Table of contents

Introduction

This work presents a synthetic approach to chess programming. Certain aspects are explored more deeply, and subject to scientific research.

• *Aim of the work*

The main aim of this work was to create a chess-playing program of reasonable strength (comparable to a strong club player of international rating about 2100 Elo points). The program should use modern chess-programming methods and employ a choice of techniques leading to improvement of playing strength.

Using this program some experiments from the field of opening book learning were to be performed. Book-learning allows program to tune its opening book usage basing on already played games in order to avoid playing poor lines in future games. The purpose of these experiments was to estimate usefulness of book-learning for improving engine performance in games against opponents.

Supplementary aim of this work was to create an overview of common knowledge about modern chess programming, including its roots, scientific and engineering background as well as methods, algorithms and techniques used in modern chess programs.

• *Layout*

This work is divided into four main parts. The first one is an overview of chess programming, mainly devoted to its scientific aspect (complexity, artificial intelligence, game theory).

The second one is a review of algorithms, data structures and techniques commonly used in modern chess programs. The alpha-beta algorithm, and its enhancements are the core of it. The most important methods (especially those related to increasing performance of alpha-beta algorithm by improvement of move ordering) are also accompanied by some quantitative data, coming both from other sources and author's own experience with his chess program.

The next chapter is dedicated to implementation of the chess program, *Nesik*, that was created as part of the work. The description includes the algorithms and techniques that are used in the program, when necessary with additional explanations over those given in the previous chapter. Results of experiments performed while creating the program,

being comparisons between different techniques and parameter values, are also supplied as part of this chapter.

In the last part there is presented data concerning experiments performed using the program described in the previous chapter. The experiments is from the field of book-learning, that is poorly described in existing sources. The data includes: purpose and description of the experiments, results, conclusions and plans for further research.

The source code and an executable for the chess program that accompanies this work is attached.

- ***Range of the work***

Since chess programming is a vast topic, some limitations on the scope of that work had to be made.

The most popular approach to chess programming, assuming usage of backward-pruning based search algorithm (alpha-beta or its derivatives) to determine best moves, is explored. The other approaches (like best-first search) are only mentioned. Parallel search algorithms were excluded from the scope of that work from the beginning.

The implementation of the chess engine was focused on ease of development, providing practical insight into behaviour of techniques described in the theoretical part, and easy modifiability for the purposes of experimentation. Strength of play was compromised, the ultimate level equal to that of a strong club player was assumed to be satisfactory.

The research part is devoted to improving quality of the opening book, part of the program that uses expert knowledge gathered by generations of human chess players to guide program's play in the initial stage of the game. One of possible solutions is described and analysed.

- ***Research problem***

The initial phase of the games of chess is quite well understood and analysed by humans. It is unlikely that current chess programs can improve significantly over this expert knowledge that is available now as a common heritage of many generations of chess players. Therefore, putting this knowledge into chess programs leads to a large improvement in their performance.

The opening book can be created manually, by adding good variations that conform to the program's playing style, but it is a very time consuming and tedious work. Most

chess programmers use another solution: take a database with many hundreds of thousands of games played by strong players, and include all lines from all games up to a certain depth in the book. This approach, although very simple and fast, has at least two big disadvantages. The first one is that such a book can contain blunders, leading to a bad or even lost position for the engine right out of the book. Another, more subtle but still a big problem is that the book may contain lines that lead to positions not correctly understood by the program – as a result, the engine may not find the correct plan, and after several unskilful moves enter a significantly inferior position.

These, and others as well, problems can be addressed to some extent by using some kind of book-learning algorithm. Book learning algorithms are especially important in case of automatically generated books. They use the feedback from the engine (like positional evaluation when out of the book, result of the game) to update information about particular moves in the database – mark them as good, interesting, bad, not usable etc. A good book learning algorithm should make program avoid inferior lines, and play profitable variations more often.

The research part focuses on gathering quantitative data about improvement of strength of play when using certain kind of book-learning algorithm. The algorithm is a variation of the solution developed by dr Hyatt ([Hyatt, 2003], author of an ex-World Champion chess program named *Cray Blitz*) for his current engine *Crafty*. Since there are only a few works devoted to book-learning in chess that are known to author, and none of them provides quantitative data, this work can be one of the first to provide this kind of data.

- ***Research questions***

The aim of the research part of this book was to verify whether the described book-learning algorithm increases the overall playing strength of the program. The properly designed and implemented solution should help the program achieve better results. However, since the books that are used by modern chess programs are huge (often contain number of positions greater than 10^5), the learning can take very long time. Feasibility of book-learning in time-constrained environment is under consideration as well.

- ***Methods and tools***

The methods used during creation of this work include:

- Literature research,

- Parameter-driven testing,
- Tournaments,
- Rating comparison, including statistical significance analysis (using ELOStat package).

For the purpose of this work a chess engine has been created, using Borland C++ Builder package. C++ Builder provides a convenient Integrated Development Environment, that provides edition, build, run and debug capabilities within a single program.

The chess program is a console application, but knows the Winboard protocol, version 2, for communicating with universal GUIs providing chessboard interface to the final user.

Profiling tool (AQTime, evaluation version) was used to help eliminating program's performance bottlenecks. Some other free chess programs were used for reference and comparison purposes. The tournaments were performed using Arena's (http://www.playwitharena.com) built-in capabilities. For estimating program's strengths and weaknesses there were standard test suites used that contained hundreds of test positions with solutions (for example, Win At Chess test suite for estimating tactical awareness of the program).

1 Overall look on chess programming

Chess programming in its modern form has been around since 1950's – soon after first electronic computer was built. Since then it has significantly improved, allowing modern chess machines and programs to achieve grandmaster strength of play. General aspects of this discipline are described in this chapter.

1.1 The very beginning – why bother with chess programming?

The idea of creating a chess-playing machine has existed among humans since 18th century. The first well-described attempt on creating such a device was a cheat – inside the famous von Kempelen's machine there was hidden a dwarf that operated the pieces on the chessboard. The dwarf was an excellent chess player, so the machine made a big impression with its play on its tour throughout Europe. One day it even played Napoleon. The secret of Kempelen's machine was revealed in one of essays by Edgar Allan Poe.

The first successful and honest approach was made in the beginning of the previous century - Torres y Quevedo constructed a device that could play and win an endgame of king and rook against king. The algorithm was constructed in 1890, and the later version of the machine was shown on Paris World's Fair in 1914.

However, the first who described an idea of creating a complete chess program and laid some theoretical foundations was Claude Shannon [Shannon, 1950].

Shannon considered chess as a nice, although demanding, test field for developing concepts that today are generally assigned to the area of artificial intelligence, e.g.: operating on symbols rather than numbers only, need for making choices instead of following a well-defined line, working with solutions that can be valued not only as "good" or "bad", but also assigned some "quality" from a given range. Chess are concerned with all these points – they are considered to require some "thinking" for a reasonable level of play, and are not trivial. Moreover, they exhibit some properties that make them suitable for computer implementation: a game of chess always ends, is performed according to strictly defined rules and is not overly sophisticated. Discrete nature of chess also favours computer implementation.

Although development of a chess program might have seemed to be a waste of scarce computing power and skilled human potential at the time, Shannon realized that solutions developed at this "test field" may be further employed in many applications of somewhat similar nature and greater significance, like design of filters or switching

circuits, devices supporting strategic military decision making or processing symbolic mathematical expressions or even musical melodies, to name just a few.

Shannon in his work also created some basic framework and defined many concepts concerning computer chess implementation. Most of them remain proper and useful to date (like a concept of evaluation function), and in somewhat improved variants are widely used.

The idea of using chess as a training ground for new computer science concepts caught on, and was developed by many scientists, engineers and hobbyists. Nowadays, chess programs have left laboratories and workshops. They are widely commercially available and, thanks to their maximum playing strength equal to that of grandmasters ([Levy, Newborn, 1991], p. 5), are used as sparring partners or game analysers by millions of chess players and trainers worldwide.

1.2 A brief look at the game of chess from a scientific perspective

Since this work is related to programming chess, the theoretical considerations about game of chess are covered only to the extent required to explain certain mechanisms of chess programming.

1.2.1 Game theory

In some cases, it is suitable to talk about chess in terms of games theory, as it allows a clear and unambiguous message. Several definitions that are relevant to chess are given below (after [Turocy, von Stengel, 2001, p.2])

Definition 1.

Game is a formal description of a strategic situation.

Definition 2.

Player is an agent who makes decisions in the game.

Definition 3.

Payoff is a number, also called utility, that reflects the desirability of an outcome to a player, for whatever reason.

Definition 4.

Zero-sum game occurs if for any outcome, the sum of the payoffs to all players is zero. In a two-player zero-sum game, one player's gain is the other player's loss, so their interests are diametrically opposed.

Definition 5.

Perfect information - a game has perfect information when at any point in time only one player makes a move, and knows all the actions that have been made until then.

Definition 6.

Extensive game (or extensive form game) describes with a tree how a game is played. It depicts the order in which players make moves, and the information each player has at each decision point.

Definition 7.

Strategy is one of the given possible actions of a player. In an extensive game, a strategy is a complete plan of choices, one for each decision point of the player.

Definition 8.

Payoff function (payoff matrix) is a concept, with payoff each player will receive as the outcome of the game. The payoff for each player depends on the combined actions of other players.

Technically, matrix is used only when number of players equals two, and the payoff function can be represented as a matrix. Each row and each column of the matrix represents a strategy for player 1 or player 2 (correspondingly), and the payoff value is on the row/column intersection.

Example:

for the even-odds game:

		odd player	
		Head	Tail
Even player	Head	1-0	0-1
	Tail	0-1	1-0

Table 1. Payoff matrix for even-odds game

Initially, one player agrees to be an “even” player, and the other becomes “odd”. Then they simultaneously show a penny: either heads or tails up. If both reveal the same side of the coins (either both heads or both tails) then the “even” player wins (marked as 1-0 in the payoff matrix), otherwise “odd” is a winner (0-1).

Definition 9.

Rationality – property of a player that makes him play in a manner which attempts to maximize his own payoff. It is often assumed that the rationality of all players is common knowledge.

Definition 10.

Finite game allows for each player only a finite number of moves and a finite number of choices at each move.

Using the above terms, game of chess can be viewed as a **two-player zero-sum** game of **perfect information**.

- There are two players (commonly referred to as White and Black),
- Exactly three possible outcomes are valid: white wins-black loses, black wins-white loses or a draw. A matrix representation shows that it is a zero-sum game indeed:

		Black player		
		Win	Draw	Lose
White player	Win			1, -1
	Draw		0, 0	
	Lose	-1, 1		

Table 2. Payoff matrix for game of chess

Remark: In the above matrix win is assigned value of 1, draw is 0 and lose attains negative value of -1. However, it is a custom to assign (1, ½, 0) for win, draw and lose respectively. That difference is only in location and scale, so it has no meaning anyway.

- At each moment either only white or only black player is on move, and all information about state of game is known, that imply the **perfect information** property.

Game of chess is **finite**, as piece movement rules indicate a finite number of legal moves at any time, and the threefold repetition rule ensures its finite length (since number of legal chess positions is finite, after certain number of moves the same position must appear on the chessboard for the third time).

Game of chess can be expressed by a tree that has the starting position as the root, branches represent all legal moves and nodes correspond to chess positions. That would be

the **extensive form** of the game. The nodes of the tree indicate decision points for the player on move.

It is assumed that both players are **rational**, always choosing strategy that maximizes their payoff.

Theorem (Minimax theorem, proved by von Neumann in 1928) Every finite, zero-sum, two-person game has optimal mixed strategies.

Formally: let X, Y be (mixed) strategies for players A and B (respectively). Then:

$$\max_X \min_Y \mathbf{X}^{\mathrm{T}} \mathbf{A} \mathbf{Y} = \min_Y \max_X \mathbf{X}^{\mathrm{T}} \mathbf{A} \mathbf{Y} = v$$

Where: A – payoff matrix.

v – value of the game (minimax value)

X, Y are called solutions.

In above equation the expression on left-hand side indicates the outcome of player A (following strategy X), who tries to maximize his payoff, and the right-hand side represents the outcome of player B (strategy Y), who aims to minimize payoff of his opponent A (and therefore maximize his own outcome, as it is a zero-sum game). The theorem states that for each game satisfying the assumptions (finite, zero-sum, two-player) these two values are expected to be equal (if only both players behave rationally), and are referred to as “value of the game”.

If value of the game is equal to zero then the game is said to be fair, otherwise either game favours one player ($v > 0$) or another ($v < 0$). Since the solution for game of chess is not known (due to its enormous complexity), its value is also unknown. Therefore, nobody knows yet if game of chess is fair or not.

Mixed strategy is an entity that chooses between (pure) strategies at random in various proportions. However, in case of chess (game of perfect information) the minimax theorem holds also when only pure strategies are involved (i.e. probability of choosing one particular strategy is 1).

The process of determining the minimax value of the position that occurs in the game (establishing the minimax value of the game is equivalent to finding the minimax value of the starting position, or the root of game tree) is described in the part related to search module of chess programs.

1.2.2 Complexity

Chess are an extremely complex task from computational point of view. Most sources and encyclopaedias indicate the number of possible games of chess to be of the order 10^{44}, that is probably larger than number of particles in the known universe (e.g. [Simon, Schaeffer 1989, p. 2]). No computer is capable of examining all possible games of chess and give the expected outcome when both sides play best moves (white wins, black wins or maybe a draw?), and it is not likely that any will be in any predictable future.

Compared to most existing decision problems (e.g. well known NP-complete Travelling Salesman Problem (TSP) – ***is there*** *a round tour through given cities of cost lower than MAX?*), chess are more complex. The key question might be here – can white always win? Put more algorithmically: ***is there*** *a move of white such, that* ***for every*** *black's move* ***there is*** *a move of white such, that* ***for every****… such that white wins?* Here, we face a large number of alternating existential / universal quantifiers. Using computer language, we must find a whole sub-tree instead of just a mere path in a decision tree (like in a, still very complex, TSP). The difference here is that in TSP we do not face any resistance, while in a game of chess there is the evil black player that always tries to minimize our chances.

The tree representing game of chess is easy to construct. At the root there is the considered position (at the beginning of the game it is the standard starting chess position), then edges represent legal chess moves and nodes represent resulting positions. Obviously, when we consider the initial chess position to be the root, and assign depth 0 to the root, then edges from nodes of even depth to nodes at odd depth represent moves of *white*, while from odd to even depth – *black's* (it is *white* who makes the first move in a game of chess).

Remark: *A ply* is a word for a single move by one of sides. In graph terminology, it corresponds to one edge in the game tree. Therefore one chess move (one of white and one of black) consists of two plies.

Remark 2: As reasoning about the game means analysing its tree, it is clear that the problem of chess is *a priori* exponential. It can be shown that a generalized game of chess belongs to EXPTIME-complete complexity class, where a "generalized game of chess" means a game that takes place on arbitrary $n_x n$ board size (since complexity classes deal with asymptotic complexity).

Number of edges from the root to nodes depth 1 in the initial position is 20 (there are twenty legal white's moves initially), and from each node depth 1 to node depth 2 is

also 20 (there are also twenty legal black's responses). However, in the chess middlegame it is assumed that on average 35 legal moves exist for each side.

If we consider a tree of depth D, and branching factor B, then number of all its nodes N (except root) can be calculated with formula:

$N = B^D$

Length of an average chess game can be considered to be about 40 moves. 40 moves mean 80 plies. As the branching factor (number of edges from each node down the tree) is 35 on average, we can estimate the number of nodes in a tree corresponding to a game of such length to be $N \approx 35^{80} \approx 3.35*10^{123}$.

If a computer browses through two million positions (nodes) per second (which is a good result achievable on a typical PC computer with x86 CPU working at 2Ghz) it would take over $5.3*10^{109}$ years to exhaust the whole tree. And even employing much faster supercomputers would not change fact that it is impossible to solve the game of chess by means of brute force analysis using hardware and software available now and in the predictable future.

1.2.3 Methods for overcoming complexity

It is impossible to solve the game of chess by means of a brute force analysis of its game tree, due to speed and storage limitations. Therefore people have developed some techniques to make computers reason sensibly about chess given restricted resources.

1.2.3.1 Limiting tree depth

As the entire game tree cannot be searched in most positions (and it is very good as chess would become uninteresting once the expected outcome in the initial position was found) some heuristics must have been developed. The general approach to writing chess programs is to explore only a sub-tree of the game tree.

The sub-tree is created by pruning everything below certain depth. Instead of assigning to the root a value from a set {win, draw, lose} (that could be assigned if the whole game tree was explored down to the leaves) there is a value assigned according to some heuristic evaluation function. Value of the function is found at some intermediate nodes (usually closer to the root than to the leaves) of the total game tree. These nodes are leaves of the considered sub-tree. Value of the evaluation function is calculated according to some rules originating from human chess knowledge, gathered throughout centuries of playing chess. One rule is obvious and natural – it is better to have material advantage, and

the material component is usually dominant for the biggest part of the game. However, there are also many positional aspects that matter – like centralization, piece mobility and king safety. Assessing all these things leads to a result that can be used to compare different positions, and therefore choose the best moves (obviously, best only according to the used evaluation function).

Obviously, the ideal evaluation function would return one of the values *win*, *draw* or *lose* only. Let *win* = 10000, *draw* = 0, *lose* = -10000. Then a real evaluation function would return values from the interval <-10000; 10000>. The bigger the score, the better position is (from viewpoint of the function). It is assumed that a position with a score of 3000 is better that one valued with 650, therefore the program would prefer to reach the first position. However, it is not necessarily true that the first position is superior. It is only so for our heuristics, and it may turn out that extending the continuation leading to the first position we find out that after next several moves it would be assessed as a (-2100) position by the same function, while the second position might unavoidably lead to a checkmate (and therefore maximum score +10000) after next 5 plies. This is the consequence of exploring just a sub-tree of the entire game tree.

As empirical tests show [Schaeffer et al., 1996, p. 1], a chess program searching one move deeper into the game tree than its opponent (other factors being equal) wins about 80% games played. In other games (e.g. draughts) search saturation has been observed [Schaeffer et al., 1993, p. 4]. It means that playing strength increases almost linearly with search depth, but after exceeding some threshold (in case of draughts – 11 plies) gain diminishes, and is negligible for greater depths (19 in case of draughts). Similar behaviour has been observed in chess [Schaeffer et al., 1996, p. 12] – difference in playing strength between the same programs searching 3 and 4 plies is much more apparent than that between programs searching 8 and 9 plies. However, this phenomenon does not manifest in current chess programs as even the best programs currently available have not yet reached the threshold, beyond which the benefits of additional search depth are negligible.

1.2.3.2 Two basic approaches – forward and backward pruning

There are two basic methods of reducing amount of work needed to determine the best move in a given position.

These two strategies were first described by [Shannon, 1950] in his original work. He called them *strategy type A* and *type B.*

Strategy type A, is used in most chess-playing programs (and in many other computer implementations of games) since 60's. Original Shannon's work considered basically browsing through the tree for a certain depth, calculating the evaluation function at leaf nodes of the sub-tree. The best move (promising highest evaluation) can be determined in that way. Since computers at that time were extremely slow, therefore browsing through the sub-tree of depth only as low as 4 (predicting for 4 plies ahead – it means about 35^4 = 1.5 million nodes) was possible in reasonable time. It was a discovery of an α-β algorithm (covered later) that allowed more efficient analysis of the tree by means of so called *backward pruning.* This technique allowed relatively efficient computer implementation of strategy type A programs. The basic idea of the α-β algorithm is to skip branches that cannot influence the choice of the "best move" using data already gathered.

Another approach – strategy type B, also known as *forward pruning* – tries to use expert knowledge to prune moves that seem poor and analyse only a few promising branches (just as human players do). Programs following this approach were written before α-β dominated the arena. There were attempts using this approach later as well, but they did not fulfil expectations – analysis performed to prune branches *a priori* was too computationally extensive.

One of the greatest supporters of the forward pruning style of chess programming was M. Botvinnik. Botvinnik was a world chess champion for 15 years (1948-1963, with two breaks). In the sixties he was involved in a team that devoted much effort to creation of a chess machine (Pioneer project). His opinion was that only highly selective search of promising moves is acceptable to achieve high level of play. Seeking a technique of solving chess problems almost without search, he hoped that methodologies developed for that purpose would find applications in addressing complex practical problems (like those encountered in control theory). The chess program, which creation Botvinnik was involved in, never exceeded a strong amateur level, however experience gathered throughout the lifetime of the project was huge [Stilman, 1997].

1.2.3.3 Search versus knowledge

There is quite a lot of expert knowledge concerning chess around. For example, **openings** (the first stage of a chess game) have been studied and there is a lot of literature available. Assuming that this knowledge is correct (it is not known if any of the openings belongs to the best possible line of play, but following them does not make any of the players enter inferior position according to human chess criterions) introducing it to the

programs might save computers a lot of calculations and possible mistakes (currently, the best human players are still considered superior to computers programs). **Opening databases** are present in all good chess programs.

Once the program cannot follow the opening book anymore, it switches to its normal mode (usually following Shannon's strategy type A). In this mode the only explicit chess knowledge (for example – a rook on an open file is good) that is available is the one put into the **evaluation function**. By feeding the function with more and more factors one must unavoidably reduce speed of program operation – computation of evaluation function value must take some time. By implementing chess knowledge directly in specialized silicon chips it is possible to avoid the slowdown, but lack of flexibility renders this approach impractical with current technology. A smarter program will be able to browse fewer nodes given a time limit, and therefore reach smaller depths in some positions than its dumber version. Therefore, addition of excessive knowledge may reduce program's playing strength, often leading to counterproductive results. Another possible negative side of adding explicit knowledge to the program might be its ability do direct search in wrong direction in certain positions, while a dumber and faster version might be able to discover better continuation by its improved ability to reveal implicit knowledge of the position by search.

One more example of using knowledge to minimize need for tedious search is usage of **endgame tablebases**. When there are not so many pieces left on the board, it turns out to be possible to precompute the optimal line of play and put it in a database. When the program reaches a position that exists in the endgame base, it can switch from search mode to tablebase-read mode, with tablebase serving as an oracle. It saves time, and also helps in positions where finding the optimal line of play is beyond capabilities of the program, or would require including additional chess knowledge (for example, king, bishop, knight against king endgame).

1.2.4 History of chess machines and programs

During the first few years after Shannon's publishing his breakthrough paper in 1950 several programs related to chess were built. They were aimed at handling certain chess problems (simple mate finding, certain kinds of endgames). The first chess program was created in 1956 by Ulan and Stein. Another early program was made in 1958 by Alex Bernstein and his team.

An important achievement, that set the way to follow in chess programming for the next years, was creation in 1959 a program by Newel, Simon and Shaw. This program attempted to use human-like way of reasoning, like the one used by human grandmasters. Its way of making decisions followed the Shannon's type B strategy. Most programs of these times were using this approach.

In 1966 a first computer chess match was played between Russian machine "Kaissa" (developed by Adelson-Velsky, Arlazorov, Donskoy) and American computer built at Massachusetts Institute of Technology (MIT) and Stanford University (development managed by McCarthy). Kaissa won the match with score 3-1.

Development of the engines went on. In 1966-1970 a program called Mac Hack was developed by Richard Greenblatt and Donald Eastlake at MIT. Its playing strength made it possible to win against human amateur player. Moreover, enough chess programs were developed to make it feasible to organize computer chess events and tournaments, both national and international. It certainly speeded up progress on the field of chess programming. An important event was a bet in 1968 between International Master David Levy and a group of scientists that no machine would be able to beat Levy at chess for the period of following 10 years. That bet increased interest in creating strong chess programs.

The next step was transition to using Shannon's strategy type A. It started in the 1970's, when American program Chess using brute force searching (improved by several techniques) won the second World Chess Championship. This program was created by Slate and Atkin, who wrote a very important book about chess programming – [Slate, Atking, 1977]. Since then most chess programs rely on brute-force search rather than imitating human reasoning. Experience showed that fast brute-force searchers outperform programs that spend a lot of time on careful choosing suitable moves for further analysis.

"Chess" by Slate and Atkin reached a level of moderately strong club player (about 2000 Elo rating points), but its playing strength was far too low to win the bet against Levy (whose rating at the time was over 2400 points).

Most leading chess programs in the 1980's used some form of alpha-beta search algorithm. Alpha-beta algorithm for backward pruning had been known for over 40 years then, so it had been analysed carefully and many enhancements has been available. Increased interest in alpha-beta search resulted in quick further development of alpha-beta, making chess programs using it unmatched for any others. The only strong program that did not fit into the brute-force framework used by other programs was Berliner's Hitech.

Two more breakthroughs were to be seen in 1980's. One was start of development of ultra-fast searchers. They used sophisticated dedicated parallel hardware to achieve nodes-per-second on the order of 750,000 (Deep Thought) or even 5,000,000 (Deep Thought II). Most programs at the time could compute only 10-20,000 nodes per second. That trend lead to creation of Deep Blue that won a match against reigning human chess champion Garry Kasparov in 1997. Deep Blue while winning this match was able to compute 200,000,000 nodes per second.

Another breakthrough was a dramatic increase of performance of microcomputer programs in late 1980's, either running on a standalone dedicated machine or on a normal PC. Earlier the strongest programs were running only on big computers, like mainframes or even supercomputers (like Cray Blitz, World Champion from 1980's).
Microcomputers could stand the fight against much heavier machines due to increase of computing power of microprocessors, and flexibility (easier modification and tuning). Some newly discovered techniques dramatically improving strength of play (like null-move, described in the 2. part of this work) could be easily implemented in PC programs, but were not so easy to include in big sophisticated machines.
Currently most best programs run on a PC, although some development employing hardware solutions is also present (e.g. Hydra project - http://www.hydrachess.com/). The best chess programs currently available play on the level of a top grandmaster.

1.2.5 Outcomes of chess programming

As Shannon predicted, chess programming has found many applications in science and technology.

1.2.5.1 Chess programming and AI

For a long time, many researchers considered chess programming the same for Artificial Intelligence as *Drosophila* (fruit fly) is for genetics. During fifty years of its existence, chess programming helped to understand the way human brain thinks while playing (and in analogous situations as well). However, at some point most effort went into researching backward-pruning-based solutions, almost neglecting forward pruning approach. Forward pruning more resembles operation of a human brain, and therefore might be considered closer to traditionally taken AI.

Backward pruning algorithms put search over expert knowledge. Currently such brute-force search gives better results than smarter, more selective knowledge-based

approaches. Here comes one of the most important experiences from chess programming – that search can be considered as a **replacement** for knowledge. Some ([Schaeffer, 2003]) state that search is in fact equivalent to knowledge, as it reduces need for explicit knowledge (that will be uncovered by search) and can be used to reveal implicit knowledge (e.g. the best move in chess).

Although search algorithms are well understood and analysed, still there is much to improve. Some techniques of search enhancements may improve performance of the generic algorithms by orders of magnitude.

However, as no major breakthroughs are expected in our understanding of search algorithms, less and less attention is paid by AI researchers to that domain. This is one of the reasons, why chess is considered by some scientists to be about to lose interest [Donskoy, Schaeffer, 1990, p. 2].

Another point where chess and AI coincide is acquiring and processing expert knowledge to use it in evaluation functions. This aspect is clearly within boundaries of modern AI research. Unfortunately, it has not been within the mainstream of chess research by now. An example of supplementing search by expert knowledge is given in [George, Schaeffer, 1990].

Recently, more attention has been put to learning in chess programs. One of the promising techniques is *temporal-difference learning*. It relies on a database of games (might be generated from games played by the program itself) that are used by the program to determine a fine combination of evaluation function features. More about this method is presented in the part describing different components of chess programs.

Chess have also proven useful for AI research in areas of tutoring systems, knowledge representation and acquisition [Schaeffer, 2002, p. 2]. An accusation from AI world toward chess researchers is that they develop techniques focusing on chess-specific aspects, neglecting general Artificial Intelligence research.

Inability of computer chess programs to dominate over human players despite huge effort put, has caused many analytics to doubt about usefulness of used techniques to help with real-world decision problems. Chess are a well defined, finite problem, while most practical problems are ill-defined, what introduces additional complexity.

Problems with defeating human players, for example the *DeepBlue* project that finally led to winning a match with human world champion Garry Kasparov, demonstrate how expensive it is (in man-years and money) to construct machines capable of competing with humans [Schaeffer, 2002, p. 2].

1.2.5.2 Other outcomes

Game of chess

Due to computers' ability to exhaustively analyse the game tree, they have changed humans' knowledge about the game of chess itself. For example, some openings that were previously considered tolerable, have been knocked down as leading to lost positions (especially sharp tactical variants, for example some lines in the king's gambit). Another example is that computers proved that established rules of chess could prevent the game from reaching the fair end. Specifically, the 50-moves rule (*game ends with a draw if 50 consecutive moves without capture and moving a pawn have been made*) in some endgames is too restrictive, as computer analysis showed that over 50 such moves may be necessary to defeat the weaker side playing optimally. In some positions of type (king + rook + bishop) vs (king + two knights) one may need as many as 223 moves without capture nor pawn move (as first determined).

Search advances

Progress made while working on increase of game-playing programs strength found its applications in solving many economical (game theory) and operations research problems. Techniques developed for use in chess (and many other game-playing programs) can often be used when good (not necessarily optimal) solution is satisfactory, and determining optimal solution is too complicated. Some concepts (for example concerning minimizing the search tree) have found applications in single agent search (solving a problem by finding an optimal, or near-optimal, sequence of moves by one player only, like finding a path through a maze).

Parallel processing

A lot of effort has been put into developing parallel search solutions. One example of a successful project is APHID [Brockington, Schaeffer, 1996]. Parallel processing in case of complex problems allows huge boost in performance by using many workstations working simultaneously, without excessive increase of costs that would be required if supercomputers were involved.

Another example is *DeepBlue* that contained 32 microprocessors, each supported by 16 specialized chess accelerator chips (512 total accelerator chips). All those processors worked in parallel, cooperating to browse about 200 million nodes per second in average positions [IBM, 1997].

Engineering

To maximise performance of chess programs, they are sometimes supplemented by chess-specific VLSI chips (for example, for move generation). By introducing specialized hardware devices one can speed up some parts of the program even by a factor of 100. Moreover, additional chess knowledge can be added to the machine without compromising search speed (as it is in the case of software solutions running on general purpose hardware). This obviously increases playing strength by a significant amount. The recent appearances of hardware-based approach are under the form of machines called Brutus (an FPGA based PC extension card, 4th place in the World Computer Chess Championship in Graz, 2003) and Hydra (hardware solution, won the prestigious International Computer Chess Championship in Paderborn, 2004). However, it is difficult to determine now whether relevance of chess-oriented chip development will concern design of other special-purpose chips.

A thing that seems strange from an engineer's point of view is that there are no widely available applications nor environments that would aid building chess programs, although there are probably thousands of chess programs developed worldwide. As basic knowledge concerning chess programming seems to be well developed already, creation of a chess-oriented set of reusable software components and tools might give a significant boost to computer chess advances, as developers might focus on researching new solutions rather than repeatedly stating the obvious.

2 Typical structure of a chess program

A chess program must be able to:

- remember what the position is;
- generate moves, given a position;
- verify moves, to make sure the move to be made is legal;
- make clever moves;

Therefore, each chess program must have a data structure to hold the **position** and **moves.** Moreover, it must contain some kind of a **move generator**. Finally, it must have some kind of a **search algorithm** and an **evaluation function**.

2.1 Basic data structures

In this subchapter I describe some data structures that are often used in all chess programs.

2.1.1 A move

Definitely, a structure describing a chess move should contain the source and destination square. Information contained in the *move* structure must be enough for the program to be able to take that move back. Therefore, it should also describe the captured piece (if any). If the move is a promotion, it must contain the piece that the pawn is promoted to. Sometimes (it depends on a type of a chessboard representation), also type of the piece that is moving should be included.

The above data may be put together into a structure having several one byte fields (e.g.: *source, destination, captured, promoted, moved*). However, the efficiency of the program often increases if the structure is contained within one 32-bit integer (for example, 6 bits per source / destination field, 4 bits per captured / promoted / moved piece) as it reduces the memory access overhead for transferring/accessing particular fields of instances of this often used structure (assuming that arithmetic and logical operations are performed relatively much faster than memory access – in most hardware environments, including the most popular PC platform, it is exactly the case).

2.1.2 Game history entry

At the moment the move is made, data contained in the *move* structure is enough. The procedure that makes moves has enough information to update locations of

pieces as well as castling rights, *en passant* status and number of moves before 50-moves rule applies. However, more information is needed to undo the move. For example, it is often impossible to recover the castling rights using only information from the *move* structure. Therefore, the game history entry should contain all the information needed to take a move back. It includes: data from the *move* structure (except *promoted* field), castling rights before the move was made, *en passant* capture field (if any) and number of moves without capture and pawn movement.

2.1.3 A chess position

There are several methods of representing a chess position in the computer memory. The most natural way seems to be an array 8x8, with each element corresponding to one of chessboard squares. Value of the elements would indicate state of the square (empty, occupied by black pawn, occupied by white queen etc.). If one byte was used to represent one square, then the whole data structure would occupy 64 bytes of computer memory (I will ignore everything but piece location for now), that is not much. Therefore, this approach was widely used in the early stages of computer chess development. For code optimizing reasons, the two dimensional array 8x8 was often replaced with a single one dimensional array of 64 elements.

Later, the idea was improved by adding two square sentinels at the edges. Sentinel squares were marked *illegal*. It speeded up move generation, as no check had to be done each time a move was to be generated to verify if edge of the board was not reached. The drawback was increase of the structure size to 144 bytes.

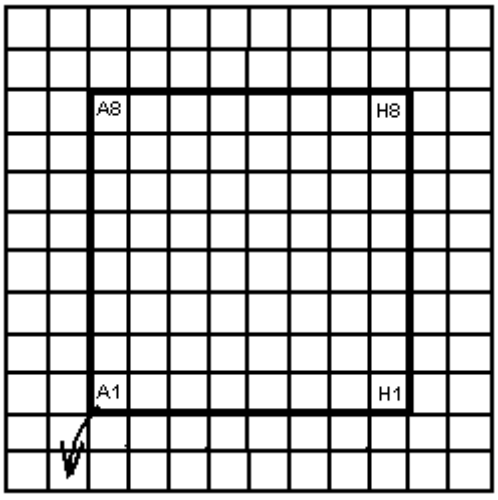


Fig. 1. A chessboard with double sentinels on the edges

On the figure 1. there is a graphical picture of that board representation, with the arrow indicating that it is impossible for any piece, including a knight, to leave the

enhanced board. The actual chessboard is surrounded by the thick line, and the fields outside are sentinels. Move generation for any piece will be stopped when the scan for possible destination squares reaches another piece, or the first layer of sentinels. However, a knight on the edge of the board can jump for two squares, therefore additional layer is required.

Another approach is to use 32 bytes only, with each byte corresponding initially to one chess piece. Value of the byte would indicate the square where the piece is standing, or the *captured* status. Special handling is required for pawn promotions into a piece that has not been captured yet, however this problem is easy to overcome.

The two techniques described above are considered as obsolete now. Most modern high-performance programs employ either the 0x88 or bitboard-based approach, that are described below.

Remark: Locations of pieces on the board **is not** all that must be remembered in order to completely describe the chess position. Other issues are: castling rights, *en passant* capture opportunities, number of moves until the 50-moves rule holds, number of times the position repeated during the game (required by the threefold repetition rule; the last one is relatively hard to track).

2.1.3.1 0x88 representation

Description of this data structure is available on many Internet sites, I based here on the one given by author of a very strong freeware chess engine (R. Hyatt).

Pushing the idea of sentinels to the extreme, one arrives at the 0x88 board representation, that uses a 2-D array of size 16x8 – twice the chess board size:

```
112 113 114 115 116 117 118 119 | 120 121 122 123 124 125 126 127
 96  97  98  99 100 101 102 103 | 104 105 106 107 108 109 110 111
 80  81  82  83  84  85  86  87 |  88  89  90  91  92  93  94  95
 64  65  66  67  68  69  70  71 |  72  73  74  75  76  77  78  79
 48  49  50  51  52  53  54  55 |  56  57  58  59  60  61  62  63
 32  33  34  35  36  37  38  39 |  40  41  42  43  44  45  46  47
 16  17  18  19  20  21  22  23 |  24  25  26  27  28  29  30  31
  0   1   2   3   4   5   6   7 |   8   9  10  11  12  13  14  15
```

Fig. 2. 0x88 board representation (taken from [Hyatt, 2004])

The actual data structure treats this 2-D array as a 1-D vector, with elements enumerated as shown above. The chess position is stored only in the fields occupying the left half of the picture. The trick is that now there is no need for memory fetch to verify whether the square is a legal one (within the board) or an edge has been reached. It is more visible when the structure is enumerated with hexadecimal numbers:

```
70 71 72 73 74 75 76 77 | 78 79 7a 7b 7c 7d 7e 7f
60 61 62 63 64 65 66 67 | 68 69 6a 6b 6c 6d 6e 6f
50 51 52 53 54 55 56 57 | 58 59 5a 5b 5c 5d 5e 5f
40 41 42 43 44 45 46 47 | 48 49 4a 4b 4c 4d 4e 4f
30 31 32 33 34 35 36 37 | 38 39 3a 3b 3c 3d 3e 3f
20 21 22 23 24 25 26 27 | 28 29 2a 2b 2c 2d 2e 2f
10 11 12 13 14 15 16 17 | 18 19 1a 1b 1c 1d 1e 1f
 0  1  2  3  4  5  6  7 |  8  9  a  b  c  d  e  f
```

Fig. 3. 0x88 board representation, hexadecimal notation (taken from [Hyatt, 2004])

Each rank and file of the board is indexed using a 4-bit number (nibble). For example, square numbered with 0x34 can be read as 0x3rd rank and 0x4th file. The valid squares, that correspond to the actual chessboard, have the highest order bits in each nibble equal to zero, while the invalid squares (the right half on the figure) do not.

Consider a bishop from chess square E1 (square number 4 on the scheme) moving along the diagonal E1-H4 (0x4-0x37) – see figure 2. The squares that the bishop visits are obtained by adding constant 0x11 to the previous square (0x4 – 0x15 – 0x26 – 0x37). All

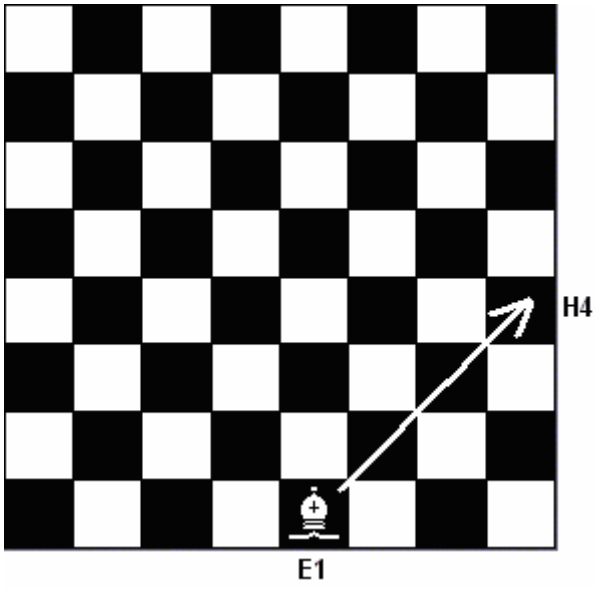


Fig. 4. Bishop moving from E1 to H4

those squares have highest order bits set to 0, and they all represent valid bishop locations. When we try to push the bishop beyond the square H4 (that lies on the very edge of the real chess board) we arrive at a square numbered 0x48. This number has the one of the highest order bits set (in the nibble 8), that can be detected by performing a logical AND of 0x48 and 0x88. The result of the AND operation is nonzero, that indicates an illegal square.

This replacement of a memory read by a single logical operation was significant in times when cache memory was not so widely used – memory access was a costly operation, and the value read from the memory served for no purpose – it was discarded as soon as validity of the move was confirmed or negated. The memory overhead (doubled size) is negligible in most of the modern computers, since there never exist many instances of this data structure.

The 0x88 data structure allows many tricks, and has proven efficient in supporting many operations typically done by a chess program – archives of Computer Chess Forum [CCC] know many examples, described by practising developers.

2.1.3.2 Bitboards

Idea of using bitboards (64-bit integers) came into being when 64-bit mainframes became available in 1960's. Sixty four happens to be the number of squares on the chessboard. Therefore, each bit in the 64-bit variable may contain a binary information for one square. Such binary information might be, for example, whether there is a white rook on the given square or not. Following this approach, there are 10 bitboards (+ one variable

for each king) necessary to completely describe location of all pieces (5 bitboards for white pieces (pawns, knights, bishops, rooks, queens) and 5 for black). There is no need to maintain bitboards for kings as there is always only one king on the board – therefore a simple one byte variable is enough to indicate location of the king.

An addition to the bitboard representation of the actual chess position is a database containing bitboards representing square attacked by a given piece located on a given square.

For example, *knight*[A4] entry would contain a bitboard with bits corresponding to squares attacked by a knight from A4 square set.

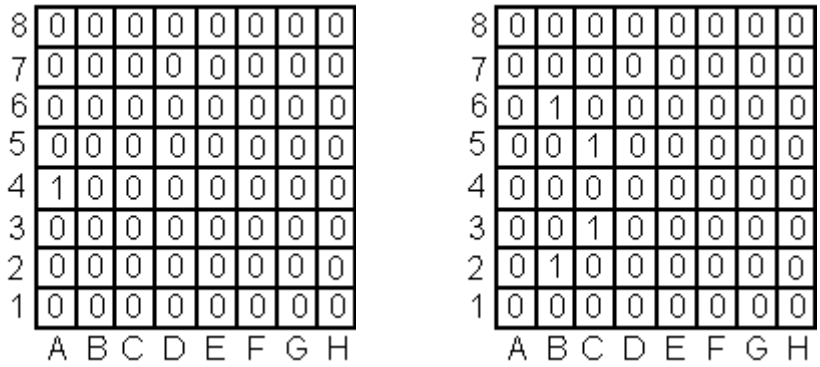


Fig. 5. Graphical representation of bitboards representing a knight (or any other piece) standing on square A4 (left) and squares attacked by this knight (right)

Assuming that the square A1 corresponds to the least significant bit of the bitboard (unsigned 64-bit integer), square H1 – eighth least significant bit, A2 – nineth, H2 – sixteenth etc., H8 – most significant bit, the bitboard from the left-hand side example can be represented in a more machine-friendly form: 0x1000000h (or 2^{24}), and the one from the right-hand side: 0x20400040200h ($2^{41} + 2^{34} + 2^{18} + 2^{9}$).

Using the bitboard and the precomputed database, it is very fast to generate moves and perform many operations using processor's bitwise operations. For example, verifying whether a white queen is checking black's king looks as follows (an example taken from Laramée 2000):

1. Load the "white queen position" bitboard.
2. Use it to index the database of bitboards representing squares attacked by queens.
3. Logical-AND that bitboard with the one representing "black king position".

If the result is non-zero, then white queen is checking black's king. Similar analysis performed when a traditional chessboard representation (64 or 144 byte) was used would require finding the white queen location (by means of a linear search throughout the array) and testing squares in all eight directions until black king is reached or we run out of legal moves. Using bitboard representation, only several processor clock cycles are required to find the result.

Remark: As most of today's widely available processors are 32-bit only, and cannot smoothly operate on 64-bit numbers, some of the performance gain is lost. Nevertheless, the technique remains fast, conceptually simple and powerful. Moreover, since the shift towards 64-bit PC architectures has already started, bitboard programs are likely to benefit from it heavily soon.

2.2 Basic procedures

There are some procedures common to all chess programs.

2.2.1 Move generation

Generation of chess moves is an obligatory activity for every chess engine. In this paragraph I will describe several issues of move generation. The implementation of this function is strongly related to the choice of data structure representing a chess board. An example of implementation using bitboard representation will be described in the third part of the work that describes my own chess program. Below general approaches to creating a generator are described.

2.2.1.1 Selective, incremental and complete generation

Three major approaches to move generation are [Laramée, 2000] :

- Selective generation – carefully analyse the position in order to find a few promising moves, and discard all the rest.
- Incremental generation – generate a few moves at a time, hoping that one of them turns out to be good enough for the rest to become insignificant – they would not need being generated, leading to time savings. The savings may be significant, since move generation in chess is not trivial (there are castlings, en passant captures, each kind of piece moves differently).
- Complete generation – generate all possible moves for the given position in one batch.

Selective generation can be viewed as a scheme for forward pruning, also named Shannon's scheme B. Only a small number of promising moves are generated, and furthermore analysed. The choice of viable moves is made using chess knowledge – the effort of choosing good moves is shifted from the search procedure to the move generator. However, as it was mentioned before in the paragraph describing forward pruning technique, computational effort required to find good moves without searching, by means of static analysis, is too high. Machine spends too much time on static analysis of the position, and cannot perform enough dynamic analysis in a time limit, that leads to blunders and overall low playing strength.

Incremental and complete generation are commonly used in modern chess programs. There are benefits and penalties from both schemes.

Search routines basing on alpha-beta algorithm for backward pruning that are used in chess programs are very sensitive to move ordering – the rule here is simple: best move first. It means that best moves should be analysed first, and in that case most moves may never need being analysed (and therefore even generated). Obviously, the program never knows whether the generated move is best in the given position – if it knew, it would not need analysing it anymore. However, there are some rules that allow us to indicate best (or at least good) moves – for example, in the game of chess a best move is often a capture, or a checking move.

Therefore, since we can achieve a reasonably good move ordering, it would seem more logical to employ an incremental generator – in the first batch it might return only captures (preferably sorted by an expected material gain), then checking moves etc. In such case, often one of the moves generated in the first batch would turn out to be the best (or at least good enough) one, and the rest would not have to be generated. The drawback of the incremental scheme is that it takes it longer to generate all possible moves than it would take a generator that returns all possible moves at once.

A complete generation is the fastest way to obtain all moves for a position. However, usually most of those moves (assuming reasonable move ordering) turn out to be useless for the engine – they never get to being analysed. An advantage of generating all possible moves at once is that it enables some search tricks (e.g. enhanced transposition cutoff (ETC) – one of the search enhancements described later), that may give benefits outweighing the cost of excessive generation.

Obviously, even when using complete generation a good move ordering remains crucial – it may lead to many branches being skipped by the search routine. Therefore,

move generator should return moves according to the “first-best” rule wherever possible, reducing the need for reordering later.

2.2.1.2 Legal and pseudolegal moves

Another question that must be answered while creating a move generator is whether to generate only moves that are absolutely legal, or to generate all moves that obey the piece movement rules, but may violate some chess constraints (e.g. leave king in check) – these moves are called pseudolegal.

Obviously, only legal moves can end up being made on the board. However, complete validation of chess moves in generation phase is costly, and cases when a move is pseudolegal but is not legal are rare. This leads to a conclusion that a program using a pseudolegal move generator may perform better than that validating all moves at generator. The reason is that by postponing the test for legality till the search phase we may avoid many such tests – branches starting at illegal moves might be pruned and never analysed. As a result, time savings may occur.

A program might use the number of legal moves as a criterion when making some decisions (for example non-standard pruning techniques) – then, moves must be validated by the generator. Otherwise, it is usually better to return all pseudolegal moves.

2.2.2 Evaluation function

A heuristic evaluation function (usually called simply evaluation function) is that part of the program that allows comparison of positions in order to find a good move at the given position. As was mentioned in the first chapter, it is not guaranteed that value of the function is the absolutely optimal guide. However, since game of chess has not been solved yet, no perfect evaluation function exists, and we must do with a heuristic version only.

Evaluation function is essential for avoiding blunders in tactical positions as well as for proper choice of a strategy by the program. While the former is easy to achieve as far as evaluation function is concerned, the latter is quite difficult. Creating a good evaluation function requires good understanding of chess and detailed work of the program. Good knowledge about chess is required to formulate the evaluation function in such a way that the program can pick sensible moves basing on its value. Insight into detailed operation of the program is important as some properties of the evaluation function (for example, granularity) influence operation of other parts of the program. For optimum performance

of the entire program, evaluation function must remain in harmony with co-working procedures (e.g. tree browsing).

There are two basic aspects of the chess position that must be considered: material and positional.

2.2.2.1 Material balance

In most positions, it is material that plays dominant role in value of the evaluation function.

Chess note: These are the relative values of the chess pieces widely established throughout the literature: ***pawn (100), knight (300), bishop (325), rook (500), queen (900)***.

Material balance can be therefore determined using a simple formula:

$$MB = \sum_i (N_{Wi} - N_{Bi})V_i$$

where i=0 corresponds to pawns, i=1 to knights etc.,

N_{Wi} stands for number of white pieces of type *i* and analogously N_{Bi} stands for number of black pieces. V_i is value of a piece of type *i*.

If MB > 0 then white leads on material, MB = 0 indicates that the material is balanced while MB < 0 shows that black is in better shape.

King cannot be captured, and each side has exactly one king, therefore it is skipped in the material balance evaluation.

Calculation of material balance in the above form is trivial. In practice, the formula is modified. For example, it is known that in most positions it is a good idea to exchange pieces of equal value if we have a material advantage. Another example is while playing *gambit*. Sacrifice of a pawn or more in the opening may be compensated by positional advantage. However, positional aspects (described in the next paragraph) usually are valued low compared to material. In most programs, evaluation function is constructed in such a way that positional advantage can hardly ever compensate being two pawns beyond the partner.

Therefore, when the program leaves the opening book and is forced to play on its own, it can suddenly find out that it has a miserable position and play inadequately, although the position may be positionally won. To avoid such mistakes, some kind of a “contempt factor” may be included to handle such positions and allow program to make a fair evaluation [Laramée, 2000].

2.2.2.2 Positional aspects

There are lots of positional aspects that should be included while computing value of the evaluation function. The list given below does not exhaust all aspects, but describes the most popular, in a nutshell:

- ✓ Development
  - The more pieces developed from starting locations, the better.
  - It is advantageous to have king castled.
  - Minor pieces (knights, bishops) should be developed before major pieces.
- ✓ Board control
  - Control over central region of the board is advantageous.
- ✓ Mobility
  - The greater the better.
    - Pitfall – assigning too large weight to mobility may have negative influence (insensible moves of type "Kg8-h8" should not count, pointless checks may be valued too high as they leave only a few legal responses for the opponent).
  - "Bad bishops" (e.g. a bishop moving on white squares is limited by own pawns occupying white squares).
  - Pinned pieces (cannot or should not move).
  - Bishop pair in open position is valuable.
- ✓ King safety
  - Important in opening and middlegame.
- ✓ Pawn formation
  - Doubled and tripled pawns are usually weaknesses as they have limited mobility, cannot defend each other.
  - Isolated pawn may be a serious weakness as it must be defended by other pieces.
  - Passed pawn is usually a good thing as it cannot be threatened by opponent's pawns and may easier get opportunity of queening.
  - Having all eight pawns is rarely a good idea as they limit mobility of own pieces; usually, at least one line should be opened to improve activity of rooks.

A practice used by some chess programmers is to penalize computer when the position is blocked – almost all chess engines perform worse then, at least against humans. Most chess engines are very good in tactics, that favours open positions. One must realize however that closed position itself is not disadvantageous for any side, therefore such tricks are somewhat artificial. Their applicability will probably diminish when programs employing this technique are given knowledge that will allow them to behave properly in those inconvenient positions. An example of how the above technique is used in a top-class program is Crafty's source code, that is freely available (e.g. ftp://ftp.cis.uab.edu/pub/hyatt/).

The factors of evaluation function that are given above are not absolute – for example, doubled pawns are usually positional flaws, however in some positions they may decide about the win. Yet, there is no simple way of determining whether it is the case. Considering position itself, it would require using much more chess knowledge (possibly encoded using some patterns) into the evaluation function. Such function is likely to work too slowly and negatively influence search effectiveness. Here comes a rule that is employed by most programmers – if in doubt, keep the evaluation function simple and leave more computing power to routine that browses the tree. Practice shows that it is very hard to tune evaluation function enough to compensate for searching one or two plies deeper. However, when chess programs achieve search depth that shows saturation of the search process (when increasing search depth will not increase playing strength anymore), fine-tuning of evaluation function will probably become the way of further improving performance.

Currently, some programs use other techniques to improve work of the evaluation function. One approach is given by George and Schaeffer (1990). The approach aimed to add some "experience" to the program. As most chess programs do not remember past games, they lack a factor that is natural for humans – possibility of using past experience in new competitions. The approach shown in that publication is to supply the program with a database of positions taken from thousands of grandmaster games. During play, the database is queried for identical or similar positions. The database is indexed with some patterns. The program extracts some patterns from the current position, queries the database, and is given back some information about the position. This information is used to modify value of the evaluation function, therefore increasing probability of choosing the move that seems proper according to the database. As size of the database increases (it

consists more positions), the probability of finding the desired position and getting useful information increases.

The method described above might be very useful for example in situation when the program leaves the opening book and starts to play on its own. It is often difficult for the program to follow the plan that is consistent with the "soul" of the opening. Here, the database and patterns becomes handy. This technique improves performance of the program. The problem is to prepare the database and add chunks – knowledge acquisition is, as usually, a difficult task.

2.2.2.3 Weights, tuning

Evaluation function result is usually a number (either real or integer) obtained as linear combination of various components. It seems strange to enclose the whole chess knowledge in a single number, but this approach has performed best by now. The quality of the function depends on weights that describe significance of particular components.

Some search algorithms alternative to a-ß (described in the next part) require evaluation function to return more than a single value. An example is Berliner's B* (described elsewhere in this paper) that needs two values – the optimistic and pessimistic position evaluation.

Most programs use different weight set for different stages of the game. It is important since influence of some aspects on the evaluation varies during the game. For example, king safety is very important during the opening and middlegame, but is insignificant in the ending as there is usually not enough mating equipment on the board to threaten the king.

A sample problem within one stage of game (for example – middlegame) might be – is it good to have a doubled pawn and compensation of a rook on an open line? If a doubled pawn penalty is 50 and a rook on an open line is valued as +35, then it seems better to avoid pawn doubling, even if the line will not be taken. However, it would change if doubled pawn penalty was lowered to 25.

Schaeffer et al. (2002) state that choice of components and weights in evaluation function seems to be the most time-consuming task in creating high-performance game-playing program. Attempts have been made towards automating this task [Buro, 1995; Anantharaman, 1991; Buro et al., 2002], however only limited successes have been achieved (as far as only game of chess is concerned). Importance of different features contributing to evaluation function has always had to be manually tuned to achieve high

performance. As manual tuning is tedious and requires much time, there lasts a continuous effort towards developing new, efficient methods. The main problems that are faced include: huge search space (depends on number of parameters in evaluation function) and difficult identification of merits of particular evaluation function components.

Two of them are briefly described below.

Genetic Algorithms (GA)

Genetic algorithms are a relatively new approach to optimization problems, but have already proven powerful in many applications (for example, solving the Travelling Salesman Problem). The idea behind the GAs is to mimic the evolutionary processes, that obey the rule “the fittest survive”. The genetic algorithm follows the following cycle:

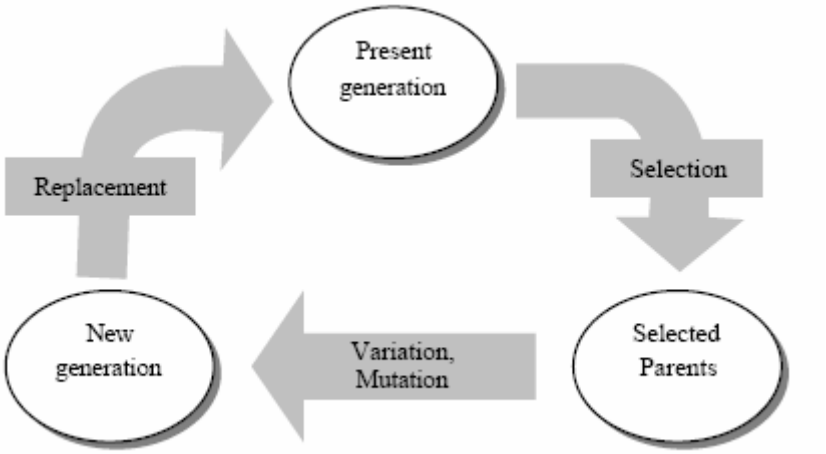


Fig. 6. Genetic algorithm - flowchart

Evolutionary process finishes when the population meets certain requirements (or simply the simulation ran out of time).

Analysis and experiment performed by Tomasz Michniewski [Michniewski, 1995] showed that GAs are basically not feasible for optimization of evaluation function parameters. The main problem is lack of a suitable method for fitness calculation. The proper form of evaluation function is not known, therefore its fitness cannot be estimated directly. Results of games between the same programs using different weights in evaluation function can be used for comparing evaluation functions. However, result of one game is not reliable – a series of tens of games is required to gain significant confidence. Since a single game of chess takes at least a few minutes, it is clear that one iteration of genetic algorithm requires either a large amount of programs simultaneously competing on different computers or a huge amount of time in case of limited amount of hardware. Since the search space is huge (in [Michniewski, 1995] there is described a

situation where only 24 components of evaluation, but the space was still of the order of 10^{45}] a lot of iterations are required to achieve clear improvement, although a good initial population can be set up using chess expert knowledge. Additionally, result achieved by playing games at short time limits can be not portable to longer games.

The results from [Michniewski, 1995] are not promising – only 70 iterations of genetic algorithm could be performed given limited time and resources. GA used elitist and roulette selection, arithmetic crossover and mutation in this experiment. The resulting program played at the similar level as the original one, and probably finely tuned hand-crafted parameters would perform much better. The outcome of the experiment could show improvement if more iterations were performed – but, on the other side, the number of parameters equal to 24 is low, making the optimization easier. Modern high-performance programs use more than 100 parameters in evaluation function that should be optimized.

Temporal difference learning (TD learning)

One solution that is still under research is to employ *temporal difference learning* (a method of unsupervised learning) to determine the optimal combination of weights. Sutton [Sutton, 1988] describes temporal difference methods as belonging to a class of incremental learning procedures specialized for prediction problems. Indeed, the problem of picking best weights for evaluation function's components might be considered as a problem of predicting correct weights. TD methods are driven by the difference between temporally successive predictions – learning takes place each time there is a change in prediction over time. An example (given by Sutton) might be prediction of the weather. Suppose a weatherman is to predict if it will rain on Saturday. Using conventional learning methods, the predictions would be faced against the actual outcome – the Saturday's rainfall or its lack. Using TD method, one compares each day's prediction with the one for the following day. So if the probability of rain on Saturday is found to be 50% on Monday and 75% on Tuesday then the TD approach increases probability for days similar to Monday (ideally, the prediction should be the same on Monday and Tuesday, so the latter, more reliable information is propagated backward).

The formula for modifying weights looks as follows (taken from [Schaeffer et al. 2002]):

$$\Delta w_t = a\,(P_{t+1} - P_t)\sum_{k=1}^{t} l^{t-k}\nabla_w P_k$$

where

Δw_t - change of weight

α - learning rate (step size)

P_t - t-th prediction (predictions $P_0 \ldots P_{t+1}$ are given, as search results)

l - a decay rate parameter (determines by how much the learning is influenced by subsequent states – whether the algorithm uses short- or long range prediction)

$\nabla_w P_k$ - gradient of the predicted value of the k-th state with respect to the weights

There are two basic advantages of TD learning. First is that it is incremental-based, thus easier to compute, while the other one is its greater efficiency compared to conventional methods – it converges faster and gives more accurate predictions.

It has not been proven yet that this method will allow for creating better evaluation functions than manually fine-tuned ones. In backgammon it has succeeded [Tesauro, 1995], but it is still unknown if it will also in other games. The first experiment with chess [Baxter et al., 2000] showed some improvement of the evaluation function, but the program did not achieve grandmaster level that is possible with manually tuned functions.

2.2.3 Search

Basically, all high-performance programs employ some kind of an alpha-beta algorithm to reduce game tree size.

2.2.3.1 Concept of a minimax tree

As it was mentioned in previous chapters, the concept of finding the best move usually

involves looking ahead several moves and evaluating the position. It is assumed that at each moment a player on the move picks the best (heuristically best) move for him. The picked move is returned to the parent position (up in the tree). Let's assume that player A plays *white* and player B tries to win with *black*. Then, at each moment, A attempts to maximize A's winning chance and B wants to maximize B's chances, that is equivalent to minimizing A's chances (since a zero-sum game is considered).

An example of a mini-max tree for an imaginary game might look as follows:

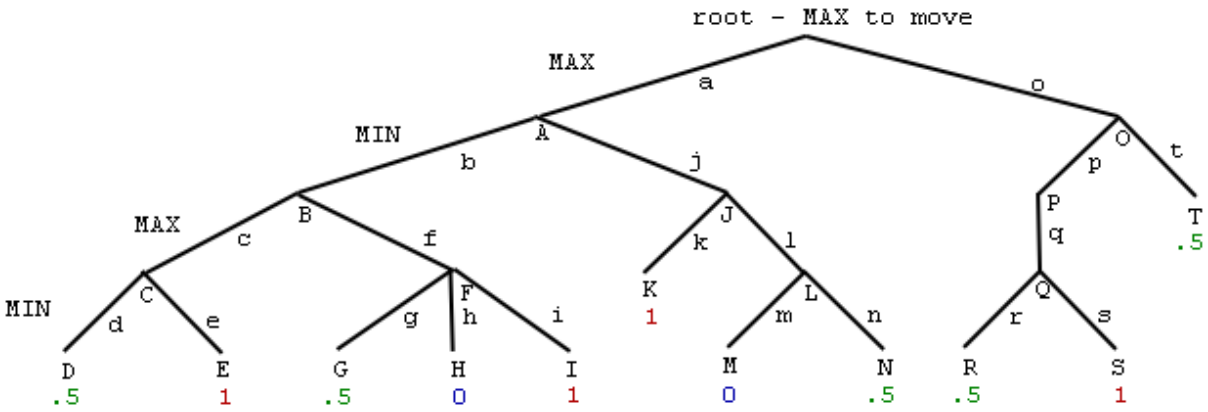


Fig. 7. Mini-max tree

The edges (corresponding to legal moves) are marked with small letters, while nodes (positions) are described by capital letters. Score "1" at leaf nodes corresponds to MAX's win, ".5" means draw and "0" to MIN's win. Following the alphabetical order of nodes one travels through the tree just as a depth-first left-to-right algorithm would do.

Determining the mini-max value of the tree (corresponding to the expected outcome of the game) is described now.

We start at leaf that is placed most to the left – leaf *D*. It represents a position that is a draw. Therefore, at position *C* (terms *node* and *position* are equivalent) the MIN player may choose move (edge) *d*, and draw the game. If he chooses move *e* then he loses (MAX wins) – he wants to maximize his outcome, therefore he picks move *d*. It is said that the mini-max value of node *C* is 0.5. Similarly, considering MIN's moves at node *F* we find out that he would choose move H that makes him win – so the mini-max value of node *F* is 0.

Now, we know that at node *B* (MAX to move) MAX will choose move *c* since it leads to a draw (always assume best moves from each side) while making move *f* makes MAX lose (MIN plays *h* and wins) – definitely not something MAX is interested in. Therefore, the mini-max score of node *B* is 0.5.

Following this approach, the following mini-max values at all nodes can be found:

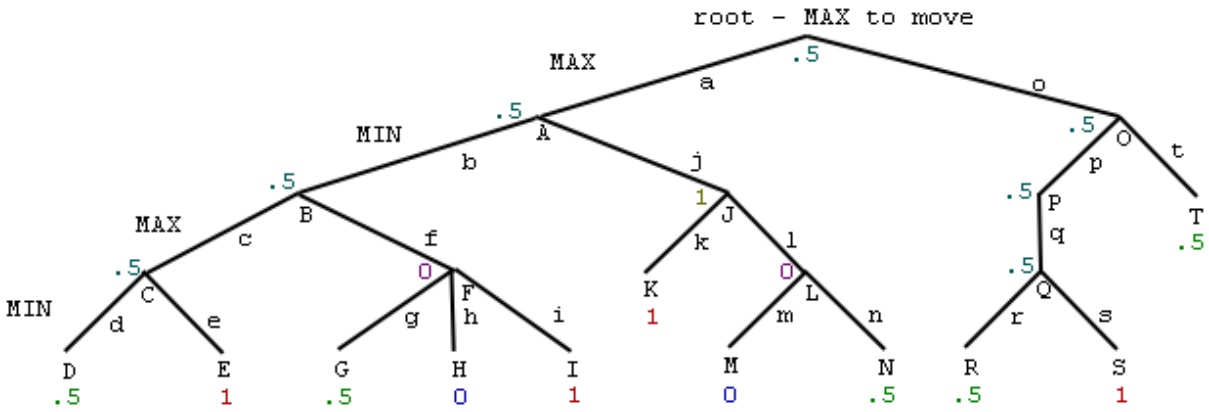


Fig. 8. Finding the mini-max value of the root (own analysis)

Therefore, the expected outcome of the game is a draw – the value of the root.

The algorithm for determining the mini-max value written in a C-like pseudocode looks as follows:

```
chSCORE minimax(chPOSITION node) {
     if (isLeaf(node) ) return evaluate(node);
     else if (isMaxOnMove(node) ) {
          g = -maxSCORE;   // maxSCORE is a huge positive
          //number
          x = left_most_child(node);
          while (isNotNull (x) ) {
               g = max(g, minimax(x) );
               x = next_brother(x);
          }
     }
     else {          // MIN on the move
          g = +maxSCORE;
          x = left_most_child(node);
          while (isNotNull (x) ) {
               g = min(g, minimax(x) );
               x = next_brother(x);
          }
     }
     return g;
} // end of minimax
```

Algorithm 1. Minimax - taken from [Plaat, 1996]

The algorithm given above shows the idea of finding the mini-max value only – in real programs, it is the move that leads to that outcome that we really need. However, changes needed for the algorithm to return the "best" move are simple and application-dependant – a sample solution will be given in the part concerning my program's operation.

For more complex games (in particular – chess) the game tree is too large to be browsed entirely – as mentioned earlier, it is explored only to a certain depth and a heuristic evaluation function is applied to determine values of leaves of resulting sub-tree. Nevertheless, the idea remains unchanged – only outputs of evaluation function are taken from a greater set.

The algorithm for determining the mini-max value of the root is very simple, and it was described by Shannon (1950) in his original work. The problem is the requirement for browsing through each and every node in the tree that renders it impractical due to exponential growth of the tree. The most popular solution is to employ an alpha-beta algorithm.

2.2.3.2 Alpha-beta

The idea beyond this algorithm is to use knowledge gathered at already analysed nodes to prune branches that cannot alter the mini-max value of the root.

Below is the tree from Fig. 7. presented again for convenience:

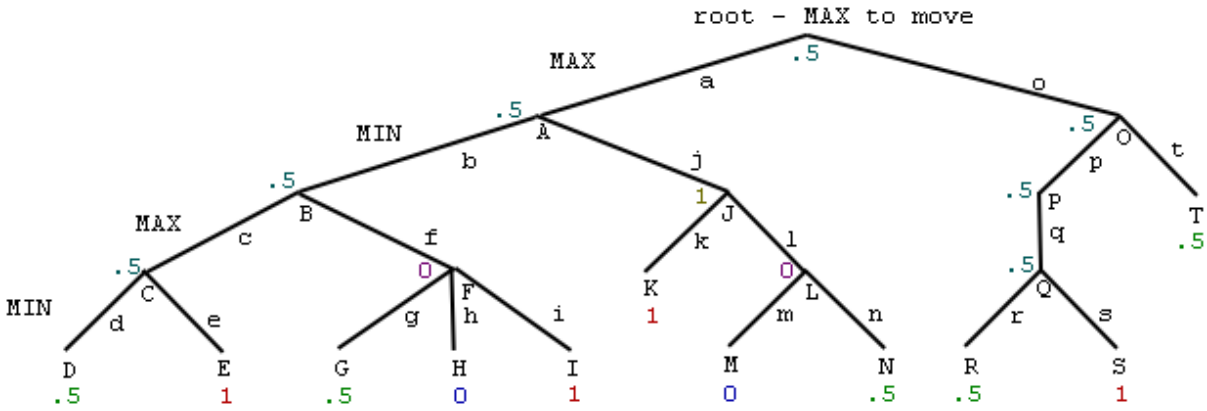


Fig. 8. (rep)

Assume that the tree is browsed according to the algorithm from the previous chapter (depth-first left-to-right). Once value of the node *C* is determined (0.5), the algorithm switches to the sub-tree starting with move *f*. The trick is that once move *g* is considered (it leads to a draw), there is no need to further consider moves *h* and *i*. MAX

has already guaranteed draw (move *c* in position *B*), so is not interested in move *h* (leading to his lose) and MIN is not concerned with his losing move *i* because move *g* in position *F* allows him to achieve draw. Therefore, moves *h* and *i* need not be explored – two less nodes need to be analyzed.

Another example taken from the above figure is when considering node *A*. MIN knows that move *b* guarantees him a draw, whatever happens next. So when he analyzes move *j* and finds that he immediately loses after MAX's response *k* he needs not to analyze move *l* anymore – the whole sub-tree *LMN* can be pruned.

Below is shown the same tree with parts that are pruned by alpha-beta algorithm coloured red:

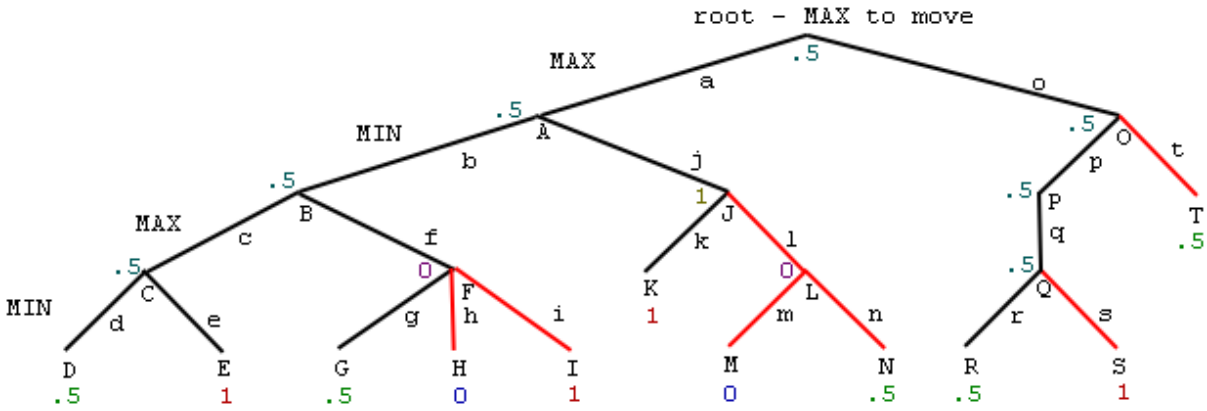


Fig. 9. Alpha-beta pruning applied to the mini-max tree

On the above example, 7 out of total 21 nodes are skipped – 33% improvement. In practical applications, the improvement is much more spectacular – in a typical chess middlegame position, alpha-beta searches about twice the square root of number of nodes browsed by a pure mini-max algorithm.

```
chSCORE alphabeta(chPOSITION node, chSCORE alpha, beta) {
     if (isLeaf(node) ) return evaluate(node);
     else if (isMaxOnMove(node) ) {
          g = -maxSCORE;  // maxSCORE is a huge
          //positive number
          x = left_most_child(node);
          while ( (g < beta) && (isNotNull (x) ) ) {
               g = max(g, alphabeta(x, alpha,beta) );
               alpha = max (alpha, g)
               x = next_brother(x);
          }
     }
```

```
        else {          // MIN on the move
             g = +maxSCORE;
             x = left_most_child(node);
             while ( (g > alpha) && (isNotNull (x) ) ) {
                  g = min(g, alphabeta(x, a, b) );
                  beta = min (beta, g);
                  x = next_brother(x);
             }
        }
        return g;
   } // end of alphabeta
```

Algorithm 2. Alpha-beta - taken from [Plaat, 1996]

The algorithm is very similar to mini-max – it includes two additional parameters sent to the function and cut-off tests. The parameters (*alpha* and *beta*) establish limits on the mini-max value (they change as tree traversal follows), and cut-off tests serve to skip branches that lead to parts of the tree that cannot influence mini-max value of the root.
In a simple alpha-beta implementation, the function would be called for the root with parameters *alpha = -maxSCORE* and *beta=+maxSCORE*.

It is easy to understand that performance of the alpha-beta algorithm depends on ordering of the moves. If moves are ordered in the order worst-first then alpha-beta will not be able to make any cut-offs at all – therefore, it will perform almost identically as pure mini-max (slightly slower due to some computational overhead). On the other hand, if the moves are ordered according to the best-first rule then alpha-beta is able to cut-off maximum number of nodes.

It has been shown by Knuth and Moore [Knuth, Moore, 1975] that the minimum number of nodes that must be explored in order to prove the mini-max value of the tree is

$$w^{\lfloor \frac{d}{2} \rfloor} + w^{\lceil \frac{d}{2} \rceil} - 1$$

where w – tree branching factor,

d – depth of the tree.

Therefore, a well-performing algorithm can search twice as deep as a pure mini-max (w^d nodes) in a given time. In other words, time complexity of alpha-beta can vary from $O(w^{\lceil \frac{d}{2} \rceil})$ in the best case to $O(w^d)$ in the worst case. The expected complexity however is closer to the best case than the worst – assuming that proper move ordering techniques are

used. The exponential complexity is preserved, but increase of practical possibilities due to alpha-beta pruning is enormous.

Techniques of move ordering are given below together with other techniques that enhance search performance.

By supplementing generic algorithms with various enhancements it is possible to improve their effectiveness drastically. Most notable enhancements are described briefly below.

2.2.3.3 Transposition table

Mini-max search is suited for tree traversal. However, in many games (e.g. chess) the search space is really an acyclic directed graph. Figure 3 illustrates, how the same position may occur after different moves' sequence:

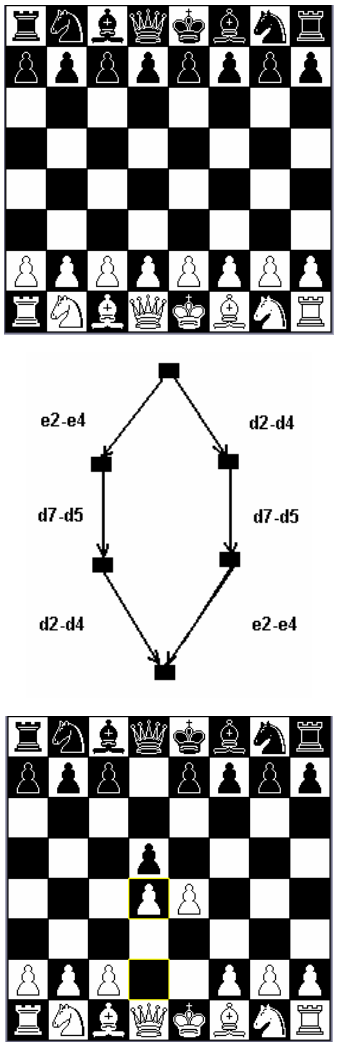


Fig. 10. – illustration of a directed acyclic graph in a game of chess

Transposition table can be used to store for reuse some information when certain node is reached again during the search process [Slate, Atkin, 1977]. It may be used to

avoid re-expansion of nodes representing positions identical to those already searched. If a position currently analysed happens to appear in the table, and the previous search was performed to at least the desired depth, then search results are collected from the table saving time on multiple expansion of identical sub-trees. Even if the data from TT does not cause a cut-off, it may still be sufficient to tighten the bounds (alpha, beta). If depth of the previous search is not satisfactory, then the suggested move (best in the previous search) may be used as the first move to be analyzed. Since search results for successive depths are strongly correlated, there is a large chance that it will remain the best choice for deeper search too, therefore dramatically improving move ordering.

All practical experiments prove the enormous usefulness of transposition tables in reducing search effort. The gain increases with increasing search depth. For 6 ply search (that is a very low depth by today standards) number of examined nodes is only about 5 % of the nodes count without transposition table [Lazar, 95, p. 17]. Obviously, the gain depends on the experimental set-up, however the configuration used by Lazar was representative of chess programs. Improvement from using transposition table in my own program was 52-95% (depending on position, at depths of 6-9 plies) over pure alpha-beta, that is on average lower than that of Lazar's, but still confirms its enormous usefulness. The smaller gain might occur due to the fact that my program inherently (due to the way its move generator operates) considers captures that look promising first, then normal moves (sorted using piece-square tables, described thoroughly in the next part of the work) and finally unpromising captures – so the plain move ordering is better than completely random.

Transposition table is usually implemented as a finite hash table ([Knuth, 1973]) – index into the table is created by performing some operations on the position (using the whole position as an index would require too much space – it is transformed to create a shorter hash key).

A sample implementation of a transposition table will be given in the part concerning my program.

2.2.3.4 Iterative deepening (ID)

The idea behind iterative deepening is to stop search at (small) depth d, then re-search to depth d+1 and so on until the target depth D is reached. Two advantages of such approach are:

- Easier time management – when program finds that it has exceeded its time limit for the analysed move, it can use the outcome of last completed iteration, and not bother with the unfinished one.
- Better move ordering – in most positions, move that performed best at depth d is likely to retain supremacy at depth d+1, thus reordering moves according to outcomes of the previous iteration may increase number of cut-offs by alpha-beta algorithm. Reordering is achieved easily when transposition table is used – it contains positions from the shallower search together with the score, thus the best-so-far move may be extracted.

The disadvantage of the ID is that it searches the same positions many times. However, it is the cost of the last iteration that dominates the total running time of the algorithm. A simplified analysis (assuming uniform branching factor) is given below:

$N_{d+1} = 35^{d+1}$ - number of nodes in a tree of depth d+1 (without root node);

$N_{1..d} = 1 + 35 + 35^2 + ... + 35^d = \frac{35^{d+1} - 1}{35 - 1} \cong \frac{35^{d+1}}{34}$ for d > 2 – total number of nodes in all trees from depth 1 up to d.

$$\frac{N_{1..d}}{N_{d+1}} = \frac{1}{34} \approx 3\%$$

The above result shows that the total number of nodes analysed for all depths up to d is only 3% of nodes analysed during the last iteration. More detailed analysis [Gillogly, 1978] says that it takes about 8 times more time to iterate one ply more to an odd depth, and about 3 times more to extend search by one ply to an even depth. Still, the time spent on re-searches (depth 1..d) is not that significant when compared to the time of iteration d+1. Practice shows that advantages of iterative deepening strategy far outweigh that disadvantage.

2.2.3.5 Killer heuristic

In many positions most moves are quickly knocked down, often by the same move. Remembering the move that caused most cut-offs at each depth (*killer* move) and trying it at the beginning whenever the same search depth is considered may drastically increase the number of cut-offs, therefore reducing effective tree size and saving time.

Effectiveness of this enhancement remains questionable – some researchers observed no benefits from it [Gillogly, 1978] while others claim huge improvement [Hyatt, 1983]. According to my observations, killer heuristic improves move ordering significantly

(from 16 up to 85%, position dependant, at depths of 6-9 plies, compared to pure alpha-beta) – some more data to prove it will be presented in the next part, devoted to my program. The improvement might be even greater if the original move ordering was completely random – however some basic move ordering (rough estimation of captures and piece-square evaluation) is done within move generator of my program.

2.2.3.6 History heuristic

A generalization of killer heuristic is history heuristic. It exploits fact that many positions differ only a little from each other (for example, by location of one minor piece), and a move that is good (sufficient) in a certain position (causes a cut-off or returns the best mini-max value) may still be good in another, slightly different one.

Schaeffer [Schaeffer, 1998, p. 8-9] describes operation of history heuristic in the following way:

> "*Every time a sufficient move is found, the history score associated with that move is increased. Moves that are consistently good quickly achieve high scores. Upon reaching an interior node, moves are ordered according to their prior history of success. Rather that using a random or knowledge-based approach for deciding the order to consider move in, a move's prior history of success can be used as the ordering criteria. The heuristic is experienced-based, acquiring information on what has been seen as an indication of what will be useful in future. In effect, previous search information is being accumulated and distributed throughout the tree.*"

Usage of history heuristic is not restricted to alpha-beta algorithm only. The idea may be used with different search algorithms, in different applications. An example from everyday life is when in a play of hide and seek our opponent often hides behind a wardrobe – it may be profitable to examine that place first when it is our turn to seek.

Experiments with my program show a 12-93% decrease in nodes-to-depth (depends on position and depth) – more detailed data is presented in the chapter concerning implementation of my program.

2.2.3.7 Null move heuristic

Null move makes use of the fact that a right to move is usually advantageous – in most positions it allows the side on move to improve its position. The basic idea is that if the MAX player may skip making a move (not a case in a real chess game – it is just an

artificial concept, useful in tree pruning), and the opponent is unable to run MAX into trouble (the search for a depth less by 2 or 3 plies than full-depth results in a value equal to or greater than *beta*) then the opponent probably will not be able to threaten MAX if a regular move is done. Thus, the sub-tree may be pruned and *beta* may be returned as a result of a shallower search.

Other uses of a null move heuristic have been found, as well. For example, if the null move search returns a mate score against the side to move, then there is a serious threat in the position, as that side gets mated if does not move. Therefore, the position should be analysed more thoroughly (i.e. the search from that position should be extended beyond the normal depth – so called mate-threat extension).

This method of forward pruning sometimes may overlook a danger on an occasion that would be revealed if full-depth search was employed. However, advantages (possibilities of using saved time to analyze deeper probably more relevant branches) outweigh the risk.

There are several conditions when the null move should not be applied. Definitely, it is illegal when the side on move is in check – skipping a move would lead to an illegal position then, with opponent having opportunity to capture the king. The null move method fails in some specific positions where each possible move leads to a worse position. These positions (called *zugzwang* positions) usually appear in late endgames, therefore this heuristic should be turned off in endgames of a very limited material, or additional steps should be taken to ensure correct zugzwang handling.

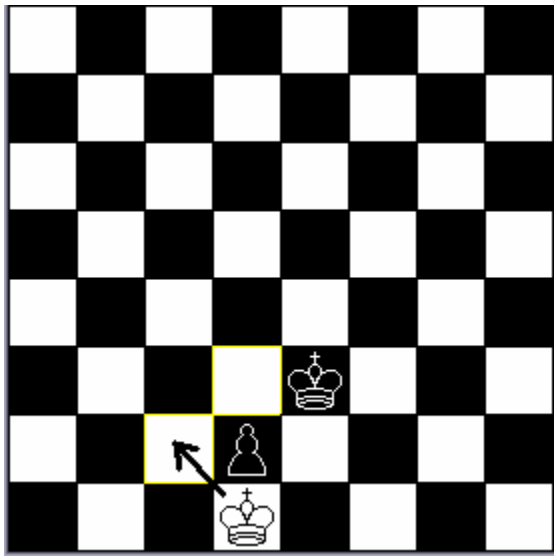


Fig. 11. White to move – a zugzwang position

An example of zugzwang position is presented on figure 6. Whte being on move is forced to move his king to C2 (indicated by an arrow). After that, black can play Ke2, securing pawn's promotion and win. However, if white could pass then black, having no threat to issue, cannot improve his position and win. Therefore, a "null" move (if was not against the game rules) would be the best choice for white here. In that kind of positions null move heuristic fails spectacularly. In general, it can make the engine blind to certain kind of tactics that bases on zugzwangs.

A null move technique is not cheap – if it fails (returns score below beta) then a significant amount of time needed to search the tree to a reduced depth is wasted. Therefore, its usage should be preceded by some tests whether it is likely to produce a cut-off. There are no universal rules about when to apply or avoid null move. Individual developers use their own techniques to maximize benefits from this heuristic in their own programs.

An enhancement of this technique is varying depth, to which the null move search should be performed. If the reduction (R) is fixed to be 1 (tree is searched one ply shallower than originally intended) then the benefits from this technique are negligible. On the other hand, reducing the search depth by 3 or more causes the search procedure to overlook too many tactics. The value considered as relatively safe and used by most programmers is 2 [Heinz, 1999]. Heinz in his paper described also a technique called adaptive null move pruning. The idea is to relate the reduction to remaining search depth. If the depth to search is large (basically greater than 6 plies, although Heinz also considers material that is left on the chess board) then the reduction can be greater (R = 3), if remaining depth is smaller or equal to 6 plies then reduction component is lowered to a safer value of R=2. According to Heinz, this scheme combines lower search effort due to greater reductions with accuracy of lower reduction scheme. The gain is noticeable (for obvious reasons) at search depths above 6. Heinz gives that the decrease in node count is about 10% at 8-ply search and increases to 30% at 12-ply search compared to the variant where a fixed reduction R=2 is used.

Another enhancement that allows null-move pruning deal well with *zugzwangs* is verified null-move pruning. In this technique the subtree is not cut-off immediately after a fail-high by a search to reduced depth. Instead, there is performed a verification search to a reduced depth with standard null-move pruning. If that verification search also fails-high then the subtree is pruned, otherwise it is searched to a full depth, as there may be some threat missed by the shallow search. This idea (with R=3, and verification search

performed to depth one less than originally intended) is described in [Tabibi, Netanyahu, 2002]. According to that publication, it leads to a larger tactical awareness of R=2 null-move pruning and smaller tree size, closer to that of R=3 scheme.

Null-move heuristic yields very large savings on the tree size, allowing for searching deeper by a ply or two – this is the technique, that allowed popular personal computers and minicomputers compete with machines using brute-force search with dedicated hardware.

This heuristic is useful in many games where it is profitable to have the right to move. However, there are some games (like Othello) where it is not the case, and null move pruning is not applicable.

2.2.3.8 Aspiration window search

A trivial approach is to call alpha-beta algorithm at the root with *alpha* and beta equal to –maxSCORE (ideally, minus infinity) and +maxSCORE (plus infinity), correspondingly. However, efficiency of alpha-beta algorithm increases significantly when the search window (interval <*alpha*, *beta*>) is smaller. Smaller window leads to greater number of cut-offs and time savings. Aspiration search calls the alpha-beta function at the root with a small window centred on the expected value.

If the returned mini-max value is from within the window then the accurate mini-max value of the tree has been found. If the returned value is greater than *beta* (*fail-high*) then it is even better – the mini-max value of the root exceeds our expectations. The trouble occurs when the search results in a value that is less than *alpha* (*fail-low)* – the expectations were too optimistic. A re-search must be done with the correct window, and if the program is forced to move (e.g. due to time constraints) then it has no sensible move prepared. Plaat and Schaeffer [Plaat, Schaeffer, 1996] have proposed a solution to this problem. They propose to restart the search in case of failing-low and use the transposition table values in a new iterative deepening search. A new best move should be found quickly and with less search effort. However, some practitioners on Computer Chess Club Forum have questioned usefulness of this technique.

2.2.3.9 Minimal window search

Reducing the search window to minimum (*alpha = beta-1* for integer-valued scores) seems insensible as re-searches are required to find the mini-max value. Alpha-beta algorithm always returns a bound in such case. However, since it is a bound that is needed

at most of the interior nodes, the highest number of cut-offs guaranteed by the minimal window can be appreciated. Most high-performance chess programs use the concept of minimal window search in one way or another. More about minimal window search techniques can be found in [Plaat, 1996].

2.2.3.9.1 Principal Variation Search (PVS)

The idea of PVS has been discovered by Marsland and Campbell [Marsland, Campbell, 1982]. It is one of the most popular algorithms in modern chess programs.

Each minimal window search returns almost immediately. Therefore, some re-searches are necessary to determine the exact mini-max value of the position. However, sometimes a bound is all we need. Specifically, with perfect move ordering all moves outside the principal variation are worse than the principal variation. The exact value of those moves is not important – all that is needed is a proof of their inferiority, and a minimal window search is the quickest way of obtaining that proof. Therefore, only the principal variation is searched with a full alpha-beta window and the other variations are analysed using quickly-returning minimal window search only.

However, the move ordering is never perfect, so sometimes the minimal window search will return a fail-high (value greater than beta). In that case, a re-search with full alpha-beta window is needed. The benefits of principal variation search (PVS) arise because those time-consuming re-searches are rarely needed (due to hopefully good move ordering), and are outweighed by time savings due to minimal window search.

2.2.3.9.2 MTD(f)

This algorithm uses minimal window searches to converge towards the true mini-max value of the position. It does not use any other windows but minimal. It makes an educated guess on the starting value of mini-max value of the position (e.g. the value from the previous iteration) and uses the results of the minimal window alpha-beta search (fail-low or fail-high) to adjust the window in the next search. Namely, in case of fail-low the window is increased and in the case of fail-high it decreases. After some (hopefully small) number of calls to alpha-beta with minimal window the algorithm settles on the true mini-max value of the position.

For MTD(f) algorithm to be efficient, re-exploration of already visited nodes must be avoided. Therefore, the implementation of alpha-beta algorithm must store the

encountered nodes. For these storing purposes a transposition table of reasonable size is sufficient.

The detailed description of the algorithm and its experimental performance are given in the original paper concerning the MTD(f) algorithm: [Plaat, 96]. According to the practitioners' posts on the Computer Chess Club Forum this algorithm is really efficient, but is sensitive to interactions with transposition table and other sources of search instabilities and performs best when the evaluation function is not very fine grained (as it requires more iterations to converge with such evaluation).

2.2.3.10 ProbCut (and Multi-ProbCut)

This technique, found by Buro, bases on a shallow null-window searches to find moves that will fall outside the search window, with certain probability. These moves can be skipped during the full-depth search. Buro's original contribution was to use means of statistical analysis (linear regression) to determine the best values of search windows.

The idea works as follows: before examining the position by a search of depth d, it is first examined by a shallow search of depth d' < d. The result of the shallow search v' is used as an estimate of the result of the deep search v. If the estimate falls outside the search window (alpha, beta) then the upper bound (beta) is returned and the deeper search is skipped, leading to time savings. However, if the estimate remains within the window, then a full depth search of that subtree is made. The cost of the shallow search in that case is supposed to be small, and the cost of shallow searches followed by deep searches should be outweighed by savings due to avoiding deep searches, leading to overall time gain. The relationship between the deep search value and the shallow search value follows a linear model:

$$v = a*v' + b + e$$

where v – full-depth search value (search depth d),

v' – shallow search value (search depth d'),

a, b – parameters,

e – normally distributed error with mean 0.

The details concerning the ProbCut algorithm (including procedure for determining the parameters) are described in [Buro, 1995].

The Multi-ProbCut enhances ProbCut by allowing different parameters for different stages of the game, using more than one depth pair (d, d') and internal iterative deepening for shallow searches.

The ProbCut method has not proven more effective than the null move heuristic in chess (it has in other games, e.g. Othello), however its improved version (Multi-ProbCut) seems promising, and has been successfully employed [Buro, 2002].

The problem with forward pruning methods (e.g. ProbCut, null move and other techniques basing on shallow searches) is that they may omit good moves that look unpromising at the first glance, or overlook blunders beyond their shallow search horizon.

2.2.3.11 Quiescence search

The evaluation function is applicable in static positions only. It does not consider hanging pieces or forced mates. Therefore, a fixed-depth search suffers from the *horizon effect* – a static evaluation function cannot correctly evaluate tactical features of the position. Therefore, branches leading to positions where checks or beneficial captures are present are explored more deeply (by considering captures and checks only) until a quiet position is reached – then, it is assessed by the evaluation function. The quiescence search is follows the alpha-beta rule, however the possible moves can be only captures or checks (although non capturing checks may be as well excluded from this phase). Moreover, it is assumed that a player can make any non capturing move (*a standing pat*) – in that case the static evaluation of the position is returned. The *standing pat* expresses the rule that captures in chess are not obligatory.

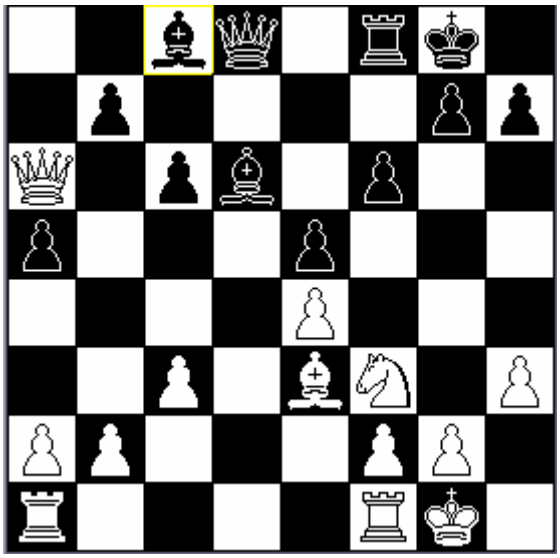


Fig. 12. Position with hanging piece

The above diagram (figure 7) shows an example of a position that would not be evaluated correctly without quiescence search. If that position happened to appear at the leaf of the search-tree, the evaluation function would return a huge advantage of the White,

since it is a rook ahead. However, it is Black's turn now, and he can capture the queen with a pawn. A quiescence search, that considers only captures (and possibly checking moves) would reveal that possibility, and properly return an evaluation showing huge Black's advantage (it will be a queen against a rook ahead soon).

The sequence of captures in most positions is short, however quiescence search is called at all leaves of the search tree, therefore the quiescence nodes can easily dominate the total number of nodes. There are two approaches to limiting the tree growth due to quiescence search: Most Valuable Victim/Least Valuable Attacker (MVV/LVA) rule and Static Exchange Evaluation (SEE).

MVV/LVA rule sorts the generated captures according to the captured piece value (so queen goes first, pawn is last), and in case when a given opponent's piece can be captured by more than one friendly piece, the capture with the least valuable one is tried first (therefore, if opponent's rook may be captured by either a bishop or a pawn, the capture with a pawn is tried first). This technique serves to sort the captures according to a rough estimation of expected material gain. This is a rough estimation (for example, QxQ capture is considered before PxR, although the queen may be defended), but MVV/LVA scheme can be easily implemented and works very fast.

SEE attempts to evaluate a sequence of captures without any calls to the search. It is much more computationally extensive, but gives better estimation of the expected material gain, therefore allowing better move ordering. For example, a capture QxN cannot be excluded using the MVV/LVA even when the knight is defended and the better move seems to be RxP, the pawn being not defended. SEE would properly evaluate the QxN as a losing capture and move it to the end of the capture list, or even allow it to be pruned.

Some old programs tried to completely replace quiescence search by SEE. This approach was not successful because lead to too many blunders. Due to computational limitations SEE implementations only consider sequence of captures to a single square, and do not consider pins, overloaded pieces etc. – as a result, values returned by SEE are good for move ordering, but not good enough to replace quiescence search completely.

SEE allows some static pruning tricks (for example, pruning of captures that appear losing without further analysis) and gives much better move ordering than MVV/LVA scheme (leading to savings in node count) but is more complex to compute (leading to reduced number of browsed nodes per second). As far as I know, none of these schemes is dominant in current chess programs, but my impression is that more and more engines shift toward SEE.

2.2.3.12 Enhanced Transposition Cutoff

Plaat [1996] suggested a technique that yields increased benefits from using transposition table. The idea is to avoid unnecessary browsing of the parts of the tree below the given node if it can be verified that one of the children of that node returns a value sufficient for the cutoff. For example, consider the drawing from the below figure 8.

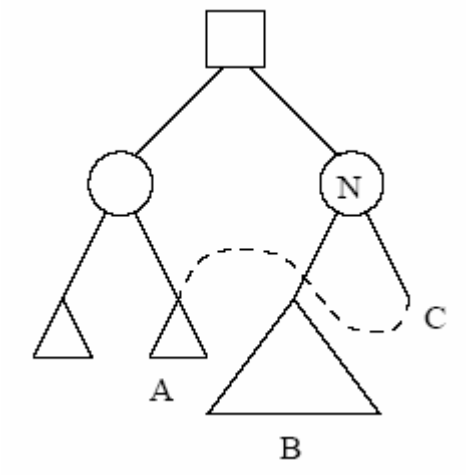


Fig. 13 Transformation of one part of the tree to another (taken from Plaat [1996])

The node under consideration is node N. The used move ordering indicates the sub-tree rooted at node B to be searched first. However, it is possible that the sub-tree starting with node C transposes into the previously searched sub-tree rooted at A. If the value obtained from searching the sub-tree A can be retrieved from the transposition table and it is sufficient to cause a cut-off, then the whole sub-tree B can be skipped, leading to (possibly significant) savings.

Therefore, by looking up in the transposition table all the children for a given node before actually considering any sub-tree of that node it is possible to avoid expanding that node. This saving in node count can be offset by introduced new complexity in node processing. A solution to that problem was given by Plaat [Plaat, 1996] in his work where he states that performing ETC at interior nodes that are more than two plies above the leaf nodes brings most of the ETC benefits, at only little additional computational effort.

Moreover, looking up all children positions for the given node cancels out the possible computational savings due to incremental move generation. However, the Plaat's solution should also solve that problem.

Plaat's experimental results indicate a reduction of node count increasing from about 8% at search depth of 5 plies to over 20% at depth 9.

2.2.3.13 Extensions

Extensions mean extending potentially interesting lines beyond the normal search depth. The sub-tree following the extended move is searched to increased depth. Deeper search is more powerful, and should be used when there are indications that the normal-depth search may be too shallow to discover possible threats. The drawback of using extensions is increase of the number of nodes searched – the usage of extensions should be carefully analysed and limited, to avoid uncontrolled search tree explosion. A method helpful at controlling extensions is usage of so called *fractional plies*. Using the concept of fractional plies one can assign the extension a fractional number – for example 3/2. Therefore, after applying that extension the search will be one ply deeper, and the value ½ will propagate down the tree. When another extension (for example by ¾ ply) is applied later somewhere in that sub-tree, the search is extended by another ply (as ½ + ¾ = 1¼ > 1) and the value ¼ propagates further etc. Therefore, although depth of the search tree is a natural number, the extensions (and reductions as well) can be measured in smaller units.

Using extensions it is possible in some positions to find a mate in tens of moves, although the fixed-depth search reaches only several plies.

An extension that is probably the most popular in chess programs is “out-of-check” extension. It is triggered when the side to move moves out of check. All sub-trees starting with “out-of-check” moves are analysed to a greater depth. The reasons for using this extension include (to name a few):

- a check can lead to a mate (the mate can be therefore found faster than it would be without the extension),
- even a useless check requires appropriate response, and it consumes two plies, postponing possible important discovery (or even throwing it beyond the search horizon)
- a check may go together with a discovered attack or double attack and lead to a material gain that might remain undiscovered if appeared close to the tree leaves.

Other popular extensions include (let the names speak):

- recapture extension,
- extension due to the pawn moving onto $7^{th}/2^{nd}$ rank,
- single response extension,
- mate-threat extension,
- singular extension.

The last one (triggered by one move being significantly better than all others) has been first described in [Anantharaman et al., 1988] and was introduced to the chess-playing computer Deep Thought, the predecessor of Deep Blue that in 1996 beat Kasparov - the human World Champion

There are no rigid rules concerning usage of extensions, and from posts on the Computer Chess Club Forum I deduce that almost any chess programmer uses (apart from the most popular also) extensions of his own, original concept.

2.2.3.14 Reductions

Reductions, being opposite to extensions, serve to reduce search depth for sub-trees that are not likely to consist principal variation. For example, in presence of a material-winning move all moves that do not win material and do not increase pressure on opponents king and do not attack opponents pieces are candidates for reduction (this example is based on notes published by Ed Schroeder, author of a premium chess program Rebel [Schroeder, 2003]).

Good reductions can significantly reduce the search effort, but care must be taken to avoid reducing important lines, as it may lead to tactical mistakes and overall decrease of program's playing strength.

2.2.3.15 Futility pruning

When the target search depth is reached, the quiescence search is usually used to avoid static evaluation of dynamic positions that can be misleading. Quiescence search browses through possible captures, applying the evaluation function only to quiet positions. However, at any moment the side on move may decide to avoid capture, if its static score is already greater than beta (meaning its position is already to good to be accepted by opponent). Beta is then returned, and cutoff occurs. It is also possible that the static score is greater than beta at the moment the quiescence search is entered, therefore avoiding searching captures at all.

Let's consider a typical case of a linear evaluation function considering material balance and positional merits. The following scenario occurs at the leaf node of full-width search: side to move has its material balance well below alpha, and it cannot restore it close to alpha by capturing opponent's piece. It has basically two possibilities (excluding degenerate cases like giving checkmate): give check (as it can allow capture of opponent's piece the next move) or count on positional rewards to cover lack of material. However, if

the side on move has no way to give check, and even by performing capture of strongest possible opponent's piece it cannot bring the score up to (alpha – maximal_positional_score), then beta cutoff will unavoidably occur when quiescence search is entered. In such case there is no point in searching that node any further – all moves are futile, and can be pruned.

The above scenario shows a case when the futility pruning is theoretically sound – it does not degrade search quality in any way. However, if the requirement about maximal_positional_score is weakened (a smaller margin is used instead) then it transforms into a truly selective pruning technique. It can introduce some errors in tactical positions, but generally it should save enough nodes to compensate for occasional tactical mistakes.

The further attempts to employ similar techniques at nodes located higher in the tree (pre-frontier and pre-pre-frontier nodes) are described in [Heinz, 1998].

2.2.3.16 Lazy evaluation

The idea is to replace the complex evaluation function at horizon depth or in quiescence search and replace it with a simple one that involves only a few components and executes much faster. This "lazy" evaluation function counts the material and maybe also some basic positional features that contribute most to the evaluation of the position. If this "lazy" value is greater (by a certain margin) than *alpha* then it can be used relatively safely as a replacement of the full evaluation function result, saving a lot of time.

This concept can turn out to be dangerous, as practice shows that not enough careful implementation of this technique may cause the program to play disastrously. Sensible play can be guaranteed by a proper choice of the margin value.

2.2.4 Disadvantages of alpha-beta procedure

Some aspects of a game of chess are difficult to incorporate into an alpha-beta based framework. They include:

- A single number is used for evaluating a position – reduction of the enormous complexity of the game to just one number seems to be an oversimplification. One number cannot describe the difficulty of the position, uncertainty of the evaluation due to its incompleteness and inexactness etc.
- Selective search – level of selectivity achieved by modern programs remains low, and there are positions where brute-force is just not enough.

- Mini-max-only based search does not consider difficulties that the opponent may face in practice when trying to find the best line. In some positions higher probability of winning might be achieved by playing not the line that seems best according to the mini-max rule, but the one that requires more subtle play from the opponent and maximizes his chances of making a mistake.
- In openings the move that is considered the best by the program is usually similarly considered by the human grandmaster, and has been thoroughly analyzed in literature. Practical chances of winning may be increased when second or third best move is picked from time to time – the human opponent may be taken by surprise.

2.2.5 Alternatives to alpha-beta

Alpha-beta algorithm is not the only known algorithm for efficient choice of best moves. Some other algorithms are possible, too.

2.2.5.1 Berliner's B*

Berliner [Berliner, 1979] formulated a new algorithm that does not enclose the evaluation of a chess position within one number. It uses two values to describe position – a pessimistic and optimistic bound. Those bounds are adjusted during exploration of the tree. The algorithm finishes when the pessimistic bound of the best move is greater or equal to the optimistic bound of all other moves. The unpromising lines of play are dropped quickly, and the rest is searched until the best move is established. This approach is closer to the human way of thinking than alpha-beta, and its inherent ability is to analyse deeper the interesting branches and refute quickly the unpromising ones. Variable-depth search is a way to go since there is never enough computational power to cover all branches to satisfying depth. With alpha-beta this variability is added artificially by means of various extensions, reductions and forward pruning techniques.

The distance between the two bounds (pessimistic and optimistic) indicates the uncertainty about the move. The larger the interval between them, the greater uncertainty of the evaluation, while a small interval suggests a high confidence. This addresses one of the drawbacks of the alpha-beta algorithm – one number being the mini-max value of the position gives no indication about the uncertainty due to imperfections of the evaluation function. As [Junghanns, 1994, p. 20] states, the interval within the bounds can be viewed as a uniformly distributed probability density function.

Since the exact mini-max value needs not to be found, B* may expanded fewer nodes than alpha-beta in order to determine the best move.

Plaat [Plaat, 1996, p. 35] states that the problem with this algorithm is determination of reliable heuristic bounds.

2.2.5.2 Conspiracy numbers

Conspiracy numbers algorithm is a mini-max search method that does not require application-dependant knowledge. It gathers the information required to determine how likely it is that exploration of a sub-tree will result in useful outcome. This "likeliness" is measured with the conspiracy numbers. Conspiracy numbers indicate the minimum number of leaf nodes that must change their value (by being explored deeper) to alter the mini-max value of the sub-tree. Therefore they serve to measure the difficulty of changing the current mini-max value of the node.

More detailed analysis of that method is given in [Lister, Schaeffer, 1991]. Below I will only present an example illustrating the concept of conspiracy numbers taken from that work.

Figure 3 presents a small mini-max tree (MAX player is on move at the root). If we want the value of the root to be 2, only one leaf node (J or K) has to change its mini-max value (as a result f a deeper search of that node). Also if both nodes F and G changed their values to 2 simultaneously it would change the root's value to 2. Therefore F and G form a *set of conspirators* (meaning that they both have to *conspire*) for changing the root value to 2. Each node J and K forms a *minimal* set in that case. Number of elements of the minimal set is called the *conspiracy number (CN) of a node for the given value*. Therefore, on the example the CN of node A for value 2 is one.

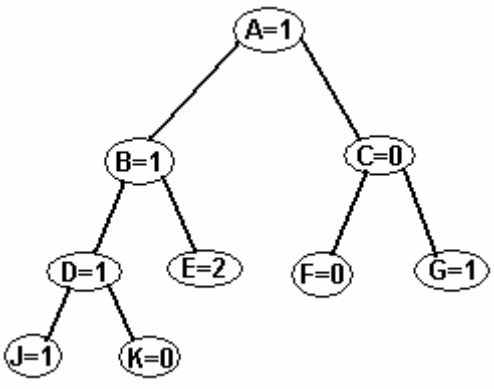


Fig. 14. Sample minimax tree

The search is controlled by a *conspiracy threshold* (CT) – it is a minimum number of conspirators required for the node to be considered stable (as conspiracy numbers represent the difficulty of changing the value of the node).

The algorithm starts with likely root values from interval $[t_{min}, t_{max}]$, with the assumption that the proper value t_{root} lies within that interval. A value is called likely if the CN for that value is smaller than the threshold CT. Then the range of likely values is reduced down to just one value. Once there is only one root value left, it is assumed that further search will not alter its value. Higher threshold of the root leads to more complex search, but also gives greater confidence about the final root value.

3 Implementation of my own chess program

In this part of my thesis I will describe implementation of my chess engine named “Nesik”.

3.1 General information about the implementation

That paragraph describes some general aspects of the implementation of my program, Nesik. It features:

- Bitboard board representation,
- Alpha-beta searching with aspiration window,
- Iterative deepening,
- Quiescence search,
- Transposition table,
- Killer heuristic,
- History heuristic,
- Adaptive null-move forward pruning,
- Futility pruning,
- Several extensions.

Moreover, it supports time control, can think on opponents time (pondering) and uses Winboard protocol for communicating with graphical user interface. It correctly recognizes rules of chess (including threefold repetition and 50-moves rules). Having played over one thousand of games (in test games, chess-server play, external computer chess tournaments) it has proved itself a reliable and quite solid player, of a good club player strength.
Below I present the more detailed description of the program.

3.1.1 Assumptions

I assumed that the program would:

- use modern and advanced chess programming techniques,
- achieve as high playing strength as possible (comparable with leading Polish chess programs),
- be easy to maintain and develop (for example in order to include self-learning code).
- use third-party graphical user interface

The choice of these assumptions lead to following decisions:

- C++ as the programming language
  - Good readability and maintainability,
  - High performance of the code,
  - Object programming support
- bitboard-based architecture
  - One of the most popular board representations,
  - High performance,
  - Good perspectives of taking advantage of new 64-bit architectures.
- alpha-beta search algorithm
  - best performance of all available search algorithms
  - well known and understood
- support for Winboard protocol (version 2)
  - one of two most popular protocols for chess engine-chess GUI in use

An important aspect of chess programming is maximizing code speed (usually measured in nodes per second). Although raw NPS speed is not the only determinant of playing strength (not even the most important, as programs with intelligent algorithmic ideas leading to savings in total node count or finely tuned evaluation function may offer better performance despite of lower NPS factor), it is important enough to force certain design and implementation choices. My program is not supposed ever to become a commercial program, neither is aiming at successes in international tournaments, but is still supposed to achieve a reasonable playing strength. Therefore I had to make some choices that would satisfy all needs: for engine's speed, for maintainability, for development speed and others. An example of such choice is resignation from using advanced object programming techniques (like virtual functions) for the purpose of maximizing code speed.

3.1.2 Programming environment

The engine is written entirely using Borland C++ Builder. The reason behind using that particular environment is that it was the one I was best familiar with, and offered a reasonable integrated development environment with debugger at hand.

I use some Borland's specific code (for example AnsiString data type as a replacement for char* and TThread base class to implement threading), however it should be reasonably easy to convert it to pure platform-independent C++.

Although code efficiency was not the major concern while creating this application, some attention to performance bottlenecks had to be paid. The profiler tool used for identifying performance issues was AQTime 3, evaluation version, by AutomatedQA Corp.

3.1.3 Description of the program's operation

Currently the engine uses two threads. One thread is used to implement ssynchronous communication with user (or a GUI). The communication follows the Winboard protocol (version 2). Description of the Winboard protocol is available at http://www.tim-mann.org/xboard/engine-intf.html (last accessed: 17th of July, 2004).
The second thread is the actual engine thread that performs chess computations. I decided to use another thread for communication in order to avoid polling the standard input during search. This approach is more suitable for parallel search implementation, that is not implemented nor planned in the nearest future, but soon it will be a must for all chess programs, so it will be nice to be prepared.

3.1.4 Program structure

Software engineering was not the main issue when design and development of the program took place. Most effort was put into understanding, analysis and implementation of chess-oriented algorithms and techniques. Certain aspects of modern software engineering were preserved in a simplified form (e.g. classes to achieve certain level of isolation between program's basic components), but proper object-oriented design was skipped for three reasons:

1. it was not required as the project's complexity lied not in its sophisticated assumptions and requirements, nor its exceptionally large size, but in designing and developing good algorithmic solutions.
2. advanced object-oriented code using virtual functions results in slower executables, due to need for late-binding at runtime – unnecessary slowdown is not welcome in a chess engine.

Below are presented diagrams representing classes used in the program. They do not represent actual classes, but model the engine's structure and behaviour.

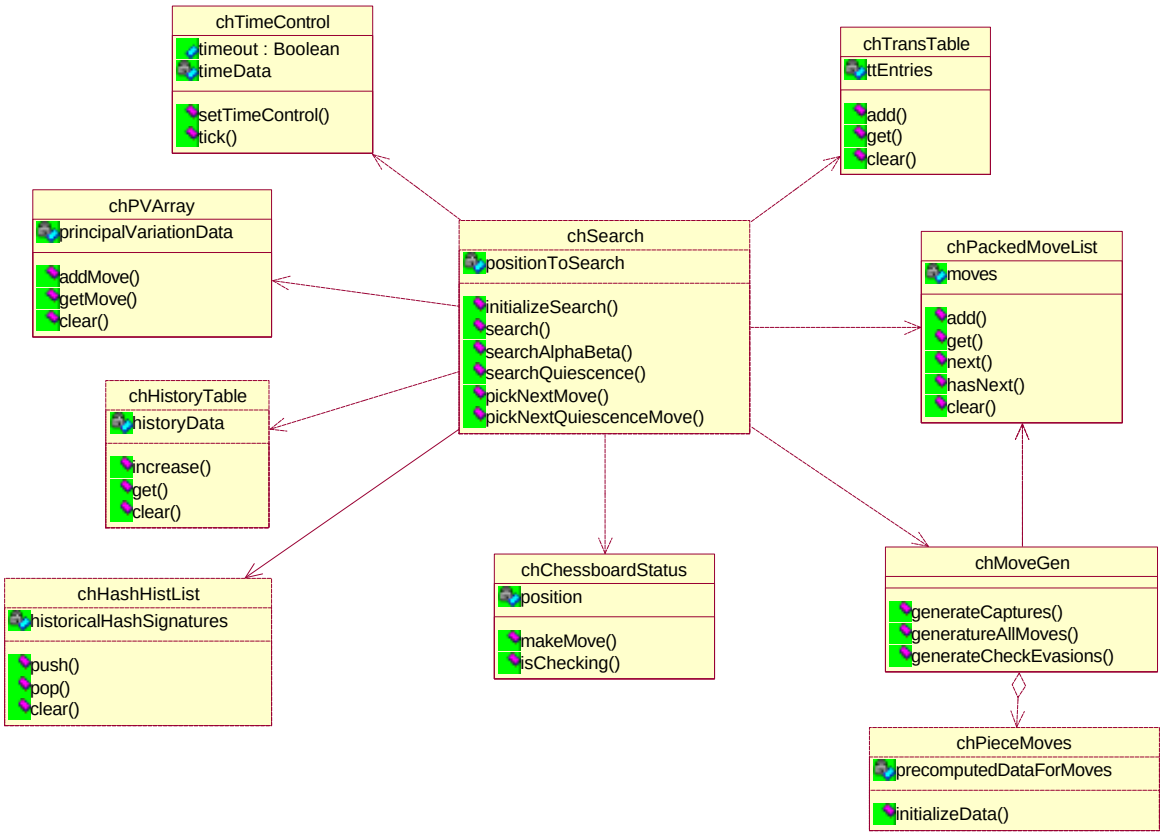


Fig. 15. Diagram presenting classes involved in the search process

Description of the parts from the above diagram:

- chSearch – main unit that performs actual search for the best move.
- chTimeControl – implements a chess clock – its method tick() is called by the search routine in regular intervals, and if the time is running out the *timeout* flag is set.
- chPVArray – implements attributes and methods for storing the best line during search.
- chChessboardStatus - represents a chessboard, supplies methods for performing moves (and some supplementary functions).
- chPieceMoves – initializes and stores precomputed data that are necessary for move generation.
- chMoveGen – implements methods for generating moves.
- chPackedMovesList – implements a data structure for storing and retrieving move data.
- chTransTable – represents the hash table and access methods to implement

transposition table.

- chHistoryTable – represents the history table and methods that operate on it.
- chHashHistList – stores hash signatures of positions that appeared in the game, either during actual course of the game or at some depth in the search tree. Used to detect threefold position repetitions.

The search process is driven by engine framework:

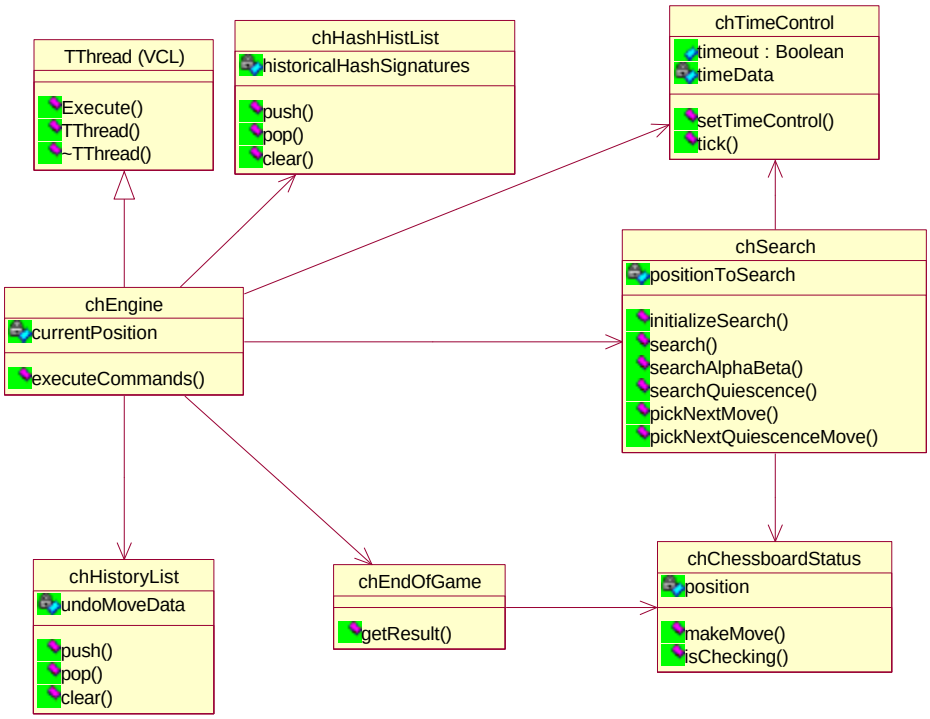


Fig. 16. Diagram presenting the classes creating the framework for the actual search process

Description of the parts from the above diagram:

- TThread – an abstract VCL class that allows its inheriting classes to run in separate threads
- chEngine – unit that coordinates program's input/output and the thinking process, inherits from VCL class TThread to allow a new thread for searching.
- chHistoryList – stores data about past moves that is required for dealing with threefold repetition rule and undoing moves.
- chChessboardStatus - represents a chessboard, supplies methods for performing moves (and some supplementary functions).
- chEndOfGame – a simple class that helps testing whether the game is ended due to checkmate or stalemate.

The last diagram presents the opening book and learning framework:

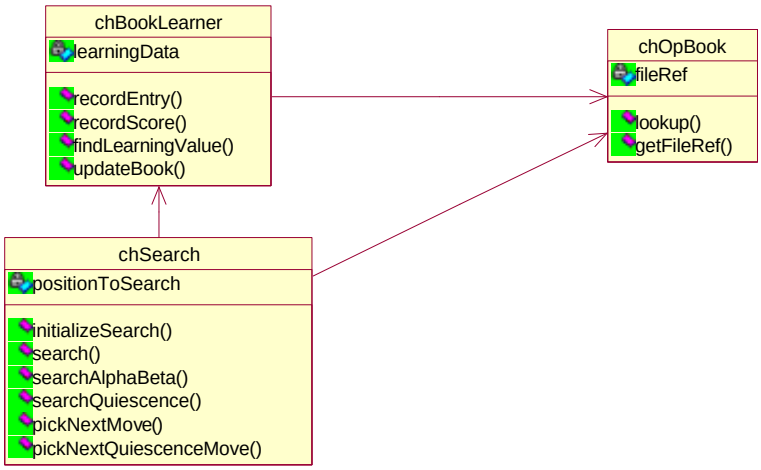


Fig. 17. Diagram presenting the classes creating the framework for opening book lookup and learning

Description of the parts from the above diagram:

- chOpBook – services the opening book lookup,
- chBookLearner – records required data during the game, and updates the book with learned data at the end of the game.

The prefix “ch” in front of class names stands for “chess” – it is there to make it easier to differentiate classes created for the purpose of chess engine from any others, that might become part of the project.

The whole source code consists of about 14,000 lines of code. It is splitted into following files:

- basic.cpp, h : declarations of some basic structures (like chM*ove*, chHistoryListEntry) and basic operations (like conversion between natural move representation in form of a structure and its packed form of 32-bit unsigned integer)
- board cpp, h : contains chChessboardStatus class that represents chess board and contains some methods to operate on it (like makeMove()).
- debug.cpp, h : several functions used during development process in order to simplify debugging (e.g. for printing bitboards into screen in a readable format).
- declarations.cpp, h : declarations of all constants, typedefs and enums that are used

in many files; functions initializing pre-computed masks and arrays (e.g. a bitboard mask representing central squares, or array chColumn[8] containing bitboards representing all columns of the chessboard).

- end_cond.cpp, h : contains a simple chEndOfGame class, that recognizes such end-of-game conditions like checkmate or stalemate.
- engine.cpp, h : contains a synchronization method (operating as a critical section) used to coordinate I/O and searching threads; contains chEngine class that manages the search process and coordinates its operation with I/O.
- eval cpp, h : contains chEvaluator class, that implements the core of evaluation function.
- history.cpp, h : contains chHistoryTable class, that stores and manages data used in history heuristic.
- lists cpp, h : contains chPVList (principal variation storage and retrieval), chPackedMoveList (stores generated moves in packed form, that is used in the search process) and chHistoryList (used for storing game history – positions and board status from past moves).
- move_gen.cpp, h : contains chMoveGenerator class that performs move generation.
- nesik.cpp : contains the main() function that initializes the search thread, waits for user's (or Winboard) input and calls appropriate engine methods via synchronizing method.
- op_book.cpp, h : contains the chOpBook class that is used to interface with opening book.
- pawn_eval.cpp, h : contains the chPawnEval class that evaluated the pawn-related components of evaluation function, and manages its own pawn hash table.
- pc_square.h : contains arrays that create piece-value tables used in move generation and evaluation.
- piecemoves.cpp, h : contains the chPieceMoves class that computes and stores pre-computed data used for move generation (like knight[64] – a set of masks representing all possible destinations of a knight from a given square).
- ponder cpp, h : contains the chPonderer class that manages the pondering process.
- search cpp, h : contains the chSearch class, that implements the actual search and supplementary methods.
- timecontr cpp, h : contains the chTimeControl class that is used for managing time.

- tt.cpp, h : contains the chTransTable class that implements transposition table.
- utility.cpp, h : contains some utility methods, mainly dealing with user interface (like parsing moves).

3.2 Description from user's perspective

Nesik is a console application that can be run on any 32-bit Windows platform. The package consists of five files that should be kept in the same directory:

- NESIK.EXE – core of the program; requires Borland's library file VCL50.BPL presence in the system (%windows\system32\ or program's current directory),
- NESIK.INI – configuration file,
- NESIK.RND – file with random numbers, used for opening book access,
- NESIK.BK – main opening book file,
- NESIK.BKI – index file for the opening book.

3.2.1 Text-mode interface

On its start, Nesik offers only an empty console window. After entering "xboard" from keyboard it displays some configuration data it has read from the "NESIK.INI", and awaits further commands (that form a subset of Winboard protocol 2 commands). Description of the Winboard protocol is available on the Internet (e.g. on the home site of the Winboard program: http://www.tim-mann.org/xboard/engine-intf.html).
The supported move format notation is algebraic notation (example of a move: d2d4, g8f6, a7a8q), and here is the list of supported commands:
xboard, protover, new, quit, random, force, go, playother, level, st, sd, time, otim, move, ?, ping, result, setboard, hard, easy, post, nopost, rating.

Fig. 18. Program's screen after accepting the initial "xboard" command issued by operator

3.2.2 Graphical interfaces: Winboard, Arena

Any program implementing the Winboard protocol can be used to provide user with a graphical chessboard and a nice interface for issuing commands to the engine. These programs can run Nesik as a separate process, redirecting its standard input and output to take control over it. Two free popular programs that can be used with Nesik (and have been tested with it) are Winboard (http://www.tim-mann.org/xboard.html) and Arena (http://www.playwitharena.com).

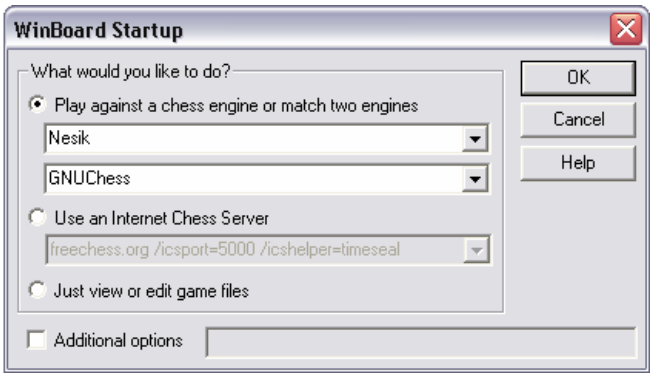


Fig. 19. Winboard startup screen

The simplest way to run Nesik under Winboard is to type its name on the Winboard's startup screen (assuming its files are placed in the Winboard directory). Another, more general way of running Nesik (e.g. when it is placed in different directory) is by editing Winboard's configuration file – it is described in Winboard's manual.

Fig. 20. Winboard – a game between two engines

Another popular program supporting Winboard protocol is Arena. It is more powerful and customizable than Winboard, including built-in capability of managing tournaments with many engines.

Fig. 21. Arena – a powerful interface that can be used with Nesik

Nesik can be ran under these, and some other interfaces as well without problems in a way that is standard for these programs (their manuals contain required details).

An important added value of these interface programs is their capability of connecting to Internet chess servers – in an automated way, the program can fight against thousands of players (both human and computer) from all around the world, of different rating, playing style etc. Nesik has played many hundreds of games on Free Internet Chess Server (FICS, http://www.freechess.org), achieving ratings of a strong club player (2100-2300 points).

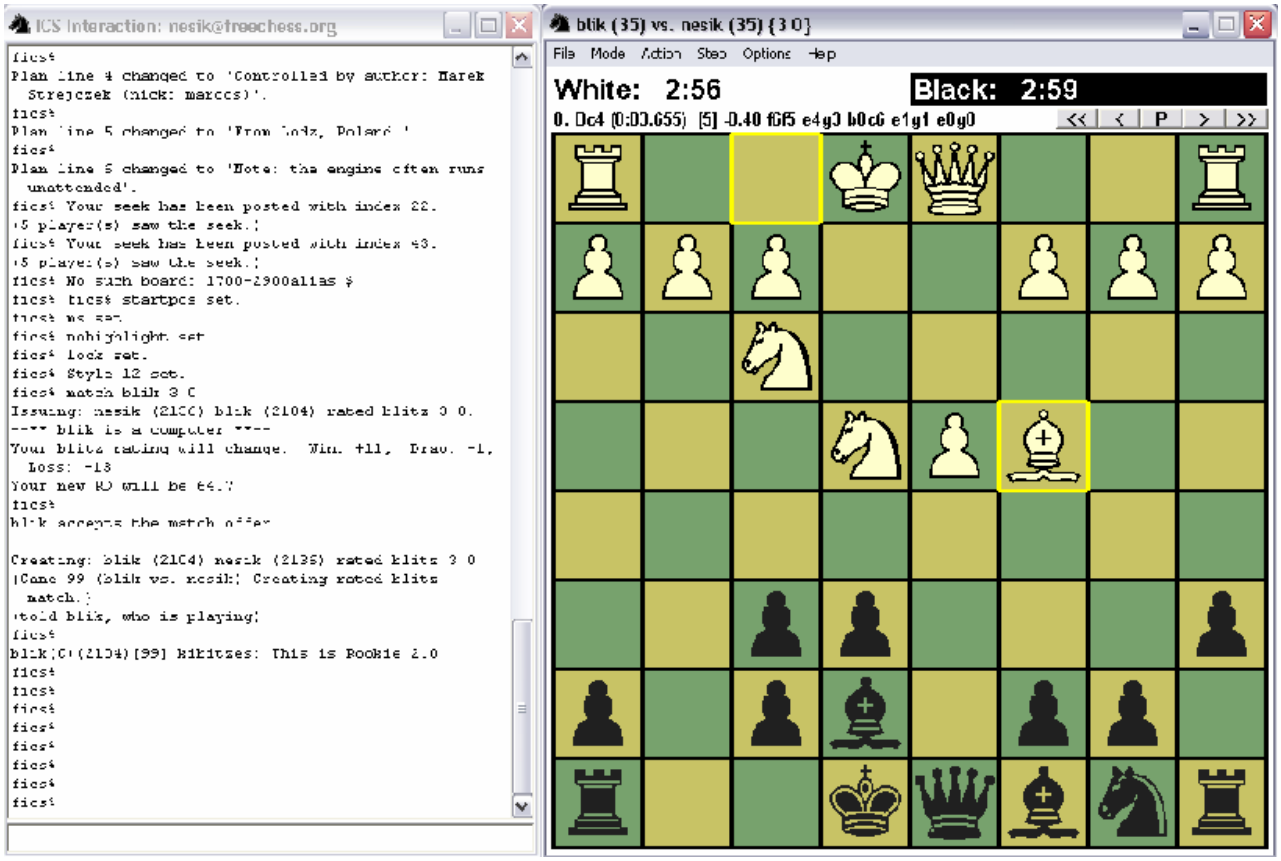


Fig. 22. Nesik playing over the Internet against Rookie on FICS

3.3 Chess programming aspects of the implementation

In that paragraph I describe some techniques that I used in my actual implementation of Nesik.

3.3.1 Move generation

Nesik uses bitboards to represent chess position. This approach determines the way of generating moves.

In the program I use a lot of precomputed data to increase move generation speed. This precomputed data is made of arrays of bitboards. One bitboard represents all possible target squares for a given piece standing on a given square. There are 4 arrays of type *bitboard[64]*: for king, knight, white and black pawns. Since targets for sliding pieces (bishops, rooks and queens) depend on positions of other pieces on the board, these other pieces must be considered while computing arrays for these pieces. The solution is to create 4 arrays (for row attacks, column attacks and two for diagonals attacks) indexed by source square and configuration of pieces on the considered row/column/diagonal. Therefore, these arrays are of type *bitboard[64][256]*.

This data are pre-computed when the program starts.

The move generation relies on bit-wise operation that are performed on bitboards together with precomputed data. Below there is shown a skeleton of code that serves to generate moves for black knights. It is a representative example of bitboard-based move generation.

```
// get bitboard representing positions of black knights
pieces = board->blackknights;
// for each black knight
while (pieces) {
      // find the index of the leftmost bit that
      // corresponds to knight's position
      source_sq = findTrailingBit(pieces);
      // unmask the bit representing the found knight
      // chGetMask returns a bitboard with a single bit set
      // (at  the given position)
      pieces ^= chGetMask(source_sq);
      // determine the possible target squares (using
      // precomputed data stored in the structure
      // piecemoves) that are not captures and do not
      // contain friendly piece (the target
      // square must not be occupied)
      target = (piecemoves->knight[source_sq] & board ->
            not_occupied);
      // for each possible target square
      while (target) {
            // find the rightmost bit corresponding to the
```

```
                // considered target square
                dest_sq = findLeadingBit(target);
                // unmask the bit of the target square
                target ^= chGetMask(dest_sq);
                // pack the move to internal representation
                // containing source and destination square
                // as well as pieces: captured and promoted into.
                move = packMove(source_sq, dest_sq, NOPIECE,
                     NOPIECE);
                // store the move
                moves->add(move);
            }
        }
```

The detailed presentation of how this algorithm works is shown below on a real example, considering the following position:

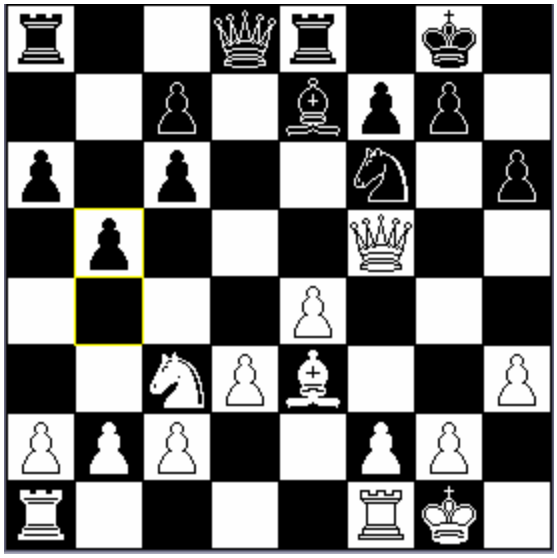


Fig. 23. A sample position, white to move

In order to determine all possible non-capturing moves for the black knight the following bitboards are used:

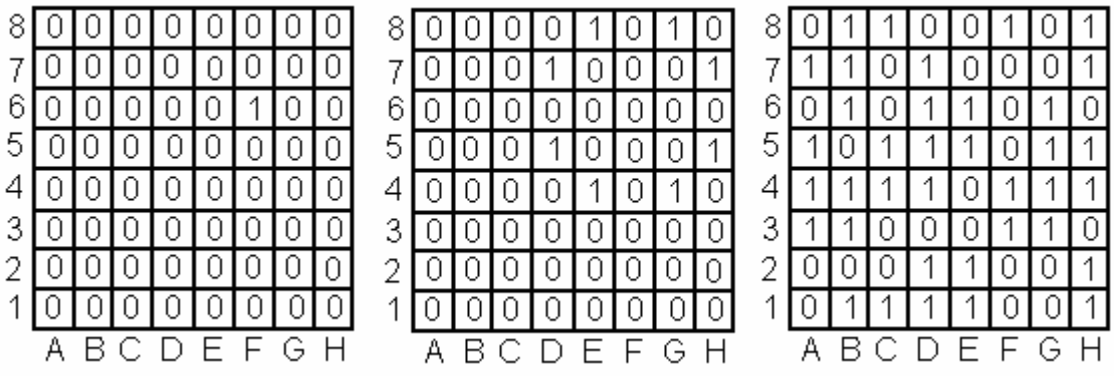


Fig. 24. Bitboards corresponding to the above position: position of black knight (left), squares attacked by a knight from square F6 (middle) and not occupied squares (right)

In this example the source square of the black knight is found to be F6 (fig. 17, left picture). Using this index the bitboard representing knight's moves is retrieved from the precomputed arrays (knights[F6] - picture in the middle). The bitboard "not-occupied" is part of the board representation – it contains all empty squares on the chessboard. The target squares are extracted from the bitboard made of the knight[F6] and "not-occupied" bitboards AND-ed together (as this bitboard will represent squares that are both accessible for the knight from square F6 and empty, as this example is for non-captures only).

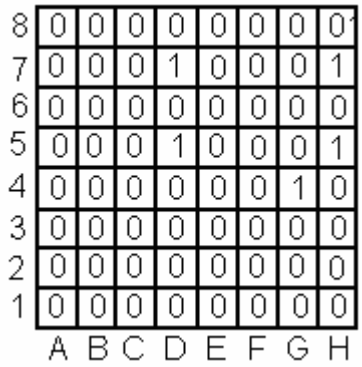


Fig. 25. Bitboard representing squares that are both not occupied and accessible by the knight from square F6

Having the source square (F6) and bitboard of target squares the move list is easily populated by looping over the target bitboard in order to determine the coordinate of each target square and unmasking it, until the bitboard becomes all zeroes: F6-H7, F6-D7, F6-H5, F6-D5, F6-G4.

After finding the source square for each knight its corresponding bit is cleared in the black-knights bitboard, so the bitboard becomes zero – no more black knights are available, and the algorithm ends. If there were more black knights on the board, the above procedure would be repeated for each of them.

Another example of pseudocode represents generation of capture moves for sliding pieces.

```
      pieces = board->whiterooks;
      while (pieces) {
            source_sq = findLeadingBit(pieces);
            pieces ^= chGetMask(source_sq);
            // compute the 8-bit index representing occupancy of
            the // row where the rook stands;
            index = (board->occupied >> shiftbitsROWS[source_sq]) &
                  (0xff);
            // determine the possible target squares
            target = (piecemoves->rowattacks[source_sq][index]) &
                  (board->blackpieces);
            // compute the index representing column occupancy
            // (rotated90R is a bitboard representing all occupied
            //  squares, but rotated right by 90 degrees (so A1
            becomes A8, H8 corresponds to A8 etc.)
            index = (board->rotated90R >>
                  shiftbitsCOLS[source_sq]) & (0xff);
            // determine the possible target squares on the column
            target |= (piecemoves->colattacks[source_sq][index]) &
                  (board->blackpieces);
             while (target) {
                   dest_sq = findLeadingBit(target);
                   captured = -board->board[dest_sq];
                    target ^= chGetMask(dest_sq);
                    move = packMove(source_sq, dest_sq, captured,
                        NONE);
                    moves->add(move);
              }
         }
    }
}
```

Move generation with bitboards relies on bit-wise operations and precomputed data. It gains significant boost (also other parts of code that perform operations on bitboards) on 64-bit architectures, as the instructions need not operate on 32-bit chunks.

The above skeleton of code does not include one important issue: assigning scores to generated moves. In Nesik, each move is assigned an estimated score, that is used for later move ordering. This estimated score consists of a piece-square component and (in case of captures) Static Exchange Evaluation score. SEE has been described in the theoretical part, so now I will briefly tell about "piece-square" technique. It uses tables (in Nesik they are fixed, written by programmer), one for each piece type, separately for white and black pieces. These tables are 64-element arrays that correspond to the chessboard. Each element of the array contains a number that represents a relative value of a given piece on the appropriate square. For example, in most positions a knight plays a greater role when standing close to the centre of the board, and has smaller meaning when placed near the band. Therefore, the array *white_knight[D4]* holds a greater value (let's say, wN_{c4} = 20) than *white_knight[B3]*
(wN_{b3} = -5). Therefore, the move Nd4-b3 would be assigned value ($wN_{b3} - wN_{d4}$) = -25, that is negative – what reflects the fact that the knight moves from a more valuable square to a less valuable one. It may happen that this move is the best possible move in the given position, but it does not change the fact that in most cases it will be suboptimal. As a result, move ordering is improved (provided the tables are filled with appropriate values, and are consistent with each other).
The move generator in Nesik also rewards (by increasing the score assigned to the move) movements of a hanging piece (that is usually good) and penalizes moves onto unprotected squares (that should be avoided).

Precomputing of data should be avoided if the data can be cheaply determined on-the-fly. For example, the function *chGetMask(int index),* that returns a bitboard with only one bit set (on the index-th position), might be implemented as *return Mask[index]* (where Mask[] is an array of precomputed bitboards) or *return (bitboard)1 << index*. Usage of precomputed data in this trivial example makes no sense, as the performance gain form using it would be nothing (or even negative), and the array data would only pollute processor's cache.
The precomputed arrays of possible target squares use a significant amount of memory (almost 600kB), and they should be cached to avoid costly memory accesses. Since most popular hardware architectures nowadays have less than 1MB of cache, the large size of these arrays can be a problem. In fact, I have observed that Nesik plays much slower (in terms of NPS) on systems that have only a little of L2 cache – probably the reason is that these arrays do not fit into the cache, leading to many very costly accesses to operating

memory. One possible way to reduce Nesik's appetite for cache might be removal of redundancies that are present in these tables, therefore reducing their sizes. An example of such redundancy is the fact, that in (the most memory-consuming) arrays for sliding pieces the position of moving piece is included twice: explicitly as the source square (the index from the range 0..64) and implicitly in the row/column/diagonal occupancy (the index from the range 0..256). By removing the piece from the occupancy component, the size of these arrays can be reduced by a factor of two. Another source of redundancy, that might be exploited at the expense of additional computations, is symmetry. However, as these low-level details associated with cache usage were not of my interest while writing this program, I left these massive arrays in their original, simplest form.

Nesik's move generator can generate separately captures (it also includes promotions, even if they are not capturing) and non-captures. Moreover, only good (in the sense of Static Exchange Evaluation) captures can be returned or only bad or all at once. As it has been mentioned in the theoretical part, such approach allows for incremental move generation (performed in several batches, in order to avoid generation of unnecessary moves when one of the first ones might cause a cut-off).

When a checking position is encountered, the generator returns only check evasions. In that case, only moves that are likely to escape check are returned. That function uses additional chess knowledge, that allows it to generate the evasions. Most of the moves returned by this generator are legal (escape the check), although it is not certain (for example, it might generate a move that captures the checker, but the capturer may be pinned on king). Since in a checking situation most moves are not legal, the search module would have to refuse most moves if the normal generator was used. Therefore, generation of mostly legal evasions in the specially designed generator should save some time. In practice, I have observed that, due to greater complexity of the check-evasions generator, the savings are minimal, if any.

3.3.2 Search

Nesik employs an alpha-beta searching algorithm with aspiration window (of size about half a pawn). It does not use Principal Variation Search nor MTD(f). Moreover, iterative deepening is use to improve search performance and for time management. The first three iterations are performed at full window width, then in each next iteration the window is determined according to the value of the previous iteration (basically: [previous_iteration_score – 35, previous_iteration_score + 40], where one pawn has a

value of 100). Due to small fluctuations of the mini-max value between odd and even iterations the aspiration window bounds are adjusted by a small value at even iterations in order to reduce the number of required re-searches.

In case of search failing low or high at the root, the window is not relaxed to minus or plus infinity at the one side. Instead, the re-search is performed using a less relaxed window (of size about 2 pawns, i.e. 200 points). The larger window should be enough for most re-searches, still offering more cut-offs than the full-width search. The full-width re-search is performed only when the search with the relaxed window fails again. It is also used when search instability occurs. Search instability is a rare phenomenon of failing high right after failing low (or vice versa). It is not possible with pure alpha-beta algorithm, but occurs in practical applications due to some search enhancements (transposition table, forward-pruning etc.) that lead to a variable rather than fixed depth search, and confuse alpha-beta algorithm.

The alpha-beta implementation in Nesik follows the negamax convention. It exploits the fact that for every node in the tree the following holds:

alphabeta(alpha, beta, depth) = - alphabeta(-beta, -alpha, depth+1)

Using this notation, there is no need for separate consideration of MAX and MIN nodes – code becomes more compact:

```
chSCORE alphabeta(chPOSITION node, chSCORE alpha, beta) {
     if (isLeaf(node) ) return evaluate(node);
     g = -maxSCORE;
     x = left_most_child(node);
     while (isNotNull (x) ) {
          g = -alphabeta(x, -beta, -alpha);
          if (g > alpha) {
               alpha = g;
               if (g >= beta) {
                    return g;
               }
          }
          x = next_brother(x);
     }
     return g;
} // end of alphabeta
```

Algorithm 3. Alpha-beta, negamax fashion

The above algorithm performs exactly the same as the version described in the previous part (algorithm 2), but its advantage is much simpler and more compact notation.

The implementations of alpha-beta algorithm described here and in part 2 are fail-soft versions. Fail-soft alpha-beta means, that the returned value can lie outside the alpha-beta window. Therefore, if the search a ply deeper returned because none of the moves could make it to alpha, or one of the moves reached beta then not only this fact is returned, but also the information about how far outside interval the result was. This information can be used for example when using aspiration window at the root to adjust the window in case of re-search more precisely (i.e. if the original search was performed with the window centred around +50 and it failed low, with the value returned -75 then this value can be used as centre of the new window during re-search). Nesik uses a fail-hard version of alpha-beta, that cannot return values outside the [alpha, beta] interval – lines “return g” are replaced with “return alpha” or “return beta” (in case of fail-low and fail-high, correspondingly).

When the desired depth of the full-width search is achieved, quiescence search is used in order to achieve reliable score for the position (on algorithm 3, the call to `evaluate()` is replaced with a call to `quiescence()`). Quiescence search in Nesik is a simple implementation that does not consider checks and does not use transposition table.
The basic implementation of quiescence search looks as follows:

```
chSCORE quiescence(chPOSITION node, chSCORE alpha, beta) {
     score = evaluate(node);
     if (score > alpha) {
          alpha = score;
          if (score >= beta) {
               return beta;
          }
     }
     g = alpha;
     x = left_most_child_captures_only(node);
     while (isNotNull (x) ) {
          g = -quiescence(x, -beta, -alpha);
          if (g > alpha) {
               alpha = g;
               if (g >= beta) {
                    return beta;
```

```
                }
            }
            x = next_brother_captures_only(x);
        }
        return alpha;
    } // end of quiescence
```

Algorithm 4. Quiescence search, negamax fashion (fail-hard)

Quiescence search in Nesik is improved by the fact that the only moves that are considered by quiescence search are “good” captures. Again, “good” captures are moves that do not give away material according to Static Exchange Evaluation.

A little trick commonly named “delta” pruning is used in the quiescence search. The idea behind that technique is simple: there is no point in considering captures that cannot reach the score close to the material balance for the worst case found by now. For example, if we are already behind by a rook then there is little point in exploring moves that can give us only a pawn back, as they are likely to be pruned anyway. In terms of alpha-beta algorithm, it makes little sense to explore captures that cannot possibly reach value (alpha-MARGIN), where MARGIN is some safeguard value.

Number of nodes searched in the quiescence search is very large, in case of Nesik it is the majority of total nodes searched. It holds although number of moves searched in each quiescence search is not large, as it considers only good captures – the reason is that QS is launched at all leaf nodes of the fixed-depth part, so the number of individual searches is huge. I have not included checks in quiescence search because it might cause a serious tree explosion. However, considering checks in QS is known to increase tactical abilities of the engine. I think that making Nesik consider checks in at least some part of QS (for example, first two plies only) might give profits. The profits might be even greater if it was combined with some clever pruning tricks in QS, as pruning even a small part of the QS branches leads to noticeable reduction of total nodes searched (due to significance of QS effort in total search effort).

3.3.3 Move ordering

I have put significant effort into achieving good move ordering, as it is crucial for the search performance. Currently, the moves are considered in the following order:

- hash table (transposition table) move,
- mate killer move,

- good and equal captures (judged by SEE) and queen promotions,
- killer moves,
- moves with highest history table scores (4 best),
- all the rest, sorted by the score assigned by move generator.

Hash table move is a move (if exists) stored in the hash table for the considered position. Mate killer move is a move that produced a mate score somewhere in the tree. These moves can be retrieved without any calls to move generator, as they are managed by the search module.

Good and equal captures as well as queen promotions are generated if any of the above moves cause a cut-off.

Killer moves again do not need being generated, as they are at hand for the search routine. Nesik uses two killer move slots for each depth, and takes care that they are never occupied by the same move (as it would be a waste of a precious resource). The search module considers the killer moves in the following order: killer_1[depth], killer_2[depth], killer_1[depth-2], killer_2[depth-2] – killer moves from not only the current depth, but also from the last depth when the current side to move was on move are considered.

Some (or even all) of the moves that are not generated from scratch (like hash table move, all killer moves) may not be valid in the current position, therefore they must be validated before use. The other approach would be to generate all moves first, and then only shift the chosen moves toward the head of the list – it would not require the validation stage for some moves, however the penalty due to the need of complete move generation first, and then pre-sorting the moves renders that approach less efficient than the presented one, as the move validation is a fast procedure.

The algorithm used for sorting the last batch of moves (after moves) is insertion sort. It is an $O(N^2)$ algorithm – but the constant hidden in the O notation is small, and the algorithm does not introduce much overhead. Moreover, it can make use of existing order in input data ([Sedgewick, 1992], p. 99-103). Therefore, this simple algorithm works well with small data sets that it has to deal with (number of moves to be sorted is usually below 30), rendering implementation of more advanced algorithms infeasible.

One point where that move ordering scheme could certainly be significantly improved is at the root of the search. Since the list of moves at the root never changes for the given position, some more attention could be paid to improve move ordering here, with negligible time overhead included. For example, the results from previous iteration could

be used to reorder the move list according to the achieved scores before launching the new iteration.

3.3.4 Transposition table

The transposition table is implemented as a hash table, using Zorbrist keys to address the table. The process of calculation of Zorbrist key is described below – it uses a three-dimensional table (in case of Nesik – computed each time the program starts) of size 6 (number of piece types) by 2 (number of colours) by 64 (number of chessboard size), filled with pseudorandom values.

In Nesik pseudorandom numbers are generated using compiler's standard library functions. Some people (e.g. some practitioners from [CCC]) claim that many random generators from C and C++ libraries offer poor randomness. Poor randomness can in that case increase number of collisions, and reduce effectiveness of the hash table. In cases when the standard random numbers are unacceptable own generator (e.g. one of presented in [Knuth, 1981], p. 1-38) can be coded manually.

Initially the key is equal to zero. Then the chessboard is traversed and each time a piece is encountered the appropriate value from the table is XORed (exclusive-or) with the key. For example, if we encounter a black bishop on square G7 then we perform the operation *key ^= Zorbrist[BISHOP][BLACK][G7]*. When the board is traversed, the resulting key is XORed with some another pseudorandom value if the side to move is black. Similarly a value corresponding to the en-passant square (if any) and castle rights are included in the key.

The great advantage of Zorbrist keys is their inherent capability of being incrementally updated. For example, to reflect the change of key due to the move Qd2xe3 (white queen captures black knight) it is necessary only to perform three operations:

1. *key ^= Zorbrist[QUEEN][WHITE][D2]* (to remove the white queen from the square D2),
2. *key ^= Zorbrist[QUEEN][WHITE][E3]* (to include the new location of the queen in the key),
3. *key ^= Zorbrist[KNIGHT][BLACK][E3]* (to remove the location of the captured knight from the key).

The simplicity comes from the properties of the XOR operation, as ((a^b)^b) == a.

Each entry of the hash table in Nesik holds the following data:

1. 64-bit hash key,

2. 32-bit hash move,
3. 18-bit score
4. 3-bit flag
5. 8-bit depth,
6. 3-bit sequence number,

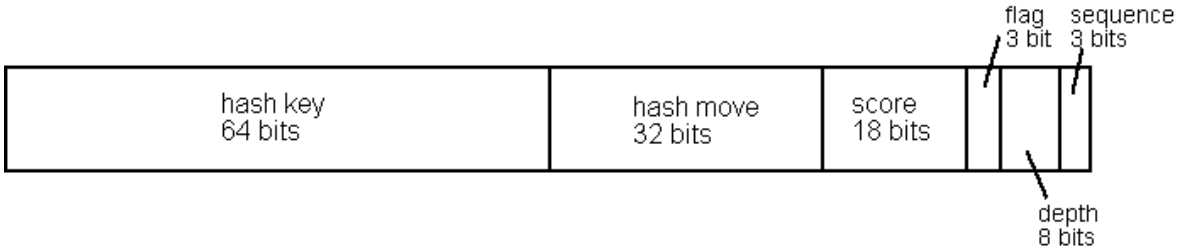


Fig. 26. Structure of a hash table entry

In order to save space and maximize the number of hash table entries in a given memory limit, the fields 3-6 are packed into a single 32-bit variable.

As the index of hash table entry is determined from the key by taking its N least significant bits (assuming that size of the hash table is 2^N entries), many different keys can point to the same entry. Therefore, the full hash key is used to verify that the stored data really represent the current position. This is the reason for storing the whole hash hey (64 bits long in case of Nesik) in the hash table entry. Of course, as the number of all possible chess positions is much larger than 2^{64}, different positions can give the same hash key, but the probability of such phenomenon is enough rare to be neglected.

The hash move field serves to store the best move encountered while searching the position. It may not be filled, as all moves might fail low and therefore no best move would be available, but if it is then that hash move is considered first when the position is explored again.

The score is the value of the position determined by search. As, due to the nature of alpha-beta operation, the exact score may not be given (because the score might only proved to be good enough or bad enough to cause a cut-off, without the need for determining the exact score), it is important to store also a flag that indicates the meaning of the data stored in the score field. This flag tells whether the score is an exact score, an upper bound (meaning that the actual score of the position is at most *score*) or a lower bound (actual score is at least *score*).

The field *depth* states how deep the search was performed from the stored position. If that value is greater or equal to the remaining depth at the moment when the entry is retrieved, then a cut-off may be done if the *flag* and value of the *score* field allow it. If it is

not then no cut-off may be performed (for example, if the data stored come from a 5-ply search then it cannot be considered enough accurate to cause a cut-off in a 8-ply search), but the hash move can still be taken as the first move to try.

Finally the *sequence* field is used to improve the replace policy for the depth-based table.

The general idea of integrating transposition table into alpha-beta framework in Nesik looks as follows:

```
chSCORE alphabetaWithTT(chPOSITION node,chSCORE alpha,beta) {
     if (isLeaf(node) ) return evaluate(node);
     if ( (entry = getFromHash(node) ) != NULL) {
          if (TT_entry_deep_enough) {
          // data in hash table comes from a search to the
          // same depth as current or deeper – so it is
          // reliable
               if (entry.flag == UPPER) {
                    if (entry.score <= alpha) {
                         return alpha
                    }
                    if (entry.score < beta)
                         beta = flag.score;
                    }
               }
               if (entry.flag == LOWER) {
                    if (entry.score >= beta) {
                         return beta;
                    }
                    if (entry.score > alpha) {
                         alpha = flag.score;
                    }
               }
               if (entry.flag == EXACT) {
                    return entry.score
               }
          }
          else {
          // TT entry represents results of a shallower
```

```
            // depth search – no cutoffs possible, but still
            // a valuable move to try first can be used
                  if (entry.move != NULL) {
                        try_hash_move_first = true;
                  }
            }
      }
      g = alpha;
      x = left_most_child(node);
      hash_flag = UPPER;
      best_move = NULL;
      while (isNotNull (x) ) {
            g = -alphabeta(x, -beta, -alpha);
            if (g > alpha) {
                  alpha = g;
                  if (g >= beta) {
                        hash_flag = LOWER;
                        best_move = current_move;
                        saveHash(node,g,hash_flag,best_move);
                        return beta;
                  }
                  hash_flag = EXACT;
                  best_move = current_move;
            }
            x = next_brother(x);
      }
      putToHash(node, g, hash_flag, best_move)
      return alpha;
} // end of alphabetaWithTT
```

Algorithm 5. Alpha-beta (fail-hard) with transposition table, negamax fashion

Nesik actually uses two hash tables for its transposition table implementation. The difference between those two is in the entry replacement policy. The first table uses the simple *always-replace* rule (that assumes the results obtained most recently are more likely to be useful) and the second one employs a more sophisticated *depth-based* policy. The

idea of depth-based replacement is based on the observation that the entries with low remaining-depth field do not provide big savings (they can rarely cause a cut-off, and the hash move they offer is less likely to prove best) while those representing the results of a deep search can easily cause a cut-off, or reliably supply the best move to achieve good move ordering. Therefore, the table basing on that policy will not replace an entry with a new one coming from a shallower search. The problem with the depth-based strategy is the danger of filling the table with a large number of deeply searched entries (that cannot be easily overwritten) that are not useful anymore, as they come from an ancient search. The solution that I use employs the *sequence* field, that is assigned a value when the entry is written. The assigned value is identical for all iterations within a single search, but differs otherwise.

Later, when the new entry is to erase the existing one, and its remaining depth value is too small, the sequence numbers are compared – if the existing entry has the sequence field different than the new one then it means that it comes from an old search, and is replaced.

When an entry is to be retrieved from the table, the depth-based table is probed first, and if the entry does not exist, the always-replace table is tried. In order to put an entry to the table the depth-based table is looked up to check if it contains a deeper result. If it does, then the new entry is put directly in the always-replace table, otherwise the entry from the depth-based table is copied to the always-replace table and the new entry is put to the depth-based table.

3.3.5 Time management

Nesik can handle different time control schemes:

1. Base time limit per game plus time increment after each move (increment can be 0),
2. Time limit per given number of moves,
3. One move per time limit.

Moreover, it can have its clock adjusted manually during the game, by the Winboard command “time”.

The way time management is performed by Nesik is given below.

At the beginning of search the time manager computes the amount of time that is assigned to that move (usually about $1/30^{th}$ of the remaining time). Then, every N nodes the search routing calls the time manager’s method *tick()*. That method checks the amount of time that has passed and compares it with the precomputed limit. If the limit has been

reached, the timeout flag is set. When the search routine notices the flag, it finishes search immediately and returns. The results of unfinished iteration are not used, the best move found in the previous iteration is made.

The universal policy of using $1/30^{th}$ of the remaining time (plus the time increment, if any) is a simple and robust scheme. However, several improvements could be made. The engine might be assigned some extra time when it leaves the opening book and starts playing on its own (since often at the moment of leaving the book it is important to choose the correct plan, more time to allow greater insight into the position seems good), when the current iteration has returned a fail-low to the root (meaning that our position may be worse than expected, and more time may be necessary in order to find a better move), or the search time might be reduced if one move seems much better than any other. Moreover, in a position where there is only one legal move it should be played immediately.

3.3.6 Pondering

Nesik is able to utilize the time when opponent is thinking. Instead of going idle after making a move, it assumes that the opponent will make the second move of the principal variation (the one that is considered best for him) and starts analyzing the resulting position as if it was its move. If no such move is available it probes the hash table for the current position in order to determine the best move (that probably will be made by opponent), and if that try is also unsuccessful it performs a quick 5-ply search from the current position and supposes the best move will be made.

If the opponent makes the predicted move, then Nesik continues its search, having gained the time opponent spent, and if he does not then the search is interrupted, the expected move – unmade, the actual move is performed and search restarted. Therefore, pondering should not hurt the performance in any way, and can provide the program with a lot of extra time for analysis.

3.3.7 Opening book

When Nesik finds the opening book available, it reads another set of random data from a file. This data is used for computing the hash key for positions, independent on the one that is used to address the transposition table. In the initial stage of the game, each time Nesik is on move it performs a book lookup. The mechanism is as follows: it computes the position's hash key using the random data from the file, and looks up the book for the entry

with identical hash signature. The book entry contains the hash signature and the list of moves with their corresponding occurrence frequency. For example, an entry might look as follows:

1734199420002361

g8f6 4 b8d7 5 c8f5 4

That means: position's hash signature 1734199420002361, book moves: g8f6 (4 occurrences), b8d7 (5 occurrences), c8f5 (4 occurrences).

If Nesik gets a book hit, it chooses one of the moves, with probability depending on the number of that move's occurrences. So, using the example above, moves g8f6 and c8f5 would be chosen about 4/13 of the time each, and b8d7 – 5/13 of the time.

Each time the book lookup fails, a counter *book_misses* is incremented (at the beginning of the game it is set to zero). When it reaches certain threshold (like 5 misses) then book probing is switched off, and Nesik does not use book in that game anymore.

Since the number of positions in the book can be large (the greatest book that I created contained over 200,000 positions) and the book file can easily exceed 10MB of size, the probing method becomes problematic. I decided to use a format in which the hash signatures are sorted, and additional index file is created. This approach provides satisfactory performance – lookup times are negligible, measured in small fractions of a second.

The book in the current version of Nesik is created automatically, from a set of games stored in a Portable Game Format (PGN) file, using simple tools that I created for that purpose. All headers and moves behind certain threshold are removed – all that remains is the raw data consisting of several initial moves from each game. Actually that might serve as a simple opening book, but (in order to allow Nesik to deal with transpositions in the openings) I decided to store the positions (hash keys) that appear in those openings and possible moves rather than move sequences. The hash keys are computed using the same random data that Nesik uses for probing, otherwise Nesik would not be able to use the book.

Quality of the book depends strongly on the content of the PGN file. To create a reasonable book I used several large files containing games of grandmasters and masters that are publicly available on the Internet. One might assume that very high level of players guarantees great opening material, but I was afraid that some of the games might contain serious blunders. It was impossible for me to verify it manually, due to a huge number of games and positions. Therefore I introduced to Nesik a threshold parameter – if number of

move occurrences in the given position is smaller than that threshold then that move is ignored. By increasing the threshold (for example to 5) one can make sure that only good moves are considered, and all blunders are eliminated (as it is not likely that a serious blunder appears in games of top masters and grandmasters more than 4 times) – but also lots of good moves are thrown away then. As the source games are supposed to be of high quality, a threshold value of 2 seems safe.

The opening book scheme that I used in Nesik is simple and works satisfactory. In the later part of the work I will describe changes that had to be made in order to allow book learning. Book learning should be useful for increasing program's strength, as automatically generated books without learning are likely to contain many holes, that can be exploited and converted to an easy win by opponents. The book itself as well could be improved in many aspects, for example forced variations should be covered more thoroughly, and in longer lines than peaceful ones (currently the largest book consists of a fixed length lines of 12 plies), as a repertoire hole in a forced variation can lead to a straightforward lose, without even having a chance to fight. Also engine's style of play should be considered when developing content of the book. Actually, I have seen many games in which Nesik lost easily due to its poor opening preparation, as development of a solid opening book was not the main concern in the initial stage of program's life - it was added only to make Nesik competitive to other engines.

3.3.8 Evaluation function

The heuristic evaluation function that guides the search plays a primary role in achieving good performance. Although it is possible to build a program that plays very good chess (much better than Nesik) that has only a simple evaluation function of several terms but a very good search capabilities, it is well-known that a finely tuned evaluation function is necessary to achieve top performance. Many positional traps and mistakes cannot be determined by means of search only, due to its limited depth. Evaluation function should be able to find and include as many of them as feasible in the score.

Evaluation function used by Nesik is moderately complex – it has a reasonable set of knowledge but still misses a lot of subtle aspects of positions. Its value can is calculated as a difference of sums of advantages and disadvantages for each player.

Below are listed major components of the function, divided into groups:

1. **Objective measures:**
    a. material balance,

b. piece-square value,
c. mobility,
d. king safety (pawn shelter, (semi)open files, closeness of enemy pieces, pressure on squares close to king),
e. pawn structure.

2. **Rewards for:**
   a. pawns in the centre,
   b. having at least one pawn in endgame,
   c. connected rooks on $1^{st}/8^{th}$ line,
   d. white (black) rook on 7^{th} (2^{nd}) rank,
   e. more than 1 heavy piece on the $7^{th}/2^{nd}$ rank,
   f. rook behind a passed pawn,
   g. rook on open file,
   h. rook on semiopen file,
   i. pawn storm on enemy king (in case of different castles),
   j. bishop pair,
   k. outposts.
   l. possession of queen in case of better king safety,
   m. presence of fianchetto bishop in case of a pawn hole on B2/G2 (B7/G7) in front of own king.
3. **Penalties for:**
   a. lost right to castle,
   b. trapped bishop,
   c. rook trapped horizontally,
   d. bad bishop,
   e. bishop blocking own central pawns (D2/E2, D7/E7),
   f. hanging pieces,
   g. enemy queen too close to own king if king's safety seems poor.

This amounts to about 25 terms, some of them are further subdivided into smaller components (for example king safety). Value of the evaluation function is determined according to the following rule:

SCORE =

(1.0+75.0/(TOTAL_MATERIAL_ON_BOARD)) * MATERIAL_BALANCE +

PIECE_SQUARE_BALANCE +

PAWN_STRUCTURE_EVALUATION_BALANCE +

KING_SAFETY_EVALUATION_BALANCE +

PENALTY_FOR_PIECES_ON_THE_FIRST/EIGHTH_RANK +

PENALTY_FOR_TRAPPED_BISHOP +

PENALTY_FOR_NO_HORIZONTAL_ROOK_MOBILITY +

BONUS_FOR_CONNECTED_ROOKS +

BONUS_FOR_ROOK_ON_$7^{TH}/2^{ND}$_RANK +

BONUS_FOR_STRONG_ROOK_ON_$7^{TH}/2^{ND}$_RANK +

BONUS_FOR_ROOK_BEHIND_PASSED_PAWN +

BONUS_FOR_ROOK_ON_SEMIOPEN_LINE +

BONUS_FOR_ROOK_ON_OPEN_FILE +

BONUS_FOR_BISHOPS_PAIR +

PENALTY_FOR_BAD_BISHOP +

PENALTY_FOR_BISHOP_BLOCKING_$3^{RD}/7^{TH}$_LINE +

BONUS_FOR_BISHOP_IN_ENDGAME +

BONUS_FOR_MOBILITY +

PENALTY_FOR_HANGING_PIECE +

PENALTY_FOR_HANGING_PIECE_ATTACKED_BY_OPPONENT +

BONUS_FOR_OUTPOST +

OTHER_COMPONENTS.

The above formula is not complete, it only gives an idea how the overall value of the position is calculated by Nesik. Terms ending with word “balance” are either computed incrementally (like material balance, piece-square values) or returned by a supplementary function (pawn evaluation, king safety). Positive “balance” indicates white’s advantage in that term, negative value means that black is better in that aspect. The same applies to SCORE as the final value: positive SCORE means white’s advantage.

Components with names starting with “bonus” have positive meaning (make position stronger) while those starting with “penalty” decrease position’s evaluation. Both are computed in a similar manner, according to the scheme:

BONUS_TERM_1 = WEIGHT_1 * (WHITE_TERM_1 – BLACK_TERM_1);

PENALTY_TERM_2 = - WEIGHT_2 * (WHITE_TERM_2 – BLACK_TERM_2);

For example, below are the steps needed to find value of BONUS_FOR_ROOK_ON_OPEN_FILE:

1. count all white rooks on open lines (assign that value to WHITE_ROOK_ON_OPEN_FILE),

2. count all black rooks on open lines (assign that value to BLACK_ROOK_ON_OPEN_FILE),

3. subtract number of black rooks on open files from the number of white rooks on open files (WHITE_ROOK_ON_OPEN_FILE – BLACK_ROOK_ON_OPEN_FILE) – result tells by how many rooks on open files white leads over black (if the result is negative then black has more rooks on open files than white).

4. multiply the value from point 3 by WEIGHT_ROOK_ON_OPEN_FILE, that tells how important that term is relative to others, to get the final result.

The value from point 4 is added to the running total SCORE. If it was not a “bonus” but a “penalty”, then that value would be added to SCORE with negative sign.
As it was written above, positive value of SCORE means that white’s position is evaluated as better than black’s – the formula given above gives evaluation from the side of white. If evaluation from black’s side is needed then the SCORE must be negated.

One more interesting thing about the evaluation function used in Nesik is that the material balance is weighted by remaining material on the board. The less material is present in the position, the more important material balance remains. It reflects the general rule in chess that material imbalance tends to pay greater role as the total material gets smaller – author has not found this trick used in source code of any publicly available chess program, although it seems sensible to add such weighing.

The most complex part is pawn structure evaluation, that penalizes doubled, isolated, backward pawns, and rewards neighbouring, passed and connected passed pawns. Pawn evaluation is one of the most time consuming components of evaluation, but its results can be stored for further reuse. It is done by means of a pawn hash table, that works similarly to the main transposition table. The hash key is calculated using the Zorbrist method, but considers pawns only. The hash table stores values of the pawn structure evaluation. As pawn structure does not change often, table probes achieve in games hit rate of 97-99%. It means that the time actually spent by program in the pawn evaluation routine is negligible, so it can be made arbitrarily sophisticated without negatively influencing engine’s speed. In Nesik, together with the pawn structure I store evaluation of king’s pawn shelter – it is quite time-consuming and does not change often, so it is a good candidate for hashing.

Due to bitboard-based representation of chess position some evaluation terms are very easy to calculate using cheap logical operators and some masks. An example of computation of one evaluation term is shown below. It shows how to determine whether a certain pawn is passed (no opponent's pawn can stop it from promoting) – passed pawns are one of terms that create pawn structure evaluation.

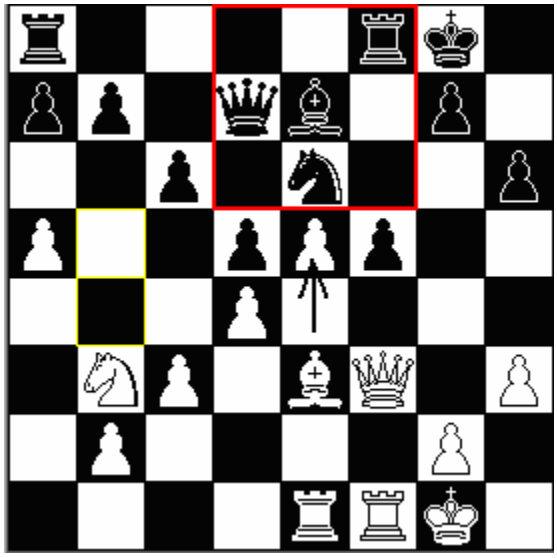


Fig. 27. Position with passed white pawn on E5

In order to verify whether white pawn on E5 is passed or not only one logical operation needs to be performed: PASSED_WHITE_PAWN_MASK[E5] & black_pawns.

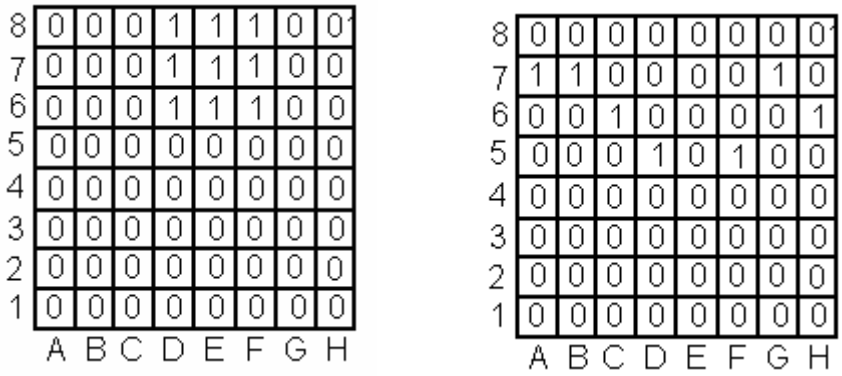


Fig. 28. Bitboards representing PASSED_WHITE_PAWN_MASK[E5] (left picture) and black pawns for the example position (right picture)

The PASSED_WHITE_PAWNS_MASK[E5] is a bitboard with all those bits set that correspond to the squares where a black pawn is needed in order to make white pawn non-passed (on the chessboard from fig. 19 they form interior of the red-bounded rectangle). In Nesik, this mask (like most others) is pre-computed on program's start.

If this bitboard is AND-ed with a bitboard representing all black pawns and the result is zero (as it is on the example) then no black pawn can prevent white pawn from reaching the promotion square E8 – it is passed. If the result is non-zero then the pawn is not passed. On the above example the PASSED_WHITE_PAWNS_MASK[E5] and “black_pawn” bitboards have no common “ones”, so the result of AND operation will be 0 – white pawn on E5 is correctly recognized as passed.
By repeating the above procedure for all pawns each passed pawn can be found and rewarded.

Passed pawns are one of several terms that are weighted depending on their position on the board. The weight in case of these components is not a single value but rather array of values. In case of passed pawns, the array is indexed by rank of the passed pawn – the closer it is to promotion, the more valuable it becomes, so its weight increases. Therefore, WEIGHT_PASSED_PAWN[D3] is significantly lower than WEIGHT_PASSED_PAWN[D7] – because the latter is likely to be much more dangerous for the opponent, being just one square from promotion.

Nesik has some knowledge about specific types of positions and parts of the game. It is able to recognize and penalize too early queen development in the opening, give mate in endgame KBBxK basing on evaluation, reward central pawns in openings, penalize trapped bishops, recognize insufficient material for mating etc. This knowledge supplements or replaces Nesik’s general evaluation terms listed earlier.

In some positions Nesik does not perform full evaluation. If the value calculated using several basic components is greater than (beta + LAZY_MARGIN) or smaller than (alpha – LAZY_MARGIN) then no further computations is done – beta or alpha (correspondingly) is returned immediately. The margin is set to such value that can not possibly be exceeded by skipped components. The idea beyond that technique is to avoid costly full evaluations when the result is already known, increasing program’s NPS speed.

The pseudocode for lazy exit in evaluation function looks as follows:

```
chSCORE evaluate(chPOSITION node) {
      chSCORE score;
      score = node.material_balance;
      score += node.pc_sq_score;
      score += evaluatePawnPosition(node);
      score += evaluateKingSafety(node);
      if ( (score – LAZY_EVAL_MARGIN) >= beta) {
```

```
            return beta;
        }
        else
        if ( (score + LAZY_EVAL_MARGIN) <= alpha) {
            return alpha;
        }
        …
        // rest of the evaluation function
}
```

Algorithm 6. Pseudocode for lazy evaluation in Nesik

The basic components of score are material balance (the essential part of evaluation function), piece-square values balance (quite important and for free, because it is part of board representation and as such is updated incrementally), pawns' evaluation (important and very cheap to compute, as it can be retrieved from pawn hash table in 96-99% cases) and king safety evaluation (computationally expensive, but too important to skip). In the real code a few other important but computationally inexpensive components are included (trapped bishops and opening-specific penalties) in order to slightly increase accuracy of lazy evaluation.

The number of smallest individual components that are assigned specific values and weights exceeds 60 in Nesik. Moreover, part of them is not described using a single value, but rather array of values (for example a bonus for a passed pawn is not constant but depends on how far the pawn is advanced). It accounts for number of tuneable parameters of the order of hundreds. Some of them had to be tuned from their initial settings – using my own chess knowledge I altered the parameters that caused the most improper evaluations. However, limitations of my own chess skills and current lack of understanding of all connections between different evaluation terms and relationships between search and evaluation do not allow me to manually create a function that would be optimal for Nesik.

4 Experiments

The program that was built for the purpose of this work was used to perform several experiments. Some of them served to help program's development by providing important information about its performance, while others were related to research about book learning.

4.1 Effects of chosen techniques on program's performance

In order to determine the most profitable set of techniques I performed some tests that showed if Nesik benefited from a particular technique or not, and if so then to what extent. These experiments are very simplified, and were performed during development of the chess program accompanying this thesis. Their purpose was to give the author a clue about their feasibility and relative performance, that was needed for the proper choice of techniques implemented in the engine. The results are not meant to be universal, and should not be treated as a reference of any kind. They are included here because someone implementing his/her own program may find them a good starting point for own experiments.

4.1.1 Move ordering techniques

An all-important issue in chess programs using alpha-beta search algorithm (or its derivatives) is good move-ordering (best move first).

In order to determine usefulness of these techniques I performed tests that compared number of visited nodes for fixed-depth searches for three different positions using different ordering schemes. The positions can be considered typical for three stages of the game: opening, middlegame and endgame. Nesik was configured to use its default (describe in the previous paragraphs) features, and during the tests I switched on/off the following features concerning move ordering: hash table (although it was still used for cut-offs), killer heuristic and history table. SEE was always used to order the moves, as it is too deeply integrated with move generator to switch it off.

In that way, I measured the benefits of different move ordering schemes in a practical environment. Since each of these techniques (apart from SEE) has very small influence (less than 5%) on the nodes-per-second speed of the program, any reduction of tree size can be considered positive. The results in the form of graphs are presented below:

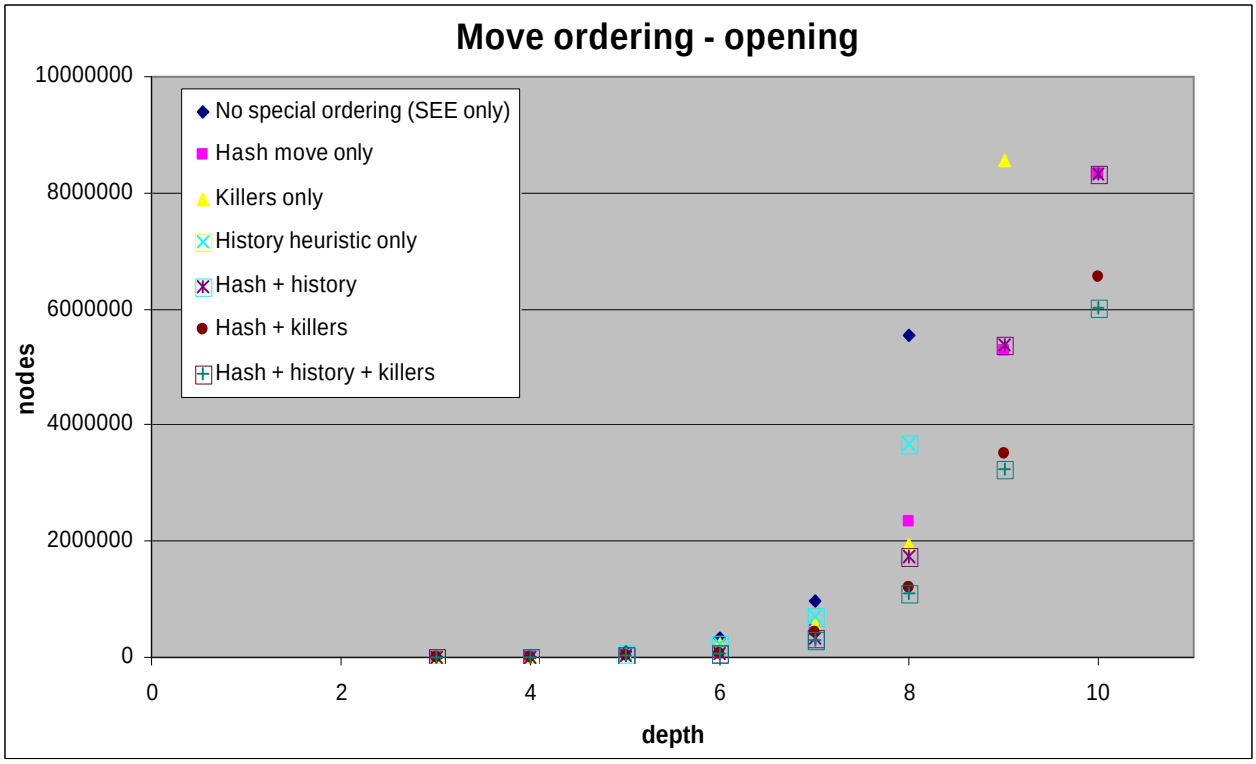


Fig. 29. Nodes-to-depth characteristic for different move ordering schemes – in a sample opening position.

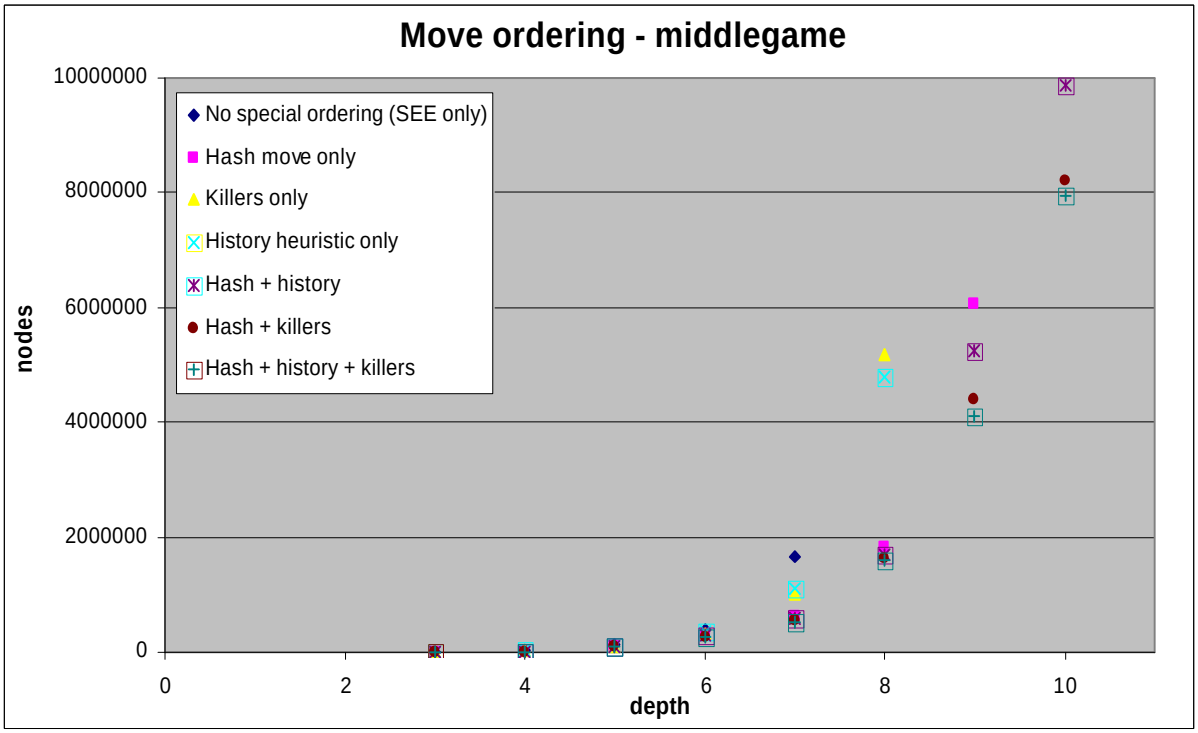


Fig. 30. Nodes-to-depth characteristic for different move ordering schemes – in a sample middlegame position.

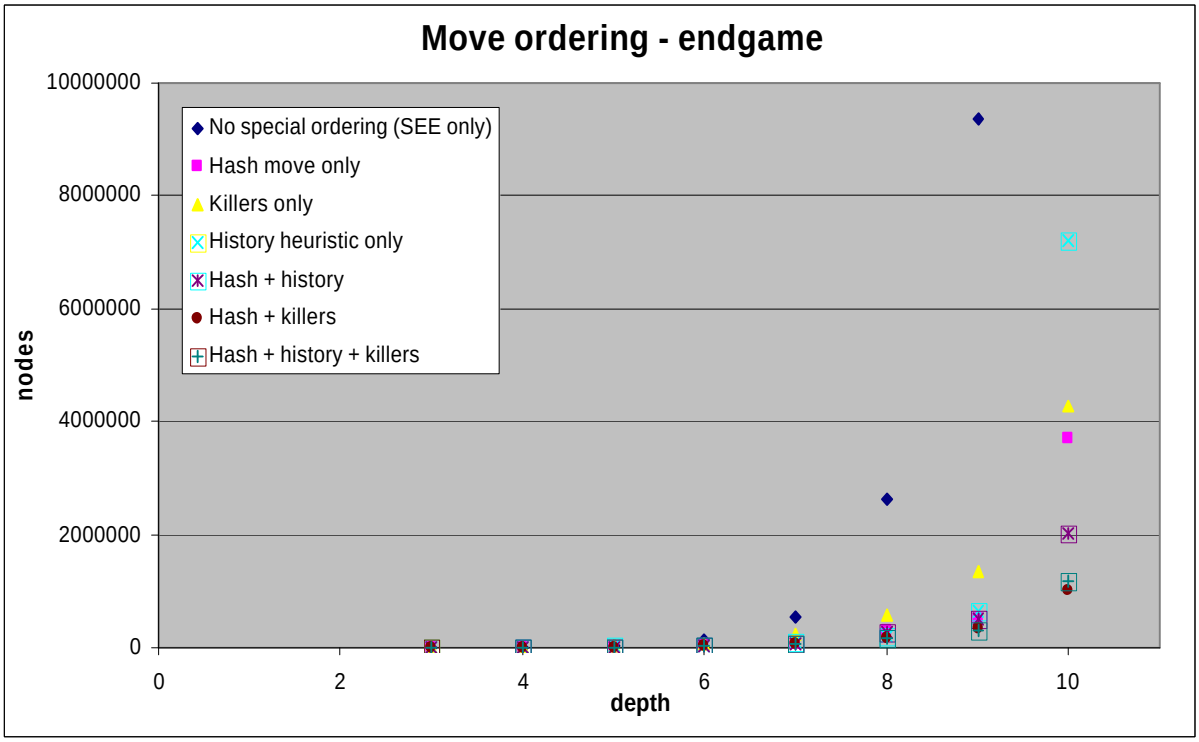


Fig. 31. Nodes-to-depth characteristic for different move ordering schemes – in a sample endgame position.

There are several conclusions from these graphs:

- History heuristic gave least (but still considerable) benefits, used alone or supplementing other techniques.
- When used alone, the technique of using the move suggested by hash table excels at move ordering.
- Implementing greater number of move ordering methods increases performance.
- Addition of history heuristic to the scheme using hash move + killer heuristic results in a little, but noticeable improvement. This is probably due to the fact, that the history heuristic keeps track of longer-term trends on the board, and its results give an overall benefit, although are subject to considerable delay in reflection of changes on the board. At the same time, killer moves (that can change more rapidly) can quickly adapt to changes of the position. Therefore, although history heuristic is considered to be a generalization of killer heuristic, they cannot be replaced one by another. At least in case of Nesik they should supplement each other in order to maximize benefits.
- The sample is too small to allow making definite statements, however it shows the general trend in position complexities: the number of nodes required to reach

certain depth is relatively high in the opening, then grows further in the middlegame, and drops significantly in endgames.

The sample size was too small to draw definite conclusions, but author's other observations made during development of the program confirmed these results, at least as far as the relative order is concerned. Therefore, Nesik uses all: hash move, killer and history heuristic to order the moves.

4.1.2 Size of transposition table

Another quick tests were performed in order to determine the smallest reasonable size of transposition table. "Smallest reasonable" means here that performance is significantly lower for smaller tables, but does not increase much (e.g. more than 5%) for greater number of entries. Nesik, because of its design, accepts only sizes of transposition table equal to the power of 2. Each TT entry consumes 16 bytes of memory.

Transposition table serves basically for two purposes: to store values and bounds of past searches (that can be used when the same position is reached again, and the previous search was deep enough) and keep the best move found in a given position (even if that search was shallower than the currently performed one, the stored move can still be useful for the purposes of move ordering, as the experiment from the previous chapter showed). Intuition suggests that the greater number of positions can be stored in the transposition table, the better program's performance is, because more useful information can be stored for reuse. However, it is hard to guess what is the threshold, beyond which further increase of TT size is not profitable.

First, the following experiment was performed: 30 similar endgame positions were analysed by Nesik to depth 10. The positions are presented in Appendix A. Depth 10 was chosen as it is typical depth reached by Nesik in middlegames at popular 5-15 min/game time controls). Size of transposition table was fixed at 524,288 entries. Number of nodes needed to finish 10^{th} iteration was recorded, and the histogram from fig. 32 was created.

Shape of the histogram does not indicate any specific distribution. Therefore, dependency of number of nodes required to reach certain depth on position, for fixed size of TT and all positions belonging to the same group, is unknown.

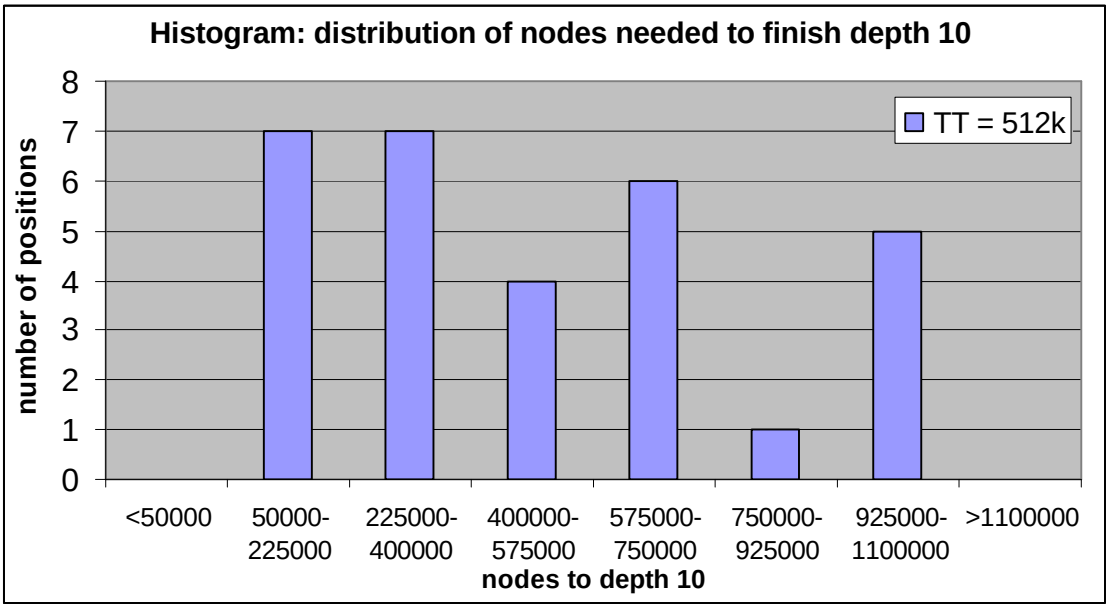


Fig. 32. Distribution of nodes needed to finish 10th iteration

Later on, in an attempt to determine influence of size of TT on number of nodes searched, the following testing procedure was used: engine was forced to think about certain positions and record number of nodes required to reach depths 9 and 10 for different sizes of transposition table. For each chess game phase (opening, middlegame, endgame) three sample positions were chosen (that can be considered typical for these phases), and the test was ran for each of them. Then the results were averaged in order to obtain final results.

Number of nodes-to-depth was chosen as the measure because, given certain nodes-per-second speed, it provides the best and easiest to track estimation of time required to reach certain depth. Size of TT does not affect NPS speed of the program by any significant amount, so the nodes-to-depth is directly proportional to time-to-depth. Given nodes-to-depth (NTD) and NPS speed of the program one can directly find time-to-depth (TTD) value: TTD = NTD / NPS.

The results are shown on the figures below:

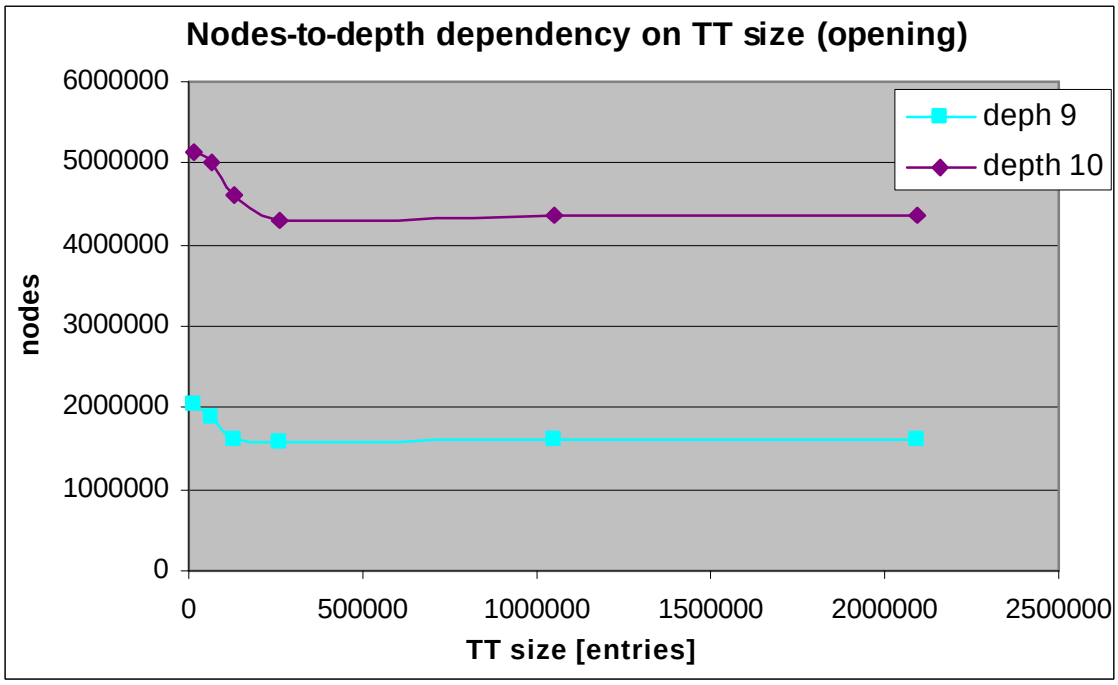


Fig. 33. Nodes-to-depth as a function of transposition table size (averaged for three opening positions)

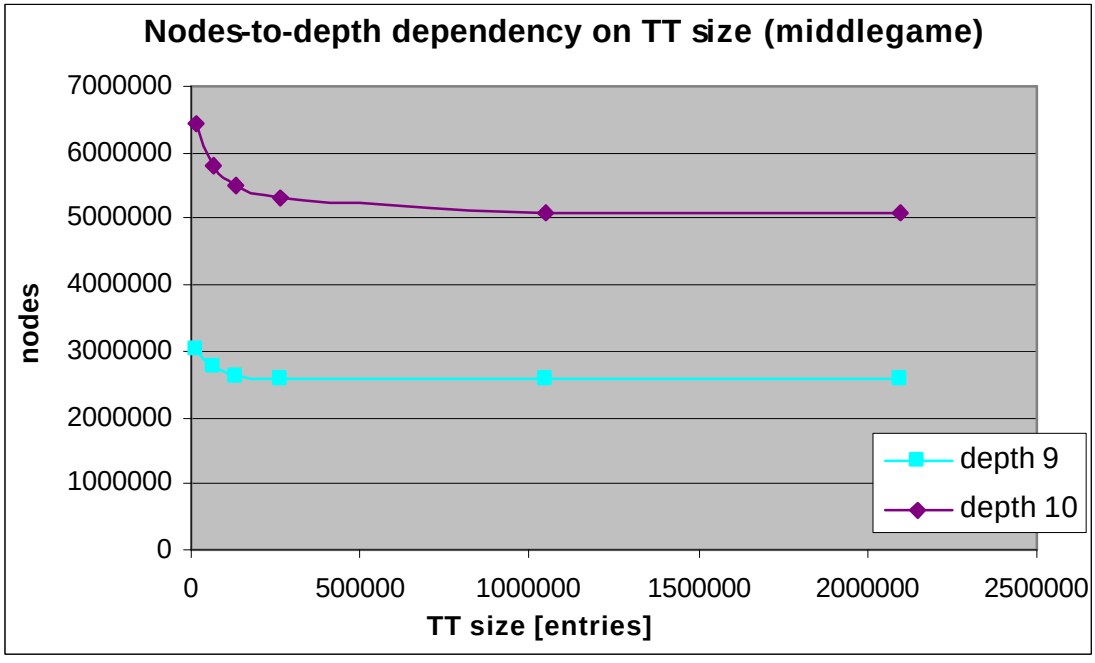


Fig. 34. Nodes-to-depth as a function of transposition table size (averaged for three middlegame positions)

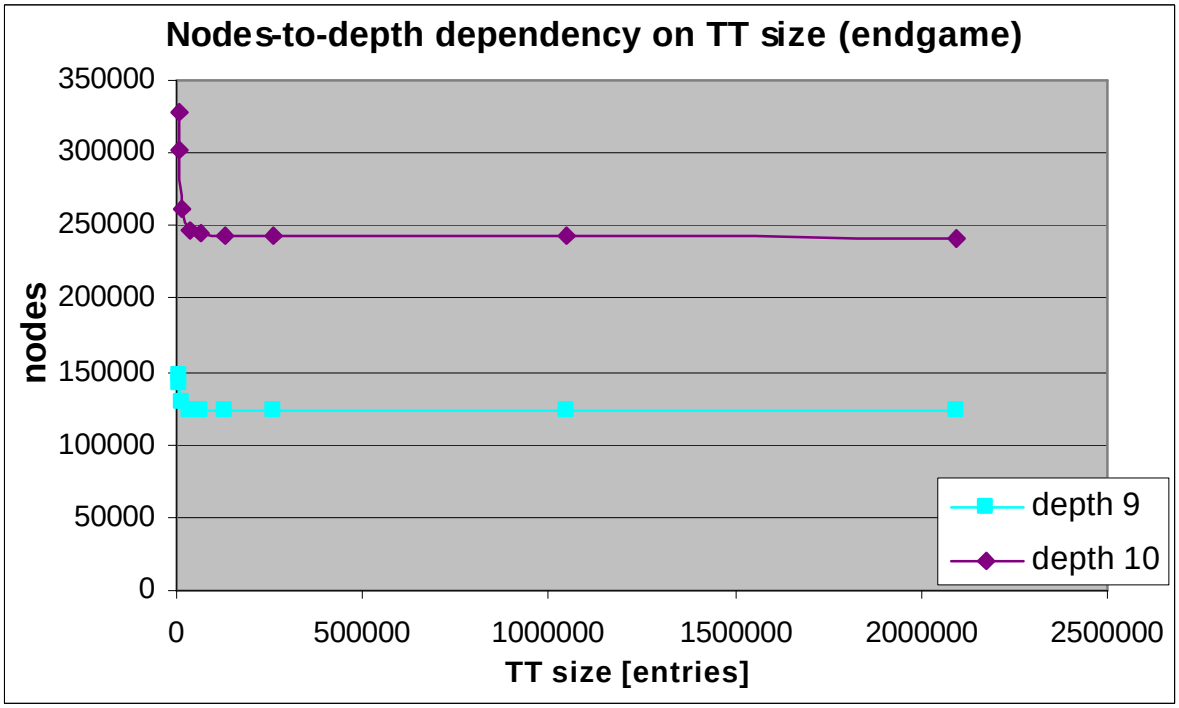


Fig. 35. Nodes-to-depth as a function of transposition table size (averaged for three endgame positions)

Some conclusions can be drawn from the above figures:

- For small transposition table increasing its size (up to 256k entries) significantly increases performance (over 20% lower nodes-to depth number for depths 9 and 10 in opening and middlegame, about 15% in the endgame),
- Middlegame performance is part of the game that is most affected. Slightly less sensitive is the opening phase, and the least – endgame. It is due to the total number of nodes searched at these depths: middlegame positions are usually the most complex in the game, and require highest number of nodes to reach certain depths, while in endgames (where positions are simple due to low number of remaining pieces) significantly less nodes are needed. The more nodes are searched, the more need to be stored to TT – therefore the effect of TT size is most apparent in middlegames.

4.2 Experiments with book learning

As it was mentioned before, programs using automatically generated books are likely to suffer from imperfectness of the books. The reasons have been described earlier in the work, but they are repeated here for convenience:

- Source games might contain blunders that make the game lost right away,
- Many lines from the book can lead to positions that the engine cannot handle well, leading to suboptimal play and falling into a poor position.

Since the books are generated from very large databases (in order to cover a wide range of possible openings and their variants), it is usually infeasible to manually tune the book. It is profitable to make engine recognize poor lines and mark them as such. Then, when the engine encounters a critical position it can choose the move that has proven itself as the best, or (if all seem poor) start thinking on its own.

4.2.1 Basic information about book learning

The problem with book learning as described above is to find a good strategy for evaluating openings. Hyatt ([Hyatt, 2003]) describes in his online paper two approaches: result-driven learning and search-driven learning.

4.2.1.1 Result-driven learning

In the first scheme the basis for evaluating openings is the result that program achieves using them. It assumes a significant correlation between correct choice of opening and future win in that game. A correlation certainly exists, but it is not likely to provide satisfactory learning performance. There are two main reasons for not using this approach as the only one:

1. Games last much longer than most opening book variations, so it is impossible to determine in case of a loss whether the opening was really poor or simply the opponent was much stronger. Similarly, a win might not be the outcome of a brilliant opening line, but simply the opponent could be weak. Such situations would confuse the learning process and lead to incorrect evaluations.
2. After playing many games against a much stronger player most book lines may be marked as “unusable” – although it is not the opening book to blame for the losses. As a result, program will switch to playing exotic variants or will refuse to use the book at all very early in the game – that would not bring any profits to the engine,

and the learning time would be wasted as the book would need being replaced with a new one.

4.2.1.2 Search-driven learning

In this case, the main factor in evaluating opening lines is the result of program's search after the book is left. The simplest solution uses the result of the first search after book line is finished. The problem in this case is that certain openings, especially including material sacrifices (called *gambits* by chess players) may be handled improperly. The reason is that the material part of evaluation is dominant is most chess programs, and they are usually unable to notice the positional compensation immediately. Therefore, the variant would be marked as poor, although the position might be really good for the program, and the engine might achieve significant advantage just a few moves out of the book. The sensible solution is to use value of the search several moves after leaving the book. Usually after a couple of moves the possible sacrifices are cleared up, and either the material equilibrium is restored, or positional advantage becomes apparent enough for the engine to compensate for existing material losses. Such approach can also catch some cases when the engine is unable to find proper plan – after making several suboptimal moves it is likely that the opponent achieves some advantage, that appears in the search value and can be reflected in the opening evaluation. The line leading to improperly understood position can then be omitted in future games.

4.2.2 Implementation of book learning in Nesik

Book learning in Nesik is mainly search-driven, but takes into consideration achieved results if two moves have similar learning records. The algorithm is based on the one described in [Hyatt, 2003].

The modified opening book entry (for each position) looks as follows:

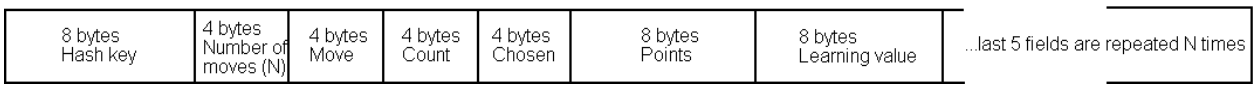


Fig. 36. Opening book entry used for book learning in Nesik

Description of fig. 15:

- Hash key - position's identifier (described in the previous part in the chapter describing Nesik's opening book),
- Number of moves (N) – number of book moves for the given position,
- Move – a single book move,
- Count – number of occurrences of the given move in the original games database,

- Chosen – number of games in which Nesik has used this move,
- Points – number of points Nesik has gathered by playing the move,
- Learning value – value obtained in the process of search-driven learning.

The fields: Chosen, Points and Learning value are updated at the end of each game. Below is the algorithm for determining learning value:

1. *SCORE = search_value_at_10th_own_move_after_leaving_the_book;*
   // ^^^ if 10th move is available, otherwise the last available.
2. *if (SCORE > 500) SCORE = 500, else if (SCORE < -500) SCORE = -500;*
3. *DELTA = SCORE * DepthWeight[SCORE_depth] * LearnLengthWeight[SCORE_length];*
4. *if ((opponent_has_significantly_higher_rating) && (DELTA > 0))*
   *DELTA *= BONUS_FOR_BETTER_OPPONENT; // (BONUS > 1)*
   else
   if ((opponent_has_significantly_lower_rating) && (DELTA < 0))
   *DELTA *= PENALTY_FOR_WORSE_OPPONENT; // (PENALTY < 1)*
5. *if ((opponent_has_significantly_higher_rating) && (RESULT == WIN))*
   *DELTA *= BONUS_FOR_WIN_WITH_BETTER_OPPONENT; // > 1*
   else
   if ((opponent_has_significantly_lower_rating) && (RESULT == LOSE))
   *DELTA *= PENALTY_FOR_LOSE_WITH_WORSE_OPPONENT; // < 1*
6. *LEARNING_VALUE = (LEARNING_VALUE + DELTA) / 2;*

Algorithm 7. Book-learning: determining learn value in Nesik

The above algorithm uses the search value at the 10th move after leaving the book. If the 10th move is not available (the game ended before, for example the opponent resigned) then the score for the last move available is used. That score is then restricted to the interval [-500, +500] (where 100 is the reference value equal to one pawn), in order to avoid too rapid changes of the evaluations (for example, opponent's blunder in one game shouldn't make the whole line marked as wonderful for a long time, as it could happen if the score could take arbitrary value).

The score is then treated with two factors. The first one, DepthWeight, is dependant on the search depth that was reached at the 10th (or last available) move beyond the book. The greater depth, the higher the weight (its value is low, below 0.2 for shallow searches (depth < 6), then rises quickly to 0.9 for depth = 13 and slowly approaches 1.0 for higher depths).

The idea beyond this scaling factor is that deeper searches are much more reliable – shallow searches can overlook many important aspects of the position, therefore they should be trusted less.

The second weight depends on the number of moves that were made after leaving the book. If the base score comes from the 10^{th} move after leaving the book (as it should) then this weight has value of 1.0 (fully trusted), if the number of moves was smaller then the trust is severely reduced (only 0.2 for 5^{th} move after leaving the book).

The DELTA, equal to SCORE modified by these two factors, can be further modified if:

- the opponent has much higher rating (two intervals are used: stronger by more then 100 points and stronger by more than 300 points) and
    - DELTA is positive – DELTA is rewarded a bonus (as it is probably a good line, as a better opponent should not let Nesik achieve advantage if Nesik played a poor opening),
    - Nesik won the game – a win against a better engine is not likely to occur if the opening was played poorly – reward DELTA.
- the opponent has much lower rating – a symmetrical case
    - DELTA is lowered if it is already below 0 (as Nesik should be able to achieve advantageous position against a weaker opponent if the opening was played correctly)
    - DELTA is decreased if the game was lost by Nesik.

To make the learning process smoother, the resulting DELTA is averaged with the old Learning value. The final value is put in the appropriate record of the opening book file, and fields Chosen and Points are updated as well. The process of updating book can be represented as follows:

```
delta = findDelta();   // find evaluation of the used opening line
for each (book_positions_in_last_game) {   // for each book position from last
                                           //game, starting with the latest one
    possible_moves = 0;
    bad_move =false;
    for (each_possible_move_in_book_position) {
        learning_value = book.readLearningValue();
        if (move_played) {
            chosen = position.readChosen();
```

```
                    position.writeChosen(chosen+1);
                    points = position.readPoints();
                    position.writePoints(points+result);
                    learning_value = (learning_value + delta) / 2;
                    position.writeLearningValue(learning_value);
                    if (learning_value < PLAYABILITY_THRESHOLD) {
                        bad_move = true;
                    }
                }
                if (learning_value > PLAYABILITY_THRESHOLD) {
                    possible = possible + 1;
                }
            }
            if (possible > 0 ) {    // if at least one playable move is present in that
                                    // position
                if (bad_move) {     // if the chosen move is not playable then don't
                                    // propagate bad score up
                    delta = 0;
                }
                else {
                    delta = delta / possible;       // decrease delta by a factor
                                                    // proportional to the number of
                                                    // possible moves in that
                                                     // position.
                }
            }
        }
```

Algorithm 8. Book-learning: updating the book

The above pseudocode illustrates, how the individual book entries are updated after the game. The main idea is that the learn value, determined using algorithm 7., is applied to the last book position reached during the game. Then, the learn value (delta) is reduced as many times as there are playable (learn value above PLAYABILITY_THRESHOLD) moves in that position. Then the learn value is updated to the previous book position etc.

If certain move that was played in the game turned out to be unplayable, but alternatives to that move exist in the book, then rest of the learning is skipped (it behaves if that one move was simply removed from the book, as it will never be used again). If however the unplayable move is the only book move, then the bad score is propagated upwards. The idea of splitting learn value in case when alternative moves are present is shown on the below figure

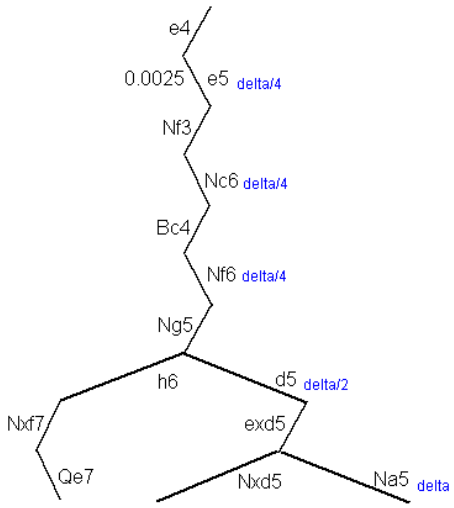


Fig. 37. Book learning – changes of the learn value along the opening line.

The above tree represents a tiny opening book. Nodes of the tree represent positions, while branches correspond to moves. The learn value for line finishing with Na5 is found, according to algorithm 7., to be “delta”. Therefore, move Na5 is assigned value:

learn_value_0 = (learn_value_0 + delta) / 2.

Delta is then divided by two, as there are two moves possible in that position: Nxd5 and Na5. Then move d5 in position one move earlier is assigned value:

learn_value_1 = (learn_value_1 + (delta/2)) / 2.

Delta is divided by two again (as d5 has an alternative: h6) etc.

Another aspect of book learning is how to choose a move when a game is played. The simplest solution considering learning – sort by learning value first, then by results (Points/Chosen), and finally by the Count field, and then pick the first one on the list may lead to the engine repeating the same line again and again. Engine that keeps playing the same line wherever possible is considered boring, and it is possible that the resulting position is not understood properly, leading to high search and learning values, but very poor actual results.

The way Nesik picks moves is described below:

```
BEST_SCORE = -INFINITY;
BEST_PERFORMANCE = -INFINITY;
BEST_COUNT = -INFINITY;
BEST_MOVE = null;
for each book move
    if ( (learning_mode) &&
    (move.CHOSEN == 0) &&
    (move.COUNT > 2*average_count) ) {
        return move;
    }
    LEARN_SCORE = move.LEARNING_VALUE;
    PERFORMANCE = move.POINTS / move.CHOSEN;
    COUNT = move.COUNT;
    if ( (move.CHOSEN > 3) && (PERFORMANCE < 0.4) )
        LEARN_SCORE /= 2.0;
    if (LEARN_SCORE > BEST_SCORE) {
        if ( (LEARN_SCORE > (BEST_SCORE+6.0) ) ||
        (BEST_PERFORMANCE < PERFORMANCE*1.5) ) {
            BEST_MOVE = move;
            BEST_SCORE = LEARN_SCORE;
            BEST_PERFORMANCE = PERFORMANCE;
            BEST_COUNT = move.COUNT;
        }
    }
    else
    if ( (BEST_SCORE – LEARN_SCORE) < 6.0) {
        if ( (PERFORMANCE > (BEST_PERFORMANCE*1.5) ) ||
          (COUNT > BEST_COUNT)  ) {
            BEST_SCORE = LEARN_SCORE;
            BEST_PERFORMANCE = PERFORMANCE;
            BEST_MOVE = move;
            BEST_COUNT = move.COUNT;
        }
```

```
        }
    }
    if (BEST_SCORE > BAD_MOVE_THRESHOLD) {
        return BEST_MOVE;
    }
    return null;
```

Algorithm 9. Book-learning: choosing a move from the book

The main idea behind this algorithm is to choose a move with the best learning value. That value is retrieved from the opening book entry, however if performance of that move is poor (percentage of achieved points is lower than 40%, and more than 4 games have been played) the learned value is divided by two, and the divided value is used for further computations. If another move has the learned value close to the best one (worse by no more than 6.0 units) then it is chosen as the best move if: its performance is better by a factor of 1.5 than performance of the best move found before, or it appeared more often in the original games database.

These additional conditions should avoid making program play the same moves over and over again if they lead to poor results.

Another aspect is "*learning mode*" – if this mode is selected then engine tries to pick moves that have not been tried yet, and occur often (count > 2*average_count) in the database games. The idea is to cover more lines from the book, that would not even be tried if learning value was the only criterion. In tournaments or rated games learning mode should probably be switched off – engine will stop experimenting then, and concentrate on utilizing learned values.

4.2.3 Demonstration of book-learning

In order to verify operation of book-learning in practice an experiment was performed. A small pre-prepared book was used for this purpose. The book contained three lines from so called "two knights defence": 1. e4 e5 2. Nf3 Nc6 3. Bc4 Nf6.
Root position of this opening is shown on the picture:

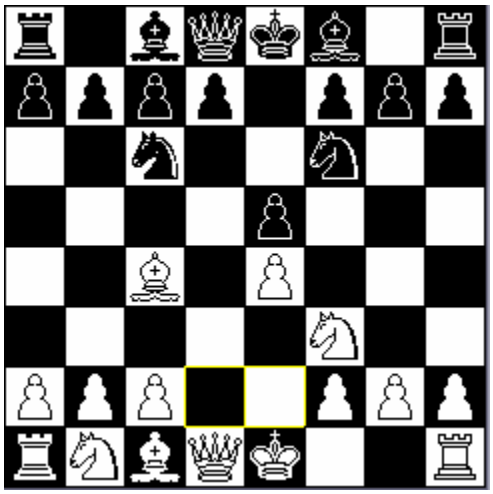


Fig. 38. Two knights defence

The lines that were present in the book were:

I. 4. Ng5 d5 5. exd5 Na5 6. Bb5 c6 7. dxc6 bxc6 8. Be2 h6 9. Nf3 e4

II. 4. Ng5 h6 5. Nxf7 Qe7 6. Nxh8

III. 4. Ng5 d5 5. exd5 Nxd5

The opening lines I, II, III can be presented as a tree, where nodes represent positions (the top-most node, root, is chess initial position) and labelled branches mean moves:

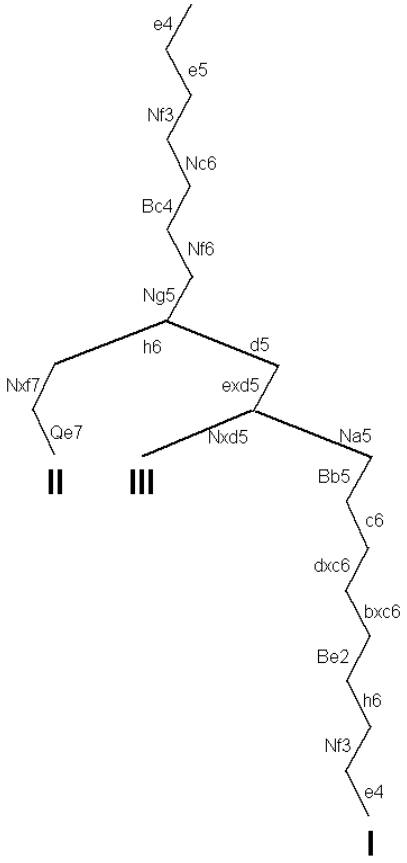


Fig. 39. Tree representing three lines from Two Knights Defence

The positions corresponding to these lines (leafs I, II, III of the tree) are shown below:

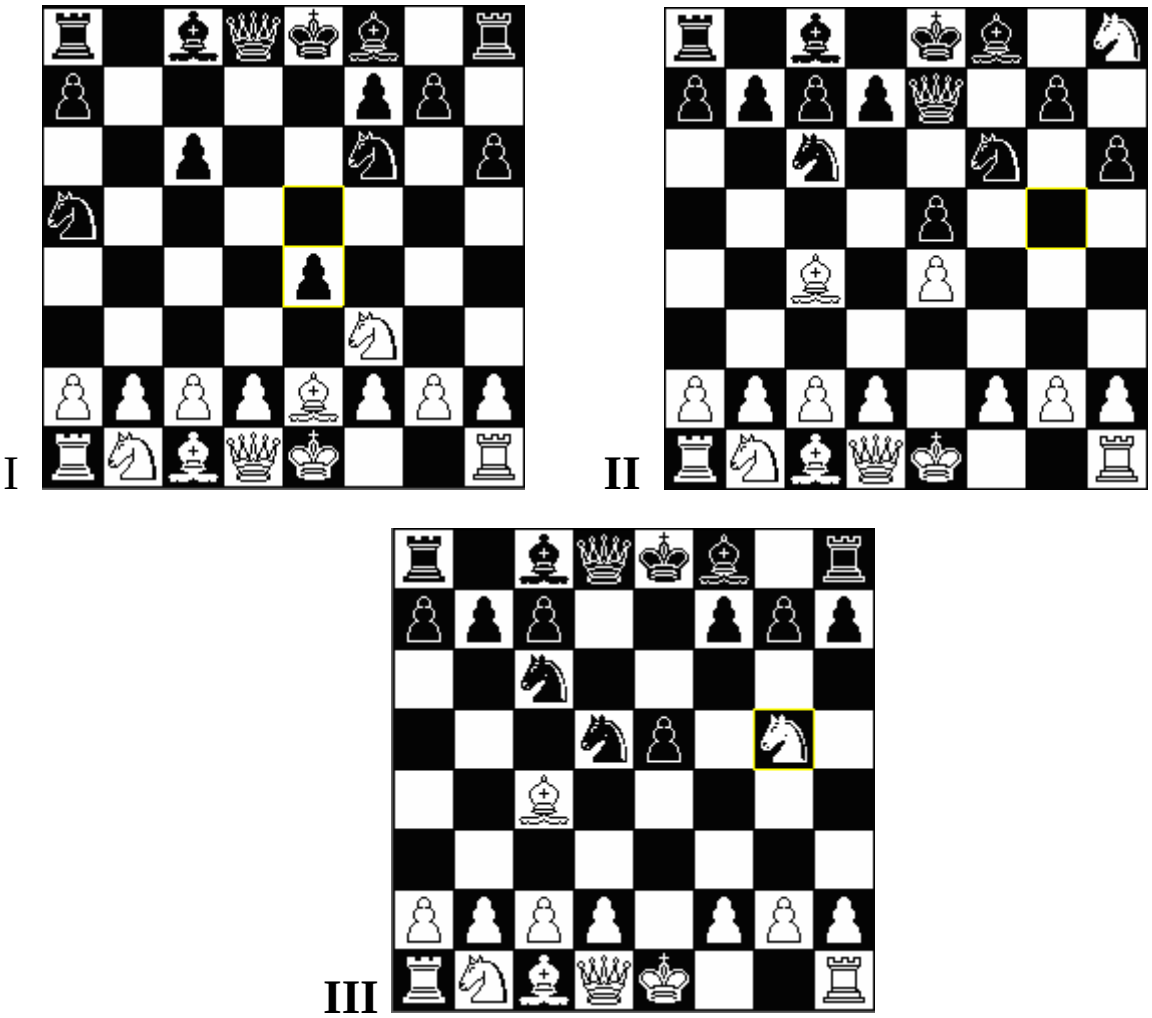


Fig. 40. Positions at the end of opening lines (I – left-top, II – right-top, III – down)

These positions correspond to three different types of variations:

I. A gambit – white is a pawn up, but black has satisfactory compensation due to activity,

II. A blunder was played (4…h6??) – black's position is totally lost,

III. Normal opening position, minimal white's advantage.

Nesik played these positions with black pieces. The idea was to verify whether Nesik is able to properly recognize and learn the first position as satisfactory and playable (despite being one pawn down), the second – as a line to avoid and the third – as a normal line that can be played.

The same lines were put into another program's book (TSCP – Tom's Simple Chess Program). The games were played at a fast pace (2 minutes per program per game). Nesik was put into "learning mode" in order to force it to try all moves.

Learning process is shown below:

Game one (line I.)

On leaving the book Nesik evaluated its position to be -0.30 (30 hundredths of a pawn down). This evaluation (minimal opponent's advantage) suggests that Nesik recognizes its being a pawn down, but sees a significant compensation for it, since the score is only a little below 0. After ten moves Nesik found its position to be +2.16. The position after these moves is shown below:

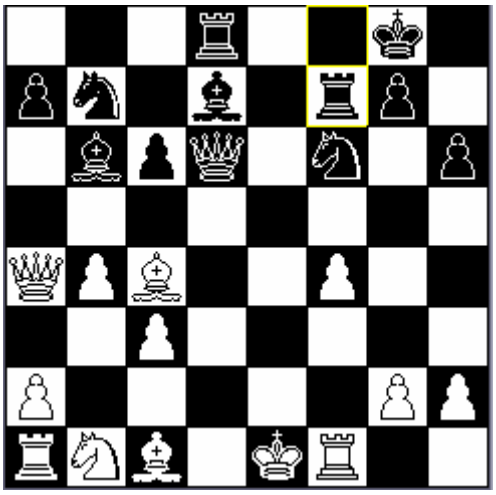


Fig. 41. Position after ten moves after the book line ended (line I)

Such a high score (equivalent to being over two pawns ahead) might be caused either by having strong position at the end of the book and brilliant play later on, or simply by weak opponent's play. This time probably they were opponent's mistakes that allowed Nesik to achieve a won position, however the engine is unable to verify any of these hypotheses on its own, so it makes a reasonable decision to mark this line as profitable, in order to repeat it in future. The learning values of this line's moves after applying the learning process (all weights and modifiers included) look as follow:

1. e4 **e5** (0.0645)
2. Nf3 **Nc6** (0.0645)
3. Bc4 **Nf6** (0.0645)
4. Ng5 **d5** (0.129) (alternative: 4…h7h6)
5. exd5 **Na5** (0.258) (alternative: 5…Nxd5)
6. Bb5 **c6** (0.258)
7. dxc6 **bxc6** (0.258)
8. Be2 **h6** (0.258)
9. Nf3 **e4** (0.258).

The bold moves were played by Nesik, the numbers in brackets are learning values that were assigned to these moves. The learned values are relatively low compared to the value 2.16 that was the base for calculations of learning values – the reason is that such high values are weakly trusted (probably a blunder was made by opponent), as mentioned in the algorithm description.

Game two (line II.)

On leaving the book Nesik evaluated its position to be -6.78 (almost seven pawns down). It is an obvious consequence of being a rook and a pawn down, without any compensation. After ten moves Nesik found its position to be -4.91 – still dead loss.

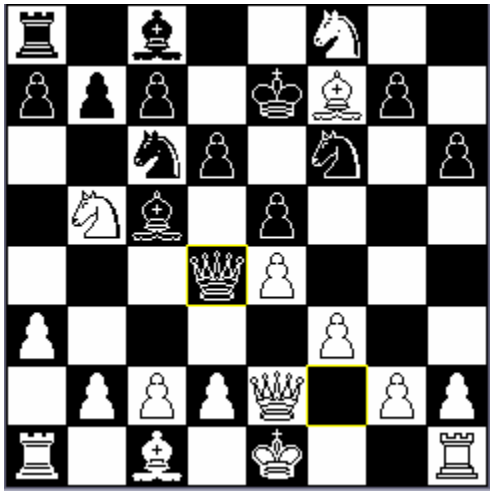


Fig. 42. Position after ten moves after the book line ended (line II)

This is how that score is reflected in the book (including changes already made to the book by the previous game):

1. e4 **e5** (0.0645)
2. Nf3 **Nc6** (0.0645)
3. Bc4 **Nf6** (0.0645)
4. Ng5 **h6** (-1.989) (alternative: 4…d5)
5. Nxf7 **Qe7** (-1.989)

The main conclusion from that game is that move 4…h6 is marked as unplayable (score below the -0.75 threshold) – Nesik will never repeat that move in any game. This conclusion is perfectly correct, as 4…h6?? is a horrible blunder that should never get into the book. And according to the algorithm, the unplayable move has no influence on the earlier part of the line if only there are playable alternatives to it. For example, instead of 4…h6 (that appears to be a blunder) black could pick move 4…d5 (that has the score above the “playability” threshold), so the bad score does not propagate up to 3…Nf6 – in accordance with the algorithm.

Game three (line III.)

This time Nesik estimated its position as -0.04 on leaving the book (equal position), and the score dropped slightly to -0.17 (corresponds to an almost equal position) after ten moves.

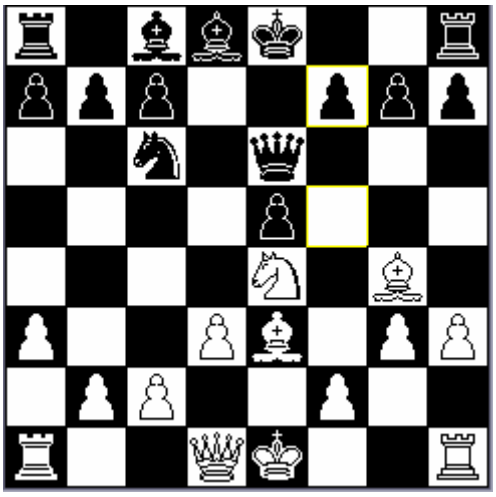


Fig. 43. Position after ten moves after the book line ended (line III)

When book-learning function applied the score -0.17 to the book, it became as follows:

1. e4 **e5** (+0.0025)
2. Nf3 **Nc6** (+0.0025)
3. Bc4 **Nf6** (+0.0025)
4. Ng5 **d5** (0.03475) (no alternative, as 4…h6 was recognized to be a blunder)
5. exd5 **Nxd5** (-0.0595)

The values learned for the first four moves are very close to zero in both playable lines (I and III, line II has an unplayable fourth move). Since Nesik achieved a much better position after 5…Na5 (line I), that move has a little higher learned value than 5…Nxd5 (line III). Another observation is that the blunder (4…h6 in line II) was determined and isolated quickly, and once found it did not influence the book anymore.

The tree with learned values included after these three games looks as follows:

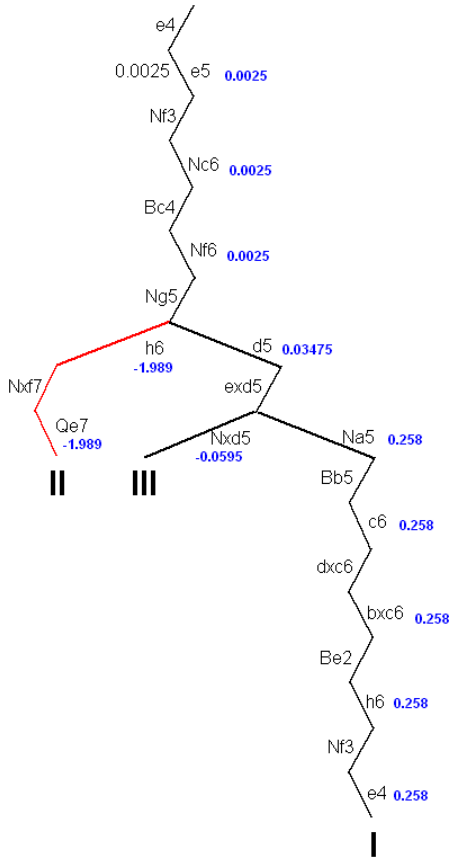


Fig. 44. Tree representing three lines from Two Knights Defence, after three learning games (red colour – unplayable line, blue – learning values)

Later on, Nesik played a gauntlet tournament against 4 engines: Movei, Skaki, Trace and TSCP. It was 6 games against each opponent, that summed up to 32 games, all played at 2 minutes per player per game. Nesik played all games with black, with book learning on. The aim of this tournament was to allow Nesik settle at some learning values.

Values after the tournament are shown on the next figure:

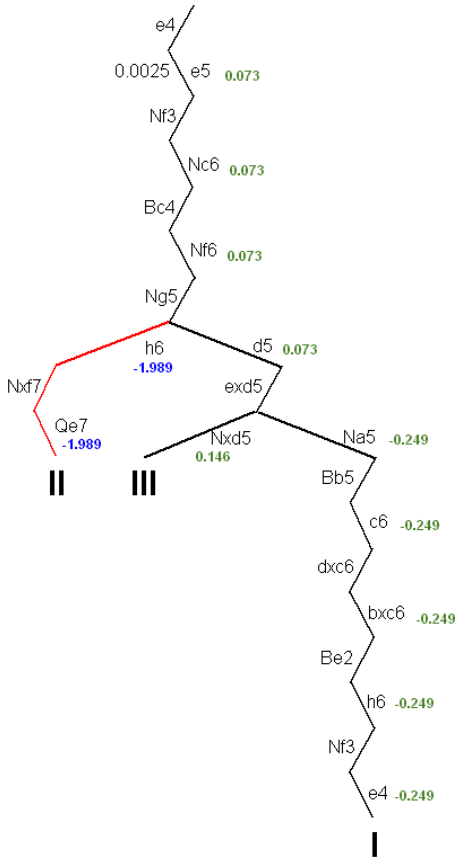


Fig. 45. Tree representing three lines from Two Knights Defence, after a 32-games tournament (green colour – learning values changed in the course of tournament)

The above figure shows that Nesik is able to recognize and dislike bad lines, and pick a favourite move out of the remaining ones. Since the learned values decrease along the way up the tree due to branching, the values at top of the tree are small (the absolute value of the root is only 29% of the absolute value of leaf of line I). They should be even smaller in case of large books, since they exhibit high branching factor and the learning value decreases fast along its way towards root. The algorithm allows Nesik to pick not only the move with highest learned value, but also the one with a little lower score if it lead to significantly better results in the past. Therefore, some variability is introduced to the choice of variations on the initial moves, however lower in the tree, where the learning values are likely to be more diversified, the best line found is followed.

4.2.4 Experimental set-up and results

Tests were performed in order to estimate usefulness of the book learning scheme used in Nesik. Each test consisted of three parts:

1. Games with a plain book, book learning switched off – in order to establish the initial performance of Nesik with respect to competitors,
2. Learning phase – games are played with book learning on, games results from that part were unimportant,
3. The same as 1., using the book from 2. – in order to determine change of engine's performance due to the phase of book learning.

Two experiments were performed: first was a gauntlet tournament, where Nesik played again 5 other engines, and the other one was a match against another program.

Arena software managed the tournaments, and a free ELOStat tool was used to establish ratings of the programs. The description of calculating the Elo ratings can be found in [Elo, 78], as well as in many online sources, for example: (http://en.wikipedia.org/wiki/Elo_rating_system).

The time limit for all games was set to 4 minutes base time and 1 second increment after each move. Hardware used was: a PC computer, processor AMD Athlon XP1700+, 512MB RAM. Each program was configured to use as close to 64MB of memory for hash tables (excluding pawn hash table – that one was set to 2MB in programs that allowed changing this parameter) as possible.

4.2.4.1 Tournament (gauntlet)

Five chess programs (freely available on the Internet) played 20 games each against Nesik. Each program used its own book, book- and position-learning was switched off for these programs that had these features implemented.

The programs used as opponents included programs of strength similar to Nesik and greater, using small and large books – diversity of opponents was supposed to ensure a reasonable degree of results' portability. Moreover, comparable strength of the engines (with the exception of Beowulf, that was supposed to be significantly stronger) should make the influence of opening book greater than in case of engines representing very different levels, as a much stronger opponent should win by a wide margin whether the opening book is good or not.

Nesik used a large 10-plies deep book consisting about 200,000 positions. The book was built from games of grandmasters and masters rated at least 2500 points (FIDE rating).

The results of the first stage (before learning) were as follows:

Programs	Outcome (for Nesik)	Wins-Losses-Draws	Results of games
Nesik - Beowulf	4 / 20	2-14-4	=000==000100000010=0
Nesik – Butcher	7,5 / 20	7-12-1	00010100010101101 00=
Nesik – Delphil	4,5 / 20	2-13-5	00=0==000100000=00=1
Nesik – Gerbil	8,5 / 20	6-9-5	11=01100100=000=0=1=
Nesik - GreKo	12 / 20	9-5-6	11=1===1011000011==1

Table 3. Results of the gauntlet tournament before learning (own research)

After finishing this tourney, book learning in Nesik was switched on and learning took place. The second stage included the same engines and tournament configuration, but this time Nesik played 35 games against each program. This accounted for 175 games that were used by Nesik to tune its opening book.

When the second part was finished, the third (verification) stage was ran. It was exactly the same tournament, as in part 1, with the exception that Nesik now used its tuned book.

The results are shown below (NesikL means Nesik after learning):

Programs	Outcome (for Nesik)	Wins-Losses-Draws	Results of games
NesikL - Beowulf	4,5 / 20	3-14-3	1000100000000==100=0
NesikL – Butcher	5,5 / 20	4-13-3	01000==000010010=010
NesikL – Delphil	9 / 20	6-8-6	===1=11000010=10001=
NesikL – Gerbil	14,5 / 20	13-4-3	001111=11=11=1101011
NesikL - GreKo	6,5 / 20	5-12-3	=00110=0000110001=00

Table 4. Results of the gauntlet tournament after learning (own research)

The results from both tournaments correspond to following ratings (calculated by ELOStat, starting point set to 2100):

Program	Number of games	Score	Ratio Score/no of games	Rating (Elo)	Rating +	Rating -
Nesik	100	36,5	36,5%	2044	69	58
NesikL	100	40	40%	2070	68	61
Beowulf	40	31,5	78,8%	2282	81	150
Butcher	40	27	67,5%	2181	92	136
Delphil	40	26,5	66,2%	2171	93	104
Gerbil	40	17	42,5%	2002	102	104
GreKo	40	21,5	53,8%	2080	109	94

Table 5. Rating results of the gauntlets (own research)

The “Rating +” and “Rating -“ columns describe the 95% confidence interval around the Rating value. Since Nesik has played the greatest number of games (100), this interval is significantly smaller for it compared to other programs.

The Elo ratings are not absolute values – they represent relative difference of strength between players. For instance, a 100 points difference indicates, that the stronger player should win roughly 2 games for each lose. Therefore, the difference between a player with rating 1500 and one rated 1600 is the same as between 2100 and 2200 points. The conclusion is that, when creating own rating list (as it is being done in this work) one can choose arbitrary reference point, that serves as the start point for the iterative process of determining ratings (that is not described here). If the reference point changes by 200 points upwards, then all ratings from the list are automatically increased by the same amount. In the case of the above ratings the starting point was set to 2100 points (a strong club player level, that all participating programs from the above list present).

As can be seen on the above table, no definite conclusion can be drawn from these data. Nesik performed better after learning, but the difference is not significant. The 95% confidence intervals for Nesik and NesikL overlap strongly, so it cannot be said convincingly that learning increases playing strength.

4.2.4.2 Match

The second experiment was a match of Nesik against Greko. Greko performed on a similar level to Nesik in the tournament described above. The aim of that experiment was to verify usability of book learning for the purpose of match preparation.

This time a smaller book was used (about 6000 positions, up to 24th ply), also made from grandmasters' games. The same time control was used (4' + 1"). Greko is supplied with a small book of its own that was used (the same as in the tournament part of the experiment).

The results from the first part (new book, learning off) are shown below:

Program	Points	Wins-Losses-Draws
Nesik	12,5 / 50	9-34-7
Greko	37,5 / 50	34-9-7

Table 6. Results of the match Nesik-Greko before learning (own research)

Then learning for Nesik was switched on and after allowing it to tune its book for 100 games, the match of 50 games was repeated:

Program	Points	Wins-Losses-Draws
NesikL	22 / 50	18-26-4
Greko	28 / 50	26-18-4

Table 7. Results of the match Nesik-Greko after learning (own research)

The rating comparison this time looks as follows:

Program	# of games	Score	%	Rating (Elo)	Rating +	Rating -
Nesik	50	12,5	25%	1969	133	74
NesikL	50	22	44%	2118	102	94
GreKo	100	65,5	65,5%	2157	56	82

Table 8. Rating results of the matches (own research)

This time the rating difference was much higher, although still not large enough to allow making definite claims about superiority of using book-learning over plain book.

4.2.5 Analysis of the results and conclusions

In both cases the rating achieved by the version after learning was higher, however the number of games was too small to clearly prove that the used method for book learning improves program's performance. Nevertheless, the results suggest some improvement. In case of the tournament increase of rating was small (although, if it persisted after more

detailed investigation, it would still be significant). The reasons for much lower improvement in the first part of experiment than in the second part could be:

- Nesik used much greater book (by a factor of 33), so it required much more time to tune due to much greater number of possible lines,
- Nesik played greater number of learning games (100) against Greko in the second part, while it played only 35 against each engine in the first part, so it was relatively better prepared to face that particular opponent. The effect was probably even more apparent since Nesik's book was smaller and easier to tune in that part, while Greko's book is not large enough to provide its game with sufficiently high diversity.

In the gauntlet tournament a certain trend could be observed. Programs that used large opening books (Butcher – 19MB, Beowulf – 9,7MB) were weakly influenced by Nesik's learning – probably Nesik did not have enough games to learn how to play against the rich repertoire they had at their disposal. Therefore, the results against these engines in the verification tournament were not better at all (even slightly worse).

On the other side, the engines that used small books (Gerbil - 13kB and Delphil – 860kB) performed clearly worse against Nesik after learning. Probably Nesik learned quite a lot about how to play against their opening books.

The exception to that rule is Greko – in spite of a small opening book (40kB) it performed better in the second tournament. The reason could lie in the fact that Nesik's book was tuned against the other four engines but at the price of decreasing its performance against the last engine. It might be a flaw of the learning algorithm or a consequence of too small number of learning games.

The sizes of opening books given in kilo- or megabytes are not precisely determining the number of lines the book contains (as different engines use different book formats) but they give a rough estimation about content of the book.

The conclusion from these two experiments is that book learning (at least as implemented in Nesik) seems to improve program's performance. However, more games are necessary to prove this hypothesis, as the results obtained in the two experiments have two wide confidence intervals, so it is possible that the observed improvements would vanish if more games were played.

The second conclusion is that it appears as it is easier to learn how to deal against one specific opponent – improvement observed in the match is significantly higher than that in the tournament. Another aspect is that the opening book used in the match was

much smaller, so it could be better tuned in the relatively small number of learning games that were performed (although a small book might be a serious handicap against opponent using a large book, as he might play variations that are not present in the small book – one of the reasons why Greko was chosen as the partner was it could use a small book, too).

Increase of rating by 26 points with wide 95% confidence intervals makes it hard to draw conclusions about feasibility of the used book learning scheme. However, the improvement of about 150 points (almost one class) suggests that a serious performance gain can be achieved in matches when a sufficient number of learning games is played again the opponent. Again, definite claims would require a larger number of games in order to increase precision of the experiment.

Better learning performance in the match might indicate that size of the book influences efficiency of learning. A small book can be tuned much faster than a large one. Therefore, although a large opening book can make program seem more "clever" in the openings, it doesn't have to improve its performance because the large book is harder to tune. That would agree with the common sense of many chess players – they use only a part of the whole chess opening theory, a fraction they know and understand well.

4.2.6 Issues for further research

The experiment described in the previous chapters suggests that book-learning as described and implemented in the author's chess program improves performance. However, due to time constraints, only limited number of test and learning games could be played, so the results are subject to large errors that make it impossible to draw definite conclusions. A more comprehensive test including several hundreds of games is necessary in order to obtain exact numerical results.

The games were played at blitz time controls (4 minutes of base time and 1 second per-move-increment). Common opinion is that almost any results obtained using data gathered at fast time control (like blitz) are not necessarily applicable to games played at tournament time controls (1 hour per game or more). So similar experiments should be performed using long time controls (due to much greater time required to complete a single game, and the need for playing hundreds of such games, it should be probably made on many machines simultaneously).

The experiment described in this work does not tell whether the main advantage from learning process is that engine learns its strong points, or opponents' weak points. That is, if program successfully trained by playing games against a set of opponents would

be able to use with similar success its book to face another opponent, not present in the set. Experiment to clarify this issue might provide important suggestions on further improvements of the book learning algorithms.

As in case of many algorithms, the performance of algorithm used in this work might be seriously affected by the choice of its parameters (for example, dependence of learning value on search depth). The parameters used in Nesik were chosen after some considerations, but certainly they could be improved and tuned.

Summary

The main aim of this work was to create a chess-playing program that would be able to compete with strong club human players. Nesik, the engine that was built for that purpose, contains many modern chess-programming techniques in order to achieve that goal. It knows all rules of the chess game (including threefold repetition and 50-moves rules) and features: fully bitboard-based move generation, alpha-beta search algorithm with aspiration window at the root and quiescence search, killer and history heuristics for move ordering, transposition table to exploit graph properties of the game tree, adaptive null-move forward pruning to reduce search effort devoted to unpromising branches, pondering to utilize opponent's thinking time, opening book to employ expert knowledge about game of chess and heuristic evaluation function equipped with a significant set of parameters. Implementation of these methods and techniques makes it play at strength about 2100 Elo rating points, that is somewhere between I category (2000 points) and candidate master (2200) level. Therefore, the main aim was fully achieved.

The program has played about 800 of games online on Free Internet Chess Server (FICS), with good results (record for September 9th, 2004: blitz - 321 wins (+), 132 losses (-), 44 draws (=), rating 2136, lightning - +174, -75, =26, rating: 2346). It also participated in several tournaments, for example:

- WBEC Ridderkerk (5th division), 6th place among 21 participants (http://wbec-ridderkerk.nl),
- Winboard Olympic Chess Tournament – place 2 out of 12 in Polish team semifinal (http://vigo.altervista.org/WOCT3.html)).

Description of the program is part of this paper. It contains also details about implementation of different techniques. Nesik is a console-mode application that uses third-party programs to provide graphical user interface (and a chessboard). It communicates with them using Winboard protocol 2.

Another aim of this work was to provide an overview of modern chess-programming techniques. This part includes a short description of the game of chess from scientific perspective (especially artificial intelligence, game theory and complexity) and state-of-the-art data structures, algorithms and techniques of chess programming. The described techniques include:

- Alpha-beta search algorithm (and its variants: PVS, MTD(f)),

- Transposition table,
- Iterative deepening
- Killer and history heuristic,
- Null move pruning
- Aspiration window technique and minimal window search,
- ProbCut and MultiProbCut pruning,
- Quiescence search,
- Enhanced Transposition Cutoff,
- Various tree shaping techniques (extensions, reductions),
- Futility pruning,
- Lazy evaluation.

Other chess programming information includes several board representations and move generation techniques.

Some non-alpha-beta-based approaches to determining best moves are also briefly presented (Berliner's B*, conspiracy number search).

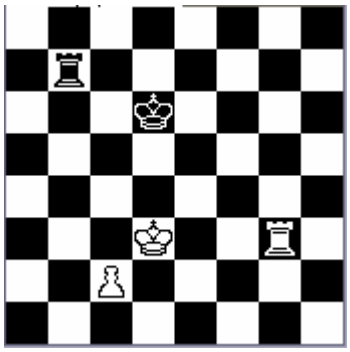
The program that was created along with this paper was used to implement an algorithm for opening book-learning. The modified program was used to perform experiments concerning book-learning. The algorithm was a modified Hyatt's approach using search-driven learning ([Hyatt, 2004]).

Time constraints made it impossible to play enough test games to obtain accurate, statistically significant, results, however the gathered data suggest increase of program's performance from using book-learning (26 Elo points in case of the tournament and about 150 in case of match against a single opponent). The increase is the greater, the more games are played against a specific opponent. It also depends on size of the book – the larger opening book is, the more games are required to tune it and achieve performance gain.

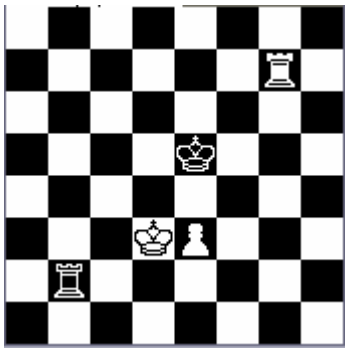
Quantitative results that are presented in this work are one of the first that author of this work has ever seen in literature. Due to lack of time they are not significant, but potential for improvement has been shown, so more accurate experiments can follow in order to confirm and measure that improvement, or reject it.

As opening book learning is not well described in literature yet, there are still many aspects that should be tried, tested and described. Some of them, especially those concerning the used algorithm are mentioned in this work.

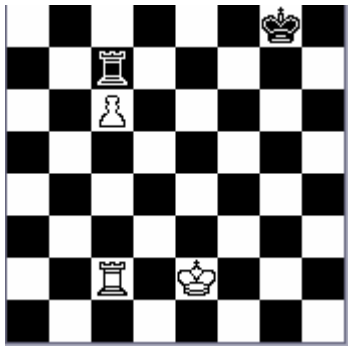
Appendix A – 30 test positions for TT



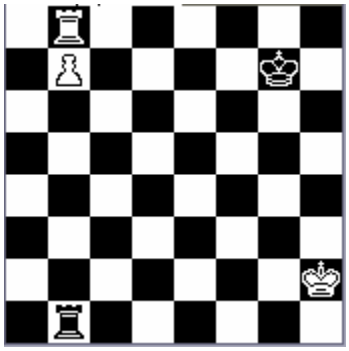
Black to move



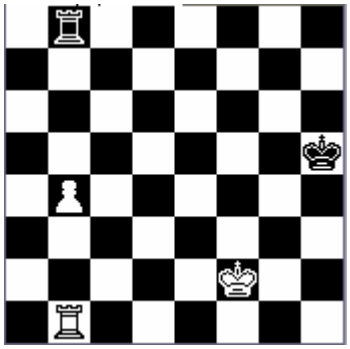
Black to move



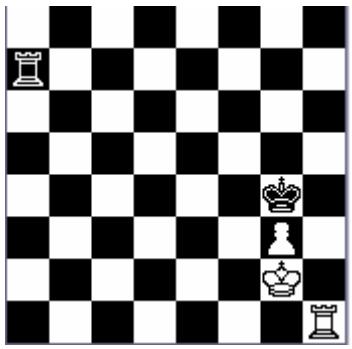
Black to move



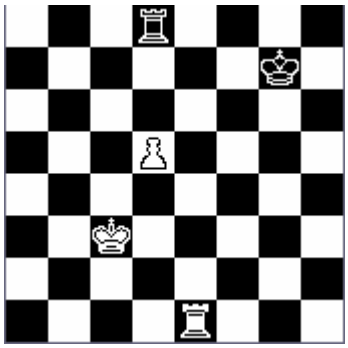
Black to move



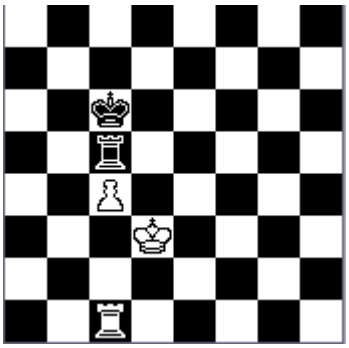
Black to move



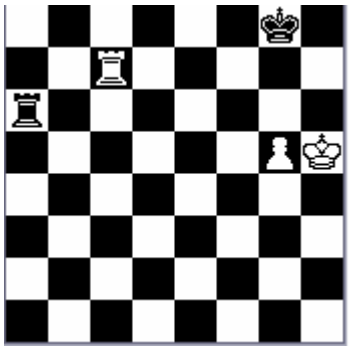
White to move



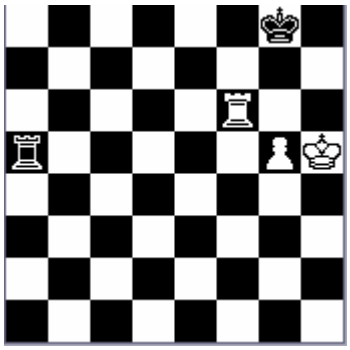
White to move



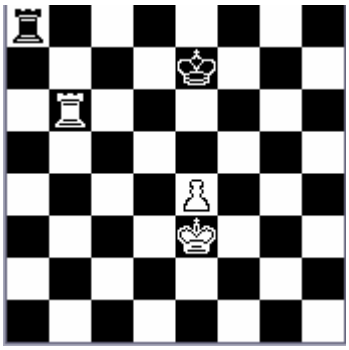
White to move



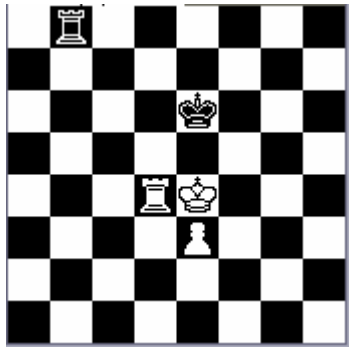
White to move



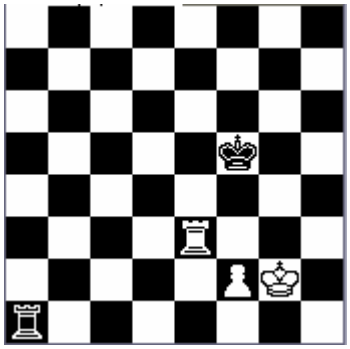
White to move



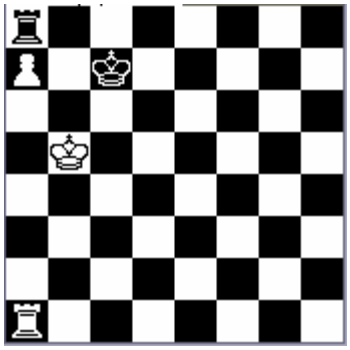
White to move



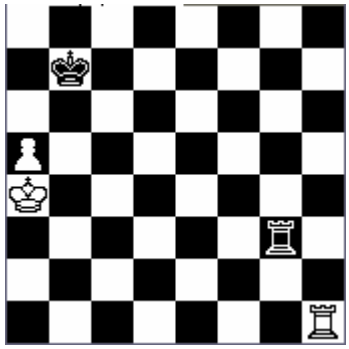
White to move



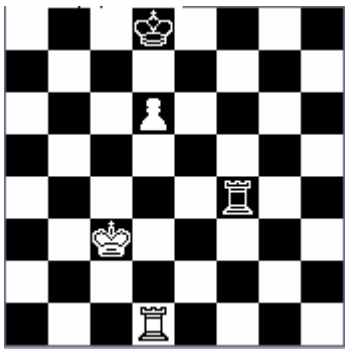
White to move



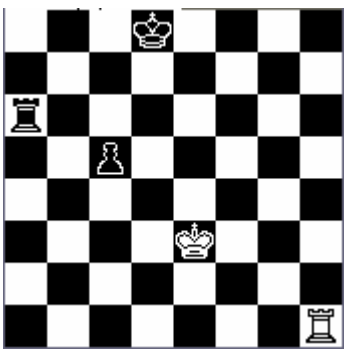
White to move



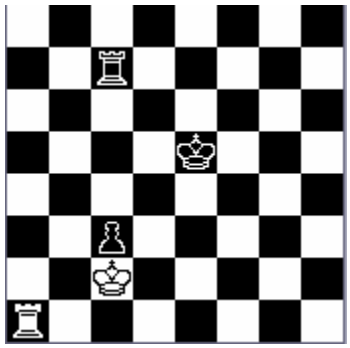
White to move



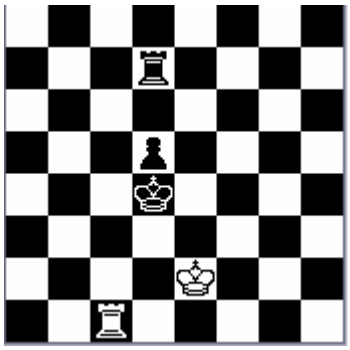
White to move



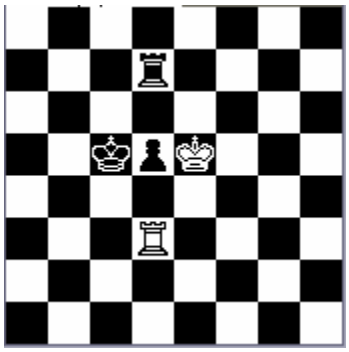
White to move



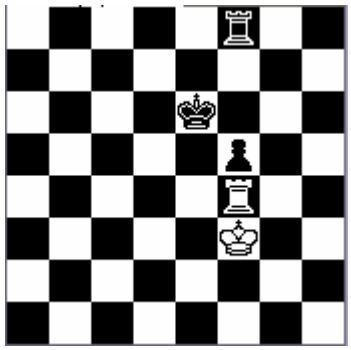
White to move



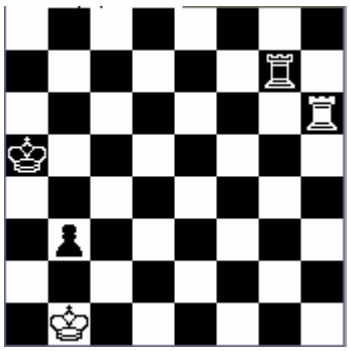
White to move



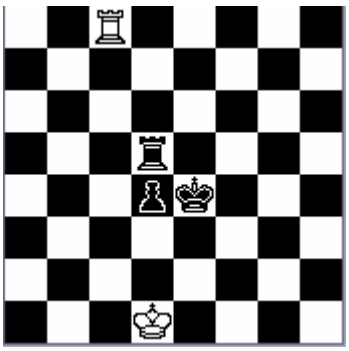
White to move



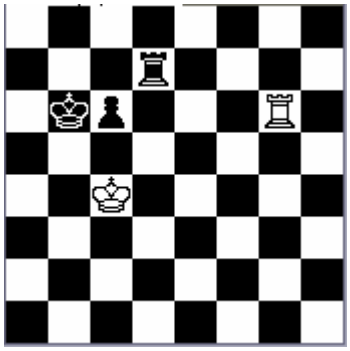
White to move



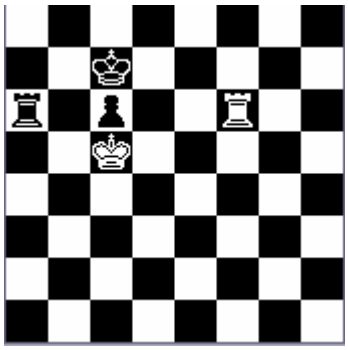
White to move



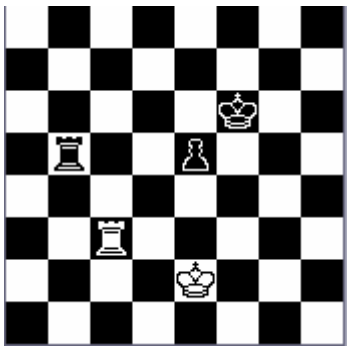
White to move



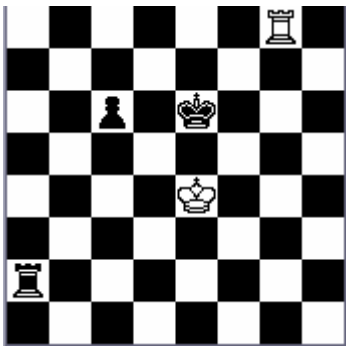
White to move



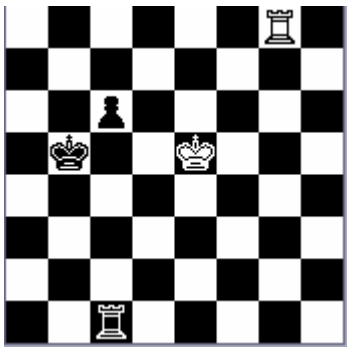
White to move



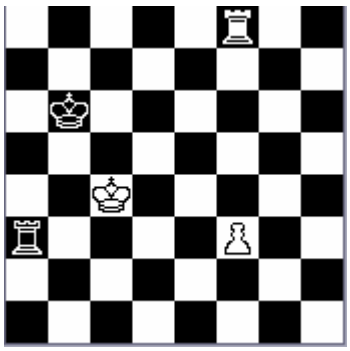
White to move



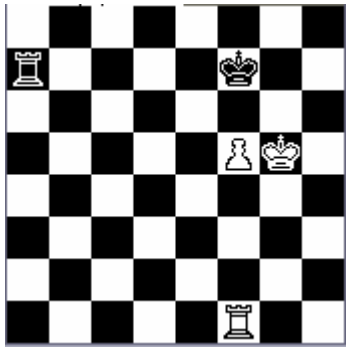
White to move



White to move



White to move



White to move

Appendix B – list of abbreviations

CN	Conspiracy Number
CT	Conspiracy Threshold
ETC	Enhanced Transposition Cutoff
FICS	Free Internet Chess Server (http://www.freechess.org)
FPGA	Field Programmable Gate Arrays
GUI	Graphical User Interface
ID	Iterative Deepening
MVV/LVA	Most Valuable Victim/Least Valuable Attacker
NPS	Nodes per second
PGN	Portable Game Format
PV	Principal Variation
PVS	Principal Variation Search
QS	Quiescence Search
TD	Temporal Difference;
TSP	Travelling Salesman Problem;
SEE	Static Exchange Evaluation
VCL	Visual Component Library

Appendix C – list of figures

Appendix D – list of tables and algorithms

Appendix E – CD content

The attached CD contains:

- Source code and executable of the chess program,
- Files with games played by Nesik,
- Some tables with numerical data used in this paper,
- Opening book and configuration files for Nesik
- Supplementary programs to create opening books from sets of games.
- Installation version of Winboard, GUI for the engine.

They are located in the following directories:

\Nesik\	- Nesik program
\Nesik\src\	- Nesik source code
\Nesik\build_book\	- Program to build books for Nesik
\Books\	- Opening books for Nesik
\data\num\	- Numerical data
\data\pgn\	- Files with games played by Nesik

Bibliography

Books

1. [Drozdek, Simon, 1996] Drozdek A., Simon, L. *Struktury Danych w Jezyku C.* WNT, Warszawa 1996.
2. [Elo, 1978] Elo, A. *The Rating of Chessplayers, Past and Present*. Arco Publishing Inc., New York 1978.
3. [Knuth, 1981] Knuth, D. E. The *Art of Computer Programming, 2nd ed.. Volume 2: Seminumerical algorithms*. Addison-Wesley Publishing Comp. 1981.
4. [Levy, Newborn, 1991] Levy, D., Newborn M. *How Computers Play Chess.* Computer Science Press 1991.
5. [Sedgewick, 1992] Sedgewick, R. *Algorithms in C++*. Addison-Wesley Publishing Comp. 1992.
6. [Slate, Atkin 1977] Slate D.J., Atking L. R. *''CHESS 4.5 - The Northwestern Univ. Chess Program''* in P. Frey, ed., *Chess Skill in Man and Machine*, Springer-Verlag, 1977, p. 82-118.

Journal articles

7. [Anantharaman et al, 1988] Anantharaman, T.S., Campbell, M.S. and Hsu, F. (1988). *Singular Extensions: Adding Selectivity to Brute-Force Searching*. ICCA Journal, Vol. 11, No. 4, pp. 135-143.
8. [Baxter et al., 2000] Baxter, J., Tridgell, A, Weaver, L. (2000) *KnightCap: A Chess Program That Learns by Combining TD(l) with game-tree search.* ICML, pages 28-36.
9. [Berliner, 1979] Berliner, H. J. (1979) *The B* Tree Search Algorithm: A Best First Proof Procedure*. Artificial Intelligence, 12, 1:23-40.
10. [Buro, 1995] Buro, M., *ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm*, ICCA Journal 18(2) 1995, 71-76
11. [Buro, 1995] Buro, M. *Statistical Feature Combination for the Evaluation of Game Positions*. JAIR, 3:373-382.
12. [Knuth, Moore, 1975] Knuth, D. E., Moore, R. W. *An Analysis of Alpha-Beta Pruning.* Artificial Intelligence, 6(4): 293-326.
13. [Shannon, 1950] Shannon, C. *Programming a Computer for Playing Chess*.

Philosophical Magazine, Ser. 7. Vol. 41. No. 314.

14. [Sutton, 1988] Sutton, R. S. *Learning to Predict by the Methods of Temporal Differences.* Machine Learning, 3:9-44.
15. [Tabibi, Netanyahu, 2002] Tabibi, O.D., Netanyahu N.S., *Verified Null-move pruning*. ICGA Journal 25(3): 153-161, 2002
16. [Tesauro, 1995] Tesauro, G. *Temporal Difference Learning and TD-gammon.* CACM, 38(3):58-68.
17. [Marsland, Campbell, 1982] Marsland, T.A, Campbell, M.S. *Parallel search of strongly ordered game trees*. Computing Surveys, 14(4):533–551, December 1982.

Electronic publications

18. [Anantharaman, 1991] Anantharaman, T. *A Statistical Study of Selective Min-Max Search.* PhD thesis, Carnegie Mellon University.
19. [Brockington, Schaeffer, 1996] Brockington, M., Schaeffer, J. *The APHID Parallel Alpha-Beta Search Algorithm.* Technical Report TR 96-07, Univeristy of Alberta.
20. [Buro et al. 2002] Buro, M., Gomboc, D., Marsland, T. A. *Evaluation Function Tuning via Ordinal Correlation*. University of Alberta.
21. [Donskoy, Schaeffer, 1990] Donskoy, M., Schaeffer, J. *Perspectives on Falling From Grace*.
22. [George, Schaeffer, 1990] George, M., Schaeffer, J. *Chunking for Experience*. University of Alberta.
23. [Gillogly, 1978] Gillogly, J. *Performance Analysis of the Technology Chess Program*. PhD thesis, Carnegie Mellon University.
24. [Heinz, 1998] Heinz, E. *Extended Futility Pruning*. University of Karlsruhe
25. [Heinz, 1999] Heinz, E. *Adaptive Null Move Pruning* University of Karlsruhe
26. [Hyatt, 1983] Hyatt, R. M. *Cray Blitz – A Computer Chess Playing Program*. MSc thesis, University of Southern Mississippi.
27. [Junghanns, 1994] Junghanns, A. *Fuzzy Numbers In Search To Express Uncertainty* University of Alberta
28. [Lazar, 1995] Lazar, S. *Analysis of Transposition Tables and Replacement Schemes*, University of Maryland.
29. [Lister, Schaeffer, 1991] Lister, L., Schaeffer, J. *An Analysis of the Conspiracy Number Algorithm.* University of Alberta.

30. [Michniewski, 1995] Michniewski T., *Samouczenie Programów Szachowych*. Uniwersytet Warszawski, 1995.
31. [Plaat, 1996] Plaat, A. *Research Re: Search & Re-Search.* PhD thesis, Erasmus University, Rotterdam.
32. [Plaat, Schaeffer, 1996] Plaat, A., Schaeffer, J. *New Advances in Alpha-Beta Searching.*
33. [Schaeffer, 1990] Schaeffer, J. *The History Heuristic and Alpha-Beta Search Enhancements in Practice*. University of Alberta.
34. [Schaeffer et al., 1993] Lake, R., Lu, P., Schaeffer, J. and Szafron, D. *A Re-Examination of Brute-Force Search.* University of Alberta.
35. [Schaeffer et al., 1996] Bjornsson, Y., Brockington, M., Junghanns, A, Marsland, T and Schaeffer, J. *Diminishing Results of Additional Search in Chess,* University of Alberta.
36. [Schaeffer, 1998] Schaeffer, J. *History Heuristics and Alpha-Beta Search Enhancements in Practice*. University of Alberta.
37. [Schaeffer, 2002] Schaeffer, J. *Applying the Experience of Building a High Performance Search Engine for One Domain to Another*. University of Alberta.
38. [Schaeffer et al., 2002] Hlynka M., Jussila, V., Schaeffer, J. *Temporal Difference Learning Applied to a High-Performance Game-Playing Program.*
39. [Schaeffer, 2003] Schaeffer, J. *lecture notes*.
40. [Simon, Schaeffer, 1989] Simon H., Schaeffer, J. *The Game of Chess*. Carnegie-Mellon University and University of Alberta
41. [Stilman, 1997] Stilman B. *Materials from 15th IMACS World Congress, 1997*
42. [Turocy, von Stengel, 2001] Turocy T. L., von Stengel B. *Game Theory.* CDAM Research Report, October 8, 2001.

Internet sites

43. [Hyatt, 2003] Hyatt R. *Book Learning - a methodology to automatically tune an opening book* (http://www.cis.uab.edu/hyatt/learning.html) accessed on 14/08/2004
44. [Hyatt, 2004] Hyatt R. *Chess program board representation - draft* (http://www.cis.uab.edu/hyatt/boardrep.html) accessed on 18/04/2004
45. [Laramee, 2000] Laramee, F. D. *Chess Programming Part VI: Evaluation Functions (online tutorial)*. (http://www.gamedev.net/reference/articles/article1208.asp) accessed on 10/10/2004

46. [IBM, 1997] Corporate website, *Deep Blue*, (http://www.ibm.com) accessed on 14/04/2004
47. [CCC] Various chess programmers from Computer Chess Club online forum (http://www.talkchess.com)
48. [Schroeder, E., 2003] Schroeder E. *Programmer Stuff* (http://members.home.nl/matador/chess840.htm) accessed on 20/04/2004

Lódz, dn. 14 wrzesnia 2004r.

Marek Strejczek
Nr albumu: 100691
Telecommunication & Computer Science
Centrum Ksztalcenia Miedzynarodowego
Wydzial Elektrotechniki i Elektroniki PL

OSWIADCZENIE

Swiadomy odpowiedzialnosci oswiadczam, ze przedkladana praca magisterska na temat: „Some aspects of chess programming” zostala napisana przeze mnie samodzielnie.

Jednoczesnie oswiadczam, ze ww. praca nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz. U. nr 24, poz. 83) oraz dóbr osobistych chronionych prawem cywilnym, a takze praca nie zawiera danych i informacji, które uzyskalem w sposób niedozwolony.

Zaswiadczam takze, ze niniejsza praca magisterska nie byla wczesniej podstawa zadnej innej urzedowej procedury zwiazanej z nadawaniem dyplomów wyzszej uczelni lub tytulów zawodowych.

……