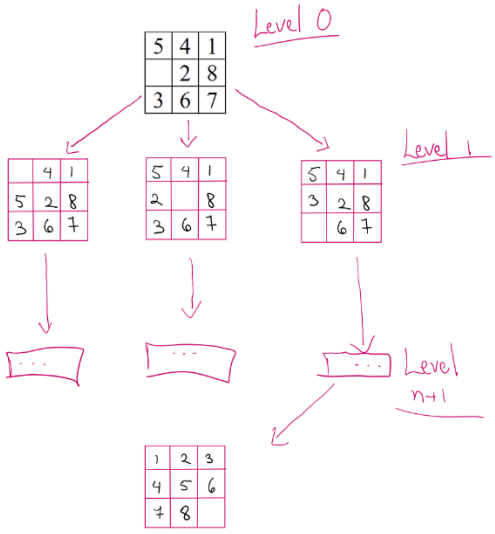
8 Puzzle Solver A* Visualization Algorithm

Part A:

For the hand demonstration, the approach was to draw a search or decision tree for each individual possible state that can be generated. As an example, our initial grid was [5, 4, 1, 0, 2, 8, 3, 6, 7], we could generate the following states from this grid which would represent the level 1 of the tree: **[0, 4, 1, 5, 2, 8, 3, 6, 7]** (move 5 down), **[5, 4, 1, 2, 0, 8, 3, 6, 7]** (move 2 left) and **[5, 4, 1, 3, 2, 8, 0, 6, 7]** (move 3 up) .



Search Tree – Drawn by hand.

We would continue to do so for each of the states generated at each level of the tree until we reach the goal state. The problem with this approach is that it is very time consuming, and it is not guaranteed to find the shortest path. For that we have several algorithms that can be used to find the shortest path, among them we have the A* algorithm.

Part B:

As we have seen in the previous part, the problem we would face is by doing it by hand, it will take us as many N iterations, and possible stuck in loops of never finding the solution. To solve this problem, we can use the A* algorithm, which is a search algorithm that finds the shortest path between two nodes in a graph. It does so by keeping track of the nodes that have already been visited and the nodes that are still to be visited. In our case, each node will contain the states of our puzzle, where every depth of the tree contains the possible moves for a prior state.

How does it work though? A*'s principle is not very different from Dijkstra's algorithm, which is also a search algorithm that finds the shortest path between two nodes in a graph. The difference is that A* uses a heuristic function to estimate the distance between the current node and the goal node. This heuristic function is used to calculate the cost of a given state. What makes this very brilliant is by following a simple formula, which is $f(n) = g(n) + h(n)$, where $f(n)$ is the total cost of the node, $g(n)$ is the cost of the path from the initial node to n, and $h(n)$ is the heuristic function, this can be applied to any problem that can be mapped into a search problem.

In our case as it's an informed search algorithm, the cost is calculated by finding the distance of each tile from its goal position. The distance is calculated based on their row and column position, only considering the number of moves required to reach the goal position. This is the heuristic function we will use for this project, with some modifications to make it more efficient.

Initially, we need a way to store and create our path, in a way we could easily revert it back to find the optimal solution. For this purpose, a decision tree, where each leaf contains the state of the puzzle, the parent leaf, the cost of the path from the initial node to the current node, and the heuristic function. The reason for me keeping the parent leaf is to be able to backtrack and find the optimal solution. This will be created within our class inside JavaScript. Then the actual tree is represented in a different class, where it contains the root leaf, states to be searched, the ones we have searched, as well as additional data we might need to store about the tree.

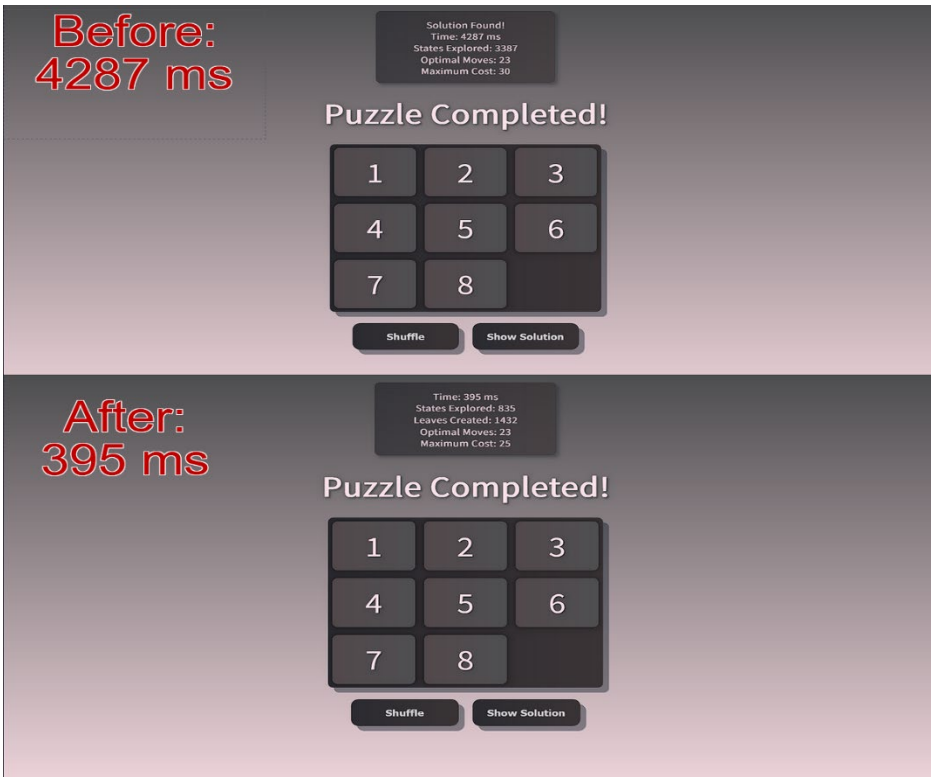
Before diving into the algorithm, we need to understand and determine if the current state/grid of the puzzle was solvable or not. This is done by counting the number of inversions in the current state, and if the number of inversions is even, then the puzzle is solvable. If the number of inversions is odd, then the puzzle is not solvable.

As for the actual algorithm is quite interesting how I have been improving it, however most of the magic happens at the heuristic function. Initially I was just counting the number of misplaced tiles, which is a very simple heuristic function, however it was not very efficient. Originally, I quickly found out that it was taking too long to find a solution.

The first few runs happened in a matter of seconds, but then it started taking minutes, and then crashing for complex cases. I did some research behind the most complex case, and I found the most complex would take 31 moves to solve, which is not that bad, but my algorithm was taking too long to find a solution. I then decided to use the distance between each tile to their goal position.

For example, for the 3x3 grid, [4, 2, 1, 0, 3, 5, 6, 7, 8], the distance of 4 to its goal position [1, 2, 3, 4, 5, 6, 7, 8, 0] is 3, as it is 3 moves away from its goal position. I implemented this approach, and finally started seeing some results. The first few runs were finding the solution and performance was very good. Average solution would be found within 4 to 5 seconds, which is not bad at all. However, I was still not satisfied, as I wanted to find a way to make it even more efficient.

I then thought two things: First, I didn't necessarily need to know the position of element 0, as it's always the last place, and I can assume if everything else is sorted, then 0 would be as well. Second, instead of finding the distance to their respective position within the array, I could look at the adjacent places, and calculate the distance of those places relative to their index. For example, back with the example of earlier, the distance of 4 to its goal position [1, 2, 3, 4, 5, 6, 7, 8, 0] is 3, but if we look at the adjacent places, we can see that 4 is 1 move away from its goal position, therefore we get a more accurate reading without overestimating the distance.



Heuristic A vs B – Time 4287ms vs 395 ms for a Hard Case

The experiment worked, and it brought down the average run down per solution to less than half a second. I was very happy with the results, and I was able to find the solution for the most complex case in less than 3 seconds. With a combination of HTML, CSS, and JavaScript, I was able to create a simple UI that allows the user to input the initial state of the puzzle, and then it will find the solution. The user can also choose to see the steps of the solution, or just the final solution. It also includes a manual mode, where the user can move the tiles around, and then it will find the solution, it would also tell you how many moves it took to solve the puzzle.