**Heuristics:**

We initially began with four admissible heuristics. We chose to use heuristics 2 and 3.

**Heuristic 1:** Give a grade of +1 for each tile that is not in its goal location.

**Heuristic 2:** For each tile out of place, give a grade of the Manhattan distance to its goal location.

**Heuristic 3:** Give a grade of +1 for each tile that is not in its goal column and +1 for each tile that is not in its goal row.

**Heuristic 4:** Solve the board with the ability to swap between any tile with the “0” tile only.

**Implementation:** We implemented our code in Python.

**Node:** We created a Node class, containing the current state of the problem, the parent of the current node (an instance of Node), f, g, h, neighbours. In the algorithms, each time we expanded a node, we created nodes for each of its children such that the only nodes created are nodes that we are planning to visit (unless we reach our goal beforehand).

**Goal State:** We used a static variable to easily reference our goal state throughout the runs.

**Initial State:** We used a randomizing algorithm in order to create out initial state to solve. We did this by keeping an arsenal of the tiles that were not yet entered to the game and entering them randomly, each time removing the inserted tile from the arsenal. Once we had an optional initial state, we had to check if the state was solvable. This was one of the difficulties we encountered. We kept randomizing solutions until a feasible one was found.

**Recreating Path:** Once we have reached our goal state, we use our goal node to trace back the parent nodes and recreate the solution path. Since each node has a single parent node, we could trace the single solution path that resulted in finding our goal.

**A\*:**

**Optimality**: Since we are using an admissible heuristic, the A\* algorithm is optimal. We learnt the proof in class.

**Completeness**: Since our graph is finite, the A\* algorithm is complete. If the algorithm receives an unsolvable board, it will return None. In our runs, we used only solvable boards.

**Soundness**: We only search in a space of feasible solutions and the parents of each node can only contain paths that led to the node, thus the algorithm is sound.

**Terminates**: Since our graph is finite, the A\* algorithm terminates. In the worst case, it will open all possible states.

**Branch and Bound:**

**Optimality**: BnB is optimal since in the worst case we will go over the entire tree. **Completeness**: Since our graph is finite, the BnB algorithm is complete. If the algorithm receives an unsolvable board, it will return None. In our runs, we used only solvable boards.

**Soundness**: We only search in a space of feasible solutions and the parents of each node can only contain paths that led to the node, thus the algorithm is sound.

**Terminates**: Since our graph is finite, the BnB algorithm terminates. In the worst case, it will search the entire tree.

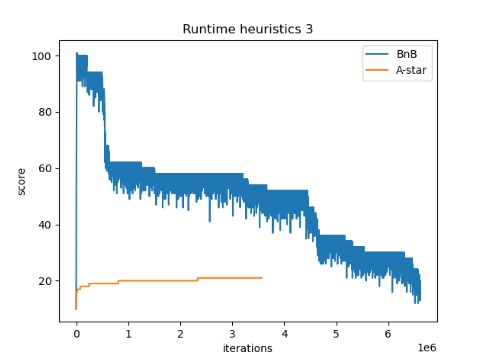
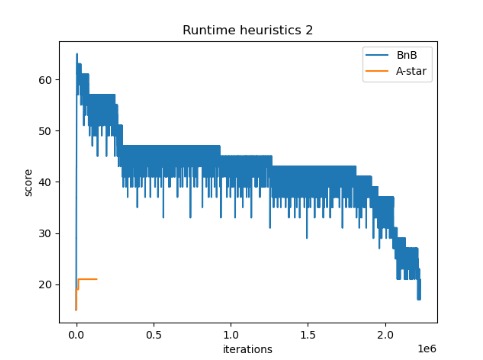
**Anytime**: Once we have found an initial solution, the algorithm is anytime - it holds the best solution and continuously compares its UB with new solutions. If we were to stop the algorithm after a certain number of iterations, we could return the best solution found.

**Problems:** We ran into a few issues.

Firstly, there was an issue of cycles in the graphs- in BnB we had to ensure that in each branch, there were no reoccurrences of a state.

Secondly, the issue of unsolvable problems, which we solved by checking if it was solvable beforehand. We could send the algorithms unsolvable problems as they are complete - they will return a “None” answer, however the runtime would be very long.

Lastly, we had the issue of defining what an iteration is in this case. As the algorithms are different in their logic, we wanted to find a mutual task between the algorithms to define the iterations.



Comparison of runtime score with heuristic 2, running on A\* and BnB algorithms.

Comparison of runtime score with heuristic 3, running on A\* and BnB algorithms.