Chess AI

James Jia

October 24, 2016

1 Introduction

In this report, I will discuss my approach to creating a Chess AI program that will be able to select good moves against both AI and human opponents. I implemented my ai move selection using the provided Chesspresso library, and in this assignment I have included 3 separate chess AI classes. The first of which, MiniMax.java will use minimax search to select moves against its opponent. The second, AlphaBeta.java, will similarly use minimax search, but with the addition of Alpha Beta pruning to search more efficiently by only searching through nodes in the game tree that are viable. Finally, the last chess AI class is AlphaBetaTransposition, which does a minimax search with both alpha beta pruning as well as the addition of a transposition table to further eliminate redundancies in the search algorithm by keeping track of a visited set of equivalent positions, or "transpositions". The transposition table utilizes the hascode function already implemented within Chesspresso's position class for hashing.

2 Utility and Evaluate Function

In order to conduct a minimax search, it is necessary to have a utility function that will be able to return the value of leaf nodes in the game tree. This will allow the value of terminal states to be passed up the tree to the starting node so it can make a move decision. I implemented a method called utility() in my classes that would take a position and return its utility. It is shown below.

```java

public int utility(Position pos) {

  int value = 0;

  //checkmate
  if(pos.isMate()){

    //ai checkmated opponent
    if (pos.getToPlay() != color) {
      value = BESTCASE;
    }
    //ai got checkmated
    if (pos.getToPlay() == color) {
      value = WORSTCASE;
    }
  }

  else if (pos.isStaleMate())
    value = 0;

  else {
    value = evaluate(pos);
  }
  return value;
}
```

This utility function will test if the current is a terminal node, indicated by a checkmate for either side or a stalemate, and then return a large positive or large negative value depending on whether the state is a win or loss for the AI. For a stalemate, the function returns a value of zero. For non-terminal states, the value will call an evaluate function to determine the utility value of the position. The evaluate() function is shown below.

```java

public int evaluate(Position position) {

  int value = 0;
```

```
    //If the turn is the AI's, then the getMaterial score is correct
    if (position.getToPlay()==color) {

      //getMaterial gets material score
      //getDomination weighs occupation of center squares more, not as important as material score
      value += position.getMaterial();
      value += 0.5*position.getDomination();

    //If the turn is the other person's, then negate score to get AI's score
    }
    if(position.getToPlay()!=color) {
      value -= position.getMaterial();
      value -= 0.5*position.getDomination();

    }
    return value;
  }
```

Here, the evaluate function will evaluate non-terminal positions by considering the material score of the position. Because the getMaterial() method in Chesspresso takes into account whose turn it is, we have to negate the score when it's not the AI's turn, because we always want to evaluate the material score of the position from the AI's standpoint. Additionally, Chesspresso has a method called getDomination(), which will increase scores for positions that occupy spaces near the center, as that signifies a dominant position for a player during a chess game, as the center of the board gives the player many options to attack from. However, this domination score is weighted by 0.5 because it is of secondary importance to the material score.

3 Cutoff Test

In addition to the utility and evaluate functions, we need a cutoff test that tells minimax search when to stop. This is because minimax search essentially does a depth first search through the game tree and requries a base case to stop the search. The two main base cases we look at are whether the search has exceeded a predetermined depth and whether the search has encountered a terminal node. The cutoff function is shown below.

```

  public boolean cutOff(Position pos, int depth) {

    if (depth >= maxD){
      return true;
    }
    if(pos.isMate()){
      return true;
    }
    if(pos.isStaleMate()){
      return true;
    }
    if (pos.isTerminal()){
      return true;
    }
    return false;
  }
```

4 Minimax Search

Finally, we have all the pieces required to conduct a Minimax search. This is shown in the miniMax(), mini(), and max() methods shown below.

```

  public short miniMax(Position pos, int startDepth) {

    //initialize variables for minimax, assuming maximizer
    short bestMove = 0;
    int bestValue = WORSTCASE;
    int currentValue;

    //get set of possible moves and try each one out
    for (short move : pos.getAllMoves()) {

      try {
```

```java
        pos.doMove(move);
        currentValue = mini(pos, 0);

        //if value is better, then update bestValue
        if (currentValue > bestValue) {
          bestValue = currentValue;
          bestMove = move;
        }

        //undo move to try more moves
        pos.undoMove();
      } catch (IllegalMoveException e) {
      }
    }
    return bestMove;
  }

  //returns value of min player's move choice
  public int mini(Position pos, int depth) {

    nodes++;
    int currentDepth = depth+1;

    //Base case: pass cutoff
    if (cutOff(pos, currentDepth)){
      return utility(pos);
    }
    //initially, minimizer's bestValue is +infinity
    int bestValue = BESTCASE;

    //get moves and try each one out
    for (short move : pos.getAllMoves()) {
      try {
        pos.doMove(move);
        //if move is better, then bestValue is updated
        bestValue = Math.min(bestValue, max(pos, currentDepth));
        pos.undoMove();
      } catch (IllegalMoveException e) {
      }
    }
    return bestValue;
  }

  //returns value of max player's move choice
  public int max(Position pos, int depth) {

    nodes++;
    int currentDepth = depth+1;

    //Base Case: Pass Cutoff
    if (cutOff(pos, currentDepth)){
      return utility(pos);
    }

    int bestValue = WORSTCASE;
    for (short move : pos.getAllMoves()) {
      try {
        pos.doMove(move);
        bestValue = Math.max(bestValue, mini(pos, currentDepth));
        pos.undoMove();
      } catch (IllegalMoveException e) {
      }
    }
    return bestValue;
  }
```

The miniMax method will take a position as well as a starting depth. The position is the initial game state that the search will start from and the starting depth is what depth the search starts at. To clarify, being passed a starting depth of 3 does not mean that the search will search 3 layers into the tree. Rather, it means the search will "start" at depth 3 and search until the max depth is reached, so in total the Minimax search will search the (maximum depth - starting depth) layers. In this class, the maximum depth is stored in the maxD instance variable in MiniMax.java.

The miniMax method will start the search assuming the AI is in the maximizer position, so it will call min on its possible moves, which will in turn call max on its moves, etc. until a base case is reached. Here, the base case is a position passes a cutoff test - if a position passes it, then the search will immediately return the utility of the current position up the tree and stop searching downward. Additionally, both max and min will keep track of the value of their best possible move in bestValue, and that will be the value they return up the tree if they are not at a leaf node. Thus, leaf nodes will return their utility up the tree after hitting the base case, and tree nodes will return the best possible value amongst the value of all their possible moves. Additionally, I'm keeping track of the node count with an instance variable called node to later compare the efficiency of the various searches.

5 Iterative Deepening MiniMax

After implementing depth-limited Minimax, I used a for loop and an instance variable called bestMove to keep track of the results from running my Minimax search up to varying depths. The method idmm() will do the work of iteratively calling the depth-limited minimax function, and getMove() will simply return the bestMove chosen by idmm(). Again, because Minimax takes into account the starting depth, and not the max depth as its parameter, the for loop will have to decrement with each loop, first starting at max depth -1, so it will search 1 layer to reach max depth. The final loop will start at depth 0 and search up to max depth, so it will in total search max depth layers.

```java

  public short getMove(Position pos) {

    Position position = new Position(pos);
    short move = idmm(position);
    System.out.println("Minimax nodes searched: " + nodes);
    return move;
  }

  public short idmm(Position pos) {

    //This loop will run minimax search for depth 1 up to depth maxD
    for (int i = maxD-1; i >= 0; i--) {

      //save results from each loop in bestMove instance variable
      bestMove = miniMax(pos,i);

      //if found checkmate, no need to go deeper
      try{
        pos.doMove(bestMove);
        if(utility(pos)==BESTCASE){
          return bestMove;
        }
        pos.undoMove();
      }
      catch (IllegalMoveException e){
      }
    }
    return bestMove;
  }
```

6 Alpha Beta Pruning

In addition to Minimax search, I also implemented Alpha Beta pruning on my Minimax search - this would prune off certain branches of the tree where it was not necessary to search any deeper. It would go about this by keeping track of two variables, alpha and beta, which represented the maximizer's worst case scenario and the minimizer's worst case scenario, respectively. If maximizer has a possible position in which the value of that position is greater than beta, then we prune off the rest of that tree and don't explore that position's sucessors. The reason for this is that the maxizer will always pick something of higher or equal value to the node we looked at, but this value is higher than minimizer's worst case scenario. Ultimately, this means that beta will not allow maximizer to even get to this node because its worst case scenario is better than letting maximizer get to choose that position. The same logic, applies to minimizer - if it reaches a node that potentially has a value less than alpha, then we prune off that branch because maximizer will not let it get to a value that's lower than its current worst case scenario. The three methods for Minimax search with Alpha Beta pruning are shown below.

```java

  //alpha beta MiniMax search
  public short abMiniMax(Position pos, int startDepth) {

    short bestMove = 0;
    int bestValue = WORSTCASE;
```

```java
    int currentValue;
    int alpha = WORSTCASE;
    int beta = BESTCASE;

    for (short move : pos.getAllMoves()) {

      try {
        pos.doMove(move);
        currentValue = mini(pos, 0, alpha, beta);

        if (currentValue > bestValue) {
          bestValue = currentValue;
          bestMove = move;
        }

        pos.undoMove();
      } catch (IllegalMoveException e) {
      }
    }
    return bestMove;
  }

  //returns mini's best move value
  public int mini(Position pos, int depth, int alpha, int beta) {

    nodes++;
    int currentDepth = depth+1;

    //Base case: pass cutoff
    if (cutOff(pos, currentDepth)){
      return utility(pos);
    }

    int bestValue = BESTCASE;
    for (short move : pos.getAllMoves()) {
      try {
        pos.doMove(move);
        bestValue = Math.min(bestValue, max(pos, currentDepth, alpha, beta));
        pos.undoMove();

        //Pruning - if value is less than alpha (max's worst case), no need to continue
        if (bestValue<=alpha){
          return bestValue;
        }

        //update beta if bestValue is better
        beta = Math.min(beta, bestValue);

      } catch (IllegalMoveException e) {
      }
    }
    return bestValue;
  }

  //returns max's best move value
  public int max(Position pos, int depth, int alpha, int beta) {

    nodes++;
    int currentDepth = depth+1;

    if (cutOff(pos, currentDepth)){
      return utility(pos);
    }

    int bestValue = WORSTCASE;
    for (short move : pos.getAllMoves()) {
      try {
        pos.doMove(move);
        bestValue = Math.max(bestValue, mini(pos, currentDepth, alpha, beta));
        pos.undoMove();

        //Pruning - if value is greater than beta (mini's worst case), no need to continue
        if(bestValue>=beta){
          return bestValue;
        }

```

```
        //update alpha if bestValue is better
        alpha=Math.max(alpha, bestValue);

      } catch (IllegalMoveException e) {
      }
    }
    return bestValue;
  }
```

The rest of the methods, including the iterative deepening component, are the same between Minimax and Minimax with Alpha Beta Pruning.

7 Minimax Search with Alpha Beta and Transposition Table

Next, I used a transposition table to keep track of visited equivalent states in a hashtable in order to further reduce redundancies in the search algorithm. This was implemented by using a hashtable that would store a the hashcode that results from calling getHashCode() on a given position (implemented in Chesspresso already), and using that as a key that mapped to the corresponding position's value and depth, stored as an integer array. For each move that is tried, the resulting position is hashed and we check if the position's hashcode is already inside the hashtable.

If it's already inside, then we check whether the depth that we're at currently is less than the depth associated with the previously found position. If it's less than the previous depth, then we update the depth associated with the position in the hashtable. Additionally, we don't need to keep searching down the tree because we already know the best value from this point, as it is already stored in the transposition table, so we can just return that value. If it's not less than the previous depth, then we keep searching down the tree.

If the position was not already inside, then we need to continue down the tree to get its best value. After we get its best value, we can record its information in the transposition table for later comparison. The code for the transposition table implementation is shown below. Note: the depth of each position is (max depth - start depth) because that tells you how many layers deep it has searched so far.

```

  //alpha beta transposition table mini max search
  public short abttMiniMax(Position pos, int startDepth) {

    short bestMove = 0;
    int bestValue = WORSTCASE;
    int currentValue;
    int alpha = WORSTCASE;
    int beta = BESTCASE;

    //try all moves and get their utility
    for (short move : pos.getAllMoves()) {

      try {
        pos.doMove(move);

        //if already in transposition table - uses position's getHashCode
        if (tt.containsKey(pos.getHashCode())) {

          //info array stores position's value and index
          int info[] = tt.get(pos.getHashCode());

          //if search depth is less than previous search depth
          if (maxD-startDepth < info[1]) {

            //update position's info with this position
            currentValue = info[0];
            int[] updateInfo = {currentValue, maxD-startDepth};
            tt.put(pos.getHashCode(), updateInfo);

          //otherwise, keep searching down the tree
          } else {
            currentValue = mini(pos, startDepth, alpha, beta);
          }
        }

        //if position not already in table, keep searching down tree
        //then add position into table with its info
```

```
        else {
          currentValue = mini(pos, startDepth, alpha, beta);
          int[] updateInfo = {currentValue, maxD-startDepth};
          tt.put(pos.getHashCode(), updateInfo);
        }

        if (currentValue > bestValue) {
          bestValue = currentValue;
          bestMove = move;
        }

        pos.undoMove();
      } catch (IllegalMoveException e) {
      }
    }
    return bestMove;
  }


  public int mini(Position pos, int depth, int alpha, int beta) {

    nodes++;
    int currentDepth = depth+1;

    //Base case: pass cutoff
    if (cutOff(pos, currentDepth)){
      return utility(pos);
    }

    int bestValue = BESTCASE;
    for (short move : pos.getAllMoves()) {
      try {
        pos.doMove(move);

        //if already in transposition table
        if (tt.containsKey(pos.getHashCode())) {
          //get the position's value and depth info
          int info[] = tt.get(pos.getHashCode());

          //if position's search depth (not current depth) <previously found
          //then update table with position's info
          if (maxD-currentDepth < info[1]) {
            bestValue = info[0];
            int[] updateInfo = {bestValue, maxD-currentDepth};
            tt.put(pos.getHashCode(), updateInfo);

          //not less than previously found - continue down tree
          }
          else {
            bestValue = Math.min(bestValue, max(pos, currentDepth, alpha, beta));
          }

        //not in table - continue down tree, then update table with position's info
        }
        else {
          bestValue = Math.min(bestValue, max(pos, currentDepth, alpha, beta));
          int[] updateInfo = {bestValue, maxD-currentDepth};
          tt.put(pos.getHashCode(), updateInfo);
        }
        pos.undoMove();

        //alpha beta pruning
        if (bestValue<=alpha){
          return bestValue;
        }

        beta = Math.min(beta, bestValue);

      } catch (IllegalMoveException e) {
      }
    }
    return bestValue;
  }

  public int max(Position pos, int depth, int alpha, int beta) {

```

```java
  nodes++;
  int currentDepth = depth+1;

  if (cutOff(pos, currentDepth)){
    return utility(pos);
  }

  int bestValue = WORSTCASE;

  //try all moves
  for (short move : pos.getAllMoves()) {
    try {
      pos.doMove(move);

      //if in table already
      if (tt.containsKey(pos.getHashCode())) {

        //get the position's value and depth info
        int info[] = tt.get(pos.getHashCode());

        //if position's search depth (not current depth) is less than previously found
        //then update table with position's info
        if (maxD-currentDepth < info[1]) {
          bestValue = info[0];
          int[] updateInfo = {bestValue, maxD-currentDepth};
          tt.put(pos.getHashCode(), updateInfo);
        }

        //if position's search depth >= previously found - continue down tree
        else {
          bestValue = Math.max(bestValue, mini(pos, currentDepth, alpha, beta));
        }
      }

      //if not already in table - continue down tree, and then update
      else {
        bestValue = Math.max(bestValue, mini(pos, currentDepth, alpha, beta));
        int[] updateInfo = {bestValue, maxD-currentDepth};
        tt.put(pos.getHashCode(), updateInfo);
      }
      pos.undoMove();


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

    } catch (IllegalMoveException e) {
    }
  }
  return bestValue;
}
```

8 Profiling

One way that I profiled my AI was by counting the number of nodes that it searched through. This was done by keeping track of an instance variable called nodes and incrementing it every time max or min was called. I then compared the number of nodes searched for, which was essentially a proxy for number of function calls. I tested this by making the AI black and the human player White. I then did the same opening move (pawn from d2 to d4) and tested how many nodes the AI searched before making its move. The output is shown below.

Minimax:
making move 5835
rnbqkbnr/pppppppp/8/8/3P4/8/PPP1PPPP/RNBQKBNR b KQkq d3 0 1
Minimax nodes searched: 1499220
making move 6834
rnbqkbnr/pp1ppppp/2p5/8/3P4/8/PPP1PPPP/RNBQKBNR w KQkq - 0 2

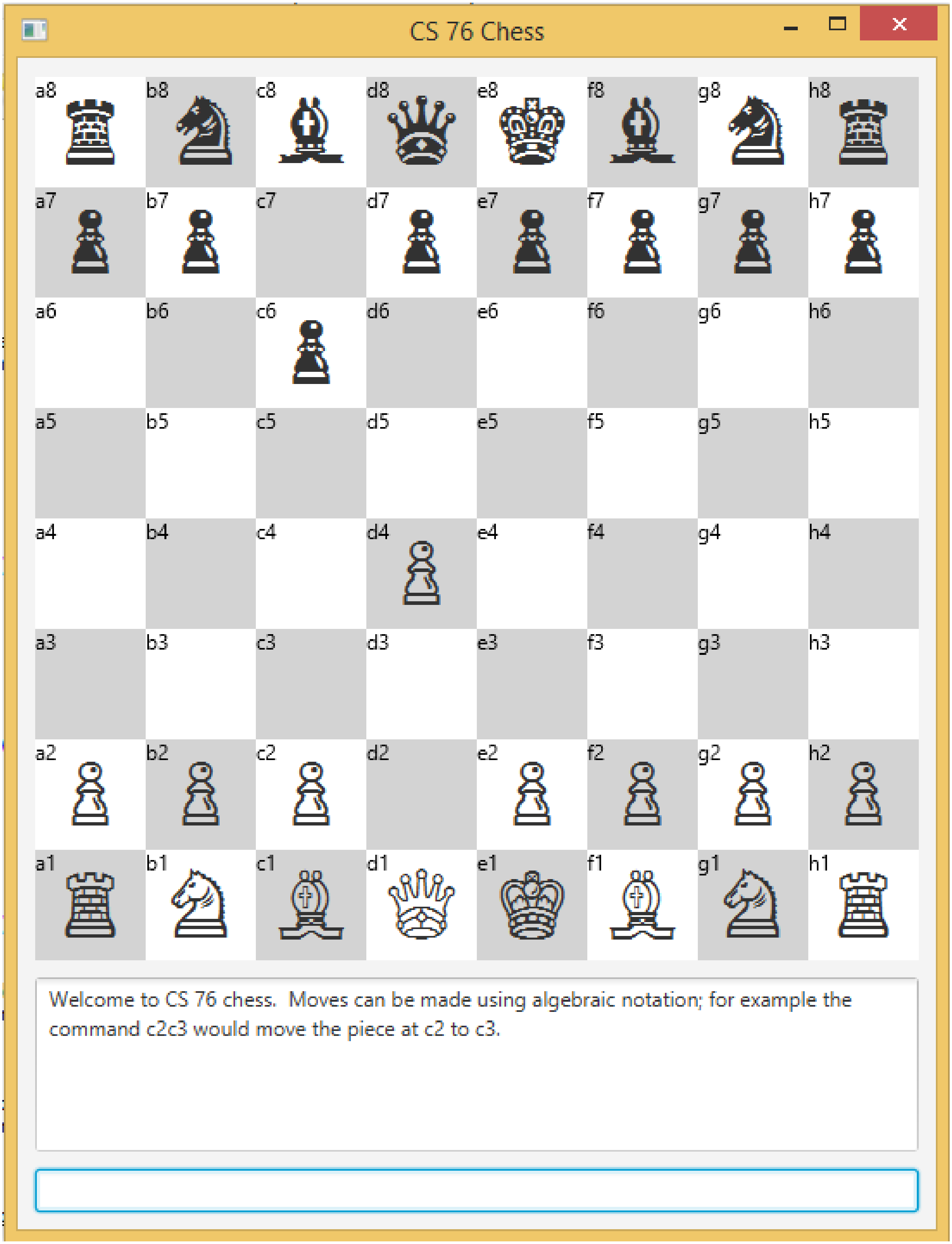
Minimax with Alpha Beta:
making move 5835
rnbqkbnr/pppppppp/8/8/3P4/8/PPP1PPPP/RNBQKBNR b KQkq d3 0 1
Alpha Beta nodes searched: 246832
making move 6834
rnbqkbnr/pp1ppppp/2p5/8/3P4/8/PPP1PPPP/RNBQKBNR w KQkq - 0

Alpha Beta and Transposition Table Minimax:
making move 5835
rnbqkbnr/pppppppp/8/8/3P4/8/PPP1PPPP/RNBQKBNR b KQkq d3 0 1
Alpha Beta W/ Transposition Table nodes searched: 66272
making move 6834
rnbqkbnr/pp1ppppp/2p5/8/3P4/8/PPP1PPPP/RNBQKBNR w KQkq - 0 2

Minimax searched around 1.5 million nodes, and with the addition of alpha beta that number was cut down to around 250,000, and with the transposition table added, it went down to around 66,000. This clearly shows that the addition of more clever search algorithms reduced the number of function calls and as a result increased the speed of the search significantly.

9 Testing

In order to test my algorithms, I made sure that they would always pick the same moves given the same opening state. I made myself the white player and manually chose the move pawn from b2 to b4. The following was the result from all three AIs, showing that they weren't pruning off branches that were important.


CS 76 Chess
a8
b8
c8
d8
e8
f8
g8
h8
a7
b7
c7
d7
e7
f7
g7
h7
a6
b6
c6
d6
e6
f6
g6
h6
a5
b5
c5
d5
e5
f5
g5
h5
a4
b4
c4
d4
e4
f4
g4
h4
a3
b3
c3
d3
e3
f3
g3
h3
a2
b2
c2
d2
e2
f2
g2
h2
a1
b1
c1
d1
e1
f1
g1
h1
Welcome to CS 76 chess. Moves can be made using algebraic notation; for example the command c2c3 would move the piece at c2 to c3.