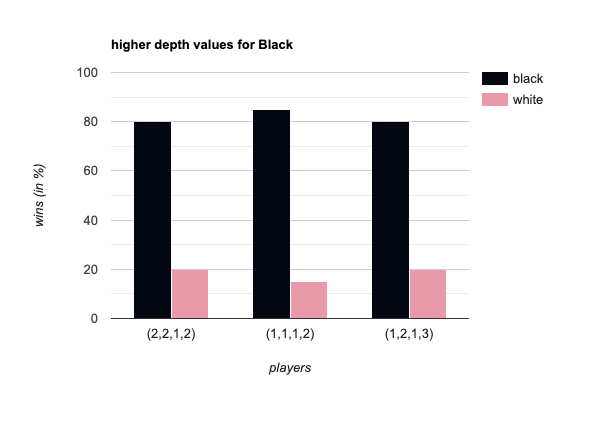
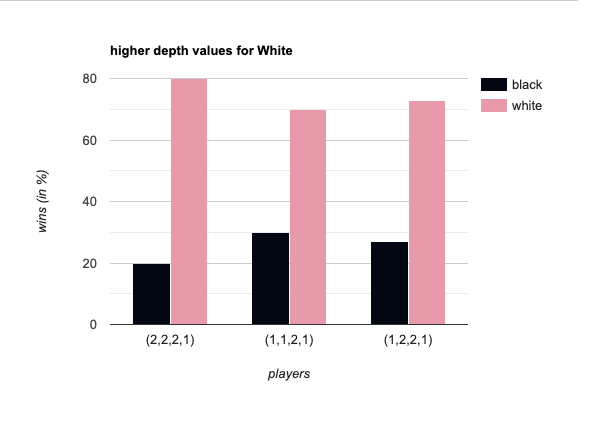
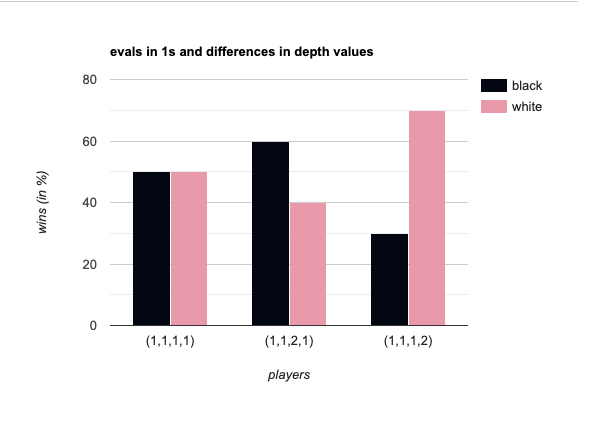
Project 1 – Chess

Hafsa Chaudhry

October 31, 2019

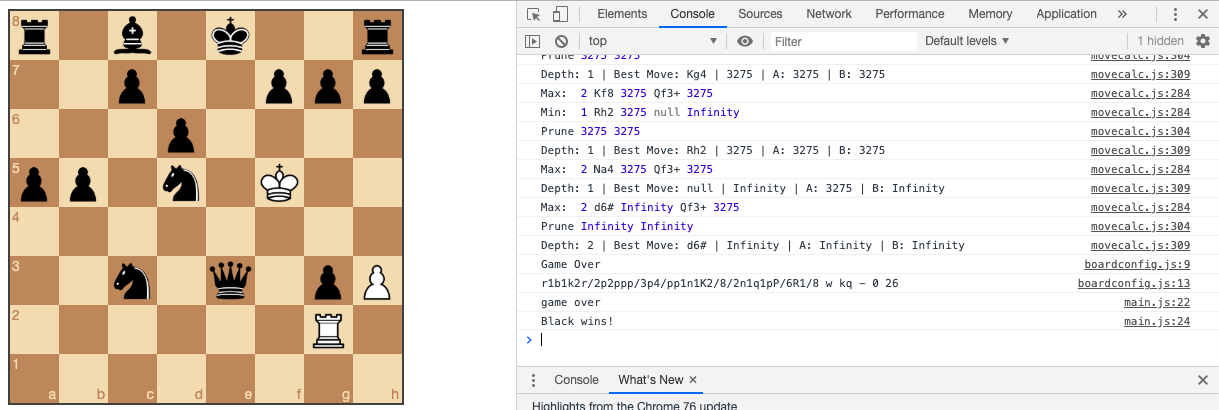
**Graphs:**

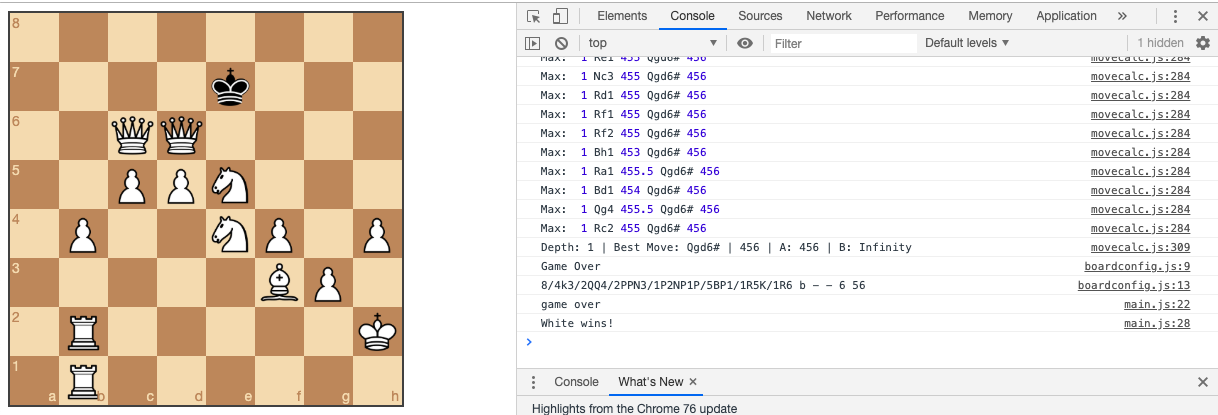




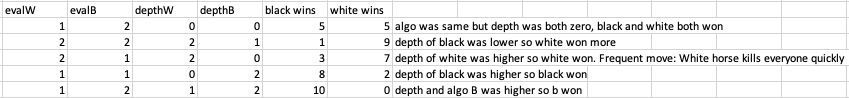
**Findings:**

While exploring the different combinations of depths and evals, I came to the conclusion that depth plays a higher significance in “who wins” then evaluation. While running multiple game rounds, I noticed that when my combination was something like, let’s say (1,1,1,2) , then the number of times that black won was substantially higher. Similarly, when my combination was (1,1,2,1), then the number of times white won was substantially higher. At first I was worried that something was wrong with my code because even after repeating the same combination 10x times, the numbers were still consistent to the team that had a higher depth value. I was only reassured my code was not broken when I changed the combination up and finally saw that the other color team was finally also able to have a higher winning streak. This was when I made my conclusion.

**Screenshots:** 



**Excel sheet with some examples and commentary:**



**My edited code for main:**

// Computer makes a move with algorithm choice and skill/depth level

var makeMove = function(algo, skill=3, moves=3) {

// exit if the game is over

if (game.game\_over() === true) {

console.log('game over');

return;

}

// Calculate the best move, using chosen algorithm

var move = calcBestMove(algo,skill, game, game.turn())[1];

// Make the calculated move

game.move(move);

// Update board positions

board.position(game.fen());

}

// Computer vs Computer

var playGame = function(algo=4, skillW=2, skillB=2,algoW=2,algoB=2) {

if (game.game\_over() === true) {

console.log('game over');

return;

}

var moves = game.turn() === 'w' ? algo : algoB;

var skills = game.turn() === 'w' ? skillW : skillB;

makeMove(algo,moves,skill);

window.setTimeout(function() {

playGame(algo, skillW, skillB, algoW, algoB);

}, 250);

};

// Handles what to do after human makes move.

// Computer automatically makes next move

var onDrop = function(source, target) {

// see if the move is legal

var move = game.move({

from: source,

to: target,

promotion: 'q' // NOTE: always promote to a queen for example simplicity

});

// If illegal move, snapback

if (move === null) return 'snapback';

// Log the move

console.log(move)

// make move for black

window.setTimeout(function() {

makeMove(4, 3);

}, 250);

};

**My edited code for movecalc**

/\*\*

\* Finds a random move to make

\* @return {string} move to make

\*/

var randomMove = function() {

var possibleMoves = game.moves();

var randomIndex = Math.floor(Math.random() \* possibleMoves.length);

return possibleMoves[randomIndex];

};

/\*\*

\* Evaluates current chess board relative to player

\* @param {string} color - Players color, either 'b' or 'w'

\* @return {Number} board value relative to player

\*/

var evaluateBoard1 = function(board, color) {

// Sets the value for each piece using standard piece value

var pieceValue = {

'p': 100,

'n': 350,

'b': 350,

'r': 525,

'q': 1000,

'k': 10000

};

// Loop through all pieces on the board and sum up total

var value = 0;

board.forEach(function(row) {

row.forEach(function(piece) {

if (piece) {

// Subtract piece value if it is opponent's piece

value += pieceValue[piece['type']]

\* (piece['color'] === color ? 1 : -1);

}

});

});

return value;

};

/\*\*

\* Evaluates current chess board relative to player

\* @param {string} color - Players color, either 'b' or 'w'

\* @return {Number} board value relative to player

\*/

var evaluateBoard2 = function (board) {

var totalEvaluation = 0;

for (var i = 0; i < 8; i++) {

for (var j = 0; j < 8; j++) {

totalEvaluation = totalEvaluation + getPieceValue(board[i][j], i ,j);

}

}

return totalEvaluation;

};

var reverseArray = function(array) {

return array.slice().reverse();

};

var pawnEvalWhite =

[

[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],

[5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0],

[1.0, 1.0, 2.0, 3.0, 3.0, 2.0, 1.0, 1.0],

[0.5, 0.5, 1.0, 2.5, 2.5, 1.0, 0.5, 0.5],

[0.0, 0.0, 0.0, 2.0, 2.0, 0.0, 0.0, 0.0],

[0.5, -0.5, -1.0, 0.0, 0.0, -1.0, -0.5, 0.5],

[0.5, 1.0, 1.0, -2.0, -2.0, 1.0, 1.0, 0.5],

[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

];

var pawnEvalBlack = reverseArray(pawnEvalWhite);

var knightEval =

[

[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0],

[-4.0, -2.0, 0.0, 0.0, 0.0, 0.0, -2.0, -4.0],

[-3.0, 0.0, 1.0, 1.5, 1.5, 1.0, 0.0, -3.0],

[-3.0, 0.5, 1.5, 2.0, 2.0, 1.5, 0.5, -3.0],

[-3.0, 0.0, 1.5, 2.0, 2.0, 1.5, 0.0, -3.0],

[-3.0, 0.5, 1.0, 1.5, 1.5, 1.0, 0.5, -3.0],

[-4.0, -2.0, 0.0, 0.5, 0.5, 0.0, -2.0, -4.0],

[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]

];

var bishopEvalWhite = [

[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0],

[ -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0],

[ -1.0, 0.0, 0.5, 1.0, 1.0, 0.5, 0.0, -1.0],

[ -1.0, 0.5, 0.5, 1.0, 1.0, 0.5, 0.5, -1.0],

[ -1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, -1.0],

[ -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0],

[ -1.0, 0.5, 0.0, 0.0, 0.0, 0.0, 0.5, -1.0],

[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]

];

var bishopEvalBlack = reverseArray(bishopEvalWhite);

var rookEvalWhite = [

[ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],

[ 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.5],

[ -0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5],

[ -0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5],

[ -0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5],

[ -0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5],

[ -0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5],

[ 0.0, 0.0, 0.0, 0.5, 0.5, 0.0, 0.0, 0.0]

];

var rookEvalBlack = reverseArray(rookEvalWhite);

var evalQueen = [

[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0],

[ -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0],

[ -1.0, 0.0, 0.5, 0.5, 0.5, 0.5, 0.0, -1.0],

[ -0.5, 0.0, 0.5, 0.5, 0.5, 0.5, 0.0, -0.5],

[ 0.0, 0.0, 0.5, 0.5, 0.5, 0.5, 0.0, -0.5],

[ -1.0, 0.5, 0.5, 0.5, 0.5, 0.5, 0.0, -1.0],

[ -1.0, 0.0, 0.5, 0.0, 0.0, 0.0, 0.0, -1.0],

[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]

];

var kingEvalWhite = [

[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],

[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],

[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],

[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],

[ -2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],

[ -1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],

[ 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 2.0, 2.0 ],

[ 2.0, 3.0, 1.0, 0.0, 0.0, 1.0, 3.0, 2.0 ]

];

var kingEvalBlack = reverseArray(kingEvalWhite);

var getPieceValue = function (piece, x, y) {

if (piece === null) {

return 0;

}

var getAbsoluteValue = function (piece, isWhite, x ,y) {

if (piece.type === 'p') {

return 10 + ( isWhite ? pawnEvalWhite[y][x] : pawnEvalBlack[y][x] );

} else if (piece.type === 'r') {

return 50 + ( isWhite ? rookEvalWhite[y][x] : rookEvalBlack[y][x] );

} else if (piece.type === 'n') {

return 30 + knightEval[y][x];

} else if (piece.type === 'b') {

return 30 + ( isWhite ? bishopEvalWhite[y][x] : bishopEvalBlack[y][x] );

} else if (piece.type === 'q') {

return 90 + evalQueen[y][x];

} else if (piece.type === 'k') {

return 900 + ( isWhite ? kingEvalWhite[y][x] : kingEvalBlack[y][x] );

}

throw "Unknown piece type: " + piece.type;

};

var absoluteValue = getAbsoluteValue(piece, piece.color === 'w', x ,y);

return piece.color === 'w' ? absoluteValue : -absoluteValue;

};

/\*\*

\* Calculates the best move looking one move ahead

\* @param {string} playerColor - Players color, either 'b' or 'w'

\* @return {string} the best move

\*/

var calcBestMoveOne = function(playerColor) {

// List all possible moves

var possibleMoves = game.moves();

// Sort moves randomly, so the same move isn't always picked on ties

possibleMoves.sort(function(a, b){return 0.5 - Math.random()});

// exit if the game is over

if (game.game\_over() === true || possibleMoves.length === 0) return;

// Search for move with highest value

var bestMoveSoFar = null;

var bestMoveValue = Number.NEGATIVE\_INFINITY;

possibleMoves.forEach(function(move) {

game.move(move);

var moveValue = evaluateBoard(game.board(), playerColor);

if (moveValue > bestMoveValue) {

bestMoveSoFar = move;

bestMoveValue = moveValue;

}

game.undo();

});

return bestMoveSoFar;

}

/\*\*

\* Calculates the best move using Minimax without Alpha Beta Pruning.

\* @param {Number} depth - How many moves ahead to evaluate

\* @param {Object} game - The game to evaluate

\* @param {string} playerColor - Players color, either 'b' or 'w'

\* @param {Boolean} isMaximizingPlayer - If current turn is maximizing or minimizing player

\* @return {Array} The best move value, and the best move

\*/

var calcBestMoveNoAB = function(depth, game, playerColor, isMaximizingPlayer=true) {

// Base case: evaluate board

if (depth === 0) {

value = evaluateBoard(game.board(), playerColor);

return [value, null]

}

// Recursive case: search possible moves

var bestMove = null; // best move not set yet

var possibleMoves = game.moves();

// Set random order for possible moves

possibleMoves.sort(function(a, b){return 0.5 - Math.random()});

// Set a default best move value

var bestMoveValue = isMaximizingPlayer ? Number.NEGATIVE\_INFINITY : Number.POSITIVE\_INFINITY;

// Search through all possible moves

for (var i = 0; i < possibleMoves.length; i++) {

var move = possibleMoves[i];

// Make the move, but undo before exiting loop

game.move(move);

// Recursively get the value of this move

value = calcBestMoveNoAB(depth-1, game, playerColor, !isMaximizingPlayer)[0];

// Log the value of this move

console.log(isMaximizingPlayer ? 'Max: ' : 'Min: ', depth, move, value, bestMove, bestMoveValue);

if (isMaximizingPlayer) {

// Look for moves that maximize position

if (value > bestMoveValue) {

bestMoveValue = value;

bestMove = move;

}

} else {

// Look for moves that minimize position

if (value < bestMoveValue) {

bestMoveValue = value;

bestMove = move;

}

}

// Undo previous move

game.undo();

}

// Log the best move at the current depth

console.log('Depth: ' + depth + ' | Best Move: ' + bestMove + ' | ' + bestMoveValue);

// Return the best move, or the only move

return [bestMoveValue, bestMove || possibleMoves[0]];

}

/\*\*

\* Calculates the best move using Minimax with Alpha Beta Pruning.

\* @param {Number} depth - How many moves ahead to evaluate

\* @param {Object} game - The game to evaluate

\* @param {string} playerColor - Players color, either 'b' or 'w'

\* @param {Number} alpha

\* @param {Number} beta

\* @param {Boolean} isMaximizingPlayer - If current turn is maximizing or minimizing player

\* @return {Array} The best move value, and the best move

\*/

var calcBestMove = function(evals, depth, game, playerColor,alpha=Number.NEGATIVE\_INFINITY, beta=Number.POSITIVE\_INFINITY, isMaximizingPlayer=true) {

// Base case: evaluate board

if (depth === 0) {

if (evals === 1) {

value = evaluateBoard1(game.board(), playerColor);

}

else if (evals === 2) {

value = evaluateBoard2(game.board());

}

        else {

            value = evaluateBoard3(game.board());

        }

return [value, null];

}

// Recursive case: search possible moves

var bestMove = null; // best move not set yet

var possibleMoves = game.moves();

// Set random order for possible moves

possibleMoves.sort(function(a, b){return 0.5 - Math.random()});

// Set a default best move value

var bestMoveValue = isMaximizingPlayer ? Number.NEGATIVE\_INFINITY : Number.POSITIVE\_INFINITY;

// Search through all possible moves

for (var i = 0; i < possibleMoves.length; i++) {

var move = possibleMoves[i];

// Make the move, but undo before exiting loop

game.move(move);

// Recursively get the value from this move

value = calcBestMove(evals,depth-1, game, playerColor, alpha, beta, !isMaximizingPlayer)[0];

// Log the value of this move

console.log(isMaximizingPlayer ? 'Max: ' : 'Min: ', depth, move, value, bestMove, bestMoveValue);

if (isMaximizingPlayer) {

// Look for moves that maximize position

if (value > bestMoveValue) {

bestMoveValue = value;

bestMove = move;

}

alpha = Math.max(alpha, value);

} else {

// Look for moves that minimize position

if (value < bestMoveValue) {

bestMoveValue = value;

bestMove = move;

}

beta = Math.min(beta, value);

}

// Undo previous move

game.undo();

// Check for alpha beta pruning

if (beta <= alpha) {

console.log('Prune', alpha, beta);

break;

}

}

// Log the best move at the current depth

console.log('Depth: ' + depth + ' | Best Move: ' + bestMove + ' | ' + bestMoveValue + ' | A: ' + alpha + ' | B: ' + beta);

// Return the best move, or the only move

return [bestMoveValue, bestMove || possibleMoves[0]];

};