Mario Garcia
Kitten Solvers ™
CS420-01
Spring Quarter
Dr. Fang “Daisy” Tang

Project 1 Research

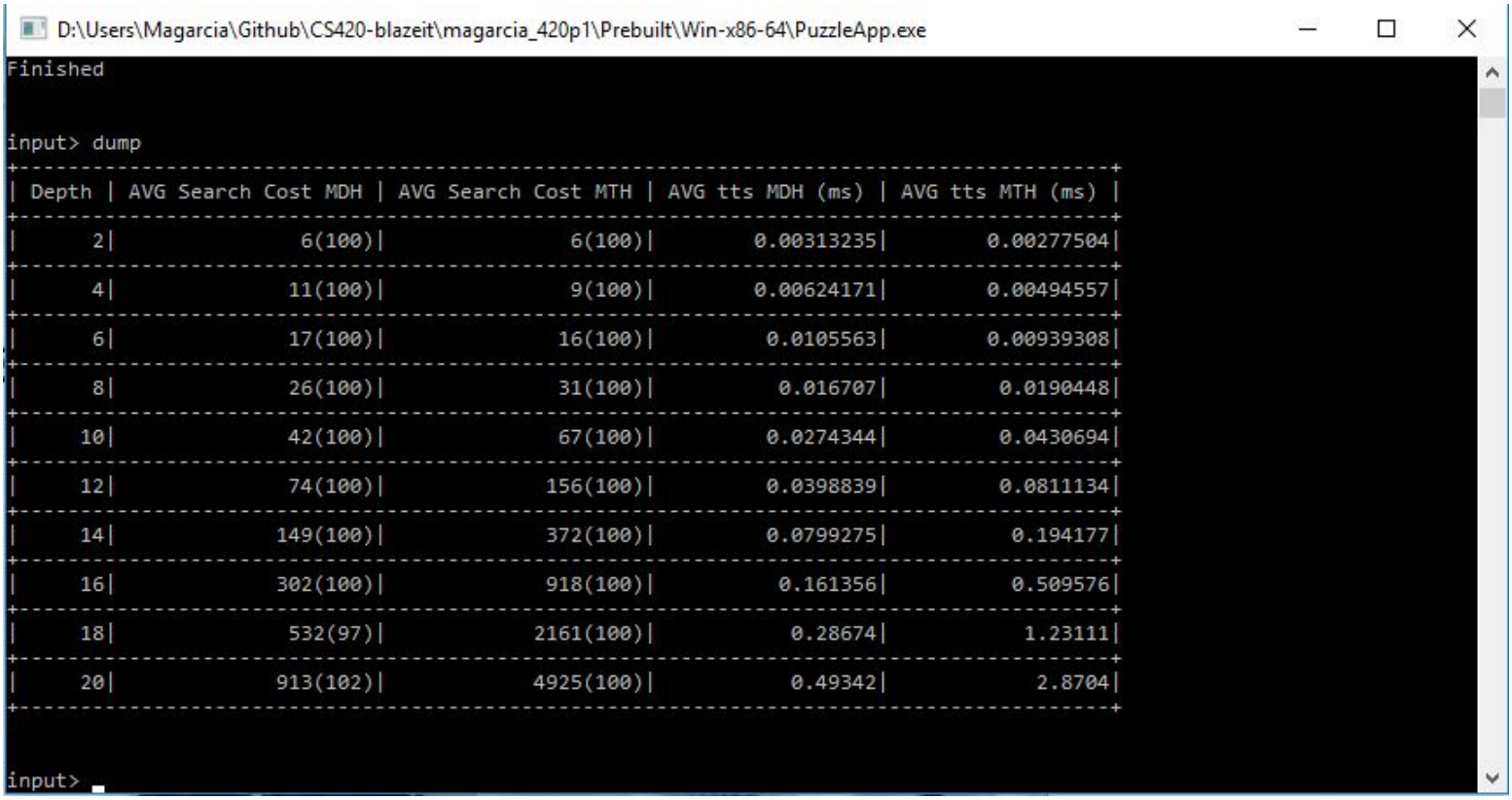
For this project, we were assigned to implement the A* Search Algorithm on the well known 8 puzzle problem. The idea behind solving this puzzle is roughly to get an understanding how informed search algorithms are intended to provide better decision making, along with improving on time complexity for finding the most optimal path. Of course, there are several ways to approach this when it comes to the 8 puzzle problem, so I would like to show you the way I went about implementing this search algorithm to solve our puzzles.

The key to my approach focuses on speed and readability, while also preventing too much burden with expensive node generation, and if needed, to cut the cost of generating nodes. Given this, the language of choice was C++, which provided the necessary ingredients to building an appropriate A* Search Algorithm. I broke down each step of the search, and finally decided to modularize most of the components needed for our algorithm. This included breaking up our Frontier, SearchTree, SearchNodes, and Heuristics into objects that will then be used to build up our final Engine object, which will be used by our application to solve for our puzzles. Within the Engine (or Solver) object, we would then take in a Puzzle (which will need to be for user input) and use our A* algorithm to solve. How this is done is rather tedious: We first check if the puzzle is unsolvable by using an invariant, to which we then initialize everything we need once we figure that our puzzle is indeed solvable. These variables to initialize would be the search tree, frontier, and heuristic of choice, to which we then push our first puzzle state (the start) to the root of our search tree. From here, we can then enter a loop which will always be checking if our frontier will no longer have any nodes to visit, for this will signify that our puzzle was unsolvable.

Within this loop, we will always be pulling a node out of a priority queue, that uses the following formula (g(n) + h(n)) to sort out the queue from least costly node to most costly. What this means, is that the cost of traversing that particular node can be quite expensive, so it would be ideal to choose the least cost node to traverse to, which will get us closer to our goal state. Once we get our least costly node, we then take it’s state, and generate multiple possible states from it. We then check if these generated states have already been visited, which we check with a hash map. If a particular state was not already visited, we would encapsulate it into a node object and store it into our hashmap for later referential use. Before storing however, we need to calculate our g(n) + h(n) cost for that particular state, along with the parent node, or the node

reference that we are currently checking. We store our current depth as the g(n) component and finally the heuristic h(n), to which we store into our node. We continuously loop for each iteration, or each node we pop from the priority queue, and perform the same steps until we finally check to see if we have reached our goal node. Once we do in fact reach our goal, we can calculate our path easily by traversing up the search tree from our goal node, to get our path and steps taken to reach our goal.

This approach proves to be rather general, and somewhat vague, yet it does explain the major portions of how my implementation was written for solving the puzzle. Now comes the interesting bit. What does the test case look like? Well… I decided that using 4 or 5 or even 10 test cases per depth would be too slow, so instead I test all 100 and 200 cases from the given text files, provided by Dr. Daisy Tang, for each depth, and calculate the average cost, and time to complete. I also test this with both our Manhattan Distance Heuristic, as well as our Number of Misplaced Nodes heuristic:



Where AVG Search Cost is the average number of generated nodes with Manhattan Distance Heuristic (MDH) and Misplaced Tiles Heuristic (MTH), along with their respective average time taken to complete each puzzle for each certain depth. Notice that there are parentheses around numbers in the search cost columns, this signifies how many puzzles were tested to get the

average cost. We discover that our Manhattan Distance happens to be rather quicker than our Misplaced Tiles heuristic, simply because of the way these are calculated. As a result, a better heuristic would yield an even better, and more efficient, solution.

The table above was calculated with all 300 cases provided by Dr. Daisy Tang. How this was accomplished was that we simply find those files, parse them, and then attempt to solve each one. After this, we then calculate our averages based on the all of the data we collected from our files, and display them as a table as shown above. We do notice that the implementation I have written does indeed show correct speed and time taken to solve our puzzles, however it is not entirely efficient. The average cost of our puzzles of 20 depth show to be much higher than the provided table reference in our book, which can in reality prove to be fatal in the long run, yet with faster computers this problem can be mitigated and possible to solve in incredible speed. We also notice that our Manhattan Distance heuristic search costs between depths 2-6 prove to be a bit worse than our Misplaced Tiles heuristic. This may be due to how the implementation of my A* Search Algorithm being improperly written, but nonetheless, when we increase the depth, our Manhattan distance proves to be the more efficient. Yet in this case, slow performance is nearly nonexistent with smaller depths.

Of course, my implementation could use a little improvement, especially when approaching the puzzle problem with a more direct approach. I did take careful consideration of avoiding heap allocations with new, or unique_ptr, since these allocations would cause bottlenecks in performance when solving our puzzles. I also took advantage of the cpu to allow for better cache coherence when dealing with node generation, as to make a more cache friendly A* Search Algorithm. All of this however, would be even better if our algorithm was further optimized to prevent needless node generation. I guess it all comes down to experience, especially since this algorithm is very well known, it can be abused or even poorly written, and still provide the best performance when it comes to searching.