World of Blocks Solver

Foundations of Artificial Intelligence

Ethan Heinlein

12/1/2023

Abstract

To solve the World of Blocks problem, a method of “Breadth-First” searching was implemented in a layered tree structure. The general approach of the search is to cast a wide net of actions by loading every action that can be performed on a given state simultaneously, then comparing each generated state to the goal state. Should the goal state not be found, each created state’s possible actions are loaded, and the cycle repeats until the goal is found. A few optimizations were added, such as filtering redundant “inverse” actions or back-to-back movement actions. When found, the correct path of states is built from the goal up, creating a cohesive series of states that visually display moving from the initial state to the goal state.

Table of Contents

I. Introduction

The World of Blocks problem is a classic Artificial Intelligence scheduling and planning problem [1] [2] [3] where a series of blocks are placed on a table in a particular configuration (an initial state), and a robotic arm is used to move the blocks around to reach a user-defined end (or goal) state. The arm can perform a variety of actions, such as picking up a block from the table, picking up a block from atop another block, stacking a block on another block, stacking a block on the table, and moving from one spot to another.

The goals of the project are to create a visual representation of the world of blocks, using 13 blocks labeled ‘a’ through ‘n’, and show real-time information displaying the current progress of the program as it performs various actions. Additionally, another goal was to create a user interface that could allow the user to perform a variety of actions, such as: entering the initial and final states to travel between, view each state individually, and report a variety of stats like how many states are in between the initial and goal state, and the time it took for the program to reach the goal state.

This project is built on Python 3.12, and uses the TKinter as a framework to build the GUI. The choice in platform and GUI library comes from personal preference, as I have experience building small programs in TKinter, and prefer it for quickly building GUI programs.

The following report details two experiments run by the program and their results, a deep dive into the methodology implemented, the justifications for said approach, different technical choices that were made, as well as the changes made to the program since the in-person presentation.

To test the functionality and capability of the program, two experiments were run: a simple problem (**Experiment 1**) and a complex problem (**Experiment 2**). Both are covered in more detail below:

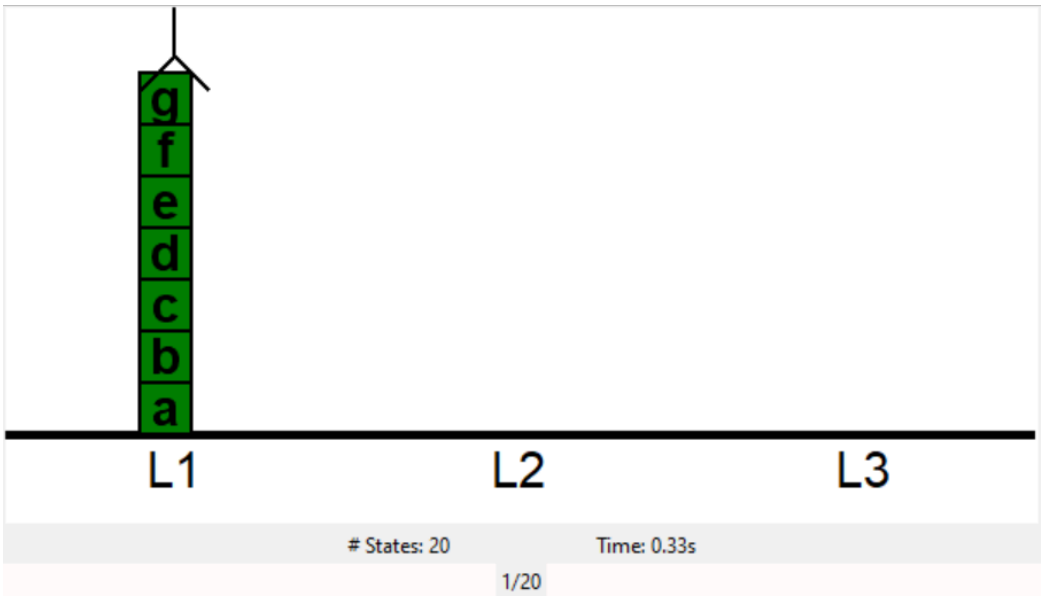
a. Experiment 1

Starting from the initial state L1: abcdefg to the final state L1: a, L2: bcd, L3: efg

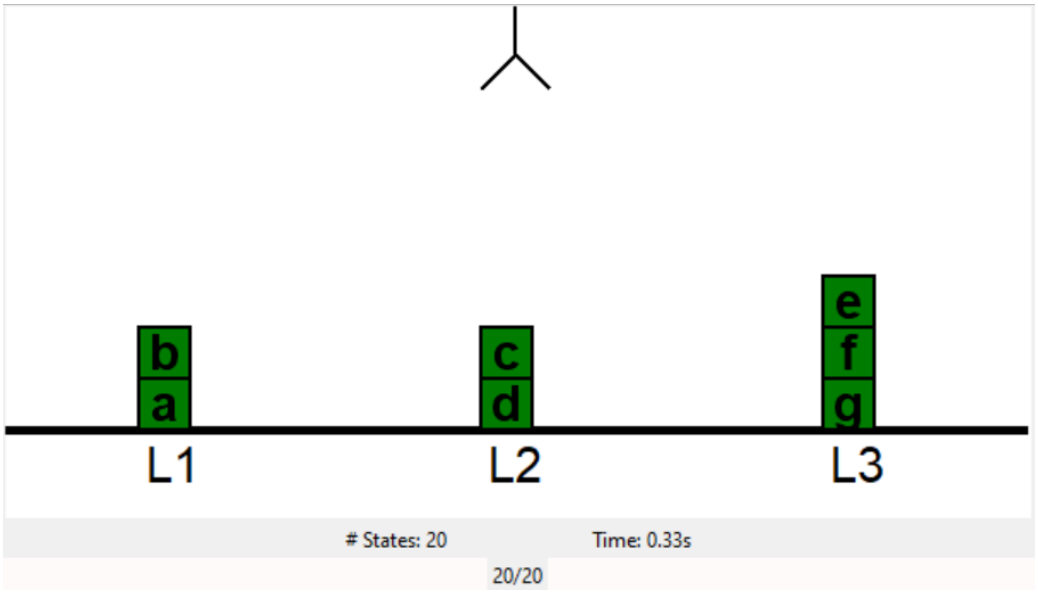
{'Found': True, 'States': 19, 'State_Data': ['abcdefg---1', 'abcdef---1g', 'abcdef---3g', 'abcdef--g-3', 'abcdef--g-1', 'abcde--g-1f', 'abcde--g-3f', 'abcde--gf-3', 'abcde--gf-1', 'abcd--gf-1e', 'abcd--gf-3e', 'abcd--gfe-3', 'abcd--gfe-1', 'abc--gfe-1d', 'abc--gfe-2d', 'abc-d-gfe-2', 'abc-d-gfe-1', 'ab-d-gfe-1c', 'ab-d-gfe-2c', 'ab-dc-gfe-2'], 'Time': 0.33}

(Pictures of each state will be available in the /Experiment 1/ folder).

19 States created in 0.33 seconds.



The initial state of **Experiment 1**.



The final state of **Experiment 1**.

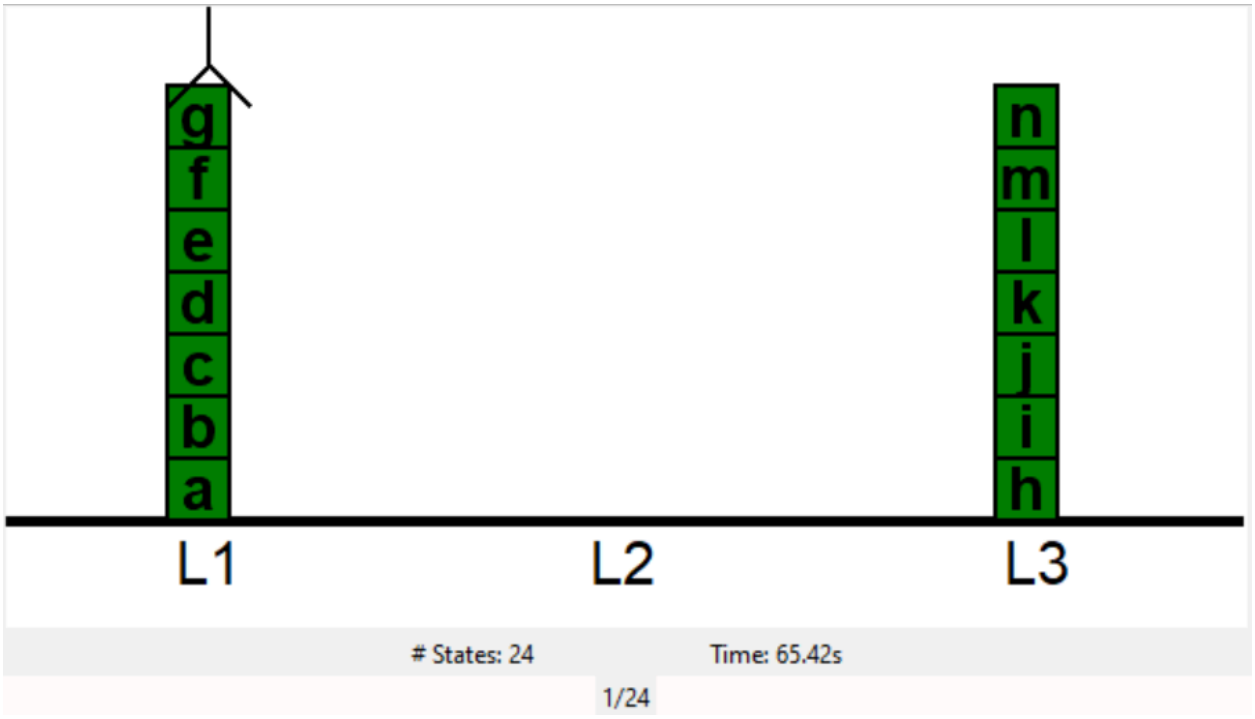
b. Experiment 2

Starting from the initial state L1:abcdfg, L3:hijklmn to the final state L1: abcd, L2: efglmn, L3: hijk

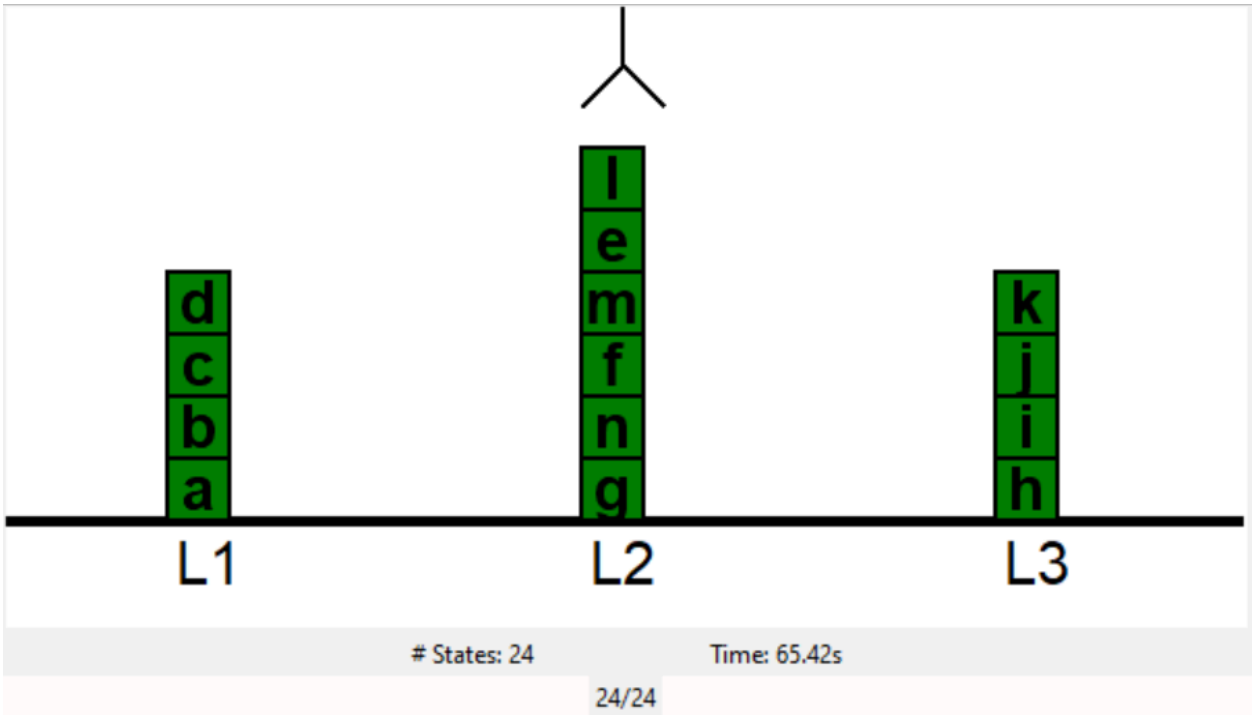
{'Found': True, 'States': 23, 'State_Data': ['abcdefg--hijklmn-1', 'abcdef--hijklmn-1g', 'abcdef--hijklmn-2g', 'abcdef-g-hijklmn-2', 'abcdef-g-hijklmn-3', 'abcdef-g-hijklm-3n', 'abcdef-g-hijklm-2n', 'abcdef-gn-hijklm-2', 'abcdef-gn-hijklm-1', 'abcde-gn-hijklm-1f', 'abcde-gn-hijklm-2f', 'abcde-gnf-hijklm-2', 'abcde-gnf-hijklm-3', 'abcde-gnf-hijkl-3m', 'abcde-gnf-hijkl-2m', 'abcde-gnfm-hijkl-2', 'abcde-gnfm-hijkl-1', 'abcd-gnfm-hijkl-1e', 'abcd-gnfm-hijkl-2e', 'abcd-gnfme-hijkl-2', 'abcd-gnfme-hijkl-3', 'abcd-gnfme-hijk-3l', 'abcd-gnfme-hijk-2l', 'abcd-gnfmel-hijk-2'], 'Time': 65.42}

(Pictures of each state will be available in the /Experiment 2/ folder).

24 states created in 65.42 seconds.



The initial state of **Experiment 2**.



The final state of **Experiment 2**.

II. Methodology

In general terms, the AI's methodology could be described as "breadth-first" searching [4] [5]. The initial state is loaded into a Tree data structure at layer 0. Then, every possible action that can be performed on the initial state is loaded into the next layer, Layer 1. If the goal state hasn't been found, the program will then individually load each state in Layer 1, queue up every possible action that can be performed upon it, into the next layer, and compare that new state to the goal state. It will continually search every action in the layer for the goal state, and move further down the tree until the goal state is found, or the depth limit is reached.

The program runs every possible action that can be performed on a state in parallel, and compares it to the goal state; if it is not the goal state, every possible action that can be performed on that state is placed into the next layer, as seen in **Figure 1** below, and the cycle repeats until the goal state is found.

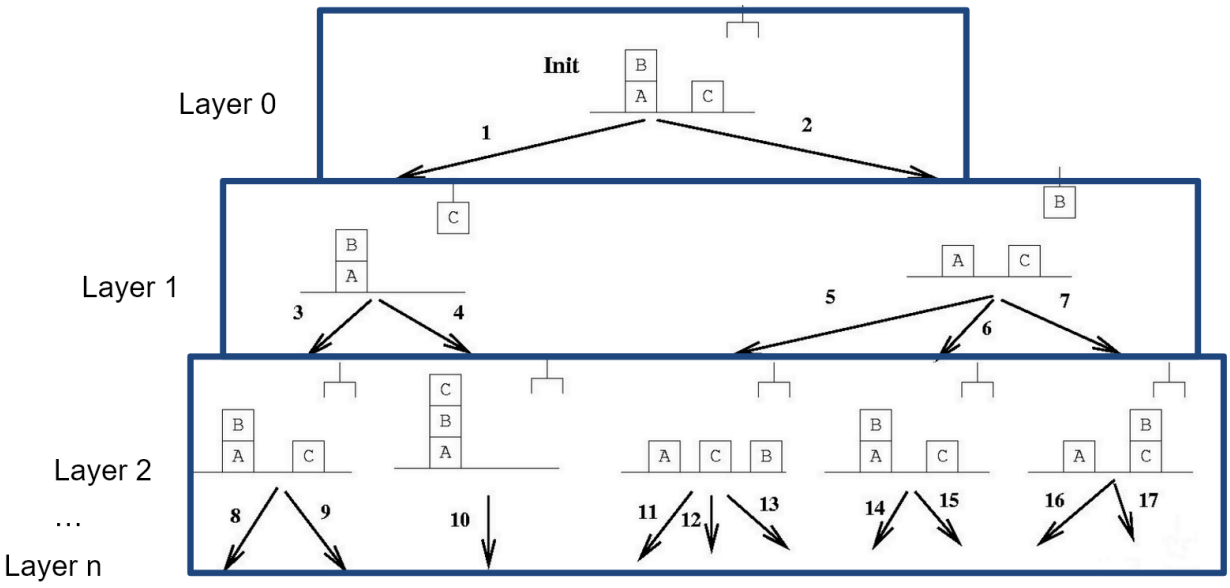


Figure 1: A visual representation of the "layered tree" search, layers shown as boxes (from https://www.cs.bham.ac.uk/~mmk/Teaching/AI/l8.html)

For this to work, the program makes two assumptions: that the goal state both exists, and is reachable from the given initial state [2]. To prevent the program from infinitely attempting to calculate states, a layer depth maximum of 2000 is enforced. Should this maximum be reached, the program reports a failure.

When the goal state is found, the path of “correct” states is traced from the goal upwards to the initial state, and placed into a data structure that can be read by the GUI.

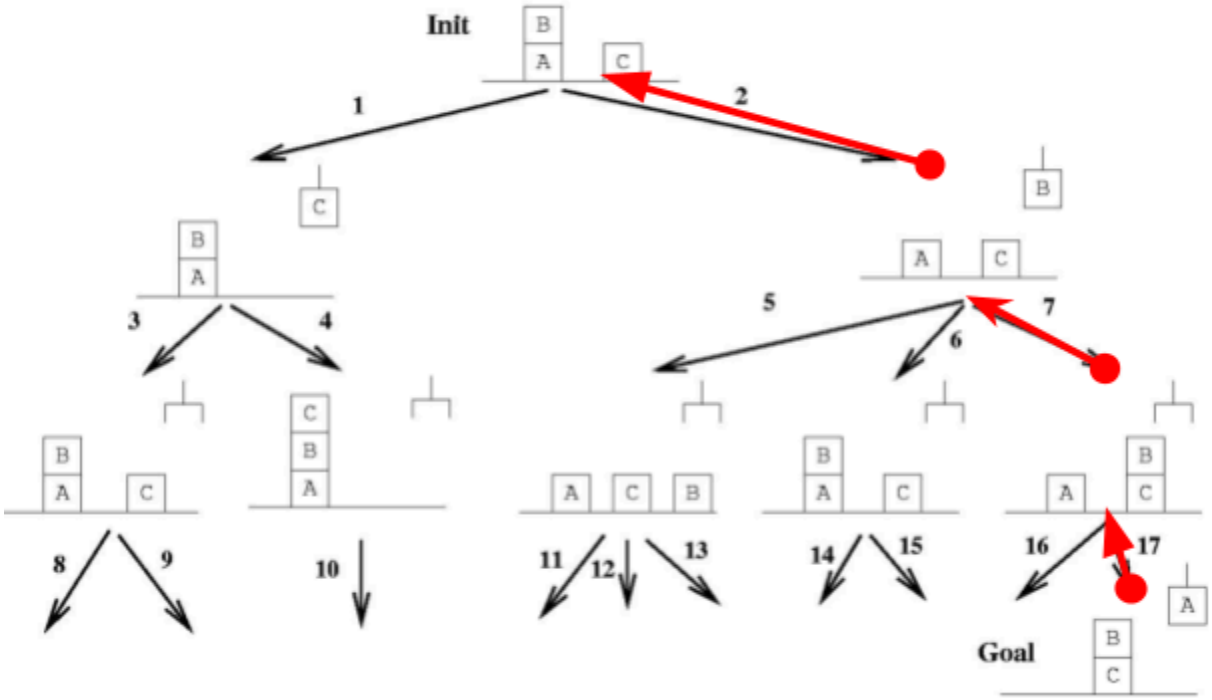


Figure 2: Tracing up from the goal state back to the initial state (from https://www.cs.bham.ac.uk/~mmk/Teaching/AI/l8.html)

This “correct” path is sent to the GUI, where each state in between the initial and final state can be viewed individually. A simple example is shown below in **Figure 3** with all states taken by the program to move block ‘a’ from spot L1 to L3.

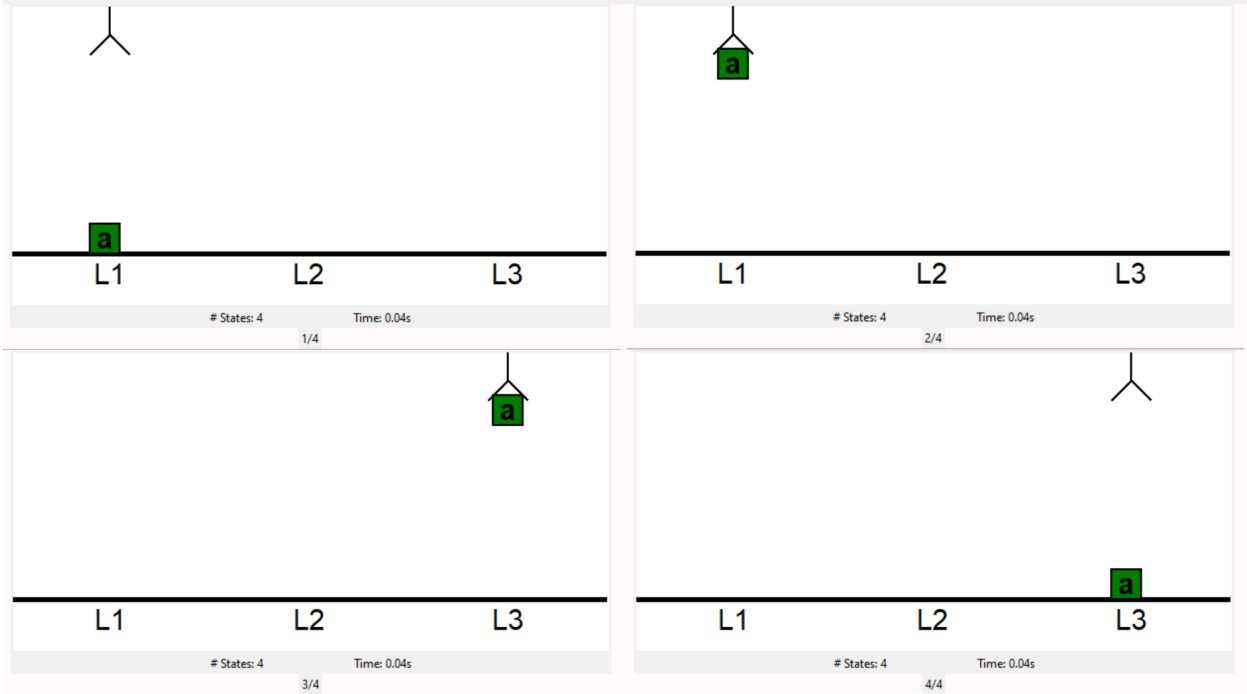


Figure 3: A simple run of the program, moving block ‘a’ from spot L1 to L3, showing every action in between (from me).

The GUI and problem “Solver” are separate entities that communicate to each other with formatted strings containing block location information. This approach was taken as to not perform large search operations on the GUI itself, that may compromise its integrity. The GUI has its own set of backend operations as well, such as command parsing and switching between visual states, so splitting up these functions keeps the codebase clean.

This data structure contains formatted strings to help the GUI draw each state to the screen, as well as some runtime information like the number of states generated, and the time taken to reach the goal state from the initial state. For example, the above run returns the following:

{'Found': True, 'States': 3, 'State_Data': ['a---1', '---1a', '---3a', '--a-3'], 'Time': 0.04}

a. Methodology Flowchart

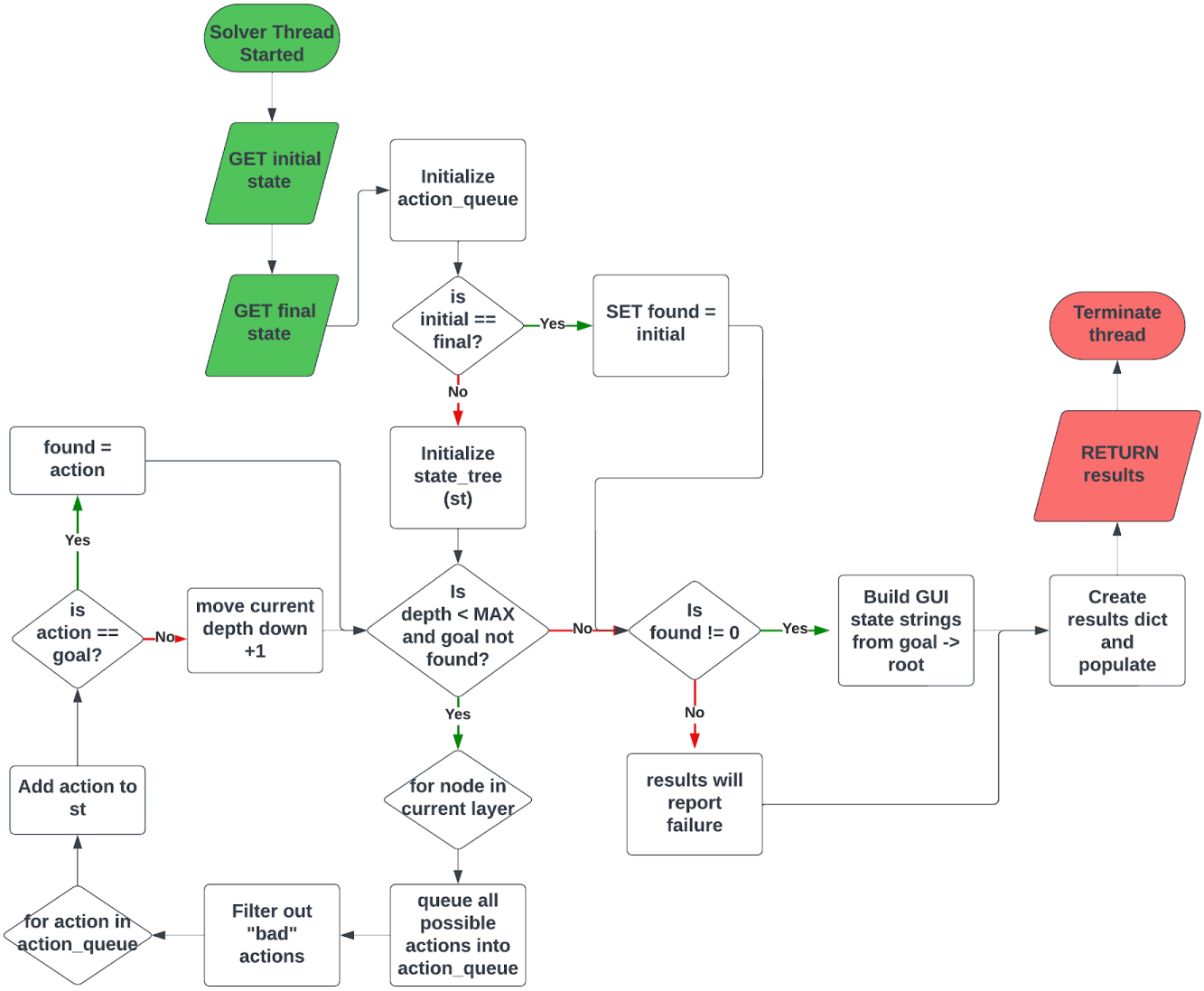


Figure 4: Program Search methodology (from me)

The general idea behind this approach is to cast a wide net of possible actions, in the hopes that the program will find its way to the goal state after enough actions are performed.

To reduce some redundancy, three filters were applied to the action queue that removed “bad” actions before they could be added to the tree. First, actions that result in an identical state to the previous state (the current state’s parent) are filtered. For

example, if we create a state from the action U(a, b, L1), then the subsequent action of S(a, b, L1) is removed from the list of possible actions. Additionally (two new additions since the in-person presentation) no two movement actions may be taken in a row without another type of action having occurred in-between and no state may be identical to any state that came before. For example a state created from the action M(L1, L2) cannot then perform M(L2, L3), as it can be assumed M(L1, L3) was accounted for on the previous layer, thus removing redundant actions.

Some of the sources referenced below covered other optimization techniques that could further improve the program, such as Bi-Directional Searching [4] [5]. This technique involves searching the states from the initial state and goal state simultaneously, searching for a match in the actions created in between. In pure theory, this could cut the complexity of a search in half, as less overall layers are evaluated.

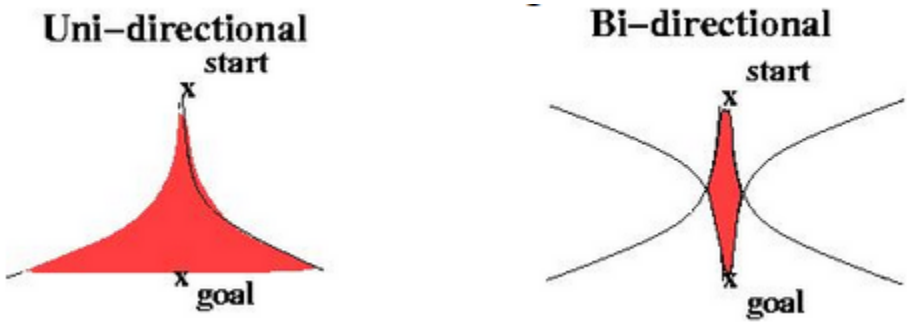


Figure 5: Complexity metric of uni-directional searching (current) vs bi-directional searching (from https://www.cs.bham.ac.uk/~mmk/Teaching/AI/l8.html)

IV. Conclusions

While various techniques from class were used to create the “Breadth-First” searching program (Green’s Method, State-plan-execute model, etc) [2] and the logic is sound, optimizations would help to improve program run-time and reduce the overall clunky feeling of the program. Thus, small-scale experiments were used to demonstrate the verbosity of the program in a shorter amount of time. The program, in theory, will work with any number of blocks given, in enough time.

References

[1] "Blocks World," Wikipedia, https://en.wikipedia.org/wiki/Blocks_world (accessed Nov. 30, 2023).

[2] M. R. Genesereth and N. J. Nilsson, Logical Foundations of Artificial Intelligence. Morgan Kaufmann, 1987.

[3] A. Dixit, "Goal stack planning for blocks world problem," Medium, https://apoorvdixit619.medium.com/goal-stack-planning-for-blocks-world-problem-41779d090f29 (accessed Nov. 27, 2023).

[4] "The blocks world," Introduction to AI - Week 8, https://www.cs.bham.ac.uk/~mmk/Teaching/AI/l8.html (accessed Nov. 27, 2023).

[5] "Search algorithms in ai," GeeksforGeeks, https://www.geeksforgeeks.org/search-algorithms-in-ai/ (accessed Nov. 27, 2023).

[6] "Tkinter Documentation," TkDocs Home, https://tkdocs.com/index.html (accessed Dec. 1, 2023).