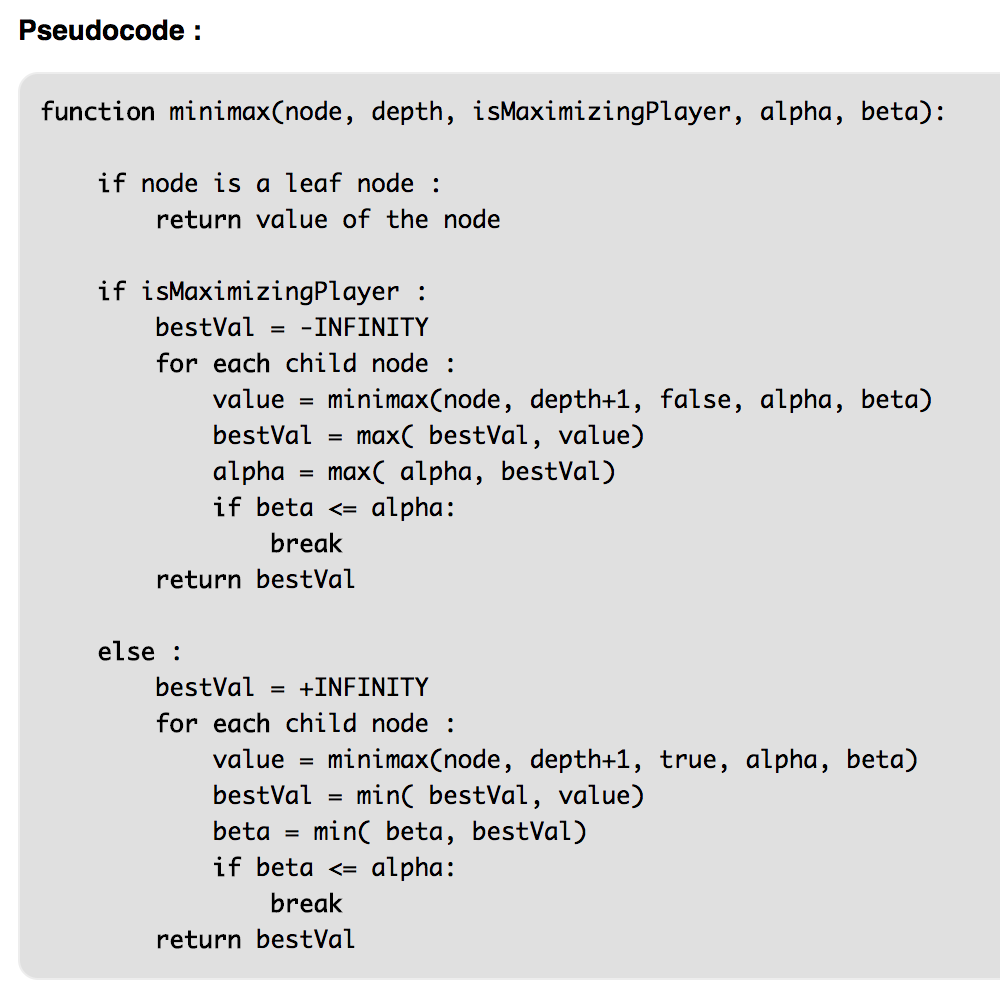
Approach:

My approach was pretty simple and straight forward. For min-max with alpha-beta pruning I used a recursive approach where each time I would call min-max I would set my isMax value to be true for when I am at the “max level” and isMax to false when I am at the “min level”. Then I would have it run for either 30 seconds or a max depth. By using a max depth value, I am essentially taking a depth limited search approach to this. The optimal max depth I found for my program was 4 which I found after trial and error. And I calculated the time by using the java LocalTime class and which uses the actual real-world time. After reaching max depth I would treat that node as a leaf and start to traverse back or I would start to traverse back if I have reached my max time limit. Traversing back just depends on what occurs first. For my evaluation function, I would check for AI win, opponent win, draw state, or inconclusive state. Both the draw state and inconclusive state would return a value of 0, an AI win state would return a value of 100 and an opponent win state would return a value of -100. Then I would either add the depth to opponent win state or subtract the depth from an AI win state. This just means that it would take x amount of moves to reach that state. For alpha-beta pruning I would just simply check to see if the beta value less than or equal to alpha, if it is just break out of the inner for loop and then continue on with min-max. After min-max finishes it will output a value and the check to see if this is the highest value if so then we will return the coordinates of move that will give us that value, then make that move on the board. This process is continued until a win, loss, or draw state has been achived.

Pseudocode that I used to code my min-max method:



Output (last couple of moves):

