# Chess using Alpha-Beta

Liam Marcassa

**Abstract**—A program was written to play chess by evaluating a minimax game tree using alpha-beta pruning. The user is allowed to specify several options, including search depth and the board's starting configuration. Two board evaluation functions were provided, both of which can evaluate the search tree to a depth of six in a reasonable amount of time.

## 1 Introduction

The goal of this project was to develop a chess-playing AI which used alpha-beta pruning to evaluate and select moves. It had to respect all the rules of chess, and decide which move to make in a reasonable amount of time. A complete chess-playing environment had to be presented: the user was able to play an entire game of chess without leaving the program. The minimax search tree had to be evaluated to a minimum depth of four. The reader is assumed to be familiar with the topics of chess and AI.

This report will cover the use, design, and testing of the program. The command line was used for I/O, a GUI being used only for the user to specify initial parameters. The emphasis of the design was on speed, as a deeper search depth was highly desirable. Testing of the finished program focused on usability, correctness, and the utility of the evaluation functions.

## 2 Use

### 2.1 "Hello World"

The `Referee` class serves as the entry point to the program. A GUI opens which allows the user to specify board configuration, which player moves first, which colour moves first, which evaluation function to use, and maximum depth of the search tree. Through the choice of starting player and colour, each player is assigned a colour. The "Reset Board" button toggles between an empty board and the default starting configuration.

Once the parameters have been specified, interaction is performed entirely through the command line. In an attempt to improve readability, the program detects *nix systems and colour-codes the pieces appropriately. Colour-coding did not work when tested on the Windows shell, so black pieces are uppercase and white pieces are lowercase. Empty squares are denoted by a "middle dot" (ASCII #183).

Moves are specified by providing a piece's current position followed by a desired position. For example, white's most common opening move can be made by typing "E2 E4". The user may enter "x" instead of a move and the program will exit. The computer informs the user when it has started it's search, and displays the selected move (in the same format as above) when finished.

As the king is the most important piece, it is denoted by the letter "k" or "K". It would have been confusing to denote knights with the same symbol, so they were given the characters "n" and "N".

The program will detect game-over states and print a message out to the console indicating the winning player or a stalemate. The program will then exit.

A game's moves are logged to a file named "output.txt" in a similar format as above (E2-E4). Each new game overwrites the file. It does not save the initial board configuration, which likely limits its functionality to games played from the default starting configuration.

### 2.2 Option Details

This section will explore depth limits, briefly describe the evaluation functions, and explain what constitutes a valid board configuration.

Depth is coerced into a value between 2 and 10 (inclusive). The minimum value of two is essential for the `Computer` class to be able to recognize game-over conditions (explained in the Code section). The maximum value of ten is largely symbolic, as the program slows dramatically around depth seven or eight. If the user enters a non-numeric depth, it is automatically set to four.

Two evaluation functions were provided: a simple one and a complex one. The simple evaluation function was aimed at speed of execution, and meant to maximize the depth of the search. The complex evaluation function was meant to emulate classical chess play more closely, at the sacrifice of some speed. This offers more choice to the user.

A board configuration must satisfy several conditions to be considered valid. First and foremost, both kings must be present. Otherwise, the AI will have no goal to achieve or outcome to guard against.

Secondly, there can be no pawns on either the first or eighth rank. Pawns only ever move forward, and get promoted upon reaching the other side. Although a white pawn on the first rank is not inconceivable in the scenario of a chess puzzle, this position cannot be achieved from the default starting configuration and is therefore considered invalid.

Lastly, there can be no more pieces than the sixteen per side that each player would start with. Nine queens of one colour is acceptable (all pawns have somehow been promoted), but ten queens or two kings is not. If a user-specified configuration violates any of these three restrictions, a fatal error occurs.

The last two restrictions for a valid board configuration could have been avoided at the expense of code complexity. The restrictions were deemed reasonable as they followed from the rules of a default starting position and allowed for certain assumptions.

# 3 CODE

## 3.1 Design Overview

A game of chess consists of a board, some pieces, and two players. This format is deeply familiar and provides a clear conceptual division of work; it was therefore translated directly into code. For each game, a single `Board` class, two `PlayerInterface` classes, and several `PieceInterface` classes were created. A `Referee` class was added to govern the game and provide an entry point into the program, as the players in this game could not be trusted to follow a turn sequence or declare victory on their own. Several other helper classes will be explained as needs arise.

There were two player classes: `Human` and `Computer`, each implementing `PlayerInterface`. The `Human` class was concerned with console I/O and preventing the user from making an invalid move (for example: putting their king in check). It asked the user to specify a current and desired position, then validated that wish, asking again if the validation failed. The `Computer` class held all the AI components of this project, and did not prompt the user for any input after the initial setup was done. It indicated when it had started computation, and then wrote the selected move out to console. As mentioned above, turn order is governed by `Referee`.

There are six piece types: king, queen, rook, bishop, knight, and pawn (in roughly descending utility). Each has it's own unique set of movement rules. A separate class implementing `PieceInterface` was created for each type. These classes tested mechanically valid moves (bishops move diagonally, knights are the only pieces allowed to "hop over" others, etc.) and were capable of generating a set of possible moves given a piece's position. These possible moves corresponded to the branches of the search tree.

Every other class communicated in part using the `Board` class, which made it the most important one. It's essential purpose was to handle making and undoing moves, but also parsed the initial board configuration and provided mechanisms to detect check and determine if a move was valid. All of this had to be accomplished as quickly and efficiently as possible so the search tree could reach it's maximum depth.

## 3.2 The Board

It was quite natural to index pieces by location, however knowledge of piece type was of equal importance. In order to reconcile both goals efficiently, the compromise detailed in Figure 1 was reached. The variables are such that:

```
board[(56&pieces[i])>>3][7&pieces[i]] == i
```

which allowed a piece to be represented completely by a single `pieces` byte.

The order of `pieceNames` (and therefore of `pieces`) was important for the alpha-beta tree; efficient searching requires that the best move be evaluated first. It was therefore decided to store pieces by descending utility, grouping colours together. `pieces[0]` was the black king, `pieces[1]` the black queen, then rooks, bishops, knights, and finally pawns. The same was true for white pieces, only they started at index 16. Unless a pawn got promoted, the program knew exactly what type of piece was at what index, without having to use `pieceNames`. This was especially useful for castling and determining check.

Fig. 1. Variables Used to Represent a Board Configuration

| Variable | Format | Use |
|---|---|---|
| board | byte[8][8] | Represents a physical board. Value is either an index of the pieces and pieceNames arrays or -128. A2 is at [0][1] |
| pieces | byte[32] | each byte's bits: 0-2→row/rank/y-coor, 3-5→column/file/x-coor, 6→0=captured, 1=in play, 7→0=white, 1=black |
| pieceNames | byte[32] | actually chars representing piece type (K,Q,B,R,N,P) |

When branching the alpha-beta search, the program could start at index 0 or 16 and increment up with the reasonable assumption of coming across a good move early on, as these pieces were of higher value (and therefore needed protecting), had greater mobility (so could capture more), or both.

## 3.3 Design Principles and Rule Coverage

### *3.3.1 Tree Evaluation*

The first major design decision was prompted by a hint given by the TAs: do not generate several hundred thousand separate boards for leaf nodes, as this would likely overflow memory and would generally be unwieldy to work with. Instead, repeatedly make and undo moves on a single board. Following this logic, the most important methods became `boardMove(PlayerInterface,byte,byte)` and `undoMove()`, both in the `Board` class. As will be explained, this decision had several repercussions.

### *3.3.2 Game-over Detection*

Three outcomes are possible: they checkmate us, we checkmate them, or stalemate. Checkmate occurs when a player is in check and they have no moves which get them out of check. Stalemate occurs when a player is not in check but all their moves put them into check. The `Board` could test for these conditions every time it evaluated a move, but this would not be optimal when searching the game tree.

The game-over conditions require testing a player's every possible move, but this already occurs when traversing the game tree. So instead of incorporating game-over testing into `Board`, it can be left entirely to `Computer`. As long as `Computer` calculates check at every node, game-over states are easy to detect throughout the course of a search. A minimum depth of two is required in order to accomplish this, but that's fairly inconsequential.

The function to detect check (`calcCheck()`) had to be modified to accommodate this logic. If the king has been captured, `calcCheck()` indicates that that king is in check. If a player moves and is in check at both the parent and current nodes, that player has lost the game. If all moves result in check at the current node but not at the parent node, the game is in stalemate.

Game-over states are detected by the methods `checkMyMoves(Node)` and `checkTheirMoves(Node)`

in the Computer class, then assigned to a variable called badMove. If badMove is non-zero, that node becomes a leaf node. If a badMove is the best move available to the root node, the game is over. At depths greater than two, badMove is modified to prevent the root node from declaring game over in the case of a forced mate (and instead plays it out). In order to indicate that the game is over, Board.gameOver (a String which is usually empty) is set to a simple message informing Referee and the user.

Testing for check has to be separate from movement execution; if the two functions were incorporated it would lead to an infinite loop. Every move would require calculating check, which would require more moves. Testing for mates was also separated from movement. As a result, the distinction between mechanically valid moves and "check-wise" valid moves is a recurring theme throughout the code.

#### 3.3.3 En Passant

One of the conditions required in order to capture en passant is that the to-be-captured pawn has moved forward two spaces immediately prior. It was therefore beneficial to track exactly which piece moved exactly from where at each step in the game. This also follows logically from the need for an undo function.

A linked list of OldPositions was used for this task. Each OldPosition holds an array of four bytes. These bytes contain all the information that undoMove() needs to know. The first byte contains several flags (such as piece captured, castle kingside, etc.) which facilitate fast reversal of moves. Further bit-level meaning of each byte is outside the scope of this report, but is explained in the code.

#### 3.3.4 Pawn Promotion

If Board detects a pawn promotion, it asks the player what to do by calling choosePawnPromo(). If a pawn promotion occurs while testing check or searching the game tree, it needs to be handled automatically. All possible promoted piece movement is a subset of a queen's or knight's movement, therefore these two pieces are sufficient to test all possible cases.

Knowing this, the Pawn class' move generation function (getMoves(byte)) will return two identical piece promotion moves (along with all the other possible moves) if promotion is viable. Board and Computer simply toggle between promoting to a queen and promoting to a knight. A "dummy" or "shell" player is used specifically to handle these promotion cases.

#### 3.3.5 Castling

Castling is often an essential move in the opening game to provide king safety. Both a rook and a king are involved in castling, but for ease of development, only the king is allowed to initiate castling. This aids the detection and handling efficiency, as only one piece has to be tested instead of two.

In the interest of speed, mechanical movement logic was once again separated from check-wise logic. The CastleSync class manages one part of mechanical castling validation: whether or not the king or rook has moved from it's default starting square. This provides a quick and easy way for any class to eliminate the possibility of castling (and thus prevent further check-wise computation) as soon as a rook or king has moved. Further mechanical validation is done entirely by the King class. Check-wise validation is performed by Board.canCastle(byte,byte).

#### 3.3.6 File Ouput

The game moves are logged out to a file called "output.txt", saved to wherever the program is run from. It overwrites this file on every run. The initial board configuration is not saved, so it may be difficult to reconstruct a game if a non-default starting configuration is used. "Computer style" chess notation is used throughout the program, as it is aptly named after being easier to parse than standard chess notation.

### 3.4 Evaluation Functions

The purpose of an evaluation function is to determine the expected utility of a board configuration. Two different functions were provided; a simple function and a complex function. They were meant to provide the user with a balanced approach to the game.

It was desirable to have some evaluation function that reflected the actual board state for testing whether or not alpha-beta pruning was implemented properly and could find the best move. The simple evaluation function (simpleEval()) arose from this need. It calculates the balance of material and encourages king safety through castling. Relative piece weights were taken as the classical values: pawn = 1, knight/bishop = 3, rook = 5, queen = 9, and the king was chosen to be 200. Castling was plus 2 and moving a rook/king before castling was minus two.

The complex function (bigEval()) attempts to follow some of the guidelines set by classical chess theory. It supplements the balance of material score with several others.

#### 3.4.1 Detecting Game State

Chess games can be broken up into three game states: the opening, the middle, and the end. Different goals are associated with each state; development should occur early on, positional play is important in the middle, and finding forced-check manoeuvres is the goal towards the close of a game. By detecting state, the evaluation function can be optimized for each stage of play.

The opening is marked by little piece development and a majority of pawns on the second rank. Both colours are factored into the algorithm in order to account for non-default start configurations: if our opponent is already developed, we should move right into the middle game.

The middle game is defined in this program as not the opening or end game. As soon as there are less than seven pieces on the board, the board is in an end game state.

The middle game is much larger than classically defined, as named openings exist past move fifteen (largely thanks to computational analysis), and the endgame may start slightly before there are six pieces left on the board. However, the implemented definitions work well with the rest of the program.

Not all classical goals could be implemented for reasons of development time, computational intensity, and poor

definition. When using the complex function, it was still desirable to traverse the game tree at least four nodes deep, and obviously deeper was even better.

#### 3.4.2 Opening Goals

Three goals were identified for the opening: centre control, development, and king safety. Centre control is calculated by counting the number of pieces and pawns which can attack the centre four squares (D4, D5, E4, E5), including how many times they can attack them. If a piece is no longer on the back rank, it is considered developed. The king is safe if it has castled, and the pawn structure in front of the king is in one of two pre-defined arrangements.

No explicit consideration is given to the protection of pieces attacking the centre (or any other pieces for that matter). Protection is somewhat incorporated into the material balance but at one depth up. A database of common (and winning) openings was briefly considered, but ultimately deemed overkill.

#### 3.4.3 Middle Goals

The king is assumed to have castled at this point. There is still a reward for proper pawn structure in front of the king, but only if it has moved to one side of the board. This reward acts as a secondary influence to castle if the king has not done so already.

Aside from king safety, three more goals are considered: pawn aggression, promotion opportunities, and rook mobility. These are significantly more subtle than the goals of the opening game, but the middle game is generally more complex.

Pawns are rewarded for being over the centre line or being a passed pawn. A pawn is considered to be passed if it cannot be captured by any of the opponent's pawns. Passed pawns become vitally important in the end game as they are strong candidates for promotion opportunities and therefore threats. Aggressive pawns can break up weak structures and pressure material advantages.

Finally, rook mobility is generally favourable. Open files (columns) are one's which have no other pieces on them. These allow rooks to protect many other pieces, and are good at threatening back-rank check. If a file contains only pawns of our colour and no other pieces, it is considered half-open, which is nearly as good. Queens tend to favour open and half-open files as well as rooks, but as implemented, the queen is left to her own devices.

The chosen goals of this middle game represent a strategy which looks ahead to the end game. If the computer can survive until then, it will likely be in a good position.

#### 3.4.4 End Goals

End games rely heavily on looking far ahead. For this reason, the maximum search tree depth is increased by two in the end game state, as it is usually computationally feasible to do so with so few pieces left on the board. A slight bonus is added if the king is in a central position, as classical theory encourages the king to be. An end game database was also considered and similarly rejected as being too laborious.

## 4 Testing

The program was mostly developed on a 2016 MacBook Pro. Performance testing was done on a Windows 10 machine with an i5 processor at 3.4 GHz. After several iterations of bug-fixing, performance testing was done to estimate reasonable maximum depth and fine-tune evaluation weights.

Edge-cases for every piece (capture, en passant, castling, etc.) were validated. Game-over detection was validated. It was discovered that the program does not safeguard against many draw conditions: king vs king, king vs king+knight, king vs king+bishop, as well as the "fifty move rule". These rules must be enforced be the user.

At a depth of six, the simple evaluation function executed in under two minutes when tested with several different configurations, including; six queens, all pieces no pawns, and the default starting configuration. At a depth of 7, simple evaluation ran over ten minutes when given a board with all pieces and no pawns.

The complex evaluation function was tested to be even faster than the simple evaluation function. This was not intended, but is a consequence of alpha-beta pruning: diverse evaluation functions will likely find a good move more quickly, so more branches can be pruned. At depth seven, the majority of tested configurations were executed in under a minute. The slowest was a four queen endgame (evaluated to an effective depth of nine) which took less than five minutes. Depth six is recommended for normal play, as it was efficient in all test configurations.

The effectiveness of the complex evaluation weights (rewards for castling, pawn aggression, etc.) was difficult to test for. Given more time, the program could have been set up to play itself with a variety of different weights. Games would have to be played several times, as the "best move" could be picked at random from a set of best moves. Optimization would take the form of a population-based search. Execution of the search would have taken several weeks if the results were to be trusted. As such, weights were validated theoretically and by playing out several games using a variety of weights. These games often ended in the middle game, as the user was not very good.

Overall, the program successfully implemented an alpha-beta search and played chess with a semblance of intelligence. The user is able to play without considerable inconvenience. The two evaluation functions provided insight into the search tree and gave the user more options. The complex evaluation function managed to mirror several aspects of classical chess theory.

## Acknowledgments

Without the TAs Adam Balint and Preston Engstrom, this project would have taken the wrong direction almost immediately.

Many YouTube videos produced by the Chess Club and Scholastic Center of St. Louis were watched to learn about chess theory and identify strategic goals. Mato Jelic's channel was also informative and humbling.