Chess AI

Mengjia Kong

February 20, 2014

1 Introduction

Chess is a two-player strategy board game played on a chessboard with 64 squares arranged in an eight-by-eight grid. Millions of people worldwide play chess at home, in parks, clubs, online, by correspondence and in tournaments, which leads to chess becoming one of the world's most popular games for a long history. Besides in life, chess also gets interests from academics. Many combinatorical and topological problems connected to chess were known of for hundreds of years. Because of the features: deterministic, turn-taking, two-player, zero-sum games, perfect information, chess becomes a focus in AI. The most famous AI is *Deep Blue* who beat World Chess Champion *Garry Kasparov*. In 1913, Ernst Zermelo used chess as a foundation for his theory of game strategies, which is considered as one of the predecessors of game theory[1].

In this report, I will introduce some algorithms in game theory applying to chess. The basic structure of chess is provided by the course with supporting from a chess library by Bernhard Seybold. In the next section, a check for a win condition which is added to the model will be introduced. In the section 3,

2 Related work

When applying minimax to chess, we always use a evaluation heuristic function to compute the value of the position for minimax search. There is another paper explains why minimax works not from value dependence. If real numbers are used for position values, they tend to be further apart at lower levels of the game tree. That leads to a larger proportion of more exact positions, where error is less probable.[2]

Alpha-Beta Pruning is one of the most powerful and fundamental improvements of minimax search, especially for perfect information games such as chess. But scholars are trying to adapt it to simultaneous move games by using a simultaneous move alpha-beta pruning(SMAB).[3]

3 Changes of the model

There is no check for a win condition. I have added one to the model at the beginning in `GameHandler` in `ChessClient.java`.

Here is my code:

```
1       // Game over!
2       if (game.position.isTerminal()) {
3         System.out.println("Game over!!!");
4
5         if (game.position.isMate()) { // checkmate
6           if (game.position.getToPlay() == Chess.WHITE)
7             System.out.println("Black Wins!!!");
8           else
9             System.out.println("White Wins");
10        }
```

```
        else
          System.out.println("Draw!!!");

        game = new ChessGame();
        moveMaker[0] = new TextFieldMoveMaker();;
        moveMaker[1] = new TextFieldMoveMaker();;

        return ;
      }
```

By adding this at the beginning of `GameHandler`, every time, before a player makes an action, check whether the game is over first. If `game.position.isTerminal()` is `true`, the game is over. Then check whether the game is someone wins or a draw. Obviously, If this is White's turn and `isMate()` is `true`, White is checkmated. Because Black did an action leading that White cannot move. If `isMate()` is `false` but the game is over, it is a draw.

4 Minimax search

The *Minimax algorithm* computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor's state. The recursion proceeds all the way down to the leaves, then the minimax values are backed up throuth the tree as the recursion unwinds. Furthermore, the minimax algorithm performs a complete depth-first exploration of the game tree. Assume the maximum depth of the tree is m and there are b legal moves at each node, the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is $O(m)$ for generating actions one at a time, like my implementation.

4.1 Implementation of Minimax

I will introduce the whole algorithm of minimax first, then explain cutoff test, utility function and evaluation function in details later.

Here is my code of minimax algorithm:

```
  private short Minimax(Position position) {
    int v = Integer.MIN_VALUE; // keep max value
    short action = 0;   // keep the action leading to max value

    short [] moves = position.getAllMoves();
    for (short move: moves) {
      doMove(position, move);
      int tmp = MIN_Value(position, 1);
      if (tmp > v) {    // find the max value
        v = tmp;
        action = move;
      }
      undoMove(position);
    }
    System.out.println("Simple Minimax Max value: " + v);
    return action;
  }

  private int MAX_Value(Position position, int depth) {
    if (Cutoff_Test(position, depth))
      return Utility(position);

    int v = Integer.MIN_VALUE;
```

```
    for (short move: position.getAllMoves()) {
      doMove(position, move);
      v = Math.max(v, MIN_Value(position, depth+1));
      undoMove(position);
    }

    return v;
  }

  private int MIN_Value(Position position, int depth) {
    if (Cutoff_Test(position, depth))
      return Utility(position);

    int v = Integer.MAX_VALUE;
    for (short move: position.getAllMoves()) {
      doMove(position, move);
      v = Math.min(v, MAX_Value(position, depth+1));
      undoMove(position);
    }

    return v;
  }
```

Here I implemented a depth-limited minimax search. Just like the pseudo-code for minimax in the book, but replaced the `TERMINAL-TEST` with `Cutoff_Test` instead. The basic idea is:

$$Minimax(position) = \begin{cases} Utility(position) & \text{if } Cutoff\_Test(position, depth) \\ max_{a \in Actions(position)} Minimax(Result(position, a)) & \text{if Player(position) = MAX} \\ min_{a \in Actions(position)} Minimax(Result(position, a)) & \text{if Player(position) = MIN} \end{cases}$$

4.1.1 Cutoff Test

In my method, I used `Cutoff_Test` instead of `TERMIANL-TEST`. Because if each minimax needs to find the leaves to get the value, the computation is verty large. I set a `depth_limit` in the class `Minimax`, then set if the game is over or the current depth reaches the `depth_limit` the minimax search needs to cutoff.

Here is my code:

```
  private boolean Cutoff_Test(Position position, int depth) {
    if (position.isTerminal() || depth >= depth_limit)
      return true;
    else
      return false;
  }
```

4.1.2 Utility function

The normal utility function is that if the player wins the game, return 1; if the player loses the game, return -1; if it is a draw, return 0. But I cut off the search before reaching the terminal. It changes a lot. In my method, if the position is not terminal, the search is stopped because of the depth limit, return a random value; else, if the position is checkmate, check who is checkmated now: if the player is the winner, return the maximum integer; if the player is the loser, return the minimum integer. If the position is not checkmate, it is a draw, return 0.

Here is my code:

```
private int Utility(Position position) {
  // if I win, return MAX_VALUE
  //    I lose, return MIN_VALUE
  //    draw, return 0
  //  else return random value
  if (position.isTerminal()) { // terminal
    if (position.isMate()) { // checkmate
      // this turn play is checkmated
      if (position.getToPlay() == myTurn)
        return Integer.MIN_VALUE;
      else
        return Integer.MAX_VALUE;
    }
    else
      return 0;
  }

  return new Random().nextInt();
}
```

4.2 Iterative deepening minimax

Iterative deppening minimax algorithm just add a loop outside of `minimax` with varying `depth_limit`, which is a private component in the class.

Here is the code:

```
private short iterative(Position position) {
  short[] bestmove = new short[depth_limit];
  int d = depth_limit;
  for (depth_limit = 1; depth_limit < d; depth_limit++) {
    bestmove[depth_limit] = Minimax(position);
  }
  //if the time is out
  depth_limit = d;
  return bestmove[depth_limit-1];
}
```

4.3 Experiments and Discussions

To demonstrate the correctness of the program, I will show the wins, loses and draws first; then, discuss the situations in different maximum depth; finally, show some checkmate cases.

4.3.1 Win, lose and draw

When the position comes to that the player will win the game(in Figure 1), the return value is the maximum integer like Figure 2. The output also shows Game over and White wins.

When the position has some steps to win, the minimax search reaches the winning leaf, like figure 3. Then the minimax search will choose the action leading to the maximum integer.

When the position comes to that the player will lose the game(in Figure 4).

If use two players with random AI, when there are few pieces left in game, it is more likely to get a draw.

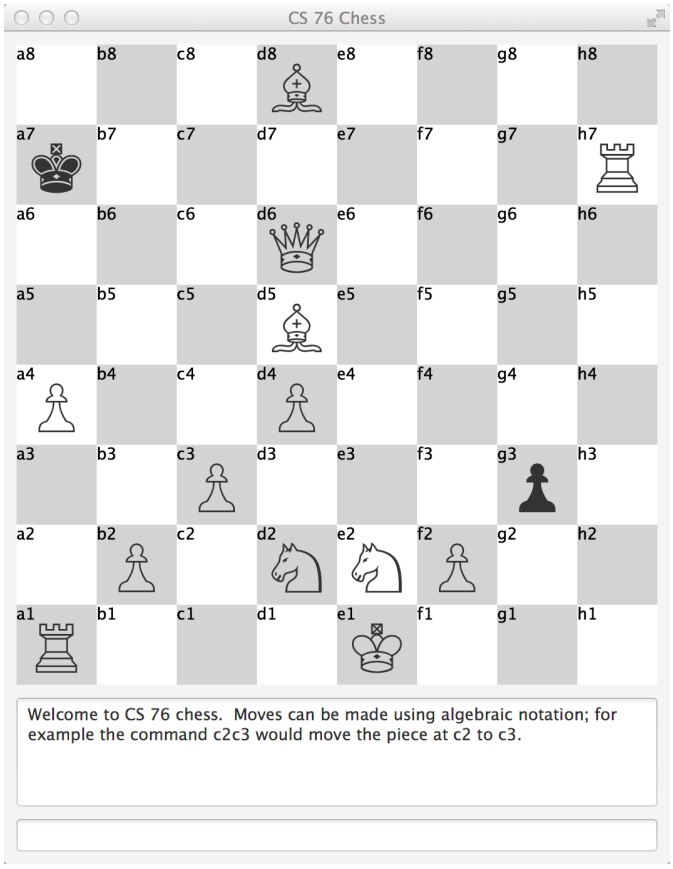

Figure 1: When White wins, the position

4.3.2 Different Maximum Depth

Now, I was using a depth-limit minimax search. Obviously, the results, time, number of visited states may different because of the depth limit.

First, I use a position: White to move, and checkmate-in-one move, to give you the example about the time consuming and number of visited states. The unit of time is ms. The position is shown in Figure 5 and the checkmate is shown in Figure 6. The output of the time and visited states in Figure 7. According to that, the time and number of states are apparently increasing with the depth limit.

I will give another example of the beginning position, in Figure 8.

4.3.3 Force a checkmate

My AI can force a checkmate apparently. The example is shown in Figure 5 and 6.

```
Simple Minimax Max value: 2147483647
Minimax visited states: 34292 time: 1102
making move 7639
3B4/k6R/3Q4/3B4/P2P4/2P3p1/1P1NNP2/R3K3 b Q - 3 33
Game over!!!
White Wins
```

Figure 2: When White wins, the outputs

```
AI try to make move 7347
1k1B4/2Q5/8/3B3p/P2P3R/2P3P1/1P1N1N1P2/R3K1N1 b Q - 1 30
Reach win leaf! value: 2147483647
```

Figure 3: When White needs to act, the outputs

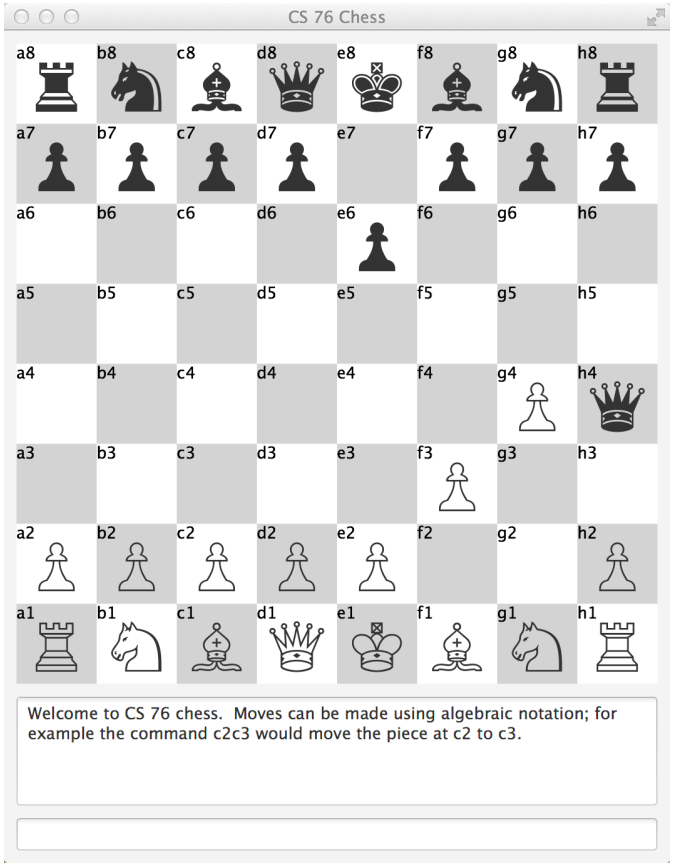

Figure 4: When White loses, the position

5 Evaluation function

An evaluation function returns an estimate of the utility of the game from a given position, like the heuristic functions in A* which returns an estimate of the cost to the goal.

There are some rules of evaluation function:

1. The evaluation function should order the terminal states in the same way as the true utility function;
2. The computation must not take too long;
3. For nonterminal states, the evaluation function should be related with the actual chances of winning strongly.

In my method, I implemented the evaluation function easily, according to the material value described in the textbook.

- If the player has n pawns more than the adversary, $value + 100 * n$;
  If the player has n pawns less than the adversary, $value - 100 * n$;

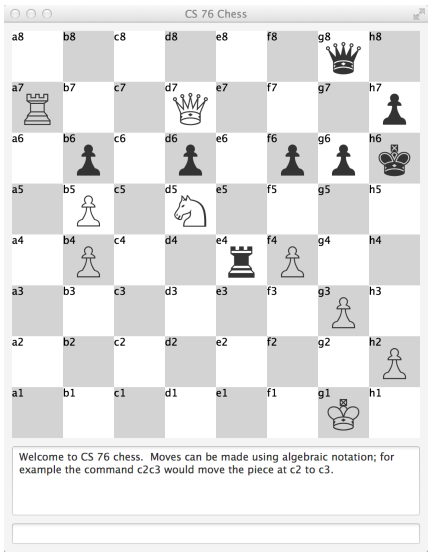

Figure 5: Board before checkmate

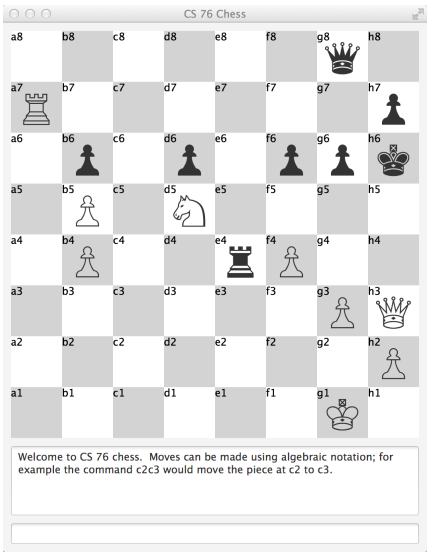

Figure 6: checkmate

```
Minimax depth:2 visited states: 923 time: 68
Minimax depth:4 visited states: 769088 time: 21313
```

Figure 7: Outputs of different depth in a same state, depth:2,4

- If the player has n knights more than the adversary, $value + 300 * n$;
  If the player has n knights less than the adversary, $value - 300 * n$;
- If the player has n bishops more than the adversary, $value + 325 * n$;
  If the player has n bishops less than the adversary, $value - 325 * n$;
- If the player has n rooks more than the adversary, $value + 500 * n$;
  If the player has n rooks less than the adversary, $value - 500 * n$;
- If the player has n queen more than the adversary, $value + 900 * n$;
  If the player has n queen less than the adversary, $value - 900 * n$;

```
1  private int Evaluation(Position position) {
2
3    // if I win, return MAX_VALUE
4    //    I lose, return MIN_VALUE
5    //    draw, return 0
6    //  else return random value
7    int v;
8    if (position.isTerminal()) { // terminal
9      if (position.isMate()) { // checkmate
10       // this turn play is checkmated
```

```
Minimax depth:3 visited states: 9323 time: 337
Minimax depth:5 visited states: 5081536 time: 187678
```

Figure 8: Outputs of different depth in a same state, depth:3,5

```
        if (position.getToPlay() == myTurn)
          v =  Integer.MIN_VALUE;
        else
          v = Integer.MAX_VALUE;
      }
      else
        v = 0;
    }
    else
      v = position.getMaterial();

    return v;
  }
```

5.1 Experiments and Discussions

Obviously, evaluation function deal with the cutoff nodes better than utility function, in which it is just random value. The winning probability of minimax with evaluation is greater than that with utility function. Then, I will show you some examples.

The checkmate state and the value are the same with the checkmate in last section, shown in Figure 6 and 9.

```
Simple Minimax Max value: 2147483647
Minimax depth:3 visited states: 32783 time: 830
```

Figure 9: When White wins, the outputs

The AI reaches the checkmate state, like in Figure 10.

```
AI try to make move 5619
6q1/R6p/1p1p1ppk/1P6/1P3P2/2N1r1PQ/7P/6K1 b - - 3 2
value 2147483647
```

Figure 10: When White needs to act, the outputs

6 Alpha-Beta Pruning

The minimax search has exponential game states to examine in the depth of the tree. To deal with this, there is a idea of prunning to cut some nodes in the tree. Here, I will introduce a alpha-beta pruning.

Alpha-beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves. Considering a node n in the tree, a player has a choice of moving to that node. If the player has a better choice m either at the parent node of n or at any choice point further up, the n will never be reached in actual play. So we can find out these nodes and prune them.

The basic idea is to keep two number:

- α A number keeps the value of the best(highest value) choice we have found so far at any choice point along the path for MAX.
- β Another number keeps the value of the best(lowest value) choice we have found so far at any choice point along the path for MIN.

The algorithm updates the values as it goes along and prunes the branches with greater value than β in MAX and branches with less value than α in MIN.

6.1 Implementation of Alpha-Beta Pruning

The algorithm is implemented in `MinimaxAI.java`. Here is the code:

```
private short AlphaBetaPruning(Position position) {

  int v = Integer.MIN_VALUE; // keep max value
  short action = 0;   //keep the action leading to max value

  short [] moves = position.getAllMoves();
  for (short move: moves) {
    doMove(position, move);
    int tmp = MIN_Value_ABP(position, Integer.MIN_VALUE, Integer.MAX_VALUE, 1);
    if (tmp > v) {
      v = tmp;
      action = move;
    }

    undoMove(position);
  }

  System.out.println("Alpha-Beta Pruning Max value: " + v);
  return action;
}

private int MAX_Value_ABP(Position position, int a, int b, int depth) {

  if (Cutoff_Test(position, depth))
    return Evaluation(position);

  int v = Integer.MIN_VALUE;
  for (short move: position.getAllMoves()) {
    doMove(position, move);
    v = Math.max(v, MIN_Value_ABP(position, a, b, depth+1));
    undoMove(position);
    if (v >= b)
      return v;
    a = Math.max(a, v);
  }

  return v;
}

private int MIN_Value_ABP(Position position, int a, int b, int depth) {

  if (Cutoff_Test(position, depth))
    return Evaluation(position);

  int v = Integer.MAX_VALUE;
  for (short move: position.getAllMoves()) {
    doMove(position, move);
    v = Math.min(v, MAX_Value_ABP(position, a, b, depth+1));
    undoMove(position);
    if (v <= a)
      return v;
    b = Math.min(b, v);
```

```
    }

    return v;
  }
```

Here I implemented a minimax search with alpha-beta pruning. Just like the pseudo-code for alpha-beta pruning in the textbook. The algorithm updates the values as it goes along and prunes the branches with greater value than β in MAX and branches with less value than α in MIN.

6.2 improvement

I found a problem that before any pieces are taken, the value is always 0. Then the algorithm always choose the first move it comes into, which results that the AI always do repeated actions. To deal with this, I did some improvement. For the actions with same values, I choose an action randomly. Ultimately, the alpha-beta pruning is like that:

```
  private short AlphaBetaPruning(Position position) {

    int v = Integer.MIN_VALUE; // keep max value
    short action = 0;   //keep the action leading to max value

    short [] moves = position.getAllMoves();
    for (short move: moves) {
      doMove(position, move);
      int tmp = MIN_Value_ABP(position, Integer.MIN_VALUE, Integer.MAX_VALUE, 1);
      if (tmp > v) {
        v = tmp;
        action = move;
      }
      // randomly choose one from equal value, to alleviate repeating
      else if (tmp == v) {
        int r = new Random().nextInt(100);
        if (r < 50) {
          v = tmp;
          action = move;
        }
      }

      undoMove(position);
    }

    System.out.println("Alpha-Beta Pruning Max value: " + v);
    return action;
  }
```

6.3 Experiments and Discussions

6.3.1 Correctness

Withut the random part, the choices will be same from both usual minimax in last section and alpha-beta pruning. There are some outputs can prove this, in Figure 11, 12, 13.

```
Alpha-Beta Pruning Max value: 2147483647
Minimax 5619 visited states: 32783 time 965 ABP 5619 visited states: 3300 time 87
```

Figure 11: Output shows both choose the same action

```
Alpha-Beta Pruning Max value: 0
Minimax 5121 visited states: 9323 time 315 ABP 5121 visited states: 1246 time 69
```

Figure 12: Output shows both choose the same action

6.3.2 Simple minimax versus Alpha-Beta Pruning

The experimental data is also shown in Figure 11, 12 and 13.

From these data, apparently, alpha-beta pruning has much less time and space consuming than usual minimax in last section. In many cases, more than half states are cut off!

7 Transposition table

In many games, repeated states occur frequently because of different ordering of same actions. So it is worthwhile to store the evaluation of the resulting position in a hash table the first time it si came into so that we can decrease computation consuming. It is like the explored list in A* in last section.

7.1 Implementation of Transposition table

I added transposition table to alpha-beta pruning. Here is the code:

```
1  private short AlphaBetaPruning_Trans(Position position) {
2    updateExplored();   // record visited states
3
4    int v = Integer.MIN_VALUE; // keep max value
5    short action = 0;   //keep the action leading to max value
6
7    short [] moves = position.getAllMoves();
8    for (short move: moves) {
9      doMove(position, move);
10     int tmp = MIN_Value_ABP_Trans(position, Integer.MIN_VALUE, Integer.MAX_VALUE, 1);
11     if (tmp > v) {
12       v = tmp;
13       action = move;
14     }
15     // randomly choose one from equal value, to alleviate repeating
16     else if (tmp == v) {
17       int r = new Random().nextInt(100);
18       if (r < 50) {
19         v = tmp;
20         action = move;
```

```
Alpha-Beta Pruning Max value: 0
Minimax 4176 visited states: 16479 time 641 ABP 4176 visited states: 1849 time 52
```

Figure 13: Output shows both choose the same action

```
        }
      }

      undoMove(position);
    }
    if (!transpositionTable.containsKey(position))
      transpositionTable.put(position, v);
    System.out.println("Alpha-Beta Pruning Max value: " + v);
    return action;
  }

  private int MAX_Value_ABP_Trans(Position position, int a, int b, int depth) {
    if (transpositionTable.containsKey(position))
      return transpositionTable.get(position);

    updateExplored();   // record visited states

    if (Cutoff_Test(position, depth))
      return Evaluation(position);

    int v = Integer.MIN_VALUE;
    for (short move: position.getAllMoves()) {
      doMove(position, move);
      v = Math.max(v, MIN_Value_ABP_Trans(position, a, b, depth+1));
      undoMove(position);
      if (v >= b)
        return v;
      a = Math.max(a, v);
    }

    transpositionTable.put(position, v);
    return v;
  }

  private int MIN_Value_ABP_Trans(Position position, int a, int b, int depth) {
    if (transpositionTable.containsKey(position))
      return transpositionTable.get(position);

    updateExplored();   // record visited states

    if (Cutoff_Test(position, depth))
      return Evaluation(position);

    int v = Integer.MAX_VALUE;
    for (short move: position.getAllMoves()) {
      doMove(position, move);
      v = Math.min(v, MAX_Value_ABP_Trans(position, a, b, depth+1));
      undoMove(position);
      if (v <= a)
        return v;
      b = Math.min(b, v);
    }

    transpositionTable.put(position, v);
    return v;
  }
```

In my method, before searching the value of the state, look up the transposition table first. If there is the same position, I can get the value without searching the nodes and their successors, just like the nodes are cut off. Because minimax search is a breath-first search, the position you find have been in transposition table must have a depth no more than the current depth. Obviouly, use the value of a position with less depth, due to it have searched deeper.

There is another change of `evaluation function`.

```
1  private int Evaluation(Position position) {
2
3    // if I win, return MAX_VALUE
4    //    I lose, return MIN_VALUE
5    //    draw, return 0
6    //  else return random value
7    int v;
8    if (position.isTerminal()) { // terminal
9      if (position.isMate()) { // checkmate
10       // this turn play is checkmated
11       if (position.getToPlay() == myTurn)
12         v =  Integer.MIN_VALUE;
13       else
14         v = Integer.MAX_VALUE;
15     }
16     else
17       v = 0;
18   }
19   else
20     v = position.getMaterial();
21   System.out.println("value "+ v);
22   transpositionTable.put(position, v);
23   return v;
24 }
```

Because `MIN_Value_ABP_Trans` and `MAX_Value_ABP_Trans` have looked up the transposition table before `Cutoff_Test`, there is no need to look up transposition table here. Just put it in, after getting the value.

7.2 Experiments and Discussions

The experimental data is shown in Figure 14, 15 and 16.

```
Alpha-Beta Pruning Max value: 2147483647
ABP 5619 visited states: 3300 time: 153 ABP with transpositionTable 5619 visited states: 922 time: 72
```

Figure 14: Experimental data for abp with and without transpositionTable

```
Alpha-Beta Pruning Max value: 0
ABP 5583 visited states: 1246 time: 86 ABP with transpositionTable 6030 visited states: 421 time: 36
```

Figure 15: Experimental data for abp with and without transpositionTable

From these data, apparently, alpha-beta pruning has much less time and space consuming than usual minimax in last section. In many cases, more than half states are cut off!

```
Alpha-Beta Pruning Max value: 0
ABP 5258 visited states: 1949 time: 39 ABP with transpositionTable 6030 visited states: 740 time: 37
```

Figure 16: Experimental data for abp with and without transpositionTable

8 Move reordering

By remember best moves in each depth, when the algorithm comes into this depth, if there is a best move have been stored, search this move first. Because this move always can help do more alph-beta pruning, cut more branches.

8.1 Implementation of Move reordering

First, add a private component in the class `bestMoves`. It is an array of short type to store the best move of each depth.

```
1  private short iterative_reorder(Position position) {
2    int d = depth_limit;
3    for (depth_limit = 1; depth_limit < d; depth_limit++) {
4      bestMoves[1] = Minimax(position);
5    }
6
7    depth_limit = d;
8    return bestMoves[1];
9  }
10
11 private short iterative_recursive(Position position) {
12   updateExplored();   // record visited states
13
14   int v = Integer.MIN_VALUE; // keep max value
15   short action = 0;   //keep the action leading to max value
16
17   int a = Integer.MIN_VALUE;
18   int b = Integer.MAX_VALUE;
19   if (bestMoves[1] != 0) {
20     short move = bestMoves[1];
21     doMove(position, move);
22     int tmp = MIN_Value_ABP_Trans(position, a, b, 1);
23     if (tmp > v) {
24       v = tmp;
25       action = move;
26     }
27     undoMove(position);
28   }
29
30   short [] moves = position.getAllMoves();
31   for (short move: moves) {
32     if (move == bestMoves[1])
33       continue;
34     doMove(position, move);
35     int tmp = MIN_Value_ABP_Trans(position, a, b, 1);
36     if (tmp > v) {
37       v = tmp;
38       action = move;
39     }
```

```
40        // randomly choose one from equal value, to alleviate repeating
41        else if (tmp == v) {
42          int r = new Random().nextInt(100);
43          if (r < 50) {
44            v = tmp;
45            action = move;
46          }
47        }
48
49        undoMove(position);
50      }
51      if (!transpositionTable.containsKey(position))
52        transpositionTable.put(position, v);
53      System.out.println("Alpha-Beta Pruning Max value: " + v);
54      return action;
55    }
56
57    private int MAX_Value_ABP_Trans(Position position, int a, int b, int depth) {
58      if (transpositionTable.containsKey(position))
59        return transpositionTable.get(position);
60
61      updateExplored();   // record visited states
62
63      if (Cutoff_Test(position, depth))
64        return Evaluation(position);
65
66       int v = Integer.MIN_VALUE;
67      if (bestMoves[depth+1] != 0) {
68        short move = bestMoves[depth+1];
69        doMove(position, move);
70         v = Math.max(v, MIN_Value_ABP_Trans(position, a, b, depth+1));
71        undoMove(position);
72        if (v >= b)
73          return v;
74        a = Math.max(a, v);
75      }
76      for (short move: position.getAllMoves()) {
77        if (move == bestMoves[depth+1])
78          continue;
79        doMove(position, move);
80        v = Math.max(v, MIN_Value_ABP_Trans(position, a, b, depth+1));
81        undoMove(position);
82        if (v >= b)
83          return v;
84        a = Math.max(a, v);
85      }
86
87      transpositionTable.put(position, v);
88      return v;
89    }
90
91    private int MIN_Value_ABP_Trans(Position position, int a, int b, int depth) {
92      if (transpositionTable.containsKey(position))
93        return transpositionTable.get(position);
94
95      updateExplored();   // record visited states
```

```
96
97      if (Cutoff_Test(position, depth))
98        return Evaluation(position);
99
100     int v = Integer.MAX_VALUE;
101     if (bestMoves[depth+1] != 0) {
102       if (move == bestMoves[depth+1])
103         continue;
104       short move = bestMoves[depth+1];
105       doMove(position, move);
106       v = Math.min(v, MAX_Value_ABP_Trans(position, a, b, depth+1));
107       undoMove(position);
108       if (v <= a)
109         return v;
110       b = Math.min(b, v);
111     }
112     for (short move: position.getAllMoves()) {
113       doMove(position, move);
114       v = Math.min(v, MAX_Value_ABP_Trans(position, a, b, depth+1));
115       undoMove(position);
116       if (v <= a)
117         return v;
118       b = Math.min(b, v);
119     }
120
121     transpositionTable.put(position, v);
122     return v;
123   }
```

8.2 Experiments and discussions

The experimental data is shown in Figure ??, ??.

```
Alpha-Beta Pruning Max value: 0
iterative 6030 visited states: 422 time: 46 iterative reorder 5583 visited states: 2 time: 2
```

Figure 17: Experimental data for abp with and without move reordering

```
Alpha-Beta Pruning Max value: 0
iterative 5900 visited states: 442 time: 31 iterative reorder 6095 visited states: 2 time: 2
```

Figure 18: Experimental data for abp with and without move reordering

From these data, apparently, iterative deepening with move reordering has much less time and space consuming than algorithm without it in last section. In many cases, more than half states are cut off!

References

[1] Zermelo, Ernst (1913), Uber eine Anwendung der Mengenlehre auf die Theorie des Schachspiels, Proceedings of the Fifth International Congress of Mathematicians 2, 5014. Cited from Eichhorn, Christoph: Der Beginn der Formalen Spieltheorie: Zermelo (1913), Uni-Muenchen.de. Retrieved 2007-03-23.

[2] Mitja Lutrek, Matja Gams, Ivan Bratko (2005). Why Minimax Works: An Alternative Explanation. IJCAI 2005

[3] Abdallah Saffidine, Hilmar Finnsson, Michael Buro (2012). Alpha-Beta Pruning for Games with Simultaneous Moves. AAAI 2012.