# Sokomind Analysis Report

Bradley Hampton

CS 580

Professor Yaohang Li

7 October 2024

## 1 Performance Statistics

Tables 1-3 illustrate the performance of each algorithm and heuristic against each provided benchmark puzzle. For each, the program was run until either a solution was generated, there was determined to be no solution, or until 20 minutes were exceeded.

| Algorithm | Heuristic | Moves | Execution Time | Visited States |
|---|---|---|---|---|
| Depth-First Search | | 27 | 89 ms | 500 |
| Breadth-First Search | | 21 | 70 ms | 395 |
| Greedy | Manhattan | 21 | 5 ms | 25 |
| Greedy | Deadzones | 21 | 21 ms | 33 |
| A-Star | Manhattan | 21 | 57 ms | 246 |
| A-Star | Deadzones | 21 | 145 ms | 91 |

*Table 1: Performance Statistics against the Tiny benchmark puzzle.*

When running the program against the Medium benchmark, some of the algorithms began to struggle. In Table 2, any algorithms that were unable to produce results are indicated by marking them with “exceeded” in the Execution Time column.

| Algorithm | Heuristic | Moves | Execution Time | Visited States |
|---|---|---|---|---|
| Depth-First Search | | - | Exceeded | - |
| Breadth-First Search | | - | Exceeded | - |
| Greedy | Manhattan | - | Exceeded | - |
| Greedy | Deadzones | 50 | 2859 ms | 1379 |
| A-Star | Manhattan | 34 | 8 min, 1 sec | 671918 |
| A-Star | Deadzones | 34 | 4 min, 22 sec | 58947 |

*Table 2: Performance Statistics against the Medium benchmark puzzle.*

As shown in Table 3, only a Greedy algorithm using the Deadzone Heuristic was able to produce a solution within the allotted time-span. Information regarding the general intuition and implementation of Deadzones can be found in the either the README.md file included with the source code but is also repeated in Section 2.

| Algorithm | Heuristic | Moves | Execution Time | Visited States |
|---|---|---|---|---|
| Depth-First Search | | - | Exceeded | - |
| Breadth-First Search | | - | Exceeded | - |
| Greedy | Manhattan | - | Exceeded | - |
| Greedy | Deadzones | 220 | 10 min, 16 sec | 318192 |
| A-Star | Manhattan | - | Exceeded | - |
| A-Star | Deadzones | - | Exceeded | - |

*Table 3: Performance Statistics against the Large benchmark puzzle.*

## 2 Deadzones Heuristic

The information shared in this Section is a duplicate of what is already included in the README.md file. I am including it again here for convenience.

Starting from the same Heatmap generated by the Heatmap Heuristic, we perform some more in-depth analysis to decide whether a state is unsolvable or not. If a state is determined to be unsolvable, then the Heuristic will return -1, which instructs the algorithm to prune this state.

The following describes the methods the Heuristic uses to decide whether the state is unsolvable, in the order they are evaluated.

### 2.1 Immovable Blocks

A block can be considered immovable if they are adjacent to at least one obstacle or another immovable block on each axis.

Immovable blocks are evaluated recursively. If any immovable block is not already on its designated storage, then we return -1. This should detect blocks that are nestled into corners.

### 2.2 Reachable Storages

For a storage to receive a block, its two adjacent spaces in any direction must be either open or moveable.

For example:

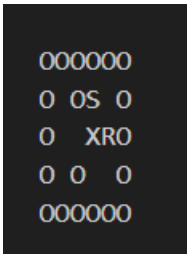
```
OOOOOOOO
OSO R  O  It is impossible for S to recieve its block, because for every 2 spaces in any direction, there are
O   X  O  immovable things (in this case, obstacles, but could be immovable blocks as defined in method 1) in the way.
OOOOOOOO
```

If there are any storages for which it will be impossible for it to receive their block for this reason, we return -1.

### 2.3 Stuck Blocks

After a block is pushed into a position, from where the robot cannot come around to do anything useful.

For example:




If the robot pushes the block to the left, then it is not considered a "immovable block" based on the criteria discussed in method 1. However, we can see that this block would desperately want to be pushed to the right to reach its storage, but the robot has no way of coming around to appear on the block's left.

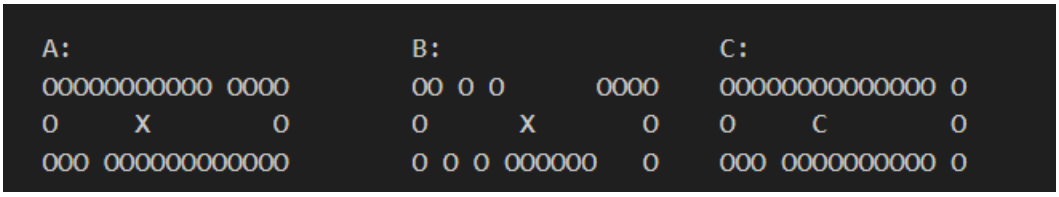
After a block is pushed, we perform a bfs on its 3 adjacent spaces (not including the one the robot is currently standing on). From these positions, we check to see if we are able to either reach the robot, or reach the block's designated storage. If neither of these are True, we return -1.

### 2.4 "Dead Zones"

Blocks may sometimes end up in a segment of a row or a column from which they can never escape. When this row/column does not contain their designated storage, this is called a deadzone. If any block is trapped in a deadzone, then we return -1.

An enclosed space like this can appear between any 2 obstacles, where every cell in between is adjacent to at least one obstacle in at a perpendicular direction.

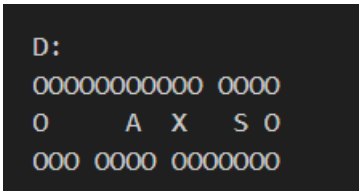
Consider the following grids:




- Grid A is an enclosed space, X cannot be pushed out of the 2nd row, because it is between 2 obstacles on its far left and far right, and every cell in between its far left and far right either has an obstacle above or below it.

- Grid B is also an enclosed space, even though there are large gaps in the 1st and 3rd row, the same rules hold so X still cannot escape the 2nd row.
- Grid C is not an enclosed space, the block X *can* escape the 2nd row, because there is one position where the cell above and below do are open.

If there is a matching Storage enclosed within the enclosed space, then this is not considered a deadzone.




- Grid D is not considered a deadzone for X, because a storage S is also enclosed within the space. Grid D is considered a deadzone for A, because there is no 'a' storage enclosed.
- Grids A and B are deadzones because they do not enclose a storage S.

A block can be in multiple enclosed spaces simultaneously, but this is only possible if it is in a corner. If a block is in a corner, the only way for this state to be valid it is directly standing on a storage.

This Heuristic does make many assumptions about what can make a state unsolvable. There could be some edge-cases that I have not accounted for in my implementation of this Heuristic. More details about the performance of this Heuristic will be discussed in Section 4.2.

## 3 Algorithm Analysis

The following section analyzes the various algorithms shared implemented in the program, along with my explanation on why I believe I have obtained the results that I have reported for each benchmark.

### 3.1 Depth-First Search

Depth-First Search obtains the worst performance of all the algorithms tested, it is not guaranteed to find the best solution, and may waste execution time traversing down incorrect paths, especially when it comes across an unsolvable state with a wide variety of legal moves.

Depth-First Search has the potential to obtain solutions faster than some algorithms, if it happens to traverse down the right path. If the execution of dfs continued past the allotted 20 minutes, I might expect the algorithm to find a solution faster than Breadth-First search, if the order successor states are generated and put onto the search-stack were convenient. There is also

the possibility that the correct solution sequence might be one of the last states that Depth-First Search investigates. Traditionally, Depth-First Search uses much less memory than Breadth-First Search, since we are not generating every possible successor where our move count equals 23 if we are to find the solution, which has a move count of 24. In an average case we use a fraction of that memory.

### 3.2 Breadth-First-Search

Breadth-First Search is guaranteed to both produce the best solution and, in the best case, use the most memory out of all the algorithms. Breadth-First Search scales incredibly poorly. For example, if the optimal solution involves 34 mores in its solution path, as is required to solve the Medium benchmark, we will have to check every 33 move sequence first. This amounts to approximately $4^{49}$, or $7.3787 \times 10^{19}$ states that must be visited, without considering states that are identical and are thus are skipped.

### 3.3 Greedy

The Greedy algorithm generally produces solutions with the fastest execution times, while also visiting the least number of states, depending on the Heuristic. Greedy algorithms are not guaranteed to find the best solution, however. When executing the Medium benchmark, the Greedy Algorithm with the Deadzones Heuristic produced a solution in 50 moves, whereas the A-Star Algorithm produced solutions in only 34 moves.

There is also a risk, like in Depth-First Search, that a Greedy Algorithm may spend much of its execution time traversing down a solution path on a state that is already rendered unsolvable. I suspect this is why I was not seeing results for the Manhattan Heuristic within the allotted time. Generally, I find the Greedy Algorithm to be poor at backtracking, a Heuristic that doesn't account for situations that renders the puzzle unsolvable or require the robot to take several steps going against the calculating Heuristic score can cause the number of visited states and execution times to increase exponentially.

### 3.4 A-Star

Unlike the Greedy algorithm, A-Star is guaranteed to find the optimal solution, if the Heuristic is configured such that it never prefers a longer path, all else equal. In all benchmark tests of the A-Star algorithm, it produced the shortest solution path. A-Star scales much better than Depth-First Search or Breadth-First Search, as it is still able to obtain a solution to the Medium benchmark for both tested Heuristics.

As the search space continues to scale, the execution time and memory usage of A-Star begins to increase exponentially. The best Heuristic for the Medium benchmark was only able to obtain a solution in 4 minutes. A-Star could not obtain a solution for the Large benchmark using any Heuristic.

## 4 Heuristic Analysis

The following Section compares the two Heuristics used by the Greedy and Astar algorithms.

### 4.1 Manhattan

This program naively implements Manhattan, it seems to push blocks into positions which renders the puzzle unsolvable. Since there is no detection for when this occurs, the robot tends to wander around for a while until it realizes that it has tried everything. This means that Manhattan could spend much of its execution time processing an already unsolvable state.

In smaller puzzles, this can still produce rather quick solves, if there are not too many blocks to push around. The more blocks that there are, the more permutations of positions of the remaining, movable blocks that exist—meaning it will take Manhattan much longer to realize that it is stuck. The Medium benchmark seems to especially illuminate this, since we see a total of 8 blocks in the grid, if one is stuck in a corner such that the puzzle can no longer be solved, there are a lot of ways the robot can proceed to arrange remaining blocks before having tried all possible configurations.

A-Star remedies this issue a bit, since it penalizes the Heuristic proportional to its move count, the algorithm avoids traversing down a path where the robot is wandering aimlessly. This is why we can see that Manhattan with A-Star produced a solution for the Medium benchmark, whereas Greedy did not.

### 4.2 Deadzones

Since it includes various checks to evaluate whether a state is unsolvable, the Deadzones Heuristic function has a much higher overhead cost than Manhattan. We can see this by how the Manhattan Heuristic outperforms Deadzones in the Tiny benchmark. Given that the proper solution is not particularly complex, the various checks that Deadzone performs unnecessary.

Once Deadzones is scaled to more complex problems, we see the benefits of the Heuristic function’s various checks. When combined with the Greedy algorithm, Deadzones is able to obtain a solution to the Medium benchmark in a 2859 ms and visiting only 1379 states. I find this to be the most impressive result obtained, by far.

The Greedy Algorithm with the Deadzones Heuristic was also the only test that was able to solve the Large benchmark, and I find the timeframe of 10 minutes, 16 seconds to be quite good, considering how much backtracking must occur for the solution, and how Greedy Algorithms are naturally reluctant to backtrack. I will note that since this Heuristic makes some assumptions about which states are unsolvable, there is the possibility that given a certain puzzle, the Heuristic will wrongly assume that a given state along the optimal solution path is

unsolvable; therefore, it is possible that we receive no solution from this Heuristic, or a suboptimal solution when we are using the A-Star algorithm.