AI and Affective Computing Report

**An Optimised Web-Based Chess Engine Using Quiescence Search Algorithm**

**Name:** Abdulelah Alshehri (W14035393)

**Module:** Artificial Intelligence and Robotics/Artificial Intelligence and Affective Computing (CM0671)

**Module Tutor:** Dr. Li Zhang

**Course:** Computer Science (AI Pathway)

**Table of Contents**

[Introduction 3](#_Toc38283015)

[Development 3](#_Toc38283016)

[System Structure 3](#_Toc38283017)

[Algorithm design 4](#_Toc38283018)

[Implementation 6](#_Toc38283019)

[Testing 7](#_Toc38283020)

[Critical Evaluation 7](#_Toc38283021)

[Conclusion and Future Work 8](#_Toc38283022)

[References 9](#_Toc38283023)

# Introduction

The efforts to develop computer chess date back to the late 1940s. Nonetheless, programming a computer to understand the tact and philosophy behind chess moves remain on of the most challenging problems. Interestingly, both humans and computer depend on the ability to look ahead and react accordingly, humans tend to be selective of branches to consider, while, computers depend on their high computational power to explore as many branches as possible, in a brute-force like manner.

Due to the difficulty of producing reliable rules allowing computers to be more selective of branches to explore, without overlooking tactful branches have been a challenge. This project aims to provide a highly optimised, accessible and scalable chess engine without the need to download nor have a powerful machine to run.

# Development

As with most of this report, this section will mainly target development process of the implemented algorithm in the chess engine. However, it was thought to be best, that the chess engine as a whole and system structure be discussed to give a wholesome image on how the system functions and interacts with implemented search algorithm.

The developed Chess Engine is written purely using JavaScript, without the use of any libraries and ASP.NET framework which was used to implement a pleasing UI and RESTful APIs for further modularity and extensibility of the engine’s functionality and efficiency. Additionally, during the process of developing the engine, three main aspects were particularly targeted; Intelligence, Efficiency and Accessibility.

## System Structure

The system was designed to while following the paradigm of good software practices and keeping SOLID – a mnemonic acronym referring to five design principles; **S**ingle responsibility, **O**pen/closed, **L**iskov substitution, **I**nterface segregation and **D**ependency inversion, (Robert and Micah, 2006) – and Keep It Simple Stupid (KISS) design principles in mind.

The designed engine contains six main components, to allow for better modularity and readability of the engine’s structure. Each component and its main responsibilities are as follows (File names are used, as well as the file description provided were naming has been but abbreviated):

* **gameDfns.js:** The engine’s definitions file. It is responsible for the initialisation of all constants in game setting, pieces positions, player risk thresholds and pieces’ direction of movement.
* **gameBoard.js:** Sets up the game board logic including, creation of game pieces and game movement bounds.
* **gameGUI.js:** Matches the game board object to a graphical representation, manages graphical interactions as a result of a gameIO component event (selection and deselection of pieces) and supplies the user with game output and result.
* **gameIO.js:** Manages user input and output and translates it to the gameGUI component.
* **searchMove.js:** The game’s Search algorithm used to generate moves.
* **evalPos.js:** Contains simple piece-position arrays for each piece type, which is then used to determine the best move by the search algorithm.

## Algorithm design

The developed chess engine solely relies on an optimised Minimax-based game search algorithm, as with most current chess engines (although they might slightly differ in implementation). The algorithm was first introduced by John von Neuman in 1928, which then modified to target the problem of chess by Claude E. Shannon in 1950 (Newell and Simon, 1976).

This section discusses the process followed to reach the current state of the engine’s evaluation algorithm.

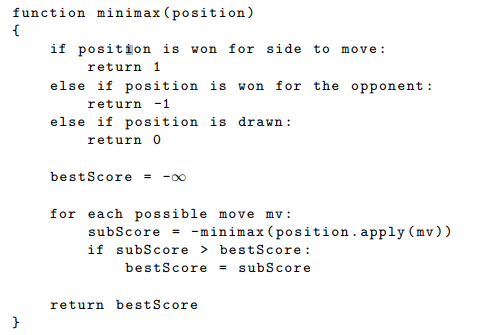
1. **Simple Minimax:** Starting with the simplest form of the algorithm. All it aimed to perform, was to recursively loop through “optimistic” self’s possible moves and “pessimistically” loop through the opponent’s, see Figure 1.

Figure 1: Simple MiniMax form. (Lai, 2015)

While the Minimax algorithm works, it is only practical in simple games such as, tic-tac-toe (where the size of the search tree would be 362880 at most), due to its high inefficiency in more complex games like chess, where possible game search tree can reach approx. 10120.

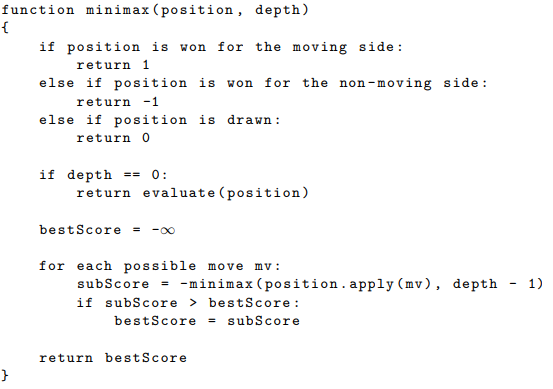
1. **Introducing Depth:** A fixed-depth search was then implemented to allow for a more efficient performance, see Figure 2. This allows the program to specify how deep down the search tree it wants to explore in a relatively timely matter. 

Figure 2: Depth-limited Minimax (Lai, 2015)

However, this introduces a fatal problem, the Horizon Effect. The horizon effect, happens when the deepest ply (the horizon) appears to be advantageous, when it’s really not, due to the search being blinded by the depth cut-off.

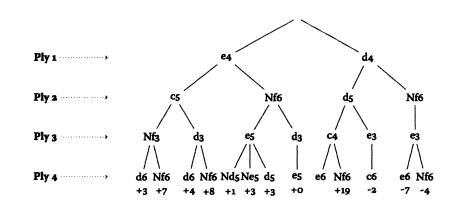


Figure 3: Game search tree (Ensmenger, 2011)

1. **Alpha-Beta Pruning:** Simply, Alpha-Beta pruning can be considered a search filter. It introduces a low-boundary and high-boundary for the current search to allow us to avoid unnecessary searches. It should also be noted that boundaries are inversed depending on the ply being searched.
2. **Search ordering:** While the order of searches in minimax does not matter as all nodes are visited at least once. However, if ordering branches is coupled with alpha-beta pruning it can, in the worst case (perform at O(bn)), and in the best case (perform at O(√bn)). As a consequence, the better the moves are ordered the more efficient the search becomes. The developed engine will uses multiple ordering methods: Most Valuable Victim - Least Valuable Aggressor (MVV-LVA), Killer Move Heuristics, Null-Move Heuristics and search history.
3. **Quiescence Search:** The Quiescent search was implemented to avoid “the horizon effect” and ensure a node is “Quiescent” or quiet before cutting-off the search. What is taken into account when checking for quiet node is a matter of debate; exchange material (capturing a higher valued piece using a lower valued piece) and king’s checks are the agreed to be essential, by all engine authors. Other popular moves include, all possible captures, queen promotions and check evasions. Nonetheless, there is a balance to be maintained when using quiescence search (including too many moves, the search becomes too large; including too few might reintroduce the horizon effect).

## Implementation

First, it is important to understand how the game is conceived by the engine, to allow for a clearer picture on how the search algorithm was implemented.

The board pieces are represented by a simple array of 64 objects (see Figure 1), each representing a board square and contains a value identifying what is on it.

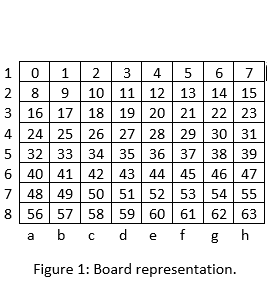
Of course, the array is representation in **Figure 1**, is simplified for better readability, as to the engine, an array is conceived only as a single row with 64 columns.

Figure 4: Board Representation

Given the above representation, the following represents how the best move generated by the engine:

1. Start a timer to a specified time limit in the GUI
2. Initiate a new search using the SearchPosition() function
3. Evaluate current position using Piece tables. PieceTables are manually generated arrays by chess experts indicating the recommended position for each piece type. The PieceTables have the exact same representation as Figure 3, which makes comparing them really simple.
4. Make sure no King threats and time limit is not up
5. Search next ply
6. Repeat step 3
7. Store best 2 non-capture beta values in the Killer table with higher lower score than normal moves.
8. Repeat step 5
9. Store best 3 non-capture alpha move in the History table with higher score than normal moves.
10. Repeat until root leaf is found or time limit is reached

## Testing

A PERFormance Test (PERFT) - A function used to debug the engine’s move generation; by comparing the number of legal moves at a specified depth to predefined values (Chessprogramming.org, 2019)- has been developed to test the move generation of the engine.

Additionally, the engine has many commented console logs which enables the user to view exactly what is happening to make sure everything is running as expected.

Lastly, after implementing multiple ordering methods, the engine runs particularly more efficient, with a 90% average nodes ordering, the number of nodes required to be evaluated is considerably low. The engine was used against [Elometer’s](http://www.elometer.net/) 76 chess problems, achieving an Elo rating of 2000 using a 2 seconds thinking time.

# Critical Evaluation

Developing the engine using JavaScript was not without difficulties. Due to the fact that JavaScript does not produce as the most informative compilation errors, especially, when using a text editor; lead to long wasted hours purely on debugging. Performance potential using multi-threading and GPU assisted processing, other languages provide are superior and much easier to implement than JavaScript.

Due to the occasional lack in performance, searches are cut-off sue to the time limit which might introduce the horizon effect. Even though, this is very unlikely due to the ordering of moves forcing the use of the most promising move.

Lastly, unlike chess engines using a Convolutional or Recurrent Neural Network the evaluation function is solely dependent on fixed piece tables. While they work really well, NN allow the use of weights which can allow the engine to adapt to new chess problems.

# Conclusion and Future Work

In conclusion, the developed chess engine showed great results, considering that it was written in JavaScript and does not have any external dependencies. Which enhanced its simplicity of deployment, accessibility and scalability. Even with the use of piece evaluation tables the intelligence of the engine is particularly high.

The enhancements implemented into the search algorithm were one of the major considerations of this project. This is due to fact that chess engines’ efficiency often correlates to how fast nodes are searched which is directly related to how intelligently it performs.

Finally, there are a couple of areas of interest to me, but due to time constraints I was not able to consider. First, the effect of GPU optimisation and multi-threading on the performance of the engine. Second, the implementation of Neural Network reinforced learning to update and further tune the evaluation function of the engine.

# Bibliography and References

Beal, D. (1990). A generalised quiescence search algorithm. Artificial Intelligence, 43(1), pp.85-98.

Boskovic, B., Greiner, S., Brest, J. and Zumer, V. (2006). A Differential Evolution for the Tuning of a Chess Evaluation Function. 2006 IEEE International Conference on Evolutionary Computation.

Berliner, H. (1978). A chronology of computer chess and its literature. Artificial Intelligence, 10(2), pp.201-214.

Chessprogramming.org. (2019). Perft - Chessprogramming wiki. [online] Available at: <https://www.chessprogramming.org/Perft> [Accessed 14 Jan. 2019].

Chessprogramming.org. (2019). MVV-LVA - Chessprogramming wiki. [online] Available at: <https://www.chessprogramming.org/MVV-LVA> [Accessed 14 Jan. 2019].

Ensmenger, N. (2011). Is chess the drosophila of artificial intelligence? A social history of an algorithm. Social Studies of Science, 42(1), pp.5-30.

Hauptman, A. and Sipper, M. (2007). Evolution of an Efficient Search Algorithm for the Mate-In-N Problem in Chess. Lecture Notes in Computer Science, pp.78-89.

Hauptman, A. and Sipper, M. (2005). GP-EndChess: Using Genetic Programming to Evolve Chess Endgame Players. Lecture Notes in Computer Science, pp.120-131.

Knuth, D. and Moore, R. (1975). An analysis of alpha-beta pruning. Artificial Intelligence, 6(4), pp.293-326.

Lai, M., (2015). Giraffe: Using deep reinforcement learning to play chess. arXiv preprint arXiv:1509.01549.

Newell, A. and Simon, H. (1976). Computer science as empirical inquiry: symbols and search. Communications of the ACM, 19(3), pp.113-126.

Scheucher, A. and Kaindl, H. (1998). Benefits of using multivalued functions for minimaxing. Artificial Intelligence, 99(2), pp.187-208.

Vázquez-Fernández, E., Coello, C. and Troncoso, F. (2013). An evolutionary algorithm with a history mechanism for tuning a chess evaluation function. Applied Soft Computing, 13(7), pp.3234-3247.