Challenges
Our group consists of members who have varying experience using Python and libraries so the challenges faced differ with each one of us. However one notable challenge that some of us faced include learning the syntax and structure used in Python. For example in C++ we need to manually traverse a list or array in order to look for an object, but python makes these tasks trivial with built in functions for them which is so handy.

Furthermore we faced a hiccup where some of us were working with a 2d array for the 8 puzzle while others passed on a 1d array. Both methods work but one had to be implemented with 2 values in mind [i][j] while the single array could just take the length into mind and modulo the index to know what row/column it belonged to.

Additionally, the primary functions of search had their obstacles. Getting user input and setting everything quite simple, but creating a custom search where the best state has to be expanded had a lot of trial and error. Ultimately once the steps are in order for expanding a node given a certain operation, the rest of the operations are really easy as only a single value changes, the swapping index.

Handling various edge cases and ensuring the correctness of the implemented algorithm is a major challenge in 8 puzzle problem we have faced. For the Uniform cost search we have to handle each edge case for various input configurations efficiently in order to not to exceed the time limit as uniform cost search expands more number of nodes than heuristic search algorithms. Therefore, the code needs to be optimized to handle these large problem spaces without causing significant increase in runtime and storage.

Design
The design of the puzzle solver consists of 2 classes. A Node class and a Puzzle class. The node class is used to keep track of a state, its parents, g & h values, and actions. The puzzle class is what contains all functions. From getting the input (that sets the initial state), printing the puzzle in a visually pleasing manner, locating the index of the blank space, and of course the search functions and their helper functions to calculate h & g values.

The implementation of the puzzle class contains various methods for each algorithm and their supporting functions. The implementation of uniform cost search implements two basic data structures, which are a Priority queue such as heap which stores nodes in search space. And Another data is a unique list that keeps track of visited nodes so as to decrease the search space for the algorithm. The main method used is uniform_cost_search to implement the algorithm and heapq for priority queue data structure and set() for unique list.

Optimization
Python has built in features that make it easier to find items in a list but also using a priority queue made getting the best heuristic state very simple

The built-in library that helps in optimization of the code and search results in uniform cost search are heapq that are in-built library in python which helps in efficient retrieval of nodes with the lowest cost and reduces the run time of the algorithm. The other inbuilt function is set() in python that helps to create the unique list in python and thereby decrease in search space for the algorithm by avoiding the visited or discarded nodes.

Search Type
Our program uses graph search that tracks all explored states (to ensure we do not have an invalid or repeated state) and every node contains a state and a parent pointer that acts like an edge. We did not try both we simply went straight to graph because it seemed more logically simpler to implement and we wouldn't have to worry about tree traversals

For the uniform cost search we have employed a graph searching method known as Breadth First Search or BFS. That keeps track of all explored nodes and traverses each level of graph fully before going in-depth. This approach helps in preventing the revisiting of nodes, which generally leads to better performance compared to search algorithms that do not store nodes.

Comparison
The Trivial, Easy, Very Easy, and doable test cases ran rather similar for all 3 types of searches, there is a small almost unnoticeable difference between the A* searches and uniform cost but once we get to the Oh Boy and our own random puzzles then the difference is much larger. From our testing Uniform cost took the longest in all tests, A* with Manhattan Distance Heuristic ran the fastest, and A* with misplaced tile was close behind but the gap between all 3 grew with each harder puzzle as exponentially more nodes and states were explored and added to the queue.

In other words, for shallow problems such as in the first 2 test cases the heuristic algorithm does not significantly impact performance. But as the complexity of the algorithm increases using heuristic algorithms leads to substantial increase in run-time.

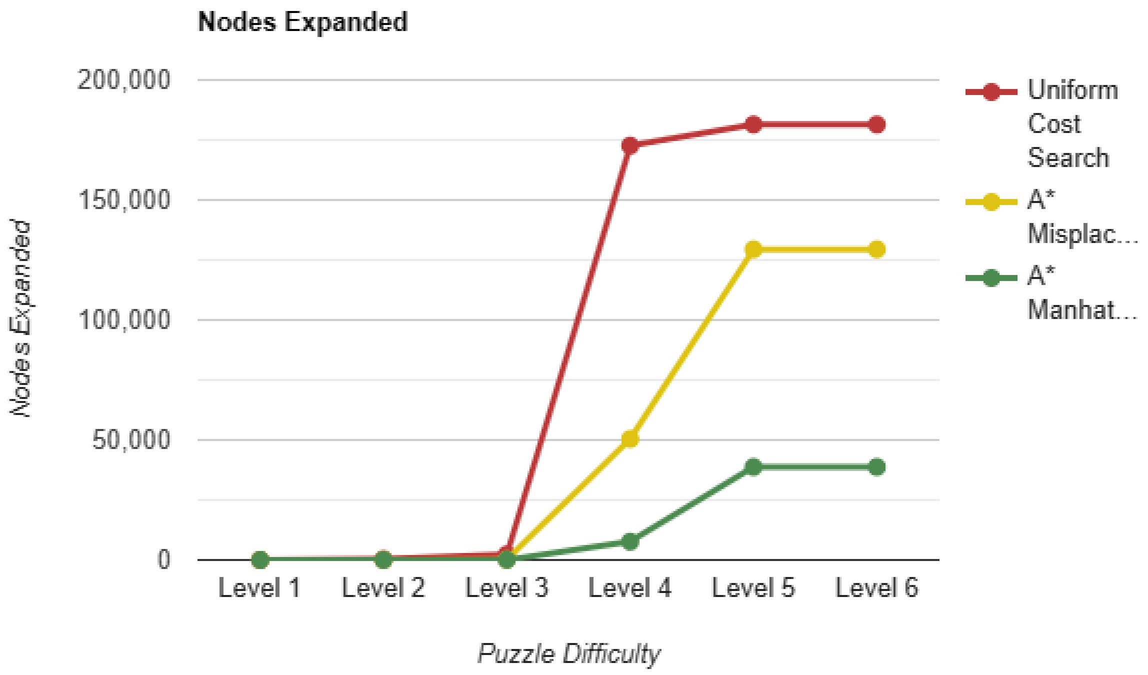
Level 1: [1 2 3 4 5 6 7 8 0] = no moves required, it is solved
Level 2: [1 8 2 0 4 3 7 6 5] = found online random medium one
Level 3: [1 2 3 5 6 0 7 8 4] = harder but not super hard
Level 4: [8 0 6 5 4 7 2 3 1] = really difficult one
Level 5: [8 6 7 2 5 4 3 0 1] = hardest 8 puzzle according to google
Level 6: [1 2 3 4 5 6 8 7 0] = literally impossible taken from slides

MAXIMUM QUEUE SIZE

	Uniform Cost Search	Misplaced Tile	Manhattan Distance
Level 1	1	1	1
Level 2	210	24	1
Level 3	1414	146	3
Level 4	32786	25795	4
Level 5	32834	42526	634
Level 6	-1 (no solution)	-1 (no solution)	-1 (no solution)

NUMBER OF NODES EXPANDED

	Uniform Cost Search	Misplaced Tile	Manhattan Distance
Level 1	0	0	0
Level 2	333	30	11
Level 3	2298	204	95
Level 4	172659	50430	7628
Level 5	181438	129445	38747
Level 6	-1 (no solution)	-1 (no solution)	-1 (no solution)



Contributions

Hritvik Gupta: Worked on implementing methods inside puzzle class to perform **Uniform cost search**, including designing the necessary data structures and optimizing the code for better performance.

Manojsai Kalaganti : Worked on the design and interface of the algorithm. Implemented A* with misplaced heuristic.

Vishal Menon: Worked on the design and implementation of the algorithm A* with euclidean distance as well as the maximum queue size calculation.

Eddie Vargas: Documented the project report, improved the user interface with each algorithm. Run all the test cases with each algorithm. Improved the program responsiveness by generating multiple output for each test case.

Citations
We referenced lecture slides and the optional reading for the project.