Chess AI Solution

Nicholas Golini

October 24, 2016

1 Introduction

The goal of this assignment is to create a Chess AI that can play a human, or another chess bot, and can make intelligent moves. We were provided with an open source chess library called chesspresso, written by Bernhard Seybold, which provided us the board GUI, a chess board object representation (Position), a getAllMoves() function to get all the legal moves that the player can make in a given turn, and much more. The Position object is used to represent the state of the board in FEN, which is a short hand way to represent a chess board in a one line string. The getAllMoves() function acts as a getSuccessors() function, as it finds all of the possible moves a player can make during a turn, thus representing all the possible states the board can be in after a move.

The way we can get the AI to choose the best move from a given position is to create a tree of all possible positions resulting from each available move given by getAllMoves(), where the edges of the tree are moves from one board position to another, and the nodes on the tree are the resulting position of the game board after a move. Each node will also have a value attached to it, where the value is an integer representation of how good a given position is for the bot. The goal of the bot is to find a move from a current position that maximizes the bot's values in future moves. It does this by finding the move that minimizes the opponents best move (in Minimax, it finds the move with the maximum value for Min, as Min is looking for the smallest number possible). For example, if there is a move from the current position that results in the bot's opponent taking its queen and another move results in no captures, the bot will choose the move resulting in no captures because the queen capture would decrease the bot's board value and increase its opponent's value more than the move that results in no captures. Since the chess branching factor is considered to be non-trivial, we cannot search this tree until we reach a terminal position (a checkmate or draw) for every move, thus we can only search a finite number of moves in advance. The more moves the algorithm is able to look at, the better the move returned by the algorithm will be because it is able to foresee more moves in advance, such as possible traps or a move that may have an immediate decrease in value, but may be good for a long term strategy. For example, some moves may seem like a good choice when looking two or three moves in advance, but may lead to possible checkmates later in the game, so as the depth of the search tree increases, the algorithm is able to foresee these future traps and choose better moves. One algorithm used to find the best move is the Minimax algorithm, which will be discussed later in the report. We can then implement alpha-beta pruning on the Minimax algorithm to be able to search deeper in the tree which will yield even better moves for the bot.

2 Cutoff and Evaluation Function

Before getting into the Minimax algorithm, it is important to understand how to tell if a game is over (whether it is due to a draw or checkmate) and to know how to evaluate the value of a position during a game. First, we will look at how to know when to stop the Minimax search.

Cutoff test: The three times we want to cut off a search is: 1) when a checkmate is found, 2) when a draw is found, and 3) when the current depth of the search is greater than the maximum depth to search. Chesspresso provides a function position.isMate(), which returns true if the current player is in a checkmate position, and false if not. It also has a function position.isTerminal(), which returns true if the current player cannot make any moves, and false if the player can move. If isMate() returns true, then the game is over and the search has reached a leaf node, thus the search should be cutoff. To handle draw situations, we must check if isTerminal() is true, and isMate() is false. This is because if there is a checkmate, then the current player cannot make a legal move, so isTerminal will also be true, so in order to detect when a draw occurs, we want to check if isTerminal() is true, and isMate() is false. If a draw is found, then the search has reached a leaf node, and should cease. The last situation for search cutoff is simply if the current depth of the search is greater than the maximum depth the program is allowed to search for. Since the branching factor of chess is so large, it is only reasonable to search a few plys in advance of a position, rather than search exclusively until a checkmate or draw is found, so this last cutoff test is important in having a reasonable runtime for the algorithms.

Evaluation function: When a cutoff position is reached, we want to calculate the value of the position. If the cutoff position is a checkmate in which the bot will win, we want to return infinity (in practice a very large int) to indicate that the move is going to end the game and so the algorithm is guaranteed to choose that move. If the checkmate would make the bot's opponent win, then return negative infinity (in practice a very negative int) to indicate that the bot's opponent will beat it if the bot makes that move, so the bot will never choose a move that puts it into checkmate. For draw conditions, return 0 because neither the bot or its opponent can win. For any other position, a basic evaluation function for chess is to calculate the sum of the differences between the number of each kind of piece between two players, Max and Min, and assign a unique weight for each piece. The weights I used for the pieces are: 1 for pawn, 3 for bishops and knights, 5 for rooks, 9 for the queen, and 200 for the king. These weights are determined based off of their relative importance to a player, and have been determined by chess experts studying the game for years. Generally speaking, pawns are less valuable than bishops and knights, which are less valuable than rooks, which are less valuable than queens. The value of the king should be greater than the sum of all of the other pieces because you should never sacrifice your king, since taking a king ends the game. To calculate the value of a position in my chess AI, I loop through the FEN string representation of the position, and count up the number of each piece for Min and Max in a switch statement. I then do the following computation, where the variables with an underscore before it represents Max's pieces, and the variables without the underscore represents Min's pieces:

$$score = (\_p - p) + 3 * (\_n - n + \_b - b) + 5 * (\_r - r) + 9 * (\_q - q) + 200 * (\_k - k);$$

By calculating the difference between the number of pieces for Max minus the number of pieces for Min, we are using the assumption that a positive score means Max has the advantage, and a negative score means that Min has the advantage. Given these cutoff and evaluation functions, we can then run the Minimax/Alpha Beta Pruning algorithms.

3 Minimax

The Minimax algorithm is used to find an optimal decision in a game between two players, Max and Min, where the AI bot is Max, and its opponent is Min. It creates a tree where the root is the current position of the board, and the children of each node is the position after a given move and represents a position in which the other play will move from. For example, the root represents Max's position on the board, and all of its children represent a move made by Max, thus a positions for Min to move from. Then all of these node's children represent moves made by Min, thus the positions for Max to move from. Thus this tree represents all of the possible positions that can occur given a current position and a certain number of moves. As explained above, a positive evaluation for a position is ideal for Max, and a negative evaluation for a position is ideal for Min (which is why the two players are named Max and Min). Therefore, Max wants to choose a move that will yield the largest evaluation for any of Min's moves after Max makes its moves (minimizing the potential of Min making a good move). As explained in the introduction of this report, the further the search gets in terms of number of moves looked at, the better the chosen move will be. Thus we want to find the value of a position as deep in the search tree as we can, in which the node will be a leaf (a checkmate or draw), or a node at the maximum depth of the search, and percolate the maximum values up to the top of the tree.

The Minimax recursion is split up into two stages: one where the nodes are evaluated down to the leaves or nodes at the max search depth, and one where the max values of the leaves are backed up to the nodes' parents. In the first stage of the recursion, the tree is recursed all the way down to the leaf and cutoff nodes of the search. The values for these nodes are then calculated, and the second stage of recursion backs up these values to the root, keeping the maximum value from the children if the parent represents Max's turn, and keeping the minimum value from the children if the parent represents Min's turn. To compute this, we have two functions, MaxValue, and MinValue, which take in a position and a depth. Let us look at MaxValue first. If the current position is a leaf or cutoff node, then the function will return the value of the position. If it is not, then we take the maximum value calculated by MinValue() after Max makes a move, which is how Max nodes get the maximum value from its children, which represent Min moves. The code for MaxValue is below:

```
1  public int MaxValue(Position curr, int depth, int currMax) throws IllegalMoveException {
2    if(cutoff(curr, depth, currMax)) {
3      int ut = UtilityFunc(curr, depth);
4      return ut;
5    }
6    int v = NEGINF;
7    depth++;
8    for(short currMove : curr.getAllMoves()){
9      v = Math.max(v, MinValue(Result(curr, currMove), depth, currMax));
10     curr.undoMove();
11   }
12   return v;
13 }
```

MinValue looks almost identical, but initializes v as positive infinity, and line 9 becomes:

$$v = Math.min(v, MaxValue(Result(curr, currMove), depth, currMax));$$

Since Min is trying to be as negative as possible, it is looking for the minimum value of all of its children, i.e. it's best move, which are Max turns.

The way that this algorithm is started, is by calling Minimax on all of the root's possible moves. This calculates the values of the different moves that min can make. The node with the smallest value represents the position that minimizes Min's possible values after moving (which is good for Min, bad for Max), and the node with the largest value represents the position that maximizes Min's possible values (which is bad for Min, and good for Max). Thus we choose the move that yields the largest value for Min, thus minimizing Min's potential for a good move. The for loop in the code below is where Minimax is called on the root's children and where the largest value for the children are kept (the while loop around it is for iterative deepening, which will be discussed later). If the return value of Minimax for a move is greater then the current best move, update the best move to represent the move with a larger Minimax value. At the end of the for loop, the best move found in Minimax will be stored in the bestMove variable.

```
public short MiniMax(Position pos) throws IllegalMoveException {
  int currV;
  int largestV = NEGINF;
4     int i;
5     short bestMove = 0;
6     int currMax = 0;
7     Position temp = pos;
8     while(currMax <= MAXDEPTH) {
9       i = 0;
10      pos = temp;
11      for (short currMove : pos.getAllMoves()) {
12        pos.doMove(currMove);
13        currV = MinValue(pos, 0, currMax);
14        if (currV > largestV) {
15          largestV = currV;
16          bestMove = currMove;
17        }
18        pos.undoMove();
19        i++;
20      }
21      currMax++;
22    }
23    return bestMove;
24  }
```

This function also utilizes iterative deepening. The while loop in line 8 is where the iterative deepening is implemented. It basically calls the for loop for Minimax with incrementing max depths. Iterative Deepening is mainly implemented here if checkmates are found in the search, because if they are, we want to find teh checkmate at the shallowest depth. Since Minimax is a depth first approach and checkmates all have the same value of infinity (10,000 in my code), the move that finds a checkmate first will always be chosen by the program without iterative deepening, even if a checkmate is found at a shallower depth than the first checkmate found. For example, if the first move from the root finds a checkmate at a depth of 4, meaning 2 moves each, and then another move results in a checkmate after a depth of 0, meaning one move by Max, the move that found the checkmate first will be chosen and the immediate possible forced checkmate will not be chosen. In iterative deepening, however, the shallowest checkmate will be found first, resulting in a move that forces a checkmate with as few moves as possible. Once a checkmate is found, the move to make that checkmate is returned, making it so that the algorithm does not have to keep searching for a better move, since there will be no better move.

The Minimax algorithm runs quite slowly at a depth of 4 (4 plys, 2 turns each player). For the opening move, there are 20 moves possible, and it takes around 30 seconds to compute the best move. For the opening move at a depth of 4, the Minimax algorithm visited 9254371 nodes, which is an awful lot. This is including the iterative deepening, so the number of nodes visited is compounded with the iterative deepening, but still, that is way too much considering how many nodes are transpositions of each other and unnecessary to visit. The number of nodes visited will decrease tremendously with the implementation of alpha-beta pruning and a transposition table.

4 Alpha-Beta Pruning

Alpha-Beta pruning is a means of improving the runtime of Minimax. If the runtime of Minimax is improved by alpha-beta pruning, we can then run a search to a depth deeper than the max depth used in Minimax, which will result in better moves made by the AI. Instead of keeping the max/min of each move, like Minimax does, Alpha-Beta pruning keeps two variables, alpha and beta. Alpha is the value of the best move found so far for Max, and beta is the value of the best move found so far for Min. By keeping these values, we keep an upper bound for values that would be good for Max, and a lower bound for values that would be bad for Max. For MaxValue, when a new v (score) is calculated, we check if alpha is less than v, and if it is then we found a new upper bound for Max and we update alpha to equal v in order to represent this upper bound. Similarly for MinValue, when a new score is calculated, we choose the minimum between beta and the new score to be the new beta, setting the lower bound lower if the new score is less than the current beta. The alpha-beta pruning algorithm is then started the same was as Minimax, and it recurses down to the tree's leaves first, then backs the alpha/beta values up to the root. The root then chooses the move with the largest value the same way Minimax chooses the best move. By keeping alpha and beta during this recursion, we only have to examine nodes that will have a larger alpha, or a smaller beta. This is because nodes found with values inside the range of alpha-beta will never have values larger than alpha, or smaller than beta, so they would never be considered to be best moves by Max or Min and do not have to be considered. By pruning out these moves, alpha-beta search can skip the analysis of up to half the nodes, thus can search up to double the depth that Minimax can search to. Below is my code for alpha-beta pruning. One note to make first is that I simplified the recursion by putting it all in one function, rather than having separate functions for MaxValue and MinValue. Lines 13-28 are the equivalent to MaxValue, and lines 29-45 are the equivalent to MinValue.

```
1  public int AlphaBetaSearch(Position pos, int depth, int a, int b, boolean maxPlayer, int
2    int v;
3    if(transpoTable.containsKey(pos.getHashCode())){
4       if((MAXDEPTH - depth) <= transpoTable.get(pos.getHashCode()).get(1)) {
5          v = transpoTable.get(pos.getHashCode()).get(0);
6          return v;
7       }
8     } else if(cutoff(pos, depth, currMax)){
9     v =  UtilityFunc(pos);
10    return v;
11    }
12    depth++;
13    if(maxPlayer){
14       v = NEGINF;
15       for(short currMove : pos.getAllMoves()){
16          Result(pos, currMove);
17          v = Math.max(v, AlphaBetaSearch(pos, depth, a, b, !maxPlayer, currMax));
18          a = Math.max(a, v);
19          pos.undoMove();
20          if(b <= a){
21            break;
22          }
23       }
24       ArrayList<Integer> newVals = new ArrayList<>();
25       newVals.add(0, v);
26       newVals.add(1, MAXDEPTH - depth);
27       transpoTable.put(pos.getHashCode(), newVals);
28       return a;
29    } else{
30        v = INF;
31        for(short currMove : pos.getAllMoves()){
32          Result(pos, currMove);
33          v = Math.min(v, AlphaBetaSearch(pos, depth, a, b, !maxPlayer, currMax));
34          b = Math.min(b, v);
35          pos.undoMove();
```

```
            if(b <= a){
                break;
            }
        }
        ArrayList<Integer> newVals = new ArrayList<>();
        newVals.add(0, v);
        newVals.add(1, MAXDEPTH - depth);
        transpoTable.put(pos.getHashCode(), newVals);
        return b;
    }
}
```

5 Transposition Table

In the above function, a transposition table is being used. A transposition table is a hash table that keeps a position as a key, and the position's value after being pruned as its value. The value also contains the depth at which this value was found. The value for my implementation of the transposition table keeps a Long as its key, which is calculated by the chesspresso .getHashCode() function, and an arraylist of two integers as the value, where index 0 in the list is the value of that position, and index 1 is the depth at which that value was found. The transposition table is useful in chess, because there are many ways to get to the same position in chess (these are called transpositions). Without a transposition table, positions will be reanalyzed in Minimax and Alpha-Beta pruning if they are found by different paths, so keeping a transposition table makes it so that the algorithm does not have to reanalyze the same position twice. At the beginning of the alpha-beta search, we check to see if the position has already been analyzed (if it's in the transposition table), and if it has been, then we use the value associated with that position from the table. One important note here too is that we also do a check to see if the previously calculated value was from a depth further away from the maxdepth than the current depth is, because the value is more valuable to the search if it is further from the max depth because it will have more information factored into the value, and thus be more accurate. If the position is not in the transposition table, then we run alpha beta pruning on it, and when we get a final value, we add the position to the table with the calculated alpha-beta value so that if we get to the same position later in the search, we do not have to run the complete alpha-beta pruning on it. This should speed up the code enough to be able to search a depth or two even more than alpha-beta search without the transposition table.

For the opening move, the number of nodes visited with alpha-beta pruning and a transposition table was 176883, which is smaller by a factor of 50. This shows how much more efficient alpha-beta pruning is with a transposition table in comparison to Minimax. Alpha-Beta can run to a depth of 6 in the same time as Minimax took to run to a depth of 4, and AB runs almost instantaneously at a depth of 4, compared to the 30 seconds Minimax took.

6 Extensions

6.1 Iterative Deepening

The iterative deepening aspect of my code is mainly discussed in the Minimax section of this report. Iterative deepending is certainly not necessary to a chess AI, but it does make it better at ending games. As a chess game is ending, there are more opportunities to force checkmates and there may be many checkmates found within the Minimax or alpha-beta search, but in order to make it hard for the opponent to evade the checkmate, we want to choose a checkmate found at the shallowest depth, which is the purpose of Iterative Deepening.

6.2 Related Work

I read a very interesting 2012 paper by Mark Montoya called ”A Bried Survey of Chess AI: Strategy and Implementation”, which discussed the problems with computing moves based on the basic chess algorithms commonly used, which are Alpha-Beta pruning with Iterative Deepening and the utilization of a Transposition table. The article then went into the newer research being done to improve chess AIs to specifically beat grandmasters. The proposed methods of better computation talked about in the article are machine learning algorithms, neural networks, and genetic algorithms. The paper first talks about how long term strategy in chess is extremely important, and is very hard to teach an AI. As grandmasters have played thousands and thousands of games of chess, they have much experience setting up positions for a long term strategy, while computer AI searches such as my own assignment have no prior knowledge of chess and choose moves based on searching possible moves from each position to a certain depth. These kinds of searches generally do not make it to the end of the game, especially at the beginning of the game, because the branching factor for chess is so large. One strategy the author talks about is having a preprocessing long-term strategy function to run before the search, so that the AI can search for moves that apply to the long term strategy found in the preprocessing stage. When ”Deep Blue” was created, which was the first machine to to beat a grandmaster with huge computational strength, and it was able to win due to the highly optimized search algorithms and computational ability. Deep Blue was not implemented with long-term strategies or anything, it was just able to calculate a search of the moves up to 40-plys in advance of the position. To give a sense of how crazy this is, my code took around 30 minutes to compute 5 of 20 moves from the opening position at a depth of 8-ply.

One important note that this paper brings up is that the Chess AIs using brute force, such as Deep Blue, do not know why some moves are better than others, and cannot use a sense of the game. One example brought up is that a human may not need much processing to realize a move is bad, but the computer may have to search a few plys deep to see that the move is going to be bad. There is no ability to sense how good a move is other than the evaluation function being used in the AI, which is why brute force algorithms need to reach depths of 40-ply to be able to beat grandmasters, which takes a huge amount of processing. Some ideas to improve this sense of the game include some kind of machine learning algorithms, which require some computing to understand the feel of the game, but these algorithms have produced poor results simply due to the huge number of possible states in a chess game (around 10^{46}). One way to improve on this is to use genetic algorithms, which basically take in data on how grandmasters have played games of chess, and try to mimic how a grandmaster would play. It is possible to have a bunch of different AIs to play each other in a round-robin style tournament, and having the winners move on and learn from each other to get even better, so that at the end of the tournament, the winner will be a combination of the strongest aspects of the rest of the AIs. This ”survival of the fittest” strategy is why this method of developing a chess AI is part of a genetic algorithm.

Source:
Montoya, Mark. ”A Brief Survey of Chess AI: Strategy and Implementation.” University of New Mexico, 2012.

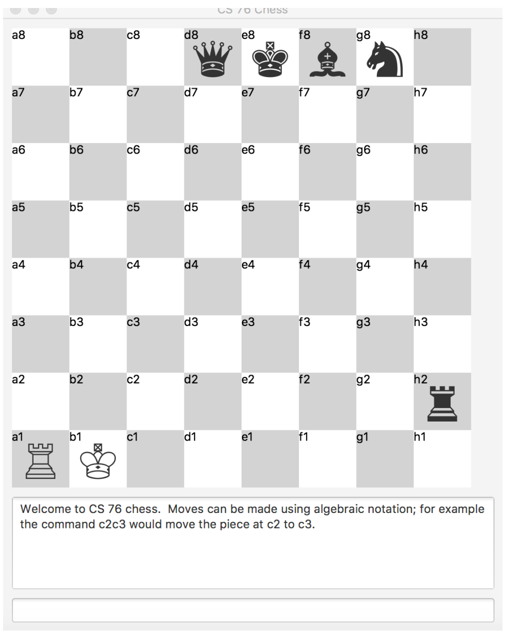
6.3 Profiling

I will focus on the Alpha-Beta pruning AI during this profiling, because Alpha-Beta pruning is an extension to make Minimax faster, and I want to now look at how I could have made my Alpha-Beta pruning even faster. The slowest part of my alpha-beta pruning AI is the alpha-beta search itself, especially implemented with Iterative Deepening. Iterative deepening is necessary for the AI if a checkmate is found at multiple places in the search because in chess, you want to force your opponent into checkmate as fast as you can, so you would always want to choose a checkmate found at a shallower depth if multiple checkmates are found at different depths. Since alpha-beta search is depth first, the search will choose the move that finds a checkmate first, not necessarily the shallowest checkmate, which is why iterative deepening is super effective. However, when checkmates are not being found in the search, the iterative deepening just slows down the search by searching a depth of 1, then 1 and 2, then 1 and 2 and 3, etc. until the max depth, which takes up more time than is saves, since it only saves time when a checkmate is found. So if there was a way to implement iterative deepening only when checkmates are found in the search, that would take away any unnecessary double searching that comes along with ID, and would only utilize ID if it would save the algorithm time.

Another way to improve the alpha-beta AI would be to order the moves being looked at during each iteration in a smart way. This reordering would reorder moves so that moves with higher values are looked at before those of lower values. This is because if moves with higher values are analyzed first, alpha will be set early on and every node visited after that will not have to be analyzed since they will be under the alpha upper bound. This reordering of moves could be implemented with a transposition table, where the values for positions resulting from moves from the position.getAllMoves() call are taken from the transposition table, and ordered in a decreasing order. Once the moves are in decreasing order of resulting values, they are put through the alpha beta search in that order to utilize the fact that we may have the values of the positions resulting from the moves and alpha-beta search improves with a smart ordering of its nodes. If a position after a given move is not in the transposition table, put them at the end of the ordered moves because it is likely that some of the unanalyzed moves (the moves not in the transposition table) are less than the current maximum move, so some moves will not have to be analyzed. If they are put before the maximum value in the ordered moves, then they are guaranteed to all be analyzed, which defeats the purpose of ordering the moves and analyzing the highest value nodes first.

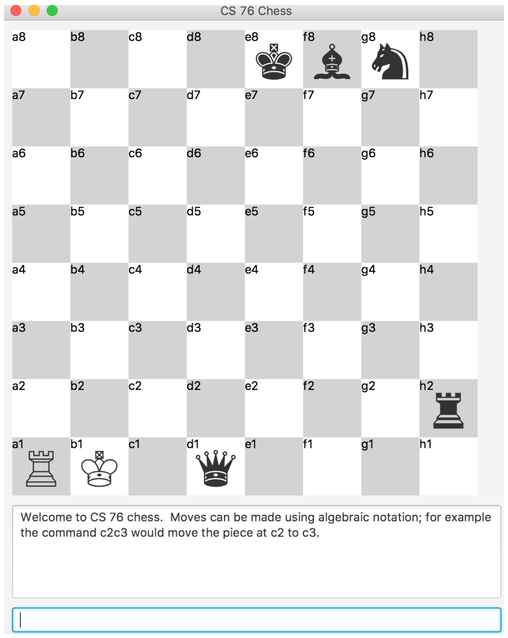
7 Testing

Forced Checkmate Test



Welcome to CS 76 chess. Moves can be made using algebraic notation; for example the command c2c3 would move the piece at c2 to c3.

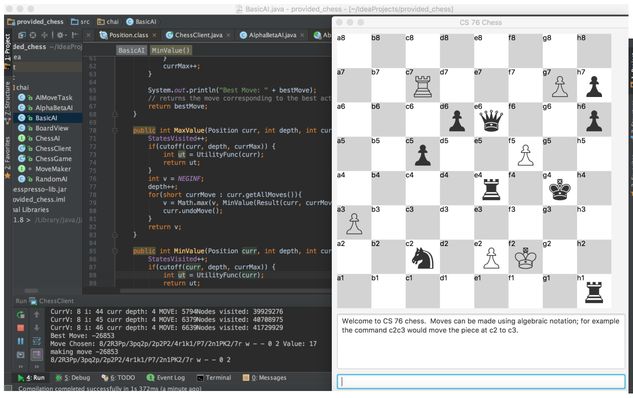
Figure 1: This is forced checkmate check. The AI should move the queen down to the bottom row for a forced checkmate. This is a tricky situation, however, because many checkmates are found with a depth of 4, so this example shows how iterative deepening takes the shallowest checkmate it finds.



Welcome to CS 76 chess. Moves can be made using algebraic notation; for example the command c2c3 would move the piece at c2 to c3.

Figure 2: Checkmate!

Same Value for Minimax Search and AB Search



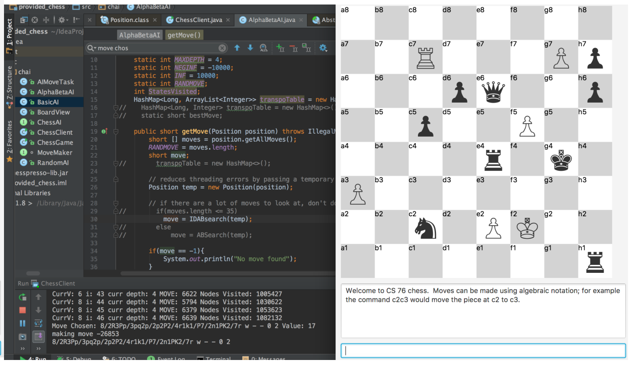
```
            }
            currMax++;
        }

        System.out.println("Best Move: " + bestMove);
        // returns the move corresponding to the best act
        return bestMove;
    }

    public int MaxValue(Position curr, int depth, int cur
        StatesVisited++;
        if(cutoff(curr, depth, currMax)) {
            int ut = UtilityFunc(curr);
            return ut;
        }
        int v = NEGINF;
        depth++;
        for(short currMove : curr.getAllMoves()){
            v = Math.max(v, MinValue(Result(curr, currMov
            curr.undoMove();
        }
        return v;
    }

    public int MinValue(Position curr, int depth, int cur
        StatesVisited++;
        if(cutoff(curr, depth, currMax)) {
            int ut = UtilityFunc(curr);
            return ut;
```

```
CurrV: 8 i: 44 curr depth: 4 MOVE: 5794Nodes visited: 39929276
CurrV: 8 i: 45 curr depth: 4 MOVE: 6379Nodes visited: 40708975
CurrV: 8 i: 46 curr depth: 4 MOVE: 6639Nodes visited: 41729929
Best Move: -26853
Move Chosen: 8/2R3Pp/3pq2p/2p2P2/4r1k1/P7/2n1PK2/7r w - - 0 2 Value: 17
making move -26853
8/2R3Pp/3pq2p/2p2P2/4r1k1/P7/2n1PK2/7r w - - 0 2
```

Compilation completed successfully in 1s 372ms (a minute ago)

Welcome to CS 76 chess. Moves can be made using algebraic notation; for example the command c2c3 would move the piece at c2 to c3.

Figure 3: I moved the rook from d7 to c7 to show how the AB search and Minimax choose moves of the same value. This board represents the Basic AI finding the value 17 move. The console shows that the move chosen has value 17. This search looked at 41729929 nodes during the iterative deepening.



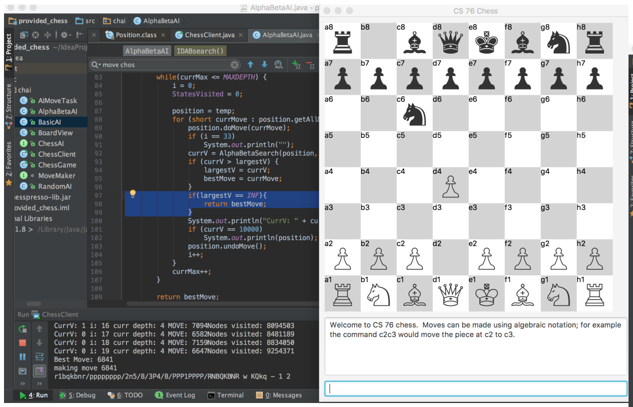
```
    static int MAXDEPTH = 4;
    static int NEGINF = -10000;
    static int INF = 10000;
    static int RANDMOVE;
    int StatesVisited;
    HashMap<Long, ArrayList<Integer>> transpoTable = new Ha
//    HashMap<Long, Integer> transpoTable = new HashMap<>(
//    static short bestMove;

    public short getMove(Position position) throws Illegal
        short [] moves = position.getAllMoves();
        RANDMOVE = moves.length;
        short move;
//        transpoTable = new HashMap<>();

        // reduces threading errors by passing a temporary
        Position temp = new Position(position);

        // if there are a lot of moves to look at, don't d
//        if(moves.length <= 35)
            move = IDABsearch(temp);
//        else
//            move = ABSearch(temp);

        if(move == -1){
            System.out.println("No move found");
        }
```

```
CurrV: 6 i: 43 curr depth: 4 MOVE: 6622 Nodes Visited: 1005427
CurrV: 8 i: 44 curr depth: 4 MOVE: 5794 Nodes Visited: 1030622
CurrV: 8 i: 45 curr depth: 4 MOVE: 6379 Nodes Visited: 1053623
CurrV: 8 i: 46 curr depth: 4 MOVE: 6639 Nodes Visited: 1082132
Move Chosen: 8/2R3Pp/3pq2p/2p2P2/4r1k1/P7/2n1PK2/7r w - - 0 2 Value: 17
making move -26853
8/2R3Pp/3pq2p/2p2P2/4r1k1/P7/2n1PK2/7r w - - 0 2
```

Welcome to CS 76 chess. Moves can be made using algebraic notation; for example the command c2c3 would move the piece at c2 to c3.

Figure 4: This shows the AB search finding a move with the same value as the Basic AI, which was 17. This can be seen in the console print statement. You can also notice that this search looks at 1082132 nodes, which is abour 40 times fewer nodes than the basic AI.

Nodes Visited Minimax and Alpha-Beta

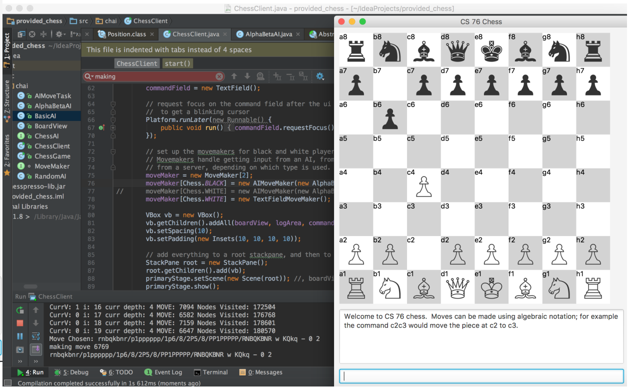


```
        while(currMax <= MAXDEPTH) {
            i = 0;
            StatesVisited = 0;

            position = temp;
            for (short currMove : position.getAll
                position.doMove(currMove);
                if (i == 33)
                    System.out.println("");
                currV = AlphaBetaSearch(position,
                if (currV > largestV) {
                    largestV = currV;
                    bestMove = currMove;
                }
                if(largestV == INF){
                    return bestMove;
                }
                System.out.println("CurrV: " + cu
                if (currV == 10000)
                    System.out.println(position);
                position.undoMove();
                i++;
            }
            currMax++;
        }

        return bestMove;
```

```
CurrV: 1 i: 16 curr depth: 4 MOVE: 7094Nodes visited: 8094503
CurrV: 0 i: 17 curr depth: 4 MOVE: 6582Nodes visited: 8481189
CurrV: 0 i: 18 curr depth: 4 MOVE: 7159Nodes visited: 8834050
CurrV: 0 i: 19 curr depth: 4 MOVE: 6647Nodes visited: 9254371
Best Move: 6841
making move 6841
r1bqkbnr/pppppppp/2n5/8/3P4/8/PPP1PPPP/RNBQKBNR w KQkq - 1 2
```

Welcome to CS 76 chess. Moves can be made using algebraic notation; for example the command c2c3 would move the piece at c2 to c3.

Figure 5: This screenshot shows how many nodes were visited by the basic AI on the opening move, and what move is chosen at the depth 4 Minimax search from the opening move.



This file is indented with tabs instead of 4 spaces

```
        commandField = new TextField();

        // request focus on the command field after the ui
        //  to get a blinking cursor
        Platform.runLater(new Runnable() {
            public void run() { commandField.requestFocus()
        });

        // set up the movemakers for black and white player
        // Movemakers handle getting input from an AI, from
        // from a server, depending on which type is used.
        moveMaker = new MoveMaker[2];
        moveMaker[Chess.BLACK] = new AIMoveMaker(new AlphaB
//      moveMaker[Chess.WHITE] = new AIMoveMaker(new AlphaB
        moveMaker[Chess.WHITE] = new TextFieldMoveMaker();

        VBox vb = new VBox();
        vb.getChildren().addAll(boardView, logArea, command
        vb.setSpacing(10);
        vb.setPadding(new Insets(10, 10, 10, 10));

        // add everything to a root stackpane, and then to
        StackPane root = new StackPane();
        root.getChildren().add(vb);
        primaryStage.setScene(new Scene(root)); //, boardVi
        primaryStage.show();
```

```
CurrV: 1 i: 16 curr depth: 4 MOVE: 7094 Nodes Visited: 172504
CurrV: 0 i: 17 curr depth: 4 MOVE: 6582 Nodes Visited: 176768
CurrV: 0 i: 18 curr depth: 4 MOVE: 7159 Nodes Visited: 178601
CurrV: 0 i: 19 curr depth: 4 MOVE: 6647 Nodes Visited: 180570
Move Chosen: rnbqkbnr/p1pppppp/1p6/8/2P5/8/PP1PPPPP/RNBQKBNR w KQkq - 0 2
making move 6769
rnbqkbnr/p1pppppp/1p6/8/2P5/8/PP1PPPPP/RNBQKBNR w KQkq - 0 2
```

Compilation completed successfully in 1s 612ms (moments ago)

Welcome to CS 76 chess. Moves can be made using algebraic notation; for example the command c2c3 would move the piece at c2 to c3.

Figure 6: This screenshot shows how many nodes were visited by the alpha beta pruning AI