Chess AI

Lexin Tang

October 25, 2016

1 Introduction

In this report, I will firstly describe the framework of chess AI, including iterative deepening, cutoff test and evaluation function. Then I will introduce the implementation of Minimax and Alpha-Beta Pruning algorithms, in which transposition table is used to improve search efficiency. Experiments are conducted to verify the correctness of the algorithm and I will compare different algorithms in terms of run-time efficiency.

2 Framework

2.1 Cut-off Test and Evaluation Function

A cut-off test is used to cut off further search once we have reached a terminal state or we have reached the specified maximum depth.

```
private boolean cutoffTest(Position position, int depth){
  return (position.isTerminal() || depth >= CUTOFF_DEPTH);
}
```

Once the cut-off test returns true, we stop searching froward and return the utility of the current state. If the chess AI wins the game, the utility is `Integer.MAX_VALUE`. If it loses the game, utility is `Integer.MIN_VALUE`. If neither cases happens, the utility of the current position is based on `position.getMaterial()` implemented in Chesspresso library. The code is shown below.

```
  private final static int WIN = Integer.MAX_VALUE;
  private final static int LOSS = Integer.MIN_VALUE;
  private final static int DRAW = 0;
  private int getUtility(Position position, int player){
  if (position.isMate()){
    if (position.getToPlay() == player){
      return LOSS;
    }
    else{
      return WIN;
    }
  }
  else if (position.isTerminal() || position.isStaleMate()){
    return DRAW;
  }
  else{
    return evaluateFunc(position, player);
  }
}

private int evaluateFunc(Position position, int player){
  if (position.getToPlay() == player){
```

```
      return position.getMaterial();
    }
    else{
      return -position.getMaterial();
    }
  }
```

2.2 Iterative Deepening

For a chess game, the number of states increases exponentially with depth. Iterative deepening search algorithm is useful to limit search within a range, which makes the computation of states feasible. The code is shown below.

```
  public short getMove(Position position) {
    Node bestMoveNode = null;
    Position pos = new Position(position);
    // iterative deepening
    for (int i = 1; i < MAX_DEPTH; i++){
      CUTOFF_DEPTH = i;
      resetStats();
      try {
        bestMoveNode = makeDecision(pos);

        printStats();
        if (bestMoveNode.utility == WIN){
          break;
        }
      } catch (IllegalMoveException e) {
        e.printStackTrace();
      }
    }

    short move = bestMoveNode.move;
    System.out.println("utility of best move = " + bestMoveNode.utility);
    return move;
  }
```

3 Minimax

The Minimax algorithm can be described in 3 parts.
a. `makeDecision()` returns the best move after calling the recursive function `minValue()`.
b. `minValue()` minimizes the worse-case outcome for Min.
c. `maxValue()` maximizes the worse-case outcome for Max.

```
  private Node makeDecision(Position position) throws IllegalMoveException{
    int player = position.getToPlay();
    int bestUtility = LOSS;
    short bestMove = 0;
    short [] moves = position.getAllMoves();
    for (int i = 0; i < moves.length; i++){
      position.doMove(moves[i]);
      int new_utility = minValue(position, player, 1);
      if (bestUtility < new_utility){
        bestMove = moves[i];
        bestUtility = new_utility;
      }
```

```
      position.undoMove();
    }

    return (new Node(bestMove, bestUtility));
  }

  // return a utility value
  private int maxValue(Position position, int player, int depth) throws IllegalMoveException {
    updateDepth(depth);
    incrementNodeCount();
    if (cutoffTest(position, depth)){
      int utility =  getUtility(position, player);
      return utility;
    }

    int result = LOSS;
    short [] moves = position.getAllMoves();
    for (int i = 0; i < moves.length; i++){
      position.doMove(moves[i]);
      result = Math.max(result, minValue(position, player, depth + 1));
      position.undoMove();
    }
    return result;
  }

  // return a utility value
  private int minValue(Position position, int player, int depth) throws IllegalMoveException {
    updateDepth(depth);
    incrementNodeCount();
    if (cutoffTest(position, depth)){
      int utility =  getUtility(position, player);
      return utility;
    }

    int result = WIN;
    short [] moves = position.getAllMoves();
    for (int i = 0; i < moves.length; i++){
      position.doMove(moves[i]);
      result = Math.min(result, maxValue(position, player, depth + 1));
      position.undoMove();
    }
    return result;
  }
```

4 Alpha-Beta Pruning

The algorithm consists of 3 parts and is very similar to minimax algorithm. The only thing different from minimax is that alpha-beta pruning is added into `minValue()` and `maxValue()`.
a. `makeDecision()` returns the best move after calling the recursive function `minValue()`.
b. `minValue()` minimizes the worse-case outcome for Min.
c. `maxValue()` maximizes the worse-case outcome for Max.

```
  private int maxValue(Position position, int player, int depth, int alpha, int beta) throws IllegalMoveException{
    updateDepth(depth);
    incrementNodeCount();
    if (cutoffTest(position, depth)){
      int utility = getUtility(position, player);
```

```
      return utility;
    }

    int result = LOSS;
    short [] moves = position.getAllMoves();
    for (int i = 0; i < moves.length; i++){
      position.doMove(moves[i]);
      result = Math.max(result, minValue(position, player, depth + 1, alpha, beta));
      position.undoMove();

      //alpha-beta pruning
            if(result >= beta){
                return result;
            }
            alpha = Math.max(alpha, result);
    }
    return result;
  }

    private int minValue(Position position, int player, int depth, int alpha, int beta)
    throws IllegalMoveException{
    updateDepth(depth);
    incrementNodeCount();
    if (cutoffTest(position, depth)){
      int utility = getUtility(position, player);
      return utility;
    }

    int result = WIN;
    short [] moves = position.getAllMoves();
    for (int i = 0; i < moves.length; i++){
      position.doMove(moves[i]);
      result = Math.min(result, maxValue(position, player, depth + 1, alpha, beta));
      position.undoMove();

      //alpha-beta pruning
            if(result <= alpha){
                return result;
            }
            beta = Math.min(beta, result);
    }

    return result;
  }
```

5 Transposition Table

A transposition table is used to reduce duplicate work when searching around the same state. It is implemented by java HashMap with key being the hash code of a position, and hash value being the utility and searched depth of the position. The code for minimax algorithm is added in function `minValue()` and `maxValue()`.

```
    long hashPos = position.getHashCode();
    if (transTable_flag && transTable.containsKey(hashPos) && transTable.get(hashPos).depth
   >= CUTOFF_DEPTH){
      return transTable.get(hashPos).utility;
    }

    incrementNodeCount();
    if (cutoffTest(position, depth)){
```

```java
        int utility =  getUtility(position, player);
              if (transTable_flag && (!transTable.containsKey(hashPos) || transTable.get(hashPos).depth < depth)) {
                transTable.put(hashPos, new TransTableNode(utility, depth));
              }
        return utility;
    }
```

6 Test Results and Discussion

6.1 MaxDepth and Time Efficiency

Here is an example of different max depth set for alpha-beta pruning algorithm. We could see from the figure that the chess AI needs fewer steps to win the game over a random player with the max depth increases.

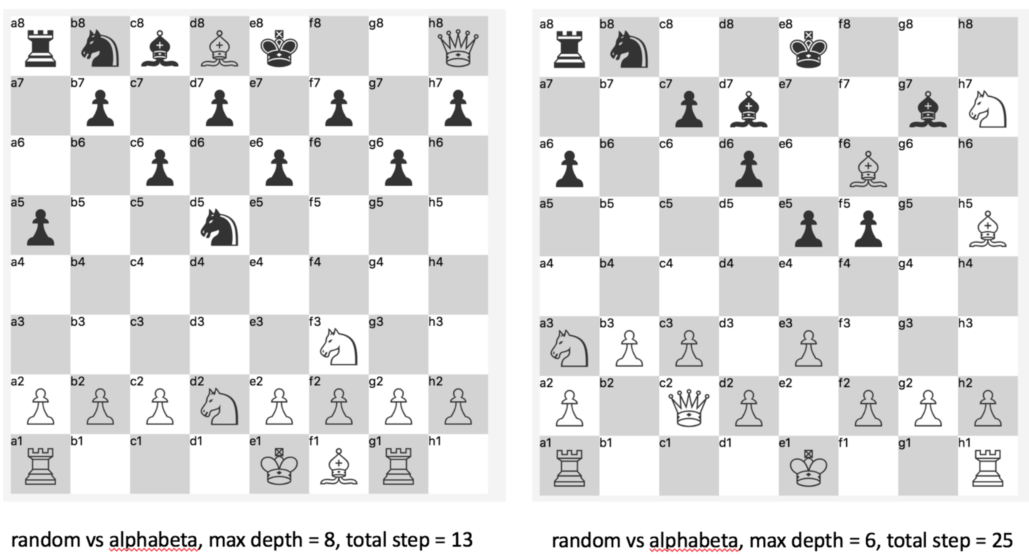

Figure 1: Example of different max depth for iterative deepening

6.2 The Use of Transposition Table

We set the maximum depth for iterative deepening to 7 and compare the number of nodes explored by Alpha-Beta algorithm with or without use of transposition table. The result is shown below. Obviously, transposition table helps to reduce nearly 80% duplicate work in this case.

```
// with transposition table:
Nodes explored during last search:  1912689
Maximum depth explored during last search 7
utility of best move = 100
making move 5249

// without transposition table:
Nodes explored during last search:  11151442
Maximum depth explored during last search 7
```

```
utility of best move = 100
making move 5249
```