Chess AI

Carter J. Bastian

February 10, 2016

Introduction

This assignment consists of a series of algorithms implemented to play chess. The first algorithm, Minimax, is implemented in `MinimaxAI.java`. The second, Alpha-Beta search, is implemented in `AlphaBetaAI.java`.

Iterative Deepening Minimax

The Minimax algorithm was implemented first. The `MinimaxAI`class uses three instance variables to get the results efficiently. The first, `maxDepth`, is the maximum allowed depth for any iteration of search. The second, `tempDepth`, is the lowest depth in which a win-value has yet been achieved. The third, `IDdepth`, is the current max-depth allowed in this iteration of deepening.

Notice that this second variable was necessary to ensure that, within a depth, we look only at the shortest path found to the win. In this way, we ensure that we never get stuck perpetually chasing a 4-move win (and thereby never achieving it at all).

This algorithm consists of three functions: `minimaxDecision`, `minValue`, and `maxValue`.

The first method runs the iterative deepening functionality with a for-loop iterating through each depth from one to the maximum depth allowed. Within this for-loop, the algorithm loops through each potential move available from the starting position, and calculates value returned from the `minValue`method. The move corresponding to the maximum of these stored values (implemented by only keeping track of the maximum seen yet) is returned.

In doing so, the AI finds the best possible outcome under the assumption that its opponent is playing optimally. A full listing of the `minimaxDecision`method is provided below:

```
private synchronized short minimaxDecision(Position position) {
  for (IDdepth = 1; IDdepth < maxDepth; IDdepth++) {
    short[] moves = position.getAllMoves();
    int currUtil = loseVal;
    int bestUtil = loseVal;
    int bestIdx = 0;
    short currMove;

    this.tempDepth = this.IDdepth;
    for (int i = 0; i < moves.length; i++) {
      try {
        currMove = moves[i];
        position.doMove(currMove);
        currUtil = minValue(position, 1);
        position.undoMove();
        // Use the last (and thus shortest) path to victory!
        if (bestUtil <= currUtil) {
          bestUtil = currUtil;
          bestIdx = i;
        }
      } catch (Exception e) {
        System.out.println(e);
        System.out.println("Invalid Move at start state...");
      }
    }
    if (bestUtil == winVal)
      return(moves[bestIdx]);

    if ((IDdepth+1) == maxDepth)
      return(moves[bestIdx]);
  }
  return -1;
}
```

The second function, `maxValue`, returns the maximum utility possible from this move by looping through all of the available moves and returning that which has the highest value returned from `minValue`. Notice that, before the algorithm begins, the `cutOffTest`is applied so as to ensure that we have not hit the base case. If we have, the calculated utility value is returned. Both of these functionalities are addressed in the next section. Below is a listing of the method.

```
private synchronized int maxValue(Position position, int currDepth) {
  if (cutOffTest(position, currDepth)) {
    return utility(position, currDepth);
  }

  int util = loseVal;
  int curr;
  short [] moves = position.getAllMoves();

  for (int i = 0; i < moves.length; i++) {
    try {
      position.doMove(moves[i]);
      curr = minValue(position, currDepth + 1);
      position.undoMove();
      if (curr > util)
        util = curr;
    } catch(Exception e) {
      System.out.println("Illegal move found...");
    }
  }
  return util;
}
```

The third and final method, `minValue`, is simply the opposite of `maxValue`. It returns the minimum of the values computed by passing each possible move into the `maxValue`method. It is listed in full below.

```
private synchronized int minValue(Position position, int currDepth) {
  if (cutOffTest(position, currDepth)) {
    return utility(position, currDepth);
  }
```

```
    int util = winVal;
    int curr;

    short [] moves = position.getAllMoves();

    for (int i = 0; i < moves.length; i++) {
      try {
        position.doMove(moves[i]);
        curr = maxValue(position, currDepth);
        position.undoMove();
        if (curr < util)
          util = curr;
      } catch(Exception e) {
        System.out.println("Illegal move found...");
      }
    }
    return util;
  }
```

This algorithm had much success on its own and was fully capable of finding solutions to simple chess problems such as those in 1. However, with more complicated test-cases, it would struggle simply because the computation required to go deeper was too time consuming. Both of the puzzles above were solved at a depth-level of 3 in a reasonable amount of time (each move taking less than 10 seconds).

The full implementation of the minimax algorithm is available in the file `MinimaxAI.java`.

Moving to Alpha-Beta Search

In Alpha-Beta search, the only modification was to constrain the search with maximum and minimum values. These changes were extremely small in the code, but ended up having large results. Namely, the maximum possible (in reasonable time) depth grew from three to six. In practice, it was found that a depth of five was able to solve all of the test cases presented satisfactorily and proved quite difficult to best.

Furthermore, when run at the same depth, fewer than half of the nodes visited by the Minimax algorithm were visited by Alpha-Beta.

In terms of implementation, only sparse modifications to the three main functions were necessary: simply adding the minimum and maximum (alpha and beta) values as parameters to the `minValue`and `maxValue`functions, and updating these values as necessary. For example, in the for-loop within `minValue`, this was implemented as:

```
position.doMove(moves[i]);
  if (position.isLegal()) {
    curr = maxValue(position, currDepth + 1, alpha, beta);
  } else {
    curr = winVal;
  }
position.undoMove();

if (curr < util)
  util = curr;
if (util < alpha)
  return util;
if (util < beta)
  beta = util;
```

Further, in `maxValue`, this was implemented as:

```
position.doMove(moves[i]);
  if (position.isLegal()) {
    curr = minValue(position, currDepth + 1, alpha, beta);
  } else {
    curr = loseVal;
  }
position.undoMove();

if (curr > util)
  util = curr;
if (util > beta)
  return util;
if (util > alpha)
  alpha = util;
```

Notice that, in `minValue,`if we've seen a minimum value less than the acceptable minimum (alpha), we simply return it without further recursion, and if we see a minimum value greater than the upper bound, we update the upper bound with this new information. This then works in reverse for `maxValue`.

As can be seen in figure 2, the Alpha-Beta algorithm proved extremely capable at solving even quite-difficult chess problems.

The Cutoff Test and Utility Function

The Cutoff test used in the algorithm is relatively simple in that it only checks to see if there is a terminal case or if the maximum depth of recursion has been reached. The implementing code is listed below:

```
private boolean cutOffTest(Position position, int currDepth) {
  return (position.isTerminal() || (currDepth > this.tempDepth));
}
```

However, the utility method is much more important, as it decides what is or is not a good position. Poor results were achieved with Chesspresso's built-in `getMaterial`method. As such, I implemented Tomasz Michniewski's "Simplified Evaluation Function" in a method called `getMaterials`.

This method looks not only at the value of the pieces on the board, but also at the value of their positioning on the board using a series of tabular lookups into arrays to positional advantages given by each piece in each spot on the board. These tabular lookups are implemented as static instance variable's belonging to the `AlphaBetaAI`class.

Further, I also embedded a few snippets of human knowledge into the evaluation function by giving slight advantages to paired bishops and slight additional penalty for having zero pawns (as this makes the end-game difficult).

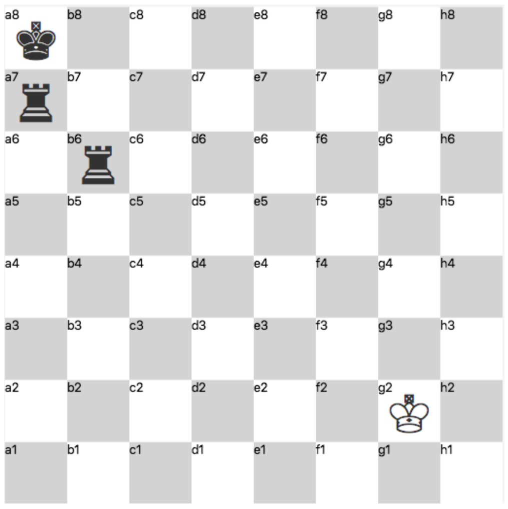
The full implementation of this method is listed below:

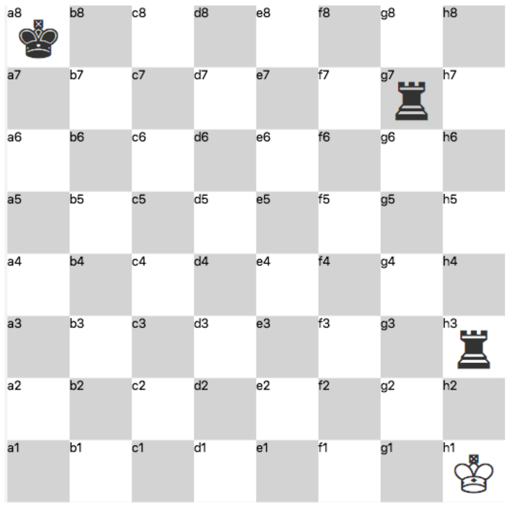
```
public int getMaterials(Position p) {
  int currStone = Chess.NO_STONE, row, col;
  int val = 0;
  int wPawns = 0, wRooks = 0, wBishops = 0, wQueens = 0, wKnights = 0, wKings = 0;
  int bPawns = 0, bRooks = 0, bBishops = 0, bQueens = 0, bKnights = 0, bKings = 0;

  for (int i = 0; i < Chess.NUM_OF_SQUARES; i++) {
    currStone = p.getStone(i);
    row = i / 8; // EG B3 = 17, 17%8 = 1, 1 = B
    col = i % 8; // With integer division => 17/8 = 2 (counting from zero)
```

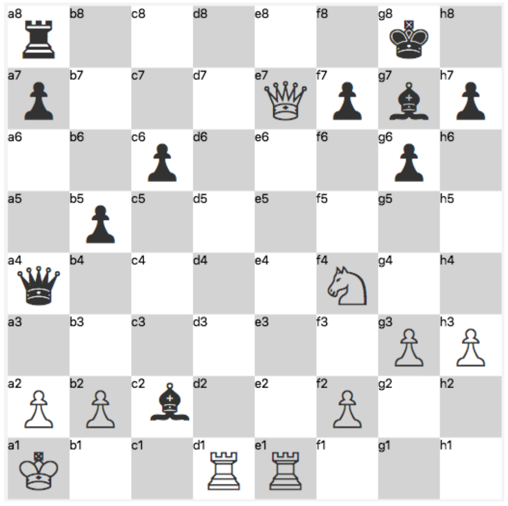
```
switch(currStone) {
  case Chess.WHITE_PAWN:
    wPawns++;
    val += (100 + (pawnTable[8*(7 - row) + col]));
    break;
  case Chess.BLACK_PAWN:
    bPawns++;
    val -= (100 + (pawnTable[8 + (8* row) - (1 +col)]));
    break;
  case Chess.WHITE_ROOK:
    wRooks++;
    val += (500 + (rookTable[8*(7 - row) + col]));
    break;
  case Chess.BLACK_ROOK:
    bRooks++;
    val -= (500 + (rookTable[8 + (8* row) - (1 +col)]));
    break;
  case Chess.WHITE_BISHOP:
    wBishops++;
    val += (330 + (rookTable[8*(7 - row) + col]));
    break;
  case Chess.BLACK_BISHOP:
    bBishops++;
    val -= (330 + (rookTable[8 + (8* row) - (1 +col)]));
    break;
  case Chess.WHITE_KNIGHT:
    wKnights++;
    val += (320 + (knightTable[8*(7 - row) + col]));
    break;
  case Chess.BLACK_KNIGHT:
    bKnights++;
    val -= (320 + (knightTable[8 + (8* row) - (1 +col)]));
    break;
  case Chess.WHITE_QUEEN:
    wQueens++;
    val += (900 + (queenTable[8*(7 - row) + col]));
    break;
  case Chess.BLACK_QUEEN:
    bQueens++;
    val -= (900 + (queenTable[8 + (8* row) - (1 +col)]));
    break;
  case Chess.WHITE_KING:
    wKings++;
    val += (20000 + (kingTable[8*(7 - row) + col]));
    break;
  case Chess.BLACK_KING:
    bKings++;
    val -= (20000 + (kingTable[8 + (8* row) - (1 +col)]));
    break;
}
```

```
    }
    /* Customizations: */
    // Penalty for no-pawns
    if (wPawns == 0) {
      val -= 20;
    }
    if (bPawns == 0) {
      val += 20;
    }
    // Bonus for bishop pair
    if (wBishops == 2) {
      val += 40;
    }
    if (bBishops == 2) {
      val -= 40;
    }
            // Everything Calculated with respect to white
    return (this.player == Chess.WHITE ? val : -1 * val);
  }
```

The difference this method made was quite drastic. In fact, when pitted against each other in 3, the better-evaluation function proved the difference between a tough loss and an easy win. Notice that the code for the poorly-evaluating AI is available in the file `BadEvalAI.java`.


(a) A Very Easy test case (unsolved by black)


(b) A Very Easy test case (solved by MinimaxAI)


(c) An Easy test case (unsolved by black)

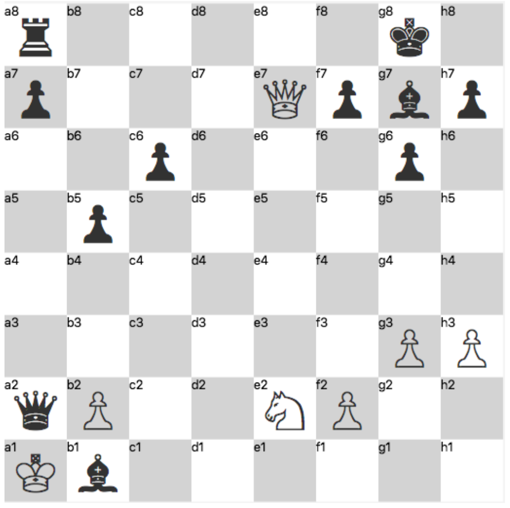
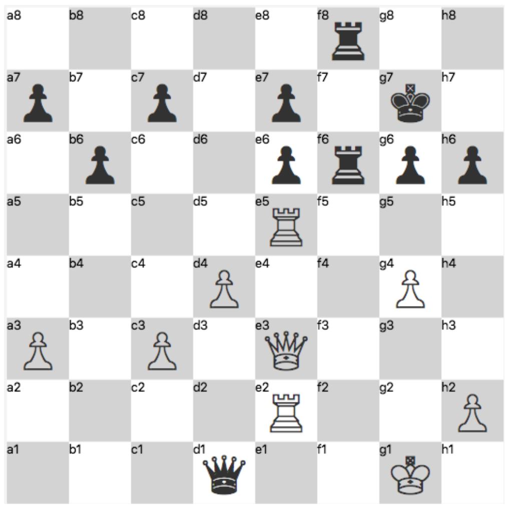
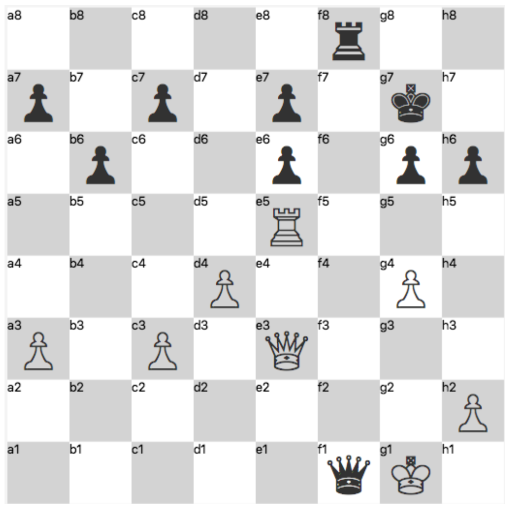
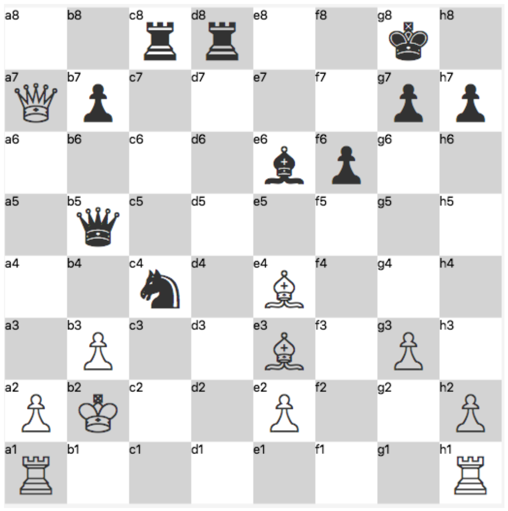
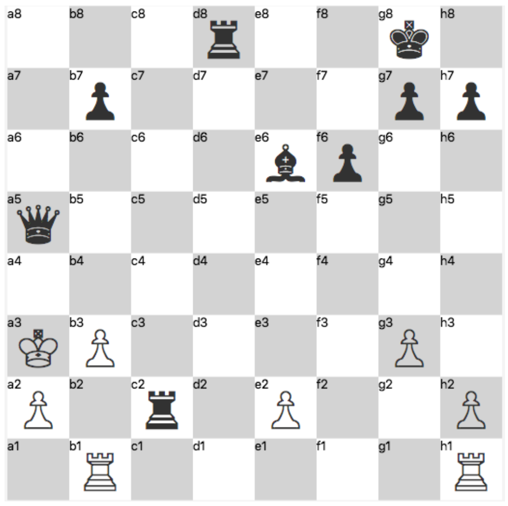

(d) An Easy test case (solved by MinimaxAI)

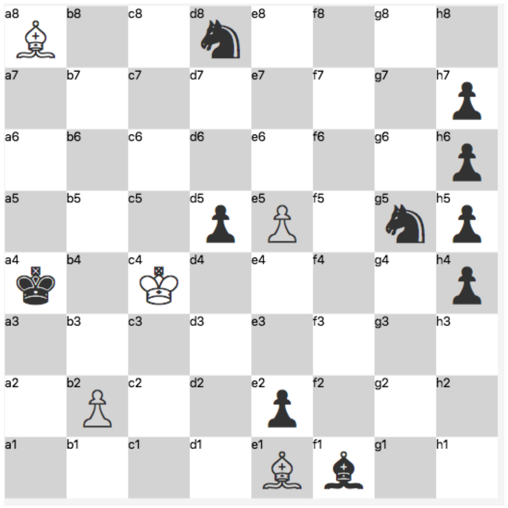
Figure 1: Minimax Solves a simple chess problem test-case to checkmate. Note that, when the Alpha Beta algorithm was used on this problem, the same results ensued. This allowed for confidence in the correctness of the Alpha Beta implementation.


(a) A medium-difficulty test case (unsolved by black)


(b) A medium-difficulty test case (solved by AlphaBetaAI)


(c) A difficult test case (unsolved by black)


(d) An difficult test case (solved by AlphaBetaAI)


(e) An very difficult (20+ moves to checkmate) test case (unsolved by black)

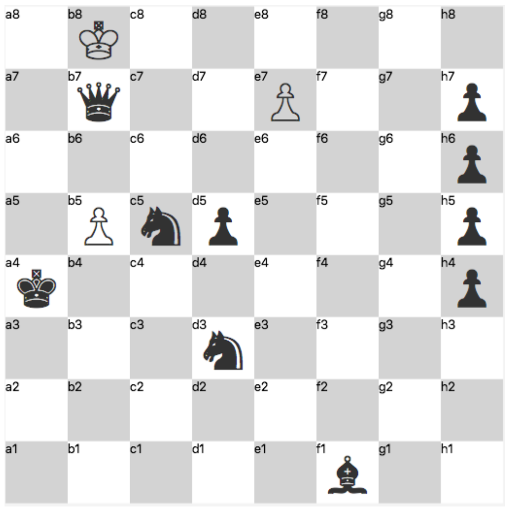
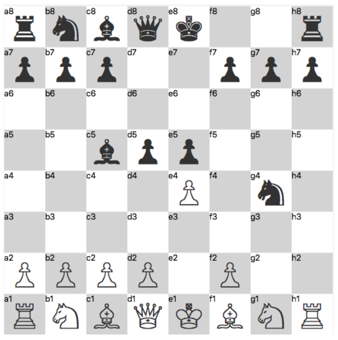
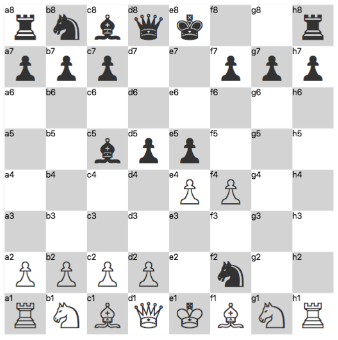
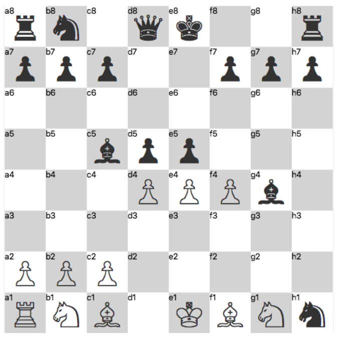
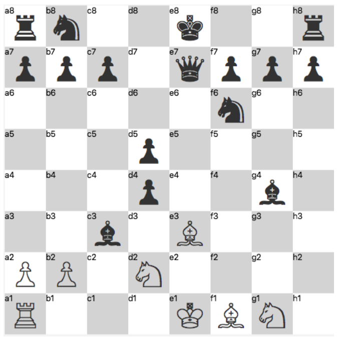

(f) An very difficult test case (solved by AlphaBetaAI)

Figure 2: Alpha-Beta search solves three increasingly difficult chess problems with a depth level of five. Each move took less than fifteen seconds.


(a) An intelligent (classical) opening


(b) A Well-Done fork F2


(c) Capturing pieces with tactics


(d) Building positional strength

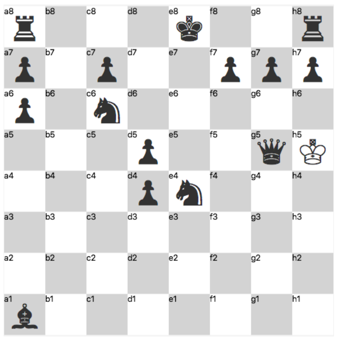

(e) Checkmate!

Figure 3: A full game between an AI using the intelligent evaluation algorithm (black) and an AI using the built-in getMaterial function. The difference in performance is striking, as the positional and tactical awareness shown by black seems almost human.