George Matta

CS4200 - Project 1

Report

8-Puzzle Solver: A*

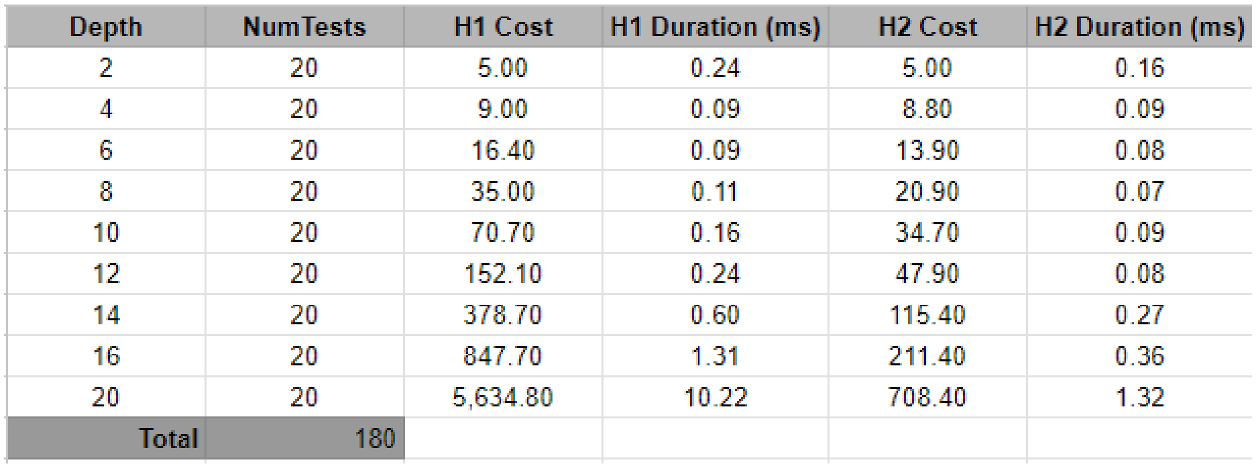
To say that this project has been the culmination of my CS career is a true statement. By far one of the most interesting projects I had been assigned in one of my classes, which I had expected to do way earlier in my schooling. I remember being a high schooler learning some basics about computer science and encountering “sorting algorithm visualizers” and “pathfinding algorithms”, and coming across the name “a star” not knowing what it meant. Now that I have learned how the A* algorithm works, and was even able to apply it (as well as 2 differing heuristics) to solve an interesting problem, I feel qualified to talk about the algorithm, and the heuristics, and compare the data I found.

Firstly, my approach. I would think that this is a pretty basic procedure for how to get started with a project like this, but I started with creating a Puzzle object that will be able to store information on the goal configuration, input configuration (with the ability to set an input with a variety of input methods (a string, a list, an array, etc)), the g-cost of the puzzle (the current depth), and helper methods that will create a new Puzzle depending on what move we are going to make (move left, right, up, and down). Additionally, I created a method that will return a list of methods of the possible moves that can be made (depending on the location of the empty tile and the walls) to allow for easy access to all the child Puzzles. Finally, there is a variable keeping track of this Puzzle’s parent puzzle, so that it can easily be traversed back once we find the solution (similar to a LinkedList). Next, I implemented the two heuristic methods.

The two heuristic methods were pretty intuitive, in my opinion. The first will count the number of misplaced tiles and use that as the H value. The second will calculate the total distance of each tile to its intended position and use that as the H value. These methods were also implemented in the Puzzle class, and an "f()" method was created that would calculate the requested heuristic value and sum it up with the g value. This will serve as the value by which two Puzzles are compared for the A* priority queue. Speaking of the comparators, a Comparator class was also created that will take two Puzzles and compare their f values. Pretty self-explanatory, and simple to implement, but allowed for a clean initialization of the aforementioned queue.

Finally, the A* algorithm itself. I created a class for this as well, the AStarSolver. This class will hold the statistics of the solution for a given input. Though we are requesting the solver to use a single heuristic method, it will end up running both heuristic methods to retrieve statistics for both of them. However, we will only print out the path generated by the requested heuristic. We also time both heuristics, store their costs (how many nodes had to be created for both), and store the depth of the found solution.

These classes culminate in the Main class. This class serves as a driver for the program and hooks up the terminal for user input to the backend for A* solving. Information on how to run the program itself (and how to navigate the 'main menu') can be found on the README.md file.



Depth	NumTests	H1 Cost	H1 Duration (ms)	H2 Cost	H2 Duration (ms)
2	20	5.00	0.24	5.00	0.16
4	20	9.00	0.09	8.80	0.09
6	20	16.40	0.09	13.90	0.08
8	20	35.00	0.11	20.90	0.07
10	20	70.70	0.16	34.70	0.09
12	20	152.10	0.24	47.90	0.08
14	20	378.70	0.60	115.40	0.27
16	20	847.70	1.31	211.40	0.36
20	20	5,634.80	10.22	708.40	1.32
Total	180				

As shown, the costs of both heuristics understandably increase as the depth increases, but the cost of the second heuristic is consistently less than that of the first heuristic. This is clear since the second heuristic is overall a better measure of a Puzzle's closeness to the goal. One thing that surprised me, however, is the duration of the heuristic methods. Firstly, I thought they would be much higher; thousands of Puzzles, each having to have their distances calculated. Secondly, I thought that H2 would consistently take longer than H1, but this is very clearly not the case (in fact, this heuristic was consistently faster than the other one). A simple explanation for this is as follows; although, yes, the value takes a longer time to calculate, since we are finding our solution in much fewer nodes than the other heuristic, we end up saving a lot of time in the end.