. The agent does borrow from this human intuition though as it only moves blocks onto other blocks if the stack below is already correctly configured and otherwise moves it to the table if it is out of place.

## 3 REFERENCES


1. A* search algorithm - Wikipedia. (2022). Retrieved 13 February 2022, from https://en.wikipedia.org/wiki/A*_search_algorithm
2. Swift, N. (2020, May 29). *Easy A* (star) pathfinding*. Medium. Retrieved February 13, 2022, from https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2