**Heuristic**

My heuristic favors men closer to the final row, where they would be crowned. For kings, it calculates the closest enemy piece to the king and favors the king being closer enemy pieces. Hopefully, that will place kings in a better position to capture enemy pieces. I considered the case where the king was more than 1 square away from the piece, and would move closer, only to be captured on the opponent’s turn. The minimax search should prevent this from ever happening, even if there is a Horizon Effect. It is possible that the search reaches a depth N, and the king gets captured at depth N+1, but the search on the next turn should be able to see that state and compensate appropriately. My heuristic also favors reducing the number of pieces on the board for the player in the lead. The goal is to implicitly encourage trading pieces when a player has an advantage.

**Board Class**

My implementation of my board class relies on pointers to my piece class. I create instances of my piece class at the start of the game and store pointers to those pieces in my board class. I use a single instance of an “empty” piece and a single instance of a “filler” piece to represent multiple locations in the 2d board array. Unlike regular pieces, empty and filler pieces do not need to store their location, meaning the same piece can be reused for every location that requires an empty or a filler piece.

**Minimax Search**

I did not use my Terminal Test function in my minimax search. My heuristic returns a very large number (much larger than a normal score) when a terminal state is reached, so that should be sufficient for the search.

In my minimax function, I use recursion, passing by reference my board object. In order to evaluate a set of moves, I would have to “play” each move and calculate the score of the resulting board using my heuristic. “Playing” each move involves my moveResult function, which directly alters the board object. Because I passed my board object by reference, changing the board object at depth D will also change the previous board objects from depth 0 to D-1. To deal with this, I make a copy of the board class at the beginning of my minimax function, which is why I decided to use pointers to my piece class, rather than using instances of my piece class; copying pointers is faster than copying the class. This ensures that the data structures (unordered\_set and list) I use in my board object will not be affected. However, the data structures contain pointers to pieces, meaning a change to the pieces at depth D will also affect the pieces from depth 0 to D-1. To handle this, I created the isolateBoard function, which will make copies of all pieces that are potentially affected by a move. This is significantly less than copying the entire board (12 for a move and 16 for a jump, compared to 64 pieces in the entire board). Then, pointers to the original pieces in the object are replaced with pointers to the copied pieces. Thus, any change made to the board object will not affect previous board objects.