A REPORT

ON

AN ARTIFICIAL INTELLIGENCE CHESS ENGINE IN HASKELL

BY

ANISH V MAHESH	2010A7PS021G
NITIN SINGH	2010C6PS550G

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

MAY 2014

ACKNOWLEDGEMENT

We take this opportunity to thank all the people who have helped and supported us during the progress of this project.

We would like to express our gratitude to **Prof. A Baskar**, Computer Science Department, BITS Goa for his continuous support, guidance and motivation which helped us immensely during the course of the project. This project has helped us learn a lot about the real life applications of concepts of Artificial Intelligence and functional programming. Prof. Baskar's guidance and support also helped us in the draft of this report.

INDEX

INTRODUCTION

The goal of our project was to design and implement a Chess Engine in Haskell which can compete with human players and other Chess Engines available online. Different areas of Artificial Intelligence are explored such as Heuristic Search, Tree Searching and Tree Pruning techniques.

A Chess Engine is a computer program that analyses chess positions and makes decisions on the best chess moves. The chess engine decides what moves to make, but typically does not interact directly with the user. This Chess Engine does not have its own Graphical User Interface (GUI) but is rather a console application that communicates with GUI XBoard. XBoard is a graphical user interface for chess. It displays a chessboard on the screen, accepts moves made with the mouse, and loads and saves games in Portable Game Notation (PGN). It serves as a front end for this Chess Engine.

The programming language used for the implementation of the Chess Engine, is Haskell. Haskell is a functional programming language that concentrates on achieving its tasks through recursive calls and a functional programming paradigm.

OBJECTIVE

- The goal was to simulate a real chess game as closely as possible. The input to the core of the program is the 8 by 8 chessboard, and the output is a single move that was deemed best by the algorithms. A visual GUI makes the input/output interaction very easy.
- Rules followed by the Chess Engine are identical to the Official Rules.
- The Engine should play Legal moves. By legal moves, what is meant is that the moves follow all the rules and regulations of a standard chess game but do not have any intelligence involved. So this can act as a move validating chess engine.
- The Chess Engine should have the following features:
    - **Castling** - to castle, move the king 2 spaces to the left or right.
    - **En passant capture** - if an en passant capture is possible, click on your pawn, and then on the empty square where it will end up (NOT on the square with the enemy's pawn you are capturing).
    - **Pawn Promotion** - pawn that reaches its eighth rank is immediately changed into the player's choice of a queen, knight, rook, or bishop of the same color.

- Chess Engine should play intelligent moves. This means engine should be able to think of some moves ahead. Also implementation of knowledge of Artificial Intelligence was expected.

ALGORITHMS

Board Representation

The Chess board has been represented by a two dimensional matrix. Programmatically, it is a list of lists (an array of arrays). The outer list has 8 elements, each one of which is a list of 8 elements. Thus, this forms an 8 X 8 matrix representing each of the 64 squares of a chess board. Further, each square is either empty or contains a piece. This is implemented in Haskell by using the data type Maybe. So each Square is either Nothing or is an instance of the Piece module. Thus every operation on the chess board takes place through this data structure. Any square can be accessed by using the list extraction operation of Haskell. For example, to access the (4, 6) position of a 'board', we do 'board!!4!!6'.
Also, there are functions to convert the position from FEN notation to the indices notation. FEN notation is the commonly accepted Chess notation, where the squares range from a1-a8 to h1-h8, with the alphabet representing the column and the numeric represents the rank of the chess board. So along these notions, functions convert any FEN notation to the indices notation for ease of handling the data structure. For example, 'a1' is (7, 0), 'a78 is (0, 0), 'h1' is (7, 7) and 'h8' is (0, 7).

Thus, as can be seen, everything originates from the board data structure. For instance, moving a piece from b2 to b4 would involve copying the contents of '(board!!6!!1)' to '(board!!4!!1)' and deleting the contents of the original square.

This kind of representation is commonly known as the Square Centric Representation.

Mini-max Searching

The core of the chess playing algorithm is a local min-max search of the game space. The algorithm attempts to MINimize the opponent's score, and MAXimize its own. Further, White tries to attain the maximum positive score while Black tries to attain the maximum negative score. At each depth (or "ply" as it's referred to in computer chess terminology), all possible moves are examined, and the static board evaluation function is used to determine the score at the leaves of the search tree. These scores propagate up the tree and are used to select the optimal move at each depth. The bigger the ply, the better the chosen move will be (as the algorithm is able to look ahead more moves). The branching factor is typically between 25 and 40 moves per ply (the average is around 35).

Alpha-beta Pruning

This common pruning function is used to considerably decrease the min-max search space. It essentially keeps track of the worst and best moves for each player so far, and using those can completely avoid searching branches which are guaranteed to yield worse results. Using this pruning will return the same exact moves as using min-max (i.e. there is no loss of accuracy). Ideally, it can double the depth of the search tree without increasing search time. The average branching factor for min-max in chess is about 35, but with the alpha-beta pruning, the program achieves a branching factor of around 25.

Static Board Evaluation Function

The board evaluation function being used is a simple material value based function. Each piece type has a certain value associated with it. The aggregate of all these values gives the total material value for one side (either White or Black). The final value returned by the function is the difference of material value of White and Black (i.e. Total White value – Total Black value). Therefore, White will want a higher positive value while Black would aim to achieve a higher negative value.
The respective piece values are as follows:
Pawn – 1
Bishop – 3
Knight – 3
Rook – 6
Queen – 9
King – Infinity
(Here, Infinity could be a very large number, in our code it is 1000)

GRAPHICAL USER INTERFACE

This Chess Engine does not have its own Graphical User Interface (GUI) but is rather a console application that communicates with GUI XBoard. This Chess Engine uses an open communication protocol – Universal Chess Interface which is used by chess engines to play games automatically, that is to communicate with other programs including Graphical User Interfaces.

Universal Chess Interface (UCI)

The **Universal Chess Interface** (UCI) is an open communication protocol that enables a chess program's engine to communicate with its user interface. It is an interface between Graphical User Interface (GUI) and the Chess Engine.

It was designed and released by Rudolf Huber and Stefan Meyer-Kahlen, the author of Shredder, in November 2000, and can be seen as a rival to the older XBoard/WinBoard Communication protocol. Like the latter, it is free to use without license fees.

Some basic functionalities of UCI are:

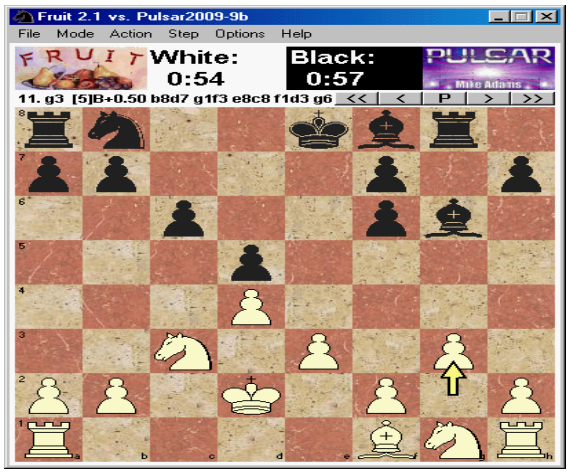
- The specification is independent of the operating system. For Windows, the engine is a normal exe file, either a console or "real" windows application.
- All communication is done via standard input and output with plain text commands
- The engine should boot and wait for input from the GUI, the engine should wait for the "isready" or "setoption" command to set up its internal parameters
- The engine must always be able to process input from stdin, even while thinking

- All command strings the engine receives will end with '\n', also all commands the GUI receives should end with '\n',
- The engine will always be in forced mode which means it should never start calculating or pondering without receiving a "go" command first.
- The engine should never execute a move on its internal chess board without being asked to do so by the GUI, e.g. the engine should not execute the best move after a search.
- Before the engine is asked to search on a position, there will always be a position command to tell the engine about the current position.
- All the opening book handling is done by the GUI, but there is an option for the engine to use its own book

XBoard

XBoard is a graphical chessboard for the X Window System. It is developed and maintained as free software by the GNU project. WinBoard is a port of XBoard to run natively on Microsoft Windows. It displays a chessboard on the screen, accepts moves made with the mouse, and loads and saves games in Portable Game Notation (PGN). It serves as a front end for this Chess Engine. XBoard supports different chess protocols like Chess Engine Communication Protocol and Universal Chess Interface.

An xboard chess engine runs as a separate process from xboard itself, connected to xboard through a pair of anonymous pipes. The engine does not have to do anything special to set up these pipes. Xboard sets up the pipes itself and starts the engine with one pipe as its standard input and the other as its standard output. The engine then reads commands from its standard input and writes responses to its standard output. Xboard sends engine a set of standard commands in string format. Engine parses the commands and act accordingly.
A simple game can be played with a small subset of commands: uci, isready, newgame, go, position, stop, quit.



HASKELL AS A PROGRAMMING LANGUAGE

Haskell is a computer programming language. In particular, it is a polymorphically statically typed, lazy, purely functional language, quite different from most other programming languages.
It is specifically designed to handle a wide range of applications, from numerical through to symbolic. To this end, Haskell has an expressive syntax, and a rich variety of built-in data types, including arbitrary-precision integers and rationals, as well as the more conventional integer, floating-point and Boolean types.

Writing large software systems that work is difficult and expensive. Maintaining those systems is even more difficult and expensive. Functional programming languages, such as Haskell, can make it easier and cheaper.

Functional programs are also relatively easy to maintain, because the code is shorter, clearer, and the rigorous control of side effects eliminates a huge class of unforeseen interactions.

Haskell offers you:

- Substantially increased programmer productivity
- Shorter, clearer, and more maintainable code.
- Fewer errors, higher reliability.
- A smaller "semantic gap" between the programmer and the language.
- Shorter lead times.

PERFORMANCE OF THE CHESS ENGINE

The performance is currently not great. Owing to the simplicity of the evaluation function, the engine merely tries to either protect its pieces or jump at a situation where it can capture an enemy's piece.

But even such a trivial function with a capability of thinking around six to seven moves ahead can be a tough opponent for the average chess player. Alpha Beta search allows us to go up to a depth of four only (taking too much time for deeper searches) and hence does not perform so well. But even saying that, the current chess engine fares decently well against beginner chess players.

Another thing to be pointed out is that at the beginning of a game, the engine behaves weirdly. This is due to the fact that at the beginning all the values it gets even for a depth of four is generally equal and hence the move made is spurious. The move is spurious as according to the current program, it always takes the first move if all the moves produce equal valuation.

This oddity can be reduced by introducing some randomness. For instance, selecting a random move from a set of moves that have equal values could lead to a better performance eventually.

Current Limitations and Scope for Future Work

Many improvements can be made to the current engine. These range from making the code efficient to introducing alternate representations to speed up the search operations.

One of the major bottlenecks in the current version is determining legal moves at each branch, and scoring the board. This must be done for each board position encountered, which of course grows exponentially with the ply search depth. The rules of chess are relatively complex, particularly with special moves such as castles and en passant captures, and the concept of checking a king.

Scoring a board is even more difficult. Currently, a very trivial evaluation function is used which just works towards material superiority. A good evaluation function should not only consider piece values, but also what pieces are protected and threatened in each position, and taking into consideration position superiority. It is possible to speed up all these calculations by creating a different representation of the chess board. Years of research have yielded viable so-called “bit boards,” which keep track of different aspects and properties of the board in data structures that allow for very fast computations. Re-implementing the program with such bit boards would certainly improve it.

Knowledge databases of opening moves and end-game positions could be added. Chess players over the years have discovered that certain move combinations at the beginning of games will develop into good positions mid-game. Since these are well-known, hard-coding appropriate responses would greatly improve the program’s performance in the initial phase of the game – which would then extend into an improved mid-game. Analogously, at the end of the

game, when there are only a few pieces left on the board, it is possible to explicitly reach the goal state via local searches, thus determining optimal moves for many possible positions. Putting these responses into a database would allow the program to perform flawlessly in the end-game; currently this stage of the game is a big weakness of the program.

Many improvements could be made to the already implemented algorithms and concepts. The alpha-beta pruning could be sped up with the null move heuristic, which could determine initial guesses for the alpha and beta values. Quiescence searching could be used to speed up the alpha beta pruning and it can be further improved by intelligently limiting the types of moves considered in the extension searches. There are also other types of pruning techniques, such as futility pruning, but those would be effective in practice only at very high plies.

Another important improvement that could be made is to incorporate learning into the program. Currently, every time a state is calculated, there is no memory involved that is everything is calculated freshly. Including algorithms for learning would make the engine remember crucial games and the sequence of moves that arose in that game. This would help it play a better game next time a similar situation is reached.

So, as can be seen, there is still scope for a lot of work in this project. Due to time constraints, a lot of it could not be implemented, but this project provides a solid enough base to go on and build a robust and high performing chess engine.

CONCLUSION

This Chess Engine is able to play legal moves which follow all the rules and regulations of a standard chess game. Given an 8 by 8 board and a side to move, Chess engine will come up with a move deemed best by the algorithms used.
The Chess Engine can also play complex moves like Castling, En-passant, and Pawn promotion. The Chess Engine also uses some intelligence and can think of n-moves ahead. Heuristic Search, Mini-max Searching and Alpha-beta are the Artificial Intelligence concepts incorporated so far.
The Chess Engine does not have its own Graphical User Interface (GUI) but is rather a console application that communicates with GUI XBoard.
The Chess Engine uses an open communication protocol – Universal Chess Interface which is used by chess engines to play games automatically, that is to communicate with other programs including Graphical User Interfaces.

REFERENCES

Haskell Links

www.haskell.org/hoogle/
https://hackage.haskell.org/

Chess Implementation Links

http://vision.unipv.it/IA1/ProgrammingaComputerforPlayingChess.pdf
https://chessprogramming.wikispaces.com/
http://www.stackoverflow.com/
http://www.wikipedia.com/

GUI Links

http://www.open-aurec.com/wbforum/WinBoard/engine-intf.html
http://wbec-ridderkerk.nl/html/UCIProtocol.html
http://home.hccnet.nl/h.g.muller/interfacing.txt
http://www.gnu.org/software/xboard/