GP-EndChess: Using Genetic Programming to Evolve Chess Endgame Players

Ami Hauptman and Moshe Sipper

Department of Computer Science, Ben-Gurion University, Israel
{amiha,sipper}@cs.bgu.ac.il, www.moshesipper.com

Abstract. We apply genetic programming to the evolution of strategies for playing chess endgames. Our evolved programs are able to draw or win against an expert human-based strategy, and draw against CRAFTY—a world-class chess program, which finished second in the 2004 Computer Chess Championship.

1 Introduction

Developing intelligent (or at least pseudo-intelligent) computer players of strategy games is a problem which AI research have been addressing since the field's onset. Because excelling at strategy games has often been considered to be a sign of intellectual excellence, many have felt that developing an intelligent game player would represent a big step towards developing a more generally intelligent machine [1].

The game of chess has always been viewed as an intellectual game par excellence, "a touchstone of the intellect," according to Goethe.[1] The game's complexity stems from two main sources. First, the size of the search space: after the opening phase, each player has to select the next move from approximately 50 possible moves on average. Since a single game typically consists of a few dozen moves, the search space is enormous. A second source of complexity stems from the amount of information contained in a single board. Since each player starts with 16 pieces of 6 different types, and as the board comprises 64 squares, evaluating a single board (a "position") entails elaborate computation, even without looking ahead.

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 [2]. According to Russell and Norvig [3], from 1965 to 1994 there was an almost linear increase in the strength of computer chess programs—as measured in their performance in human-rated tournaments. This increase culminated in the defeat in 1997 of Gary Kasparov—the former World Chess Champion—by IBM's special-purpose chess engine, Deep Blue (see [4])

Deep Blue, and its offspring Deeper Blue, rely mainly on brute-force methods to gain an advantage over the opponent, by traversing as deeply as possible the

[1] Some basic chess terms are explained in the appendix.

game tree [5]. Although these programs have achieved amazing performance levels, Noam Chomsky [6] has criticized this aspect of game-playing research as being “about as interesting as the fact that a bulldozer can lift more than some weight lifter.”

The number of feasible games possible (i.e., the size of the game tree), given a board configuration, is astronomical, even if one limits oneself to endgames. While endgames typically contain but a few pieces, the problem of evaluation is still hard, as the pieces are usually free to move all over the board, resulting in complex game trees—both deep and with high branching factors. Thus, we cannot rely on brute-force methods alone. We need to develop better ways to approximate the outcome of games with “smart” evaluation functions. The automated learning of evaluation functions is a promising research area if we are to produce stronger artificial players [5].

We will use the *Genetic Programming* (GP) paradigm to evolve board-evaluation functions, the basic idea of GP being to breed computer programs to solve a particular problem [7]: Start with a population of random, (usually) low-fitness individuals. Every individual plays a few games with its peers, and is assigned a score according to its level of success (or failure), i.e., its fitness. The next generation is stochastically constructed, based on individuals’ fitness values. This process repeats itself until the single best individual is returned as the solution, at the time of the evolutionary program’s termination.

This paper is organized as follows: In the next section we describe previous work on on automated methods for developing chess endgame strategies. Section 3 describes our GP setup for the evolution of chess endgame players, followed by results in Section 4. Finally, we end with concluding remarks and future work in Section 5.

2 Previous Work

GP has recently been argued to deliver ”high-return, human-competitive machine intelligence” [8]. Indeed, over the years, several strategies or agents that play games have been evolved using GP (or some other form of evolutionary algorithm).

Ferret and Martin [1] had a computer play the ancient Egyptian board game of Senet, by evolving board-evaluation functions using tournament-style fitness evaluation. Gross *et al.* [9] introduced a system that integrates GP and Evolutionary Strategies to learn to play chess. This system did not learn from scratch, but instead a “scaffolding” algorithm that could perform the task already was improved by means of evolutionary techniques.

Kendall and Whitwell [5] used evolutionary algorithms to tune evaluation-function parameters. The resulting individuals were successfully matched against commercial chess programs, but only when the lookahead for the commercial program was strictly limited.

Previous works only used simple board-evaluation functions as the building blocks for the evolutionary algorithm. For example, some typical functions used by Gross *et al.* [9] are: material values for the different pieces, penalty for bishops in initial positions, bonus for pawns in center of chessboard, penalty for doubled pawns and for backward pawns, castling bonus if this move was taken and penalty if it was not, and rook bonus for an open line or on the same line of a passed pawn. Kendall and Whitwell [5] used fewer board-evaluation functions, and focused on the weights of the remaining pieces.

3 Evolving Chess Endgame Strategies using Genetic Programming

We evolve chess endgame strategies using Koza-style GP [7]. Each individual—a LISP-like tree expression—represents a strategy, the purpose of which is to evaluate a given board configuration and generate a real-valued score. The tree's internal nodes are called *functions*, and the leaves—*terminals*. We used simple Boolean functions (AND, OR, NOT), and IF functions; terminals were used to analyze certain features of the game position. We included a large number of terminals, varying from simple ones (such as the number of moves for the player's king), to more complex features (for example, the number of pieces attacking a given piece). A full description of functions and terminals used is given in Section 3.3.

In order to better control the structure of our programs we used *Strongly Typed Genetic Programming* (STGP) [10]. This method allows the user to assign a type to a tree edge. Each function is assigned both a return type and a type for each of its arguments; each terminal is assigned a return type. Assigning more than one type per edge is also possible. All trees must be constructed according to these conventions, and only compatible types are allowed to interact. Thus, a user-defined typing scheme is imposed, although in fact all data passed within the tree consists of real numbers. We used the ECJ GP System of Luke [11].

3.1 Board evaluation

We wish to develop evaluation strategies that bear similarity to human board analysis. Thus, instead of looking deep into the game tree, we traverse less nodes, but consider each node more thoroughly. As such, our strategies use only limited lookahead.

The current player receives as input all possible board configurations reachable from the current position by making one legal move (this is quite easy to compute). After these boards are evaluated, the one that received the highest score is selected, and that move is made. Thus, an artificial player is had by combining an (evolved) board evaluator with a program that generates all possible next moves.

Although this approach has been successfully used in several game-strategy evolution scenarios (see [1]), it has not yet been applied to chess endgames.

3.2 Tree topology

Our programs play chess endgames consisting of kings, queens, and rooks (in the future we shall also consider bishops and knights). Each game starts from a different (random) legal position, in which no piece is attacked, e.g., two kings, two rooks, and two queens in a KQRKQR endgame. Although at first each program was evolved to play a different type of endgame (KRKR, KRRKRR, KQKQ, KQRKQR, etc.), which implies using different game strategies, the same set of terminals and functions was used for all types. Moreover, this set was also used for our more complex runs, in which GP chess players were evolved to play several types of endgames. Our ultimate aim is the evolution of general-purpose strategies.

Still, as most chess players would agree, playing a winning position (e.g., with material advantage) is very different than playing a losing position, or an even one (see Appendix). For this reason, each individual contains three trees: an advantage tree, an even tree, and a disadvantage tree. These trees are used according to the current status of the board. The disadvantage tree is smaller, since achieving a stalemate and avoiding exchanges requires less complicated reasoning.

3.3 Tree nodes

While evaluating a position, an expert chess player considers various aspects of the board. Some are simple, while others require a deep understanding of the game. Chase and Simon found that experts recalled meaningful chess formations better than novices [12]. This lead them to hypothesize that chess skill depends on a large knowledge base, indexed through thousands of familiar chess patterns.

We assumed that complex aspects of the game board are comprised of simpler units, which require less game knowledge, and are to be combined in some way. Our chess programs use terminals, which represent those relatively simple aspects, and functions, which incorporate no game knowledge, but supply methods of combining those aspects. As we used STGP, all functions and terminals were assigned one or more of two data types: *Float* and *Boolean*. We also included a third data type, named *Query*, which could be used as any of the former two.

The function set used included the If function, and simple Boolean functions. Although our tree returns a real number, we omitted arithmetic functions, for several reasons. First, a large part of contemporary research in the field of machine learning and game theory (in particular for perfect-information games) revolves around inducing logical rules for learning games (for example, see [13], [14] and [15]). Second, according to the players we consulted, while evaluating positions involves considering various aspects of the board, some more important than others, performing logical operations on these aspects seems natural, while mathematical operations does not. Third, we observed that numeric functions sometimes returned extremely large values, which interfered with subtle calculations. Therefore the scheme we used was a (carefully ordered) series of Boolean

Table 1. Function set of GP individual. B: Boolean, F: Float.

F=If3(B_1, F_1, F_2)	If B_1 is non-zero, return F_1, else return F_2
B=Or2(B_1, B_2)	Return 1 if at least one of B_1, B_2 is non-zero, 0 otherwise
B=Or3(B_1, B_2, B_3)	Return 1 if at least one of B_1, B_2, B_3 is non-zero, 0 otherwise
B=And2(B_1, B_2)	Return 1 only if B_1 and B_2 are non-zero, 0 otherwise
B=And3(B_1, B_2, B_3)	Return 1 only if B_1, B_2, and B_3 are non-zero, 0 otherwise
B=Smaller(B_1, B_2)	Return 1 if B_1 is smaller than B_2, 0 otherwise
B=Not(B_1)	Return 0 if B_1 is non-zero, 1 otherwise

queries, each returning a fixed value (either an ERC or a numeric terminal, see below). See Table 1 for the complete list of functions.

We developed most of our terminals by consulting several high-ranking chess players [2]. The terminal set examines various aspects of the chessboard, and may be divided into 3 groups:

1. Float values, created using the ERC (*Ephemeral Random Constants*) mechanism (see [7] for details). An ERC is chosen at random to be one of the following six values $\pm 1 \cdot \{\frac{1}{2}, \frac{1}{3}, \frac{1}{4}\} \cdot MAX$ (MAX was empirically set to 1000), and the inverses of these numbers. This guarantees that when a value is returned after some group of features has been identified, it will be distinct enough to engender the outcome.

2. Simple terminals, which analyze relatively simple aspects of the board, such as the number of possible moves for each king, and the number of attacked pieces for each player. These terminals were derived by breaking relatively complex aspects of the board into simpler notions. More complex terminals belong to the next group (see below). For example, a player should capture his opponent's piece if it is not sufficiently protected, meaning that the number of attacking pieces the player controls is greater than the number of pieces protecting the opponent's piece, and the material value of the defending pieces is equal to or greater than the player's. Adjudicating these considerations is not simple, and therefore a terminal that performs this entire computational feat by itself belongs to the next group of complex terminals.

The simple terminals comprising this second group are derived by refining the logical resolution of the previous paragraphs' reasoning: Is an opponent's piece attacked? How many of the player's pieces are attacking that piece? How many pieces are protecting a given opponent's piece? What is the material value of pieces attacking and defending a given opponent's piece? All these questions are embodied as terminals within the second group. The ability to easily embody such reasoning within the GP setup, as functions and terminals, is a major asset of GP.

[2] The highest-ranking player we consulted was Boris Gutkin, ELO 2400, International Master (see appendix), and fully qualified chess teacher.

Other terminals were also derived in a similar manner. See Table 2 for a complete list of simple terminals. Note that some of the terminals are inverted—we would like terminals to always return positive (or true) values, since these values represent a favorable position. This is why we used, for example, a terminal evaluating the player's king's *distance* from the edges of the board (generally a favorable feature for endgames), while using a terminal evaluating the *proximity* of the opponent's king to the edges (again, a positive feature).

Table 2. Simple terminals. Opp: opponent, My: player.

B=NotMyKingInCheck()	Is the player's king not being checked?
B=IsOppKingInCheck()	Is the opponent's king being checked?
F=MyKingDistEdges()	The player's king's distance form the edges of the board
F=OppKingProximityToEdges()	The player's king's proximity to the edges of the board
F=NumMyPiecesNotAttacked()	The number of the player's pieces that are not attacked
F=NumOppPiecesAttacked()	The number of the opponent's attacked pieces
F=ValueMyPiecesAttacking()	The material value of the player's pieces which are attacking
F=ValueOppPiecesAttacking()	The material value of the opponent's pieces which are attacking
B=IsMyQueenNotAttacked()	Is the player's queen not attacked?
B=IsOppQueenAttacked()	Is the opponent's queen attacked?
B=IsMyFork()	Is the player creating a fork?
B=IsOppNotFork()	Is the opponent not creating a fork?
F=NumMovesMyKing()	The number of legal moves for the player's king
F=NumNotMovesOppKing()	The number of illegal moves for the opponent's king
F=MyKingProxRook()	Proximity of my king and rook(s)
F=OppKingDistRook()	Distance between opponent's king and rook(s)
B=MyPiecesSameLine()	Are two or more of the player's pieces protecting each other?
B=OppPiecesNotSameLine()	Are two or more of the opponent's pieces protecting each other?
B=IsOppKingProtectingPiece()	Is the opponent's king protecting one of his pieces?
B=IsMyKingProtectingPiece()	Is the player's king protecting one of his pieces?

3. Complex terminals. These are terminals that check the same aspects of the board a human player would. Some prominent examples include: the terminal OppPieceCanBeCaptured considering the capture of a piece; checking if the current position is a draw, a mate, or a stalemate (especially important for non-even boards); checking if there is a mate in one or two moves (this is the most complex terminal); the material value of the position; comparing the material value of the position to the original board—this is important since it is easier to consider change than to evaluate the board in an absolute manner. See Table 3 for a full list of complex terminals.

Since some of these terminals are hard to compute, and most appear more than once in the individual's trees, we used a memoization scheme to save time [16]: After the first calculation of each terminal, the result is stored, so that further calls to the same terminal (on the same board) do not repeat the calculation. Memoization greatly reduced the evolutionary run-time.

Table 3. Complex terminals. Opp: opponent, My: player. Some of these terminals perform lookahead, while others compare with the original board.

F=EvaluateMaterial()	The material value of the board
B=IsMaterialIncrease()	Did the player capture a piece?
B=IsMate()	Is this a mate position?
B=IsMateInOne()	Can the opponent mate the player after this move?
B=OppPieceCanBeCaptured()	Is it possible to capture one of the opponent's pieces without retaliation?
B=MyPieceCannotBeCaptured()	Is it not possible to capture one of the player's pieces without retaliation?
B=IsOppKingStuck()	Do all legal moves for the opponent's king advance it closer to the edges?
B=IsMyKingNotStuck()	Is there a legal move for the player's king that advances it away from the edges?
B=IsOppKingBehindPiece()	Is the opponent's king two or more squares behind one of his pieces?
B=IsMyKingNotBehindPiece()	Is the player's king not two or more squares behind one of my pieces?
B=IsOppPiecePinned()	Is one or more of the opponent's pieces pinned?
B=IsMyPieceNotPinned()	Are all the player's pieces not pinned?

3.4 Fitness evaluation

As we used a competitive evaluation scheme, the fitness of an individual was determined by its success against its peers. We used the random-2-ways method (see [17] for full details), in which each individual plays against a fixed number of randomly selected peers (typically 5). Each of these encounters entails a fixed number of games, each starting from a randomly generated position. Since random starting positions can sometimes be uneven (for example, allowing the starting player to attain a capture position), every starting position was played twice, each player playing both black and white. This way a better starting position could benefit both players and the tournament was less biased. In addition, in each encounter several games were played, to further reduce the element of chance.

The score for each game is derived from the outcome of the game. Players that manage to mate their opponents receive more points than those that achieve only a material advantage. Draws are rewarded by a score of low value and losses entail no points at all.

The final fitness for each player is the sum of all points earned in the entire tournament for that generation. We used the standard reproduction, crossover, and mutation operators, as in [7]. The major parameters were: population size – 80, generation count – between 150 and 250, reproduction probability – 0.35, crossover probability – 0.5, and mutation probability – 0.15 (including ERC).

4 Results

We conducted several experiments to test our evolving chess players. The scoring method was based on the one used in chess tournaments: victory—1 point, draw—$\frac{1}{2}$ point, loss—0 points. In order to better differentiate our players, we rewarded $\frac{3}{4}$ points for a material advantage (without mating the opponent).

The final score is the sum of all scores a player has received, divided by the number of games. This way, a player who always mates its opponent will receive a perfect score of 1. The score for a player that played against an opponent of comparable strength (where most games end in a draw), is 1/2 on average.

4.1 Experiment 1: Competing against a human-defined strategy

As noted above, we developed most of our terminals by consulting several high-ranking chess players. In order to evaluate our system, we wished to test our evolved strategies against some of these players. Because we needed to play thousands of games in every run, these could not be conducted manually, but instead we programmed an optimal strategy, based on the guidance from the players we consulted. We wrote this evaluation program using the functions and terminals of our GP system.

During evolution, our chess programs competed against each other. However, every 10 generations the best individual was extracted and pitted in a 150-game tournament against the human-defined strategy. The results are depicted in Figure 1, showing runs for KRKR, KRRKRR and KQRKQR, respectively.

These figures clearly show that starting from a low level of performance, chess players evolve to play as good as high-ranking humans for all groups of endgames, in one case even going beyond a draw to win (KQRKQR endgame, where a high score of 0.63 was attained). Improvement was rapid, typically requiring only a few dozens of generations (about 15 hours on a standard workstation).

4.2 Experiment 2: Competing against a world-class chess engine

Having attained good results against a human-defined strategy based on expert chess players, we went one step further and competed against a highly powerful

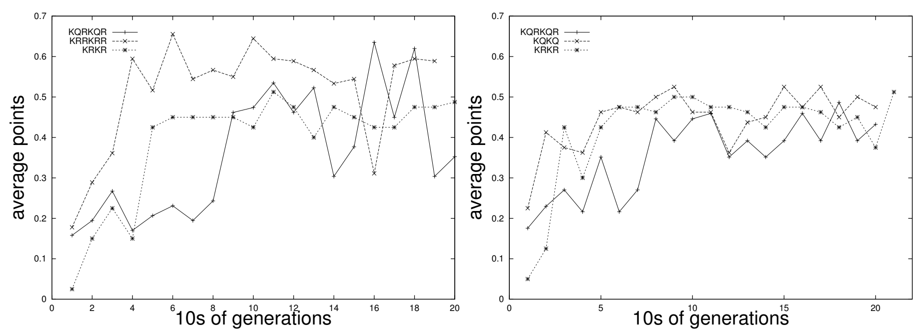


Fig. 1. Left: Results against a strategy defined by a chess Master. The three graphs show the average score over time of the best run of 50 runs carried out, for three types of endgames: KRKR, KRRKRR, KRQKRQ. A point represents the score of the best individual at that time, pitted in a 150-game tournament against the human-defined strategy. Right: Results against CRAFTY. The three graphs show the average score over time of the best run of 15 runs carried out, for three types of endgames: KRKR, KQKQ, KQRKQR. A point represents the score of the best individual at that time, pitted in a 50-game tournament against CRAFTY.

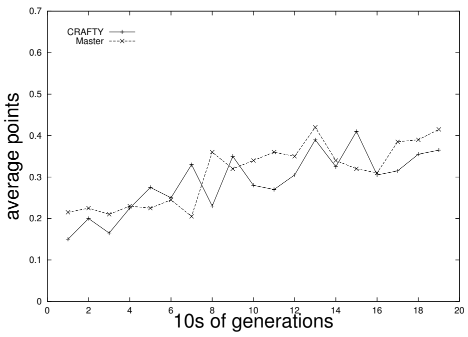
chess engine. For this task, we used the CRAFTY engine (version 19.01) by Hyatt [3]. CRAFTY is a state-of-the-art chess engine, using a typical brute-force approach, with a fast evaluation function, NegaScout search, and all the standard enhancements [18]. CRAFTY finished *second* at the 12th World Computer Speed Chess Championship, held in Bar-Ilan University on July 2004. According to `www.chessbase.com`, CRAFTY has a rating of 2614 points, which places it at the human Grandmaster level. CRAFTY is thus, undoubtedly, a worthy opponent.

As expected, CRAFTY proved to be a formidable opponent, constantly mating the GP opponent at early generations. However, during the process of evolution, substantial improvement was observed to occur. As shown in Figure 1, our program managed to achieve near-draw scores, even for the complex KQRKQR endgame. Considering our evolved 2-lookahead programs' competing against a world-class chess player, our method seems quite viable and promising.

4.3 Experiment 3: Multiple-endgame runs

Aiming for general-purpose strategies, this third experiment involved the playing of one game *of each type* (rather than a single type)—both during evolution and in the test tournaments. Evolved players were pitted against the Master-defined strategy and CRAFTY. As can be seen in Figure 2, near-draw scores were achieved under these conditions as well. We observed that performance kept improving and are confident that it would continue doing so with added computational resources.

[3] CRAFTY's source code is available at `ftp://ftp.cis.uab.edu/pub/hyatt`.



	%Wins	%Advs	%Draws
Master	6.00	2.00	68.00
CRAFTY	2.00	4.00	72.00

Fig. 2. Left: Results for multiple-endgame runs—wherein all endgames were used during evolution—against both CRAFTY and the Master-defined strategy. Each graph shows the average score over time of the best run of 20 runs carried out. A point represents the score of the best individual at that time, pitted in a 50-game tournament against CRAFTY, or a 150-game against the Master. Right: Percent of wins, advantages, and draws for best tournament of run (i.e., fitness peak of graph).

5 Concluding Remarks and Future Work

We presented a method by which chess endgame players may be evolved to successfully hold their own against excellent opponents. One of the major prima facie problems with our scheme is its complexity, as evidenced by Tables 1, 2, and 3. In the time-honored tradition of computer science, we argue that this is not a bug but rather a feature—to be more precise, a somewhat overlooked feature of genetic programming.

We believe that GP represents a viable means to automatic programming, and perhaps more generally to machine intelligence, in no small part due to its being *cooperative with humans*. More than many other adaptive search techniques (e.g., genetic algorithms, artificial neural networks, ant algorithms), the GPer, owing to GP's representational affluence and openness, is better positioned to imbue the genomic language with his or her own intelligence. While artificial-intelligence (AI) purists may wrinkle their noses at this, taking the AI-should-emerge-from-scratch stance, we argue that a more practical path to AI involves man-machine cooperation. GP is a forerunning candidate for the 'machine' part.

We did not design our genome (Tables 1, 2, 3) in one fell swoop, but rather through an incremental, interactive process, whereby man (represented by the humble authors of this paper) and machine (represented by man's university's computers) worked hand-in-keyboard. To wit, we began our experimentation with small sets of functions and terminals, which were revised and added upon through our examination of evolved players and their performance and through consultation with high-ranking chess players. GP's design cooperativeness, often overlooked, is thus perhaps one of its major boons.

In addition, the number of terminals we used is small, compared to the number of patterns used by chess experts when evaluating a position: According to Simone and Gilmartin [19] this number is close to 100,000. Since most pattern-based programs nowadays are considered to be far from competitive (see [13]), the results we obtained may imply that we have made a step towards developing a program that has more in common with the way humans play chess.

In the future we aim to follow a number of paths: 1) improve the evolved programs' performance against the above and other endgames, 2) branch out beyond endgames, and 3) analyze the evolved cognition as to its resemblance and difference from human cognition.

Appendix: Brief Glossary of Basic Chess Terms

(More at `www.arkangles.com`)

Material value. Sum of all numerical values (see Point Count) for player's pieces (which are given positive values), and the opponent's (negative values).

Point count. Queen is worth 9 points, rooks – 5 points, bishops – 3 or 3.25 points, knights – 3 points, and pawns – 1 point. King is typically assigned an infinite value.

Advantage. When the current configuration of the game favors one side over another; includes: material advantage, permanent advantage, positional advantage, and temporary advantage.

Capture. Moving a piece to a square occupied by an enemy piece, thereby removing the enemy piece from the board.

Fork. A form of double attack where one piece threatens two enemy pieces at the same time. In a triple fork, three enemy pieces are threatened.

Endgame. The final phase of the game when there are few pieces left on the board. Endgame abbreviations are used to represent the remaining pieces (e.g., KRKR).

Ranking chess players. Both professional and amateur chess players may obtain a nationally (or internationally) recognized numerical rating (sometimes referred to as ELO). Independently, professional players may earn titles, gained in special official tournaments, in which title-holders must participate. A title, once earned, is the player's for life, while the point rating can oscillate. The lowest international title is Master (usually not gained before the player reaches ELO 2200). The highest titles are International Master (IM) and Grandmaster (GM). In 2003 there were only about 3000 IMs and GMs worldwide.

The research was partially supported by the Lynn and William Frankel Fund for Computer Science.

References

1. Ferrer, G.J., Martin, W.N.: Using genetic programming to evolve board evaluation functions for a board game. In: 1995 IEEE Conference on Evolutionary Computation. Volume 2., Perth, Australia, IEEE Press (1995) 747–752

2. Bernstein, A., de V. Roberts, M.: Computer versus Chess-Player. Scientific American **198** (1958) 96–105
3. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice-Hall, Englewood Cliffs , NJ (1995)
4. DeCoste, D.: The Significance of Kasparov vs Deep Blue and the Future of Computer Chess. ICCA Journal **21** (1998) 33–43
5. Kendall, G., Whitwell, G.: An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In: Proceedings of the 2001 Congress on Evolutionary Computation CEC2001, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, IEEE Press (2001) 995–1002
6. Chomsky, N.: Language and Thought. Moyer Bell, Wakefield, RI (1993)
7. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)
8. Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, W., Yu, J., Lanza, G.: Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, Norwell, MA (2003)
9. Gross, R., Albrecht, K., Kantschik, W., Banzhaf, W.: Evolving chess playing programs. In Langdon, W.B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M.A., Schultz, A.C., Miller, J.F., Burke, E., Jonoska, N., eds.: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, Morgan Kaufmann Publishers (2002) 740–747
10. Montana, D.J.: Strongly typed genetic programming. Evolutionary Computation **3** (1995) 199–230
11. Luke, S.: ECJ: A Java-based Evolutionary Computation and Genetic Programming Research System. (2000) http://www.cs.umd.edu/projects/plus/ec/ecj/.
12. Charness, N.: Expertise in chess: The balance between knowledge and search. In Ericsson, K.A., Smith, J., eds.: Toward a general theory of Expertise: Prospects and limits. Cambridge University Press, Cambridge (1991)
13. Fürnkranz, J.: Machine learning in computer chess: The next generation. International Computer Chess Association Journal **19** (1996) 147–161
14. Bonanno, G.: The logic of rational play in games of perfect information. Papers 347, California Davis - Institute of Governmental Affairs (1989) available at http://ideas.repec.org/p/fth/caldav/347.html.
15. Bain, M.: Learning Logical Exceptions in Chess. PhD thesis, University of Strathclyde, Glasgow, Scotland (1994)
16. Abelson, H., Sussman, G.J., with J. Sussman: Structure and Interpretation of Computer Programs. Second edn. The MIT-Press (1996)
17. Panait, L.A., Luke, S.: A comparison of two competitive fitness functions. In Langdon, W.B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M.A., Schultz, A.C., Miller, J.F., Burke, E., Jonoska, N., eds.: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, Morgan Kaufmann Publishers (2002) 503–511
18. Jiang, A.X., Buro, M.: First experimental results of ProbCut applied to chess. In: Proceedings of 10th Advances in Computer Games Conference, Kluwer Academic Publishers, Norwell, MA (2003) 19–32
19. Simon, H., Gilmartin, K.: A simulation of memory for chess positions. Cognitive Psychology **5** (1973) 29–46