

Report

Introduction

The 8-Puzzle game, shown in Figure 1, is a sliding tiles game where you want to slide the tiles in such a way as to create a certain pattern or state. Usually, a solved state would look like Figure 1, where the numbers are increasing and the blank tile is at the bottom right. With the 8-Puzzle, there are $9!$ possible combinations of the puzzle. This is equal to 362,880 possible combinations. However, half of the combinations are impossible to solve, leaving us with $9!/2$ solvable combinations, or 181,440 solvable arrangements. When solving this puzzle with computer software, instead of thinking of the moves as “moving the 8-tile to the right once”, we can think of it as moving the blank tile to the left. In other words, we can attribute all tile movements to moving the blank tile in a certain direction.



In this report, I explain my conclusions, results, and numbers. I have attached my code to the end of this document, which simulates the 8-Puzzle game written in C++. The code also contains Uniform Cost Search, A* with Misplaced Tile Heuristic, and A* with Manhattan Distance to solve the puzzle. I have also attached 4 sample problems, containing a very easy configuration, an easy configuration, a difficult configuration, and the most difficult configuration possible. It has been found that the diameter of this puzzle is 31, meaning in the most difficult configuration, it would take a minimum of 31 tile moves to solve the puzzle.

Figure 1. A visual of the 8-Puzzle [1]¹

Uniform Cost Search

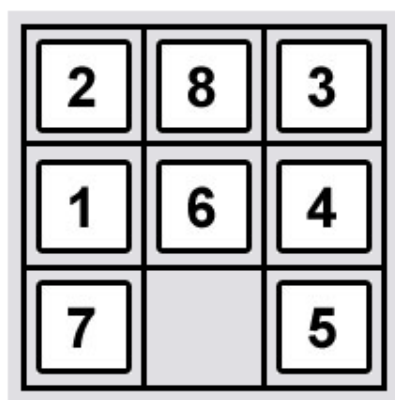
Uniform Cost Search in this puzzle is equivalent to Breadth First Search. In other words, this search for this puzzle expands every node in order of increasing depth. This search

¹ https://play.google.com/store/apps/details?id=com.faramira.puzzle&hl=en_US&gl=US

algorithm searches through nodes in the order it is given, which effectively means it searches every possible combination from the initial position. Arrangements that are a small number of moves away from the initial position are checked first, with positions that take more moves to perform to reach being checked later. This is why the most difficult puzzle, which takes a minimum of 31 moves to solve, takes a very long time with this algorithm. It is also important to note that as you perform more moves from the initial state, more possible arrangements arise. This means checking all possible combinations 10 moves away from the initial state is much quicker than checking all possible combinations 27 moves away from the initial state. For my code in for Uniform Cost Search, repeated states are not checked another time and are tracked. This is implemented due to the extremely long run times without this feature.

A* with Misplaced Tile Heuristic

The Misplaced Tile Heuristic tells the algorithm how many pieces are misplaced from the goal state, shown in Figure 1. The blank tile is not counted, meaning the value can go up to 8, where all the tiles are misplaced. The algorithm then starts by checking the arrangements with the fewest misplaced tiles and continues checking based on the number of misplaced tiles in the arrangement. For consistency purposes, this algorithm also does not check for repeated states in my code, which may influence the output.



2	8	3
1	6	4
7		5

As an example seen in Figure 2, the Misplaced Tile Heuristic for this arrangement would be 6. This is because all the tiles are misplaced except for the 3 and 7 tiles, which are in the correct position compared to the goal state.

Figure 2. Example of a configuration of the 8-Puzzle [2]²

² http://www.8puzzle.com/8_puzzle_problem.html

A* with Manhattan Distance Heuristic

The A* with the Manhattan Distance Heuristic is very similar to the A* with the Misplaced Tile Heuristic, except that the Manhattan Distance Heuristic calculates the value differently. The Manhattan Distance is the minimum amount of horizontal and vertical movements needed to reach your destination square from your current position. This applies to all misplaced tiles, and the total is summed up into a value. The algorithm then uses this value and checks the arrangements starting from the lowest value. Similar to the previous algorithms, this algorithm does not check for repeated states in my code to stay consistent.

Manhattan Distance

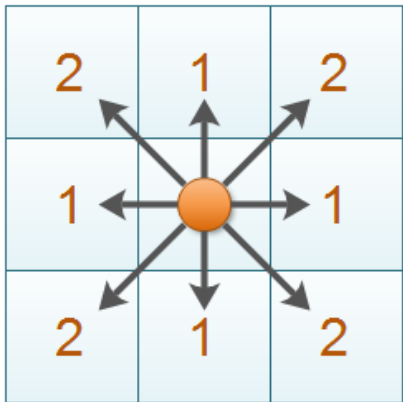


Figure 3 shows an example of the manhattan distance from the center square. For example, the minimum distance required to reach the top right square from the center square would be 2, since diagonal movements are not allowed. This process is done for every misplaced tile and added up.

Figure 3. Manhattan Distance

Example [3]³

Algorithm Comparisons with Sample Puzzles

Sample Puzzle Inputs: (9 Represents blank tile)

	[1 2 3]	[1 2 3]	[1 3 6]	[1 6 7]	[7 1 2]	[9 7 2]	[8 6 7]
	[4 5 6]	[5 9 6]	[5 9 7]	[5 9 3]	[4 8 5]	[4 6 1]	[2 5 4]
	[7 8 9]	[4 7 8]	[4 8 2]	[4 8 2]	[6 3 9]	[3 5 8]	[3 9 1]
Depths:	0	4	12	16	20	24	31

³ <https://iq.opengenus.org/manhattan-distance/>

Data:

Figure 4

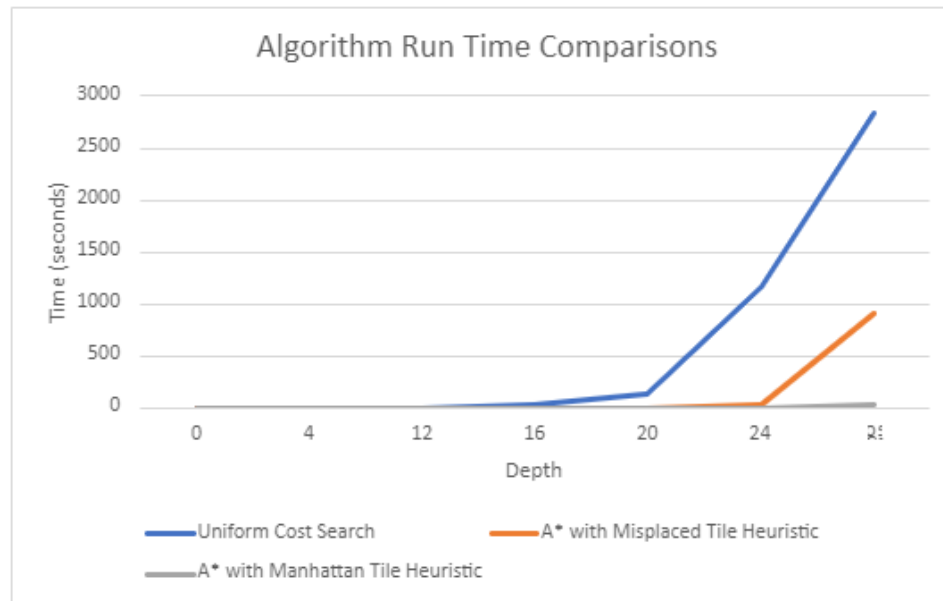
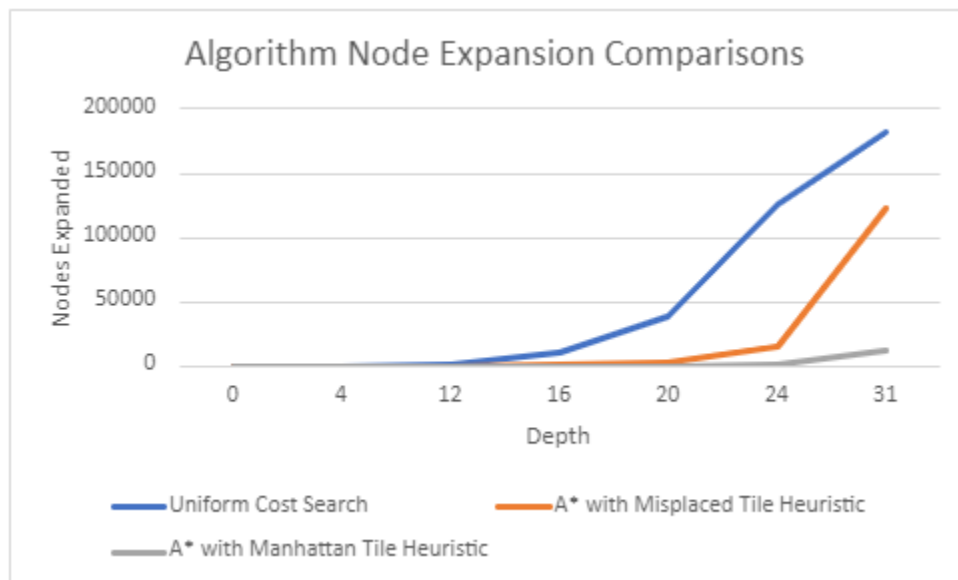


Figure 5



Conclusions

- Comparing the 3 algorithms, it is very apparent that the A* algorithms with heuristics perform much better comparatively in the majority of instances.
- When the inputted puzzle was easier, the differences in runtime and nodes checked were pretty negligible. The runtimes, such as shown in the very easy

problem example and the easy problem example, show that the difference in time is almost practically non-meaningful.

- However, when the puzzles got more difficult, the differences in runtime and nodes checked were massively increased, with the Uniform Cost Search taking multitudes of more time compared to the other 2 algorithms. This can be seen in the difficult problem example and the most difficult problem example.
- The A* with Manhattan Distance Heuristic is shown to be better compared to the A* with the Misplaced Tile Heuristic in the vast majority of cases
- Similar to the above point, the same can be said about the nodes checked when comparing A* with Manhattan Distance Heuristic and A* with the Misplaced Tile Heuristic. A* with Manhattan Distance Heuristic checks fewer nodes most of the time.
- As the puzzles get more difficult, the runtime for all algorithms, except the A* with Manhattan Value Heuristic, increases quite a bit, as can be seen in Figure 4. The rate of increase compared to all 3 algorithms is very apparent, especially as the puzzles get more difficult. The same conclusion can be said for the number of nodes checked, as seen in Figure 5.

Example with Very Easy Problem

Puzzle Input: $\begin{bmatrix} 1 & 2 & 3 \\ 5 & 9 & 6 \\ 4 & 7 & 8 \end{bmatrix}$ (9 is a blank tile)

- Solution Depth: 4

Uniform Cost Search:

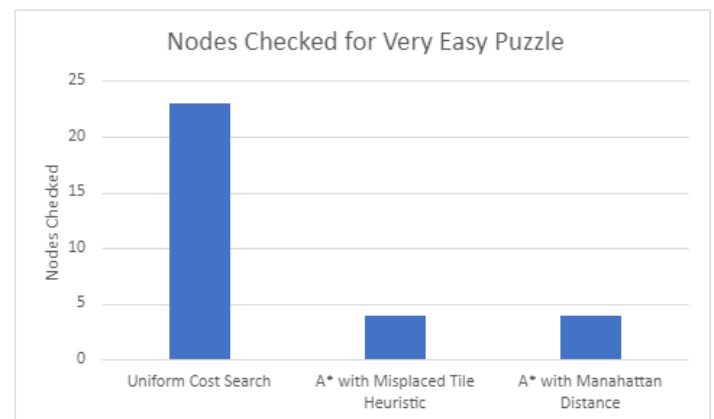
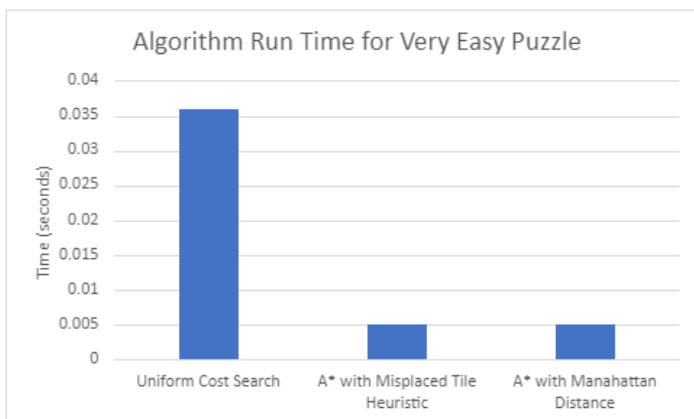
- Duration: Approximately 36 milliseconds
- Nodes Expanded: 23
- Max Queue Size: 24

A* with Misplaced Tile Heuristic:

- Duration: Approximately 5 milliseconds
- Nodes Expanded: 4
- Max Queue Size: 6

A* with Manhattan Distance Heuristic:

- Duration: Approximately 5 milliseconds
- Nodes Expanded: 4
- Max Queue Size: 6



Example with Easy Problem

Puzzle Input: $\begin{bmatrix} 1 & 3 & 6 \\ 5 & 9 & 7 \\ 4 & 8 & 2 \end{bmatrix}$ (9 is a blank tile)

- Solution Depth: 12

Uniform Cost Search:

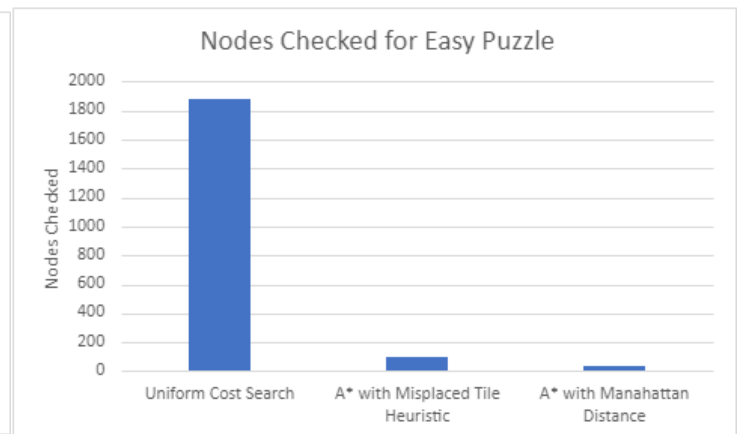
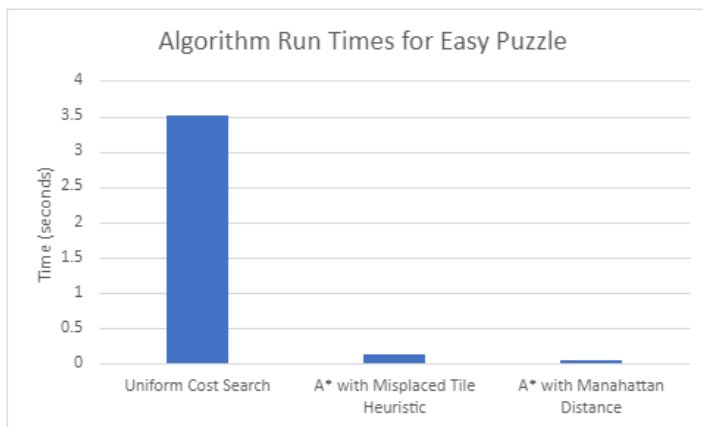
- Duration: Approximately 3.51 seconds
- Nodes Expanded: 1,885
- Max Queue Size: 1,946

A* with Misplaced Tile Heuristic:

- Duration: Approximately 0.13 seconds
- Nodes Expanded: 99
- Max Queue Size: 123

A* with Manhattan Distance Heuristic:

- Duration: Approximately 0.05 seconds
- Nodes Expanded: 34
- Max Queue Size: 44



Example with Difficult Problem

Puzzle Input: $\begin{bmatrix} 9 & 7 & 2 \\ 4 & 6 & 1 \\ 3 & 5 & 8 \end{bmatrix}$ (9 is a blank tile)

- Solution Depth: 24

Uniform Cost Search:

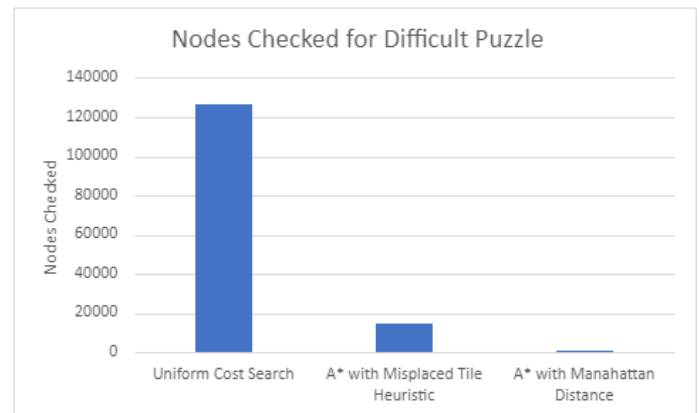
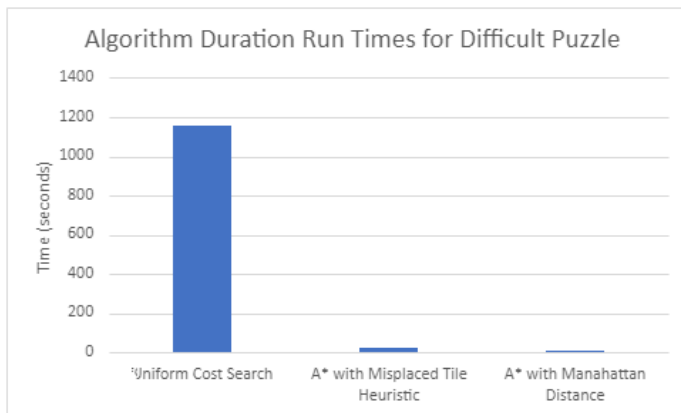
- Duration: Approximately 19.31 minutes (Approximately 1158 seconds)
- Nodes Expanded: 126,183
- Max Queue Size: 61,382

A* with Misplaced Tile Heuristic:

- Duration: Approximately 27.83 seconds
- Nodes Expanded: 14,955
- Max Queue Size: 14,172

A* with Manhattan Distance Heuristic:

- Duration: Approximately 1.18 seconds
- Nodes Expanded: 896
- Max Queue Size: 957



Example with Most Difficult Problem

Puzzle Input: $\begin{bmatrix} 8 & 6 & 7 \\ 2 & 5 & 4 \\ 3 & 9 & 1 \end{bmatrix}$ (9 is a blank tile) Note: This is 1 out of 2 most difficult states

- Solution Depth: 31

Uniform Cost Search:

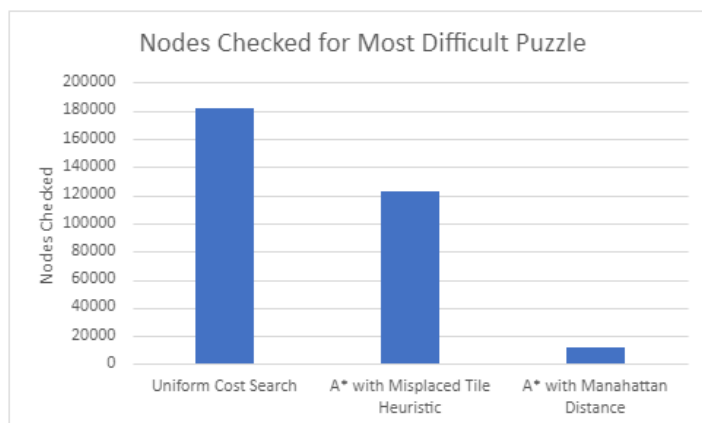
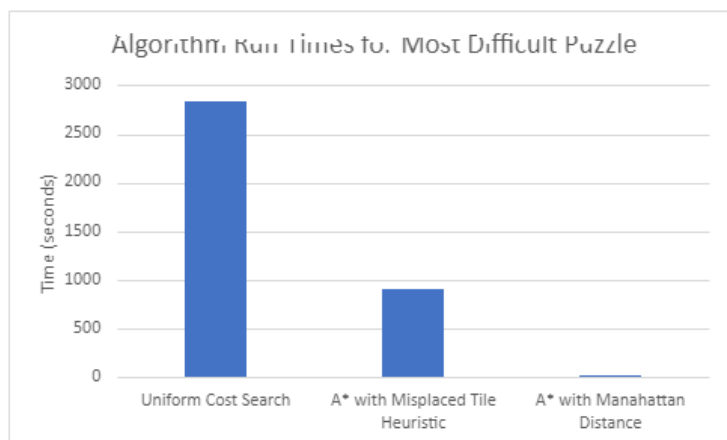
- Duration: Approximately 47.16 minutes (Approximately 2829 seconds)
- Nodes Expanded: 181,439
- Max Queue Size: 61,114

A* with Misplaced Tile Heuristic:

- Duration: Approximately 15.17 minutes (Approximately 910 seconds)
- Nodes Expanded: 122,699
- Max Queue Size: 66,570

A* with Manhattan Distance Heuristic:

- Duration: Approximately 22.49 seconds
- Nodes Expanded: 11,795
- Max Queue Size: 12,301



Example Traceback of a Very Easy Problem

```

Enter initial state, 9 for the blank tile. Valid puzzles only, numbers will be inputted from the 1st to last row.
1
2
3
4
5
6
7
8
9
Initial State
-----
1 2 3
4 5 6
7 8
Press 1 for Uniform Cost Search (Skip Repeats), 2 for A* with Misplaced Tile Heuristic, or 3 for A* with Manhattan Distance Heuristic
1
-----
1 2 3
4 5 6
7 8
Depth: 0
g(n): 0
h(n): 0
-----
1 2 3
4 5 6
7 8
Depth: 1
g(n): 1
h(n): 0
-----
1 2 3
4 5 6
7 8
Solution Found!
Depth: 1
Duration: 2 milliseconds
Nodes: 2
Max Queue Size: 3

```

```

Enter initial state, 9 for the blank tile. Valid puzzles only, numbers will be inputted from the 1st to last row.
1
2
3
4
5
6
7
8
9
Initial State
-----
1 2 3
4 5 6
7 8
Press 1 for Uniform Cost Search (Skip Repeats), 2 for A* with Misplaced Tile Heuristic, or 3 for A* with Manhattan Distance Heuristic
2
-----
1 2 3
4 5 6
7 8
Depth: 0
g(n): 0
h(n): 1
-----
1 2 3
4 5 6
7 8
Solution Found!
Depth: 1
Duration: 2 milliseconds
Nodes: 1
Max Queue Size: 1

```

```
Enter initial state, 9 for the blank tile. Valid puzzles only, numbers will be inputted from the 1st to last row.
1
2
3
4
5
6
7
8
9
Initial State
-----
1 2 3
4 5 6
7 8
Press 1 for Uniform Cost Search (Skip Repeats), 2 for A* with Misplaced Tile Heuristic, or 3 for A* with Manhattan Distance Heuristic
3
-----
1 2 3
4 5 6
7 8
Depth: 0
g(n): 0
h(n): 1
-----
1 2 3
4 5 6
7 8
Solution Found!
Depth: 1
Duration: 7 milliseconds
Nodes: 1
Max Queue Size: 1
```

Program Inputs and Outputs

Inputs: The program first asks for 9 numbers to be inputted, with the blank tile represented with a 9. The numbers are taken in one by one, each filling the first row, then the second row, and the last row from left to right. For example, an input of “1 3 2 4 5 6 7 9 8” will create a puzzle looking like $\begin{bmatrix} 1 & 3 & 2 \\ 4 & 5 & 6 \\ 7 & & 8 \end{bmatrix}$. One important point to note is that there is no validation checking for user input. In other words, invalid puzzles will not be filtered out. After the user finishes inputting the puzzle, the puzzle they have entered will appear, with the 9-tile represented with a blank tile, and the program will prompt the user to enter a certain number. An input of 1 will run the Uniform Cost Search, 2 will run A* with Misplaced Tile Heuristic Search, and 3 will run A* with Manhattan Distance Search. Any other input will cause the program to exit.

Outputs: Running any search will result in the program outputting every state the program checks, with its current depth, $g(n)$, and $h(n)$. When the program eventually finds the goal state, it will display the depth the state is in, the duration of the program runtime in milliseconds, the max queue size, and the number of nodes the program has checked. For efficiency and hardware purposes, repeated states are not checked a second time, which may produce a lower checked node count than expected for all searches. For any of these searches, a -1 depth indicates no solution.