KTH Computer Science
and Communication

Evolutionary Tuning of Chess Playing Software

JONAH SCHREIBER AND PHILIP BRAMSTÅNG

Degree Project in Engineering Physics, First Level at CSC
Supervisor: Johan Boye
Examiner: Mårten Olsson

Abstract

In the ambition to create intelligent computer players, the game of chess is probably the most well-studied game. Much work has already been done on producing good methods to search a chess game tree and to statically evaluate chess positions. However, there is little consensus on how to tune the parameters of a chess program's search and evaluation functions. What set of parameters makes the program play its strongest? This paper attempts to answer this question by observing the results of tuning a custom chess-playing implementation, called *Agent*, using genetic algorithms and evolutionary programming. We show not only how such algorithms improve the program's playing strength overall, but we also compare the improved program's strength to other versions of *Agent*.

Contents

Chapter 1

Introduction

In artificial intelligence, the famous Turing Test is often used as a platform to designate whether a machine or computer is "intelligent" or not. The test expresses a central motivation of the field: creating machines that can imitate or even surpass the actions of a human expert. As stated, this motivation is misleading, however, because it imposes no requirements on how the machine actually works. A machine that also imitates the *processes* by which humans acquire some particular behavior is much more useful in software applications than one without this capability, because such a machine can solve problems that were not considered in its design. The hope to achieve such *machine learning* is a fundamental motivation of artificial intelligence.

Unfortunately, despite extensive study, designing such machines or programs has proven difficult, primarily because the exact processes by which learning can happen are not well understood. While a program can learn in ways unlike those of humans, human learning usually has more apparent computative approaches.[1] One such commonly cited approach is to let a program "learn from its mistakes", like humans tend to do, but this approach is not only vaguely formulated, it is inherently problematic. In particular, for a game like chess, it can be infeasible to let a computer program learn from its mistakes, because computing the mistakes (or at least *when they took place*) would be equivalent to finding the best move in the first place. In fact, the way most chess programs work today is by actually doing this computation (that is, finding the best move), making any learning seem unnecessary (see Figure 1.1). The problem is analogous to a parent giving a child a job, and, in having to teach the child how to do it, the parent ends up having to do the job herself.

Machine learning is, of course, still very desirable because there are often inputs on which such mistake calculations are incomplete, incorrect or inaccurate. For example, if a chess program is somehow punished for making a move that was incorrectly labeled as a mistake, it would be taught a false lesson. Hence, in order to attain any useful mistake-based machine learning, mistakes must be very clearly

[1] Such approaches are still complicated since human learning is not well understood, either.

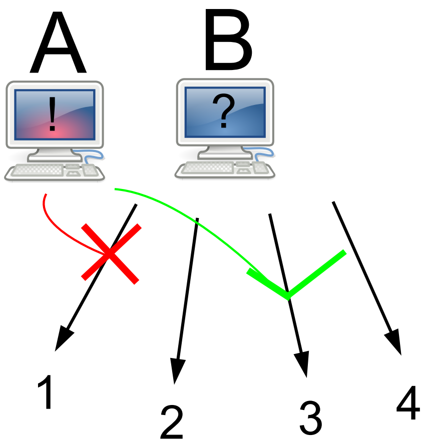


Figure 1.1. If the machine A can already readily identify the mistakes of the learning machine B, it might seem unnecessary for B to exist. In most cases, however, the machine A is incomplete, justifying the existence of machine B, since it could learn to generalize from and surpass A's capabilities.

classified. In most cases, this means the mistake feedback mechanism cannot be too complicated. However, such a decrease in complexity means there is less information for the program to learn from. The problem is therefore to reduce the complexity of the feedback mechanism, while at the same time increasing the sophistication of the learning mechanism. Regrettably, the former is often much easier to achieve than the latter.

Fortunately, various generalized learning methods have been proposed over the years to complement a more simple feedback system. One such method, called *genetic algorithms* (studied in detail by Fraser and Burnell, 1970), is the focus of this paper and its description follows in the next sections. In fact, as this paper will prove, the method is so powerful that an extremely simple feedback system proves to be sufficient. Specifically, this paper shows, among other things, that designating mistakes based only on losing or winning a game of chess is sufficient for a chess program to learn and improve its playing strength, even though one might normally expect such information to be insufficient.

1.1 Problem description

Chess has long been a quintessential showcase of the intelligence of computers. While hopes have historically been high to produce a self-learned chess program, most programs today still use a heavily improved type of brute-force search coupled with an *evaluation* strategy. This observation leads one to ask if machine learning of chess is an intractable problem. Can a chess program teach itself anything inside the programmer's framework, or must the programmer define all its routines explicitly?

Fortunately, while most chess programs use very conventional search techniques which leave little room for automatic learning, the best way to compute a strategic evaluation of a chess position is more of a mystery. There is great disagreement on what game elements to evaluate strategically, and by how much. In fact, evaluation strategies are some of the most well-kept industry secrets. Because of this method's evasive nature, we might ask instead: "Given basic rules for searching for the best move, can a chess program teach itself to play better by learning what makes a chess position advantageous?"

This question marks the foundation of our research. With so many elements contributing to strategy in chess, what selection and emphasis of such elements makes a program play its strongest? Can the program be made to answer this question itself? Understanding the learning capabilities a chess program has in this area is the driving force of this research, and is what motivated this project to use genetic algorithms as a program's learning basis.

This paper will answer the question on how much a chess program can learn and improve its evaluation through genetic algorithms by letting our own chess program *Agent* evolve in a simulated ecological environment. This paper shows how the program has improved in various simulation cycles by letting the different generations play games against other versions (such as earlier generations, "greedy" versions, etc.), and observing the win rate. It also shows an unsuccessful attempt at producing *triangular specialization*, which will be described later.

1.2 Genetic algorithms

The purpose of this paper is not only to show how genetic algorithms can be employed to optimize a chess-playing program, but also how they embody a flexible learning environment in a broader context. In other words, the intent is to illustrate the versatility of genetic algorithms by highlighting the learning obstacles it manages to overcome in a rather complex game such as chess. To achieve this end, we will first establish precisely what we mean with genetic algorithms.

Genetic algorithms fall under the category of *evolutionary programming*, and are designed specifically to imitate the evolutionary processes found in nature. Strictly speaking, they are actually a form of *search heuristic*. The goal is to search for some problem-specific maximum - as such, they aim to *optimize* some set of parameters to this end. They are heuristic because parameter candidates are selected through an algorithmic process that mimics *natural selection*, which acts as a kind of educated guess. Genetic algorithm applications also usually have a method for mimicking *reproduction with variation*, the other core component for evolution in nature.

Genetic algorithms usually work on a population (or "pool") of virtual organisms. Each such organism has an associated *genome*: an information string which expresses features of the organism. For example, the importance a chess program assigns to castling might be part of its genome. Through this genome, mating two individual organisms to produce an offspring is then possible, by crossing over the respective genomes with a chance of mutation. This crossover then produces an offspring which inherits half its genome from one parent, and half from the other.

These crossover and mutation processes are usually simulated during the "reproduction with variation" phase of a genetic algorithm. The genome is important to define because it forms the basis by which an organism's fitness can be measured.

In the case of chess, we usually want to maximize the win rate, so that the program wins as often as possible. A chess program which wins often could be called *more fit* than others which win less. Measuring such fitness is done by evaluating a *fitness function* and these are central to a genetic algorithm. Their input is usually an organism's genome and their output is a measure of the organism's fitness. Hence, a fitness function directly corresponds to whatever quantity the application wants to maximize. In short, a fitness function measures how "good" an organism is.[2]

Once these simulation elements have been decided, they can be combined into a genetic algorithm. A common algorithm is illustrated in Figure 1.2. First, three organisms are sampled from the organism pool. They are then ranked after fitness using the fitness function, and the weakest is eliminated. Finally, the remaining organisms reproduce, recombining and possibly mutating their genomes into a third organism which is reinserted into the pool.

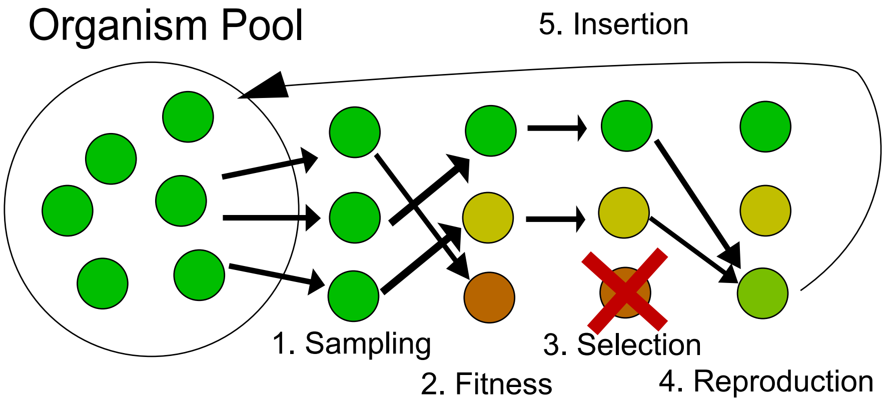


Figure 1.2. Common approach for a genetic algorithm.

This algorithm is simple to implement but has a disadvantage in that organisms are only given one chance to survive. If the fitness function has significant statistical variance, potentially fit individuals may be eliminated too hastily. For example, in chess, the outcome of a single game does not certainly decide which player is best. Hence, a sum of averages may be required. Algorithms which remedy this problem exist, and our own implementation is described in detail in Section 2.2.

A genetic algorithm usually defines borders between *generations*. They are analogous to "family" generations found in nature, in that they approximately delimit a population's parents from their children. In practice, what constitutes a generation varies between genetic algorithms, but they generally separate populations into categories where one might expect a significant genome change from a previous gen-

[2] In practice, the fitness function is also the most computationally intensive part of any genetic algorithm, because it is usually a very complicated non-linear function.

eration. For example, if the algorithm described above is repeated as many times as there are organisms in the pool, we may call this new pool the "next" generation. It is useful to speak of generations because they provide a standardized measure of how much a population has advanced.

By exercising the steps described above, a genetic algorithm should in theory maximize the fitness of a population. The time taken for this to happen may, however, be very long. Nevertheless, genetic algorithms are very useful because they will almost always find partial solutions (optimizations) along the way, and will sometimes find unexpected (but very fit) solutions to a problem, unlike many other optimization algorithms.

1.3 Background of chess programs

In 1997, the chess computer Deep Blue defeated grandmaster Garry Kasparov in a very famous match. It was the first time in history that a machine had won a game of chess against a reigning world champion. Naturally, a great deal of public interest arose as to why Deep Blue was such a powerful chess player, and it was revealed that a key component to its success was the way its play parameters were tuned. By letting Deep Blue analyze many thousand grandmaster games ahead of time, its parameters could be adjusted so as to match results in these games.

Since then, however, chess program strength has improved even further. For example, the chess program Rybka won four consecutive World Computer Chess Championships between 2007 and 2010 and is considered one of the strongest programs today. In addition, unlike Deep Blue, Rybka could also be run on ordinary hardware and still play very well. This was also the case for Deep Fritz, which even managed to defeat Classical World Chess Champion Vladimir Kramnik in 2006 while running on relatively ordinary hardware. Like Deep Blue, both programs use very carefully designed evaluation.

However, while Deep Blue's tuning indeed helped it play very strong moves, there is no clear consensus on what tuning method is truly best. Moreover, while many of the techniques to search a chess game tree for the best move have been well established and conventional for decades, the best way to compute a strategic evaluation of a chess position is more of a mystery. Indeed, it can be argued that this task of *static evaluation*, first described in 1950 by Shannon, defines a program's style of play and even identity in some sense.

Having learned of the significance of static evaluation, our interest was sparked as to how evolutionary programming influences it. Previous work by David B. Fogel et al on the possibility to improve a chess program with evolutionarily programmed neural networks was particularly inspiring. We already had some experience in how useful genetic algorithms could be to find unexpected solutions or optimal ratios to some problems. Hence, to apply these algorithms to a game like chess seemed to be a very lucrative experiment.

Chapter 2

Methodology

The following section describes the methods used to answer the questions given in Section 1.1. Firstly, this section expands on how our custom chess program *Agent* works and how it can be used to simulate chess games with custom "profiles". Next, this section describes how the genetic algorithm was constructed and how the simulation was set up. Finally, it describes the actual simulations that took place and how their results were measured. Many of the terms used in this chapter are defined in greater detail in this paper's glossary section.

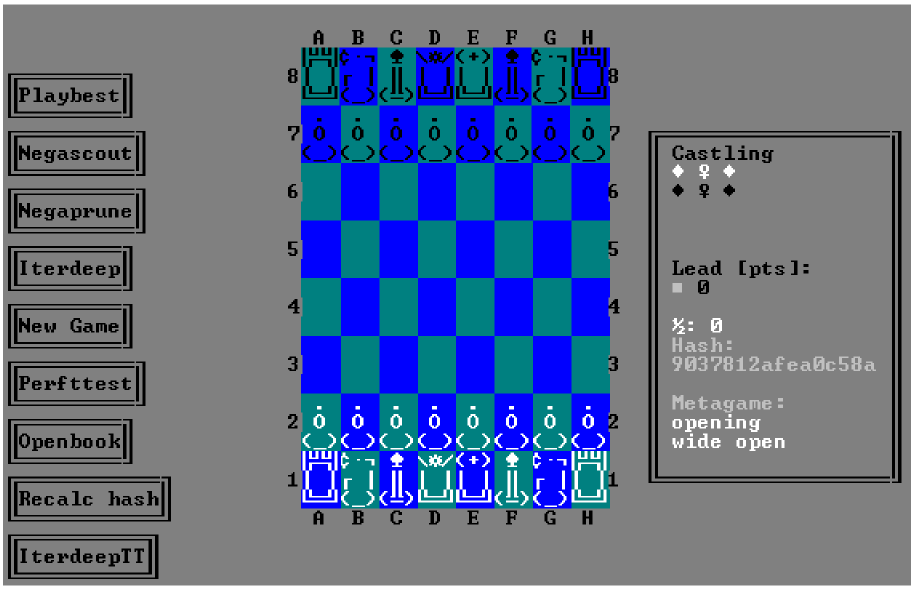
2.1 Chess program implementation

Agent is our custom chess program, written entirely in C. Its main structure is fairly traditional; its search mechanism and evaluation mechanism being the most important modules. Figure 2.1 shows its standalone Microsoft Windows user interface when launched without any options. In total, *Agent* consists of 7,000 lines of C code.

As the predominant focus in this paper is how *Agent* was made to improve itself, we will not comment extensively on all its modules, although much project time was devoted to them. Instead, we refer to external literature for more traditional information, of which there is a comprehensive list of in this paper's Bibliography section. However, *Agent*'s more novel features will be expanded on.

The general design philosophy of *Agent* was to fully exploit any speed gains by using precomputation, function inlining and bitwise operations wherever possible. As speed is essential to a chess program, optimization was a high priority throughout most of the program.

Agent's internal data representation of a chess position is somewhat novel. It is reminiscent of a traditional *mailbox* approach (first described by Spracklen, 1978a), but uses two mutually corresponding arrays instead of one. One array maps a piece to its square on the board, and the other does the opposite. The advantage of this approach is that it makes *move generation* (a function chess programs use to generate all legal moves from a position) very fast, at the cost of more memory.



```
Castling
◆ ♀ ◆
◆ ♀ ◆

Lead [pts]:
■ 0

½: 0
Hash:
9037812afea0c58a

Metagame:
opening
wide open
```

Figure 2.1. The default Microsoft Windows appearance of *Agent*.

Ultimately, this structure was designed to be more informational than a more traditional mailbox system, yet as fast but less awkward as a *bitboard* (named and described by Adelson-Velsky et al, 1970) system. Along with piece positions, the structure also contains information on whose turn it is to move, castling rights, en passant states, a half-move clock and also a 64-bit hash signature.

2.1.1 Chess profiles

A central structure in *Agent* supports different *chess profiles*. These are structures containing variable information on how the program should play a move. They contain information pertaining to (mostly numerical) expressions in its search and, in particular, its evaluation. For example, the material value of having a pawn, the importance of having knights close to the center, and the threshold at which bad captures should be discarded or re-ordered can be changed within such a chess profile. This enables *Agent* to simulate two different opponents playing against each other, along with other interesting uses. One important use is their role as genomes during simulation, which is described more in Section 2.2.

2.1.2 Search

Agent uses, like many chess programs, a search function based on *negascout* (Reinefeld, 1983), which in turn uses *alpha-beta pruning* (improved by Knuth, 1975) to reduce the search space. Like any algorithm based on *minimax*, the main idea is to assume both players always play optimally. Through negascout, the search space is reduced through an implicit *principal variation search* (described by Marsland and

Campbell, 1982) and *null window searches.* To improve the performance of these, Agent uses heuristics to improve move ordering, of which the chief ordering of principal variation nodes is accomplished through *iterative deepening* (described by de Groot, 1965). Through the iterative deepening framework, *Agent* automatically stops deepening once its time limit is (approximately) exceeded.

While *killer heuristics* (Akl, 1977) were a priority to implement, *Agent* conspicuously lack them because of time constraints. In place, *Agent* orders moves at each search tree node by first checking good captures, then regular moves, and finally bad captures. Captures are ordered in this fashion through a custom *static exchange evaluation* (SEE) (Spracklen, 1978b) implementation which uses a rather fast and novel approach. This move ordering was found to increase the search speed significantly.

The SEE implementation is also (traditionally) used in *Agent's quiescence search* (first described by Harris, 1975), which is a reduced "dramatic-moves-only" alpha-beta search done at the search tree's leaf nodes to avoid the *horizon effect.* These dramatic moves include captures and promotions. Checking moves that evaded check was planned, but never implemented. *Agent* uses SEE to order the captures and also discard bad ones according to a defined profile threshold. This means for instance that if this profile threshold is set to 0 (default), the quiescence search will not look at any capture sequences in which it loses more material than it takes. If this parameter is instead set to -2000 however, net material losses of up to 2000 points (the worth of two pawns) will be considered.

Agent is also conspicuously devoid of *transposition table* lookups (Warnock and Wendroff, 1988), despite having functionality implemented to support them (such as *Zobrist hashing*, Zobrist, 1969). This is unfortunate because in theory, such tables dramatically improve search speed, particularly in the endgame. For this project's simulations, they were disabled because during testing, they produced odd results and bad moves while quiescence search was also on. Despite prolonged debugging, the problem was never found and their usage during search was simply omitted.[1] The functionality remains, however, to implement the three-fold repetition rule in chess, for which the tables work without problems.

2.1.3 Evaluation

The second major module, *evaluation* (or *static evaluation*, first described by Shannon, 1950) is in some sense the heart of the program and is responsible for statically (without looking ahead at further moves) evaluating a chess position and returning a numerical score based on how good the position looks. It observes that chess is a zero-sum game in that a loss for one side is exactly equal to the gain for the other. In this instance, this results in positive scores for white, and negative for black.

Almost all of the factors considered during evaluation are also represented in chess profiles. This enables all such factors to be tuned independently of each

[1] The most likely explanation we have for the moment is that some form of *search instability* appears with the tables.

other. They determine how much emphasis the evaluation puts on different aspects of the position. The following is a comprehensive list of the considered factors:

- Material strength
- Knight outposts, rook outposts
- Connected pawns
- Doubled or tripled pawns
- Principal diagonals for bishops
- Knights in the board's center
- Castling and removal of opponent's castling rights
- Value of bishops/knights in open/closed games
- Centralization of the king in the endgame
- Rooks on open files

When *Agent* evaluates a position each chess piece receives an individual score, which are then summed up to determine which player the position favors. An example of how such evaluation functions are implemented is presented below in Figure 2.2:

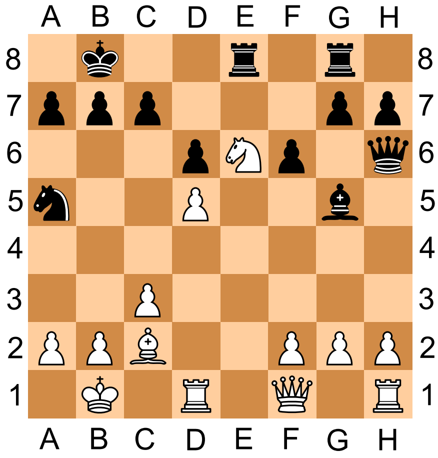


Figure 2.2. Example position.

Consider the board position and specifically the white knight on E6 as an example. First, the knight's material value is considered. This is set to 3000 in the default profile (three times as much as a pawn, which is the standard material value of a knight in chess).

Secondly, the knight is given a bonus based on its proximity to the center of the board (knights are generally stronger in the center, where they can threaten many squares at once). This is done by using a precomputed array, which has 64 entries corresponding to the 64 squares on a chessboard. The entries representing squares closer to the center contain greater values than those on the edges (see Figure 2.3). These values are then multiplied by a profile factor (default 9) which corresponds to the emphasis that profile places on having knights in the center. In the case of the knight on E6, this calculation becomes $29 \times 9 = 261$.

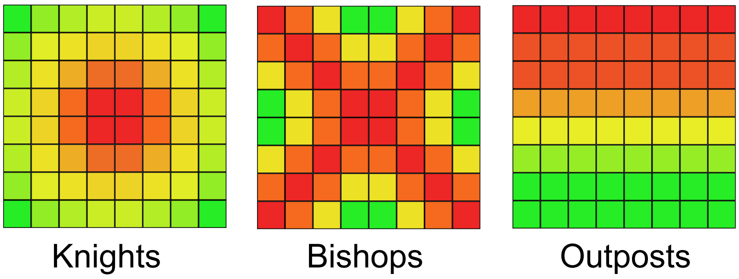


Figure 2.3. Visualisation of bonus arrays. Red indicates higher values.

To gain more opportunities for tuning, a common convention in *Agent* is for each such precomputed array to be given several versions of itself. For example, consider the "Knights" array in Figure 2.3. At what rate should the values increase towards the center? It is tempting to only have, say, a linear growth, but this leaves fewer opportunities for tuning. Instead, *Agent* often has several versions of arrays that grow with different rates. Conventionally, there are six versions for each array, named after in which manner they grow towards their maximal value: *linear*, *quadratic*, *cubic*, *square root*, *sigmoid* and *inverse sigmoid* (see figure 2.4). This allows a profile to have even more opportunities to fine-tune its evaluation.

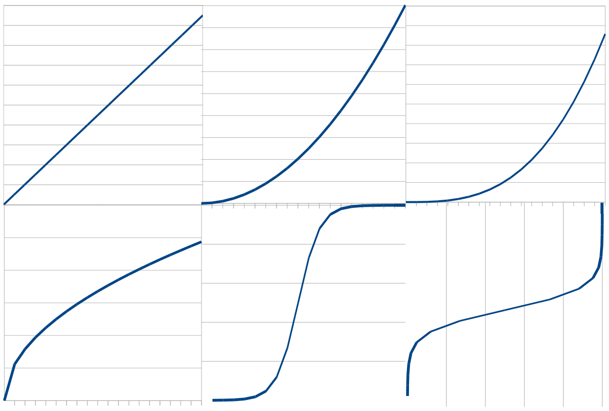


Figure 2.4. The different "growth" paradigms *Agent* uses in evaluation. They are in order *linear*, *quadratic*, *cubic*, *square root*, *sigmoid* and *inverse sigmoid*.

Let us return to the knight on E6. This knight is also an outpost, meaning it is standing on a square that is protected by one's own pawns while it cannot be attacked by an opponent's pawn. This is a tactical advantage, and as such merits an evaluation bonus. The procedure is much like the previous one, where the "outposts" array in Figure 2.3 is used instead to emphasize outposts closer to the enemy. As before, there is also a profile factor which decides what emphasis a profile places on outposts (default is 10). Hence, similarly, the outpost calculation becomes $24 \times 10 = 240$.[2]

Knights also receive a bonus in closed games because of their ability to jump over other pieces (meaning they are not as restricted in their movements as other pieces when the center is crowded). A measure of openness is governed by a "metagame" variable (described in greater detail in the next section) which is based on the number of pieces currently in the 6×4 center squares on the board as well as the number of pawns that are locked. The considered position however (Figure 2.2), is very open and thus warrants no such bonus.

Finally, the evaluation scores are summed up for a total of $3000 + 261 + 240 = 3501$ for the white knight on E6, which is the value that Agent will consider it worth. In contrast, the same calculations for the black knight on A5 (which is not an outpost and is very restricted by its position at the edge of the board), would result in the value 3081. The fairly large difference between these two values is a prime example of how different the evaluation module can consider two pieces to be. In theory, two knights are equal and should have the same worth, but in practice one has to take strategical considerations into account.

Another, more complex example of an evaluation function is the way Agent handles pawn structures. This calculation is inherently more complicated than those of all the individual piece evaluations, since it not only involves several pieces but also their positions relative to each other. Therefore, the main challenge comprises finding a sufficiently quick and efficient way to compute an accurate bonus or penalty.

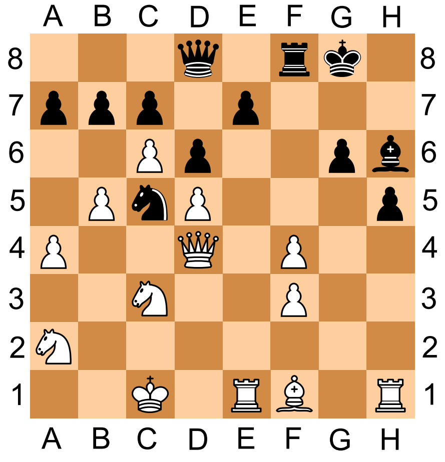


Figure 2.5. Example pawn structure position.

[2] The outpost array is, of course, reversed when evaluating black pieces.

The implementation uses the fact that a lot of information is contained in each sequence of only two rows. If two rows of pawns is represented as a number, with bit 1 for "pawn here" and bit 0 for "pawn not here", then there are $2^{8\times 2} = 2^{16} = 65536$ arrangements.[3] These numbers can be treated as indices in a large (65536 indices) array containing a precomputed structure factor for each arrangement, which can then be used for evaluating the whole structure.[4] Consider the position in Figure 2.5.

Agent would then, for the the position in this figure, evaluate the pawn structure according to the procedure given in Figure 2.6 below. Each "2-row" receives a bonus based on how well-structured it is. In this example, the first 2-row receives no bonus. The second receives a penalty because it looks like an isolated pawn. The third receives a bonus for having two connected pawns, and the fourth for having connected pawns (but penalized for having two isolated pawns). The fifth receives a large penalty for doubled pawns, and the sixth for having a single isolated pawn.

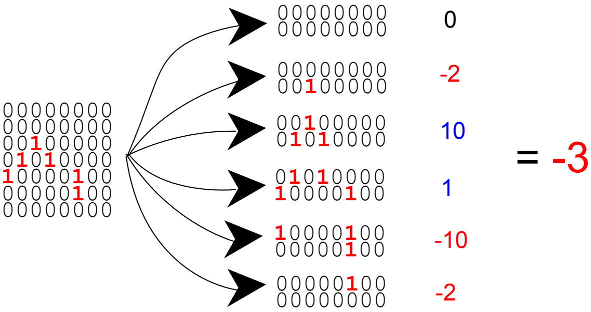


Figure 2.6. The procedure by which a pawn structure score can be extracted.

The scores are added up resulting in $-2+5+5+5-4-10-2+0 = -3$ which is then multiplied by a profile factor (5 by default) resulting in a pawn structure evaluation for white of $-3 \times 5 = -15$. This is a small number (compared to the value of a pawn 1000) which means that white's pawn structure is considered to be fairly neutral (some connected pawns, but also doubled pawns on an isolated file).

Part of the idea is that the 2-rows overlap, allowing quite a lot of information to be guessed and extracted like the above illustrates. While the precomputed array only considers elementary structure factors, it leaves much room for improving the strategic accuracy of the array. In addition, the calculated 64-bit number can also be used to simplify other important strategic calculations, such as passed pawns. This approximation method, coupled with the fact that all of the calculations use extremely fast bitwise operations, allows for a very efficient overarching evaluation of the pawn structure.

[3] *Somewhat less* actually since there can only be 8 pawns (the highest number would be 1111111100000000 = 65280).

[4] *Agent* also uses three such versions for more tuning: *unbiased*, *hates doubled* and *loves connected*.

Overall, *Agent's* evaluation is comparatively simple but manages computer processing power well by being very fast. It was also designed to be easy to expand on, using established paradigms.

2.1.4 Miscellaneous

Chess programs tend to play poorly in the opening, because humans are very familiar with what strategies and openings work best. Therefore, an *opening book*, which is a data structure containing common opening lines, is often deployed so as to mitigate this problem. *Agent* has a custom opening book which includes a 10 megabyte text file containing 180,000 recorded high level chess games to a depth of 8 moves by each side. *Agent* also has a function that reads the file and chooses a suitable move, including somewhat complex rules on how and when to leave the book. (In short, the program attempts to stay in the book for as long as possible).[5] The move is chosen randomly among the available moves but each move is weighted with its frequency of occurrence in the recorded games.

Each position is also supplied with *metagame* information, which includes an estimation of how far progressed the current game is (opening, mid-game, endgame) and of how open or closed the center of the board is. Since these variables change slowly during a game, their progress is only approximately updated during the search. Instead, after each move is made, a more thorough calculation of these factors are made. These factors are useful not only for evaluation, but changing the search slightly as a game approaches the end.

2.2 Simulation setup

The different profile parameters which *Agent* uses in its evaluation of a position are largely based on guesswork. For instance, how much value should the evaluation put into castling? Is it worth being down a pawn for that added king safety? Half a pawn? And what is the relative value of taking away your opponent's castling rights?

The tuning of these parameters is a complex optimization problem. As stated in Chapter 1, it is one this project intended to solve using genetic algorithms. In this specific case, the base concept is to create a base population of random profiles, letting them play against each other and then using the top performers as parents for a new generation of profiles. This way each generation is stronger than the previous, until finally the algorithm stabilizes (for some time) because a part-way optimal configuration of the parameters has been reached.

The genetic algorithm is fairly straightforward and is illustrated in Figure 2.7. The organisms it treats are virtual chess players whose playing style is defined by a chess profile. Hence, the chess profile acts as the organism's genome. The algorithm's fitness function, then, is simply a simulation of a chess game between

[5] As a result, the program will always play out all 16 moves if played against itself.

two players (which acts as competition between two organisms). This way, better fitness corresponds to winning more often, which is, of course, the quantity we wish to maximize - we wish to find the optimal chess player.

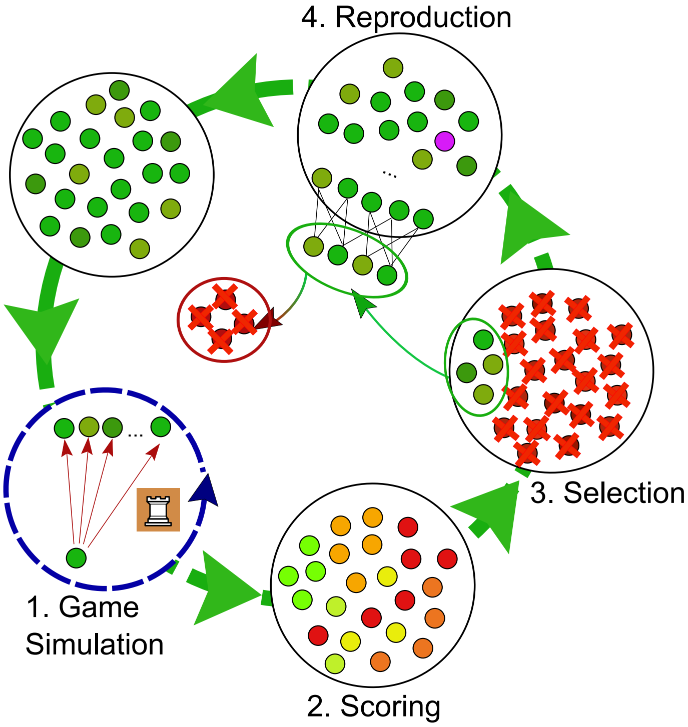


Figure 2.7. Flowchart illustrating the different phases of the genetic algorithm.

The first step in the algorithm is to create a base population. The base population was created using randomization (within certain reasonable bounds) of *Agent's* default profile, creating very diverse profiles. During a simulation cycle, the algorithm first computes "matchups" for the current generation. Each profile (organism) gets to play against a number of opponent profiles chosen randomly from the same (or a different population, depending on the simulation mode; see section 2.3.1). This way, a single game does not decide an organism's fate.

After matchups have been computed, the algorithm simulates each matchup by using calls to *Agent*. The results of each game are collected and each matchup gives points or penalties to both opponents. This phase is equivalent to evaluating the fitness of each organism. After scoring each organism based on the results of the games it played, the organisms are ranked after fitness and only a percentage of the best are kept. Finally, pairs of these survivors are randomly selected to produce recombined, possibly mutated offspring, until the population has become as large as it originally was, resulting in a new generation.

2.2.1 Simulating a game

By calling *Agent* with some command-line options, it simulates a game between two profiles (or organisms). Specifically, passing a random seed (used for the opening book and Zobrist hash keys), a maximum memory parameter, a time factor parameter and a minimum depth parameter along with values corresponding to the two profiles to simulate makes *Agent* simulate a game between these two profiles (the first profile given plays as white). Importantly, the value of a pawn is always fixed to 1000, to provide a frame of reference.

The time factor is a number used to determine how long Agent may spend calculating each move. Agent maintains time by using a precomputed array containing how much time most high-level games tend to be spent on each move versus the number of moves into the game (see Figure 2.8). This number is then multiplied by the time factor to get the maximum time allowed for each move. A time factor of 45 ms leads to a game of length usually less than one minute. The minimum depth is a number to which Agent will always search for the current move, which overrides any time settings. This is used to make *Agent* search to a reasonable depth even when the time factor is small.

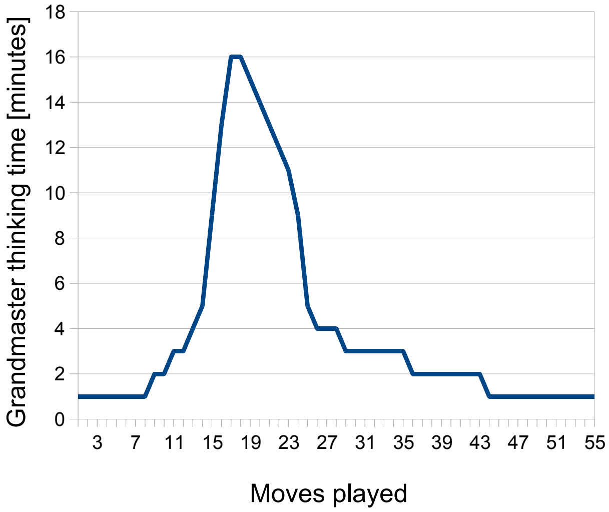


Figure 2.8. An approximation of grandmaster thinking time versus moves played (reproduced from Hyatt, 1984)

After a simulation is complete, *Agent* will output four numbers corresponding to scores given to the involved profiles, along with the entire game in a textual format (the latter is only of interest for debugging purposes). The scores can then be used to determine fitness.

2.2.2 Scoring & natural selection

Agent returns a number of scores for a simulated game. A win gives more than a draw, which gives more than a loss. To minimize randomness in the results, the first two scores are adjusted based on how quickly the game was won/lost (a quick win for instance would indicate a stronger winning player than a drawn out win after a long endgame). The two other scores are equivalent to traditional chess scores, but doubled as to avoid using floating point numbers: 2 points for a win, 1 point for a draw, and 0 points for a loss. Each profile (or organism) maintains a sum of its scores to which each individual game result is added.

After all computed matchups and games has been played out, the population is sorted according to this sum of scores, and a percentage of the worst are eliminated, leaving the top percentage of profiles as survivors which become parents for the new generation.

2.2.3 Reproduction & variation

During reproduction, two parents are chosen randomly from among the survivors for each child. Each child is produced by letting it inherit half of its profile parameters from each parent. Which parameter comes from which parent is random, but the child always gets precisely half from each. The process is repeated until the population becomes a desired size (usually the size of the previous population).

Finally there is a chance for each parameter to mutate (change randomly). This is a core mechanism in genetic algorithms, since it maintains diversity in a population that could otherwise become too homogeneous to yield any new solutions. Each child has a chance to mutate: if it does, two events take place with equal probability. Half of the time, each parameter of the child gets this same chance to mutate. The other half of the time, only one parameter of the child mutates. A mutation means that a parameter is changed by a random term, which is never larger than a certain factor times the parameter's maximum interval. In addition, the result never exceeds this maximum interval.

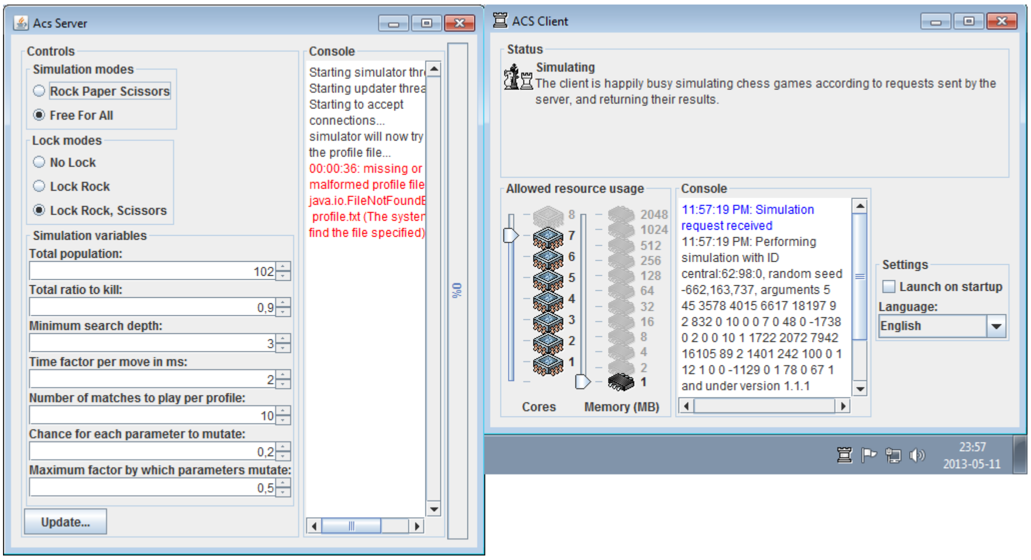
This whole process is repeated until a full new generation has been created, at which point the the simulation step is repeated, this time with a new (and most likely stronger) generation.

2.2.4 Distributed computing setup

As mentioned in Chapter 1, evaluating the fitness function of a genetic algorithm is typically a simulation's most time-consuming part. The simulation of a chess game is no different. Naturally, this means pursuing opportunities to parallelize the simulation process becomes a high priority.

Since much of the simulation process (Section 2.1.1) could be parallelized, a simulation client was developed in Java that could simulate a chess game on behalf of a single server (which was also written in Java) that distributes simulation jobs. It is the server that executes the genetic algorithm described above, but its clients are the ones evaluating the fitness function (simulating the games). These jobs precisely correspond to the matchups the genetic algorithm computes (see Section 2.2).

The client uses calls to Agent to perform the simulations. By using threads, multiple calls in parallel are possible. The client includes functionality to update the Agent program, report simulation results, adjust simulation timing variables per request as well throttle as the number of cores and memory it may use, along with other miscellaneous features. Much effort went into properly synchronizing the code. Communication happens via the TCP/IP protocol. The client totals 4,200 lines of code, and its interface may be seen in Figure 2.9.



Simulation modes
○ Rock Paper Scissors
● Free For All

Lock modes
○ No Lock
○ Lock Rock
● Lock Rock, Scissors

Simulation variables
Total population: 102
Total ratio to kill: 0,9
Minimum search depth: 3
Time factor per move in ms: 2
Number of matches to play per profile: 10
Chance for each parameter to mutate: 0,2
Maximum factor by which parameters mutate: 0,5

Console
```
Starting simulator thr
Starting updater threa
Starting to accept
connections...
simulator will now try
the profile file...
00:00:36: missing or
malformed profile file
java.io.FileNotFoundE
 profile.txt (The syster
find the file specified)
```

Status
Simulating
The client is happily busy simulating chess games according to requests sent by the server, and returning their results.

Console
```
11:57:19 PM: Simulation
request received
11:57:19 PM: Performing
simulation with ID
central:62:98:0, random seed
-662,163,737, arguments 5
45 3578 4015 6617 18197 9
2 832 0 10 0 0 7 0 48 0 -1738
0 2 0 0 10 1 1722 2072 7942
16105 89 2 1401 242 100 0 1
12 1 0 0 -1129 0 1 78 0 67 1
and under version 1.1.1
```

Settings
□ Launch on startup
Language: English

Figure 2.9. The interface for the server (left) and client (right) which mutually run the genetic algorithm.

During the simulation stage, the server distributes calculated matchup jobs to any clients connected over a network. It also collects statistics and carefully notes results of its many tasks. Its interface includes controls to adjust some of the simulation parameters described above. Like the client, much work was put into properly synchronizing the server's code, in particular to ensure correctness at all times. The server totals 3,400 lines of code, and its interface may be seen in Figure 2.9.

2.3 Performing the simulations

Once the simulations were set up, they could be carried out proper. The details are specified below.

2.3.1 *Free-for-all* and *rock-paper-scissors* strategies

Two simulation strategies were used. In *free-for-all* (*FFA*), the genetic algorithm described above was run on a single population, and competition was restricted to taking place inside the population alone.

In *rock-paper-scissors* (*RPS*), three (appropriately-named) populations were used. These were designed in mind to try to specialize each in defeating its neighbour (hence the name). Points were only awarded for succeeding to defeat a population, and not for losing; after all, scissors should not learn to defend against rock, but rather how to defeat paper. The hope was to see if what may be called *triangular specialization* would occur.

To help RPS along, a mechanism was designed by which populations could be "locked," which restricted their development. At the start, for instance, both rock and scissors were locked. This was thought so that paper could properly specialize against rock. Next, only rock was locked, so that scissors would specialize against paper. Finally, none were locked to stabilize the development.

2.3.2 Simulation runs

Excluding debugging cycles, three simulation cycles were run, whereof two were of type FFA and one was of type RPS.

The first FFA cycle was run with a total population of 102,[6] and 90% of profiles were eliminated before reproduction in each cycle. There were 10 matchups per profile, meaning that each profile was matched with 10 other profiles (for a grand total of 1020 matchups to simulate per generation). The mutation chance was 20%, and the mutation range (described above) was ±50%. Each move had a "minimum depth" of 5 ply, and a time factor of 45. The run was terminated after 124 generations.

The second run was of type RPS with a population of 102, elimination ratio of 90%, 10 matchups per profile, 20% mutation chance and 50% mutation range. Both rock and scissors were "locked" until generation 20, after which scissors was locked until generation 40, after which none were locked. The run was terminated after 57 generations.

The third run was of type FFA and was changed so that the fitness was only dependent on the traditional chess score (that is, unlike the first FFA run, winning the game faster had no impact on fitness evaluation). It had the same parameters as the first, except that after 28 generations, the mutation chance was increased to 60% for about 17 generations, and then further to 80% for 45 generations, then back to 20%.[7] The run also had a slightly different profile setup: in particular, the maximum intervals were extended and always positive. Finally, because of these new

[6] This strange number was chosen because it was close to 100 (a reasonable population) as well as being divisible with 3 and 2, for later running with RPS.

[7] By taking a somewhat questionable divine role, this was done in hope to resolve what was seen as an abnormally high queen material value.

intervals, the mutation range was also decreased to 35%. The run was terminated after 108 generations.

2.3.3 Testing the simulation results

To test the end results (that is, the final generations), a modified version of the server was used to compute matchups for a target versus all generations (approximately 100 matchups versus random profiles from a generation), and summing up the scores for an approximate win rate. In order to get the dominant parameters in a generation, an *archetype* could be created by taking the most common parameters (the median) for some generation. For the FFA runs, matchups were performed against all previous generations, as well as matching each generation against specific pre-designed profiles. These included a "random" profile (one with random parameters), a "greedy" profile (one that only considers material), and our default profile.

For the RPS run, instead, the latest rock/paper/scissors populations were matched against their neighbours in matchups and summed their scores, to see if they had a strong win rate against their neighbours.

Chapter 3

Results

Below follows the results from the testing and from the actual simulations. Recall that each profile always evaluates pawns as being worth 1000 points, to provide a frame of reference.

The meaning of the headers of the tables can be inferred by the list of considered factors in Section 2.1, save for "exchanges", whose use was disabled in each run (so its value is irrelevant). The values may be compared to *Agent's* default profile, which is presented in Table 3.1. A "greedy" profile was also created, whose material values match that of the default profile, but which is otherwise 0. (This emulates a player who only cares about material).

The vertical axis of each diagram (win rate) is actually the sum of scores of all matchups divided by the highest possible score. This means draws also influence this rate. The curve which profiles the data points has been "smoothed" using B-spline interpolation with a data point range of 10.

Knight value	**Bishop value**	**Rook value**	**Queen value**
3000	3200	5000	9000

Knight center	**Knight growth**	**Has castled**	**Castle penalty**
9	linear	700	350

Knight outpost	**Rook outpost**	**Outpost growth**	**Bishop principals**
10	6	linear	7

Bishop growth	**Rook open file**	**Exchanges**	**Capture threshold**
linear	53	14	0

Pawn structure	**Pawn growth**	**Closed knight**	**Open bishop**
5	unbiased	10	10

King center	**King growth**		
3	linear		

Table 3.1. Parameter values for the default profile.

3.1 First *FFA* run

Table 3.2 presents the parameter values for the archetype of the last (124th) generation.

Knight value	**Bishop value**	**Rook value**	**Queen value**
3194	3204	5738	10714
Knight center	**Knight growth**	**Has castled**	**Castle penalty**
6	linear	1197	627
Knight outpost	**Rook outpost**	**Outpost growth**	**Bishop principals**
-10	-10	cubic	28
Bishop growth	**Rook open file**	**Exchanges**	**Capture threshold**
quadratic	53	0	-1938
Pawn structure	**Pawn growth**	**Closed knight**	**Open bishop**
5	loves connected	-10	-10
King center	**King growth**		
15	quadratic		

Table 3.2. Parameter values for the archetype of generation 124 for the first FFA run.

The performance of this archetype versus previous generations is shown in Figure 3.1. This win rate was based on 100 matchups. Similarly, shown in Figure 3.2, 3.3 and 3.4 are the win rates versus the greedy profile, *Agent's* default profile, and a random profile.

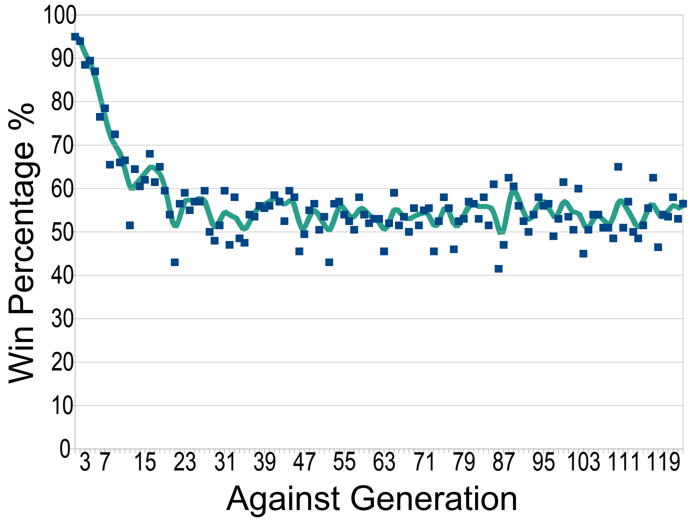


Figure 3.1. The win rate, based on 100 matchups, of the 124th generation versus earlier generations.

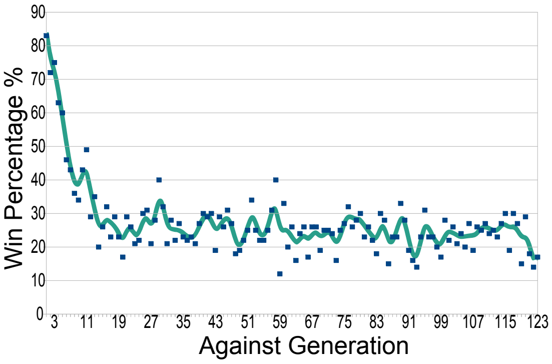


Figure 3.2. The win rate, based on 50 matchups, of the greedy profile versus all generations.

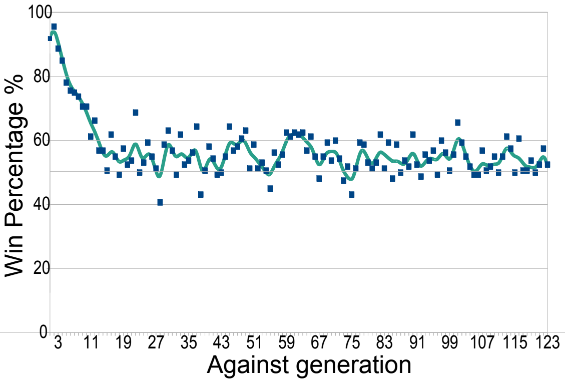


Figure 3.3. The win rate, based on 80 matchups, of the default profile versus all generations.

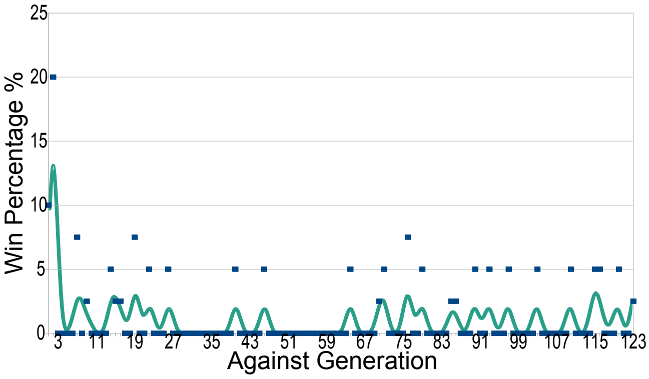


Figure 3.4. The win rate, based on 40 matchups, of a randomized profile versus all generations.

Finally, the archetype's performance versus the default profile in 1000 matchups was measured to 49%.

3.2 *RPS* run

Tables 3.4, 3.5 and 3.6 present the parameter values for the archetypes of "rock", "paper" and "scissors" respectively of the last (57th) generation. After these archetypes were computed, they were run against their respective neighbors in 100 matchups each. The results are presented in Table 3.3.

Matchup	Win rate
rock vs. scissors	54%
scissors vs. paper	47%
paper vs. rock	43%

Table 3.3. Win rates for rock-paper-scissors matchups.

Knight value	**Bishop value**	**Rook value**	**Queen value**
3599	4508	5686	11501
Knight center	**Knight growth**	**Has castled**	**Castle penalty**
0	linear	789	350
Knight outpost	**Rook outpost**	**Outpost growth**	**Bishop principals**
189	0	square root	28
Bishop growth	**Rook open file**	**Exchanges**	**Capture threshold**
square root	53	372	-1938
Pawn structure	**Pawn growth**	**Closed knight**	**Open bishop**
0	unbiased	366	0
King center	**King growth**		
11	linear		

Table 3.4. Parameter values for the "rock" archetype of generation 57.

Knight value	Bishop value	Rook value	Queen value
1861	3200	5148	10144

Knight center	Knight growth	Has castled	Castle penalty
6	quadratic	2219	350

Knight outpost	Rook outpost	Outpost growth	Bishop principals
73	0	square root	0

Bishop growth	Rook open file	Exchanges	Capture threshold
linear	53	222	-1013

Pawn structure	Pawn growth	Closed knight	Open bishop
0	loves connected	0	0

King center	King growth		
11	linear		

Table 3.5. Parameter values for the "scissors" archetype of generation 57.

Knight value	Bishop value	Rook value	Queen value
2208	4194	6165	11236

Knight center	Knight growth	Has castled	Castle penalty
39	linear	337	0

Knight outpost	Rook outpost	Outpost growth	Bishop principals
37	0	square root	3

Bishop growth	Rook open file	Exchanges	Capture threshold
square root	212	0	-1938

Pawn structure	Pawn growth	Closed knight	Open bishop
5	hates doubled	0	10

King center	King growth		
15	sigmoid		

Table 3.6. Parameter values for the "paper" archetype of generation 57.

3.3 Second *FFA* run

Table 3.7 presents the parameter values for the archetype of the last (108th) generation.

Knight value	**Bishop value**	**Rook value**	**Queen value**
3578	4015	6617	18197
Knight center	**Knight growth**	**Has castled**	**Castle penalty**
9	cubic	832	0
Knight outpost	**Rook outpost**	**Outpost growth**	**Bishop principals**
10	0	linear	7
Bishop growth	**Rook open file**	**Exchanges**	**Capture threshold**
linear	48	0	-1738
Pawn structure	**Pawn growth**	**Closed knight**	**Open bishop**
0	loves connected	0	0
King center	**King growth**		
10	quadratic		

Table 3.7. Parameter values for the archetype of generation 108 for the second FFA run.

The performance of this archetype versus previous generations is shown in Figure 3.5. This win rate was based on 100 matchups. Similarly, shown in Figure 3.6, 3.7 and 3.8 are the win rates versus the greedy profile, *Agent's* default profile, and a random profile.

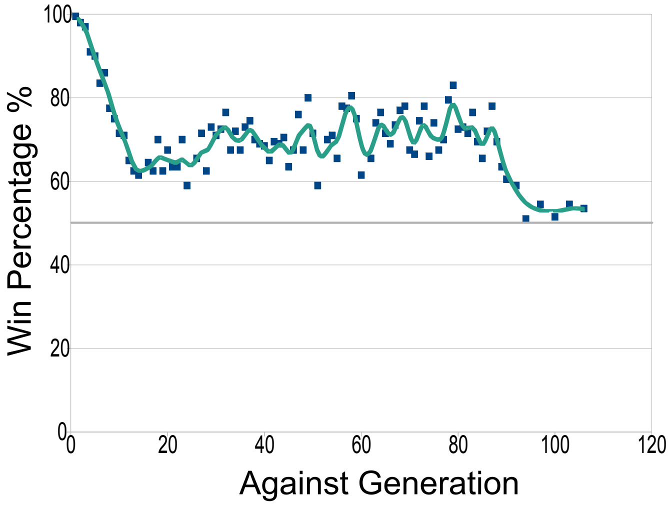


Figure 3.5. The win rate, based on 100 matchups, of the 108th generation versus earlier generations.

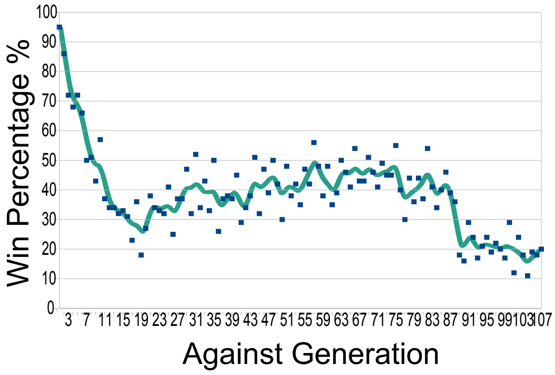


Figure 3.6. The win rate, based on 50 matchups, of the greedy profile versus all generations.

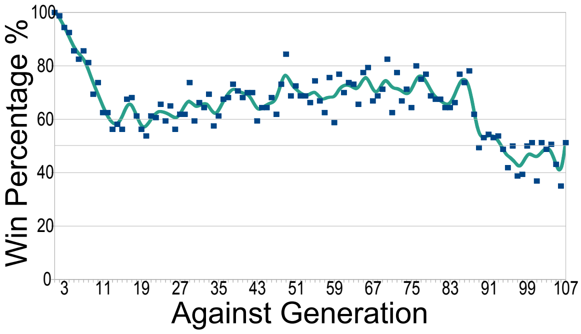


Figure 3.7. The win rate, based on 80 matchups, of the default profile versus all generations.

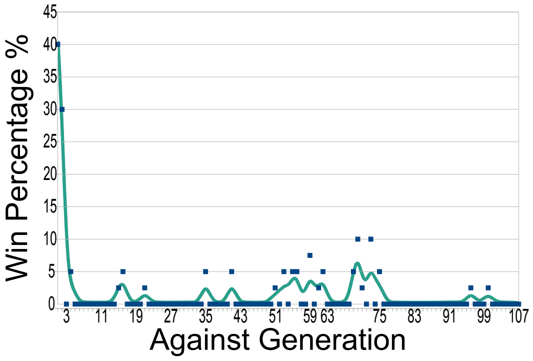


Figure 3.8. The win rate, based on 40 matchups, of a randomized profile versus all generations.

Finally, the archetype's performance versus the default profile in 1000 matchups was measured to 52%.

3.4 Statistics

In total, there were approximately 450,000 games played by the clients, for both simulation and testing purposes. The simulation time (with parallelization considered) amounted to 323 days. Representing all these chess games in plain text requires approximately 400 MB of memory.

Chapter 4

Discussion

This chapter presents an analysis of the results. They can be analyzed in respect to the feasibility of genetic algorithms, and the game of chess in general.

4.1 Analysis

The most accessible trend of all considered diagrams is that the win rate rapidly drops in the first few generations. Most considered archetypes enjoy a 90%+ win rate against the first, mostly random generation, but this quickly changes. Each considered archetype (latest, greedy, etc) starts becoming less and less effective as the program learns and evolves. This clearly indicates that learning is taking place, and quite rapidly so, as the program becomes more difficult to defeat. In addition, no archetype manages to win more than 50% of the time against the latest generation with any reasonable statistical significance. This evidence clearly supports this paper's thesis.

The rate at which the program adopts reasonable parameters (that is, parameters that guarantee approximately 50% wins) was much greater than expected. In all cases, within the first 15-20 generations, the program is already reasonably strong. This decline is gradual, as well, just as evolution is in nature: it can be clearly seen that this change does not happen immediately.

All tests of the second FFA generation also show a curious "notch" at generation 90, along with a seemingly strange decrease of program strength in the middle generations. The reason behind these will be analyzed and explained in Section 4.1.2.

It is also clear, however, judging from both the appearance of each generation, and the win rate of the diagrams in the first FFA run, that this learning process stagnates quickly. This is likely because the algorithm has found a *local maximum*: a point where most simple mutations and recombinations seem to generate less fit offspring. There may be a *global maximum*, the most fit version of *Agent*, but finding it from a local maximum may be very difficult. This maximum makes the population settle, and the time until a superior recombination or mutation comes along may be very long. This process is also seen in nature when large-scale environmental changes are seldom.

It must be emphasized, however, that genetic algorithms are much more limited than the processes that happen in nature, restricting us from speaking too much of any likenesses. In particular, nature is much more complex, and even the simplest organism's genome is much more complicated than our simple chess profiles, for instance.

4.1.1 Parameter analysis

Overall, both FFA runs yielded material values that are close to the expected, traditional values. The first FFA run, in particular, differed from the traditional values by only 6%, 7%, 15% and 19% for knight, bishop, rook and queen respectively.

One of the most surprising results is the pervasive choice of a rather large negative value for discarding captures, which all populations ended up with. For example, the first FFA run yielded an archetype which discards captures worse than losing approximately two pawns. This effectively means that the genetic algorithm has decided that programs which sacrifice some search depth by also considering losing captures in the quiescence search (and a changed ordering during normal search) are more fit, and hence better.

This is a profound statement. It means the program has taught itself that reasonable sacrifices in chess can be very important to consider, even at the leaf nodes during quiescence search. Indeed, in most cases in chess, a sacrifice of up to a minor piece can be very sound if it gains the player an advantage otherwise, which *Agent* seems to have learned.

Other notable factors include a much heavier emphasis on having a centralized king in the endgame, performing castling, and on having bishops on the principal diagonals than was expected. The first is particularly interesting, as it can probably be attributed to the tendency towards having drawn-out endgames. In addition, the material value of the queen was very emphasized, especially in the second FFA run. More generally, both runs had high emphasis on non-pawn pieces, which can probably be attributed to *Agent's* comparatively weak understanding of pawns. In particular, with no "passed pawn" bonus, this emphasis does not come across as surprising.

Something particularly notable in both runs, but in particular the first, was the appearance of very low, even negative, scores on certain strategic elements, such as open-game bishops and outposts, which is unfortunate. This may indicate a limitation of genetic algorithms, as such strategic elements are quite subtle, unlike material and mobility.

4.1.2 Performance analysis

While the program evidently improved itself against all its considered opponents, the performance differences are of considerable interest. In particular, while the greedy profile started out by winning over 90% of games, it quickly lost ground to the improving program, ending up winning only *20-25%* of games. This means that the evolved versions of *FFA* had a much better grasp of chess than a simple greedy version. In particular, this proves that an overly simple theoretically deduced profile cannot hope to compete with a profile that has had the chance to evolve.

The results against the random profile, however, do not come across as surprising. While showing some ability against the first generation or so, the random profile very, very quickly degrades to winning almost 0% of the time. In fact, the only time it wins or draws later is likely against mutated opponents in generations. It is neither very surprising that the latest generation is stronger, or at least as strong, as previous generations, as this is what the theory behind genetic algorithms predicts; a progressive refinement. This serves as further evidence to our thesis as it indicates a progressively stronger program.

However, the results of the RPS run are disappointing. With win rates that are no different from 50% with any statistical significance, there appears to have been no triangular specialization, despite very varied final populations for "rock", "paper" and "scissors". While these results do not prove there is little room for triangular specialization in chess, they do show that there may be difficulties in constructing a learning mechanism by which they can be found.

Of considerable interest is the sudden increase in program strength at generation 90 of the second FFA run, as well as the decreasing strength in the middle. The explanation, however, is likely simple. At generation 28, the mutation chance was increased significantly, and even further at generation 45. This probably causes the poor middle performance, as the profiles are more likely to be matched versus a mutated profile in the generation, increasing their chance of victory. At generation 90, when the mutation is reset to 20%, the profile immediately strengthens.

This deviation indicates a *very* important point to consider in the results. Because approximately 20% of each generation is mutated in both FFA runs, the archetypal, fixed profiles run against them automatically have an advantage over many such mutated profiles. This means the results are biased in the archetype's favor, meaning that each generation's strength appears lower than it really is. The alternative approach, however, which would be to calculate a single archetype of each generation, is more problematic, because it introduces artificial strength through the *law of large numbers*. That is, a collective archetype[1] of some generation will very likely be stronger or at least as strong as any single profile in the same generation. Hence, this bias is important to consider in all results, especially the ones where the opponent is our default profile.

Considering this bias when looking at the results versus our default profile is important. In both FFA runs, the diagrams indicate an approximate 50% win chance against the default profile (or slightly higher in the case of the second FFA run). While this seems to indicate that the evolved program is only equal in strength

[1] As before, taking the median of each parameter across the population to create a collective archetype (as opposed to choosing a random profile from the population)

to the default profile, the aforementioned bias must be considered as it may influence this result. However, the tests in which 1000 specific matchups were performed of the individual archetypes versus the default profile indicate that the strength is approximately equal. In any case, the evolved programs are at least as effective as our default profile, and perhaps better, indicating that genetic algorithms are a very feasible choice of optimization algorithm, even for a game such as chess.

4.2 Error & stability analysis

The outcome of a single game of chess is usually a poor indicator of the players' individual strengths. Hence, a sum of averages is required to attain any useful information, which leads to errors. These errors are no different in this case. The standard deviation, defined as

$$s_N = \sqrt{\frac{1}{N}\sum_{i=1} N(x_i - \bar{x})^2}$$

can be used as an estimate of data precision. In this case, for the settled data points of the first FFA run, the standard deviation was calculated to

$$s_N = 7,22\%$$

where % indicates procentual units. This means that one can only really speak of statistical significance of approximately 7 procentual units. In other words, a win rate of 43% may not be statistically different to a win rate of 50%. This deviation, however, is still quite low, and it does not influence any ability to make a qualitative statement of strengths particularly much.

4.3 Conclusions

This paper conclusively proves that genetic algorithms are very well suited to complex problems such as playing chess. While evaluating a suitable fitness function for chess is particularly computationally intensive, it is still a feasible way to improve a program's playing strength. This paper also proves that genetic algorithms help to highlight potential implementation considerations, as it did in this case by showing that a low capture threshold leads to better play than a high one.

Some problems with genetic algorithms are also highlighted in this paper. In particular, many genetic algorithms which can be feasibly run with much bigger populations and for much longer, can be used to capitalize on speciation phenomena. This phenomena does not really appear in this application, as the population settles relatively early. Hence, not all applications can enjoy certain benefits of genetic algorithms, because of time and computation limitations.

Finally, we turn back to the question of machine learning. Sadly, the results of this paper are not exempt from the *AI effect*, which occurs when a problem has been solved by an artificial intelligence whereupon many onlookers discount the AI with statements such as "that's just computation". In addition, genetic algorithms do not exactly replicate any known processes in the human brain. This paper, however, tries to resolutely oppose this view. Without any significant human intervention, the program, when viewed from a perspective ignorant of its workings, *must* be said to have learned. While clearly not the best chess program available, *Agent* has certainly improved itself and acquired its skills in a fashion that is hard for even humans to fully grasp.

Appendix A

Access to *Agent* & surrounding software

Details for accessing and downloading *Agent* and surrounding content is given below.

A.1 *Agent* program for Windows

An executable version of *Agent* for Windows, which can be run to play against different profiles of *Agent* can be downloaded at: `http://host-a.net/u/Schreib/agent.zip`. This version has been slightly improved over the one used in this paper, in that *Agent* now also recognizes a penalty for early queen development, and a bonus for passed pawns. The file "openingbook.obf" must be placed in the same directory as "chessagent.exe".

A.2 *Agent* source code

The C source code for the version of *Agent* given in Section A.1 can be downloaded at: `http://host-a.net/u/Schreib/Chess_Dev_VI.zip`

A.3 Simulation client

The simulation client for performing simulations on behalf of a server may be downloaded at: `http://host-a.net/u/Schreib/acsclient.jar`. This download includes the Java source files inside the .jar file as well as being independently runnable.

A.4 Simulation server

The basic simulation server for running our genetic algorithms may be downloaded at: `http://host-a.net/u/Schreib/acsserver.zip`. This download includes the Java source files inside the .jar file as well as being independently runnable. Basic generation files are also included.

Appendix B

Acknowledgements

We would like to thank our supervisor Johan Boye for his guidance through this project. We would also like to thank everyone who helped us simulate, without which this project would not have been possible. Statistics from the genetic algorithm runs are presented in Table B.1 excluding the authors' runs. Finally, we would like to thank our friends and family for their encouragement and support.

Simulator	Games simulated
Ludvig Bramstång	50,025
Calle Svensson	33,211
CSC computers	24,157
Allan Mossiaguine	10,377
Johannes Wennberg	5,233
Ramona Mayer	2,730
Niclas Höglund	912

Table B.1. Table of simulation partners and the number of games simulated by each during the genetic algorithm runs.

Bibliography

[1] Adelson-Velsky, G. M., Arlazarov, V. L., Bitman A.R., Zhivotovsky, A.A and Uskov, A. V, 1970. Programming a computer to play chess. *Russian Mathematical Surveys*, 25(2).

[2] Akl, S. Newborn, M., 1977. The Principal Continuation and the Killer Heuristic. *ACM Annual Conference Proceedings*, pp. 466-473. ACM, Seattle, WA.

[3] Fogel, D. B., Hays T. J., Hahn, S.L. and Quon, J, 2004. A self-learning evolutionary chess program. *Proceedings of the IEEE*, 32(12), [online] Available at: <`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.72.4267`>

[4] Fraser, A. and Burnell, D., 1970. *Computer Models in Genetics*, New York: McGraw-Hill.

[5] de Groot, A., 1965. *Thought and Choice in Chess.* Mouton & Co Publishers, The Hague, The Netherlands. Second edition 1978.

[6] Harris, L., 1975. The Heuristic Search And The Game Of Chess - A Study Of Quiescence, Sacrifices, And Plan Oriented Play. *IJCAI Tbilisi*, Georgia, pp. 334-339.

[7] Hyatt, R. M., 1984. Using time wisely. *ICCA Journal*, 7(1).

[8] Hyatt, R. M. and Cozzie, A. E., 2005. The effect of hash signature collisions in a chess program. *Journal of The International Computer Games Association*, 28(3), pp. 131-139, [online] Available at: <`http://www.cis.uab.edu/info/faculty/hyatt/collisions.html`>

[9] Knuth, D. E. and Moore, R. W., 1975. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4), pp. 293-326.

[10] Lefler, M. et al. Chess Programming Wiki. [online] Available at: <`http://chessprogramming.wikispaces.com/`>

[11] Marsland, T. and Campbell, M., 1982. Parallel Search of Strongly Ordered Game Trees. *ACM Computer Survey*, 14(4), pp. 533-551, [online] Available at: <`http://webdocs.cs.ualberta.ca/~tony/OldPapers/strong.pdf`>

[12] Pollock, N. Norm's chess downloads. [online] Available at: <http://www.hoflink.com/~npollock/chess.html> last updated 13th of April,

[13] Reinefeld, A., 1983. An improvement to the Scout tree search algorithm. *ICCA Journal*, 6(4), [online] Available at: http://www.top-5000.nl/ps/An%20improvement%20to%20the%20scout%20tree%20search%20algorithm.pdf

[14] Shannon, C., 1950. Programming a Computer for Playing Chess. *Philosophical Magazine* 41(314).

[15] Spracklen, D. and K., 1978a. First steps in Computer Chess Programming. *Byte Publications Inc*, [online] Available at: <http://archive.computerhistory.org/projects/chess/related_materials/text/4-4.First_Steps.Byte_Magazine/First_Steps_in_Computer_Chess_Programing.Spracklen-Dan_Kathe.Byte_Magazine.Oct-1978.062303035.sm.pdf>

[16] Spracklen, D. and K., 1978b. An Exchange Evaluator for Computer Chess. *Byte Publications Inc*, 3(11).

[17] Warnock, T. and Wendroff, B., 1988. Search tables in computer chess. *ICCA Journal*, 11(1), pp. 10-13.

[18] Zobrist, A. L., 1969. A new hashing method with application for game playing. Technical Report 88, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, [online] Available at: <http://research.cs.wisc.edu/techreports/1970/TR88.pdf>

Glossary

AI effect an effect that occurs when a problem has been solved by an artificial intelligence, whereupon many onlookers discount the AI arguing it is not really thinking

alpha-beta pruning a pruning technique applied to *minimax* searches to vastly decrease the needed search space by stopping the search of a position when it can be proved that a player would not put herself in the position because a better alternative exists elsewhere in the game tree

archetype a chess profile that represents some population as a whole; computed by taking the median of each set of parameters

bitboard a type of data structure representing a 64-square game position in which 64-bit numbers record different types of states for each square on the board

evaluation see static evaluation

evolutionary programming a set of programming paradigms that try to imitate evolutionary processes in nature

FFA see free-for-all

fitness function a core component in genetic algorithms; a function which determines the fitness of an individual in a population

free-for-all a genetic algorithm simulation strategy in which a single population is used and competition is restricted to happening inside it

genome a string of data representing the genetic coding for an individual

horizon effect a phenomenon in which a game playing program selects suboptimal moves because it cannot see crucial events past its search depth horizon

iterative deepening a search strategy in which the planned search depth is successively incremented; used to speed up an *alpha-beta pruned* search using *move ordering*

killer heuristics a type of *move ordering* in which moves that caused an *alpha-beta pruned* search to stop searching early in some other section of the game tree are tried first; the expectation is that they too will cause the search to stop earlier at the current position

law of large numbers a mathematical predication stating that the average value of some data set goes to its expected value as the data set grows in size

mailbox a type of data structure representing a game position in which each square on the game position board is mapped to the piece standing on it

minimax a recursive algorithm for finding the best move in a game; its basic assumption is that both players play optimally in each position

move generation a component of a game playing program responsible for generating all legal moves from a given position

move ordering an attempt to speed up an *alpha-beta pruned* search by searching the expected best moves first

natural selection a core component in the theory of evolution; a process in which only well-performing individuals are selected to survive in some context

negascout a *minimax* search with an improved *alpha-beta pruning* technique; it enables more pruning by searching supposed bad moves with the sole intent of proving they are worse than a given move

null window search a selective *minimax* search of moves not expected to be played; searched with the intent of proving this

opening book a precomputed source of moves game playing programs use during the opening of a game; used since such programs often play poorly in the opening

ply one half-move; for example, searching to a depth of 6 plies means looking forward 3 full moves (both players have made 3 moves)

principal variation search an exhaustive *minimax* search of the expected line of play for both players

quiescence search a reduced *minimax* search done at the deepest part of the search to mitigate the *horizon effect*; used to ensure the search stops in a quiet position

reproduction with variation a core component in the theory of evolution; a process in which individuals produce offspring with possibly varied characteristics

rock-paper-scissors a genetic algorithm simulation strategy in which three populations compete in order to achieve *triangular specialization*

RPS see rock-paper-scissors

search heuristic a method to search some structure by using educated guesses, rule of thumbs, etc.

SEE see static exchange evaluation

static evaluation a central strategy or algorithm of a game playing program which tries to estimate how much some player is ahead in some game position without considering moves ahead

static exchange evaluation an algorithm for determining the gain for the optimal sequence of capturing moves on a selected square

transposition tables data structures used to save the search scores of positions so that they can be used if the position occurs in a different section of the game tree

triangular specialization a situation in which three players are adept at defeating exactly one other player and poor at defeating exactly one other player

Turing Test a classical intelligence test conceived by famous computer scientist Alan Turing in which a machine may pass this test if it, by holding a text conversation with a human, cannot be distinguished from a second human doing the same

Zobrist hashing a method to compute a hash value of a game position by only considering the changes between two positions; named after Albert Zobrist