



University
of Glasgow | Department of
Computing Science

Mind Game - Chess

Gary Blackwood
Fergus Leahy
Andrew Meikle
Aleksandr Radevic
Joshua Schaeuble

Level 3 Project — March 19, 2012

Abstract

Today, there are countless mind game programs available, ranging from the simple, tic-tac-toe to the complex such as Go and Chess. We set out not to try and reinvent the wheel for one of these games but instead to learn and document the processes of creating one.

For our project weve chosen to create a Chess program. This dissertation presents our program, JChess, and describes its development life-cycle from inception through to deployment and evaluation. JChess has been designed to be able to play and win a game against a casual player whilst playing by the full rule set of Chess i.e. en passant, castling.

From analysing other Chess programs available such as XChess, researching material from David Levys Compendium of Chess, Chessprogramming.com and advice from our supervisor Dr. David Watt we decided upon what features and characteristics our program should have.

As a result weve successfully created a working Chess program meeting our goals and learnt from it along the way.

Education Use Consent

We hereby give our permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Name: _____ Signature: _____

Contents

1	Introduction	5
1.1	Prospect	6
1.2	Aim	6
1.3	Preliminaries	7
1.4	Structure and Outline	8
2	Background	9
2.1	What is Game Theory	9
2.1.1	Definition of Games	9
2.1.2	Essential Game Features	10
2.1.3	Extensive Games with Perfect Information	11
2.1.4	Game Tree	11
2.2	History	12
2.2.1	History and Impact of Game Theory	12
2.2.2	History of Computer Chess	12
2.3	Deep Blue	14
3	Requirements	15
3.1	Requirements Gathering	15
4	Design	17
4.1	Class Design	17

4.1.1	Model Package	18
4.1.2	Model package	22
4.2	User Interface	24
4.2.1	Overview	24
4.2.2	GUI Builder vs Manual Prototyping	24
4.2.3	Main Window	25
4.2.4	Additional Features	28
4.2.5	New Game Window	29
5	Implementation	31
5.1	Chess Board	31
5.1.1	Board Representation	31
5.1.2	Summary	36
5.1.3	Piece Representation	36
5.2	Artificial Intelligence	37
5.2.1	Move Generation	37
5.2.2	Minimax Algorithm	38
5.2.3	Efficiency	41
5.3	Pruning	41
5.3.1	Alpha Beta Pruning	42
5.4	Static Evaluation	44
5.4.1	Early stages	44
5.4.2	Improvements	44
5.4.3	Difficulties	45
5.4.4	Implementation Specific	45
6	Evaluation	46
6.1	Introduction	46

6.2	Evaluation Techniques	47
6.3	Evaluation Result Analysis	47
6.3.1	Detailed Questionnaire Feedback & Additional Features	51
6.3.2	Improvements	52
6.4	Summary	54
7	Conclusion	56
7.1	Future Improvements	58
7.1.1	Transposition Table	58
7.1.2	Iterative Deepening	58
7.1.3	Move Ordering	59
7.1.4	Quiescence Search	59
7.1.5	Databases	60
7.1.6	GUI Improvements	61
7.2	Appendices	62
7.2.1	Testing Strategy	62
7.2.2	Contributions	63
7.2.3	Acknowledgements	65

Chapter 1

Introduction

“Play the opening like a book, the middle game like a magician, and the endgame like a machine” (Rudolph Spielmann)

The introductory quotation describes not only how to play chess, but also a metaphor for our programming project, which is, as already indicated, the development of a chess program. Therefore we are opening this project by research and planning. We are playing the middle game by metamorphosing these plans into a chess program and finally. We are playing the endgame (not necessarily “like” but) against our own chess machine while we are evaluating the outcome.

1.1 Prospect

“Yet another chess program, but what makes it special?” the reader of this dissertation might ask. And obviously he has a point. Based on our resources and skills there is nothing our programming team could do to develop a chess game that works better than any existing programs. But why did we ever decide to take up this hopeless challenge and what makes this report worth being read nevertheless?

The first question is simple to answer: We all like to play chess, we all like to program and we are all interested in understanding and developing artificial intelligence from scratch. Programming a chess game unites all of these interests with game theory and complex algorithms, which makes it a perfect task for our team.

The challenge is not to program the best chess game but to understand the foundations of zero-sum games and chess programming and to apply theoretical software engineering concepts to solve chess problems in a practical setting. This prospect is also what we can offer the reader by this report. We will document our journey to research, plan, design and finally, implement a chess game in an understandable and hopefully interesting way.

1.2 Aim

Although building the strongest chess game is not our main goal, we aim for a smart and responsive solution that is not only interesting to read about but also to play against. We decided to have three difficulty levels of which the strongest will be able to play against ordinary players (like ourselves) in an entertaining and challenging way. In order to satisfy these requirements our artificial intelligence engine evaluates the most common techniques like central-control and pawn-line-up and will calculate moves within less than one minute on the highest difficulty level. We will also implement our own graphical user interface which will contain functionality to undo moves, save and reload chess games and to highlight previous moves.

1.3 Preliminaries

As we attempt to describe and document our work in the most understandable and comprehensive way there are no preliminaries necessary to understand and enjoy this document.

We will describe the game theoretical and algorithmic foundations as well as the basic software engineering concepts and patterns we adhere to.

1.4 Structure and Outline

Background: Briefly defines what game theory is and how it relates to Computing Science. Section also describes essential features of the game and short introduction to a game tree. Then gives a short overview on history of the computer chess programs from the early years, 1950s to Deep Thought and Deep Blue era.

Requirements: Section outlines project must have, should have and might have requirements. Covers the targets which the team set in the beginning of the design and implementation process and the process of requirement gathering during team meetings and research on computer chess programs. Rules, functional and non-functional requirements which were initially planned to be implemented are included in the end of the dissertation.

Design: Class design section covers domain model of the program and detailed information of the Model and View package. Section also describes the relationships between classes, the importance of particular classes (e.g. Board class) and justification of the design decisions which were made. The second part presents an extensive overview of the User Interface. This reveals the program's main components and additional features which were implemented as should have requirements.

Implementation: Begins with description of board and piece representation. Outlines a short summary of why we chose to use 0x88 method instead of using one and two dimensional arrays. There are advantages and disadvantages of each particular approach, which can be found in this section. Artificial Intelligence subsection thoroughly describes algorithms: Minimax, Negamax and Alpha-beta pruning. Later on the reader can find the number of Minimax positions compared to Alpha-beta positions in relation to the depth of the tree. Lastly the section outlines benefits of using Alpha-beta and is concluded with static evaluation in the early and later stages of the development.

Evaluation: Section goes through evaluation techniques team used to conduct an evaluation. How evaluation results influenced future improvements of the graphical user interface and what could be done better. Users' feedback is graphically analysed and major points are outlined. Feedback form can be found in the end of the dissertation.

Conclusion: In the end of dissertation team concludes what experience it had before working on a project, and how this experience changed in terms of work collaboration, weekly meetings e.t.c. What difficulties we encountered and what could have been done better. Section briefly describes about work of both sub teams AI and GUI. Finally, dissertation conclusion ends with a discussion of a number of techniques and methods that would improve the functionality, efficiency and overall experience of our Chess engine.

Chapter 2

Background

2.1 What is Game Theory

Game Theory is a method aimed at studying decision making in situations of competition and conflict under specific rules. It is used in games of strategy but not in games of chance (such as rolling a dice). A strategic game represents a situation where two or more participants are faced with decisions, by which each may gain or lose, depending on choices made by other players. The final outcome of a game, therefore, is determined mostly by the strategies chosen by participants here lies the challenge because no player knows for sure what the other participants are going to do.

2.1.1 Definition of Games

The object of study in game theory is the game, which is a formal model of an interactive situations and typically involves several players. The formal definition included players, their information, the strategic actions available to them, and how these influence the outcome. There are different kinds of game theory, but this section focuses primarily on two-person game theory.

2.1.2 Essential Game Features

Game theory is concerned with situations which have the following features.

1. There must be at least two players.
2. The game begins by one or more of the players making choices given a number of possible actions, in chess such a choice is called a move. The game of chess begins with a range of twenty alternatives open to the player.
3. After the first move is made the current situation determines who is next to move. For example after white has moved, black has twenty alternatives regardless of how white has moved.
4. The choices by the player are known. These games are called games of perfect information. Games like checkers, go and tic-tac-toe are examples of this type of game. The importance of games of such type is that there is always a “best way to play” which can be specified without mentioning chance.
5. There is always a termination rule. Each choice made by a player determines a certain situation. Following each move, the arrangement of the pieces on the chessboard defines a situation. Game ends then “checkmate” situation occurs.
6. Every play of a game ends in a certain situation. Each situation determines a payoff to each player.

If the above six criteria are satisfied, we can speak of a game.

2.1.3 Extensive Games with Perfect Information

As it was mentioned, an extensive game with perfect information is there every player is at any point aware of the previous choices of all other players. Furthermore, only one player can move at a time, thus no simultaneous moves are possible. Below is a figure that shows a choice game with a game tree containing perfect information.

Every branch point, or node, is associated with a turn which contains all moves that can be made from that position. The connection lines are labelled with the player's choices. The game starts at the initial node, the root of the tree, and ends at a terminal node, which establishes the outcomes, there can be many of these, and determines the player's payoff. In Figure 1, the tree grows from left to right.

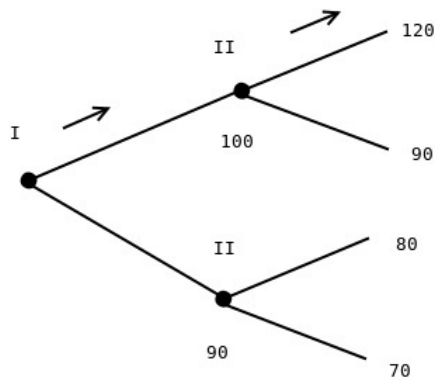


Figure 2.1: Illustration of tree growing from left to right

2.1.4 Game Tree

In order to evaluate a board state and choose the best move to play, all possible legal moves must be generated into a structure which can be traversed and evaluated.

A game tree representation provides the necessary structure to the generated moves, both to search for the best move and also to help visualise the game. In this representation the tree nodes are board states and edges are moves played.

In simple games such as Tic-tac-toe, the game in its entirety can be easily represented in a game tree showing all possible moves from start to finish. This means the game can be solved, allowing an AI to find a sequence of moves which can guarantee a win or draw.

However in a complex game such as Chess, solving the game is impossible within normal time constraints. This is because there is a far greater set of possible moves in a Chess board than in a Tic-tac-toe board. Therefore only a partial game tree is feasible, in which only a set number of plies(turns) are evaluated.

2.2 History

2.2.1 History and Impact of Game Theory

The earliest example of a formal game-theoretic analysis is the study of a duopoly by Antoine Cournot in 1838. The mathematician Emile Borel suggested a formal theory of games in 1921, which was furthered by the mathematician John von Neumann in 1928 in a “Theory of parlor games”. Game theory was established as a field in its own right after the 1944 publication of the monumental volume “Theory of Games and Economic Behavior” by von Neumann and the economist Oskar Morgenstern. In 1950, John Nash demonstrated that finite games have always contained an equilibrium point, at which all players choose actions which are best for them given their opponents choices. As a mathematical tool for the decision-maker the strength of game theory is the methodology it provides for structuring and analysing problems of strategy. The process of formally modelling a situation as a game requires the decision-maker to enumerate explicitly the players and their strategic options, and to consider their preferences and reactions. The discipline involved in constructing such a model already has the potential of providing the decision-maker with a clearer and broader view of the situation

2.2.2 History of Computer Chess

Some of the first non-numerical computer programs were chess programs.

Kirste (1957) wrote a program for the early computers that played on a reduced 6 X 6 board. This program was neither sophisticated nor a very good player. It simply followed up all moves to a fixed depth of the two moves (for each player) and backed up its evaluations using a modified minimax procedure.

Bernstein (1958) progressed to a full-size board and produced a much better chess playing AI by improving the static evaluation function. On the machines available in 1958, following up all moves on a full-size board would have taken a very long time, particularly with a complex evaluation function. Bernstein, therefore, devised a method for choosing seven moves to follow up in more depth, using routines called plausible move generators. These move generators considered such factors as which of the program’s pieces needed defending and which of the opponent’s could be attacked. The use of plausible move generators makes the program’s operation more like that of a human chess player. However, it did not look as far ahead as human players do and it, therefore, appeared to have “blind spots”.

Since 1958, there have been two lines of development in research on chess playing. On one hand people have continued to write programs that play games of chess. The level of play of such programs has improved significantly. However, most of this improvement can be attributed to increase in the size and speed of computers.

By the early 1960, students at almost every major university had access to computers, which inevitably led to more research on computer chess. It was in 1959 that MIT freshmen Alan Kotok, Elwyn R. Berlekamp, Michael Lieberman, Charles Niessen and Robert A. Wagner started working on a chess-playing program for the IBM 70790 mainframe computer. Their program was based on research by artificial intelligence pioneer John McCarthy. The program was written in Fortran and FAP and did not use a move generator. Instead it had a program, called REPLY, which scanned the legal move table, updated, evaluated and reverted each move and ordered them according to a

single ply evaluation. Evaluation functions were written for material balance, centre control, and piece development. The “alpha-beta” heuristic proposed by John McCarthy was introduced during mid 1961. This technique ignored or “pruned” branches of the search tree that would yield less favourable results, thus saving time by reducing the size of the tree.

Work on chess programs was further encouraged by the Association for Computing Machinery’s (ACM) computer chess tournaments, which began in 1970. Soon the became dominated by first Series Three programs CHESS3.0-CHESS3.6, and then Series Four programs. The Series Three programs were in the Bernstein’s mould, but Series Four reverted to the brute-force methods used in early computer programs. The plausible move generator was discarded and all moves were followed up to a fixed depth. The increase in computer power since the 1950s meant that such brute-force methods were much more successful. CHESS4.6 performed at the level of chess master, by simply looking ahead for four to five moves and considering about 400,000 positions. Its limited lookahead meant that it could not develop the kinds of plan that characterise human chess.

The second line of research on chess has focused on simulating the way people play. For two reasons, programs from this tradition have been restricted to parts of the game. They do not play a full game of chess. First, the programs have become highly complex. Second, different skills are required in different parts of the game. This fact is recognised in the chess, in which openings, mid-games and end-games are analysed separately.

In the mid-game, human chess players develop complex plans, which require a comparatively lengthy sequence of moves for their execution. At each point in the sequence the opponent’s reply must also be predicted. The plans that these programs develop are complex control strategies that direct the programs’ attention to a very small part of the search tree. The programs can, therefore, pursue ideas to a much greater depth. Other examples of programs that play only part of the game are the endgame programs. These programs treats each combination of pieces in the endgame as a separate problem and has a knowledge base of stored advice for each combination. For example, when king and knight play against knight and rook, advice used to guide the search for moves includes “do not loose the knight” and “keep the king and knight together”.

In 1977 at Bell Laboratories, Ken Thompson and Joe Condon took the brute force approach one step further by developing a custom chess-playing computer called Belle. By 1980 it included highly specialised circuitry that contained a “move generator” and a “board evaluator”, allowing the computer to examine 160,000 positions per second. This approach was so effective that in 1982 at the North American Computer Chess Championships (NACCC), this chess machine beat the Cray Blitz program running on a million supercomputer.

In the mid-1980s, two competing research groups developed separate chess computers, one named Hitech and the other ChipTest. While both machines took different programming approaches, they shared advances in custom chip technology, allowing them to further implement brute force search strategies in hardware that had previously been performed by software. This allowed faster and thus deeper searching. ChipTest was capable of searching approximately 50,000 moves per second, Hitech 200,000 moves.

Building on the initial ChipTest machine, the team developed a second machine, called Deep Thought. It cost about 5000 dollars and ran at 500,000 positions per second. The Deep Thought chess engine was built around a pair of custom-built processors, each of which included a VLSI chip to generate moves. These were mounted on a single printed circuit board along with an additional 250 integrated circuit chips that controlled the search and evaluated positions. Each processor could evaluate at a rate of a half-million positions per second by adding values retrieved from tables. could weight 120 board features, including centre occupation, mobility, pawn structure, King-protection, and Rook control of files. A special program provided off-line learning by automatically fine-tuning the weights associated with the board features according to the moves played in a training set games. A two-processor engine, controlled by a main program running on a SUN workstation, could exhaustively examine a middle game tree to a depth of ten plies in a three minutes. This machine won the Fredkin International Prize in 1989 for the first system to play the Grandmaster level (above 2400). Both won against Grandmaster in 1988. Concurrently, microcomputers were steadily advancing, leading to David's Kittenger's microprocessor-based program, WChess, which in 1994 achieved worldwide acclaim when it won against American Grandmaster at the Intel-Harvard Cup "Man vs Machine" tournament.

2.3 Deep Blue

After graduation from Carnegie Mellon, Hsu, Thomas Anantharaman, and Murray Campbell from the Deep Thought team were hired by IBM Research to continue their quest to build a chess machine that could defeat the world champion. The functional features for Deep Blue were of special purpose VLSI chess chip; a move generator, a smart move stack, and evaluation function, and a search controller. The move generator offered a major advance over the move generator in the earlier deep Thought chip. In addition to generating captures at high speeds, the new version generated checking moves, check evasion moves, and attacking moves very quickly. the basic algorithm for move generation remain the same as used by the move generator in ken Thompson's Belle. The circuitry dealing with the evaluation function was sophisticated and advanced, allowing the chip to try a fast evaluation and then, if necessary, a slow evaluation.; The fast evaluation was mainly concerned with the value of material on the board. The slow evaluation, carried out on only about 15% of the positions, considered a slew of factors, including square control, pins, x-rays, king safety, pawn structure, passed pawns, ray control, outposts, pawn majority, rook on the 7th, blockade, restraint, colour complex, trapped pieces, development and so on. The smart move stack helped the chip recognise draws through repetition of position. It also helped recognise when a move led to a repetition, showing that the side to move can at least claim a draw. It gave Deep Blue a far greater understanding of a draw. The search controller was in charge of the search of a position delivered from the host computer. It typically carried out a rapid four-five level search, returning information about the position to the host when done. Chess playing program was written in C and ran under the AIX operating system. In the match against World Champion Garry Kasparov Deep Blue was capable of evaluating 200 million positions per second to a minimum depth of 10 to 12 levels in the middlegame and deeper in the endgame.

Chapter 3

Requirements

Our team has chosen our Third year Team Project to design and create a chess program. Before proceeding with the design and actual implementation, it is important to outline requirements for the program's functionality and efficiency. Therefore this section will define targets for our project.

3.1 Requirements Gathering

Requirements gathering. Before making the list of must have, should have and might have chess requirements team made a research in computer chess development. We went through a number of chess programs which were already written by other programmers, discussed our plans and goals with our project advisor, as well got some brief knowledge of history of computer games development. These all information assisted us in setting our final requirements for our project. It was important to assure that stated requirements are clear, complete, consistent and unambiguous for all team member. Although because of the time constraints of this project some of goals we were aiming for were not possible to implement, team came up with the following goals.

- **Must have goals.** Initially it was important for us to start with basic functionality of the program. Which included basic difficulty level, command line interface and essential chess game rules. At early stage of development we have omitted the possibility for both players to castle. Chess was using minimax algorithm with a single material evaluator function. Using command line interface human player could type the move coordinates, game ended if one of the sides king was check mated. At this development stage testing against another AI player was not possible therefore work on other evaluator functions was postponed.
- **Should have goals.** To make interaction with the system more enjoyable graphical user interface features were prerequisite. It was agreed that GUI for chess will be written in Java Swing. As one of our should have features, chess will have timer, undo facility, possibility to castle. To meet our initial requirement to make chess reasonably fast and efficient we needed to optimise move search generator and in a as much as possible quicker way to calculate best move for current position. So it was decided to transfer minimax into alpha-beta algorithm. More description about these two algorithm are going to follow later on. It is important to mention that our aim was to implement AI searching algorithm so it will take less than 30

seconds with depth 4 and at most one minute for depth 5. In order to make AI more intelligent evaluator features such as centre control, pawn structure, king safety were important to have. To put more emphasis on human interaction with the system, team thought that following game board attributes will make game more intuitively interactive and support a novice player during the game.

- Highlight previous move and possible moves for selected piece.
- Display the move that the user has played, as well as the computers counter move.
- Display state of play such as whos move it is, check and checkmate.

For more advanced players team decided to implement possibility to choose between various difficulty levels. As well player could choose whether to play against human player or AI player. By default human could play white pieces but either choices were available.

- Additional Goals These are extra goals that should be implemented if time is available, each improving the experience and performance of the Chess program but yet not essential to its purpose. The chess program could have:
 - A looped animation to show thinking time for computer.
 - Different sounds for illegal moves, check, and checkmates.
 - Auto save feature in case of failure.
 - Pre-warning of possible opponent moves.
 - Player tutorials or summary of rules.
 - A table of achievements for each player.

Chapter 4

Design

4.1 Class Design

For the design of the program we chose to split it up into several packages, one for the User Interface, the Chess engine, the Evaluators and for the Piece representation look ups. This choice was made to separate concerns and to loosely follow the MVC pattern. Doing this enabled us to not only split the work up in a more organised way but to also easily distinguish between what belongs in the main Chess “engine” and graphical user interface packages as well as what belongs in the other auxiliary packages. Following the MVC pattern we created a Model and a View, which represent the Chess engine and GUI respectively.

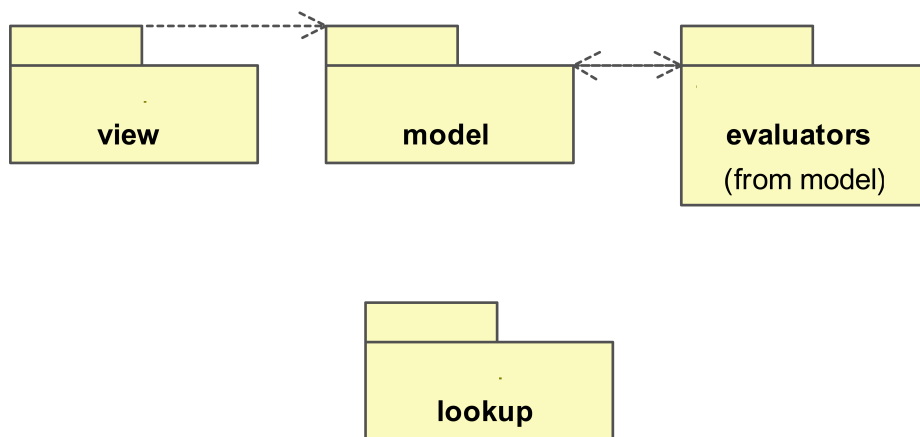


Figure 4.1: Package view of JChess

4.1.1 Model Package

Board and Move

We chose to design board as its own class so that each instantiation maps to a particular board state after a set of moves. This representation works well because each board has its own particular set of moves, scores and pieces taken which we want to keep with the board as its used. Of all classes in the Model, board is by far the largest containing all operations related to making, generating, validating and evaluating moves. We chose to keep these within the board as opposed to separating them out to other classes to try and reduce the overhead of making calls across multiple objects

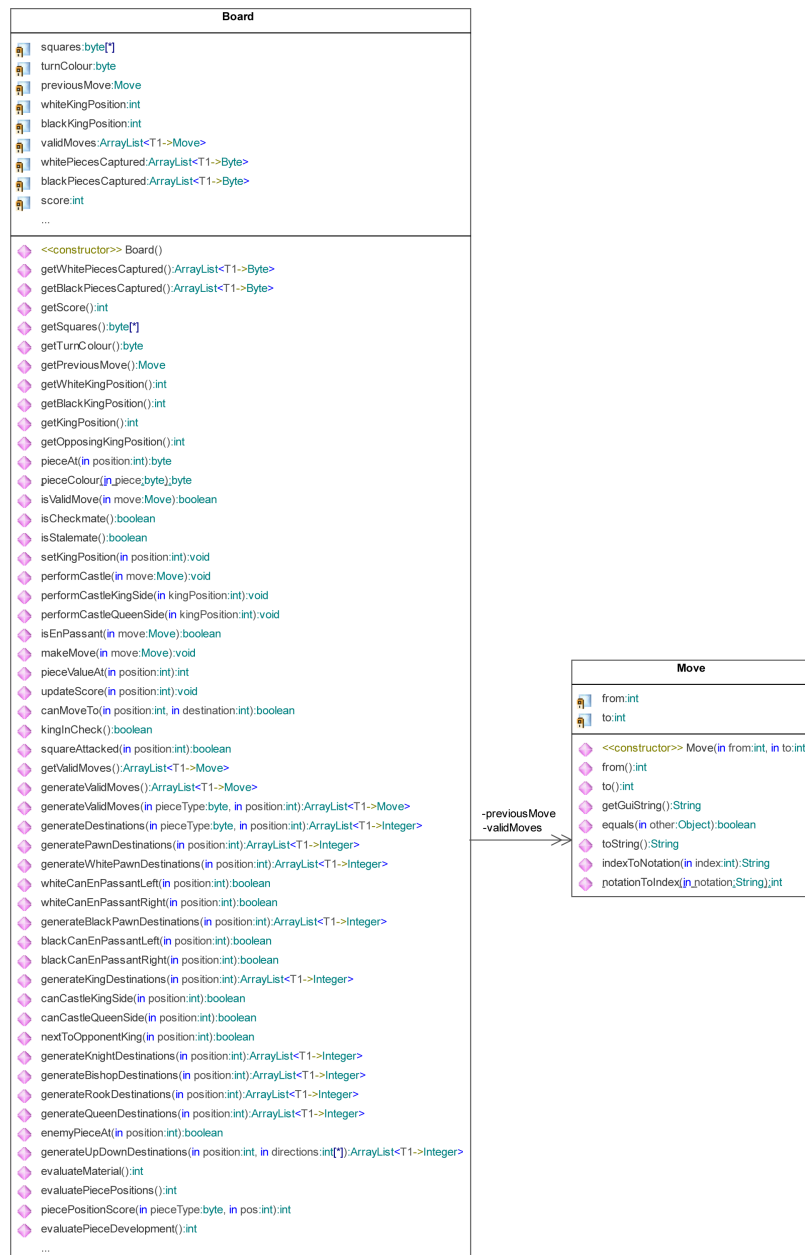


Figure 4.2: Class diagram of Board and Move classes.

When considering designing how a move should be implemented we decided on creating a move class we had two options. The first, a move would only exist as “to” and “from” values which were created by the player and then passed as parameters to the board to make a move. The second was to create a separate object for each move. This object would keep all the information relevant to a move and provide useful methods (i.e. array index to chess notation) It would also allow us to store the moves easily for later use, possibly enabling the undoing of a move or saving a game.

Player

For the players of the game we decided on an design which could enable the switching of players very easily in code. This led to the inception of a player class which could then be sub-classed into an AI or a human player. Because both players have the same set of operations i.e. `getMove`, `getColour` e.t.c. The only differences occur in how the move to be made is acquired. For a human player it is from a selection on a user interface and for the AI its from generating and evaluating moves. Doing it via this method allows the program to have games such as Human vs Human, AI vs Human and AI vs AI without much effort in coding.

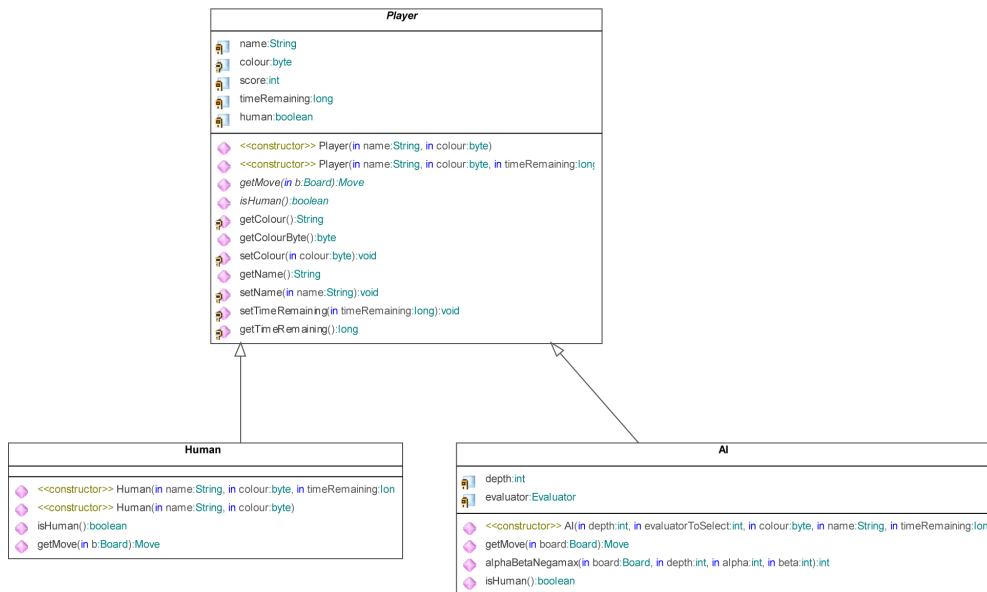


Figure 4.3: Class diagram of Player classes.

Evaluator

For the Evaluator design we wanted to make sure all evaluations we as efficient as possible minimising calls to across objects but at the same time we wanted to try a keep to an object orientated style. We solved this problem by placing the evaluation algorithms within the board and then creating a set of Evaluator classes which accessed those methods and computed a score for a given board. This keeps calls efficient as the Evaluator functions and the board they are to be performed on are within the same object. Our design included two main evaluators, easy and medium. These would use a different set of evaluation functions to compute the score of a board. The hard difficulty would be implemented by using the medium evaluator and increasing the look ahead depth.

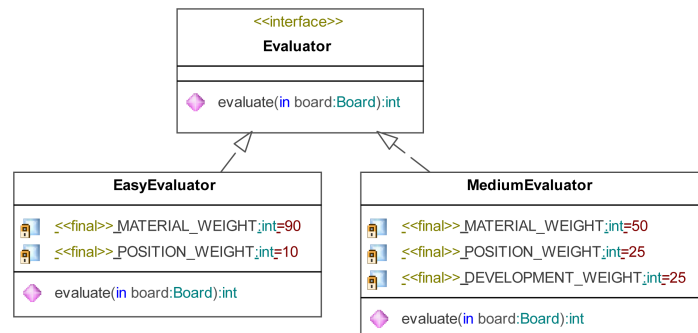


Figure 4.4: Class diagram of Evaluator classes.

4.1.2 Model package

While the model package clearly fulfils the task that is indicated by its name the view package performs two tasks of the model-view-controller pattern we used to design the application: It illustrates the game by providing a graphical user interface (presentation = view task) and it controls the communication between the visual representation (in form of user input/output) and the model package (=controller task). We consciously decided to unify view and controller because of the limited time we had to implement the Graphical User Interface. By combining the presentation task with the controller in one package (and within this package even in one class) we were able to speed up the development (everything is in one place, we did not have to think about separation) at the cost of future maintainability. From our perspective this was the only option, especially because future maintainability is subsequent less important than when compared to the risk of not delivering in time with such a short implementation period.

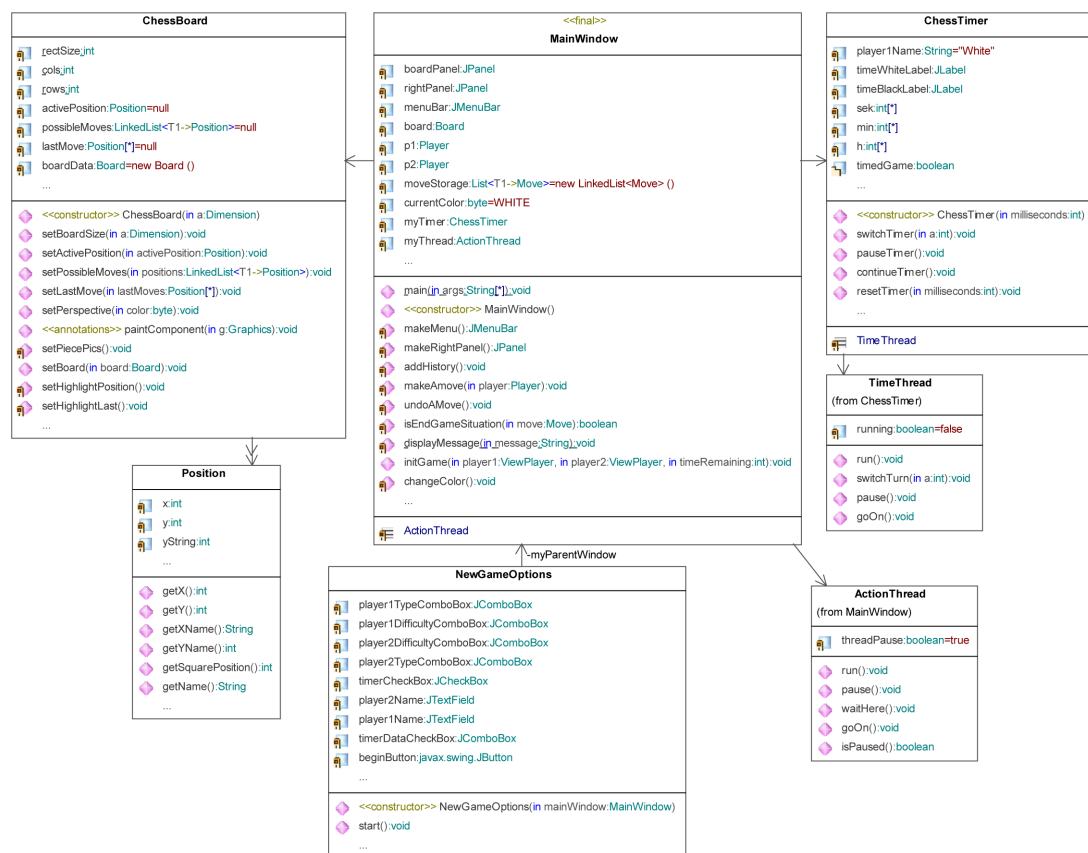


Figure 4.5: The set of classes which the View package comprises of.

Main Window and Chess board

The main class within the view package is MainWindow. This class strictly handles all the user interactions and controls the communication with the model by instantiating a Board and two Player objects. We designed the classes with the approach to balance between outsourcing graphical functionalities from the main window into separate classes (for future maintainability) and keeping

everything in one place to speed up the initial implementation. Therefore the other view-classes represent specific areas of the graphical interface, which are all instantiated, ordered and organised by the MainWindow but which do not contain user input/output.

The most important one is the ChessBoard class (that must not be confused with the Board class in the model). The ChessBoard class is the graphical representation of the board; it provides a squared panel including all of the graphical elements of the board (nested in a graphic) like the fields, the pieces and the background colours for highlighting. Due to our design all the user input/output is handled by the MainWindow, the MainWindow listens to mouse events performed on the printed instance of the ChessBoard, calculates changes and repaints the updated ChessBoard. For these calculation and to create an adapter from the model's board representation (an integer array sized 128) to the view's representation (a two-dimensional area of pixel that is mathematically divided into coordinates) we implemented the class Position. Its instances can be built with one representation of a board position (like "1" or "a1") and provide all the representations (like "0/0" as coordinates). We also created a NewGameOptions class which displays a window to set-up and create a new game.

Timer

Finally the timer provides a chess clock that is inspired by traditional mechanical chess clocks: it connects two clocks for both players in one class, of which only one can run at a time. The MainBoard instantiates the clock with a given time and can then either pause, continue or switch the timer from one user to the other. For the timer to run continuously and to refresh the actual time within the GUI it was absolutely necessary to thread the timer manually along with the game thread that controls the game and that is provided by the MainWindow. As both threads are closely related to specific classes in our design we decided to build subclasses within the class that a Thread controls. This is also matching with our approach to balance the GUI implementation between keeping things in one place as far as possible and keeping the project maintainable for future developments.

4.2 User Interface

4.2.1 Overview

In this section we will discuss the design for our user interface, the different options that were considered, the arguments for the different approaches and why we chose our current configuration.

When we were designing the GUI we wanted it to be functional and easy to use while still providing useful extra features and different game options.

Our initial requirements specification included a history panel that would allow a user to interact with the previous moves made in the game. The history feature was to allow them to undo moves and resume the game from any point, dynamically. Additionally we planned to implement the following graphical features: graphical representations of the player clocks for timed mode games, extra features that would be helpful for beginners such as possible move highlighting and highlighting the previous moves made.

4.2.2 GUI Builder vs Manual Prototyping

The first weighty decision we had to make was whether we preferred to use an IDE that supporting GUI building (e.g. Netbeans) or to program the GUI manually. Despite probably the most common industry practise of using GUI builders to design interfaces we decide to code it manually. Choosing a GUI building was, more than likely the faster, easier and probably more professional option. However, we decided that learning more about the intricacies of GUI development by manually producing the interface was a better course of action and would also lead to more efficient debugging. Nonetheless, this decision made the GUI development a more time and resource heavy section but it did allow us to gain a larger exposure to different interface elements, e.g. threading.

4.2.3 Main Window

Our main page, which include the board representation and history panel mentioned before, never underwent major redesigns, instead there were design decisions that had to be made about the components that were included within.

The first important decision that faced our GUI design team was whether or not to have every square on the board as an individual button(64 buttons with black and white backgrounds) or to have the board display as one large display and the the location of mouse clicks would relate to an appropriate square.

The benefits of the button approach are that the buttons can have events linked directly to them such as when they are clicked - which would provide us with a easy and simple way of establishing what square had been selected. They are also easily re-sizable which would reduce scaling issues when going to full screen mode and do not require any complex mathematical calculations to do so. Nevertheless as they are not dynamic and have a fixed position they have limited graphical opportunities. This is illustrated, for example, when a piece would try to be moved the picture could not be animated to move from button to button as it would require significantly more work, computationally.

On the other hand, a manually painted (and constantly repainted) single graphics object compensates for all the weaknesses of buttons. It is a single object that can be generated with mathematical operations and repainted within milliseconds. The components that are included are dynamic and the numerous “clicked” listeners can be replaced by one listener that can record the position of a mouse click. Then based on the position calculates and returns the clicked position mathematically in terms of the selected square. This is the solution we chose to use although it is more complex to implement it is a worthwhile risk due the additional flexibility that it provides as well as they decreased computation time which is a the most important thing for a chess program.

The main window that the program uses is included below. As discussed previously it contains the main board, history panes. It also includes a menu bar and timers both of which had no great impact on the interface design.

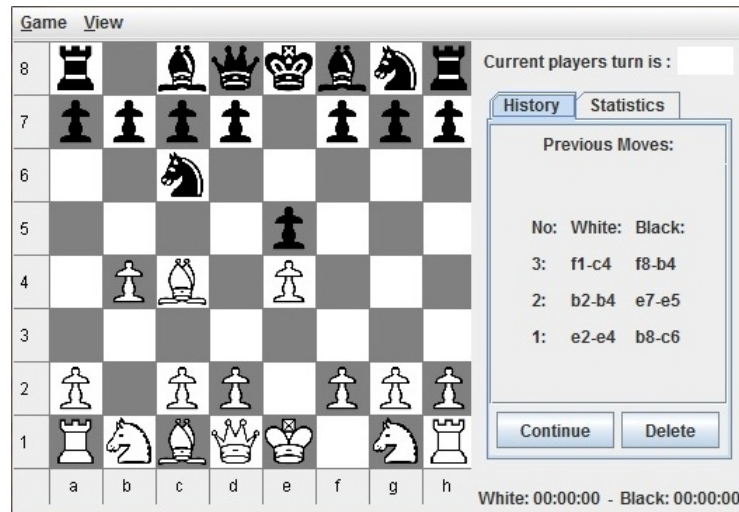


Figure 4.6: Illustration of the main window design with the initial history feature design.

However, although the main window did not undergo any revisions the history feature underwent two revisions, the second which is illustrated below was carried out after user feedback suggested it was not intuitive enough.



Figure 4.7: Illustration of final main window design, with the revised history feature.

We decided to change the feel of the history tab from looking as if it was one large entity to a collection of individual objects that upon selection changed colour, in order to provide the user with some additional feedback whereas before there was no feedback at all. Another option we decided to change was the two buttons. The reason we had them before was because it made it clear what could happen when you selected a move to delete this was proven incorrect as will be illustrated in the evaluation section of this dissertation. Instead of using two separate buttons we went for three buttons but they would change depending on the users interaction with the system, these buttons were also named more sensibly, instead of delete it became undo. We opted for the undo button because it meant that we conformed to the norm as any chess program has a option for undoing one move. This was a problem in the old design because in order to undo a move in the old design you had to select it first then undo it which adds unnecessary complexity to would should have been a simple process. Finally, we opted to change the style of the timers to make it more visually pleasing for users.

Finally, the order that the history was populated was reversed, we initially had it filling up in reverse because it would keep the recent move at the top which, we believed, would be the most useful thing to see as it would become tedious for the user if they had to keep scrolling down in order to undo a move every time they wanted. However, in the revised version we decided to order it normally because it is similar to most existing program that have a history feature to order it in this manner and make slightly more sense because in general interface design principles the top of an object normally signals the start and not the end.

4.2.4 Additional Features

The last design issue we had to consider when designing the board was how we would represent the two GUI orientated requirements which were highlighting the previous move (denoted as the red line in the image below) and also highlighting the possible moves (denoted as green).

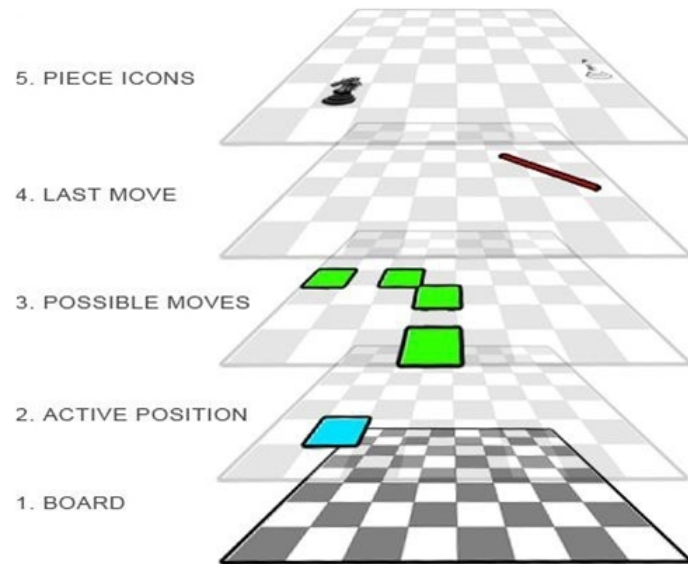


Figure 4.8: Illustration of layering within the GUI

We choose this layered approach because it gave us the ability to enable/disable layers 3 and 4 dynamically through a drop-down menu on the main screen.

Below are some examples of how these additional features were presented in the program.

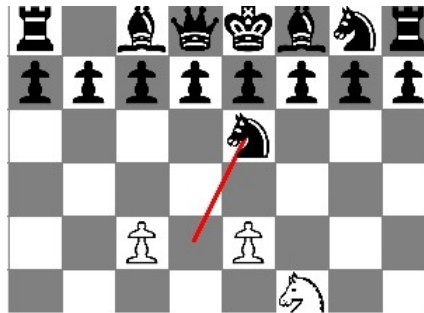


Figure 4.9: Illustration of highlighting the previous move.

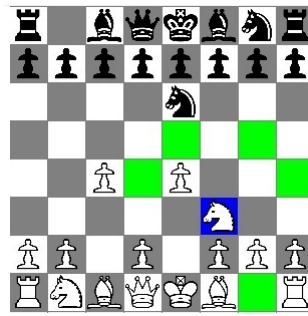


Figure 4.10: Illustration of highlighting possible moves

4.2.5 New Game Window

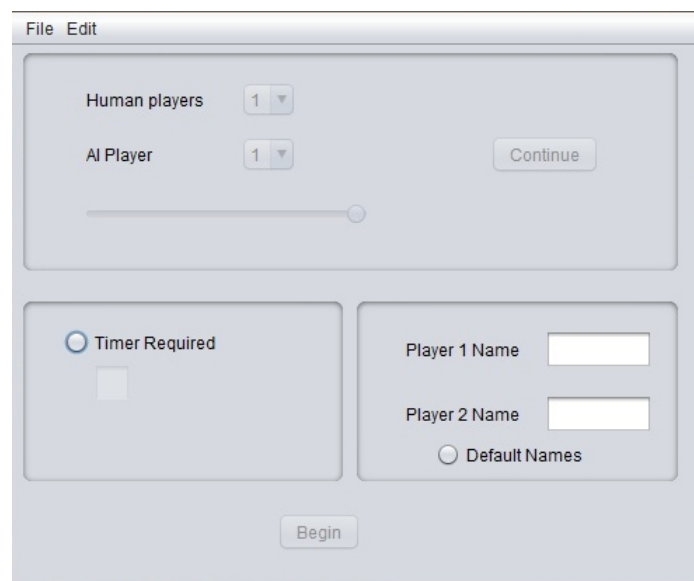


Figure 4.11: Initial new game options.

This was the initial way the new game option screen looked. It had validation throughout every panel so before progression onto the next panel the input in the initial field had to be correct (which is illustrated by the greyed out continue button as this has already been pressed). This ensured that all of the game creation was regulated and no bugs could be entered this way.

However the amount of validation quickly became cumbersome during testing due to the tedious process of creating the game. Although the combination of text boxes and drop-down boxes made the input process fairly easy to compute it meant there was no fast way through to create a game even though some options were already selected by default.

This led to the main disadvantage which was that the number of steps required to create a game was too high.

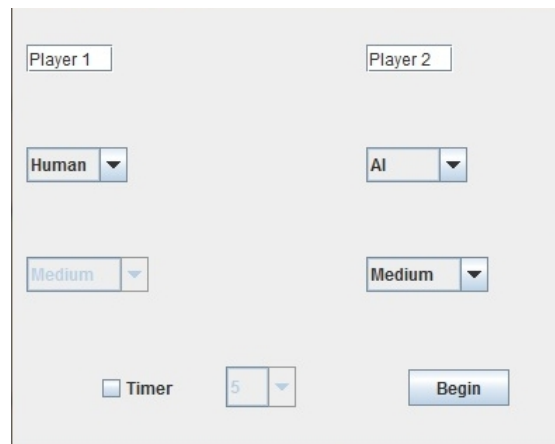


Figure 4.12: Other new game options.

This was the other prototype version of the new game options panel. It involves the user selecting solely from drop-down boxes, thus creating minimal need for validation. Additionally, we decided to make the player1 and player2 label into a text field that could be altered to change the player name. We also changed the timer section so that instead of it being a component that required users to type in the time which increased the chance of user error and complicated input validation we decided to make it a drop-down menu with a healthy time selection which both reduced possibility for user error and made the input validation easier.

Our final decision on the design of our layout was with the second option. Mostly due to the fact that it reduced the number of steps required to create a game and the reduced need for input validation mostly thanks to removing text fields. The text fields were also removed from being a separate section because when people play chess games it is not detrimental to their playing experience being unable to select a name, especially for local(non Internet based) chess games.

However, the initial prototype looked somewhat more logical because as it contained a flow of execution throughout the various steps of game creation -which was the main reason why it was slow- whereas the chosen prototype presents all of the options from the beginning.

Chapter 5

Implementation

5.1 Chess Board

5.1.1 Board Representation

Choosing a board representation is the most important aspect of computer chess program design. The wrong decision may introduce future difficulties and as a result it is frequently re-designed and changed to improve operation efficiency. Representation shortcomings are often discovered while implementing other system components such as move generation and static evaluation. It is therefore critical that careful consideration is taken beforehand in selecting the most efficient and maintainable representation so that any future complexities may be avoided.

Chess board representations exist in two distinct variations; the array based and as a set of bit-vectors. Using an array based representation, the chess board is represented by an array of elements where there exists a mapping between each element of the array and each square of the chess board. A bit-vector approach differs greatly in that a number of 64-bit words are used in combination to represent the board.

The following sections are used to discuss the advantages and disadvantages of various representations that we considered.

Two-dimensional array

0	⇒	0	...						7
1	⇒	0	...						7
2	⇒	0	...						7
3	⇒	0	...						7
4	⇒	0	...						7
5	⇒	0	...						7
6	⇒	0	...						7
7	⇒	0	...						7

Figure 5.1: A two dimensional array.

After considering that a chess board has sixty four squares, the most obvious and logical approach is to create a 8x8 two-dimensional array in which each element would store information associated with the particular square it mapped to. At first this approach seems reasonable given that it is simple to implement and provides a one-to-one mapping from elements to chess board squares.

However, problems arise during move generation as we discover efficiency drawbacks that mark this board representation as unacceptable.

It is also necessary to check each move and ensure that its destination is a valid position on the board. This is achieved by ensuring all rank and file values are ≥ 0 and ≤ 7 . These checks are performed so often that they noticeably hinder performance. A variation exists, the 12x12 array, that solves this particular problem by filling the outer elements that are not mapped to a chess board square with invalid values. The move generator can now determine if a move is invalid or terminate its search of the board if it finds an invalid value. The 10x12 provides the same functionality with an improved memory usage by filling only the left and right outermost edges with invalid values.

Although this representation provides a logical view of a chess board, we determined that it would not be a suitable choice due to efficiency concerns.

One-dimensional array

56	...						63
48	...						55
40	...						47
32	...						39
24	...						31
16	...						23
8	...						15
0	...						7

Figure 5.2: A one dimensional array.

This representation improves upon various aspects including an improved real world view of the board itself. Figure 5.2 maps the square a1 to the element at index 0, h1 at index 7 and h8 at index 63. The array provides the flexibility to store square h8 at index 0 and leaves it up to the

programmer to decide upon his preferred method of implementation. The key improvement is that it is now possible to perform piece movement operations by simply adding a constant, therefore avoiding the costly operations required by the two-dimensional array.

56	...						63
48	...						55
40	...						47
32	...						39
24	...						31
16	...	→	→	→	→	→	23
8	...	↖					15
0	...						7

Figure 5.3: An example of an illegal “wrap” around move.

The one dimensional array, much like its multidimensional counterpart, requires the same conditional tests to check whether or not the destination of a move is valid. In many scenarios it is possible to “wrap” around the board and therefore perform an illegal move. For example, consider the scenario shown in figure 5.3. The move generator is generating moves for the bishop located on square b2 (index 9) and checking the upper left directions by adding the offset 7 to the previous position each time. The array index sequence is 9, 16 and 23 which would now move the bishop from its starting position to h3. This type of illegal move must be caught by the move generator and we therefore do not make any efficiency gains over the multidimensional array.

As arrays are stored in a linear manner at the hardware level, to access the information stored at any particular element it is necessary to compute the associated index beforehand by using the following formula:

$$index = rank * 8 + file$$

The result is then used as an offset from the base address of the array. A smart compiler may optimise the calculation however the number of operations taken to compute the result would still be too high to be considered efficient.

We concluded that the efficiency gains were not sufficient enough to make this representation a viable option. The next step is to eliminate the computation performed while checking if a move has a valid destination.

0x88

112	...							X	X	X	X	X	X	X	127	X
96	...							X	X	X	X	X	X	X	111	X
80	...							X	X	X	X	X	X	X	95	X
64	...							X	X	X	X	X	X	X	79	X
48	...							X	X	X	X	X	X	X	63	X
32	...							X	X	X	X	X	X	X	47	X
16	...							X	X	X	X	X	X	X	31	X
0	...							X	X	X	X	X	X	X	15	X

Figure 5.4: Two chess boards side by side.

The 0x88 method combines two side by side chess boards to create a 16x8 one dimensional array. The squares on the left half of figure 5.4 represent the playable chess board whereas those on the right side are marked as illegal positions which are used solely to determine invalid moves.

The main advantage of the 0x88 representation is that it provides an extremely efficient method of determining whether or not a given square index is on the chess board. Although not obvious at first, the hexadecimal number 0x88, when shown in its binary form 10001000, has either the seventh or third bit set. This is key property which takes advantage of the fact that the chessboard's dimensions are an even power of two. It allows us to perform a simple and fast check to determine whether a square is on the playable left hand side chessboard.

Legal Board Index	Binary	Illegal Board Index	Binary
0	00000000	127	01111111
1	00000001	126	01111110
2	00000010	125	01111101
3	00000011	124	01111100
4	00000100	123	01111011
5	00000101	122	01111010
6	00000110	121	01111001
7	00000111	120	01111000

Table 5.1: A comparison of the left and right hand side of an 0x88 array.

Table 5.1 highlights the bit pattern differences between the valid board square indexes in comparison to the those in the illegal right hand side. All indexes located in the right hand side of the board have either their seventh or third bit set high while those valid indexes on the left side have their seventh and third bits set low. Using this knowledge, we can now determine whether or not a square index is legal by performing the following operation:

$$index \& 0x88$$

If the resulting value of the operation is anything other than zero, we know that the index is invalid as a binary AND operator is used to lower all bits other than the seventh and third. As the valid board indexes have their seventh and third bits set low, the result will always be zero. In contrast, the illegal board indexes will always result in a value greater than zero.

Another advantage of this representation is that if the distance between any two valid board squares is a multiple of the normal distance increment for a particular piece then we know that both squares are on the same diagonal, file or rank. This also solved the “wrap” around issue that previous representations had as each distance increment will either move to a legal square or an illegal square in which case we stop.

In comparison to the previous array representations, the 0x88 method offers various improvements and provides advantages worthy of consideration.

Bitboard

A bitboard is a 64-word that stores positional information associated with a single piece type. Using the property that a chess board has 64 squares, a bitboard is able to store exactly enough information in each bit, which is mapped to an associated chess board square, to determine the location of all pieces of that type on the board.

The bitboard representation consists of a number of bitboards that when combined together determine the current positions of all piece types on the board. For example, the simplest implementation uses twelve bitboards; one for each piece type and colour.

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 10000001
rank 8    rank 7    rank 6    rank 5    rank 4    rank 3    rank 2    rank 1
```

Figure 5.5: A bitboard for white rooks.

Figure 5.5 displays the bit pattern for a bitboard representing the white rooks at the start of a game of chess. This implementation maps the least significant bit to the square a1 and as such is set high to represent a white rook being present on the square. The example also shows a second white rook on the square h1 which is mapped by bit seven. Also shown is the property that every byte of the bitboard stores information related to an associated rank of the chess board.

The most obvious advantage is that significantly less memory is used to store all of the required information; 96 bytes if we use twelve bitboards. However, the major advantage of bitboards is that they allow for efficient computation of the game logic itself. Not only are operations which are expensive in array based representations much more efficient, they are also simple and fast. Array based representations require cycling through the board to locate the appropriate pieces needed for move generation. Bitboards provide a much faster method; using bitwise operations that complete in one CPU cycle. Many CPUs also have additional bit instructions that improve performance such as locating the first bit.

It is clear that the Bitboard is the most efficient representation to use and one which is likely to achieve the deepest move look ahead. The major drawback is that the approach to computing game logic is drastically different from the popular array based representations and considered more difficult to implement. Its popularity has grown in recent years due to the explosion of 64-bit CPU processors on the market and their ability to significantly improve the speed of the Bitboard representation.

5.1.2 Summary

After considering each representation it was clear that only two of the options would be viable implementations. Both the one and two dimensional arrays simply did not provide the efficient operations required to create an intelligent chess AI.

We decided upon the 0x88 representation as, although more efficient, the bitboard representation would be much harder to implement.

5.1.3 Piece Representation

After deciding upon our chosen board representation, the next step was to determine how to represent the chess pieces. The array based 0x88 board representation stores positional information within its array elements therefore we had a number of possible implementations to consider.

Objects

Taking an object orientated approach was the most logical option as we were using the Java programming language for implementation, which is inherently object orientated. However, in practice objects are far too inefficient and there are much simpler alternatives.

Integers

Piece Type	Integer Value
Pawn	1
Knight	2
Bishop	3
Rook	4
Queen	5
King	6

Table 5.2: An example of pieces represented by integers.

Using integers as the elements of our array is simple and uses less memory than an object. As each array element maps to a corresponding chess board square, we can distribute different meanings to the individual bits of the integer. For example, the three lowest order bits could be used to represent whether which type of piece was present on the square. Table 5.2 shows how the piece types are represented by values in the range 1 – 6. It is now possible to mask the associated bits to retrieve the type of piece on a square.

Bytes

As we were not using more than eight bits within the integer, switching to a byte required no further changes and improved the memory usage of our program.

5.2 Artificial Intelligence

In order to create an AI which can compete and possibly beat a human opponent it must use an efficient algorithm for deciding upon the best move to make given a particular board state. This section will discuss the different algorithms and variations which can be used.

5.2.1 Move Generation

The move generator is a fundamental component of any chess engine that is used to generate the legal moves that can be played from a given position. The implementation relies heavily upon the board representation and it is therefore critical that the correct decision is made regarding this to avoid any future difficulties.

As most computer chess programs rely on brute force to analyse all of the possible moves it is important to implement the move generator as efficiently as possible.

There are three main generation strategies that a move generator may use; each with their own advantages and disadvantages.

Selective Generation

The move generator will analyse the board, select a small number of good moves and discard the others. This type of generation is no longer used due to the hardware advancements that have made the advantages obsolete. It provides extremely efficient generation of moves however the quality of AI was poor due to the selection of bad moves.

Incremental Generation

The move generator generates a few moves and look for a move that is either so good or bad that the search can be terminated. This strategy avoids having to generate every move however it is much more difficult to implement.

Complete Generation

The move generator generates all of the moves that can be played from the given position. Implementation is simple and it is the most commonly used strategy due to the brute force nature of computer chess programs.

Due to its simplicity, we chose to implement the complete generation strategy within our move generator. This will increase the maintainability of our program due to the readability of our implementation.

5.2.2 Minimax Algorithm

In order to create an AI which can compete and possibly beat a human opponent it must use an algorithm for deciding upon the best move to make on a given board state.

Because Chess is a two player zero-sum game, this means that when adding the two players scores the result will always be zero. Therefore one players loss directly correlates to the others gain. This zero-sum nature of Chess means that the Minimax algorithm can be used to effectively calculate the best move to play for a given board state.

The Minimax algorithm created by Jon Von Neumann, works by minimising the opponents maximum gain across a game tree. Conversely this also maximises the players minimum gain because Chess is a zero-sum game. For example given a situation (such as in figure 5.6) where: X represents a starting chess board position and B & C are the board after two different moves made by player 1. Both moves will result in losing the piece moved. By using the minimax algorithm the move which minimises player 2s maximum gain is chosen, which in this case is the pawn move¹.

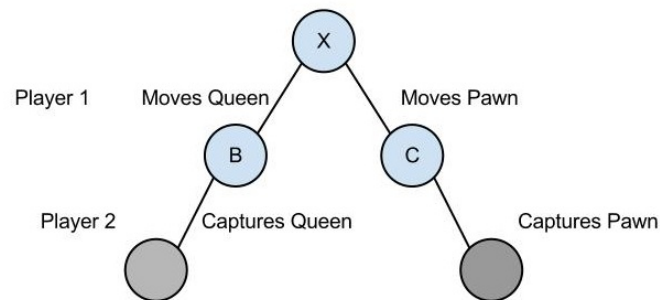


Figure 5.6: Basic minimax representation

¹Based on material balance using standard piece weightings.

To expand upon this further each player is assigned to either be Max or Min. Max always tries to maximise the heuristic score of the board whilst Min tries to minimise this score.

Figure 5.7 below shows a game tree of depth four with a limited spanning factor to illustrate the evaluation process. Moves are generated in a depth first manner and the heuristic evaluation is only performed at the leaf nodes of the tree i.e. the last ply. The evaluation scores are always from the Max players perspective(AI) therefore: +ve == good, -ve == bad. These evaluation scores are then rippled back up the tree to the root node from which the best move is then chosen.

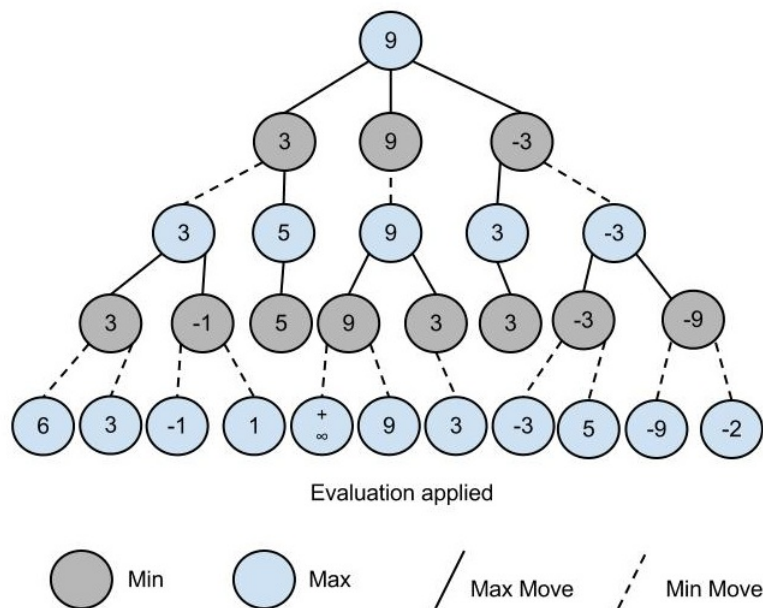


Figure 5.7: Example of the minimax algorithm

As illustrated in figure 5.7 Min makes some bad move which could cause a fork resulting in either check or the cumulative loss of a Queen². Min will choose to minimise the score as much as possible and chooses the score of 9. Max will then try to ripple this score back up to the root as it is the highest score.

When writing the code two methods must be implemented, one for Max and another for Min. This creates unnecessary extra maintenance and documentation than needed.

²Based on material balance using standard piece weightings.

Negamax

The Negamax algorithm is a variation of the Minimax algorithm which reduces the implementation code. This is possible because of the following relation:

$$\max(a, b) = -\min(-a, -b)$$

More precisely because of the zero sum nature of the game one players score is the exact negation of the other players score. Therefore only one function is required to evaluate and find the best move in comparison to the two methods required Minimax.

As shown in figure 5.8 at each node the maximum negated score is chosen no matter whose turn it is. This should always give the exact same result as the standard Minimax algorithm if implemented correctly.

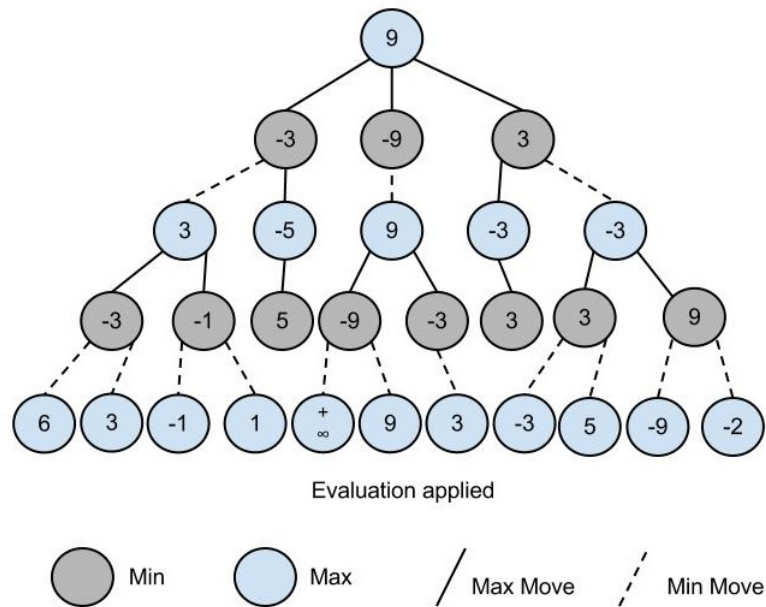


Figure 5.8: Example of Negamax, a minimax derivative

5.2.3 Efficiency

To ensure that playing a game of chess against our artificial intelligence is fun, it is important to minimise the time it takes for a the computer to determine the best move. To do so it must first analyse all legal moves available.

The number of legal moves that can be played from a given position, the branching factor of the game tree, is approximately 35 for the mind game Chess. Thus, to determine the best move in a game tree it is required that 35^d positions be generated and evaluated where d is the number of moves to look ahead by, the depth of the tree.

By examining the total number of positions that must be evaluated in relation to the depth of the tree we highlight the efficiency concerns that must be considered during our implementation decisions.

Depth	Positions
2	1,225
3	42,875
4	1,500,625
5	52,521,875
6	1,838,265,625

Table 5.3: The number of positions in relation to the depth of a game tree.

In Table 5.3 we see that the number of positions grows exponentially as the depth increases. It is therefore essential that any implementation decisions consider possible efficiency consequences.

5.3 Pruning

Using just Minimax or Negamax alone would create either a very slow and time consuming game or an AI whose move look ahead is minimal reducing its difficulty to play against.

This is where pruning such as Alpha-beta is essential. Pruning involves cutting unnecessary parts of the tree to reduce the overall evaluation time. This can typically reduce the tree size dramatically which consequently greatly increase the speed of evaluation and ultimately finding the best move.

By reducing the game tree significantly, pruning also allows much greater depths to be reached within the same time frames as achieved by lower depths in a standard Minimax algorithm.

There are many different pruning algorithms to use such as Alpha-beta, Razer and Mate Distance. These all offer efficiency improvements but Alpha-beta pruning offers the most significant improvement when compared to the rest which is why we've chosen to implement it in our project.

5.3.1 Alpha Beta Pruning

Alpha Beta pruning is perhaps the most well known pruning algorithm for speeding up evaluation.

It works by keeping track of two values throughout the tree, Alpha and Beta. These values keep track of the Max players maximum possible score and the Min players minimum possible score respectively.

The difference between the two is called the window, initially set very large i.e. alpha -infinity and beta +infinity. Once the window closes and the values cross over(beta becomes less than alpha) the evaluation algorithm knows that traversing any further down the branch which caused this crossover would yield no better result than previous evaluations, in essence pruning that branch.

A good example of this is shown in figure 5.9. Evaluating the tree in a depth first manner and rippling up the scores from the leaf nodes at sub-tree (a) where Min chooses the minimum value (maximum negated value) -3. On returning to sub-tree (z) 3(alpha) is rippled up to Max. Then sub-tree (b) is searched, where it evaluates the first leaf node and ripples up the score to 1(beta). Because this is less than the previous best score(alpha) so far and we know that Min will choose at maximum this score the algorithm simply stops evaluating any other children of (b).

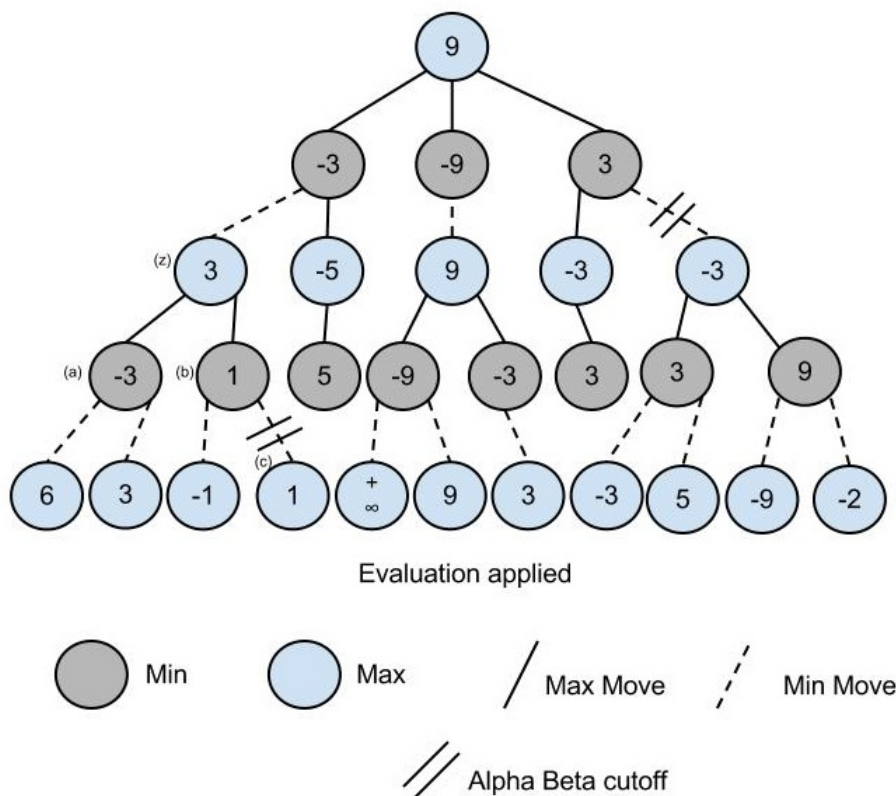


Figure 5.9: Example of Negamax with Alpha Beta pruning, an efficient minimax derivative

Benefits

With Alpha-beta in mind what does this mean in actual savings? Using the formula below we can calculate the best case pruning scenario where optimal move ordering is in place. [6, Alpha Beta Savings]

$$b^{\lceil n/2 \rceil} + b^{\lfloor n/2 \rfloor} - 1$$

In table 5.4 it clearly demonstrates the effectiveness of using Alpha-beta over the standard Minimax algorithm.

Depth	Minimax Positions	Alpha-beta Positions
2	1,225	69
3	42,875	1,259
4	1,500,625	2,449
5	52,521,875	44,099
6	1,838,265,625	85,749

Table 5.4: The number of Minimax positions compared to Alpha-beta positions in relation to the depth of a game tree.

5.4 Static Evaluation

The evaluation function is the most influential component of a computer chess program in dictating the strength of the artificial intelligence. It is possible to search the game tree at a much shallower depth and still play to a standard that may surpass those engines that search deeper. The evaluation function makes this possible.

5.4.1 Early stages

Piece development, piece positioning and king safety were among the very first metrics that we evaluated. The piece development evaluator encouraged the central pawns to advance and rewarded minor pieces to station themselves on key control squares. In addition, it ensured that bishops and knights were placed on optimal squares, away from the edges of the board. The king safety evaluator ensured that a king was adequately protected and surrounded by friendly pieces.

Our initial implementation traversed every square on the board to determine the evaluation scores for various pieces. This method was extremely inefficient; we had to do better.

5.4.2 Improvements

Our new evaluation functions improved the overall performance of our program. The material evaluator was left unchanged, as it was functioning as intended and was very simple. During development, we adopted an iterative approach, testing each new evaluator against the existing ones to ensure that we were making progress.

In the new evaluator each field on the board was given a score. Pawns by high score were encouraged to move to the centre or to reach the first opponents line and promote to Queen. Knights on the other hand, were penalised to stay on the tables first and last rows and files as these fields make them passive in attack and defence. The same constraints were given to Bishops, however penalty points were half less. Important to mention is that, both these figures - Knights and Bishops are most efficient in the centre. This refers to the start and mid game, then both sides are trying to take control of the centre as soon as possible. Therefore, fields which potentially can support or attack pawns in the centre are rewarded by 10 to 20 points for each occupied centre field. The same tactics we have applied to Queen. Although, Queen in most powerful piece, its efficiency increases in the centre. But this may be true not for all positions. Therefore, we have found a trade off to reward Queen being in the centre only for minimal score - 5 points and not penalise it being on the board edges or penalise with minimal score -10 points. It is usually taken for rule not to use Rooks in the start and mid game stages. Because rooks are not efficient to fight for centre control, we have rewarded them to control Kings and Queens fields as this files are pointing to the centre and opposite Kings and Queen fields and to attack the second opponents row, making it easy to destroy pawn structure and have possibility for Kings attack.

5.4.3 Difficulties

Minor modifications can have a big impact on the strength of a chess engine. We took advantage of this property to create multiple difficulty levels for our users. By developing multiple evaluators and using different weighting for each, we were able to effectively alter the experience depending on exactly which evaluations were being computed.

5.4.4 Implementation Specific

As we implemented the negamax variant of the minimax algorithm we were required to provide a symmetric evaluation in relation to player whose turn it is to move. The following expression is used where `turnColour` is 1 for the white player and -1 for black.

$$relativeScore = turnColour * evaluationScore$$

Chapter 6

Evaluation

6.1 Introduction

We conducted our evaluation in order to test chess program functionality and usability. In particular we wanted to answer these questions:

- How efficient AI plays against beginners and upper level users?
- Does the graphical user interface and its corresponding features make user interaction with the chess program enjoyable and easy?
- Which problems evaluation participants have encountered while playing game?
- What can be improved and modified?

In order to get comprehensive evaluation results we had sample size of fourteen participants in the first study and nine in the second study. We were aiming to target three user groups: beginner, intermediate and advanced players. Beginners - are players who have basic experience in chess, know game rules. Intermediate, on the other hand have better understanding of the game strategy, posses basic knowledge of game stages and can provide a good challenge for our chess program. Advanced users are participants who play chess quite often or are members of the local chess club. Evaluation process will mostly concentrate on first two groups, because evaluation team expects to face difficulties in finding players above intermediate level. We expect that evaluation for each participant will take no longer than 45 minutes. 35 minutes for actual game and the rest for filling out the questionnaire. For this time constraints the depth will be 3, 4 depending on the players preferences. This will make whole interaction with the system not tiring and enjoyable. Player can play game more than once, however feedback form and results will be considered only once. In the end, this section evaluation process will be summarised, statistical data will be presented in a bar chart form as well. The reader will find suggested improvements for current chess program. Reader can get acquainted with the questionnaire's in the appendix section.

6.2 Evaluation Techniques

Evaluation team has decided that the best techniques for gathering data will be questionnaires. In particular this includes eleven questions with optional answers and three question requiring written response for first evaluation and five questions for second evaluation. We have chosen questionnaires as it is the most suitable method for data gathering; it is easy to apply to a group of users. Besides, answers are quick to analyse and to group. However, we did not limit ourselves only in questionnaires. Each participant will be supervised by evaluation team member to provide support, make and document significant observations while evaluation takes place.

6.3 Evaluation Result Analysis

History panel

The first evaluation revealed quite a lot of weaknesses most of them concerned with graphical user interface (GUI).

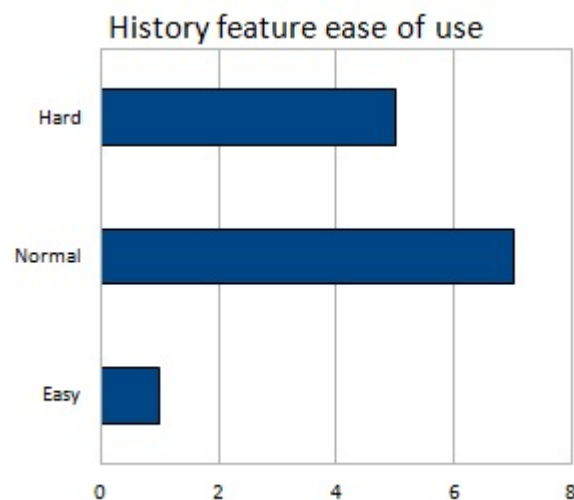


Figure 6.1: Chart illustrating the ease of use of the history panel

More than a half of participants who took part in evaluation noticed flaws in the history tab functionality. Most of them have found this feature unclear and difficult to use. Players did not really find it obvious to interact with and suggested to improve it in future development, if this takes place. In addition, history tab was not fully corresponding to its initial purpose e.g. to undo a move user had to find and select this move in a history tab, press “Delete” button which on occasion was removing more than moves specified. It obviously caused frustration and disappointment while interacting with the system. Some participants suggested to highlight the move line which player wanted to undo as it was hard to keep track of it. As an alternative to delete move user wanted to undo, it was proposed to add “Undo” and “Redo” buttons to undo each move one by one rather than selecting particular move in history tab and deleting it. On the other hand, three users liked undo feature and history tab. Another feature which players found handy to use was highlighting of previous

move and possible moves for selected piece. It assisted a lot in playing game against AI and made experience more enjoyable.

General user friendliness

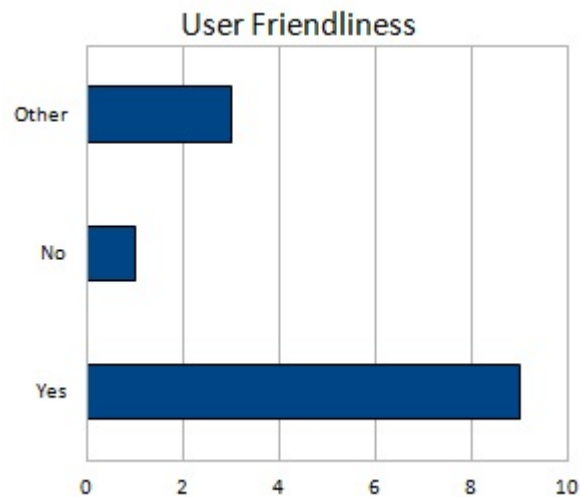


Figure 6.2: Chart illustrating the user friendliness of the GUI

Despite the criticism we received from users mentioned above, it is clear from the other graph that it did not hinder their experience of using the system. More than likely because the main fault was with the history panel which was designed so poorly than most users did not realise it was interactive so it ignored it for their experiment.

It is important to explain the “Other” section of the user friendliness survey, it denotes an additional choice that should have been included into that question as some people rated the user experience well but included as a side note beside the question statements regarding the history panel.

Skill level of the participants

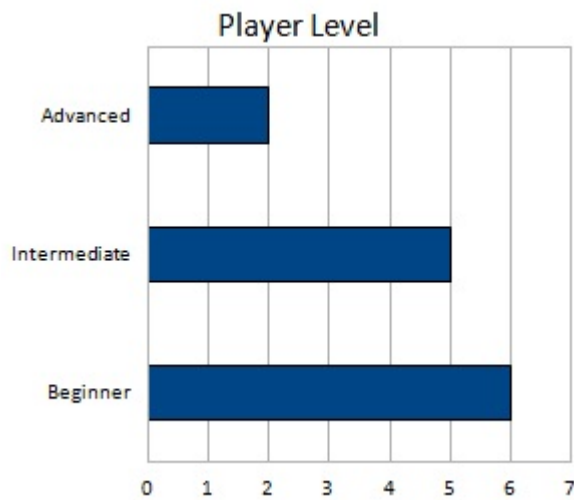


Figure 6.3: Chart illustrating the ease of use of the history panel

This chart shows the player level of the participants in our user test. As the reader can see from the graph the vast majority of our participants were within the intermediate and beginner age bracket. This data suggests that targeting our program at the “casual” chess player was sensible because out of a possible 13 participants only two were advanced players (15% of the total 13 participants). As the “Result” data shows our program won 12 out of the thirteen tests we carried out which proves that our application has met the requirement, “to challenge a casual player” perhaps even a bit too well.

Program was adequate challenge

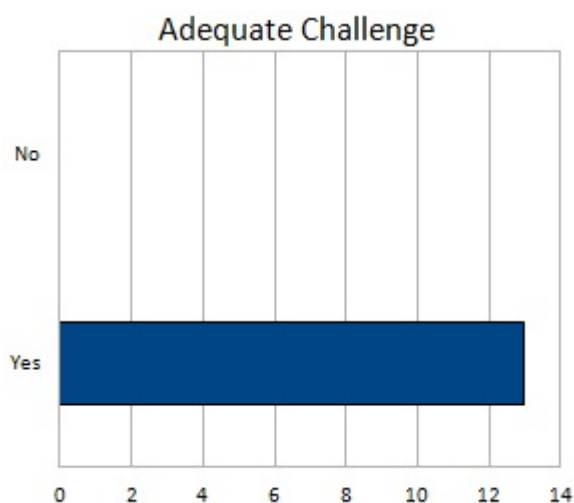


Figure 6.4: Chart illustrating the ease of use of the history panel

This chart shows that every use who took part in the experiment were challenged by our AI, which

was one of our main requirements we had to satisfy.

Response time of AI & Game length

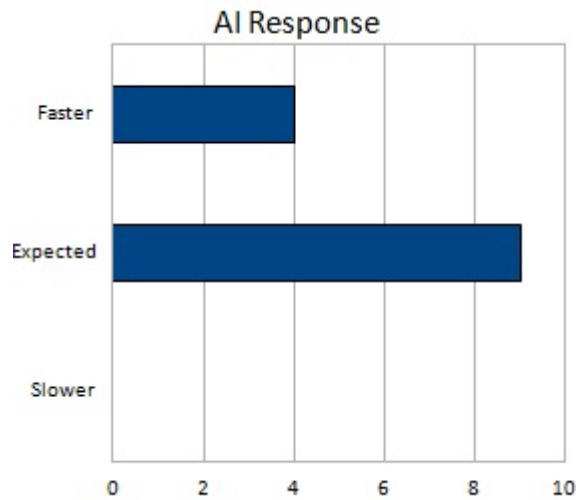


Figure 6.5: Chart illustrating the ease of use of the history panel

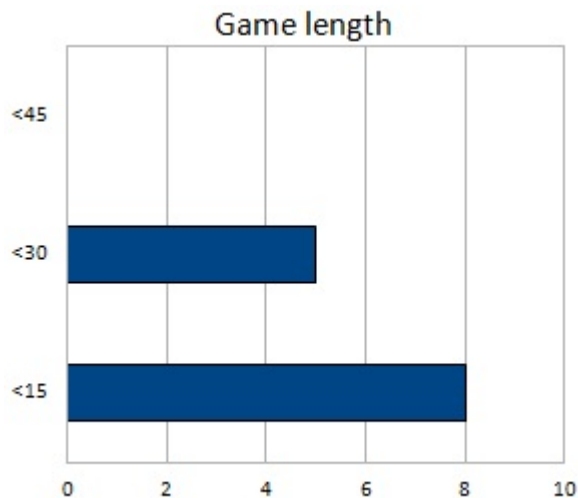


Figure 6.6: Chart illustrating the ease of use of the history panel

Most of the users found the AI response to be as the expected, this may be due to large number of beginner users, whom may not have been exposed to chess programs previously therefore it would be the baseline time for them. Whereas the advanced or intermediate users were more likely to find it faster than other programs they had played. Average game lasted less than 15 minutes and was around 40 moves long. One of the reasons the average game length was under fifteen minutes may have been due to the average skill level of the participants. For example, one of our advanced users was within the less than thirty minute bracket. This suggests that if we had managed to include more advanced users in our sample then the average time may have been greater. Additionally, as a beginner or intermediate user, they may not be inclined to think the moves through completely

and consider all the options which lead to blunders which, consequently, lead to early defeat. Our AI response time may have also been a factor in the short length of each game as it may have intimidated the participants into making their next move quickly, as it responded quickly they have felt they had to respond in kind and thus there were frequent blunders.

Results of individual experiments

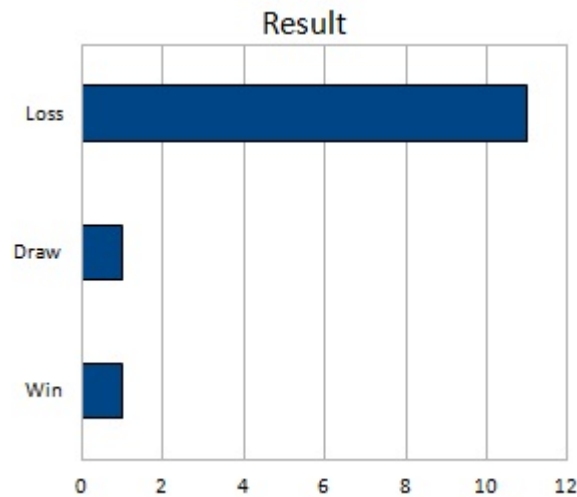


Figure 6.7: Chart illustrating the results of all the experiments

The outcome of our experiments are clear, that despite possible inconsistencies in the AI, it still more than adequately challenged the casual user. One of the victories was produced by one of our advanced users, therefore it can be deduced that perhaps if we had an increased number of advanced users then perhaps the results may have been less favourable for the program. On a side note, it is worth mentioning that the program never crashed during any of the experiments illustrating the defect absence in our chess implementation.

6.3.1 Detailed Questionnaire Feedback & Additional Features

It was quite disappointing to get only a few reviews (detailed) on AI functionality. Three participants found the AI level quite good and only two have noticed weakness in how it behaves during actual game. It was unexpected that on occasions the AI player would not make obvious moves that the human player could detect e.g. losing a Rook in exchange for Bishop or losing a Knight in exchange for Pawn. However this behaviour might depend on various different factors such as current position. On the other hand, this aspect might indicate that the Evaluator functions are not properly calculating a final score for particular positions and this inconsistency may lead to the strange AI behaviour. Another example of why AI should be improved is while playing more than one time, user has noticed that AI each game was moving same pieces (pawns) and in the same order. This and other AI flaws mentioned here should be added as a suggestions for further improvement.

Additional Features

There were some additional features that participants would like to see in future developments of the program, but not all of them can be implemented. Here are just some of them to mention:

- Include sound effects for capturing, moving a piece and making check and mate.
- Highlight when King is in check state
- View pieces you have lost to get an idea of material balance. As an alternative statistics bar can be implemented.
- Have a variety of board themes and pieces styles to choose from
- Include a general timer which shows how much time whole game took place.

6.3.2 Improvements

This section will explain the improve that we could make with the results of our evaluation.

Interface Improvements

Thanks to the feedback we received from our user questionnaires were able to redesign the main focus of the negative of the feedback, the history panel (see the main window section in design for details of the redesign). After producing the new UI version we carried out a second set of experiments, with the same participants, to check if they found the user experience any better. The results of these experiments will be explained below.

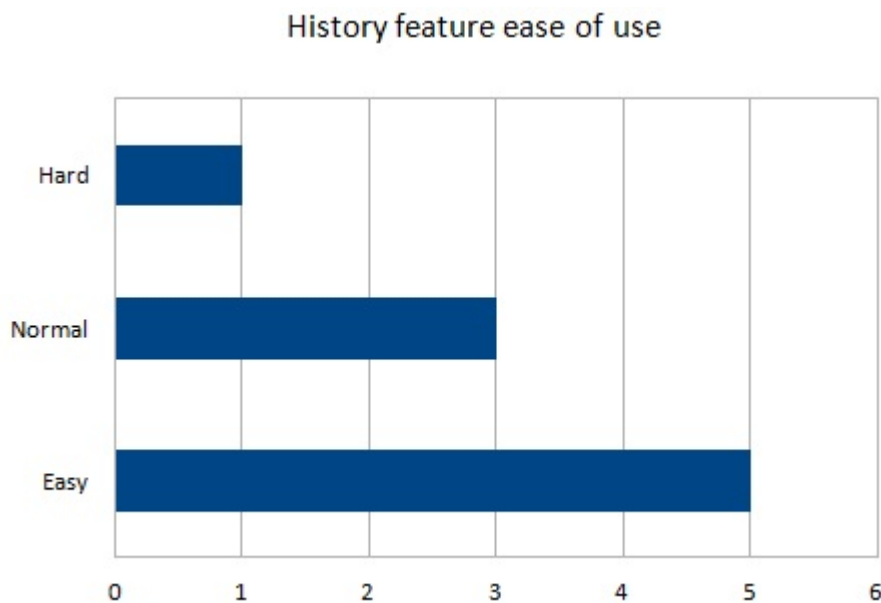


Figure 6.8: Chart illustrating the results of all the experiments

The charts above illustrate the progression in usability going from the first version of the user interface to the second version. We are using a reduced sample size because some of the participants from the previous experiments were unable to take part in this second test. The reason we only used the participants that had already taken part was to ensure the test results accurately measured the results of redesigning the UI vs previous design.

We included player level in the second questionnaire to ensure that perception of the UI was not dependant on exposure to other chess programs. As only a subset of the users that originally took part initially were available for additional tests (we had 6 beginners and 5 intermediates in the initial test group versus six and three in the second test group) we cannot draw any concrete conclusions from the data but the data does suggest that the redesign of the GUI was successful and the number of people that found it easy to navigate was higher than before (prior to the redesign one user found it easy to use, seven said it was average and 5 people said it was hard to use, compare that with the post redesign questionnaire five people found it easy to use, three people found it normal and one person found it hard).

Questionnaire Improvements

As a result of our user testing it was clear we could have designed the questionnaires better as we missed out some key questions and research on the good questions we had included. The first improvement would be to have some way to quantify what makes a player a beginner, intermediate and advanced players such as the Elo rating system, which is described below.

“The Elo rating system is a method for calculating the relative skill levels of players in two-player games such as chess.”(courtesy of Wikipedia)

This would have been an very good way to gauge chess player standards although we were aware that most of our participants were not advanced chess players (e.g someone that could not rate

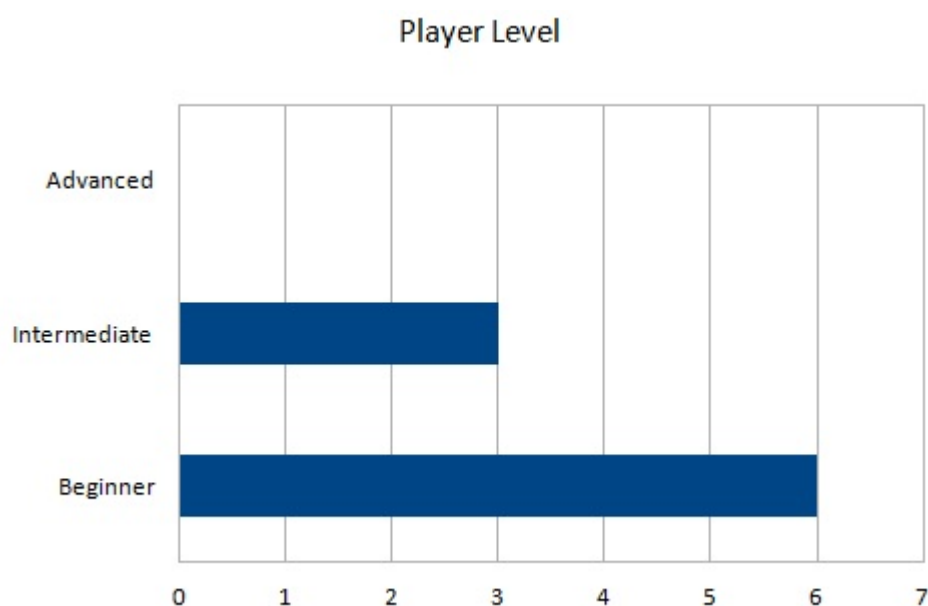


Figure 6.9: Chart illustrating how the usability of the history panel altered with the second revision

themselves according to the Elo rating method). This may have been a good question to include before we inquired as to the skill level.

Additionally, we should also have asked whether or not they had ever used a computer chess program before. If we had asked this question we could have more accurately gauged the improvement going from the version 1 history panel to version 2, because we would be able to tell whether they thought it was better compared to the old one (and only the old one). Compared to the users of other chess programs who would provide us more accurate feedback on the usability of the interface and history feature compared to other interfaces they have used.

If we were to carry out an additional survey, we would include a section for the test leader to use. That would take an average time for all of the user moves and AI moves which would allow us to make a judgement on the result table such as if the number of loses were due to players being inexperience and rushing through the game. Also possibly ask future participants to take part in two experiments one of easy and one medium difficulty. We would also structure the tests so that one group of participants were informed about how to use the history panel features prior to test commencement and one group was not. Therefore we could draw better conclusions of the usability and if a redesign was required or just a help section for the panel.

6.4 Summary

In conclusion, the initial evaluation we performed has revealed that users had difficulty with the history feature and also that the undo feature was complicated and hard to use. While these factors did not inhibit the user experience with the interface we decided to redesign the problematic features and check if the user found the experience more pleasurable.

Taking into account that chess program was evaluation was aimed on mostly chess beginners and

intermediate level players AI has archived showed good results. Thus from our evaluation, we can conclude that current chess program version may act and assist as a good tool for beginners and intermediate level players in practising and developing their game skills as well as perhaps challenging the more advanced users.

Chapter 7

Conclusion

In this chapter we will discuss the various different things we learned through doing this project

Team Structure

In this section we will discuss the knowledge we gained through working as a team

As this was our first exposure to team work on this scale, for most of the team members, it was a steep learning curve and errors were made along the way. Some of the mistakes which we learnt from were poor organisational skills. The effects of these could be seen in the quality of our deliverables mostly due to the fact we left starting the documents late or were of poor quality. Additionally we also learnt the benefit of defining roles as well as creating roles for all of the tasks we might need. For example during the first few weeks we lacked a method of quality control for all the deliverables and after this period we decided a specific role was required to handle the quality control aspect of each deliverable. We also learnt about a variety of different tools that can help you when working in teams such as SVN(revision control system) and facilities offered by Google such as Google docs. Finally, the last important lesson learnt was to not underestimated was the importance of good communication. On occasion certain group members lagged behind with documents the rest of the team had written because they had not been briefed on the communication and tools the team were using.

AI

Through our development of the AI we learnt why complex algorithms are useful and how to implement and tweak these algorithms to perform at optimum levels. These issues will be discussed in the section below.

When going through the motions of designing and implementing an AI that can be multifaceted it is paramount to consider only a limited number of possible evaluation factors as too many can weaken and slow down the AI instead of improving it. The use efficient computation such as Alpha-beta pruning was also vitally important when developing a program which requires a fast response time.

GUI

We also learnt a great deal about GUI development particularly about the tedious nature of developing user interfaces using Java swing without the use of a GUI builders such as Netbeans. During the development of the GUI we also became more aware of the intricacies that come with using threads in Java and, more importantly, the issues that can arise through using them. Furthermore, we learnt about the building and painting of complex graphical elements, specifically the how the process of repainting those graphical elements can impact the speed of the program.

Finally, some of the most important lessons we learnt where in respect to efficiency. We learnt a great deal about optimising code through trying to achieve $O(1)$ operations for algorithm and scoring methods (for example our evaluation methods that use static board representations in order to increase the evaluation speed).

General Project

In this section I will discuss the aspects of the project that could have been done better, in terms of team structure.

During the project we had regular team meetings, these were where the core of our improvements could be made. We had an adequate amount of meetings per week, so the problem did not lie in the frequency of the meetings, but that we lacked a fixed schedule going into every meeting. What often started as meetings with an rough agenda turned into a problem solving session wasting significant amounts of time of team members who werent working on that particular problem. By having a defined schedule going into every meeting which would have helped provide a greater focus and prevent unnecessary issues clouding the main points of the meeting. Finally, we could have designed a more efficient role structure. We initially choose the Laissez-faire structure but upon reflection that may not have been the best choice. As group members were reluctant to fill the leadership position, we decided this would be the best option for us. Upon reflection however, a more hierarchical system would have been better for the same reason because no one wanted to take charge the quality of the work suffered initially.

Summary

In conclusion,we feel that doing this project has taught us a significant amount about the various different aspects of the software development cycle. Additionally it was also a great way to put to use all of the techniques we had learnt throughout the last few years at university such as algorithms, data structures and particularly for GUI development.

7.1 Future Improvements

7.1.1 Transposition Table

A Transposition Table is a database used to store the results of previously performed searches. They are used to improve the performance of computer chess programs by avoiding the need to re-search positions that have previously been reached via alternate move sequences called transpositions. It is very likely that a chess program will analyse the same position multiple times. This allows a transposition table to drastically reduce the search space and increase game speed.

A common method of implementation is to use a hash table to store each of the positions that have been analysed so far. The table is checked every time a new position is encountered to see if the position has been analysed previously. The transposition table allows constant time access to pre-analysed positions therefore if the position is found in the table its value is used directly. If not, the value for the new position is computed and inserted into the table.

As the number of positions analysed by the program is greater than the memory available, old positions must be removed from the table to allow new insertions. The scheme by which the program determines which entries to remove is known as the replacement strategy.

7.1.2 Iterative Deepening

In its current state our chess AI does not increase the speed at which it makes a move if it is running out of time. A human player may take advantage of this by playing very defensively until his or her computer opponent forfeits the game as a result. Most computer chess programs solve this problem by implementing a technique known as iterative deepening.

Iterative deepening provides a time management strategy that performs repeated depth-limited searches until the time which has been allocated for the search has been exhausted. Throughout the search the computer keeps track of the best move so far therefore allowing early termination of the search if the computer needs to make a move quickly.

The following is one possible implementation which highlights the simplicity of the technique.

```
for  $depth = 1 \rightarrow max$  do  
     $best \leftarrow \alpha\beta\text{Negamax}(depth, board)$   
    if  $timeRemaining \leq 0$  then  
        break  
    end if  
end for
```

The advantage of such a depth-limited search is that it provides a good foundation to build upon. If the computer returned the best move as soon as its time ran out instead of completing the search at the current depth it is likely that a blunder move will be made. It also allows you to use the results of shallower depths to increase the speed at which deeper searches are completed which often reduces the time for the overall search in comparison to starting at the deepest depth.

7.1.3 Move Ordering

In order for the alpha-beta algorithm to perform optimally, it is important to order the result of move generation in such a way that the best moves occur first.

Captures and Check

It is often the case that moves which result in capturing an opponents piece or moves which leave the opponents king in check will provide pruning cut-offs. One common move ordering technique is to simply rank these moves above other, less serious, moves.

Killer Move Heuristic

It is common in chess to encounter situations where your opponent only has one good move that you must stop him or her from playing. This move, known as a *killer move*, has a high chance of being played unless explicitly prevented. Killer moves are likely to repeatedly cause beta cutoffs; a property which can be used to increase the effectiveness of our program.

By keeping track of the moves which repeatedly cause beta cutoffs at each depth we can order the moves in such a way that the killer moves are at the top of the list. This allows the computer to prune much more efficiently as it will not have to waste time searching other weaker moves first.

7.1.4 Quiescence Search

Every computer chess program that is limited by the depth at which it can search is prone to a problem called the horizon effect. The horizon effect is an artificial intelligence problem where there exists a possibility that the chosen best move is actually detrimental to the computer players position. However, this detrimental effect is never analysed by the computer as it is limited to a search depth which is less than that required for visibility. For example, the white play may capture a black rook and the evaluation function therefore indicates that the white player is up a piece in material. However, during the very next move the black player can respond with bishop captures queen resulting in an overall poorer position for white.

Quiescence search is a technique used to mitigate the horizon effect by extending the search where appropriate. By applying the static evaluation function to only those positions which are tactically quiet, quiescence search minimises the horizon effect as key passages of play such as piece exchanges are evaluated fully.

Common situations in which to extend the search include piece exchanges, a king being placed in check and pawn promotion.

Although the overall search time increases while using this technique, the accuracy of the algorithm improves so much that the strength of the artificial intelligence will noticeably increase.

7.1.5 Databases

Opening Book

A substantial amount of literature exists on the topic of chess openings therefore it is possible to create a database or some form of look-up functionality to increase the quality of computer chess programs. Opening books save time computing the best move for a given situation if it already exists within the database and are commonly used to increase the variety of play. Searching for the best move provides deterministic results whereas several alternatives may be available using an opening database.

Once the game has reached a state in which the positions are no longer database entries, the computer must continue to use the normal search function to determine the next best move. It is therefore logical to store the more positional information for those openings that are played most frequently.

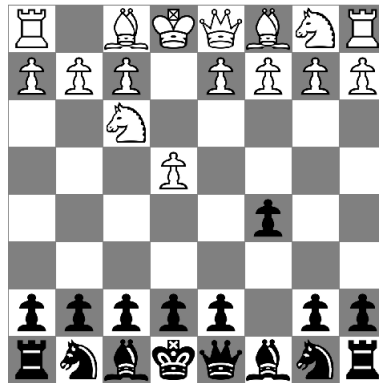


Figure 7.1: The Sicilian defence.

Figure 7.1 shows one of the most popular chess openings against the white opening move e4. The Sicilian defence has many variations which would be excellent entries in an opening database.

End game

Much like an opening book, an end game database provides efficient access to the best move for a given end game position. As there are a huge number of possible end game positions the size of database is limited.

Although the most profound advantage is the increased speed computing the best move for the computer, an end game database can also be used by human players to analyse their games.

7.1.6 GUI Improvements

Load/Save Games

The most important future improvement from a usability perspective is the functionality to store, load and replay games. This would also be an important improvement that could be used to exchange games noted and stored by other chess programs in standard notation. We planned this functionality and it was mentioned in every meeting we had about the GUI design. Until a very late point in the implementation period we were so sure that we would still implement it, that we even added a menu entry for storing and loading games into the GUIs game menu. Unfortunately we had to remove this entry in the end because of the very limited time we had to implement the GUI.

Real-time interaction with the model

Amongst all the other ideas we had while we were brainstorming and researching (mainly by looking at other chess applications) the one improvement that we really liked was an asynchronous communication between the model and the view so that not only the GUI could request changes from the model but also the model could fire events to the GUI. This would have given us the chance to implement functionalities like a move suggestion for the best future human move or an information box that warns beginners and casual players with messages like “Your queen is in danger” whenever the evaluators detect a high probability of such dangers. This would have also been useful to speed up the decision making of the AI in timed games, as the evaluators could constantly submit the best move evaluated so far to the GUI which could decide based on the timer state when to do the last received move.

Standard Algebraic Notation

Another small feature we would have liked to implement for more advanced users was the possibility of entering moves in via Standard Algebraic notation (SAN). This would allow advanced users the same speed enhancements as advanced users of operating systems who use the command line over a GUI. In relation to this the history panel could also be upgraded to support the display of the SAN of moves for those same advanced players. Although we had managed to implement a working SAN parser we simply didn’t have time to incorporate it into the UI.

7.2 Appendices

7.2.1 Testing Strategy

Chess board

During the initial development of the chess board we implemented a simple command line interface to test the basic functionality of the board and make sure the basic rules of chess were working i.e. a bishop only moved diagonally and not through pieces. This allowed us to guarantee that certain functionality worked before creating the AI.

AI

Throughout the development of the AI component we had to make sure we were in fact improving the AIs “smartness” and speed. For comparing speed improvements such as the depth increases, the switch of board representations styles and the switch to Alpha-beta pruning, we used a simple side by side comparison running on two identical computers(lab machines) of the two versions of the program and compared response times. Whilst we didnt collect any precise time records it did allow us to get a good understanding of whether the changes were in fact improvements at all.

Another important part to test was the “smartness” of the AI. As we designed and re-factored the AI we had to make sure what we doing was having a positive effect. To do this we would play the two against each other with middle-man playing the moves on either machine. A refinement of this would have been to implement a way of allowing two different AIs to play a game to alleviate the need of a middle-man.

7.2.2 Contributions

Dissertation

- Gary Blackwood
 - 5.1.*** Chess Board
 - 5.2.1** Move Generation
 - 5.2.3** Efficiency
 - 5.4.*** Static Evaluation
 - 7.2.*** Future Improvements
- Fergus Leahy
 - Abstract
 - 2.1.4** Game Tree
 - 4.1.*** Class Design
 - 5.2.2** Minimax/Negamax Algorithms
 - 5.3.*** Pruning
 - 7.2.1** Testing Strategies
- Andrew Meikle
 - 4.2.*** User Interface
 - 6** Evaluation
 - 7** Conclusion
- Aleksandr Radevic
 - 1.4** Structure and outline
 - 2.*** Background
 - 3.*** Requirements
 - 5.4.*** Static Evaluation
 - 6.*** Evaluation
- Joshua Schaeuble
 - 1.*** Introduction
 - 4.1** Class Design (Images)
 - 4.1.2** View Package
 - 4.2.*** User Interface (Images and alternative draft)

Implementation

- Gary Blackwood

Board.java Game logic and move generation.

Move.java Representation of a chess move.

Evaluator.java Evaluator interface.

EasyEvaluator.java Evaluator for easy difficulties.

MediumEvaluator.java Evaluator for intermediate difficulties.

example

- Fergus Leahy

AI.java Negamax and Alpha-beta algorithms.

Evaluators.java Old evaluation interface.

EasyEvaluators.java Old evaluator for easy difficulties.

MediumEvaluators.java Old evaluator for medium difficulties.

SANParser.java Standard Algebraic Notation Parser.

- Andrew Meikle

NewGameOptions.java Creation of new game.

ViewPlayer.java Player object used in the view package.

SANParser.java Standard Algebraic Notation Parser.

- Aleksandr Radevic

Evaluator.java Evaluator

EasyEvaluator.java Evaluator for easy difficulties.

MediumEvaluator.java Evaluator for intermediate difficulties.

- Joshua Schaeuble

Example GUI

Roles

Gary Blackwood AI team member and dealt with game logic.

Fergus Leahy AI team member and quality control.

Andrew Meikle UI team member and Organiser.

Aleksandr Radevic AI team member and Risk manager.

Joshua Schaeuble UI team member.

7.2.3 Acknowledgements

Our thanks to our supervisor Prof. David Watt for his comments and suggestions and to all those who participated in evaluating our project.

Requirements Specification

Team J

October 25, 2011

1 Rules

1. Queen and King side castling.
2. En passant.
3. If a pawn advances to its eighth rank it will be promoted to either a queen, rook, bishop or knight.
4. A player may not make any move which places or leaves his or her King in check.
5. The game ends when a players King is placed in check and there are no legal moves that can be used to escape.
6. The game ends as a draw if any of these conditions occur:
 - (a) Stalemate - The player is not in check and has no legal moves remaining.
 - (b) King v King
 - (c) King and Bishop v King
 - (d) King and Knight v King
 - (e) King and Bishop v King and Bishop where both Bishops are on diagonals of the same colour.
7. If a player runs out of time during a timed game, he or she will lose the game.
8. Only legal moves will be implemented.

2 Functional Requirements

2.1 Essential

Fundamental Gameplay A playable game of chess that supports the very basic rules.

Command-line Interface A very basic form of user interaction.

Rules The fundamental rules of chess must be implemented that allow for legal gameplay.

2.2 High Priority

Undo Utility Allow a user to undo his or her previous move.

Player v Player Allow two human players to play each other locally.

Save and Load Users should be able to save and load games of chess.

2D Graphical User Interface Allow users to interact via the mouse.

Time Constraints/Clock If a time game is being played a clock shall be displayed for each player.

Multiple Levels of AI Provide AI scaling to accommodate various players.

Additional Rules Implement more complex rules such as En passant.

2.3 Medium Priority

Highlight Possible Opponent Moves Highlight the move(s) that an AI opponent is considering.

Leaderboards Allow users to track their scores.

2.4 Low Priority

Networking Allow two human players to play each other over a network.

Portability Develop the game to run on various platforms.

Time Modes Implement various time constraints for games.

Sound Provide audio feedback in various situations.

Achievements Encourage and congratulate the user for completing a task or doing well.

Game Replays Allow users to archive a game they have played for viewing later.

Algorithm Animation Play an animated algorithm while the AI opponent is considering a move.

3 Non-Functional Requirements

Response Times -

Easy Less than one second.

Medium Less than five seconds.

Hard Less than 15 seconds.

Auto Save Implement an auto save feature as a system failure precaution.

User Tutorials Provide a link or implement various tutorials and learning materials.

Evaluation Questionnaire

Chess program 2012

1. How would you rate yourself as a chess player?

Beginner intermediate advanced

2. Given your previous rating, did the program provide an adequate challenge?

Yes No

3. Which of these best describe the response time of the AI player (keep in mind harder difficulties will take longer)

Slower than expected As expected Faster than expected

4. While using our program, did the AI opponent make any glaring errors?

Yes No

If yes, could you briefly explain what has happened?

5. With your experience of the user interface, was it user friendly?

Yes No

6. How easy was it to use the history feature?

Easy Normal Hard

7. Your result playing against AI?

Win Lost Draw

8. How long did your game last?

less than 15 min less than 30 min over 45 minutes

9. Which features did you really like about this program?

10. Which features you did not like?

11. Which features would you add or what can be improved?

Bibliography

- [1] Chess Programming. Chess programming website.

Bibliography

- [1] David Levy, *Computer Chess Compendium*. Ishi Press International, 2009.
- [2] Rapoport Anatol, *Two-person game theory : the essential ideas*. Univ. of Michigan Press, 1966.
- [3] Routledge & Kegan Paul *Artificial intelligence : an introduction*. Alan Garnham, 1988.
- [4] George W. Atkinsonl, *Chess and Machine Intuition*. Chess and Machine Intuition, 1998.
- [5] Monty Newborn : foreword by Charles E. Lieserson, *Deep Blue : an artificial intelligence milestone*. Springer, 2003.
- [6] Chess Programming website, a repository of information about programming computers to play chess, chessprogramming.wikispaces.com.
- [7] Article on Deep Blue project, [http://en.wikipedia.org/wiki/Deep_Blue_\(chess_computer\)](http://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)).
- [8] Article on Chip Test project, <http://en.wikipedia.org/wiki/ChipTest>.
- [9] Article on HiTech project, <http://en.wikipedia.org/wiki/HiTech>.