Chess AI

Jing Ge

February 9, 2016

1 Introduction

This assignment aims to implement a chess AI using iterative deepening minimax search with alpha-beta pruning and a transposition table. In this paper, the first part shows the implementation of minimax search, which includes the utility function and non-random evaluation function of the chess game. This search algorithm was tested by several cases and the efficiency was not very well. Then the second part discusses alpha-beta pruning which could reduce the number of unnecessary searching by pruning them off. In the last part we uses a transposition table to record states we visited, which sacrifices space complexity but could accelerate the whole search algorithm.

2 Minimax and Cutoff Test

2.1 Implementation

This is a imperfect algorithm of the chess problem without any duplicates detection and pruning. The basic idea of this algorithm is assuming the best choice of a player comes from all choices of its opponent while its opponent offers the worst cases to him. In this algorithm, the `getMove` function returns the best move of this turn by using iterative deepening minimax search. The maximal search depth is customized and we try to find a the best move in every search depth. It is important to note that we will stop searching if we find the game will end in a specific depth, then we have no need to increase search depth. (In the chess problem, there are some end states check (win/lose/draw) and evaluations of different states may use in the algorithm, we will introduce implementations of these functions in the next parts.)

We only paste lines that are important here to explain it and the code of this algorithm is in `MiniMaxAI.java`.

```
public short getMove(Position ThisPosition) {
    // we'd better use a new Object since this is a multi threads program
    // if two AIs are playing against each other, it will cause collision
    Position position = new Position(ThisPosition);
    // indicates AI is black or white
    AI = position.getToPlay();
    // we will stop searching if we find that
    // 1. in a specific depth, the best move could guarantee a win
    // 2. in a specific depth, the best move will cause a lose (we will definitely lose)
    stopSearch = false;
    short bestMove = 0;
    for(int i = 1; i <= DEPTH && !stopSearch; i++)
        bestMove = minimaxSearch(position, i);
    return bestMove;
}
```

In order to determine the best move, minimaxSearch loops all possible moves from a given position and start from minValue, then use maxValue and minValue alternately. (As we mentioned above, you choose best move from all choices that your opponent offers, and he will definitely give you the worst, so we start from minValue and initialized max value as Integer.MIN_VALUE at first.)

```java
public short minimaxSearch(Position position, int maxDepth) {
    int maxVal = Integer.MIN_VALUE;
    short bestMove = 0;
    for (short move : position.getAllMoves()) {
        try {
            position.doMove(move);
            // start from a minValue search
            int temp = minValue(position, 1, maxDepth);
            // compare to current max value
            if (maxVal <= temp){
                maxVal = temp;
                bestMove = move;
            }
            position.undoMove();
        } catch (IllegalMoveException e) {
            e.printStackTrace();
        }
    }
    // if we find that we will definitely win or definitely lose
    // then we know we don't need to search in a deeper depth
    if((maxVal == WIN_UTILITY || maxVal == LOSE_UTILITY))
        stopSearch = true;
    return bestMove;
}

private int minValue(Position position, int depth, int maxDepth) {
    if (cutoffTest(position, depth, maxDepth))
        return utility(position);
    int v = Integer.MAX_VALUE;
    for (short move : position.getAllMoves()) {
        try{
            position.doMove(move);
            v = Math.min(v, maxValue(position, depth + 1, maxDepth));
            position.undoMove();
        }catch (IllegalMoveException e){
            e.printStackTrace();
        }
    }
    return v;
}

// maxValue is same as minValue
private int maxValue(Position position, int depth, int maxDepth) {}

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

We need to check whether we should stop searching or not at the beginning of maxValue and minValue. The function cutoffTest helps us to check it and its implementation is very simple. If we find that the search should be stopped, we then return a value (utility), which represents the evaluation of this move.

2.2 Utility function

The implementation of utility function is very simple. If the game reaches a terminal state, in this case, is win/lose/draw, this function will return a default value to represent the end of the game:

1. max integer if AI wins. (obviously, winning is the most important)

2. min integer if AI loses . (vice versa)

3. zero if it's a draw.

Otherwise, it will return the evaluation of this move.

```
private int utility(Position position) {
    if (position.isTerminal()) {
        if (position.isStaleMate())
            return DRAW_UTILITY; // Integer.Min
        else if (position.isMate() && AI == position.getToPlay())
            return LOSE_UTILITY; // Integer.Max
        else if (position.isMate() && AI != position.getToPlay())
            return WIN_UTILITY; // 0
    }
    return evaluation(position);
}
```

2.3 Evaluation function

There are two parts of the chess game evaluation function, simple piece values and piece-square values:

1. Piece values: we give different values to different pieces (such as 100 for a pawn and 20000 for a king), and calculate each piece's value on the board for both sides. The `Chesspresso lib` already has a material value heuristic function `Position.getMaterial()`. And the evaluation function in this problem could be very simple.

```
private int evaluation(Position position) {
    if(AI == position.getToPlay())
        return position.getMaterial();
    else
        return position.getMaterial() * -1;
}
```

2. Piece-square values: it is not very accurate only using piece values since the position of different pieces also has influence on evaluation. For example, the heuristic value of a rook's position could be like that:

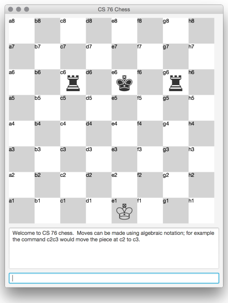
```
private static int[] RookTable = new int[] {
            0,  0,  0,  0,  0,  0,  0,  0,
            5, 10, 10, 10, 10, 10, 10,  5,
           -5,  0,  0,  0,  0,  0,  0, -5,
           -5,  0,  0,  0,  0,  0,  0, -5,
           -5,  0,  0,  0,  0,  0,  0, -5,
           -5,  0,  0,  0,  0,  0,  0, -5,
           -5,  0,  0,  0,  0,  0,  0, -5,
            0,  0,  0,  5,  5,  0,  0,  0,
    };
```

We'd better give bonuses for pieces standing well and penalties for pieces standing badly. I've implemented a position based evaluation function.

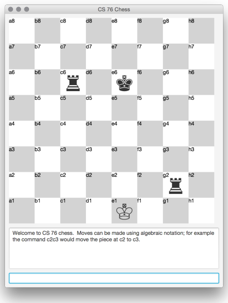
...Found that my piece values && piece-square values evaluation works even worse than a normal piece values heuristic. Cannot find why. I used normal piece values heuristic function to test all cases. The implementation is in `MiniMaxAlphaBetaAI.java`.

2.4 Testing

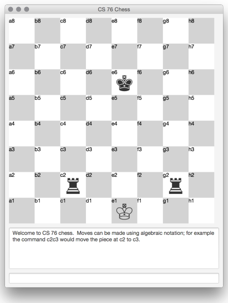
`Test 1:` This is a simple test where black has two rooks, and white has one king. Black will move first and win the game. Search depth is 5.



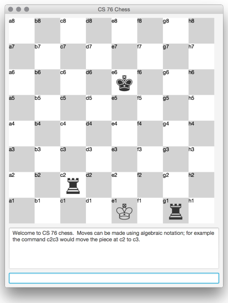
Welcome to CS 76 chess. Moves can be made using algebraic notation; for example the command c2c3 would move the piece at c2 to c3.



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.

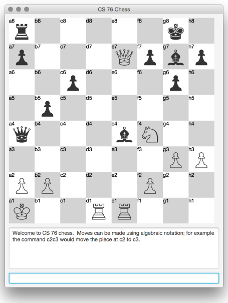


Welcome to CS 76 chess. Moves can be made using algebraic notation; for example the command c2c3 would move the piece at c2 to c3.

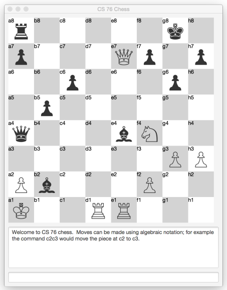


Welcome to CS 76 chess. Moves can be made using algebraic notation; for example the command c2c3 would move the piece at c2 to c3.

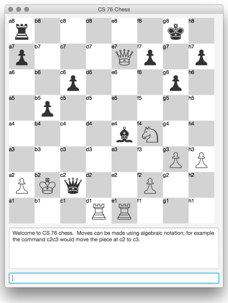
`Test 2:` In this case, black will move first and win the game. Search depth is 5.



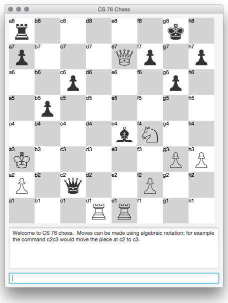
Welcome to CS 76 chess. Moves can be made using algebraic notation; for example the command c2c3 would move the piece at c2 to c3.



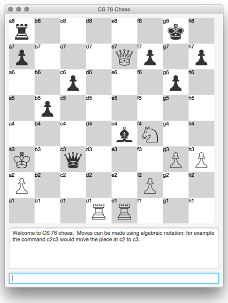
Welcome to CS 76 chess. Moves can be made using algebraic notation; for example the command c2c3 would move the piece at c2 to c3.



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.



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.



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.

There are more test cases in my code and you could test them freely.

3 Alpha-Beta Pruning

3.1 Implementation

Alpha-beta pruning could decrease the number of searching in the minimax search. It stops searching if the current search could not generate a better solution in comparison with the previously examined paths. The new version of the minimax search with alpha-beta pruning is in `MiniMaxAlphaBetaAI.java`.

In this problem, this could be done by passing an alpha and beta value to next depth. These two numbers are initialized in `alphabetaSearch` as `Integer.MIN_VALUE` and `Integer.MAX_VALUE`.

```java
int temp = minValue(position, 1, maxDepth, Integer.MIN_VALUE, Integer.MAX_VALUE);
```

In each step of `maxValue` and `minValue`, we check if the return value has an influence of the upper bound and the lower bound or not, if so, we modify the value and return it.

```java
private int minValue(Position position, int depth, int maxDepth, int a, int b) {
    int alpha = a, beta = b;
    if (cutoffTest(position, depth, maxDepth))
        return utility(position);
    int maxVal = Integer.MAX_VALUE;
    for (short move : position.getAllMoves()) {
        try{
            position.doMove(move);
            maxVal = Math.min(maxVal, maxValue(position, depth + 1, maxDepth, alpha, beta));
            position.undoMove();
            // if max value is less than the lower bound, return it
            if(maxVal <= alpha)
                return maxVal;
            // if max value is less than the upper bound, change the upper bound to that value
            beta = Math.min(beta, maxVal);
        }catch (IllegalMoveException e){
            e.printStackTrace();
        }
    }
    return maxVal;
}
```

3.2 Testing

`Test 1`: Same as the test of minimax search without alpha-beta pruning. Below is the information of the first move. We could see that both of them are have same moves, but pruning would reduce the chance to visit node we don't need to visit again.

`Test 2`: Same as the test of minimax search without alpha-beta pruning. Below is the information of the first move. We could see that both of them are have same moves, but pruning would reduce the chance to visit node we don't need to visit again.

We could see that the effects of pruning are very significant. Actually, minimax search with alpha-beta pruning could search in deeper depth within the same time time while minimax search without alpha-beta pruning can only search in a shallower depth. We will apply alpha-beta search with a larger max depth in the next part: the comparison of with or without transposition table.

Algorithm 1 Table

`Without Pruning`

```
Search in depth 1
States visited is 27
The best move in this depth is 7532
Search in depth 2
States visited is 148
The best move in this depth is 7532
Search in depth 3
States visited is 3769
The best move in this depth is 7532
Search in depth 4
States visited is 20132
The best move in this depth is 7532
Reach end in depth 5. We will definitely Win this game.
Search in depth 5
States visited is 525885
The best move in this depth is 5038
The final move is: 5038
```

`With Pruning`

```
Search in depth 1
States visited is 27
The best move in this depth is 7532
Search in depth 2
States visited is 148
The best move in this depth is 7532
Search in depth 3
States visited is 1050
The best move in this depth is 7532
Search in depth 4
States visited is 2474
The best move in this depth is 7532
Reach end in depth 5. We will definitely Win this game.
Search in depth 5
States visited is 39059
The best move in this depth is 5038
The final move is: 5038
```

Algorithm 2 Table

`Without Pruning`

```
Search in depth 1
States visited is 41
The best move in this depth is -28456
Search in depth 2
States visited is 1742
The best move in this depth is 6492
Search in depth 3
States visited is 63943
The best move in this depth is 4760
Search in depth 4
States visited is 2597818
The best move in this depth is 4760
Reach end in depth 5. We will definitely Win this game.
Search in depth 5
States visited is 95879430
The best move in this depth is -28042
The final move is: -28042
```

`With Pruning`

```
Search in depth 1
States visited is 41
The best move in this depth is -28456
Search in depth 2
States visited is 1742
The best move in this depth is 6492
Search in depth 3
States visited is 23569
The best move in this depth is 4760
Search in depth 4
States visited is 313225
The best move in this depth is 4760
Reach end in depth 5. We will definitely Win this game.
Search in depth 5
States visited is 2676367
The best move in this depth is -28042
The final move is: -28042
```

4 Transposition table

4.1 Implementation

Another way to accelerate the search algorithm is to implement a transposition table. We could save <key (hash code of a position), value (an Entry)> pair in this table. If we have met this position in a shallower depth, we will just use this Entry's value instead of going on search deeper. This may take a lot of space but could reduce the run time of the whole algorithm.

In order to use hashtable to record position information. We write a new class to save both the value and depth of that position. The new class is in the `Entry.java`.

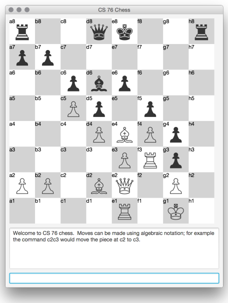
```
// need to save value and depth
public class Entry{
    public int value;
    public int depth;

    public Entry(int value, int depth){
        this.value = value;
        this.depth = depth;
    }

    public void setEntry(int value, int depth){
        this.value = value;
        this.depth = depth;
    }
}
```

In each turn, we create a new hashtable. In each step of `maxValue` and `minValue`, we check that if we have met this position or not. We only need to modify `maxValue` and `minValue`, below is the implementation in `maxValue`.

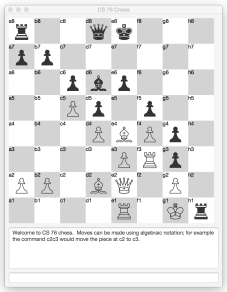
```
// use Transposition table
Long hashcode = position.getHashCode();
if(table.containsKey(hashcode)) {
    if(table.get(hashcode).depth < depth) {
        minVal = table.get(hashcode).value;
    } else {
        minVal = Math.max(minVal, minValue(position, depth + 1, maxDepth, alpha, beta));
        table.get(hashcode).setEntry(minVal, depth);
    }
} else {
    minVal = Math.max(minVal, minValue(position, depth + 1, maxDepth, alpha, beta));
    table.put(position.getHashCode(), new Entry(minVal, depth));
}
```

4.2 Testing

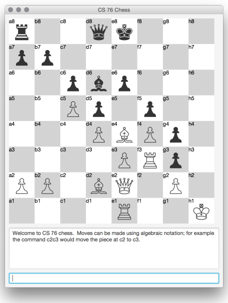
`Test 3`: In this case, black will move first and win the game. Search depth is 5 and 8 (both have same result).



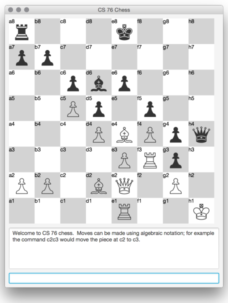
Welcome to CS 76 chess. Moves can be made using algebraic notation; for example the command c2c3 would move the piece at c2 to c3.



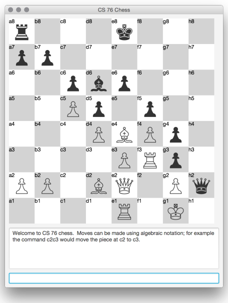
Welcome to CS 76 chess. Moves can be made using algebraic notation; for example the command c2c3 would move the piece at c2 to c3.



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.



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.



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.

We could see that transposition table does improve the efficiency but the effects are not statistically significant. We also found that if the search depth is larger than 7 in a game which has lots of possible moves(many pieces on board), transposition table will take up significant memory in the last few depths (and it took so much time to find a best move). But, if we keep cleaning the hash table, then how can it make use to the whole search? We think another way to use transposition table is record visited positions during different moves. But if two AIs are playing against each other, then we need to maintain two hashtable and it will also occupy lots of memory.

Algorithm 3 Table

`Without Table, Search Depth 5`

```
Search in depth 1
States visited is 41
The best move in this depth is -27298
Search in depth 2
States visited is 1241
The best move in this depth is -27298
Search in depth 3
States visited is 37348
The best move in this depth is -27298
Search in depth 4
States visited is 326772
The best move in this depth is 4607
Reach end in depth 5.
Search in depth 5
States visited is 8377561
The best move in this depth is 4607
Table size after this search: 0
The final move is: 4607
```

`With Table, Search Depth 5`

```
Search in depth 1
States visited is 41
The best move in this depth is -27298
Search in depth 2
States visited is 1241
The best move in this depth is -27298
Search in depth 3
States visited is 37348
The best move in this depth is -27298
Search in depth 4
States visited is 323558
The best move in this depth is 4607
Reach end in depth 5.
Search in depth 5
States visited is 7984788
The best move in this depth is 4607
Table size after this search: 2675999
The final move is: 4607
```

Algorithm 4 Table

```
Without Table, Search Depth 8
______________________________
Search in depth 1
States visited is 41
The best move in this depth is -27298
Search in depth 2
States visited is 1241
The best move in this depth is -27298
Search in depth 3
States visited is 37348
The best move in this depth is -27298
Search in depth 4
States visited is 326772
The best move in this depth is 4607
Reach end in depth 5.
Search in depth 5
States visited is 8377561
The best move in this depth is 4607
Search in depth 6
States visited is 38348737
The best move in this depth is -27298
Reach end in depth 7.
Search in depth 7
States visited is 1681390034
The best move in this depth is 4607
The final move is: 4607
```

```
With Table, Search Depth 8
______________________________
Search in depth 1
States visited is 41
The best move in this depth is -27298
Search in depth 2
States visited is 1241
The best move in this depth is -27298
Search in depth 3
States visited is 37348
The best move in this depth is -27298
Search in depth 4
States visited is 323558
The best move in this depth is 4607
Reach end in depth 5.
Search in depth 5
States visited is 7984788
The best move in this depth is 4607
Search in depth 6
States visited is 35605645
The best move in this depth is -27298
_______________________________-
Failed
Cannot find a solution in a similar amount of time
Take too much time and memory
_______________________________-
```