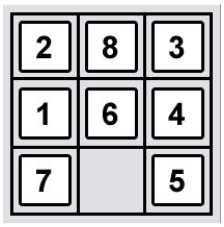
Searching for Answers
Alec Frey – CSCI 4350 – OLA1

Introduction

There are countless puzzles we grew up solving, but one of the most common is the 8-Puzzle Problem, or some variation of it. I personally played a variation whose goal was to move a special tile out of a slit in the side of the puzzle box. The rules were very simple, no removing tiles and replacing them just to get to the desired state. This simple puzzle lends us a perfect opportunity to practice our search methods, as it is small enough for most computer to handle, yet long enough that proper methodology is required for a quick answer.



Taken from 8puzzle.com

Theory

When looking for a solution to a problem, we often time have a systematic approach that we perform every time. More often than not, we use some sort of memory to help us find what we are looking for. At the very least, we know where something is more than likely to be. Thus it makes sense to help a computer use a similar technique.

Informed search strategies make use of helpful information called heuristics. These helper guidelines will direct the computer towards a path that will most likely lead to its destination. On the other hand, there are also uninformed search techniques, which make use of a predetermined order of node expansion. To help demonstrate why these values are so important to the searching process, we performed searches with various different versions of heuristics, as well as one without any at all.

Heuristics

Our project involved using four different heuristics. Two of them were tried and true functions, one was of my own creation, and one did not give any useful information. The first method tested was the search with no guidance at all, and is an uninformed search called Uniform Cost Search (UCS). As the later statistics will show, this search was greatly out performed by the others.

The second search involved adding up all tiles not in their final destination, I have called this the Displacement strategy. This represented the minimum number of moves required to solve the puzzle at that exact moment, ignoring the fact that you can only swap two tiles at a time.

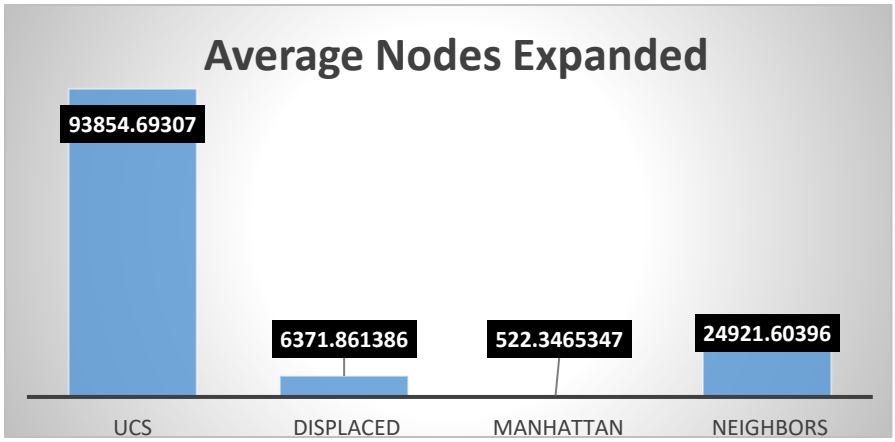
The third, used a similar method of adding the distance a tile was from its goal, called the Manhattan distance due to it spanning both the X and Y axes, similar to a city. This distance covered the amount of moves required to legally move your tile into location. Again, this represented a hypothetical amount of moves remaining, even though it was an underestimate.

The final heuristic was of my own design, and involved counted how many of the possible moves involved tiles that were not located at their final state. I dubbed this the Neighbor approach, as it was focused on the tiles neighboring the empty cell. I had planned for this to mimic the idea of pick up your surroundings.

A key part of all of these functions is that they returned a value less than the actual amount of moves to the goal. That is what makes heuristics work. By underestimating the total

distance to cover, the computer can select which move will give it the best "score". As you get closer to the goal, the heuristics will get smaller until they eventually lead you to your goal state.

Performance



As previously mentioned, two of the heuristics I implemented were methods commonly used to solve this problem. Thus, it was beyond reasonable to expect they would work rather efficiently. That being said, the Manhattan heuristic reigned supreme. This chart displays the average amount of nodes the program had to expand before finding the goal state. It is easy to see the benefit of a good heuristics. That being said, even my own approach greatly out performed UCS, which again highlights that having an informed search versus an uninformed search can make a huge difference.

Results Analysis

The values detailed below are statistics gathered for each search. N, is the amount of nodes in memory at the end of the search. V, is the total amount of nodes visited by the computer when calculating the path. B, is the branching factor, and represents the average amount of children to each parent node. Finally, D is the depth of the optimal solution.

UCS	N	V	B	D
Min	13	13	1.60	2
Median	44,052	51,933	1.83	18
Mean	65,281.82	93,854.69	1.87	17.09
Maximum	222,141	489,486	3.61	28
STD Dev	65,359.74	108,509.23	0.24	5.29

Neighbor	N	V	B	D
Min	10	10	1.55	2
Median	10,648	10,917	1.67	18
Mean	23,489.20	24,921.60	1.70	17.09
Maximum	179,087	209,820	3.16	28
STD Dev	31,802.47	35,064.53	0.16	5.29

When considering search patterns, it is important to know whether or not the search will find the best solution, not simply a solution. By looking at the D column, we can see all of the searches found the same depth solutions. From that, we can confirm these strategies were all finding an optimal solution.

The two tables above were the two least efficient search techniques. Though it was not as fast as the Manhattan and Misplaced strategies, my Neighbor heuristic was still able to easily out-perform UCS. This can be noted by both the N and V values.

The two tables below, were the two fastest and most efficient. Manhattan was able to consistently dominate the Misplaced heuristic. They were both incredibly fast techniques, but Manhattan was able to find the path by only expanding an average of 522 nodes. That is 1/12th the amount of nodes averaged by the Misplaced heuristic.

Manhattan	N	V	B	D
Min	6	6	1.24	2
Median	232	233	1.36	18
Mean	516.28	522.35	1.38	17.09
Maximum	3,593	3661	2.45	28
STD Dev	743.15	753.74	0.13	5.29

Misplaced	N	V	B	D
Min	6	6	1.43	2
Median	1,783	1,851	1.52	18
Mean	5,897.08	6,371.86	1.53	17.09
Maximum	84,796	99,096	2.50	28
STD Dev	10,980.83	12,452.77	0.10	5.29

Limitations

The biggest limitation I ran into throughout this project was ensuring that the program was behaving efficiently. The fact that these searches can expand thousands of nodes can really put a handle on your system. Thus, it is imperative to have quick data structures.

The nodes themselves can also be a problem, as they have to be stored in some fashion or another. Though that is a serious problem as the puzzle gets bigger, I felt that it did not necessarily impact this scenario.

Implementation

Throughout this lab, I entirely re-implemented my board structure three times. I originally started with a hash map implementation, but ran into issues with my comparing function. Without that function, I was not able to effectively maintain my frontier or check the list of previously visited node.

I then switched to using vectors to hold the board state. Holding these vectors, were a combination of sets and unordered maps. In hindsight, this implementation would probably have been fine, but issues I ran into at the time prompted me to make a final change.

After googling various tips on ways for storing the board structure, I came across a forum post that said they used a string to hold the state. I immediately thought that sounded like an interesting and good idea. Thus, I re-implemented one last time and stayed with it for the rest of the project.

The string led to very easy comparisons and a very intuitive tile location search. If you allow the index of the string to be the “location”, it makes searching and determining the final goal for each tile very easy. The built in comparator was always very handy for use in sets.

The final thing I found helpful, was preprocessing look up tables for data. Specifically for the Manhattan distance, the tables allowed me to feed in the tile’s location, and get back the distance immediately, with no need for calculation. When implemented with unordered maps, these provided constant time lookups.

Memorable Moments

By far the most memorable problem I ran into was my processing speed. I had re-implemented the program three times, but was still unable to get the performance I needed to run the UCS quickly. I had poured over my code countless times, eliminated more than a few for loops and cut all unnecessary calls.

Then I remembered a rather fundamental factor of programming, pass by reference. I had been passing by value in order to preserve the board states. Though our professors have told us to use a constant reference, I had never ran into this problem before. After I changed all my variables and look up tables to pass by reference, I shaved nearly five minutes off my execution time. That is most definitely a lesson I will never forget.

Citations

tokenjoker187, and alwaysLearning0 39. “8 Puzzle Breadth First Algorithm Help.” *DaniWeb*, 1 Sept 2011, https://www.daniweb.com/programming/software-development/threads/321727/8-puzzle-breadth-first-algorithm-help

The picture of the 8 Puzzle problem was gathered from the website, 8puzzle.com but no other information could be gathered.