Search Strategy Implementation with the A Algorithm*

The chosen search strategy for this task is A* search, which is a best-first search algorithm. It incorporates a heuristic function (h) to estimate the total cost (f) from the starting point to the goal. A priority queue, implemented using Python's heapq function, is utilised to sort nodes by their total cost.

To determine available moves, all empty coordinates adjacent to the current red pieces on the board are identified. Then, each of the 72 different possible pieces is attempted to be placed on these coordinates. If successful, any filled lines on the board are removed. If the resulting board state does not match the goal state, the search continues.

Data structures include dictionaries for tracking the costs (g and f), with the current board state/node as the key. To ensure hashability, the board states are transformed using frozenset(). Additionally, a predecessor dictionary facilitates backtracking from the goal state to determine the path. A visited set prevents redundant exploration of nodes.

Time and Space Complexity Analysis

The worst-case time complexity of the A* algorithm implementation is O(b^d), where b represents the branching factor (average number of successors per state) and d denotes the depth of the optimal solution. In this case, b can be possible boards after all possible moves and d is number of steps to get the optimal solution. This complexity arises due to the necessity of exploring and storing all potential nodes. The worst-case space complexity would be also O(b^d) since it needs to store all the nodes in memory.

The average-case time and space complexity are a lot tricker and hard to get a big O notation for that. The complexity exponential in [relative error in h times length of solution]. Our heuristic performs better in hard cases with less room to expand and can break empty space surrounded by blue pieces. For case like that, even the optimal solution needs 7 steps it will get it in 10k nodes.

Heuristic Functions

Two heuristics are computed for each node: the minimum of two horizontal and vertical line heuristics. These heuristics ensure the filling of the correct goal line while guiding the search algorithm effectively. The following heuristics are applied:

- goal_line_completion: Estimates the distance to the goal state based on the number of empty coordinates in the goal line.
- distance_to_goal_line: Determines the minimum Manhattan distance from any red piece's coordinate on the board to any coordinate in the goal line without obstacles in

the way.

If distance_to_goal_line cannot find the minimum distance because it is completely blocked, it defaults to the distance_to_furthest_goal, since we are assuming that a line needs to be cleared.

- empty_around_by: Checks if goal line cords are surrounded as this implies that nearby lines need to be cleared
- path_continuity: Encourages a relatively straight and direct line approach for improved goal line filling.

These heuristics are admissible and consistent as they do not overestimate the number of cords to reach the goal. The heuristic is finally multiplied by the float 8/7, as this brings h exactly equal to the value of g and thereby greatly increases the efficiency of the search. This float was found through vigorous testing.

Scenario: All blue tokens need to be removed

While the worst-case complexity of O(b^d) would remain the same for both time and space for obvious reason, the average-case complexity would be dramatically altered. The programm will need to do the search and calculate heuristic based on multiple rows/columns to fill. Therefore, it may have time cost times the total rows or columns to fill. And the space complexity still depends on the optimal solutions.