My First Chess Engine

Nicholas Fiacco

February 9, 2016

1 Introduction

Chess has long stood as one of the most troublesome AI problems to receive a lot of media attention. Here, we will explore some of the progress made towards potentially finding a solution, and in doing so we will demonstrate some clever ways of searching a decision tree in which two agents make alternating moves. One of the most ubiquitous approaches to this problem is known as Minimax, which seeks to maximize the upside of one agent, our hero, while minimizing the best possible upside of the adversary. However, before we can discuss Minimax we must first understand the representation of the state space corresponding to the game of chess.

1.1 State Space

The most straightforward way to represent the various states in a chess game are to distinguish each unique arrangement of pieces as a single "position" and let the successor states be all other positions reachable by moving a single piece in a legal manner. However, it is highly important to note that we must also keep track of the player who's turn is up, since two otherwise identical positions may indeed have different successors depending on this factor. Thus, with 64 locations on the board and any subset of 32 pieces, in addition to the two possibilities for which color is allowed to move, there are very large number of states and typical search is ineffective. The number is something like 10^{43} based on a reputable source (Wikipedia), and knowing that the average branching factor is about 30 tells us that A* is doomed.

2 Minimax

The concept behind minimax is that you are searching for a position that maximizes the guaranteed utility of one agent while minimizing the best possible utility of the other. This utility can refer to anything; in our game of chess there are several factors that come into play. When we have reached a terminal position, one in which there is a checkmate or stalemate, we attribute a utility corresponding to the outcome. Intuitively, a victory is `Integer.MAX_VALUE`, loss is the negation of this, and stalemate is 0. The process of determining such a utility for a non-terminal position is discussed later in the section on evaluation functions. In order to use Minimax, we define two functions in addition to the namesake function that handles our initial request, called `getMinValue` and `getMaxValue` which serve to determine the minimum or maximum value of the successors from a particular position. This code can be found in `MinimaxAI.java`, and since you probably have a good idea of what it looks like, I have chosen not to include it in the report. However, I would like to draw attention to the manner in which I make the move and undo it, cleanly wrapped around each recursive call as such:

```
// minimize the value of the user's moves
position.doMove(move);
min = Math.min(min, getMaxValue(position, depth-1));
position.undoMove();
```

Furthermore, I do not use an express function to determine the utility of terminal positions, since it would be necessary to take into account which color made the last move. By handling this directly in the min/max functions, it is obvious who made the last move depending on which of the two functions we are in. In addition to this basic implementation, we use an iteratively deepening wrapper that stops the second a solution, or victory checkmate, is found. Nothing too interesting here either. One bug that I think some people saw when they were actually able to checkmate the AI is that there was an error when the AI tried to find a move when there were indeed none. A simple check to see if the length of the move list was empty rectified this issue.

3 Evaluation Function

We define a simple evaluation function based on the pieces remaining, with each piece accorded value based on traditional knowledge of the game. Here is our code for determining this value, a veritable evaluation function based on material:

```
for(int r = 0; r < 8; r++){
    for(int c = 0; c < 8; c++){
        sqi = Chess.coorToSqi(c, r);
        color = 1;
        stone = position.getStone(sqi);

        // utility is negative for each of the opponent's pieces
        if((stone < 0 && ai_color == 1) || (stone > 0 && ai_color == 0)){
            color = -1;
        }

        // based on the piece type, sum up the utilities
        if(Math.abs(stone) == Chess.PAWN){
            material += (100 * color);
        }
        if(Math.abs(stone) == Chess.KNIGHT){
            material += (320 * color);
        }
        if(Math.abs(stone) == Chess.BISHOP){
            material += (330 * color);
        }
        if(Math.abs(stone) == Chess.ROOK){
            material += (500 * color);
        }
        if(Math.abs(stone) == Chess.QUEEN){
            material += (900 * color);
        }
    }
}
```

Also, I created an evaluation function using the FEN representation of the position. For the record, it was no faster despite the fact that at most around 34 characters are iterated over instead of looking at all 64 squares on the board. Here is that code as well (note that it is a few lines longer):

```
String fen = position.getFEN();
for(int i = 0; i < fen.length(); i++){
    char c = fen.charAt(i);
    // space character means you have reached the end of the line
    if(c == ' '){
        break;
    }

    // black is lowercase in FEN and the constant in chesspresso for
    // black is 1. the opposite of both statements is true for white
    if((Character.isLowerCase(c) && ai_color == 0)
    || (Character.isUpperCase(c) && ai_color == 1)){
        // utility is negative for each of the opponent's pieces
        color = -1;
    }
    else{
        color = 1;
    }

    // based on piece type, sum up the utilities
    if(c == 'p' || c == 'P'){
        material += (100 * color);
    }
    if(c == 'n' || c == 'N'){
        material += (320 * color);
    }
    if(c == 'b' || c == 'B'){
        material += (330 * color);
    }
    if(c == 'r' || c == 'R'){
        material += (500 * color);
    }
    if(c == 'q' || c == 'Q'){
        material += (900 * color);
    }
}
```

4 Alpha-Beta Pruning

With this technique, it is possible to significantly decrease the number of positions evaluated by the Minimax algorithm. The main idea is that within each `getMaxValue` call, there is a current maximum value, alpha, associated with the best move M. When this function calls `getMinValue` on another move M' to get its minimum value, we know that this value will only matter if it exceeds alpha. Thus, if one of the possible successors to M' has a value V less than alpha, we know that

the highest possible value for M' will be V since we take the minimum of these values. Therefore, we can stop evaluating M' immediately. The reverse is used to lop off branches pertaining to moves that result in a lowest maximum value exceeding a value beta from the `getMinValue` call one frame above it in the recursion stack. Alpha-Beta pruning proves to be highly effective if implemented correctly, and we augment our Minimax algorithm to include parameters for alpha and beta that are updated periodically and used as a window for future moves. Here is the bulk of the added code (the opposite for `getMaxValue` of course):

```
// update values
if(min < beta){
    beta = min;
}
if(min <= alpha){
    nodesPruned++;
    break;
}
```

As you can see, it did not take more than a few added parameters and checks to significantly improve the algorithm's runtime. Indeed, we implemented an iterative deepening version of this as well.

5 Transposition Table

Now, we will describe one of the trickier aspects of improving the original Minimax algorithm. The reader will notice that many of the positions explored may have been previously discovered from some other sequence of moves, especially in the iterative deepening version. Thus, it might seem effective to store the value associated with this position in some sort of lookup table of transpositions, which refers to identical positions reached through different means. However, it is clear that not all values are created equally. As such, the value associated with a node that is at the maximum depth is determined via the evaluation function and thus is a direct reflection of this node by itself, without regard to its successors.

The astute reader will notice that it is more valuable to have a value (semantics, apologies) that is determined through multiple levels of subsequent recursion. Since depth is kept track of in reverse (the first ply explored has a depth of max depth and we decrement for future plies), it is trivial to determine the "quality" of a value, which we will store in this transposition table as well. Furthermore, it is possible that we have values associated with an upper or lower bound on the value of a position if we choose to prune the rest of the branch using the Alpha-Beta method. Thus, it will only be useful if this upper/lower bound can be used to prune future encounters with this position, since it is unwise to use it directly as a value. Here is an example of adding the upper bound of a position to the transposition table:

```
// make sure we are still in the window, otherwise we have an upper bound
if(min <= alpha){
    // add as an upper bound
    transposition.put(position.getHashCode(),
                      new Entry(depth, min, Entry.UPPER));
    return min;
}
```

Here is some of the associated code for retrieving values from the transposition table and determining their usefulness:

```
// try to get existing value from transposition table
if(transposition.containsKey(position.getHashCode())){
    // we only want high quality values, depends on how much depth is left
    Entry entry = transposition.get(position.getHashCode());
    if(entry.getQuality() >= depth){

        // we can just use the exact value
        if(entry.getBound() == Entry.EXACT){
            transpositionUsed++;
            return entry.getValue();
        }

        // if we found an upper bound that is less than alpha, we can prune
        if(entry.getBound() == Entry.UPPER && entry.getValue() <= alpha){
            transpositionUsed++;
            return entry.getValue();
        }
    }
}
```

6 Testing

In order to ensure the quality of the various engines that I built, I personally played several games and lost miserably to each of them. In no case was I able to take a piece with ease, and there were never "stupid" moves by the engine, even the basic Minimax version. I also ran extensive tests to ensure that given the same sequence of moves by me, each of the engines responded identically, and showed identical values. It is also clear that with each improvement the number of nodes explored to reach this identical move was lessened. Finally, I tested all of the end games with less than 4 moves required and it was clear that the solution was found. Two test cases I found particularly effective for testing were:

8/8/8/8/8/k1B5/BN6/K7 w - - 0 1 *// white wins in 6*
r5k1/p3Qpbp/2p3p1/1p6/q3bN2/6PP/PP3P2/K2RR3 b - - 0 1 *// black wins in 3*

Even with these tricky situations, all of my engines produced the right result, in the same manner. It is important to note that iterative deepening wrappers are necessary for the accurate determination of the optimal sequence of moves. Knowing that each engine performed identically to the basic Minimax is key, since there aren't many moving parts to the original implementation and thus it was far easier to ensure that this version was producing correct results, then progressively cross-checking with each of the other versions.

7 MTD(f)

This method augments the version of transposition table enhanced Alpha-Beta pruning engine. The basic idea is that an initial value f is fed into the search and a window of "zero" size is centered around this initial value. Zero-window essentially means that any value returned by the Minimax search will be a new upper or lower bound outside the alpha/beta bounds used by the pruning method. Thus, it will quickly prune large segments of the tree and return a new focus point with which the algorithm can adjust the center of the window. This process is iteratively continued in order to tighten the upper and lower bounds until they converge. Here is the important code used to implement the algorithm, using the original Alpha-Beta pruning implementation of Minimax with a transposition table:

```
while(lowerbound < upperbound){
    // the window should be targeted above the lower bound
    if(g == lowerbound){
        beta = g + 1;
    }
    else{
        beta = g;
    }

    result = TransAlphaBeta(position, beta-1, beta, MaxDepth);
    g = result.getValue();

    // adjust the bounds of the window
    if(g < beta){
        upperbound = g;
    }
    else{
        lowerbound = g;
    }
}
```

In order to ensure that the values and moves returned by this algorithm were accurate, we employed the same testing methods as before. Furthermore, we see a time improvement over the Alpha-Beta pruning with a transposition table in most cases. This implementation uses as an f value the previously found value associated with the search two plies earlier. This is due to the fact that there are oscillations in the value determined between searches of even depth and those of odd depth, and in order to speed up MTD(f) it makes sense to use the more accurate value. Here is the code demonstrating this, from the iterative deepening calls to MTD(f):

```
// oscillation between even and odd depths is handled
if(i % 2 == 0){
    result = MTDF(position, secondGuess, i);
    secondGuess = result.getValue();
}
else{
    result = MTDF(position, firstGuess, i);
    firstGuess = result.getValue();
}
```

You might notice that the `TransAlphaBeta` call has been modified to accept initial alpha and beta values, and that it returns a "result." This object is specified below and enables the function to return the associated value as well as the move itself:

```
private class Tuple{
    private short move;
    private int value;
    private Tuple(short move, int value){
        this.move = move;
        this.value = value;
    }
    private short getMove(){ return move; }
    private int getValue(){ return value; }
}
```

8 Profiling

When I noticed that the transposition table enhanced version of Alpha-Beta was actually slower than the original despite fewer nodes being explored, I decided that the best way to find the root of the issue was to profile. I found that a large amount of time was being spent in the evaluation function, since I was adding the value of these nodes to the transposition table and obviously every leaf position (a position at the max depth) would be added, and checked for existence. This had marginal value, since the evaluation function itself was pretty quick, and there wasn't any possibility that a non-leaf position would be pruned since the quality of this value would be the lowest possible. Thus, there was little benefit to adding evaluation function values to the transposition table, and I commented out this code. You can see the remnants of this in `TransAI.java` and `MTDFAI.java`.

9 Move Ordering

Rather than ordering every single possible move, which would require sorting and thus take a little bit of time, possibly more than it would save, I decided to use a killer heuristic that saves the last two moves that caused a cutoff at each depth. It is easy to save the killer moves for each depth, simply store an object that contains both moves in a list with the depth as an index. Here is the code for such adding moves to this object:

```
private void addMove(short move){
    // don't add duplicates
    if(move != first && move != second){
        if(last){
            first = move;
        }
        else{
            second = move;
        }
        // flip the value so next time the other is updated
        last = !last;
    }
}
```

This is accompanied by storing the move whenever a cutoff is found, and the code for this is relatively simple. However, another interesting aspect of this heuristic is figuring out how to try these moves first. I ultimately decided to check the list of all possible moves from a certain position and check if the killer moves at this depth were present in the list. If so, I swapped them up to the front of the list. Here is that code:

```
short killer1 = killerMoves[depth].getFirst();
short killer2 = killerMoves[depth].getSecond();

// make sure they aren't both negative
if(killer1 >0 && killer2 > 0){
    int index = 0;
    for(int i = 0; i < moves.length; i++){
        // move the killer move to the front
        if(moves[i] == killer1){
            short temp = moves[index];
            moves[index] = moves[i];
            moves[i] = temp;
            index++;
        }
        // move the other killer to the front
        if(moves[i] == killer2){
            short temp = moves[index];
            moves[index] = moves[i];
            moves[i] = temp;
            index++;
        }
    }
}
```

This implementation of MTD(f) with a killer heuristic is even faster, and passes all the unit tests that were applied to previous engines. Many more nodes are pruned, and it returns identical solutions in much less time.

10 References

I don't really know how to cite the chesspresso docs or chessprogramming.com but they were both very helpful. Wikipedia should be an assumed reference these days. I also used the MTD(f) page, so here is that citation:

Plaat, A., 1997, "MTD(f), A Minimax Algorithm Faster than NegaScout", Available on the Internet at http://www.cs.vu.nl/ aske/mtdf.html