Ivan Lorna 862035775

ilorn001@ucr.edu

CS170: Introduction to Artificial Intelligence

Dr. Eamonn Keogh

17 February 2021

Project 1 Report

Table of Contents

Introduction

A sliding tile puzzle is a puzzle that consists of a grid of tiles that can move vertically and horizontally. The puzzle can vary in size, but will always be of NxN shape and consist of N^2-1 tiles and one empty space. To solve this puzzle, you must slide tiles into the empty space to rearrange the pattern on the tiles to its 'solution state.'

For this project, the sliding tile puzzle we will consider is a 3x3 tile space with numerically numbered tiles. The solution state is as shown below in figure 1, as well as an example of an initial state, or 'problem state.'

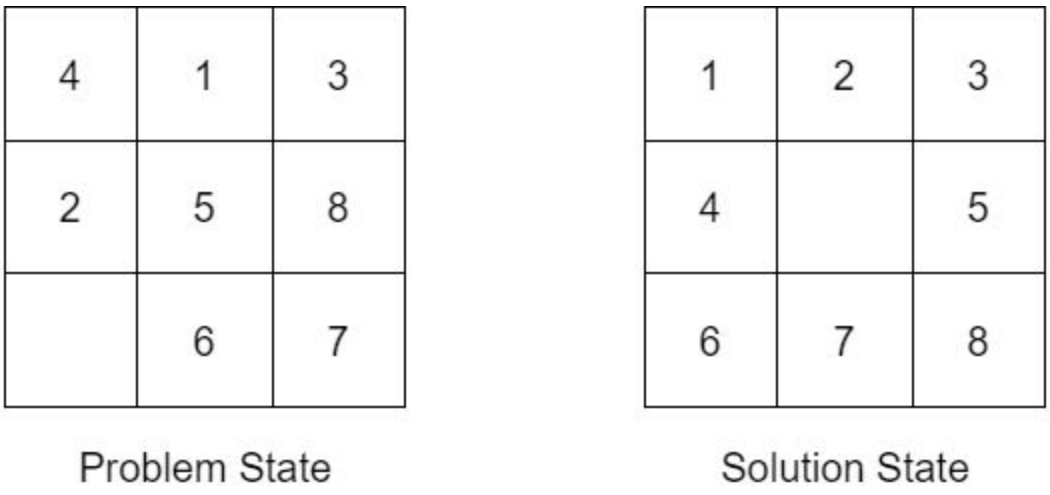


Figure 1: An example of a sliding tile puzzle's problem state and solution state.

This project is meant to explore different methods of developing A.I. This report will contain the results of a comparison of three different algorithms that all attempt to solve the sliding tile puzzle. This project was conducted using Matlab.

Comparison of Algorithms

The three algorithms that will be compared in this project are:

- Uniform Cost Search
- A* using Misplaced Tile Heuristic
- A* using Manhattan Distance Heuristic

A* is an algorithm where the queue of nodes to be evaluated is sorted to prioritize the minimization of the distance to the solution state by whichever heuristic it uses to determine that.

Uniform Cost Search

Although it is worded differently, the Uniform Cost Search algorithm is very similar to the others, as it is also an A* algorithm. It just uses path weight as its heuristic. Every time a node is expanded, its children inherit a total weight of the parent plus one. The algorithm will prefer to expand on nodes that have been expanded less. This way, it evaluates all nodes of a certain depth before beginning the evaluation of nodes of a further depth.

This algorithm searches all possible branches and paths, one depth level at a time similarly to Breadth-First Search. It ensures a solution that is optimal so long as it exists. Its downside is its runtime, because checking every depth gets expensive very quickly when every node can expand in at least two ways.

Misplaced Tile Heuristic

The Misplaced Tile heuristic measures how close a node is to the solution by the number of spaces that match the solution state. Weight is calculated as the sum of binary classifiers of each space on the grid. If the tile in the space matches the tile in the corresponding space in the solution state, it is classified 0, otherwise 1. Figure 2 shows an example of how new nodes are evaluated by the Misplaced Tile heuristic.

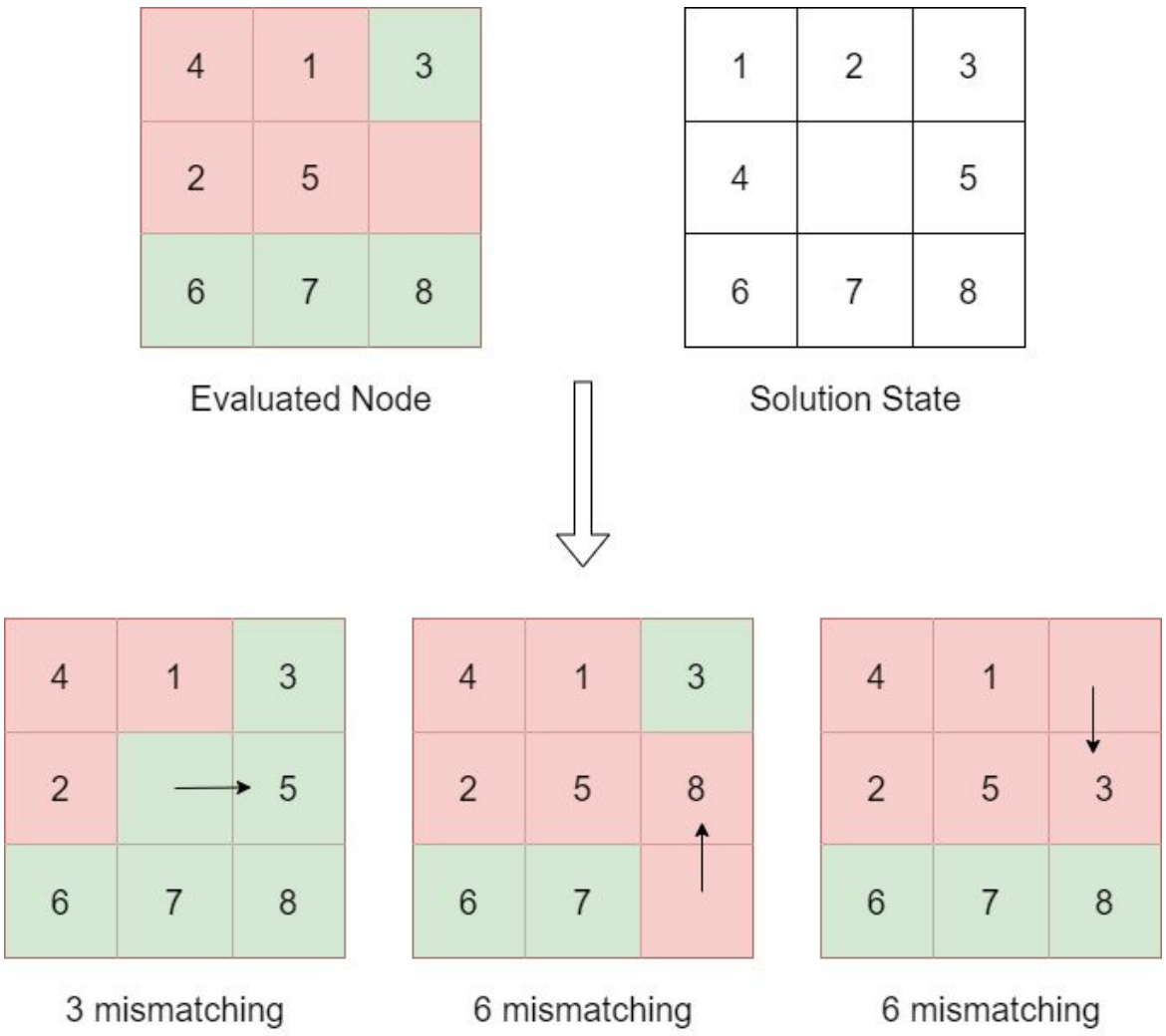


Figure 2: An example of Misplaced Tile Heuristic evaluation

When the node at the top of the queue is visited, the children are assigned a weight based on the number of mismatching tiles it has, then sorted into the queue of nodes. Nodes are evaluated in order of the least-to-greatest number of mismatches.

Manhattan Distance Heuristic

The Manhattan Distance heuristic determines how far a node is from the solution by the sum of manhattan distances of its tiles location to their location in the solution state.

Historically, a manhattan distance between two points is the sum of differences in their coordinates. You could also think of it as the sum total of steps one point needs to take in each

axis to reach the other. For this heuristic, we consider the manhattan distance to be the number of times a tile must slide to reach its desired location while unimpeded by the other tiles.

The manhattan distance of a tile at its ideal location is 0. In this way, the Manhattan Distance heuristic can be considered as an extension of the Misplaced Tile heuristic. Whereas the Misplaced Tile heuristic uses binary, the Manhattan distance heuristic classifies in a range [0,4].

The range [0,4] comes from noticing the natural upper and lower bounds of possible manhattan distances in our sliding tile puzzle. The worst place a tile could be is at the corner opposite than it should be, and this requires at least 4 slides to fix. The best place a tile could be is at its correct position, requiring 0 slides to fix.

Figure 3 below shows the expansion of a node and the evaluation of its children nodes under the Manhattan Distance heuristic.

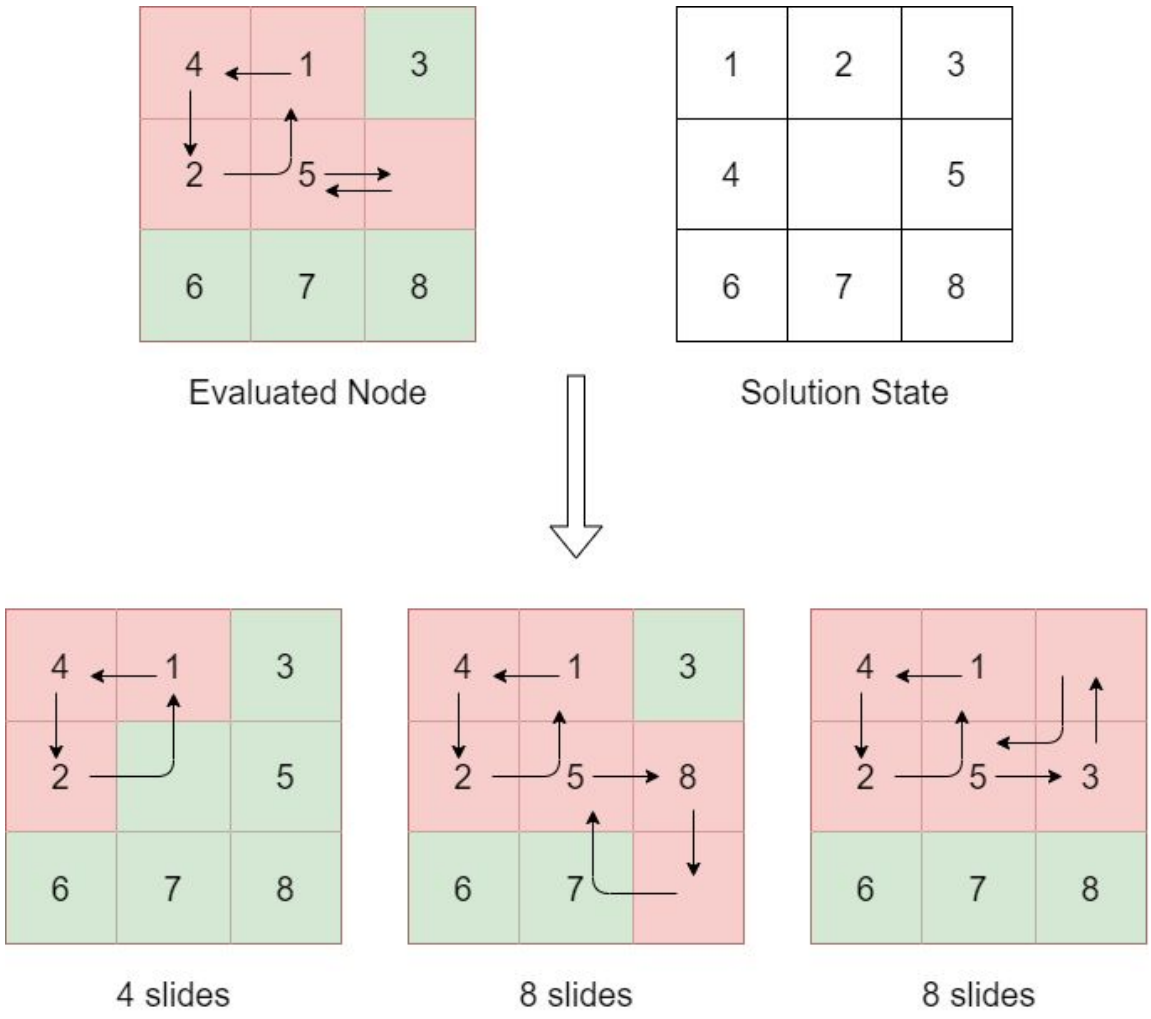
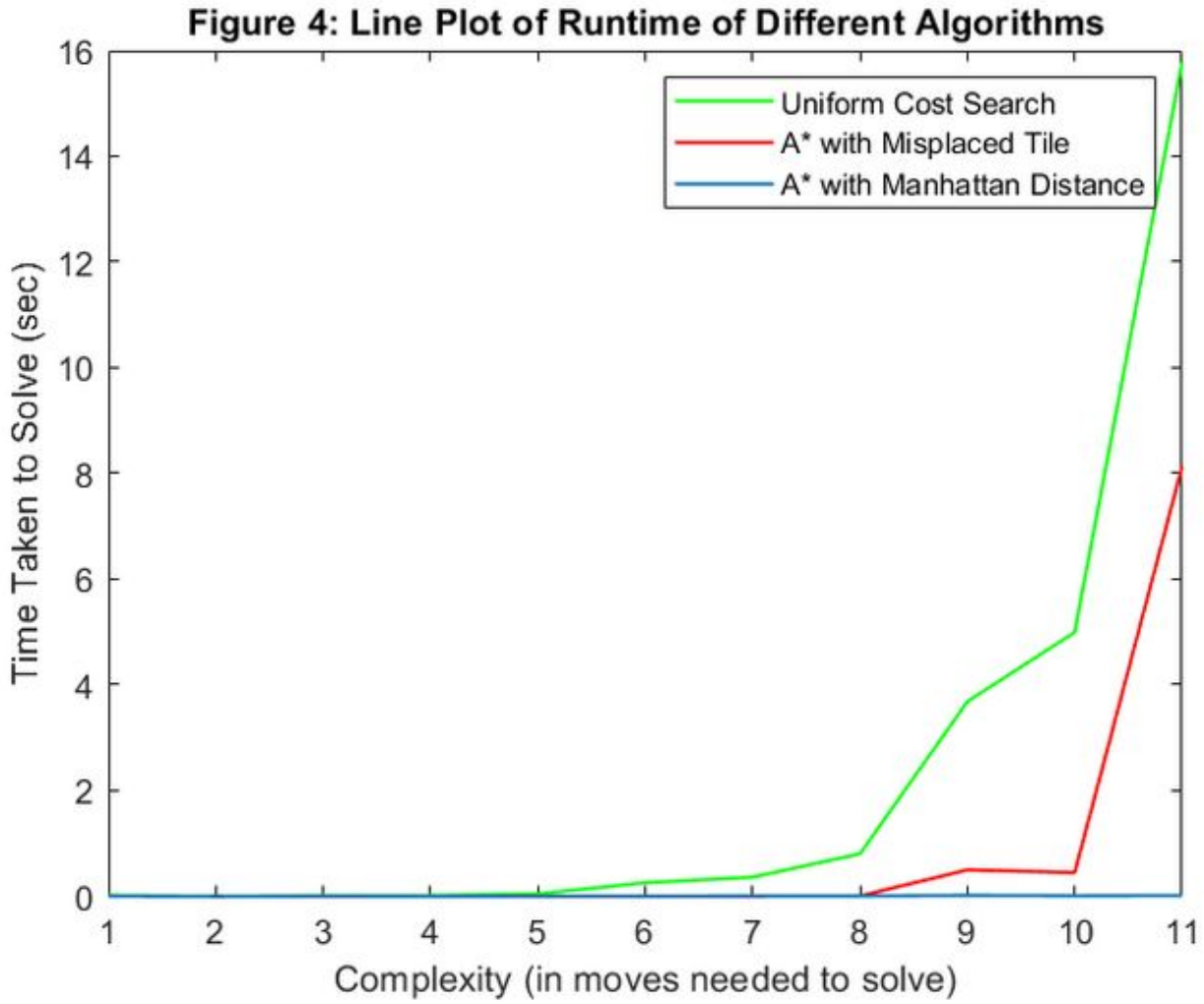


Figure 3: An example of Manhattan Distance Heuristic evaluation

Comparisons of Algorithms on Sample Puzzles

To compare the algorithms, we can input a series of problem states with increasing complexity, and measure each algorithm's turnaround time. The hardcoded problem states can be seen in my Matlab script provided at the end of the report. Figure 4 plots the results of each algorithm running these problem states.



The contrast in the turnaround time is immediately noticeable.

It is expected that Uniform Cost Search begins to become much more expensive at higher complexities. As mentioned before, Uniform Cost Search is similar to Breadth-First Search. It is Optimal and Complete. But the queue of nodes to evaluate expands exponentially as depth increases.

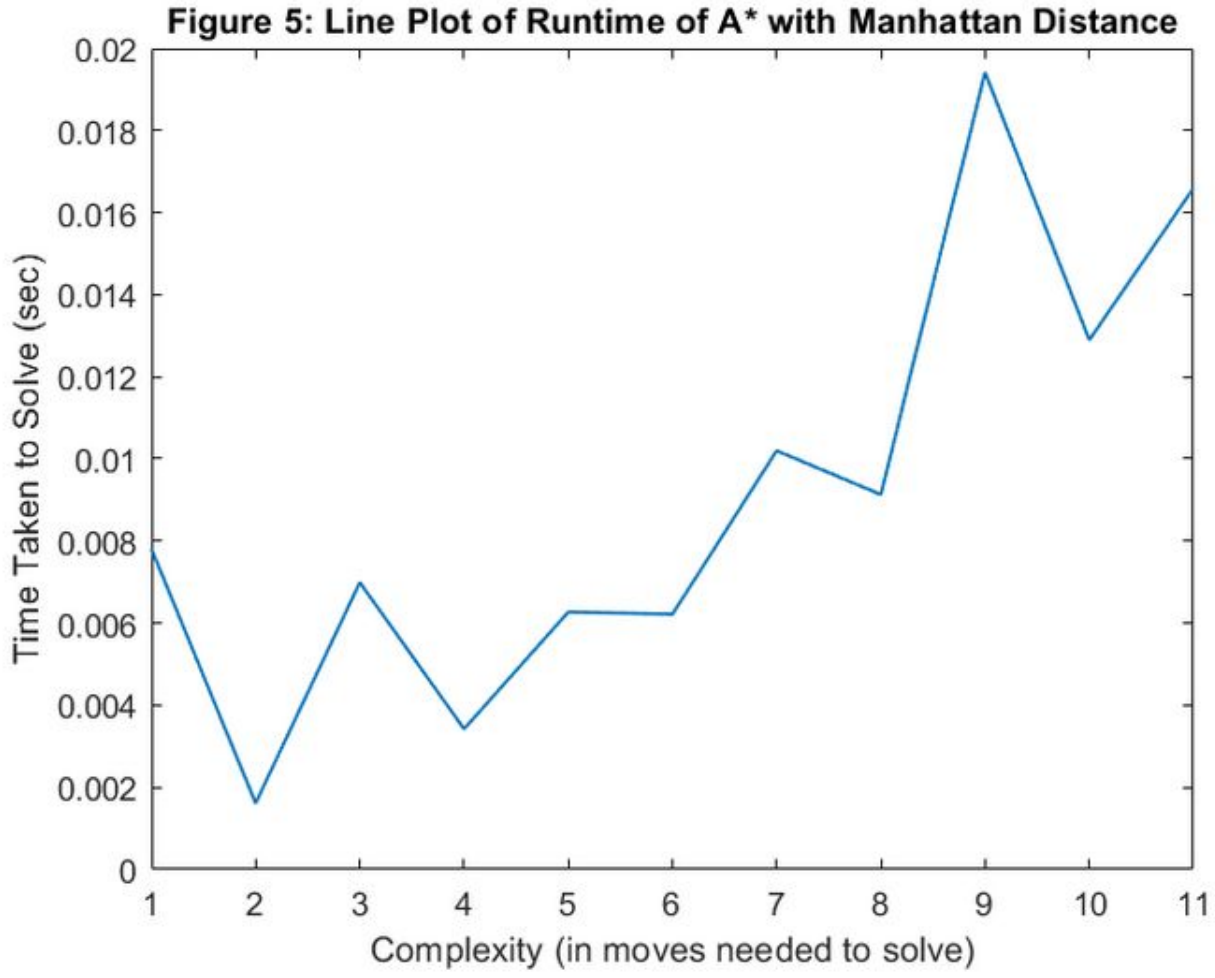
Misplaced Tile Heuristic becomes increasingly ineffective as complexity rises. This occurs because as nodes become more complex, the distance from the solution state remains

capped at 9, where all the spaces contain incorrect tiles. There is no guarantee that a node with a weight of 9 will contain children that have less weight. If the queue of nodes contains only nodes of weight 9, it randomly selects a node to expand. For sufficiently complex problems, it becomes random as to how much time this algorithm will take evaluating nodes with a weight of 9 before it finds one of less weight, and it takes much longer to find a node of less weight that will lead to the solution state.

A list of 'visited' states was implemented to reach a solution in a significantly smaller amount of time, but the algorithm's inefficiency at complexity 9 and higher is still clear.

The Manhattan Distance heuristic can be considered an extension of the Misplaced Tile heuristic, but it does not experience this issue. The range of its weight values is significantly greater. Weights in the Manhattan Distance heuristic range from [0,24] as opposed to the Misplaced Tile heuristic's range of [0,9]. Measuring weight with the Manhattan Distance heuristic allows the algorithm to more accurately classify nodes and continue to move towards the solution state.

Figure 4 does not really show the growth of the Manhattan Distance heuristic's turnaround time. Figure 5 is provided below as a plot solely of the Manhattan Distance heuristic. Notice the significantly smaller scale of the time axis compared to that of Figure 4.



Example of Simple Problem

The following will be snippets of my code's generated output for a simple problem.
Note: the puzzle state is represented by the leftmost 3x3 of the 3x4 matrices presented, the other 3x1 holds extraneous information for computation (x/y coordinates of the blank tile, weight of node).

```
simple_problem = 3×4
     1     3     5     2
     4     0     2     2
     6     7     8     0

front_of_queue = 3×4
     1     0     5     1
     4     3     2     2
     6     7     8     6

front_of_queue = 3×4
     1     3     5     2
     4     2     0     3
     6     7     8     4

front_of_queue = 3×4
     1     3     5     2
     4     0     2     2
     6     7     8     4

front_of_queue = 3×4
     1     3     0     1
     4     2     5     3
     6     7     8     4

front_of_queue = 3×4
     1     0     3     1
     4     2     5     2
     6     7     8     2

front_of_queue = 3×4
     1     2     3     2
     4     0     5     2
     6     7     8     0

solution found
max depth: 6
nodes expanded: 6
max queue size: 12
```

Example of Complex Problem

The following will be snippets of my code's generated output for a complex problem. Note: only the first few and last few expanded nodes will be shown to save space.

```
complex_problem = 3×4
       5     8     6     3
       3     7     4     2
       2     0     1    24

front_of_queue = 3×4
       5     8     6     2
       3     0     4     2
       2     7     1    22

front_of_queue = 3×4
       5     0     6     1
       3     8     4     2
       2     7     1    22

front_of_queue = 3×4
       5     8     6     2
       0     3     4     1
       2     7     1    22

front_of_queue = 3×4
       5     8     6     2
       3     4     0     3
       2     7     1    22

front_of_queue = 3×4
       0     5     6     1
       3     8     4     1
       2     7     1    22
```

…

```
front_of_queue =
       0     2     3     1
       1     7     5     1
       4     6     8     6

front_of_queue =
       1     2     3     2
       0     7     5     1
       4     6     8     4

front_of_queue =
       1     2     3     3
       4     7     5     1
       0     6     8     4

front_of_queue =
       1     2     3     3
       4     7     5     2
       6     0     8     2

front_of_queue =
       1     2     3     2
       4     0     5     2
       6     7     8     0

solution found
max depth: 22
nodes expanded: 264
max queue size: 270
```

Code

All the code can be found on Github: https://github.com/IvanLorna/TheEightPuzzle

Conclusion

This report details my findings and observations on three different algorithms to solve the same solution. As taught in lecture, a breadth-first search becomes inefficient at high depth, and so Uniform Cost Search behaves similarly. A* using Misplaced Tile Heuristic performs better in comparison but suffers from the limited weight range. A* using Manhattan Distance Heuristic solves this constraint and performs the best overall. All implemented algorithms perform well within expectations.