Creating a Chess-Playing Computer Program

Ulysse Carion,
Junior at La Jolla High School

February 2, 2013

Abstract

The goal of this project is to create a computer program that plays a relatively strong game of chess using programming techniques used by the top engines in use today. The result of this project is Godot, a Java program that plays a redoubtable game of chess.

Godot uses bitboards (64-bit numbers representing a chessboard) to implement board representation. When searching for moves, the Godot uses the most common methods of the day, including alpha-beta pruning, principal variation searching, history heuristics, iterative deepening, quiescent searching, static exchange evaluation, and null move pruning. The program evaluates positions by taking into account many factors that are typically an indication of a strong position. Godot can evaluate tens of thousands of positions per second.

Godot also has an opening book based on a large database of thousands of very high-quality games. At the time of this writing, Godot's opening database has a little over 252,000 positions pre-programmed into it.

The program is based on other chess engines, especially open-source ones such as Stockfish, Carballo, and Winglet. Despite being based on other programs, Godot has a distinctive "style" of play that has been repeatedly described as appearing "creative".

Godot has achieved an Elo ranking online of above 2100 at 1-minute chess. It has also defeated multiple FIDE[1]-titled players. Though certainly not nearly as strong as commercial chess engines, Godot certainly plays a very respectable game of chess.

[1] *Fédération Internationale des Échecs*; the International Chess Federation

Contents

1 Acknowledgements **1**

2 Introduction **2**

3 Statement of Purpose **3**

4 Review of Literature **4**
- 4.1 Board Representation 4
  - 4.1.1 Square-Centric Methods 4
  - 4.1.2 Piece-Centric Methods 4
  - 4.1.3 Hybrid Methods 5
- 4.2 Bitboards 5
  - 4.2.1 Binary Operators 5
  - 4.2.2 Implementation 6
  - 4.2.3 Zobrist Hashing 9
- 4.3 Searching 10
  - 4.3.1 Claude Shannon's Types A & B 10
  - 4.3.2 Minimax 10
  - 4.3.3 Alpha-Beta Pruning 12
  - 4.3.4 Quiescent Searching 13
  - 4.3.5 Iterative Deepening, Principal Variation, and History Heuristic 15
  - 4.3.6 Null Move Pruning 16
- 4.4 Evaluation 17
  - 4.4.1 Material 18
  - 4.4.2 Pawn Structure 18
  - 4.4.3 Piece-Square Tables 19
  - 4.4.4 King Safety 19

5 Development **21**
- 5.1 Version 0.1 21
- 5.2 Version 0.2 23
- 5.3 Version 0.3 23
- 5.4 Version 0.4 28
- 5.5 Version 0.5 34
- 5.6 GodotBot 34

6 Results **36**

7 Recommendations **37**

1 Acknowledgements

I'd like to thank Mr. Greg Volger for helping me write my notebook, Dr. David Groce for his support in the creation of this project's presentation, and Mr. Martin Teachworth for guiding me and acting as my mentor.

In addition, I'd like to thank the chess programming community for helping me with more technical issues I faced.

2 Introduction

> "We stand at the brief corona of an eclipse - the eclipse of certain human mastery by machines that humans have created."
> —*Stephen Levy*

Chess-playing programs have captivated the interest of computer scientists since before computers have existed. Alan Turing, perhaps the first computer scientist ever, created his program *Turochamp* and, lacking a computer to run his program with, completed the arithmetic operations himself. In what could be considered the first chess match between man and machine, Turing's "paper machine" lost against a colleague of Turing's (Friedel, 2002).

Much progress has been made since Turing's paper-and-pencil computer. While many chess players asserted that computers would never play at a serious level, claiming that computers lacked the "understanding" of the game necessary to defeat high-rated chess programs, programmers and programs were only getting better. Finally, in 1997, world champion Garry Kasparov lost in a six-game match against IBM's *Deep Blue*, a program that would be considered mediocre by today's top programmers.

Creating a chess-playing program is extremely interesting for many reasons. For one, it is a remarkably large challenge requiring a lot of research, planning, coding, and testing. A chess program is not a trivial assignment; learning the fundamentals of computer chess requires a thorough knowledge of many different "levels" of a computer, from low-level bit manipulation to high-level artificial intelligence algorithms. Secondly, due to the vast amount of work that has been done in the field of computer chess, there exists a great amount of creative freedom for programmers; no two programs will use the same techniques to arrive at the same point. Finally, a chess program is interesting to create because it endows the programmer with the capacity to create "intelligence" out of simple 1s and 0s, a power that has inspired computer scientists for over half a century to keep on pushing the limits of computer chess.

3 Statement of Purpose

Chess has always been at the heart of computer science and artificial intelligence. The goal of this project is to research and develop a chess-playing program that is unique and plays a strong game.

Creating a program that can compete at the level of play of current top-level programs is almost impossible and not the objective of this project—the goal is to create an original program that uses a carefully-selected combination of the hundreds of ways to assault the challenge of teaching a computer to play chess.

4 Review of Literature

Since the beginning of chess-playing programs in 1956, computer scientists and avid programmers have created hundreds of chess engines of differing strengths. Despite the large amount of different chess programs, most programmers choose to use the same time-tested techniques in almost all of their creations. Vasik Rajlich, creator of Rybka, one the strongest chess programs ever, remarked that "when two modern top chess programs play against each other maybe 95% of the programs are algorithmically the same. What is classing is the other 5%" (Riis, 2012).

There exist three major elements to a chess program that must be carefully chosen and experimented with: representation, searching, and evaluation. A strong combination of these three elements is what Rajlich calls the "classing" elements of a program.

4.1 Board Representation

Board representation is the usage of some data structure to contain information about a chessboard. A good board representation makes move generation (determining what moves are available in a position) and board evaluation (determining how strong a position is—more on this in Section 4.4) very fast. There are overall three different methods of representing a board: square-centric, piece-centric, and hybrid solutions.

4.1.1 Square-Centric Methods

A square-centric method is perhaps the most obvious: a program holds in memory what is on each square. Square-centric programs are optimal for quickly figuring out what piece is on a square—going the other way around (viz., determining what square a piece is on) is more difficult. There are many ways to represent a board in a square-centric method. The most obvious (and least effective) method is to use an 8x8 array. Though this may seem simple, it is in fact very cumbersome and requires a lot of work to make sure we do not go outside the array's bounds.

A more common square-centric method is the "0x88" method. This technique works by representing a chessboard as an 8 by 16 array. Though the details of this implementation are complicated (and are not particularly useful to know), 0x88 is quite fast because it takes little effort to determine if a square is in bounds—if a square's index ANDed[2] with the hexadecimal number 0x88 returns anything else than 0, then the index is out of bounds. (Hyatt, 2004).

4.1.2 Piece-Centric Methods

The alternative method, piece-centric board representation, focuses on being able to determine where a piece is rather than what is on a given square. Piece-

[2] See Section 4.2.1

centric methods are optimized for situations where we must find out where a given piece is, but are far from optimal if we must find out what, if anything, stands on a given square. The very first chess programs used such a method called “piece-lists”, wherein a list storing where each piece stands on the board is maintained. These piece-lists were optimal in the early days of computing, when memory was very valuable, because they required very little space on the computer. (Kotok, 1962). The modern variant of piece-centric representation, bitboards, will be explained later in section 4.2 (Hyatt, 2004).

4.1.3 Hybrid Methods

Hybrid solutions involve both piece-centric and square-centric methods, and have the advantages of both methods, at the cost of extra computation to maintain two separate data structures.

4.2 Bitboards

Bitboards are a piece-centric board representation that remains popular to this day. Because most of the strongest engines of this day, including Houdini, Critter, and Stockfish (respectively 1st, 2nd, and 3rd in world ranking at the time of this writing), use bitboards, it is worthwhile to understand how they work.

Bitboards are optimized for computers because they work directly with bitwise operations. Bitwise operations are simple processes that a computer processor uses to work with binary numbers. The five major bitwise operations are the bitwise AND, bitwise OR, bitwise XOR, bitwise left shift, and bitwise right shift.

4.2.1 Binary Operators

The bitwise operations AND, OR, and XOR all take two binary numbers as arguments, and return another binary number as a result. The bitwise AND, for instance, goes through each bit of its arguments, setting the nth bit of the result to be 1 if the nth bit of both of its arguments is 1. For example, if we were to try 10110100 AND 10011000, the result would be:

$$\begin{array}{r} 10110100 \\ \&\, 10011000 \\ \hline 10010000 \end{array}$$

Notice how in the bitwise operator AND, we set the nth bit to “1” if the nth bit of the first number is “1” and the nth bit of the second number is “1” as well. Similar logic works for the bitwise OR operator, which sets the nth bit to “1” if the nth bit of the first number or the second number is “1”. Thus, taking the same example as last time,

$$\begin{array}{r} 10110100 \\ \mid 10011000 \\ \hline 10111100 \end{array}$$

Finally, the XOR operator, or the "exclusive or" operator, sets the nth bit to "1" if the nth bit of the first or second (but not both) is "1". With the same example as before[3]

$$\begin{array}{r} 10110100 \\ \oplus\, 10011000 \\ \hline 00101100 \end{array}$$

The left-shift and right-shift operators are much simpler than the AND, OR, and XOR operators. These operators simply move the bits of a number left or right (as they appear when written out). Thus, 00100 left-shifted by 2 gives 10000, and right-shifted by 2 gives 00001. Note that the left-shift and right-shift operators do not "replace" numbers if they are shifted out; new bits being inserted at either end of a number being shifted are unconditionally 0. If we work with a number that has strictly eight bits,

$$\begin{array}{r} 10001010 \\ \text{left-shift by } 1 = 00010100 \end{array}$$

(Note how the 1 at the left-end of the number was not "replaced" ("carried") back into the result—a zero will unconditionally be placed at the end of the number.)

$$\begin{array}{r} 11001010 \\ \text{right-shift by } 1 = 01100101 \end{array}$$

(Note how 0 was inserted where there was an "opening" as the number was shifted right.)

All of these operators have symbols in programming languages. In C (and related languages, such as C++ or Java) the operations AND, OR, and XOR are represented as "&", "|", and "^", respectively. In Java, left-shift is $<<$, and right-shift is $>>>$ [4].

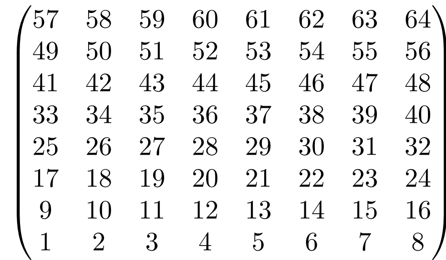
4.2.2 Implementation

Bitboards are considered the best way to represent a chessboard because they are optimized for usage with bitwise operations. The idea is to represent the 64 squares on a chessboard with the 64 bits in a 64-bit number. In Java, these numbers are called "longs" (in C, these are called "unsigned long longs"). We

[3] The example below denotes XOR with the symbol $\oplus$, which is also commonly used to represent the *logical XOR* operator; however, this symbol is not used in any programming languages.

[4] Many languages represent right-shifting with $>>$, but in Java this symbol is used for a variant of bit-shifting that preserves the state of the leftmost bit, an operation called an *arithmetic shift*.

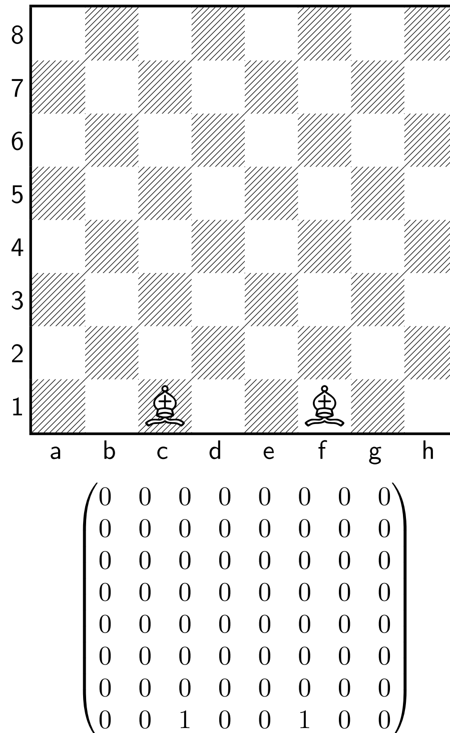
can represent the bottom-left square of a chess board as the last (rightmost) bit of a long, and assign the top-right square to the first (leftmost) bit, as shown in Figure 1.



$$\begin{pmatrix} 57 & 58 & 59 & 60 & 61 & 62 & 63 & 64 \\ 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\ 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\ 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{pmatrix}$$

Figure 1: The position of each bit; board shown from White's perspective.

We can represent where any type of piece is using a single number thanks to bitboards. For instance, if we wanted to represent the position of the white bishops in the initial setup of the board, we would use the number "0000 [...] 000100100", which would correspond to the board shown in Figure 2.



$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Figure 2: White bishops at the starting position.

In the previous example, we saw how we could use a single bitboard to represent the white bishops. In the more general case, we have to represent 6 types of pieces for 2 different colors, resulting in 12 total bitboards minimum. However, most programs also use three extra bitboards, representing white pieces, black pieces, and all pieces. Thus, we use 15 bitboards to represent a chess position. There exist other factors as well (such as whose turn it is to move), but these are trivial to implement.

Representing a board as a number is especially useful when we want to evaluate a position. If we want to find all white pawns that are on black's side of the board, we can do this using one quick bitwise operation. Let's consider the pawn position shown in Figure 3.

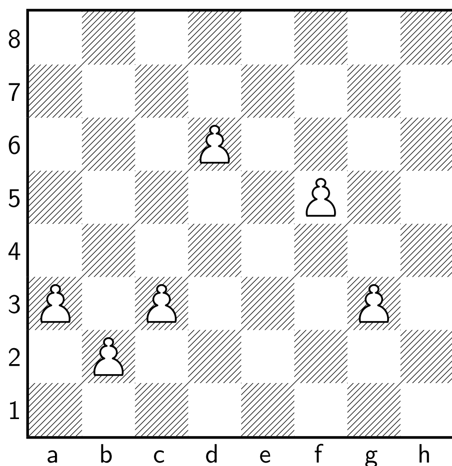


Figure 3: A pawn formation.

The bitboard to represent white's pawns is shown in Figure 4.

$$
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

Figure 4: The bitboard for the pawns in Figure 3.

If we wanted to find all pawns that are on black's half the board, we would simply construct a bitboard representing black's side and AND this number

with our white-pawns bitboard. The result is a bitboard of all white pawns on black's side, as shown in Figure 5.

$$\begin{pmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&0\\ 1&0&1&0&0&0&1&0\\ 0&1&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0 \end{pmatrix} \& \begin{pmatrix} 1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0 \end{pmatrix} = \begin{pmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0 \end{pmatrix}$$

Figure 5: Finding advanced pawns using bitboards.

In one fell swoop, we have can find all white pawns on the opponent's side of the board. Bitboard programs are very fast because they use many of these bitboards to quickly find pieces. This is only a brief summary of a few of the tricks we can enjoy with bitboards—there exist countless methods people have found to optimize their programs thanks to fancy operations with this useful method.

4.2.3 Zobrist Hashing

One of the more difficult-to-handle aspects of board representation is the three-fold repetition rule. This rule states that if a position is visited three times during a game, either side can declare a draw. This rule must be accounted for whenever the program considers a move, but poses a problem: how do we efficiently keep a record of the positions our board has been in? The obvious technique would be to keep a record of our complete previous state (in other words, to have the board keep an array of previous states, or perhaps create a linked list of boards), but this is extremely inefficient for two reasons: it would require much more effort to make a move, and determining if two positions are the same could require a lot of comparisons. There exists an easier solution: Zobrist hashes.

Zobrist hashes rely on representing an entire board using a single 64-bit number as hash of a position. Each element of a position, such as where pieces are, whose turn it is to move, etc. is represented with a 64-bit bitstring which we XOR into/out of the Zobrist hash for our position. For example, if we are moving a pawn from b4 to c5, and we have a Zobrist hash "Z" of our position, we would do the following changes:

$$Z = Z \oplus PawnAt[B4]$$

$$Z = Z \oplus PawnAt[C5]$$

To undo this change, we could repeat the same process. Because XOR is commutative ($A \oplus B \Leftrightarrow B \oplus A$) and associative ($A \oplus (B \oplus C) \Leftrightarrow (A \oplus B) \oplus C$), it doesn't matter what order XOR things "in" or "out", and we can always undo our actions by simply repeating the XOR we have executed.

By using a Zobrist hash to represent our board position, to check for a repeated position we simply need to maintain an array of previous Zobrist hashes of our position and check to see if the same number appears three times. This approach makes our process much faster and easier to implement, if at the price of a (very rare) case of a hash collision[5](Zobrist, 1970).

4.3 Searching

Board representation is only the foundations for the actual point of a chess program: searching. Searching is the process a chess program undergoes to look for and find the best move available in its position.

4.3.1 Claude Shannon's Types A & B

Chess artificial intelligence was formalized and developed in a paper, *Programming a Computer for Playing Chess*, by Claude Shannon. In his seminal work, Shannon described two types of programs, which he called types A and B. Type A would be a "brute force" algorithm—all moves are examined, and the best one is selected; type B would only examine "plausible" moves. Though it may seem that a type B approach would be more effective, all strong programs are nowadays type A—this is due to the fact determining if a move is "plausible" takes too much work to be worthwhile. In fact, because type B programs are essentially nonexistent, I will use the word "search" to mean a Shannon type A search from this point forward.

Shannon's work is also notable for his estimation of the number of possible chess positions possible–he provided the estimated number of different chess positions that can arise. He finds that the number of positions that can arise is on the order of

$$\frac{64!}{32! \times (8!)^2 \times (2!)^6} \approx 10^{43}$$

This number is stunningly high; so large, in fact, that it dispels any hope of creating a chess-playing program that does not perform limited-depth searching (Shannon, 1950).

4.3.2 Minimax

The basis of chess artificial intelligence was explored far before computers existed; in 1928 John von Neumann described an algorithm nowadays known as *minimax* that could help computers make decisions in games including chess (von Neumann, 1928).

Minimax works by considering two players, aptly named "min" and "max". Min's goal is to try minimize the amount of points Max gets. Max's goal, obviously, is to try to maximize his score. Max's technique is to attribute a

[5]The likelihood of a collision in a Zobrist hash is remarkably unlikely—so unlikely that Robert Hyatt (creator of Crafty) and Anthony Cozzie (creator of Zappa) remark that the danger of collisions "isn't a problem that should worry anyone"

score to every move, and then choose the one with the highest score. Min's technique is precisely the same thing, except he chooses the move the lowest score. Max finds out how good a score is by asking Min what he thinks of it, and Min finds out how good a score is by asking Max the very same question.

Of course, there is a problem here—if Max and Min just ask each other what the score is all day, we will never find out what Max's move is. To fix this, we simply say that when we reach a certain *search depth*, we stop searching and just give an estimation of how good a position is using an *evaluation function*. In psuedocode, minimax is shown in Figure 6.

```
function MAX(depth)
    if endOfGame() ∨ depth = 0 then
        return estimation()
    end if
    max ← −∞
    for all moves do
        makeTheMove()
        score ← min(depth − 1)
        unmakeTheMove()
        if score > max then
            max ← score
        end if
    end for
    return max
end function
```

(a) Algorithm for "Max"

```
function MIN(depth)
    if endOfGame() ∨ depth = 0 then
        return estimation()
    end if
    min ← ∞
    for all moves do
        makeTheMove()
        score ← max(depth − 1)
        unmakeTheMove()
        if score < min then
            min ← score
        end if
    end for
    return min
end function
```

(b) Algorithm for "Min"

Figure 6: The Minimax algorithm

Minimax is perhaps best understood as a tree-searching algorithm. If we imagine all the possible moves max and min can make as branches in a tree (branches which, in turn, will have more sub-branches until we reach "leaf" nodes), minimax looks for the best possible score knowing that min will always chose the lowest possible value and max will always chose the highest possible value. A graphical version of this tree is shown in Figure 7.[6]

There exist many optimizations for this basic minimax routine. For example, in zero-sum games such as chess, any gain one player gains is to the detriment of his opponent. This property can be summarized by saying that $max(a, b) = -min(-a, -b)$. This property allows us to summarize Minimax by replacing min and max's cross-calling with a single function calling itself, as shown in Figure 8.

This technique of summarizing minimax into one function is called *negamax*.

[6]Image credit: Wikimedia

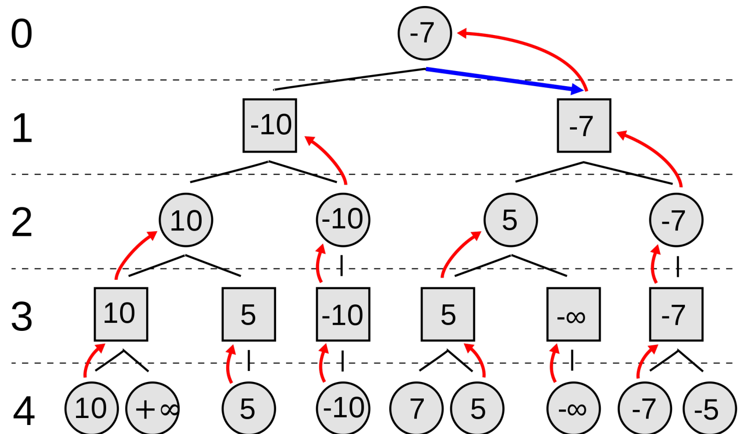


Figure 7: Minimax as a Tree-Searching Algorithm

4.3.3 Alpha-Beta Pruning

Possibly the most difficult-to-grasp improvements to minimax is alpha-beta pruning. This remarkable improvement was developed by Hart and Richards in 1961, and makes searching faster by avoiding searching when a move will never be chosen (Adel'son-Vel'skii et al., 1970; Edwards and Hart, 1961).

M. A. Weiss describes alpha-beta pruning with an example of alpha-beta for a tic-tac-toe-playing program.

> Suppose that the computer is considering five moves: C_1, C_2, C_3, C_4, C_5. Suppose also that the recursive evaluation of C_1 reveals that C_1 forces a draw. Now C_2 is evaluated. At this stage, we have a position from which it would be the human player's turn to move. Suppose that in response to C_2, the human player can consider H_{2a}, H_{2b}, H_{2c}, and H_{2d}. Further, suppose that an evaluation of H_{2a} shows a forced draw. Automatically, C_2 is at best a draw and possibly even a loss for the computer (because the human player is assumed to play optimally). because we need to improve on C_1, we do not have to evaluate any of H_{2b}, H_{2c}, and H_{2d} ... (Weiss, 2002)

An implementation of alpha-beta pruning that uses negamax would look as shown in Figure 9.

The value beta establishes an "upper limit" which, if exceeded, causes alpha-beta to return a value early (which is why alpha-beta is faster than minimax). Alpha serves as a "reasonable score to beat"—if alpha-beta returns alpha, then the position being evaluated was not deemed "too good". Note also that alpha changes value as it finds new positions that return better values.

This particular implementation of alpha beta has the condition that a returned value is always between alpha and beta. In this case, alpha and beta are "hard bounds" and the implementation is said to be *fail hard*.

```
function MINIMAX(depth)
    if endOfGame() ∨ depth = 0 then
        return estimation()
    end if
    max ← −∞
    for all moves do
        makeTheMove()
        score ← −minimax(depth − 1)
        unmakeTheMove()
        if score > max then
            max ← score
        end if
    end for
    return max
end function
```

Figure 8: The Negamax algorithm.

4.3.4 Quiescent Searching

One of the most dangerous aspects of limited-depth searching is when a strong move the opponent can make is missed because the program did not search far enough. For instance, a program examining a long sequence of consecutive captures and re-captures might conclude that it wins an exchange just because it's done searching when in fact, if it had looked a move further, it would have realized that it would in fact gain nothing (or lose material). A remarkable instance of a top-class program losing due to this *horizon effect* of naive fixed-depth searching occurred in 2005 in a "Man versus Machine" triad of games. Grandmaster Ruslan Ponomariov squared up against the program Fritz; the computer made a remarkable error on move 39 (see Figure 10) (Levy, 2005).

Though the sub-optimal quality of this move is only apparent to those who are already strong in chess, it becomes evident that Fritz's hunger for material was myopic. The game continued with

39. . . ♗c2 40 ♕×h4 g×h4 41 ♖c1 ♖×b3 42 ♘×b3 ♗×b3 43 a5 ♘c4 44 b5 ♗a4 45 b×a6 ♗c6 46 a7 ♔g7 47 a6 ♗a8 48 ♖b1

Whereupon black resigned.

As it is clear here, it is important that computer programs should not simply quit searching at some depth. The solution is to use *quiescent searching*. This technique consists of searching deeper (to an indefinite depth, in fact) whenever the computer searches through moves that are tactically potentially explosive—that is, positions where one side that can make a move that can drastically change the evaluation of a position. Quiescent searching forces the program to only evaluate "quiet" ("quiescent") nodes, searching further if there exist moves that have tactically strong moves.

An implementation of quiescent searching would replace alpha-beta's call to an evaluation function with a call to a quiescent-searching function, which

function ALPHABETA($\alpha, \beta, depth$)
  if endOfGame() $\vee$ $depth = 0$ **then**
    return estimation()
  end if
  for all moves **do**
    makeTheMove()
    $score \leftarrow$ alphabeta($-\beta, -\alpha, depth - 1$)
    unmakeTheMove()
    if $score \geq \beta$ **then**
      return β
    end if
    if $score > \alpha$ **then**
      $\alpha \leftarrow score$
    end if
  end for
  return α
end function

Figure 9: The Alpha-Beta algorithm

Ponomariov vs. Fritz

Man vs. Machine, Bilbao—21 Nov 2005

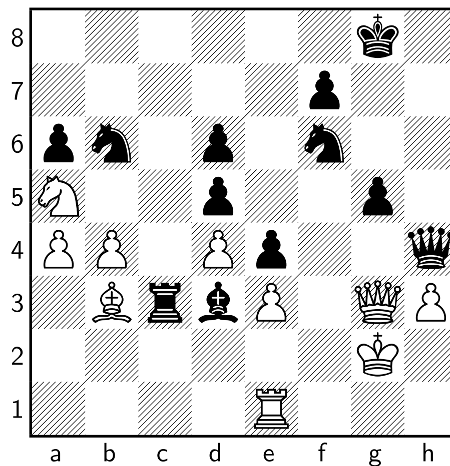


Figure 10: Fritz (playing black) here plays ...Bc2?

would look as shown in Figure 11.[7]

[7] The implementation of quiescent searching shown in Figure 11 uses an early call to an estimation function (a “stand pat” score) in hopes of faster performance by failing hard or by

```
function QSEARCH(α, β)
    standPat ← estimation()
    if standPat ≥ β then
        return β
    end if
    if standPat > α then
        α ← standPat
    end if
    for all capturing moves do
        makeTheMove()
        score ← −qsearch(−β, −α)
        unmakeTheMove()
        if score ≥ β then
            return β
        end if
        if score > α then
            α ← score
        end if
    end for
    return α
end function
```

Figure 11: The Quiescent Searching algorithm

However, this technique can still be improved—we can limit ourselves to only considering captures that are actually winning (this can be done using a technique called *Static Exchange Evaluation*, which I do not explain here). We can also use a clever (if perhaps a bit risky) technique called *delta pruning*, which will skip quiescent-searching if no captures will allow the player to overcome alpha.

4.3.5 Iterative Deepening, Principal Variation, and History Heuristic

Iterative deepening is a creative and powerful optimization for searching, and opens many new doors for improvements to searching. The idea behind iterative deepening is to begin one's search at a shallow depth and iteratively deepen the search until we reach a full depth. Although this may seem pointless (after all, it entails re-searching the same position multiple times), this is in fact very powerful because as we progressively search, we also identify what moves have been the best. We do this mainly through two methods: *Principal Variation Searching (PVS)*, and *History Heuristic* (Korf, 1985).

Before we discuss any optimizations to the alpha-beta routine, it is important to note that most of these adaptations are *Late Move Reductions (LMRs)*—

improving alpha.

because alpha-beta works fastest if the best moves are found first (if good moves are found early on, more moves will be eliminated due to a narrower "window" between alpha and beta), it is oftentimes worthwhile to spend a little extra effort to identify moves that appear to have a lot of potential.

Principal Variation Searching (PVS) identifies the "best move" at the current depth (the move that raises alpha) and stores it in order for it to be found again at the next search depth. Oftentimes (in fact, most of the time), this best move (the *prinipal variation*) remains the best move even as we increase our depth-search, so after we find the principal variation at a low depth (which takes little time), we quickly find the best move at a high depth because we begin our search with the principal variation (Marsland and Campbell, 1982).

History heuristic is the technique of maintaining two 64x64 arrays (one for each side) that represents "from" and "to" squares for moves that cause a value to be returned (be it due to beta cutoff or alpha improvement). When we progress to the next search depth, we will prioritize moves with high values in the history heuristic arrays because these are more likely to cause a beta-cutoff or alpha improvement (which, either way, makes the process of searching faster). History heuristic is a controversial method–though simple to implement, top-level programs avoid it because it ceases to be useful at very high search depths (Schaeffer, 1989).

4.3.6 Null Move Pruning

Null move pruning is a creative method to quickly detect positions that are almost certainly going to cause a beta-cutoff. The technique works based on the assumption that passing one's turn (doing a "null move") is detrimental to one's position.[8] If, even despite the handicap of a null move, a player is still winning in a position (to be more specific, if the player is still causing a beta-cutoff), then it is reasonable to assume that the position is ridiculous and we can move on to search more plausible moves (Adelson-Velsky et al., 1975).

There exist many aspects to null move pruning that vary from program to program. This includes the value of "R", the value representing by how much we reduce our search depth after doing a tentative null-move pruning. Some programs use the value 2, others 3, and yet others chose between these two values depending on the current search depth.

A very important aspect of null-move pruning to keep in mind is the possibility of (relatively rare) positions where there doesn't exist any moves better than the "null move". These positions, which are known as *zugzwang*, occur when any move a side can make will be to the detriment of his / her position. Figure 12 shows an example of a zugzwang position; black's only option is the move Kb7, whereupon white will be guaranteed to be able to promote his pawn and win.

To avoid using null-move pruning in zugzwang positions, one simple solution is to require a certain minimum amount of non-pawn pieces on the board before

[8] Null moves are illegal in chess—nonetheless, chess programs can still use the assumption that a null move is detrimental (this is called the "null move observation") to their advantage.

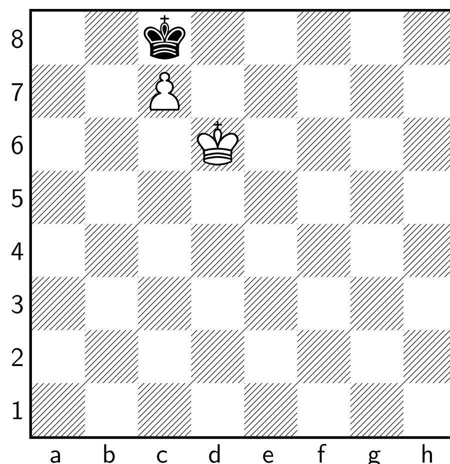


Figure 12: An example of *zugzwang*—Black to move, loses

we attempt a null move; zugzwang positions where there exists even a minor piece are extraordinarily rare and so many programs simply assume that a position with a piece on the board cannot have zugwzwang, though high-level programs may spend more time identifying zugzwang positions.

4.4 Evaluation

Position evaluation is the third and final element to a chess engine. Search optimizations can help a program go faster, but only a strong evaluation function can make it "stronger" (in terms of its knowledge of chess). Deep Blue, the famous IBM computer that defeated world champion Garry Kasparov in a 6-game match in 1997, had an extremely complex position evaluator that had been developed for years by multiple grandmasters.

All searching algorithms eventually reach rock bottom and stop searching any further; instead of going to deeper search depths, they call an evaluator that "estimates" how good a position is for whoever's turn it is to move. If a position is advantageous for the side to move, the evaluator should return a positive value; if the position is not good for the side to move, the result should be negative. A dead draw should return 0.

Most chess players are familiar with the idea of certain pieces being "worth more" than others. For example, it is relatively common knowledge that a rook is more or less worth five pawns. However, some other elements to playing, such as having pieces in good places, are rarely worth as much as a pawn is, even in the most contrived of scenarios. So how do we represent things that are worth less than a pawn? The key is to come up with an imaginary unit, the *centipawn*, and use this value when evaluating. The advantage to using centipawns is that we can easily handle things worth less than a pawn (for instance, something

worth a quarter of a pawn is worth 25 centipawns), and it allows us to work only with integers, which are faster to work with than floating-point decimals are.

A strong evaluation function can take many forms, but always works by assigning a bonus for some element of a position that brings about an advantage to the person whose turn it is to move, and assigning penalties if the opponent has good elements to his/her position. Most evaluators take many factors into consideration when evaluating a position.

4.4.1 Material

By far the most important aspect of position evaluation is material—having more pieces on the board. However, specifically determining what we should assign as values remains a topic of debate. A very popular essay on the matter comes from GM Larry Kaufman, who assigns the values Pawn = 1, Knight = 3½, Bishop = 3½, Rook = 5, and Queen = 9¾. Kaufman also specifically encourages an additional "bonus" of ½ for having both bishops (the "bishop pair") (Kaufman, 1999).

Some programmers may chose to assign variable values for pieces—for example, it may be desirable to make knights be worth more when a position is "closed" (where there is little open space on the board) and make bishops more valuable in "open" positions (where the center of the board is open).

4.4.2 Pawn Structure

In chess, a solid position oftentimes comes from having a very strong pawn structure. Because of the particular way in which pawns can move, these little pieces can simultaneously be remarkably strong or hopelessly useless. It is up to the evaluator to detect strong or weak pawn structure. Most programs penalize three types of weak pawn structure, which are labeled in Figure 13.

In the first diagram, the pawns on a5, b3, and e4 are isolated. In the second diagram, the pawns on the b and f files are doubled. In the third diagram, the pawns on d5 and g7 are backward.

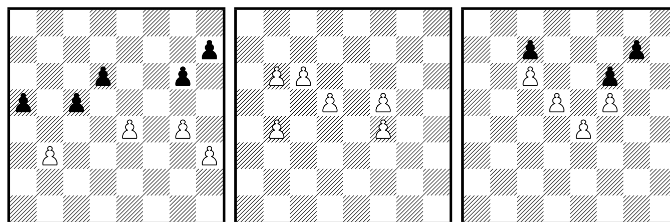


Figure 13: Isolated, doubled, and backward pawns.

4.4.3 Piece-Square Tables

An easy way to encourage a chess program to put its pieces in good squares is to use Piece-Square Tables. These tables are in fact arrays that give a given bonus (or penalty) for a piece on a specific square. For example, a piece-square table for white pawns would likely give bonuses for pawns in the center and for those about to promote. A piece-square table for knights would probably penalize having the knight along the edge of the board ("a knight on the rim is grim").

An example of a piece-square table for white pawns could look like the array shown in Figure 14. This particular piece-square table encourages rapid development of the center pawns and gives bonuses for pawns that have moved forward, especially if they are closer to the central files. Such a piece-square table will likely encourage good piece positioning.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 10 & 15 & 20 & 20 & 15 & 10 & 5 \\ 4 & 8 & 12 & 16 & 16 & 12 & 8 & 4 \\ 3 & 6 & 9 & 12 & 12 & 9 & 6 & 3 \\ 2 & 4 & 6 & 8 & 8 & 6 & 4 & 2 \\ 1 & 2 & 3 & -10 & -10 & 3 & 2 & 1 \\ 0 & 0 & 0 & -40 & -40 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 14: A possible piece-square table for white pawns.

4.4.4 King Safety

King safety is another important aspect to many evaluators. It is generally useful to maintain a strong defensive position for one's king during a game of chess. King safety in a program's evaluator can encourage the program to have a solid pawn shield in front of one's king and discourage the program from allowing the enemy to place his/her pieces in places that are too close to the king.

An easy way to implement king safety would be to give a bonus for adjacent pawns in front of the king (using bitboards to represent places where a "shielding pawn" could stand, and ANDing this bitboard with the bitboard for white's pawns), and give penalties for opponent pieces that are near the king (using an array and a method of finding distances between squares—perhaps with Chebyshev distances[9]).

The world of computer chess is a well-developed one and top-of-the-line programs take years to create. While my program will certainly not do as well

[9]The Chebyshev distance bewteen two points represents the number of turns it would take a king to move between the two points.

as the current reigning champions, it will certainly represent its own unique take at the challenge. The decisions I make when developing my program will determine what techniques will be useful to me further down the line. Each chess program has its own defining characteristics in representation, searching, and evaluation that distinguish it from others and perhaps adds a very human aspect to a deterministic process.

5 Development

5.1 Version 0.1

My first version of my chess engine is finished by early November. This basic program uses basic searching techniques: alpha-beta searching with bitboards, as well as basic evaluation (based on Tomasz Michniewski's simplified evaluation function). During this time I reached preliminary results and very high speeds when searching (especially when compared to my very first attempt, using an 8x8 array with only minimax).

Due to the fact that I spent much of my time waiting for my program to play when testing old versions, I decided to name my engine *Godot* (in reference to Samuel Beckett's play *Waiting for Godot*).

One particularly interesting test to assure that a move generation works correctly is *perft*. This is simply a test case where the program is given a position and is asked to generate all possible legal moves at progressively increasing depths. For instance, if we begin from the starting position, the number of legal moves we have is:

Depth	Moves
1	20
2	400
3	8902
4	197281
5	4865609
6	119060324

Which my programs reproduces correctly. What is perhaps most exciting is the rate at which the program generates these moves: at a rate of just below 1.4 million nodes / second. However, that number will quickly go down when I actually have to evaluate those positions, which will take most of my time.

I make my program use alpha-beta pruning and I now have my first presentable artificial intelligence, although it isn't terribly smart yet. I decide to test my program with a basic chess puzzle, shown in Figure 15. I use this position because the solution is relatively simple and completely empirically decisive. The winning moves are 1 ♕×d1+ ♔×d1 2 ♘f1+ ♔f2 3 ♘×h1, requiring a depth search of 4 (though there are five moves in the sequence, implementations do not count the first move as a search depth).

The preliminary results were slightly disappointing. The data I found is shown in Figure 16. Considering a game of lightning chess (1 min. each side) that takes 25 moves of actual thinking, we would need to take no more than 2.4 seconds of thought per move, thus leaving us with a depth of 2, which is not enough for serious play. Given a 3-minute game of 30 thinking moves, we'd need 6 seconds. For 40 moves in a 5-minute game, we would need 7.5 seconds. So at my current implementation, I could play to depth 2 for anything except the longest of games, where I could perhaps reach depth 3, which remains extremely shallow. Much work remains.

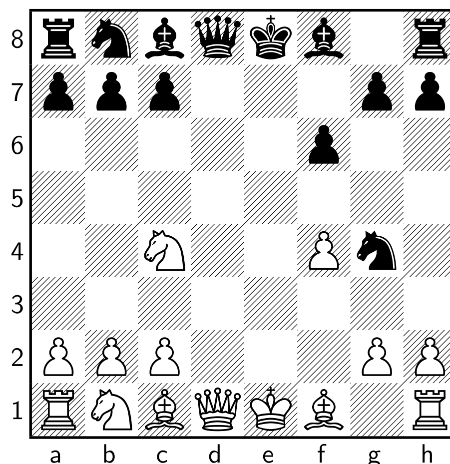


Figure 15: The test position for speed evaluation.

Depth	Time (s)	Nodes	Rate (node/s)
1	.163	1206	7398.773
2	.854	25605	29983.606
3	22.018	793889	36056.363
4	224.256	8307900	37046.500

Figure 16: Results with only alpha-beta pruning.

My next endeavor, then, was PV-searching (PVS). The particular implementation I used was based off of Stef Luijten's Winglet program, which uses a triangular array where the PV at depth N is stored in the Nth row of the array. Of course, with fixed-depth searching, that would mean that the array would have $MaxDepth - N - 1$ entries at Depth N. The results of PVS are shown in Figure 17.

As with most speed-ups, three things can be noted with this optimization: 1) the program is faster (time dropped by 9.44%); 2) fewer nodes are examined (a 16.96% decrease); 3) rate has slowed down (by about 12%). This is due to the

Depth	Time (s)	Nodes	Rate (node/s)
1	.167	1367	8185.628
2	.966	29355	30388.199
3	15.545	506818	32610.281
4	203.086	6898488	33968.309

Figure 17: Results after adding PV searching.

Depth	Time (s)	Nodes	Rate (node/s)
1	.042	43	1023.810
2	.114	157	1377.193
3	.146	1198	8205.479
4	.218	3829	17564.220
5	.764	29777	38975.130
6	7.319	340529	46526.711

Figure 18: Results after adding Iterative Deepening.

program working more effectively with respect to time, at the price of having to do more work for each node. Despite the improvements, Godot still cannot play even semi-decently. More work must be done.

5.2 Version 0.2

Godot becomes a serious program once I've completed iterative deepening. It turns out that most of the moves I make are a total waste of my time—thankfully with PV-searching and history heuristics, I now have a major speedup. The results of the previous test case, now with much deeper searching are shown in Figure 18.

As expected, the program is (much, *much*) faster (nearly a thousand times faster, in fact), examines less nodes, and has a lower node/s efficiency. However, we can see that rate slowly increases with the new implementation, indicating that despite the fact that we are doing much more work each node, we are getting much more payoff. By depth 6, we have a much higher rate than we did with any other implementation. Iterative deepening marks the beginning of a serious program.

At this point, I play my first game against the fledgling Godot, as shown in Figure 19.

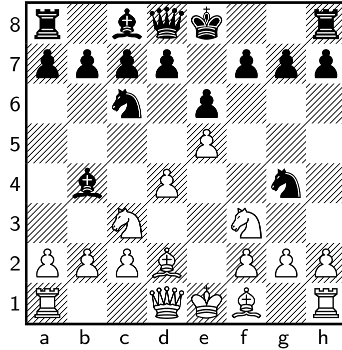
5.3 Version 0.3

What is immediately apparent from the first game (apart from the fact that I am awful at chess, which is perhaps a little ironic at this point) is that Godot needs quiescent searching. In this last game, Godot sacrifices a knight with the move ... ♘×d4 due to the fact that he doesn't see through the whole exchange, only noticing he has the last move and naively concluding this means he wins the exchange. The final result of this type of searching is the program playing embarrassing blunders.

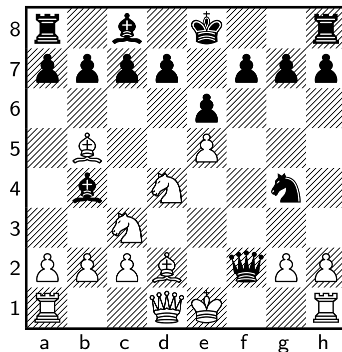
My next goal, then, was to implement quiescent searching. This required that I implement SEE, or Static Exchange Evaluation. This can be implemented using the SWAP algorithm, which avoids the ostensibly recursive nature of a program that looks through all possible exchanges at a square. The final result is a program that is quite a bit slower, but makes very few blunders.

Ulysse Carion vs. Godot
5 minute game

The game begins in a rather usual way, though Godot's lack of an opening repertoire is obvious from move 1.
1 e4 ♘c6 2 ♘f3 ♘f6 3 ♘c3 e6 4 d4 ♗b4 5 e5 ♘g4 6 ♗d2



At this point Godot makes a flat-out blunder with the startlingly dumb move
6... ♘×d4??
I happily accept the sacrifice.
7 ♘×d4 ♕h4
Unfortunately for me I was even more short-sighted than Godot was here;
8 ♗b5?? ♕×f2#



Godot's first victory.

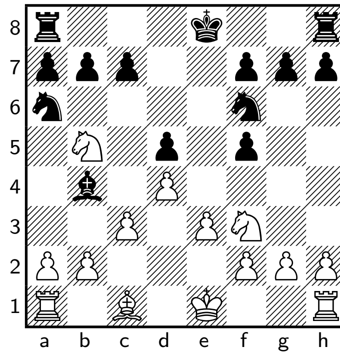
Figure 19: The machine versus his creator.

In order to prove to myself that I've been productive in my efforts, I square off Godot 0.2 against 0.3 to see if my improvements are sufficiently conclusive for a victory.

Godot 0.2 vs. Godot 0.3

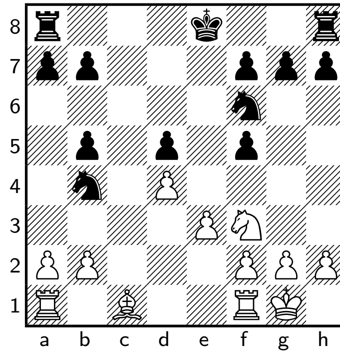
1 minute game

1 ♘c3 d5 2 e3 ♘f6 3 d4 ♗f5 4 ♗d3 e6 5 ♗×f5 e×f5 6 ♕d3 ♕d7 7 ♕b5 ♕×b5 8 ♘×b5 ♘a6 9 ♘f3 ♗b4+ 10 c3



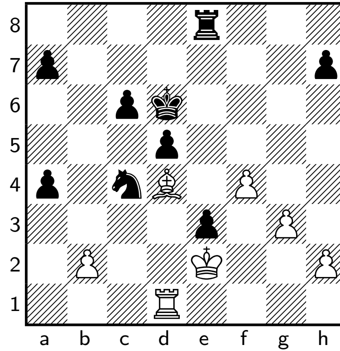
Godot 0.3 here plays a clever . . . c6!, which forces Godot 0.2 to juggle too many balls at once ...

10. . . c6! 11 c×b4 c×b5 12 O-O ♘×b4



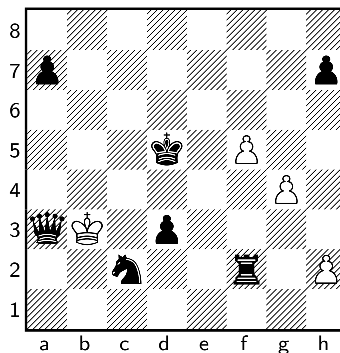
... and Black is now up a pawn and has a bishop for a knight. The game then drifts into an uncomfortable endgame for white.

13 ♗d2 ♘d3 14 a4 b×a4 15 ♘e5 ♘×e5 16 d×e5 ♘e4 17 ♗e1 ♘c5 18 ♖d1 O-O-O 19 ♗b4 ♘b3 20 ♖d3 f6 21 ♖c3+ ♔b8 22 e×f6 g×f6 23 ♗e7 ♖c8 24 ♗×f6 ♖he8 25 ♖d1 ♖c6 26 ♖×c6 b×c6 27 ♔f1 ♔c7 28 ♗c3 ♔d6 29 ♔e2 f4 30 g3 f×e3 31 f4 ♔c5 32 ♗f6 ♘d2 33 ♗c3 ♘b3 34 ♗f6 ♘d2 35 ♗c3 ♘c4 36 ♗d4+ ♔d6



In the style of Cronus with Uranus and Zeus with Cronus, Godot 0.3 defeats its predecessor with great gusto.

37 b4 a×b3 38 ♗c3 c5 39 ♖a1 d4 40 ♗e1 b2 41 ♖b1 ♖b8 42 ♗a5 ♘a3 43 ♖d1 ♔d5 44 ♗c3 b1♕ 45 ♖×b1 ♖×b1 46 ♗×d4 c×d4 47 ♔f3 ♖f1+ 48 ♔e2 ♖f2+ 49 ♔d1 ♘c2 50 f5 d3 51 ♔c1 e2 52 ♔b2 e1♕ 53 g4 ♕a1+ 54 ♔b3 ♕a3#



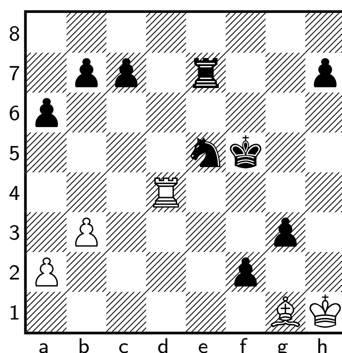
Godot 0.3 Victorious

At this point, I wanted to have an actual evaluation of how good my program really is. This meant having it play online. Because I don't have the money to pay for online membership-based services that would allow my program to play on an online chess server (i.e. Chessbase's PlayChess or ICC), I opted for creating a bot that played games automatically on chess.com's free server. I had Godot play 1-minute chess and to my surprise, it played at a remarkable 2110 Elo, defeating multiple FIDE-titled players in the process.

I include below some of the more interesting games, though I omit any commentary because it would not be worthwhile given the blunder-prone nature of lightning chess.

- **FM** Julian Landaw (2396) v. Godot (First match against titled player)

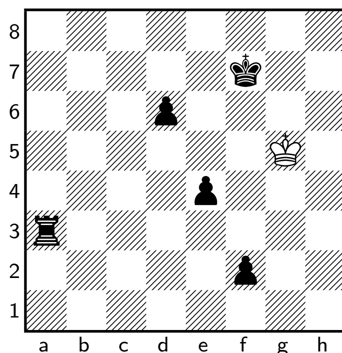
1 e4 ♘f6 2 e5 ♘d5 3 d4 ♘c6 4 c4 ♘b6 5 d5 ♘×e5 6 b3 e6 7 ♗b2 d6 8 d×e6 ♗×e6 9 f4 ♘g6 10 g3 d5 11 c5 ♗×c5 12 ♕c2 ♗b4+ 13 ♘d2 O-O 14 f5 ♗×d2+ 15 ♔×d2 ♕g5+ 16 ♔d1 ♕×f5 17 ♗d3 ♕g4+ 18 ♘e2 ♕f3 19 ♔d2 ♖ac8 20 ♖hf1 ♕h5 21 h4 ♘d7 22 ♘d4 ♘ge5 23 ♗e2 ♕h6+ 24 ♔e1 ♕e3 25 ♘f5 ♗×f5 26 ♕×f5 ♕×g3+ 27 ♕f2 ♕×f2+ 28 ♖×f2 ♖ce8 29 ♔f1 ♖e6 30 ♖c1 ♖c8 31 ♔g1 ♖g6+ 32 ♔h1 ♖d6 33 h5 d4 34 ♖g2 d3 35 ♗g4 d2 36 ♖d1 ♘×g4 37 ♖×g4 f6 38 h6 g5 39 ♖g2 ♔f7 40 ♖d×d2 ♖×d2 41 ♖×d2 ♘e5 42 ♖f2 ♘g4 43 ♖g2 ♘×h6 44 ♖f2 ♘g4 45 ♖f3 ♔e6 46 ♗c3 ♘e5 47 ♖e3 ♔d5 48 ♖e2 ♖e8 49 ♗e1 f5 50 ♖d2+ ♔e4 51 ♗f2 a6 52 ♖e2+ ♔d5 53 ♖c2 ♖e7 54 ♗g1 f4 55 ♖d2+ ♔e4 56 ♖e2+ ♔d5 57 ♖h2 g4 58 ♖d2+ ♔e4 59 ♖h2 g3 60 ♖h6 f3 61 ♖h4+ ♔f5 62 ♖d4 f2



Final position after Landaw loses on time.

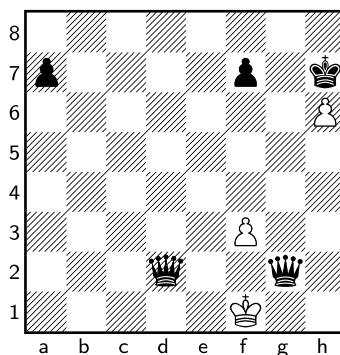
- **NM** Jared Defibaugh (2372) v. Godot (Victory against chess.com's 5th-best player.)

1 e3 ♘c6 2 d4 ♘f6 3 ♘f3 d5 4 ♗e2 e6 5 O-O ♗d6 6 h3 O-O 7 c4 ♗b4 8 ♘c3 ♗×c3 9 b×c3 ♘e7 10 ♕c2 ♗d7 11 ♘e5 ♗c6 12 ♗a3 ♖e8 13 ♗×e7 ♕×e7 14 ♘×c6 b×c6 15 c5 e5 16 ♖ab1 ♖ec8 17 ♖b7 ♖cb8 18 ♖fb1 ♖c8 19 a4 ♕e6 20 a5 h6 21 ♕d1 a6 22 ♕f1 ♘e4 23 ♗×a6 ♘d2 24 ♕e2 ♘×b1 25 ♖×b1 ♖d8 26 ♖c1 e4 27 ♗b7 ♖×a5 28 ♖b1 ♖b8 29 ♖b2 ♖a7 30 ♗×c6 ♖a1+ 31 ♔h2 ♖c8 32 ♗b5 ♖c1 33 c6 ♕d6+ 34 g3 ♖×c3 35 ♗a6 ♖d8 36 ♗b7 ♖×c6 37 ♗×c6 ♕×c6 38 ♖c2 ♕b6 39 h4 ♖a8 40 ♕d2 ♖c8 41 ♕c3 ♕b1 42 ♕c5 ♕b3 43 ♕c3 ♕b1 44 ♕c6 ♕b3 45 ♖e2 h5 46 ♕d7 ♕b7 47 ♕f5 g6 48 ♕g5 ♕c6 49 g4 h×g4 50 ♕×g4 ♖a8 51 ♕g3 ♕c4 52 h5 ♕×e2 53 h×g6 ♕h5+ 54 ♔g2 ♕×g6 55 ♕×g6+ f×g6 56 ♔g3 ♔f7 57 f3 ♖a3 58 ♔f4 g5+ 59 ♔×g5 e×f3 60 e4 d×e4 61 d5 f2 62 d6 c×d6



Final position after Defibaugh loses on time.

- Godot v. **GM** Evan Ju (2353) (Loss against chess.com's then-#1 player.)
  1 ♘c3 g6 2 ♘f3 ♗g7 3 d4 d6 4 e4 ♘f6 5 ♗f4 O-O 6 e5 ♘fd7 7 ♗c4 ♘b6 8 ♕e2 ♘×c4 9 ♕×c4 c6 10 e×d6 e×d6 11 ♗g5 ♖e8+ 12 ♔f1 ♕b6 13 ♘a4 ♕a5 14 ♗d2 ♕h5 15 ♖e1 ♖×e1+ 16 ♔×e1 b5 17 ♕b4 b×a4 18 ♕×d6 ♘d7 19 ♕×c6 ♖b8 20 ♔f1 ♘f6 21 ♔g1 ♗g4 22 ♕d6 ♖×b2 23 ♕d8+ ♗f8 24 ♕×f6 ♗×f3 25 g×f3 ♕h3 26 ♕f4 ♖b1+ 27 ♗c1 ♗h6 28 ♕×h6 ♕×h6 29 ♔g2 ♖×c1 30 ♖×c1 ♕×c1 31 a3 ♕×c2 32 d5 ♕b3 33 f4 ♕×d5+ 34 ♔g3 ♕b3+ 35 ♔g2 ♕×a3 36 h3 ♕b3 37 h4 a3 38 f5 a2 39 f×g6 a1♕ 40 g×h7+ ♔×h7 41 h5 ♕d5+ 42 f3 ♕ad4 43 h6 ♕g5+ 44 ♔f1 ♕d1+ 45 ♔f2 ♕dd2+ 46 ♔f1 ♕gg2#



Final position after ...Qgg2#

The results of online play indicated two things–first, that Godot was definitely playing at an extremely high level (to the extent that I was no longer capable of judging how well it plays purely by looking at its moves), and second, that Godot could still be improved. 1-minute chess is highly tactical, and that humans could defeat it meant that it was not searching deeply enough. I still had many optimizations I could explore at this point. Also, it was becoming quite evident that my evaluation was not very good–by analysing my games using a stronger program, it became evident that Godot was not putting enough emphasis on things such as pawn structure or king safety.

5.4 Version 0.4

Godot 0.4 marked two large improvements over its predecessor: null move pruning and better evaluation.

Null move pruning causes a remarkable speed-up over previous implementations. In order to detect potential zugzwang, positions where null move pruning won't work, Godot does not do null-move pruning if either side has no pieces ("pieces" here meaning any chessman except pawns). The results, which are considerably better than any previous test, are conclusively better and shown in Figure 20.

Null-move pruning again adds a new level of achievable search depth, making Godot even stronger. At 5-minute games, Godot can now reasonably play at depth 6, and is incredibly fast at depth 5. For longer games, Godot can now reach depth 7. However, likely just as important is the new evaluator, which takes into consideration:

Depth	Time (s)	Nodes	Rate (node/s)
1	.138	58	420.290
2	.173	483	2791.908
3	.249	1336	32610.281
4	.425	6271	33968.309
5	.52	9424	5365.462
6	3.928	103167	18123.077
7	13.785	377588	27391.222

Figure 20: Results after adding Null-move pruning.

Piece Values Godot attributes values to pieces according to GM Larry Kaufman's values, which I described in Section 4.4.1.

Pawn Structure Godot penalizes 10 cp[10] for doubled or isolated pawns, and penalizes 8 cp for backward pawns.

Passed Pawns Godot offers a 20 cp bonus for a pawn that is passed (that is, no pawn ahead of it is in an adjacent file).

Rooks Behind Passed Pawns Godot offers an additional 20 cp bonus for a rook that is behind and on the same file as a passed pawn.

Pawn Shield Godot gives 9 cp for a pawn immediately ahead and adjacent to the king, and also gives 4 cp for a pawn two ranks ahead of the king.

Nearby Threats Godot penalizes for having any opponent's piece too close to the king.

Piece Placement Godot also uses piece-square tables, but unlike before they have a relatively small impact on evaluation.

This new evaluator does not cause any measurable slowdown in searching, but certainly does cause the computer to play better.

I also implemented delta pruning, a technique that attempts to speed up searching in quiescent searching by returning alpha if one side is so far above alpha that there is no hope of reaching an equalizing position. To my surprise, this optimization slightly slowed down my program–the extra work necessary to test the "potential value change" in a position, as well as the extra logic involved made the optimization not worthwhile in my program.

At this point, the new program plays considerably better than I do, making testing the program in specific scenarios very difficult. Thankfully, a very strong chess player, Varun Krishnan (a USCF[11] Life Master), volunteered to play against my program. The game went as:

[10]cp = Centipawns (see Section 4.4)

[11] *United States Chess Federation*

1 e4 ♘c6 2 d4 d5 3 e5 e6 4 ♘f3 ♗b4+ 5 c3 ♗a5 6 ♗d3 ♘ge7 7 O-O O-O 8 ♗×h7+ ♔×h7 9 ♘g5+ ♔g8 10 ♕h5 ♖e8 11 ♕×f7+ ♔h8 12 f4 ♘b8 13 ♖f3 ♘f5 14 ♖h3+ ♘h6 15 ♖×h6+ g×h6 16 ♕h7#

Krishnan remarked that my program mostly lacked was an opening book—the moves 1 e4 ♘c6 2 d4 d5 3 e5 e6 lead to a position that strongly favors white. Shredder's opening database, which has nearly 3,000 positions in the position following 1 e4 e6 2 d4 d5 3 e5 (the French Defence, Advance Variation), only finds 2 games where the move . . . ♘c6 was made. It was apparent that it was time to create an opening book.

Although many opening databases exist online and in other programs, I decided to create my own opening database. To do this, I needed a large sample size of high-quality games. I found a large database of games from the Dutch Chess Championship, ranging 25 years from 1981 to 2006. The database had a total of 104,020 games.

To parse the database, the first step was to convert the PGN-formatted database into something my program could use. A PGN game takes the format of:

```
[Event ""]
[Site ""]
[Date "1994.??.??"]
[Round "?"]
[White "Broll, Egon"]
[Black "Fischer, Max Dr(33)"]
[Result "1-0"]
[WhiteElo "1625"]
[ECO "A10"]

1.c4 b6 2.g3 Bb7 3.Nf3 e6 4.Bg2 d6 5.O-O g6 6.d4 Bg7 7.Nc3 Nf6
8.Bg5 O-O 9.Qd2 Qd7 10.Bh6 Re8 11.Bxg7 Kxg7 12.Rae1 c5 13.e4
cxd4 14.Qxd4 Nc6 15.Qd2 e5 16.Nb5 Red8 17.a3 a6 18.Nc3 Na5
19.Qd3 Rac8 20.Nd5 Nxc4 21.b3 Bxd5 22. exd5 Nxa3 23.Qxa6 Nc2
24.Rd1 b5 25.Ng5 Nb4 26.Qa5 Nc2 27.Bh3 Ng4 28.f3 Nce3 29.Bxg4
Nxg4 30.fxg4 Rf8 31.Qb4 h6 32.Nf3 Rce8 33.Rc1 e4 34.Nd4 e3
35.Qxb5 Qa7 36.Ne2 Rb8 37.Qd3 Rfe8 38.Rf4 Qa3 39.Rcf1 Re7 40.Qc3+
Re5 41.Rxf7+ Kg8 42.Qc7 1-0
```

Using a RegEx (*Regular Expression*) (a type of string that can find and replace character sequences), we can parse the entire database into a long sequence of lines, each of them representing a game. I removed any character sequence matching the RegEx:

```
[\\{.+?\\} | \\d+\\. | \\s\\s+]
```

Which removes all comments (which are enclosed between {. . .}), numbers (which take the form of digits followed by a period), and extra whitespace. I also completely ignore any line that begins with "[". The aforementioned PGN now becomes the line:

```
c4 b6 g3 Bb7 Nf3 e6 Bg2 d6 O-O g6 d4 Bg7 Nc3 Nf6 Bg5 O-O [...]
```

At this point, all we have is a long list of moves in what is called *algebraic notation*[12]. My goal at this point was to create a large database of positions and the most common move in that position. Though it may be tempting to simply create a large table of lines of algebraic notation and replies in that position, this would be inadequate. Such an implementation would not consider the move sequences 1 e4 e5 2 ♘c3 and 1 ♘c3 e5 2 e4 to be equivalent, despite the fact that they are[13].

For my program to work correctly with transpositions, it must enter the moves into the actual chess-playing logic, and ask the chess program itself if two lines are equivalent. This means I must convert SAN into computer-formatted moves. Algebraic notation is not optimal for computers, who must try to convert a move represented in legible format into a move usable by the program. This is due to the fact that algebraic notation accounts for the "ambiguity" of a move. Normally, a move representing a rook moving to f3 would be represented as Rf3 in algebraic notation, but if there are two rooks that can move there, then we disambiguate by specifying the file or rank (preferably the former) of the piece moving. This is demonstrated in Figure 21.

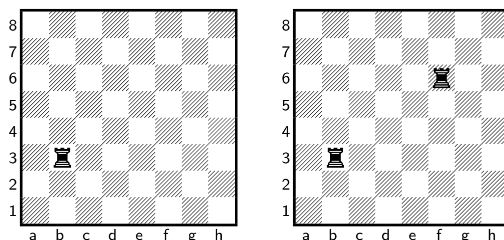


Figure 21: With SAN, Rb3→f3 is Rf3 in the first position, but Rff3 in the latter.

Once I created a simple utility to convert a SAN move into a computer-usable format (using something very similar to Stockfish's method, implemented in Stockfish's *notation.cpp*), I could then easily keep track of what positions had what replies. However, even with a large table of positions and most common replies, I still need a way to easily find the move I'm looking for (assuming it is even there).

The solution was to create a large Tree Map, a data structure that can link keys with values. For each position, I used the position's Zobrist hash as its key, and the move to reply with as its value. The advantage with using a Tree Map is that its implementation is such that the time to find a move is proportional to the logarithm of the size of the Map (in computer science terms, this is referred to as "running in $O(\log n)$"), meaning my opening database will not become much slower even if I make my opening database larger.

[12] Also known as *SAN*, which is short for *Standard Algebraic Notation*.

[13] In chess terminology, two lines leading to the same position are known as *transpositions*.

My final implementation has a very large mapping of positions with moves, giving Godot the capacity to respond to 252,360 different positions without ever having to think. Effectively, this means that Godot will likely be able to play without searching for the first 7 moves of the game.

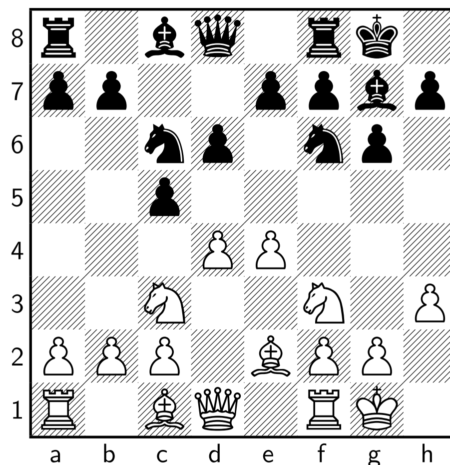
Krishnan against volunteered to play against Godot, this time in the format of a 1-minute game.

Varun Krishnan (2217) vs. Godot

1 minute game

Godot's new opening book, though adding theoretically very little (to no) skill the Godot's actual play, has a strong "psychological" effect when a human faces it. Godot accurately plays the first moves of Pirc Defense, Classical Variation.

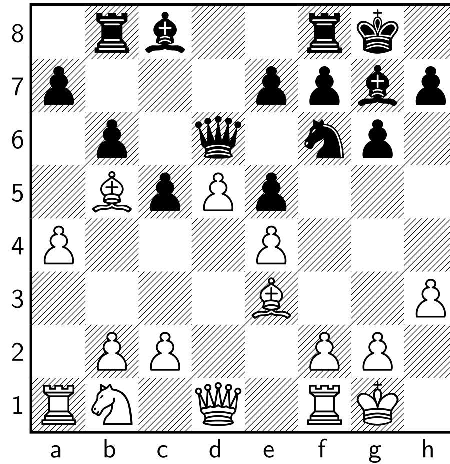
1 ♘f3 g6 2 e4 ♗g7 3 d4 d6 4 ♘c3 ♘f6 5 h3 O-O 6 ♗e2 c5 7 O-O ♘c6



With Godot having consumed less than half a second of thought, the psychological effects of a relatively aggressive computer with time to spare causes a blunder on white's part.

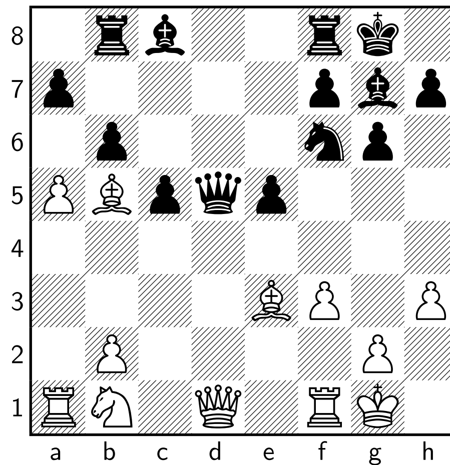
8 d5 ♘e5 9 ♘xe5 dxe5 10 ♗e3 b6 11 ♗b5 ♖b8 12 a4 ♕d6 13

♘b1??



Godot pushes on to capture another pawn...

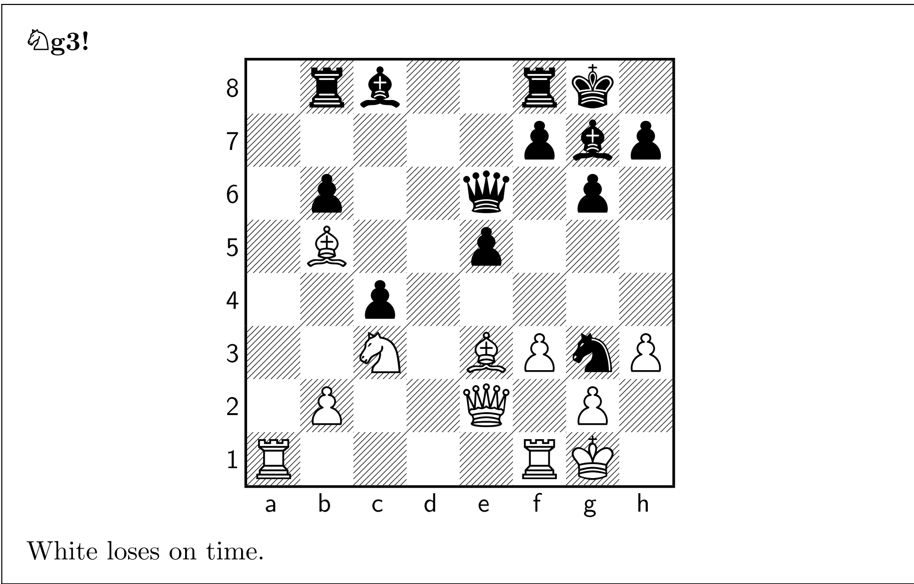
13. . . ♘×e4 14 f3 ♘f6 15 c4 e6 16 a5 e×d5 17 c×d5 ♕×d5



With only a few seconds on the clock and Godot with still over two thirds of his time remaining, white makes another final blunder.

18 ♘c3 ♕e6 19 ♗a4 c4 20 a×b6 a×b6 21 ♕e2 ♘h5 22 ♗b5??

♘g3!



White loses on time.

Other games demonstrate the same pattern—Godot wins most of its games by playing relatively solid moves consistently, winning by taking advantage of its opponent's blunders. Against a very careful opponent, Godot struggles to outperform its rivals, but in rapid chess, when perfect opening play and fast thinking have a powerful psychological effect, the computer can incite mistakes in the humans that face it.

5.5 Version 0.5

In order to make my program quite a bit faster, I decided to change my program from being a hybrid-representation program (containing both piece-centric and square-centric data) to a completely bitboard-based program. My initial program had a few dependencies that ran faster with square-centric implementations (namely evaluation), but Godot 0.4 could easily be implemented in a completely piece-centric way. The only operation that would have to be slowed down with the change is outputting the board, an operation that is almost never executed.

With the new, completely bitboard-based program, Godot is quite a bit faster. With perft, Godot can loop through 2.4 million nodes / second, making the new change a 70% optimization on its predecessor.

5.6 GodotBot

GodotBot is the nickname gave to the new, final online version of Godot. Playing on chess.com, GodotBot is how I evaluate exactly how good Godot really is by putting it up against chess players online from around the world.

The main challenge with GodotBot is that it must communicate over the internet. I chose to implement this by using Selenium, a tool for simulating a web browser. Specifically, I used Selenium WebDriver with ChromeDriver, which allows me to automate a Chrome browser in Java. The algorithm I used is shown in Figure 22.

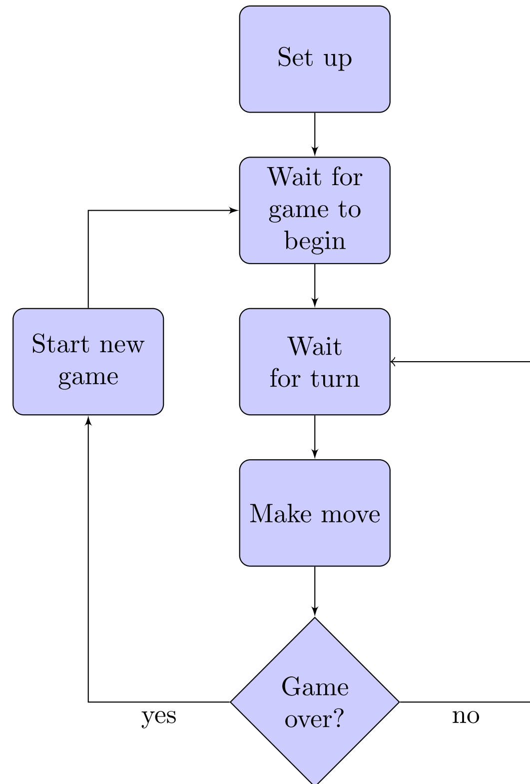


Figure 22: A flowchart of GodotBot.

A very tricky element to master in making the program is to configure how long GodotBot waits for the website to update; it takes a (relatively small) while for the website to be updated after Godot makes a move. Trying to refresh slowly wastes time (which is problematic when playing 1-minute games), but refreshing too quickly raises the chances of Selenium not getting a reply back from the browser, which can cause an "Unreachable Browser Exception", which causes the program to crash.

Through experimentation, I've found that waiting 400-500 milliseconds between each reply is the ideal time; GodotBot will still occasionally (perhaps one in eight games) crash, but the update time is not prohibitively slow.

6 Results

The final result of the project is Godot, a chess program that is surprisingly strong. In sifting through the many options available to me, I made Godot into a bitboard-based program relying on a combination of null-move pruning, iterative deepening, quiescent searching, static exchange evaluation, alpha-beta, PVS, and history heuristics.

For the vast majority of chess players, Godot would be very challenging to play against, especially in short time controls. However, using Godot to set up a position and find the best move available takes only two lines of code. The Godot source code is very user-friendly.

Though Godot is strong, it will not be able to defeat current world-class programs. Godot is not Deep Blue, and will almost certainly lose against Stockfish, Rybka, Critter, or other top-level engines.

The program is

- *Fast* — Godot can generate in excess of 2.4 million nodes per second, and can evaluate over 30 thousand positions per second.
- *Fine-Tuned* — Godot's position evaluation takes into account material, pawn structure, king safety, rook-pawn cooperation, and piece placement to determine how good a position is.
- *Unique* — No single program is the basis for the project's result; Godot is a combination of techniques inspired from world-class programs and is original.
- *Maintainable* — Godot is written in clean code that is easy to maintain.

7 Recommendations

Though Godot is certainly a strong program, there exist countless optimizations and improvements to explore. If I could extend my program, I would look into re-writing Godot in a faster language than Java—a C++ version of my program would likely run considerably faster. In addition, I would also like to look into other popular optimizations, such as transposition tables, aspiration windows, razoring, and others.

References

G. M. Adel'son-Vel'skii, V. L. Arlazarov, A. R. Bitman, A. A. Zhivotovskii, and A. V. Uskov. Programming a computer to play chess. *Russian Mathematical Surveys*, 25(2):221, 1970.

Georgy Adelson-Velsky, Vladimir Arlazarov, and Mikhail Donskoy. Some methods of controlling the tree search in chess programs. *Artificial Intelligence*, 6 (4):361–371, 1975.

D. J. Edwards and T. F. Hart. The alpha-beta heuristic, December 1961.

Frederic Friedel. A short history of computer chess. *Chessbase*, 2002.

Robert Hyatt. Chess program board representations, 2004.

Larry Kaufman. The evaluation of material imbalances. *Chess Life*, March 1999.

Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

Alan Kotok. A chess playing program. Technical report, MIT, December 1962.

David Levy. Bilbao: The humans strike back. *Chessbase*, November 2005.

T. A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. Technical report, Department of Computing Science, University of Alberta, Edmonton, 1982.

Søren Riis. A gross miscarriage of justice in computer chess. *Chessbase*, February 2012.

Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:1203–1212, 1989.

Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(314), March 1950.

John von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

Mark Allen Weiss. *Data Structures & Problem Solving Using Java*. Addison Wesley, 2 edition, 2002.

Albert L. Zobrist. A new hashing method with application for game playing. Technical Report 88, Computer Sciences Department, University of Wisconsin, April 1970.