Mitchell Griffith

Dr. Burns

CSC 575

5 March 2020

A\* 8 Puzzle

For this project we were tasked to use A\* search. In A\* I use as priority queue which uses heuristics to prioritize which puzzle in the frontier has the best chance of solving the puzzle most efficiently. For 575 we also had to create a random walk agent, a hill climb agent, as well as a second heuristic to use with A\* search. Creating A\* initially was certainly the most challenging part of this project, while the other tasks made this project very interesting and fun to explore.

As stated previously the heuristic is a way of determining which puzzle has the least amount of distance to go to reach the goal state. The puzzle with the least distance to go should be the one that is prioritized to make the next move first. This clearly helped. This makes the search more informed (I know informed is not the right term, but it felt right to say it here) than Breadth First Search. Compared to this Breadth first search seems brute force and takes much more space.

The first heuristic I used is Manhattan Distance. Manhattan distance is total distance each element has to move to be in its desired goal location (h) added together plus the total distance of the path it has taken the node to get to where it is at that moment in time (g) to get the f value which is what I used for the h() function. This differs from greedy best search because greedy best only takes into account the moves left to be made in the puzzle and does not take into account the distance traveled so far. That can cause you to not get the best possible path in some scenarios. This heuristic is fairly effective at improving the search only requiring the agent to explore 34 total nodes on the first A\* test to solve the puzzle. The 34 nodes also includes all of the nodes that are explored but have a bad heuristic so they are stuck at the back of the priority queue.

The other heuristic I decided to use is misplaced tiles. This heuristic counts the number of tiles that are not in the correct place instead of the distance the misplaced tiles must travel. I found this to be almost two times less efficient than Manhattan distance. This required 64 total nodes to be explored to complete the puzzle. This is almost twice as many nodes that have to be stored so it takes up much more space and could get much more out of hand with a larger puzzle. The table below also shows how much longer the misplaced tiles heuristic takes time wise.

The difference between these is fairly easily explainable. With each move that is made by the agent, the Manhattan distance changes. It either moves further away or closer to the goal by one piece having to move one less tile or one more tile, and the f value shows this. So the priority queue has can prioritize the better f value. While the misplaced heuristic can make a move and a tile also not move into place. This means the heuristic does not get better so many moves are just random and uninformed until some tiles eventually fall into place and cause the heuristic to get better.

This table represents the average of 3 tests per heuristic and puzzle.

|  |  |  |
| --- | --- | --- |
| Test | Manhattan Distance | Misplaced Tiles |
| Astar\_1 | 0.0029 s | 0.0059 s |
| Astar\_2 | 0.0434 s | 0.483 s |
| Astar\_3 | 122 s | 1587 s |

A\_star 3 took a very long time to complete with the misplaced tiles heuristic. This is a puzzle that really exposed how much less effective misplaced tiles is than Manhattan distance. While Manhattan distance still took a long time, it was significantly better than misplaced tiles.

Our next task was to create a random walk agent. This was the most interesting agent to watch solve the puzzle. It became a game to try to guess how many moves it would take to randomly solve the puzzle. This agent would start look at the possible moves to be made from the starting puzzle and then choose a random one to make. It continuously do this until it solved the puzzle. I saw the puzzle get solved in the triple digits and I saw it go over 1 million many times. It was fascinating to see how quickly the computer could make 1 million moves of the puzzle.

Random walk agent is obviously very inconsistent compared to a\* search. Both agents find the solution eventually, a\* is just much more reliable and much quicker on average. While it is possible for random walk to make the perfect ten moves off the bat, the chances are very low that it picks all of those moves first. A\* is just much more reliable and consistent when it comes to time. While random walk can take a very long time, it does not store any puzzles in memory other than the current node. This can save a lot of space on a big puzzle, but with a bigger puzzle the chances of solving the puzzle randomly get much lower and therefore the puzzle takes much longer to solve.

This table shows the results from 5 runs of random walk:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 |
| Nodes explored | 1709804 | 983433 | 453987 | 391821 | 236 |

The other agent we had to create was a hill climb agent. Hill climbing goes until its current neighbors do not have a better heuristic than it then it stops. The hill climb did not solve the first puzzle given in main() It would do a few moves then just finish without solving the problem. This was very confusing at first but the more times I read the read me file, the more I thought it was acceptable. This is one of the flaws of hill climbing.

In conclusion, this was by far the most challenging project I have worked on at WCU so far. It challenged me conceptually and code wise unlike any other project. I really enjoyed eventually working through and solving this A\* algorithm even if it is not perfect. My heuristic can definitely be improved I think and the way I find the path can be improved, but I learned a lot about how the heuristic works and how it helps improve search. The additional 575 work was also challenging but also fun to play around with them and compare them to A\* search. I am excited to see what is coming in the future of this class and how we can improve on these algorithms.