Jing Su 6839039

Architecture:

I used a 2D vector to represent the game board. Since we're required to use <letter><number> format to represent a move, I created two functions that convert between the "move" entered by the user and the index number for the 2D vector.

I also put two markers, one at the upper left and the other at the bottom right, on the 2D vector such that they form a rectangle that includes all the pieces at the current iteration. When doing searches, I simply extend the marker by two blocks out and that would be the search space because it doesn't make sense to take a step that's far away from the current pieces.

Search:

I used the minimax algorithm to determine the best move, and the scores are calculated as follows:
Score = max(score(I take step 1) – min(score(enemy take step 2)))
To determine the score at each step, my program traverses all the search space. For each block in the search space, the program expands towards four directions (right, bottom, bottom left, bottom right) to find connected pieces. The program will also check if the pieces are being blocked at any ends or if there's a gap in between.
My program will then use the score table to sum up the total score:

Open three – 200, Capped three – 50, Open four – 20000
Capped four – 195, Two open two – 49, Connected five – 50000

I also used alpha-beta pruning to reduce the running time. Once the program finish traversing the deepest level, d, and return to level d-1 (using dfs), it stores current_max and current_min to the corresponding depth level depending on who is the max/min player. If the program finds a node that's worse than the existing max or min at level x, than the program returns to level x and go to the next iteration.

There're certain combinations that will lead to the victory, such as two open three or an open three with a capped four. When encountering such situations, there're bonus points being added to the score.

Naturally, we want to avoid taking a step near the edge of the board, so there's also points deducted when taking a step near the edge of the board, depending on the distance to the edge.

Challenges:

The minimax algorithm is not a simple algorithm, it took me a while to discover the appropriate method to perform this algorithm while traversing the game board at the same time.

Also, there are so many edge cases in the heuristics function that I have to deal with, especially when you are moving closer to the edge of the board.

Weaknesses:

For my algorithm, the search depth is always 2. I tried to increase the search depth, but it makes my program a lot more unreliable (timeout, edge cases in heuristics function). A possible solution would be dynamically changing the search depth throughout the game. Also, writing an estimator function for the search time would also be helpful to avoid timeout.