Genetic algorithms for optimising chess position scoring

Petr Aksenov

06.04.2004

University of Joensuu
Department of Computer Science
Master's Thesis

Abstract

Since the invention of computer, people have been utilising its fast computation ability, and nowadays, computer is far better than human in many routine tasks. However, there are the fields, such as computer chess, in which computer is not superior. Best human chess players still rival best computer chess programs successfully. This might happen due to the fact that chess includes something beyond pure calculations, which is beyond the capabilities of a computer, but what is quite normal for human. As an alternative, this might happen because of inaccuracy how computers estimate the goodness of a given chess position. In this thesis, an effort on the second assumption is made by means of using the advantages of genetic algorithms, which are very well known as an excellent tool for solving optimisation problems.

In this thesis, the main components of a computer chess program are first reviewed. These include the well-known tree search and alpha-beta pruning. Concepts such as quiescence search, null move, and table bases are also discussed. Basic introduction to genetic algorithms is given. An own evaluation function of a chess position is constructed. A genetic algorithm is used for optimising the values of the parameters. The system is implemented and tested using an own chess engine, and the results are presented.

1 INTRODUCTION

1.1 The game of chess

1.1.1 Invention

There is a wide-spread legend that narrates about the invention of the game of chess. I retell it here how I heard it during the school lesson on mathematics when we were introduced the exponential operation. Approximately at the same time I started to make my first steps in playing chess.

In ancient times one Indian sage came to the rajah and showed a game he invented. The rajah liked the game so much that he wanted to award the sage and said that the sage could ask whatever he wanted for the invention of such an interesting game. The sage asked to put 1 grain of wheat onto the first square of a game board, 2 grains onto the second square, 4 grains onto the third, and so on, putting onto every next square twice as many grains as there were on the previous one. And this was the quantity of wheat he would like to get. With cheerfulness, the rajah agreed to accomplish sage's wish. Unfortunately, he could not fulfill his promise because the number of grains the sage asked for, 2^{64}, is much greater than the overall number of grains on the whole Earth.

I think, this legend with one or another variant is known to many people, not only chess players. But the real story of the invention of chess, to which a lot of investigations have been devoted and a number of different scientifically based conclusions have been reported, is not so simple and is still a subject of controversy.

The first hypothesis assumes that the game came to Europe from India [19, 35, 52]. The primary reason is that chess was earliest recorded in a number of Persian and Arabic manuscripts, and in them the game was told to be invented in India. The basic work supporting this version is [35]. It is a fundamental investigation of about 900 pages created in 1913, which still remains the most popular reference for those working on the history of chess.

The second hypothesis states for China as being homeland of chess [19, 42]. In [42] there is a very detailed discussion about different types of Persian-Arabic-Indian, Japanese, Chinese

and modern chess, from which the author concludes the evident similarity of latter two. However, Sloan [42] stated that chess did appear first in China and not in India, but he was heavily criticised by Ghory [18]. Ghory tried to show by means of commenting the cites extracted from [42] that most of the conclusions were unsubstantiated. However, Ghory agrees with Sloan in the opinion that there is no solid evidence about Indian origin of chess. Moreover, there is more evidence for Chinese, rather than for Indian priority.

Thus, there is no universal opinion in the scientific world of historians about the origin of chess. I have never played either Chinese or Japanese chess. But my opinion is that modern chess, which is nowadays so popular and is played in all over the world, is an evolutional aggregation, originally born somewhere in the East and being modified on its way to Europe.

There is a description of modern chess in the following two sections. This information is enough to become able to play and understand how to move and the notations used to record games.

1.1.2 Basic concepts

The modern chess is played on a special square board divided into 64 squares of two different colours (usually black and white, but not necessarily) as shown in Fig. 1.

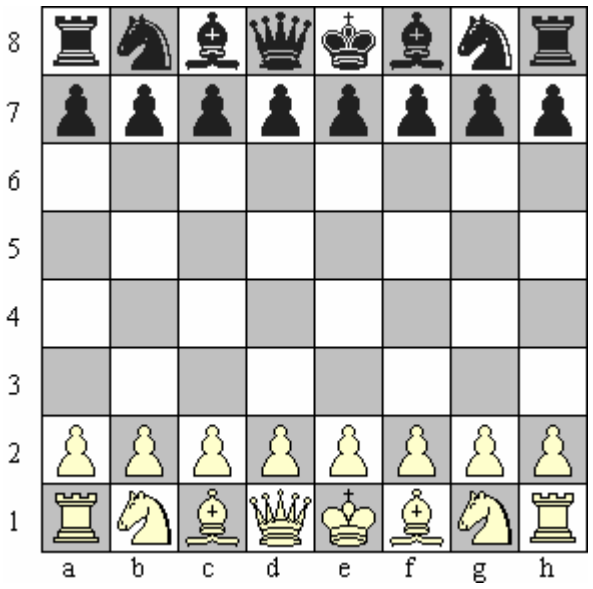

Fig. 1: Initial chess game position.

This board is called *chessboard*. Vertical lines are called *files* and horizontal lines are called *lines*. The words *vertical* and *horizontal* are used as well. Straight line that connects adjacent

squares of the same colour is called *diagonal.* There are two opponent sides, which are referred to as *white* and *black*. In the beginning of the game both sides have an equal set of 16 pieces. The starting position of the pieces is shown in Fig. 1.

1.1.3 Moving rules

White and black, in turn, displace pieces on the chessboard. White starts. Chessboard is put so that the leftmost bottom square is black. A single or in several cases a set of displacements made by one player is called *move.* To distinguish, in computer chess one move by one side is called *ply* and was introduced by Samuel in [39] in order to represent one level in the game search tree (see section 2.2.2). Each piece has its own name and it moves according to a particular rule. The pieces and their moving rules are described next.

Above all, piece cannot move to a square occupied by a piece of the same colour. If a piece of one colour moves to a square occupied by a piece of the other colour, the latter is considered to be *captured* and is removed from the chessboard within the same move. A piece is said to *attack* the square, if it can move to or perform a capture on this square.

Pawn. Pawn moves (but not attacks) one square ahead on the same vertical, if this square is empty. In the initial position, pawn can move two squares ahead (but not attacks), if both squares are empty. Pawn can move one square ahead diagonally, and this is the square it attacks, if this square is occupied by opponent's piece. If pawn moves from its initial position to two squares so that it passes through the square attacked by opponent's pawn, then it can (but not obliged to!) be captured by the corresponding opponent's pawn in immediate reply as if it moved only one square. This type of capture is called *en passant capture*. An example of en passant capture move is shown in Fig. 2 in section 1.1.4. If pawn reaches the last (of its initial position) horizontal, it must be replaced either with knight, bishop, rook, or queen of the same colour. This move is called *promotion*.

Knight (N). Knight can move to one of the nearest squares of its current position, which are not on the same horizontal, vertical, or diagonal. Knight does not affect the squares it passes through.

Bishop (B). Bishop can move any number of squares along the diagonal(s).

Rook (R). Rook can move any number of squares along the horizontal or vertical.

Queen (Q). Queen can move any number of squares along the horizontal, vertical, or diagonal(s), on which it stays at the moment.

Bishop, rook and queen cannot pass through the square occupied by a piece of either colour.

King (K). King can move in two different ways. It can move to any square adjacent to its current location, if the destination square is not attacked by a piece of different colour. King can also *castle*. *Castling* is performed by displacing king along the edge horizontal from its initial position to two squares towards one of the rooks of the same colour, and the corresponding rook is then placed to the square king has just passed through. This manipulation is considered to be one king's move. Castling becomes impossible once king has moved, or if the corresponding rook was moved. Castling becomes temporarily impossible if the square, on which king stays at the moment, which king must pass through, or which king must occupy, is attacked by one or more pieces of different colour. Castling also becomes temporarily impossible if there is a piece between king and the corresponding rook, with which king is to castle.

King is said to be in *check* if its current location is attacked by one or more pieces of different colour. Player cannot make move that keeps his king in check. If player does not have a move that will preserve a capture of his king during the next opponent's move, the king has been *checkmated*, and the player loses the game. If player does not have a valid move and his king is not in check, this means that his king has been *stalemated*, and the game is *drawn*. The game is also considered to be drawn, if none of the players is able to checkmate opponent's king.

1.1.4 Game recording

Every square on the chessboard has a unique address that consists of two symbols. The first symbol stands for the vertical the square belongs to and it is a Latin letter from *a* to *h*. The second symbol stands for the horizontal the square belongs to and it is a number from 1 to 8. For example, from white's point of view left bottom square has the address of *a1*, because it belongs to the first vertical and the first horizontal. Right bottom square has the address of *h1*, because it belongs to the last vertical and the first horizontal. The square, which belongs to

the fourth vertical and the fifth horizontal, has the address of *d5*. The entire picture is shown in Fig. 1.

Every official chess game, i.e. one that occurs in an official competition of any level, is recorded. It allows chess players to analyse chess games and try to find better moves, which, in case of success, is announced publicly. It is done in order to continually improve the quality of playing, and this is the reason why there exist *chess databases*. This issue will be discussed more in section 2.2.11.

Since only one piece can occupy one square in a certain position, and vice versa, the universal way to record a chess move is to write down address of the departure square followed by address of the destination square. Simply, a game can begin with the following move sequence: *b1c3, e7e5, e2e4, b8c6.*

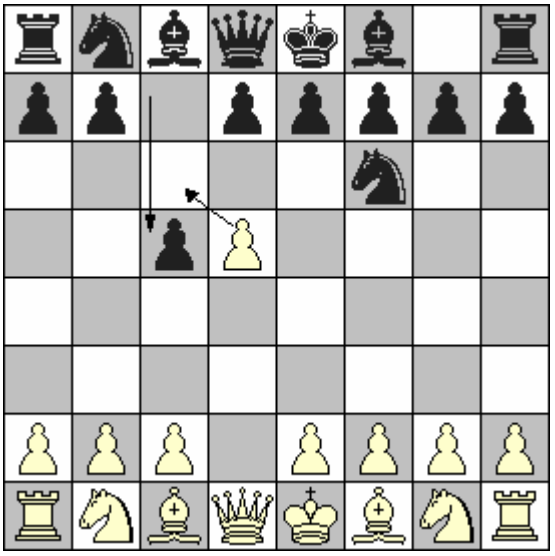

Fig. 2: En passant capture move. Position after *1. d4 Nf6 2. d5 c5*. Black pawn has just moved from c7 to c5, white pawn on d5 can capture it en passant, *3. dxc6*, just like if it moved to c6.

But for better readability and faster understanding, some simplifications were involved into the recording process. Instead of indicating the departure square, a letter for the moving piece is written. These letters were given in section 1.1.3 in brackets after the names of the pieces. The only exception is on pawn, for which no letter is used. Usually, moves in the game are also numbered. Thus, the above sequence usually looks like *1. Nc3 e5 2. e4 Nc6.* In some positions, several (usually, two) pieces of the same kind are able to move to the same square. In this case, a qualifying symbol is added after piece's letter. For example, *Nbd2, N7e5,*

Rad1, R7g8, Rff8. Castling with *h* rook is written as *0-0,* and castling with *a* rook is written as *0-0-0*.

If move is a capture move, symbol *x* is appended right before the destination square address *(exd4)*. If move produces a check, symbol + is appended to the end of this move *(Nbxd8+)*. If move checkmates a king, symbol *X* is appended to the end of this move *(Qxg7X).*

1.2 The game of chess and Artificial Intelligence

If asked, most people now would answer that at least once they have heard the phrase *Artificial Intelligence,* or its abbreviature, *AI.* A part of them would say that it concerns mostly computers. Only a few people would be ready to tell properly what AI is, the problems it solves, the methods that are used, and other related issues. I do not belong to the group of experts in the area of AI, and present work is not a research in that field. Nevertheless, I decided to start this thesis exactly with referring to the notion of Artificial Intelligence, and there are some reasons below.

During the whole period of existence of computers, the human being has been studying the main question of Artificial Intelligence: "Can a machine think?" In spite of the range of the problems considered to be the problems of AI is quite comprehensive, one of its most interesting and valuable sub-branches is the problem of search very well known from the strategic game playing. In *checkers, chess, othello, go* and other strategy games (sometimes called *games of perfect information* [30]) the search has to be done before a move is made. For those who are familiar at least with one of these games, it is obvious that there is no possibility to check every position that may become a possible continuation at some intermediate stage of a certain game, since the number of positions grows exponentially with every move made. For example, in chess, there are 400 for the first, 8902 for the second, 197281 different variants for the third full move. Therefore, it is understood that human uses some different method from complete search to find a good move (according to the rules of the game). Which exactly?

One of the possible answers is that it is a property of intelligence. Can a machine simulate the human's thinking process? This is still an open question, and the solution, if any, is vague. Nevertheless, nowadays there are computer programs that are highly competitive against the strongest human players. These world famous programs are *Deep Blue, Deep Junior, Deep*

Fritz, REBEL, Mephisto. Chapter 2 and the beginning of Chapter 3 of the present work discuss the basics of the methods, which make these programs play so strong.

1.3 Purpose of this research

The most important thing for a chess playing computer program is the way, in which the moves are evaluated. Since the computer is nothing but a device that is able to operate with numbers, then from the computer point of view every chess position is nothing but a combination of numbers. Therefore, ideally, there must exist a numerical equivalent for every positional component. The question of constructing such conformity is not trivial at all, and there are, in my opinion, two main reasons for it. First of all, every chess position is highly individual, and some observations that are valuable for one position can be completely senseless for another. Secondly, in general, there can not be any limits set for the number of parameters, which are to be taken into account when analysing the position, since this kind of things depend on the goals every different player sets for himself in a certain position. And finally, this numerical analogy, i.e. set of numbers, must be exactly such that every number, is the precise value of a positional component it stands for. Obviously, human uses more complicated technique for playing chess, but he does not deal with numbers. All said above makes the creation of a perfect computer chess player worlds apart from being completed, but only with all this the best numerical evaluation can be obtained.

And coming back to the topic: in other words, these numbers must be optimised. And *genetic algorithms* are known to be of a very successful use in the area of *optimisation problems*, i.e. the problems where the best of all possible solutions is desired, and they, with one or another peculiarity, have already been applied to the problem of computer chess [7, 45]. This thesis is one more effort in this area.

2 COMPUTER CHESS

2.1 History of computer chess

The time period of 1949 and 1950 is considered to be the birth of computer chess. In 1949, Claude Shannon, an American mathematician, wrote an article titled *"Programming a Computer for Playing Chess"* [40]. The article contained the basic principles of programming a computer for playing chess. It described two possible search strategies for a move with taking into account the impossibility of considering all the colossal number of variants. These strategies will be described in section 2.2 when directly talking about implementing chess as a computer program. No fundamentally different strategy has been invented ever since, which would appear to be successful enough in comparison of those two.

About a year later, in 1950, an English mathematician Alan Turing [46] (published in 1953) invented an algorithm aimed at teaching a machine to play chess. Unfortunately, at that time there was no machine, which could be programmed with that algorithm. Therefore Turing performed algorithm's work himself using pen and paper and played against one of his colleagues. The program lost, but the start was given for computer chess.

In the same year in USA, John von Neumann created a calculating machine huge in its size and very powerful to that time. The machine was built in order to hasten calculations on the production of atomic weapon but before the giant was used for its direct purposes, a test had been made and the chess-playing algorithm for a simplified variant of the game (6x6 board without bishops, no castling, no two-square move of a pawn, and some other restrictions) had been programmed into it. The machine played three games: it beat itself with white, lost to a strong player, and beat a young girl who had been taught how to play chess a week before [16].

In 1958, a great research in the area was made by a group of American scientists from Carnegie-Mellon University in Pittsburgh. Their algorithm, called *alpha-beta algorithm*, or *alpha-beta pruning*, the modern version of which is considered in details in section 2.2.4, allowed pruning away a considerable number of moves without having any penalties in the further process. Undoubtedly, it was a great discovery and made computers able to calculate

about 5 times more positions, but it still was not enough. The main problem again was how to reduce the number of required calculations.

The next interesting idea to improve computer's game level was proposed by another American scientist Ken Thompson. He reorganised the structure of an ordinary computer and built up a special device named *Belle* [8], whose only purpose was to play chess. That machine appeared to be much stronger than any existing computer and held the leading position among all chess playing computers for a long period in the 1980's, until there appeared *HiTech*, a chess computer developed by Hans Berliner from Carnegie-Mellon University, and *Cray X-MPs* [16].

Since then the progress in computer chess was mainly the result of permanently increasing computers' computing power. At the end of 1980's, an independent group of students made their own chess computer *Deep Thought* that appeared to be the prototype of the following *Deep Blue*, which later won the match against the human world chess champion Garry Kasparov in 1997 [27].

2.2 Making a chess playing computer program

The whole computer chess game involves a number of parameters for different purpose into the action process at every step. The chess board, the pieces on the board, moving rules, castling possibilities, en passant capture possibilities, and king in check. All these and some others must be considered and handled correctly when implementing a chess playing computer program. In this chapter I am going to give a more or less detailed description of every part that is required to construct a chess playing program.

Every chess engine consists of three components. First of all, a program has to have a computer equivalent of a *real chessboard*, so that it is able to understand what is going on in the game. Next, there must exist a method that decides if a position under consideration is worth playing or an attempt to find better one must be made. This method is referred to as *search technique,* and it is still a question of great discussion in the world of computer chess. And finally, a function, on which the search technique is based, and that tells, which of two positions is better than another, must be invented. This is called the *evaluation function,* and its description is given in a separate chapter (see Chapter 3).

2.2.1 Chess board

Chessboard representation is a significant part of every chess program, independently of which search technique is chosen and how complicated evaluation function is developed. All the processes, which occur in the game, operate with the chessboard, so that it is always worth spending time to think over chess board's the most convenient mode. Two reasonable approaches have been given in [15] to represent a chessboard in computer, the *mailbox* and the *bit boards* structures.

The mailbox representation was the original one sketched out by Shannon in [40]. In his time the power of the mightiest computers was totally insignificant to the one that a usual generally-purposed personal computer has nowadays, as well as there were no this plenty amount of computer memory to operate with. Therefore the programmers always aimed at reducing the capacity of memory required by the program, sometimes at the cost of increasing its calculation time. According to Shannon, chessboard itself consists of 64 integer numbers each from -6 to 6 (-6 for black king, -5 for black queen, 0 for empty square, 5 for white queen, and 6 for white king). One more number indicates the moving side, he used +1 and -1 for white and black, respectively. However, Shannon noted that the proposed method was not the most efficient one, but it was convenient for calculations. A move is described by specifying three parameters: index of the departure square, index of the destination square, and one more number considered in the case of pawn promotion move stores the value of a piece, to which a pawn promotes. The program then assigns the value of the departure square to the value of the destination square (or the value of the third parameter in the case of promotion) and then 0 to the departure square. This is obviously a convenient and efficient way of describing a move, and a similar (if not the same) idea is used in most computer implementations of the game of chess, or any other, which has the same playing manner.

Using these notations, there might arise some difficulties in detecting the edges of the board when determining legally possible moves for a certain piece. This circumstance must be handled somehow, and therefore an improvement of the Shannon's method was invented for this purpose.

Instead of using 8x8 board a 10x12 board is considered [15]. Squares that belong to chessboard are assigned addresses in the way shown in Fig. 3. The squares of all other

addresses serve as the auxiliary squares. They contain some big value to indicate that the square is off the playing board.

92	93	94	95	96	97	98	99
82	83	84	85	86	87	88	89
72	73	74	75	76	77	78	79
62	63	64	65	66	67	68	69
52	53	54	55	56	57	58	59
42	43	44	45	46	47	48	49
32	33	34	35	36	37	38	39
22	23	24	25	26	27	28	29

Fig. 3: Mailbox representation of the chessboard.

Thus if a square, to which a piece is thought to move, is occupied by this number, it means that the square in question is out of the board and it is impossible to move there.

After a time, another approach, called bit boards, was invented independently by two groups of scientists: one from the Institute of Theoretical and Experimental Physics[1] in Moscow, USSR, and one from Carnegie-Mellon University, Pittsburgh, USA, led by Hans Berliner. They suggested to represent each square of the chess board by a single bit, thus having one 64-bit computer word (and this is the one that is called the bit board) to represent any state of the board. The whole chess board consists then of 12 such words: for every chess piece, the corresponding word contains 1 on that bit, which number within the word corresponds to a square of the chess board that piece occupies. All other bits are set to 0. Two more bit boards that contain all white pieces and all black pieces present on the board, respectively, are usually used. The bit boards containing the squares, to which a certain piece on a certain square is allowed to move, can be easily constructed also. There is no reason to enumerate any further, since these are the developers, who decide what of the game they want to be represented with the help of bit boards. Of course, en passant and castling possibilities must also be kept in separate variables.

[1] Later they continued their work in the Institute of the Problems of Control, also in Moscow.

The presented approach is much faster and more convenient with respect to obtaining the necessary interesting information. For instance, it greatly simplifies move finding procedure: all the program has to do is to perform logical AND operation on the bit board of all possible moves of a piece on the square and the negation of "all-this-colour-pieces" bit board. For some other more complicated examples a reader is referred to [15], as well as for comparing the performance of the same operations using two discussed approaches.

2.2.2 Move search

Moves in a chess game are done in turn by two players until some final result – victory/loss, or draw – is reached. Therefore the game of chess can be represented as an enormously huge, but yet finite, tree with all possible chess positions as its nodes and final positions, from which no continuation is possible (checkmate or stalemate) or all continuations are meaningless, as its leaves.

Therefore it is possible to scan the tree and find the path leading to a victory from any given position. For example, for a standard game of *Tic-Tac-Toe* (using the 3x3 board), the overall number of final positions is less than 9! = 362880, it takes less than one second for a modern computer to find the best path. For the game of chess, meaning the reasonability of computations required, the problem is that, in general, a chess position allows about 30-35 moves as the replies to the move leading to this position. Thus, having about 60 (and this is quite low bound) moves per one game, i.e. 120 plies, we get $30^{120} \approx 1{,}8 \cdot 10^{177}$ leaves. This is more than the square of the assumed number of atoms in the Universe.

Of course, you may fairly mention that only a few moves are really valuable in any position. Moreover, all others lead to an immediate loss. For example, the sequence "queen takes pawn, pawn takes queen" makes sense only when the sacrifying side is going to mate or it will for sure take the opponent's queen *en prise* later. Otherwise the game will be lost. The number of valuable moves varies for different positions, but on average there are not more than 4 or 5 such moves. It does not solve the problem, since the number $5^{120} \approx 7{,}5 \cdot 10^{83}$ is still inadmissible, but this question, i.e. selecting only several moves for consideration and ignoring the rest, was described already by Shannon in [40]. It was named as *type-B strategy*, whereas the other way, in which no move is omitted, was called *type-A strategy*. But since human players, without doubt, use type-B strategy, then it is a matter of perception rather than of anything else, and, hence, it is, in general, inapplicable to a computer. And the history

only proved that fact: most computer chess programs eventually overlooked the losing move, which seemed to be unlikely to happen, according to the algorithms they had. Therefore the question of creating a plausible move generator that will never fail in its choice is far from being solved [15].

2.2.3 Full-search

Due to all said above, a given position could be searched for some reasonable number of plies using the so-called *minimax* technique, which is very popular among the most effective computer implementations of the games of perfect information. After every next ply has been made, the algorithm rescans the game tree to the same depth, but starting at a level lower (in other words, going one level deeper), and thus obtaining acceptable results each time.

The word "minimax" already expresses the idea of minimal maximum and maximal minimum. The maximal value for one side is the minimal, or least desirable, for the other, and at every step a corresponding selection must be made, depending on the side, which is on move. It means that two types of best values, the minimal and the maximal ones, are always considered. The best move is the one that leads to the position with the best score for the side to move. In order to get the clear understanding, let's consider an example of such a process of choosing minima and maxima.

Assume that we have an initial position with white to move, and the depth, to which the search is performed, is 4 plies (2 full moves). Assume also that the evaluation function returns white's score and it is symmetrical, i.e. white's score is equal to black's score, but with the opposite sign. This situation is shown in Fig. 4. Here, grey squares are the nodes, in which minimum is chosen, and white squares are the nodes, in which maximum is chosen. Positions are evaluated using the evaluation function I developed for my program. Arrows along the lines indicate the corresponding choices.

The last move is made by black. It means that end positions must be considered from black's point of view, and the best move is, hence, the one that provides the lowest score, or simply the minimum. At a higher level all the numbers, which were lifted up, must be again compared within one node, and the greatest one must be selected, since white's purpose is to achieve the position with the highest score. This procedure of alternation is repeated until the root of the search tree has been reached, where the maximal value is finally picked and the appropriate move is made.

Surely, the actual game path may likely differ from the one obtained for the current position, but this is because different part of the tree with new leaves will be examined at every new step and different results will be achieved.

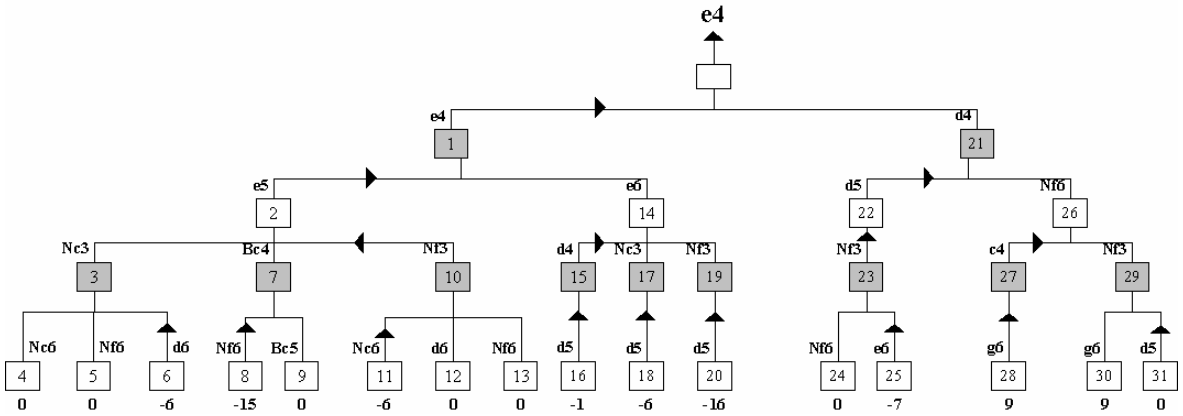

Fig. 4: Example of a game tree.

The minimax procedure can be implemented by a recursive function that searches the tree. Fig. 5 shows its sketch in a C-like pseudo-programming language [29].

```
int Minimax(position p)
{
     int m,i,t,d;
     Descendants(p,d); //Define all descendant positions p1,...,pd
     if d = 0
          return EvaluatePosition(p);
     else
     {
          m = - infinity;
          for i = 1 to d
          {
               t = -Minimax(pi);
               if t > m
                    m = t;
          }
     }
return m;
}
```

Fig. 5: Minimax algorithm.

2.2.4 Alpha-Beta (a-ß) pruning

Let's again consider the example in Fig. 4 and the depth-first method of observing the tree, which is usually used in modern computer chess programs [32]. Assume that we have already searched the tree to leaf 8 with position score -15. If you now look at node 3, its score is -6,

i.e. if white chooses the path to node 3, the worst score it may get is -6. And if it plays to node 7, it gets -15 already after the first reply considered during the search. It simply means that there is no reason to consider all other replies to node 7, which have not been considered by the moment, since white will in any case choose the path leading to node 3. In other words, we can simply skip considering leaf 9. Using this line of reasoning, we can skip also leaves 12 and 13. The same procedure, but only using the opposite reasoning, is applied at a level above. To be precise, there is no reason to consider nodes 17 and 19 (and, hence, all of their children), since black will always choose the path leading to node 2. And finally, all the sub-tree of node 26 can be skipped, since white will play to node 1, which gets the final score -6, rather than to node 21, which already has the score -7, and the search is not complete yet.

The method described above is called α-β algorithm, or α-β pruning, and according to [32], was first presented in [36]. Later, in [29], the topic was reviewed and supplemented with the proof of correctness and time complexity evaluation.

```
int  AB_Search(position p, int a, int b)
{
     int m,i,t,d;
     Descendants(p,d); //Define all descendant positions p1,...,pd

     if d = 0
          return EvaluatePosition(p);
     else
     {
          m = a;
          for i = 1 to d
          {
               t = -AB_Search(pi, -b, -m);
               if t > m
                    m = t;
               if m >= b
                    break;
          }
     }
     return m;
}
```

Fig. 6: α-β algorithm.

Every situation when there is no need to examine the rest of the sub-tree is called *cut-off*. Values α and β are, respectively, white's and black's best move scores found so far. The main advantage of this method is that the result is the same as if there were no cutting at all and the whole tree were examined. The sketch of this algorithm [29] is given in Fig. 6.

Table 1 shows the results of running the program I have created to play against itself using two above methods of move finding. Value in MOVE NUMBER column is the ordinal number of move, for which the search is made, since the beginning of the game (1 is the first move of white, 2 is the first move of black, 3 is the second move of white, and 16 is the 8th move of black, respectively). The numbers in other cells stand for the average time (in seconds) the program spent to find every next move in the test game.

Table 1: Full search and α-β search.

MOVE NUMBER	2 PLIES (sec)		3 PLIES (sec)		4 PLIES (sec)	
	FULL SEARCH	α-β SEARCH	FULL SEARCH	α-β SEARCH	FULL SEARCH	α-β SEARCH
1	0,61	0,38	7,8	3,7	177,0	45,5
2-6	1,31	0,95	15,0	7,35	348,0	80,2
7-11			22,2		490,0	
12-16			31,6		900,0	

In fact, time was different for each ply. For example, checks were recognised immediately and almost no time at all was spent in reply. But for the ordinary moves the difference was insignificant in all of α-β cases, and the separation between the moves (see the first column) was made in order to mark out the great distinction of the full search. In addition, the first ply was separated from the rest in each case, since its values were striking in comparison to all others.

Visual comparison of both search methods, in terms of the total number of positions evaluated, applied on the same game and using my own evaluation function is given in Fig. 7. To make the first move, the full search evaluated 8938 leaf positions, whereas the α-β search evaluated 4245, or 47% of the full search work. After the 8th move, the numbers were 136813 and 29577 (21%), respectively. And after the program had found the 16th move, the overall number of positions evaluated in the game was 377367 for the full search and only 63515 (17%) for the α-β search. Of course, the more the search depth is, the more positions are skipped by the α-β search (when searching to 4 plies, α-β considered only 0,07% of the positions the full search evaluated after 16 moves).

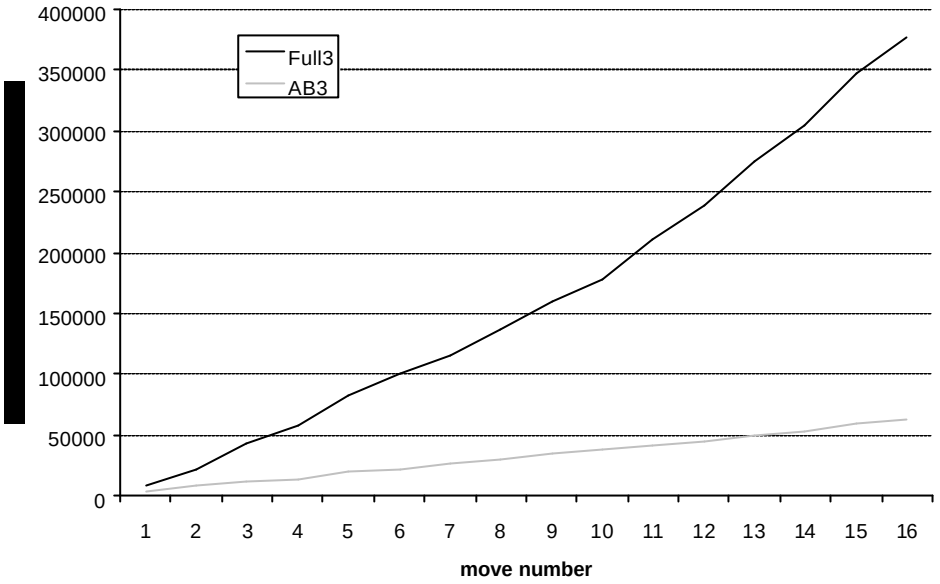

Fig. 7: Full search and α-β search to 3 plies for the same game.

The α-β algorithm can be applied to different search problems, not only computer chess. But each game has its own features, which might not be peculiar to any other at all. Due to this fact, the following discussion deals only with the game of chess. However, it does not mean that the refinements, which are going to be introduced, are not applicable to other games. On the contrary, it is very likely that being revised and modified appropriately they are. The only thing is that these peculiarities must be carefully determined and analysed in every case.

2.2.5 Transposition Tables

There are many ways to reach one position in chess. For instance, four sequences *1. e4 e5 2. Nf3 Nc6, 1. Nf3 Nc6 2. e4 e5, 1. e4 Nc6 2. Nf3 e5,* and *1. Nf3 e5 2. e4 Nc6* all result in the same position. *1. e4 e5* and *1. e4 e5 2. Nf3 Nc6 3. Ng1 Nb8* also both lead to the same position, but in different number of moves. Therefore it seems natural to prevent a program from considering the same position two or more times. This prevention is especially important when a position considered before appeared somewhere in the middle of the search. To make it clear, consider two positions: a) *1. e4 e5*, b) *1. e4 e5 2. Nf3 Nc6.* Consider also a 4-ply search technique. Assume now that position b) is now being searched and the sequence *3. Ng1 Nb8* has been just examined, so that there are still two plies to go. The appeared position is exactly the same as position a). But the latter one has been already searched earlier to 4 plies. Therefore spending time on further calculations for position b) will

get only 2 plies search score whereas using the results of the past search of position a) provides 4-ply search score, which is obviously better.

Moreover, every chess position is unique. Two positions with the same pieces on the same squares but with different en passant status or castling possibility are different. So, positions that have been already searched by the algorithm can be stored in some repository containing information like the side to move, the score it got, the depth at which this score was obtained. This repository is looked through first each time new position is to be examined. The repository is called *transposition table* and it is usually implemented as a *hash dictionary* [23, 30].

From here the question of assigning a unique value to a position arises. One of the solutions (the most famous and widely used one) was suggested by Zobrist in [50]. I have not managed to find neither the original article nor its later reprint, but the method Zobrist proposed became so popular that nowadays it is possible to find *Zobrist method's* description in a number of Internet resources devoted to computer chess [30, 32]. Zobrist suggested to assign one 32- or 64-bit random number to each piece on each square, or $12 \cdot 64 = 768$ numbers altogether. Empty square is assigned 0. A set of numbers is also generated for different castling possibilities and for en passant capture status. Then, starting with null hash key, XOR operation is performed between current hash key and a random number generated for the piece on the square in question if this piece is met on this square. The procedure is repeated for every square of the board. The value is then XORed with random numbers of castling and en passant possibilities. And finally, if black is to move, the number is XORed with a random number. The resulting number is a hash key for the current position.

Of course, the overall number of positions is much larger than the maximal 64-bit number. The probability of two positions to have the same hash key is small but greater than zero. Therefore, it is possible to repeat the whole procedure described above with different random numbers, thus, obtaining the second hash key for the position. The probability of two positions to have both hash keys equal is low enough to provide uniqueness of the positions within a single game.

The way of filling the transposition table in general is not confined by any rules. The reason is that this is the author of a chess engine who decides what is the most convenient dynamics of the content the tables are filled with. Of course, there are many more possible hash keys

than the maximal size of a transposition table could be because of the limits of computer memory available. Therefore, the hash keys must be somehow mapped onto the table indices. The method proposed in [30] and the one that I used in my program simply obtains the index as a residue of division of the current hash key by the size of the transposition table. Hence, in every game there will be a number of positions that will point to the same entry. And to handle this, there should exist a measure of the age of the position so that the engine knows if a certain position is old enough to be replaced with the new one. However, this question (to create an effective measure) is not simple. In my program a position was replaced with the new one each time they both point to the same entry and when all other parameters, which described the position in terms of a transposition table entry, allowed to do this.

The use of transposition tables allows a program to avoid considerable amount of unnecessary calculations. Obviously, the more entries the transposition table has, the higher the probability of a position to be found in it. The table becomes progressively filled, and it is more likely to find more positions in it with every move made, even using such a simple and ineffective method of renewing the table like mine. In my program I used the transposition table of $2^{20} = 1048576$ entries, and according to my experiments, it increased the speed of finding a move by one third, on average (see Table 2).

2.2.6 Move ordering and killer move heuristic

Coming back to α-β pruning, it is very important to have the moves ordered in such a way that there are as many cut-offs during the search as possible. Evidently, the best ordering is when the best moves are searched first. This problem is considered to be among the most important ones in the area of α-β search in computer chess.

Generally speaking, it is impossible to know in advance, which move proves to be the best one in every particular case, for otherwise there would be no need to search at all. Therefore, all we can do is to try to use the prior results combined with the information about the current situation on the chess board in order to make up a sequence of moves, which will be likely to stay in the best order. To start with, all capture moves are worth considering first. For replying with a simple pawn or piece moving to a move that took either knight, bishop, rook, or queen is unlikely a good choice. We do not talk about special cases, such as when reply makes a check with a “fork” to opponent’s queen or starts a mating attack. These situations are much rarer and are handled in some special way. Pawn promotion can also be considered

as a capture move, because it changes the material balance on the board. Later, all checks could be considered, and then the rest of moves. This approach, however, uses only information at hand and is obtained for every position independently of the game history.

Another refinement consists in storing for a while the details of the search performed so far. For instance, if a reply is so that the queen is taken, it does not really matter if some pawn was moved one or two squares (*1. … h5 2. Qxa5* or *1. … h6 2. Qxa5* in position in Fig. 8). Therefore the reply that took the queen during examining the first pawn move and made a cut-off (i.e. the position became absolutely hopeless for black), should be put to the top of possible replies during examining the second pawn move.

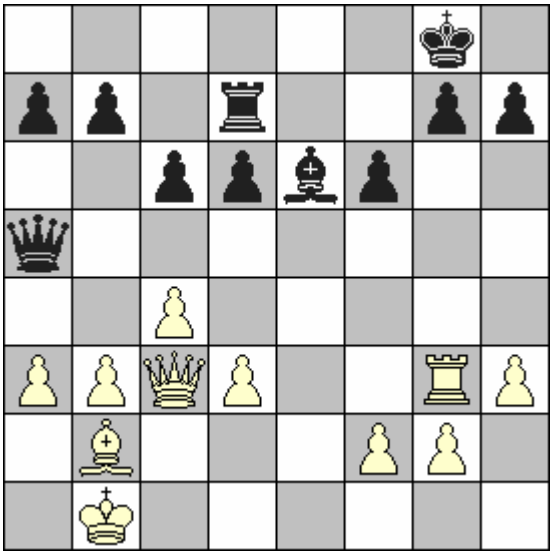

Fig. 8: Valuable and useless moves.

This idea was referred to as *killer heuristic*, and the moves that caused quick cut-offs were named *killer moves.* All this and some additional detailed information on the techniques used for move ordering can be found in [23, 30, 32].

The results of running my program (in seconds required to calculate every next move) for different combinations of transposition tables and move ordering are given in Table 2. Moves were ordered in two separate steps. 4 best and 4 worst moves of the previous search were always kept for the next search procedure. All new moves, generated during the next search, were compared to them and were placed onto the top of the final move list, if a match with the best move was found, or onto the bottom if a match with the worst move was found. The rest of the moves were first ordered in decreasing order, according to the safety of the destination square. The more friendly pieces and the less opponent's pieces attacked the

destination square, the more its safety was. Then the moves were reordered, according to their types. Capture and promotion moves were placed above other moves.

Table 2: Transposition tables and move ordering in α-β search to 4 plies.

	Transposition Table	Move Ordering	Move number	
			1	2-16
α-β search	-	-	45,5	80,2
	Yes	-	32,7	67,4
	Yes	Yes	23,6	29,2

So, using both transposition table and move ordering techniques improves the speed of play considerably. Moreover, the first move of the game is separated since its improvement is much less, and better improvement is achieved for the later moves. It seems to be correct since the program starts with an empty transposition table.

2.2.7 History Table

We have just discussed that looking one move back, we place some moves of the current position onto the top of the search list based on the scores they achieved a move ago. *History table* approach suggests to store information about all recently examined moves, not only the killer moves [23, 30]. The apparent advantage is that using history table it is possible to accumulate information about previous effectiveness of each move throughout the whole game tree, unlike it is for killer moves when they are considered only within a certain sub-tree.

Here, each time one or another move proved to be good (caused a quick cut-off or achieved high position score), its characteristic, which indicates how good this move is, is increased, and the greater this characteristic is later, the higher is move's privilege in the current move list. For example, the move that was placed among the best ones a move ago still has high probability to be such, even if a different piece can move so now. Thus, in Fig. 8, after the game continued *1. … Qxc3 2. Bxc3*, white's move *Bxf6* (instead of *Qxf6* a move ago) is still dangerous. Of course, all this makes sense only for a reasonable time period, so that the history table must be cleaned periodically in order not to mislead the computer with some old move.

2.2.8 Iterative Deepening

In any case, it is a factor of time that restricts the quality of playing both by human and machine. And if a skilled enough human chess player uses type-B strategy and focuses his attention on several but mostly acceptable moves, the machine, as it was already discussed above, cannot behave the same way all the time. And due to this the restriction in quality can sometimes be suicidal. It is especially important when time limits for one move are only of a few seconds. In this case a program can simply fail to consider every move it is supposed to by its algorithm.

Iterative deepening tries to solve this problem. According to this method, a program always starts with searching to 1 ply. After the search is complete, the program starts a new search, but to 2 plies already. In other words, it starts the search to N+1 plies only after it has finished the search to N plies. In case of no time left, it returns the best move of the last complete search. As stated in [23], the advantage of this method is that the number of nodes visited by all successive iterations taken together is generally much smaller than that of a single non-iterative search to the full depth in modern chess programs. It happens because of the possibility to re-order moves dynamically after every iteration has been completed, thus, obtaining better move ordering, so necessary for the effective use of α-β pruning algorithm.

2.2.9 Quiescence search

Let's now consider the following example. Assume that the search depth is 5 plies and the fifth ply is the move with which the side to move takes opponent's pawn with the rook. As it is the last move and no further game path is allowed to be considered, then the evaluation function must be called and the score must be returned. The result is that the moving side gets the advantage of a pawn, and therefore the move leading to this position will be estimated as a favourable one. But the real situation can be so that with his reply the opponent simply takes the rook, which, in fact, results in "a-rook-for-a-pawn" disadvantage. This kind of behaviour, when a program is not able to see the actual situation, was called *horizon effect* [6]. Of course, after the corresponding game path is chosen, the game will not be played so since during the next search the game tree is available at one level deeper and the retake is reachable. But what if the chosen move leads to the loss in any case? For example, instead of "a-rook-for-a-pawn" there will be "a-piece-for-a-pawn" disadvantage. This situation is for sure unfavourable.

A solution is that not every position during the game process is ready for the evaluation. Only the *relatively quiescent positions*, [40], where the least possible action takes place should be evaluated (such positions were also called *dead* in [46]). This is why all capture moves and pawn promotions are usually considered separately and searched to the end when the result of all material changes finally appears. In addition, moves are ordered in a special way depending on the taking piece and the piece to be taken in *most valuable victim/least valuable aggressor* manner. For more details a reader is referred to [23]. Check moves should also be taken special care of because check move always allows only a few forced replies and the actual situation is also vague. Here, the problem may lie in possibly too long check move sequence, which can explode the game tree. This situation can be resolved by limiting the number of extra plies given to inspect check moves. In [15] a value of 2 is said to be the mostly used one.

2.2.10 Null Move

Null move [4, 12, 20] means that the side to move skips its turn and allows the opponent to do two moves in a row. The idea is to see if the opponent is able to change the situation on the board playing twice. If the result of applying the null-move procedure is still acceptable for the skipping side, there is hardly any need to continue the full search because it most likely leads to a cut-off too [23].

The significance of this technique is in the fact that it takes away the whole ply of the current *N*-plies search tree and makes a machine search one sub-tree of a depth *N*-1. In the middle of the game, when the number of legal moves is about 30-35, using null move takes only 3% of the entire *N*-depth search effort. In case of success, i.e. null move search achieves acceptable results, the program saves 97% of the time required to make a move by pure searching, and if the results are not good, the program has additionally spent only 3% [30].

However, there exist zugzwang (German for "compulsion to move", "forced to move" [55]) situations (see Fig. 9), in which null move is the only way to avoid the loss. Applying null move procedure to such positions will certainly lead to a mistake.

In position A of Fig. 9 black do not have a move that will keep the present material balance. Namely, any of the moves *Ke8, Kf8, Kf7, Kf6, Rb7* allows white to win a pawn on *d6*. Every other possible move (*Bc7, Rc7,* or *Ra7*) loses even more material. Situation in position B is

not so evident at first glance and requires a bit deeper analysis. I analysed it myself and made sure that any black move would led to forthcoming mate or big material losses.

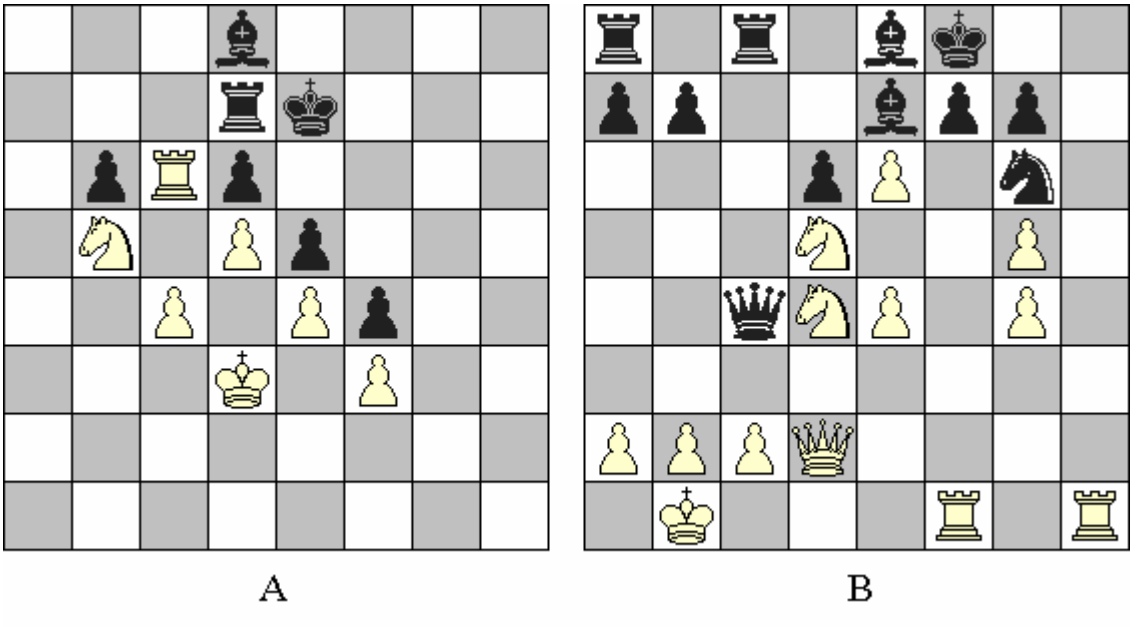

Fig. 9: Zugzwang. Any black move loses. Position A appeared in the game Kosteniuk – Paehtz, Lausanne, 2003 (taken from [41]). Position B was considered in the analysis of the game Vasiukov – Van Wely, Moscow, 2002 (taken from [47]).

Nevertheless there are two observations on this issue: on one hand, it is said in [30] that these positions are hopeless anyway; therefore the loss of performance is not very traumatic. On the other hand, according to [23], zugzwang happens extremely rarely in chess with the notable exception of late endgames. The latter circumstance is usually handled by stopping using null move procedure when a number of pieces left in a game is less than some pre-defined value. So the null move refinement is worth being implemented, especially if the aim is to increase search speed.

2.2.11 Opening and end game databases

Since it was realised that the best way to memorise the best move for every position is to write it down and save, people started to document every game officially played by highly skilled players, which was then carefully analysed. It has been done in order to always find the best reply for the position appeared in one or another game and to use it whenever the position is met again.

Nowadays, *chess theory*, as all this analysis together with the basic principles of playing is called, is very huge. There are hundreds of books, each of which is entirely devoted to the description of a single opening line. These books discuss in all details the main possibilities that can appear when starting the opening in question and bring the ideas and ways of further development of the game, which historically proved to show the best performance. I mean here that almost for every opening there are only several possible game paths that keep the best reply, and all others allow opponent to improve his position. Trying to say this in terms of computer chess, allowing the use of database gives a possibility to a computer to make the best move irrespectively of how high/low the score calculated by its evaluation function is. Moreover, by making a database move, program does not need to search at all!

The very same idea can be implemented for the endgames, which are also being analysed a lot, and in many cases even solved to the end. Every position in the endgame database is assigned a value of $+\infty$ (victory), $-\infty$ (loss), or 0 (draw) - the result the position ends with when assuming perfect playing of both sides. If during the search a match with the database entry is found for some position, this position becomes a leaf of the search tree, and it receives the corresponding value from the database without calling the evaluation function. According to [23], there are three different kinds of endgame databases available: *Thompson's collection* of 5-piece databases, *Edwards' tablebases* and *Nalimov's tablebases*, of which the last gained more popularity among recent chess programs due to their considerable advantages in indexing and size. Thompson's databases were the first in the area and had a number of disadvantages, especially those of very slow search at deep levels of the game tree. It finally made Edwards try a different approach, which became a big success but with the only disadvantage of their huge size. Nalimov's tablebases are actually the improvement of Edwards' originals with advanced index schemes. You can find much more information in [23], where there is an entire chapter devoted to endgames databases used by the famous chess program *Dark Thought*.

3 EVALUATION FUNCTION

As we said earlier, there is no possibility to look through the whole chain of subsequent moves at some intermediate stage of a single chess game. Therefore the problem of the most accurate position fitness estimation arises.

In general, evaluation function is a multivariable function that measures a goodness of a chess position. Every input parameter of this function stands for some factor that characterises the position. We have already discussed in the beginning the difficulty in constructing a numerical analogy for the game of chess, but at the moment there is no other way to make computer play chess. And the more precise this measure is, the better.

A chess position is a certain set of black and white pieces located on the chessboard. Each piece is of different importance and is assigned different value. The difference between the pieces of one colour and the pieces of the other colour, or *material balance*, is the major factor in every position evaluation, and it always must be considered above all. Two special cases are also used always: position score is $+\infty$ if opponent's king is checkmated, and $-\infty$ if own king is checkmated. Trivial situations, e.g. "king against king", "king against king and knight", "king against king and bishop", are known to be drawn, and they can be easily included with position score 0 into the evaluation function directly. Using database it is possible to assign 0 to more complicated positions that have been analysed before by chess experts.

However, in general, it is not possible to claim the equality of two positions taking into account only the equality of pieces of both sides. In several opening lines one side is ready to sacrifice a pawn in purpose, e.g. *king's gambit accepted (1. e4 e5 2. f4 exf4), queen's gambit accepted (1. d4 d5 2. c4 dxc4).* Here the program that uses opening database must still somehow consider itself in an advantageous position, even having one pawn less. Sometimes player can sacrifice *quality* (i.e. exchanging rook for a bishop or knight) and, hence, achieve some non-material advantage he considers to be worth the material. Moreover, highly skilled chess players often agree to call the game a draw even when there is inequality of pieces. So in all these situations other factors, apart from the material balance, should be involved into position evaluation. We will call these factors *strategic* or *positional*.

Ideally, evaluation function is a sum of all factors that influence to the result of the game. We need to analyse the board to see, which pieces and other factors are present, and sum up their values. Mathematically, evaluation function can be expressed by the formulae:

$$F = \sum_{i=1}^{N} x_i \cdot v_i \ , \ \begin{cases} x_i \in \{0,1\} \\ v_i \in \Re \end{cases} \ , \ i = \overline{1, N} \ ,$$

where x_i is an indicator of the presence of the i-th parameter, v_i is the parameter's importance weight, and N is the overall number of the parameters involved into the evaluation. In the following sections we will discuss which parameters may be important.

3.1 Summary of parameter selection

Since the top-priority purpose of this work is not to create a high-level chess-playing program, but to make an attempt on improving the values of the parameters for position scoring, I tried to select the most valuable set of parameters based on my own understanding of the game of chess. It was done by the detailed study of different parameters involved into the chess programs I managed to find out [3, 22, 28]. The chosen parameters are shown in Table 3. The assigned values were picked also with taking into account my own opinion about the importance of the particular parameter. In the next two sections there is a description of every selected parameter.

Table 3: Selected parameters of the evaluation function and their values.

#	PARAMETER	RANGE	RECOMMENDED VALUE
0	queen	[800-1000]	900
1	rook	[440-540]	500
2	bishop	[300-370]	340
3	knight	[290-360]	330
4	pawn	[85-115]	100
5	bishop pair (+)	[0-40]	not given
6	castling done (+)	[0-40]	not given
7	castling missed (-)	[0-50]	not given
8	rook on an open file (+)	[0-30]	not given
9	rook on a semi-open file (+)	[0-30]	not given
10	connected rooks (+)	[0-20]	not given
11	rook(s) on the 7^{th} line (+)	[0-30]	not given
12	(supported) knight outpost (+)	[0-40]	not given
13	(supported) bishop outpost (+)	[0-30]	not given
14	knights' mobility >5 (>6) (+)	[0-30]	not given
15	adjacent pawn (+)	[0-5]	not given
16	passed pawn (+)	[0-40]	not given
17	rook-supported passed pawn (+)	[0-40]	not given
18	centre (d4,d5,e4,e5) pawn (+)	[0-30]	not given
19	doubled pawn (-)	[0-30]	not given
20	backward (unsupported) pawn (-)	[0-30]	not given
21	blocked d2,d3,e2,e3 pawn (-)	[0-15]	not given
22	isolated pawn (-)	[0-10]	not given
23	bishop on the 1^{st} line (-)	[0-20]	not given
24	knight on the 1^{st} line (-)	[0-30]	not given
25	far pawn (+)	[0-30]	not given

3.2 Material count

Pawn: Pawn is the unit of the measure of material count. It can be set to 1 (pawn), or scaled to 100 as it is done here.

Knight and bishop: Originally, in the pioneer work on computer chess [40], both bishop and knight were given the value of 3 pawns. The recommended values are a bit higher and a little difference between them proved to be the result of longstanding (and still being continued)

experiments in the field of computer chess. Here we set the value of bishop to 340 and knight to 330.

Rook and queen: Usually rook is considered to have value of 5 pawns and queen of 9 pawns. However, in some applications [3] you can meet higher values, up to 650 for a rook and 1200 for a queen. Here we set the value of rook to 500 and queen to 900.

King: It is the main piece in the game and simply cannot be captured or exchanged. Officially, in blitz and fast games there is a possibility to capture the king when the opponent misses a check and makes an illegal move. Of course, the game is immediately lost with the loss of the king. For example, in the blitz game Shirov-Tkachiev, Bastia, 2003, [53], black, having clearly winning position, black simply ignored a check, which allowed Shirov to win and move further in the knock-out tournament. But we do not consider such a possibility in the present research, since this rule was invented for humans in order not to waste opponent's time when playing with real pieces and real chess boards where illegal moves can be made. Besides, computer simply can not make an illegal move, hence, there is no reason of assigning any value to king.

3.3 Positional factors

The selected important positional factors will be next explained and illustrated in Figures 10, 11 and 12.

Castling done and castling missed: Castling is missed, when there is no possibility to castle any more. It is very important for a king to be surely defended by friendly pieces in the opening and most time in the middle of the game. Pawns in the corner serve as excellent defenders, of course, supported by other pieces, if necessary. Castling is therefore considered as a positional benefit.

Rook on an open file: Rook is the second powerful piece on the board after the queen. It can move far, so it always aims at having a lot of space to attack in order to catch opponent's weaknesses and rush into enemy's camp through this open file as soon as a player decides.

Rook on a semi-open file: File is called semi-open one when there is no friendly pawn but there is an enemy pawn on it. Placed onto a semi-open file, rook does not allow opponent to leave his pawn unprotected, thus reducing mobility of his pieces.

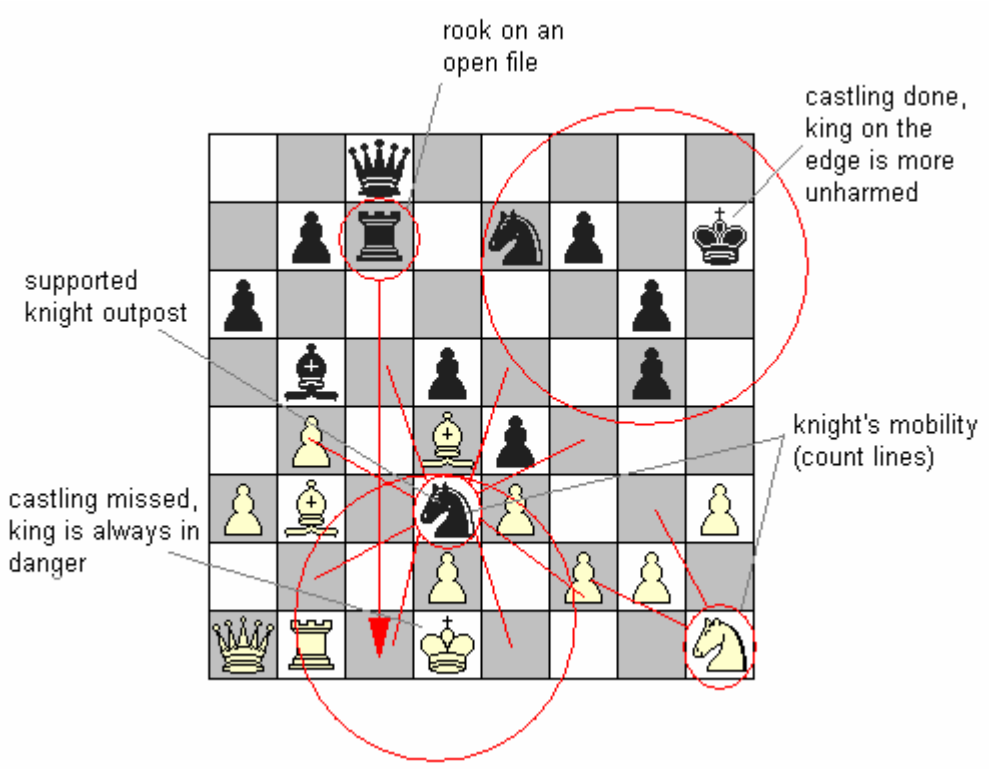

Fig. 10: Positional factors.

Knight's mobility: Knight has at most 8 moves. The more moves it has at the moment, the better.

(Supported) knight/bishop outpost: Both can highly reduce enemy's mobility, especially if achieved in the openings.

Bishop pair: With this only difference it is usually considered for a side (whichever has) to be in a slightly advantageous position. Two bishops might become very powerful when aiming at enemy's king position.

Centre pawn (d4,d5,e4,e5): Usually, in the opening one of the most important aims of both sides is to control central squares, as they are the most valuable ones for the mobility, and these are the central pawns that can do it best.

Doubled pawn: Good pawn structure is very important. Doubled pawns are usually considered as a weakness: the upper one blocks the lower and in many cases both are in need to be defended by a piece.

Blocked d2,d3,e2,e3 pawn: It complicates co-operation of the pieces within the camp, and can therefore be blocked by opponent's outposts.

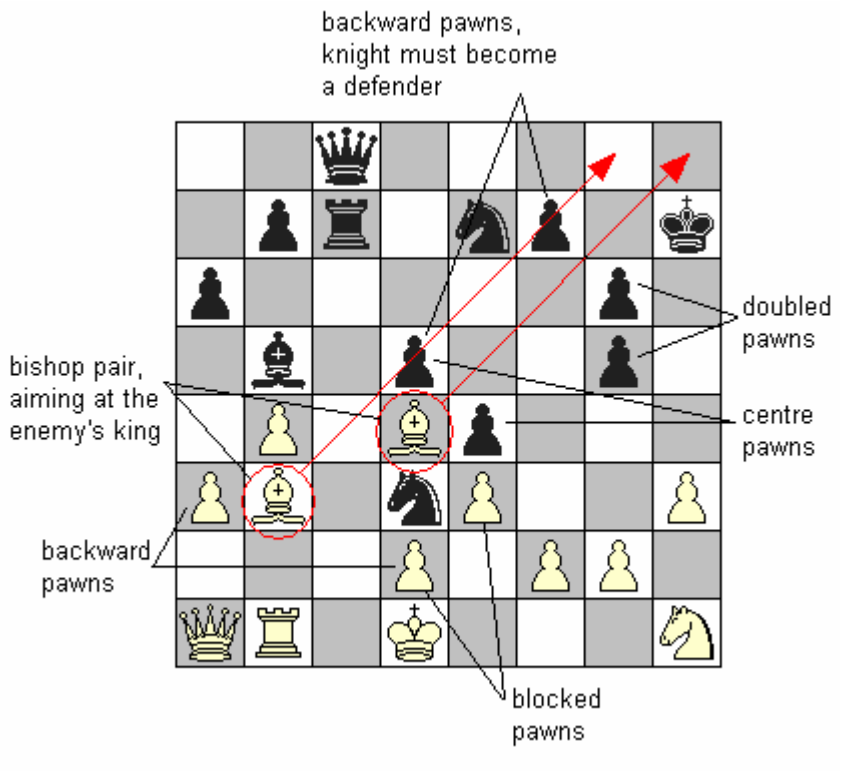

Fig. 11: Positional factors.

Backward pawn: Backward pawn does not have a pawn on the neighbouring verticals, which is nearer to the first rank, and it cannot move forward due to one or more enemy pawns on the adjacent files.

Rook(s) on the 7th line: Rook(s) on the 7th line usually considerably impede co-operation of enemy's pieces during the late stages of the game.

Connected rooks: When two rooks are on the same file or line without pawns or pieces between them. Connected rooks are very strong, as they support each other.

Passed pawn: A pawn is passed when it is not hindered by a pawn and can never be captured by a pawn. This pawn is usually aimed at to be promoted, so that it is of special importance in end games.

Rook-supported passed pawn: It is a kind of a rule: if your pawn is on the way to promotion, try to support it with a rook (if rooks exist on a board, of course). The opponent has then to block this pawn by placing a piece in front of it whereas your pieces are still free to choose their location.

Adjacent pawn and isolated pawn: Isolated pawn has no friendly pawns on the neighbouring files. Adjacent pawn is the opposite. It is always better when a pawn is

defended by another pawn; otherwise it requires more powerful pieces to take care of it, thus reducing their own mobility.

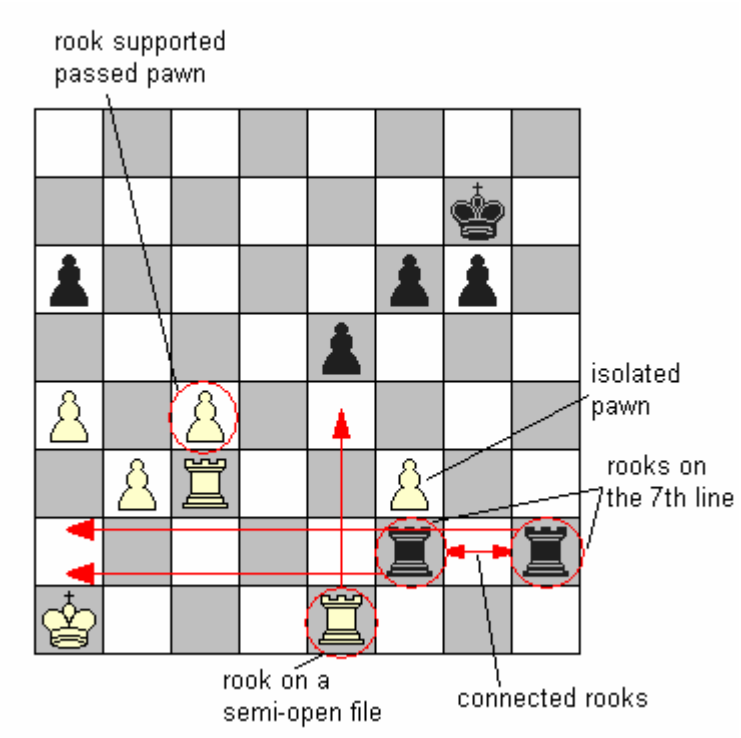

Fig. 12: Positional factors.

Bishop on the 1st line. Knight on the 1st line: Mobility of minor pieces is very important during the entire game process. And at the same time they should not disturb the connection of heavy pieces.

Far pawn: If a friendly pawn is close to its promotion square, it becomes very dangerous. Opponent must permanently make some of his pieces try to hinder further moving of such a pawn. But the bonus is given only when the pawn is not in danger of being taken with the next opponent's reply.

There are also some other parameters I paid my attention to during the selection. However, they require much deeper position analysis, and were eventually rejected for some reasons, one of which is that before including these parameters, I should improve my own chess skills. But these parameters are discussed briefly here.

Pawn structure analysis: This is the minimax search (see section 2.2.3) performed for pawns and kings only with all other pieces removed from the chess board. The number of moves is not big, and in principle there is a possibility to perform this search to the end. It can

be valuable only since ending stage happens. The main meaning is to define a position, to which to conduct the game path.

Position closeness (openness): In [22] the position is defined as a *closed position* if more than 6 pawns occupy the 16 centre squares of the board. In [24] there is an exemplary analysis of such closeness as being an advantage of one of the sides. For the illustration the situation in Fig. 13 is considered.

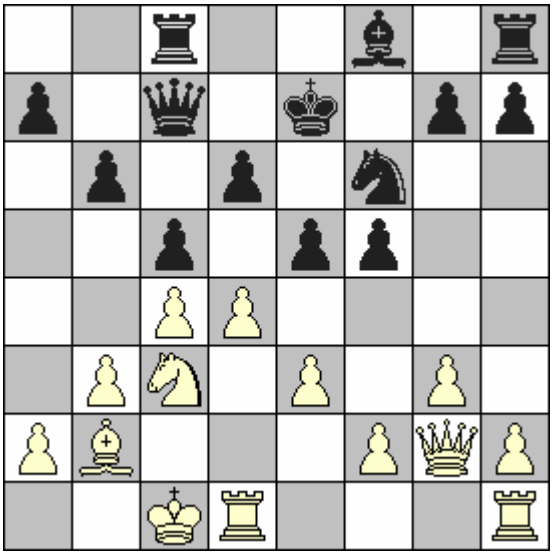

Fig. 13: White to move. Close (*1. d5*) or open (*1. dxe5*) the position?

In this position with white to move the material balance is equal, but positional one is not at all. White stays much better: all its army is active, king is safe, and rooks are connected, whereas black has none of these. Therefore, white should try to open the position in order to reach the black king faster, which, besides, hinders the communication of his own army.

Black, on the other hand, should aim at closing the position, to provide time for king to escape. So *1. d5* would be an enormous mistake, because by playing so, white's biggest advantages would be negated. Thus the qualitative analysis of the closeness is not as trivial as just 6 pawns in the centre. It depends on many more other things.

Pieces' activity: This is also worth to consider, since in many chess programs every potentially possible move adds some very small positive value to the overall evaluation value of the current position. In my opinion, if this parameter were included, it would also require detailed analysis. Probably, something like dividing the pieces into different categories with differently directed moves available, each having its own value for being performed, and so

on. This demands excellent positional understanding. An interesting approach to consider pieces' activity was proposed in [31]. The author introduces the notion of a *chess chunk* as a special distance measure, according to which some pieces are relative. The discussion is based on the human style of interpreting a chess position, so that understanding this style may let a machine make predictions about human players' performance.

4 GENETIC ALGORITHMS

4.1 Overview

Genetic Algorithms (GAs) aim mainly at solving various optimisation problems by means of applying the principle of natural selection to the creation of the solution. Moreover, during GA's work an internal function is applied to determine how good a particular solution is. If you nowadays made an effort and tried to search for and name the areas in which GAs proved to be a good solution tool, you would receive a large number of very different problems, from *scheduling* [13], *query optimising* [5] and *optimal control problems* [34] to *neural network training* [2], *network design* [37] and *game playing* [43, 49].

The advent of the concept of GA is the result of generalisation and imitation in artificial systems of such wildlife properties as natural selection, adaptability to environment's changing conditions, inheritance of the vital properties by descendants; thus following the rule "survival of the fittest" of Darwin's evolution theory [10]. In scientific literature, the idea of GA was first proposed by John Holland in 1975 [25]. In his work, Holland suggested a scheme, or a sketch, how genetic algorithm should look like. In 1989, David Goldberg described in details the *Simple Genetic Algorithm* [21], the first famous computer implementation (using programming language *Pascal*) of a genetic algorithm.

At present, the appearance and the complication of a wherever implemented genetic algorithm is much more difficult and time taking than these of the very first ones. But the basic structure still remains the same. Let's consider it in more detailed in the following.

4.2 Terminology

In nature, in biological systems, from where the concept of genetic algorithm has come to computer science, the main concept of genetics is the *chromosome.* A set of chromosomes of every living organism is called the *genotype,* and the organism itself is called the *phenotype.* The parts constituting a chromosome are referred to as *genes,* which are located on different *loci*. Each gene controls the inheritance of one or several *alleles*.

In artificial genetic systems a different terminology is accepted [21]. The comparison of the corresponding terms and also the terms, which seem to be the most convenient for the current thesis, are given in Table 4.

Table 4: Comparison of genetic terminology.

Natural genetics	Genetic algorithm	Current thesis
phenotype	parameter set, alternative solution, decoded structure	set of parameters
genotype	structure, population	population
chromosome	string	individual, representative, player
gene	feature, character, detector	parameter
locus	string position	position
allele	feature value	parameter's value

4.3 Population

So, the basic notion of the GA considered in computer science is an entity called *string*. A string in turn consists of *features* or *characters*, each of which controls the inheritance of one or several *values*. Features are located at certain places of the string, which are called *positions* [21].

Feature can be whatever makes sense for a problem to be solved. For example, in [33] *the function maximisation problem* was considered for a function of two variables and binary values were used to code the variables. In [7] author considered *the chess-playing computer program problem* and decided a feature, indicating the importance of a parameter involved into the solution, to be represented by a fixed-length set of binary values. In [9, 26] an integer number stood for a feature of *the N-Queens problem's* string, and in [13] a string for *the job shop scheduling problem* was constructed having some special structure as a feature. In each of these articles a different problem was dealt with, but all used the concept of GA as a solution tool.

When applying GA, the string is the only information storage for the problem, i.e. it entirely describes a potential solution. GA operates with *population*, a set of actually different strings each representing one solution. Numerous strings are considered and evaluated as the algorithm goes on (*the evolution process* happens), always obtaining new population from the

previous one. It happens by means of applying *genetic operators* with the following evaluation, thus providing vast information exchange and modifications of the characteristics. This is why it is worth much fixing carefully and thinking over the most valuable attributes of the problem before creating a string. And this is why the key aspect of the algorithm is the representation of the solution, i.e. inventing the problem-specific internal structure of the string that does find the best solutions.

4.4 Genetic operators

According to GA's principles, every individual in the population undergoes some modifications as the algorithm goes on, producing the *offspring*. There are two classical genetic operators, *crossover* and *mutation*, which perform these modifications. Usually, crossover takes two strings from the current population, selects a position, sets the crossover point, and makes an interchange of parts, separated by the chosen position, between two strings, thus forming two new ones. Single mutation usually changes the feature value of one position in one string. The way these operators do the selection usually utilizes problem specification and, hence, is of a special discussion.

To understand the idea better, let's consider its visual representation. Assume x_1 and x_2 are two strings of length 8 selected arbitrary from the population of candidate solutions for some problem and every feature occupies one bit in a string. Assume also that the third bit was selected as the crossover point. Then the operator performs as shown in Fig. 14.

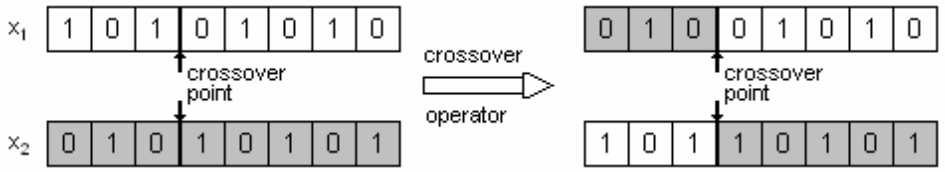

Fig. 14: Example of a simple crossover.

Next, assume that one of two new strings, y_1, is selected for the mutation, and the first bit is to be switched on this step of the algorithm. This mutation is shown in Fig. 15.

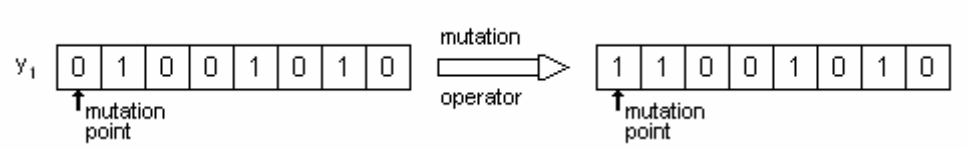

Fig. 15: Example of a simple mutation.

There exist more complicated extensions of genetic operators: *two-point (multi-point) crossover* and *two-point (multi-point) mutation*. In two-point crossover, a string is divided into three parts, and the middle ones are exchanged. In two-point mutation, values of two positions are changed. A similar procedure can be defined for multi-point operators [33].

In spite of the fact that usually only two genetic operators are used, *inversion* is sometimes called the third genetic operator. Simple inversion selects two points along the length of the string, which is cut at these points, and the sub-string between these points is reversed [33]. It is still an issue of great discussion if this operator is necessary, since evolution can happen without it [11]. However, inversion has been applied successfully in many differently purposed applications, and a reader is referred to [1, 37, 44] for the examples.

4.5 Evaluation and selection

Before the crossover and the mutation can be applied, all the strings are evaluated and the *selection* is conducted to make up a set for undergoing the genetic operators. This operation can be very different. There is a possibility to give preference to better solutions, or to make the worst individual be denied from the selection process, or use some randomised heuristics (*roulette wheel* selection), in which better solutions have higher selection probability. If the best individual is always preserved for the next generation, then it is called *elitist model*. There are also *static* and *dynamic* selection methods, with the constant and the varying selection probabilities of the representatives over the generations, respectively [33]. Finally, one is able to develop an own selection method private to the problem. But the most important part of each step of GA is its evaluation process. Every solution is assigned a fitness value, which characterises how good the solution is. Different solutions are achieved from the present ones, and it cannot be known in advance if these new candidates are better or worse than their parents. And this is the *fitness function* that decides it. Obviously, fitness function calculates fitness value of a string depending on the data this string contains. That is why it is very important for a string to carry the most valuable characteristics of the current problem in order to converge to a better solution, indeed.

5 PROPOSED GENETIC SYSTEM

After all the parameters were selected, described and assigned the ranges to vary, the time has come to start dealing with GAs, and the first task is to develop an optimal representation of the future solutions. Here, we will start using, as appropriate, the terminology of the current thesis given earlier in Table 4.

5.1 Solution representation

The main problem when thinking over the issue is should an individual have integer, binary, or any other possible representation. As we have mentioned before, this very important choice depends highly on the problem specification.

The simplest and the straightforward way is to operate with a usual binary string representing every parameter from Table 3 as a sub-string at a certain position. For example, a possible parameter containing a bonus value for a pawn in the centre might look like the one shown in Fig. 16.

s	3	2	1	0
1	0	1	0	1

Fig. 16: Binary representation of "centre pawn" value.

Here, the first bit is a sign bit and the rest is a value between 0 and 15, which, when added to 15, gives the corresponding bonus value for an individual. Every other parameter can be coded in similar manner with some additions to several, thus constituting a binary string of the approximate length of 150 bits. This representation is close to the one used in [7]. The genetic operators are then the ones illustrated in Fig. 14 and Fig. 15, with a note that multi-point crossover might perform better for such a long string.

The idea of the second approach is to represent each parameter directly as an integer number within the corresponding range, according to Table 3. For instance, queen gets a value between 800 and 1000, rook between 440 and 540 and so on. In this case an individual would have a set of integers and would be much shorter than the one of binary structure. Surely, it seems more illustrative with respect to the problem because the parameters have the property of atomicity, but then the genetic operators are not clear and should be weighed somehow.

Applying similar crossover would always change a subset of parameters whereas in binary case a parameter could get a part from one parent and the rest from another if at least one crossover point was in the middle of the corresponding gene, which might provide better solution diversity than changing the whole parameter. As for mutation then it is not understood well how this operator should change the value of the parameter. For example, it could generate some random number within the range, or it could be a function depending on the current value, the range and the importance of the parameter, or any other specific technique.

All said above results in the dilemma of choice: should an individual have binary and long structure, but with the genetic operators already developed, or should it be short and contain integer numbers, but with vague genetic operators, which are to be developed? Or in brief: which approach should show better results?

Therefore a number of articles on the relative topics have been studied for the purpose of examining the representations used and the results achieved, which were analysed and compared to the present problem before making the final conclusion.

In many researches on various *parameter optimisation problems* much attention has been paid to the solution process and the results, but the way the problem origins are dealt with has been omitted so that the work acts well with the representation involved. For instance, in [45] there is a big discussion about the time required to perform the evolution process using binary chromosomes. The chromosomes are simply involved, but no other possibility is considered. In [28] an integer representation is chosen, but also without reasons, just as a fact. In [7] the author selects a binary representation due to its prevalence throughout scientific literature and good understanding of the corresponding genetic operators. However, some other possibilities are mentioned in this work, but they stand apart of binary. The punch line of the examples I have given does not purpose to cavil or make any critics, but on the contrary, to note that in each of these papers good and valuable results were achieved in spite of the representation; so that it still stood unclear which of two was better.

Finally, a comparison of two approaches, binary and floating point, applied on the same problem, was found in [33]. The following *dynamic control problem* was considered:

$$\min\left(x_N^2+\sum_{k=0}^{N-1}\left(x_k^2+u_k^2\right)\right),\ x_{k+1}=x_k+u_k\,,\ k=\overline{0,N-1}.$$

Here, x_0 is a given initial state, $x_k \in R$ is a state, and $u \in R^N$ is the sought control vector.

Its optimal value can be expressed as

$$J^* = K_0 x_0^2,$$

where K_k is the solution of the Riccati equation:

$$K_k = 1 + K_{k+1} / (1 + K_{k+1}) \text{ and } K_N = 1.$$

A string represented a vector of the control states u, x_0 = 100 and N = 45. In his work, the author describes in details how the differences are handled, what is done for each approach at every phase and gives the reasons why the floating-point representation achieves better results. The final conclusion was that the floating-point representation appeared to be faster, more consistent from run to run, and more precise. Applying some special operators improved its performance in terms of achieved accuracy. Also it was stated that the floating-point representation was easier for designing the problem specific operators.

This experiment appeared to be even more useful, since the parameter optimisation problem was considered, to which the problem of the present research belongs.

5.2 Individual

So it was agreed to use integer representation of an individual, or a chess player. An individual has 26 parameters, according to Table 3. Each parameter keeps one integer number that corresponds to the value this parameter takes on. The parameters must be located in some order. And an interesting problem arises from this: is the order of parameters important?

There are two basic possibilities that come to mind: when parameters appear in no special order, for example, as listed in Table 3, and when some combinations of parameters are placed together.

The idea is that if the answer to the question above is positive and the parameters, which are truly correlated, appeared to stay one after another, then such an individual may show better performance than that of the separately placed parameters (of course, assuming that there are no other correlated groups). For example, passed pawn and adjacent pawn (the same pawn)

when placed one by one might produce the maximal bonus value greater than 40+5=45; or connected rooks together with rooks on the 7^{th} line might achieve more than 20+30+30=80. And if the answer is negative, the result does not depend on the order and, hence, must be exactly the same.

5.3 Operators

Genetic operators are not necessarily both present in every GA. For example, in [28] there is no crossover, in [17] there is no mutation, and in [9] the same operation is used for both. This circumstance greatly depends on the solution technique the problem is managed by. For the present research, it was decided to involve both crossover and mutation, and since an integer representation was used, the specific operators had to be defined for both.

We discussed already in section 4.4 basic principles of usual genetic operators, and in this chapter we will only discuss the problems, which were forced during the development stage. The operators selected for the use are presented at the end of every discussion.

5.3.1 Crossover

The development of the operator consisted of two parts. First, to fix how many crossover points it should have, and second, to create a rule, according to which the new individuals will be obtained.

Since the number of parameters is small, one-point crossover and two-point crossover are of the most interest, of which the second seems to be more preferable. Of course, it cannot be known in advance, which variant appears to be better, and a similar but simpler reason for such a favour was already given in section 5.2. Namely, it can happen that some parameters will not be in harmony with the others, thus making individual show worse results. And when replaced with some other set, they will, hopefully, improve. This idea can seem close to the one of correlated parameters, but, actually, it slightly differs, since it is quite difficult, or ever feasible, to discover a correlation between some 5 parameters. However, it can be considered and studied as a generalisation.

So two-point crossover was agreed. The crossover points will be chosen randomly for every new generation, perhaps, providing more population diversity.

Next, there is a lot of possibilities for obtaining new values of the parameters. The very well known one was already explained earlier: it is a simple exchange (see Fig. 14). A middle part is interchanged in two-point crossover. According to the author's opinion, the drawback of this simple exchange under quite small population size is that there is higher probability for two offspring to be selected as a pair for the crossover one iteration after. It will result in producing their parents and will be useless in terms of information exchange. Another possibility is to apply some mathematical formulae, which can make sense. For example, the resulting value can be one of the following:

$$\frac{x_1^c + x_2^c}{2} \quad \text{(a)}; \qquad 2\left\lfloor \frac{x_1^c \cdot x_2^c}{x_1^c + x_2^c} \right\rfloor \quad \text{(b)} \tag{1}$$

These expressions take two numbers and produce one. Since two generally different values must appear for a parameter after the crossover (so as to produce two different offspring), the idea of *arithmetic crossover* [33] was involved. This type of crossover is expressed by the formulas:

$$\begin{aligned} y_1 &= a \cdot x_1 + (1-a) \cdot x_2 \\ y_2 &= (1-a) \cdot x_1 + a \cdot x_2 \end{aligned} \tag{2}$$

where x_1 and x_2 are the parameters on the same position of the first and the second individual, respectively, y_1 and y_2 are the resulting values for offspring, and $a \in [0..1]$ is some coefficient. Using (2), it is easy to generalise (1):

$$\begin{cases} y_1 = a \cdot x_1 + (1-a) \cdot x_2 \\ y_2 = (1-a) \cdot x_1 + a \cdot x_2 \end{cases}$$

$$\begin{cases} y_1 = \left\lfloor \dfrac{x_1 \cdot x_2}{a \cdot x_1 + (1-a) \cdot x_2} \right\rfloor \\ y_2 = \left\lfloor \dfrac{x_1 \cdot x_2}{(1-a) \cdot x_1 + a \cdot x_2} \right\rfloor \end{cases}$$

In general, coefficient a can be globally fixed before the algorithm starts, randomly generated for every next step of the algorithm, or can be new for every pair. Like that of selecting the

crossover type, it is impossible to know which combination performs best. Therefore it was decided to obtain new chromosomes as shown in Fig. 17.

y_1 :	$a \cdot x_1 + (1-a) \cdot x_2$	$(1-a) \cdot x_1 + a \cdot x_2$	$a \cdot x_1 + (1-a) \cdot x_2$
y_2 :	$(1-a) \cdot x_1 + a \cdot x_2$	$a \cdot x_1 + (1-a) \cdot x_2$	$(1-a) \cdot x_1 + a \cdot x_2$

Fig. 17: Selected crossover.

Since a is a floating-point number, the resulting values must be rounded to an integer. The result is rounded up (to the nearest integer) if its fractional part is greater than or equal to 0.5, and is rounded down (to the nearest integer) otherwise.

To illustrate how this crossover works, let's consider two individuals shown in Table 5.

Coefficient a = 0.783197 and crossover points are (after) 8 and (before) 13. They are marked out by the double lines. Values of the parameters of the first offspring are calculated using the upper rule of Fig. 17, and values of the parameters of the second offspring are calculated according to the lower rule.

Let's perform calculations for the value of the first parameter (the queen) for both. For the first offspring we have 0.783197 · 861 + (1 - 0.783197) · 888 = 866.853681, and this number is rounded up (0.853681 > 0.5) so that p_1 = 867. For the second offspring we have (1-0.783197) · 861 + 0.783197· 888 = 882.146319, and this number is rounded down (0.146319 < 0.5) so that p_1 = 882.

The selected crossover has the *averaging effect*, i.e. it produces the values from the range between the minimum and the maximum of the corresponding values of parents.

Table 5: Example of individuals (two top rows) and their offspring (two bottom rows).

861	468	310	292	112	4	34	38	1	28	15	23	27	10	17	1	31	24	26	16	15	14	8	0	26	6
888	527	366	299	92	27	2	3	7	29	5	22	27	18	12	3	24	31	8	17	5	11	13	2	10	8
867	481	322	294	108	9	27	30	6	29	7	22	27	12	16	1	29	26	22	16	13	13	9	0	23	6
882	514	354	297	96	22	9	11	2	28	13	23	27	16	13	3	26	29	12	17	7	12	12	2	13	8

The *probability of crossover* p_c is one more component of the crossover operator. It decides if an individual undergoes crossover or not. There are no universal instructions for setting the probability, because this issue highly depends on the problem. The value of p_c for the current

genetic system is discussed later in the experimental part in section 6.1, since there must be some reasoning given.

5.3.2 Mutation

The idea of mutation is to provide some variance for the values of the parameters. Like in crossover, there are the key issues: 1) defining the rule, according to which the mutation happens, 2) fixing the probability of the mutation, i.e. how often the parameters should be mutated.

Undoubtedly, the development of the proper mutation operator is a consequence of the analysis of the problem specification and the representation of the solution. And the conclusion may even be that of applying no mutation at all, like it happened in [17]. For the problem of this thesis one mutation operation affects one parameter's value, and this change can happen differently. The simplest idea is to replace the current value with a randomly generated new one from the parameter's range. This is of pure chance and is by no means connected with the parameter's previous value.

Quite often the natural method for the mutation is to swap two elements of the string. It is usually applied when all parameters are equivalent, thus, it is not applicable for the present genetic model.

Another possibility of modifying the parameter is to use *normal (Gaussian) distribution* [14] as its basis. Up to a random number between 0 and 1 (vertical axe of Fig. 18), the corresponding number of the horizontal axe (positive or negative, up to a random binary number) is added to the current parameter value.

One more way is to alter a parameter taking into account its current value. For example, the following formula was proposed in [33]:

$$y_m = y \pm y \cdot \left(1 - r^{\left(1 - \frac{t}{T}\right)^b} \right), \tag{3}$$

where $r \in [0..1]$ is a random number, b is a parameter, t is the number of the current generation, and T is the overall number of generations. The sign (plus or minus) is chosen randomly. This change decreases with time, and the reason is to vary the value greatly and

search the entire space in the beginning, and make very minor alterations at larger stages, so that to increase the probability of generating a number close to its successor [33].

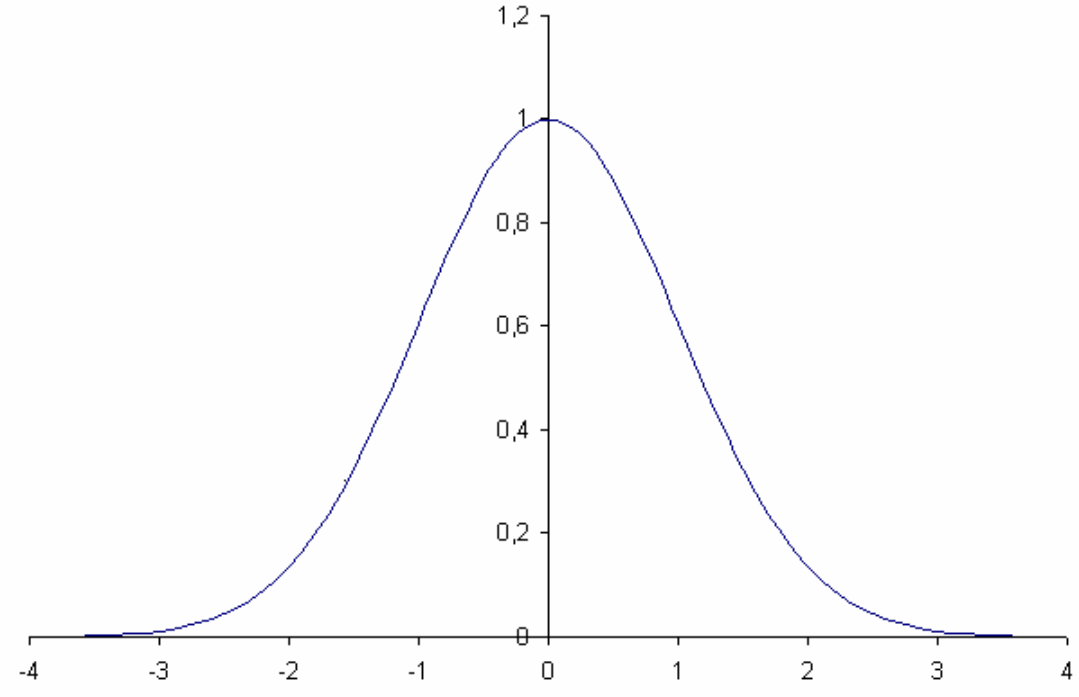

Fig. 18: Normal density function.

All the above possibilities of mutation seemed to serve well, and there were no good reasons found that could give privilege to any. Moreover, all parameters are integers and do not have large ranges to vary. It restricts the best possible accuracy to 1, and using, for example, (3) in this case seems quite unreasonable, because the change produced by this formulae in late generations will not have any effect. Finally, I chose a simple random mutation.

The second fundamental issue of the mutation operator is its frequency, i.e. *mutation rate* p_m. It is much likely that the most appropriate frequency is obtained always empirically. There are, however, some commonly used variants: for large population sizes the probability of around 0.001 is chosen, and for small population sizes, when the diversity within one population is much smaller, this number can increase up to 0.1 [33].

The mutation rate p_m of the present work is also discussed in the experimental part in section 6.1, together with the probability of crossover p_c.

5.4 Evaluation and selection

As it was already mentioned earlier, the most important part of every GA is its fitness function. While preparing this thesis, I have studied some works on the same topic, i.e. applying GAs for the evaluation in chess [7, 28, 45].

In both [7] and [45] the idea is that every individual is given a certain number of positions from the famous games played by grandmasters. The player has to find the best move in these positions using its own evaluation function. Assuming that the grandmasters' moves are perfect in the overwhelming majority of the positions, fitness of the individual is then calculated by means of the number of matches between the grandmasters' moves and the ones made by individual and then divided by the total number of positions it was given to analyse.

In my opinion, such an approach has some disadvantages. First of all, there is still a big discussion about computer's playing in the middle stage of the game. For example, it is well known to a highly experienced human chess-player that in some positions there exists the so-called *positional pawn sacrifice*. If accepted, it allows the sacrifying side to possess an initiative and thus create some enduring pressure on the opponent's pieces with the sound hope that it will finally lead to a deciding advantage in the game. Humans are applying this technique successfully. But as for a computer, it cannot play in such way. The main thing for a computer is the position scoring, and if it is unable to find whichever alternative for that pawn, it will never play so.

To prove the opinion, there is a game, Deep Blue vs. G. Kasparov, California, USA, 1996 [48, 51], where Deep Blue left a central pawn en prise. And only as that game continued, it became clear that the machine simply calculated the return of the material, which was not obvious.

Next, every grandmaster thinks differently in a large number of complicated positions (otherwise there will be no resulting games). Therefore it can not be excluded that a computer plays other move than the grandmaster, whose game is being analysed, played, which means that making "wrong" move computer does not get a point. This is actually somewhat unfair. There is, undoubtedly, another grandmaster, which would make the same move as the computer did. Moreover, his move would lead to the exactly same result. But if the latter

grandmaster's game was considered (even assuming different position) instead, the computer would much more likely get a point for the evaluation of the given position.

As one more curious situation relative to the move selection, let's consider the position depicted in Fig. 19 and discussed in [15]. In this position, there is a clear and obvious win for white that even a novice player can recognise in a moment. The white pawn on *a2* is closer (in moves) to its promotion square *a8* than the black king and its advance cannot be hindered. Therefore the only correct move for white in this position is *1. a4*.

But a static analysis of this position results in a doubtless advantage for black, since black has three pawns and white has only two. And if the machine took on white and were offered a draw, it would readily accept even though this would be utter folly.

The chess engine used in [15] after 3-ply search played *1. b4 f5 2. a4* with the negative score for white. The comment given in the book says that the machine obviously "misunderstands" the position. After 5-ply search, it concluded with *1. b4 f5 2. a4 g5 3. Kf2* and negative score, still being blind to the promotion. After 7-ply search, it played *1. b4 f5 2. a4 g5 3. Kf2 h5 4. Kg3* and negative score. And only after 9-ply search, the move line was correct *1. a4 h5 2. a5 h4 3. a6 h3 4. a7 h2+ 5. Kxh2*, and the score finally became positive. The comments for this choice say that the reason why white play *1. a4* is that they win a pawn, but not because of the promotion!

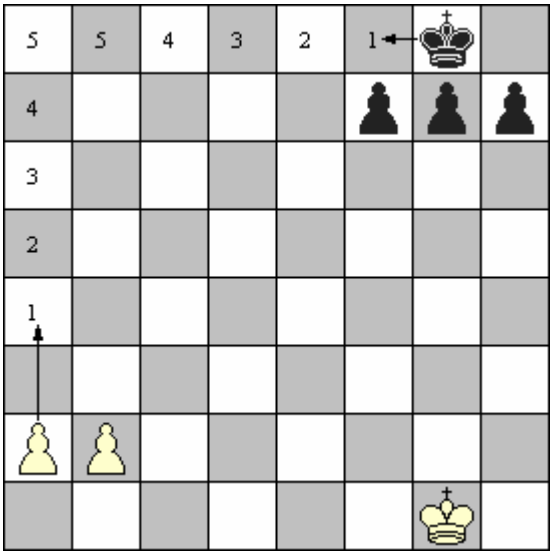

Fig. 19: White to move and win.

I repeated the above procedure using my own program, and qualitatively the result proved to be absolutely the same, i.e. the machine finally achieved positive score only after 9-ply search. The main conclusion of this small but priceless try is that the computer picks the right move for the wrong reason, and it needs a 11-ply search to understand that *1. a4* leads to the promotion and not the pawn capture.

So, you see how it goes with the computer. Of course, the situation above is very simple and nowadays will never happen to a high-level chess-playing computer. But the number of chess positions is enormous, the search is not limited to 11 plies, and the main feature I concluded is that encouragement given for making the right move, but using the wrong reason, is not the best way to train a computer to play chess, or even to train at all. Nevertheless, the technique of move matching can be surely successfully applied, and works [7, 8] are an excellent proof, but it seems to be too subjective and makes a computer depend greatly on how humans play, whereas the latter is too far away from being understood.

Another approach, proposed in [28], organises a match of two games, one with white and one with black, between two individuals from the population. The winner is selected for the next generation and the loser is replaced with the winner's mutated clone. The method seems to be more suitable for the computers if for no other reason than its independence from human chess.

Each-with-each tournament system, also called *Round Robin* [38, 54], can be considered as an extension of the above method. With time the practice proved that it achieves the most truthful results wherever used, not only in chess. In such a tournament every participant plays against every other, and one mistake made here still most likely leads to the actual final standings whereas it might not and even often does not in other systems such as *Swiss* system invented by J. Muller in 1895 [38, 54], or *Knock-out* system (in which the loser is eliminated from the tournament). The only essential drawback of Round Robin is that it is very time taking and hence often simply not acceptable in real-life tournaments. Nevertheless, I chose Round Robin for my experiments.

Number of participants of the tournament is discussed later in section 6.1. There are also some estimation numbers presented. One player will always have a set of predefined values, will never take part in the genetic process and will be moved to the next generation independently of the results it achieved in the tournament. This individual, called here *the*

reference player, is involved into the population in purpose. Its values for the pieces given in the right column of Table 3 are close to the values, which proved to show better performance in various previous works. As for the other parameters, I could not find any sufficient information on their exact values, and therefore each was assigned middle value of the corresponding range. The number of points (win is one point, and draw is half a point) an individual obtains in a single tournament stands for its fitness measure, and fitness value f_i of the ith individual is then his score divided by the maximal possible number of $2\cdot(N-1)$ points, where N is the population size.

The selection is done by means of *roulette wheel*, in which an individual is chosen for breeding according to the scheme given in [33]. It is described as a C-like program in Fig. 20.

```
population  RouleteWheelSelection(population POP)
{
     int N = GetPopulationSize(POP);
     float f[N-1],p[N-1],q[N-1];
     float r;
     int F;
     int i,j;

     //Assign a fitness value f[i] to each individual of POP
     for i=1 to N-1
          f[i] = (POP[i].points)/MAX_POINTS;

     //Calculate the total fitness of the population as the sum
     //of all fitness values
     F = 0;
     for i=1 to N-1
          F = F + f[i];

     //Calculate selection probability for each individual
     for i=1 to N-1
          p[i] = f[i]/F;

     //Calculate the cumulative probability q[i] for each
     //individual
     for i=1 to N-1
          for j=1 to i
               q[i] = q[i] + p[j];

     //Start roulette wheel
     for i = 1 to N-1
     {
          //Generate random number r from the range [0..1]
          r = random(0..1);

          //Select the first individual if r < q[1],
          //the j-th one if q[j-1] < r <= q[j]
          j = 1;
          while (r > q[j])
               j++;

          NEW_POP[i] = POP[j];
     }

     return NEW_POP;
}
```

Fig. 20: Roulette wheel selection.

6 EXPERIMENTS AND RESULTS

6.1 Some preliminary calculations

In my experiments, due to the speciality of the goals of this research, I have not set any time limits for a game so that every game was played to the end. Therefore a factor of time was the main obstacle that I came across with before starting the evolution process. To solve out the situation, a number of test games were played. Searching to 3 plies, one game took on average three minutes to complete, whereas searching to 4 plies, one game lasted more than thirty minutes. For the latter case, assuming that the evolution process would have, for example, 40 generations, and each Round Robin tournament would have 20 games for only 5 participants (if there are N players in the tournament and each pair plays twice, then $N \cdot (N\text{-}1)$ games must be played), the lower bound for the entire evolution process appears to be around $20 \cdot 40 \cdot 30 = 24000$ minutes, or 400 hours of work for one computer. And if we assume also that population size must be increased in order to provide better diversity, then this number becomes unacceptable, indeed. Using 3-ply search depth, the entire evolution process appears to be already around $20 \cdot 40 \cdot 3 = 2400$ minutes, or 40 hours of work for 5 players, and around $90 \cdot 40 \cdot 3 = 10800$ minutes, or 180 hours of work for 10 players. Taking into account that some time should also be spent to process all the results and create the next generation, 3-ply search seemed to be the most suitable.

Thus, we have 3-ply search, 10 individuals (including the reference player) and 40 generations to complete the evolution process. And now, as there are all numbers fixed, the probabilities of crossover p_c and mutation p_m must be specified. For example, in [33] the probabilities of $p_c = 0.25$ and $p_m = 0.01$ were set for the population size of 20 individuals, each having 33 binary parameters. My population is two times less, but each individual has 26 parameters, and these parameters are integers instead of binaries, so that they have more values to take. I selected $p_c = 0.5$ and $p_m = 0.01$.

6.2 General playing style and its problems

The only possibility to win a chess game is to checkmate an opponent's king. In several types of positions, like "king against king and queen", "king against king and rook", "king against

king and two bishops", "king against king, bishop and knight" there is an algorithm that tells how to checkmate. None of them was implemented in the current program. Instead, for the simplicity, each game was played unless one of the players achieved the best score below some threshold of hopelessness two times in a row. In this case the program immediately resigned and the game ended. Of course, it does not mean that there could no checkmate happen: there were the positions, from which no escape was possible and king was eventually checkmated. The threshold was set so that, according to the range of the parameters, the mentioned positions were supposed to satisfy that condition.

However, when the experiment had already begun and a number of generations were already performed, it turned out that there were the positions from the list above, in which a losing side didn't resign as it should have had to. It happened because of no other reason than the score it achieved was above the threshold due to one or the other factor that was trifling from the quantitative point of view, but that appeared to have a tremendous qualitative effect. For example, white castled its king in the beginning and black did not. It means that white always gets a bonus value for castling, whereas black gets a punishment for missing its castling opportunity. And the game continued so that eventually black has king and rook, whereas white has only king left. The evaluation function in this case still punishes black and gives a bonus to white. Therefore, depending on the value of the rook, white may get the score, which is above the threshold, so that it continues playing. And the game becomes drawn, because black does not know how to checkmate. The evident solution could have been to decrease the threshold, and it would certainly have solved that problem. But that circumstance was found out quite late after the beginning and it would have meant to break the already running evolution process and to start a new one, but that was not possible because of the time limits. Therefore, it was taken as a matter of fact, taking into account a considerable sparsity of such positions.

Draw could only happen in three cases: 1) there was a stalemate, 2) the same position appeared three times, 3) there were 50 moves played, during which no change of material balance happened and no pawn move was made. The last two are the official rules adopted by *World Chess Federation (FIDE).*

A draw of the second case can also mislead the natural continuation of the game. Assume, there is a complicated position in the middle game, in which one side has a "piece-for-a-pawn" advantage. In general, this side must win the game. But it behaves so that the best

moves it selects do not use the advantage, e.g. it places a knight from one square to another and back. This leads to three-times repetition of the same position. As a result there is a drawn game, what is actually not the expected result. In this case a program must not make the best move in order to use the advantage it has, but this is not a trivial circumstance to handle. And surely, there were the games observed during the experiment, in which the situation took place.

All described cases had an effect on the final standings of the individuals in every tournament, which, in turn, had an indirect effect on the selection process and, hence, on the further evolution process.

Since there was no opening database used, most of the games of the late tournaments were mainly of a few game paths up to some moment. Later on, the program became able to find several best moves and had a choice of the possible continuations, thus providing a natural diversity. This fact has the same meaning as the database of openings has, i.e. storing the best known move for each position, but only much smaller in its size and according to a particular evaluation function.

Many games ended with the different score than they were expected to. It happened because of the three factors. The factor of incorrect draws (instead of wins) was already discussed above. The second possibility deals with the set of the evaluation parameters and will be paid a special attention to. The third factor, the one of the search depth, we will now discuss.

As it was already said, the deepest acceptable depth search was 3 plies. Obviously, it greatly reduces the quality of playing. The modern chess computers search for 10 and more plies. Moreover, they are usually limited in time. It becomes possible, because the authors of those programs have implemented most of the techniques discussed in section 2.2 of this thesis and may have also added some additional, very specific and complicated improvements to their algorithms. Surely, they must have spent a considerable amount of time making everything work correctly, since, in my opinion, some issues require a great number of tests run in order to be sure that the feature is functioning perfectly. In addition, they already have an evaluation function developed.

It appeared during the running of the experiment that a quiescence search must play the decisive role for a computer chess program, and it is, in fact, closely interconnected with the depth of the search. I realised that a side on move always tries to make qualitatively the same

move in reply. For example, it attempts to attack an opponent's piece if an own piece just was attacked, and such a style of playing can last while the program is able to do so. It also gives a check almost whenever possible, since this type of move very often does not change material balance. These two factors, when coupled, can cause the game to end in just a few moves after the piece exchange starts. Simply because of the possibility for one side to check an opponent's king by means of moving a piece that was under attack. And after the check, to take opponent's piece, which is still being attacked.

The other possible mistake sets out as a part of the previous one but it concludes more in insufficiently deep search than in the explosiveness of the position. To clear up, there were the positions, in which a queen (a rook) was trapped in three and more moves. Here, the trick was that up to some moment a piece was placed to a safe square every time, but finally, after a sequence of such "safe" escapes, the best move was to sacrifice that piece for a less valuable one. To prevent this, it is necessary to search deeper.

6.3 The results

The results of each generation were processed manually. Game results were stored in separate cross tables. An exemplar cross table (for the 23rd generation) is given in Table 6.

Table 6: Tournament cross-table.

ROUND 23											**points**	**place**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**const**		
1	+	0 0	0 0	0 1	1 0.5	0 0.5	0 0	0 1	0 0.5	1 0.5	**6**	**ix**
2	1 1	+	1 1	1 0	0.5 1	0.5 1	0 1	0 0	0 1	1 1	**12**	**ii**
3	1 1	0 0	+	0 1	1 0.5	1 0	1 1	1 1	1 0.5	1 0.5	**12.5**	**i**
4	0 1	1 0	0 1	+	0 1	0 0	0 0.5	0 0.5	0 0.5	0 0	**5.5**	**x**
5	0.5 0	0 0.5	0.5 0	0 1	+	1 1	1 0	1 0.5	0 0	1 1	**9**	**v-vi**
6	0.5 1	0 0.5	1 0	1 1	0 0	+	1 1	0 0.5	0.5 1	1 0	**10**	**iii-iv**
7	1 1	0 1	0 0	0.5 1	1 0	0 0	+	0 0	0 1	1 1	**8.5**	**vii**
8	0 1	1 1	0 0	0.5 1	0.5 0	0.5 1	1 1	+	0 1	0 0.5	**10**	**iii-iv**
9	0.5 1	0 1	0.5 0	0.5 1	1 1	0 0.5	0 1	0 1	+	0 0	**9**	**v-vi**
const	0.5 0	0 0	0.5 0	1 1	0 0	1 0	0 0	0.5 1	1 1	+	**7.5**	**viii**

As it was already said above, fitness of each individual is its score divided by the maximal number of points (18). Those numbers were given as an input for an independent program

that was used to create the next generation. The output of the program was then copied into the files of evaluation functions, which individuals used every time a new game started.

The initial set of parameters for every individual was generated randomly by means of assigning every parameter a value within its range of alteration (see Table 3). These sets are presented in Appendix A. The reference player (referred to as *const*) stood unmodified and took part in tournaments of all generations regardless of the score it achieved in the previous one. In Fig. 21 the places, with which this player scored in every generation, are depicted.

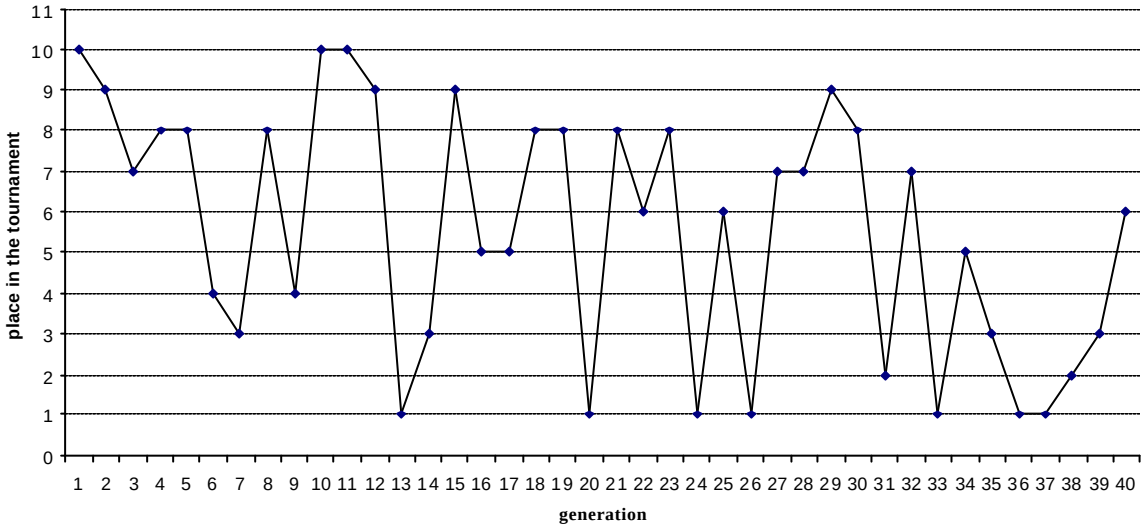

Fig. 21: Performance of the reference player in the evolution process.

The plot has both successful values (i.e. victories in tournaments of generations 13, 20, 24, 26, 33, 36, 37), unsuccessful values (i.e. losses in tournaments of generations 1, 10, 11) and neutral values (i.e. standing in the middle of the cross table) throughout the entire evaluation process. It means that the reference player showed quite unsteady performance.

An interesting moment was noticed in the fifth generation. A fragment of its cross table is shown in Table 7. Here, players 5, 6 and 7 have an identical set of parameters of their evaluation functions given in Table 8.

Table 7: Performance of individuals 5, 6, 7 in the tournament of the fifth generation.

5	0 0	0 1	0 0	0.5 0	+	0 0	0 1	0.5 0.5	0.5 0	0 0.5	**4.5**	**x**
6	0.5 0.5	1 1	1 0	0 0	1 1	+	1 1	0.5 0	1 1	1 0.5	**12**	**i**
7	0 0	0 0.5	0.5 1	0.5 1	0 1	0 0	+	0 1	1 1	1 0	**8.5**	**vii**

Table 8: Parameters of the evaluation function of individuals 5, 6, 7 of the fifth generation.

816	490	316	301	105	25	33	23	1	16	4	20	8	3	23	4	30	29	5	23	13	6	6	14	29	9

However, their standings vary greatly. This fact was detected already when working on the results of the experiment and was taken as a considerable argument in making the conclusions. It somewhat explains the unstable behaviour of the reference player, and I suppose, it appeared to be mostly the result of a weak search procedure, independently of the values of the evaluation function parameters.

After the evaluation process had finished, every parameter converged to some value. The last (40^{th}) generation is presented in Appendix B. There, the value of the second parameter of individual 6 differs from the ones of every other because of the mutation. Otherwise, the difference between the corresponding values of the rest of the parameters is small enough to state the fact of convergence. Moreover, in cases of parameters 4, 5, 12, 14, 15, 17, 18 and 24 the difference is absent at all.

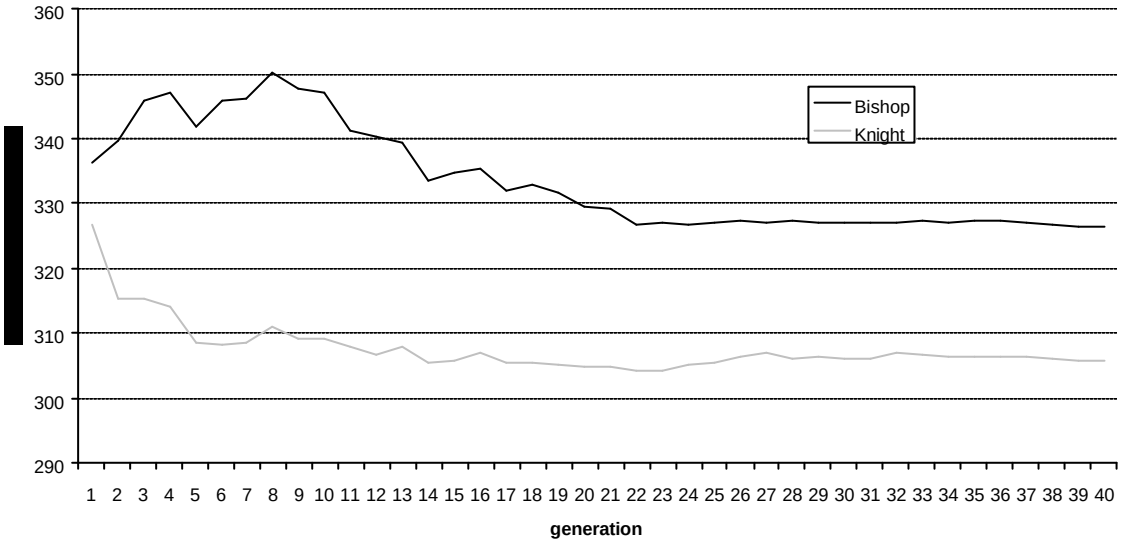

Fig. 22: Development of the average value of bishop and knight.

Now, let's discuss in details the set of the evaluation parameters, which were selected for the current chess program, in terms of the values they converged to. Predictably, the first five parameters standing for the exchangeable pieces eventually got the values they were qualitatively expected to have. The similar situation proved to be for the punishment given for having a bishop or a knight on the first line. Obviously, for bishop it is not so bad, since it is able, in general, to reach the other side of the board in one move while knight must make at

least three moves. And you can see from the final values of the parameters (see Appendix B) that they are worth one third and three fourth of their maximal values (see Table 3), respectively.

Fig. 22 shows the evolution of the population average value of bishop and knight. The plots become smooth already since the second half of the evolution process begins. It means that they have converged to the final values quite soon. Fig. 23 shows the development of the best non-reference player's values for knight and bishop through the evolution process. In the beginning the values varied a lot, and in later generations the roughness is rarer and it arises only when the reference player appeared to win the tournament.

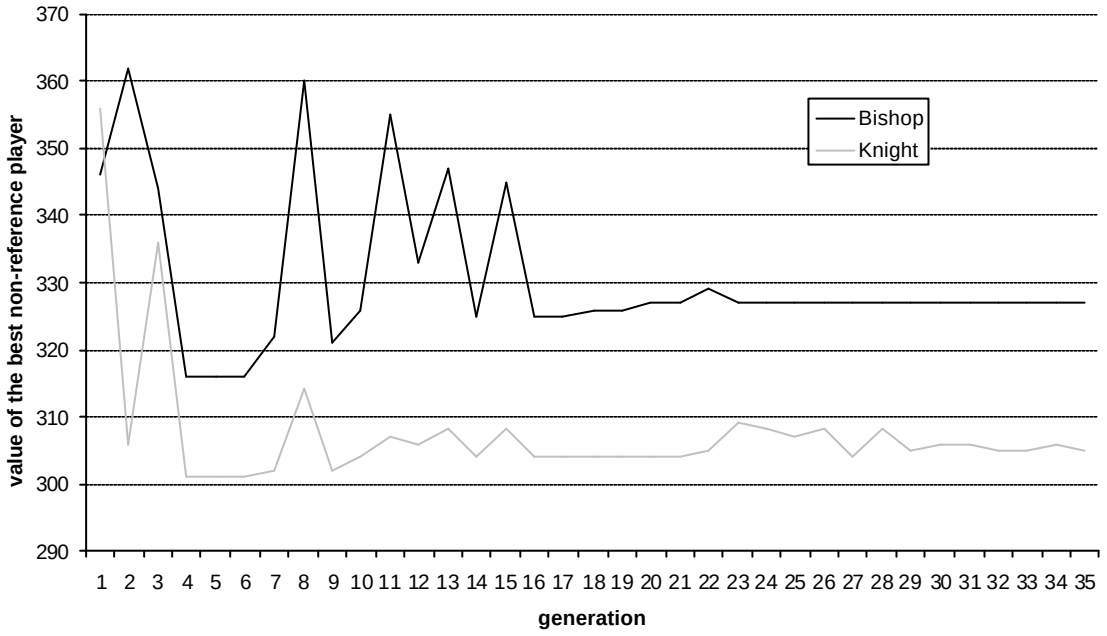

Fig. 23: Development of the best values of bishop and knight of non-reference players.

As for the others then the answer is not so trivial. First of all, it appeared that every other parameter is of very different importance in different situations in the game. You were already given one example on this issue: castling parameters had an influence in the very end of the game whereas they had not to. A pawn in the centre is a desirable position achievement in the beginning when one needs more space to mobilise his forces faster. Later on, when the main action often happens on the flank, such a pawn should be paid less attention. On the contrary, a rook on an open file and a rook on the 7^{th} line are much more important during the later game period as well as far pawn is more indefensible and less useful when there are still a lot of pieces on the board.

All the above and some other facts were in a varying degree being observed when watching how programs were playing against each other in the tournaments. And the main conclusion is that regardless of the depth of the search, there are two possibilities to make evaluation function perform better.

The first one is to use the universal set of criterion, every member of which has approximately the same importance through the whole game duration. In this case, a possible evaluation function seems to be not very complicated and this is the efficient search algorithm that may make a program play better.

The second variant is to divide a chess game into stages and use a separate, perhaps, very different set of parameters during each stage. This question looks quite knotty, for the division cannot be done at once. Moreover, the stage bounds are very particular in every game. It is impossible, in general, to say at some arbitrary position if one stage of the game has gone and the next has already come. And it is not so trivial to invent/develop such criterions. One possible idea of determining a stage of the game might be to consider the overall weight of the army by summing up pieces' values only. A linear function, reducing with every piece moved away from the game board might serve as an indicator. And within some range it can be considered to be either the leaving or the coming stage. But certainly, the parameters like castling must not have an influence on the game flow in endings. In my opinion, this variant of improving the evaluation function seems to be more promising, as it tries to utilise the individuality of a chess position.

As a final testing, two matches of 20 games each were organised. The first one was played by the best individual of the last generation and an individual with a random set of parameter values (the values of individual 6 of the first generation were taken). The match was won by the first player with the score 12-8, of those 4 games were drawn. The second match was between the best individual of the last generation and the individual, which only had the same values for the pieces in its evaluation function, so that every other parameter was set to 0. The match was one by the first player with the score 13-7, of those also 4 games were drawn. Thus, in both cases the developed player gained a victory. The advantage did not appear to be striking, but it indicates an improvement.

7 CONCLUSIONS

The work has shown that genetic algorithms can be successfully applied for determining the fittest values for the parameters involved into the numerical evaluation of a chess position.

Taking into consideration all the discussion of Chapter 6, the values obtained in the current work can be considered as the optimal values for the selected parameters and the implementation developed. Being of the primary importance for a chess position, the parameters for the pieces have converged to more or less expected values, and they are likely to show successful performance when coupled with a more complicated search algorithm. I think, they should not either dramatically change the quality of playing when involved into another evaluation function. Other parameters all together might not be correct from the above point of view. The main reason is that their tuning could have been ruined and misled by a couple of "insufficient-search-based" piece sacrifices. It means that the optimality of such a subtle parameter as, for example, "adjacent pawn" must be based on the excellent search technique. Similarly, such values as bonus for making castling and punishment for its omission can be disputed.

The current work has shown that the selected topic is of interest to be continued and amended. Therefore the future work should aim at improving the present results by means of expanding the search procedure and mastering the second approach of complicating the evaluation function.

REFERENCES

[1] Andris, P., Frollo, I., Optimisation of NMR Coils by Genetic Algorithms, *Measurement Science Review*, Vol. 2, Section 2, pp. 13-22, 2002.

[2] Bartlett, P., Downs, T., Training a Neural Network with a Genetic Algorithm, *Technical Report*, Dept. of Electrical Engineering, University of Queensland, 1990.

[3] Baxter, J., Tridgell, A., Weaver L., Learning to Play Chess Using Temporal-Differences, *Machine Learning*, 40 (3), 2000.
(Note: in fact, the source code of the program used in this experiment was examined to inspect the values).

[4] Beal, D.F., Experiments with the Null Move, *Advances in Computer Chess 5*, (ed. Beal, D.F.), pp. 65-79, Elsevier Science Publishers, Amsterdam, 1989.

[5] Bennett, K., Ferris, M.C., Ioannidis, Y.E., A Genetic Algorithm for Database Query Optimization, *Proceedings of the 4th International Conference on Genetic Algorithms*, pp. 400-407, 1991.

[6] Berliner, H.J., *Chess as Problem Solving: The Development of a Tactics Analyzer*, Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA, 1974.

[7] Cogley, J., *Designing, Implementing and Optimising an Object-Oriented Chess System Using a Genetic Algorithm in Java and its Critical Evaluation*, M.Sc. thesis, Open University, 2001.

[8] Condon, J.H., Thompson, K., Belle Chess Hardware, *Advances in Computer Chess 3*, (ed. Clarke, M.R.B.), pp. 45-54, Pergamon Press, Oxford, 1982.

[9] Crawford, K.D., Solving the n-Queens Problem Using Genetic Algorithms, *Proceedings of ACM/SIGAPP Symposium on Applied Computing*, pp. 1039-1047, 1992.

[10] Darwin, C.R., *The origin of species*, Murray, London, 1859.

[11] Davis, L., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.

[12] Donninger, C., Null Move and Deep Search: Selective Search Heuristics for Obtuse Chess Programs, *ICCA Journal*, 16 (3), pp. 137-143, 1993.

[13] Falkenauer, E., Bouffouix, S., A Genetic Algorithm for Job Shop, *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 824-829, 1991.

[14] Feller, W., *An Introduction to Probability Theory and Its Applications*, volume I, John Wiley & Sons, Inc., third edition, 1968.

[15] Frey, P.W., *Chess Skill in Man and Machine*, Springer-Verlag, New York, 1977.

[16] Friedel, F., A Short History of Computer Chess, (http://www.chessbase.com/columns/column.asp?pid=102), visited 17.03.2004

[17] Fränti, P. Genetic Algorithm with Deterministic Crossover for Vector Quantization, *Pattern Recognition Letters*, 21 (1), pp. 61-68, 2000.

[18] Ghory, I., Commentry on Sam Sloan's Origin of Chess, (http://www.bits.bris.ac.uk/imran/games/sloan.html), visited 15.03.2004

[19] Ghory, I., The history and origin of chess, (http://www.bits.bris.ac.uk/imran/games/chesshistory.html), visited 15.03.2004

[20] Goetsch, G., Campbell, M.S., Experiments with the Null-Move Heuristic, *Computers, Chess, and Cognition*, (eds. Marsland, T.A., Schaeffer, J.), pp. 159-168, Springer-Verlag, New York, N.Y, 1990.

[21] Goldberg, D.E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, MA, 1989.

[22] Groß, R., Albrecht, K., Kantschik, W., Banzhaf, W., Evolving Chess Playing Programs, *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 740-747, 2002.

[23] Heinz, E.A., *Scalable Search in Computer Chess*, Vieweg Verlag, 2000.

[24] Heisman, D., *Evaluation Criteria*, 2003, (http://www.chesscafe.com/text/heisman27.pdf), visited 29.03.2004

[25] Holland, J.H., *Adaptation in Natural and Artificial Systems,* The University of Michigan Press, Ann Arbor, 1975.

[26] Homaifar, A., Turner, J., Ali, S., The n-Queens Problem and Genetic Algorithms, *Proceedings of IEEE SOUTHEASTCON-92,* pp. 262-267, 1992.

[27] Hsu, F., *Behind Deep Blue: Building the Computer That Defeated the World Chess Champion*, Princeton University Press, Princeton, NJ, 2002.

[28] Kendall, G., Whitwell, G., An Evolutionary Approach for the Tuning of a Chess Evaluation Function using Population Dynamics, *Proceedings of Congress on Evolutionary Computation*, pp. 995-1002, 2001.

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

[30] Laramée, F.D., *Chess Programming*, Series of Articles, 2000.
(http://www.gamedev.net/reference/articles/article1014.asp),
(http://www.gamedev.net/reference/articles/article1046.asp),
(http://www.gamedev.net/reference/articles/article1126.asp),
(http://www.gamedev.net/reference/articles/article1171.asp),
(http://www.gamedev.net/reference/articles/article1197.asp),
(http://www.gamedev.net/reference/articles/article1208.asp),
visited 29.03.2004

[31] Linhares, A., Data Mining of Chess Chunks: a Novel Distance-Based Structure, *Data Mining IV*, 2003.

[32] Marsland, T.A., *Computer Chess and Search, Computer Science Department,* University of Alberta, Edmonton, Canada, 1991.

[33] Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs,* Springer-Verlag, 1996.

[34] Michalewicz, Z., Krawczyk, J., Kazemi, M, Janikow, C., Genetic Algorithms and Optimal Control Problems, *Proceedings of the 29th IEEE Conference on Decision and Control*, pp. 1664-1666, 1990.

[35] Murray, H., *History of Chess*, 1913.

[36] Newell, A., Shaw, J.C., Simon, H.A., Chess Playing Programs and the Problem of Complexity, *IBM Journal of Research and Development* 4(2), pp. 320-335, 1958. Also Feigenbaum, E., Feldman, J. (eds.), *Computers and Thought*, pp. 39-70, 1963.

[37] Pierre, S., Legault, G., A Genetic Algorithm for Designing Distributed Computer Network Topologies, *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 28(2), pp. 249-258, 1998.

[38] Pribut, S., Frequently Asked Questions (with answers) for Chess Players, 2002. (http://www.faqs.org/faqs/games/chess/part1/), visited 17.03.2004

[39] Samuel, A.L., Some Studies in Machine Learning Using the Game of Checkers, *IBM Journal of Research and Development*, 3 (3), pp. 210-229, 1959. Also Levy, D. (ed.), *Computer Games 1*, Springer-Verlag, pp. 335-365, 1988.

[40] Shannon, C.E., Programming a Computer for Playing Chess, *Philosophical Magazine*, 41 (4), pp. 256-275, 1950.

[41] Shipov, S., Review of "Lausanne Young Masters 2003" chess tournament, (in Russian), 2003. (http://www.worldchessrating.ru/themes/basic/review-content.asp?folder=22&matID=1463), visited 29.03.2004

[42] Sloan, S., *The Origin of Chess*, Sloan Publishers, 1985. (http://www.ishipress.com/origin.htm), visited 29.03.2004

[43] Sun, C.-T., Wu, M.-D., Multi-Stage Genetic Algorithm Learning in Game Playing, *Proceedings of NAFIPS/IFIS/NASA-94*, pp. 223-227, 1994.

[44] Tao, G., Michalewicz, Z., Inver-Over Operator for the TSP, *Proceedings of the 5th Parallel Problem Solving from Nature*, Lecture Notes in Computer Science, pp. 803-812, 1998.

[45] Tunstall-Pedoe, W., Genetic Algorithms Optimising Evaluation Functions, *ICCA Journal* 14 (3), pp. 119-128, 1991.

[46] Turing, A.M., Strachey, C., Bates, M.A., Bowden, B.V., Digital Computers Applied to Games, in Bowden, B.V. (ed.), *Faster Than Thought*, Pitman, pp. 286-310, 1953.

[47] Vasiukov, Y., *Comments on the game Vasiukov – Van Wely, Aeroflot Open, Moscow, 2002.* (http://www.aeroflotchess.com/rus/best/bestgame.htm), visited 29.03.2004

[48] Veretnov, L., Coach, Computer, Analysis, (in Russian). (http://www.chessib.com/rus/veretnovtca.html), visited 29.03.2004

[49] Wu, M.-D., Liao, Y.-H., Sun, C.-T., Network Tournament Pedagogical Approach Involving Game Playing in Artificial Intelligence, *Journal of Information Science and Engineering*, 19, pp. 589-603, 2003.

[50] Zobrist, A.L., A New Hashing Method with Application for Game Playing, *Technical Report 88*, Computer Science Department, The University of Wisconsin, Madison WI, USA, 1970. Also in *ICCA Journal*, 13 (2), pp. 69-73, 1990.

[51] *Deep Blue – Kasparov, 1996, Game 1 (chess)*, (http://chess-guide.fateback.com/famous_games/deep_blue-1996.html), visited 27.03.2004

[52] *History of Chess*, (in Russian), 2000, (http://webcenter.ru/~brdgames/brdgames/chesshistory.htm), visited 29.03.2004

[53] *Napoleonic Battles in Corsica*, Overview of the Quarter Finals, 2003. (http://www.chessbase.com/newsdetail.asp?newsid=1279), visited 29.03.2004

[54] *Swiss System vs Round Robin Pairings*, (http://scichess.org/faq/swiss.html), visited 17.03.2004

[55] *Wikipedia, the Free Encyclopedia*, (http://www.wikipedia.org), visited 29.03.2004

APPENDICES

Appendix A: Initial generation.

	individual:									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**const**
parameter:	**assigned values:**									
0	905	818	965	824	880	888	808	861	995	900
1	522	526	476	470	456	527	494	468	470	500
2	346	344	359	308	340	366	317	310	336	340
3	356	333	321	338	344	299	302	292	356	330
4	93	105	111	88	102	92	104	112	110	100
5	19	16	36	11	7	27	28	4	31	20
6	14	2	2	5	39	2	26	34	29	20
7	14	8	43	3	15	3	21	38	48	25
8	5	13	25	23	13	7	1	1	21	15
9	4	24	28	20	15	29	14	28	23	15
10	1	1	3	16	18	5	1	15	11	10
11	19	26	25	15	27	22	20	23	27	15
12	13	34	39	14	22	27	5	27	32	20
13	13	23	15	2	26	18	2	10	3	15
14	9	5	20	13	27	12	24	17	26	15
15	0	3	1	3	3	3	4	1	3	3
16	38	20	24	7	20	24	30	31	10	20
17	20	32	24	29	9	31	30	24	11	20
18	3	7	16	19	1	8	2	26	3	15
19	6	26	7	3	23	17	24	16	5	15
20	0	29	25	23	4	5	13	15	28	15
21	5	9	13	3	13	11	5	14	4	7
22	0	1	17	5	17	13	6	8	4	10
23	1	1	16	16	15	2	16	0	1	10
24	29	11	21	5	12	10	29	26	20	15
25	12	6	17	19	4	8	10	6	20	15

Appendix B: Last generation.

	individual:									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**const**
parameter:	**assigned values:**									
0	860	860	860	860	861	861	860	860	860	900
1	485	485	486	485	485	519	485	485	485	500
2	327	327	327	327	327	327	325	326	326	340
3	306	306	306	306	305	305	306	306	306	330
4	105	105	105	105	105	105	105	105	105	100
5	27	27	27	27	27	27	27	27	27	20
6	27	27	27	27	26	26	26	27	26	20
7	30	30	29	30	30	30	30	30	30	25
8	3	3	3	3	3	3	4	3	4	15
9	19	19	21	19	20	20	18	19	18	15
10	9	9	9	9	10	10	9	9	9	10
11	23	23	23	23	23	23	22	23	22	15
12	17	17	17	17	17	17	17	17	17	20
13	17	17	17	17	18	18	18	17	18	15
14	17	17	17	17	17	17	17	17	17	15
15	2	2	2	2	2	2	2	2	2	3
16	23	23	23	33	24	28	23	33	27	20
17	27	27	27	27	27	27	27	27	27	20
18	12	12	12	12	12	12	12	12	12	15
19	12	12	13	12	12	12	12	12	12	15
20	13	13	14	13	13	13	13	13	13	15
21	8	14	8	7	7	8	7	7	8	7
22	12	12	12	12	12	12	12	12	12	10
23	7	7	6	6	5	6	6	6	7	10
24	22	22	22	22	22	22	22	22	22	15
25	15	15	12	14	13	14	14	15	14	15