Project 3, Fascicle 3:
Better AI

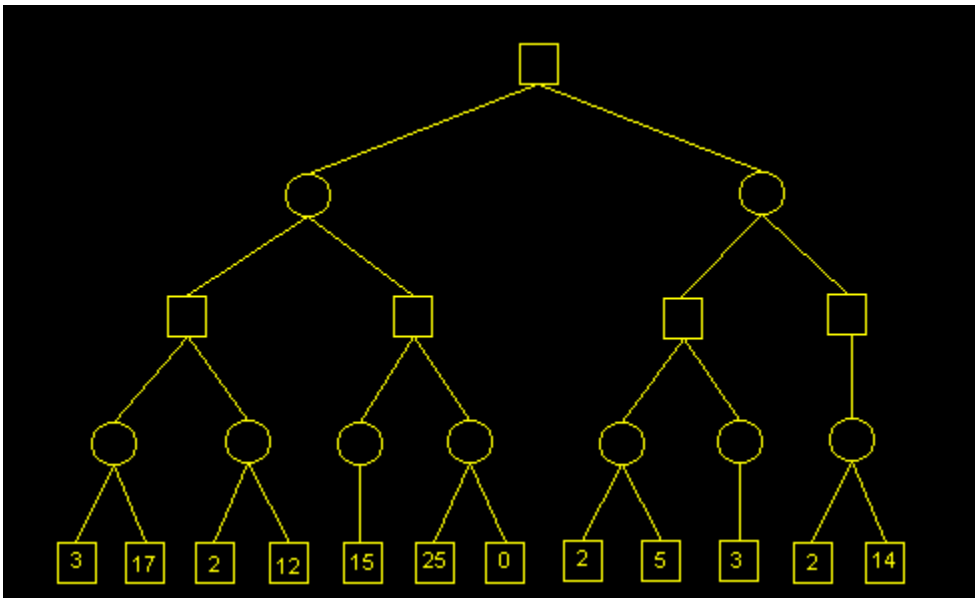
Suggested completion date for this Fascicle: May 8

Chess AI

By now you have hopefully tried integrating your minimax AI into your chess implementation, and likely been disappointed in the result: with our implementation of the game rules, you're probably only able to search to a depth of 2 or 3 without having to wait several-seconds of delay for the AI to find a move. An AI that only looks 2-3 moves ahead is going to make some very poor decisions, so we will try to do a little better. We'll never approach a "real" professional chess AI for several reasons, but we should be able to increase the search depth by 1 or 2, and give the AI better rules for evaluating the weight of board configurations.

Alpha-Beta Pruning

Alpha-beta pruning is an adaptation to search algorithms like Minimax that seeks to reduce the number of nodes searched in the tree, by stopping the evaluation of a move when it is known to be worse than a previously examined move. Consider the following search tree, where square nodes indicate White moves (MAX value desired) and circle nodes indicate Black moves (MIN value)



Examine the lower left circle node at depth 3, which has two possible moves for Black. Of the two moves, Black will choose the one with weight 3 (minimum value), and report that back to the White node above it at depth 2. When we descend to the second Black child of that White node (with the 2 and 12 children), then examine the first possible move for Black, we see a move of weight 2. Upon seeing that value, we consider these facts:

1. The White node at depth 2 has already explored one possible move resulting in a board value of 3.
2. The Black node to the right of the White node now has a possible move leading to a board value of 2.
3. The value of 2 is not necessarily the *best* (smallest) value for Black at that level, but it is smaller than the 3 found in the "cousin" branch.
4. The White node at depth 2, having already found one branch of value 3, will ***never under any circumstance*** choose the right-most branch under its node, in which its opponent will be able to get

an even better board value of ***2 or less***. Therefore, we do not need to apply and evaluate the board resulting in the value of 12, or any other board position in the same sub-tree (if there were any). None of those moves will ever be relevant because White will never actually choose the move leading to this board state.

This series of observations is the key to **alpha-beta pruning**, in which each level of the recursion keeps track of two values α and β indicating the best-known move for White (α) and Black (β) in each subtree. These values help us "prune" our search tree when we find ourselves in situations like before: when White has a current-known best move of α whose value is greater than or equal to Black's best known move of β, then we can abandon a branch of the search tree and reduce the amount of work in our minimax algorithm.

To implement alpha-beta pruning, we modify our `FindBestMove` private implementation to take two additional integers, `alpha` and `beta`, initially set to the smallest and largest integers, respectively, and we pass our current values for these variables to each recursive call. Upon receiving a best move back from a recursive call, we update `alpha` or `beta` if the move's weight is better than the value corresponding to the current player – so if we are maximizing, and the move's weight is better than `alpha`, we update `alpha` and record a new local best move. **If at any time** `alpha` **becomes not-less-than** `beta`, then we prune the search by immediately returning our current best move, with a weight equal to our **opponent's** best score – so `beta` if we are maximizing, and `alpha` otherwise.

And that's it!

Static Board Evaluation

The term **static board evaluation** refers to our concept of **board weight** – an integer value given to a particular board layout, indicating which player has the stronger position, used to guide our AI in selecting best moves. Our current idea for chess weight is to return our chess board's `Value`: the difference between the piece values of the two player's pieces. This works, but we can do better.

This lecture from Cornell University outlines many ideas regarding improving a chess AI:
https://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html. We will use some of the ideas in the Static Board Evaluation Function section to implement our own evaluator. Read the link and then return here.

You will need to write a full `BoardWeight` property that calculates and returns a static board weight using some of the techniques mentioned in the link. Specifically, your chess board weight should sum these values for each player, then subtract White's score minus Black's score:

1. Points for the ownership of each piece, using the same calculation as in `CurrentAdvantage` – in other words, you don't need to calculate this explicitly, you already have it in your `CurrentAdvantage` property.
2. 1 point for each square a pawn has moved forward. (So a pawn that started in a2 and is now in b5 is worth 3 points.)
3. 1 point for each friendly piece that is threatening (protecting) your own knight or bishop. ("Protects.") We will **not** consider whether the threatening piece can actually move to the threatened square, particularly with respect to check/mate – we will just use the simple definition of threatening.
4. A certain number of points for each of your opponent's pieces that you threaten: 1 for knight or bishop; 2 for rook; 5 for queen; 4 for king.

You will want to implement this function as efficiently as possible. LINQ statements can be cute, but if they result in having to do multiple scans through a possible moves list, then you may be better off doing a single manual loop and updating multiple variables during that loop.

What You Need to Do

1. If you have not already, follow Othello's example to implement a computer opponent in your `ChessViewModel`. Make the `BoadWeight` property of your `ChessBoard` class return the integer `Advantage` value of your

`CurrentAdvantage` property (positive if player 1 has the advantage, otherwise negative). Try it out at level 4, and note its slowness.

2. Change your `ChessBoard`'s `BoardWeight` property so it calculates the static board weight described above. Test your code by having your `ChessViewModel` show the board's `BoardWeight` in a `MessageBox` after applying a move.

3. Implement alpha-beta pruning in your `FindBestMove` method. Do not change the signature of the `public` version of the method, only the private overload.