CS 170
Kristian De Castro
kdec001@ucr.edu
862025678 || kdec001

CS170 8 Puzzle Project

7th May 2020

OVERVIEW

The following report contains my results and complications of the 8-puzzle project. As a disclaimer, all major code is original (with some influence from lecture slides) which includes but is not limited to: algorithm searches and calculations. Most, if not all minor code has been either taken or influenced by sources cited at the end (2D vectors, priority queues). All references will be posted below.

GOALS

1. Solve the 8-puzzle as fast as possible
2. Optimize code to where runtime is really quick

SPECIFICATIONS

1. Uniform Cost Search
    a. This search is a best-first search algorithm that only accounts for step-cost in a priority queue. For this, we set the heuristic value to 0 and run through the algorithm as is.
2. A* Misplaced Tile
    a. This search accounts for misplaced tiles in the puzzle and finds the lowest cost tile to move to. The algorithm increments the heuristic value by one for each misplaced tile and would expand to those options with lower cost values.
3. A* Euclidean Distance
    a. This search takes the distance from point a to point b (sqrt[(x1 - x2)^2 + (y1 - y2)^2)]) as if there were no obstacles between them. This would then become the heuristic value and like all A* algorithms, it would find the lowest cost and expand towards that node.

ANALYSIS

This algorithm was a bit tricky to get working properly. In the end, it runs and it runs well, but has some difficulties on the Oh Boy case. Uniform cost search takes some time to complete, but I believe that is expected as it does not use h(n). However, I believe that I keep track of too many nodes so that it does not run on harder puzzles and ends up crashing because of memory overload. I thought I incorporated a way to ignore extra nodes but that will need some further optimization. The A* Misplaced tile algorithm runs the fastest. It isn't supposed to, but that is because my A* Euclidean has such a slow runtime. I was not able to figure out a more optimal algorithm for it so it has a runtime of O(n^4). This is a known bug and I will try to optimize it if I have the time in the future. So A* Misplace tile is the breadwinner, which it really is not supposed to be, but simply because finding misplaced tiles is faster than comparing the goal state to the end state, it runs the most optimally for me.

EXAMPLES

These are the test cases I used. I will show the runtimes and effectiveness of my algorithm on medium

Easy

$$\begin{bmatrix} 1 & 2 & 0 \\ 4 & 5 & 3 \\ 7 & 0 & 6 \end{bmatrix}$$

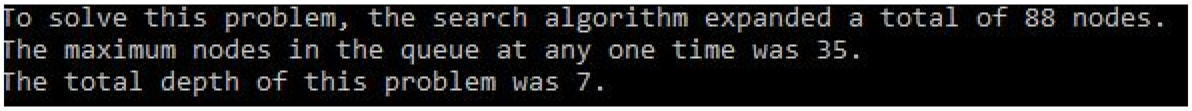
Medium

$$\begin{bmatrix} 4 & 1 & 2 \\ 5 & 8 & 3 \\ 7 & 0 & 6 \end{bmatrix}$$

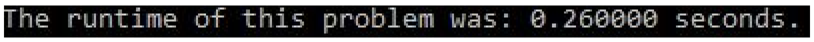
Hard

$$\begin{bmatrix} 1 & 4 & 0 \\ 7 & 3 & 2 \\ 6 & 5 & 8 \end{bmatrix}$$

Uniform Cost Search on Medium

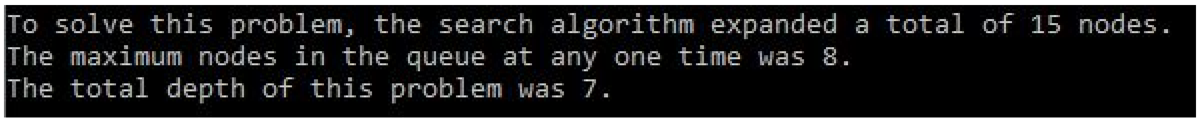


```
To solve this problem, the search algorithm expanded a total of 88 nodes.
The maximum nodes in the queue at any one time was 35.
The total depth of this problem was 7.
```

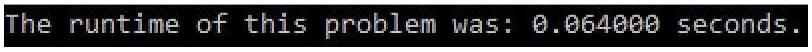


```
The runtime of this problem was: 0.260000 seconds.
```

A* Misplaced Tile on Medium

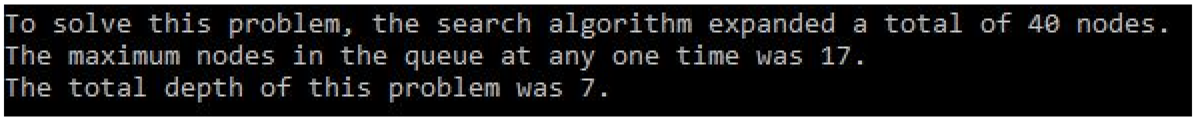


```
To solve this problem, the search algorithm expanded a total of 15 nodes.
The maximum nodes in the queue at any one time was 8.
The total depth of this problem was 7.
```

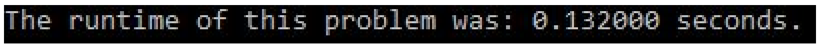


```
The runtime of this problem was: 0.064000 seconds.
```

A* Euclidean on Medium



```
To solve this problem, the search algorithm expanded a total of 40 nodes.
The maximum nodes in the queue at any one time was 17.
The total depth of this problem was 7.
```



```
The runtime of this problem was: 0.132000 seconds.
```

EXAMPLE ANALYSIS

Obviously more details are shown when you run the code yourself, such as the amount of nodes expanded, how many were pushed into the frontier, and the depth. But we can see that the UCS algorithm is the slowest, and the A* Misplaced is the fastest. Now by no means does this prove that A* Misplaced is faster than A* Euclidean Distance, and this is an error on the programmers part (me, sadly). I am not sure why A* Euclidean explores more nodes, it probably has to do with confusion when it's calculating the Euclidean Distance.

CONCLUDING THOUGHTS, PERSONAL REVIEW

This project was very insightful. I had the ability to conceptualize how to build this program, but at first I was completely lost on how to start. It was like having the lego pieces to build the death star without having the instructions. In all honesty, thanks to the piazza posts, I was able to grasp at least what the beginnings should be and what kind of storage system I should use. I ended up using a tree-search algorithm which I think led to my downfall as it kept track and built too many children nodes. Since the space complexity of Uniform Cost Search is O(b^d) where b is branching factor and d is depth, it overloaded memory and wasn't able to complete puzzles with depths greater than 20. I definitely learned that if I were to optimize this problem I would probably try to implement graph search instead. I would also want to find a better algorithm to calculate Euclidean Distance using 2D vectors. I could have maybe used a map or pair instead of 2D vectors. At least I was able to see that Uniform Cost Search is definitely slower than the A* searches. It demonstrates how important a good heuristic is for performance.