Chess AI

Siyang Li

February 21, 2014

1 Introduction

Chess is a very commonly known and relatively complex game. Though it is complex, it is a perfect information game-all information about the game is known and determined (not random). Therefore, in theory, a computer algorithm can create a perfect decision game tree to decide how to ensure a perfect win. However, because there exists many different possible states of the game and because there exists many moves that can be made from each state, the game tree would grow exponentially large and computations would take an unfeasible amount of time. Because chess is a timed game, there needs to be faster algorithm which can compute a best move for the computer to do. The move may not ensure a win, but should play relatively "smart". This is the topic of our paper. In this paper, we will discuss first min-max algorithm and then alpha-beta pruning modified version of the min-max algorithm.

2 Min-Max Algorithm

The first algorithm we will look at is the min-max algorithm. This algorithm searches the tree to termination conditions using recursive depth first search. Termination condition is determined by the `cutoffTest()` which will be discussed later. The idea behind min-max algorithm is to go deep into the game decision tree via recursive DFS. Once a terminal leaf node is reached, a value is derived to resemble the "strength" of that position relative to player 1 (positive values if player 1 is winning and negative if player 2 is). Afterwards, that value is passed up the tree and compared to previous values passed up the parent node. At every even level (player 1's move), the maximum value of all the children nodes is recorded in each node. At odd levels (player 2's move), the minimum value of all the children nodes is recorded. This makes sense given the way our utility method is set up relative to player 1. Player 1 is therefore the maximizer and aims to make moves to maximize utility and win the game. Player 2 is minimizer and aims to make moves to minimize utility and win the game. Our recursive min-max function takes 3 different methods- `getMinValue()` method that recursively calls `getMaxValue()` at odd levels, `getMaxValue()` that recursively calls `getMinValue()` at even levels, and a `getMove()` method which starts the recursive function by calling previous methods.

```
//depth-limited get move function; takes in a depth and uses Min-Max algorithm
//to return best move
  public short getDepthLimitedMove(Position position, int depthLimit) {
    PositionDataNode positionNode = new PositionDataNode(position);

    short[] moves = positionNode.getPosition().getAllMoves();

    int startMaxValue = -2147483647;
    int tempMinValue;
    short bestMove = 0;

    for (int i = 0; i < moves.length; i++) {
```

```
      try {
        positionNode.getPosition().doMove(moves[i]);
        positionNode.incrementDepth();
        }

        catch (IllegalMoveException exception) {
          System.out.println("IllegalMoveException");
        }

      tempMinValue = getMinValue(positionNode, depthLimit);

      if (startMaxValue < tempMinValue) {
        startMaxValue = tempMinValue;
        bestMove = moves[i];
      }

      positionNode.getPosition().undoMove();
      positionNode.decrementDepth();
    }

    return bestMove;
  }
```

This is the `getMove()` method which iterates through all possible moves from the current position. It calculates the minMaxValue for each of the children nodes and returns the maximum of these values as the best move for the current player (the maximizer).

```
//getMaxValue method; recursively calls getMinValue method; one of two
//components for the min-max algorithm
  public int getMaxValue(PositionDataNode positionNode, int depthLimit) {
    if (cutoffTest(positionNode, depthLimit)) {
      return utilityCalculator(positionNode.getPosition());
    }

    int max = -2147483647;

    short[] moves = positionNode.getPosition().getAllMoves();

    for (int i = 0; i < moves.length; i++) {

      try {
        positionNode.getPosition().doMove(moves[i]);
        positionNode.incrementDepth();
        }

        catch (IllegalMoveException exception) {
          System.out.println("IllegalMoveException");
        }

      max = Math.max(max, getMinValue(positionNode, depthLimit));

      positionNode.getPosition().undoMove();
      positionNode.decrementDepth();
```

```
    }

    return max;
  }
```

This is the `getMaxValue()` method and is called upon at even levels while recursively searching our game tree. This is the move of the maximizer (player 1). The method goes through all possible moves from that position in the tree. For each move, it calls upon `getMinValue()` to find the value that the opponent (minimizer) would have had if that move were chosen. The maximum value out of all these values is returned. This is the recursive case. The base case tests if the position is a terminal state (either reached depth limit or game is over with checkmate/stalemate). It does so by calling the `cutoffTest()` method. If the position is terminal, the utility is returned. The utility is calculated via the `utilityCalculator()` method.

```
//getMinValue method; recursively calls getMaxValue method; one of two
//components for the min-max algorithm
  public int getMinValue(PositionDataNode positionNode, int depthLimit) {
    if (cutoffTest(positionNode, depthLimit)) {
      return utilityCalculator(positionNode.getPosition());
    }

    int min = 2147483647;

    short[] moves = positionNode.getPosition().getAllMoves();

    for (int i = 0; i < moves.length; i++) {

      try {
      positionNode.getPosition().doMove(moves[i]);
      positionNode.incrementDepth();
      }

      catch (IllegalMoveException exception) {
        System.out.println("IllegalMoveException");
      }

      min = Math.min(min, getMaxValue(positionNode, depthLimit));
      positionNode.getPosition().undoMove();
      positionNode.decrementDepth();
    }

    return min;
  }
```

This is the `getMinValue()` method that is called upon at odd (player 2/minimizer) levels of the game tree. The method works like the `getMaxValue()` method except that it recursively calls `getMaxValue()` and returns the minimum value for all possible moves.

3 Cut-off Test

Because we cannot, in reasonable time, search the entirety of the tree to all terminal states, we must have some sort of cut-off test which allows us to know when to stop going down via DFS. This is our `cutoffTest()` method.

```
//cut-off test to check if search/evaluation should be stopped at node
//returns true if either (1) maximum depth is reached or (2) game is over
  public boolean cutoffTest(PositionDataNode positionNode, int depthLimit) {
    return (positionNode.getDepth() >= depthLimit ||
     positionNode.getPosition().isTerminal());
  }
```

The method returns true (meaning we cut-off the search) if the depth limit is reached or if the game is over in stalemate or checkmate (as determined by `isTerminal()` method of `Position.java` class.

4 Utility Evaluation

At terminal states, we must calculate a utility or "worth" of that state to determine which state is stronger for the minimizer/maximizer. This task is done by our `utilityCalculator()` method which calls upon `evaluatePosition()` method.

```
//returns utility of state; if terminal state, returns high MAX, MIN or 0; if
//not terminal state, return an evaluated value
  public int utilityCalculator(Position position) {

    if (position.getToPlay() == Chess.WHITE) {
      if (position.isMate()) {
        return -2147483647;
      }

      if (position.isStaleMate()) {
        return 0;
      }

      else {
        return evaluatePosition(position);
      }
    }

    if (position.getToPlay() == Chess.BLACK) {
      if (position.isMate()) {
        return 2147483647;
      }

      if (position.isStaleMate()) {
        return 0;
      }

      else {
        return -evaluatePosition(position);
      }
    }

    return 0;

  }
```

The `utilityCalculator()` method uses if-statements to determine whether or not a checkmate has been reached. If a checkmate has been reached and it is the minimizer/player 2's turn, then the maximizer has won. We can return a value that is highly positive (positive maximum int value). Likewise, if checkmate has been reached and it is the maximizer/player 1's turn, then we return the negative minimum int value. If there is a stalemate, 0 is returned to indicate tie. If the state is not an end-game state, we must call upon our `evaluatePosition()` method to give an estimate heuristic analysis of the strength of that position.

This is our `evaluatePosition()` method shown below:

```
//evaluates a utility for non-terminal nodes based on material value;
//also takes into consideration check positions
  public int evaluatePosition(Position position) {
    int materialValue = position.getMaterial();

    if (position.isCheck()) {
      materialValue = materialValue - 500;
    }

    return materialValue;
  }
```

The `evaluatePosition()` method calls upon `getMaterial()` from `Position.java` to get an estimated strength value based on material value of the position. The material value of the position is determined by taking the pieces of the current player, assigning a value to each (pawns, knights, rook, bishop, and queen are all worth different values respectively), and adding them together. The pieces and total value of pieces of the opponent are subtracted from current player value. Our \verbutilityCalculator‘ method takes into consideration this player-dependent value and negates the return value of `evaluatePosition()` if it is the minimizer's turn. In addition, our `evaluatePosition()` method has been modified to not only take into consideration of the material value of the position. In addition, certain configurations of the pieces such as that of a check are beneficial as they force opponent to not be able to move. This is factored into our `evaluatePosition()` method by subtracting from material value if check is detected (since check on your turn is negative).

5 Iterative Deepening Search

Our above min-max search is a depth-limited DFS search. However, we might like to make an iterative deepening search which searches to a certain depth by searching previous depths 1–¿set depth. If the search is interrupted early (time runs out), the bestMove from the deepest search so far can be returned. Although iterative deepening search can take longer, it runs less risk of not finding a move in time.

```
//getMove method that uses iterative deepening and our depth-limited get move
//function; if time ever runs out, best move found so far is returned
  public short getMove(Position position) {
    short bestMove = 0;
    boolean timeOut = false;

    for (int i = 1; i <= iteratedDepthLimit; i++) {
      bestMove = getDepthLimitedMove(position, i);

      if (timeOut) {
        return bestMove;
      }
    }
```

```
    return bestMove;
    return getDepthLimitedMove(position, iteratedDepthLimit);
  }
```

The implementation for our iterative depth min-max search is simple. We call the depth limited min-max search successively from depths 1 to a set depth. Each time, we replace bestMove with a new bestMove (assuming with more information, we get better move). If at any time, timeOut = true, we return the bestMove. Since we do not have a timer, timeOut will never equal true. This could be easily implemented though.

6 Min-Max Results and Discussion

In the end our min-max algorithm along with our utility evaluation was able to play relatively "smart" moves which always won significantly against the RandomAI. `Figure 1` (shown below) shows the end result for a chess game between our MinMaxAI and RandomAI. The depth was set at 5. As shown, the MinMaxAI (white) was able to force the RandomAI into checkmate. This means that both our `cutoffTest()` and `evaluatePosition()` were effective at guiding the AI to make smart choices. It should be noted that the algorithm did take a long (5 seconds) time for each move. This is because it had to search a large amount of nodes for each move. Print statements showed that each decision searched around 500k to 900k nodes.

Next, we varied the depth of the min-max algorithm to see depth's impact on decision making and run-time. In `Figure 2` (shown below), a min-max test run with depth 3 is shown beow. The impact on run-time was obvious. Run-time was much faster (1 second) and the nodes explored decreased greatly to 5k-20k nodes. This is because the each position has many moves and the game tree expands exponentially. The difference between a tree at depth 3 and depth 5 is great. The impact on decision making with varying depth was not as apparent. It seemed that decreasing the depth increased the number of moves needed for the win. This makes sense as the AI often fails to see ahead to winning positions and might make a move away from a win. However, it is also noted that at lower depths, the AI seems to be better at protecting its own pieces (we see that less pieces were lost). This might be because the opponents moves were unpredictable and a look ahead to far might have overlooked more immediate threats and so a further depth was counter-productive. Either way, the results were interesting.

In the end however, it is clear that our MinMaxAI was able to force a checkmate and does not lose pieces or chances to capture pieces stupidly.

7 Alpha-Beta Pruning Algorithm

Although the min-max algorithm is effective at reducing the need to search the entire tree to all end-game states while still maintaining relatively smart game play, it does search many unnecessary branches that could have been "pruned" from the search. Alpha-beta pruning algorithm is one approach to removing from the search a large portion of the game decision tree.

The idea behind alpha-beta pruning is that if a move has a successor move/branch which (1) returns a utility value greater than the min value of its minimizer parent (current position is maximizer) or (2) returns a utility value less than the max value of its maximizer parent (current position is minimizer), then all other moves made from the current position do not need to be searched as the parent would never have selected that node to begin with. Those sections of the game tree can be pruned. The modified min-max algorithm with alpha-beta pruning is shown below:

```
//depth-limited get move function; takes in a depth and uses Min-Max algorithm
```

```
//to return best move
//Modified for alpha-beta pruning
public short getDepthLimitedMove(Position position, int depthLimit) {
    PositionDataNode positionNode = new PositionDataNode(position);

    short[] moves = positionNode.getPosition().getAllMoves();

    int startMaxValue = -2147483647;
    int tempMinValue;
    short bestMove = 0;

    int alpha = -2147483647;
    int beta = 2147483647;


    for (int i = 0; i < moves.length; i++) {
      try {
        positionNode.getPosition().doMove(moves[i]);
        positionNode.incrementDepth();
        }

        catch (IllegalMoveException exception) {
          System.out.println("IllegalMoveException");
        }

      tempMinValue = getMinValue(positionNode, depthLimit, alpha, beta);

      if (startMaxValue < tempMinValue) {
        startMaxValue = tempMinValue;
        bestMove = moves[i];
      }

      positionNode.getPosition().undoMove();
      positionNode.decrementDepth();
    }

    return bestMove;
  }
```

The `getMove()` method with alpha-beta pruning is very similar to the `getMove()` method for regular min-max algorithm. The only thing that changed is that alpha-beta versions of `getMinValue()` is called. The method still iterates through all moves and returns the one that gets the highest min-max value.

```
//getMaxValue method; recursively calls getMinValue method; one of two
//components for the min-max algorithm
//modified for alpha-beta pruning

public int getMaxValue(PositionDataNode positionNode, int depthLimit,
int alpha, int beta) {
    if (cutoffTest(positionNode, depthLimit)) {
      return utilityCalculator(positionNode.getPosition());
    }
```

```
    int score;

    short[] moves = positionNode.getPosition().getAllMoves();

    for (int i = 0; i < moves.length; i++) {

      try {
        positionNode.getPosition().doMove(moves[i]);
        positionNode.incrementDepth();
        }

        catch (IllegalMoveException exception) {
          System.out.println("IllegalMoveException");
        }

      score = getMinValue(positionNode, depthLimit, alpha, beta);

      if (score >= beta) {
        positionNode.getPosition().undoMove();
        positionNode.decrementDepth();
        return beta;
      }

      if (score > alpha) {
        alpha = score;
      }

      positionNode.getPosition().undoMove();
      positionNode.decrementDepth();
    }

    return alpha;
  }
```

The modified alpha-beta pruning `getMaxValue()` is shown above. The overall framework of the algorithm is much like that for regular min-max. However, now, the algorithm keeps track of an alpha value and beta value. Alpha represents the maximum value for the maximizer found so far to the root. Beta represents the minimum value for the minimizer found so far to the root. The `getMaxValue()` method essentially checks the values returned by all moves and sees if any of them is greater than beta, the lowest value found by the parent so far. If a value is found greater than beta, no further children moves are explored and beta is simply returned back to the parent. This is because the parent minimizer node would never have picked the current maximizer node if it had an option greater than the best option for the minimizer.

```
//getMinValue method; recursively calls getMaxValue method; one of two
//components for the min-max algorithm
//modified for alpha-beta pruning
  public int getMinValue(PositionDataNode positionNode, int depthLimit,
  int alpha, int beta) {
    if (cutoffTest(positionNode, depthLimit)) {
      return utilityCalculator(positionNode.getPosition());
    }

    int score;
```

```
    short[] moves = positionNode.getPosition().getAllMoves();

    for (int i = 0; i < moves.length; i++) {

      try {
      positionNode.getPosition().doMove(moves[i]);
      positionNode.incrementDepth();
      }

      catch (IllegalMoveException exception) {
        System.out.println("IllegalMoveException");
      }

      score = getMaxValue(positionNode, depthLimit, alpha, beta);

      if (score <= alpha) {
        positionNode.getPosition().undoMove();
        positionNode.decrementDepth();
        return alpha;
      }

      if (score < beta) {
        beta = score;
      }

      positionNode.getPosition().undoMove();
      positionNode.decrementDepth();
    }

    return beta;
  }
```

The code for for `getMinValue()` is above and is very similar in implementation to the `getMaxValue()` function. The method goes through all children moves and finds the utility value for each. If a move returns a utility value less than that of alpha, the best value for the parent maximizer node, all other moves are not searched and alpha is simply returned back to the parent. The logic is that the maximizer parent would never have selected the current node for expansion if there was an option for the minimizer child to choose a lower value than its own best value so far.

8 Transposition Table

After min-max and alpha-beta pruning algorithms are implemented, our ChessAI runs very fast and searches minimum amounts of nodes. However, there is a way to reduce the run-time of our algorithm and number of nodes searched further. This can be achieved by using a transposition table. The idea behind a transposition table is to store visited/searched positions in a table with their returned values. When the position is visited again, the value can be obtained from the table instead of a redundant search. Below are the modifications made to the `getMaxValue()` and `getMinValue()` methods in alpha-beta pruning min-max to implement the transposition table. The transposition table itself is a ConcurrentHashMap¡Position, Integer¿. Because `Position.java` has a hash function that uses Zorbists hashing, all positions can be ensured not to collide with other different positions in the hash map.

```
if (! transpositionTable.contains(positionNode.getPosition())) {
  score = getMinValue(positionNode, depthLimit, alpha, beta);
  transpositionTable.put(positionNode.getPosition(), score);
}

else {
  score = transpositionTable.get(positionNode.getPosition());
}
```

The above code is what is used to obtain score for alpha-beta comparison in `getMaxValue()` instead of previous simpler score calculations (a simple call to `getMinValue()`). Essentially, it checks if the transposition table has the current position. If it does not, the score is calculated like regular and the position with the score is put into the transposition table. If the the position is in the table already, the score is obtained simply from the table. This saves upon the amount of nodes searched and therefore reduces run-time in our algorithm.

9 Alpha-Beta Pruning Results and Discussion

With alpha-beta pruning implemented in our min-max search, we can expect a huge decrease in run-time and nodes explored. In In `Figure 3` (shown below), we ran the AlphaBetaPruningAI against a RandomAI at the original depth of 5. The results were as expected. AlphaBetaPruningAI made decisions much like MinMaxAI and was indeed able to force a checkmate and win significantly. However, run-time per move was greatly improved (2 seconds). This is clear from the number of nodes explored (12k-30k). This is a huge improvement from the trial shown in `Figure 1` (shown below) where nodes explored was 500k-900k @ same 5 depth. This indicates that AlphaBetaPruningAI was able to prune off large sections of the tree and therefore save great amounts of time/decrease nodes explored.

Next, we must see that AlphaBetaPruningAI and MinMaxAI return the same move decisions. This is important because alpha-beta pruning should not change the decision in the end; all it does it prune unnecessary parts of the tree which could never contain a better terminal state. In In `Figure 4` (shown below), we test the decision making similarities between the two algorithms. Both algorithms were confronted with similar situations/initial positions. Indeed, both algorithms did decide on the same best move, the overtaking of the pawn. This shows that our alpha-beta pruning algorithm was efficient at saving time without sacrificing accuracy compared to min-max algorithm.

10 Extra: Better Cutoff Test

A possible idea for a better cutoff test is shown below:

```
//cut-off test to check if search/evaluation should be stopped at node
//returns true if either (1) maximum depth is reached or (2) game is over
//modified to check further into tree if we are at check (very bad position)
  public boolean cutoffTest(PositionDataNode positionNode, int depthLimit) {

      if (positionNode.getPosition().isCheck()) {
            return false;
            //maybe also decrement depth or increase depth limit so
            //search can go as much as necessary, further than just one more
            //level
      }
```

```
        else {
    return (positionNode.getDepth() >= depthLimit ||
     positionNode.getPosition().isTerminal());
     }
}
```

The idea behind this altered cutoff test is that in addition to checking if we are at an end-game state and if we are at depth limit, we also check if we are being checked. If we are being checked, we return false so we can go search move even further. The idea/assumption behind this addition to the cutoff test is that if we are being checked, we MIGHT be really in danger of being checkmated. By allowing to go steps further, we can see if we can avoid this checkmate. This cutoff addition is useful because sometimes the checked state might have a good material value; this might trick the AI to go towards the move when just beyond the horizon, we might be headed towards a losing checkmate state.

11 Extra: Related Works

A related work on Chess AI is the paper, "Parallelizing a Simple Chess Program, by Dr. Brian Greskamp. In the paper, Greskamp discusses several enhancements that could be made to the standard ChessAI (iterative deepening, alpha-beta pruning w/ transposition table). Greskamp introduces the idea of a history table in addition to a transposition table. The history table is a second HashMap which assists in move ordering at all levels in tree. The idea is to keep track of nodes that are likely to return good values so as to be able to search them first. Greskamp's biggest addition to the standard ChessAI design is the parallel alpha-beta search. He talks about assigning different processors to different parts of the tree so as to concurrently calculate subtrees. The threads and multiple processes can then come together at the end to achieve the same results as a single non-parallel algorithm. It would reduce run-time greatly (from exponential run-time to closer to linear run-time). In the end, he concludes that his concurrency program, mscp-par, does have bugs which will need to be addressed in future research.

Source: http://iacoma.cs.uiuc.edu/ greskamp/pdfs/412.pdf

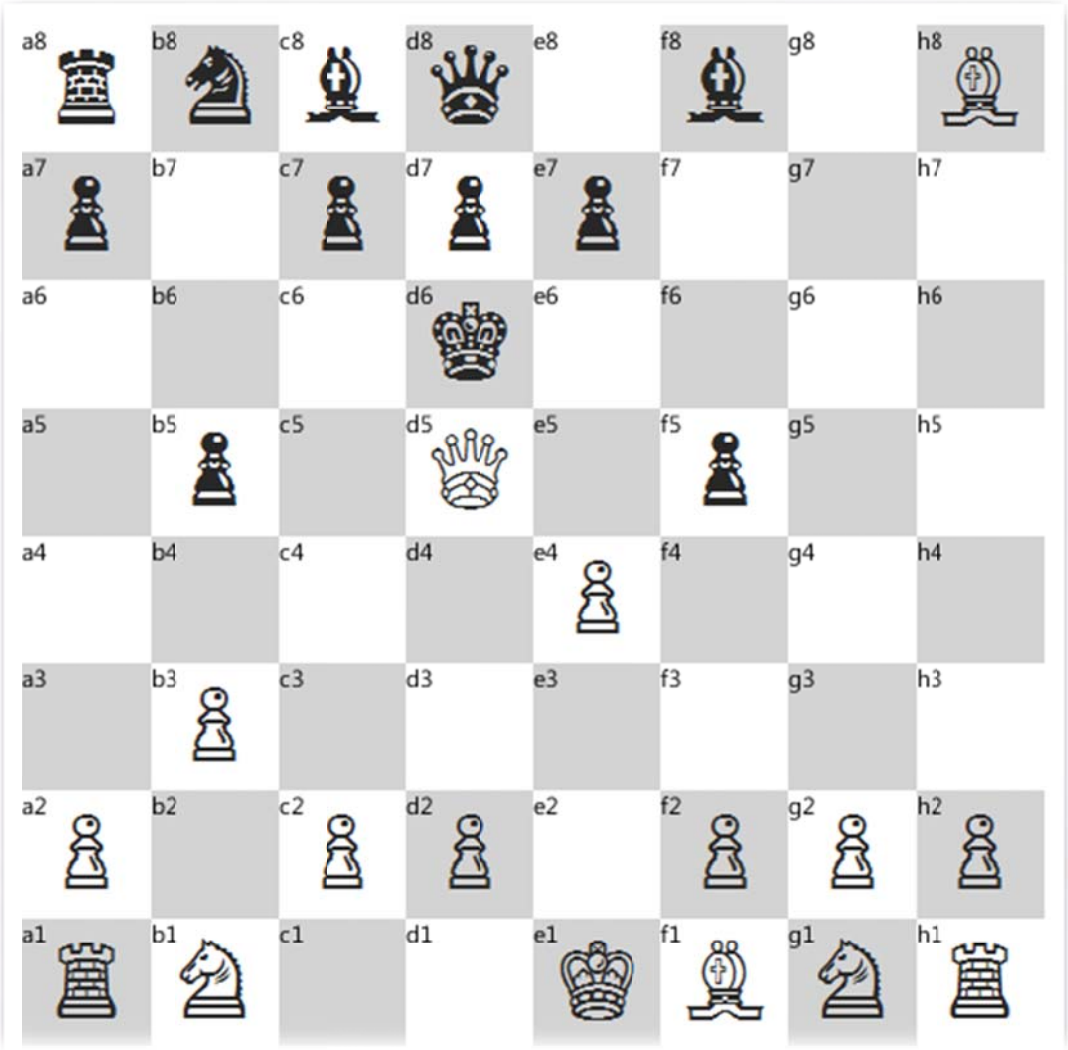


Figure 1: MinMaxAI (white) vs RandomAI (black) @ depth 5

Nodes Explored: ~500k-900k

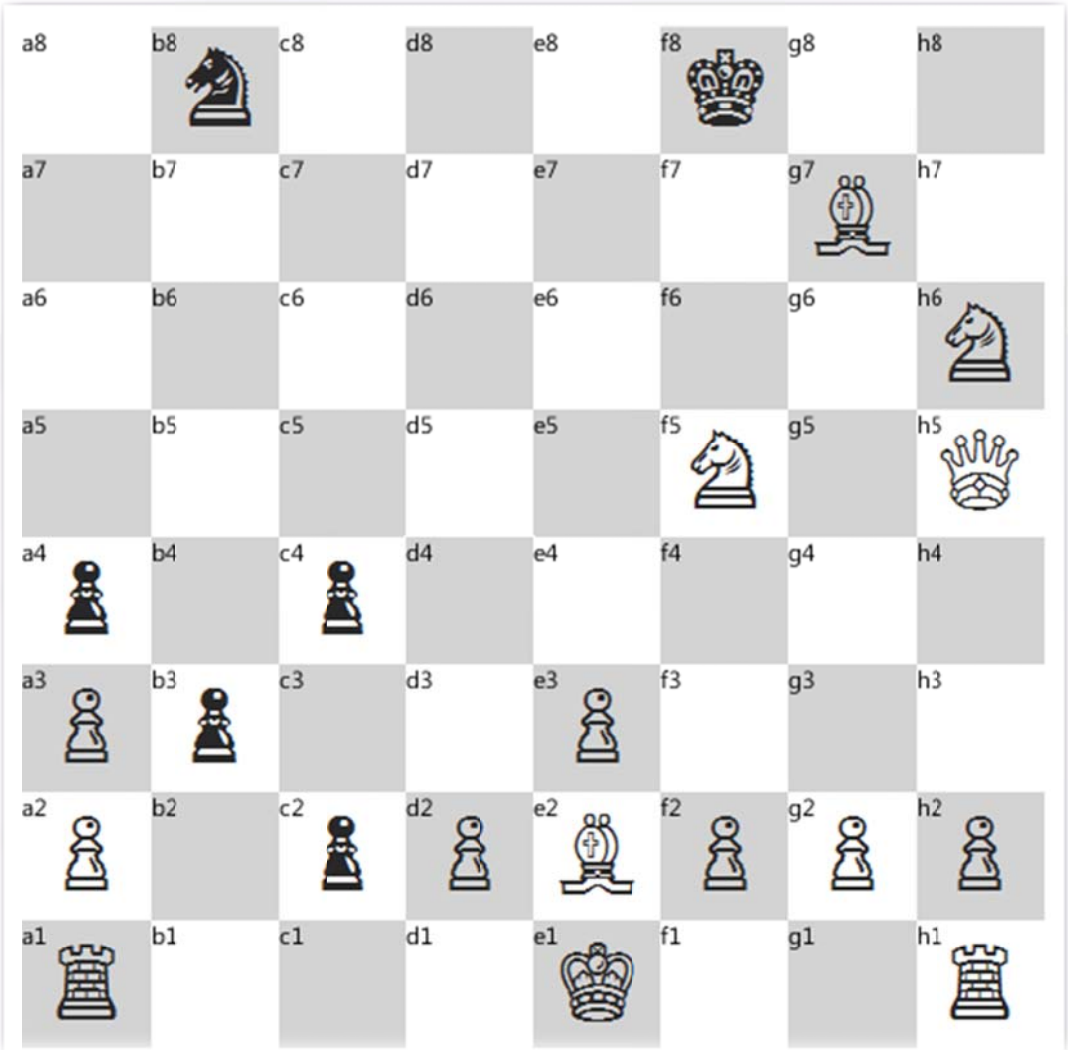


Figure 2: MinMax*AI* (white) vs Random*AI* @ depth 3

Nodes Explored: ~5k-20k

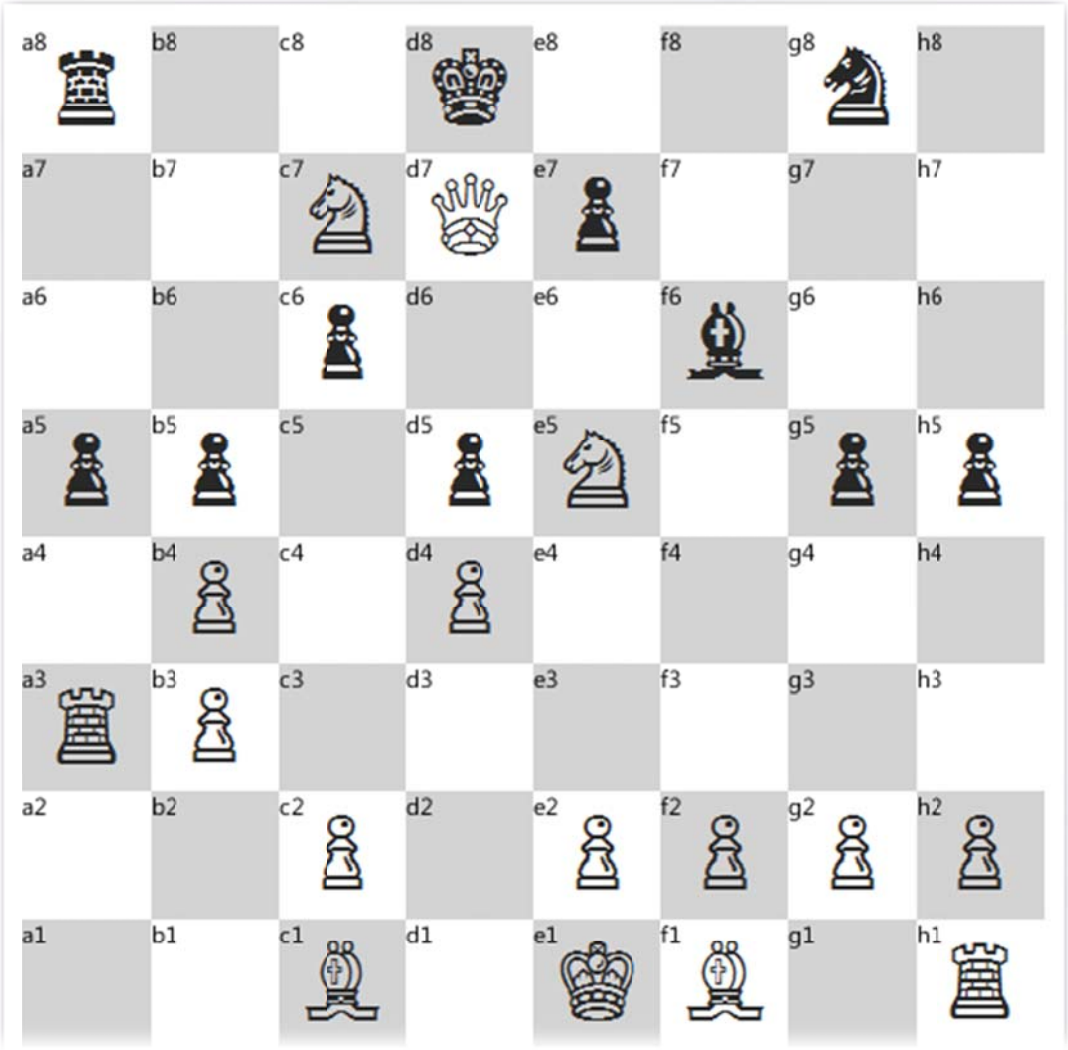
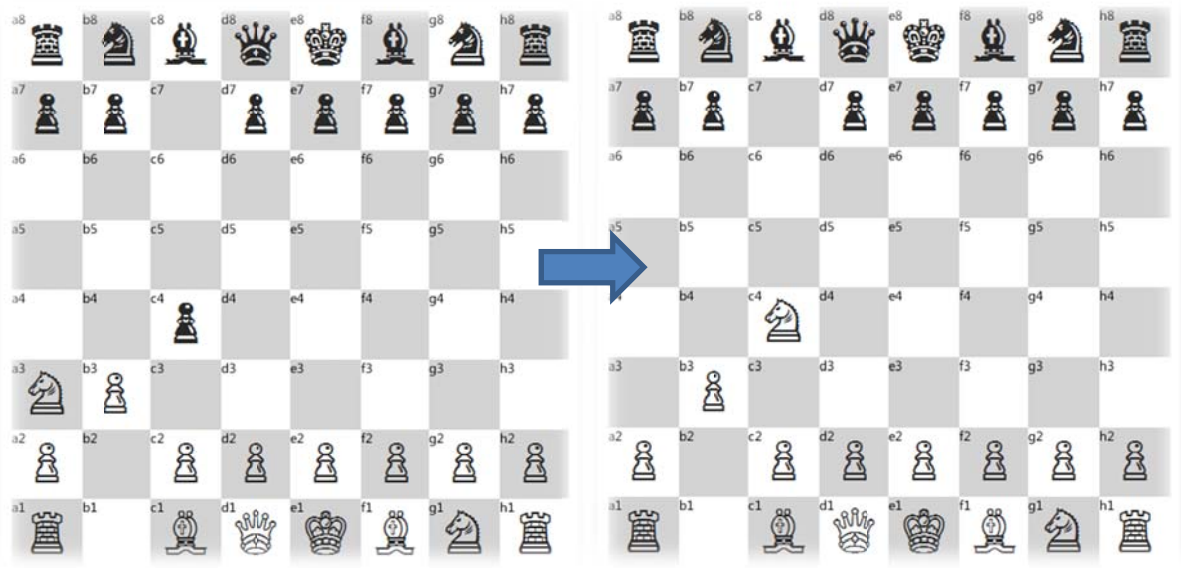
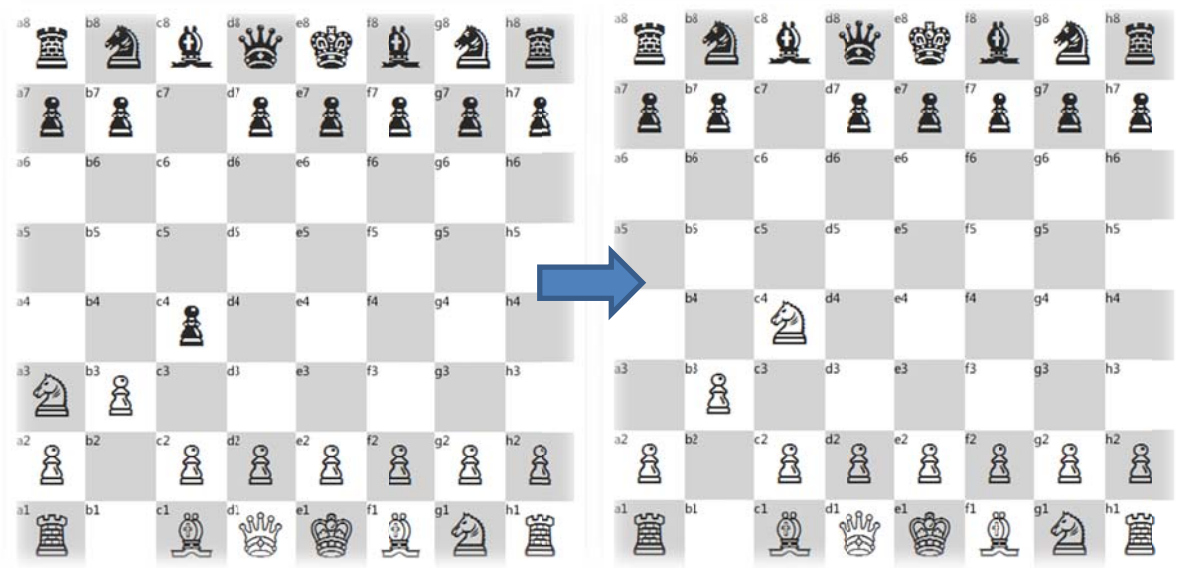


Figure 3: AlphaBetaPruning*AI* (white) vs Random*AI* (black) @ depth 5

Nodes Explored: ~12-30k



Min-Max Algorithm



Alpha-Beta Pruning Algorithm

Figure 4: Comparison of Decisions between Min-Max and Alpha-Beta Algorithms