Chess Game Solution

Boying Shi

February 19, 2014

1 Introduction

In this assignment we are presented with a game tree searching problem and particular to deal with chess game problem. To deal with this problem, there are several search algorithm implemented including the ***Minimax Search Algorithm***, ***Alpha-Beta Search Algorithm*** and ***Alpha Beta Search Algorithm with Transposition Table***.

The basic search algorithm is Minimax Search, which simply enumerates each possible states for each depth for either player. To improve by reducing the number of nodes that are searched, Alpha-Beta Search Algorithm has been raised based on Minimax Search. Then, more improved searching algorithms were proposed bash on Alpha-Beta Search like the one adding a transposition table to it.

Since I have done all the bonus works, I also implemented more improved algorithms for Alpha-Beta Search like ***Move Reordering Search***, ***Null-move Heuristic Search*** and ***improved cut-off Test***. Also, for a bonus part, I implemented to import opening book to my project and give a better begin for each test and it is more interesting. And I use profiling to record for my implemented algorithms and give some analysis for them about running time and space. Finally, I read a paper related to Alpha-Beta Search and give a short review of it.

Check for a win condition

Since the provided code has not give a win condition, I personally add it to help me clarify white player wins or black player wins. My code for it is as followed and I add it at `ChessGame.java` doMove function:

Listing 1: Win condition

```
//Check Win Condtion
  if(position.isMate()==true&&position.getToPlay()==1){
    System.out.println("White wins");
    return;
    }else if(position.isMate()==true&&position.getToPlay()==0){
      System.out.println("Black wins");
      return;
      }else if(position.isTerminal()==true){
       System.out.println("time out");
       return;
       }

//Check Draw Condition
  if(position.isStaleMate()==true){
  System.out.println("Draw");
  return;
  }
```

Also, in order to get a clear right result, I let the `bestmove` to be equal to -1000 when there there is no legal moves for the loser. Then, when the handle receive a move equals to -1000. Then it will return and stop here. The code is as followed:

Listing 2: Bestmove value for win condition

```
short[] moves = position.getAllMoves();
//if there is no move for a player then return -1000
  if(moves.length==0){
    bestmove = -1000;
    }
```

2 Minimax Search

A minimax algorithm is a recursive algorithm for choosing the next move in an n-player game, usually a two-player game. A value is associated with each position or state of the game. This value is computed by means of a position evaluation function and it indicates how good it would be for a player to reach that position. The player then makes the move that maximizes the minimum value of the position resulting from the opponent's possible following moves. If it is A's turn to move, A gives a value to each of his legal moves[1].

For the simplest Minimax Search Algorithm, the detailed steps are as followed: From the beginning position, it will calculate all the possible moves that could be made from that position, and in a standard chess game there are 20 possible successors for the beginning position. Then, it will expand each of these new positions so that they will include the possible moves for the other player. Continually, this process will end until reach a winning position for the required player is found[2]. So, my first task is to implement the winning position test called *cutoffTest*. My implementation of it is shown below:

Listing 3: cutoffTest

```
public boolean cutoffTest(Position p,int depth){
  /* A winning state or a draw state or reach max depth or is timeout
   * will return true, otherwise it will return false*/
  if(p.isMate()||p.isStaleMate()||depth == 0||p.isTerminal()){
      return true;
    }else{
     return false;
     }
  }
```

Thanks to Bernhard Seybold, I can use the Chesspresso to help me build the basic Chess Game structure. And then I can use functions that are written already in the provided chess library. Here in my cutoffTest code, I use `isMate()` to check if the game hass already reached a win state. And `isStaleMate()` to help me check if the game has reached a draw state. Also use the `isTerminal()` function to check if it is already time out. Moreover, since the algorithm should be depth-limited minimax, and then to check if we have reach the specified maximum depth. Then, the cutoffTest can help me to check if it can stop a particular search.

Little Discussion

However, even I just put a cutoffTest to help stop searching for each depth, the algorithm seems can get some results of `bestmove` at each depth. However, the cost of performing that calculation is enormous[2]. Therefore, I need to implement utility function to take care of what we return when we reach a terminal states.

2.1 With Utility Function

As is mentioned in assignment three[3], the utility function is implemented as followed: large positive utility for a win for MAX, such as the maximum integer value for a Java int, its negative for a win for MIN, and zero for a draw. For other states, return a random value r, where r is smaller than the value of MAX and larger than the value for MIN.

My code of utility function is as below:

Listing 4: Utility Function

```
public int utility(Position p,int player){
  if(p.isMate() == true&&player==0){
    return Integer.MIN_VALUE;
    }else if(p.isMate() == true&&player==1){
      return Integer.MAX_VALUE;
      }else if(p.isStaleMate()==true){
        return 0;
        }else{
          return (new Random().nextInt(1000));
         }
 }
```

The code is very simply and straight forward regarding to the description in the assignment. The only thing that needs to be consider is how to choose a random value, since it returns a score for a state that has not reach the cut off test. I give a random value among 1000.

2.1.1 Implementation

With cutoff test and utility function implemented, I can now simply implement the basic Minimax Search, and my code is here (Since the following works are all based on this search, I put the entire class (except `cutoffTest` and `utility`) of it to give a detailed look for this implementation and after that I will just post the changed part):

Listing 5: Minimax Search

```
public class MiniMaX implements ChessAI{
  int maxdepth;
  short bestmove;//At each depth, the best move might be saved in an instance
      variable
  long states=0;//record the number of states that have been searched
  int tempdepth=0;//record the current maximum depth used
  int bestevluationscore = 0;//record the best utility value

  public MiniMaX(int maxdepth){
    this.maxdepth = maxdepth;
    }

  //calculate the best move
  public void minimax_decision(Position position) throws IllegalMoveException{
    //iterative deepening search by adding for loop of j
    for (int j = 1; j <= maxdepth; j++){
      states = 0;
      short[] moves = position.getAllMoves();
      if(moves.length==0){
        bestmove = -1000;//a win condition value for best move
```

```
        }else{
          bestmove = moves[0];//give best move a initial value
          int[] score = new int[moves.length];
          states = states+moves.length;//record states
          int max = -10000;
          short pos = 0;
          for (int i=0;i<moves.length;i++){
            position.doMove(moves[i]);
            score[i] = minimax(position,j);
            if(score[i]>=max){
              max = score[i];
              pos= moves[i];
              bestmove = pos;
              bestevluationscore = max;
              }
              position.undoMove();
              }
            }

       if(bestevluationscore == Integer.MAX_VALUE){
         tempdepth =j;
         return;
           }
         }

   tempdepth = this.maxdepth;
    }

//starter
 public int minimax(Position p, int depth){
   return min_value(p,depth);
   }

//calculate max value
 public int max_value(Position p, int depth){
   int best = Integer.MIN_VALUE;//initial best value
   if(this.cutoffTest(p, depth)==true){
     return utility(p,0);
     }

    short [] moves = p.getAllMoves();
    states = states+moves.length;//record states
    if(moves!=null){
      for(short move: moves){
        p.doMove(move);
        int val = min_value(p,depth-1);
        if(val>best){
          best = val;
          }
        p.undoMove();
        }
```

```
      }
       return best;
   }

//calculate max value
  public int min_value(Position p, int depth){
    int best = Integer.MAX_VALUE;//initial best value
    if(this.cutoffTest(p, depth)==true){
      return utility(p,1);
      }

    short [] moves = p.getAllMoves();
    states = states+moves.length;//record states

    if(moves!=null){
      for(short move: moves){
        p.doMove(move);
        int val = max_value(p,depth-1);
        if(val<best){
          best = val;
          }
        p.undoMove();
        }
      }
      return best;
  }

 @Override
  public short getMove(Position position) {
    minimax_decision(position);

    if(bestmove == -1000){
      return bestmove;
      }else{
        System.out.println("Maximum depth: "+tempdepth);
        System.out.println("Number of states has been visited: "+states);
        states=0;
        tempdepth=0;
        return bestmove;
              }
        }
  }
```

2.1.2 Test Result

Now, in my project I have three method to play chess: human player(given), random AI player(given) and my MiniMax with basic utility function player. Since I have already implemented Opening Book, I can use those beginnings for my test.

The test shown in figure1 is a maximum depth = 2 and the black player is using MiniMax and the white player is using random AI.

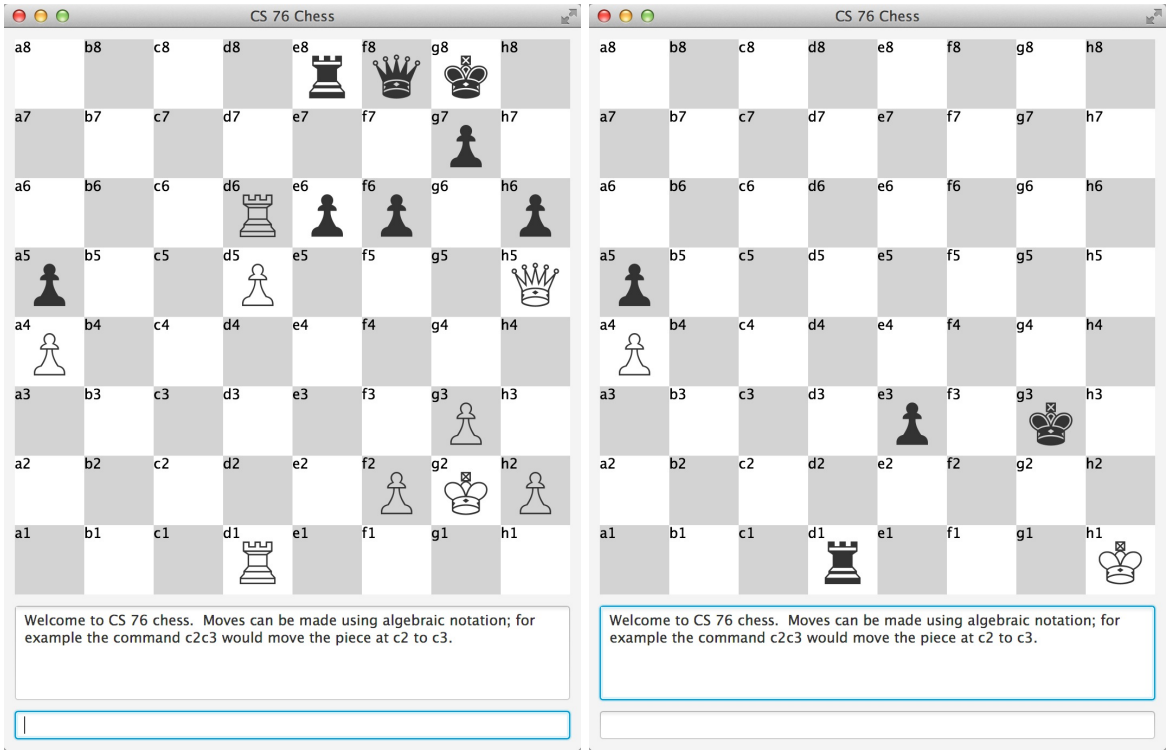


Figure 1: Results of maxdepth=2 and black is MiniMax player(Left beginning, Right win)

Listing 6: Data for this move

```
Maximum depth: 2
Maximum number of states has been visited: 28985
Minimum number of states has been visited: 432
Black player wins with 13 turns
Opening Book: Random
```

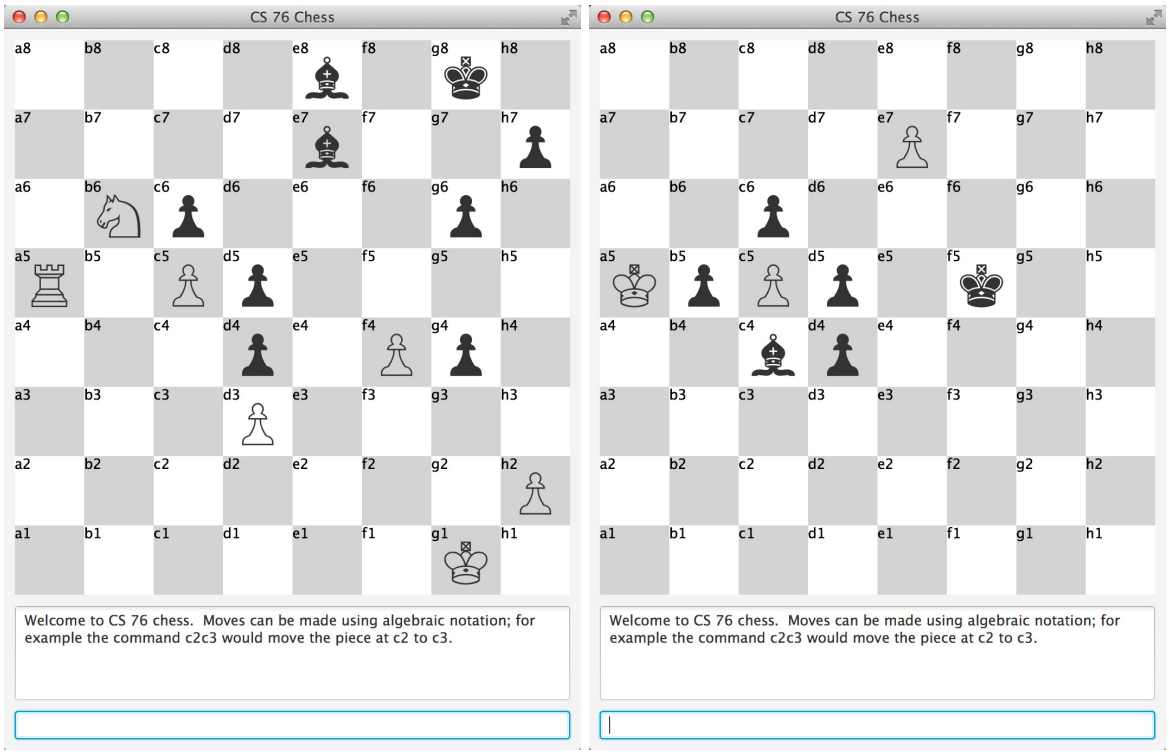


Figure 2: Results of maxdepth=3 and black is MiniMax player(Left beginning, Right win)

Listing 7: Data for this move

```
Maximum depth: 3
Maximum number of states has been visited: 144033
Minimum number of states has been visited: 3711
Black player wins with 20 turns
Opening Book: 23
```

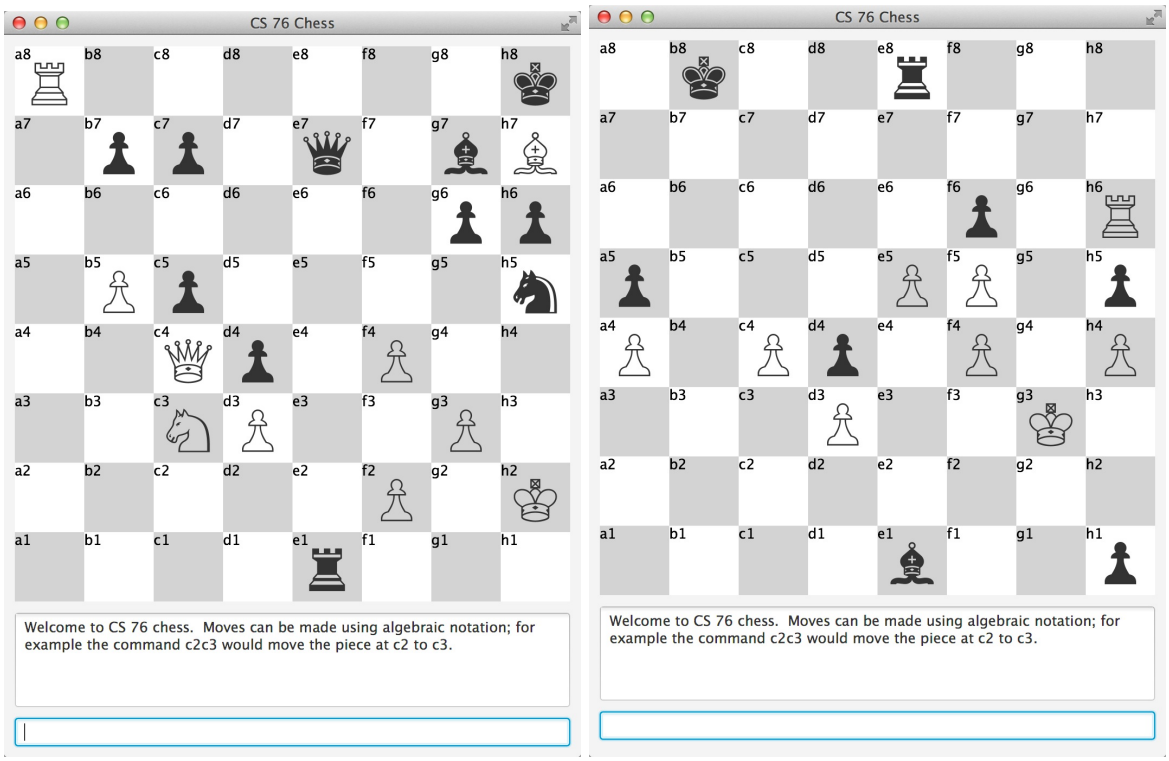


Figure 3: Results of maxdepth=3 and black is MiniMax player(Left beginning, Right win)

Listing 8: Data for this move

```
Maximum depth: 3
Maximum number of states has been visited: 1035029
Minimum number of states has been visited:  5096
Black player wins with 28 turns
Opening Book: 31
```

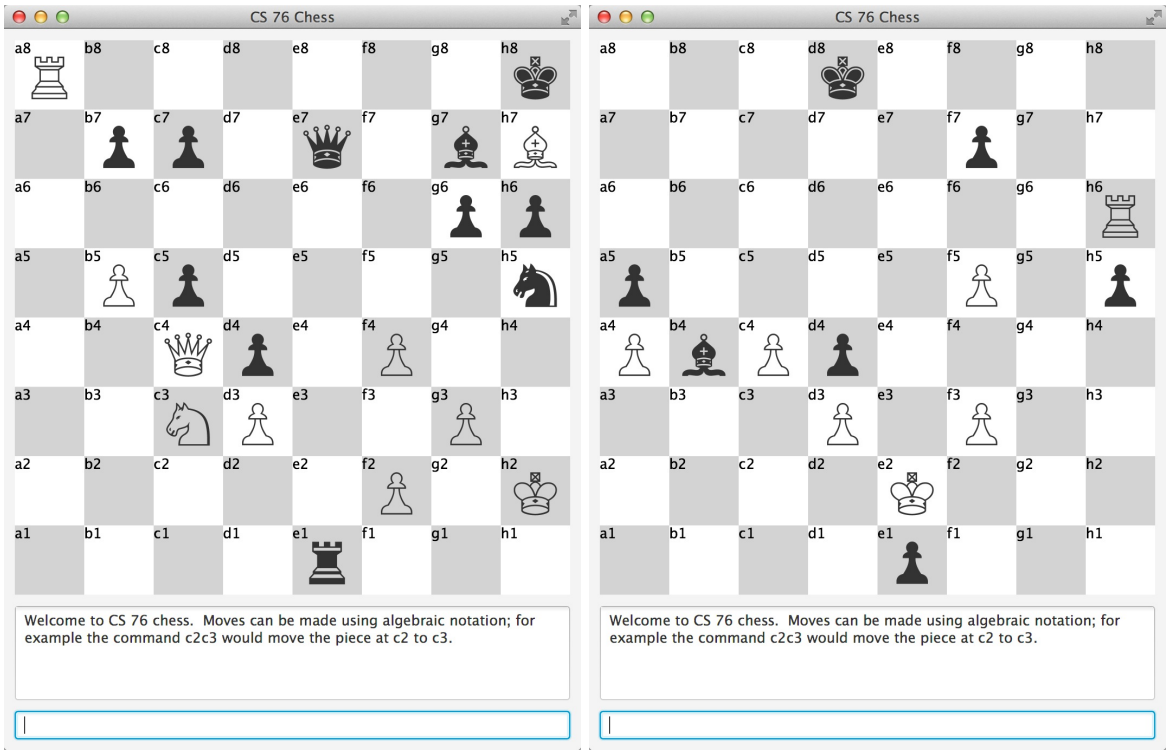


Figure 4: Results of maxdepth=2 and black is MiniMax player(Left beginning, Right win)

Listing 9: Data for this move

```
Maximum depth: 4
Maximum number of states has been visited: 5451532
Minimum number of states has been visited:  4326
Black player wins with 6 turns
Opening Book: 31
```

From figure 1, figure 2 and figure 3, I test these three cases with the same maximum depth but different beginnings and it shows that for a maximum depth of 2 that these Minimax AI can always win with a Random AI and from figure 3 and figure 4, they use the same beginning, the 31th in opening book, that if I set a bigger maximum depth then the winning turns will reduce from 9 to 6 but the maximum number of states thad have been visited increased a lot from 14377 to 5451532.

Therefore, from my test cases, I can see that using such a simply utility function can already help me to make it win any random AI. But the main problem has already been exposed, that is, it searches too many states and it need to be improved. Then, when the maximum depth increase, the number of states will increase exponentially and result in a long running time.

To improve the algorithm, firstly, I will improve the utility function. (A random score in the previous utility function is just like a random AI).

2.2 With Evaluation Function

2.2.1 Implementation

To improve the evaluation function, I need to modify the part for giving the score when it has no reach terminal state. The basic evaluation function is to calculate the material scores. To sum of various factors that are thought to influence the value of position. And for each kind of chess piece, it gives a weight for different value of a same kind of chess piece. The given code has already implement a evaluation function call `getMaterial()` in Position class. It has the same idea of what I have explained former for material scores.

However, only use a material score is not enough and it will result in a stupid moving problem, that is, when the number of each kind of chess piece on the board for either player is the same, then, the material score will always be 0 (white king = black king, white queen = black queen, white rook = black rook, white bishop = black bishop, white knights = black knights, white pawns = black pawns). And the stupid moving will be it will move one by one in order until there is some piece being killed and therefore the number is unbalanced. So, only one evaluation factor is not enough.

And in Position class, there is another evaluation function called `getDomination()`. It calculates the position on the board that give weight value for each board which the center is higher than the margin. Then, I add both getMaterial() and getDomination() together will give a better reasonable evaluation function. And the new evaluation function is shown below:

Listing 10: Evaluation Function

```
public int utility(Position p,int player){
  if(p.isMate() == true&&player==maxplayer){
    return Integer.MIN_VALUE;
    }else if(p.isMate() == true&&player==minplayer){
    return Integer.MAX_VALUE;
    }else if(p.isStaleMate()==true){
    return 0;
    }else{
      //using new evaluation function rather than random
      evluationscore =  (int) evaluate(p,player);
      return (int) evaluate(p,player);
              }
      }

public double evaluate(Position position,int player){
  //calculate the sum of material score and domination score
  double value = position.getMaterial()+position.getDomination();
  if(player == minplayer){
    return -value;
    }else{
    return value;
      }
   }
```

2.2.2 Test Result

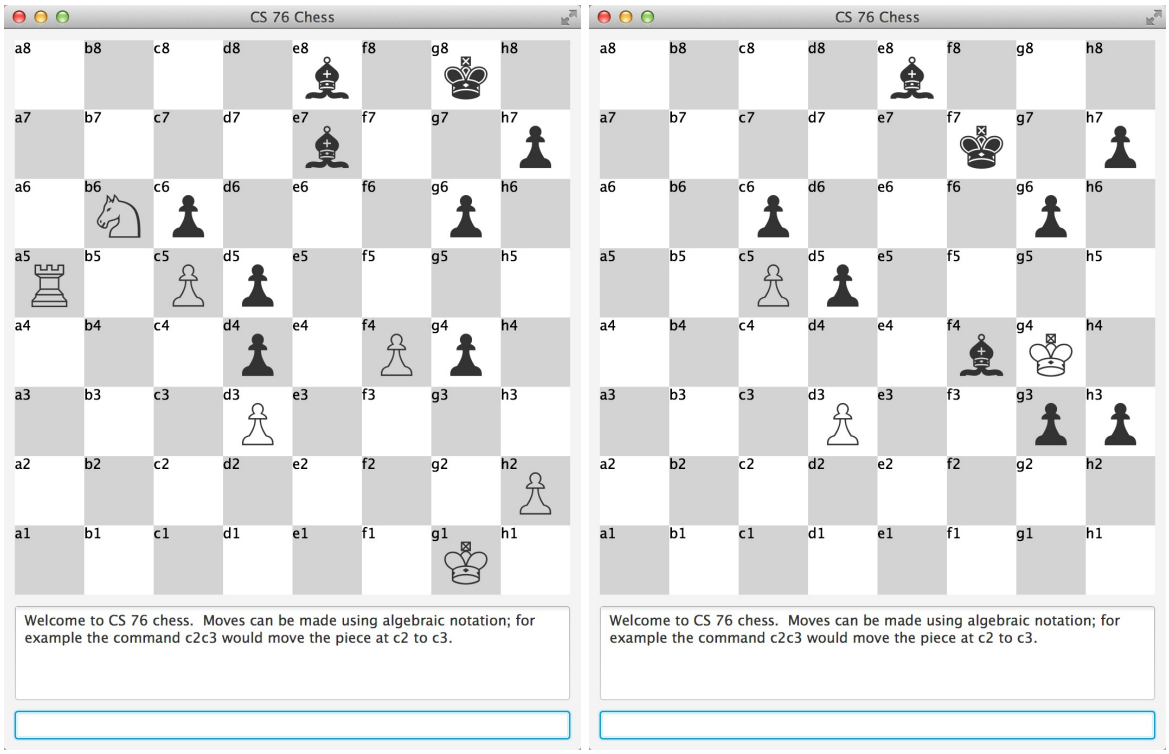


Figure 5: Results of maxdepth=3 and black is EvluationMiniMax player(Left beginning, Right win)

Listing 11: Data for this move

```
Maximum depth: 3
Evluation score: 174
Maximum number of states has been visited: 59761
Minimum number of states has been visited: 1211
Black player wins with 9 turns
Opening Book: 23
```

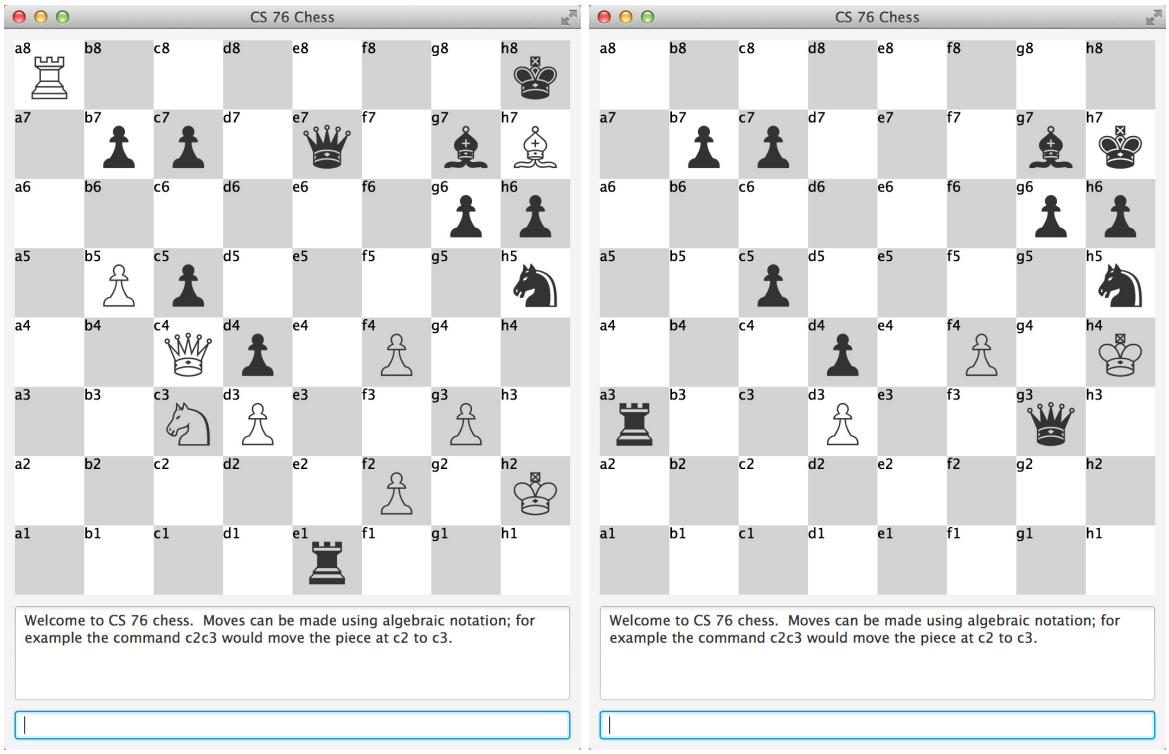


Figure 6: Results of maxdepth=3 and black is EvluationMiniMax player(Left beginning, Right win)

Listing 12: Data for this move

```
Maximum depth: 3
Evluation score: 209
Maximum number of states has been visited: 1864942
Minimum number of states has been visited:  2404
Black player wins with 6 turns
Opening Book: 31
```

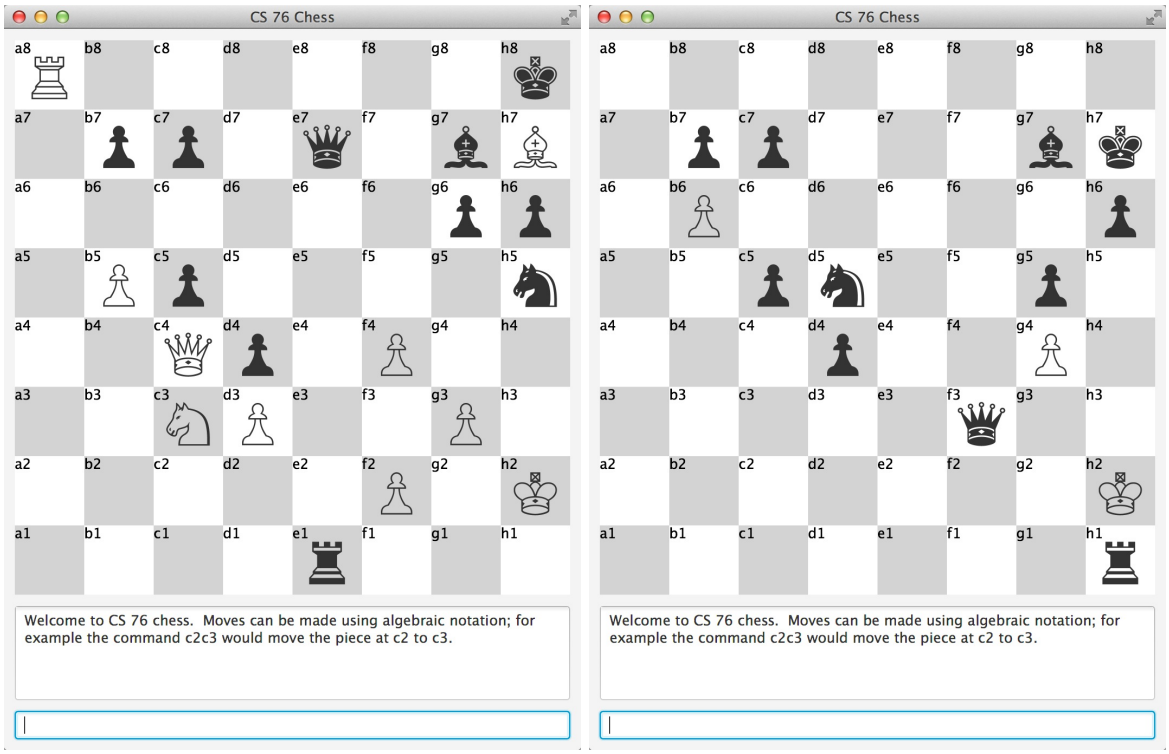


Figure 7: Results of maxdepth=4 and black is EvluationMiniMax player(Left beginning, Right win)

Listing 13: Data for this move

```
Maximum depth: 4
Evluation score: 2404
Maximum number of states has been visited: 54679191
Minimum number of states has been visited:  4934
Black player wins with 6 turns
Opening Book: 31
```

From the test results shown above, the figure 5 and figure 6 using the same maximum depth and the same beginnings with which I have tested in the figure 2 and figure 3. Clearly, although they use the same MiniMax algorithm, the different utility functions have different results. And from the result that the evaluation MiniMax search results in quicker win, it means that by using the evaluation function, the AI player performs much smarter than previous.

Now the algorithm did improved somehow, but there is still a problem, that is the running time for a bigger depth. From figure 7, the maximum states has been visited become 54679191 from 1864942. It is really a big number and takes some times. And as is known that the algorithm did visited some states that are useless which need to be pruned. Then, there comes out the Alpha-Beta pruning search.

3 Alpha-Beta Pruning Search

Alpha?beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Go, etc.). It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision[4].

This new algorithm just helps to solve the problem of visiting too many useless states and then can lead to a quicker running time but the IQ of this algorithm is still the same with Evaluation MiniMax Search since they use the same evaluation function.

3.1 Implementation

My implementation of this search is shown below:

Listing 14: Alpha-Beta Pruning Search

```
public int max_value(int alpha, int beta, Position p, int depth){
  if(this.cutoffTest(p, depth)==true){
    return utility(p,maxplayer);
    }

    int v = alpha;
    short [] moves = p.getAllMoves();
    states = states+moves.length;//record visited states

    if(moves!=null){
      for(short move: moves){
        p.doMove(move);
        int score = min_value(v,beta,p,depth-1);

        if(score >v){
          v = score;
          }

        //beta is upper bound
        if(v >= beta){
          p.undoMove();
          return beta;
          }

        p.undoMove();
           }
         }

          return v;
       }



 public int min_value(int alpha, int beta, Position p, int depth){
   if(this.cutoffTest(p, depth)==true){
```

```
    return utility(p,minplayer);
    }

    int v = beta;
    short [] moves = p.getAllMoves();
    states = states+moves.length;//record visited states

    if(moves!=null){
      for(short move: moves){
        p.doMove(move);

        int score = max_value(alpha, v, p,depth-1);

        if(score< v){
          v = score;
          }

        //alpha is upper bound
        if(v <= alpha){
          p.undoMove();
          return alpha;
          }

          p.undoMove();
          }
       }
        return v;
   }
```

3.2 Test Result

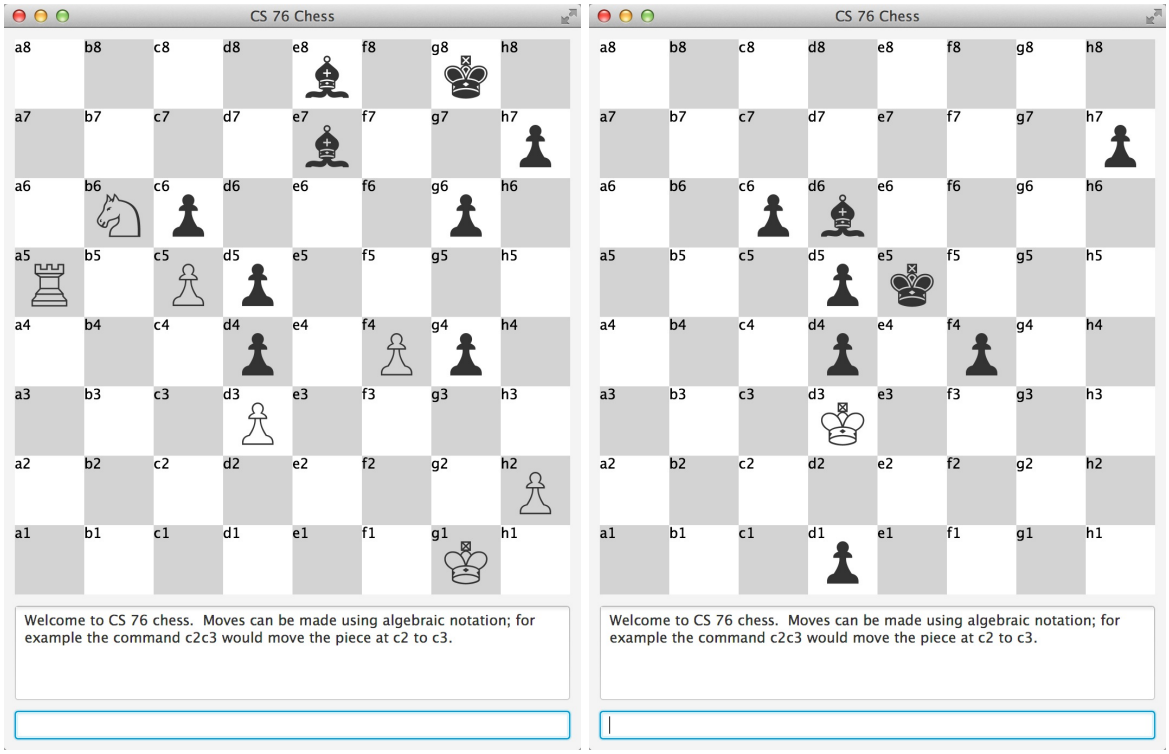


Figure 8: Results of maxdepth=3 and black is Alpha-Beta player(Left beginning, Right win)

Listing 15: Data for this move

```
Maximum depth: 3
Evluation score of first step: 174
Maximum number of states has been visited: 16433
Minimum number of states has been visited: 98
Black player wins with 9 turns
Opening Book: 23
```

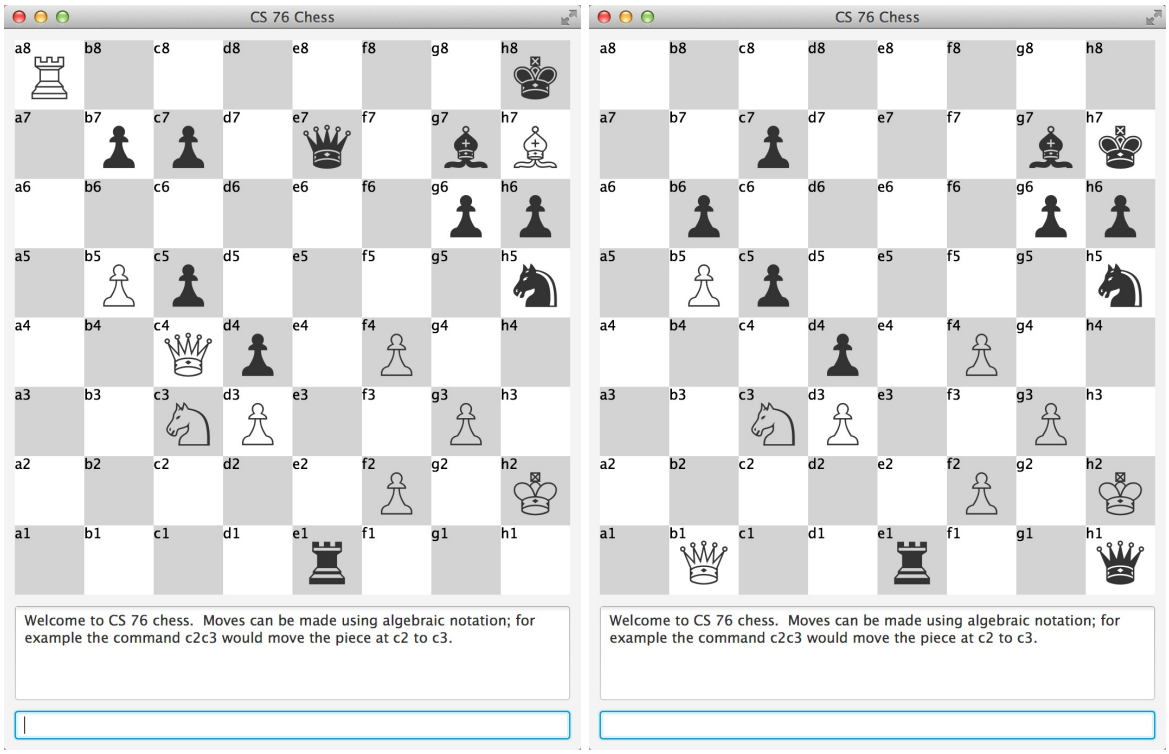


Figure 9: Results of maxdepth=3 and black is Alpha-Beta player(Left beginning, Right win)

Listing 16: Data for this move

```
Maximum depth: 3
Evluation score of first step: 209
Maximum number of states has been visited: 467458
Minimum number of states has been visited:  924
Black player wins with 6 turns
Opening Book: 31
```

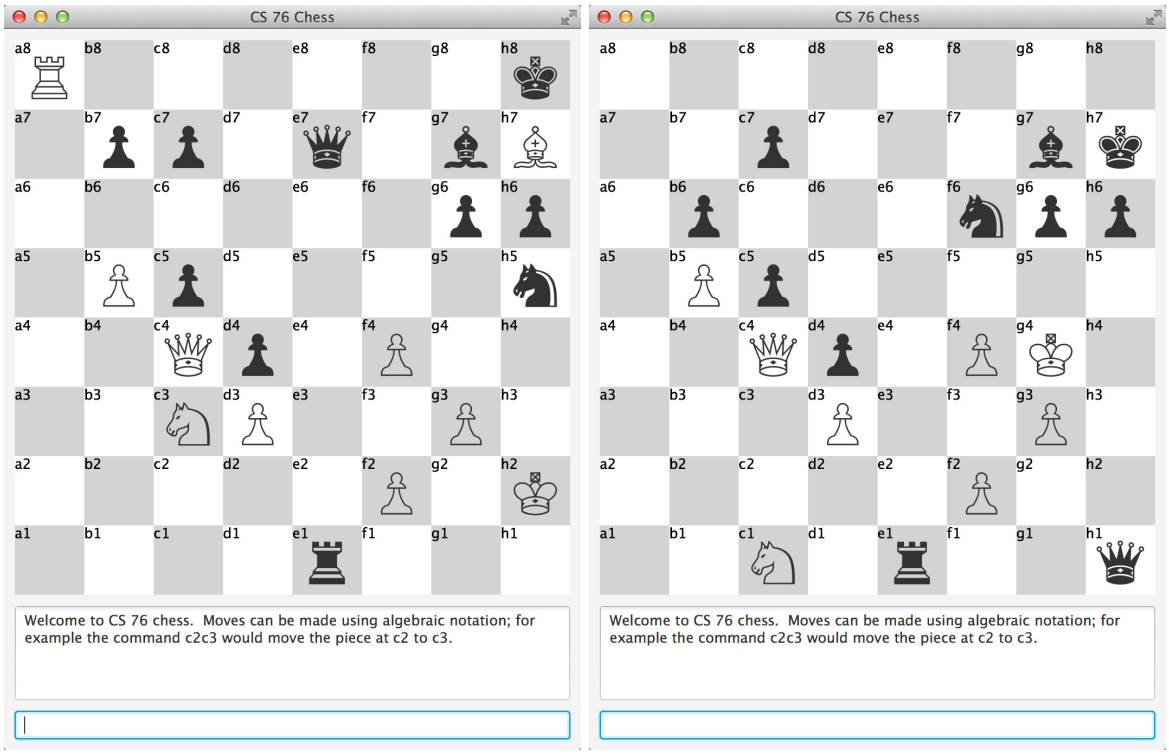


Figure 10: Results of maxdepth=4 and black is Alpha-Beta player(Left beginning, Right win)

Listing 17: Data for this move

```
Maximum depth: 4
Evluation score of first step: 315
Maximum number of states has been visited: 829502
Minimum number of states has been visited:  1554
Black player wins with 3 turns
Opening Book: 31
```

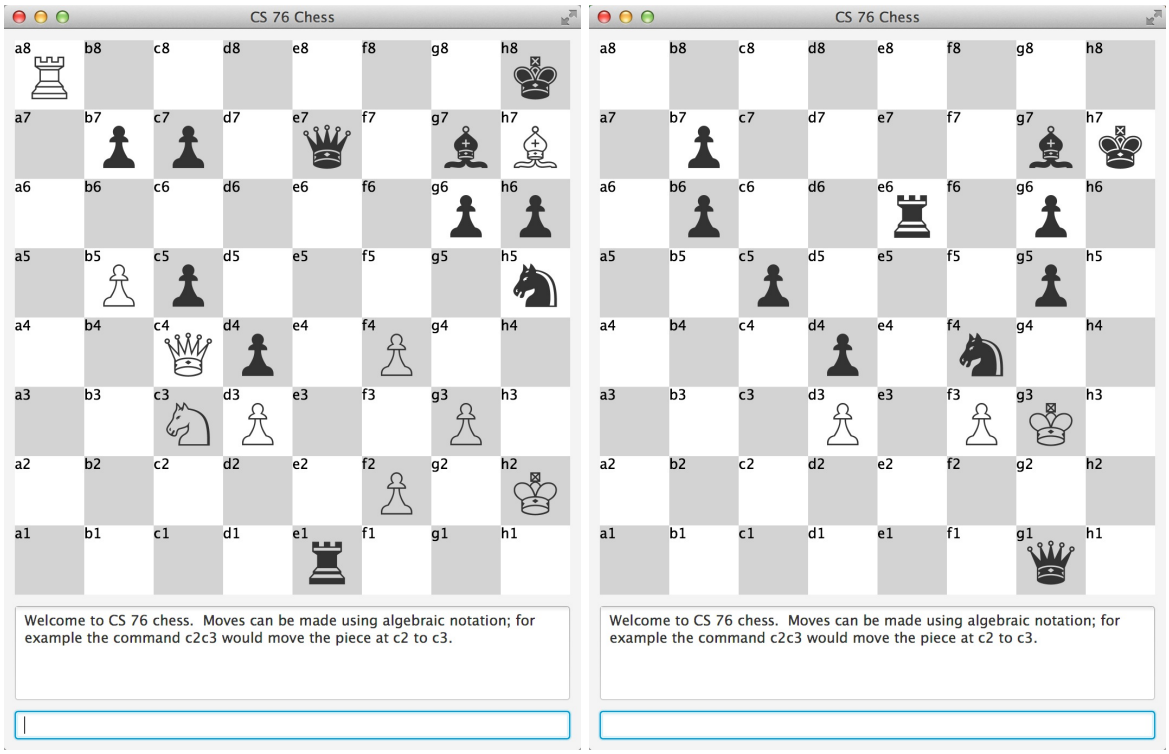


Figure 11: Results of maxdepth=4 and black is Alpha-Beta player(Left beginning, Right win)

Listing 18: Data for this move

```
Maximum depth: 6
Evluation score of first step: 414
Maximum number of states has been visited: 1614726
Minimum number of states has been visited:  2087
Black player wins with 6 turns
Opening Book: 31
```

From the test results, figure 8, figure 9 and figure 10 are using the same beginning positions and evaluation functions and same maximum depth with figure 5, figure 6 and figure 7. And the result is clear, that using the Alpha Beta Pruning function, the states that have been visited are reduced a lot and result in a quicker running time and also it can run with a maximum depth of 6 which is twice maximum depth as previous ones. And as I predicted, these test result are not smarter than before because they have the same evaluation score for the first step. Therefore, Alpha Beta pruning is good to reduce running time for each step and can help to get a bigger maximum depth to play a game. However, the AI player with Alpha Beta Pruning looks "smarter" than usual is because first, if they use the same maximum depth, it moves faster (Think quicker). Second, if they use different maximum depth, it can think deeper and in turn have a better move selection.

4 Transposition table

Transposition tables (often referred to as hash tables) are used for storing chess positions during the search to avoid re-searching positions that can be reached via several different move sequences. This can speed up the search dramatically, particularly in the endgame[5].

Such a table is a hash table of each of the positions analyzed so far up to a certain depth. On encountering a new position, the program checks the table to see if the position has already been analyzed; this can be done quickly, in expected constant time. If so, the table contains the value that was previously assigned to this position; this value is used directly. If not, the value is computed and the new position is entered into the hash table. This is essentially memorization applied to the search function[6].

4.1 Implementation

To have such a table, I need to have firstly have a table structure to know what I will store in the table. The code is shown below:

Listing 19: Table structure

```
public class tableEntry{
  int type;//Exact = 0, Upper = 1, Lower = -1
  int score;
  int depthleft;

  public tableEntry(int t, int s, int d){
    this.type = t;
    this.score = s;
    this.depthleft = d;
    }
}

//Using HashMap to store transposition table
HashMap<Long, tableEntry> table = new HashMap<Long, tableEntry>();
```

The first position I will put the key in and the second place I will put the type in. Node types as established by the search are stored inside the transposition table, indicating whether the score is either exact, lower or upper bounded[7]. Then, I will store the score value and the depth left of current situation in.

After creating a table structure, I write a function to add each entry to the table. The code is below:

Listing 20: addNodeToTable

```
void addNodeToTable(HashMap<Long, tableEntry> t,long zobristKey, int TYPE,
    int SCORE, int DEPTHLEFT)
{
  // add node with parameters
    tableEntry existingEntry = t.get(zobristKey);
    if (existingEntry == null || DEPTHLEFT >= existingEntry.depthleft) {
      t.put(zobristKey, new tableEntry(TYPE, SCORE, DEPTHLEFT));
      }
   }
```

Then, the prepare works for transposition table is done and now I need to simply modify maxvalue() function and minvalue() function to get using the transposition table based on Alpha Beta Search with evaluation function. As is known, the Chesspresso has already done the zobrist hash work in Position class. So I use it here. The code is below:

Listing 21: maxvalue()

```
public int max_value(int alpha, int beta, Position p, int depth){
  if(this.cutoffTest(p, depth)==true){
    return utility(p,maxplayer);
    }

 //Check if the position has already in the hash table
 if (table.containsKey(p.getHashCode()) == true) {
   tableEntry existNode = table.get(p.getHashCode());
   if (existNode.depthleft >= currentmaxdepth - depth){
     if (existNode.type == 0){
       return existNode.score;
       }
     else if (existNode.type == 1 && existNode.score <= alpha){
       return alpha;
       }
     else if (existNode.type == -1 && existNode.score >= beta){
       return beta;
       }
     }
   }

 //The original search work
  int v = alpha;
  short [] moves = p.getAllMoves();
  states = states+moves.length;//record the number of states
  if(moves!=null){
    for(short move: moves){
      p.doMove(move);
        int score = min_value(v,beta,p,depth-1);
        if(score >v){
          v = score;
          }
        if(v >= beta){
          //add it to table(lower bound)
          addNodeToTable(table,p.getHashCode(),-1,
              beta,currentmaxdepth-depth);
          p.undoMove();
          return beta;
          }

        //add it to table(exact)
        addNodeToTable(table,p.getHashCode(),0,score,currentmaxdepth-depth);
        p.undoMove();
        }
      }
      return v;
    }
```

Listing 22: minvalue()

```
public int min_value(int alpha, int beta, Position p, int depth){
public int max_value(int alpha, int beta, Position p, int depth){
  if(this.cutoffTest(p, depth)==true){
    return utility(p,maxplayer);
    }

 //Check if the position has already in the hash table
 if (table.containsKey(p.getHashCode()) == true) {
   tableEntry existNode = table.get(p.getHashCode());
   if (existNode.depthleft >= currentmaxdepth - depth){
     if (existNode.type == 0){
       return existNode.score;
       }
     else if (existNode.type == 1 && existNode.score <= alpha){
       return alpha;
       }
     else if (existNode.type == -1 && existNode.score >= beta){
       return beta;
       }
     }
   }

 //The original search work
  int v = alpha;
  short [] moves = p.getAllMoves();
  states = states+moves.length;//record the number of states
  if(moves!=null){
    for(short move: moves){
      p.doMove(move);
      int score = max_value(alpha, v, p,depth-1);
      if(score< v){
        v = score;
        }

      if(v <= alpha){
        //add it to table(upper bound)
        addNodeToTable(table,p.getHashCode(),1, alpha,currentmaxdepth-depth);
        p.undoMove();
        return alpha;
        }

        //add it to table(exact)
        addNodeToTable(table,p.getHashCode(),0,score,currentmaxdepth-depth);
        p.undoMove();
        }
      }
    return v;
   }
```

Now, the transposition table has been implemented and it can improve the speed of running program.

4.2 Test Result

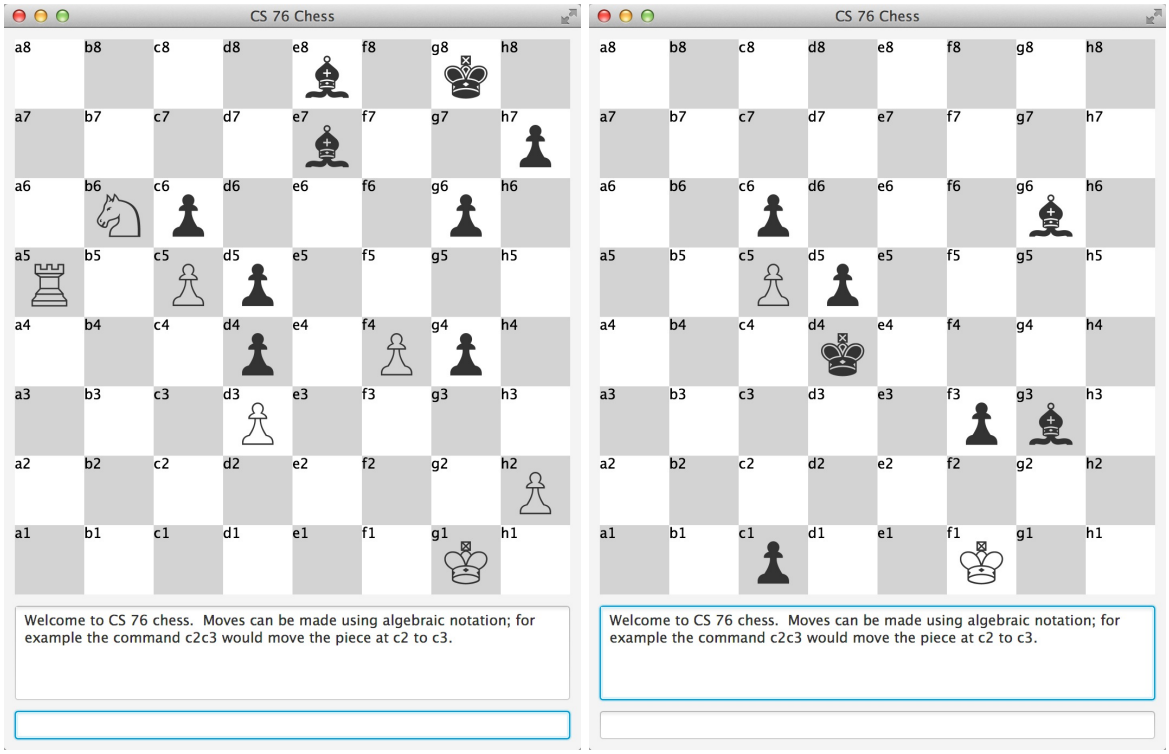


Figure 12: Results of maxdepth=3 and black is TT player(Left beginning, Right win)

Listing 23: Data for this move

```
Maximum depth: 3
Evluation score of first step: 174
Maximum number of states has been visited: 19248
Minimum number of states has been visited: 30
Black player wins with 19 turns
Opening Book: 23
```

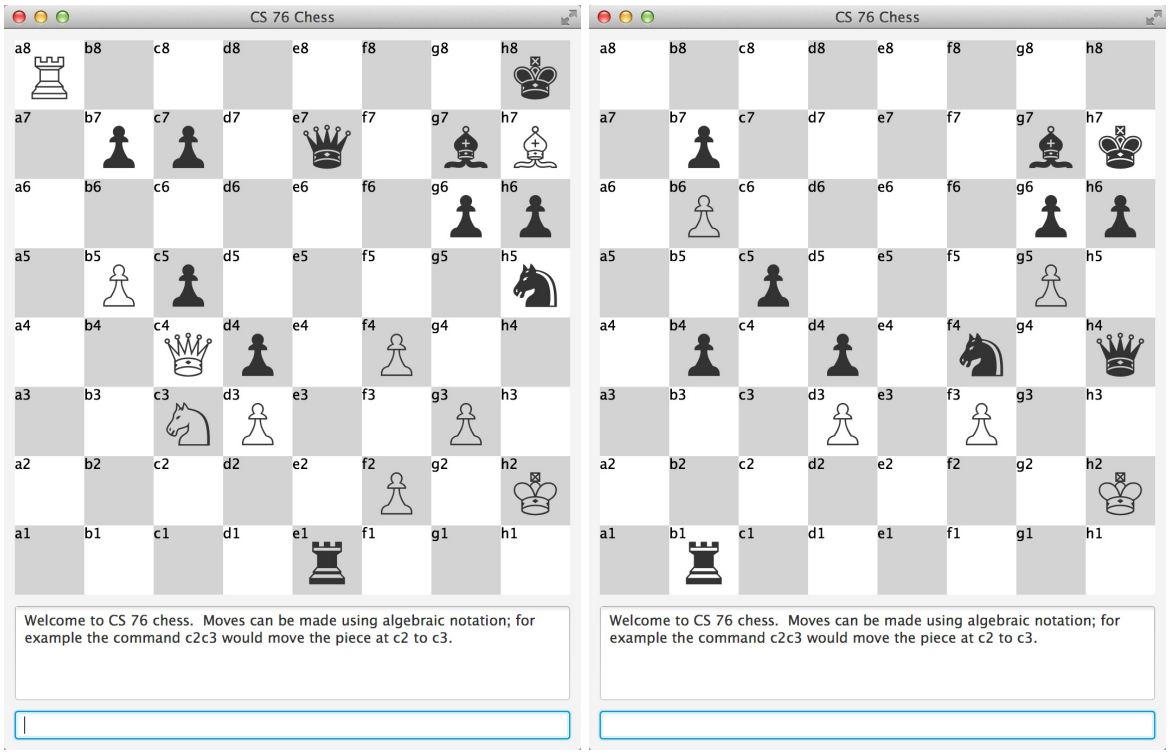


Figure 13: Results of maxdepth=3 and black is TT player(Left beginning, Right win)

Listing 24: Data for this move

```
Maximum depth: 3
Evaluation score of first step: 209
Maximum number of states has been visited: 43064
Minimum number of states has been visited:  66
Black player wins with 8 turns
Opening Book: 31
```

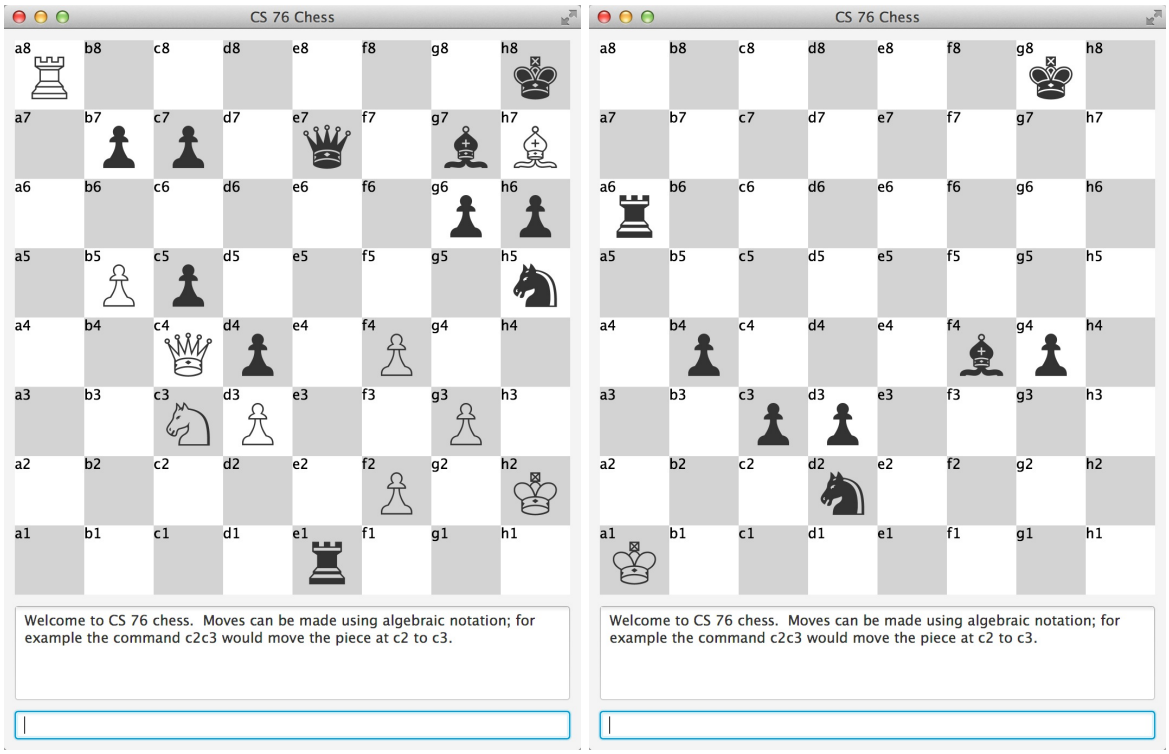


Figure 14: Results of maxdepth=4 and black is TT player(Left beginning, Right win)

Listing 25: Data for this move

```
Maximum depth: 4
Evaluation score of first step: 315
Maximum number of states has been visited: 38743
Minimum number of states has been visited:  20
Black player wins with 14 turns
Opening Book: 31
```

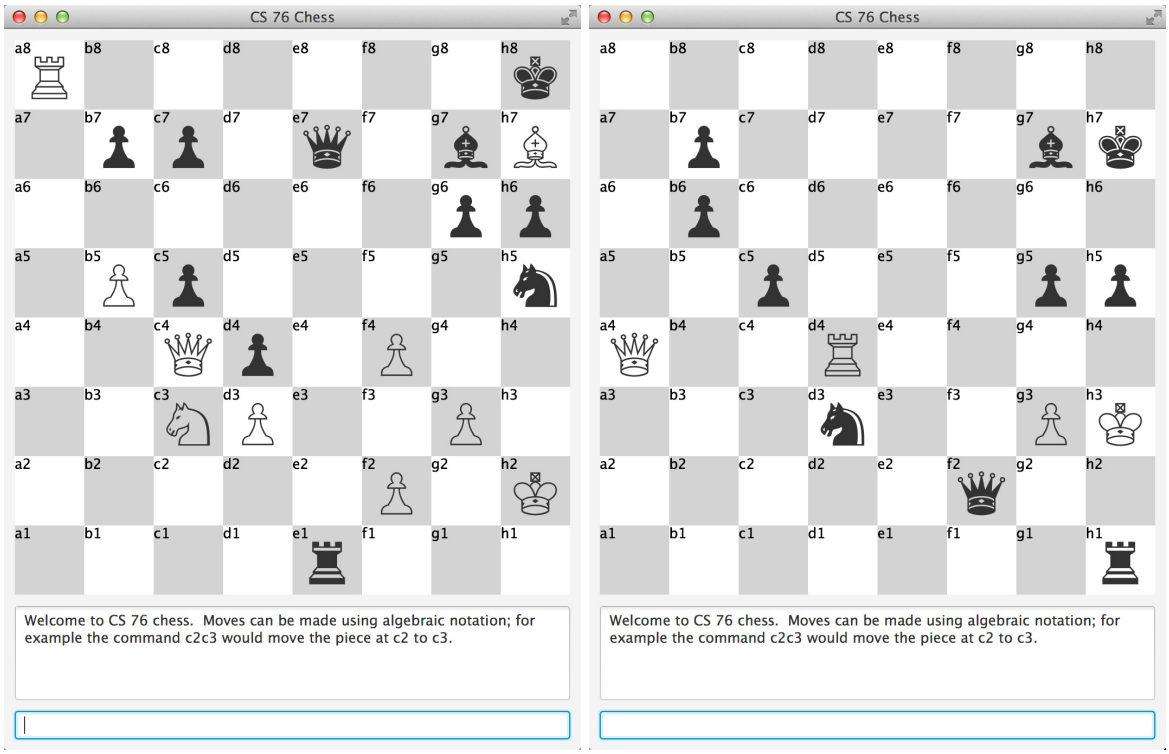


Figure 15: Results of maxdepth=6 and black is TT player(Left beginning, Right win)

Listing 26: Data for this move

```
Maximum depth: 6
Evaluation score of first step: 414
Maximum number of states has been visited: 65315
Minimum number of states has been visited:  5
Black player wins with 8 turns
Opening Book: 31
```

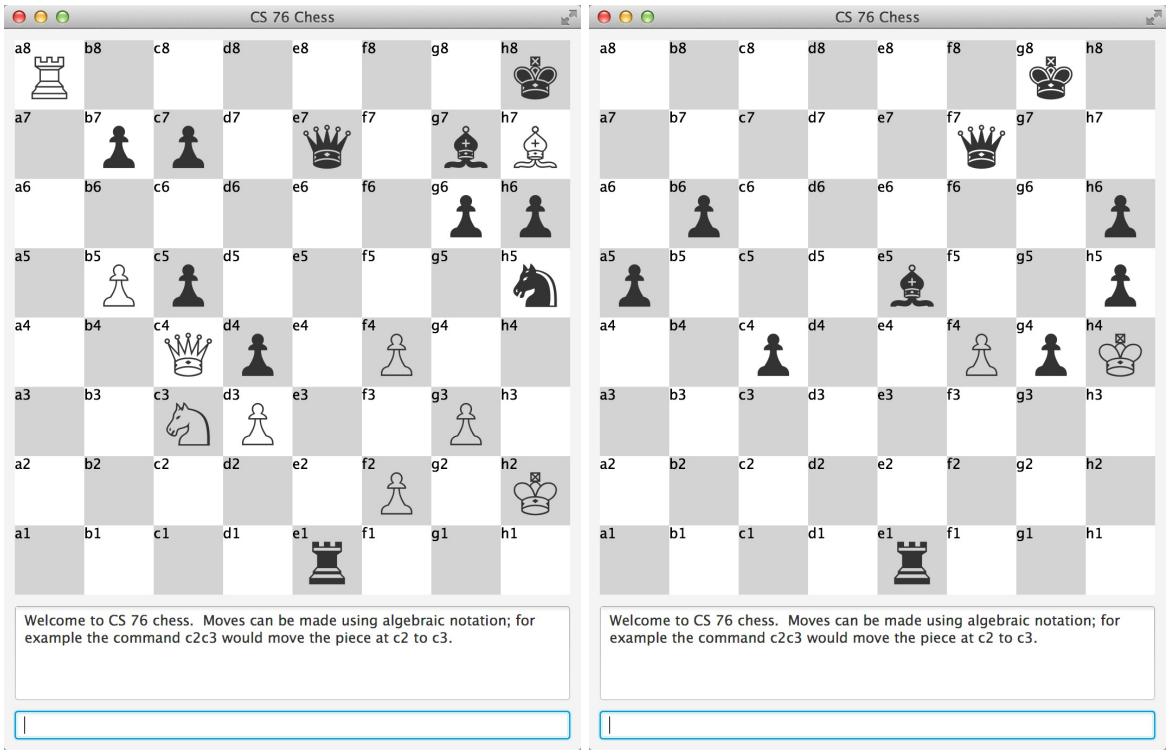


Figure 16: Results of maxdepth=16 and black is TT player(Left beginning, Right win)

Listing 27: Data for this move

```
Maximum depth: 25
Evaluation score of first step: 456
Maximum number of states has been visited: 8064
Minimum number of states has been visited:  5
Black player wins with 14 turns
Opening Book: 31
```

I use the same test cases(figure 12, figure 13, figure 14, figure 15) as I test in the alpha beta search with evaluation function and the result clearly shows that the number of states that has been visited and especially the minimum number of states decrease a lot. Thanks to transposition table, during the chess game of each test, I can see the number of states are always in a small number, just some case, it will come back to the level like in alpha beta search since the hash table did not help. And also because of transposition table, I can now using a maximum depth of 25(figure 16) to get a win state quickly, which will be a extremely slow process if I use it with any previous search methods. But all in all, the hash table or here we call it the transposition table can help a lot in speeding up the search process.

5 Opening Book (Bonus)

Openning Book is very helpful to help me get a good and interesting and practical beginning states. And I have use a lot in my previous test work.

5.1 Implementation

To use the games in book.pgn, I show firstly add a constructor to ChessGame.java. Like here:

Listing 28: New Constructor of ChessGame

```
//read a game and get its position
public ChessGame(Game g){
  position = g.getPosition();
  }
```

Then, I will add the following code into the Client.java, inside start().The implementation of it is as below:

Listing 29: Openning Book

```
  game = new ChessGame();

  //read the given book.pgn
  File f = new
      File("/Users/shiboying/Desktop/14winter/ai/hw4/provided_chess/book.pgn");
  FileInputStream fis = new FileInputStream(f);
  pgnReader = new PGNReader(fis, "book.pgn");

  //store each game in book.pgn
  ArrayList<ChessGame> openingBook = new ArrayList<ChessGame>();

  //hack: we know there are only 120 games in the opening book
  for (int i = 0; i < 120; i++)  {
    ChessGame g = new ChessGame(pgnReader.parseGame());
    openingBook.add(g);
    }

    //get random beginning states from the 120 games
    game = openingBook.get(new Random().nextInt(120));
```

5.2 Test Result

Here are four opening results from book.pgn, the first one is the 119th in book.pgn, the second one is the 88th in book.pgn, the third one is the 77th in book.pgn, the final one is 26th in book.pgn:

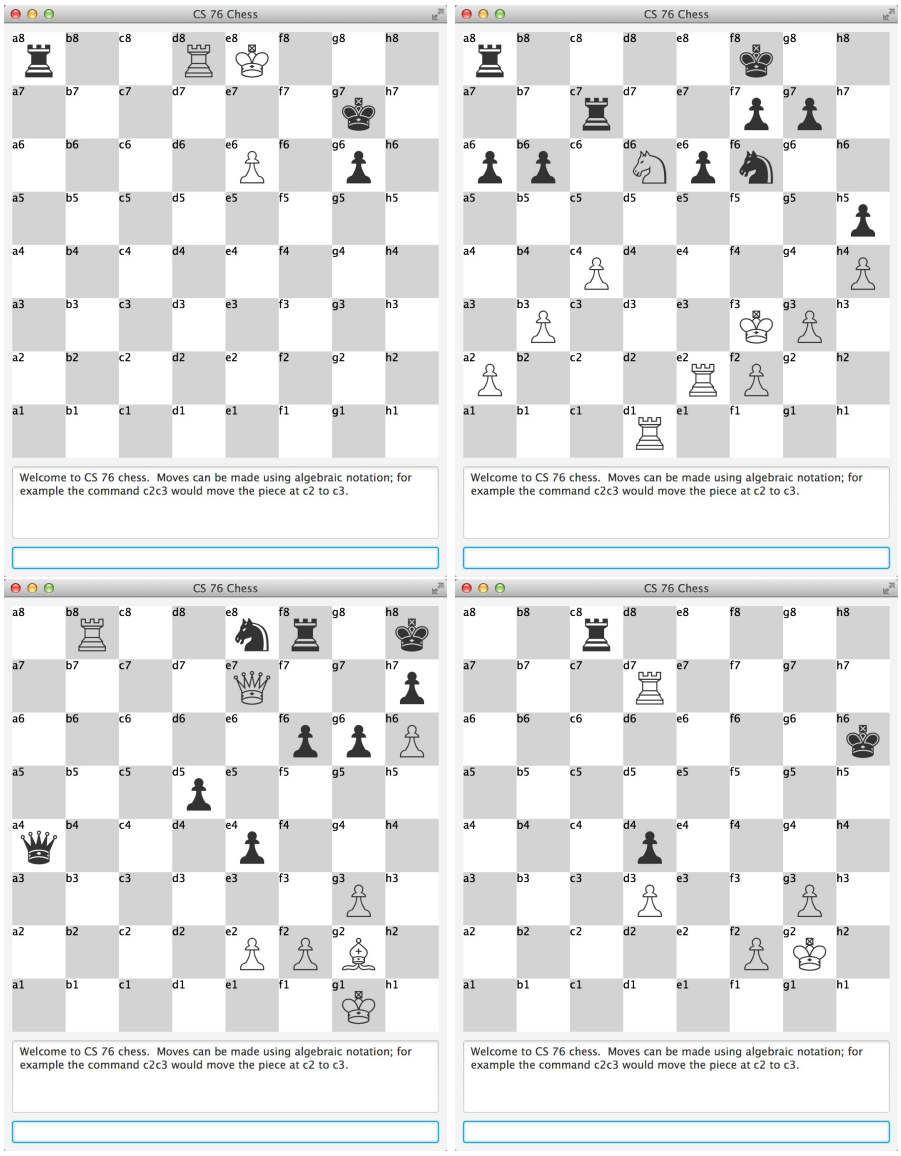
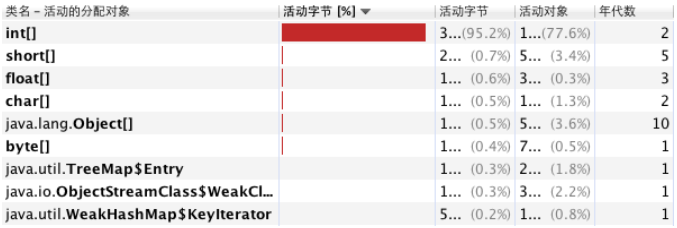


Figure 17: Openning Book Results

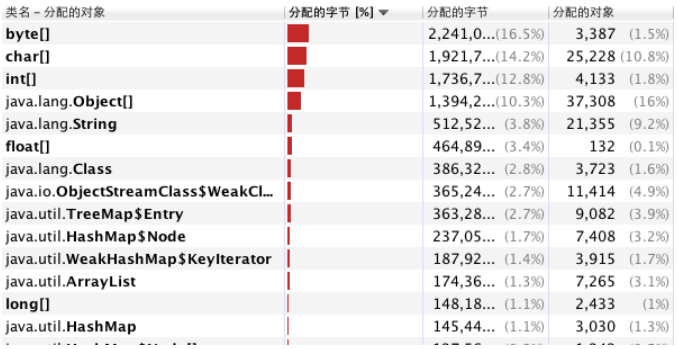
6 Profiling (Bonus)

In order to keep track of how the program runs and the data in it, I use a tool named VisualVM 1.3.7 to help me record the data. Here are results of profiling for the same maximum depth = 3, for each using Minimax Search with evaluation function, alpha beta pruning and transposition table on the standard board.



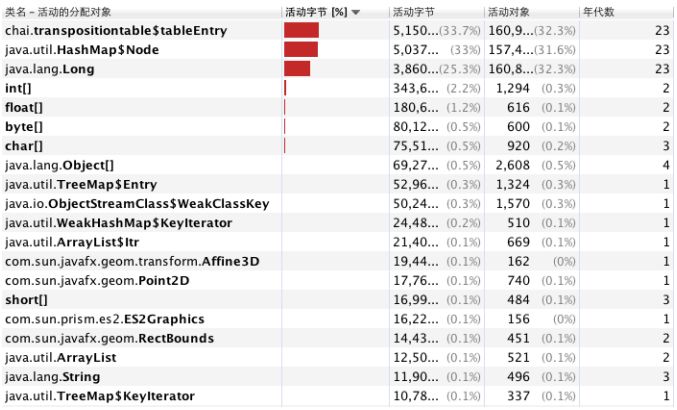
类名 – 活动的分配对象	活动字节 [%]	活动字节	活动对象	年代数
int[]		3...(95.2%)	1...(77.6%)	2
short[]		2... (0.7%)	5... (3.4%)	5
float[]		1... (0.6%)	3... (0.3%)	3
char[]		1... (0.5%)	1... (1.3%)	2
java.lang.**Object[]**		1... (0.5%)	5... (3.6%)	10
byte[]		1... (0.4%)	7... (0.5%)	1
java.util.**TreeMap$Entry**		1... (0.3%)	2... (1.8%)	1
java.io.**ObjectStreamClass$WeakCl...**		1... (0.3%)	3... (2.2%)	1
java.util.**WeakHashMap$KeyIterator**		5... (0.2%)	1... (0.8%)	1

Figure 18: Minimax Search with evaluation function



类名 – 分配的对象	分配的字节 [%]	分配的字节	分配的对象
byte[]		2,241,0...(16.5%)	3,387 (1.5%)
char[]		1,921,7...(14.2%)	25,228 (10.8%)
int[]		1,736,7...(12.8%)	4,133 (1.8%)
java.lang.**Object[]**		1,394,2...(10.3%)	37,308 (16%)
java.lang.**String**		512,52... (3.8%)	21,355 (9.2%)
float[]		464,89... (3.4%)	132 (0.1%)
java.lang.**Class**		386,32... (2.8%)	3,723 (1.6%)
java.io.**ObjectStreamClass$WeakCl...**		365,24... (2.7%)	11,414 (4.9%)
java.util.**TreeMap$Entry**		363,28... (2.7%)	9,082 (3.9%)
java.util.**HashMap$Node**		237,05... (1.7%)	7,408 (3.2%)
java.util.**WeakHashMap$KeyIterator**		187,92... (1.4%)	3,915 (1.7%)
java.util.**ArrayList**		174,36... (1.3%)	7,265 (3.1%)
long[]		148,18... (1.1%)	2,433 (1%)
java.util.**HashMap**		145,44... (1.1%)	3,030 (1.3%)

Figure 19: alpha beta pruning



类名 – 活动的分配对象	活动字节 [%]	活动字节	活动对象	年代数
chai.**transpositiontable$tableEntry**		5,150...(33.7%)	160,9...(32.3%)	23
java.util.**HashMap$Node**		5,037... (33%)	157,4...(31.6%)	23
java.lang.**Long**		3,860...(25.3%)	160,8...(32.3%)	23
int[]		343,6... (2.2%)	1,294 (0.3%)	2
float[]		180,6... (1.2%)	616 (0.1%)	2
byte[]		80,12... (0.5%)	600 (0.1%)	2
char[]		75,51... (0.5%)	920 (0.2%)	3
java.lang.**Object[]**		69,27... (0.5%)	2,608 (0.5%)	4
java.util.**TreeMap$Entry**		52,96... (0.3%)	1,324 (0.3%)	1
java.io.**ObjectStreamClass$WeakClassKey**		50,24... (0.3%)	1,570 (0.3%)	1
java.util.**WeakHashMap$KeyIterator**		24,48... (0.2%)	510 (0.1%)	1
java.util.**ArrayList$Itr**		21,40... (0.1%)	669 (0.1%)	1
com.sun.javafx.geom.transform.**Affine3D**		19,44... (0.1%)	162 (0%)	1
com.sun.javafx.geom.**Point2D**		17,76... (0.1%)	740 (0.1%)	1
short[]		16,99... (0.1%)	484 (0.1%)	3
com.sun.prism.es2.**ES2Graphics**		16,22... (0.1%)	156 (0%)	1
com.sun.javafx.geom.**RectBounds**		14,43... (0.1%)	451 (0.1%)	2
java.util.**ArrayList**		12,50... (0.1%)	521 (0.1%)	2
java.lang.**String**		11,90... (0.1%)	496 (0.1%)	3
java.util.**TreeMap$KeyIterator**		10,78... (0.1%)	337 (0.1%)	1

Figure 20: transposition table

From figure 18, figure 19 and figure 20. I can easily figure out the improvement for searching process one by one. At first, the Minimax Search with evaluation function just add a evaluation function and give a score for a move, and it does not give any pruning for the search and traverse all states for each depth. Therefore it need to store a lot of integer value. And then by using Alpha Beta pruning, the search has a

sense of pruning the search results and need not to traverse all states. And finally, by using a transposition table, it stores those positions that have been known in a hash table and improve the speed of the program.

From my test of profiling, the most of the executing time being spent is to traverse each possible positions or store possible positions, if the number of possible states are a large number, then it is time consuming. While if the number of states is not large or putting some of them into a hash table, then the time is acceptable.

When talking about improving the program and especially regarding to speed up the entire process, I think the part is still to reduce nodes that need to be visited. If the searching process can take only a short time, then the entire process will be quick. Also, if talking about making the AI player much smarter, it is very necessary to improve the heuristic function, that is the evaluation function. Here, I only take consideration of the material scores and pieces domination scores as branch factors. In fact, there are still many other factors like mobilitiy(as the number of legal moves available to White), king safety and so on[8]. Moreover, the weighted value for each factor is also need to be consider. After giving a better heuristic function, then the score for each move will be more reliable.

7 Move Reordering (Bonus)

Just as the assignment indicated, I need to use the previous results from the previous iterative deepening search. So here, I have a temp variable to help me store the best evaluation value and the best move from the previous iterative deepening search. And add a check condition to find the best results including the evaluation value and the best move between the previous iterative deepening search result and current iterative deepening search result. And finally, return the really best results from them.

7.1 Implementation

Here is my implementation for move reordering and it is based on transposition table:

Listing 30: Reording

```
short finalmove = 0;
int finalvalue = Integer.MIN_VALUE;

bestmove = moves[0];//initial a best move
int[] op = new int[moves.length];//store each possible position
states = states+moves.length; //record visited states
int max = -10000;//initial a temp max
short pos = 0;//initial temp move
for (int i=0;i<moves.length;i++){
  position.doMove(moves[i]);
  op[i] = minimax(position,j);

  if(op[i]>=max){
    max = op[i];
    pos= moves[i];
    bestmove = pos;
    bestevluationscore = max; //record current best evaluation score
    }
    position.undoMove();
      }
    }

    //Move reordering
```

```
if(bestevluationscore > finalvalue){
   finalvalue = bestevluationscore;
    finalmove = bestmove;
    }
```

In my implementation, I use a `finalmove` to store the finally real best move between the previous search and the current search; and `finalvalue` to help me store the finally real best evaluation score between the previous search and the current search. As the same with the original searching process, I have a temp best results, here are `bestmove` and `bestevluaitonscore`. And since I added `if(bestevluationscore > finalvalue)` to find the real best result, I finally got the move reordering.

7.2 Test Result

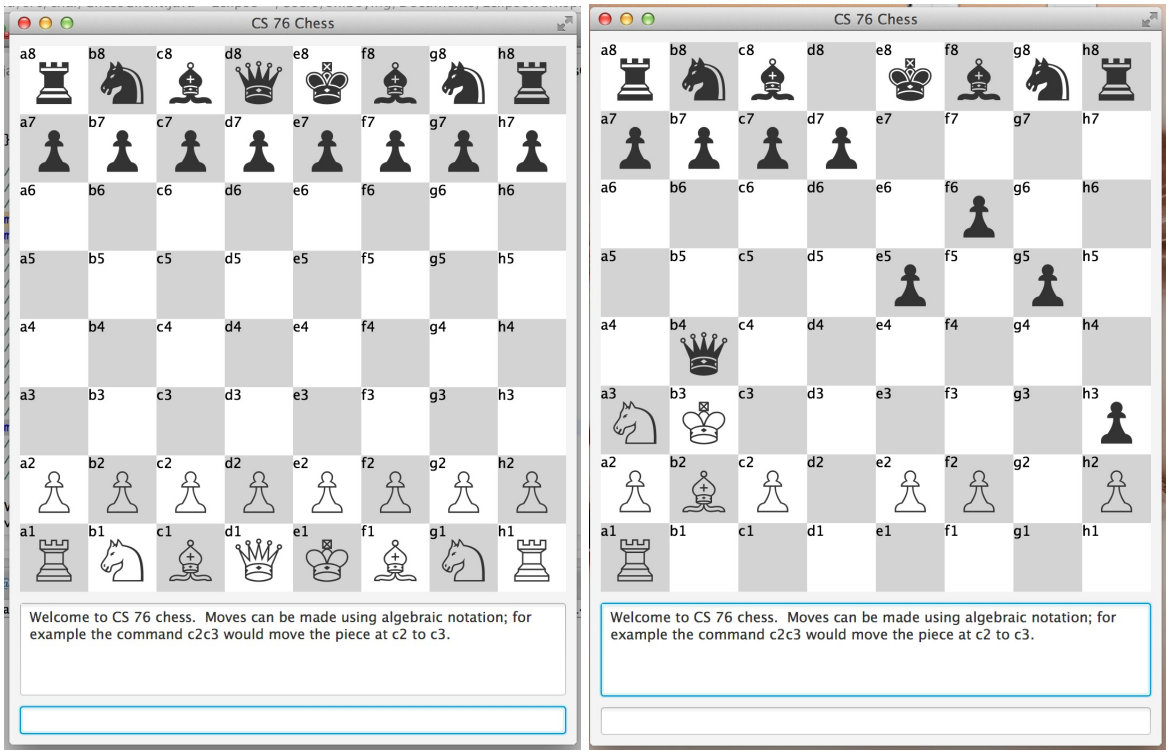


Figure 21: Results of maxdepth=3 and black is move reordering player(Left beginning, Right win)

Listing 31: Data for this move

```
Maximum depth: 3
Evaluation score of first step: 42
Maximum number of states has been visited: 48336
Minimum number of states has been visited:  37
Black player wins with 13 turns
Standard Opening
```

8 Null-move Heuristic (Bonus)

The null-move heuristic is a way to help speed up the heuristic function for alpha beta search. It is based on the fact that most reasonable chess moves improve the position for the side that played them. So, if the player whose turn it is to move can forfeit the right to move (or make a "null move" - an illegal action in chess) and still have a position strong enough to produce a cutoff, then the current position would almost certainly produce a cutoff if the current player actually moved[9]. In my implementation of null-move heuristic, I just to add a check condition for my alpha beta search to check each evaluation score that if the result can have a better situation or not, if a move will not get a better situation, then it is better to make no move at this turn. And therefore, it can promise ever step for a player is a good or no loss move.

8.1 Implementation

My implementation of it is here. In my previous maxvalue() function, I add the following check condition:

Listing 32: Null-move change in my maxvalue()

```
if(!p.isCheck() && null_flag==0 ){
 //null_flag is initial to be 0. And when there is null move, change it to 1
  null_flag = 1;
  //make no move, and change player to the other side
  p.setToPlay(1-p.getToPlay());
  //get new evaluation score, beta-1 to make the window between alpha and
     beta to be small
  // R=2
  //smaller the window between alpha and beta and reduced depth
  score = min_value(beta-1,beta,p,depth-R-1);
  if(score>=beta){
    return score;
    }
  }
```

And also, I need to have a modify in my minvalue() function too, and here is my code:

Listing 33: Null-move change in my minvalue()

```
if(!p.isCheck() && null_flag==0 ){
 //null_flag is initial to be 0. And when there is null move, change it to 1
  null_flag = 1;
  //make no move, and change player to the other side
  p.setToPlay(1-p.getToPlay());
  //get new evaluation score
  // R=2
  //smaller the window between alpha and beta and reduced depth
  score = max_value(alpha,alpha-1,p,depth-R-1);
  if(score<=alpha){
    return score;
    }
  }
```

The idea of the implementation is to make sure the side who get a free move cannot reach up to alpha and do some other searching but at the same time I need to use a reduced depth. In order to make a reduced depth, the window between alpha and beta is minimal because we are just interested if the opponent can read up to alpha, it does not matter how much worse he is. And the R is how much we reduce the depth,

and here I use R = 2 because it is faster than doing the full search and is commonly accepted as a good reduction to search a null-move. R=1 is usually too small, making the null-move search too slow. And R=3 is too large, making the null-move too likely to miss tactics[10]. Here in my implementation code, I use a flag to help me know if I encounter a free move and if there is a free move, I make it change player to finish the actual action of null-move.

8.2 Test Result

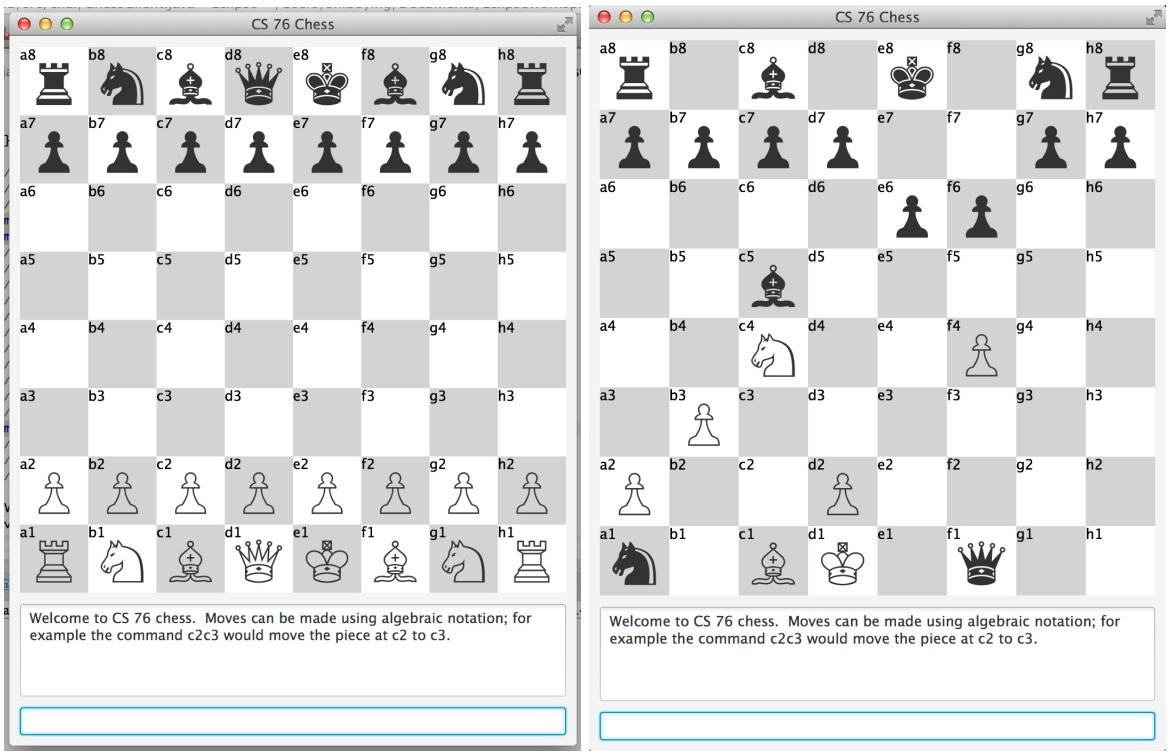


Figure 22: Results of maxdepth=3 and black is null-move alpha beta player(Left beginning, Right win)

Listing 34: null-move alpha beta

```
Maximum depth: 3
Evaluation score of first step: 89
Maximum number of states has been visited: 238606
Minimum number of states has been visited:  694
Black player wins with 16 turns
Null-move step: the 8th step of black player
Standard Opening
```

9 Improved cut-off Test (Bonus)

A quiescence search attempts to emulate this behavior by instructing a computer to search "interesting" positions to a greater depth than "quiet" ones (hence its name) to make sure there are no hidden traps and, usually equivalently, to get a better estimate of its value. As the main motive of quiescence search is usually to get a good value out of a poor evaluation function, it may also make sense to detect wide fluctuations in values returned by a simple heuristic evaluator over several player[11].

9.1 Implementation

My implementation is shown here, a quiesce search need to firstly get a calculate result of utility function and then traverse all moves that contain capturing:

Listing 35: quiesce search

```
int QuiesceSearch(int alpha, int beta,Position p, int player){
  //calculate a temp utility value
  int temp = utility(p,player);
  if (temp >= beta)
    return beta;
  if(alpha<temp)
    alpha = temp;

  //Get all moves that contain capturing
  for (short m :p.getAllCapturingMoves()) {
    p.doMove(m);
    temp = -quiesce(-beta,-alpha,p,player);
    p.undoMove();

    if (score >= beta)
      return beta;
    if (score > alpha)
      alpha = score;
      }
      return alpha;
  }
```

In chess game, quiescence searches usually include all capture moves, so that tactical exchanges do not mess up the evaluation. In principle, quiescence searches should include any move which may destabilize the evaluation function?if there is such a move, the position is not quiescent. Not all search algorithms require a quiescence search, but depth-limited search algorithms really need this search[12].

9.2 Test Result

Listing 36: quiescence search

```
Maximum depth: 3
Evaluation score of first step: -21
Maximum number of states has been visited: 392621
Minimum number of states has been visited:  831
Black player wins with 15 turns
Standard Opening
```

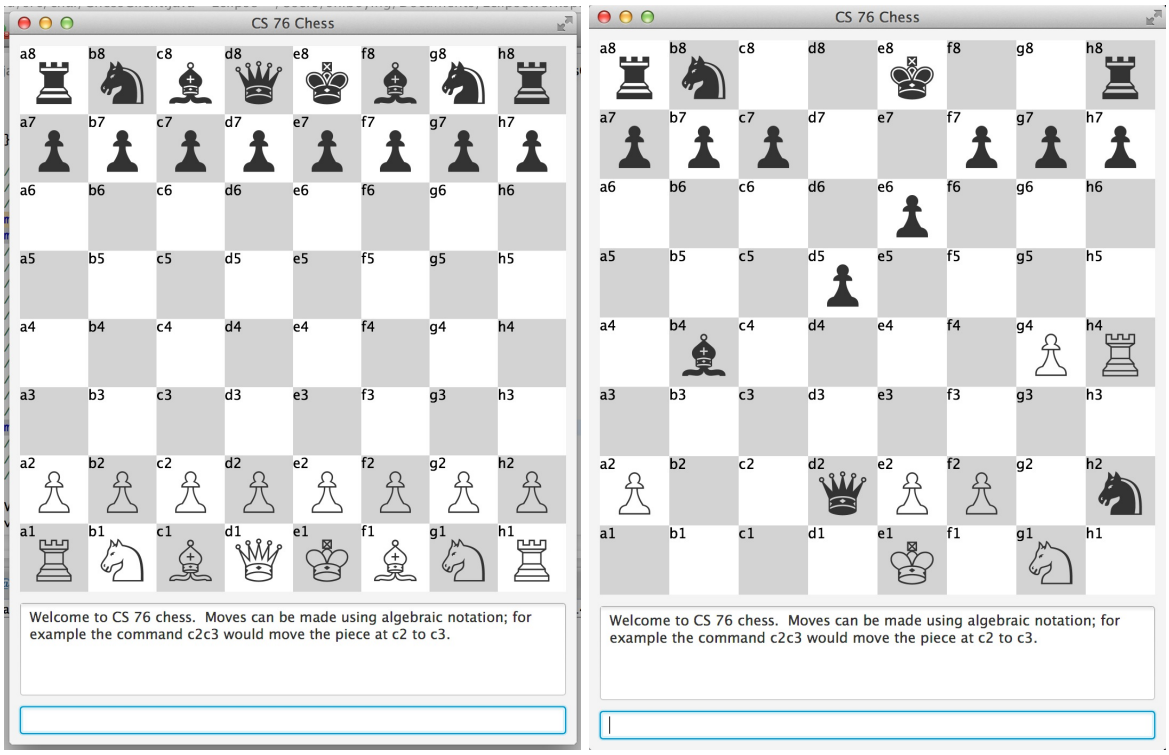


Figure 23: Results of maxdepth=3 and black is quiescence search player(Left beginning, Right win)

10 Related work[13] (Bonus)

As what I have implemented above, I can make a Alpha-Beta Pruning Search AI player play chess with Random AI player and human beings. Although I can have two Alpha-Beta Pruning Search AI players to play chess with each other, however, I always have a dead lock situation that when they have already played for a while and then they will move the same step forever.

Considering this simultaneous moves problem, Abdallah Saffidine, and Hilmar Finnsson and Michael Buro has come up with a solution to deal with two Alpha-Beta Pruning Search AI players playing problem. The idea they have, is to maintain upper and lower bounds for achievable payoffs in states with simultaneous actions and domination action pruning based on the feasibility of certain linear program.

In their paper named *Alpha-Beta Pruning for Games with Simultaneous Moves*, they gave details about their Simultaneous Move Alpha-Beta (SMAB) pruning. Specifically, they traverse a given game tree in depth-first manner like the original Alpha-Beta algorithm, and for each position they will have a lower bound and an upper bound and a evaluation score. And as long as there is a evaluation score lies outside the window of upper bound and lower bound, then they will do pruning. Then, by looping through all joint action pairs first checking trivial exit conditions and if these fail, proceeding with computing optimistic and pessimistic bounds for the entry in questions, and then recursively computing the entry value, their SMAB pruning can be incorporated in a depth-first search.

Finally, after testing their algorithm, they showed that SMAB pruning procedure can reduce the node count and run-time when solving nontrivial games.

11 Citation

Minimax Search
[1].http://en.wikipedia.org/wiki/Minimax#cite_note-5
[2].https://www.cs.tcd.ie/Glenn.Strong/3d5/minimax-notes.pdf
[3].https://canvas.dartmouth.edu/courses/744/assignments/7180
Alpha Beta Pruning
[4].http://en.wikipedia.org/wiki/Alpha-beta_pruning
Transposition Table
[5].http://www.sigmachess.com/_usersmanual/ManualData/TransTables.html
[6].http://en.wikipedia.org/wiki/Transposition_table
[7].http://chessprogramming.wikispaces.com/Node+Types#PV
Profiling
[8].http://en.wikipedia.org/wiki/Evaluation_function
Null-move
[9].http://en.wikipedia.org/wiki/Null-move_heuristic
[10].http://mediocrechess.blogspot.com/2007/01/guide-null-moves.html
Improved cut-off Test
[11].http://en.wikipedia.org/wiki/Quiescence_search
[12].http://satirist.org/learn-game/methods/search/quiesce.html
[13].https://skatgame.net/mburo/ps/aaai12-sbs.pdf
Code Work
Used Chesspresso.
Got idea from discussion on Piazza.
Discussed with TA Yinan Zhang.