

# AI Assignment 4 - Chess Game

Tianyuan Zhang

October 20, 2016

## 1 Introduction

The adversarial search in Zero-Sum games is an important area of AI, which can also be applied to many real-world applications also. The Chess game is the representative among these games. In this assignment, I built a Chess AI and tried to optimize it in terms of time and space. In this assignment, I implemented the Minimax Method, Alpha-Beta Pruning, Transposition table, Profile Analysis and Move reordering.

## 2 Minimax Search

### 2.1 Basic Idea

The basic idea of Minimax Search is that whenever we need to make a decision in a game, we are not only evaluating according to the current state of the game, but also the opponent's play - the reaction of the adversary in his/her turn. As it is obviously a tree shape prediction model, we can use a computer to process the tree and try to find the best movement for the current player according to the prediction for the next few moves. In Minimax Search algorithm, the current play always wants its score to be maximum, while its opponent always wants its score to be minimum. Then for each level of the prediction tree we have an evaluation for each node corresponding to the current player, so basically we just enumerate all the possible moves of the whole game. However, as the limitation of the computation power of the computer, we need to limit the max-depth of the search tree and pick the best move found so far.

### 2.2 Code Implementation

#### 2.2.1 Class : MoveVal

This class helps store the move-evaluation pair.

```
1 public class MoveVal{
   private short move;
3  private int val;

5  public MoveVal(){
      move = 0; val = 0;
7  }
   public MoveVal(short m, int v){
8      move = m; val = v;
9  }
11 public void setVal(int v){val = v;}
   public void setMove(short m){move = m;}
13 }
```

### 2.2.2 minimaxIDS

This method is iterative deepening returns a *short* type result which represents the next best move for the current player. This method will call method *maxMove*.

```
1 private short miniMaxIDS(Position position, int maxDepth)
    throws IllegalMoveException{
3     MoveVal bestMove = new MoveVal();
    for(int i = maxDepth - 1; i >= 0; i--){
5         bestMove = maxMove(position, i);
    }
7 }
```

### 2.2.3 maxMove

This method is the *Max* part of the Minimax Search. For a given position of the current player, this method try each possible move for the next player and call *minMove* method to get the evaluation of the opponent, then according to the evaluation of the opponent to choose the best move for the current player.

```
1 private MoveVal maxMove(Position position, int depth)
    throws IllegalMoveException{
3     MoveVal bestMove = new MoveVal((short) 0, Integer.MIN_VALUE);
    if(depth == max_depth || position.isTerminal()){
5         bestMove.setMove(position.getLastShortMove());
        bestMove.setVal(utility(position));
7     } else{
        for(short move : position.getAllMoves()){
9             position.doMove(move);
            int val = (minMove(position, depth + 1)).val;
11            if(bestMove.val < val){
                bestMove.setVal(val);
                bestMove.setMove(move);
13            }
            position.undoMove();
15        }
    }
17 }
    return bestMove;
19 }
```

### 2.2.4 minMove

This method is the *Min* part of the Minimax Search. It will get the position from the method *maxMove* and call *maxMove* for each possible next move. Then it will choose the minimum evaluation among all the evaluation it gets.

```
1 private MoveVal minMove(Position position, int depth)
    throws IllegalMoveException{
3     MoveVal bestMove = new MoveVal((short)0, Integer.MAX_VALUE);
    if(depth == max_depth || position.isTerminal()){
5         bestMove.setMove(position.getLastShortMove());
        bestMove.setVal(utility(position));
7     } else {
```

```

9      for(short move : position.getAllMoves()){
10         position.doMove(move);
11         int val = (maxMove(position, depth + 1)).val;
12         if(bestMove.val > val){
13             bestMove.setMove(move);
14             bestMove.setVal(val);
15         }
16         position.undoMove();
17     }
18     return bestMove;
19 }

```

### 2.2.5 utility

This method is for handling the terminal. When the tree reaches the max-depth or reaches a draw or check mate, it will return the corresponding evaluation.

```

1 private int utility(Position position){
2     if(position.isTerminal() && position.isMate()){
3         if(position.getToPlay() == curPlayer)
4             return Integer.MIN_VALUE;
5         else return Integer.MAX_VALUE;
6     }
7     else if(position.isTerminal()) return 0;
8     else{
9         int value = position.getMaterial() + (int)position.getDomination();
10        if(curPlayer == position.getToPlay())
11            return value;
12        else return -value;
13    }
14 }

```

## 2.3 Result

Here I would like to show two experiments I played with the Maximini Search AI. I set up two games, both of them are Minimax AI v.s. Random AI, but one of them is with max-depth = 1 and another one is with max-depth = 3. The difference between the decision that the AI made is obvious.

### 2.3.1 max-depth = 1

This AI plays like a Chess newbie - it always want to capture the opponent whenever possible. The result is that it can always win the random AI but it will spend a lot of steps. The result shown as follow, we can see that the random AI (Black) has almost nothing left and it cost Minimax AI (White) 19 turns to win.

```
White Wins!  
making move 7534  
N4k2/5Q2/2P1B3/7p/3P1p2/P4P2/P1P3PP/R3K1NR b KQ - 1 19
```

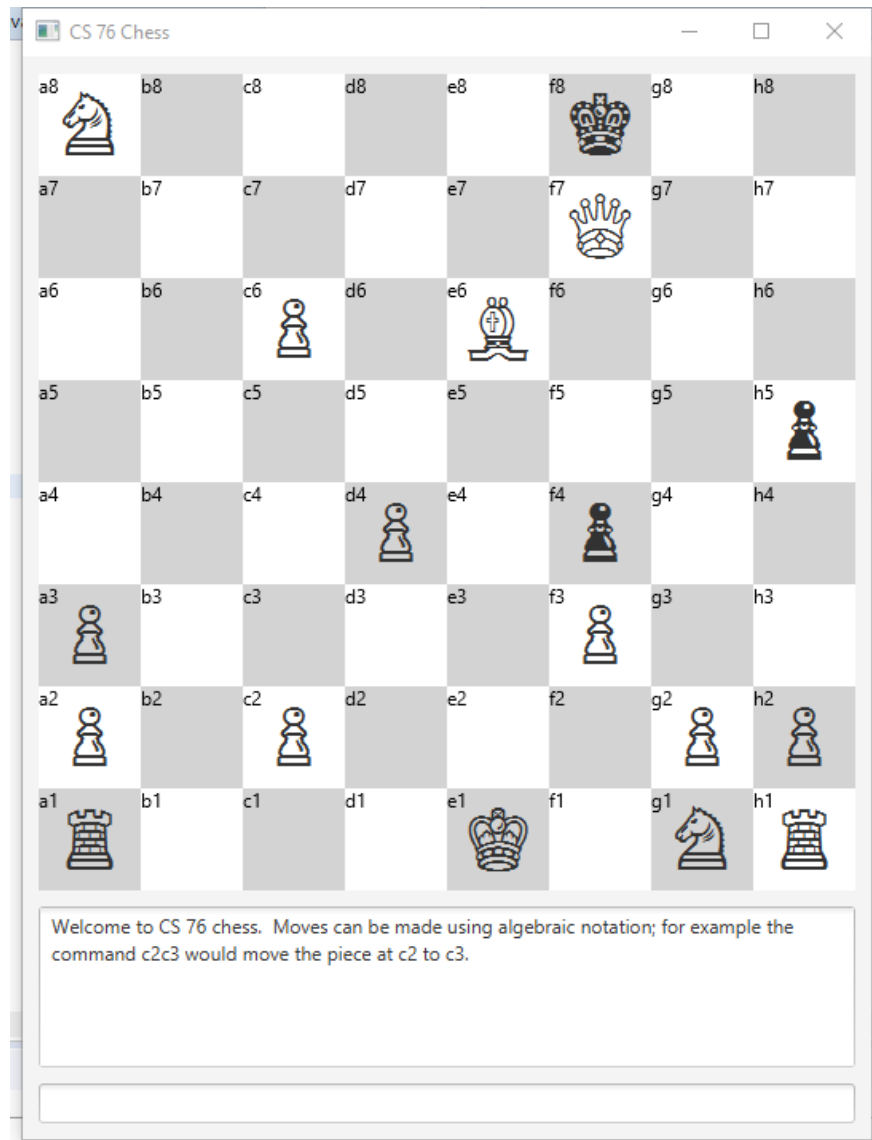


Figure 1: Minimax(depth = 1) AI vs Rnd AI

### 2.3.2 max-depth = 3

This AI plays smarter - it may not capture as many as the AI with depth = 1, but it can always win quickly. We can see that White only capture one Pawn but wins in 8 turns.

```
1 White Wins!  
making move -25251  
3 rnbqk2r/2pp1Q1p/pp1bp1p1/3nP1N1/8/2N5/PPPP1PPP/R1B1KB1R b KQkq - 0 8
```

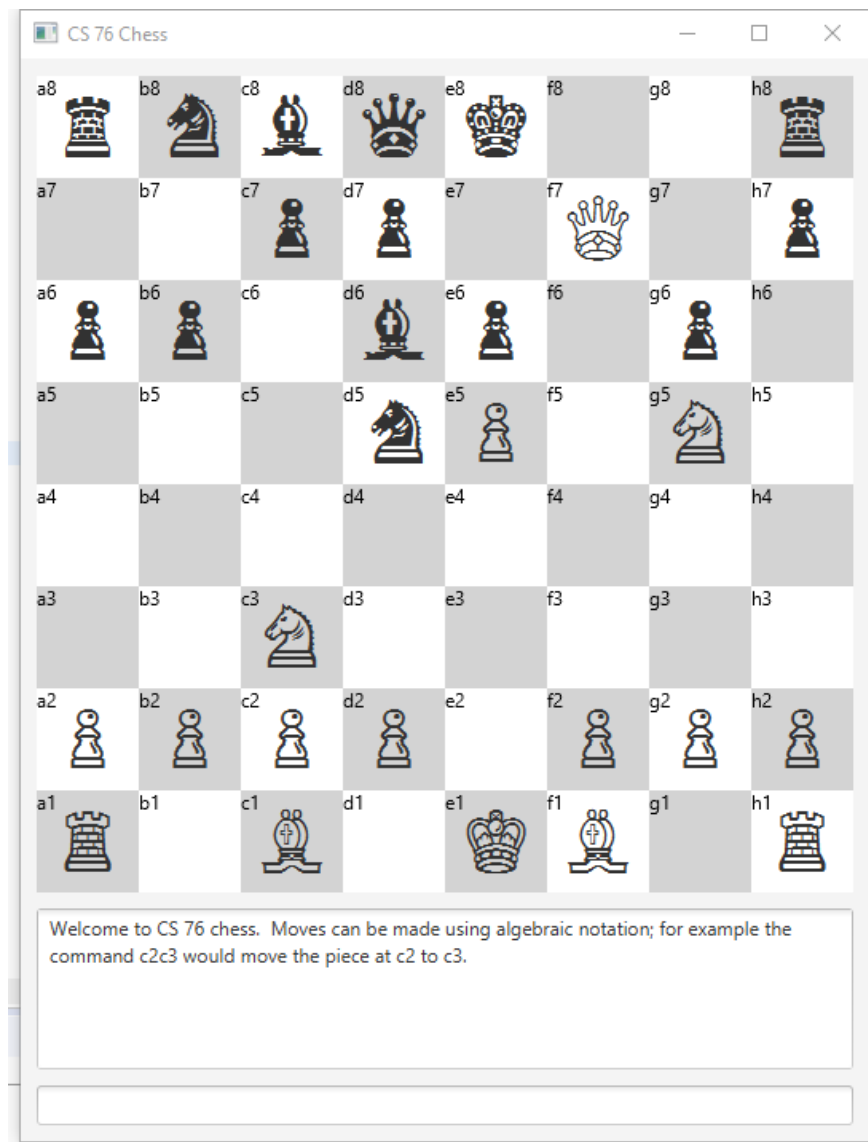


Figure 2: Minimax(depth = 3) AI vs Rnd AI

## 3 Alpha-Beta Pruning

### 3.1 Basic Idea

The basic idea of Alpha-Beta Pruning is to prevent traversal the branch of the tree as much as possible. For the AB Pruning, it will record the current minimum evaluation and maximum evaluation for the node, and if the next step is obviously worse than the current step, then we can prevent traversal that branch.

### 3.2 Code Implementation

Basically the implementation of AB Pruning is a lot like the Minimax Search except that the Alpha-Beta constraints are added.

#### 3.2.1 maxMove

The difference is that lines 16-20 are added to jump out of the loop earlier if possible.

```
1  private MoveVal maxMove(Position position, int depth, int alpha, int beta) throws IllegalMoveException {
2      MoveVal bestMove = new MoveVal((short) 0, Integer.MIN_VALUE);
3      if (depth == max_depth || position.isTerminal()) {
4          bestMove.setMove(position.getLastShortMove());
5          bestMove.setVal(utility(position));
6      } else {
7          for (short move : position.getAllMoves()) {
8              position.doMove(move);
9              int val = (minMove(position, depth + 1, alpha, beta)).val;
10             if (bestMove.val < val) {
11                 bestMove.setVal(val);
12                 bestMove.setMove(move);
13             }
14             position.undoMove();
15             alpha = bestMove.val;
16             if (alpha >= beta) {
17                 bestMove.setVal(beta);
18                 break;
19             }
20         }
21     }
22     return bestMove;
23 }
```

#### 3.2.2 minMove

The difference is that lines 16-20 are added to jump out of the loop earlier if possible.

```
1  private MoveVal minMove(Position position, int depth, int alpha, int beta) throws IllegalMoveException {
2      MoveVal bestMove = new MoveVal((short) 0, Integer.MAX_VALUE);
3      if (depth == max_depth || position.isTerminal()) {
4          bestMove.setMove(position.getLastShortMove());
5          bestMove.setVal(utility(position));
6      } else {
7          for (short move : position.getAllMoves()) {
8              position.doMove(move);
9              int val = (maxMove(position, depth + 1, alpha, beta)).val;
10             if (bestMove.val > val) {
11                 bestMove.setVal(val);
12                 bestMove.setMove(move);
13             }
14             position.undoMove();
15             beta = bestMove.val;
16             if (alpha >= beta) {
17                 bestMove.setVal(alpha);
18                 break;
19             }
20         }
21     }
22     return bestMove;
23 }
```

```

9      position.doMove(move);
      int val = (maxMove(position, depth + 1, alpha, beta)).val;
11     if (bestMove.val > val) {
        bestMove.setMove(move);
13         bestMove.setVal(val);
    }
15     position.undoMove();
    beta = bestMove.val;
17     if(beta <= alpha){
        bestMove.setVal(alpha);
19         break;
    }
21 }
    }
23 return bestMove;
}

```

### 3.3 Result

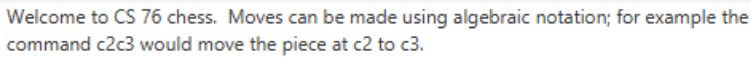
For this part, I majorly want to show the correctness of my AB Pruning implementation and the better performance of AB Pruning over Minimax Search.

#### 3.3.1 Correctness

To check the correctness of the AB Pruning implementation is that given the same situation, AB Pruning should perform the same moves as Minimax Search. So my method is to first let a White(Minimax with depth = 3) vs Black(ABP with depth = 3), then let a White(ABP with depth = 3) vs Black(Minimax with depth = 3). If my AB Pruning was right, then the result of these two players should be exactly the same. The results show that though I switch two AI's position, the winner is always the White side with 29 turns and with the same ending position. Here is the result:

White Wins!

1Bbkq2r/1p3p1p/3p1Q2/2pP4/5P2/2P5/P5PP/2KnR3 b - - 1 29



8



### White(ABP) vs Black(Minimax)

```
1 making move 7030  
1Bbkq2r/1p3p1p/3p1Q2/2pP4/5P2/2P5/P5PP/2KnR3 b - - 1 29  
3 White Wins!
```

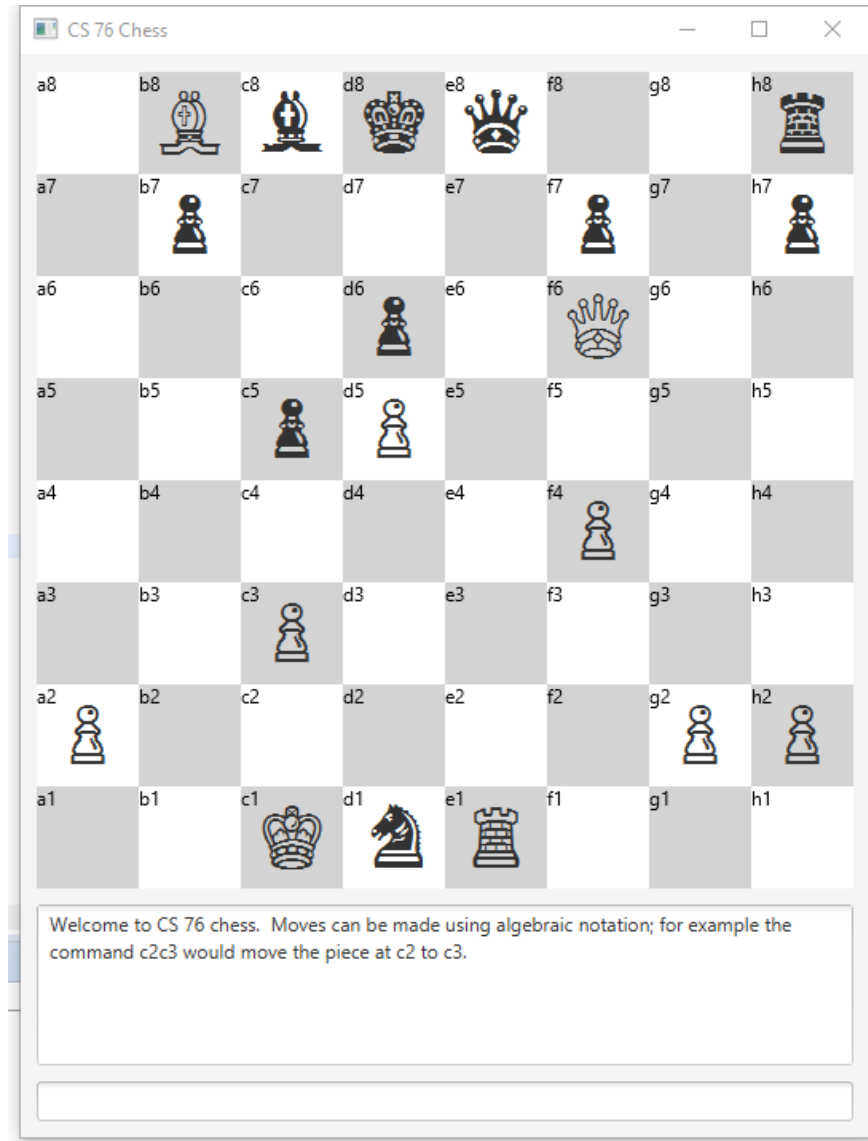


Figure 4: White(ABP) vs Black(Minimax)

### 3.3.2 Performance

To show that the performance of AB Pruning is better than Minimax Search, I also set up two experiment sets - one is Minimax v.s. Minimax and the other one is Minimax v.s. ABP. All the AI have the same max-depth = 3. By showing the total time each AI spend, we can see that AB Pruning beats Minimax regarding performance. Here is the result:

### Minimax vs Minimax

As there are two AIs and both of them are using Minimax Search, we can see that the time spent by them is similar - one is 4080 ms and the other one is 3674 ms.

```
1 White Wins!  
Time Spent: 4080  
3 making move 7030  
1Bbkq2r/1p3p1p/3p1Q2/2pP4/5P2/2P5/P5PP/2KnR3 b - - 1 29  
5 White Wins!  
Time Spent: 3674
```

**Minimax vs ABP** This shows that ABP improves the performance significantly comparing to Minimax even with max-depth = 3. The Minimax AI spent almost the same amount of time as the previous one, 4051 ms, while the ABP AI just spent 888 ms - 3 times quicker than Minimax Search!

```
2 White Wins!  
Time Spent: 4051  
making move 7030  
4 1Bbkq2r/1p3p1p/3p1Q2/2pP4/5P2/2P5/P5PP/2KnR3 b - - 1 29  
White Wins!  
6 Time Spent: 888
```

## 4 Transposition Table

### 4.1 Basic Idea

The basic idea is to have a HashMap store the position we have visited. Need to mention that we also need to consider the depth of the evaluation. We only use the HashMap's move when the HashMap's move are found in a deeper branch than the current one. Otherwise we need to do the search and update the HashMap. In this way, we ensure the accuracy of the search.

### 4.2 Implementation

The implementation is similar to the AB Pruning, I just add a HashMap and each time before I call *maxMove* or *minMove*, I first check the HashMap to see if the result already exists. Here is the code.

```
private MoveVal maxMove(Position position, int depth, int alpha,
2   int beta, int curMaxDepth) throws IllegalMoveException {
    MoveVal bestMove = new MoveVal((short) 0, Integer.MIN_VALUE);
4   if (depth == curMaxDepth || position.isTerminal()) {
        bestMove.setMove(position.getLastShortMove());
6       bestMove.setVal(utility(position));
        // System.out.println(Integer.toString(bestMove.val));
8   } else {
        for (short move : position.getAllMoves()) {
10            position.doMove(move);
            if (transTable.containsKey(position.getHashCode())
12                && (transTable.get(position.getHashCode()).depth
                    >= curMaxDepth) {
14                TransValue tv = transTable.get(position.getHashCode());
                bestMove.setMove(move);
16                bestMove.setVal(tv.val);
            } else {
18                int val = (minMove(position, depth + 1, alpha,
                    beta, curMaxDepth)).val;
20                if (bestMove.val < val) {
                    bestMove.setVal(val);
22                    bestMove.setMove(move);
                }
24                transTable.put(position.getHashCode(),
                    new TransValue(val, curMaxDepth, move));
26            }
            position.undoMove();
28            alpha = bestMove.val;
            if (alpha >= beta) {
30                bestMove.setVal(beta);
                break;
32            }
        }
34    }
    return bestMove;
36 }

private MoveVal minMove(Position position, int depth, int alpha,
38     int beta, int curMaxDepth) throws IllegalMoveException {
```

```

40     MoveVal bestMove = new MoveVal((short) 0, Integer.MAX_VALUE);
41     if (depth == curMaxDepth || position.isTerminal()) {
42         bestMove.setMove(position.getLastShortMove());
43         bestMove.setVal(utility(position));
44     } else {
45         for (short move : position.getAllMoves()) {
46             position.doMove(move);
47             if (transTable.containsKey(position.getHashCode())
48                 && (transTable.get(position.getHashCode()).depth
49                     >= curMaxDepth) {
50                 TransValue tv = transTable.get(position.getHashCode());
51                 bestMove.setMove(move);
52                 bestMove.setVal(tv.val);
53             } else {
54                 int val = (maxMove(position, depth + 1, alpha,
55                     beta, curMaxDepth)).val;
56                 if (bestMove.val > val) {
57                     bestMove.setMove(move);
58                     bestMove.setVal(val);
59                 }
60                 transTable.put(position.getHashCode(),
61                     new TransValue(val, curMaxDepth, move));
62             }
63             position.undoMove();
64             beta = bestMove.val;
65             if (beta <= alpha) {
66                 bestMove.setVal(alpha);
67                 break;
68             }
69         }
70     }
71     return bestMove;
72 }

```

## 5 Profiling(Bonus)

I implemented profiling codes in class *ABP*, *Minimax* and *ABP\_Trans* to record the runtime for each move evaluation. After you run with them, the corresponding *txt* log file will be created. It turns out that for each step the time spent is as follows, from left to right is the time spent on each step for ABP with Transposition Table, ABP, and Minimax. It is clear that Minimax spent most of the time in each step. But interesting things happened when to compare ABP with Transposition table and pure ABP. Sometimes ABP with Transposition table spends more time than ABP but sometimes it saves time. I think it is because insert a new value into the table would cost time, while if a result is already in the table, then ABP with Transposition table can save a lot of time. I think ABP with Transposition table would perform better if max-depth is larger.




 File	 File	 Minimax_L	Edit	For
29	18	93		
26	11	67		
40	39	177		
33	12	172		
13	16	160		
49	27	160		
44	38	137		
58	40	116		
34	38	117		
27	17	92		
55	46	97		
26	20	116		
46	53	118		
41	47	179		
27	26	195		
40	37	179		
57	36	198		
62	96	217		
76	57	242		
76	32	205		
38	19	100		
10	9	127		
7	12	158		
14	44	127		
9	8	104		
16	22	98		
28	23	126		
35	22	132		
20	2	130		

Figure 5: Profiling, left to right, ABP\_Trans, ABP and Minimax

## 6 Move Reordering(Bonus)

The basic idea for the move reordering is that for maximum, it is more likely trigger the cut-off condition if sort the evaluation from a lower level in decreasing order. Similar to minimum step, it is more likely to trigger the cut-off condition if we sort the evaluation from the lower lever in increasing order. The way is simple - just to sort the all next move's evaluation as needed. Need to mention that we just sort the evaluation that has been found before, which means that the evaluation that is in the Transposition Table. If it does not exist, then we assign the evaluation worst possible value.

```
private ArrayList<MoveVal> sortMoves(Position position, boolean turn)
2      throws IllegalMoveException{
    ArrayList<MoveVal> res = new ArrayList<MoveVal>();
4      MoveVal curMove = null;
    for(short move : position.getAllMoves()){
6          position.doMove(move);
          if(transTable.containsKey(position.getHashCode()))
8              curMove = new MoveVal(move,
                  transTable.get(position.getHashCode()).val);
          else
10             curMove = new MoveVal(move, turn ? Integer.MIN_VALUE
12                 : Integer.MAX_VALUE);
          position.undoMove();
14         res.add(curMove);
    }

16     Collections.sort(res, new Comparator<MoveVal>(){
18         @Override
            public int compare(MoveVal m1, MoveVal m2){
20                 return (int) (Math.signum(m1.val - m2.val));
            }
22     });

24     return res;
}
```