Chess

Mauricio Esquivel Rogel

October 24, 2016

1 Introduction

I implemented an iterative deepening minimax search with the following extensions: alpha-beta pruning, transposition table, null-move pruning, quiescence search with delta pruning, opening book, and move ordering.

2 Running the code

Open the project in Eclipse and run `ChessClient.java`. Unfortunately, I wasn't able to fix a compiling bug when you first add the project to Eclipse that tells you some library can't be found. To fix that, you just go to the `chai` package right-click menu, and then: **Build path** > **Configure Build Path...**> **Libraries** > Select **JRE System Library [Java**... > **Add Library...** > **JRE System Library** > Check **Workspace default JRE**... > **Finish** > **Apply**.

3 Opening book

As suggested in *Artificial Intelligence: A modern approach*, I decided that for the first 10 plys, the computer will actually attempt to make the moves directly out of some game from the opening-book. Here is the code for this:

```
// Get the first 10 moves from the opening book but stop if at
// any point the move is invalid or not quiescent, then start using
// AI
if (position.getPlyNumber() < 10 && !invalidMove) {
  opening.goForward();
  short move = opening.getNextMove().getShortMoveDesc();
  boolean empty = position.isSquareEmpty(Move.getToSqi(move)),
    sacrifice = false, capture = Move.isCapturing(move),
      recapture;

  position.doMove(move);
  opening.goForward();

  if (!position.isLegal() || !empty) {
    position.undoMove();
    invalidMove = true;
    return IDMinimaxSerach(position);
  }

```

```
169     Double v = eval(position, 0);
170
171     recapture = position.getAllCapturingMoves().length > 0;
172     sacrifice = recapture ? !capture : quiescenceSearch(position,
173       Double.NEGATIVE_INFINITY,
174         Double.POSITIVE_INFINITY, 0) <= v;
175
176     // check if the horizon effect makes this a bad move
177     if (sacrifice) {
178       position.undoMove();
179       invalidMove = true;
180       return IDMinimaxSerach(position);
181     }
182
183     position.undoMove();
184     return move;
185 }
```

As the code shows, there is a lot of move validation going on. Invalid moves are immediately rejected, and they trigger a normal search as soon as their detected in lines 163-167. On the other hand, non-quiescent moves are too risky to allow, so we also run a small Quiescent-Search (see 4.2.1) and resort to a full, normal search if necessary (lines 171-181).

4 Minimax

To show how I implement the search, I will split it into subsections for each extension:

4.1 Iterative-deepening

ID-Minimax-Serach runs the iterations of the search, and Minimax-Alpha-Beta-Serach starts the depth-limited Minimax search.

4.2 Alpha-beta pruning

Implemented from ***Artificial Intelligence: A modern approach***

The calls to Max-Value(position, d, maxDepth, a, b) and Min-Value(position, d, maxDepth, a, b) take alpha, a, and beta, b, parameters to recursively find that best possible solution up to the max-depth. Each method starts out by checking if the search should terminate with the cutoff test,

```
// CUTOFF test with quiescence search
    if (cutoffTest(position, d, maxDepth)) {
      Double value = eval(position, d), q;

      if (position.isStaleMate()) {
value = 0.0;
      } else if (position.isTerminal() && position.isMate()) {
value = position.getToPlay() == computerId ?
    Double.NEGATIVE_INFINITY : Double.POSITIVE_INFINITY;
      } else if
(Move.isCapturing(position.getLastMove().getShortMoveDesc()) &&
```

```
  (q = quiescenceSearch(position, Double.NEGATIVE_INFINITY,
    Double.POSITIVE_INFINITY, d)) < value) {
value = q;
      }

      return value;
    }
```

Here, given that the current position terminates the search, we check if this is a win, a draw, a loss, or simply the cutoff evaluation. This code is from Max-Value, but the Min-Value version is analogous.

4.2.1 Quiescence Search

Implemented from ***https://chessprogramming.wikispaces.com/Quiescence+Search***

In order to make sure that the *horizon effect* is not causing unnecessary sacrifices, we check every non-quiescent position and extend the search until we land on a quiescent position. In this implementation, I consider captures non-quiescent positions. To do this, Quiescnece-Search runs a reduced Minimax search with alpha-beta and delta pruning. Here is the code for the max part of the search (the min version is analogous):

```
// beta cutoff
if (standPat >= b) {
  return standPat;
}

for (int i = 0; i < capturingMoves.length; i++) {
  move = capturingMoves[i];

  position.doMove(move);
  counter = quiescenceSearch(position, a, b, d);

  v = Math.max(counter, v);

  position.undoMove();

  if (v >= b) {
    return v;
  }

  bigDelta = 975.0; // queen eval
  if (Move.isPromotion(move)) {
    bigDelta += 775.0;
  }

  // delta cutoff
  // if not near the end of the game
  if (v < 2000.0 && v < a - bigDelta) {
     return a;
  }

  a = Math.max(a, v);
```

```
544 }
```

We know we have a quiescent position when `standPat` $\geq$ `b`, so we keep searching until that happens. Although these trees can grow quite large as well, the size of the trees decreases as less pieces are available. So to make it more efficient, we ignore moves that are not good enough to make up for the loss of a queen in lines 532-541, which serves as a good parameter to evaluate captures. Near the end of the game, there are relatively few captures to be made, so we deactivate delta pruning at this point because it no longer makes sense.

4.2.2 Move ordering

Before exploring the child min moves and max moves in MAX-VALUE and MIN-VALUE, respectively, these moves are ordered in a convenient way to make the alpha-beta pruning more efficient. In MAX-VALUE, we sort the moves in decreasing evaluation values, and in MAX-VALUE, we do the opposite. Here is the code for the move ordering in MAX-VALUE:

```
297 // MOVE ORDERING
298 move = moves[0];
299 position.doMove(move);
300
301 while (evalValues.containsKey(position) && j < moves.length) {
302   orderedMovesHeap.insert(new
303   FibonacciHeapNode<EvalMove>(new EvalMove(move,
304   evalValues.get(position))));
305
306   position.undoMove();
307
308   if (j < moves.length) {
309     move = moves[j++];
310     position.doMove(move);
311   } else {
312     j++;
313   }
314 }
315
316 if (j > 1) {
317   if (orderedMovesHeap.size() == moves.length) {
318     canOrder = true;
319   }
320 }
321
322 position.undoMove();
```

The Fibonacci Heap takes care of the sorting, and here is its initialization:

```
comparator = new EvalMoveComparator(-1);
PriorityFibonacciHeap<EvalMove> orderedMovesHeap =
        new PriorityFibonacciHeap<EvalMove>(comparator);
```

The `comparator` instantiates `EvalMoveComparator` with an integer, 1 or -1, to represent increasing and decreasing orders, respectively. Once this is ready, the **while** loop in lines 301-314 inserts all possible moves into the fib-heap but only if they're already in the transposition table, `evalValues`, because otherwise, the ordering would actually hurt the efficiency of the search. If all necessary moves were found in `evalValues`, then lines 316-320 toggle the boolean `canOrder` to **true** so that in the actual alpha-beta pruning search we know if we can get the next move from the ordered heap or if we need to resort to normal ordering:

```
x = canOrder ? orderedMovesHeap.poll().getValue() : null;

move = canOrder ? x.getMove() : moves[i];
```

4.2.3 Null-move pruning

Implemented from ***https://chessprogramming.wikispaces.com/Null+Move+Pruning***

As a forward-pruning method, null-move pruning makes the search more efficient by simulating a `Move.NO_MOVE` from the current player and then seeing if even with that largely disadvantageous move, the search can still reach a satisfactory cutoff evaluation value (which would of course be to a shallower depth, since a null-move is equivalent to decreasing `maxDepth` by 1). If the search is able to accomplish this, then we know there's no point exploring a larger tree, so we just cutoff the search. This process gets done every recursion of both Max-Value and Min-Value to cutoff unnecessary trees, where the codes for the two methods are once again analogous to each other. Here is the code for null-move pruning in Max-Value:

```
324 // NULL-MOVE PRUNING
325 if (position.isCheck() && position.getToPlay() == computerId) {
326   v = Double.NEGATIVE_INFINITY;
327 } else {
328   position.setToPlay(0);
329   v = minValue(position, d+1, maxDepth, a, b);
330   position.setToPlay(computerId);
331 }
332
333 if (v > Double.NEGATIVE_INFINITY) {
334   if (v >= b) {
335     return v;
336   } else {
337     v = Double.NEGATIVE_INFINITY;
338   }
339 }
```

Since I wasn't able to make Chesspresso allow a `Move.NO_MOVE` for either player, I equivalently give the turn to the next player. Once the cutoff value for this shallower search has been obtained in line 329, we check if the value is good to cutoff the current search tree in lines 333-339.

4.3 Evaluation function with Transposition Table

Idea from ***Artificial Intelligence: A modern approach***

Here is the code for the evaluation function I'm using:

```
Double material;

```

```java
// TRANSPOSITION TABLE
if (evalValues.containsKey(position.hashCode())) {
  // repeated positions are less valuable
  material = evalValues.get(position.hashCode()) - PENALTY;
} else {
  // more weight on the material
  material = position.getMaterial() + position.getDomination()
    / RATIO;
  evalValues.put(position.hashCode(), material);
}

if (position.getToPlay() == computerId) {
  return material;
}

return material * -1;
```

Chesspresso conveniently provides the three key pieces for this evaluation:

1. The position's `hashCode`
2. The position's `getMaterial` method
3. The position's `getDomination` method

The first item provides the necessary keys to create the transposition table and thus drastically speed up running time. The last two items determine how valuable is a position, and I just simply ranked them according to what I considered more important. Since I consider defense more important than offense (mostly because the computer doesn't make the first move), I add only a fraction of the position's domination (how much has the player destroyed the other player) to the the position's material. Although I initially considered taking into account other factors such as the possibility of a castle, good pawn structure, etc. but I realized that it works well enough as it is.