Chess AI Solution

Faizan N. Kanji

February 21, 2014

1 Introduction

Over the years, several attempts have been made to create an excellent Chess program. In this assignment, we make our own little attempt at this. Fortunately, we are given wonderful resources in the Chesspresso library as well as the GUI code provided by professor Balkcom. Our task, then, is simply to get an AI player to pick the best possible move at each point. We will build our way up to such an AI player improving the way it picks the next move both in terms of time and the optimality of the move. We will start with the basic Minimax algorithm with a simple heuristic, followed by speeding it up using alpha beta pruning. We will then implement some interesting ideas to further improve minimax such as using a transposition table, a more advanced heuristic, using opening strategies from some famous games, using quiescence search, using a null move heuristic to forward prune and using move reordering.

2 Minimax

2.1 Basic implementation and utility function

Our first task is to implement the famous minimax algorithm. The basis of the minimax algorithm is that each time the AI calls the algorithm to find a move, it treats that AI as the maximizing player and its opponent as the minimizing player. The algorithm then searches down a game tree, finding out the best move for each player at different depths. It will attempt to maximize the utility for the maximizing player and minimize it for the minimizing player. Since going down the chess game tree fully is impossible with the current state of technology, we can only search the game tree upto a maximum depth. Therefore, in our minimax implementation, we will cut off the search when either of two things happen:

- We have reached a terminal state (a win or a draw)
- we have reached the relevant maximum depth

Each time the minimax is cut off, it returns a value associated with current state of the chess board at cutoff. When the board is in a terminal state, we can use values of `Integer.MAXVALUE` for a win, `Integer.MINVALUE` for a loss and 0 for a draw. Finding values for a state which is not terminal, however, is more complicated. In our first implementation of minimax, we simply pick a random number if the state is not terminal, while using the above mentioned values for when the state is terminal. The basic implementation of minimax is given in the following three methods `miniMaxMove`, `maxValue` and `minValue`:

```
1
2  public short miniMaxMove(Position position, int maxDepth){
3      nodeCount++; //to keep track of time.
4      //get all moves from the current posiiton.
```

```
    short [] moves = position.getAllMoves();

    int max = Integer.MIN_VALUE;
    short maxMove = 0;
    int currentDepth = 1;
    int player = position.getToPlay();

    //for each move, get the utility using minValue.
    for(short move:moves){
      int minVal = minValue(move, position, maxDepth, currentDepth, player);
      //update max if below condition holds.
      if(minVal>max){
        max = minVal;
        maxMove = move;
      }
    }
    //Does this move eventually lead to a checkmate?
    if(max == Integer.MAX_VALUE){
      mated = true;
    }

    //return the corresponding move.
    return maxMove;
  }



  public int minValue(short move, Position position, int maxDepth, int currentDepth, int player){

    //do the move.
    try {
      position.doMove(move);
    } catch (IllegalMoveException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
    nodeCount++;
    //increment depth.
    int newDepth = currentDepth + 1;

    //do the cutoff test.
    if(position.isTerminal() || newDepth>maxDepth ){
      int util = utility(position, player, currentDepth);
      position.undoMove();
      return util;
    }

    //loop through the moves finding the one with the minimum value.
    int min = Integer.MAX_VALUE;
    for(short nextMove:position.getAllMoves()){
      int maxVal = maxValue(nextMove, position, maxDepth, newDepth, player);
      if(maxVal<min){
        min = maxVal;
      }
    }

```

```
    position.undoMove();
    //return that min value.
    return min;
  }


  public int maxValue(short move, Position position, int maxDepth, int currentDepth, int player){
    //do the move.
    try {
      position.doMove(move);
    } catch (IllegalMoveException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
    //increment depth.
    int newDepth = currentDepth + 1;
    nodeCount++;

    //do the cutoff tests.
    if(position.isTerminal()){
      int util = utility(position, player, currentDepth);
      position.undoMove();
      return -1*util; //-1 is multiplied as this is a loss for the player.
    }

    if(newDepth > maxDepth){
      int util = utility(position, player, currentDepth);
      position.undoMove();
      // get appropriate utility based on whether the starting player is white or black.
      // this ensures that we can pick either color for the computer.
      if(player==0){
        return util;
      }else{
        return -1*util;
      }
    }

    //loop through moves to find one with maximum value.
    int max = Integer.MIN_VALUE;
    for(short nextMove:position.getAllMoves()){
      int minVal = minValue(nextMove, position, maxDepth, newDepth, player);
      if(minVal>max){
        max = minVal;
      }
    }

    position.undoMove();
    //return that move.
    return max;
  }
```

We can clearly see the cutoff test and the transition from minValue to maxValue in the code above. It is important to note that the code has been made general enough so that the computer can be any color, black or white. Now we present our first utility function that was described before `utility`

```
public int utility(Position position, int player, int depth){
    //if the state is terminal, see if it's a mate or a draw.
    if(position.isTerminal()){
      if(position.isMate()){
        return Integer.MAX_VALUE;
      }else{
        return 0;
      }
    }else{
      //otherwise return a random integer between Integer.Min_Value and
      //Integer.Max_Value.
      int rand = new Random().nextInt(Integer.MAX_VALUE);
      if(new Random().nextDouble()<0.5){
        return rand;
      }else{
        return -1*rand;
      }
    }
  }
```

As we can see, the utility function gives a very high value for a checkMate, a 0 for a draw and a random integer for any other move. Recall that we mentioned that the utility of a checkMate move depends on who won. This is taken care of in the minimax code where the utility is negated if the respective player lost giving a large negative number for that case.

Our final step is to write the `getMove` method of our program. The `getMove` does an iterative deepening version of minimax, incremeneting the depth each time and storing the best move for the most recent depth. This has two main advantages. Firstly, suppose the best move at depth 1 leads to a checkmate. It would be pointless to search higher depths because we can't do better than a checkmate. We can then break off the iterative deepening search and return that move. Secondly, if we are playing a timed version of chess, iterative deepening will allow us to search until time runs out. when time runs out, we can simply return the best move found at the highest depth. Keeping all of this in mind, below is the method:

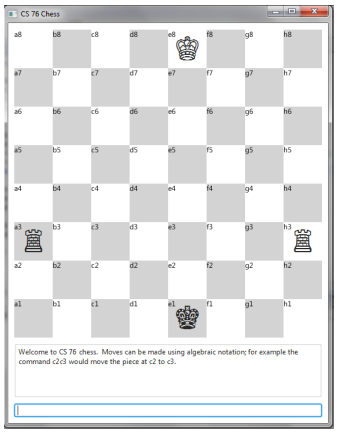
```
public short getMove(Position position) {
    nodeCount = 0; //keep track of time.

    // copy the current position.
    String copyFEN = position.getFEN();
    Position copy = new Position(copyFEN);

    short move = 0;
    int depth = 1;
    //iterative deepening minimax.
    while(depth<MAXDEPTH){
      move = miniMaxMove(copy, depth);
      if(move==0){
        break;
      }

      if(mated){
        mated = false;
```

```
19          break;
20        }
21        depth++;
22      }
23      System.out.println("The node count is: " + nodeCount);
24      return move;
25    }
```

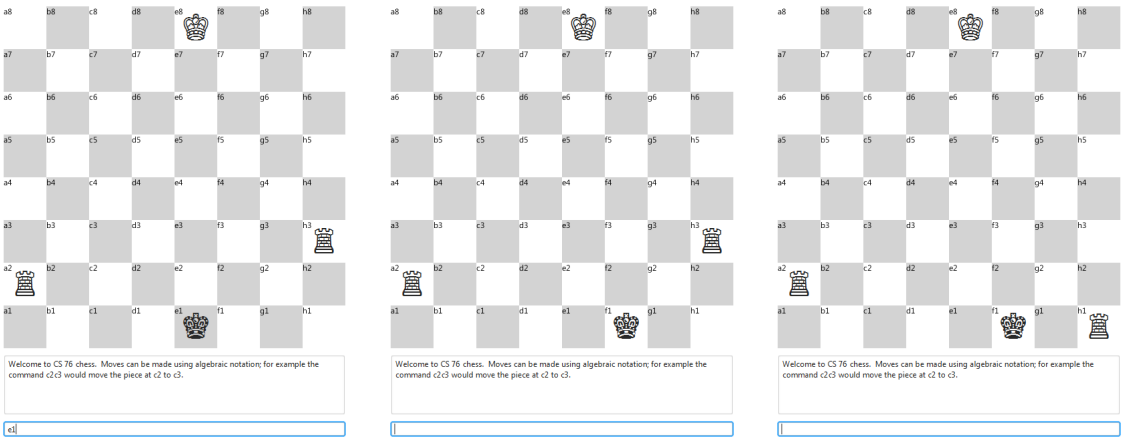
We can now see how all the methods fit together. `getMove` calls `miniMaxMove` for each iteration, which then alternates between `minValue` and `maxValue` calling `utility` if the cutoff tests are satisfied. Our algorithm should now be able to find checkmates, and also prevent itself from being checkmated. We can now move on to test our algorithm.

2.2 Testing Basic Implementation

consider the following start position:



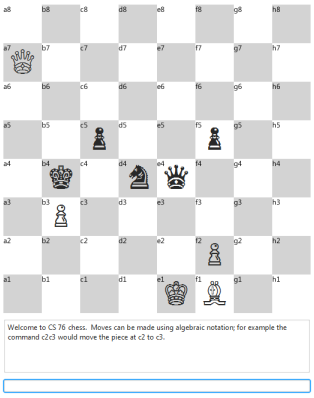
We play minimax to a MAXDEPTH of 7 with minimaxAI as white. Since a checkmate is possible within that depth, we expect it to go for it. That is exactly what it does as seen by the following images:



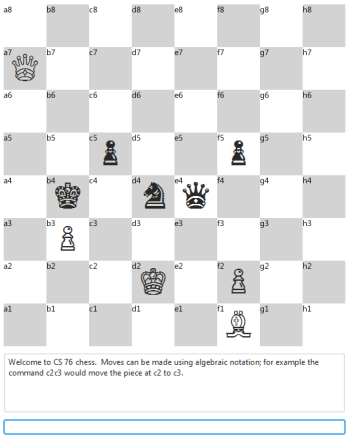
The node count for each move and the respective board positions are given below:

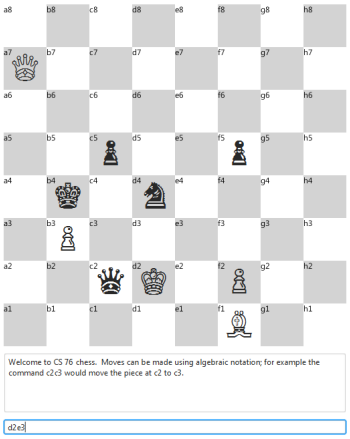
```
The node count is:  4488
making move 4624
4K3/8/8/8/8/7R/R7/4k3 b - - 1 1
making move 4420
4K3/8/8/8/8/7R/R7/5k2 w - - 2 2
The node count is:  34
making move 4567
4K3/8/8/8/8/8/R7/5k1R b - - 3 2
CHECKMATE! White Wins!
```

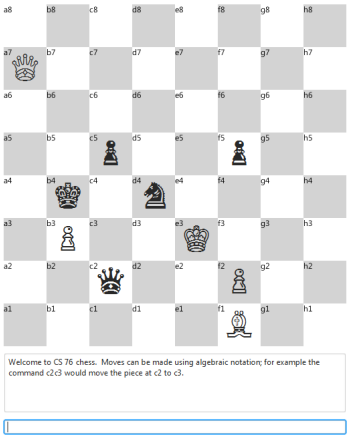
Let's now try a more complex checkmate. In this case, a checkmate can be done by black in 3 moves (source: http://www.wtharvey.com/m8n4.html). Below is the respective start position:

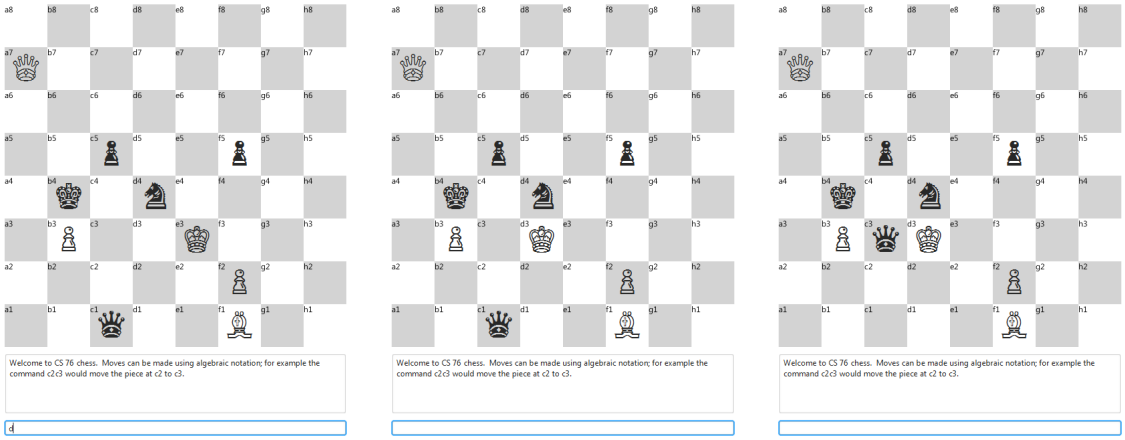


This time we set the minimaxAI to black and we play white keeping the MAXDEPTH to 7. As we expect, the AI does the respective 3 move checkmate!









The node count for each move and the respective board positions are given below:

```
making move 4804
8/Q7/8/2p2p2/1k1nq3/1P6/3K1P2/5B2 b - - 1 1
The node count is:  8230249
making move 4764
8/Q7/8/2p2p2/1k1n4/1P6/2qK1P2/5B2 w - - 2 2
making move 5387
8/Q7/8/2p2p2/1k1n4/1P2K3/2q2P2/5B2 b - - 3 2
The node count is:  8077
making move 4234
8/Q7/8/2p2p2/1k1n4/1P2K3/5P2/2q2B2 w - - 4 3
making move 5332
8/Q7/8/2p2p2/1k1n4/1P1K4/5P2/2q2B2 b - - 5 3
The node count is:  27
making move 5250
8/Q7/8/2p2p2/1k1n4/1PqK4/5P2/5B2 w - - 6 4
CHECKMATE! Black Wins!
```

Note: we will use the above case (let's call it **3-move-mate**) with MAXDEPTH 7 as a benchmark case for each of our improvements. For improvements that improve efficiency in terms of time, we expect the node count to decrease for each of the moves. The reason we use this as benchmark is because no matter how good our robot's performance becomes from now on, it will always do the same moves in the above case as that is the best and fastest way to get to a checkmate, therefore it allows us to compare node counts for same moves.

We can now believe that our Minimax AI works for when a checkmate is in sight. However, in order to get to a position where a checkmate is in sight, it will have make some intelligent moves when a checkmate is not in sight. In order for it to do that, we will need to improve our utility function.

2.3 Improved utility function and further testing

For an improved utility function, we will need to find a way to evaluate a configuration of the chessboard, giving it a score that shows how good or bad that configuration is for a specific player. In order to do this, we use a material value evaluation function. Without loss of generality, suppose our player is white. In order to find the evaluation value of a certain configuration, for each type of piece (pawn, bishop, queen, etc.) we

take the difference between the number of pieces of that type that white has on the board and the number of pieces of that type that black has on the board. We then take a weighted sum of these differences (for example pawns will be weighted quite low while queens will be weighted very high). This value will be the score for that configuration of the board. To get the score with respect to black, we can simply negate this value.

Chesspress already has a built in material evaluation function, `getMaterial`. We will start by using this function and will later write our own enhanced evaluation function. Therefore, we write a new utility function, `utility2`:

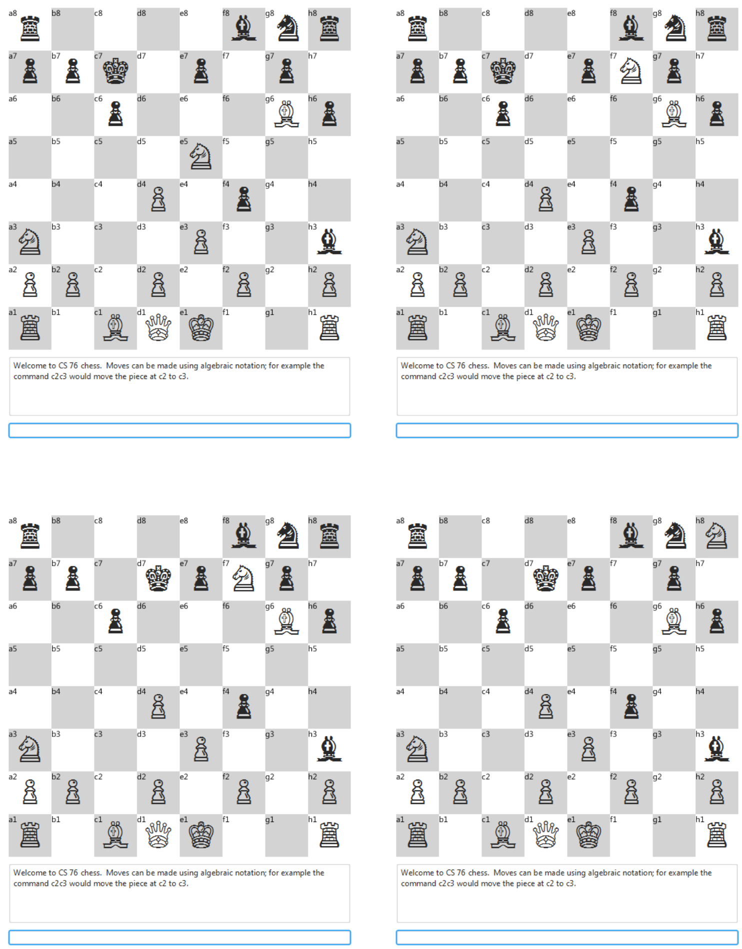
```
public int utility2(Position position, int player, int depth){
    //same as utility for terminal
    if(position.isTerminal()){
      if(position.isMate()){
        return Integer.MAX_VALUE;
      }else{
        return 0;
      }
    }else{
      // to ensure we get the right value for the right player.
      if(player == 0 && position.getToPlay() == player){
        return position.getMaterial();
      }else{
        return -1*position.getMaterial();
      }
    }
  }
```

Now that we have a slightly more intelligent utility function for configurations which are not terminal, we have made our chess program more intelligent. Let us now test its intelligence. We now pit our program against `RandomAI` written by professor Balkcom. With our AI as white and RandomAI as black and a MAXDEPTH of 5, we managed to get the checkmate in 23 full moves. The last few moves along with the node count and evaluation values are printed below:

```
making move 6769
Q5nN/3kp1p1/ppp3Bp/8/3P4/4P2b/PP3P1P/RNB1K2R w KQ - 0 20
Evaluation value of move: 2825
The node count is:  271424
making move 7992
4Q1nN/3kp1p1/ppp3Bp/8/3P4/4P2b/PP3P1P/RNB1K2R b KQ - 1 20
making move 7347
4Q1nN/2k1p1p1/ppp3Bp/8/3P4/4P2b/PP3P1P/RNB1K2R w KQ - 2 21
Evaluation value of move: 2825
The node count is:  321689
making move -24644
6QN/2k1p1p1/ppp3Bp/8/3P4/4P2b/PP3P1P/RNB1K2R b KQ - 0 21
making move 7383
6QN/2kbp1p1/ppp3Bp/8/3P4/4P3/PP3P1P/RNB1K2R w KQ - 1 22
Evaluation value of move: 2925
The node count is:  421490
making move -25154
7N/2kbp1Q1/ppp3Bp/8/3P4/4P3/PP3P1P/RNB1K2R b KQ - 0 22
making move 6898
```

```
7N/3bp1Q1/pppk2Bp/8/3P4/4P3/PP3P1P/RNB1K2R w KQ - 1 23
Evaluation value of move: 2147483647
The node count is:  39
making move 6454
7N/3bp3/pppk2Bp/4Q3/3P4/4P3/PP3P1P/RNB1K2R b KQ - 2 23
CHECKMATE! White Wins!
```

As the game progressed, our robot did indeed makes some intelligent moves. For example, as you can see in the screenshots below, after rendering the rook unmovable through the bishop, the AI moved the knight forwards to take the rook.



It is understandable that increasing the MAXDEPTH will improve the performance of the AI. In my testing, I found that playing with a MAXDEPTH of 1 to 5 was very quick. Increasing to 6 made the moves slower and 7 took impractical amounts of times. Our intelligent robot is therefore limited by the limitations of modern computing. All hope is not lost, however, as we can improve minimax search using a clever technique called alpha beta pruning!

3 Alpha Beta Pruning

Alpha beta pruning is a clever technique which makes use of the fact that we actually don't need to search the entire game tree when deciding what move to play and can prune some branches of the tree. At any point, we have the following definitions of α and β from our book:

α = is the best (i.e. highest-value) choice we have found so far at any choice point along the path for MAX. β = is the best (i.e. lowest-value) choice we have found so far at any choice point along the path for MIN.

α and β are updated as we go along with minimax. As soon as we know that the value of a node is worse than the current α or β value for MAX and MIN respectively, we can prune off the remaining branches of the node. Here are the updated functions `miniMaxABMove`, `minValue` and `maxValue` (note that only the changed portions are commented):

```
1  public short miniMaxABMove(Position position, int maxDepth){
2      nodeCount++;
3      short [] moves = position.getAllMoves();
4      //set initial upper and lower bounds.
5      int alpha = Integer.MIN_VALUE;
6      int beta = Integer.MAX_VALUE;
7      int max = Integer.MIN_VALUE;
8      short maxMove = 0;
9      int currentDepth = 1;
10     int player = position.getToPlay();
11     for(short move:moves){
12       int minVal = minValue(move, position, maxDepth, currentDepth, player, alpha, beta);
13       if(minVal>max){
14         max = minVal;
15         maxMove = move;
16       }
17       //check mate move.
18       if(max>=beta){
19         mated = true;
20         return maxMove;
21       }
22       //update alpha.
23       if(max>=alpha){
24         alpha = minVal;
25       }
26     }
27     return maxMove;
28   }
29
30   public int minValue(short move, Position position, int maxDepth, int currentDepth, int player,
         int alpha, int beta){
31     try {
32       position.doMove(move);
33     } catch (IllegalMoveException e) {
34       // TODO Auto-generated catch block
35       e.printStackTrace();
36     }
37     int newDepth = currentDepth + 1;
38     nodeCount++;
39     if(position.isTerminal() || newDepth>maxDepth ){
40       int util = utility2(position, player, currentDepth);
```

```
41        position.undoMove();
42        return util;
43      }
44
45
46      int min = Integer.MAX_VALUE;
47      for(short nextMove:position.getAllMoves()){
48        int maxVal = maxValue(nextMove, position, maxDepth, newDepth, player, alpha, beta);
49        if(maxVal<min){
50          min = maxVal;
51        }
52
53        //if value is worse than alpha, prune off everything else.
54        if(min<=alpha){
55          position.undoMove();
56          return min;
57        }
58        //update beta.
59        if(min<beta){
60          beta = min;
61        }
62      }
63
64      position.undoMove();
65      return min;
66    }
67
68
69    public int maxValue(short move, Position position, int maxDepth, int currentDepth, int player,
          int alpha, int beta){
70      try {
71        position.doMove(move);
72      } catch (IllegalMoveException e) {
73        // TODO Auto-generated catch block
74        e.printStackTrace();
75      }
76      nodeCount++;
77      int newDepth = currentDepth + 1;
78      if(position.isTerminal()){
79        int util = utility2(position, player, currentDepth);
80        position.undoMove();
81        return -1*util;
82      }
83
84      if(newDepth > maxDepth){
85        int util = utility2(position, player, currentDepth);
86        position.undoMove();
87        if(player==0){
88          return util;
89        }else{
90          return -1*util;
91        }
92      }
93
94      int max = Integer.MIN_VALUE;
95      for(short nextMove:position.getAllMoves()){
```

```
96        int minVal = minValue(nextMove, position, maxDepth, newDepth, player, alpha, beta);
97        if(minVal>max){
98          max = minVal;
99        }
100
101       //if value is worse than (greater than) beta, prune off everything else.
102       if(max>=beta){
103         position.undoMove();
104         return max;
105       }
106       //update alpha.
107       if(max>alpha){
108         alpha = max;
109       }
110     }
111
112     position.undoMove();
113     return max;
114   }
```

For now we will keep the same utility function as before. It is important to understand that alpha beta does not improve the program's performance compared to simple minimax at the same MAXDEPTH. Therefore we expect both alpha beta and minimax to return the same moves for same positions of the board. To test this, we will play three moves with both alpha beta and minimax and ensure that they return the same move. To ensure that we illustrate this fairly, we will test it on a random configuration. With white controlled by me and black as the AI and with a MAXDEPTH of 5 for both simple minimax and alpha beta, we do indeed get the same moves. Let's call this the **Comparer-Case** and we will also use this case in addition to our 3-move-mate case to compare future improvements.

The results for simple minimax are as follows:

```
making move -26796
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P2/N1P2N1q/PP1P1PPP/R1BQK2R b KQ - 0 10
Evaluation value of move: -300
The node count is:  1070064
making move -27753
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P2/N1P2N2/PP1P1PqP/R1BQK2R w KQ - 0 11
making move 6101
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1P1PqP/R1BQK2R b KQ - 1 11
Evaluation value of move: 300
The node count is:  934868
making move -28210
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1P1P1P/R1BQK2q w Q - 0 12
making move 4868
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1PKP1P/R1BQ3q b - - 1 12
Evaluation value of move: 300
The node count is:  631791
making move 6074
r2k1bnr/pp2p1p1/2p3Bp/3p4/5PbN/N1P5/PP1PKP1P/R1BQ3q w - - 2 13
```

On the other hand with alpha beta, we get the following results:

```
making move -26796
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P2/N1P2N1q/PP1P1PPP/R1BQK2R b KQ - 0 10
```

```
The node count is: 210317
making move -27753
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P2/N1P2N2/PP1P1PqP/R1BQK2R w KQ - 0 11
making move 6101
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1P1PqP/R1BQK2R b KQ - 1 11
The node count is: 16632
making move -28210
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1P1P1P/R1BQK2q w Q - 0 12
making move 4868
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1PKP1P/R1BQ3q b - - 1 12
The node count is: 23104
making move 6074
r2k1bnr/pp2p1p1/2p3Bp/3p4/5PbN/N1P5/PP1PKP1P/R1BQ3q w - - 2 13
```

As stated, both the AI's make the same moves when responding to identical moves given by me. However, for each of the three nodes, alpha beta visits considerably much less nodes than simple minimax (210,317 compared to 1,070,064 for the first move, 16,632 compared to 934,868 for the second move and 23,104 compared to 631,791 for the third move). These similar tests were conducted on several different start positions and different depths and in each case, whenever MAXDEPTH for alpha beta and minimax were they same, they both made the same move but alpha beta visited considerably less nodes.

Since visited nodes are being used by us to represent time taken, this means that alpha beta takes considerably less time than simple minimax. This means we can search to greater depths using alpha beta. In my program, we can now search to a MAXDEPTH of 6 almost instantaneously, 7 was slower while 8 on wards became quite impractical. We were therefore able to search at least 1 step deeper using alpha beta.

Before proceeding further, let's test alpha beta with our 3-move-mate benchmark case and ensure that we have improved the nodeCounts:

```
making move 4804
8/Q7/8/2p2p2/1k1nq3/1P6/3K1P2/5B2 b - - 1 1
The node count is: 38473
making move 4764
8/Q7/8/2p2p2/1k1n4/1P6/2qK1P2/5B2 w - - 2 2
making move 5387
8/Q7/8/2p2p2/1k1n4/1P2K3/2q2P2/5B2 b - - 3 2
The node count is: 936
making move 4234
8/Q7/8/2p2p2/1k1n4/1P2K3/5P2/2q2B2 w - - 4 3
making move 5332
8/Q7/8/2p2p2/1k1n4/1P1K4/5P2/2q2B2 b - - 5 3
The node count is: 18
making move 5250
8/Q7/8/2p2p2/1k1n4/1PqK4/5P2/5B2 w - - 6 4
CHECKMATE! Black Wins!
```

We can indeed see that node counts have improved.

Now that we have seen how alpha beta improves efficiency of our program in terms of time, the question we ask as computer scientists is "can we do better?"

4 Transposition Table

The answer to the question asked in the last section is yes! When we do our iterative deepening search, we encounter several positions more than once. We can store information about a position when we encounter it the first time so that we can make use of it in the future when we encounter it again. Clearly the first thing that comes to mind is to store the evaluation value of the position in the table. However, we have to be very careful with that. The value may have been discovered at a different depth than the one we are searching for at currently. In addition to that, the value may not be an exact value, but rather simply a lower or upper bound if it resulted in the pruning of several branches of the tree.

In order to take care of these issues, I created a new class called `TranspositionNode`. The code for the functions and variables in this class is given below:

```
1  public class TranspositionNode {
2    private int height;
3    private int nodeType; //0 for PV Node (exact score), 1 for Cut Node (lower bound)
4              // or 2 for All Node (upper bound).
5    private int value;
6
7    public TranspositionNode(int currHeight, int type, int evaluation){
8      height = currHeight;
9      nodeType = type;
10     value = evaluation;
11   }
12   public int getHeight(){
13     return height;
14   }
15
16   public int getType(){
17     return nodeType;
18   }
19
20   public int getValue(){
21     return value;
22   }
23
24 }
```

As seen there are three pieces of information stored in an instance of this class. First we have the height (notice that this is different from depth). This is simply a measure of how far deeper below the depth that the position was processed was the relevent evaluation value found. The second piece of information is the type of the node. A node can be one with an exact evaluation value, a lower bound or an upper bound. This distinction will make sure we don't make any errors when using a transposition table with our alpha beta and give us some added efficiency. Finally, the last information stored in a transposition node is the value of the position.

We now modify our minimax functions to incorporate the transposition table. We will use a hashMap with keys as the Zobrist hash values of each position and values as the transpositionNode with the greatest height corresponding to that position. A new transposition table is initialized each time `getMove` is called.

The updated function `miniMaxABMove` is printed below with comments noting the updates:

```
1  public short miniMaxABMove(Position position, int maxDepth){
2      short [] moves = position.getAllMoves();
3      nodeCount ++;
```

```
    int alpha = Integer.MIN_VALUE;
    int beta = Integer.MAX_VALUE;
    int max = Integer.MIN_VALUE;
    short maxMove = 0;
    int currentDepth = 1;
    int player = position.getToPlay();
    for(short move:moves){
      int minVal = minValue(move, position, maxDepth, currentDepth, player, alpha, beta);
      if(minVal>max){
        max = minVal;
        maxMove = move;
      }
      if(max>=beta){
        //add the value to the transposition table as an upper bound value.
        transpositionTable.put(position.getHashCode(), new TranspositionNode(maxDepth, 2, max));
        mated = true;
        return maxMove;
      }
      if(max>=alpha){
        alpha = minVal;
      }
    }
    //add the value to the transposition table as an exact value.
    transpositionTable.put(position.getHashCode(), new TranspositionNode(maxDepth, 0, max));
    return maxMove;
  }
```

Similarly, the updated functions `minValue` and `maxValue` are given below:

```
public int minValue(short move, Position position, int maxDepth, int currentDepth, int player, int
    alpha, int beta){
    try {
      position.doMove(move);
    } catch (IllegalMoveException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
    int newDepth = currentDepth + 1;
    nodeCount++;

    if(position.isTerminal() || newDepth>maxDepth ){
      int util = utility2(position, player, currentDepth);
      //add to transposition table as exact value.
      transpositionTable.put(position.getHashCode(), new TranspositionNode(maxDepth-currentDepth,
          0, util));
      position.undoMove();
      return util;
    }


    //check if in transposition table.
    if(transpositionTable.containsKey(position.getHashCode())){
      TranspositionNode current = transpositionTable.get(position.getHashCode());
      //if height is greater than the height we will be searching to
      if(current.getHeight()>=maxDepth-currentDepth){
```

```
        // if the node is an exact value node.
        if(current.getType()==0){
          //simply return that value.
          position.undoMove();
          return current.getValue();
        }else if(current.getType()==1){
          //if it is a lower bound node, check if it is less than alpha.
          if(current.getValue()<=alpha){
            //if so then return alpha.
            position.undoMove();
            return alpha;
          }
        }
      }
    }
    int min = Integer.MAX_VALUE;
    for(short nextMove:position.getAllMoves()){
      int maxVal = maxValue(nextMove, position, maxDepth, newDepth, player, alpha, beta);
      if(maxVal<min){
        min = maxVal;
      }
      if(min<=alpha){
        //add node as lower bound node.
        transpositionTable.put(position.getHashCode(), new TranspositionNode(maxDepth-currentDepth,
            1, min));
        position.undoMove();
        return min;
      }
      if(min<beta){
        beta = min;
      }
    }

    //add node as exact node.
    transpositionTable.put(position.getHashCode(), new TranspositionNode(maxDepth-currentDepth, 0,
        min));
    position.undoMove();
    return min;
  }
```

maxMove is very similar to the code above as shown below:

```
public int maxValue(short move, Position position, int maxDepth, int currentDepth, int player, int
    alpha, int beta){
    try {
      position.doMove(move);
    } catch (IllegalMoveException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
```

```
8       int newDepth = currentDepth + 1;
9       nodeCount++;
10
11      if(position.isTerminal()){
12        int util = utility2(position, player, currentDepth);
13        //add node as exact node.
14        transpositionTable.put(position.getHashCode(), new TranspositionNode(maxDepth-currentDepth,
              0, util));
15        position.undoMove();
16        return -1*util;
17      }
18
19      if(newDepth > maxDepth){
20        int util = utility2(position, player, currentDepth);
21
22        if(player==0){
23          transpositionTable.put(position.getHashCode(), new TranspositionNode(maxDepth-currentDepth,
                0, util));
24          position.undoMove();
25          return util;
26        }else{
27          transpositionTable.put(position.getHashCode(), new TranspositionNode(maxDepth-currentDepth,
                0, -1*util));
28          position.undoMove();
29          return -1*util;
30        }
31      }
32
33
34      //similar to what was done in minValue().
35      if(transpositionTable.containsKey(position.getHashCode())){
36        TranspositionNode current = transpositionTable.get(position.getHashCode());
37        if(current.getHeight()>=maxDepth-currentDepth){
38          if(current.getType()==0){
39            position.undoMove();
40            return current.getValue();
41            // if node is an upper bound node.
42          }else if(current.getType()==2){
43            //check if it is greater than beta.
44            if(current.getValue()>=beta){
45              //if so then return beta.
46              position.undoMove();
47              return beta;
48            }
49          }
50        }
51      }
52
53
54
55      int max = Integer.MIN_VALUE;
56      for(short nextMove:position.getAllMoves()){
57        int minVal = minValue(nextMove, position, maxDepth, newDepth, player, alpha, beta);
58        if(minVal>max){
59          max = minVal;
60        }
```

```
61        if(max>=beta){
62          //add node as upper bound node.
63          transpositionTable.put(position.getHashCode(), new TranspositionNode(maxDepth-currentDepth,
                2, max));
64          position.undoMove();
65          return max;
66        }
67        if(max>alpha){
68          alpha = max;
69        }
70      }
71
72      //add node as exact node.
73      transpositionTable.put(position.getHashCode(), new TranspositionNode(maxDepth-currentDepth, 0,
            max));
74      position.undoMove();
75      return max;
76    }
```

The comments in the code of the above two functions clearly show where and how the transposition table is used and what we should do when the node type is an upper or lower bound rather than an exact value. For example, in `minValue` if the node turns out to be a lower bound, then we compare it to the current alpha. If it is lower than alpha then we simply return alpha as (as it's a tighter bound). You can also see in the above code that we only consider using a value in the transposition table if the height of that node is less than the remaining height of the search.

Once again, like alpha beta, transposition search does not improve performance of the program at a specific depth, rather it simply makes it faster. Therefore, given the same configuration and the same MAXDEPTH it should return the same moves as minimax and alpha beta. To show this, we test it on the Comparer-Case. As predicted, our program did indeed return the same results:

```
making move -26796
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P2/N1P2N1q/PP1P1PPP/R1BQK2R b KQ - 0 10
The NodeCount is  179384
making move -27753
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P2/N1P2N2/PP1P1PqP/R1BQK2R w KQ - 0 11
making move 6101
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1P1PqP/R1BQK2R b KQ - 1 11
The NodeCount is  14488
making move -28210
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1P1P1P/R1BQK2q w Q - 0 12
making move 4868
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1PKP1P/R1BQ3q b - - 1 12
The NodeCount is  20956
making move 6074
r2k1bnr/pp2p1p1/2p3Bp/3p4/5PbN/N1P5/PP1PKP1P/R1BQ3q w - - 2 13
```

The above results also show that using alpha beta with transposition table is indeed faster. This is verified by the fact that for each move, alpha beta with transposition table visits less nodes than simple alpha beta (179,384 compared to 210,317 for the first move, 14,488 compared to 16,632 for the second move and 20,956 compared to 23,104 for the third move). In fact, we were once again able to increase search depth by 1 on most occasions with a MAXDEPTH of 7 performing almost instantaneously while MAXDEPTH of 8 performing more slowly.

It is extremely important to note that the increase in speed by using a transposition table comes at a cost in terms of memory. The transposition table stores a very high amount of chess positions and therefore takes up a lot of memory. In fact, for higher depths, even if speed is not an issue, memory certainly may become one. Therefore, when using a transposition table, we should be aware of this tradeoff between speed and memory.

Finally, let's test our transformation table for our benchmark 3-move-mate case. We get the following results:

```
making move 4804
8/Q7/8/2p2p2/1k1nq3/1P6/3K1P2/5B2 b - - 1 1
The NodeCount is  35234
making move 4764
8/Q7/8/2p2p2/1k1n4/1P6/2qK1P2/5B2 w - - 2 2
making move 5387
8/Q7/8/2p2p2/1k1n4/1P2K3/2q2P2/5B2 b - - 3 2
The NodeCount is  936
making move 4234
8/Q7/8/2p2p2/1k1n4/1P2K3/5P2/2q2B2 w - - 4 3
making move 5332
8/Q7/8/2p2p2/1k1n4/1P1K4/5P2/2q2B2 b - - 5 3
The NodeCount is  18
making move 5250
8/Q7/8/2p2p2/1k1n4/1PqK4/5P2/5B2 w - - 6 4
CHECKMATE! Black Wins!
```

We have now covered most of the basic improvements to our program. However, we won't let that prevent us from asking "can we do better?"

5 Bonus Improvements

5.1 Improved Evaluation Function

While the creator of Chesspresso put in a great deal of work to build this library and we are ever so thankful to him, his evaluation function can certainly be improved. I therefore write my own `getMaterial` function. There are multiple pieces to my evaluation function and I will explain each one of them and their purpose. I would like to cite http://www.chessbin.com/post/chess-board-evaluation.aspx for most inspiring my evaluation function. I have also used piece weights and piece matrices from that website.

5.1.1 Weighted Piece Difference

Our first component is essentially what Chesspresso's original evaluation function was, a weighted sum of piece differences between the two opposing sides. I used the following piece weights:

- Pawn weight = 100
- knight weight = 320
- bishop weight = 325
- rook weight = 500
- queen weight = 975

5.1.2 Piece Square Tables

Different pieces are more effective at different positions on the chess board. For example, a pawn is definitely better the further up it is. We therefore add/subtract to the evaluation score different numbers based on where each piece is on that board. I use the following piece square tables:

```
//We want to essentially encourage pawns to move forward.
  private static short[] pawnTable = new short[]
      {
       0,  0,  0,  0,  0,  0,  0,  0,
      50, 50, 50, 50, 50, 50, 50, 50,
      10, 10, 20, 30, 30, 20, 10, 10,
       5,  5, 10, 27, 27, 10, 5, 5,
       0,  0,  0, 25, 25, 0,  0,  0,
       5, -5,-10, 0,  0,-10, -5, 5,
       5, 10, 10,-25,-25, 10, 10, 5,
       0,  0,  0,  0,  0,  0,  0,  0
  };

  //knights are better towards the center, so push them there.
  private static short[] knightTable = new short[]
      {
          -50,-40,-30,-30,-30,-30,-40,-50,
          -40,-20, 0,  0,  0,  0,-20,-40,
          -30,  0, 10, 15, 15, 10, 0,-30,
          -30,  5, 15, 20, 20, 15, 5,-30,
          -30,  0, 15, 20, 20, 15, 0,-30,
          -30,  5, 10, 15, 15, 10, 5,-30,
          -40,-20, 0,  5,  5,  0,-20,-40,
          -50,-40,-20,-30,-30,-20,-40,-50,
      };

  //Again bishops are better in the center.
  private static short[] bishopTable = new short[]
      {
          -20,-10,-10,-10,-10,-10,-10,-20,
          -10,  0,  0,  0,  0,  0,  0,-10,
          -10,  0,  5, 10, 10, 5,  0,-10,
          -10,  5,  5, 10, 10, 5,  5,-10,
          -10,  0, 10, 10, 10, 10, 0,-10,
          -10, 10, 10, 10, 10, 10, 10,-10,
          -10,  5,  0,  0,  0,  0,  5,-10,
          -20,-10,-40,-10,-10,-40,-10,-20,
      };

  //King should stay back and let the other pieces defend him.
  private static short[] kingTable = new short[]
      {
        -30, -40, -40, -50, -50, -40, -40, -30,
        -30, -40, -40, -50, -50, -40, -40, -30,
        -30, -40, -40, -50, -50, -40, -40, -30,
        -30, -40, -40, -50, -50, -40, -40, -30,
        -20, -30, -30, -40, -40, -30, -30, -20,
        -10, -20, -20, -20, -20, -20, -20, -10,
         20,  20,   0,   0,   0,   0,  20,  20,
         20,  30,  10,   0,   0,  10,  30,  20
```

```
51      };
```

5.1.3 Doubled, Isolated and Passed Pawns

James Mason said "Every pawn is a potential queen". Indeed pawns are extremely important in a game of chess. We therefore take into account three important states of a pawn, namely doubled, isolated and passed.

A doubled pawn is when two pawns of the same color reside on the same file. Doubled pawns are considered a weakness as they are incapable of defending each other. Therefore, we deduct a few points for a doubled pawn in our evaluation function.

An isolated pawn is a pawn which has no friendly pawn on an adjacent file. Once again, isolated pawns are a weakness as they cannot be protected by any other pawn. Therefore, we deduct a few points for an isolated pawn in our evaluation function.

A passed pawn is a pawn with no opposing pawns to prevent it from advancing to the eighth rank. This pawn is most likely to be a potential queen compared to all other pawns. Therefore, we add a few points to the score for every passed pawn.

5.1.4 Mobility

A chess configuration where a player has 15 playable moves is certainly better than a chess configuration where a player has only 3 playable moves. Higher mobility is indeed an advantage. Therefore we add a few points for mobility.

5.1.5 Function

Taking all of the above factors into account, below is my evaluation function:

```
1  public int getMaterial (Position position){
2      int player = position.getToPlay();
3      int score = 0;
4      int piece = 0;
5      for (int i = 0; i<64; i++){
6        piece = position.getStone(i);
7        if(piece == Chess.WHITE_PAWN){
8          //add the pawn weight.
9          score += pawnMultiple;
10         score += pawnTable[i]; //add the piece table value.
11
12         //Doubled Pawn.
13         for (int j = i; j<64; j=j+8){
14           if(j==i){
15             continue;
16           }
17           if (position.getStone(j)==Chess.WHITE_PAWN){
18             score -= 50;
19             break;
20           }
21         }
22
```

```
        // Isolated Pawns.
        if(i%8==0){
          int pawn = 0;
          for(int j = i+1; j<64; j=j+8){
            if(position.getStone(j)==Chess.WHITE_PAWN){
              pawn++;
              break;
            }
          }
          if(pawn<1)
            score -=50;
        }else if(i%8 == 7){
          int pawn = 0;
          for(int j = i-1; j<64; j=j+8){
            if(position.getStone(j)==Chess.WHITE_PAWN){
              pawn++;
              break;
            }
          }
          if(pawn<1)
            score -=50;
        }else{
          int pawn = 0;
          for(int j = i-1; j<64; j=j+8){
            if(position.getStone(j)==Chess.WHITE_PAWN){
              pawn++;
              break;
            }
          }

          for(int j = i+1; j<64; j=j+8){
            if(position.getStone(j)==Chess.WHITE_PAWN){
              pawn++;
              break;
            }
          }
          if(pawn<1)
            score-=50;
        }



        //Passed Pawn.
        int pawn = 0;
        for(int j = i; j<64; j = j+8){
          if(position.getStone(j)==Chess.BLACK_PAWN){
            pawn++;
            break;
          }
        }
        if(pawn<1){
          score+=50;
        }

      }else if(piece == Chess.WHITE_BISHOP){
        score += bishopMultiple;
```

```
        score += bishopTable[i];
      }else if(piece == Chess.WHITE_KNIGHT){
        score += knightMultiple;
        score += knightTable[i];
      }else if(piece == Chess.WHITE_QUEEN){
        score += queenMultiple;
      }else if(piece == Chess.WHITE_ROOK){
        score += rookMultiple;
      }else if(piece == Chess.WHITE_KING){
        score += kingTable[i];
      }else if(piece == Chess.BLACK_PAWN){
        score -= pawnMultiple;
        score -= pawnTable[63-i];

        //Doubled Pawn.
        for (int j = i; j>0; j=j-8){
          if(j==i){
            continue;
          }
          if (position.getStone(j)==Chess.BLACK_PAWN){
            score += 50;
            break;
          }
        }

        // Isolated Pawns.
        if(i%8==0){
          int pawn = 0;
          for(int j = i+1; j>0; j=j-8){
            if(position.getStone(j)==Chess.BLACK_PAWN){
              pawn++;
              break;
            }
          }
          if(pawn<1)
            score +=50;
        }else if(i%8 == 7){
          int pawn = 0;
          for(int j = i-1; j>0; j=j-8){
            if(position.getStone(j)==Chess.BLACK_PAWN){
              pawn++;
              break;
            }
          }
          if(pawn<1)
            score +=50;
        }else{
          int pawn = 0;
          for(int j = i-1; j>0; j=j-8){
            if(position.getStone(j)==Chess.BLACK_PAWN){
              pawn++;
              break;
            }
          }

          for(int j = i+1; j>0; j=j-8){
```

```
            if(position.getStone(j)==Chess.BLACK_PAWN){
              pawn++;
              break;
            }
          }
          if(pawn<1)
            score+=50;
        }


        //Passed Pawn.
        int pawn = 0;
        for(int j = i; j>0; j = j-8){
          if(position.getStone(j)==Chess.WHITE_PAWN){
            pawn++;
            break;
          }
        }
        if(pawn<1){
          score-=50;
        }


      }else if(piece == Chess.BLACK_BISHOP){
        score -= bishopMultiple;
        score -= bishopTable[63-i];
      }else if(piece == Chess.BLACK_KNIGHT){
        score -= knightMultiple;
        score -= knightTable[63-i];
      }else if(piece == Chess.BLACK_QUEEN){
        score -= queenMultiple;
      }else if(piece == Chess.BLACK_ROOK){
        score -= rookMultiple;
      }else if(piece == Chess.BLACK_KING){
        score -= kingTable[63-i];
      }
    }
    //get Mobility
    int mobility = position.getAllMoves().length;
    position.toggleToPlay();
    mobility = mobility - position.getAllMoves().length;
    position.toggleToPlay();
    if(player == 0){
      return score + 10*mobility;
    }else{
      return (-1*score) + 10*mobility;
    }

  }
```

Keep in mind that this time, the improvement in our program is in performance and not necessarily in time. Indeed this comprehensive evaluation function will make our program stronger and more likely to find the best move for a configuration.

5.2 Opening Book

One notable weakness of our program is at the beginning of the game. At the beginning, we may not have any potential captures in sight when given a certain MAXDEPTH and therefore our program may end up making somewhat of a random move. Fortunately people have been playing chess for hundreds of years and we can learn from them. We have been provided a book of 120 openings by some very good chess players. We will get our program to learn from these openings. Basically, we will store all these openings, and whenever we encounter a position that corresponds to a position in one of the openings, we will play the move that the expert chess player played in that game without doing any search. This will mostly occur at the beginning of the game. If our position does not correspond to a position in any of the openings, then we simply do our regular search. This will allow us to do more interesting openings at the start of the game.

I read the file and create the list of openings in my constructor as follows:

```
public MinimaxABAIEVbookQ(){
    openingBook = new ArrayList<Game>();
    URL url = this.getClass().getResource("book.pgn");
    File f = null;
    try {
      f = new File(url.toURI());
    } catch (URISyntaxException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
    FileInputStream fis = null;
    try {
      fis = new FileInputStream(f);
    } catch (FileNotFoundException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
    PGNReader pgnReader = new PGNReader(fis, "book.pgn");
    //hack: we know there are only 120 games in the opening book
    for (int i = 0; i < 120; i++) {
      Game g = null;
    try {
      g = pgnReader.parseGame();
    } catch (PGNSyntaxError e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
      openingBook.add(g);
    }
  }
```

In addition to that, we add the following code at the beginning `getMove` method:

```
//if it is the absolute start position, simply play a random opening
    //from the list of openings.
    if(position.isStartPosition() && position.getToPlay()==0){
      int rand = new Random().nextInt(120);
      Game g = openingBook.get(rand);
```

```
      g.gotoPosition(position);
      System.out.println("We are doing a book position! " + g.getInfoString());
      return g.getNextShortMove();
    }
    //cycle through openings.
    for(Game g:openingBook){
      //if the opening does not have our current position, ignore it.
      if(!g.containsPosition(position)){
        continue;
      }else{
        //go to that position on the opening and play the next move.
        g.gotoPosition(position);
        if(g.hasNextMove()){
          System.out.println("We are doing a book position! " + g.getInfoString());
          return g.getNextShortMove();
        }
      }
```

While using openings from experts does save time each time an opening is played, since openings will usually only be played at the very start of the game, it is overall not a time saving improvement. However, it does improve the performance of the program by making it play good moves right from the start.

5.3 Null Move Heuristic

Having discussed two performance enhancing improvements, let's look at another time saving improvement. Null move heuristic is based on an observation that in chess for the side to move, there is almost always a better move than doing nothing. So if that player performed a null move at that point and still has a strong enough position to produce a cutoff, then the current position would almost certainly produce a cutoff if the current player actually moved. Note that the word *almost* was used in the observation. Indeed there are positions where the best move would be to make a null move called Zugzwang moves. Since these are quite rare, for our program we will ignore these moves.

In order to implement Null Move Heuristic, before searching the deeper game tree at any level, we do a null move and check the subsequent position upto a depth of maxDepth-2. If that value is less than alpha in minValue or greater than beta in maxValue, we simply return alpha or beta respectively without searching anything else. If it isn't, then we proceed with our search as normal. Since my implementation of `minValue` and `maxValue` depended on the move, I had to create helper functions `minValue2` and `maxValue2`. To our previous `minValue` function, we add the following code right after doing our cutoff tests but before starting our search:

```
if(!position.isCheck()){
      //make a null move.
      position.toggleToPlay();
      if(position.isLegal()){
        int nullVal = maxValue2(position, maxDepth - 2, newDepth, player, alpha, beta);
        if(nullVal<=alpha){
          position.toggleToPlay();
          position.undoMove();
          return alpha;
        }
      }
           //undo null move.
      position.toggleToPlay();
```

```
    }
```

In addition to that, the code for `minValue2` is given below:

```
public int minValue2(Position position, int maxDepth, int currentDepth, int player, int alpha, int
    beta){
    int newDepth = currentDepth + 1;
    nodeCount++;
    if(position.isTerminal() || newDepth>maxDepth ){
      int util = utility2(position, player, currentDepth);
      return util;
    }

    if(!position.isCheck()){
      position.toggleToPlay();
      if(position.isLegal()){
        int nullVal = maxValue2(position, maxDepth - 2, newDepth, player, alpha, beta);
        if(nullVal<=alpha){
          position.toggleToPlay();
          //System.out.println("Nulled!");
          return alpha;
        }
      }
      position.toggleToPlay();
    }

    int min = Integer.MAX_VALUE;
    for(short nextMove:position.getAllMoves()){
      int maxVal = maxValue(nextMove, position, maxDepth, newDepth, player, alpha, beta);
      if(maxVal<min){
        min = maxVal;
      }
      if(min<=alpha){
        return min;
      }
      if(min<beta){
        beta = min;
      }
    }

    return min;
  }
```

The code for the additions to `maxValue` and `maxValue2` is very similar and therefore omitted.

Since this improvement is simply a speed improvement, it is again expected to make the same moves as minimax and alpha beta at the same depth with the same evaluation function. To verify this, we test it on the Comparer-Case. As predicted, our program did indeed return the same results:

```
making move -26796
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P2/N1P2N1q/PP1P1PPP/R1BQK2R b KQ - 0 10
The node count is:  27458
making move -27753
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P2/N1P2N2/PP1P1PqP/R1BQK2R w KQ - 0 11
making move 6101
```

```
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1P1PqP/R1BQK2R b KQ - 1 11
The node count is:  8275
making move -28210
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1P1P1P/R1BQK2q w Q - 0 12
making move 4868
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1PKP1P/R1BQ3q b - - 1 12
The node count is:  9553
making move 6074
r2k1bnr/pp2p1p1/2p3Bp/3p4/5PbN/N1P5/PP1PKP1P/R1BQ3q w - - 2 13
```

We can clearly note very considerable decreases in the node counts. Finally, we test this with our benchmark 3-move-mate case with the usual MAXDEPTH of 7 and get the following result:

```
making move 4804
8/Q7/8/2p2p2/1k1nq3/1P6/3K1P2/5B2 b - - 1 1
The node count is:  15801
making move 4764
8/Q7/8/2p2p2/1k1n4/1P6/2qK1P2/5B2 w - - 2 2
making move 5387
8/Q7/8/2p2p2/1k1n4/1P2K3/2q2P2/5B2 b - - 3 2
The node count is:  452
making move 4234
8/Q7/8/2p2p2/1k1n4/1P2K3/5P2/2q2B2 w - - 4 3
making move 5332
8/Q7/8/2p2p2/1k1n4/1P1K4/5P2/2q2B2 b - - 5 3
The node count is:  18
making move 5250
8/Q7/8/2p2p2/1k1n4/1PqK4/5P2/5B2 w - - 6 4
CHECKMATE! Black Wins!
```

This time we explore less nodes for two moves and equal nodes as alpha beta for the third move.

Clearly, null move heursitic increases speeds tremendously. With null move heuristic, my program was able to reach the previously unreachable MAXDEPTH of 9!

5.4 Quiescence search

The cutoff in minimax and alpha beta suffers from a common problem known as the horizon effect. Suppose we are searching to a maxdepth of 5. The best move at that depth could be detrimental at depth 6 (for example could result in the loss of a queen). However, since our program only searched to a depth of 5, it was unable to see that coming.

One way to reduce the horizon effect is to use quiescence search. Under quiescence search, we distinguish positions between quiet positions and interesting positions. We will search quiet positions to a lesser depth and interesting positions to a higher depth, under the assumption that most problems from the horizon effect will occur at interesting positions (for example, it could be a trap!). We define quiet to be a position from which no capturing moves can be made and an interesting position as one where at least 1 capturing move can be made. In my implementation MAXDEPTH is usually 4 while interesting positions are searched upto a depth of 7.

Like with null move heuristic, I created helper methods, `minQuiescence` and `maxQuiescence` to help with quiescence search. Part of our cutoff test in maxValue and minValue now becomes:

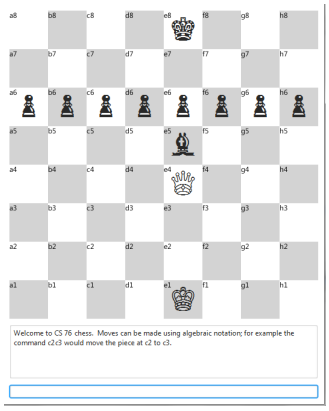
```
if(newDepth>maxDepth){
```

```
if(position.getAllCapturingMoves().length==0){
int util = utility3(position, player, currentDepth);
position.undoMove();
return util;
}else{
int util = minQuiescence(position, player, alpha, beta, currentDepth);
position.undoMove();
return util;
}
}
```

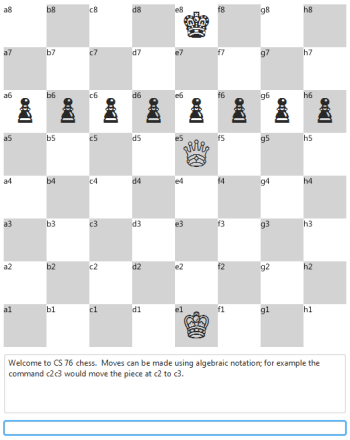
Below is the code for the `minQuiescence` function:

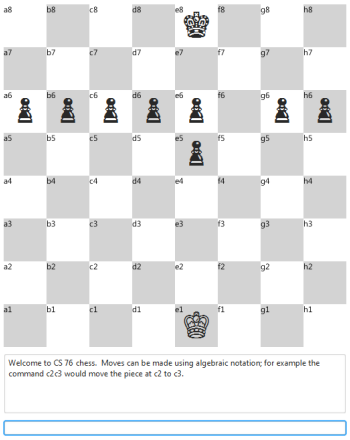
```
public int minQuiescence(Position position, int player, int alpha, int beta, int depth){
    nodeCount++;
    if(position.isTerminal() || position.getAllCapturingMoves().length==0 || depth>7){
      int util = utility3(position, player, 0);
      return util;
    }

    int min = Integer.MAX_VALUE;
    for(short move:position.getAllCapturingMoves()){
      try {
        position.doMove(move);
      } catch (IllegalMoveException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
      int maxVal = maxQuiescence(position, player, alpha, beta, depth+1);
      if(maxVal<min){
        min = maxVal;
      }
      if(min<=alpha){
        position.undoMove();
        return min;
      }
      if(min<beta){
        beta = min;
      }
      position.undoMove();
    }
    return min;
  }
```

Once again, the code for `maxQuiescence` is similar and therefore omitted. The improvement can clearly be seen by the following example. Consider the following start position:

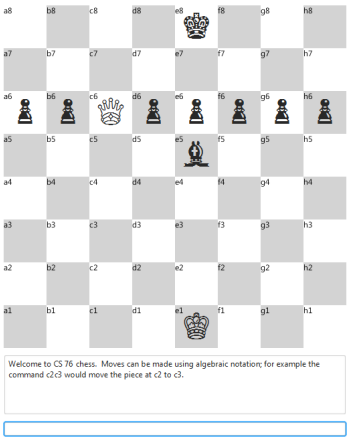


If we play minimaxAI with a depth of 1 as white, it will see the opportunity to take the bishop with the queen but will fail to see the pawn taking the queen after that, giving the following two moves:





However, with running quiescence search with a MAXDEPTH of 1, it will realize the queen capture. Therefore, instead of capturing the bishop, it will do the following move thereby avoiding queen capture:



5.5 Move Reordering

The power of the transposition table does not only lie in providing the dynamic programming that it does. We can also use it to further improve our alpha beta search. If a move was thought of as the best move at a search of lower depth, that increases the probability that it is also the best move at a higher depth. Using this rationale, we know that the transposition table stores position information for the previous depth and can therefore sort the moves based on that information for the alpha beta search in the following depth in iterative deepening.

We will use a priority queue to sort the moves and a linked list to store the sorted moves. We will add the following code to our `maxValue` function right before we start our alpha beta search:

```
1  short [] moves = position.getAllMoves();
2
3
4      PriorityQueue<MoveNode> q = new PriorityQueue<MoveNode>();
5      LinkedList<Short> ordered = new LinkedList<Short>();
6          //get all moves.
7      for(short theMove:moves){
8          //do the move.
9        try {
10         position.doMove(theMove);
11       } catch (IllegalMoveException e) {
12         // TODO Auto-generated catch block
13         e.printStackTrace();
14       }
15           //if that move was in the transpostion table.
16           if(transpositionTable.containsKey(position.getHashCode())){
17         //add that move to the priority queue.
18         q.add(new MoveNode(theMove, transpositionTable.get(position.getHashCode()).getValue()));
19       }
20       position.undoMove();
21     }
22
23         //add the moves in the priority queue based on priority first to the linked list.
24     while(!q.isEmpty()){
25       short theMove = q.remove().move;
26       ordered.add(theMove);
27     }
28
29         //add all remaining moves at end of linked list.
30     for(short theMove:moves){
31       if(!ordered.contains(theMove)){
32         ordered.add(theMove);
33       }
34     }
```

After that, instead of doing our alpha beta search by iterating over the usual array of all moves, we will iterate over the linked list `ordered`. We add similar code to `minValue` but instead of doing ordered.add(theMove) in line 26, we will do ordered.addFirst(theMove) to reverse the order.

Again since this is simply a time saving algorithm, we can test it on the Comparer-Case for the usual depth of 5. We get the following result:

```
making move -26796
```

```
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P2/N1P2N1q/PP1P1PPP/R1BQK2R b KQ - 0 10
The node Count is:  11360
making move -27753
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P2/N1P2N2/PP1P1PqP/R1BQK2R w KQ - 0 11
making move 6101
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1P1PqP/R1BQK2R b KQ - 1 11
The node Count is:  6307
making move -28210
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1P1P1P/R1BQK2q w Q - 0 12
making move 4868
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1PKP1P/R1BQ3q b - - 1 12
The node Count is:  8450
making move 6074
r2k1bnr/pp2p1p1/2p3Bp/3p4/5PbN/N1P5/PP1PKP1P/R1BQ3q w - - 2 13
```

Once again, the node counts for all three moves decrease dramatically (even less than null move heuristic!). At the same time, as predicted, the program does the same moves as before.

Lastly, let us check this with our bechmark 3-move-mate case with a depth of 7. We get the following result:

```
making move 4804
8/Q7/8/2p2p2/1k1nq3/1P6/3K1P2/5B2 b - - 1 1
The node Count is:  16394
making move 4764
8/Q7/8/2p2p2/1k1n4/1P6/2qK1P2/5B2 w - - 2 2
making move 5387
8/Q7/8/2p2p2/1k1n4/1P2K3/2q2P2/5B2 b - - 3 2
The node Count is:  455
making move 4234
8/Q7/8/2p2p2/1k1n4/1P2K3/5P2/2q2B2 w - - 4 3
making move 5332
8/Q7/8/2p2p2/1k1n4/1P1K4/5P2/2q2B2 b - - 5 3
The node Count is:  18
making move 5250
8/Q7/8/2p2p2/1k1n4/1PqK4/5P2/5B2 w - - 6 4
CHECKMATE! Black Wins!
```

Once again, for two of the moves, we beat simple alpha beta in terms of nodes explored while for the last move we get the same value.

6 Conclusion and Summary of Results

Now that we have looked at several improvements to the program both in terms of performance and speed, let us now combine all the speed improving additions to see the result. We will proceed with testing on our usual two cases. For the Comparer-Case and the usual depth of 5 if we combine alpha beta with a transposition table, move ordering and a null move heuristic, we get the following result:

```
making move -26796
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P2/N1P2N1q/PP1P1PPP/R1BQK2R b KQ - 0 10
The node Count is:  1806
```

```
making move -27753
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P2/N1P2N2/PP1P1PqP/R1BQK2R w KQ - 0 11
making move 6101
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1P1PqP/R1BQK2R b KQ - 1 11
The node Count is:  1639
making move -28210
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1P1P1P/R1BQK2q w Q - 0 12
making move 4868
r1bk1bnr/pp2p1p1/2p3Bp/3p4/5P1N/N1P5/PP1PKP1P/R1BQ3q b - - 1 12
The node Count is:  7681
making move 6074
r2k1bnr/pp2p1p1/2p3Bp/3p4/5PbN/N1P5/PP1PKP1P/R1BQ3q w - - 2 13
```

A highly significant decrease in all three moves.

Similarly, if we run this program on our benchmark 3-move-checkmate case with the usual depth of 7, we get:

```
making move 4804
8/Q7/8/2p2p2/1k1nq3/1P6/3K1P2/5B2 b - - 1 1
The node Count is:  4215
making move 4764
8/Q7/8/2p2p2/1k1n4/1P6/2qK1P2/5B2 w - - 2 2
making move 5387
8/Q7/8/2p2p2/1k1n4/1P2K3/2q2P2/5B2 b - - 3 2
The node Count is:  353
making move 4234
8/Q7/8/2p2p2/1k1n4/1P2K3/5P2/2q2B2 w - - 4 3
making move 5332
8/Q7/8/2p2p2/1k1n4/1P1K4/5P2/2q2B2 b - - 5 3
The node Count is:  18
making move 5250
8/Q7/8/2p2p2/1k1n4/1PqK4/5P2/5B2 w - - 6 4
CHECKMATE! Black Wins!
```

Once again giving lower node counts for the first two moves and equal node count for the third move.

On the next page are two tables containing the summaries of our results for the two cases that we have used to compare the speed improvements. The values in the cells are the node counts for the respective moves (remember lower node count means faster algorithm). We have improved both the speed and quality of the chess program significantly through our improvements! Maybe we can beat the world chess champion one day?

Table for Comparer-Case with MaxDepth = 5

	Minimax	Alpha Beta	Alpha beta with Transposition Table	Transposition Table with Move Ordering	Alpha beta with Null Move Heuristic	Transposition Table with Move Ordering and Null Move Heuristic
First Move	1,070,064	210,317	179,384	11,360	27,458	1,806
Second Move	934,868	16,632	14,488	6,307	8,275	1,639
Third Move	631,791	23,104	20,956	8,450	9,553	7,681

Table for 3-move-mate case with MaxDepth = 7.

	Minimax	Alpha Beta	Alpha beta with Transposition Table	Transposition Table with Move Ordering	Alpha beta with Null Move Heuristic	Transposition Table with Move Ordering and Null Move Heuristic
First Move	8,230,249	38,473	35,234	16,394	15,801	4,215
Second Move	8,077	936	936	455	452	353
Third Move	27	18	18	18	18	18