I had a lot of trouble writing the evaluation function for this program. My initial evaluation function scanned only the top layer of pieces. If a piece had a like piece it could connect with, the function added value to the piece. If instead the function saw a dissimilar piece, then it would zero the value of the piece being evaluated. This method missed some edge cases, most notably one where if the top layer piece was in the middle of a connect 4, but the program saw an opposing piece less than four tiles away in the opposite direction, the program would inappropriately zero the piece being assessed.

In my second attempt at an evaluation function, I tried to amend the function so that some of the edge cases the first function missed were now accounted for. I modified the function so it would now add value to the original piece if a like piece was found, continuing until an unlike piece was found, which would raise a flag that that direction was dead and could no longer accumulate value. I also added a flag to check if the space was contiguous. If it was, the program then summed the number of contiguous pieces for each vertical, horizontal and diagonal axis. These changes did not produce the desired effect.

For my third and final evaluation function, I had the program scan more than just the top layer. That choice originated as an effort to speed the function up. This function first checks if a *isevlauated* flag is set. If the flag is not set the function will first set it, then begin checking every piece on the board. However, it only checks up, right, up and to the left, and up and to the right for piece adjacency. Because it checks every piece, it does not need to check to the left or down, as those pieces have already been evaluated. When a like piece is encountered in one of these directions, a point is added to the score for the owner of that piece. If an unlike piece is met, that direction is marked as dead and can no longer accumulate score. Using this method means two points are added for groups of three, and one point is added for groups of two. Once every piece has been scanned the minimizing player’s score is saved as well as the maximizing player’s. Finally, the Minimizing player’s score is subtracted from the Maximizing player’s score and returned. If the *isEvaluated* flag is set, this process has already occurred and the minimizing player’s score minus the maximizing player’s score that was previously saved is returned.

Minimax function is recursive, and so is the alpha-beta enhancement. The functions take a board data object, the turn of that board, and in the case of alpha-beta, alpha and beta scores. Both functions operate nearly identically, except the alpha-beta function also checks if the previously returned value is greater than Beta or less then Alpha, depending on turn number, and returns that previous value immediately if it is. The turn number is modulus 2 to determine if it is the minimizing player or the maximizing players turn. First, the function checks if the input board is a winning board, if it is that value is immediately returned. Next the function checks if the provided board has previously generated children, if it does not then children are generated. Each valid child is then stepped through and the function is called recursively on them until the desired depth is reached. Next the returned value from the recursive call is compared to Min and Max, Min is replaced when the return value is lower, Max is replaced when the returned value is greater. This is also where Alpha-Beta differs from minimax and checks the value against the alpha and beta inputs. After each child is stepped through either the max or the min is returned depending on the turn for the input Board

Because I am calling minimax recursively, at the top level I use a different function called *PickMove*. *PickMove* is similar to minimax except rather than returning the min and max values, logic has been added to return the index of the minimum or maximum evaluated board (which is also the column for that move). The *minimax* function serves to evaluate all of the possible moves from the current board and the *PickMove* function serves to first decide which of those boards is ideal, then translate that information into a move command.

Neither the minimax nor alpha-beta pruning functions demonstrated a clear speed advantage over the other. The target depth is hard coded at eight moves. Minimax averaged 13.71ms to make a move. Alpha-beta Pruning averaged 13.62ms to make a move. I was never able to beat either version of the game, which indicates to me that either I’m terrible at connect four or my algorithm is functioning splendidly.