CS 205: Artificial Intelligence
Project 1: The Eight Puzzle, Dr. Eamonn Keogh

Name: Milan Maragiri
SID: 862548940
NetID: mmara031

In completing this assignment I consulted:

- The Blind Search and Heuristic Search lecture slides and notes annotated from lecture.
- C++ Standard Library documentation: https://en.cppreference.com/w/
- For generating and testing 8-puzzle instances: https://deniz.co/8-puzzle-solver/

All important code is original. Unimportant subroutines that are not completely original are:

- Standard utility functions from the C++ STL such as priority_queue for managing the open list in A* search.
- Functions from <algorithm> for common operations like find, swap, and copy.
- #include <chrono> used for timing execution, utilizing std::chrono::high_resolution_clock to measure runtime duration
- #include <mach/mach.h> is used specifically on macOS to measure memory usage, using “task_basic_info” via the Mach API to query the resident memory size of the running process. This method is **not portable** and only works on macOS. If the program needs to be run on a linux machine, then the corresponding code to measure memory usage must be commented out.

Outline of this report:

CS205 Assignment 1: The eight-puzzle

Milan Maragiri, SID 862548940, May-07-2025

Introduction

The 8-puzzle is a classic problem in artificial intelligence involving a 3x3 grid of tiles, where eight numbered tiles are arranged with one blank space. It is a smaller version of the well-known 15-puzzle. The objective is to reach a specific goal configuration by sliding tiles into the blank space, one move at a time. The tiles are numbered from 1 to 8 so that each tile can be uniquely identified. Though the puzzle appears simple at first glance, finding an optimal solution path can become computationally challenging as depth increases, making it a useful benchmark for evaluating search algorithms.

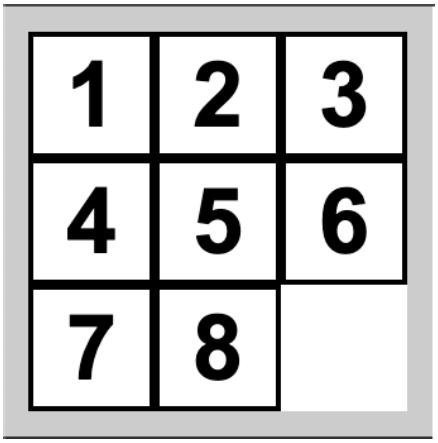

Figure 1. A solved 8-puzzle game [1]

In this project, we were asked by Dr.Keogh to write a program in any language that would solve this puzzle. I used C++ to write a program that solves the 8-puzzle using 3 algorithms: Uniform Cost Search, A* with the Misplaced Tile heuristic, and A* with the Manhattan Distance heuristic.

I compared their performance using time taken, memory used and nodes expanded as comparison metrics for solution depths ranging from 0 to 31. The program starts with an initial puzzle configuration and searches for the goal state using valid moves under the selected strategy. The code and the results for the same are attached in the report below. The program also supports giving a custom goal state as user input. For the structure of the report, I have followed the sample report structure provided in the “project handout”. [2]

Algorithms Explained

Each of the algorithm used is explained below

1.) Uniform Cost Search

Uniform cost search is a search algorithm that considers only the cost of traversing the search tree. Because there are no weights to the operators, each operator will have a cost of 1. In other

words, the heuristic function is hardcoded to 0 and the evaluation function becomes strictly the cost function

Evaluation function f(n) = g(n) + h(n)
h(n) = 0
Hence, f(n) = g(n)

2.) A* Search Using The Misplaced Tile Heuristic

In this algorithm, the evaluation function must consider the misplaced tile heuristic which is the number of tiles that are not in the position they would have been had it been the goal state. For example, consider the initial state as shown in Figure 2.

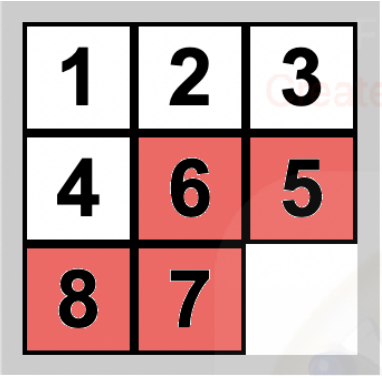

Figure 2. Randomized 8-puzzle where misplaced tiles are highlighted in red [2]

The tiles highlighted in red (i.e. the tiles 6 5 8 and 7), are misplaced from their position in the goal state. Hence, the initial configuration shown in Figure 2. would have a heuristic value of 4. We use this heuristic value that counts the number of misplaced tiles as our heuristic function along with the cost as our evaluation function and continue with the A* search algorithm.

f(n) = g(n) + h(n)
Where h(n) = Number of misplaced tiles in the current state.

3.) A* Search Using The Manhattan Distance Heuristic

In this algorithm the evaluation function must consider the Manhattan distance heuristic which is similar to the misplaced tile heuristic but also considers the cost to move to the position required by the goal state. For example, in Figure 2., all the tiles are 1 tile away from each other i.e. they need to move one position to reach their respective goal state. Hence, the heuristic value would be 4 here (Each tile requires 1 move and there are 4 misplaced tiles).
We use this heuristic that counts the number of tiles not in their goal positions. This value is combined with the path cost to form the evaluation function and continue with A* search

f(n) = g(n) + h(n)
Where h(n) = total cost to move all misplaced tiles to their goal state position

Algorithm Execution and Metric Collection

To compare the efficiency of each algorithm, I used the test cases given by Dr. Keogh's handout [2] that is sorted by depth as shown in Figure 3. In addition, I found the hardest test case for the 8 puzzle from Jason Wolfe's blog [3]

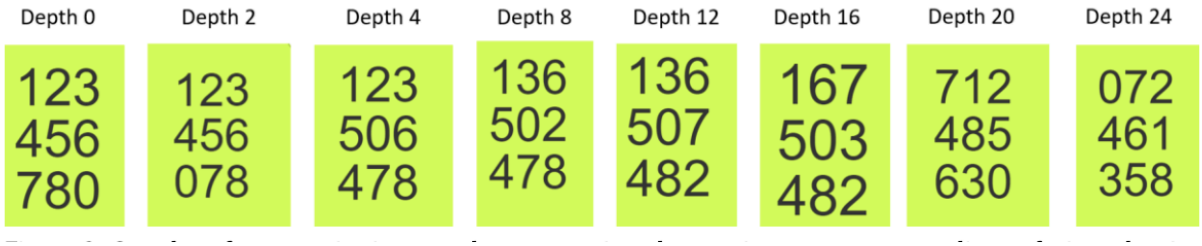

Figure 3. 8-puzzle test cases as mentioned in Dr. Keogh's project handout

These test cases were run on my implementation of the 8-puzzle solver using the 3 algorithms and the metrics time, nodes expanded, were captured. These metrics are show in the tables below

Nodes expanded vs solution depth, three lines for 3 heuristics
Time taken vs depth, three lines for 3 heuristics
Memory vs depth, three lines for 3 heuristics

Uniform Cost Search

Solution Depth	Time taken in ms	Nodes Expanded	Max queue size	Memory used in bytes	Test case
0	0	1	1	81920	1 2 3 4 5 6 7 8 0
2	0	6	6	81920	1 2 3 4 5 6 0 7 8
4	1	23	18	147456	1 2 3 5 0 6 4 7 8
8	9	297	180	262144	1 3 6 5 0 2 4 7 8
12	42	1886	1133	1032192	1 3 6 5 0 7 4 8 2
16	287	13443	6596	5259264	1 6 7 5 0 3 4 8 2
20	1076	52841	17894	18071552	7 1 2 4 8 5 6 3 0
24	2470	119600	24188	36290560	0 7 2 4 6 1 3 5 8
31	3732	181440	25135	47726592	8 6 7 2 5 4 3 0 1

A* with misplaced tile heuristic

Solution Depth	Time taken in ms	Nodes Expanded	Max queue size	Memory used in bytes	Test case
0	0	1	1	81920	1 2 3 4 5 6 7 8 0
2	0	3	3	180224	1 2 3 4 5 6 0 7 8
4	0	5	6	81920	1 2 3 5 0 6 4 7 8
8	1	19	16	114688	1 3 6 5 0 2 4 7 8
12	4	120	84	196608	1 3 6 5 0 7 4 8 2
16	24	659	413	524288	1 6 7 5 0 3 4 8 2
20	57	2705	1611	1376256	7 1 2 4 8 5 6 3 0
24	268	13625	7039	5488640	0 7 2 4 6 1 3 5 8
31	2553	124538	24713	28884992	8 6 7 2 5 4 3 0 1

A* with manhattan distance heuristic

Solution Depth	Time taken in ms	Nodes Expanded	Max queue size	Memory used in bytes	Test case
0	0	1	1	81920	1 2 3 4 5 6 7 8 0
2	0	3	3	81920	1 2 3 4 5 6 0 7 8
4	0	5	6	49152	1 2 3 5 0 6 4 7 8
8	0	13	12	114688	1 3 6 5 0 2 4 7 8
12	1	37	28	147456	1 3 6 5 0 7 4 8 2
16	7	181	114	163840	1 6 7 5 0 3 4 8 2
20	12	375	239	311296	7 1 2 4 8 5 6 3 0
24	42	1778	1007	1064960	0 7 2 4 6 1 3 5 8
31	129	6863	3634	3178496	8 6 7 2 5 4 3 0 1

Comparing The Algorithms

The data obtained from executing the code was used to plot graphs comparing the algorithms using different metrics. I have included four graphs which I made on “Plotly Chart Studio” [4] comparing each of the metrics.

Comparing Execution Time

Figure 4. Shows the time taken at each solution depth by the 3 algorithms. It can be seen that after a solution depth of 10, the uniform cost search starts to converge and take significantly longer to complete than the A* algorithm. Within the A*, the manhattan heuristic appears to be significantly faster than the misplaced tile heuristic.

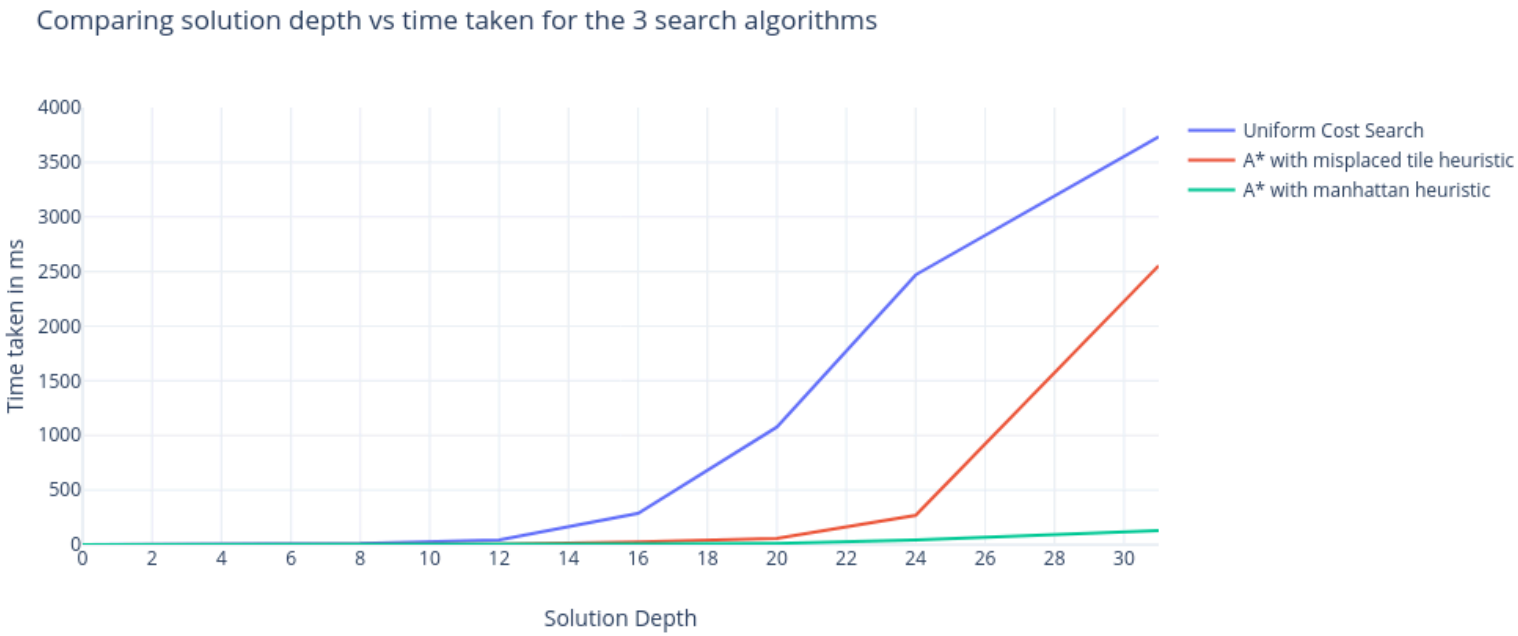

Figure 4. Depth vs Time taken in ms for the 3 algorithms

A* with manhattan heuristic is the fastest, taking a significantly smaller amount of time compared to the rest. This is followed by the A* with misplaced tile heuristic and then the uniform cost search which is the slowest.

Comparing Nodes Expanded

Figure 5. Shows the number of nodes expanded at each solution depth by the 3 algorithms. Similar to Figure 4. It is evident that uniform cost search expands the search tree a lot more than the A*. Within the A* algorithm, the misplaced tile heuristic expands a greater number of nodes than the manhattan distance heuristic to find the same solution.

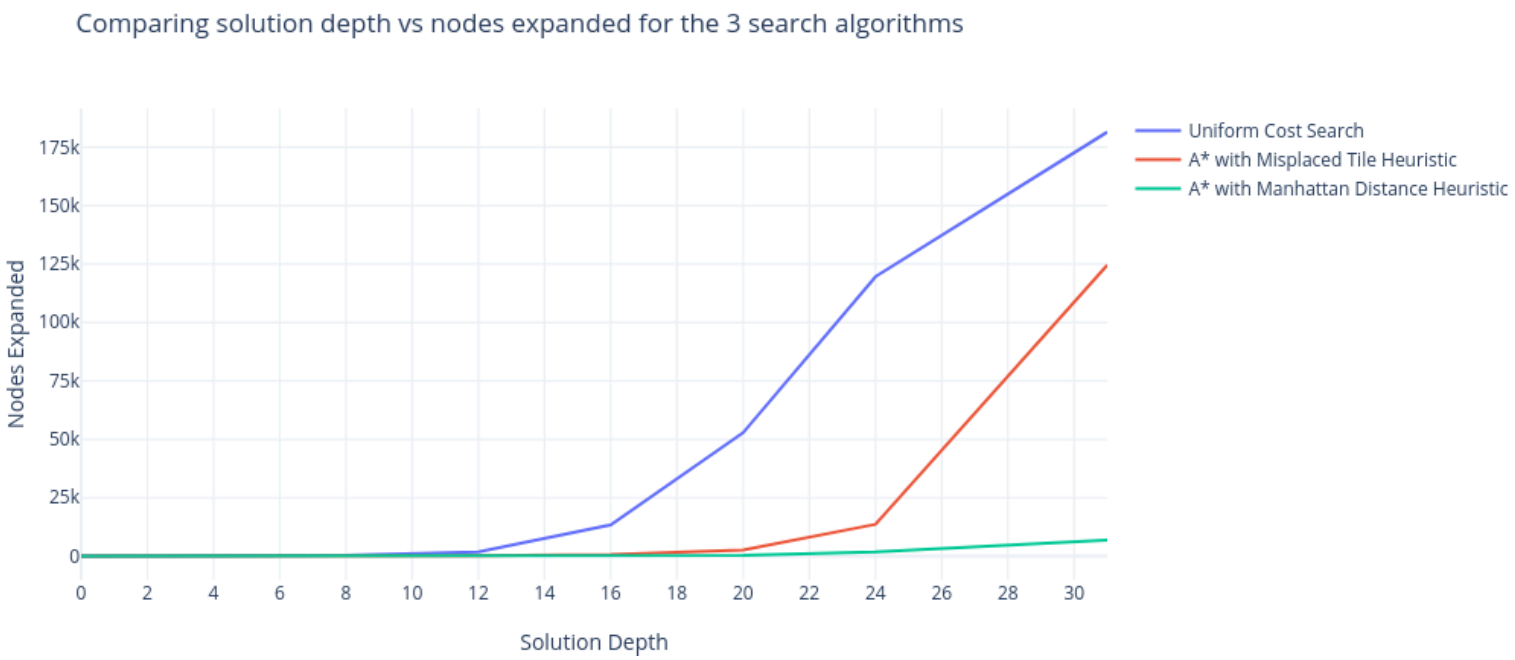

Figure 5. Solution depth vs nodes expanded for the 3 algorithms

A* with manhattan heuristic expands the least number of nodes followed by the A* with misplaced tile heuristic and then the uniform cost search which expands the most amount of nodes.

Comparing Max Queue Size and Memory Used

Figure 6. shows the maximum queue size at each solution depth, and Figure 7. shows the corresponding memory used (in KB) by the 3 algorithms. A* with the manhattan heuristic performs the best, using the smallest queue size and the least amount of memory across all depths.

There is a clear correlation between max queue size and memory usage, algorithms with larger queues generally use more memory. However, Uniform Cost Search uses much more memory than A* with misplaced tiles even though queue size is similar at higher depth. This is because Uniform Cost Search generates and retains more nodes in memory, particularly in the already expanded list, due to the absence of a heuristic. A* with a heuristic can avoid some of these extra nodes by cutting off bad paths earlier.

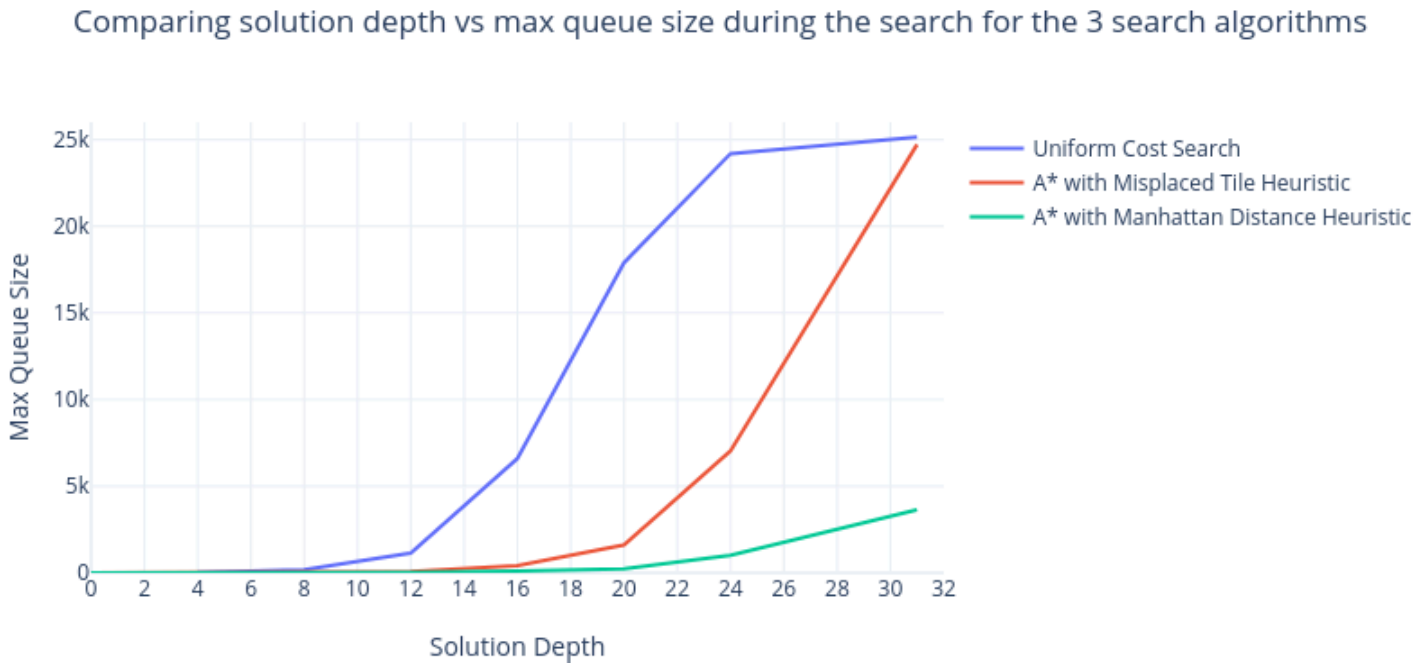

Figure 6. Solution depth vs Max Queue Size

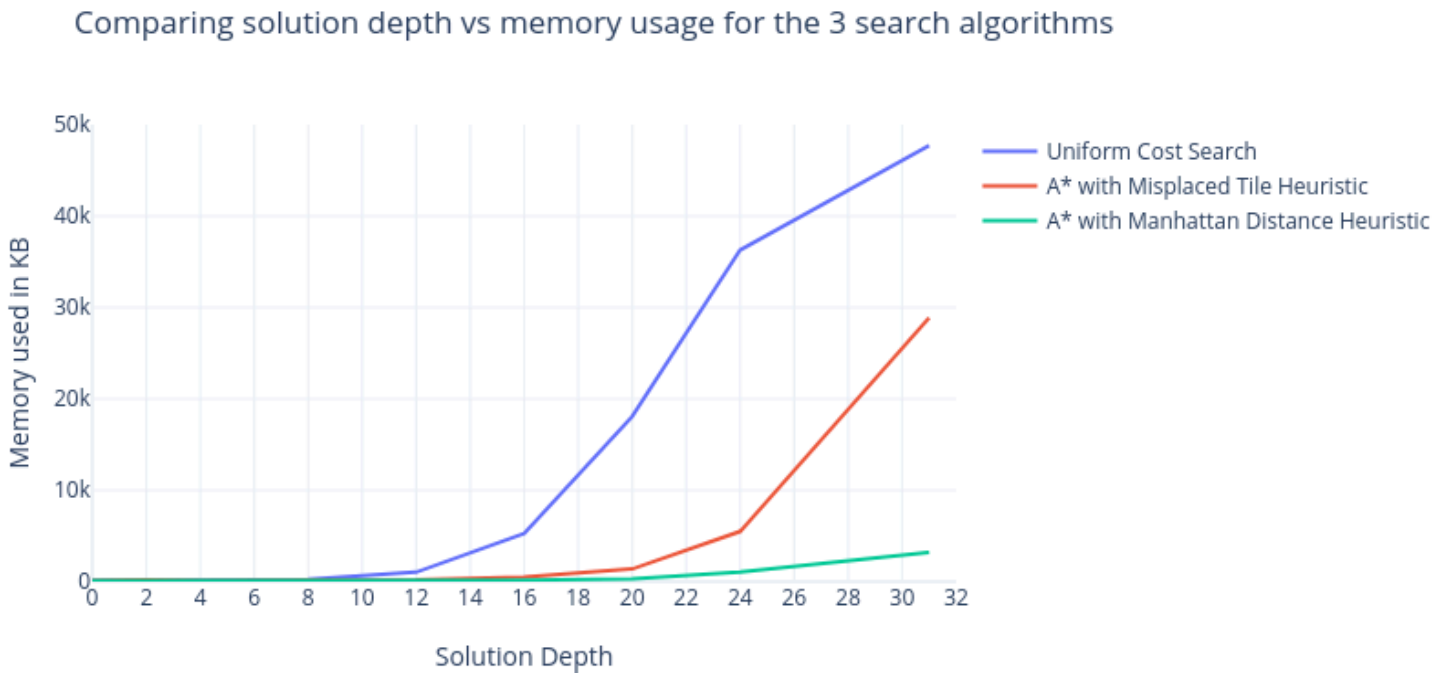

Figure 7. Solution depth vs memory used for the 3 algorithms

A* with the manhattan heuristic is the most efficient both in terms of queue size and memory consumption, followed by A* with misplaced tile, while Uniform Cost Search is the least efficient.

Conclusion

Considering the three algorithms compared (Uniform Cost Search, A* with Misplaced Tile heuristic, and A* with Manhattan Distance heuristic) across the different metrics (Time taken, Memory used, Nodes expanded) it can be said that:

- Out of the three, **A* with Manhattan Distance performed the best overall** in all metrics, followed by A* with Misplaced Tile, and finally Uniform Cost Search, which was the worst

- A* was faster and used less memory because it uses a heuristic to guide the search. Uniform Cost Search doesn't use any heuristic (it sets h(n) = 0), so it ends up working like Breadth-First Search and explores way more nodes than needed.

- Both heuristics helped the algorithm perform better, but the Manhattan Distance heuristic did a better job than Misplaced Tile. It was faster, expanded fewer nodes, and used less memory. This shows that using a smarter heuristic makes the search more efficient.

Sample Trace On An Easy Problem

Sample trace using a custom goal state for a solution depth of 1

```
milanmaragiri@ucr-secure-36-10-13-195-213 search_algorithm % g++ main.cpp -o main
milanmaragiri@ucr-secure-36-10-13-195-213 search_algorithm % ./main
Welcome to my 8-Puzzle Solver.
Type '1' for default goal state or '2' to enter your own goal state.
2
Enter the goal puzzle row by row (use 0 for blank):
Row 1: 1 2 3
Row 2: 4 5 6
Row 3: 7 0 8
Enter the initial puzzle row by row (use 0 for blank):
Row 1: 1 2 3
Row 2: 4 5 6
Row 3: 7 8 0
Select algorithm. (1) for Uniform Cost Search, (2) for Misplaced Tile Heuristic, or (3) the Manhattan Distance Heuristic.
1
The best state to expand with a g(n) = 0 and h(n) = 0 is...
[1 2 3 ]
[4 5 6 ]
[7 8 0 ]
The best state to expand with a g(n) = 1 and h(n) = 0 is...
[1 2 3 ]
[4 5 0 ]
[7 8 6 ]
The best state to expand with a g(n) = 1 and h(n) = 0 is...
[1 2 3 ]
[4 5 6 ]
[7 0 8 ]
Goal state!
Solution depth was 1
Number of nodes expanded: 3
Max queue size: 3
Time taken: 0 milliseconds
Memory used: 114688 bytes
milanmaragiri@ucr-secure-36-10-13-195-213 search_algorithm %
```

Sample Trace On A Hard Problem

Sample trace using the default goal state and a solution depth of 31

```
milanmaragiri@Mac search_algorithm % ./main
Welcome to my 8-Puzzle Solver.
Type '1' for default goal state or '2' to enter your own goal state.
1
Enter the initial puzzle row by row (use 0 for blank):
Row 1: 8 6 7
Row 2: 2 5 4
Row 3: 3 0 1
Select algorithm. (1) for Uniform Cost Search, (2) for Misplaced Tile Heuristic, or (3) the Manhattan Distance Heuristic.
3
The best state to expand with a g(n) = 0 and h(n) = 21 is...
[8 6 7 ]
[2 5 4 ]
[3 0 1 ]
The best state to expand with a g(n) = 1 and h(n) = 20 is...
[8 6 7 ]
[2 5 4 ]
[0 3 1 ]
The best state to expand with a g(n) = 1 and h(n) = 20 is...
[8 6 7 ]
[2 5 4 ]
[3 1 0 ]
The best state to expand with a g(n) = 1 and h(n) = 22 is...
```

Skipping the search tree till the end

```
[4 7 6 ]
The best state to expand with a g(n) = 30 and h(n) = 1 is...
[1 2 3 ]
[4 5 0 ]
[7 8 6 ]
The best state to expand with a g(n) = 14 and h(n) = 17 is...
[5 8 7 ]
[2 6 0 ]
[1 4 3 ]
The best state to expand with a g(n) = 25 and h(n) = 6 is...
[1 2 0 ]
[8 5 3 ]
[4 7 6 ]
The best state to expand with a g(n) = 31 and h(n) = 0 is...
[1 2 3 ]
[4 5 6 ]
[7 8 0 ]
Goal state!
Solution depth was 31
Number of nodes expanded: 6863
Max queue size: 3634
Time taken: 81 milliseconds
Memory used: 3604480 bytes
milanmaragiri@Mac search_algorithm %
```

References

[1] Deniz 8-puzzle solver: https://deniz.co/8-puzzle-solver/
[2] Dr. Keogh, Project handout: https://www.dropbox.com/scl/fi/hc9y86duxfc1toujylc2o/Project_1_The_Eight_Puzzle_CS_205_2025.pdf?rlkey=1r544xlegi1eddil3ua8ck6sv&e=4&dl=0
[3] Jason Wolfe's blog: https://w01fe.com/blog/2009/01/the-hardest-eight-puzzle-instances-take-31-moves-to-solve/
[4] Plotly Chart Studio (https://chart-studio.plotly.com/)
[5] URL to my code: https://github.com/3rst/search_algorithm