Chess AI

Niranjan Ramanand

October 24, 2016

1 Introduction

In this report, we discuss variations of the minimax algorithm applied to a chess AI. We use Chesspresso, a library created by Bernhard Seybold, for graphics and retrieval of certain board information.

2 General Setup

There are two AIs included in the file bundle: MMAI and ABAI, the AI's using normal minimax and Alpha-beta pruning, respectively. Note that the random evaluation function is commented out and discussed below. We replace it by a much more reasonable function based on material cost that is built into Chesspresso. The choices of who should play who can be made by adjust the following lines in ChessClient:

```
        ...
      moveMaker = new MoveMaker[2];
    moveMaker[Chess.BLACK] = new AIMoveMaker(new ABAI());//BLACK
    //oveMaker[Chess.WHITE] = new TextFieldMoveMaker();
    moveMaker[Chess.WHITE] = new AIMoveMaker(new ABAI());//WHITE
    ...
```

3 Minimax

By recursing through "minimum" and "maximum" states, minimax aims to minimize the maximum loss of a certain move. This means that, given a certain state, one descendant path from that state that generally loses slightly in the immediate future could triumph over another descendant path that is generally profitable in the immediate future, except for in one instance where it fails miserably. This is because the minimax algorithm assumes that both players play optimally, and so it does not matter how good a descendant path of actions may generally be, but rather how good is, specifically, its worst case scenario.

3.1 Naive, Random Minimax

The naivety in this minimax comes from its random evaluation function. While the evaluation function makes some sense in preferring wins to losses, it does not take any stance on game states that are not either. This leads to random sort game play, as even if a pawn could capture a queen, it may not choose to do so because a checkmate may not be in immediate view.

We use iterative deepening, allowing us to perform successive minimax calculations for a single move. The benefit in doing this is that we can choose to exit early on if we find a winning move, without needing to carry out calculations where currMaxDepth takes on values between zero and MAX DEPTH. When we have seen iterative deepening used in DFS, we note that no visited nodes were stored. We circumvent this in minimax by move reordering (discussed later briefly).

We use recursion to traverse our game tree by performing a move, passing the new position back into the method, and then undoing the move, resulting in our original board position. This saves a considerable amount of space by not creating new positions too often.

```java
  public short minimaxDecision(Position pos1) {
    Position pos = new Position(pos1);

    short bestMove = Short.MIN_VALUE;
    short moveValue = Short.MIN_VALUE;

    short[] moves = pos.getAllMoves();

    try {
      for (short move : moves) {
        depth = 0;
        pos.doMove(move);
        short value = minValue(pos);

        if (value > moveValue) {
          bestMove = move;
          moveValue = value;
        }
        pos.undoMove();

      }
    } catch (Exception e) {
      e.printStackTrace();
    }
    return bestMove;
  }
```

```java
public short minValue(Position pos1) {
    if (hash && transTable.containsKey(pos1.getHashCode())
        && transTable.get(pos1.getHashCode()).entryDepth > (currMaxDepth - depth)) {
      saved++;
      return transTable.get(pos1.getHashCode()).util;
    } else {
      explored++;
      Position pos = new Position(pos1);
      short util = Short.MAX_VALUE, max;
      short[] allMoves;

      if (pos.isMate()) {
        return (pos.getToPlay() == turn ? MIN_UTIL : MAX_UTIL);
      } else if (pos.isStaleMate()) {
        return DRAW_UTIL;
      } else if (depth < currMaxDepth) {
        allMoves = pos.getAllMoves();

        try {
          for (short move : allMoves) {
            pos.doMove(move);
            depth++;
            max = maxValue(pos);
            util = util > max ? max : util;
            pos.undoMove();
            depth--;
          }
        } catch (IllegalMoveException e) {
          e.printStackTrace();
        }
      } else if (depth == currMaxDepth) {
          return (short) (new Random().nextInt(2*Short.MAX_VALUE) -
          Short.MAX_VALUE); //Change to meaningful
        //return eval(pos); // A meaningful eval
      }

      transTable.put(pos1.getHashCode(), new Entry((currMaxDepth - depth), util));
      return util;
    }
  }
  }
```

With this implementation, a max depth of 4 seems to be the highest achievable depth that does not slow the program down an undesirable amount (i.e. hard to wait for).

3.2 Alpha-beta pruning

We can think of alpha as an infimum and beta as a supremum. The idea when used with minimax is that it can prevent the evaluation of game states that have no likelihood of being picked by an optimal player. Let us say that we have a state that is trying to find a minimum child node. When its children are being evaluated for value, they, in turn, will try to find the maximum of their children's values. Let us say that the first child of the original node is evaluated to be something very small. Now, we look at the second child and notice that its children include a value that is higher than the evaluation of the first child of the original node. We do not need to bother with this second child any more, as we know that the optimality with which these players are playing prohibits any part of this sub-minimax-tree from being traversed. We really only change the minValue and maxValue methods to enact such changes in the MMAI, creating ABAI.

```
public short minValue(Position pos1, short a, short b) {
  if (hash && transTable.containsKey(pos1.getHashCode())
      && transTable.get(pos1.getHashCode()).entryDepth > (currMaxDepth - depth)) {
    saved++;
    return transTable.get(pos1.getHashCode()).util;
  } else {
    explored++;
    Position pos = new Position(pos1);

    short util = Short.MAX_VALUE, max;
    short[] allMoves;

    if (pos.isMate()) {
      return (pos.getToPlay() == turn ? MIN_UTIL : MAX_UTIL);
    } else if (pos.isTerminal()) {
      return DRAW_UTIL;
    } else if (depth < currMaxDepth) {
      allMoves = pos.getAllMoves();

      try {
        for (short move : allMoves) {
          if (b > a) {
            pos.doMove(move);
            depth++;
            max = maxValue(pos, a, b);
            util = util > max ? max : util;
            b = (b < util) ? b : util;
            pos.undoMove();
            depth--;
          } else {
            return b;
          }
        }
      } catch (IllegalMoveException e) {
        e.printStackTrace();
      }
    } else if (depth == currMaxDepth) {
      // return (short) (new Random().nextInt(2*Short.MAX_VALUE) -
      // Short.MAX_VALUE);

      return eval(pos);
    }

    transTable.put(pos1.getHashCode(), new Entry((currMaxDepth - depth), util));
    return util;
  }
}
```

3.3 Transposition Table and Move Reordering

Chesspresso includes a getHashCode() function that hashes a board setup, so that we can use the long value retrieved to map to the utility value of the board. We could also use position objects to map to the utility, but that would be an enormous waste of space considering all the data packed into one position object. Thus, we construct a HashMap with the desired domain/co-domain. Some care needs to be put in when to access this HashMap, as it might initially seem that when the board reaches a certain configuration, all that we need to do is store the value. What this leaves out is that the utility function is depth-dependant, and so we will want the utility in the map to be the one that most closely resembles the actual utility (the deepest). This is the same as saying that, if we run minimax on a position with depth x and depth y, x being greater than y implies that the utility of the first minimax run is more accurate. This is why (currMaxDepth - maxDepth) makes a few appearances. This latter mechanism, called move reordering, is something that is omitted in typical iterative deepening and was something I came up with as a bonus to resolve the frustration of not using the information in our previous iterative deepening searches. Not storing information in ID-DFS is one thing, but there is no reason to have a lack of storage when we are already keeping a transposition table.

We make use of the following data structure as the co-domain type, which provides not only util, but also the entryDepth, allowing us to determine how accurate the entry really is and if it can become more accurate.

```
public class Entry {
  public int entryDepth;
  public short util;

  public Entry(int entryDepth, short util) {
    this.entryDepth = entryDepth;
    this.util = util;
  }
}
```

After every iteration of getMove(), we see how many states were explicitly computed (explored) and how many were retrieved from the table (saved) Iterative deepening is present in all of the AIs. He is its implementation in MMAI:

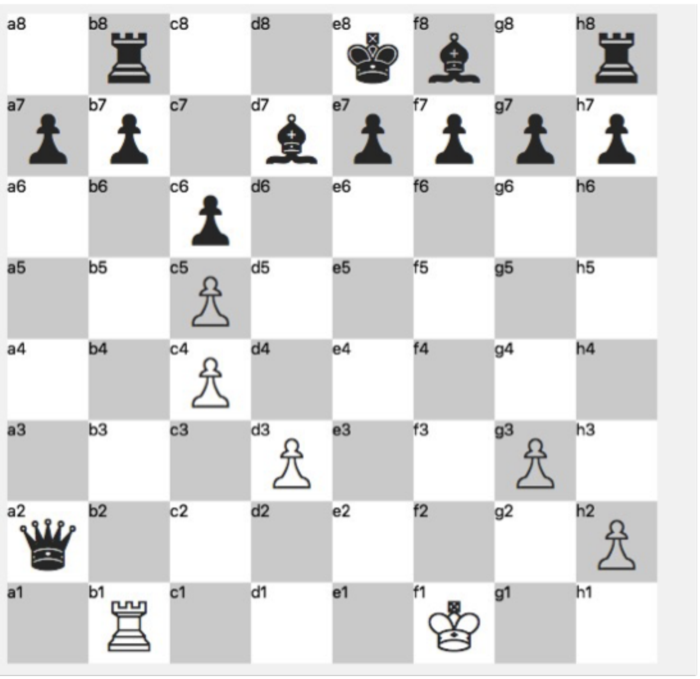
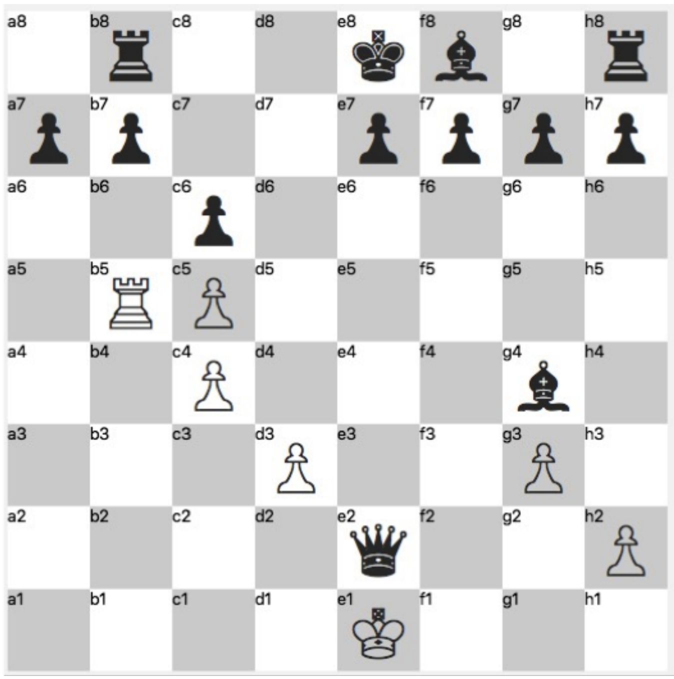
```
    @Override
public short getMove(Position position) {
  short nextMove = 0;

  turn = position.getToPlay();

  for (currMaxDepth = 1; currMaxDepth < MAX_DEPTH; currMaxDepth++) {
```

```java

      nextMove = minimaxDecision(position);

      try {
        Position temp = new Position(position);
        temp.doMove(nextMove);

        if (temp.isTerminal()) {
          System.out.println("MMAI Nodes Explored: " + explored);
          System.out.println("MMAI Nodes Saved: " + saved);
          explored = 0;
          saved = 0;
          return nextMove;
        }
      } catch (IllegalMoveException e) {
        e.printStackTrace();
      }
    }

    System.out.println("Max Depth: " + currMaxDepth);
    System.out.println("MMAI Nodes Explored: " + explored);
    System.out.println("MMAI Nodes Saved: " + saved);
    explored = 0;
    saved = 0;
    return nextMove;

  }
```

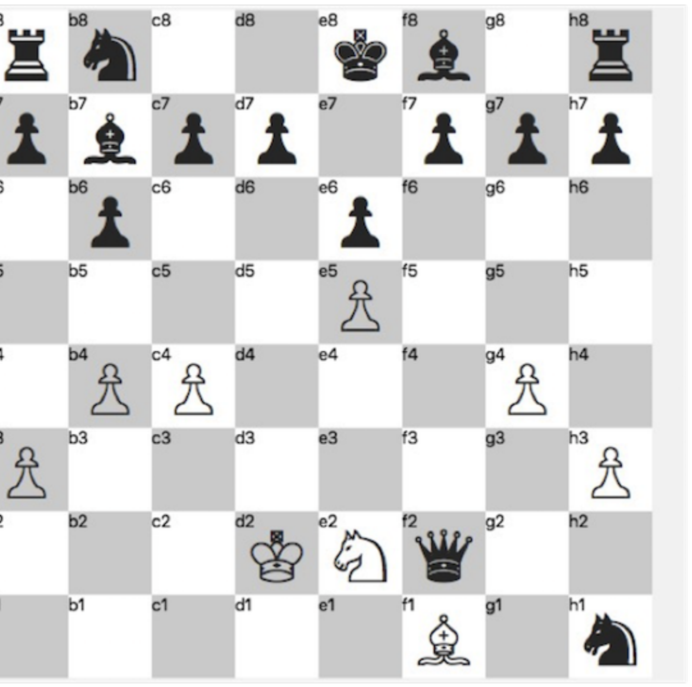
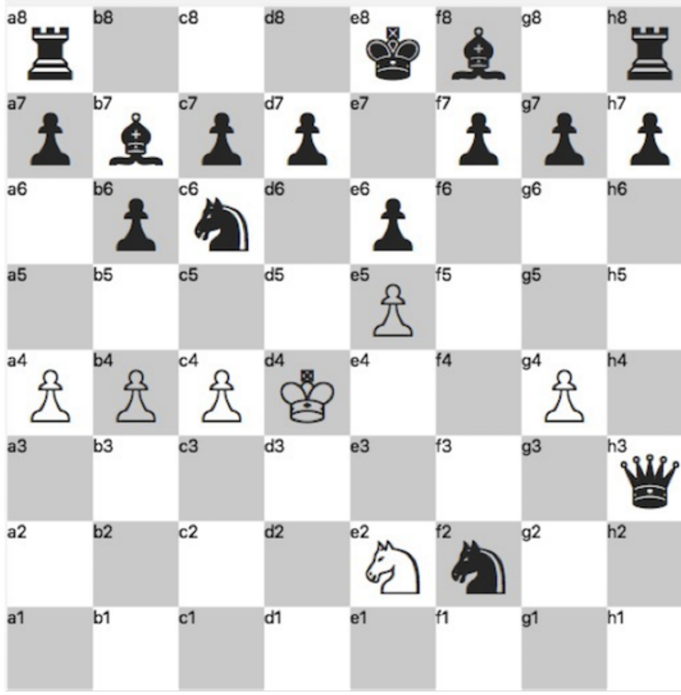
This getMove() function is run everytime the player's turn occurs. These threads are triggered to start through the GameHandler class in ChessClient.

Together with a transposition table and alpha-beta pruning, the improved algorithm is able to take on a maximum depth of 6. This is 2 more than the previous possible maximum depth!

Here are some pictures of ABAI of depth 6 working against ABAI (with slight randomness) of lower depth in various configurations. These checkmates may give some insight into the sort of complexity with which these algorithms operate.

ABAI Checkmate in 3 moves

ABAI Checkmate in 5 moves

4 Closing Remarks

The runs pitting various AI's built from MMAI will definitely draw when run with the same depth. Even when depth is different, a stalemate could result. This is because the AI that sees further may still think the option it keeps repeating is the best option. Certainly ABAI and MMAI are very similar, as ABAI, when operated at the same depth, will perform the exact same way as MMAI. In order to get a game to conclude without stalemate, it is necessary to either induce randomness in the game or have one AI see sufficiently and significantly further ahead. The latter is just theoretical, as even with ABAI's barely feasibly computable MAX DEPTH of 6, the difference is not sufficient.

Also, the algorithm mainly evaluates states based on material cost. This is unfortunate when it comes to actually spread out pieces in the beginning. There is no incentive for the AI to do so, as no pieces can be taken. Therefore it engages in repetitive behavior until something can lead to a difference in material cost.