Evolving chess playing programs

R. Groß, K. Albrecht, W. Kantschik and W. Banzhaf
Dept. of Computer Science
University of Dortmund
Dortmund, Germany

Abstract

This contribution introduces a hybrid GP/ES system for the evolution of chess playing computer programs. We discuss the basic system and examine its performance in comparison to pre-existing algorithms of the type alpha-beta and its improved variants. We can show that evolution is able to outperform these algorithms both in terms of efficiency and strength.

1 Introduction

Computer programs capable of playing the game of chess have been designed for more than 40 years, starting with the first working program that was reported in 1958 [BR58]. Since then countless numbers of programs were developed and appropriate hardware was designed.

This article introduces a chess program which learns to play the game of chess under limited resources. We want to demonstrate the capabilities of Computational Intelligence (CI) methods to improve the abilities of known algorithms. More precisely we investigate the power of Genetic Programming (GP) [BNKF98] and Evolutionary Strategies (ES) [Sch96] using the example of computer chess. The relevance of computer chess is probably comparable to that of fruit flies in genetics, it is a laboratory system with paradigmatic character.

In previous work we have studied the evolution of chess-playing skills from scratch [BKA+00]. In this contribution we do not want to create an entirely new program to play chess. Instead, we start with a scaffolding algorithm which can perform the task already and use a hybrid of GP and ES to find new and better heuristics for this algorithm. We try to improve a simple algorithm, the *alpha-beta* algorithm. In order to evolve good standard heuristics we use evolutionary techniques.

It is very time consuming to evolve chess playing individuals. Thus the basis of our evolutionary system is a distributed computing environment on the internet called ***qoopy***. Distributed computing is necessary because of the high costs of one fitness evaluation for a chess program. Each individual has to perform several games against computer programs and a game might last several hours in worst case. The additional computer power needed to perform the evolution of chess programs is borrowed from participating users worldwide through the internet.

The performance of our evolved programs is neither comparable to Deep Blue [Kor97] nor to other computer chess programs playing at expert level. This was not intended at the present stage of development.

2 The Chess Individuals

We use an *alpha-beta* algorithm [Sch89] as the kernel of an individual which is enhanced by GP- and ES-modules. The goal of these modules is the evolution of smart strategies for the middle game. So no opening books or end game databases have been used to integrate knowledge for situations where a tree search exhibits weak performance. Also, GP/ES individuals always play white and standard black players are employed to evaluate the individuals. The black players are fixed chess programs which think a certain number of moves ahead and then choose the best move (according to the minimax-principle, restricted by the search horizon) [Bea99, BK77]. The individuals are limited to search not more than an average of 100,000 nodes per move to ensure an acceptable execution speed.

Like standard chess programs, individuals perform a tree search [IHU95]. In particular, they use an alpha-beta-algorithm. Three parts of this algorithm are

Table 1: The pseudocode shows the $\alpha\beta$ algorithm with the evolutionary parts (bold).

$\alpha\beta_{max}$ (position K; integer α, β) {
  integer $i, j, value$;
  $nodes = nodes + 1$;
  IF POSITION_DECIDED(K) *THEN*
   RETURN **position module** (K);
  IF ($depth == maxdepth$) *THEN*
   RETURN **position module** (K);
  $restdepth$ = **depth module** ($restdepth$);
  IF ((($restdepth == 0$) OR
  ($nodes > maxnodes$))
   AND ($depth >= mindepth$)) *THEN*
   RETURN **position module** (K);
  determine successor positions $K.1, \ldots, K.w$;
  move ordering module ($K.1, \ldots, K.w$);
  $value = \alpha$;
  FOR $j = 1$ *TO* w *DO* {
    $restdepthBackup = restdepth$;
    $restdepth = restdepth - 1$;
    $value$=max($value, \alpha\beta_{min}(K.j, value, \beta)$);
    $restdepth = restdepthBackup$;
    IF $value \geq \beta$ *THEN* {
     ADJUST KILLERTABLE;
     RETURN $value$;
     }
  }
  RETURN $value$;
}

evolved:

- The depth module, which determines the remaining search depth for the given node.
- The move ordering module, which changes the ordering of all possible moves.
- The position module, which returns a value for a given chess position.

Evaluation of a chess individual is performed in the following way (see table 1): The depth module determines the remaining depth for the current level in the search tree. If the position is a leaf then the position module is called to calculate a value for it. Otherwise the node (move) will be expanded and all possible moves will be calculated. Subsequently the move ordering module for these moves is called and changes the order of the moves, so that moves which are more important can be evaluated first in the search tree.

2.1 Depth module

Table 2: The terminal set of the depth module with a short description. With chess-specific operations the depth module receives chess knowledge.

terminal	description of the terminal
accumulator	Default register for all functions, initialized each node.
level register	Special register which holds information of the current level in the search tree, initialized each search.
search/game register	Special register, initialized each search/game.
search depth	Returns the current search depth of the tree.
search horizon	Value of the current search horizon.
piece value	Value of a piece given by the accumulator.
captured piece	Value of a captured piece, if the last move was a capture move.
alpha/beta bound	Value of alpha/beta bound.
move number	Number of current move, given by the move ordering module.
# pieces	Number of knights, bishops, rooks and pawns of the board.
# expanded nodes	Number of expanded nodes for the current position, in percent.
value of move	Value of the move which led to the current position, given by the move ordering module.
branching factor	Number of moves of the precedent position.
position score	Value of the current position, given from the position module.

The depth module decides for each position whether the search tree should be expanded and to what depth.

Normally chess programs have a fixed search horizon [PSPDB96]. This means, that after a certain number of plies the expansion in the search tree will be terminated. In contrast, the depth module should give the individual a higher flexibility in the search to avoid the search horizon effect.

The depth module has two limitations, the search depth and the amount of nodes used in a search tree. The maximal depth is 10 plies but if all moves until ply 10 would be executed approximately 10^{14} nodes would be expanded. Therefore the amount of expanded nodes in the search tree was limited to 100,000 nodes on average per move. On average means, that the individual might save nodes in particular periods of the game to spend them later.

The depth module is a GP-program of branched operation sequences. The structure is an acyclic graph where each node holds a linear program and each if-then-else condition makes a decision which node of the program will be executed next (see section 2.4). Its function set holds very simple functions but its terminal set is more chess-specific, see tables 2 and 3.

Table 3: The function set of the depth module with a short description. No function of the set has chess knowledge.

function	description of the function
+, -, *, /	Arithmetic functions; result is written to the accumulator.
inc/decHorizon	Function to increment/decrement the search depth by one, but the search depth can only be increased/decreased by 2 per level.
sine	The sine function.
sigmoid	The sigmoid function $\frac{1}{1+e^{-terminal}}$.
store in level/game/ search register	Stores the terminal in the level/game/search register.
load	Loads the terminal to the accumulator.
if	If the condition is true the left branch will be executed, otherwise the right one. A condition can be a comparison of two terminals.

The depth module does not define the depth of search directly, rather it modifies how much depth is left for searching - the *restdepth*. It can be incremented or decremented by the operation *incHorizon* and *decHorizon*, or stay untouched. The *restdepth* is initialized with a value of 2. Once a move is executed the *restdepth* is automatically decremented by 1.

2.2 Position module

The position module of an individual calculates a value for the current position.

The position module is a fixed algorithm which evaluates the position by using evolved values for the different chess pieces and some structural values of a check. These values are accumulated whereas *bonus* values are added and *punish* values are subtracted - a higher value corresponds to a better position. We used a simple ES to evolve these values. The idea of this module is to find a better set of parameters than a fixed position evaluation algorithm would provide. Values of hand-written programs are determined through experience of the programmer or by taking parameters from the literature.

Our ES evolves the following weights for the position evaluation algorithm. The first two numbers in brackets reflects the range within which the values can be chosen, the last number is the standard value which was chosen for the black players (see 3.3).

- Values of different pieces: pawn [85-115, 100], knight [290-360, 340], bishop [300-370, 340], rook [440-540, 500], queen [800-1000, 900].
- Bishops in the initial position are punished [0-30, 15].
- Center pawn bonus: Pawns in the center of the chessboard get a bonus [0-30, 15].
- Doubled pawn punishment: If two pawns of the same color are at the same column [0-30, 15].
- Passed pawn bonus: A pawn having no opponent pawn on his and the neighboring columns [0-40, 20].
- Backward pawn punishment: A backward pawn has no pawn on the neighboring columns which is nearer to the first rank [0-30, 15].
- If a pawn in end game is near the first rank of the opponent it gets a promotion bonus depending on the distance, this value fixes the maximal bonus [100-500, 300].
- Two bishops bonus: If a player has both bishops it gets a bonus [0-40, 20].
- A knight gets a mobility bonus, if it can reach more than 6 fields on the board [0-30, 15].
- Knight bonus in closed position: A closed position is defined if more than 6 pawns occupy the center of the board. The center consists of the 16 fields in the center of the board [0-40, 20].
- Knight punishment: If opponent pawns are on each side in end game [0-50, 25].
- Rook bonus for a half open line: A half open line is a line with no friendly pawn that does have an enemy pawn [0-30, 15].

- Rook bonus for an open line: An open line is a line without a pawn on this line [0-30, 15].
- Rook bonus for other positional advantages [0-20, 10].
- Rook bonus: If a rook is on the same line as a passed pawn [0-30, 15].
- King punishment, if the king leaves the first rank during the opening and the middle game [0-20, 10].
- Castling bonus, if castling was done [0-40, 20].
- Castling punishment for each weakness of pawn structure (exception: end game) [0-30, 15].
- Castling punishment, if the possibility was missed [0-50, 25].
- Random value, this is a random value which will be added or subtract from the position value [0-30, 20].

The structure of the position evaluation algorithm for the chess individual and the black player is identical. However there is a difference: Values for individuals are evolved, values for black players are predefined and fixed.

2.3 Move ordering module

The move ordering module of an individual orders the moves for each chess position by assigning a real number to every possible move. The value of a move is the sum of several weighted features of the move. Moves are sorted according to these values and moves will be expanded by this order. By default the value of a feature is in $[0, 100]$.

An ES evolves the following weights for the sorting algorithm:

- Piece values in the opening/middle and end game: Each piece are assigned three values which reflect how important this piece is in the opening/middle and end game.
- Most valuable victim/Least valuable aggressor: The ratio of aggressor and victim move values is calculated. A position with a high ratio is better than one with a smaller value.
- Check: If the move leads to a check position then the move fulfilling this feature gets a bonus.
- Capture move: These are moves which can capture a piece of the opponent.
- Pawn moves that can attack a piece of the opponent.
- Pawn moves that do not attack a piece of the opponent.
- Pawn moves that lead to a promotion of a queen.
- Center activity: Pieces which move from and/or to the center of the chess board gets a bonus.
- Killer moves: Killer moves are moves which often lead to a cut in the search tree. The table contains at most 4 killer moves for each level in search tree. The table will be filled during the search, and if a move is in the killer table it gets a bonus depending on its rank, a lower and upper bound is given by this feature. Besides, the composition of the killer table which changes during search is influenced by the move ordering module of an individual.

Based on these weights, the value for moves will be calculated. Sorting of the moves is very important for the *alpha-beta* search algorithm. If the best move is visited first, the following moves don't need to be considered. A very good move ordering module results in a better performance of the *alpha-beta* algorithm.

2.4 GP structure of the depth module

The depth module of an individual, as illustrated in Figure 1, is represented by a program with nested if-then-else statements [KB01]. This representation has been developed with the goal of giving a GP-program greater flexibility to follow different execution paths for different inputs. It also achieves a reuse of the evolved code more frequently than is the case in linear GP [Nor94, NB95].

A program consists of a linear sequence of statements and if-then-else statements, that contain a bifurcation into two sequences of statements. Nested if-then-else statements are allowed up to a constant recursion depth. The resulting structure is a graph where each node contains a linear GP-program and a decision part. During the execution of the program only one path of the graph will be executed for each input.

Crossover of two programs can be realized in different ways. We have chosen the following two types. The first crossover operator selects a sequence of statements in each program. In case of selected if-then-else statements, the associated statements of the then- and

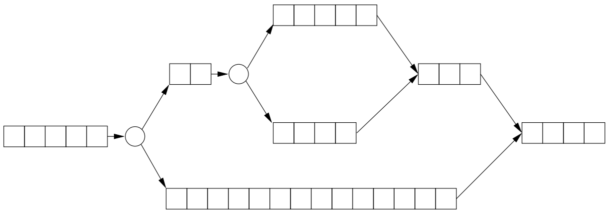

Figure 1: The representation of the GP program, which is a graph structure. The squares represents the linear programs and the circles represents the if-then-else decisions.

else- parts are selected, too. Then the selected sequences are swapped. Secondly, a swapping of branch-free sequences is allowed for an exchange of information between individuals.

Mutation is performed subsequently to crossover or independently from it. There are two types of mutation operators. The first one performs a crossover with a randomly generated individual. The second one selects a sequence of statements (in case of if-then-else statements including the statements belonging to the associated then- and else- parts). Afterwards each statement other than an if-then-else one will be mutated with an adjustable probability (see [KB02]).

3 Evolution

Evolution is based on populations of several individuals. Each individual has to be evaluated by determining its fitness. In our system a fitness case is a chess game and an individual has to play several chess games before its fitness can be assigned. We used the approach of distributed computing [GCK94] to allow for enough computing power. We developed the ***qoopy*** system in order to spread our task among the internet. As opponents of GP-/ES-programs, chess programs with fixed depth were used. Fitness was calculated based on the number of wins, losses and draws against these opponents.

In the following sections we describe this system in more detail.

3.1 Internet-wide evolution based on *qoopy*

qoopy is an environment for *distributed computing* tasks [GCK94, Far98]. It is possible to develop distributed programs for the ***qoopy*** environment and use ***qoopy*** to run these programs.

The first application of ***qoopy*** is **EvoChess**, a distributed software system which creates new chess programs in an evolutionary process. After ***qoopy*** is installed on a machine each participant runs a deme containing a variable number of individuals (default value is 20). In each deme evolution begins and, during the evolution, individuals might be copied between demes (pollination) to create offsprings. ***qoopy*** provides the necessary infrastructure for communication between demes and the connection to an application server.

The application server is necessary because ***qoopy*** has to register users being online to let them connect to each other and to exchange data. The server holds results of the internet evolution, and each deme sends its best individuals and other statistics back to the server on a regular basis.

3.2 Fitness evaluation

The fitness of an individual is a real number between 1 and 15, with higher values corresponding to better individuals. In order to determine fitness, individuals have to play chess games against fixed algorithms of strength 2, 4, 6, 8, 10, 12 and 14. For fitness evaluation an individual always plays white (see 3.3).

The result of a game is a real number between -1 and 1. It is 1 in case that the individual wins the game, -1 if the standard algorithm wins the game and 0 in case of draw. Sometimes it is obvious that one side can win or that the game has to end draw. In such a case the game is stopped to save time. In rare cases lengthy games are aborted because nothing happens anymore.[1] Then the position is evaluated and the result reflects the advantage of white (positive) or black (negative) as a value in the range of $[-1, \ldots, 1]$.

Fitness is initialized with a value of 1. Resulting values are weighted with the number of games played relative to the strength of the opponent. If, e.g., the individual loses twice and wins once against an opponent of strength 6 $(-1, -1, 1)$, this results in values $(5.0, 5.0, 7.0)$, and the fitness is 5.667.

In general, the fitness of an individual is calculated by the following function:

$$fitness = \sum_{j \in \mathcal{C}} \sum_{i=1}^{n^j} \frac{j + \mathrm{result}_i^j}{n^j * |\mathcal{C}|}$$

Classes $\mathcal{C}$ are the classes with wins, draws and losses of the individual. These classes lie in an interval whose

[1] There are several criteria to prevent games to be canceled in *interesting* situations, e.g. when a king has been checked or a piece has been captured within the last moves.

bounds are defined by the following rules: If an individual wins all its games up to class i, these results are ignored and if an individual loses all its games from class i to the highest class, these results are ignored.

For example if an individual i wins all games of classes 2, 4, and 8, and has wins, draws and losses in the classes 6 and 10, and loses all games in the classes 12 and 14, $\mathcal{C}_i$ holds the classes 6, 8 and 10. The first rule defines the lower bound of the interval (4), and the second rule defines the upper bound of the interval (12). The first rule does not hold for class 8 because in class 6 the individual has had a draw or loss.

In general, the fitness of an individual is calculated in four phases. Thus weak individuals can be dropped from fitness evaluation in the first or second phase. Fitness evaluation in phase three and four is very CPU time expensive and we try to reduce the computation time by removing inviable programs.

In phase one the individual plays two games against 2. If the individual is very weak it can be identified by the fitness function and replaced immediately. In the second phase the individual plays against 4, 6, 8 and 10. If the fitness is at least 4.5 at the end of phase two, the evaluation is continued in phase three. In phase three the individual plays 1-2 games against 2, 4, 6, 8 and 10. Successful individuals might skip games against 2 and 4. These are individuals which win each game up to strength 6 and receive good results in games against 8 and 10. In phase four games are performed against 12 and 14. Only the best individuals play in this phase, however. Games against the standard algorithm of class 12 and 14 are very expensive.

To play more than twice against class 12 (or 14) it is required to win in one of the two games before. Every draw results in 1 point, every loss in 2 points. If the individual has more than 6 (5) points it does not play any more against class 12 (14). If the individual is good enough it will play 4 times against 12 and then 3 times against 14.

3.3 Opponents of the individuals (black players)

The opponents of individuals are chess programs which can fully traverse the search tree up to a fixed depth. We use these players to calculate the fitness of an individual, by playing against fixed programs of different search depth. Fixed programs can play to a depth of 1, 2, 3, 4, 5, 6 and 7. Each of these programs defines a corresponding fitness class of 2, 4, 6, 8, 10, 12 and 14. The value for a fitness class is the search depth multiplied with 2, so that an individual which defeats an individual of class 4 but loses against an individual of class 6 can be inserted into class 5.

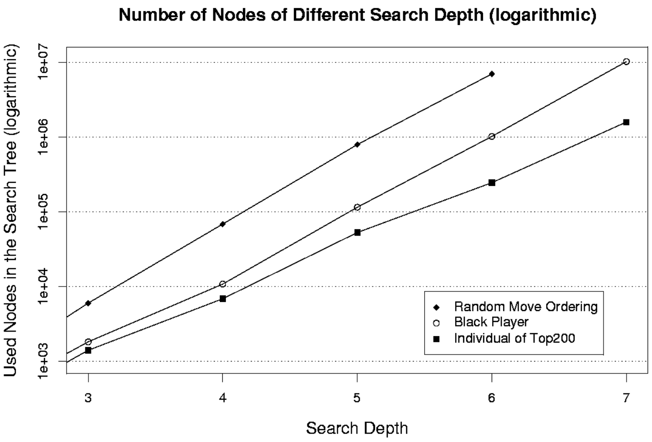

Figure 2: Average number of nodes used in the search tree for different fixed search depths. For search depth 6 the plot shows a large difference between a random move ordering and an evolved move ordering module. Even the f-negascout search algorithm requires more resources than an evolved individual. The data are average values of more than 1000 moves (from reference games).

The GP/ES individuals use a position evaluation of the same structure and the same criteria - but their weights are determined by the individual's genotype.

To reduce the number of nodes of the game tree an f-negascout algorithm [Rei89] combined with iterative deepening is performed for the black players. The f-negascout algorithm is an improved variant of alphabeta, which is the most wide-spread tree search algorithm. Iterative deepening performs a search to depth $d-1$ before searching to depth d (recursively). In addition, so-called killer moves are stored and tried first whenever possible. Killer moves are moves which result in the cut of a subtree. This means that much of the game tree can be discarded without loss of information!

4 Results

In this section we describe the current results of an ongoing evolution on the internet.

First we look at the evolved individuals and their efficiency in search. The question is: How many resources are needed by the evolved move ordering modules in case of a fixed-depth search in comparison to other move ordering strategies. Figure 2 shows the number

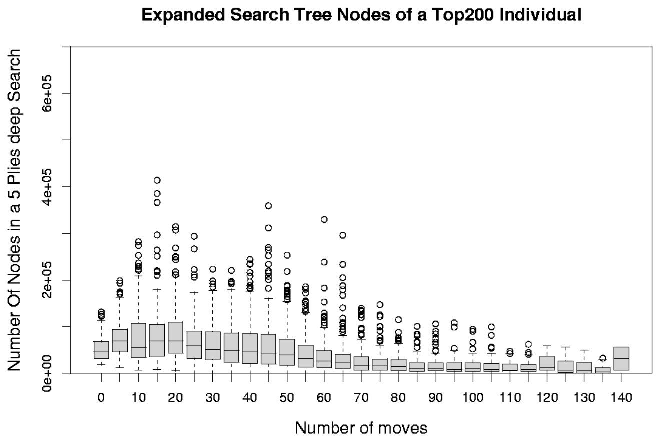

Figure 3: Box plot diagram of average number of used nodes during a game with one of an evolved individual combined with a simple fixed-depth alpha-beta algorithm, average taken over more than 5000 moves from 50 reference games. The bar in the gray boxes is the median of the data. A box represents 50 % of the data, this means that 25 % of the data lies under and over a box. The smallest and biggest *usual* values are connected with a dashed line. The circles represents outliers of the data.

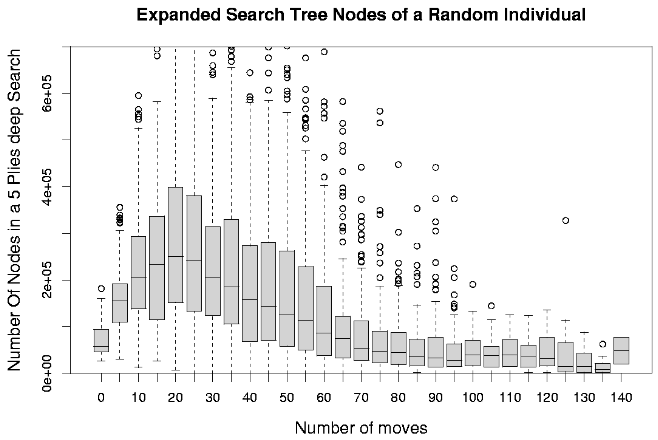

Figure 4: Box plot diagram of average number of used nodes during a game with a random individual (see also figure 3).

of nodes examined in the search tree of an alpha-beta-algorithm with a random move ordering, an evolved individual and the f-negascout algorithm (the opponent of an individual (the black player) is always an f-negascout algorithm). The figure shows that a random move ordering algorithm calculates seven million nodes with a search tree of depth 6. The f-negascout algorithm needs one million nodes. An evolved individual only needs 250,000 nodes. So evolution has managed to create individuals which perform a very efficient search through the tree.

Figures 3 and 4 show a box plot diagram investigating the number of search nodes visited by an evolved and a random move ordering module combined with a simple fixed-depth alpha-beta algorithm. The evolved individual clearly outperforms the random one. Besides, the figures show that most nodes during a game are used between ply 10 and 60.

The other aspect of the evolved chess programs is the quality of the selected moves. Currently evolution succeeded to evolve a chess playing program, with a fitness of 10.48. This means that the evolved program is better than the opponent program of fitness class 10. The fitness value was measured by a post-evaluation of best programs: An individual plays 20 games against class 8, 10 and 12, so that the fitness value is the result of 60 games. Note that the individual achieves this result by expanding on average 58,410 nodes per move in the search tree. A simple *alpha-beta* chess program needs 897,070 nodes per move for search depth of 5, which corresponds to class 10. The f-negascout algorithm which is an improved variant of *alpha-beta*, needs 120,000 nodes per move for this search depth. In other words evolution has improved the search algorithm, so that it wins by only using 50% of the resources of a f-negascout algorithm which, in turn, outperforms an *alpha-beta*-algorithm. Evolved individuals win against a simple *alpha-beta*-algorithm by using only 6% of the resources.

5 Summary and Outlook

We have shown, that it is possible to evolve chess playing individuals superior to given algorithms. At this time evolution is still going on and results are still improving.

Next we shall develop this approach by using other search algorithms as the internal structure, and by exchanging the different modules. A further feature will be that individuals will play against each other.

The ultimate goal of our approach is to beat computer programs like Deep Blue, which to this day use brute-force methods to play chess.

ACKNOWLEDGMENT

The authors gratefully acknowledge the enthusiastic support of a large group of **EvoChess** users. A list of participants is available at http://qoopy.cs.uni-dortmund.de/qoopy_e.php?page=statistik_e . All of them have helped to produce these results and to improve both **EvoChess** and the ***qoopy*** system considerably. Support has been provided by the DFG (Deutsche Forschungsgemeinschaft), under grant Ba 1042/5-2.

SUPPLEMENTARY MATERIAL

More information on the ***qoopy*** system, the EvoChess application, and experimental data are available from:

http://www.qoopy.net.

References

[Bea99] D.F. Beal. *The Nature of Minimax Search.* PhD thesis, University of Maastricht, 1999. Diss.Nr.99-3.

[BK77] J. Birmingham and P. Kent. Tree-Searching and Tree-Pruning Techniques. In M.R.B. Clarke, editor, *Advances in Computer Chess 1*, pages 89–97. Edinburgh University Press, 1977.

[BKA+00] J. Busch, W. Kantschik, H. Aburaya, K. Albrecht, R. Gross, P. Gundlach, M. Kleefeld, A. Skusa, M. Villwock, T. Vogd, and W. Banzhaf. Evolution von GP-Agenten mit Schachwissen sowie deren Integration in ein Computerschachsystem. SYS Report SYS-01/00, ISSN 0941-4568, Systems Analysis Research Group, Univ. Dortmund, Informatik, 10 2000.

[BNKF98] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, San Francisco und dpunkt verlag, Heidelberg, 1998.

[BR58] A. Bernstein and M. de V. Roberts. Computer v Chess-Player. *Scientific American*, 198:96–105, 1958.

[Far98] J. Farley. *Java Distributed Computing.* O'Reilly, 1998.

[GCK94] J. Dollimore G.F. Coulouris and T. Kindberg. *Distributed Systems, Concepts and Design.* Addison-Wesley, 2 edition, 1994.

[IHU95] H. Iida, K. Handa, and J. Uiterwijk. Tutoring Strategies in Game-Tree Search. *ICCA Journal*, 18(4):191–205, December 1995.

[KB01] W. Kantschik and W. Banzhaf. Linear-tree GP and its comparison with other GP structures. In J. F. Miller, M. Tomassini, P. Luca Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 302–312, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[KB02] W. Kantschik and W. Banzhaf. Linear-graph gp - a new gp structure. In J. F. Miller, M. Tomassini, P. Luca Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2002*, LNCS. Springer-Verlag, 2002.

[Kor97] R.E. Korf. Does Deep Blue Use Artificial Intelligence? *ICCA Journal*, 20(4):243–245, December 1997.

[NB95] Peter Nordin and Wolfgang Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 1995. Morgan Kaufmann.

[Nor94] J. P. Nordin. *A Compiling Genetic Programming System that Directly Manipulates the Machine code*, pages 311–331. MIT Press, Cambridge, 1994.

[PSPDB96] A. Plaat, J. Schaeffer, W. Pijls, and A. De Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87:255–293, 1996.

[Rei89] A. Reinfeld. An improvement of the scout tree search algorithm. *ICCA Journal*, 6(4):4–14, June 1989.

[Sch89] J. Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.

[Sch96] H-P. Schwefel. *Evolution and Optimum Seeking.* John Wiley & Sons, Inc., 1996.