The Zephyr Project: Development of a Strong Chess Engine

Author: Sam Gibbon (c1013107)
Project Supervisor: Christine Mumford
Project Moderator: Kirill Sidorov

School of Computer Science and Informatics, Cardiff University
CM0333 Individual Project (30 credits)

Abstract

This project has revolved around the design and implementation of a chess engine. The primary aim was to develop its playing abilities to as high an extent as possible within the time available. In other words, the engine was intended to win as many games as possible against both human and computer opponents. While a project of this nature can never be claimed to be totally complete, it was established that the latest version of the program plays chess to a standard well above that of a typical club player. The program uses an array of modern computer chess techniques to achieve this level of play, such as transposition tables, null move pruning and late move reduction.

Acknowledgements

My thanks go to Jonatan Pettersson and Stef Luijten, whose open source chess engines (Mediocre and Winglet respectively) and excellent guides were extremely helpful throughout the project. The enthusiasm of my friends in Cardiff University Chess Society was much appreciated, especially Steve Gardner who put Zephyr through its paces on several occasions. Finally, I should thank my supervisor Christine Mumford for her continued support throughout the year, even when my attendance record for meetings was somewhat lacking!

Table of Contents

Section 1: Introduction

Chess has been a game of interest to computer science for perhaps as long as computer science itself has existed as a subject. As a result, extensive research has been conducted into computer chess, developing techniques that enable computers to not only play a legal game, but to excel at it, consistently defeating even the strongest human chess grandmasters.

This strength is to be commended, but it comes at a price - clarity. Examine the source code of practically any strong chess engine - for example, the otherwise excellent Stockfish - and without prior knowledge of the program structure and the precise techniques used, you are very unlikely to have any idea what any particular block of code is intended to do, or why it is present at all. To truly understand what is happening in a chess engine, there is no better way than to implement your own.

This report is intended as a summary of techniques used in a prototypical chess engine. It details the process of designing and implementing the author's own chess engine - Zephyr - in the hope that the reader may then easily take the concepts presented and apply them to an engine of their own. Finally, a demonstration of the engine's abilities is given in the form of results achieved against a variety of opponents.

Section 2: Background

The concept of a machine capable of playing chess has been around for hundreds of years. While early automatons were mere hoaxes, such as the infamous Mechanical Turk constructed in the late 18th century, genuine chess algorithms were available as early as 1948. Turochamp - the name of which is derived from Alan Turing and David Champernowne, the men who invented the algorithm - was implemented by hand, as computer technology at the time was simply incapable of the task. Each move required roughly half an hour of tedious, error-prone calculation that ultimately resulted in rather poor play. However, the algorithm introduced many concepts that are still used by engines today.

Two years later in 1950, a seminal paper[1] by Claude Shannon outlined a formal structure for a chess engine, discussing in detail various techniques by which the computer may compute a move for any given position. Many aspects of the Shannon paper will be discussed - with modern improvements - in this paper.

It was not until the late 50s that computer hardware caught up with the ideas of Turing, Champernowne and Shannon and the first programs capable of a full game of chess appeared. These programs were weak players, easily beaten by all but the most inexperienced beginner, but they set in motion a wave of improvements. As computers grew more powerful and available chess techniques became more numerous, the strength of the top programs grew ever greater until eventually even the strongest of grandmasters were considered lucky to hold a draw.

What can we draw from this (heavily abridged) history of computer chess? Certainly that the field has enjoyed active research for well over half a century. Modern techniques are cutting-edge as a result, with any further gains marginal at best. Engine strength is not an area in dire need of improvement.

Where, then, is computer chess lacking? What remains to be done? One problem, already mentioned in the Introduction, is a lack of clarity in the top engines. Code becomes horrifically dense and malformed, good programming practice ignored for the sake of trimming excess nanoseconds off a subroutine. While this is to an extent a necessary evil, much could be done to improve the situation without serious loss of speed.

Computer chess is also an incredibly difficult field for newcomers to acquaint themselves with. There are plenty of resources available on the Internet, including an entire wiki[2] with its own didactic engine and countless forums, but these often read like an expert's manual rather than a gentle introduction. If one is not already familiar with the material presented, little progress will be made.

There are notable exceptions to these issues; in particular, the Mediocre[3] and Winglet[4] projects were of great assistance to the author when first studying the field, and these will be referenced throughout this paper. Mediocre provides an excellent coverage of the thought process behind engine development and the potential pitfalls, while Winglet goes out of its way

1 Shannon, C. 1950. Programming a Computer for Playing Chess. *Philosophical Magazine, Series 7, Volume 41*
2 http://chessprogramming.wikispaces.com/
3 http://mediocrechess.blogspot.co.uk/
4 http://www.sluijten.com/winglet/

to ease the process of implementation.

The Zephyr project seeks to combine these strengths, providing a summary of techniques that may then be easily implemented in the reader's own engine, bearing in mind the limited development timespan of a single academic year.

Section 3: Design

We will now consider the design of our own chess program. Section 3.1 discusses programming methodology and what the engine is required to achieve. Section 3.2 covers the design proper for a bare-bones program that will meet the specification. Section 3.3 then suggests improvements to the base program, increasing its playing strength and usability.

3.1: Overall Design & Program Decisions

3.1.1: Agile Development

The Zephyr project, almost by necessity, is to be conducted using an agile software development scheme. This is due to the massive number of potential techniques that may be used in a chess engine. We do not know ahead of time which of these techniques will prove to be most useful; indeed many concepts that work well in one engine are an overall loss in another. Details like this may only be determined with extensive testing, and testing is only possible with a basic framework already in place which we can later build off.

Our initial focus should therefore be to construct this framework, keeping in mind that it is sure to be expanded upon.

3.1.2: Initial Requirements + Derived Classes

What goes into a chess engine? That is, what elements are necessary for a program to play a legal game of chess? All chess engines, however basic, must achieve the following:

An internal representation of the chessboard that allows for the execution of a given move
Finding the legal moves in a given position
Accepting (legal) moves from an opponent
Assessing a position to determine which side is better - evaluation
Looking ahead to determine the move that gives it the best position - search
Determining when the game has ended

These six requirements are the bare minimum for a chess engine; anything extra is done to improve either the playing strength or the interface. Each of these will be examined in more detail in later chapters, but we can already begin to break the program down into classes and objects. For example, consider the first requirement. We can quite easily conceive a class called (say) `GameState` that contains everything relevant to a position; where each piece is, whose turn it is and so on. A method called `executeMove` should be present which modifies the class attributes accordingly. The class may also contain a method for determining if the game has ended, satisfying requirement six.

The second requirement is met with another class called `MoveGeneration`, or something equally descriptive; this contains methods that, when passed an instance of `GameState`, return a list of valid moves, where a "move" in this context is simply another class - Move. The third requirement calls for a user interface of some kind - class name Zephyr - which can use MoveGeneration to validate the user's move.

The fourth and fifth requirements concern the "meat" of the engine; it is the evaluation and search that primarily dictates the playing strength and style of any given program. Two more classes, Evaluation and Search, will cover everything we need. Evaluation has a method that takes an instance of GameState and returns a number representing how good the position is for the player to move. Search contains a method that takes an instance of GameState and returns a Move that should be played.

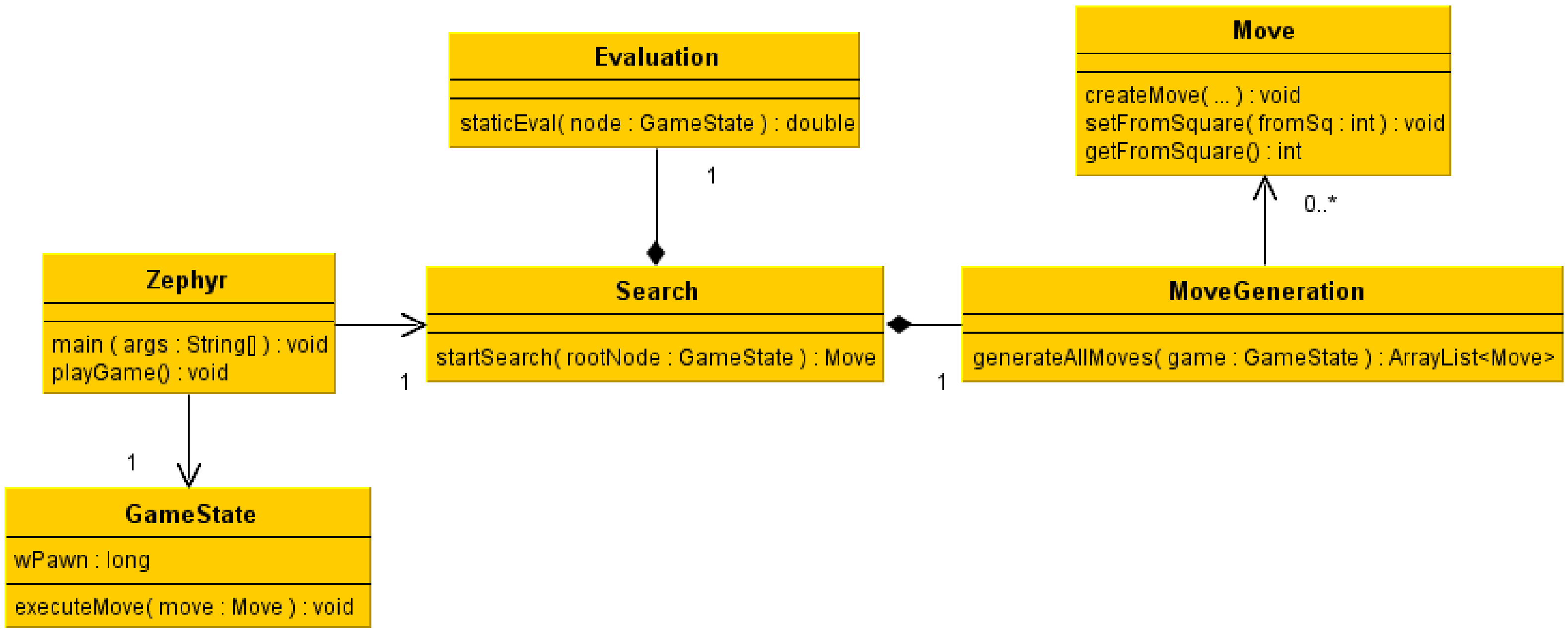

Evaluation
staticEval(node : GameState) : double
1
Move
createMove(...) : void
setFromSquare(fromSq : int) : void
getFromSquare() : int
0..*
Zephyr
main (args : String[]) : void
playGame() : void
1
Search
startSearch(rootNode : GameState) : Move
1
MoveGeneration
generateAllMoves(game : GameState) : ArrayList<Move>
1
GameState
wPawn : long
executeMove(move : Move) : void

Fig. 1: Basic class diagram of the described system.

This is a very simplistic system, but it features everything we need to play full games of chess against a computer-based opponent. While not in the specification, it could be easily modified to allow human-human and self-play games, if so desired. More to the point, we can expand this system as necessary to include any further optimisations. For example, if we were to implement the common transposition tables (see sections 3.3.4 and 4.2) we can easily do so by including a new class called Transposition connected to the rest of the system in the appropriate manner.

Desirable requirements are as follows:

Providing a GUI rather than relying on a command line interface
Obeying time controls and allocating a suitable amount of time per move
Improving playing strength

These will be discussed in more detail in Further Section Design (see section 3.3).

3.1.3: Programming Language

An important decision in any software engineering project is the choice of programming language(s) to be used throughout. A chess engine is an extremely complex project with high resource demands. Our chosen language should therefore be efficient (in the sense of compiling to fast assembly/machine code) and relatively easy to use. Above all, the programmer must be familiar with the language, with a comprehensive understanding of its subtleties and pitfalls.

For the author's personal choice, the three languages being considered were C, Python and Java. The former is by far and away the single most popular language for top-end engines, and it is easy to see why. The relatively low-level nature of C allows the programmer utmost control; indeed, many of the base routines that an engine is built on are implemented using embedded assembly code. Assuming this is done well, this corresponds to a significant speed increase over other languages.

However, for those starting out, C presents many difficulties, not least of which being the complexity of the language. C syntax can be rather obtuse, and the potential for obscure bugs runs much higher. Consequently, C may not be a good choice for a first chess engine, although it would certainly suit an expert well.

Python represents the opposite end of this scale; it is easy to grasp and the syntax does not get in the way of the program logic. However, the triple folly of dynamic typing, automatic garbage collection and being an interpreted language work against it in terms of pure performance. While chess engines written in Python do exist (most notably the engine bundled with the PyChess client), these are very much the exception and do not primarily aim for pure playing strength.

Java aims to strike a balance between these two extremes, attaining reasonable performance while still maintaining a good amount of clarity in the code. Although Java chess engines are still very much a minority, many examples do exist, including the aforementioned Mediocre.

The language used for Zephyr will therefore be Java. However, the reader should select whichever language they feel is most suitable for themselves.

3.2: Basic System Design

3.2.1: Chessboard Representation

The chessboard representation, handled by our `GameState` class, is the backbone of the chess engine. The full list of details we need to keep track of is as follows:

Where each piece is
Which player is to move in the current position
Positions encountered since the start of the game (to enforce draws by threefold-repetition)
Plies[5] elapsed since a capture or pawn move (to enforce draws by fifty move rule)
Castling status for each player
Possibility of an en-passant capture

Other attributes may be added to this list, such as the plies elapsed since the start of the game, or the time left on each player's clock, but the above represents the fundamental subset, the attributes completely necessary in order to support a legal game of chess. Some of these attributes are easily represented; the player to move is a boolean called `whiteToMove`, and the fifty-move counter is obviously an integer. For the others, however, we need to make some decisions.

First, we must have some means of referencing each square. A chessboard has 64 squares, arranged in an 8x8 pattern. It is reasonable to assign each square an integer from 0 to 63 as in Figure 2:

r \ f	a	b	c	d	e	f	g	h
8	56	57	58	59	60	61	62	63
7	48	49	50	51	52	53	54	55
6	40	41	42	43	44	45	46	47
5	32	33	34	35	36	37	38	39
4	24	25	26	27	28	29	30	31
3	16	17	18	19	20	21	22	23
2	8	9	10	11	12	13	14	15
1	0	1	2	3	4	5	6	7

Fig. 2: A possible mapping of integers to squares. Formally, this is a little-endian rank-file mapping.

5 The terms "move" and "ply" are not synonymous; a move consists of a turn by each player while a ply is a "half-move" - a turn taken by either White or Black. In common parlance, however, a chess move is often considered identical to a ply.

From this, we may define two short routines:

```
int file(int sqID)
{ return sqID % 8; }

int rank(int sqID)
{ return sqID / 8; }
```

These return an integer in the interval [0, 7]; if so desired, the reader may modify them to add 1 before returning. This would mean file A corresponds to file 1, and the first rank corresponds to rank 1 (rather than file 0 and rank 0). Zephyr does not make this transformation and any later pseudocode will assume the routines given above.

Next, we must decide upon a method of representing where each piece stands on the board. There are a vast number of ways this can be done, but in general there are two approaches - square-centric and piece-centric. In a square-centric representation, we state what each square contains. In a piece-centric representation, we state where each piece stands and assume any other squares are empty.

Judging by the capability and design of top programs, neither approach has an advantage over the other, though piece-centric is perhaps more common. There exist simple and complex representations of each type:

	Square-centric representation	Piece-centric representation
Simple	8x8 array	Piece list
Complex	0x88, mailbox	Bitboards

Fig. 3: Examples of various board representations

The terms "simple" and "complex" refer to the ease of understanding how the representation is used. In general, a simple representation is easy to work with but is inefficient or inappropriate for many tasks required in a chess engine, and vice-versa for a complex representation.

An interesting approach is to combine a square-centric and a piece-centric representation together, and for each task use whichever is more appropriate. This is of course redundant and comes at a price, as we must now update both representations with every move. However, the expectation is that the speed gained from always using the appropriate data structure will more than compensate for this.

Zephyr will use a combination of bitboards and an 8x8 array. The other viable combination, 0x88/mailbox and piece lists, will not be covered here, though most of the content of this report is still applicable to such programs.

A two-dimensional 8x8 array (or, equivalently, a one-dimensional array of length 64) is perhaps the most intuitive method of tracking piece location. The square-index mapping detailed above is used as an index into the array, so that in the initial position, `board[0]` stores a white rook, `board[59]` stores a black queen and `board[27]` stores a blank space. These may be denoted in any sensible fashion, say the integers 1 through 6 for the white pieces, -1 through -6 for the corresponding black pieces and 0 for an empty space.

r \ f	a	b	c	d	e	f	g	h
8	-2	-3	-4	-5	-6	-4	-3	-2
7	-1	-1	-1	-1	-1	-1	-1	-1
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1
1	2	3	4	5	6	4	3	2

Fig. 4: The initial position, using the mapping 1 = pawn, 2 = rook, 3 = knight, 4 = bishop, 5 = queen, 6 = king, 0 = empty and multiplying by -1 for black pieces.

As mentioned, this is a highly intuitive method of representing the board. It is also very good at answering certain questions, such as which piece, if any, stands on a specific square. However, move generation is rather inefficient using the 8x8 array, especially for sliding pieces; we must check every single square in each direction, always making sure we haven't fallen off the edge of the board. This is a rather laborious task, and must be done for every single piece.

A bitboard is a set of 64 boolean values that represent, square by square, some specific aspect of a chess position. For example, we may have a bitboard that tracks the position of each white rook, or the squares on the fourth rank. The idea is to implement this concept using a 64-bit data type, such as Java's `long`, and then use efficient bitwise operations to carry out move generation, evaluation and so on.

```
8100000000000000
```

Fig. 5: Hexadecimal representation of the white rook bitboard in the initial position. Two bits are set, corresponding to squares 0 and 7 - a1 and h1.

Needless to say, bitboards are rather unintuitive at first, especially compared to the array-based approaches. They are highly abstract, representing an aspect of a position using a single number and manipulating them with low-level bitwise operations. This abstraction leads to difficulty in implementation and debugging. Another issue is that they are poor at answering certain questions, such as which piece (if any) stands on e4.

However, the advantages of bitboards are numerous. The main advantage is the exceptionally fast move generation that becomes possible. We may generate the moves for each piece type in parallel by simply bit-shifting the bitboard the appropriate number of spaces (for more detail, see section 4.1). Fast evaluation techniques are also possible - determining (say) the set of passed pawns may be achieved using only a few bitwise operations.

It should be noted that the performance of bitboards relies on the use of a 64-bit processor. This enables the various operations to be performed using a single clock cycle. A 32-bit processor must break the operations up into stages and combine the result, causing each to take at least twice as long. While the difference is not perceptible to humans, a 32-bit processor will have a detrimental effect on the performance of a bitboard-based chess engine.

3.2.2: Moves Encoding

To construct an adequate encoding for a chess move, we must consider all potential chess moves and their effects on the board. A standard chess move removes a piece from one square and places it on another. The move potentially captures an opposing piece. If it is a king move, it might be kingside or queenside castling. If it is a pawn move, it may be a promotion to any of four pieces, or an en-passant capture in which the destination square is not where the captured piece stands.

In Java, a standard approach to this problem would be to construct a new class called `Move`, containing variables to record the nature of the move and providing accessors to these variables for use in move generation and execution. Indeed, this is the design adopted in our preliminary class diagram (see section 3.1.2). However, while this approach is conceptually pleasing from an object-oriented viewpoint, it runs into practical problems.

A typical middlegame position has around 35 legal moves to choose from. With our object-oriented scheme, this translates to 35 new `Move` objects being created for every position encountered during the search. We can reasonably expect to be searching several hundred thousand positions per second, so we are creating tens of millions of `Move` objects every second. Each of these objects must be instantiated, accessed and destroyed by the JVM, leading to a massive overhead that bottlenecks the engine's performance.

Better is to use a primitive data type. We may fully encode a chess move using a single Java `int` - a 32-bit data type - as in Figure 6:

000000000	001100	011100	0101	000	00	00
Unused	Departure Square	Destination Square	Moving Piece	Captured Piece	Castling	Promotion

Fig. 6: Move encoding bit-for-bit breakdown. The move shown is a white pawn moving from e2 to e4.

The departure and destination squares require six bits each, since the range of possible values are 0-63. This is adequate to describe "normal" moves.

We include the moving piece in our encoding as this saves us having to perform an array lookup every time we want to execute a move. The first bit of the "moving piece" field denotes the colour (0 = White, 1 = Black), and the other three bits denote the piece type, using the semi-arbitrary mapping K=0, Q=1, B=2, N=3, R=4, P=5.

To ease reversal of moves (which will be necessary in the search), we also include the piece to be captured, if any. The "captured piece" field has a similar mapping to the moving piece, but there are a few differences. First, we drop the colour bit - a captured piece must necessarily be of the opposite colour to the moving piece. Second, 0 maps to "no piece captured" rather than "king captured", which is acceptable as a king cannot ever be captured. Finally, we add the mapping E=6, which denotes an en-passant capture. This allows us to differentiate between the moves exd6 and exd6e.p., which would otherwise appear identical.

The castling field is technically unnecessary; we could record castling moves by marking the departure and destination squares as normal and using the fact that no other king move can travel across two files. However, we have the storage space available (the bits will be present regardless) and explicitly stating the castling type saves us some computation when it comes to executing the move. The field uses the simple mapping none=0, K-side=1, Q-side=2.

Finally, we have the promotion field. Since it is only examined when a pawn advances to its final rank (upon which promotion is compulsory), we do not need to include a mapping for "no promotion", meaning we can squeeze the four possibilities into two bits. The mapping used is Q=0, B=1, N=2, R=3.

This scheme leaves 9 bits unused. We will make use of these bits later when discussing the concept of move ordering (see section 3.3), but for now we shall leave them blank. Note that since we are not using the most significant bit in our encoding scheme, any valid move must be positive. This gives us a potentially useful sanity check when examining the legality of move integers; if it is negative, it cannot possibly be legal.

To construct and decipher move integers, we will need a class called `Move` that contains a collection of static methods. Fields are updated and accessed using a mask and bitshift; for example, setting the departure square is achieved using the following snippet of code…

```
private static int setFromSquare(int move, int fromSq)
{
     move &= fromSqMask;
     move |= (fromSq << 17);
     return move;
}
```

…where `fromSqMask` is the number -8,257,537 (FF81FFFF in hexadecimal), chosen so that the bitwise AND sets the departure square field to zero and leaves the rest of the encoded move intact.

Generation

Move generation in chess is a non-trivial task. Given a set of bitboards corresponding to a chess position, we must return a set of integers that represent all legal moves available. If the move generation algorithm misses a legal move, or includes a move that is illegal, then it is essentially useless (though the latter is more dangerous than the former; playing an illegal move is tantamount to immediate resignation, while missing a legal move merely means we do not have the possibility of playing it).

We introduce a new class, `MoveGeneration`, that holds static methods for generating moves given an instance of `GameState`. To begin with we need only one public method called (say) `generateAllMoves` that returns an `ArrayList<Integer>`. However, later down the line we may wish to introduce new methods, such as capture generation and check generation, as these can be useful for various tricks in the search.

Since this method would be very long and complicated as is, we will factor it out into generation methods for each piece type. These methods return a bitboard representing the valid destination squares for pieces of that type, which may then be converted into moves through a process called serialisation. These are useful not only in move generation but in evaluation too.

Zephyr will use a pseudo-legal move generator (as opposed to pure legal). A pseudo-legal move is similar to a legal move except that we do not ensure the king is not left in check. Dropping this constraint simplifies and accelerates the move generation algorithm considerably. To avoid illegal moves caused by this omission, we use a scheme the author has taken to calling "lazy check detection" - delaying the legality check until after the move has been executed by the search. Since the search will not have to search every move[6], this saves us the effort of legality checking moves that will never be executed anyway.

Execution

Executing a move involves updating the piece bitboards and various statistics contained in an instance of `GameState`. To do this, we introduce a void method `executeMove` (and its inverse method `reverseMove`) that takes a move integer as a parameter and modifies `GameState` accordingly. In the interests of speed, the method assumes that the passed integer represents a valid chess move; if this assumption is incorrect, the board may become corrupted or the program may crash completely. Note the use of "valid" rather than "legal" - executing a move that is illegal due to the king being moved into check does not cause any damage, but attempting to move off the edge of the board or to capture a piece of the same colour does.

This is a safe assumption to make during the search, as our move generation returns only valid moves. More dangerous is accepting moves made by the user, who may (accidentally or otherwise) attempt to play an illegal move. Here we must fully validate the move before passing it to `executeMove`, and prompt the user for another move if the first move turns out to be illegal.

6 The reason for this is made evident in section 3.2.4, in the discussion of alpha-beta pruning.

3.2.3: Evaluation

Before we may construct a search algorithm, we must have something to search *for*. The `evaluate` function is a central part of any chess engine. It returns a number that, from the perspective of the player to move, rates how "good" the position is. The unit used in evaluation is typically the *centipawn* (one pawn = 100 units), and this is the unit used throughout Zephyr, but the granularity of `evaluate` is left as a decision for the reader; *millipawn* is probably the next most common unit.

At its core, the evaluation is a collection of heuristics. If a particular aspect of chess is often a good one (passed pawns, bishop pair, etc.), we assign it a bonus to be applied whenever the feature is present. Correspondingly, if a feature is often a bad one (exposed king in the middlegame, doubled pawns, etc.), we assign it a penalty.

The most obvious heuristic - certainly the one given most weight in practically any chess evaluation function - is the amount of material controlled by each side. All else being equal, if White has two rooks while Black has only one, then White has a sizable advantage and most likely will win the game. This being the case, `evaluate` should return a score heavily in White's favour. Pseudocode for an evaluation function containing this heuristic is as follows:

```
     double evaluate(GameState game)
     {
          If this position is a draw by insufficient material,
return a draw score

          int score = 0;

          score += total value of White's pieces
          score -= total value of Black's pieces

          if (White to move){ return score; }
          else { return score * -1; }
     }
```

A few things to note regarding `evaluate`:

Other types of draw will be handled by the search. We could handle insufficient material there too, but the information required is more readily available in the evaluation.
The total material value of a set of pieces is merely the sum of the base material values of each piece, excluding the king. The base values used for each piece vary widely; humans tend to use a scale of P=100, N=300, B=300, R=500, Q=900 for its simplicity, but engines can (and generally do) fine-tune this.
The final IF statement causes `evaluate` to return a score relative to the side to move, rather than strictly from White's perspective. The reason for doing so will become evident when discussing Zephyr's search algorithm (see section 3.2.4).

This function is highly simplistic, but is sufficient for the engine to play better-than-random moves. It will seek to capture enemy pieces while avoiding the capture of its own. However, the complete lack of positional knowledge means that any human over a beginner standard will readily beat the engine. To counteract this, we may introduce some simple positional concepts.

```
     double evaluate(GameState game)
     {
          If this position is a draw by insufficient material,
return a draw score

          int score = 0;

          score += total value of White's pieces
          score -= total value of Black's pieces

          int pawnEval = 0;
          for (each doubled pawn White has)
          { pawnEval -= doubledPenalty; }
          for (each doubled pawn Black has)
     { pawnEval += doubledPenalty; }
          for (each isolated pawn White has)
          { pawnEval -= isolatedPenalty; }
          for (each isolated pawn Black has)
     { pawnEval += isolatedPenalty; }
          for (each passed pawn White has)
          { pawnEval += passedBonus; }
          for (each passed pawn Black has)
          { pawnEval -= passedBonus; }

          score += pawnEval;

          int kingEval = 0;
          if (white king not on first or second rank)
          { kingEval -= exposedKingPenalty; }
          if (black king not on eighth or seventh rank)
          { kingEval += exposedKingPenalty; }

          score += kingEval;

          if (White to move){ return score; }
          else { return score * -1; }
     }
```

This function is still very simplistic (for example, in the endgame we'd rather have the king in the centre than on the edge), but it demonstrates how the evaluation procedure works. We check the position to see if some property holds, and if so, we adjust the score appropriately.

The main difficulty then becomes determining appropriate weights for each of the evaluation terms. Consider, for example, `isolatedPenalty` in the above pseudocode. We know that (in general) an isolated pawn is weaker than one that is not, but how much weaker? This essentially amounts to a series of optimisation problems, and many methods exist for solving these; in particular, CLOP (Confident Local Optimisation)[7] is commonly used for chess engines. However, for the moment we may satisfy ourselves with a reasonable guess, say 20 centipawns. Optimisation is generally reserved for much later down the line of development and requires tens of thousands of games to be played before much confidence can be placed on the results.

Development of an evaluation function is a never-ending endeavour; indeed, it is often claimed that no perfect chess evaluation function exists. Quirks in the evaluation play a major role in the engine's style of play; for example, if we wish for our engine to be especially aggressive, we may introduce large bonuses for placing its pieces near the opposing king. Care should be taken, however, to avoid an excessively complicated evaluation, as this will be more difficult to maintain and risks slowing the search to a crawl.

3.2.4: Search

Chess is a zero-sum game; in any given position, if White has an advantage of x units, then Black’s disadvantage is x units in magnitude. There are several well-known approaches to computing a strategy/move for a position in a zero sum game, such as the Nash equilibrium and the minimax algorithm. These are equivalent in the sense that they produce identical strategies.

The Nash equilibrium is not especially suited for the game of chess, however, as it assumes knowledge of the other player’s plan in a game where at best we can only make educated guesses. It is also hindered by the fact that moves are made alternating between White and Black, whereas the Nash equilibrium is more helpful in those games where players make moves simultaneously (e.g. rock-paper-scissors, iterated prisoner’s dilemma). While refinements to the Nash equilibrium that deal with these issues do exist, such as the subgame perfect Nash equilibrium, we will be basing our search algorithm on the minimax tree-search algorithm.

Minimax Tree Search

A minimax strategy is one that aims to *minimise potential loss*. In the context of chess, say White is trying to decide between move A, move B, move C and move D. For each of these moves, we look at the potential replies from Black and assume he will play the “best” move - the one that gives him the best overall position (from his perspective). Since chess is zero-sum, this is equivalent to the move that gives White the worst overall position.

So in our example, if Black’s best reply to move A results in a position score of -200 (two pawns advantage to Black), +125 in reply to move B, -500 in reply to move C and +50 in reply to move D, White’s best choice would be move B, as this minimises the loss he is about to incur.

7 Coulom, R. 2011. CLOP: Confident Local Optimization for Noisy Black-Box Parameter Tuning. *Advances in Computer Games 13*. Tilburg, Netherlands, 20-22 November 2011.

The minimax algorithm itself is rather naive, however, in that it looks at every single legal move at every single ply, often evaluating moves that cannot possibly be better than something already found or moves that have already been evaluated in similar positions. This behaviour leads to exponential growth of the game tree, quickly making deep searches intractable - a typical chess position might have 35 legal moves available, making for roughly 1.9 billion nodes searched after six plies. At a rather generous rate of one million nodes per second, this search would take over half an hour, and three moves by each side is not especially deep analysis!

Fortunately, there are many techniques available to cut down the branching factor of this tree, making it far easier to conduct deep searches. Note that reducing the effective branching factor to 15 results in a tree with roughly 12 million nodes after six plies, less than 1% of the case with branching factor 35 and taking twelve seconds to evaluate fully. Further note that a chess engine with effective branching factor 15 would be considered rather poor!

The pseudocode that follows is for what is known as the negamax algorithm:

```
      double negamax(GameState game, int depth)
      {
            If this position is a draw by threefold repetition or
fifty-move rule, return a draw score

            if (depth == 0){ return evaluate(); }

            double bestScore = -1 * Infinity;
            Generate all legal moves
            for (each move)
            {
                  Execute the move
                  score = -1 * negamax(game, depth-1);
                  Reverse the move

                  if (score > bestScore){ bestScore = score; }
            }

            return bestScore;
      }
```

This function is essentially a simplified version of the original minimax algorithm that is easier to implement and maintain. It relies on the mathematical identity `max(a, b) == -min(-a, -b)` together with the zero sum game property of chess; the player to move thus looks for the move that maximises the *negation* of the resulting position - in other words, the move that does the most damage to the opponent's position.

Observant readers will notice `negamax` is listed as returning a `double` despite `evaluate` always returning an integer. This is done to accommodate draw scores, which are important to keep distinct from the regular scores. Zephyr uses the scores 0.5, 0.25, 0.125 and 0.0625 to denote draws by stalemate, insufficient material, threefold repetition and fifty-move respectively, chosen for their exact representation in floating point, along with the fact that they fall within the grain of `evaluate` (a standard non-drawn position will never return these scores by chance) and so are treated the same as an exact zero score in the `if (score > bestScore)` inequality above.

Alpha-Beta Pruning

The negamax algorithm is sufficient for a chess engine to traverse the game tree and select a move (actually, we need a way to extract the move associated with the returned minimax score; see the next section). However, the speed of this traversal will be rather slow, even on high-performance machines; as mentioned previously, even at the very optimistic one million nodes per second, a search of six ply will take over half an hour. Eight ply - still a rather shallow search for anything outside the early middlegame - would require twenty-six days.

An improvement to this performance would seem to be necessary. Fortunately, there exists a modification that is trivial to implement and yet boosts the performance remarkably. Specifically, in the best case with branching factor `b` and search depth `d`, naive negamax has computational complexity `O(b^d)` while our improved algorithm will have computational complexity `O(sqrt(b^d))`[8]. Essentially, in the best case we may search twice as deep with the same amount of computation.

The improvement is called alpha-beta pruning, and explaining how it works is probably best done by example. Say we are White, and using a two-ply search to decide between moves A and B. We fully examine move A, and find that Black's best reply leaves the game completely even. We then move onto move B, and upon checking Black's first possible reply, we find that this would give a position evaluated at -100, a pawn's advantage to Black. Since this reply exists, the best we can hope for by making move B is the loss of a pawn. Other moves by Black are either better for White (and thus will not be played) or even worse for White, so there's no point checking them - we can skip the rest of the subtree and play move A.

Implementing this concept is done by expanding the `bestScore` variable into two variables `alpha` and `beta`; the pseudocode is given below.

8 Knuth, D. and Moore, R. 1975. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, pp. 305-306.

```
    double negamaxAB(GameState game, int depth, double alpha,
double beta)
    {
         If this position is a draw by threefold repetition or
fifty-move rule, return a draw score

         if (depth == 0){ return evaluate(); }

         Generate all legal moves
         for (each move)
         {
              Execute the move
              score = -1 * negamaxAB(game, depth-1, -beta,
-alpha);
              Reverse the move

              if (score >= beta){ return beta; }
              if (score >= alpha){ alpha = score; }
         }

         return alpha;
    }
```

`alpha` is identical to our old `bestScore` variable, corresponding to the maximum score that the player to move can achieve in this node. `beta` is similar, corresponding to the maximum score that their opponent can achieve. If our opponent finds a good reply (`if score >= beta`), we do not want to play the move that led to this position; leave it (`return beta`) and move on to the next subtree.

The tricky part to understand is the call to `negamaxAB(game, depth-1, -beta, -alpha)`. The crucial point is that we have swapped and negated the values of `alpha` and `beta` - this is because we are now examining the position from the perspective of the other player. `alpha` is the score associated with our best move so far; when searching later moves, if the opponent has a reply that is scored better than the negation of `alpha`, the move cannot be better, so we stop searching it immediately. If none of the opponent's replies are scored better, then we have a new best move and we may update our `alpha` to reflect this.

A useful idea is to think of `negamaxAB` as a black box which we feed two parameters, `alpha` and `beta`. If `x` is the true minimax value of the node, then `negamaxAB` may be thought of as a function `f` defined as follows:

```
    f(alpha, beta) = alpha, if x <= alpha
              beta, if x >= beta
              x, otherwise
```

In other words, if the true score `x` is (say) above `beta`, we do not really care how *much* higher than `beta` the true score is, so we might as well just return `beta`. This is referred to as a fail-hard framework. An alternative known as fail-soft returns `x` in this situation instead, which has some advantages but loses the black box property; in fail-soft, the returned score is not necessarily within the interval `[alpha, beta]`. For the moment, we shall remain with the conceptually simpler fail-hard.

Move Extraction

The algorithm in the previous section is sufficient to traverse the game tree to a reasonable depth. However, as it is, it only returns the score associated with the best move, and not the best move itself.

There are many methods of solving this problem, but they ultimately fall into one of two categories - handling the root node separately to the rest of the search, or tracking the principal variation (PV) as the search progresses. Both approaches have their advantages and disadvantages, but Zephyr will use the second approach. This is because handling the root node separately duplicates a large chunk of code, making the program harder to maintain.

We track the PV using a triangular array. This is an array of principal variations indexed by the distance from the root node. The maximal length of the PV decreases as this distance increases, hence the triangular structure of the array.

The code is embedded inside our alpha-beta routine as follows:

```
    double negamaxAB(...)
    {
        triangularLength[rootDist] = rootDist;

        ...

        for (each move)
        {
            ...

            if (score >= alpha)
            {
                alpha = score;
                triangularArray[rootDist][rootDist] =
thisMove;
                for (int i = rootDist+1; i <
triangularLength[rootDist+1]; i++)
                {
                    triangularArray[rootDist][i] =
triangularArray[rootDist+1][i];
                }
                triangularLength[rootDist] =
triangularLength[rootDist+1];
            }
        }
```

```
            ...
    }
```

When we arrive at a node, we record the current length of the variation. Then, whenever we find a move that improves alpha (and thus is our current best move for the node), we record it in our triangular array and append the PV of the subsequent node. Finally, we copy down the length of this new PV.

This procedure causes the PV to propagate up to the root. Once the search is complete, the PV for the root node is stored in `triangularArray[0][i]` for `i = 0, 1, ..., triangularLength[0]`. Specifically, the best move for the root node may be found in `triangularArray[0][0]`.

3.3: Further System Design

3.3.1: GUI Communication Protocol

The system as currently designed uses a command line interface. While this will work perfectly fine, a better solution is to use a GUI. Computer chess practitioners are rather fortunate in that there exist plenty of ready-made chess GUIs, many of them free and open-source. The GUI used for Zephyr is called Arena[9], chosen for its relative ease of installation.

Engines may communicate with a GUI using a standardised protocol; the choice here is primarily between the Chess Engine Communication Protocol (also known as the WinBoard protocol) and the Universal Chess Interface (UCI) protocol. Each protocol has its advantages and disadvantages, but the WinBoard protocol is more suited to Zephyr's overall design and so this is the one that will be used, assuming enough time is available to implement it. Since handling the protocol commands is a matter of interface, the task is delegated to the `Zephyr` class, although having a separate `ZephyrGUI` class is also feasible.

The protocol itself is very simple. Commands are submitted from the GUI to the engine through standard input, and from the engine to the GUI through standard output. While waiting for a command, the engine should do nothing. Once a command is read in[10], we respond to it appropriately; for example, the command `new` requires us to restore the game back to the initial position with the engine playing as Black. Once the command has finished executing, we resume listening for commands.

Some commands from the GUI have associated arguments. An example is `usermove`, which is sent when the opponent makes their move. If the opponent moved their queen from g5 to d2, the submitted command will be `usermove g5d2`, and our protocol handler must be able to handle this. A few commands have multiple arguments.

Many commands, such as `go`, require a response from the engine. The `go` command tells the engine to search the current position and respond with the move to be played. This is done by simply printing a `move` command with the appropriate argument to the console; if we wish to move our rook from b1 to d1, we print the command `move b1d1`.

3.3.2: Time Controls

A time control is essentially a time limit imposed upon a game of chess; if a player runs out of time, they immediately forfeit the game[11]. In the context of computer chess, a time control is necessary for the computer to spend a reasonable amount of time per move, rather than always searching to a fixed depth no matter how long it takes. Once this functionality is in place, games against the engine will be more balanced and we will be able to conduct tournaments against other engines.

The implementation of time controls requires several classes to be modified. The

9 Freely available from http://www.playwitharena.com/

10 Zephyr uses Java's `BufferedReader` wrapped around an `InputStreamReader` for this purpose, though any standard library reader class should suffice.

11 This is not strictly accurate; see glossary entry for Time Control.

available time itself is an aspect of the chess position, so we may expand `GameState` to include variables `wTime` and `bTime`, representing each player's remaining clock time. Some time controls include an increment - increasing the available time after each move - so another variable `increment` is necessary. To allow for the possibility of time odds games, where the weaker player is given more time than the stronger, we may wish to split this variable into `wIncrement` and `bIncrement`.

The user must be able to select a time control before beginning each game; this is to be handled by `Zephyr`, modifying the interface as necessary. The WinBoard protocol includes the `level` command, which sets the time control, and the `time` and `otim` commands which update the clock for the side to move and the side not to move respectively.

Finally, the engine must be able to determine how much time is available to search each move. It must also be able to halt the search and immediately return a move once this time has elapsed. To achieve this, `Search` will be expanded too.

Determining the appropriate amount of time to search for is essentially trivial but may become arbitrarily complex. Certainly when using a time control such as 40/5+2 (meaning the clock starts at 5 minutes, every 40 moves another 5 minutes are added, and an increment of 2 seconds is added after every move), the computation is very simple; if we have 3 minutes left and must make 12 more moves to reach the time control, we search for 3/12 minutes + 2 seconds, or 17 seconds. More sophisticated algorithms might modify this time depending on the position, how the search is progressing, and so on.

Blitz time controls, where we must make all the moves within a given time limit, are more difficult. Zephyr handles these types of time control by making an educated guess as to how many moves are left in the game, and spreading the available time across them. For an example, we may guess that in the opening there are 60 moves left, in the middlegame there are 35 moves left and in the endgame there are 15 moves left. The problem is thus reduced to determining which game phase we are in, which we'll probably be doing regardless as this information can also be useful for the evaluation function.

A further extension is to introduce a "panic mode", which dictates the engine's behaviour when extremely short on time. This is especially useful with a time control that uses increments, as we can try to build up some time by forcing searches to be faster than the increment. Once the time is sufficiently built up (either by meeting the time control or using increments), we may leave panic mode and resume normal time management.

3.3.3: Iterative Deepening

In order to obey a time control, we must make use of so-called iterative deepening. The problem with conducting a fixed depth search is that we do not know in advance how long it will take to complete. A depth 10 search may take a long time in a messy middlegame position while completing almost immediately in a KPK endgame. We could attempt to guess how long the search will take based on various characteristics of the position, but this is error-prone and difficult to maintain. Iterative deepening provides a much more elegant solution to the problem and even has a side effect of speeding up the search.

Instead of the fixed depth search, we first conduct a search of depth one. Once this has completed, we conduct a search of depth two, then depth three and so on. This process continues until the allotted time has elapsed, at which point we abandon the search and return the best move found by the most recent completed iteration. Since a depth one search completes almost instantaneously, we are all but guaranteed to have some move to play when the time runs out. With a fixed depth search, we would have no choice but to keep going until the search completes, potentially losing on time in the process.

One would expect that reaching a search depth of `n` through iterative deepening would take longer than simply conducting a fixed depth `n` search, but paradoxically this is not the case. The reason is that the shallow searches provide a principal variation, which we may use to guide the deeper search. The moves in the principal variation are referred to as "PV moves" and are helpful in move ordering, a technique discussed in the next section.

3.3.4: Methods of Increasing Playing Strength

Our final desirable requirement was to "improve playing strength". This is admittedly a very vague goal, but is essentially an extension of what this project sets out to achieve. Once all other requirements have been satisfied, we may devote attention to improving the engine's performance - making more precise moves and winning more games. Classes will be added / modified as necessary to achieve this aim.

Quiescence Search

There is a serious flaw with our algorithm as it stands, due to a phenomenon known as the horizon effect.

To keep the example simple, say it is our move and we conduct a search with a depth of four plies. We're in a bad position, and most variations cause us to lose our queen on ply four. However, the computer finds a move that, by means of sacrificing a bishop, delays the queen trap until ply eight. This ply is beyond the so-called horizon of our search, and thus the computer is unaware of it; as far as it is concerned, it only lost a bishop, nowhere near as bad as the alternative of losing its queen. The move is played and we end up losing a bishop and a queen. The computer may even continue sacrificing material in order to delay the inevitable queen loss, throwing away even more pieces for no good reason.

Needless to say, this is a stark example, but the idea crops up in practice (albeit in a more subtle manner) extremely often, such as sacrificing a pawn to save a knight that turns out to be doomed regardless, or a queen capturing an opposing pawn that an extra ply reveals to be supported by a second pawn. We therefore need a way to combat the horizon effect and minimise the frequency of this type of scenario.

The simple answer, of course, is merely to search deeper into the game tree. In an ideal world, we could look at every single move available and follow it all the way to the end of the game. No horizon results in no horizon effects. In reality, however, even with our optimised search algorithm this would be prohibitively expensive to compute. We cannot avoid the horizon without waiting literally billions of years per move.

Quiescence search is a reasonable compromise between these two extremes. Once we reach a leaf node in the search, instead of immediately returning an evaluation we conduct a further selective search with the aim of reducing the position to one that is quiescent (quiet). Once this has been achieved, we return the evaluation as normal.

The precise definition of what construes a quiescent position varies from engine to engine, but most will at least consider the possibility of a capture or pawn promotion as indicative of a non-quiescent position. The point is, if these types of moves are possible, the evaluation risks being inaccurate - a static snapshot of a dynamic position. In a static position with no immediate threats, the evaluation is much more likely to be accurate.

Conceptually, quiescence search is very similar to the main search; indeed, we may use the negamax alpha-beta framework with quiescence search too. However, there are some tweaks:

> Instead of generating and examining every legal move, we use a special move generator that returns only captures and pawn promotions. If no such move is available, we consider the position quiescent and return a static evaluation score. Note that when we are in check, we should avoid using this generator in order to avoid false checkmate scores.
> We allow for the possibility of "standing-pat" - not moving. This allows the engine to avoid being forced into unfavourable moves, such as QxP with the pawn defended. Essentially, if no capturing/promoting move is capable of increasing the score above the static evaluation, the position is again considered quiescent and the evaluation returned.
> No depth limit is placed on the search - it continues for as long as the position is not quiet.
>
> This last condition may seem to risk search explosion; however, as we are examining only captures and promotions, the depth of any one branch has an inherent limit and the typical branching factor of the tree by this point is very close to 1. If we later expand the quiescence search, say by including moves that give check, we may consider placing a depth limit as a safeguard, but this then reintroduces a horizon effect (albeit a slight one).

Piece-Square Tables

Piece-square tables encode the concept of pieces being particularly effective on certain squares and ineffective on others. For example, it is rarely wise to place a knight on the edge of the board, as this limits its mobility. A knight on d5 is much more effectively placed, easily able to access almost anywhere on the board.

Many heuristics may be expressed using these piece-square tables. Figure 7 shows an example of such a table for White pawns:

r \ f	a	b	c	d	e	f	g	h
8	0	0	0	0	0	0	0	0
7	5	10	15	20	20	15	10	5
6	4	8	12	16	16	12	8	4
5	3	6	9	12	12	9	6	3
4	2	4	6	8	8	6	4	2
3	1	2	3	-10	-10	3	2	1
2	0	0	0	-50	-50	0	0	0
1	0	0	0	0	0	0	0	0

Fig. 7: Sample piece-square table for White pawns.

According to the table, a white pawn on c5 would be worth 109 centipawns rather than the default 100. It covers the following heuristics:

Pawns close to promotion are more dangerous than pawns far from promotion
Pawns should prefer capturing towards the centre
A pawn on d2 or e2 blocks our position; move it as soon as possible.

Similar tables are produced for each other piece. Tables for Black are generally mirrored versions of the tables for White; however this is not a rule and we may well wish the engine to play differently as Black. Zephyr will use colour-symmetrical piece-square tables.

Further tables may be introduced for endgame play. For example, the table for White pawns might change into the following:

r \ f	a	b	c	d	e	f	g	h
8	0	0	0	0	0	0	0	0
7	100	100	100	100	100	100	100	100
6	60	60	60	60	60	60	60	60
5	30	30	30	30	30	30	30	30
4	20	20	20	20	20	20	20	20
3	10	10	10	10	10	10	10	10
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0

Fig. 8: Sample endgame piece-square table for White pawns.

The table in Figure 8 reflects the priorities of a white pawn in the endgame far better than the first table. It is no longer so important to capture towards the centre, or avoid blocking development. Instead, we wish to race pawns towards promotion and afford a lot of protection to any far-advanced pawns we might have.

There are downsides to the piece-square table approach, not least of which is the amount of data entry involved when implementing them. With six types of pieces, two colours and two game phases to deal with, we must include twenty-four piece-square tables, a total of 1,536 pieces of data. Even if we write a quick program to spit out Black's tables given those for White, we're still manually entering twelve tables with a total of 768 fields. This carries a very high potential for typos to find their way into the tables, so thorough bug-checking must be applied. Another potential issue is speed; array lookups are not especially cheap operations in Java, and we will need to carry one out for every piece on the board, every single time we call `evaluate`.

That said, the flexibility and power afforded by piece-square tables is hard to pass up. We will store them in a new class called `PieceVal`, which holds constants that are used by `Evaluation`.

Transposition Tables

A transposition table, as the name implies, deals with transpositions in the search tree. The idea is simple enough - when we finish searching a position and determine the best move and associated score, we store it in a hash table. If we then encounter the exact same position again while searching some other branch of the tree, we can simply return the move and score immediately.

The advantages of this should be obvious enough; if we do not have to search the same position over and over, we save ourselves a significant amount of effort. It should be noted that transposition tables are notorious for introducing a host of rather subtle bugs; however the payoff is well worth the effort.

To accommodate the transposition table, we add a new class called `Transposition`. This class will be an implementation of a hash table, together with methods for adding and retrieving entries. The challenge is then to translate chess positions into hash indices.

Zephyr will make use of the common Zobrist hashing scheme, chosen for its simplicity and relative ease of use. At program initialisation, we generate a set of 781 random 64-bit numbers:

One number for each piece at each square (12 * 64 = 768)
One number to show which side is to move
Four numbers to show castling rights
Eight numbers to show the valid en-passant file (if any)

To obtain the Zobrist hash for a given position, we XOR the relevant random numbers together. For example, the starting position hash would be given by (White rook on a1) ^ (White knight on b1) ^ … ^ (Black rook on h8) ^ (White K-side castle) ^ … ^ (Black Q-side castle).

Once we have the initial hash, we can update it much more efficiently than rebuilding from scratch by using the fact that XOR is its own inverse function. If a White pawn on g4 captures a Black bishop on f5, we compute (initial hash) ^ (White pawn on g4) ^ (Black bishop on f5) ^ (White pawn on f5) ^ (side to move). This allows us to hash the various positions we come across as we traverse the game tree fairly efficiently.

There are several uses for the Zobrist representation of a position, including detection of threefold repetition draws, but here we use it as an index into our hash table. We cannot use the entire key as 2^64 entries would occupy far more storage space than exists in the world even at a single byte per entry. Instead, we use only the last `n` bits of the Zobrist hash, where our transposition table has a maximum of `2^n` entries and store the rest of the key in the entry itself.

The number of possible chess positions (estimated at around 10^47) far exceeds the available space in our table, and so by the pigeonhole principle there exists a slot which two or more completely different positions all hash onto. We must therefore determine how to cope with collisions, both when saving new entries and retrieving old ones.

Several replacement schemes exist when looking to save a new entry. The simplest is to always overwrite any existing entry with the new one, and this is the approach Zephyr will use for the time being. The rationale is that the more recent entry is more likely to be reused, and in practice the scheme works relatively well. Other schemes include favouring entries based on deep searches or entries that required searching more positions; a detailed analysis of these schemes has been conducted by Breuker et al. (1994)[12].

When retrieving old entries, we first compare the stored key fragment to ensure it matches that of the position we are interested in searching. If the fragments do not match, we have a collision; the stored position is different, so we cannot use it and thus we conduct the search as normal. If the fragments do match, we assume that the entry corresponds to the position we are searching. This is not a theoretically sound assumption; it is perfectly possible for two different positions to share a 64-bit Zobrist hash. However, the chances of this type of collision are remote, and the practical effect of this perceived flaw on the accuracy of a search are not noticeably detrimental.

Once we have retrieved an entry, we examine the depth of the search that the entry was based on. This is necessary because if we wish to search the position to a depth of six ply, a score based on a four-ply search is no good! We proceed as follows: if the entry depth is equal to or greater than the desired search depth, we return the stored score immediately. Otherwise, we conduct a search as normal, but mark the move listed in the entry as a "hash move" - it is still likely to be a strong move, so we should search it first in order to produce more cutoffs. This technique will be discussed in more detail in the next section, which concerns move ordering.

12 Breuker, D.M. et al. 1994. Replacement Schemes for Transposition Tables. ICCA Journal, Volume 17: Number 4, pp. 183-193.

Move Ordering

Alpha-beta pruning is most effective on strongly-ordered trees, where the best move is searched first, the second-best move is searched second, and so on up to the worst move which is searched last. Conversely, on a tree in the exact opposite order, alpha-beta is no better than naive negamax. We should therefore endeavour to search the best move first.

Of course, we do not yet know the best move; if we did, we would not need to search the position. However, we can make an educated guess. Moves like promotions and captures are far more likely to be good moves than putting a rook en-prise, or advancing the pawns protecting the king. A basic move ordering scheme might look as follows:

Hash move (see previous section on Transposition Tables)
PV move associated with this ply, where legal (see section 3.3.3)
Captures (including capturing promotions)
Quiet promotions
Other moves

This is a very simplistic scheme, but even this will be a vast improvement over none at all. It covers a large fraction of the positions that we will encounter when traversing the tree; if, for example, the opponent is trying out a move that places a piece en-prise, then frequently the best move is to capture the piece, gaining material for nothing. It should be noted that the hash move and PV move will sometimes coincide, so care should be taken not to search the same move twice.

We may improve the move ordering further by breaking down the captures category. Some captures are better than others; a pawn capturing a queen is almost always an excellent move, while a rook capturing a pawn is dubious as the pawn may be defended. We should therefore search the queen capture before the pawn capture.

Zephyr will use the common MVV/LVA scheme for ordering captures, where MVV/LVA stands for Most Valuable Victim / Least Valuable Attacker. We make the semi-arbitrary decision that the victim's value is a more important consideration, ordering queen captures first, then rook captures and so on down to pawns. Within the set of queen captures, we search pawn moves first, then knight moves, and so on up to queens.

Another method of improving move ordering, this time for quiet moves, is to reuse the piece-square tables. If a piece is on a square valued at -10 and can move to a square valued at +30, this is probably a reasonable move (net gain +40) and so should be searched before moves with a lesser gain.

Needless to say, our move ordering will not always be correct. Capturing the opponent's queen with a pawn will, on occasion, lead to being checkmated. However, the important point is that in the vast majority of positions where such a capture is possible, it is a vastly superior move to anything else and searching it early saves us a great deal of effort. A good target to aim for is achieving a cutoff with the first move on around 90% of all positions searched.

We define a new class `MoveOrdering` to hold public static methods that sort and return a passed `ArrayList<Integer>`. Instead of performing a (relatively) time-consuming sort on the elements of the `ArrayList`, we iterate through and assign each move a weight based on its characteristics, making use of the spare bits in the move encoding. This method of "sorting" the moves has the advantage of reducing the number of expensive `ArrayList` operations necessary, together with a computational complexity of O(n).

Other Methods

The techniques discussed above barely scratch the surface of computer chess. While the engine designed here will be capable of winning games, there remain plenty of methods to improve its strength further. Unfortunately, due to a lack of space in this report, these cannot be expanded upon to any great detail.

Other techniques include (but certainly are not limited to):

Lazy Evaluation

Extending the concept of alpha-beta pruning to evaluation, we may terminate the evaluation early (thus approximating the position's true value) if the score exceeds beta by a given margin (or falls below alpha by the same margin). This saves time evaluating positions that do not require a detailed analysis, but may lead to inaccuracies if the margin used is too small.

Null-Move Pruning

Allow the opponent to make two consecutive moves and search the resulting position to a fairly shallow depth. If the score is still above beta, our position must be very good, so we immediately fail high.

NMP is carried out after the transposition table probe and before move generation, so getting a cutoff here saves us a lot of time.

The heuristic assumes the so-called "null move observation" - there exists some move in the position better than the null move. This is not true when the player to move is in zugzwang, so we should disable NMP when the risk of zugzwang is high. Most programs disable NMP in the endgame, though some opt for the risky (but potentially most potent) method of allowing NMP whenever any piece other than king or pawn remains on the board.

Check Extensions

When the move we are searching gives check, extend the search by one ply. That is, instead of searching the resulting position to depth `n-1`, search it to depth `n`.

The idea behind the check extension is to encourage the program to examine forcing variations. Deep checkmates are generally found faster, and the horizon effect is reduced - the search can no longer push a piece loss over the horizon by means of a series of checks.

Since any extension causes the program to search more nodes, this will necessarily slow the search down a little. However, it is expected that the extra tactical capabilities afforded by the extension will more than compensate for this.

Late Move Reduction

Intended for use in conjunction with move ordering. Since strong moves are searched first, it stands to reason that later moves in the list are not as good. We therefore reduce the search by one ply when searching later moves, reducing the amount of effort spent on moves we expect to be bad.

The starting point of LMR varies between programs, but generally the first four or five moves are searched fully and any moves after this are reduced.

LMR will be counterproductive in engines with poor move ordering, ruining the tactical capabilities of the search.

History Heuristic

If a move from square X to square Y causes a beta-cutoff, increase a history table entry at index [sideToMove][X][Y] by some amount (this amount is typically related to the distance from the quiescence search). When ordering moves, search moves with a higher history score before those with a lower history score.

More effective with lower search depths; at very high search depths, the heuristic produces an essentially random noise.

Aspiration search

By default, we start searching a position with alpha and beta set to negative infinity and infinity respectively (a so-called "infinite window"). Using an aspiration search, we make a guess as to the score that will be returned, say `x`, and set alpha and beta to `x - d` and `x + d` respectively, where `d` is some integer. The guessed value of `x` is typically the score returned by the previous iteration of the search. In order to provide an estimate to start with, the first few iterations are typically done with an infinite window.

If the true score is indeed within the narrowed search window, we will have saved time due to the additional cutoffs obtained. If our guess was wrong, however, the score returned will be useless, and we must waste time searching the position again with an infinite window to obtain the true score.

The value of `d` should be set as small as possible while avoiding too many researches. The optimal value will vary between programs, but typically a value of 50 centipawns is used.

Pondering

While the opponent is looking for a move, instead of sitting idle, guess their most likely move and start analysing the resulting position. If we guess right, we have a head start searching; if we guess wrong, we have wasted no time and will have at least somewhat populated the transposition tables.

More useful against humans, and engines running on separate machines. On the same machine, each engine will disrupt the other with their pondering efforts, resulting in a lot of noise.

Futility Pruning

At a frontier node (one ply away from quiescence), if the static evaluation plus some margin is less than alpha, proceed straight to quiescence or even return an evaluation immediately.

Works best in combination with lazy evaluation.

Lower margins give a faster - but less accurate - search. If the margin is larger than the maximum possible score improvement, futility pruning is completely theoretically sound but does not prune anywhere near so many nodes.

Razoring

Similar in concept to futility pruning. Precise definitions of the technique vary, but typically razoring is applied two or three plies away from quiescence, and if the static evaluation plus some (relatively large) margin is less than alpha, we reduce the search depth by one.

Internal Iterative Deepening

If we do not have a hash move to search first, perform a shallow search (say two ply less). The best move returned from this search is then searched first. Note that if we do not have a hash move at depth eight, we also will not have a hash move at depth six, four, two and finally zero, where the quiescence search will provide us with a move to start off with - hence "internal iterative deepening".

The extra searches have a negligible cost compared to the time saved by improving the move ordering.

Static Exchange Evaluation

Provides a means of estimating whether a given capture is likely to pay off - PxQ is almost always a winning move, while RxN is only good if the knight is not defended. SEE is only an estimation; we do not actually execute the moves on the board and only consider raw material score.

Useful in several areas, including move ordering and a criteria for reductions, but comes into its own in quiescence search. If we search only those moves with SEE >= 0, we save ourselves having to analyse captures that are almost certainly poor.

Notoriously difficult to implement correctly, with many cases to handle that are not immediately apparent.

This is still by no means an exhaustive list. Sixty years of dedicated computer chess research has - if nothing else - been remarkably productive.

Final System

We stated earlier (section 3.1.2) that the system design would by necessity expand and evolve as development progressed. Zephyr went through many periods of redesign and refactoring in order to keep the code reasonably structured, concise and fast. The class diagram of Zephyr 0.7.2, the latest version as of the time of writing, is shown in Figure 7.

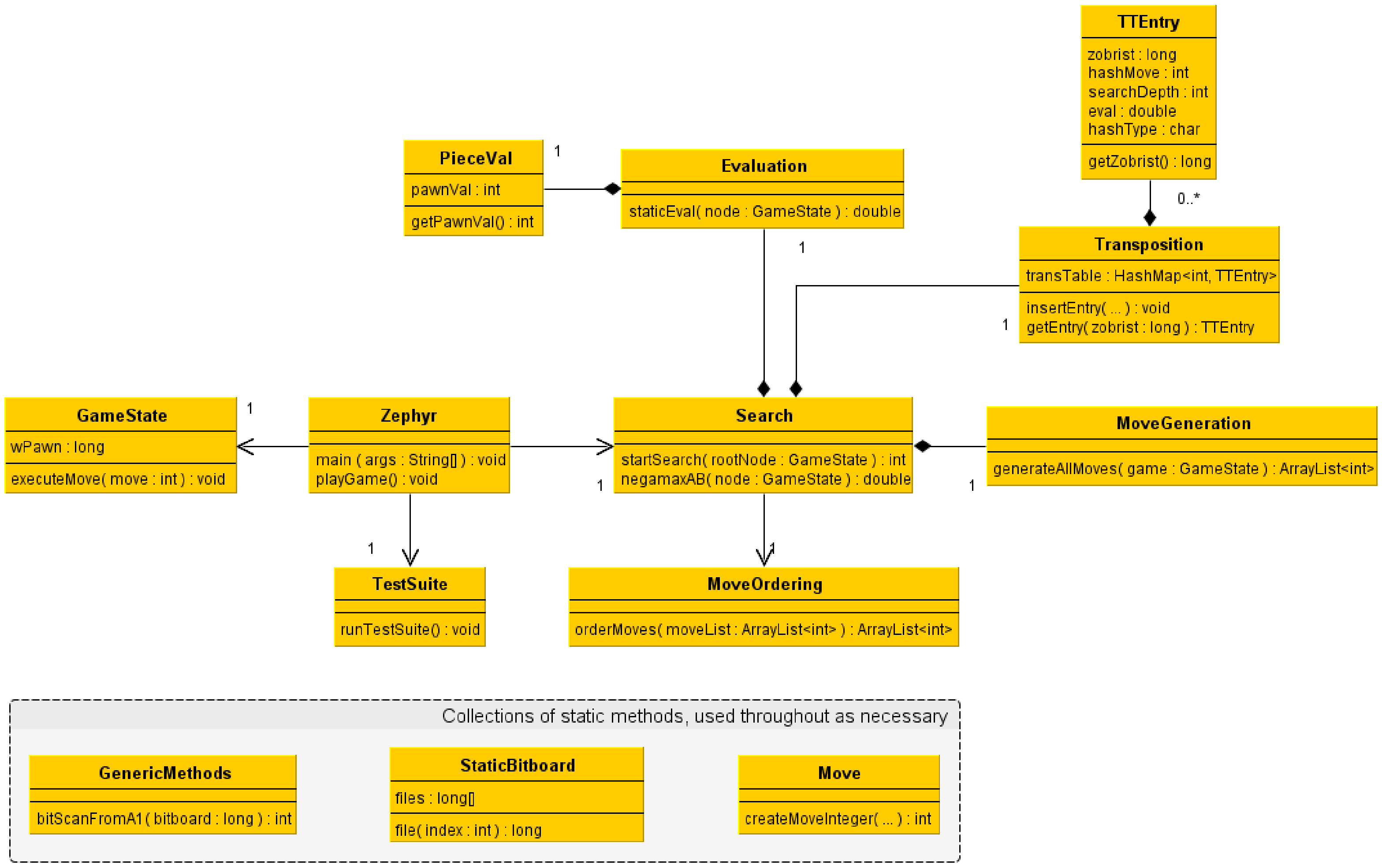

TTEntry
zobrist : long
hashMove : int
searchDepth : int
eval : double
hashType : char
getZobrist() : long
PieceVal
pawnVal : int
getPawnVal() : int
Evaluation
staticEval(node : GameState) : double
Transposition
transTable : HashMap<int, TTEntry>
insertEntry(...) : void
getEntry(zobrist : long) : TTEntry
GameState
wPawn : long
executeMove(move : int) : void
Zephyr
main (args : String[]) : void
playGame() : void
Search
startSearch(rootNode : GameState) : int
negamaxAB(node : GameState) : double
MoveGeneration
generateAllMoves(game : GameState) : ArrayList<int>
TestSuite
runTestSuite() : void
MoveOrdering
orderMoves(moveList : ArrayList<int>) : ArrayList<int>
Collections of static methods, used throughout as necessary
GenericMethods
bitScanFromA1 (bitboard : long) : int
StaticBitboard
files : long[]
file(index : int) : long
Move
createMoveInteger(...) : int
1
0..*

Fig. 7: Class diagram of Zephyr 0.7.2. To save space, only a small subset of attributes/methods are shown for each class.

This is quite a bit more complex than the original design, but the original structure is still more or less intact, the one change being the conversion of move encoding. Many of the new classes, such as `StaticBitboard`, contain static constants and methods used throughout the engine. The three classes listed at the bottom of the diagram are used directly; for example, the bitscan method necessary to interpret bitboards as lists of squares is called with the code `GenericMethods.bitScanFromA1(bitboard)`. The system is of course open to further expansion, if it is deemed necessary for future versions.

Section 4: Implementation

This section will discuss some of the more important aspects of the implementation of Zephyr. The intention is not to provide a complete overview - there is by no means sufficient space for this - but to highlight some of the potential pitfalls in chess programming and how they may be avoided. A full listing of the source code is available alongside this report.

4.1: Bitboards

The bitboard data structure forms the backbone of the chess engine, and often results in potentially obscure code. We therefore discuss bitboards in more detail in this section.

```
    public class GameState
    {
         ...

         // bitboards for each piece type and colour
         private long wPawn; private long wRook; private long
wKnight;
         private long wBishop; private long wQueen; private
long wKing;
         private long bPawn; private long bRook; private long
bKnight;
         private long bBishop; private long bQueen; private
long bKing;

         ...
    }
```

As described in Section 3.2.1, the location of each piece is recorded using a 64-bit data type - Java's `long`. We need use only one `long` for each piece type, since pieces of the same type are essentially indistinguishable from one another; hence the data structure described above.

Next, we wish to be able to execute a move. Fully executing a chess move requires a lot of work, to the extent that Zephyr's `executeMove` method is around 400 lines long. Due to this length, we will examine only the parts concerned with updating the relevant bitboards when a white queen captures a black knight. This allows us to skip over the special cases - castling, en-passant captures and pawn promotion.

```
      public void executeMove(int thisMove)
      {
            ...

            switch(movingPiece)
            {
                  ...

                  case 'Q':
                        wQueen = moveBitboard(wQueen, fromSq, toSq);
                        ...
                        break;
                  ...
            }

            switch(capturedPiece)
            {
                  ...

                  case 'n':
                        bKnight ^= StaticBitboard.singleSquare(toSq);
                        ...
                        break;
                  ...
            }

            ...
      }

      ...

      private long moveBitboard(long bitboard, int fromSq, int
toSq)
      {
            long fromTo = StaticBitboard.singleSquare(fromSq) |
StaticBitboard.singleSquare(toSq);
            return bitboard ^ fromTo;
      }
```

The above code requires some explanation. First, `StaticBitboard.singleSquare(int thisSq)` is an accessor method to an array of bitboards that have only one bit set - the bit corresponding to `thisSq`. In the `moveBitboard` method, `bitboard` always has the `fromSq` bit set and the `toSq` bit empty, and so the method has the effect of moving a set bit from `fromSq` to `toSq`.

```
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 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 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
1 1 1 1 1 1 1 1    0 0 0 0 1 0 0 0    1 1 1 1 0 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
```

Fig. 7: Moving a pawn from e2 to e4 in the initial position. The first diagram is the `wPawn` bitboard. The second is `fromTo` in the `moveBitboard` method. The third is `wPawn` after the move, obtained by XORing the first two together.

Capturing a piece is similarly achieved, XORing the appropriate piece bitboard with the bitboard corresponding to the destination square. This clears the set bit, removing the piece from the game.

Next, we must deal with move generation. That is, given a set of bitboards, we must generate the set of legal chess moves, which are encoded as 32-bit integers. With the exception of the queen (which moves like a combined rook and bishop), every piece requires its own generation, and so the code again takes up a lot of space. Here we shall examine the rook move generation algorithm; the so-called "sliding pieces" are not so easy to implement correctly.

```
     public static long rookMovesBB(int sqID, long
occupiedSquares, long targetSquares)
     {
          long northSquares = StaticBitboard.northSquares(sqID);
          long northMoves = northSquares & occupiedSquares;
          // fill all squares to the north, and clip off
overflowing bits
          northMoves = (northMoves >>> 8) | (northMoves >>> 16)
| (northMoves >>> 24) | (northMoves >>> 32) | (northMoves >>>
40) | (northMoves >>> 48);
          northMoves &= northSquares;

          // get the squares we can move to, removing the last
square if a friendly piece stands there
          northMoves ^= northSquares;
          northMoves &= targetSquares;

          ... (similar for east, south and west)

          return northMoves | eastMoves | southMoves | westMoves;
     }
```

The algorithm presented above is actually a simplified version that deals with only one rook at a time; a few other tricks can be used to generate moves for all rooks simultaneously. There exist myriad ways to generate moves with bitboards, and there seems to be a strong negative correlation between speed and clarity in this area! The algorithm chosen aims to strike a balance between the two extremes.

That said, it is difficult to claim that the above code is even remotely intuitive. Let us examine the following sample position and consider how the north moves for the rook on c1 are generated:

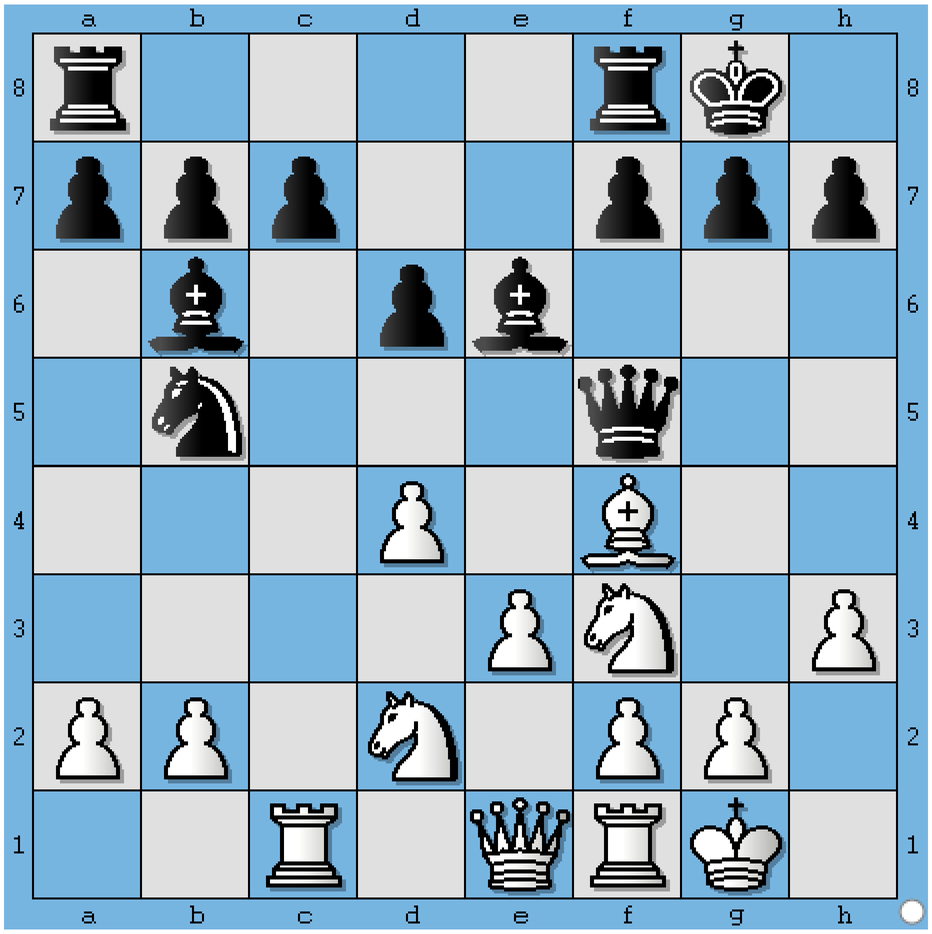


Fig. 8: A sample middlegame position. Screenshot taken from within the Arena GUI.

First, we should clarify what each of the method parameters represent. `sqID` is the square holding the rook we are interested in; here its value is `2`, which corresponds to the c1 square. `occupiedSquares` is a bitboard representing the location of every piece regardless of colour or type, which may be obtained by ORing every piece bitboard together. Finally, `targetSquares` is a bitboard representing the set of potentially valid destination squares and is here equal to `~whitePieces` (or equivalently `emptySquares | blackPieces`). This is done to avoid the capture of a friendly piece.

```
1 0 0 0 0 1 1 0
1 1 1 0 0 1 1 1
0 1 0 1 1 0 0 0
0 1 0 0 0 1 0 0
0 0 0 1 0 1 0 0
0 0 0 0 1 1 0 1
1 1 0 1 0 1 1 0
0 0 1 0 1 1 1 0
```

Fig. 9: `occupiedSquares` for the position in Fig. 2.

```
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
1 1 1 0 1 0 1 1
1 1 1 1 0 0 1 0
0 0 1 0 1 0 0 1
1 1 0 1 0 0 0 1
```

Fig. 10: `targetSquares` for the position in Fig. 2 with White to move.

Let us now step through the algorithm.

```
long northSquares = StaticBitboard.northSquares(sqID);
long northMoves = northSquares & occupiedSquares;
```

The `StaticBitboard.northSquares` method returns a bitboard with bits "north" of `sqID` set. That is, squares `x` satisfying `file(sqID) == file(x) && rank(sqID) < rank(x)` have a set bit. We then take the intersection of this set with the set of occupied squares.

```
0 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 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

Fig. 11: `northMoves` for the position in Fig. 2 after the first intersection.

We now have a bitboard representing the "blockers" - pieces standing in the way of otherwise valid squares - on this file. We next "flood" the file north to set all bits behind the closest blocker.

```
     northMoves = (northMoves >>> 8) | (northMoves >>> 16) |
(northMoves >>> 24) | (northMoves >>> 32) | (northMoves >>> 40)
| (northMoves >>> 48);
     northMoves &= northSquares;

     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 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
```

Fig. 12: `northMoves` for the position in Fig. 2 after the flood step.

The intersection here is not actually necessary for north/south moves, as the extra bits simply "drop off" and disappear. However, it is necessary for west/east moves to prevent wrapping around the edge of the board.

```
     northMoves ^= northSquares;
     northMoves &= targetSquares;

     0 0 0 0 0 0 0 0
     0 0 1 0 0 0 0 0
     0 0 1 0 0 0 0 0
     0 0 1 0 0 0 0 0
     0 0 1 0 0 0 0 0
     0 0 1 0 0 0 0 0
     0 0 1 0 0 0 0 0
     0 0 0 0 0 0 0 0
```

Fig. 13: `northMoves` for the position in Fig. 2 after the final intersection.

XORing `northMoves` and `northSquares` results in a bitboard with bits set corresponding to the north squares the rook may move to, including the blocker. The final intersection removes the blocker if it was a friendly piece; otherwise, the operation has no effect.

This technique is repeated for east, south and west moves, where we change the bit-shift used in the flood step to `>>> 1`, `<< 8` and `<< 1` respectively. Finally, we combine the four bitboards together to produce a destination bitboard representing all possible rook moves from `sqID`.

This section is not a complete documentation of Zephyr's move generation. After obtaining the destination bitboard, we serialise it to convert it into a list of actual moves, and finally we must ensure the move does not leave us in check. However, the details of these steps are not of such great interest to us here, so we will move on to the next topic.

4.2: Transposition Tables

Transposition tables are conceptually quite simple. However, it is extremely easy to make a mistake when implementing the necessary infrastructure. Incorrect transposition table code can lead to subtle bugs that are difficult to track down. Thorough testing is crucial.

The easy part is setting up the transposition table itself. Zephyr implements the table using Java's `HashMap`, mapping from integers to instances of `TTEntry`. To compute the index for a given position, we can use the simple hash function `zobrist % TT_SIZE`, where our transposition table has `TT_SIZE` possible entries. However, the modulus operation is (relatively) expensive, so Zephyr uses an optimised version of this hash function.

```
     private static int SIZE_EXPONENT = 24;
     private static long FETCH_MASK = (1 << SIZE_EXPONENT) - 1;
     private static int TT_SIZE = (int)Math.pow(2,
SIZE_EXPONENT);

     public int getTTIndex(long zobrist)
     { return (int)(zobrist & FETCH_MASK); }
```

By requiring the transposition table size to be a power of two, we may use this modified hash function - a bitwise AND with `FETCH_MASK`, a string of `SIZE_EXPONENT` 1s. When `TT_SIZE == 2^SIZE_EXPONENT`, this function is identical to `zobrist % TT_SIZE`, isolating the lower `SIZE_EXPONENT` bits of `zobrist`.

In order to keep the implementation simple to begin with, Zephyr uses the Always Replace replacement scheme when inserting new entries. More elaborate schemes are something to be considered for future work; for now, we merely require something functional. Under this scheme, inserting an entry is extremely simple; we compute the index, construct a new instance of `TTEntry` and insert it into the table, overwriting anything that might be there already.

Retrieving an entry is also quite simple. We compute the index as before, and then check if this index has an entry associated with it. If so, we return it; otherwise we return `null` to indicate that no entry exists.

The tough part is putting the new transposition table to use. If we insert an entry that does not make sense, or misinterpret a stored entry, the effectiveness of the search will be reduced, sometimes disastrously so.

We shall tackle entry retrieval first, as it is probably the simpler of the two interactions. The code to perform this task - with some comments stripped to save space - appears below.

```
     TTEntry thisNodeTT = tr.getEntry(zobrist);
     Boolean haveHashMove = false;
     int TTMove = -1;

     if (thisNodeTT != null)
     {
          if (thisNodeTT.getZobrist() != zobrist)
          { hashCollisions++; }
          else
          {
               hashHits++;

               // if the stored move is of sufficient depth, we
can return straight away
               if (thisNodeTT.getSearchDepth() >= depth)
               {
                    hashDeepEnough++;
                    double hashScore = thisNodeTT.getScore();
                    switch (thisNodeTT.getHashType())
                    {
                         case 'A':
                              // let x be the true minimax score
of this node
                              // here, we know that x <= score
                              // if score <= alpha, then x <=
alpha and thus we fail-low
                              hashMoveUppers++;
                              if (hashScore <= alpha){ return
hashScore; }
                              break;

                         case 'B':
                              // similar logic to above, we know
x >= score
                              // if score >= beta, then x >=
beta and we fail-high
                              hashMoveLowers++;
                              if (hashScore >= beta){ return
hashScore; }
                              break;

                         default: // exact minimax score known;
return it
                              hashMoveExacts++;
                              // if the hash move has a mate
score attached, we must adjust it to
                              // correspond to this node's
distance from the root position
                              if (hashScore < -MATE_CUTOFF)
                              { hashScore = MATE_VAL +
pliesFromRoot; }
                              else if (hashScore > MATE_CUTOFF)
                              { hashScore = -MATE_VAL -
pliesFromRoot; }
                              return hashScore;
                    }
               }

               haveHashMove = true;
               TTMove = thisNodeTT.getHashMove();
          }
     }
```

If an entry does not exist (that is, if `thisNodeTT == null`), we move on immediately. Otherwise, we compare the full Zobrist key of the stored position to the key of the position we wish to search. If these do not match, the stored position is not the one we're searching; the lower `SIZE_EXPONENT` bits of the two Zobrist keys just happened to coincide.

When the Zobrist keys match, to the best of our knowledge the positions are identical and so the retrieved information is valid. Our next step is to check whether the stored entry is based on a search of sufficient depth; it is no good using a result from a search of depth 5 when we need to search the current position to depth 8. Assuming it is, we check what type of entry we have (either fail-low, fail-high or exact) and use the score appropriately. Comments have been left intact in this part to explain the logic behind each possibility.

If the stored entry was not of sufficient depth, or we were unable to immediately return a score, the entry has not gone to waste, as the move associated with the entry is still likely to be a good one. It was after all selected as the best move in the position by an earlier search, and the majority of the time throughout the search the best move will be "obviously" good (e.g. capture of a valuable hanging piece), almost certainly the best move no matter how much further we search that position. Our move ordering therefore benefits from searching this move first, which is handled by assigning the variable `TTMove` to be the hash move.

The code itself may seem reasonably simple, but not pictured is the vast array of pitfalls surrounding the issue. For example, when retrieving a stored beta cutoff, we may decide to modify the code to reduce `beta` to `hashScore` if the latter is less than the former:

```
     case 'B':
          hashMoveLowers++;
          if (hashScore >= beta){ return hashScore; }
          else { beta = hashScore; }
          break;
```

This seems reasonable enough; if we previously failed high with a score of `hashScore`, then any score higher than this ought to fail high too, and the narrowed window produces more cutoffs, speeding up the search. However, it simply does not work. The modification practically turns the `negamaxAB` function into an elaborate random number generator, destroying the tactical strength of the engine.

The reason for this is that a move that previously failed high may merely increase alpha or even fail low when some tactical flaw is uncovered by the deeper search. Our modified code assumes the move will continue to fail high, which simply is not true. The overall effect of the modification is to cause certain poor moves to be considered very strong. When expressed this way, the flaw seems obvious. However, recall that this disaster was brought about by adding a single line of intuitively correct code!

All of this holds true for storing values too. The code itself is about as basic as can be; for example, when we have a beta cutoff:

```
      if (score >= beta)
      {
            if (!searchTimedOut){ tr.insertEntry(zobrist, 
nextMove, depth, score, 'B'); }

            ...
      }
```

…where `if (!searchTimedOut)` is merely a safeguard against storing invalid values at the end of a search (when out of time, any calls to `negamaxAB` immediately return 0). We store the Zobrist key of the position, what the move causing the cutoff was, etc. and note that this entry was due to a beta cutoff.

The difficulty comes from ensuring that every node searched has its result stored correctly. What happens if no move ever increases alpha, and so no move is considered "good"? What happens if the node brings about a threefold repetition in this line of play, when it would not in some different line of play? How about when the player to move has been checkmated or stalemated? All of these things are important to consider, and it is extremely easy to introduce bugs by handling the situation incorrectly. Zephyr currently uses a rather conservative approach, refusing to store threefold repetition scores at all and storing an invalid placeholder move upon a fail-low or checkmate/stalemate, in order to lower the risk of odd bugs appearing. However, even with this approach, the implementation of transposition tables took several weeks before the new version reached a state considered functional!

4.3: Move Ordering

The process in which moves are ordered and then selected is a little obscure, so we document it in further detail here. The following code is taken from the `MoveOrdering` class, and is responsible for ordering captures using MVV/LVA.

```
     public static ArrayList<Integer> MVV_LVA(ArrayList<Integer>
moveList)
     {
          for (int i = 0; i < moveList.size(); i++)
          {
               int thisMove = moveList.get(i);

               // order using MVV
               switch (Move.getPieceCaptured(thisMove))
               {
                    case 'q': case 'Q': thisMove =
Move.setWeight(thisMove, 120); break;
                    case 'r': case 'R': thisMove =
Move.setWeight(thisMove, 110); break;
                    case 'b': case 'B': thisMove =
Move.setWeight(thisMove, 100); break;
                    case 'n': case 'N': thisMove =
Move.setWeight(thisMove, 90); break;
                    case 'p': case 'P': thisMove =
Move.setWeight(thisMove, 80); break;
                    default: continue;
               }
               // order sub-divisions using LVA
               switch (Move.getPieceMoved(thisMove))
               {
                    case 'p': case 'P': thisMove =
Move.addWeight(thisMove, 5); break;
                    case 'n': case 'N': thisMove =
Move.addWeight(thisMove, 4); break;
                    case 'b': case 'B': thisMove =
Move.addWeight(thisMove, 3); break;
                    case 'k': case 'K': thisMove =
Move.addWeight(thisMove, 2); break;
                    case 'r': case 'R': thisMove =
Move.addWeight(thisMove, 1); break;
                    // leave Q at default
               }
               moveList.set(i, thisMove);
          }
          return moveList;
     }
```

The code itself is actually rather simple; for each move in the list, we set its weight to a base value depending on the piece being captured (MVV) and add a bonus if the moving piece is weak (LVA). We then plug the move back into the list. After all moves have been processed, we pass back the modified list.

Once all moves have been assigned a weight, we must search them one-by-one, starting with the hash move (if one exists) and then proceeding in descending order of weight. To select the next move to be searched, we perform one step of a slightly modified selection sort (the reason for using selection sort over a more efficient sorting algorithm will be discussed momentarily).

Each move in turn is checked to see if it is the hash move. If so, we remove it from the move list and search it immediately. Otherwise, we proceed as in standard selection sort, comparing its weight to the highest weight found so far. After iterating through the move list, we have found the move out of those remaining with the highest weight, and so we move on to searching it.

The choice of selection sort is due to the use of alpha-beta pruning in combination with the assumption that our move ordering is reasonable. We thus expect that we will achieve a beta-cutoff within the first few moves, saving us from selecting and searching the other moves at all. Experimentation showed the selection sort approach to be faster than sorting all moves ahead of time using quicksort or radix sort.

Section 5: Testing and Evaluation

Testing is of critical importance when developing a chess engine. Without developing a strong testing framework, it is all too easy for bugs to develop. Buggy engines quickly become unmanageable, to the point where it can be easier to simply start over than to attempt fixing the problem. This actually occurred (thankfully early) in the development of Zephyr, and the root cause was the lack of a rigorous testing strategy.

It cannot be stressed enough that any new code, however trivial, must be tested to ensure that it functions as expected, including any special cases. Documentation should always be kept up to date, detailing the preconditions and postconditions. Bugs in chess engines have a tendency to be extremely specific and obscure, and following these ideals will help keep debugging effort to a minimum.

Once it is established that the new code has not reduced the integrity of the system, we must test whether the changes brought about an improvement (or a regression!) to the engine's playing strength. Even seemingly trivial modifications may have a dramatic effect in either direction, so unless there is truly no way for the change to affect performance (say, updating the console interface), testing will be required.

This is a long and error-prone process, with tests often taking days to fully complete. Since chess engines are very resource-intensive and time-sensitive, it is recommended not to use the computer for any other tasks while a test is being carried out in order to not influence the results.

Many of the unit tests carried out on Zephyr are fairly generic, for example ensuring the pre- and post-conditions of each method are satisfied, profiling the engine while it searches a position, and so on. However, there exist many testing strategies that are unique to the domain of computer chess; this section will detail those that the author found valuable throughout Zephyr's development.

Perft

Perft (short for "performance test") is a method of testing the move generation of a chess engine. It is defined in terms of a function perft, where

```
    perft(x) = number of variations of length x from the
current position
```

There are some important points to bear in mind about the perft function. Firstly, it ignores any possible draws by threefold-repetition, insufficient material or fifty-move rule. Secondly, it ignores transpositions; the variations 1. d4 d5 2. e4 e5 and 1. e4 e5 2. d4 d5 are considered distinct variations of length two despite resulting in identical positions. Finally, it ignores variations that stop due to checkmate/stalemate before reaching length `x`.

Pseudocode for a basic perft function is as follows:

```
int perft(int depth)
{
    if (depth == 0){ return 1; }

    int nodes = 0;
    Generate all legal moves
    for (each move)
    {
        Execute the move
        nodes += perft(depth - 1);
        Reverse the move
    }

    return nodes;
}
```

Various optimisations are possible for the above function, but these are concerned only with speed of execution. Perft is notoriously slow; it has the same computational complexity as the standard negamax algorithm, but we cannot use the trick of alpha-beta pruning to avoid traversing sections of the tree. Assuming an average branching factor of 35, we see that if perft(4) completes in, say, 0.1 seconds, perft(5) takes 3.5 seconds, perft(6) takes 122.5 seconds, perft(7) takes over seventy-one minutes and perft(8) takes nearly forty-two hours. Thus care should be taken when using this function not to invoke it with too high a parameter.

The purpose of this function, as mentioned above, is to test the move generation of the engine. We may compare the values returned by the above function to a table of values known to be correct; typically the starting position is used for this purpose, though other standard positions exist to further assist with catching bugs.

Depth	Variations
1	20
2	400
3	8,902
4	197,281
5	4,865,609
6	119,060,324
7	3,195,901,860
8	84,998,978,956
9	2,439,530,234,167
10	69,352,859,712,417

Fig. 14: Perft values for depths up to 10 ply from the initial position[13].

As a rule of thumb, it is worth testing with at least perft(6) from the initial position, if not higher. This is not so deep that the computation is unreasonably large, but still catches a large number of potential bugs, including en-passant captures, early checkmates and check evasion. If the returned value differs from the known value, the move generation must be incorrect. It should be noted that correct values do not necessarily imply that the move generation is correct, but due to the complex nature of the rules of chess, it is highly likely that this is the case.

Apart from the computational complexity, there is another issue with using perft; it is a non-constructive testing strategy. That is, it can tell you that a problem exists, but not what the problem precisely is. Say you run perft(4) and get a result of 197,283. This is slightly higher than the correct value of 197,281, so the move generation is incorrect. However, there is no indication of where the extra two variations are coming from.

A solution to this problem exists in the form of what is commonly called the *divide* function. Divide is essentially a verbose form of perft; for each legal move in the current position, divide executes it and calls perft on the resulting position. Pseudocode for divide is as follows:

```
int divide(int depth)
{
     Generate all legal moves
     for (each move)
     {
          Execute the move
          Output move notation
          Output perft(depth - 1)
          Reverse the move
     }
}
```

13 Bean, R. 2003. Counting Nodes in Chess. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.4388 [Accessed: 17th February 2013]

Again, the resulting values are compared to a source that is known to be correct. Due to the large amount of data that results from such a function, tables with explicit values are not available; instead we compare the values with those generated by another program. Most freely available engines with a command line interface implement this function.

The following is output by Zephyr after issuing the command `divide 4` in the initial position:

```
divide 4
Calculating… (may take a while!)
     h2h3: 8457
     g2g3: 9345
     f2f3: 8457
     e2e3: 13134
     d2d3: 11959
     c2c3: 9272
     b2b3: 9345
     a2a3: 8457
     h2h4: 9329
     g2g4: 9328
     f2f4: 8929
     e2e4: 13160
     d2d4: 12435
     c2c4: 9744
     b2b4: 9332
     a2a4: 9329
     g1h3: 8881
     g1f3: 9748
     b1c3: 9755
     b1a3: 8885
Total moves: 197281
Time taken: 0.2912 seconds
```

The precise order of moves might vary for different programs, but the output will essentially be the same as Zephyr's. Now, suppose for the sake of argument that the value after `g1f3` is 8885 rather than 8881. We now know that there is a problem in a variation following the opening move Nf3. We set up the position following this move and repeat the procedure, this time using `divide 3`. Again, the results are compared to those of a program known to be correct, and we see that (say) the replies `e7e5` and `g7g5` by Black each have values two higher than expected.

We continue in this manner until we have homed in on a position where an incorrect number of moves are being generated. Examining the list of generated moves for the position should help reveal the problem, making the process of debugging the move generator much easier.

Human-Engine Tournaments

Once it has been established that the engine plays legal chess, we may focus on the task of improving its playing strength. When we make modifications with this goal in mind, we must test the new version to ensure that the changes were beneficial.

There are several ways of going about this, but perhaps the most obvious, and the most important, is to play a large number of games and observe its performance in real situations. The question then becomes one of whether to pit the engine against human players or other engines. Each approach has advantages and disadvantages, and ultimately it is best to use a mixture of both. This section will discuss the use of human-based testing.

The strengths and weaknesses of chess engines are well known; they handle chess tactics admirably. If the current position holds a forced mate in four, the engine will find it with anything more than a handful of seconds. This feat does not require any particularly advanced knowledge of the game; as long as it knows the rules and is capable of searching deep enough, the mate will be found and played.

Where engines suffer is their positional knowledge. We can implement a bonus for a passed pawn and a penalty for a knight on the edge of the board, but there will often be gaps in the available knowledge, or worse still, the implemented knowledge is handled incorrectly. The "positional noise" generated by an imperfect evaluation function (and it is difficult to argue a perfect evaluation function even exists) is responsible for a great majority of bad moves.

Humans have the same effect, only in reverse. The same mate in four generally will be much more difficult for a human to find, and may well be missed completely. However, our positional knowledge is far more complete. Humans are much more capable of learning what works and applying this knowledge to future games. Essentially, humans rely on positional heuristics to decide on a good move while not looking ahead particularly far, while computers look a long way ahead to compensate for their inaccurate evaluation.

From the perspective of the programmer, then, a human-engine match may be viewed as a test of the evaluation function. A strong human will give the engine a rigorous positional workout, seeking to inflict long-term disadvantages the ramifications of which are far beyond the search depth of the computer.

Observe the following transcript. This game was played between Zephyr and the then-president of Cardiff University Chess Society, widely regarded to be the strongest regular attendee. It is reproduced here - with permission - to illustrate the type of understanding to be gained from a human-engine chess match.

```
Steven Gardner - Zephyr 0.5.2
Time Control: 20 minutes each for all moves

1. e4 c6
2. d4 d5
3. exd5 Nf6
```

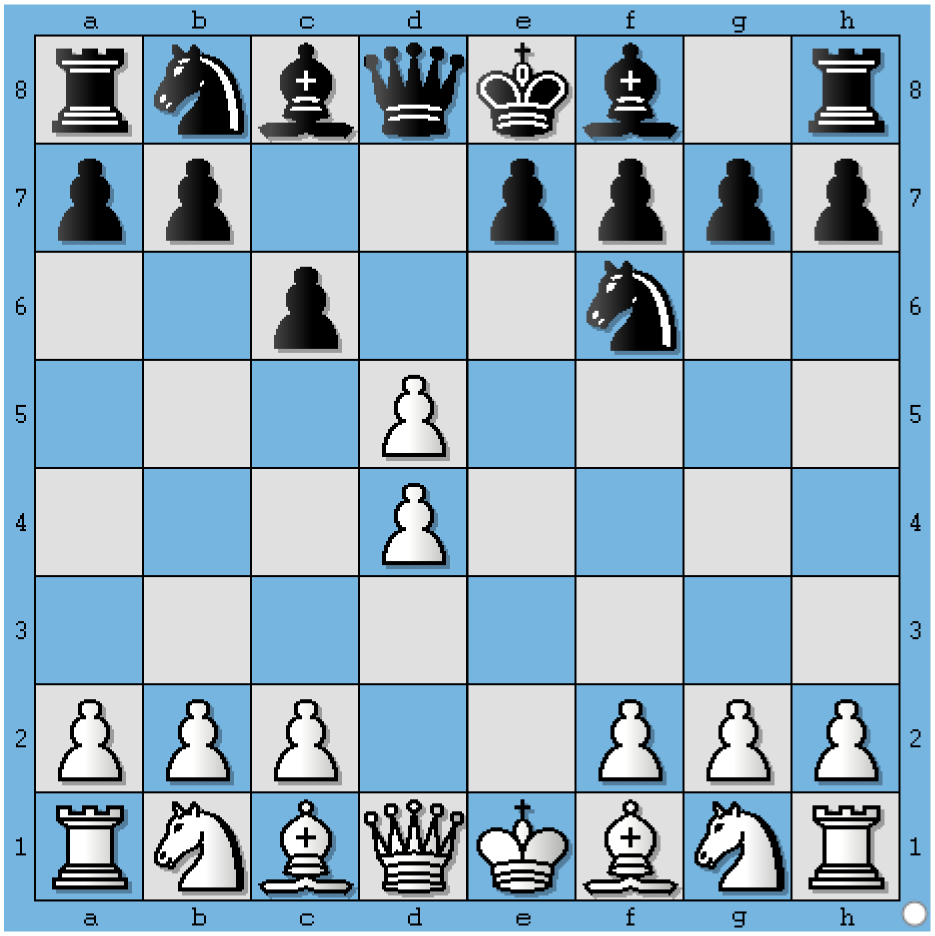


Fig 15: Position after 3. … Nf6.

This was the first move made by Zephyr, rather than the opening book. `3. … Nf6` is not unheard of in this position, but `3. … cxd5` is certainly far more common, and for good reason. Zephyr is now a pawn down, though the lead in development helps to compensate for this; a gambit, in chess parlance. Whether this style of play is to be encouraged or condemned is perhaps a matter of taste, though truly bad lines may be removed by modifying the opening book.

```
4. dxc6 Nxc6
5. Nf3 Bg4
```

Pinning the newly developed knight. Zephyr 0.5.2 does not account for pins in the evaluation, but the search has seen that this creates a threat to win the d4 pawn.

```
6. Be2 Qb6
```

This is, again, perhaps an odd choice of move, to the extent that Steve commented on it during the game. It is not truly bad, but generally in chess it is inadvisable to develop the queen too early, and Zephyr 0.5.2's evaluation does not take this into account. Thus we might consider introducing a penalty for moving the queen before other pieces have been moved. Needless to say, the idea would require thorough testing to ensure it produces an increase in strength, or at least avoids a regression, but this is the idea behind human-engine testing; identifying positional flaws in the engine's play and seeking methods of resolving them.

```
7. c3 O-O-O
8. O-O e5
```

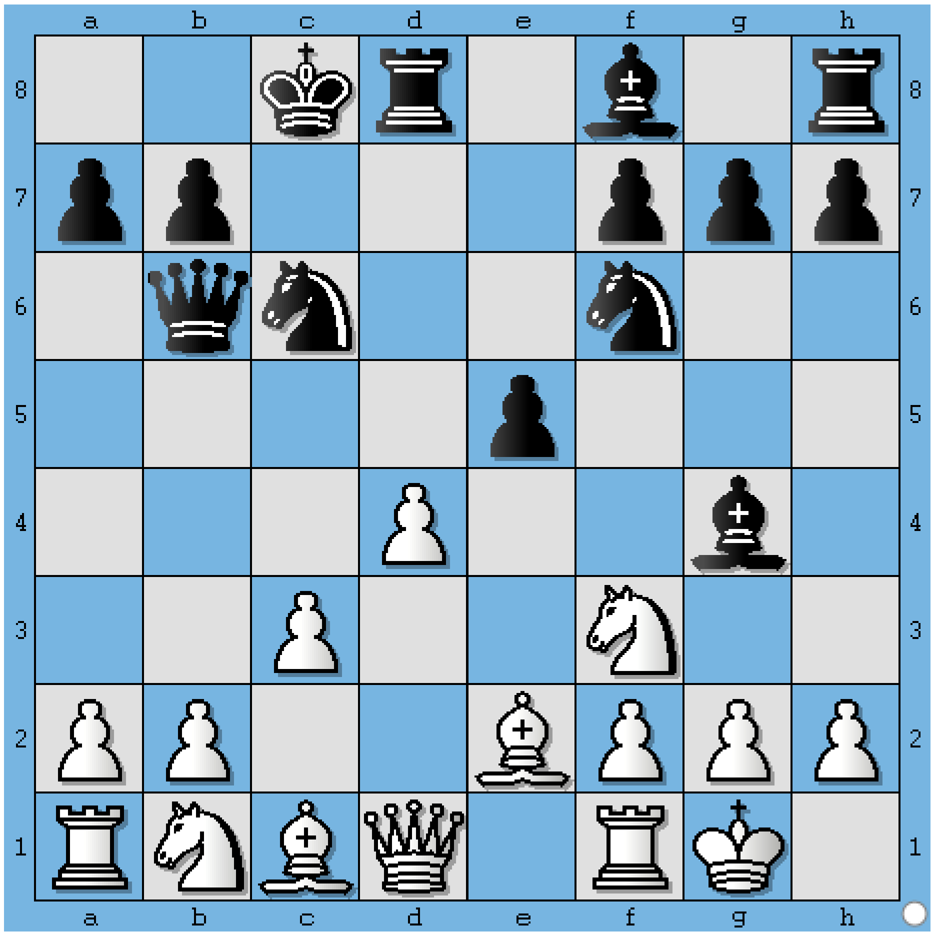


Fig 16: Position after 8. … e5.

"Alright, fair enough!" - Steve, after some time examining the new position. The black e5 pawn cannot be captured due to the pin on the d4 pawn; if it moves, White's queen is lost for a rook and pawn. To the computer, this is mere tactics, an essentially brute force calculation, but it makes for a very appealing move.

```
9. Ng5 Bxe2
10. Qxe2 Qc7
11. dxe5 Nxe5
12. Bf4 Bd6
13. Bxe5 Bxe5
14. Nxf7
```

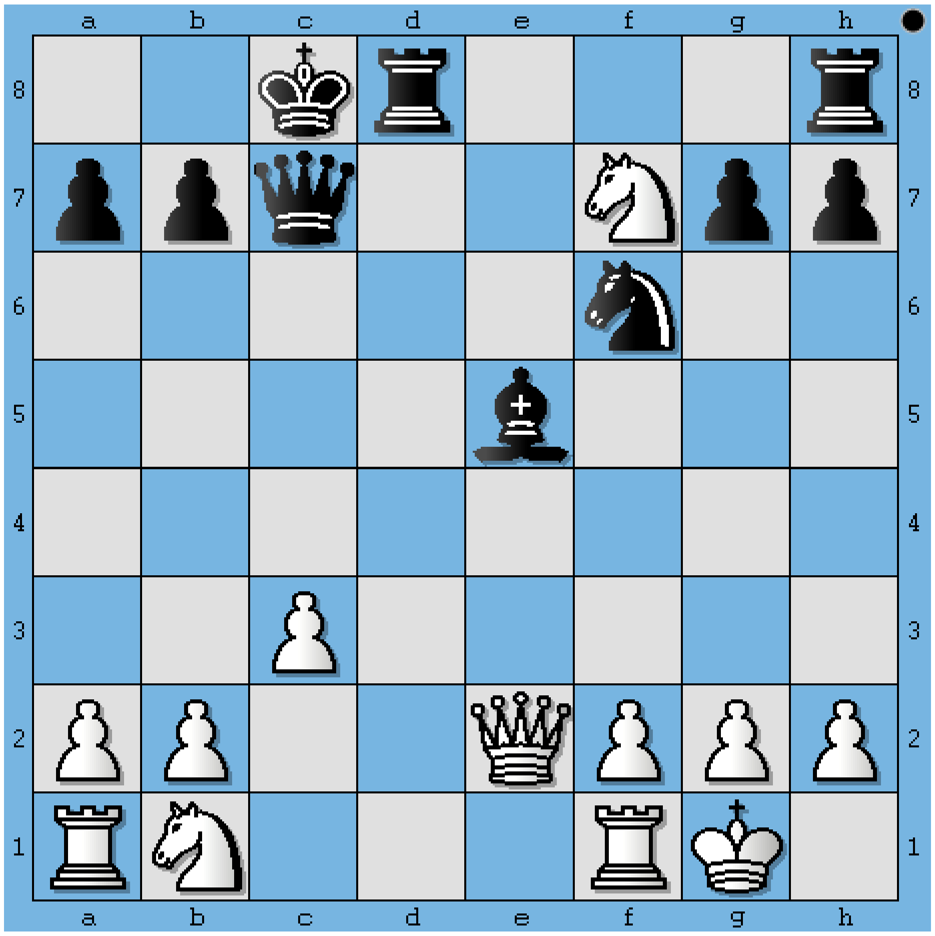


Fig 17: Position after 14. Nxf7.

This move by Steve has a very powerful visual effect, capturing a pawn and forking the two Black rooks. Recapturing with the queen leaves the bishop on e5 *en prise*, so a superficial evaluation of the position suggests White has won a pawn in broad daylight; indeed, this was the reaction of all those observing the game, including the author. However, Zephyr has it all covered.

```
14. ... Bxh2+
15. Kh1 Rhe8
```

Regaining the pawn with check, then threatening the queen. White has no respite to capture the remaining rook.

```
16. Qf3 Rd7
17. Qh3 Bf4
18. g3 Qc6+
19. Kg1 Bxg3
20. Qxg3 Rxf7
```

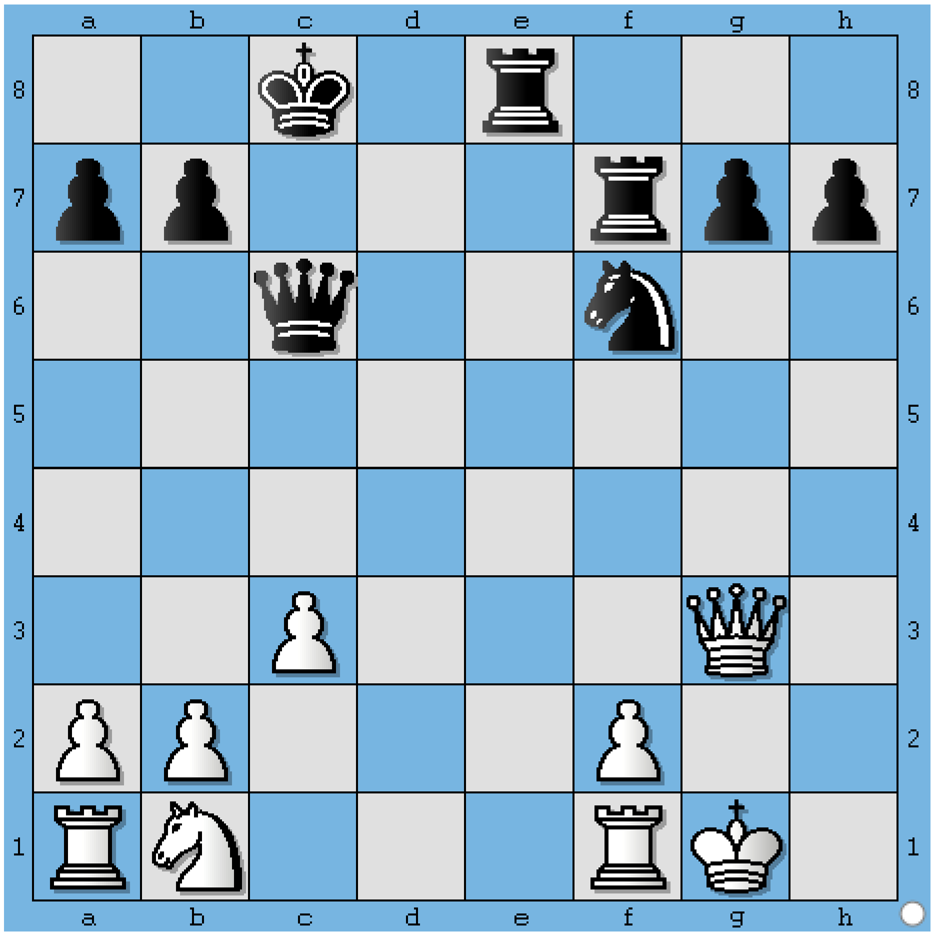


Fig 18: Position after 20. ... Rxf7.

After a shaky opening, Zephyr has pulled the material level again. Its evaluation scores - which had been hovering around the -0.5 area ever since the beginning - are now decidedly positive, a seven-ply search showing scores around +0.8. It can be instructive to examine positions like this one (middlegame with level material) to determine what factors are influencing the evaluation score. Here, the Black rooks on open files and queen pointing dangerously at the opposing king are probably the most significant contributors, together with White's rather passive queenside. If the score is very different to what you might expect, examine why this is the case; more often than not, the root cause is a missing or ineffective evaluation term.

```
21. Na3 Re4
22. f3 Re2
23. Rf2 Qc5
24. Kg2 Re5
```

The score is gradually increasing with every move. By this point, Steve is under some pressure and has settled down to think. Zephyr was proving to be a tougher match than he'd expected.

```
25. f4 Rf5
26. Re1 Qd5+
27. Qf3 Qxa2
```

On the whole, this is probably just another greedy, materialistic computer move. Certainly there are more relevant things on the board than a pawn trapped on a2; this sort of thing might be avoided by (say) encouraging pieces to move towards the enemy king. However, White has no immediate threats, so the move is acceptable enough.

```
28. Re5 Nd5
29. Rxf5 Rxf5
30. Qh3 g6
31. Nb5 Nxf4+
```

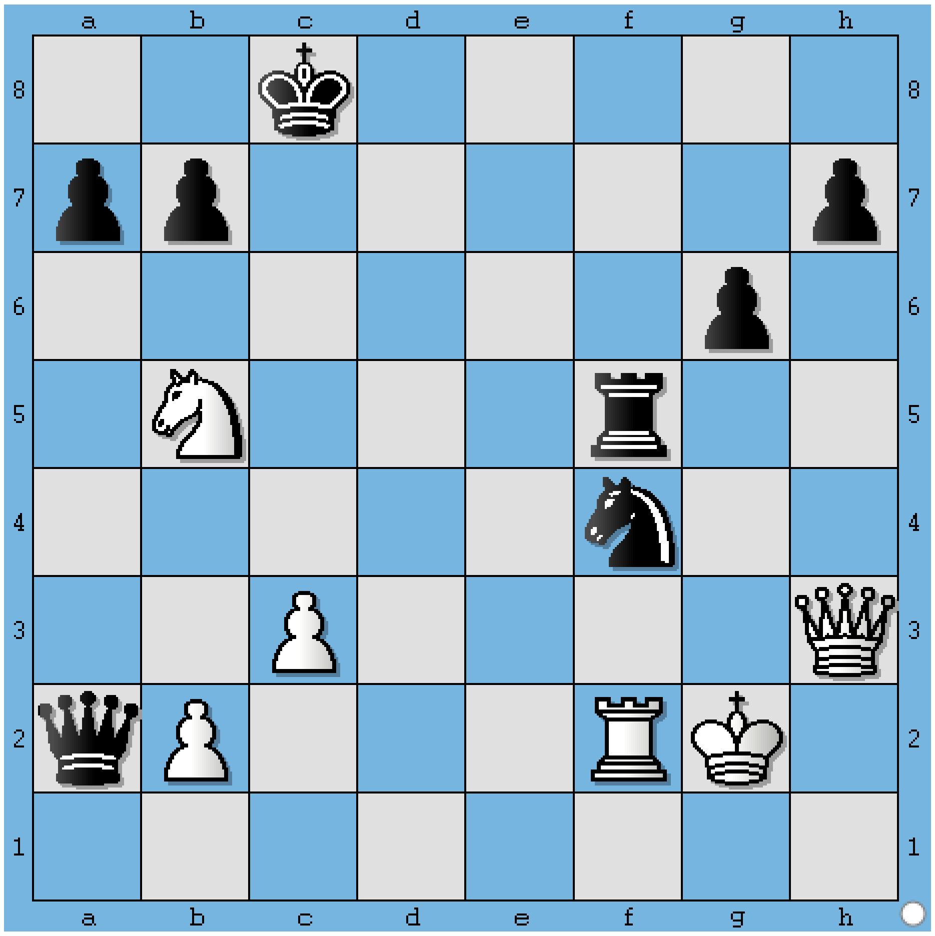


Fig. 19: Position after 31. … Nxf4+.

Deja vu, perhaps. Zephyr captures a pawn with an apparent knight fork just as White did on move 14. The queen is safe, but Zephyr's score still jumped up considerably; it had spotted a way to grab two pawns for nothing.

```
32. Rxf4 Qxb2+
```

When the king moves out of check, Black can simultaneously regain its knight and provide adequate defence to its pinned rook, maintaining a considerable advantage. However, Steve, now under considerable pressure, finds a way to accelerate the loss:

```
33. Rf2 Qxf2+
```

Disastrous. The rules governing check in chess are rather complicated; even grandmasters have made this type of mistake in the past. The Black rook is pinned by the White queen, so it normally cannot move until the pin is released. However, it is still considered to be defending the Black queen; the White king cannot capture it, as it would then be in check.

This kind of misunderstanding can easily lead to errors in a chess engine. Tests must be rigorous to a fault or the engine risks crashing at an inopportune moment. Fortunately in this case, Zephyr happened to handle the rule correctly, but the possibility of this situation occurring had never even crossed the author's mind.

The game was fairly short-lived after the mistake, but for completion's sake it is reproduced here:

```
34. Kh1 Kb8
35. Qxh7
```

Threatening mate, but Zephyr gets in first.

```
35. ... Qf1+
36. Kh2 Rf2+
37. Kg3 Rg2+
38. Kh3 Qh1# 0-1
```

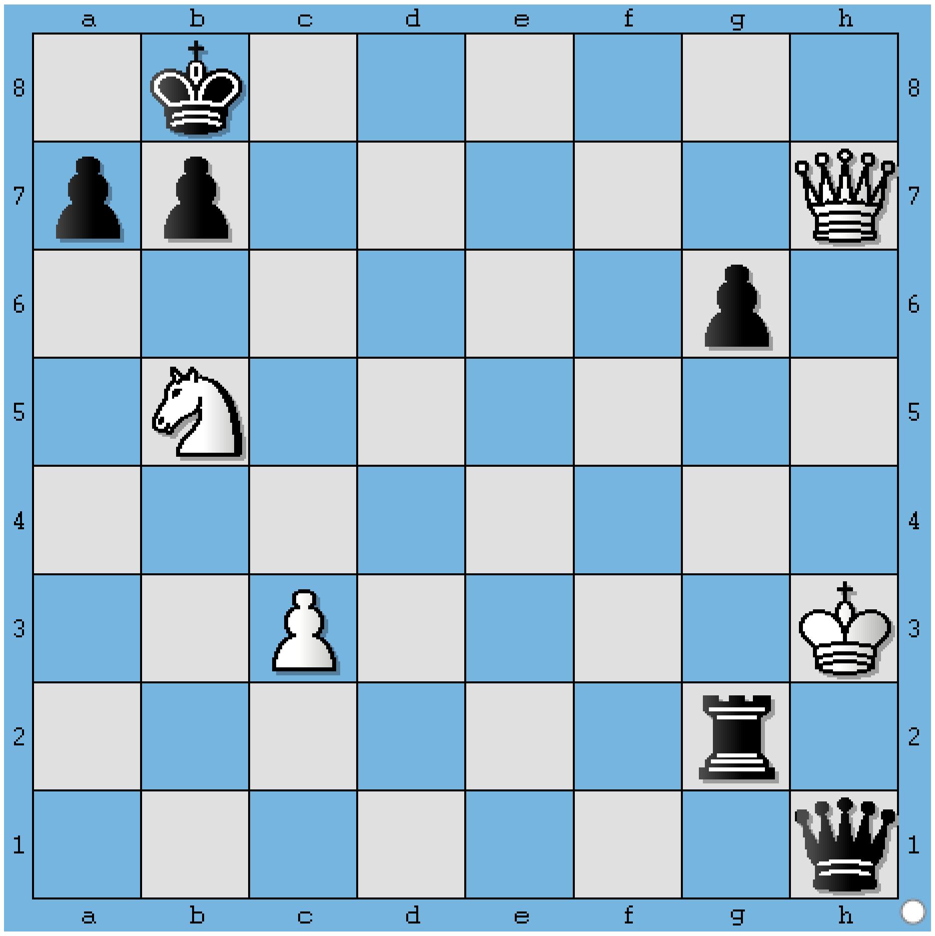


Fig 20: Final position after 38. … Qh1#.

And that is that. A highly entertaining yet informative game that pushed both participants to their limits. This is the essence of human-engine testing; a strong human can keep up with an engine tactically and give it a stern test positionally. However, as can be seen from the above, any mistake often means a fast game over for the human.

Needless to say, while the advantages of human-based testing are many, it is infeasible to conduct a truly large-scale test. It is estimated that at the time of writing, the author has played over two hundred full games of chess against Zephyr, losing the great majority of them, but given the relatively low skill level of the human involved this cannot be regarded as especially significant! At the other end of the scale, the above game is the second of two played against Zephyr by Steve, and on both occasions the game was highly competitive. The 100% score would superficially seem to be damning evidence of the engine's superiority, but two games do not make for a rigorous test and it would not be especially surprising to see Steve hold Zephyr to a draw - if not win - in a rematch.

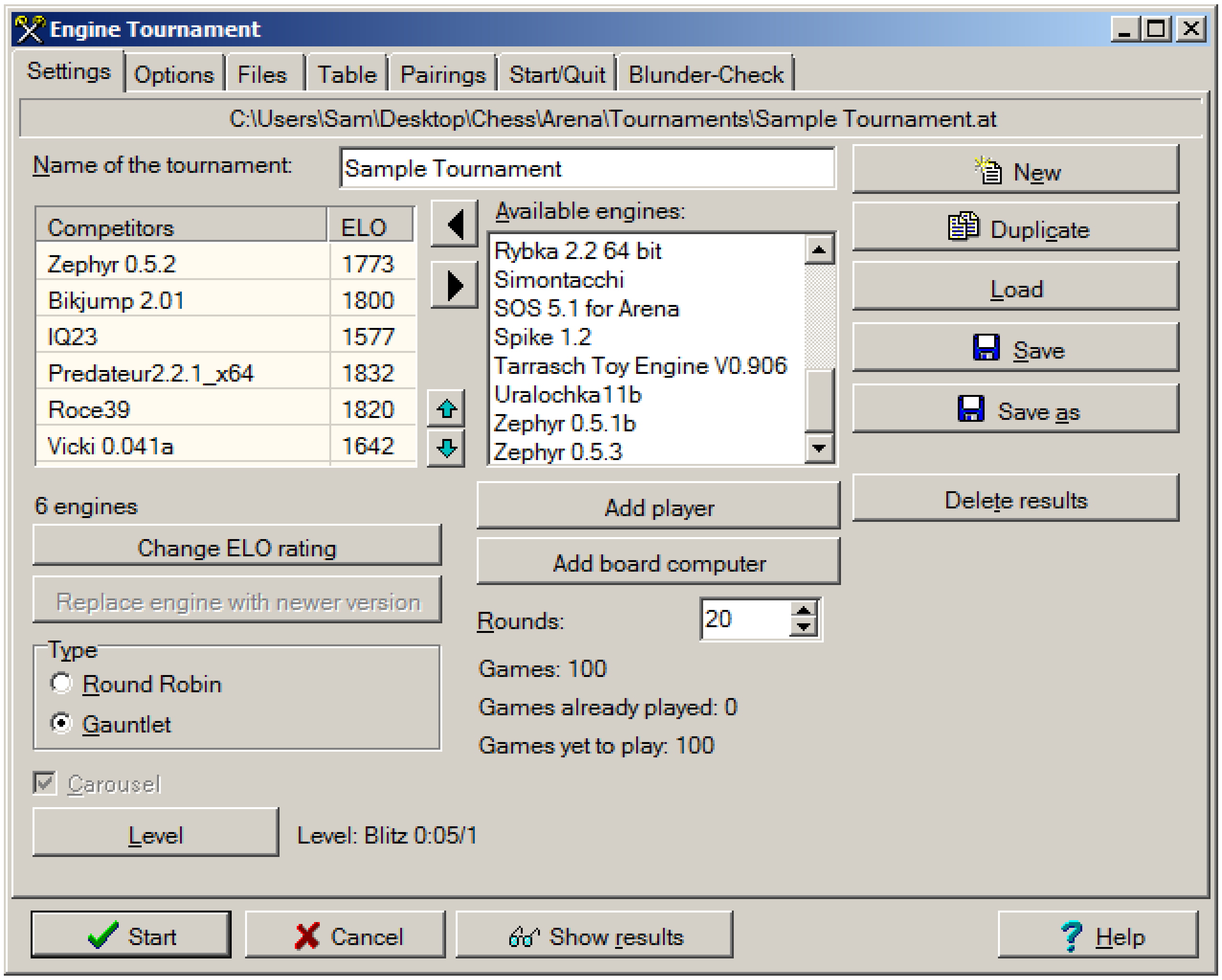
Engine-Engine Tournaments

To gather truly statistically relevant results, we must play a very large number of games. Humans cannot feasibly do this; under the fastest standard time control of five minutes for each side for all the moves ("blitz chess"), and assuming each side uses half their time, a tournament of 500 games would take nearly two days of solid chess playing. To make matters worse, this 500-game tournament might not even be enough to be completely certain which competitor is better!

In situations like this, we can take advantage of the myriad engines freely available on the Internet. The strength of these programs ranges from absolutely dreadful[14] to well above grandmaster standard, along with everything in between. For the purposes of testing, it is recommended to gather a wide range of these engines and pit them against your own to get a rough idea of their relative ability. Typically, a test tournament does best by including, say, two engines that are a little stronger, two engines around the same strength and two engines that are a little weaker. These numbers are of course flexible, but ideally your engine should fall somewhere in the middle of the pack.

Frameworks for setting up automated engine-engine tournaments are again freely available. Most GUIs have the option, although command-line tools exist. Figure 1 (below) shows an example tournament being set up. A round robin tournament has every engine playing every other engine the listed number of times. While this is sometimes useful, for the purposes of testing a new version, we generally want to use a gauntlet tournament. This has the first engine in the list play each other engine the listed number of times, allowing us to focus our attention on this one engine and avoid wasting time on irrelevant games.

14 Notably, there exist engines with a *negative* Elo rating!



C:\Users\Sam\Desktop\Chess\Arena\Tournaments\Sample Tournament.at

Name of the tournament: Sample Tournament

Competitors	ELO
Zephyr 0.5.2	1773
Bikjump 2.01	1800
IQ23	1577
Predateur2.2.1_x64	1832
Roce39	1820
Vicki 0.041a	1642

Available engines:
Rybka 2.2 64 bit
Simontacchi
SOS 5.1 for Arena
Spike 1.2
Tarrasch Toy Engine V0.906
Uralochka11b
Zephyr 0.5.1b
Zephyr 0.5.3

6 engines

Type
Round Robin
[✓ Gauntlet]

[✓ Carousel]

Rounds: 20

Games: 100

Games already played: 0

Games yet to play: 100

Level: Blitz 0:05/1

Fig 21: Setting up a tournament under the Arena GUI. Here, Zephyr 0.5.2 is staged to play 20 games against each of the other five engines, at a time control of 5 seconds plus 1 second per move.

Once the tournament has begun, there is no need to oversee it; throughout Zephyr's development, tournaments were often left to run overnight. A PGN[15] file is created allowing you to look over individual games later if you so wish, but unlike human-engine games, this is not of such great importance. Here, we are more interested in the overall result.

15 Portable Game Notation; see Appendix A

	Engine	Score	Games
1	Zephyr 0.5.2	225.5/500	························· ·························
2	Flux 2.2.1	48.5/50	111111111=111111111111111 111111111=11111=111111111
3	Predateur2.2.1_x64	41.5/50	10=011=11111111111=01110 101111111=1111=1110111111
4	Roce39	41.0/50	111=1110=101111=0=1=11111 =11111011111101=1111=1111
5	Vicki 0.041a	31.0/50	==0100101=1=0100=01101101 =0111===111110=1110110111
6	IQ23	26.5/50	=1=1==00=11===0=1===1010= ===1110==1=0010===1=0====
7	Tarrasch Toy Engine V0.906	21.5/50	0=0100001011=00=001111000 =111==01100=0=00=00110101
8	Philemon	17.5/50	=00=10=1001=110000100=000 ==0001=0=0=0=01010=10==0=
9	Zephyr 0.5.1b	16.0/50	=0==00==0=0000000===00000 ===1110011100=0000101=001
10	Rocinante 101 (64-bit)	15.5/50	01=1=0000=0=000=00=010110 000=001=1000=00=000=11001
10	Piranha	15.5/50	1100000000000000000100100=1 0000=0=1=0110010100110=10

Fig 22: Results from a 500-game gauntlet tournament. Zephyr 0.5.2 played 50 games against each other engine at the 0:05/1 time control, taking around 20 hours to complete. Individual game scores are from the perspective of that engine.

Figure 2 shows the type of output to expect from a typical gauntlet tournament. The fine detail provided by the PGN file is removed, allowing us to easily analyse the results from a higher level. We see that the test version scored less than half of the available points, and that several engines were especially dominant, Flux scoring a full 97% against it. However, we may also observe that half of the engines were beaten overall by the test version, including the previous version of the program. This last point is important; our test version is almost certainly better[16] than the old version, and so we can accept the new version as an improvement.

If the Elo rating of the gauntlet engines are known, we may obtain an estimate of the Elo rating of the test version. It should be noted that 500 games is by no means sufficient to derive a completely accurate rating, but if all we are looking for is a ballpark figure, the method is quite suitable. The above result against these engines gives the Elo rating of Zephyr 0.5.2 as 1674 ± 27 Elo, around the standard of a reasonable human club player.

The results demonstrate a potential pitfall that can be difficult to avoid in engine-engine tournaments. Examine the results of the fifty games between Zephyr 0.5.2 and Flux. Forty-seven of these games were won by Flux, with the other three being draws; none of the games were won by Zephyr at all. Clearly Flux is by far the stronger engine, but this is all we can really

16 Using a rule of thumb on the obtained score of 27-9-14 (W-L-D), we can estimate the new version to have 99.86% *likelihood of superiority* over the old version.

conclude. Say we develop a new version, Zephyr 0.6, and this version scores 2.0/50 against Flux rather than 1.5/50. This score is a full 33% better than before, but even this is likely to be mere noise. We cannot draw any meaningful conclusions from the result.

Likewise, if version 0.5.2 scores 49.0/50 against a particularly weak engine, then a score of 49.5 or 50.0 by version 0.6 does not necessarily mean 0.6 is stronger than 0.5.2; it is far more likely to be due to random chance. Contrast this with a score of 25.0 improving to 26.0; while the absolute score increase is the same, the fact that the engines were previously so equally matched means that an increase is much more likely to be significant[17].

We conclude that engines significantly weaker or stronger than the test engine should be avoided. The results of our tournament were heavily distorted by the presence of such a strong engine; ignoring the results against Flux improves the obtained score from 45.1% to 49.8%, a massive difference. Future tournaments would be better off leaving Flux out, perhaps replacing it with one closer to Zephyr's playing standard. Other methods of improving the quality of the results include increasing the time control or simply playing more games.

Test Suites

A test suite is a standardised collection of positions, each listed together with a so-called "best move". An engine's playing standard may be assessed by giving it a set amount of time per position (one minute and fifteen minutes are common) then seeing whether it finds the best move in each position, and if so how long it takes to find it. Typically this procedure may then be used to produce an estimate of the engine's Elo rating. A simplified version of this, which Zephyr uses for its tests, is to ignore the time taken and simply award a point whenever the chosen move is the best move (although still obeying the time limit).

Test suites are often highly specialised and thematic; for example, a suite may be geared towards how an engine deals with positions revolving around passed pawns. Consequently, the results should be interpreted only within this limited domain. A high score on the aforementioned suite suggests the engine handles passed pawns well, a low score not well, if at all. This score is not, however, a reflection on its playing strength as a whole.

Used correctly, however, test suites can be highly useful to the early development of a chess engine, especially when resources for testing are limited. The general purpose Bratko-Kopec test suite takes 48 minutes to carry out (24 positions, two minutes each), much faster than the 500-game tournament discussed in the previous section. This comes at the cost of a less precise result, but it can serve as a litmus test; if a change causes the score of a test suite to drop significantly, we might consider discarding it immediately rather than wasting time going through the full test cycle.

Test Results

To play a strong game of chess, the system must first be capable of playing chess. To demonstrate the correctness of Zephyr's move generation, we use the Perft test detailed in the previous section. In the initial position, the values for perft(x) produced by Zephyr match the expected values for all x from 1 to 6 (values of perft(x) for higher x take a very long time to

17 This statement is only true in theory; in practice, we'd probably ignore such a small increase regardless.

compute for relatively little gain). Many other positions were subjected to Perft testing, each producing the expected value. These positions - together with the rationale for using them - are listed in Appendix [INSERT REFERENCE].

There is no formal way to test overall stability of the system, but the empirical evidence provided by the thousands of games played by Zephyr over the year seems quite adequate. It has always correctly detected when the game is over and assigned the appropriate score. Zephyr is capable of playing at extremely fast time controls with little difficulty; the fastest used throughout testing was 10s+0.1s, which leads to games lasting less than a minute for all the moves by both sides.

When satisfied the engine plays legal chess, we may turn our attention to its playing strength. The following results are all based on Zephyr 0.7.2, the latest version at the time of writing.

We have already observed Zephyr's performance against other engines, so here we shall focus on results from test suites. The amount of time recommended per position in each test suite varies considerably; for the sake of consistency and keeping the duration down, we will allow Zephyr one minute per position across the board. Under these conditions and running on the author's laptop,[18] Zephyr's score is as follows:

```
Bratko-Kopec: 16/24
Fine #70: 1/1
Kaufman: 18/23
Mates: 40/50
```

The Bratko-Kopec and Kaufman test suites are both general purpose. The scores are generally high, indicative of a rating around 2,100 Elo. The Kaufman score in particular is impressive. It should be noted that as a workaround to a known flaw in the test suite code, two positions were omitted from the standard Kaufman suite. These positions require the engine to *avoid* the given move rather than play it, but the code assumes the given move is always the best one.

Fine #70 is a particular position, also known as the Lasker-Reichhelm position, which serves as a test for the transposition tables. The position is in the late endgame and highly blocked, so a working transposition table implementation allows the search to very quickly reach depths of over 25 ply. In any engine with a reasonable evaluation function, the sole winning move Kb1 is only possible to find when searching to these extreme depths. A program without a functional transposition table will opt for Kb2 instead, leading only to a draw.

The "Mates" suite is a collection of positions where it is possible to force a checkmate. Some of the positions are extremely basic - the first position is a mate in one - and serve more as a test of whether the engine is capable of discovering and playing a checkmate at all. Later positions serve as a stern test of the engine's tactical capabilities, with the last position being a mate in seventeen that many grandmasters failed to solve correctly. Overall, the score achieved is quite good, especially given the one minute time limit.

It is unreasonable to expect a perfect score on any of these test suites (with the exception of Fine #70!), but certainly some of the scores could be further improved. It is also

18 Intel Pentium T4500 (2.3GHz), 4GB DDR3 RAM

evident that there exist far stronger engines than Zephyr, even when considering only the subset of engines also written in Java.

Bugs and Issues

There are some bugs that still remain within the system. For the most part these issues lie in non-critical components of the command line interface, unlikely to ever be seen by a typical end user. To the author's knowledge, the system does not experience any significant issues during standard gameplay.

Bugs with the gameplay do exist, however, which have the effect of reducing its effective playing strength. Most notably, an issue lies in the interaction between the transposition table and the triangular arrays used to gather the principal variation. When we have a hash hit of depth higher than the depth we wish to search to, we return the score immediately; no move is stored in the triangular arrays. This is not a problem for the first search conducted, as we must have searched the position properly at some point, but since subsequent searches wipe the triangular arrays before commencing we will not have any move to play unless the search reaches a depth not covered by the transposition table. Since this is not guaranteed (the search might not have time to finish the first non-hash-hit iteration), we run the risk of not having a move available to play.

Currently, Zephyr gets around this issue by also clearing the transposition table between searches, thus forcing the triangular arrays to be filled as normal. Needless to say, this is rather wasteful; we would prefer to keep entries from previous searches, as they can still be helpful in this search. However, as of the time of writing, attempts to solve the issue "properly" have yet to succeed. While it is unlikely that Zephyr stands to gain a lot of strength from the fix, it would certainly seem to be an unnecessary limitation!

Another problem, albeit slightly less severe, is that the engine sometimes takes a draw by threefold repetition in a position where it has a reasonable chance of winning the game. This occurs due to draws having an assigned score of (effectively) zero. It works well if the best non-drawing move has a score of -477 centipawns, as the drawing move saves the engine from almost certain loss; however if the best non-drawing move gives a score of only -12 centipawns, the game is still more or less level but the draw is taken anyway. It should be noted that this behaviour is good against superior opposition which is more likely to later force a win, and vice-versa for inferior opposition.

A final issue lies with time management. That is, on occasion the engine loses on time, despite efforts to the contrary. This is very rare, occurring perhaps one game in every two hundred at the fastest time control used throughout testing (and so may be considered a non-issue), but ideally the possibility would be completely eliminated.

Both of these last two issues are fixable; the former with the introduction of a so-called contempt factor, the latter with a rethink of how the engine assigns and obeys time limits. However, the fixes were always considered relatively low priority, and so were never carried out.

Section 6: Future Work

Future work on Zephyr will be primarily concerned with further increasing its playing strength, using the methods detailed in Further System Design (section 3.3.4). They are not listed in any particular order, but the next feature to be implemented is likely to be futility pruning or an improvement to the move ordering scheme (such as the history heuristic), as one common observation in games against stronger engines is their higher search depth leading to a simple - but long - tactical combination that had not yet been seen by Zephyr. Another area in need of improvement is the evaluation function, which remains rather basic compared to that of a typical engine. A full list of planned changes is maintained at the top of the `Zephyr` class; implementing everything on the list would culminate in a serious increase of ability.

Work is of course also required to remove the bugs and issues that remain in the system, in particular the problem with the transposition tables and the triangular arrays. An increasingly tempting approach is to rewrite entire classes of the program (especially `Search`) in order to clean up code and remove any unseen inefficiencies.

Section 7: Conclusions and Reflections

The primary aim of this project has been to design and implement a chess engine from scratch with as high a playing standard as possible. In light of Zephyr's results against various opposition and in the test suites, we may conclude that it has achieved a good level of strength and remains stable. The requirements have therefore been met.

It is difficult to say whether anything would be changed, if the project were to be started over. One early decision that shaped much of the rest of Zephyr's development was the choice of bitboards as the primary board representation. At the time this was a calculated gamble, opting for a somewhat unintuitive structure in exchange for a potentially faster engine once everything was in place. Certainly the learning curve became steeper as a result of this choice. It would be an interesting experiment to develop a separate program using a different board representation such as 0x88 in order to compare the two approaches, as in retrospect bitboard engines and 0x88 engines are completely different beasts.

Overall, the project has been a fascinating experience, and development of Zephyr will be continuing past the submission date of this report.