The Eight Puzzle

Author: Adrian Ramos

Table of Contents

Resources

For this project, I consulted the following resources:

- Dr. Eamonn Keogh's lecture slides on Blind Search and Heuristic Search
- Dr. Eamonn Keogh during Office Hours
- C++11 Documentation from cplusplus.com (URL: https://cplusplus.com/doc/)

All important code is original in this project. The following are libraries and predefined subroutines I included in my program to help with node manipulation:

- The Queue Library to access the **priority_queue** data type
- The Copy Constructor Syntax in order to perform **Deep Copy** on nodes
- The **operator()** Overload Syntax in order to compare f(n) costs between nodes to correctly input them into the **priority_queue**
- The **operator=** Overload Syntax to assign member variables from one node to another

Overview

The 8-Puzzle is a well-known puzzle that is a somewhat easier version of the original 15-Puzzle. The 8-puzzle contains a 3 x 3 grid with tiles numbered one through eight. There is one empty space in this grid that allows one tile to move at a single time. As shown in Figure 1, the goal of the puzzle is to arrange all eight tiles in an ordered fashion from least to greatest.

Figure 1: An example of what the 8-Puzzle goal state should look like once all tiles are ordered. This image has been provided by Google Play online.

For this project, Dr. Eamonn Keogh has assigned his students to create an 8-Puzzle program that solves this puzzle. The students are required to solve this puzzle using three established search algorithms we learned during lecture. These algorithms include Uniform Cost Search, A* with the Misplaced Tile Heuristic, and A* with the Manhattan Distance heuristic. The goal of this report is to summarize my findings with all three algorithms. I will note the performance of each algorithm using runtime and the number of nodes each algorithm expanded.

The following is the link to my GitHub repository that shows my entire code for this project as well as my commit history.

The Search Algorithms

As noted before, the three search algorithms that we were required to implement are Uniform Cost Search (UCS), A* with the Misplaced Tile Heuristic, and A* with the Manhattan Distance Heuristic. I will explain the characteristics of each algorithm in this section of my report.

Uniform Cost Search

Uniform Cost Search, also known simply as UCS, is a type of Blind Search Algorithm. As explained by Dr. Keogh in his Blind Search PowerPoint slides, this algorithm expands the cheapest node. This is done by enqueueing the nodes in order of cumulative cost. For this algorithm, the **g(n)** is simply the cost to get to state **n** from the initial state. Uniform Cost Search is complete and optimal only if the "*path cost is a nondecreasing function of depth*." As explained in the Project 1 requirements, UCS is just A* with **h(n)** hardcoded to zero. Its Time and Space Complexities are both $O(b^d)$, where **b** is the average branching factor and **d** is the depth of the tree.

A* with the Misplaced Tile Heuristic

The A* algorithm is known as the fastest search algorithm. It combines the completeness and optimality of Uniform Cost Search with the fast time complexity of Hill-Climbing Search. As explained by Dr. Keogh, the algorithm enqueues nodes "*in order of estimated cost to goal **f(n)***." Here, **f(n)** is the cost to get to a particular node, **g(n),** plus the estimated distance to the goal, **h(n)**. The **h(n)** is known as the type of heuristic used in the algorithm. A **heuristic** is a function that approximates how far the current state is to the goal state. We only use heuristics that are admissible, which means they never overestimate the "*merit of a state*." One of the heuristics used in this project is known as the **Misplaced Tile Heuristic**. For this heuristic, we simply compare the current state with the goal state by counting the number of misplaced tiles.

A* with the Manhattan Distance Heuristic

This algorithm is similar to the Misplaced Tile Heuristic in that it uses A* to search through the nodes. In this case, we use the **Manhattan Distance Heuristic** instead. For this heuristic, we compare the current state with the goal state by counting the total number of spaces all misplaced tiles are away from their individual correct positions.

Algorithm Analysis

For this project, I decided to compare all three algorithms in terms of their runtime and the number of nodes they expanded when searching for the goal state. To do this, I used eight test

cases that were provided by Dr. Keogh in the Project 1 Requirements. The test cases can be seen in Figure 2. Tracebacks of an easy and hard test case are provided at the end of the report. I did not include the full traceback for the hard puzzle in order to save space, given its depth is 20.

Depth 0	Depth 2	Depth 4	Depth 8	Depth 12	Depth 16	Depth 20	Depth 24
123 456 780	123 456 078	123 506 478	136 502 478	136 507 482	167 503 482	712 485 630	072 461 358

Figure 2: The test cases provided by Dr. Keogh to test our program

The following are my results of these comparisons.

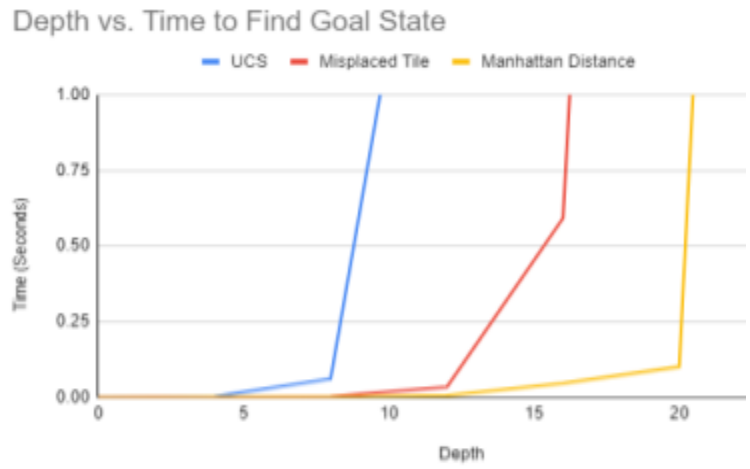

Figure 3: A comparison between all three algorithms' runtime for each depth

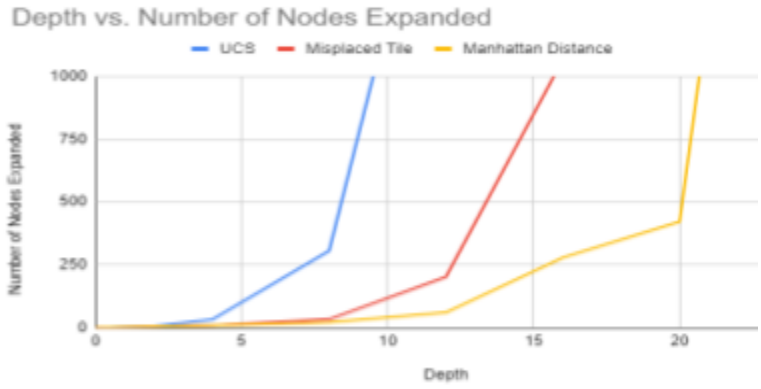

Figure 4: A comparison between the number of nodes expanded for all three algorithms for each depth

As depicted in Figure 3, we can see that the Manhattan Distance Heuristic has the fastest runtime. For example, at depth 16, UCS finds a solution in about 2 minutes, Misplaced Tile finds a solution in about half a second, and Manhattan finds a solution in about one-twentieth of a second. This means that Manhattan runs about 3,000 times faster than UCS and about 10 times faster than Misplaced Tile.

As depicted in Figure 4, we can see that the Manhattan Distance Heuristic also expands the least amount of nodes, making it very efficient and optimal. For example, at depth 16, UCS expands 17,960 nodes, Misplaced Tile expands 1,066 nodes, and Manhattan expands only 279 nodes. This means that Manhattan expands about 17,680 less nodes than UCS and about 785 less nodes than Misplaced Tile. This makes it the most efficient when finding the goal state.

Conclusion

After careful analysis of these three algorithms, it can be said that A* with the Manhattan Distance Heuristic is indeed the **fastest search algorithm** that is **complete** and **optimal**. Its runtime is much quicker than those of the other two, while expanding less nodes as well. From this, I can conclude that the use of a heuristic is greatly beneficial to an algorithm searching for a particular goal state. It allows the algorithm to perform an **Informed Search,** which allows its total cost, **f(n)**, to be minimized. My results also imply that some heuristics perform better than others. Although this is evident, using any heuristic will bring better results than those brought by any **Blind Search** algorithm.

Traceback of an Easy Puzzle (Depth 2)

```
Here is your puzzle.
1 2 3
4 5 6
0 7 8


Please select the algorithm you would like to use to solve your puzzle.
Type '1' for Uniform Cost Search.
Type '2' for A* with the Misplaced Tile heuristic.
Type '3' for A* with the Manhattan Distance heuristic.

3
The best state to expand with a g(n) = 0 and h(n) of 2 is?
1 2 3
4 5 6
0 7 8


The best state to expand with a g(n) = 1 and h(n) of 1 is?
1 2 3
4 5 6
7 0 8



1 2 3
4 5 6
7 8 0

Goal State!
Solution depth was 2
Number of nodes expanded: 5
Max queue size: 6
Total time: 0.00055 seconds
```

Traceback of a Difficult Puzzle (Depth 20)

```
The best state to expand with a g(n) = 14 and h(n) of 6 is?
1 5 2
7 4 3
0 8 6


The best state to expand with a g(n) = 15 and h(n) of 5 is?
1 5 2
0 4 3
7 8 6


The best state to expand with a g(n) = 16 and h(n) of 4 is?
1 5 2
4 0 3
7 8 6


The best state to expand with a g(n) = 17 and h(n) of 3 is?
1 0 2
4 5 3
7 8 6


The best state to expand with a g(n) = 18 and h(n) of 2 is?
1 2 0
4 5 3
7 8 6


The best state to expand with a g(n) = 19 and h(n) of 1 is?
1 2 3
4 5 0
7 8 6



1 2 3
4 5 6
7 8 0

Goal State!
Solution depth was 20
Number of nodes expanded: 422
Max queue size: 297
Total time: 0.09825 seconds
```