Chess AI

Vivek Sah

February 9, 2016

1 Introduction

In this paper, we will implement chess algorithms such as MiniMax Algorithms, Alpha-beta search and we will then try to improve their efficiency by applying modifications such as move ordering and Transposition Table. To run our chess program, we will use the ***chesspresso*** library written by Bernhard Seybold. Throughout this paper, we will discuss the implementation of each algorithm and test of they are working against various test cases. The implementation is based on the pseudocode provided in the Artificial Intelligence textbook by Russel and Norvig. We will also discuss two different ways of calculating utility of a chess position. The utility function will help us decide between different moves.

2 Overview of Project Structure

The main files are *ChessGame.java*, *ChessClient.java*, *MiniMaxAI.java* , *AlphaBetaAI.java*, *FasterAlphaBetaAI.java*. We define the starting positions in *ChessGame.java* ,and assign players in *ChessClient.java*. The chess algorithms are implemented in *MiniMaxAI.java* , *AlphaBetaAI.java*, *FasterAlphaBetaAI.java*.

3 MiniMax and CutOff test

We will first implement depth-limited minimax search algorithm in MiniMaxAI class. We will first try all the moves for a given position and get the minValue of the resulting positions and then, return the move which corresponds to the resulting state with the maximum value. The three main functions are minimax, minValue and maxValue. Minimax calls minvalue on the resulting states after the first move. Then, minValue calls maxValue on the resulting states after applying the moves at that position. This goes on unless we get to the terminal states or the depth exceeds the max-depth predefined. Here is the snippet for limited-depth minimax function and minValue. The code for maxValue is very similar to minValue.

```
public short minimax(Position position, int maxDepth)  {
        int bestMoveValue = Integer.MIN_VALUE;
        short bestMove = Move.NO_MOVE;
        for (short move : position.getAllMoves()){ //loop through the
           moves and get the best move
            Position newPosition= new Position(position); //make a copy
               of position
            try {
                newPosition.doMove(move); //try to do move to get new
                   position
            } catch (IllegalMoveException e) {
                e.printStackTrace();
            }
            int val =minValue(newPosition,0,maxDepth);
            if(bestMove==(Move.NO_MOVE)){
                bestMove =move;
```

```
                bestMoveValue = val;
            }
            if(val > bestMoveValue){
                bestMoveValue = val;
                bestMove =  move;
            }
        }
        return bestMove;
    }

public int minValue(Position position, int currentDepth, int maxDepth)  {

        //cut-off tests
        if(position.isStaleMate()){
            return 0;
        }
        if(position.isTerminal()){
            return Integer.MAX_VALUE;
        }
        if(currentDepth == maxDepth){
            return getUtility(position);
        }
        int val = Integer.MAX_VALUE;
        for (short move: position.getAllMoves()){
            Position tempPosition = new Position(position);
            try {
                tempPosition.doMove(move);
            } catch (IllegalMoveException e) {
                e.printStackTrace();
                continue;
            }
            val = Math.min(val,maxValue(tempPosition, currentDepth+1,
               maxDepth));
        }
        return val;
    }
```

As we can see, for cutoff tests, we check to see if its terminal state. If it is terminal, we check if its a draw(stalemate). If it is a draw, we return 0. Otherwise, we return MAX_VALUE as it is a minValue function. In maxValue, we return MIN_VALUE. If we reach the maximum depth, we calculate the utility of that position and return it.

4 Utility and Evaluation function

While doing depth-limited search, we have to calculate utility of the position if it is not a terminal state. Utility for terminal states is already discussed above. To calculate utility of non-terminal states, we have two ways:

- Random Utility: We will return a random integer between Integer.MAX_VALUE and Integer.MIN_VALUE for any non-terminal positions.
- Use position class methods: Here, we create a evaluation function which uses Position class' ***getMaterial*** and ***getDomination*** methods. Let's look at the code snippet for this evaluation function.

```
    private int eval(Position position) {
```

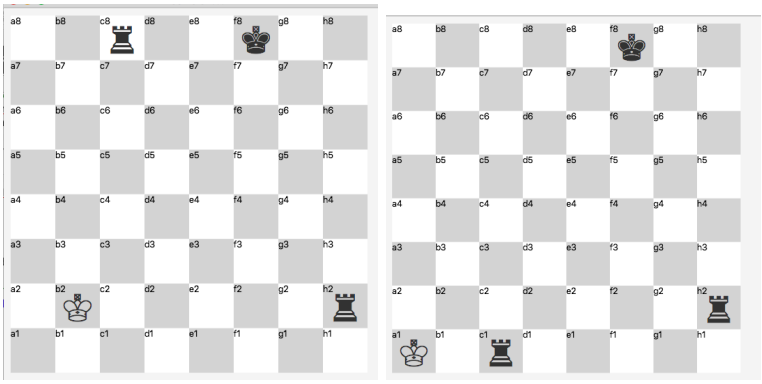
```
        int val = position.getMaterial() +
            (int)position.getDomination();
        if (startingPlayer == position.getToPlay()) //check if its
            the starting player's turn when eval was called
             return val;
        else //otherwise return negative
             return -1*val;
    }
```

The ***getMaterial*** method returns the material value of pieces on the chessboard but does not account for their position. That is where, ***getDomination*** method comes in. It evaluates the score based on the positions of the pieces on the board. The sum of these two values will be used to represent the score for a certain position.

5 Results for MiniMax

We want to show that both ways of calculating utility force checkmate if it is possible. So, we will set the start state such that black has two rooks and a king whereas the white has only one king. The figures below demonstrate what happens:

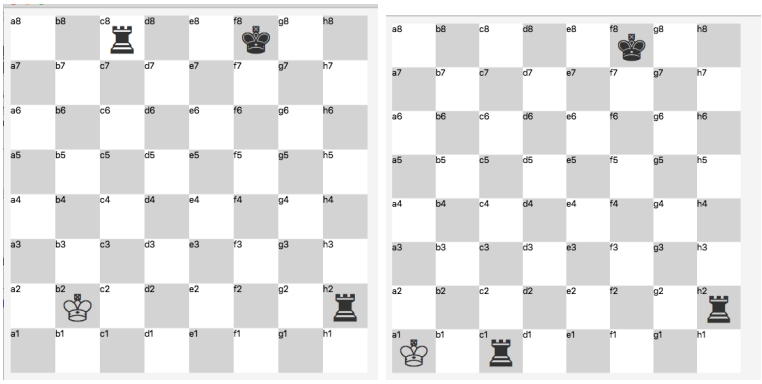
5.1 With random Utility



Here, the white player moved from b2 to a1 and the black forces a checkmate and wins. Here are further analysis:

Max_Depth =1
time taken:5ms visitedNodes: 95 maxDepthReached:1
Max_Depth = 2
time taken:79ms visitedNodes: 1570 maxDepthReached:2
Max_Depth =3
time taken:327ms visitedNodes: 12347 maxDepthReached:3

5.2 With Eval function



The evaluation functions results in the same moves but takes more run time. Max_Depth =1
time taken:9ms visitedNodes: 95 maxDepthReached:1
Max_Depth = 2
time taken:80ms visitedNodes: 1570 maxDepthReached:2
Max_Depth =3
time taken:381ms visitedNodes: 12347 maxDepthReached:3

6 Alpha-Beta Search

Minimax traverses all the nodes in the tree and the number of nodes it has to explores increases exponentially. Although, we can't eliminate the exponent, we can reduce the number of nodes it has to explore. It follows from the observation that to calculate the minimax decision for a node, we do not have to look at all the nodes, that is, we can prune some parts of the tree with a technique called alpha–beta pruning. When we apply alpha-beta pruning to MiniMax algorithm, we get the same move but in a shorter time as we explore fewer nodes. The implementation is done in *AlphaBetaAI.java*. The implementation is really similar that of MiniMax algorithm except few additions. When we implement minValue or maxValue, we compare the calculated value to alpha and beta and depending on that, break the loop. Lets look at a section of code from minValue method to clarify it.

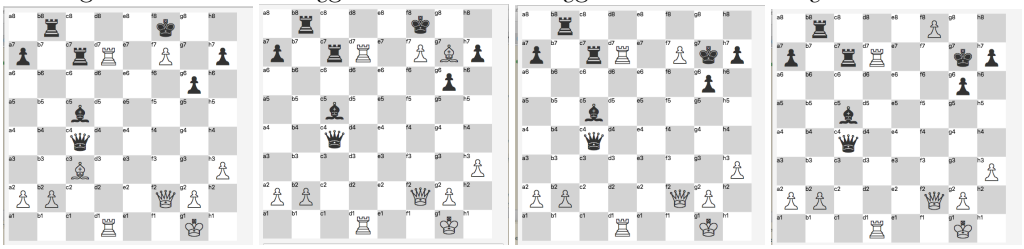
```
for (short move: position.getAllMoves()){
            Position newPosition = new Position(position);
            try {
                newPosition.doMove(move);
            } catch (IllegalMoveException e) {
                e.printStackTrace();
                continue;
            }
            val = Math.min(val,maxValue(newPosition, alpha, beta,
                currentDepth+1, maxDepth));
            if (val <= alpha ){
               break;
            }
            beta= Math.min(beta,val);
        }
```

7 Comparison of AlphaBetaAI and MiniMaxAI

Starting state: *"1r3k2/p1rR1P1p/6p1/2b5/2q5/2B4P/PP3QP1/3R2K1 w KQkq - 0 1"*

7.0.1 MiniMaxAI vs MiniMaxAI

starting state c3-¿g7 f8-¿g7 f7-¿f8

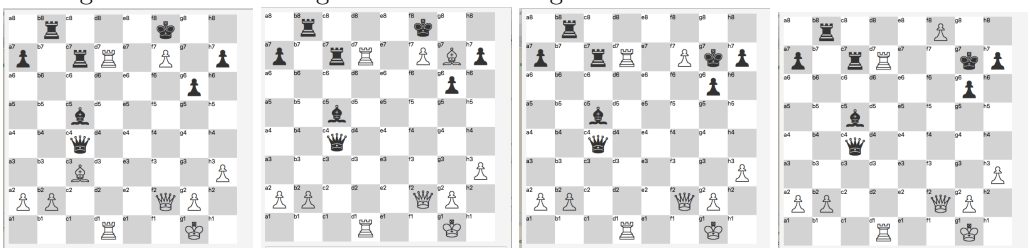


Result: Black Wins

making move 7570 time taken:21924ms visitedNodes: 2436355 maxDepthReached:3
making move -25155 time taken:716ms visitedNodes: 54888 maxDepthReached:3
making move 24437 time taken:20894ms visitedNodes: 2358355 maxDepthReached:3

7.0.2 AlphaBetaAI vs AlphaBetaAI

starting state c3-g7 f8-g7 f7-f8



Result: Black Wins
making move 7570 time taken:2512ms visitedNodes: 143504 maxDepthReached:3
making move -25155 time taken:94ms visitedNodes: 6137 maxDepthReached:1
making move 24437 time taken:1757ms visitedNodes: 157884 maxDepthReached:3

We can see that alphaBeta and Minimax AI use the same moves in both cases and nodes visited in alphabetaAI is less than the nodes visited in MinimaxAI.

8 Faster Alpha Beta AI

To make Alpha Beta even faster, we will add some modifications: **move ordering** and use of **transposition table**

8.1 Move Ordering (Bonus Problem)

In AlphaBetaAI, we can come across situations when we could avoid traversing branches of tree if the move corresponding to that branch was generated later. Hence, we can apply a heuristic to order the branches. We will do a iterative alpha-beta search and store the best move at a max depth of ,say k, in an array at index 'k'. When we do iteration for depth 'k+1', we will first try the bestMove at depth k and then try all the capturing moves and then the rest moves. With this heuristic, we expect to further prune alpha-beta nodes and decrease the run-time of Alpha Beta search.

Here is the code snippet which shows move ordering (part of alphaBeta function in FasterAlphaBetaAI.java).

```
if (maxDepth > 0 ){
            short currentMove = bestMoves[maxDepth-1];
            try {
                newPosition.doMove(bestMoves[maxDepth-1]); //try to do
                    move to get new position
            } catch (IllegalMoveException e) {
                e.printStackTrace();
```

```
        }
        triedMoves.add(bestMoves[maxDepth-1]);
        int val =minValue(newPosition,alpha, beta,0, maxDepth);
        if(bestMove==(Move.NO_MOVE)){
            bestMove =currentMove;
            bestMoveValue = val;
        }
        if(val > bestMoveValue){
            bestMoveValue = val;
            bestMove =  bestMoves[maxDepth-1];
        }
        newPosition.undoMove();

    }
    for (short move : position.getAllCapturingMoves()){
        if(triedMoves.contains(move)){
            continue;
        }
        newPosition = new Position(position);
        try {
            newPosition.doMove(move); //try to do move to get new
                position

        } catch (IllegalMoveException e) {
            e.printStackTrace();
            continue;
        }
        int val =minValue(newPosition,alpha, beta,0, maxDepth);
        if(bestMove==(Move.NO_MOVE)){
            bestMove =move;
            bestMoveValue = val;
        }
        if(val > bestMoveValue){
            bestMoveValue = val;
            bestMove =  move;
        }
    }
```

8.2 Transposition table

Our evaluation function uses *getMaterial* and *getDomination* function which take some time to calculate. Instead of evaluating all the time, we can store these evaluations corresponding to in a hash map. We will use LinkedHashMap with a max size of 5 million to store the evaluations. To not exceed heap space, we will start clearing hashmap if the size of hashmap exceeds the max size. When we enter the evaluation function, we will calculate hashcode of the current position using zobrist hashcode. Then we check in the transposition table if there is an entry. If there is, we will return the corresponding score. Here is a snippet of the modified evaluation function.

```
int currentHashCValue = position.hashCode()%MAX_HASHMAPSIZE;
if(transpositionTable.containsKey(currentHashCValue)){
    return transpositionTable.get(currentHashCValue);
}
int val = position.getMaterial() + (int)position.getDomination();
```

```
if (startingPlayer != position.getToPlay()) //check if its the
    starting player turn when eval was called
     val = -1*val;
transpositionTable.put(currentHashCValue,val);
return val;
```

We will run our normal alpha beta AI and faster alpha beta AI side by side and compare the results. Starting state: *"r5k1p3Qpbp2p3p11p6q3bN26PP/PP3P2K2RR3 b - - 0 1"*

Result: both algorithms complete the game in 3 steps where black wins.

Normal Alpha Beta:

time taken:37927ms visitedNodes: 4160873 maxDepthReached:4 move -28042
time taken:965ms visitedNodes: 57594 maxDepthReached:3 move -28096
time taken:22438ms visitedNodes: 2404820 maxDepthReached:4 move 4760
time taken:1486ms visitedNodes: 112450 maxDepthReached:1 move 4105
time taken:21500ms visitedNodes: 2306426 maxDepthReached:4 move 5258

Faster Alpha Beta AI

time taken:36147ms visitedNodes: 3750974 maxDepthReached:4 move -28042
time taken:1010ms visitedNodes: 57684 maxDepthReached:3 move -28096
time taken:21433ms visitedNodes: 2444145 maxDepthReached:4 move 4760
time taken:4096ms visitedNodes: 117821 maxDepthReached:1 move 4105
time taken:24012ms visitedNodes: 2632316 maxDepthReached:4 move 5258

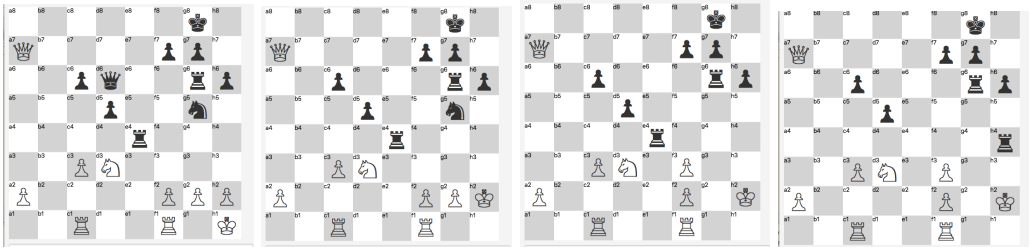
However, normal alpha beta explores 9,042,163 nodes whereas FasterAlphaBetaAI explores 6,279,126 nodes, which is significant improvement.

9 TestCases

To show that our algorithm works, we will test against known test cases and see if our algorithm achieves the result in given number of steps. We will show two additional test cases.

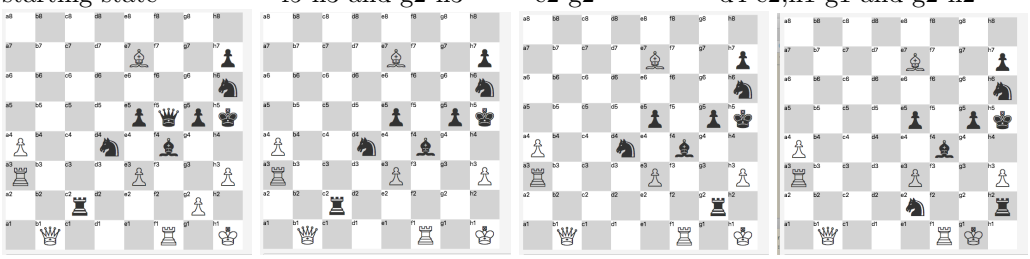
- Starting point: "6k1/Q4pp1/2pq2rp/3p2n1/4r3/2PN4/P4PPP/2R2R1K b - - 0 1" result: black wins in 3 moves

starting state | d6-h2 and h1-h2 | g5-f3 and g2-f3 | e4-h4



- Starting point: "8/4B2p/7n/4pqpk/P2n1b2/R3P2P/2r3P1/1Q3R1K b - - 0 1" result: black wins in 3 moves

starting state | f5-h3 and g2-h3 | c2-g2 | d4-e2,h1-g1 and g2-h2



The faster algorithm works on other given test cases too.