Name: Trung Kien Nguyen

Student ID: 104053642

SWINBURNE UNIVERSITY OF TECHNOLOGY

COS10009

Introduction to Programming

REPORT

Custom Project – Chess AI of WAXING CRESCENT

TABLE OF CONTENTS

INTRODUCTION

Simple Chess AI – Minimax Algorithm – Alpha-Beta pruning using Ruby (with Gosu library)

TRUNG KIEN NGUYEN – 104053642

Bachelor of Computer Science – AI major

Swinburne Unirversity of Technology – Hawthorn Campus – Semester 2 – Year 2022

Nowadays, along with the development of chess, including its games and available data of matches, chess artificial intelligence has more and more conditions to develop strongly. Since the last time a human, Ruslan Ponomariov, beat a top-level chess engine, Fritz, in a serious game in 2005, software engineers of major technology firms have created and constantly upgraded a series of chess engines, with computing capabilities surpassing any grandmasters, typically IBM's Deep Blue, KomodoChess's Komodo, or more recently, the chess supercomputers Stockfish and AlphaZero.

The goal of this project *is to create a simple AI algorithm (minimax) to find a suitable move for the computer in a chess game. In addition, it will also discuss ways to optimize the algorithm (that is, reduce the number of computations and execution time). Therefore, this report will not focus much on how to make and design a chess program using Gosu, instead will give a detailed explanation of the algorithm (Minimax with alpha-beta pruning) used for the Black to turn it into a simple AI in this program (Note that my example program is only for elementary illustration of minimax algorithm with alpha-beta pruning, it is not a 100% accurate and standard program).*

OVERVIEW OF THE EXAMPLE PROGRAM – WAXING CRESCENT CHESS

My example program used for this project is developed in Ruby programming language, version 2.5.1p57 [x64-mingw32], with the library of gosu version 1.4.3. It consists of 17 ruby files, including one main file which is *Waxing_Crescent.rb*, 16 other subfiles for the program's function and features, including initial Gosu setting up (*Chess.rb*), drawing chessboard and all pieces of the two sides (*Board_and_pieces_drawing.rb*), checking control areas of each side, player's choosing pieces (*All_valid_moves_checking.rb*), moving pieces (both *Pieces_choosing_of_white.rb* and *Pieces_choosing_of_black.rb*), checking possible moves of each piece (in the directory of *Each_piece_checking*), checking all valid moves (*All_valid_moves_checking.rb*), checking if the game should finish (*Game_finishing_checking.rb*), and the most important file for this project – *Minimax_algorithm.rb*, as well as a *Media* folder that contains all the media files (.png) used for the program.

I have produced the Custom Project Explanation Video (download in [a]) that basically demonstrates the uses of the algorithms to make a simple AI opponent, supporting for this report.

MINIMAX ALGORITHM - DEFINITIONS AND EXPLANATIONS

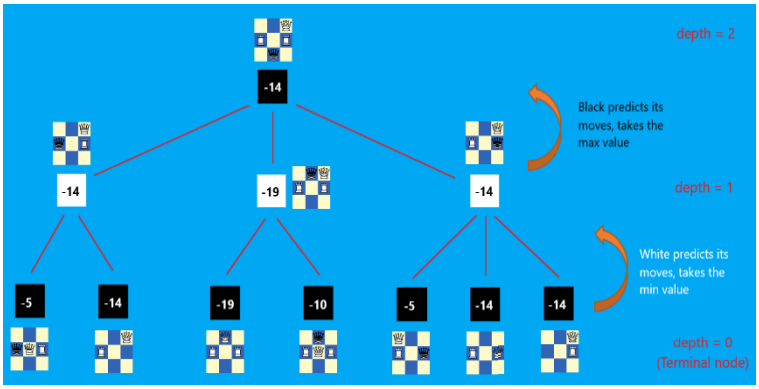
The Minimax algorithm is a recursive algorithm, or rather a backtracking algorithm, aiming to find the best solution for the next move, usually in a two-player game, in which this project is chess. Corresponding to a chess stage (a set of pieces and their positions) will be a value. In the example program, I assign the values of the pieces similar to the common way that grandmasters and computers evaluate, as follows:

- *Black King: +100*
- *Black Queen: +9*
- *Black Rook: +5*
- *Black Knight/Black Bishop: +3*
- *Black Pawn: +1*
- *If white has no way to move*
    - *If white king is checked (WHITE lose) +100*
    - *Else (Stalemate DRAW) value = 0*

- *White King: -100*
- *White Queen: -9*
- *White Rook: -5*
- *White Knight/ White Bishop: -3*
- *White Pawn: -1*
- *If black has no way to move:*
    - *If black king is checked (BLACK lose) -100*
    - *Else (Stalemate DRAW) value = 0*

Because this is only a simple algorithm, I will default that the maximum value will be the best for Black, while the minimum value will be the best for White, so basically every chessboard status, assigned a specified value, the Black will select the maximized value, while the White will select the minimized one. The algorithm is to find the best possible move for the two players, which can be done by choosing the node with the best evaluation value. The best move will be made by Black after evaluating all of the potential moves of both sides. The method has an argument of "*depth*", which defines how many moves (for both sides) will be calculated. Once this method, which is named "*minimax*" in *Minimax_algorithm.rb* in my program, is executed, it will be called again within itself, with the argument *depth* = (currently) *depth* – 1. When the value of this argument is 0, which is the endpoint (terminal nodes) of the potential moves that the algorithm wants to calculate, it will return a value corresponding to the state of the chessboard at that time. By default, when the value of "*depth*" is an even number, the algorithm is trying each move of Black, then returns the largest of the chessboard values. On the other hand, when it is an odd number, the algorithm tries every possible move of White, then the return value must be the smallest of

the potential values. Finally, when the algorithm has executed to the "root", meaning "*depth*" is now equal to the number of moves to be calculated initially (for simplicity, I set it as DEPTH = 2 in the example program), the program returns the indices of the best move found (these are defined in an array of a method that finds all possible moves of Black ("*check_all_valid_moves_of_black*" in *All_valid_moves_checking.rb*). Assuming a depth of 2 (as in my example program), the algorithm's diagram is represented as follows:



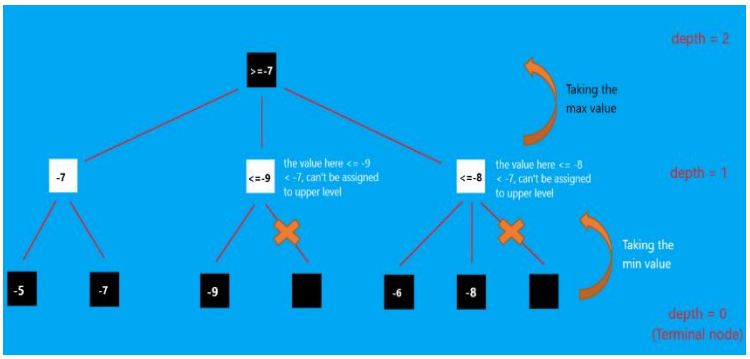
With a depth greater than 2, similarly, Black and White will switch roles continuously in trying out each of their moves. Black will take the largest value of the found values, while white will prioritize the smallest value (See more in the attached folder Diagrams).

The efficiency of the minimax algorithm is mainly based on the search depth we can achieve. However, the larger the value of the depth, the longer the search time will be. Because the current algorithm will look for all valid moves by each side, the total number of calculations will therefore increase exponentially as the algorithm gets deeper.

ALPHA-BETA PRUNING – ALGORITHM'S OPTIMIZATION

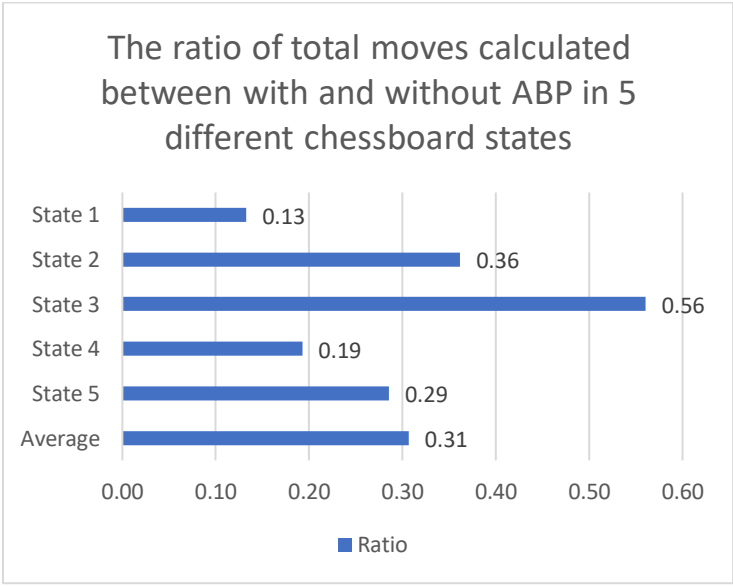
Alpha-beta pruning (ABP) is one of the minimax algorithm's optimization methods that allow the program to skip some branches in the search tree. This shortens the execution time and makes the minimax search tree evaluation much deeper.

The main idea of ABP is trying to stop evaluating a part of the search tree if we find a move leading to a worse situation than a previously detected move. Note that alpha-beta pruning does not affect the final result of the minimax algorithm, but only makes it faster. The illustration below describes an example of Alpha-beta pruning in the Minimax algorithm:



These prunings are only made in the min-taking level, in an algorithm at a higher depth, there will be similar pruning with max-taking level (See more in the attached folder Diagrams).

A small experiment was done to compare all calculated moves needed so that Black can make its move using the minimax algorithm, between with and without alpha-beta pruning, in 5 typical chessboard states. The result, not surprisingly, is demonstrated in the following graphs:



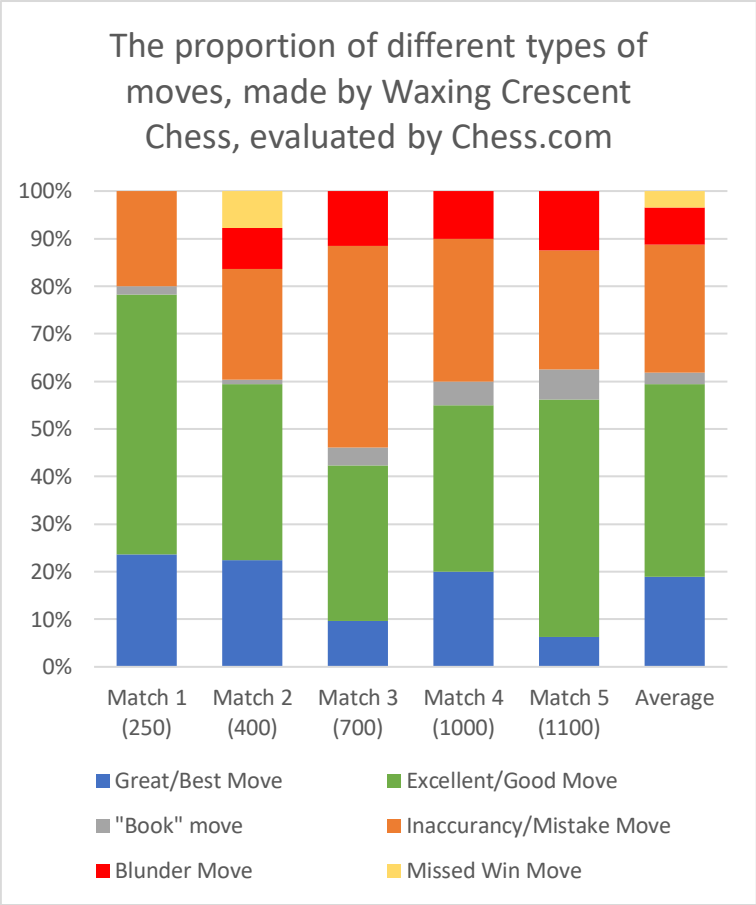
	STATE 1	STATE 2	STATE 3	STATE 4	STATE 5
MOVES WITHOUT ABP	460	1606	1026	389	543
MOVES WITH ABP	61	581	575	75	155

These example chessboard states are available in the attached directory of Experiments' Results/Alpha-beta pruning's Benefits.

The code for this part has been removed from the finished version of the program in an attempt to simplify it by removing some unnecessary features.

TESTING THE ALGORITHM EFFICIENCY

To test the efficiency of my algorithm, I did an experiment, through the chess engine of chess.com. I let my chess program play 5 games with 5 different levels of the chess.com's computer respectively (Elo from 250 to 1100). Of course, my program could only beat Martin – Elo 250, and lose all the other matches. However, more significantly, I recorded the chess.com computer's evaluation of each program's move, shown in the following chart:



Types	G/B	E/G	Book	I/M	Bl	MW
Match 1	13	30	1	11	0	0
Match 2	26	43	1	27	10	9
Match 3	5	17	2	22	6	0
Match 4	4	7	1	6	2	0
Match 5	1	8	1	4	2	0

The detail information about each match is saved in the text (.txt) files, along with the screenshots in the attached directory of Experiments' Results/Chess.com's Evaluating

LIMITATIONS OF THE ALGORITHM AND MY PROGRAM

A such simple algorithm as **Minimax**, together with a simple value-evaluation system, cannot avoid limitations. For example, a knight in the corner of the board cannot be as strong as a knight in the center, and a pawn in the endgame always outperforms itself in the early stages, while I have set their values to constants. unchanged number. Besides, the algorithm is also very poor in understanding king safety, which means it only knows how to calculate the value of the chessboard, and when the King is in danger (is checked), it will start sacrificing pieces to protect the King.

In addition, I also tried a few ways to make the moves given by Black look more reasonable, including arranging the moves to my preference (Other Pieces -> Rook -> Queen -> King) (done via arrange_valid_moves_by_priority in *All_valid_moves_checking.rb*), prioritizing the castling moves and avoiding repeats when possible (in *Pieces_moving_of_black.rb*), and eliminating complicated-to-predict moves like En passant (also in *All_valid_moves_checking.rb*), however, sometimes it may still not possible to completely eliminate the silly cases (but still ones of the best last value moves).

Furthermore, I have also carefully placed the code that executes Black's move in *Pieces_moving_of_black.rb* in Ruby's begin .. rescue instruction, in order to make a random move when the Minimax algorithm can't find the best move (some potential errors).

When it comes to my code, even with alpha-beta prunning, the program still takes a lot of time to make the optimal move for Black, that's because I was not able to optimize the calculation for other functions and methods, including the function that calculates all valid moves, calculates the control area of each side, ... (As I said, my example is not a perfect one).

CONCLUSION

My project, according to the positive results of the experiments, has fulfilled its original purpose - Creating a simple AI opponent for the chess game:

- More than 60% of moves were judged "reasonable" by chess.com.
- With Alpha-beta pruning I have reduced the number of computations in the Minimax algorithm to only about 1/3 of the original number on average.

Nevertheless, since this program is my first 2D 2-player board game containing a simple algorithm of AI, I believe it is very simple and has many shortcomings. In the near future, specifically the next related units next semester, I will consider trying out to optimize the chess program, by making it so that it may learn some moves in the previous matches. Also, It is necessary for me to optimize the core methods for operating the game, as well as the sub-methods for the minimax algorithm, including calculating the chessboard value, calculating all valid moves, ... of the two sides.

Thank you for reading these words.
If you have any questions or comments about this project, including errors or misunderstandings, please email 104053642@student.swin.edu.au.

REFERENCES

[1] Rules of chess. Retrieved from https://en.wikipedia.org/wiki/Rules_of_chess.

[2] Coding Adventure: Chess AI – Youtuber Sebastian Lague. Retrieved from https://www.youtube.com/watch?v=U4ogK0MIzqk&t=809s&ab_channel=SebastianLague and https://sebastian.itch.io/chess-ai.

[3] A step-by-step guide to building a simple chess AI - Lauri Hartikka. Retrieved from https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977.

[4] Tool for testing and evaluating my algorithm – Chess.com. Retrived from https://www.chess.com/home.