Maulana Azad National Institute of Technology

(An Institute of National Importance)
Bhopal – 462003 (India)

Department of Computer Science & Engineering

A I PROJECT REPORT

Chess AI using Minimax and Alpha-beta Pruning

Jishan Shaikh	161112013	jishanshaikh9893@gmail.com
Ankit Chouhan	161112051	ankitchouhan.dw1@gmail.com
Saurabh Jha	161112008	saurabhjhajha5@gmail.com
Senjeet Gautam	161112056	sonukumarpcm@gmail.com

Supervisors and Reviewers

Dr. Nilay Khare
Dr. Mansi Gyanchandani
Department of CSE, MANIT Bhopal

Session 2018-19

Contents

Chapter 1
Introduction

1.1 Chess

Chess is a two player strategy game. It remains the state of the art solving problem from past century so far. Computer chess is one of the most researched fields within AI, machine learning has not been successful yet at producing grandmaster level players. Top computer engines select features manually and after years of hard work results in a strong rated engine. They tune their evaluation function based on that. The complexity of the game makes it infeasible for computers to calculate each possible move in complete game, hence some computer engines use brute force techniques at certain depths for less time complexities. Still, it takes huge amount of time. Some techniques for improving brute force evaluation of chess moves search tree are minimax and alpha-beta pruning.

1.2 Methodology

In this project, we implemented simple minimax algorithm and Alpha-beta pruning to brute force moves from a certain position to produce a descent computer chess player. For this we used two libraries chess.js for move generation and chessboard.js for board representation. The evaluation function used is based on positional matrices and piece values (Pawn = 10, Knight/Bishop = 30, Rook = 50, Queen = 90, and King = 900). This evaluation function is applied to depth limited depth first search. The expected ELO rating of developed program is 1300-1900 for depth 3-5.

1.3 Problem Statement

Develop a computer chess program based on minimax algorithm and alpha-beta pruning applied on game tree. Program must able to generate and play best moves according to its evaluation. There should be an option to limit the depth upto which is should check for move (Usually < 10). Design an evaulation function based on position of pieces on the board e.g. Knight on a1 should have lesser value than Knight on e5 due to positional advantage.

Create a simple user interface for playing a human against it, at certain depth according to human's input. Interface may be based on web, mobile, or command line.

Chapter 2
Literature Survey

There are plenty of application in AI but games are the most important thing in today's world and nowadays every where else comes with two player games like chess, checkers etc. There are algorithm that help to design these games. These algorithm not only just help in making games but they also help in making the life of the player (i.e. user) tough in the game-play. Minimax Algorithm and Alpha-beta pruning techniques are such kind of methods and techniques.

2.1 Minimax Algorithm

The Min-Max (Minimax) algorithm is applied in two player games, such as tic-tac-toe, checkers, chess, go, and so on as it helps us to find the accurate values of the board position. All these games have at least one thing in common, they are logic games. This means that they can be described by a set of rules and premises. With them, it is possible to know from a given point in the game, what are the next available moves. So they also share other characteristic, they are 'full information games'. Each player knows everything about the possible moves of the adversary, so we assume that the player will always try to play his/her best move. Minimax is a recursive algorithm which is used to choose an optimal move for a player assuming that the other player is also playing optimally.

The values here represents how good a move is. So the MAX player will try to select the move with highest value in the end. But the MIN player also has something to say about it and he will try to select the moves that are better to him, thus minimizing MAX's outcome.

How does the Algorithm work?

Our goal is to find the best move for the player. To do so, we can just choose the node with best evaluation score. To make the process smarter, we can also look ahead and evaluate potential opponent's moves. Technically, we start with the root node and choose the best possible node. We evaluate nodes based on their evaluation scores. In our case, evaluation function can assign scores to only result nodes (leaves). Therefore, we recursively reach leaves with scores and back propagate the scores.

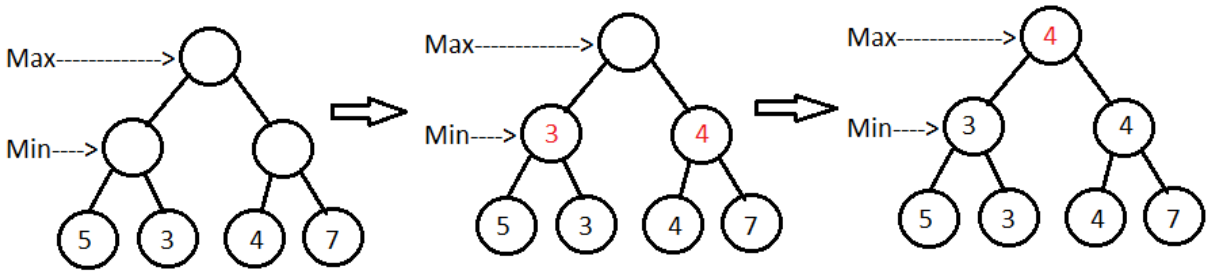

Figure 1: Minimax (Min-Max) example at depth = 2.

1. Construct the complete game tree
2. Evaluate scores for leaves using the evaluation function
3. Back-up scores from leaves to root, considering the player type:
    - For max player, select the child with the maximum score
    - For min player, select the child with the minimum score
4. At the root node, choose the node with max value and perform the corresponding move

2.2 Alpha-Beta Pruning

If we apply alpha-beta pruning to a standard minimax algorithm, it returns the same move as the standard one, but it removes (prunes) all the nodes that are possibly not affecting the final decision. Alpha-beta pruning is based on the Branch and bound algorithm design paradigm, where we will generate uppermost and lowermost possible values to our optimal solution and using them, discard any decision which cannot possibly yield a better solution than the one we have so far.

Alpha: It is the best choice so far for the player MAX. We want to get the highest possible value here.

Beta: It is the best choice so far for MIN, and it has to be the lowest possible value.

How does alpha-beta pruning work?

1. Initialize alpha = -infinity and beta = infinity as the worst possible cases. The condition to prune a node is when alpha becomes greater than or equal to beta.
2. Start with assigning the initial values of alpha and beta to root and since alpha is less than beta we don't prune it.

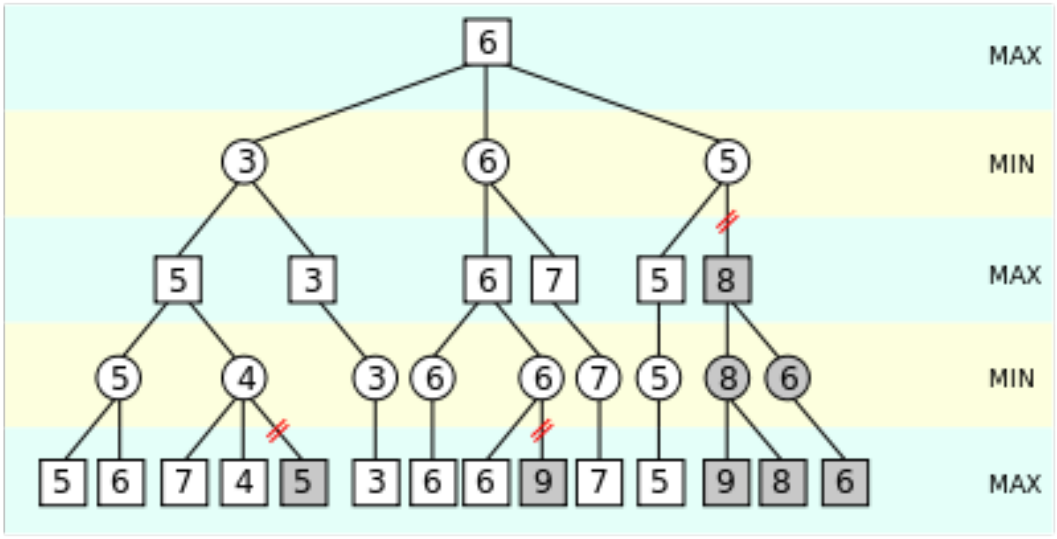

Figure 2: Alpha-beta Pruning example at depth = 4.

3. Carry these values of alpha and beta to the child node on the left. And now from the utility value of the terminal state, we will update the values of alpha and be, so we don't have to update the value of beta. Again, we don't prune because the condition remains the same. Similarly, for the third child node also. And then backtracking to the root we set alpha=3 because that is the minimum value that alpha can have.
4. Now, alpha=3 and beta=infinity at the root. So, we don't prune. Carrying this to the center node, and calculating MIN{2, infinity}, we get alpha=3 and beta=2.
5. Prune the second and third child nodes because alpha is now greater than beta.
6. Alpha at the root remains 3 because it is greater than 2. Carrying this to the rightmost child node, evaluate MIN{infinity,2}=2. Update beta to 2 and alpha remains 3.
7. Prune the second and third child nodes because alpha is now greater than beta.
8. Hence, we get 3, 2, 2 at the left, center, and right MIN nodes, respectively. And calculating MAX{3,2,2}, we get 3.

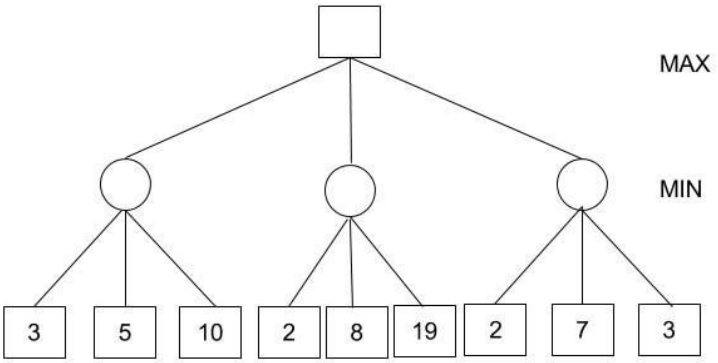

Figure 3: Alpha-beta pruning explained example.

Chapter 3
Code and Implementation

3.1 Code

3.1.1 HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <link rel="stylesheet" href="lib/chessboardjs/css/chessboard-0.3.0.css">
   <link rel="stylesheet" href="style.css">
</head>
<body>
<div id="board" class="board"></div>
<div class="info">
   Search depth:
   <select id="search-depth">
      <option value="1">1</option>
      <option value="2">2</option>
      <option value="3" selected>3</option>
      <option value="4">4</option>
      <option value="5">5</option>
   </select>
   <br>
   <span>Positions evaluated: <span id="position-count"></span></span>
   <br>
   <span>Time: <span id="time"></span></span>
   <br>
   <span>Positions/s: <span id="positions-per-s"></span> </span>
   <br>
   <br>
   <div id="move-history" class="move-history">
   </div>
</div>
<script src="lib/jquery/jquery-3.2.1.min.js"></script>
<script src="lib/chessboardjs/js/chess.js"></script>
<script src="lib/chessboardjs/js/chessboard-0.3.0.js"></script>
<script src="script.js"></script>
<script>
</script>
</body>
</html>
```

3.1.2 CSS

```
.board{
```

```
    width: 500px;
    margin: auto
}
.info {
    width: 500px;
    margin: auto;
}
.move-history {
    max-height: 200px;
    overflow-y: scroll;
}
```

3.1.3 JavaScript (ONLY MINIMAXTop AND ALPHA-BETA FUNCTION)

```
var minimaxTop = function(depth, game, isMAX) {
    var totalMoves = game.ugly_moves();      // Generating total moves from current position, function from chess.js
    var bestMove = -9999999;
    var bestMoveFound;
    for(var i = 0; i < totalMoves.length; i++) {            // Iterating through all moves generated
        var newGameMove = totalMoves[i]
        game.ugly_move(newGameMove);
        var value = minimax(depth - 1, game, -10000, 10000, !isMAX);     // Applying minimax at one depth lower
        game.undo();            // Recovering original state
        if(value >= bestMove) {         // Adjusting bestMove values
            bestMove = value;
            bestMoveFound = newGameMove;
        }
    }
    return bestMoveFound;
};

// ASSUMPTION: Opponent plays close to perfect moves.
var minimax = function (depth, game, alpha, beta, isMAX) {
    positionCount++;
    if (depth === 0) {
        return -evaluation(game.board());       // Evaluation function for a particular state
    }
    var totalMoves = game.ugly_moves();
    if (isMAX) {     // Maximizing value for first player; MAX
        var bestMove = -9999999;
        for (var i = 0; i < totalMoves.length; i++) {
            game.ugly_move(totalMoves[i]);
            bestMove = Math.max(bestMove, minimax(depth - 1, game, alpha, beta, !isMAX));  // Applying minimax at d-1
            game.undo();
            alpha = Math.max(alpha, bestMove);            // Selecting max value
            if (beta <= alpha) {            // Condition of pruning
                return bestMove;
            }
        }
        return bestMove;
    } else {          // Minimizing value for second player; MIN
        var bestMove = 9999999;
```

```
        for (var i = 0; i < totalMoves.length; i++) {
            game.ugly_move(totalMoves[i]);
            bestMove = Math.min(bestMove, minimax(depth - 1, game, alpha, beta, !isMAX));    // Applying minimax at d-1
            game.undo();
            beta = Math.min(beta, bestMove);   // Selecting min value
            if (beta <= alpha) {            // Condition of pruning
                return bestMove;
            }
        }
        return bestMove;
    }
};
```

3.2 Implementational Screenshots

Search depth: 3
Positions evaluated: 15361
Time: 1.156s
Positions/s: 13288.062283737025

g3 Nf6

Figure 5: Playing Kings Indian Attack.

Search depth: 3
Positions evaluated: 12670
Time: 0.861s
Positions/s: 14715.447154471545

c4 Nf6

Figure 4: Playing English opening.

Search depth: 3
Positions evaluated: 14457
Time: 1.453s
Positions/s: 9949.759119064005

d4 Nc6

Figure 7: Playing Queen's Gambit.

Search depth: 3
Positions evaluated: 17834
Time: 1.208s
Positions/s: 14763.245033112582

e4 Nf6

Figure 6: Attempting scholar's mate.

Chapter 4
Results

4.1 Results summary

The developed computer engine is able to play chess at sophomore level (Expected ELO 1300-1900 for depth = 3-5)[1]. Yet it is far away from master's level, it shows quite descent results for its level. Occassionally it sacrifices material for positional or tactical advantage or threats such as mating threat. It has been played against multiple times and it shows exceptionally strong endgame strategies for rook-pawn endgames, queen-pawn endgames, king-pawn endgames, etc for depth = 4 and 5. It plays close to accurate moves in middlegame and didn't blunder more than once in a single game for depth = 4.

We have played some popular openings against it and it plays very well without any blunders. It makes use of optimistism much; whenever opponent blunders, it creates additional advantages for it. It does not let opponent takes advantage in famous gambits such as Queen's gambit, Scotch gambit, etc. It is tactically more stronger than its expected ELO rating player, it usually plays for *pinning* (Holding down a piece against queen/rook/king) and *forking* (create multiple threats in single move time). From black's position, it usually plays attacking chess starts from beginning. It's favorite openings are four knight's game, symmetric English, symmetric reti, petrov's defense, nimzo-indian defense, etc. which are also considered as good at master's level.

Since it is based on very simple methods it has certains disadvantages too for lower depths (depth = 1, 2, 3) e.g. it does not follows basic chess principles used to train beginner chess players such as controlling the center based on positional advantage, not moving a single piece twice in the opening phase of game, early castling, and not exposing the queen early. It usually takes more time at depth = 5 and 4, as there is no restriction against it for brute force searching other than alpha-beta pruning.

1 **ELO** 2300-2400 FIDE Master, 2400-2500 International Master, 2500-2700 Grand Master.