Evolution of an Efficient Search Algorithm for the Mate-In-N Problem in Chess

Conference Paper · June 2007

DOI: 10.1007/978-3-540-71605-1_8 · Source: DBLP

2 authors:

Ami Hauptman
Sapir Academic College
13 PUBLICATIONS **226** CITATIONS
SEE PROFILE

Moshe Sipper
Ben-Gurion University of the Negev
264 PUBLICATIONS **5,169** CITATIONS
SEE PROFILE

Evolution of an Efficient Search Algorithm for the Mate-In-N Problem in Chess

Ami Hauptman and Moshe Sipper

Dept. of Computer Science, Ben-Gurion University, Beer-Sheva, Israel

Abstract. We propose an approach for developing efficient search algorithms through genetic programming. Focusing on the game of chess we evolve entire game-tree search algorithms to solve the *Mate-In-N* problem: find a key move such that even with the best possible counterplays, the opponent cannot avoid being mated in (or before) move N. We show that our evolved search algorithms successfully solve several instances of the Mate-In-N problem, for the hardest ones developing 47% less game-tree nodes than CRAFTY—a state-of-the-art chess engine with a ranking of 2614 points. Improvement is thus not over the basic alpha-beta algorithm, but over a world-class program using all standard enhancements.

1 Introduction

Artificial intelligence for board games is widely based on developing deep, large game trees. In a two-player game, such as chess or go, players move in turn, each trying to win against the opponent according to specific rules. The course of the game may be modeled using a structure known as an adversarial game tree (or simply game tree), in which nodes are positions in the game and edges are moves [10]. The complete game tree for a given game is the tree starting at the initial position (the root) and containing all possible moves (edges) from each position. *Terminal nodes* represent positions where the rules of the game determine whether the result is a win, a draw, or a loss.

When the game tree is too large to be generated completely, only a partial tree (called a search tree) is generated instead. This is accomplished by invoking a *search algorithm*, deciding which nodes are to be developed at any given time, and when to terminate the search (typically at non-terminal nodes due to time constraints) [17]. During the search, some nodes are evaluated by means of an *evaluation function* according to given heuristics. This is done mostly at the leaves of the tree. Furthermore, search can start from any position, and not just the beginning of the game.

In general, there is a tradeoff between *search* and *knowledge*, i.e., the amount of search (development of the game tree) carried out and the amount of knowledge in the leaf-node evaluator. Because deeper search yields better results but takes exponentially more time, various techniques are used to guide the search, typically pruning the game tree. While some techniques are more generic and domain independent, such as alpha-beta search [14] and the use of hash tables

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 78–89, 2007.

(i.e., transposition and refutation) [2], other methods rely on domain-specific knowledge. For example, quiescence search [12] relies on examining capture move sequences in chess (and relevant parts of the game tree) more thoroughly than other moves. This is derived from empirical knowledge regarding the importance of capture moves. Theoretically speaking, perfect domain knowledge would render the search futile, as is the case in solved endgames in chess [3]. However, constructing a full knowledge base for difficult games such as chess is still far from attainable.

While state-of-the-art chess engines integrate both search and knowledge the scale is tipped towards generic search enhancements, rather than knowledge-based reasoning [15]. In this paper we evolve a search algorithm, allowing evolution to "balance the scale" between search and knowledge. The *entire search algorithm*, based on building blocks taken from existing methods, is subject to evolution. Some such blocks are representative of queries performed by strong human players, allowing evolution to find ways of correctly integrating them into the search algorithm. This was previously too difficult a task to be done without optimization algorithms, as evidenced by the authors of Deep Blue [5]. Our results show that the number of search-tree nodes required by the evolved search algorithms can be greatly reduced in many cases.

2 Previous Work

Our interest in this paper is in evolving a search algorithm by means of genetic programming. We found little work in the literature on the evolution of search algorithms. Brave [4] compared several genetic-programming methods on a planning problem involving tree search, in which a goal node was to be found in one of the leaves of a full binary tree of a given depth. While this work concluded that genetic programming with recursive automatically defined functions (ADFs) outperforms other methods and scales well, the problems he tackled were specifically tailored, and not real-world problems.

Hong *et al.* applied evolutionary algorithms to game search trees, both for single-player games [10], and for two-player games [11]. Each individual in the population encoded a path in the search tree, and the entire population was evolved to solve single game positions. Their results show considerable improvement over the minimax algorithm, both in speed and accuracy, which seems promising. However, their system required that search trees have the same number of next-moves for all positions. Moreover, they did not tackle real-world games.

Gross *et al.* [7] evolved search for chess players using an alpha-beta algorithm as the kernel of an individual which was enhanced by genetic-programming and evolution-strategies modules. Thus, although the algorithmic skeleton was predetermined, the more "clever" parts of the algorithm (such as move ordering, search cut-off, and node evaluation) were evolved. Results showed a reduction in the number of nodes required by alpha-beta to an astonishing 6 percent. However, since the general framework of the algorithm was determined beforehand, the full power of

evolution was not tapped. Moreover, there is no record of successfully competing against commercial programs, which are known to greatly outperform alpha-beta (with standard enhancements) on specific game-playing tasks.

Previously [9], we evolved chess endgame players using genetic programming, which successfully competed against CRAFTY, a world-class chess program (rated at 2614 points, which places it at the human Grandmaster level), on various endgames. Deeper analysis of the strategies developed [8] revealed several important shortcomings, most of which stemmed from the fact that they used deep knowledge and little search (typically, they developed only *one* level of the search tree). Simply increasing the search depth would not solve the problem, since the evolved programs examine each board very thoroughly, and scanning many boards would increase time requirements prohibitively.

And so we turn to evolution to find an optimal way to overcome this problem: How to add more search at the expense of less knowledgeable (and thus less time-consuming) node evaluators, while attaining better performance. In the experiment described herein *we evolved the search algorithm itself*. While previous work on evolving search used either a scaffolding algorithm [7], or searching in toy problems [4], we present a novel approach of *evolving the entire search algorithm*, based on building blocks taken from existing methods, integrating knowledge in the process, and applying our results to a real-world problem.

We consider all endgames, as opposed to our previous set of experiments [9], in which we only considered a limited subset of endgames. However, an important limit has been imposed: Since efficiently searching the entire game (or endgame) tree is an extremely difficult task, we limited ourselves *for now* to searching only for game termination (or mate positions) of varying tree depths, as explained in the next section.

3 The Mate-In-N Problem

The *Mate-In-N* problem in chess is defined as finding a key move such that even with the best possible counterplays, the opponent cannot avoid being mated in (or before) move N, where N counts only the player's moves and not the opponent's. This implies finding a subtree of forced moves, leading the opponent to defeat in $(2N - 1)$ plies (actually, $2N$ plies, since we need an additional ply to verify a mate). Typically, for such tactical positions (where long forcing move sequences exist), chess engines search much more thoroughly, using far more resources. For example, Deep Blue searches at roughly half the usual speed in such positions [5].

Allegedly, solving the mate problem may be accomplished by performing exhaustive search. However, because deep search is required when N is large, the number of nodes grows exponentially, and a full search is next to impossible. For example, finding a mate-in-5 sequence requires searching 10 or 11 plies, and more than $2 * 10^{10}$ nodes. Of course, advanced chess engines search far less nodes due to state-of-the-art search enhancements, as can be seen in Table 1. Still, the problem remains difficult.

Table 1. Number of nodes required to solve the Mate-in-N problem by CRAFTY—a top machine player—averaged over our test examples. Depth in plies (half-moves) needed is also shown.

Mate-in	1	2	3	4	5
Depth in plies	2	4	6	8	10
Nodes developed	600	7K	50K	138K	1.6M

A basic algorithm for solving the Mate-In-N problem through exhaustive search is shown in Figure 1. First, we check if the search should terminate: successfully, if the given board is indeed a mate; in failure, if the required depth was reached and no mate was found. Then, for each of the player's moves we perform the following check: if, after making the move, *all* the opponent's moves lead (recursively) to Mate-in-$(N-1)$ or better (procedure $CheckOppTurn$), the mating sequence was found, and we return $true$. If not, we iterate on all the player's other moves. If no move meets the condition, we return $false$.

```
MATE-IN-N?(board, depth)

procedure CHECKOPPTURN(board, depth)
 // Check if all opponent's moves lead to Mate-in-(N-1)
 for each oppmove ∈ GETNEXTMOVES(board)
       { MAKEMOVE(board, oppmove)
       { result ← MATE-IN-N?(board, depth − 1)
   do  { UNDOMOVE(board, oppmove)
       { if not result
       {   then return (false)
 return (true)

main
 if ISMATE(board)
   then return (true)
 if depth = 0
   then return (false)
 for each move ∈ GETNEXTMOVES(board)
       { MAKEMOVE(board, move)
       { result ← CHECKOPPTURN(board, depth)
   do  { UNDOMOVE(board, move)
       { if result
       {   then return (true)
 return (false)
```

Fig. 1. A basic algorithm for solving the Mate-In-N problem through exhaustive search

This algorithm has much in common with several algorithms, including alpha-beta search and proof-number (pn) search [1]. However, as no advanced schemas (for example, move-ordering or cutoffs) are employed here, the algorithm becomes infeasible for large values of N.

In the course of our experiments we broke the algorithmic skeleton into its component building blocks, and incorporated them, along with other important elements, into the evolving genetic-programming individuals.

4 Evolving Mate-Solving Algorithms

We evolve our mate-solving search algorithms using Koza-style genetic programming [13]. In genetic programming we evolve a *population* of individual LISP expressions, each comprising *functions* and *terminals*. Since LISP programs may be readily represented as program trees, the functions are internal nodes and the terminals are leaves. (NB: There are two types of tree involved in our work: game search tree and the tree representing the LISP search algorithm.)

Since we wish to develop intelligent—rather than exhaustive—search, our board evaluation requires special care. Human players never develop the entire tree, even when this is possible. For example, Mate-in-1 problems are typically solved by only developing checking moves, and not all possible moves (since non-checking moves are necessarily non-mating moves, there is no point in looking into them). As human players only consider 2–3 boards per second yet still solve deep Mate-in-N problems fast (for example, Grandmasters often find winning combinations more than 10 moves ahead in mere seconds), they rely either on massive pattern recognition or on intelligent pruning, or both [6].

Thus, we evolved our individuals (game search-tree algorithms) accordingly, following these guidelines: 1) Individuals only consider moves adhering to certain conditions (themselves developed by evolution). 2) The amount of lookahead is left to the individual's discretion, with fitness penalties for deep lookahead (to avoid exhaustive search). Thus, we also get evolving lookahead. 3) Development of the game tree is asymmetrical. This helps with computation since we do not need to consider the same aspects for both players' moves. 4) Each node examined during the search is individually considered according to game knowledge, and move sequence may be developed to a different depth.

4.1 Basic Program Architecture

Our individuals receive a chessboard as input, and return a real-valued score in the range $[-1000.0, 1000.0]$, indicating the likelihood of this board leading to a mate (higher is more likely). A representation issue is whether to evolve boards returning scores or moves (allowing to return no move to indicate no mate has been found). An alternative approach might be evolving the individuals as move-ordering modules. However, the approach we took was both more versatile and reduced the overhead of move comparison by the individual—instead of comparing moves by the genetic programming individual, the first level of the search is done by a separate module. An evolved program thus receives as input all possible board configurations reachable from the current position by making one legal move. After all options are considered by the program, the move that received the highest score is selected, and compared to the known solution for fitness purposes (described in Section 4.3).

4.2 Functions and Terminals

We developed most of our terminals and functions by consulting several high-ranking chess players.

Domain-specific functions. These functions are listed in Table 5. Note that domain-specific functions typically examine if a move the player makes adheres to a given condition, which is known to lead to a mate in various positions. If so, this move is made, and evaluation continues. If not, the other child is evaluated. Also, a more generic function, namely $IfMyMoveExistsSuchThat$, was included to incorporate other (possibly unknown) considerations in making moves by the player. All functions undo the moves they make after evaluation of their children is completed. Since some functions are only appropriate in MAX nodes (player's turn), and others in MIN nodes (opponent's turn), some functions in the table were only used at the relevant levels. Other functions, such as $MakeBestMove$, behave differently in MAX nodes and in MIN nodes.

Sometimes functions that consider a player's move are called when it is the opponent's turn. In this case we go immediately to the false condition (without making a move). This solution was simpler than, for example, defining a new set of return types. Some of these functions appear as terminals also, to allow considerations to be made while it is the opponent's turn.

Generic functions appear in Table 4. As in [9], these domain-independent functions were included to allow logic and some numeric calculations.

Chess terminals, some of which were also used used in [9], are shown in Table 6. Here, several mating aspects of the board, of varying complexity levels, are considered. From the number of possible moves for the opponent's king, through checking if the player creates a fork attacking the opponent's king, to one of the most important terminals—$IsMateInOneOrLess$. This terminal is used to allow the player to easily identify very close mates. Of course, repeated applications of this terminal at varying tree depths might have solved our problem but this alternative was not chosen by evolution (as shown below). Material value and material change are considered, to allow the player to make choices involving not losing pieces.

Mate terminals, which were specifically constructed for this experiment, are shown in Table 3. Some of these terminals resemble those from the function set, to allow building different calculations with similar (important) units.

4.3 Fitness

In order to test our individuals and assign fitness values we used a pool of 100 Mate-in-N problems of varying depths (i.e., values of N). The easier 50 problems ($N = 1..3$) were taken from Polgar's Book [16], while those with larger Ns ($N \geq 4$) were taken from various issues of the Israeli Chess Federation Newsletter (`http://www.chess.org.il`). All problems were solved offline by CRAFTY.

Special care was taken to ensure that all of the deeper problems could not be solved trivially (e.g., if there are only a few pieces left on the board, or when the opponent's king can be easily pushed towards the edges). We used CRAFTY's

feature of counting nodes in the game tree and made sure that the amount of search required to solve all problems was close to the average values given in Table 1 (we saved this value for each problem, to use for scoring purposes).

The fitness score was assigned according to an individual (search algorithm's) success in solving a random sample of problems of all depths, taken from the pool (sample size was 5 per N). For each solution, the score was calculated using the formula:

$$fitness = \sum_{i=1}^{s \cdot MaxN} Correctness_i \cdot 2^{N_i} \cdot Boards_i$$

with the following specifications:

- i ,N, and s are the problem instance, the depth, and the sample size, respectively. $MaxN$ is the maximal depth we worked with (currently 5).
- $Correctness_i \in [0, 1]$ represents the percentage of the correctness of the move. If the correct piece was selected, this score is 0.5^d, where d is the distance (in squares) between the correct destination and the chosen destination for the piece. If the correct square was attacked but with the wrong piece, it was 0.1. In the later stages of each run (after more than 75% of the problems were solved by the best individuals), this factor was only 0.0 or 1.0.
- N_i is the depth of the problem. Since for larger Ns, finding the mating move is exponentially more difficult, this factor also increases exponentially.
- $Boards_i$ is the number of boards examined by CRAFTY for this problem, divided by the number examined by the individual[1]. For small Ns, this factor was only used at later stages of evolution.

We used the standard reproduction, crossover, and mutation operators, as in [13]. We experimented with several configurations finally setting on: population size – between 70 and 100, generation count – between 100 and 150, reproduction probability – 0.35, crossover probability – 0.5, and mutation probability – 0.15 (including ERC—Ephemeral Random Constants). The relatively small population size helped to maintain shorter running times, although possibly more runs were needed to attain our results.

5 Results

After each run we extracted the top individual (i.e., the one that obtained the best fitness throughout the run) and tested its performance with a separate problem set (the *test set*), containing 10 problems per each depth, not encountered before. The results from the ten best runs show that all problems up to $N = 4$ were solved completely in most of the runs, and most $N = 5$ problems were also solved.

[1] In order to better control running times, if an individual examined more than 1.5 the boards examined by CRAFTY, the search tree was truncated, although the returned score was still used.

Table 2. Number of search-tree nodes developed to solve the Mate-in-N problem by CRAFTY, compared to the number of nodes required by our best evolved individual from over 20 runs. Values shown are averaged over the test problems. As can be seen, for the hardest problems ($N = 5$) our evolved search algorithm obtains a 47% reduction in developed nodes.

Mate-in	1	2	3	4	5
CRAFTY	600	7K	50K	138K	1.6M
Evolved	600	2k	28k	55K	850k

Table 3. Mate terminal set of an individual program in the population. Opp: opponent, My: player. "Close" means 2 squares or less.

B=IsNextMoveForced()	Is the opponent's next move forced (only 1 possible)?
F=IsNextMoveForcedWithKing()	Opponent must move its king
B=IsPinCloseToKing()	Is an opponent's piece pinned close to the king
F=NumMyPiecesCanCheck()	Number of the player's pieces capable of checking the opponent
B=DidNumAttackingKingIncrease()	Did the number of pieces attacking the opponent's king's area increase after last move?
B=IsPinCloseToKing()	Is an opponent's piece pinned close to the king
B=IsDiscoveredCheck()	Did the last move clear the way for another piece to check?
B=IsDiscoveredProtectedCheck()	Same as above, only the checking piece is also protected

Table 4. Domain-independent function set of an individual program in the population. B: Boolean, F: Float.

F=If3(B, F_1, F_2)	If B 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)	Return 0 if B is non-zero, 1 otherwise
B=Or2(B_1, B_2)	Return 1 if at least one of B_1, B_2 is non-zero, 0 otherwise

Due to space restrictions we do not present herein a detailed analysis of runs but focus on the most important issue, namely, *the number of search-tree nodes developed by our evolved search algorithms*. As stated above, mates can be found with exhaustive search and little knowledge, but the number of nodes would be

Table 5. Domain-specific function set of an individual program in the population. B: Boolean, F: Float. Note: all move-making functions undo the move when the function terminates.

F=IfMyMoveExistsSuchThat(B, F_1, F_2)	If after making one of my moves B is true, make that move and return F_1, else return F_2
F=IfForAllOpponentMoves(B, F_1, F_2)	If after making each of the opponent's moves B is true, make an opponent's move and return F_1, else return F_2
F=MakeBestMove(F)	Make all moves possible, evaluate the child (F) after each move, and return the maximal (or minimal) value of all evaluations
F=MakeAnyOrAllMovesSuchThat(B, F_1, F_2)	Make all possible moves (in opp turn) or any move (my turn), and remember those for which B was true. Evaluate F_1 after making each of these moves, and return the best result. If no such move exists, return F_2.
F=IfExistsCheckingMove(F_1, F_2)	If a checking move exists, return F_1, else return F_2
F=MyMoveIter(B_1,B_2,F_1,F_2)	Find a player's move for which B_1 is true. Then, develop all opponent's moves, and check if for all, B_2 is true. If so, return F_1, else return F_2
F=IfKingMustMove(F_1, F_2)	If opponent's king must move, make a move, and return F_1, else return F_2
F=IfCaptureCloseToKingMove(F_1, F_2)	If player can capture close to king, make that move and return F_1, else return F_2
F=IfPinCloseToKingMove(F_1, F_2)	If player can pin a piece close to opponent's king, make that move and return F_1, else return F_2
F=IfAttackingKingMove(F_1, F_2)	If player can move a piece into a square attacking the area near opponent's king, make that move and return F_1, else return F_2
F=IfClearingWayMove(F_1, F_2)	If player can move a piece in such a way that another piece can check next turn, return F_1, else return F_2
F=IfSuicideCheck(B,F_1, F_2)	If player can check the opponent's king while losing its own piece and B is true, evaluate F_1, else return F_2

prohibitive. Table 2 presents the number of nodes examined by our best evolved algorithms compared with the number of nodes required by CRAFTY. As can be seen, a reduction of 47% is achieved for the most difficult case ($N = 5$). Note that *improvement is not over the basic alpha-beta algorithm, but over a world-class program using all standard enhancements.*

Table 6. Chess terminal set of an individual program in the population. Opp: opponent.

B=IsCheck()	Is the opponent's king being checked?
F=OppKingProximityToEdges()	The player's king's proximity to the edges of the board
F=NumOppPiecesAttacked()	The number of the opponent's attacked pieces close to its king
B=IsCheckFork()	Is the player creating a fork attacking the opponent's king?
F=NumNotMovesOppKing()	The number of illegal moves for the opponent's king
B=NumNotMovesOppBigger()	Has the number of illegal moves for the opponent's king increased?
B=IsOppKingProtectingPiece()	Is the opponent's king protecting one of its pieces?
F=EvaluateMaterial()	The material value of the board
B=IsMaterialChange()	Was the last move a capture move?
B=IsMateInOneOrLess()	Is the opponent in mate, or can be in the next turn?
B=IsOppKingStuck()	Do all legal moves for the opponent's king advance it closer to the edges?
B=IsOppPiecePinned()	Is one or more of the opponent's pieces pinned?

6 Concluding Remarks

Our results show that the number of search-tree nodes required to find mates may be significantly reduced by evolving a search algorithm with building blocks that provide a-priori knowledge. This is reminiscent of human thinking, since human players survey very few boards (typically 1-2 per second) but apply knowledge far more complex than any artificial evaluation function. On the other hand, even strong human players usually do not find mates as fast as machines (especially in complex positions). Our evolved players are both fast *and* accurate.

GP-trees of our best evolved individuals were quite large, and difficult to analyze. However, from examining the results it is clear that the best individuals' search was efficient, and thus domain-specific functions and terminals play an important role in guiding search. This implies that much "knowledge" was incorporated into stronger individuals, although it would be difficult to quantify it.

The depths (Ns) we dealt with are still relatively small. However, as the notion of evolving the entire search algorithm is new, we expect to achieve better results in the near future. Our most immediate priority is generalizing our results to larger values of N, a task we are currently working on. In the short term we would like to evolve a general Mate-In-N module, which could replace a chess engine's current module, thereby increasing its rating—no mean feat where top-of-the-line engines are concerned!

In the longer term we intend to seek ways of combining the algorithms evolved here into an algorithm playing the entire game. The search algorithms we evolved

may provide a framework for searching generic chess positions (not only finding mates). Learning how to combine this search with the evaluation functions previously developed by [9] may give rise to stronger (evolved) chess players.

Ultimately, our approach could prove useful in every domain in which knowledge is used, with or without search. The genetic programming paradigm still bears great untapped potential in constructing and representing knowledge.

References

1. L. V. Allis, M. van der Meulen, and H. J. van den Herik. Proof-number search. *Artificial Intelligence*, 66:91–124, 1994.
2. D. F. Beal and M. C. Smith. Multiple probes of transposition tables. *ICCA Journal*, 19(4):227–233, December 1996.
3. M. S. Bourzutschky, J. A. Tamplin, and G. McC. Haworth. Chess endgames: 6-man data and strategy. *Theoretical Computer Science*, 349:140–157, December 2005.
4. Scott Brave. Evolving recursive programs for tree search. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 10, pages 203–220. MIT Press, Cambridge, MA, USA, 1996.
5. Murray Campbell, A. Joseph Hoane, Jr., and Feng-Hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1–2):57–83, 2002.
6. C. F. Chabris and E. S. Hearst. Visualization, pattern recognition, and forward search: Effects of playing speed and sight of the position on grandmaster chess errors. *Cognitive Science*, 27:637–648, February 2003.
7. R. Gross, K. Albrecht, W. Kantschik, and W. Banzhaf. Evolving chess playing programs. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 740–747, New York, 9-13 July 2002. Morgan Kaufmann Publishers.
8. A. Hauptman and M. Sipper. Analyzing the intelligence of a genetically programmed chess player. In *Late Breaking Papers at the Genetic and Evolutionary Computation Conference 2005. Washington DC*, June 2005.
9. A. Hauptman and M. Sipper. GP-endchess: Using genetic programming to evolve chess endgame players. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 120–131, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.
10. Tzung-Pei Hong, Ke-Yuan Huang, and Wen-Yang Lin. Adversarial search by evolutionary computation. *Evolutionary Computation*, 9(3):371–385, 2001.
11. Tzung-Pei Hong, Ke-Yuan Huang, and Wen-Yang Lin. Applying genetic algorithms to game search trees. *Soft Comput.*, 6(3-4):277–283, 2002.
12. Hermann Kaindl. Quiescence search in computer chess. *ACM SIGART Bulletin*, (80):124–131, 1982. Reprint in Computer Game-Playing: Theory and Practice, Ellis Horwood, Chichester, England, 1983.
13. John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

14. T. Anthony Marsland and Murray S. Campbell. A survey of enhancements to the alpha-beta algorithm. In *Proceedings of the ACM National Conference*, pages 109–114, November 1981.
15. Monty Newborn. Deep blue's contribution to AI. *Ann. Math. Artif. Intell*, 28(1-4):27–30, 2000.
16. Laszlo Polgar. *Chess : 5334 Problems, Combinations, and Games*. Black Dog and Leventhal Publishers, 1995.
17. S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs , NJ, 1995.