Starting off this 8-puzzle project, I ran into a few problems in regards to where to start. I was not sure if I should create the nodes of a matrix first or a problem class to take in the initial state and goal state. I eventually ended up working with the user interface first and referencing the sample output to understand what functions I need to call and what functions would work as helper functions inside the algorithms. After coding the main I decided to think more in depth as to what the algorithms actually do and how to implement them with my project partner Kevin Gao. As I decided that I would need to call these 3 functions of search in main, I began piecing the helper functions together in order to traverse throughout the matrix itself. After I got the helper functions, I began implementing basic object oriented code and using data structures to make uniform cost search work. From there I only needed to write the other A* searches that calculates/adds distances to the priority queue to determine the cheapest node to expand next and call them with the same code used to call uniform cost search. I did also face some challenges when finding bugs and errors in my code, but after reviewing and talking about the possible errors in my code with my partner, I realized I made simple mistakes such as flipping the less than or greater than signs or not using pointers when needed in my priority struct causing it to expand the most expensive nodes rather than the cheapest! However by implementing unit testing in my program before fully integrating the functions, I made sure that writing the code would be more efficient and agile as I progressed.

As recommended in the project's guidelines, I decided to implement both a problem class and node class. My problem class obtains the initial and goal state of the puzzle while also performing the actual searches using private helper functions that I created to move the current position of the 0. My node class on the other hand has kept track of everything relating to the

node's state such as its position, move cost, depth, and my heuristic functions that calculate distance. Furthermore I decided to implement the data structures that were recommended such as the priority queue for my frontier to expand the cheapest node on the explored tree.

As demonstrated in my reported tables, using heuristic search on shallow problems does not matter too much. In the custom puzzles, it becomes clear that as the problems become more complex, heuristics cut down the costs of expanded nodes (time) and the max number of nodes (space) in the queue drastically. Although all A * misplaced tile, A * euclidean, and uniform cost search are optimal and complete, * search definitely beats out the uniform cost. Adding a few more steps to the puzzle exponentially increases the amount of time required because the algorithm is considering every possible path from the root to the goal state. This means that we also have an exponential increase in the amount of space taken up in order to keep track of which nodes were visited.

Through this 8-puzzle solver, I came to a realization of how beneficial using A * search is because in a sense it has a brain and is learning! I believe that the main advantage of this is that A * search is an optimal search algorithm in terms of heuristics. However it is clear that a good heuristic outperforms a weak heuristic since having a heuristic algorithm to compute h(n) may be inaccurate.
I can truly see why there are other real life applications other than puzzle solving such as maps because it can find the shortest path completely and optimally.

Tables compare different searches on different puzzles

Uniform Cost search

Puzzles	Nodes expanded	Max nodes in queue	Depth of search
1 (Default)	12	10	3
2 (Very Easy)	2	3	1
3 (Easy)	5	6	2
4 (Doable)	16	18	4
5 (Custom)	396	260	9

Misplaced Tile Heuristic

Puzzles	Nodes expanded	Max nodes in queue	Depth of search
1 (Default)	3	6	3
2 (Very Easy)	1	3	1
3 (Easy)	2	3	2
4 (Doable)	4	4	4
5 (Custom)	23	22	9

Euclidean Distance Heuristic

Puzzles	Nodes expanded	Max nodes in queue	Depth of search
1 (Default)	5	7	3
2 (Very Easy)	1	3	1
3 (Easy)	3	5	2
4 (Doable)	5	6	4
5 (Custom One)	15	17	9

Note: Custom Puzzle One uses the inputs 1 8 2, 0 4 3, 7 6 5. The rest of the puzzles tested are from the given pdf guidelines.

References used:

http://www.cplusplus.com/forum/beginner/98334/

https://stackoverflow.com/questions/12962778/in-graph-algorithm-what-is-the-best-way-to-determine-if-a-node-is-visited

https://visualstudiomagazine.com/articles/2015/10/30/sliding-tiles-c-sharp-ai.aspx

https://www.geeksforgeeks.org/stl-priority-queue-for-structure-or-class/

http://www.cplusplus.com/forum/beginner/9126/