Abraham Yepremian

CS 4200.01

Artificial Intelligence

8-Puzzle Problem

Project 1 Report

3/3/2019

Introduction

In this assignment, our goal is to solve the 8-puzzle problem using the A* searching algorithm. We use two different heuristics to solve the 8-puzzle problem. The first known heuristic will be h1, that is the number of misplaced tiles, also known as the hamming score. The next known heuristic will be h2, that is the sum of the distances of the tiles from their goal positions. Beyond solving the 8-puzzle problem using both heuristics, I analyzed the efficiency of both solutions in terms of depth of the solution and search costs. To test the program and the efficiency of the heuristics used in the A* algorithm to solve the 8-puzzle problem, I generated 1020 random puzzle problems with a variety of solution depths (ranging from 2 to 24 in depth). I collected the data on the depths of the 1020 random puzzle problems tested, with their average search costs by depth. I also collected the average run time per depth. Using this data, I made a comparison of the average search costs for the A* algorithm using heuristics h1 and h2. The data was averaged over 85 instances of the 8-puzzle for each depth, for a total of 1020 cases.

The program I developed has an interactive menu with a few options. The first option is to randomly generate a solvable 8-puzzle problem and solve it using both heuristics, then outputting the optimal sequence of states was taken to solve the problem, the the solution depth, the search cost for each heuristic, and the time it took to perform the search. The second option is to have the user manually input a specific 8-puzzle configuration and it will be solved with the same output as above. The third option is to test 1020 random test cases with various solution depths. This option will output the search costs by heuristic and depth, including how much time each averages and the number of cases that was tested at each depth. The last option is to exit the program.

Approach to Project

My approach to the project was to begin working on this in Java. I studied the A* searching algorithm from the lecture slides in class. I also watched resource videos on YouTube in order to fully understand how this algorithm works, given the 8-puzzle problem. I decided to make two files to make this project work, EightPuzzle.java and Node.java. EightPuzzle.java would represent the main class for the project that is the executable. Node.java is a helper class that represents a puzzle node, that has information on the puzzle and the path/heuristic costs.

I also had to consider if I should use a single dimensional or 2d array for this project. I decided to go with a single dimensional array for this project because I was doing my best to keep it as simple as possible. I figured that if I kept it as simple as possible, then I would be able to finish the project while maintaining a full understanding throughout, and having ease of debugging throughout the process.

Since I knew that I would need to run and get data on over a thousand test cases, I did my best to keep the code as optimized as possible with the smallest overhead I could make. For this reason, I strived to use byte instead of int wherever I could. This is because byte is much smaller and takes less memory to use. I knew that for the numbers in the puzzle that I would never need a number larger than 128, so it seemed logical for me to use.

Using a single array made it easy for me to find where the zero index was at all times. I just ran a search through the array. Then to find out what the children nodes would be, I hardcoded the possible children into the program by index. For example, depending on where the

0 would be in an array like 1 2 3 4 5 6 7 8 0, I would know what direction the zero can be swapped. For example, if it was in index 0, I would know that it can only move down or right. In the Node.java class I coded the cases in for all indexes 0-8. I was able to swap the zero index with the new index, depending on the current position. This is how I generated the new children. The children would then be handled by the A* algorithm, where the lowest cost one would reach the top of the priority queue and be handled next.

I was also sure to check that each puzzle is solvable by checking that the sum of the inversions were able to be divided by 2. This is important to check in order to avoid infinite loops where the puzzle is unsolvable.

For my A* searching algorithm implementation, I essentially followed the pseudocode that was posted in class. I implemented the above to make it all come together for the 8-puzzle problem. I also have my goal state for the puzzle set to 1 2 3 4 5 6 7 8 0.

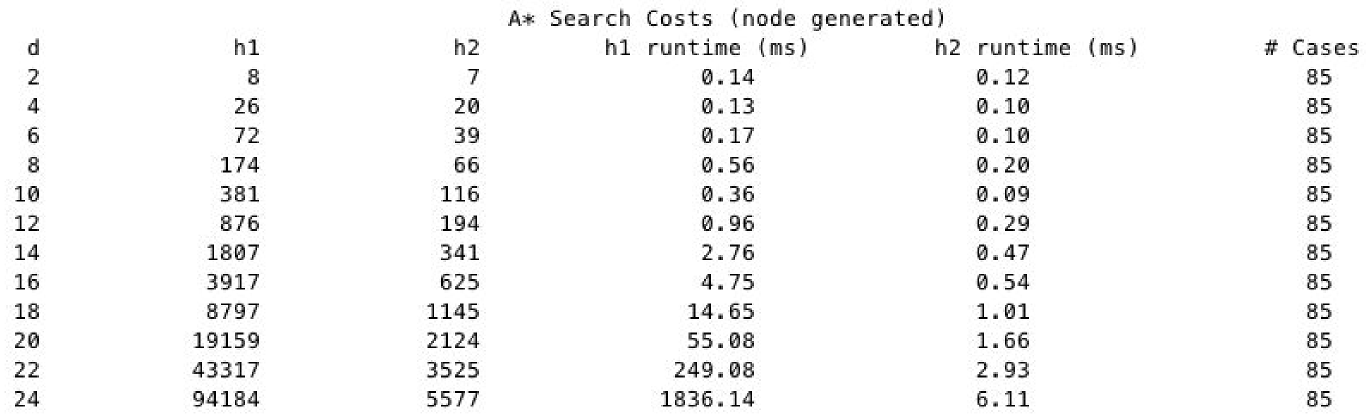
Data

Data is collected from a total of 1020 randomly generated puzzles, with 85 cases of each depth from 2 to 24 in depth. The data is captured for both heuristics, hamming score and manhattan score, labeled h1 and h2 respectively.

Table of Average A* Search Costs by Depth and Heuristic

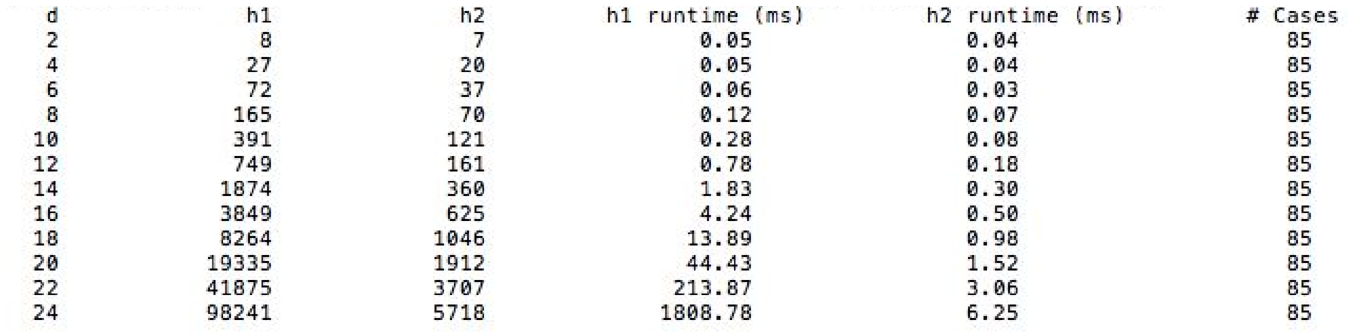
Depth	h1	h2	h1 runtime (ms)	h2 runtime (ms)	# Cases
2	8	7	0.14	0.12	85
4	26	20	0.13	0.10	85
6	72	39	0.17	0.10	85
8	174	66	0.56	0.20	85

10	381	116	0.36	0.09	85
12	876	194	0.96	0.29	85
14	1807	341	2.76	0.47	85
16	3917	625	4.75	0.54	85
18	8797	1145	14.65	1.01	85
20	19159	2124	55.08	1.66	85
22	43317	3525	249.08	2.93	85
24	94184	5577	1836.14	6.11	85



```
                                    A* Search Costs (node generated)
  d            h1            h2       h1 runtime (ms)        h2 runtime (ms)          # Cases
  2             8             7               0.14                0.12                   85
  4            26            20               0.13                0.10                   85
  6            72            39               0.17                0.10                   85
  8           174            66               0.56                0.20                   85
 10           381           116               0.36                0.09                   85
 12           876           194               0.96                0.29                   85
 14          1807           341               2.76                0.47                   85
 16          3917           625               4.75                0.54                   85
 18          8797          1145              14.65                1.01                   85
 20         19159          2124              55.08                1.66                   85
 22         43317          3525             249.08                2.93                   85
 24         94184          5577            1836.14                6.11                   85
```

Another example of data collected from the program



```
  d            h1            h2       h1 runtime (ms)        h2 runtime (ms)          # Cases
  2             8             7               0.05                0.04                   85
  4            27            20               0.05                0.04                   85
  6            72            37               0.06                0.03                   85
  8           165            70               0.12                0.07                   85
 10           391           121               0.28                0.08                   85
 12           749           161               0.78                0.18                   85
 14          1874           360               1.83                0.30                   85
 16          3849           625               4.24                0.50                   85
 18          8264          1046              13.89                0.98                   85
 20         19335          1912              44.43                1.52                   85
 22         41875          3707             213.87                3.06                   85
 24         98241          5718            1808.78                6.25                   85
```

Comparison of the average search costs for the A* algorithm using heuristics h1 and h2. The data was averaged over 85 instances of the 8-puzzle for each depth, for a total of 1020 cases.

Similar data can also be reproduced by running option 3 in the interactive menu in my program.

Analysis

We will compare the A* algorithm given the heuristics that are the hamming score and manhattan score, which will be synonymous with h1 and h2 respectively. The data collected is collected directly from me running my program and testing the heuristics on 1020 cases, with 85 each for each depth.

Comparison of Hamming Heuristic and Manhattan Heuristic

Given the data above, it is very clear that the manhattan heuristic is much more cost efficient than the hamming heuristic. The search costs generated by the manhattan heuristic are much lower than those generated by the hamming heuristic, especially as the depth increases. The search costs are directly correlated with the time average runtime per depth and heuristic as well when you look at the overall data trend. In order to understand this, we should understand what the heuristics are looking at. The hamming heuristic looks at the total number of misplaced tiles in the puzzle, whereas the manhattan heuristic looks at the sum of the distances of the tiles from their goal positions. We can see that the manhattan heuristic is a more accurate representation of the of the distance to a goal state per puzzle. The A* algorithm determines its next move using g(n) + h(n), where g(n) is the path cost of the node and h(n) is the heuristic cost of the node. The hamming heuristic is not looking at the whole problem necessarily and returns a rather rudimentary high overview number that is the total number of misplaced tiles. It is missing some information that the manhattan heuristic provides us, which is why the manhattan heuristic enjoys better performance.

Other Analysis and Findings

The hamming heuristic has costs which increase exponentially as the depth is increased, whereas the manhattan heuristic does not have costs which increase nearly as quickly. It is clear that the manhattan heuristic is superior for the A* algorithm given my work, research on this project, and the data found. It is notable that I have made my goal state in the project to be 1 2 3 4 5 6 7 8 0 rather than 0 1 2 3 4 5 6 7 8. I was having trouble in the beginning making my program work with the goal state of 0 1 2 3 4 5 6 7 8, so I made it 1 2 3 4 5 6 7 8 0 and it has been working fine since then. I think my program works for both and this can easily be changed in the first few lines of the EightPuzzle.java class, where I declare a static final goal state array.

This was an interesting project that made me use utilize my object oriented programming skills and learn deeply about the A* searching algorithm to solve the 8-puzzle problem. It took a ton of time to complete this project. The main takeaway from my analysis is that the manhattan heuristic is significantly more cost effective than the hamming heuristic given the A* searching algorithm. I hope you enjoyed my report on my analysis of the project.

Thank you,

Abraham Yepremian