CS76 Assignment 4: Chess AI

Benjamin Hannam

Fall 2016

NB: I collaborated with Brian Keare

1 Minimax and Cutoff Test

CutOffTest

The first step in creating the Chess AI was to implement a simple Minimax search to get the best move. The basis of this was to search each possible moves from the current state, and the possible moves from the next states and so on until we either hit an end game scenario (checkmate or stalemate) or we reached the maximum depth(this was done by checking for if the depth was 0 as the depth was decremented throughout). This was checked in the CutOffTest function:

```java
//cut off test method
public boolean CutOffTest(Position current, int depth){
 return depth == 0 || current.isMate() || current.isStaleMate();
}
```

code/Minimax.java

Search

The actual minimax search function was split up into three main functions: MinimaxDecision, MinValue and MaxValue. They worked as follows:

1. MinimaxDecision: Set up the variable to hold the best move, looped over all the possible moves from the start position, called MinValue with the max depth and then got the largest score that was MinValue returned and returned the bestMove associated with that score:

```java
//minimax based off the pseudocode in the book
public short MiniMaxDecision(Position current, int max_depth) throws
IllegalMoveException {
        short bestMove = 0;
         int bestScore = Integer.MIN_VALUE;
         //loop over all the moves
        for(short move : current.getAllMoves()){

            //increment the count
             nodes_visited++;
            //do the move
            current.doMove(move);

            //if more than best score then set bestmove
            int temp = MinValue(current, max_depth);
            if (temp > bestScore) {
                bestScore = temp;
                System.out.println(move);
                bestMove = move;
            }
            //undo the move
            current.undoMove();
        }
        return bestMove;
    }
```

code/Minimax.java

2. MinValue: Initially tests to see if we are at a CutOff state. After this it loops over all the possible moves from the current position, calls the MaxValue on the corresponding next position and then compares the returned values. It takes the smallest value returned and passes that back to the function that called it.

```java
    //MinValue function
    public int MinValue(Position current, int depth) throws IllegalMoveException{

        //if a final state or maxDepth
        if(CutOffTest(current, depth)){
            return Utility(current, false);

        }
        int v = Integer.MAX_VALUE;

        //loop over moves
        for(short move : current.getAllMoves()){
            //do the move and increment the count
            nodes_visited ++;
            current.doMove(move);

            //get the MaxValue for the next level
            int temp = MaxValue(current, depth - 1);
            v = Math.min(temp, v);

            //undo the math
            current.undoMove();
        }
        return  v;
    }
```

code/Minimax.java

2. MaxValue: Very similar to the MinValue function except that it gets the maximum value returned. The loop over the possible moves is shown below:

```
        //loop over the moves
        for (short move : current.getAllMoves()) {

            //do the move and increment count
            nodes_visited++;
            current.doMove(move);

            //get the minvalue
            int temp = MinValue(current, depth);
            //get the max of the two
            v = Math.max(temp, v);
            //undo
            current.undoMove();
        }
```

code/Minimax.java

Utility and Evaluation function

I assigned a very large negative value to a checkmate if it was the max players turn and a very large positive value if it was the min players turn. I then also assigned a stale mate to 0 as it is even for both players.

```
    //utility function
    public int Utility(Position current, boolean max) {

        //if checkmate and max
        if (current.isMate() && max) {
            return Integer.MIN_VALUE + 1;
        }
        else if(current.isMate()){
            return Integer.MAX_VALUE - 1;
        }
        else if(current.isStaleMate()){
            return 0;
        }
        else{
            return Evaluation(current);
        }
    }
```

code/Minimax.java

I used a materialistic based approach to assessing the value of each non-terminal position. In this function I called the getMaterial() method for the position class which assigned the following values to each piece: Pawn-100, Knights-300, Bishops-325, Rooks-500 and Queens- 900. I also added a random integer between 0-20 into the value to avoid the potential danger of falling into endless loops where pieces alternate back and forth without doing anything. The value of 20 was chosen because it is less than the biggest difference between the pieces(25) so it will never override losing a piece.

```
    public int Evaluation(Position current){

        //random int used to prevent repetitive loops
        int rand = (int)(Math.random() * 20);

        return current.getMaterial() + rand;
    }
```

code/Minimax.java

2 Alpha-Beta

The Alpha-Beta Pruning helped optimize the minimax search by reducing the number of subtrees that we have to look at by ruling out trees that we know are already going to be worse. The alpha and beta values are used to keep track of the current best/worst possibility for the state depending on who is playing. Whenever we find a move that is definitely going to be worse on that branch of the tree we just stop searching through that tree.These branches will have no influence on the final result and so there is no need to evaluate deeper into them.

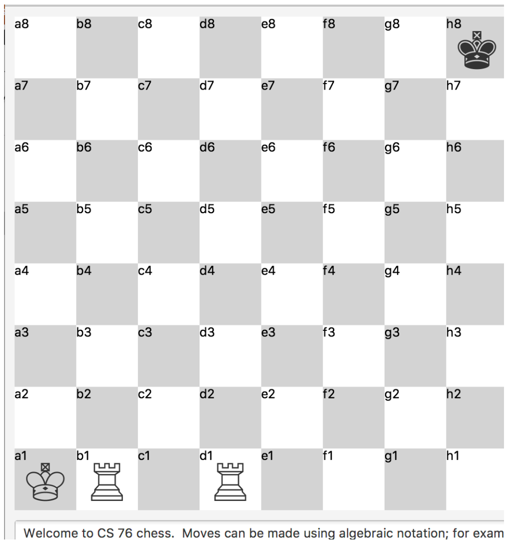
```
    public int MaxValue(Position current, int depth, int alpha, int beta) throws
    IllegalMoveException {

        //if the position is already in the transposition table
        if(transposition.containsKey(current.getHashCode())){
            //get the array
            ArrayList<Integer> arr_temp = transposition.get(current.getHashCode());
            //if the depths sync up
            if(arr_temp.get(1) >= depth){
                //get the value
```

```java
                int temp_value = transposition.get(current.getHashCode()).get(0);
                //if the position was a max negate the value
                if(arr_temp.get(2) == 0){
                    temp_value = -temp_value;
                }
                return temp_value;
            }
        }
        //if a final state or maxDepth
        else if(CutOffTest(current, depth)){
            int value = Utility(current, false);
            return value;

        }
        int v = Integer.MIN_VALUE;

        //loop over the moves
        for (short move : current.getAllMoves()) {

            //do the move and increment count
            nodes_visited++;
            current.doMove(move);

            //get the minvalue
            int temp = MinValue(current, depth, alpha, beta);
            //undo
            current.undoMove();
            //get the max of the two
            v = Math.max(temp, v);

            if(v >= beta){
                ArrayList<Integer> arr = new ArrayList<Integer>();
                arr.add(0, v);
                arr.add(1, depth);
                arr.add(2, 1);
                transposition.put(current.getHashCode(), arr);
                return v;
            }
            alpha = Math.max(alpha, v);

        }
        ArrayList<Integer> arr = new ArrayList<Integer>();
        arr.add(0, v);
        arr.add(1, depth);
        arr.add(2, 1);
        transposition.put(current.getHashCode(), arr);
        return v;
    }
```

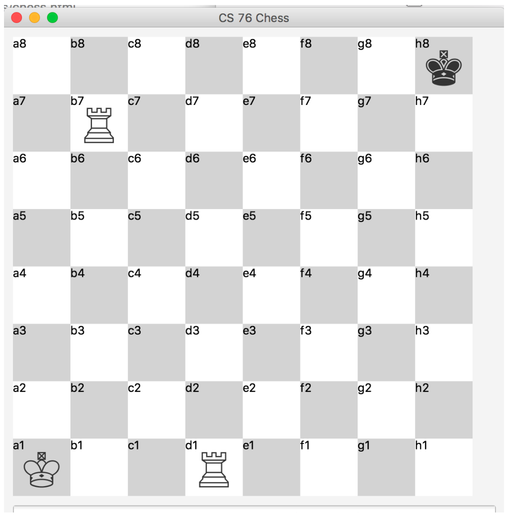
code/AlphaBetaMiniMax.java

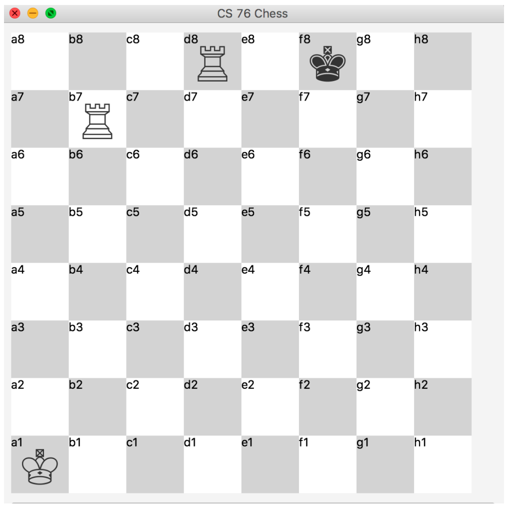
3 Minimax vs. Alpha-Beta Results

Due to the random factor that I introduced into the starting move was not always consistent so it was not possible to test the two functions to see if they did the same start moves. However I did run them both on an end game situation and they both closed out the game the same way, running on a depth of 2.

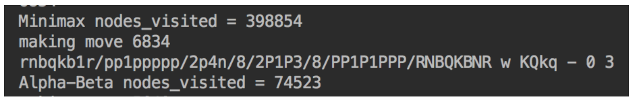


Welcome to CS 76 chess. Moves can be made using algebraic notation; for exam





As for the number of positions visited in each search it was very clear that the Alpha-Beta search was vastly superior in efficiency and produced the following numbers on depth 2:



```
Minimax nodes_visited = 398854
making move 6834
rnbqkb1r/pp1ppppp/2p4n/8/2P1P3/8/PP1P1PPP/RNBQKBNR w KQkq - 0 3
Alpha-Beta nodes_visited = 74523
```

4 Transposition Table

The transposition table is a Hashtable that keeps track of all the positions that we have already explored as the key and the values associated with those positions. This means that we can save time by not having to explore positions multiple times if they can be reached from different states. E.g At the beginning of a game, white moving its pawn, black moving its pawn and then white moving its knight would result in the same position as white moving its knight, black moving its pawn and the white moving its pawn.

When checking to see if a position has been visited we also have to take into consideration the depth at which it was explored. If it was explored at a depth greater than our current one then we would be better off continuing to explore it ourself as we will be able to get more information. We also have to check wether we explored that position at a max or a min node because this could affect the value.

With all these checks in place it was fairly easy to implement the Hashtable using the hash-code as the key and and Arraylist of integers as the value. In the zeroth slot was the value for that position, the first slot had the depth it was explored at and the last slot had whether it was visited by min or max.

We checked the table at the top of the function and set the position in the table when returning the value.

```
    public int MinValue(Position current, int depth, int alpha, int beta) throws
    IllegalMoveException{
```

```java

        //if the position is already in the transposition table
        if(transposition.containsKey(current.getHashCode())){
            //get the array
            ArrayList<Integer> arr_temp = transposition.get(current.getHashCode());
            //if the depths sync up
            if(arr_temp.get(1) >= depth){
                //get the value
                int temp_value = transposition.get(current.getHashCode()).get(0);
                //if the position was a max negate the value
                if(arr_temp.get(2) == 1){
                    temp_value = -temp_value;
                }
                return temp_value;
            }
        }
```

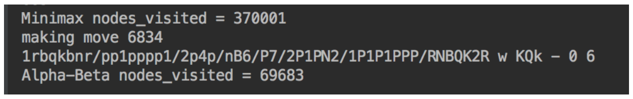
code/AlphaBetaMiniMax.java

```java
        //loop over moves
        for(short move : current.getAllMoves()){
            //do the move and increment the count
            nodes_visited ++;
            current.doMove(move);

            //get the MaxValue for the next level
            int temp = MaxValue(current, depth - 1, alpha, beta);
            //undo the math
            current.undoMove();
            v = Math.min(temp, v);

            if(v <= alpha){
                ArrayList<Integer> arr = new ArrayList<Integer>();
                arr.add(0, v);
                arr.add(1, depth);
                arr.add(2, 0);
                transposition.put(current.getHashCode(), arr);
                return v;
            }

            beta = Math.min(beta, v);
        }
        ArrayList<Integer> arr = new ArrayList<Integer>();
        arr.add(0, v);
        arr.add(1, depth);
        arr.add(2, 0);
        transposition.put(current.getHashCode(), arr);
        return  v;
    }
```

code/AlphaBetaMiniMax.java

Transposition Results

As we can see from the graphic below there is a noticeable difference when the transposition table is added.



```
Minimax nodes_visited = 370001
making move 6834
1rbqkbnr/pp1pppp1/2p4p/nB6/P7/2P1PN2/1P1P1PPP/RNBQK2R w KQk - 0 6
Alpha-Beta nodes_visited = 69683
```

5 Extra Credit: Quiescent Search

The idea behind Quiescent search is that you run the normal minimax search until you reach the cutoff test. In the cutoff test you do the same thing if we have reached a final state but if it is a non-game ending state then we check all the moves from the current state. We check each move to see if it is a ”quiet” move which basically means that it is a non-capturing or non check-making move, otherwise it is a ”noisy” move. In the Quiesce function we set the base value to be the standard evaluation for the the position. We then check this value relative to alpha and beta and act accordingly. From there we loop over all the noisy moves and recursively call quiesce with negative values to get the value for the noisy states before again checking the returned score again alpha and beta. After looping over all the moves we return alpha if beta was not previously returned.

The idea behind quiescent search is that at the max depth we may have believed that we have won by for example our queen taking their pawn, however what we do not know is that the next move down may involve some other piece capturing that queen and so it that case we want to keep going deeper until we reach a move where no pieces are taken. This prevents us stupidly losing pieces due to a bad capture or a bad check.

The Quiescent Search consistently beat out the AlphaBetaMiniMax search.

```
//Quiescent search based off of code from https://chessprogramming.wikispaces.com/
Quiescence+Search;
public int Quiesce(Position current, int alpha, int beta, boolean max) throws
IllegalMoveException{

    //if we are at a check/stale mate
    if(current.isMate() && max){
        return Integer.MIN_VALUE + 1;
    }
    else if (current.isMate()) {
        return Integer.MAX_VALUE - 1;
    }
    else if(current.isStaleMate()){
        return 0;
    }

    //base value to be used as the lower bound
```

```java
        int base = Evaluation(current);

        //if we are bigger than a value we have already found from a different node, return the other value
        if(base >= beta){
            return beta;
        }

        //if we are less that the small, reset the lower bound
        if(alpha < base){
            alpha = base;
        }
        int score;
        //loop through all the capturing moves
        for(short move : current.getAllMoves()){
            //do the move
            current.doMove(move);
            //check if we are at a quiet state
            if(Move.isCapturing(move) || current.isCheck()){
                current.undoMove();
                continue;
            }
            //get the score from the recursive call
            score = -Quiesce(current, -beta, -alpha, !max);
            current.undoMove();

            //if greater than the upper bound
            if (score >= beta) {
                return beta;
            }
            //if greater than the lower bound
            if (score > alpha) {
                alpha = score;
            }
        }
        return alpha;
    }
```

code/QuiescentSearch.java