# CSC 490: A basic chess AI

Jonah Rankin
V00808910

April 21, 2017

## Introduction

Chess is an interesting programming problem because its complexity. There are approximately $10^{120}$ different positions which is far greater than the number of atoms in the universe ($\tilde{}10^{80}$). People have been theorizing and developing chess AI's since the dawn of computers [1].
The focus of this project is not to build the strongest AI but rather one that is intelligent and responds quickly. This project is an examination of the minimax algorithm, alpha-beta pruning, and how simple heuristics and board evaluation procedures can produce a chess AI which is able to challenge casual chess players.

## Technology

This project was implemented in JavaScript. The driving factors in the choice of language were ease of implementing the GUI, personal familiarity, availability of move generations libraries and portability. The inspiration for this project was provided by the blog post "A step-by-step guide to building a simple chess AI"[2] and its the source code provided on the project's github page[3].

### Chess.js

Chess.js is a JavaScript chess engine providing move generation, piece placement, and game over detection. It was difficult to find the best version of the library. The blog post used a non-standard version which exposes the internal move generation methods which were substantially faster but not human readable. The first bottleneck in performance was gathering the list of possible moves. The master branch of Chess.js only provides the `.moves()` method which was roughly one order slower than the `.ugly_moves()` method. The algorithms implemented in this project were able to process 5,000 to 10,000 positions per second. Ideally the average search time will be under 10 seconds permitting around 100,000 positions to be analyzed per move.

### Chessboard.js

Chessboard.js provides a chess board interface and when integrated with chess.js can provide a rich user experience. For this project the "Highlight Legal Moves" example[4] from the Chessboard.js documentation was the basis for the user interface.

## Minimax algorithm

The minimax algorithm is a basic way to model a competitive multi-agent environment. The search traverses a tree of possible game sequences depth first. At the leaves the utility is evaluated and values are propagated back to the root alternate between minimizing (min node) and maximizing (max node) the value of its children. For a tree with a branching factor B and depth D the number of nodes is $B^D$ [5]. For most applications the minimax algorithm is impractical because of the exponential increase in runtime. The branching factor for chess is approximately 35 [6].

## Alpha-beta pruning

Alpha-beta pruning is an optimization on minimax which prunes out sections of the tree that cannot be reached by optimal play. In the best case the number of nodes can be reduced to approximately $B^{D/2}$. Reaching the best case bound requires visiting the optimal path first.

## Board evaluation

All algorithms evaluate the board in the same way. By calculating the material value of the pieces on the board and additionally applying additional weighting based on the positioning using static arrays for each piece. The evaluation scheme is based on the "Simplified Evaluation Function" and "Point Value" articles on the chess programming wiki [7][8].

## Results

Due to time constraints it is difficult to quantify the differences between the four searches implemented. Ideally the algorithms would be evaluated over hundreds or thousands of iterations but each game takes at least 3 minutes to complete.

Table 1: Approximate performance of the four move search algorithms

| Algorithm | Search Depth | Avg. positions per turn | Avg. time |
|---|---|---|---|
| Minimax | 3 | 8000 | 1 |
| | 4 | 200000 | 30 |
| Alpha-beta basic | 3 | 10000 | 3 |
| | 4 | 48000 | 7 |
| Alpha-beta captures first | 3 | 2500 | 0.5 |
| | 4 | 45000 | 7 |
| Alpha-beta transposition | 3 | 4000 | 1 |
| | 4 | 15000 | 4 |

These statistics were gathered from a single game where both AI's were the same. The values were an approximate average of their average positions per turn and time to generate a move. These are not statistically significant but it does highlight the basic performance differences between the algorithms.
There is a trade-off between the number of positions that can be calculated per second and the complexity of the evaluation and heuristic. For smaller depth searches the benefits of pruning are not fully realized. There is also no benefit gained by iterative deepening at small search depths but

the benefits are significant even at a search depth of 4. When run test against each other there were a large number of drawn games where sides would repeat the same move until the game was over.

## Conclusion

The transposition table algorithm is buggy and occasionally makes sub-optimal moves. While the AI's are challenging for a casual player, i.e. myself, they perform poorly against each other by struggling in the end game to reach checkmate instead drawing due to move repetition. Some of the end game issues could be solved though a better evaluation function but there is a performance impact by making the evaluation more complex. To expand on the work details in this report there are many modifications to the search, Memory-enhanced Test Driver algorithm, MTD(f), [9] which also uses a transposition table and iterative deepening is used in many modern chess programs. Additionally, more sophisticated heuristics, such as the Killer and History heuristics, could improve node ordering and pruning.

## References


[1]"chessprogramming - Alan Turing", Chessprogramming.wikispaces.com, 2017. [Online]. Available: https://chessprogramming.wikispaces.com/Alan+Turing. [Accessed: 21- Apr- 2017].
[2]L. Hartikka, "A step-by-step guide to building a simple chess AI – freeCodeCamp", freeCodeCamp, 2017. [Online]. Available: https://medium.freecodecamp.com/simple-chess-ai-step-by-step-1d55a9
[Accessed: 21- Apr- 2017].
[3]2017. [Online]. Available: https://github.com/lhartikk/simple-chess-ai/blob/master/. [Accessed: 21- Apr- 2017].
[4]"chessboard.js » Examples", Chessboardjs.com, 2017. [Online]. Available: http://chessboardjs.com/examples#5003. [Accessed: 21- Apr- 2017].
[5]S. Russell and P. Norving, Artificial intelligence, 3rd ed. New Jersey: Pearson, 2010.
[6]"chessprogramming - Point Value", Chessprogramming.wikispaces.com, 2017. [Online]. Available: https://chessprogramming.wikispaces.com/Point+Value. [Accessed: 21- Apr- 2017].
[7]F. Laramée, L. Crazy and L. Crazy, "Chess Programming Part IV: Basic Search - Artificial Intelligence - Articles - Articles - GameDev.net", Gamedev.net, 2017. [Online]. Available: https://www.gamedev.net/resources/_/technical/artificial-intelligence/chess-programming-part-iv-basic-search
[Accessed: 21- Apr- 2017].
[8]"chessprogramming - Simplified evaluation function", Chessprogramming.wikispaces.com, 2017. [Online]. Available: https://chessprogramming.wikispaces.com/Simplified+evaluation+function. [Accessed: 21- Apr- 2017].
[9]"Aske Plaat: MTD(f), a new chess algorithm", People.csail.mit.edu, 2017. [Online]. Available: http://people.csail.mit.edu/plaat/mtdf.html#abmem. [Accessed: 21- Apr- 2017].