Chess playing program

Harmandeep Mangat
December 17, 2020

Abstract

A chess playing AI that uses iterative deepening with the mini-max algorithm optimized by alpha beta pruning to find the next best move given a current board state.

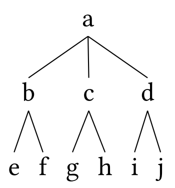
1 Background

1.1 Iterative Deepening

Iterative deepening search is a breadth first search with a limit on depth. A breadth first search starts a the root and explores all its children before moving onto the next depth. With iterative deepening, it limits the depth the breadth first search can search. In the game chess, according to Claude Shannon, an American mathematician, the conservative lower bound of game tree complexity is 10^{120}. This proves that it is impractical to try to generate the entire game tree to find the next best move. This can be solved by a depth limit. Given a board state, search til the depth limit to find the next best move. This increases efficiency and space.

1.2 Mini-max

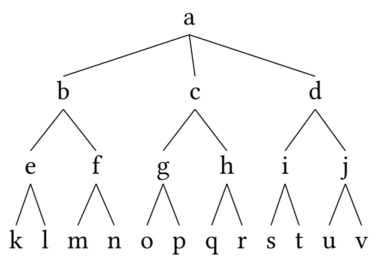
The mini-max algorithm is a recursive algorithm for choosing the next best move in an n-player game. Given a game tree, the algorithm does a depth first search, once a leaf node is found, it returns the value of the state according to the heuristic function. The mini-max algorithm is a turn based algorithm were the maximizing player chooses the node with the greatest value and the minimizing player chooses a node with the lowest value. For example, if the maximizing player was searching for the next best move; at depth of zero (the root), the algorithm would pick the node with the highest value. Then at a depth of 1, the algorithm would pick the node with the lowest value and so on.



Using the tree above, the algorithm would do a depth first search, stopping at "e" first. It will then evaluate "e", and return that value to b. It will then do the same for f. Since at a depth of 1, it is the minimizing players turn, the algorithm would pick the lowest of the two to be the value of b. Once the values of "c" and "d" have been found, at depth of zero, maximizing players turn, it would pick the node with the highest value.

1.2.1 Alpha beta Pruning

Alpha beta pruning is an optimized version of of the mini-max algorithm. It follows the same procedure as the mini-max algorithm but with an additional step of a beta cutoff. Essentially, there would be a alpha and beta variable that would be updated by the maximizing and minimizing player's turn respectively, and once beta is less or equal to alpha, the algorithms breaks, meaning it stops constructing the children nodes of the parent.



For example, if $k = 1$ and $l = 3$, the parents alpha would be 3, and beta would be infinity. Then "e" parent's beta gets set to 3 and the alpha remains a negative infinity. These values are then passed down to "f". If $m = 5$, then is parent's alpha would equal 5. Since beta is less then or equal to alpha, we break. The reason for this is, "f's" parent is searching for the min while "f" itself is searching for max, meaning at a depth of 1, it will never choose to go towards "f". This improves the efficiency and space of mini-max, since less of the tree needs to be constructed.

2 Piece Representation

Each piece has its own class that extends an abstract class. The abstract class has the methods; set name, which sets board representation of the piece name. Such as p for white pawn and N for black knight (capital letter for black, lowercase for white). Set value, which gives the piece a material value and it supplied to the heuristic function. Set en-passant, which is a flag for the pawn piece to indicate that an opposing pawn could use en-passant to capture to capture it. Lastly, set moved, which indicates whether a piece has moved from its starting position. It also has helper methods to retrieve everything that was set as well as the colour of the piece. The classes that hold the individual pieces also provide all the legal moves that piece can make along with the value of moving to a certain square. The pawn class and the king class have additional methods, appropriately setting the en-passant flag and checking to see if the king is in check respectively. To achieve setting the en-passant flag appropriately, after every move, all pawns en-passant are set to false except for if a pawn has moved up two squares. Then only its en-passant is kept true. To check to see if the king is in check, all that is needed to be done is to examine the entire board and see if any piece can capture the king and the return the appropriate value for the heuristic function.

3 Game Representation

The game class holds; an array of pieces that represent the board. The array is a 8x8 2d array that represents the board with each index storing a piece or blank space represented by a "-". It also holds a flag indicating whether white or black is the winner or the game is at a stalemate, check value, number of white pieces, and number of black pieces. The check value is calculated by seeing if the king is in check, if so, its plus ten points or minus 10 points for white or black respectively. The number of white pieces and the number of black pieces are used to determine how far along the game is and change the board scoring appropriately.

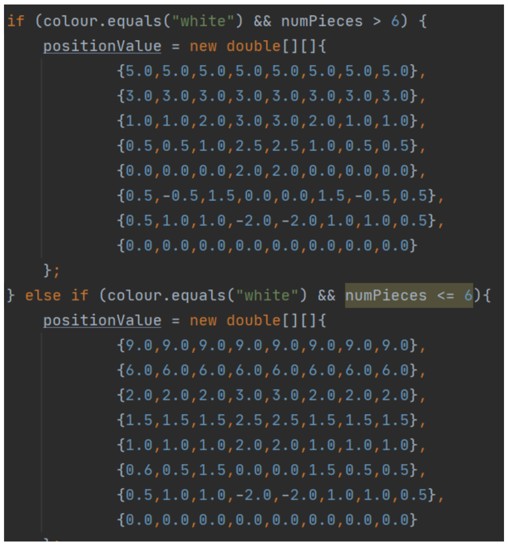
4 Move Representation

All possible moves are generated by looping over the board and using the piece classes to return a legal move in array coordinates. All these moves are then sorted according to the heuristic value. So, white's moves are sorted from highest to lowest and black's moves are sorted from lowest to highest. This is called move ordering which allows alpha beta to prune the tree sooner. Pruning occurs when beta is less then or equal to alpha, so if the best move is always looked at first, it can decrease the time it takes from beta to become less then or equal to aplha.

5 Heuristic

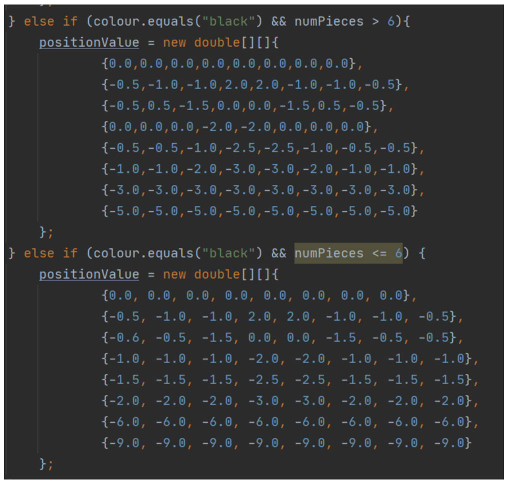
5.1 Board Scoring

5.1.1 Pawn



```
if (colour.equals("white") && numPieces > 6) {
    positionValue = new double[][]{
            {5.0,5.0,5.0,5.0,5.0,5.0,5.0,5.0},
            {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0},
            {1.0,1.0,2.0,3.0,3.0,2.0,1.0,1.0},
            {0.5,0.5,1.0,2.5,2.5,1.0,0.5,0.5},
            {0.0,0.0,0.0,2.0,2.0,0.0,0.0,0.0},
            {0.5,-0.5,1.5,0.0,0.0,1.5,-0.5,0.5},
            {0.5,1.0,1.0,-2.0,-2.0,1.0,1.0,0.5},
            {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}
    };
} else if (colour.equals("white") && numPieces <= 6){
    positionValue = new double[][]{
            {9.0,9.0,9.0,9.0,9.0,9.0,9.0,9.0},
            {6.0,6.0,6.0,6.0,6.0,6.0,6.0,6.0},
            {2.0,2.0,2.0,3.0,3.0,2.0,2.0,2.0},
            {1.5,1.5,1.5,2.5,2.5,1.5,1.5,1.5},
            {1.0,1.0,1.0,2.0,2.0,1.0,1.0,1.0},
            {0.6,0.5,1.5,0.0,0.0,1.5,0.5,0.5},
            {0.5,1.0,1.0,-2.0,-2.0,1.0,1.0,0.5},
            {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}
```

(a) White.



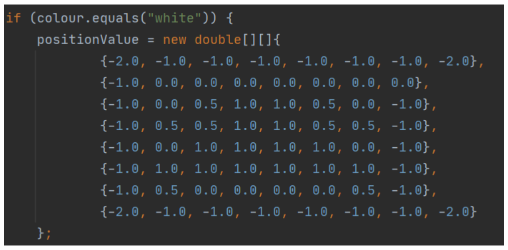
```
} else if (colour.equals("black") && numPieces > 6){
    positionValue = new double[][]{
            {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0},
            {-0.5,-1.0,-1.0,2.0,2.0,-1.0,-1.0,-0.5},
            {-0.5,0.5,-1.5,0.0,0.0,-1.5,0.5,-0.5},
            {0.0,0.0,0.0,-2.0,-2.0,0.0,0.0,0.0},
            {-0.5,-0.5,-1.0,-2.5,-2.5,-1.0,-0.5,-0.5},
            {-1.0,-1.0,-2.0,-3.0,-3.0,-2.0,-1.0,-1.0},
            {-3.0,-3.0,-3.0,-3.0,-3.0,-3.0,-3.0,-3.0},
            {-5.0,-5.0,-5.0,-5.0,-5.0,-5.0,-5.0,-5.0}
    };
} else if (colour.equals("black") && numPieces <= 6) {
    positionValue = new double[][]{
            {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
            {-0.5, -1.0, -1.0, 2.0, 2.0, -1.0, -1.0, -0.5},
            {-0.6, -0.5, -1.5, 0.0, 0.0, -1.5, -0.5, -0.5},
            {-1.0, -1.0, -1.0, -2.0, -2.0, -1.0, -1.0, -1.0},
            {-1.5, -1.5, -1.5, -2.5, -2.5, -1.5, -1.5, -1.5},
            {-2.0, -2.0, -2.0, -3.0, -3.0, -2.0, -2.0, -2.0},
            {-6.0, -6.0, -6.0, -6.0, -6.0, -6.0, -6.0, -6.0},
            {-9.0, -9.0, -9.0, -9.0, -9.0, -9.0, -9.0, -9.0}
    };
```

(b) Black.

Figure 1: Pawn board score

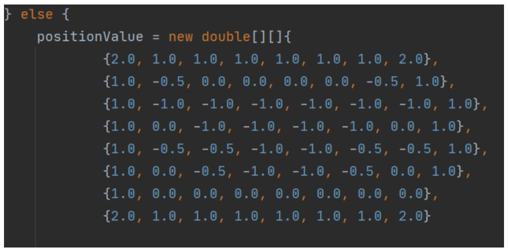
In the beginning of the game, the best move the pawn should make is to take control of the center of the board. As in figure 1-a, the best move for pawn would be moving to either d4 or e4. Since controlling the middle gives more options to other pieces further in the game. Once the game has progressed further, the white has less then 7 pieces remaining, the most advantageous move would be to get a pawn promotion and have more powerful pieces on he board. For black, the score board is inverted and the values are negative since black is tying to minimize the score (see fig 1-b).

5.1.2 Bishop



```
if (colour.equals("white")) {
    positionValue = new double[][]{
            {-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0},
            {-1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
            {-1.0, 0.0, 0.5, 1.0, 1.0, 0.5, 0.0, -1.0},
            {-1.0, 0.5, 0.5, 1.0, 1.0, 0.5, 0.5, -1.0},
            {-1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, -1.0},
            {-1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0},
            {-1.0, 0.5, 0.0, 0.0, 0.0, 0.0, 0.5, -1.0},
            {-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0}
    };
```

(a) White.



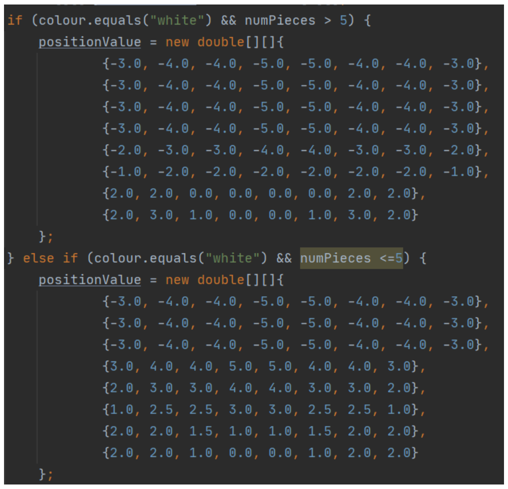
```
} else {
    positionValue = new double[][]{
            {2.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0},
            {1.0, -0.5, 0.0, 0.0, 0.0, 0.0, -0.5, 1.0},
            {1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, 1.0},
            {1.0, 0.0, -1.0, -1.0, -1.0, -1.0, 0.0, 1.0},
            {1.0, -0.5, -0.5, -1.0, -1.0, -0.5, -0.5, 1.0},
            {1.0, 0.0, -0.5, -1.0, -1.0, -0.5, 0.0, 1.0},
            {1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
            {2.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0}
```

(b) Black.

Figure 2: Bishop board score

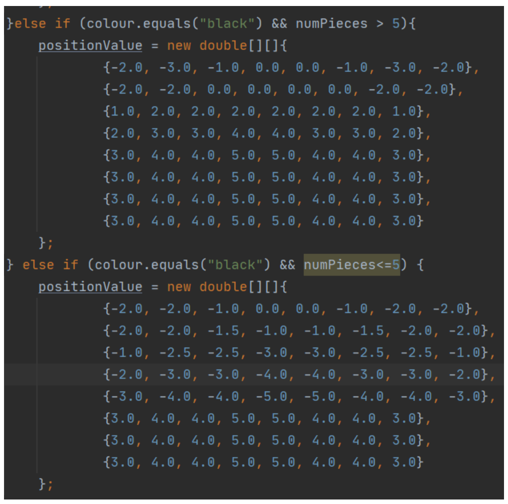
The most advantageous squares for the bishop would be around the center, avoiding the ends. This allows the bishop to have more options available to it when it needs to move. It gives it a higher probability to be used efficiently as it could take more pieces or move into a better position to give the player an advantage further into the game(see fig 2).

5.1.3 King



```java
if (colour.equals("white") && numPieces > 5) {
    positionValue = new double[][]{
            {-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0},
            {-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0},
            {-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0},
            {-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0},
            {-2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0},
            {-1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0},
            {2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 2.0, 2.0},
            {2.0, 3.0, 1.0, 0.0, 0.0, 1.0, 3.0, 2.0}
    };
} else if (colour.equals("white") && numPieces <=5) {
    positionValue = new double[][]{
            {-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0},
            {-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0},
            {-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0},
            {3.0, 4.0, 4.0, 5.0, 5.0, 4.0, 4.0, 3.0},
            {2.0, 3.0, 3.0, 4.0, 4.0, 3.0, 3.0, 2.0},
            {1.0, 2.5, 2.5, 3.0, 3.0, 2.5, 2.5, 1.0},
            {2.0, 2.0, 1.5, 1.0, 1.0, 1.5, 2.0, 2.0},
            {2.0, 2.0, 1.0, 0.0, 0.0, 1.0, 2.0, 2.0}
    };
```

(a) White.



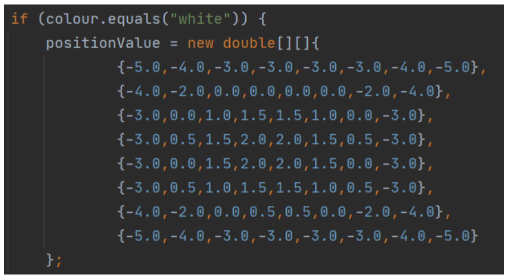
```java
}else if (colour.equals("black") && numPieces > 5){
    positionValue = new double[][]{
            {-2.0, -3.0, -1.0, 0.0, 0.0, -1.0, -3.0, -2.0},
            {-2.0, -2.0, 0.0, 0.0, 0.0, 0.0, -2.0, -2.0},
            {1.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 1.0},
            {2.0, 3.0, 3.0, 4.0, 4.0, 3.0, 3.0, 2.0},
            {3.0, 4.0, 4.0, 5.0, 5.0, 4.0, 4.0, 3.0},
            {3.0, 4.0, 4.0, 5.0, 5.0, 4.0, 4.0, 3.0},
            {3.0, 4.0, 4.0, 5.0, 5.0, 4.0, 4.0, 3.0},
            {3.0, 4.0, 4.0, 5.0, 5.0, 4.0, 4.0, 3.0}
    };
} else if (colour.equals("black") && numPieces<=5) {
    positionValue = new double[][]{
            {-2.0, -2.0, -1.0, 0.0, 0.0, -1.0, -2.0, -2.0},
            {-2.0, -2.0, -1.5, -1.0, -1.0, -1.5, -2.0, -2.0},
            {-1.0, -2.5, -2.5, -3.0, -3.0, -2.5, -2.5, -1.0},
            {-2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0},
            {-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0},
            {3.0, 4.0, 4.0, 5.0, 5.0, 4.0, 4.0, 3.0},
            {3.0, 4.0, 4.0, 5.0, 5.0, 4.0, 4.0, 3.0},
            {3.0, 4.0, 4.0, 5.0, 5.0, 4.0, 4.0, 3.0}
    };
```

(b) Black.

Figure 3: King board score

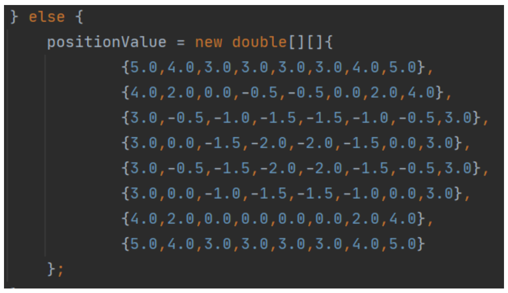
In beginning of the game, the king needs to stay in the back where it can be protected by the other pieces. So the board scoring is higher for row 1 and 2, insuring the king stays in the back. But as the game progresses and there are only 5 or less pieces remaining, the kings will have to move towards the middle of the board to support the remaining pieces and also to be protected by them. Being in the middle of the board allows the surrounding pieces to both attack and defend since they have more options to move to.

5.1.4 Knight



```java
if (colour.equals("white")) {
    positionValue = new double[][]{
            {-5.0,-4.0,-3.0,-3.0,-3.0,-3.0,-4.0,-5.0},
            {-4.0,-2.0,0.0,0.0,0.0,0.0,-2.0,-4.0},
            {-3.0,0.0,1.0,1.5,1.5,1.0,0.0,-3.0},
            {-3.0,0.5,1.5,2.0,2.0,1.5,0.5,-3.0},
            {-3.0,0.0,1.5,2.0,2.0,1.5,0.0,-3.0},
            {-3.0,0.5,1.0,1.5,1.5,1.0,0.5,-3.0},
            {-4.0,-2.0,0.0,0.5,0.5,0.0,-2.0,-4.0},
            {-5.0,-4.0,-3.0,-3.0,-3.0,-3.0,-4.0,-5.0}
    };
```

(a) White.



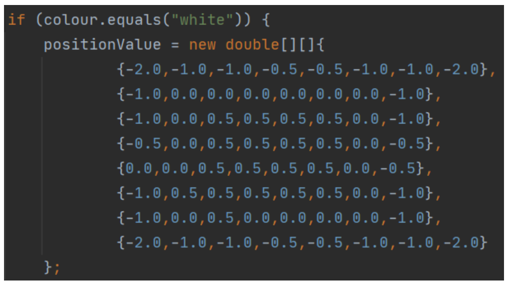
```java
} else {
    positionValue = new double[][]{
            {5.0,4.0,3.0,3.0,3.0,3.0,4.0,5.0},
            {4.0,2.0,0.0,-0.5,-0.5,0.0,2.0,4.0},
            {3.0,-0.5,-1.0,-1.5,-1.5,-1.0,-0.5,3.0},
            {3.0,0.0,-1.5,-2.0,-2.0,-1.5,0.0,3.0},
            {3.0,-0.5,-1.5,-2.0,-2.0,-1.5,-0.5,3.0},
            {3.0,0.0,-1.0,-1.5,-1.5,-1.0,0.0,3.0},
            {4.0,2.0,0.0,0.0,0.0,0.0,2.0,4.0},
            {5.0,4.0,3.0,3.0,3.0,3.0,4.0,5.0}
    };
```

(b) Black.

Figure 4: Knight board score

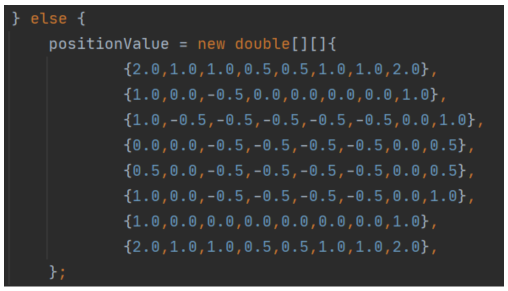
It is more advantageous to have the knight play around in the middle of the board. This allows it to both defend itself of attack since it has more options available to it to move to. Therefore the scoring around the middle of the board is better then at the ends (see fig 4).

5.1.5 Queen



```
if (colour.equals("white")) {
    positionValue = new double[][]{
            {-2.0,-1.0,-1.0,-0.5,-0.5,-1.0,-1.0,-2.0},
            {-1.0,0.0,0.0,0.0,0.0,0.0,0.0,-1.0},
            {-1.0,0.0,0.5,0.5,0.5,0.5,0.0,-1.0},
            {-0.5,0.0,0.5,0.5,0.5,0.5,0.0,-0.5},
            {0.0,0.0,0.5,0.5,0.5,0.5,0.0,-0.5},
            {-1.0,0.5,0.5,0.5,0.5,0.5,0.0,-1.0},
            {-1.0,0.0,0.5,0.0,0.0,0.0,0.0,-1.0},
            {-2.0,-1.0,-1.0,-0.5,-0.5,-1.0,-1.0,-2.0}
    };
```

(a) White.



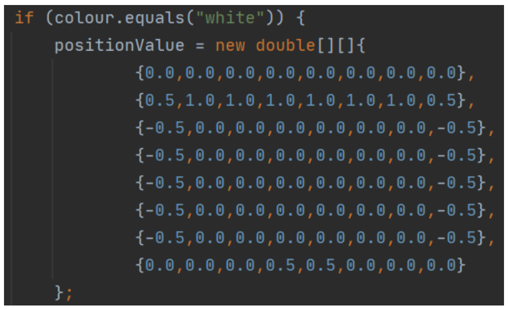
```
} else {
    positionValue = new double[][]{
            {2.0,1.0,1.0,0.5,0.5,1.0,1.0,2.0},
            {1.0,0.0,-0.5,0.0,0.0,0.0,0.0,1.0},
            {1.0,-0.5,-0.5,-0.5,-0.5,-0.5,0.0,1.0},
            {0.0,0.0,-0.5,-0.5,-0.5,-0.5,0.0,0.5},
            {0.5,0.0,-0.5,-0.5,-0.5,-0.5,0.0,0.5},
            {1.0,0.0,-0.5,-0.5,-0.5,-0.5,0.0,1.0},
            {1.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0},
            {2.0,1.0,1.0,0.5,0.5,1.0,1.0,2.0},
    };
```

(b) Black.

Figure 5: Queen board score

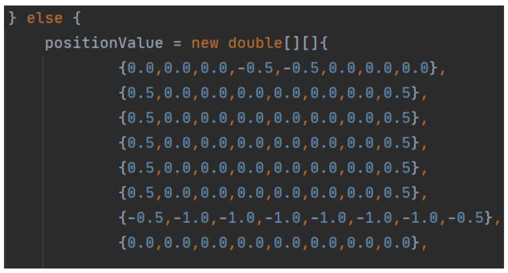
The scoring around the inside of the board is better the the outside, this allows the queen to have more options to move too(see fig 5). But the scoring is worse compared to the other pieces, this ensures the queen only moves if no other piece gives a better result, ensuring the queen doesn't get put into a vulnerable position early on in the game since it is one of the strongest pieces on the board.

5.1.6 Rook



```
if (colour.equals("white")) {
    positionValue = new double[][]{
            {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0},
            {0.5,1.0,1.0,1.0,1.0,1.0,1.0,0.5},
            {-0.5,0.0,0.0,0.0,0.0,0.0,0.0,-0.5},
            {-0.5,0.0,0.0,0.0,0.0,0.0,0.0,-0.5},
            {-0.5,0.0,0.0,0.0,0.0,0.0,0.0,-0.5},
            {-0.5,0.0,0.0,0.0,0.0,0.0,0.0,-0.5},
            {-0.5,0.0,0.0,0.0,0.0,0.0,0.0,-0.5},
            {0.0,0.0,0.0,0.5,0.5,0.0,0.0,0.0}
    };
```

(a) White.



```
} else {
    positionValue = new double[][]{
            {0.0,0.0,0.0,-0.5,-0.5,0.0,0.0,0.0},
            {0.5,0.0,0.0,0.0,0.0,0.0,0.0,0.5},
            {0.5,0.0,0.0,0.0,0.0,0.0,0.0,0.5},
            {0.5,0.0,0.0,0.0,0.0,0.0,0.0,0.5},
            {0.5,0.0,0.0,0.0,0.0,0.0,0.0,0.5},
            {0.5,0.0,0.0,0.0,0.0,0.0,0.0,0.5},
            {-0.5,-1.0,-1.0,-1.0,-1.0,-1.0,-1.0,-0.5},
            {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0},
```

(b) Black.

Figure 6: Rook board score

It is more advantageous for the rook to play on the opponents side of the board. This allows it to attack pieces from behind, attack the enemy king, and protect a pawn for pawn promotion. But if those squares are unavailable or moving there would let the opponent capture the rook, then it would be better playing on the inside of the board where it has more options to move too.

5.2 Piece Value

5.2.1 White

Pawn:	10
Bishop:	30
King:	900
Knight:	30
Queen:	90
Rook:	50

5.2.2 Black

Pawn: -10
Bishop: -30
King: -900
Knight: -30
Queen: -90
Rook: -50

5.3 Check Value

If white king is in check, black get -10 points. If black king is in check white gets +10 points. Otherwise check value is zero.

5.4 Heuristic Function

Total Score = Board Scoring + Piece Value + Check Value

5.5 Stalemate Value

If the white's move results in a stalemate and according to the heuristic white is winning, the heuristic value become -1000. But is white is losing, then the heuristic value becomes 1000. This is used to ensure if white is winning, it is disadvantageous to have the game result in a stalemate, but if it is losing, it has a less likely chance of winning. Therefore making the game end in a stalemate is more advantageous.It would be a similar situation for black, if it was winning, the heuristic equals 1000, but if it is losing, the heuristic equals -1000.

6 Checkmate Value

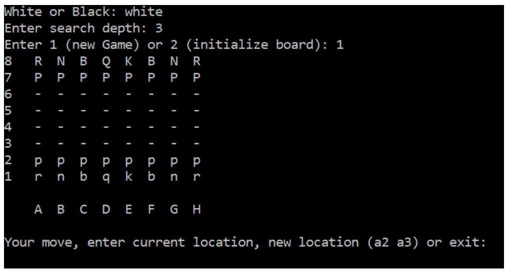
if white can get a checkmate in one move, the heuristic value equals 10,000, else if equals 5000. If black's can get a checkmate in one move, the heuristic value becomes -10,000, else it becomes -5000. This enforces the algorithm to choose the next move to be a checkmate move instead of picking a checkmate that may occur some number of moves ahead.

6.1 Heuristic Penalty

There are times when the AI gets stuck in an infinite loops, where the best move keeps repeating. In order to compensate for this, a penalty is imposed. If a move has been repeated three or more times in a row, the next best move is chosen. This opens up more possibilities and allows the game to advance.

7 Running Instructions

Once the code is run, it will ask the user what colour they want to play as, white or black. Then it will ask the search depth, how far down should the mini-max optimized with alpha beta should look. Finally it will was whether you want to play a new game of initialize board. A new game places all the pieces in their designated location while initialize board will ask the user to place the pieces on the board (see fig 7).

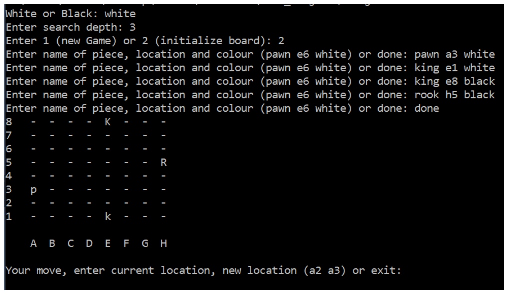


```
White or Black: white
Enter search depth: 3
Enter 1 (new Game) or 2 (initialize board): 1
8   R  N  B  Q  K  B  N  R
7   P  P  P  P  P  P  P  P
6   -  -  -  -  -  -  -  -
5   -  -  -  -  -  -  -  -
4   -  -  -  -  -  -  -  -
3   -  -  -  -  -  -  -  -
2   p  p  p  p  p  p  p  p
1   r  n  b  q  k  b  n  r

    A  B  C  D  E  F  G  H

Your move, enter current location, new location (a2 a3) or exit:
```

(a) New game.



```
White or Black: white
Enter search depth: 3
Enter 1 (new Game) or 2 (initialize board): 2
Enter name of piece, location and colour (pawn e6 white) or done: pawn a3 white
Enter name of piece, location and colour (pawn e6 white) or done: king e1 white
Enter name of piece, location and colour (pawn e6 white) or done: king e8 black
Enter name of piece, location and colour (pawn e6 white) or done: rook h5 black
Enter name of piece, location and colour (pawn e6 white) or done: done
8   -  -  -  -  K  -  -  -
7   -  -  -  -  -  -  -  -
6   -  -  -  -  -  -  -  -
5   -  -  -  -  -  -  -  R
4   -  -  -  -  -  -  -  -
3   p  -  -  -  -  -  -  -
2   -  -  -  -  -  -  -  -
1   -  -  -  -  k  -  -  -

    A  B  C  D  E  F  G  H

Your move, enter current location, new location (a2 a3) or exit:
```

(b) Initialize Board.

Figure 7: Running Example

8 Bibliography

freeCodeCamp.org, "A step-by-step guide to building a simple chess AI," freeCodeCamp.org, 15-Mar-2017. [Online]. Available: https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/. [Accessed: 07-Dec-2020].

"Shannon number," Wikipedia, 04-Nov-2020. [Online]. Available: https://en.wikipedia.org/wiki/Shannon_number. [Accessed: 07-Dec-2020].