Genetic Algorithms for Evolving Computer Chess Programs

Omid E. David, H. Jaap van den Herik, Moshe Koppel, and Nathan S. Netanyahu

Abstract—This paper demonstrates the use of genetic algorithms for evolving: 1) a grandmaster-level evaluation function, and 2) a search mechanism for a chess program, the parameter values of which are initialized randomly. The evaluation function of the program is evolved by learning from databases of (human) grandmaster games. At first, the organisms are evolved to mimic the behavior of human grandmasters, and then these organisms are further improved upon by means of coevolution. The search mechanism is evolved by learning from tactical test suites. Our results show that the evolved program outperforms a two-time world computer chess champion and is at par with the other leading computer chess programs.

Index Terms—Computer chess, fitness evaluation, games, genetic algorithms, parameter tuning.

I. INTRODUCTION

DESPITE the many advances in machine learning and artificial intelligence (AI), there are still areas where learning machines have not yielded a performance comparable to the top performance exhibited by humans. Computer chess is one of the most difficult areas with this aim.

It is well known that computer games have served as an important testbed for spawning various innovative artificial intelligence AI techniques in domains and applications such as search, automated theorem proving, planning, and learning. In addition, the annual world computer chess championship (WCCC) is arguably the longest ongoing performance evaluation of programs in computer science, which has inspired other well known competitions in robotics, planning, and natural language understanding.

Computer chess, while being one of the most researched fields within AI, has not lent itself to the successful application of conventional learning methods, due to its enormous complexity. Hence, top chess programs still resort to manual tuning of the parameters of their evaluation function and their search functions, typically through years of trial and error. The evaluation function assigns a score to a given chess position, and the selective search function decides which moves to search more deeply and which moves to prune at an earlier stage in the search tree.

Our previous work described how the parameters of an evaluation function can be evolved by learning from other chess programs [21] and human chess players [19]. It also provided a method for tuning the search parameters of a chess program [20]. That is, we developed a method for evolving the parameters of a chess program's evaluation function given a tuned search mechanism, and a method for evolving the parameters of the search mechanism given a highly tuned evaluation function.

In this paper, we extend our previous work by providing a methodology for evolving the parameters of both the evaluation function and the selective search mechanism of a chess program using genetic algorithms (GA). That is, we assume no access to either a tuned function or a tuned search mechanism, but rather initialize the parameters of both components randomly (i.e., the initial program includes only the rules of the game, a set of randomly initialized evaluation function parameters, and search function parameters). Additionally, we provide extensive experimental results demonstrating that the combined approach is capable of evolving highly tuned chess programs, on a par with top tournament-playing chess programs.

We observe that although the experimental results of the resulting approach presented in this paper differ from those of our previous work, in terms of the evolved parameter values, the number of positions solved using a test suite, and the results of matches against other top programs, the overall performances are comparable. This suggests that the new algorithm is advantageous in the sense that it initially relies only on databases of (human) grandmaster games.

At the first stage we evolve the parameters of the evaluation function, relying on a straightforward mechanism for conducting 1-ply searches only. That is, since we do not yet have a sophisticated selective search mechanism, we use a plain mechanism for making legal 1-ply moves. After evolving the parameters of the evaluation function and incorporating its evolved values, we evolve at the second stage the parameters of our selective search mechanism.

In Section II, we review past attempts at applying evolutionary techniques in computer chess. We also compare alternative learning methods to evolutionary methods, and argue why the latter are more appropriate for evolving an evaluation function. Section III presents our new approach, including a

O. E. David, and M. Koppel are with the Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel (e-mail: mail@omiddavid.com; koppel@cs.biu.ac.il).

H. J. van den Herik is with the Tilburg Center for Cognition and Communication, Tilburg University, Tilburg 5037 AB, The Netherlands (e-mail: h.j.vdnherik@uvt.nl).

N. S. Netanyahu is with the the Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel, and also with the Center for Automation Research, University of Maryland at College Park, College Park, MD 20742, USA (e-mail: nathan@cfar.umd.edu).

detailed description of the framework of the GA as applied to evolving the parameters of a chess program's evaluation function. Section IV provides a GA-based method for tuning the parameter values of the selective search mechanism of the program. Section V provides our extended experimental results, and Section VI contains the concluding remarks.

II. LEARNING IN COMPUTER CHESS

While the first chess programs could not pose a challenge to even a novice player [42], [46], the current advanced chess programs have been outperforming the strongest human players, as the recent man versus machine matches clearly indicate [48]. This improvement is largely a result of the deep searches that are possible nowadays, thanks to both hardware speed and improved search techniques. While the search depth of early chess programs was limited to only a few plies, nowadays tournament-playing programs easily search more than a dozen plies in the middlegame, and tens of plies in the late endgame.

Despite their groundbreaking achievements, a glaring deficiency of today's top chess programs is their severe lack of a learning capability (except in the most negligible ways, e.g., "learning" not to play an opening that resulted in a loss, etc.). In other words, despite their seemingly intelligent behavior, those top chess programs are mere brute-force (albeit efficient) searchers that lack true underlying intelligence.

A. Conventional Versus Evolutionary Learning in Computer Chess

During more than 50 years of research in the area of computer games, many learning methods have been employed in less complex games. Chellapilla and Fogel [11]–[13], [24] created an expert-level checkers program by using coevolution to evolve neural network board evaluators. Temporal difference learning has been successfully applied in backgammon [44] and checkers [40]. Although temporal difference learning has also been applied to chess [3], the results showed that after three days of learning, the playing strength of the program was merely 2150 Elo (see Appendix B for a description of the Elo rating system), which is quite a low rating for a chess program. These experimental results confirmed Wiering's [47] formal arguments, presented five years earlier, for the failure of these methods in rather complex games such as chess. Runarsson and Lucas [38] compared least squares temporal difference learning [LSTD(λ)] with preference learning in the game of Othello, using samples of games from human competitions held by the French Othello Federation. Their results showed that preference learning produces policies that better capture the behavior of expert players, and also lead to higher levels of play when compared to LSTD(λ). It would be interesting to apply this approach to chess, which is more complex than Othello. For example, while the average branching factor (i.e., the average number of legal moves available in each position) in chess is 38, it is seven in Othello, and smaller than three in checkers (when considering both capture and non-capture positions).

The issue of learning in computer chess is basically an optimization problem. Each program plays by conducting a search, where the root of the search tree is the current position, and the leaf nodes (at some predefined depth of the tree) are evaluated by some static evaluation function. In other words, sophisticated as the search algorithms may be, most of the knowledge of the program lies in its evaluation function. Even though automatic tuning methods, that are based mostly on reinforcement learning, have been successfully applied to less complex games such as checkers, they have had almost no impact on state-of-the-art chess engines. Currently, all top tournament-playing chess programs use hand-tuned evaluation functions, since conventional learning methods cannot cope with the enormous complexity of the problem. This is underscored by the following four points.

1) The space to be searched is huge. It is estimated that there are about 10^{46} different positions possible that can arise in chess [14]. As a result, any method based on exhaustive search of the problem space is so far infeasible.

2) The search space is not smooth and unimodal. The evaluation function's parameters of any top chess program are highly co-dependent. An example from David's computer chess career [15], [18] may illustrate this. In many cases, increasing the values of three parameters results in a worse performance, but if a fourth parameter were also increased, then an improved overall performance would be obtained. Since the search space is not unimodal, i.e., it does not consist of a single smooth hill, any gradient-ascent algorithm, such as hill climbing, will perform poorly. In contrast, genetic algorithms are known to perform well in large search spaces, which are not unimodal.

3) The problem of tuning and learning is not well understood. As will be discussed in detail in the next section, even though all top programs are hand-tuned by their programmers, finding the best value for each parameter is based mostly on educated guessing and intuition. (The fact that all top programs continue to operate in this manner attests to the lack of practical alternatives.) Had the problem been well understood, a domain-specific heuristic would have outperformed a general-purpose method such as GA.

4) We do not require a global optimum to be found. Our goal in tuning an evaluation function is to adjust its parameters so that the overall performance of the program is enhanced. In fact, a unique global optimum most probably does not exist for this tuning problem.

In view of the above four points it seems appropriate to employ GA for automatic tuning of the parameters of an evaluation function. Indeed, at first glance this appears like an optimization task, well suited for GA. The many parameters of the evaluation function (bonuses and penalties for each property of the position) can be encoded as a bit-string. We can randomly initialize many such chromosomes, each representing one evaluation function. Thereafter, one needs to evolve the population until highly tuned fit evaluation functions emerge.

However, there is one major obstacle that hinders the above application of GA, namely the fitness function. Given a set of parameters of an evaluation (encoded as a chromosome), how should the fitness value be calculated? A straightforward approach would be to let the chromosomes in each generation play a series of games against each other, and subsequently, record the score of each individual as its fitness value (each individual is a chess program with an appropriate evaluation function).

The main drawback of this approach is the large amount of time needed to evolve each generation. As a result, limitations should be imposed on the length of the games played after each generation, and also on the size of the population involved. With a population size of 100, a ten seconds limit per game, and assuming that each individual plays each other individual once in every generation, it would take 825 minutes for each generation to evolve. Specifically, reaching the 100th generation would take up to 57 days. These figures suggest that it would be rather difficult to evolve the parameter values of a chess program relying on coevolution alone. (When some *a priori* knowledge regarding material and positional values is used, as in [25]–[27] for example, good results are attainable in a reasonable amount of time.)

In Section III we present our approach for using GA in evolving state-of-the-art chess evaluation functions. Before that, we briefly review previous works of applying evolutionary methods in computer chess.

B. Previous Evolutionary Methods Applied to Chess

Despite the abovementioned problems, there have been some successful applications of evolutionary techniques in computer chess, subject to some restrictions. Genetic programming was successfully employed by Hauptman and Sipper [30], [31] for evolving programs that can solve Mate-in-N problems and play chess endgames.

Kendall and Whitwell [35] used evolutionary algorithms for tuning the parameters of an evaluation function. Their approach had limited success, due to the very large number of games required (as previously discussed), and the small number of parameters used in their evaluation function. Their evolved program managed to compete with strong programs only if their search depth (lookahead) was severely limited.

Similarly, Aksenov [1] employed genetic algorithms for evolving the parameters of an evaluation function, using games between the organisms for determining their fitness. Again, since this method required a very large amount of games, it evolved only a few parameters of the evaluation function with limited success. Tunstall-Pedoe [45] also suggested a similar approach, without providing an implementation.

Gross *et al.* [28] combined genetic programming and evolution strategies to improve the efficiency of a given search algorithm using a distributed computing environment on the Internet.

Fogel *et al.* [25]–[27] used coevolution to successfully improve the parameters of an existing chess program. Their algorithm learns to evaluate chessboard configurations by using the positions of pieces, material and positional values, and neural networks to assess specific sections of the chessboard.

The method succeeded in modifying the parameter values of an existing chess program to gain a respectable performance level (the program scored three wins, 11 draws, and ten losses against FRITZ 8 and defeated a human master). As far as we know, no successful attempt has been described at using coevolution to evolve the parameters of a chess program from fully randomized initial values, without relying on any a priori knowledge. Fogel *et al.* showed that coevolution could be employed successfully in chess when the initial material and positional parameters are already initialized within sensible ranges. Furthermore, Chellapilla and Fogel [11]–[13], [24] successfully employed coevolution to evolve the parameters of a checkers program. However, the applicability of their approach to chess is not clear, as the game of chess is by orders of magnitude more complex than checkers (the fact that checkers is by now a solved game [41] further attests to its relative simplicity in comparison to chess, which seems far from being solved).

Samothrakis *et al.* [39] used covariance matrix adaptation evolution strategy (CMA-ES) for coevolution of weighted piece counters (WPCs) in Othello. These WPCs were used to operate as value functions in a 1-ply minimax player. The aim was not to find Othello players that are strong in absolute terms, but rather use Othello as an interesting domain of study in which to measure performance and intransitivities in coevolution. Using CMA-ES within a coevolutionary setting, the authors succeeded in evolving the strongest WPCs for Othello, published as of yet. While this coevolutionary approach yields interesting results for Othello, it would be challenging to apply it to chess, which is more complex than Othello (as noted in the previous section).

There are two main components of any chess program, an evaluation function and a search mechanism. In our previous work, we provided a mentor-assisted approach [21] for reverse engineering the evaluation function of a target chess program (the mentor), thereby evolving a new comparable evaluation function. This approach relied on the fact that the mentor provides an evaluation score for any given position, and so, using a large set of positions, the evaluation score of the target program can be obtained for each position. Subsequently, the parameters of the evaluation function can be evolved to mimic the scores provided by the mentor. This mentor-assisted approach produces an evaluation function that mimics a given chess program, but is dependent on access to the evaluation score of the target program.

In order to extend the concept of mentor-assisted evolution to learning from humans as well, we combined evolution and coevolution for evolving the parameter values of the evaluation function to simulate the moves of a human grandmaster, without relying on the availability of evaluation scores of some computer chess programs [19]. In both these versions we had assumed that the program already contained a highly tuned search mechanism.

While most past attempts at automatic learning in chess focused on an evaluation function, few efforts have been made to evolve automatically the parameters controlling the search mechanism of the program. Moriarty and Miikkulainen [37] used neural networks for tuning the search of an Othello

program, but as mentioned in their paper, their method is not easily applicable to more complex games such as chess. Cazenave [10] presented a method for learning search-control rules in Go, but the method cannot be applied to chess (the search mechanism in Go is of a different, less critical nature than chess).

Björnsson and Marsland [8] presented a method for automatically tuning search extensions in chess. Given a set of test positions (for which the correct move is predetermined) and a set of parameters to optimize (in their case, four extension parameters), they tune the values of the parameters using gradient descent. Their program processes all the positions, and for each position it records the number of nodes until a solution is found. The optimization goal is to minimize the total node count for all the positions. In each iteration, their method modifies each of the extension parameters by a small value, and processes all the positions, recording the total node count. Thus, given N parameters to be optimized ($N = 4$ in their case), in each iteration their method processes all the positions N times. The parameter values are updated after each iteration, such that the total node count decreases. Björnsson and Marsland applied their method for tuning the values of four search extension parameters: check, passed pawn, recapture, and one-reply extensions. Their results show that their method optimizes the values for these parameters such that the total node count for solving the test set is decreased.

Despite the success of this gradient-descent method for tuning the values of four search extension parameters, it cannot be used efficiently to optimize a large set of parameters consisting of all the critical selective search parameters, of which search extensions comprise only a few parameters. This is due to the fact that the gradient-descent approach requires processing the whole test set for each of the parameters in each iteration. While this might be practical for four parameters, it becomes more difficult when the number of parameters involved is considerably larger. Additionally, unlike the optimization of search extensions, where parameter values are mostly independent, other search methods would exhibit a high interdependency between the parameter values, and consequently it would be more difficult to apply gradient-descent optimization.

In our previous work [20] we demonstrated that given a chess program with a highly tuned evaluation function, we can use GA to evolve the search parameters by evolving the parameters such that the overall performance in test suites is improved. As mentioned previously, we have also demonstrated [19], [21] how to evolve an evaluation function using GA, given a highly tuned search mechanism.

In this paper we combine the two approaches in order to evolve both the evaluation function and the search mechanism, assuming that the parameters of none of these two components are tuned *a priori* (but rather initialized randomly).

III. EVOLUTION AND COEVOLUTION OF EVALUATION FUNCTIONS

The parameters of an evaluation function can easily be represented as a chromosome. However, applying a fitness

---

1) Using a database of grandmaster-level games, select a list of positions and the move played in that position.
2) For each position, let the organism perform a 1-ply search and store the move selected by the organism.
3) Compare the move suggested by the organism with the actual move made by the player. The fitness of the organism will be the total number of moves where the organism's move is the same as the player's move.

Fig. 1.   Fitness function for evolution of evaluation functions using database of grandmaster-level games.

function is more complicated. As previously noted, establishing a fitness evaluation by means of playing numerous games between the organisms in each generation (i.e., single-population coevolution) is quite difficult (when the organisms are initialized randomly).

In our previous work on mentor-assisted evolution [21], we described how the fitness value can be issued by running a grandmaster-level chess program on a set of positions, and recording, for each position, the difference between the evaluation score computed by the organism and the score computed by the target program. We define this difference as the evaluation error and a magnitude inversely proportional to it as the fitness function. However, this approach is not practical when trying merely to learn from a player's moves (whether a human player or a computer program, assuming no access to the program itself).

This paper significantly extends our previous framework. Specifically, we present a learning approach, which relies only on widely available databases of grandmaster-level games. This task is significantly more difficult than using existing chess programs as mentors, since for any position taken from a (human) game, the only available information is the move actually played (and not the associated score). Our approach is based on the steps shown in Fig. 1.

Although performing a search for each position seems to be a costly process, in fact it consumes little time. Conducting a 1-ply search amounts to less than a millisecond for a typical chess program on an average machine, and so 1000 positions can be processed in one second. This allows us to use a large set of positions for the training set.

After running the abovementioned process a number of times, we obtain several evolved organisms. Due to the random initialization of the chromosomes, each time the process is applied, a different best evolved organism is obtained. Comparing the best evolved organisms from different runs, we observe that even though they are of similar playing strength, their evolved parameter values differ, and so does their playing style.

In order to further improve the obtained organisms (each organism is the best evolved organism from a complete run of the abovementioned process), we next use a single-population coevolution phase. During this phase the evolved organisms play against each other, and the fitness function applied is based on this relative performance. After running this coevolution for a predetermined number of generations, the best evolved organism is selected as the best overall organism.

As the results in Section V indicate, this best of best organism improves upon the organisms evolved from the evolutionary phase only. As noted before, it is difficult to evolve the parameter values of a chess program from randomly initialized values relying on coevolution alone. The difference in our case is that the population size is small (we used ten), and the initial organisms are already well tuned, rather than randomly initialized.

In the following subsections, we describe in detail the chess program, the implementation of the evolutionary method, and the GA parameters used.

A. Chess Program and the Evaluation Function

The evaluation function of our program (which we are interested in tuning automatically) consists of 35 parameters (see Fig. 3). Even though this is a small number of parameters in comparison to other top programs, the set of parameters used does cover all important aspects of a position, e.g., material, piece mobility and centricity, pawn structure, and king safety.

The parameters of the evaluation function are represented as a binary bit-string (chromosome size: 224 bits), initialized randomly (note that we use binary encoding since it is the most basic type of encoding for GA [33], although it is not necessarily superior to an alternative encoding method for this problem). The value of a pawn is set to a fixed value of 100, which serves as a reference for all other parameter values. Except for the four parameters representing the material values of the pieces, all the other parameters are assigned a fixed length of 6 bits per parameter. Obviously, there are many parameters for which 3 or 4 bits suffice. However, allocating a fixed length of 6 bits to all parameters ensures that *a priori* knowledge does not bias the algorithm in any way. Note that at this point, the program's evaluation function is merely a random initialization, so that apart from the rules of the game, the program has essentially no game skills at all.

B. Evolution Using Grandmaster Games

For our experiments, we used a database of 10 000 games of grandmasters rated above 2600 Elo, and randomly picked one position from each game. We picked winning positions only, i.e., positions where the side to move had ultimately won the game (e.g., if it was white's turn to move, the game would be won eventually by white). Of these 10 000 positions, we selected 5000 positions for training and 5000 for testing.

In each generation, for each organism, the algorithm translates its chromosome bit-string to a corresponding evaluation function. For each of the N test positions (in our case, $N = 5000$), it then performs a 1-ply search using the decoded evaluation function, and the best move returned from the search is compared to that of the grandmaster in the actual game. The move is deemed correct if it is the same as the move played by the grandmaster, and incorrect otherwise. The fitness of the organism is calculated as the square of the total number of correct moves.

Note that unlike the mentor-assisted approach for mimicking existing chess programs, which provide numeric values for each position, here we only have 1-bit of information for each processed position (correct/incorrect). This underscores why learning merely from a player's moves is much more challenging than using computer programs as mentors.

Other than the special fitness function described above, we used a standard GA implementation with gray coded chromosomes, fitness-proportional selection, uniform crossover, and elitism (the best organism is copied to the next generation). The following parameters are used by the algorithm:

    population size = 100;
    crossover rate = 0.75;
    mutation rate = 0.005;
    number of generations = 200.

C. Coevolution of the Best Evolved Organisms

We ran the evolution process ten times, thus obtaining ten organisms, each being the best organism from one run. The parameter values of these ten organisms differ due to the random initialization in each run. Consequently, although these ten programs are of similar playing strength, their playing style is different. Note that using the top ten evolved organisms from one of the runs is not equivalent to taking ten organisms from ten different runs, as in the former case the top ten organisms from the same run will have mostly similar parameter values.

An alternative method for generating multiple evolved organisms would be to use different training sets for each run. For example, we might use a specific grandmaster for each run in the hope of obtaining organisms that mimic the styles of various grandmasters. However, our tests indicate that this approach does not improve over the method used. Apparently, using 1-ply searches only enables mimicking general grandmaster style, rather than the subtleties of a specific player.

In the coevolution phase, the ten best organisms selected serve as the initial population, which is then co-evolved over 50 generations. In each generation, each organism plays four games against each other organism (to obtain a more reliable result). Note that for each game, a different sequence of opening moves is selected from an opening book file. This ensures that each game is unique (i.e., each game contains a unique set of opening moves). At the end of each generation, rank-based selection is applied for selecting the organisms for breeding. Elitism is used here as well, which ensures that the best organism survives for the next generation. This is especially critical in light of the small population size. Other GA parameters remain unchanged, that is, uniform crossover with crossover rate of 0.75 and mutation rate of 0.005.

Note that at this phase we still do not have a tuned search mechanism. So, for the coevolution phase described above, we use a basic alpha-beta search mechanism, without any of the advanced selective search mechanisms (for which we have yet to evolve their parameter values). As a result, the playing strength of the program at this stage is substantially limited. However, this does not pose a problem, since this limitation equally applies to all ten programs which play against each other.

TABLE I
CHROMOSOME REPRESENTATION OF 18 SEARCH PARAMETERS
(LENGTH: 70 BITS)

Parameter	Value range	Bits
Null-move use	0–1	1
Null-move reduction	0–7	3
Null-move use adaptivity	0–1	1
Null-move adaptivity depth	0–7	3
Futility depth	0–3	2
Futility threshold depth-1	0–1023	10
Futility threshold depth-2	0–1023	10
Futility threshold depth-3	0–1023	10
Multi-cut use	0–1	1
Multi-cut reduction	0–7	3
Multi-cut depth	0–7	3
Multi-cut move num	0–31	5
Multi-cut cut num	0–7	3
Check extension	0–4	3
One-reply extension	0–4	3
Recapture extension	0–4	3
Passed pawn extension	0–4	3
Mate threat extension	0–4	3
Total chromosome length		70

In the following section we present our method for evolving the parameter values of the selective search mechanism.

IV. EVOLUTION OF SELECTIVE SEARCH

After evolving the parameter values of the evaluation function and incorporating the values of the best evolved organism into the program, we now focus on evolving the parameter values of the search mechanism.

Several popular selective search methods are employed by top tournament-playing programs. These methods allow the program to search more selectively, i.e., prune uninteresting moves earlier, thus spending additional time on more promising moves. That is, instead of searching all the moves to a certain fixed depth, some moves are searched more deeply than others.

The most popular selective search methods are null-move pruning [4], [17], [22], futility pruning [32], multicut pruning [6], [7], and selective extensions [2], [5]. A description of these methods can be found in [20]. Each of these selective search methods requires several critical parameters to be tuned. Normally these parameters are manually tuned, usually through years of experiments and manual optimizations.

In order to apply GA for tuning the parameters of these selective search methods automatically, we represent these parameters as a binary chromosome where each parameter's number of allocated bits is based on the reasonable ranges for the values of the parameter. Table I presents the chromosome and the range of values for each parameter. Note that for search extensions the notion fractional ply is applied, where 1 ply = 4 units (e.g., an extension value of two is equivalent to half a ply, etc.).

We employ the optimization goal in Björnsson and Marsland [8] as our fitness function. A set of 879 tactical test positions from the encyclopedia of chess middlegames (ECM) was used for training. Each position in the ECM test suite has a predetermined best move. Each chromosome processes all of the 879 positions, and for each position it attempts to find this predetermined best move as fast as possible.

Instead of counting the number of correctly solved positions (number of positions for which the organism found the best move), we used the number of nodes the organism had to process in order to find the best move. For each position, we recorded the number of nodes the organism searched before finding the correct move. The total node count for each organism is the total node count for all the 879 positions. We imposed an upper search limit of 500 000 nodes per position. That is, if the correct move is not found after searching 500 000 nodes, the search is stopped and this upper limit is returned as the number of nodes for that position. Naturally, more positions will be solved if a larger search limit is chosen, but also more time will be spent, and subsequently the whole evolution will take much longer.

The lower the total node count for all the positions, the higher the fitness of the organism will be. Using this fitness measure instead of taking, in a straightforward manner, the number of solved positions has the benefit of deriving more fitness information per position. Rather than obtaining a 1-bit information (correct/false) for each position, a numeric value is obtained, which also measures how quickly the position is solved. Thus, the organism is not only encouraged to solve more positions, but also to find quicker solutions for the already solved test positions.

Similar to [20], we used a standard GA implementation with Gray coded chromosomes, fitness-proportional selection, uniform crossover, and elitism. All the organisms are initialized with random values. The following parameters are used for the GA:

population size = 10;
crossover rate = 0.75;
mutation rate = 0.05;
number of generations = 50.

The next section contains the experimental results using the GA-based method for the parameters of the evaluation function and the search mechanism.

V. EXPERIMENTAL RESULTS

We now present the results of running the evolutionary process described in the previous two sections. We also provide the results of several experiments that measure the strength of the evolved program in comparison to CRAFTY, a former two-time world computer chess champion that is commonly used as a baseline for testing chess programs. The details of our experimental setup are provided in Appendix A.

A. Results of Evolution

Fig. 2 shows the number of positions solved (i.e., the number of correct moves found) for the best organism and the population average for 200 generations (the figure shows the results for the first run, out of a total of ten runs). Specifically, the results indicate that the average number of solved positions is about 800 (out of 5000) in the first generation. Note that even without any chess knowledge an organism would
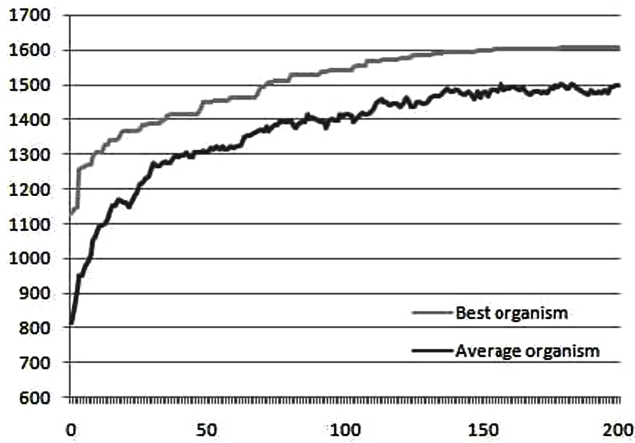
Fig. 2. Number of positions solved (out of 5000) for the best organism and the population average in each generation (total time for 200 generations ≈ two hours). The figure shows the results for the first run, out of a total of ten runs.

PAWN_VALUE	100
KNIGHT_VALUE	521
BISHOP_VALUE	572
ROOK_VALUE	824
QUEEN_VALUE	1710
PAWN_ADVANCE_A	3
PAWN_ADVANCE_B	6
PASSED_PAWN_MULT	10
DOUBLED_PAWN_PENALTY	14
ISOLATED_PAWN_PENALTY	8
BACKWARD_PAWN_PENALTY	3
WEAK_SQUARE_PENALTY	5
PASSED_PAWN_ENEMY_KING_DIST	7
KNIGHT_SQ_MULT	6
KNIGHT_OUTPOST_MULT	9
BISHOP_MOBILITY	4
BISHOP_PAIR	28
ROOK_ATTACK_KING_FILE	51
ROOK_ATTACK_KING_ADJ_FILE	8
ROOK_ATTACK_KING_ADJ_FILE_ABGH	26
ROOK_7TH_RANK	30
ROOK_CONNECTED	6
ROOK_MOBILITY	4
ROOK_BEHIND_PASSED_PAWN	40
ROOK_OPEN_FILE	27
ROOK_SEMI_OPEN_FILE	11
ROOK_ATCK_WEAK_PAWN_OPEN_COLUMN	15
ROOK_COLUMN_MULT	6
QUEEN_MOBILITY	2
KING_NO_FRIENDLY_PAWN	35
KING_NO_FRIENDLY_PAWN_ADJ	10
KING_FRIENDLY_PAWN_ADVANCED1	6
KING_NO_ENEMY_PAWN	17
KING_NO_ENEMY_PAWN_ADJ	9
KING_PRESSURE_MULT	4

Fig. 3. Average evolved parameters of the evaluation function of the best individual after ten runs (the values are rounded).

occasionally select the correct move by random guessing. Additionally, since the randomly initialized parameters contain only positive values, an organism can find some basic captures of the opponent's pieces without possessing any real chess knowledge.

The average number of solved positions increases until stabilizing at around 1500, which corresponds to 30% of the positions. The best organism at generation 200 solves 1620 positions, which corresponds to 32.4% of the positions. ~~Due to the use of elitism,~~ the number of solved positions for the best organism is monotonically increasing, since the best organism is preserved, due to the use of elitism. The entire 200-generation evolution took approximately two hours.

At first glance, a solution rate of 32% might not seem too high. However, in light of the fact that the organisms base their move on a 1-ply search only (as opposed to the thorough analysis and consideration of the position by a grandmaster prior to their move), this figure is quite satisfactory. In other words, by conducting merely a 1-ply search, the evolved organism selects successfully a grandmaster's carefully analyzed move in one out of three cases.

With the completion of the learning phase, we used the additional 5000 positions set aside for testing. We let our best evolved organism perform a 1-ply search on each of these positions. The number of correctly solved positions was 1521 (30.4%). This indicates that the first 5000 positions used for training cover most types of positions that can arise, as the success rate for the testing set is close to the success rate for the training set.

B. Results of Coevolution

Repeating the evolutionary process, we obtained each time a best evolved organism with a different set of evolved parameter values. That is, each run produced a different grandmaster-level program. Even though the performance of these independently evolved best organisms is fairly similar, our goal was to improve upon these organisms and create an enhanced best of best organism.

We applied single-population coevolution to enhance the performance of the program. After running the evolution ten times (which ran for about 20 hours), ten different best organisms were obtained. Using these ten organisms as the starting population, we applied GA for 50 generations, where each organism played each other organism four times in every round. Each game was limited to ten seconds (five seconds per side). In practice, this coevolution phase ran for approximately 20 hours.

Fig. 3 provides the average evolved values of the best individual obtained after running our method ten times (the values are rounded). Conventionally, knight and bishop are valued at three pawns, and rook and queen are valued at five and nine pawns, respectively [9]. These values are used in most chess programs. However, it is well known that grandmasters frequently sacrifice pawns for positional advantages, and that in practice, a pawn is assigned a lower value. Interestingly, the best organism assigns a pawn about half the value it is usually assigned, relative to the other pieces, which is highly unconventional for chess programs. This implies that the evolved organism, which learns its parameter values from
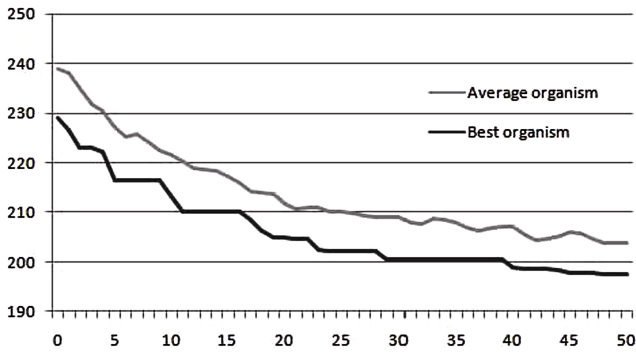
Fig. 4. Total node count (in millions) for 879 ECM positions for the best organism and the population average in each generation.

TABLE II
LEARNED VALUES FOR THE SEARCH PARAMETERS

Parameter	Learned value
Null-move use	1
Null-move reduction	4
Null-move use adaptivity	1
Null-move adaptivity depth	6
Futility depth	3
Futility threshold depth-1	112
Futility threshold depth-2	227
Futility threshold depth-3	506
Multi-cut use	1
Multi-cut reduction	4
Multi-cut depth	6
Multi-cut move num	15
Multi-cut cut num	3
Check extension	4
One-reply extension	4
Recapture extension	2
Passed pawn extension	3
Mate threat extension	2

human grandmasters, ends up adopting also their less materialistic style of play. This is also reflected in the playing style of the ultimate evolved program, as it frequently sacrifices pawns for gaining positional advantages.

C. Evolving Search Parameters

After incorporating the best evolved parameters for the evaluation function into the program, we then evolved the search parameters. Fig. 4 shows the total node count for 879 positions for the best organism and the population average in each generation. Table II provides the evolved values of the best individual.

We incorporate the values obtained here together with the values obtained from evolution of the parameter values of the evaluation function into our program (which is an experimental descendant of the program FALCON [15], [18]). We call this program EVOL*. The program contains all our obtained parameter values for the evaluation function and the search mechanism.

To measure the performance of EVOL*, we conducted a series of matches against the chess program CRAFTY [34]. CRAFTY has successfully participated in numerous WCCCs, and is a direct descendent of CRAY BLITZ, the WCCC winner of 1983 and 1986. It is frequently used in the literature as a standard reference. In order to obtain a measure for the performance gain due to the coevolution phase, we also

TABLE III
NUMBER OF ECM POSITIONS SOLVED BY EACH PROGRAM (TIME: FIVE SECONDS PER POSITION)

EVOL*	EVOL$_0$	CRAFTY
654	623	593

TABLE IV
EVOL* VERSUS EVOL$_0$ AND CRAFTY (W% IS THE WINNING PERCENTAGE AND RD IS THE ELO RATING DIFFERENCE). TIME CONTROL: FIVE MINUTES PER GAME

Match	Result	W%	RD
EVOL* - EVOL$_0$	180.0 - 120.0	60.0%	+70.4
EVOL* - CRAFTY	181.5 - 118.5	60.5%	+74.1

compared the performance of EVOL* to a version of EVOL*, which uses the parameters in its evaluation function that are evolved after the evolutionary phase (i.e., parameters evolved before the coevolution phase). We call this program EVOL$_0$. For the search mechanism we used the same parameter setting in EVOL* and EVOL$_0$.

For a proper comparison, we let EVOL*, EVOL$_0$, and CRAFTY process the ECM test suite with five seconds per position. Table III provides the results. As can be seen, both EVOL* and EVOL$_0$ solve significantly more positions than CRAFTY.

The superior performance of EVOL* on the ECM test set is not surprising, as it was evolved on this training set. Therefore, in order to obtain an unbiased performance comparison, we conducted a series of 300 matches against CRAFTY, using a time control of five minutes per game for each side. Table IV provides the results.

The results further show that EVOL* outperforms EVOL$_0$ and CRAFTY, not only in terms of solving more tactical test positions, but more importantly in test matches. These results establish that although the search parameters were evolved from randomly initialized chromosomes, the resulting organism outperforms a grandmaster-level chess program in five minute games.

We extended our experiments to compare the performance of EVOL* against several of the world's top commercial chess programs. These programs included JUNIOR, FRITZ, and HIARCS. JUNIOR won the World Microcomputer Chess Championship in 1997 and 2001 and the World Computer Chess Championship in 2002, 2004, 2006, and 2011. In 2003, JUNIOR played a six-game match against the legendary former world champion, Garry Kasparov, that resulted in a 3–3 tie. In 2007, JUNIOR won the ultimate computer chess challenge organized by the World Chess Federation (FIDE), defeating FRITZ 4–2. FRITZ won the World Computer Chess Championship in 1995. In 2002, it drew the Brains in Bahrain match against former world champion, Vladimir Kramnik, 4-4, and in 2003 it drew again a four-game match against Garry Kasparov. In 2006, FRITZ defeated the incumbent world champion, Vladimir Kramnik, 4–2. HIARCS won the 1993 World Microcomputer Chess Championship. In 2003, it drew a four-game match against Grandmaster Evgeny Bareev, then

TABLE V

EVOL* VERSUS JUNIOR, FRITZ, AND HIARCS (W% IS THE WINNING PERCENTAGE, AND RD IS THE ELO RATING DIFFERENCE). TIME CONTROL: FIVE MINUTES PER GAME

Match	Result	W%	RD
EVOL* - JUNIOR	133.5 - 166.5	44.5%	−38.4
EVOL* - FRITZ	162.0 - 138.0	54.0%	+27.9
EVOL* - HIARCS	180.5 - 119.5	60.2%	+71.6

TABLE VI

EVOL* VERSUS CRAFTY, JUNIOR, FRITZ, AND HIARCS (W% IS THE WINNING PERCENTAGE AND RD IS THE ELO RATING DIFFERENCE). TIME CONTROL: ONE HOUR PER GAME

Match	Result	W%	RD
EVOL* - CRAFTY	61.5 - 38.5	61.5%	+81.4
EVOL* - JUNIOR	41.0 - 59.0	41.0%	−63.2
EVOL* - FRITZ	51.5 - 48.5	51.5%	+10.4
EVOL* - HIARCS	58.5 - 41.5	58.5%	+59.6

TABLE VII

NUMBER OF ECM POSITIONS SOLVED BY EACH PROGRAM (TIME: FIVE SECONDS PER POSITION)

EVOL*	JUNIOR	FRITZ	HIARCS
654	681	640	642

chromosomes initialized randomly, the resulting organism is on a par with top commercial chess programs.

VI. CONCLUDING REMARKS

In this paper, we presented a novel approach for evolving the key components of a chess program from randomly initialized values using genetic algorithms.

In contrast to our previous successful attempts, which focused on mimicking the evaluation function of a chess program acting as a mentor, the approach presented in this paper focuses on evolving the parameters of the evaluation function and the search mechanism. This is done by observing solely (human) grandmaster-level games, where the only available information to guide the evolution is the moves played in these games.

Learning from the actual moves of grandmaster-level games in the first phase of the evolution, we obtained several evaluation functions. Specifically, running the procedure ten times, we obtained ten such evolved evaluation functions, which served as the initial population for the second coevolution phase. Using coevolution in our case proved successful as the initial population was not random, but relatively well tuned due to the first phase of the evolution.

We further used genetic algorithms to evolve the parameters of the search mechanism. Starting from randomly initialized values, our method evolves these parameters. Combining the two parameter sets (i.e., those of the evaluation function and the search mechanism) resulted in a performance that is on a par with that of top tournament-playing programs.

Overall, this paper provides, to the best of our knowledge, the first methodology of automatic learning the parameters of the evaluation function and the search mechanism from randomly initialized values for computer chess. We note that although the experimental results of the module presented here differ from those of our previous work (which assumed access to a highly tuned evaluation function [20] or a highly tuned search mechanism [19], [21]), the overall performance observed was comparable. Thus, the approach in this paper is superior to these previous efforts, in the sense that it achieves grandmaster-level performance without relying on any assumptions.

The results presented in this paper point to the vast potential in applying evolutionary methods for learning from human experts. We believe that this approach could be applied to a wide array of problems for essentially reverse engineering the knowledge of a human expert. While we successfully used genetic algorithms to evolve the parameter values of a chess program, it would be of interest to apply other natural and evolutionary optimization methods in future research [e.g.,

the 8th ranked player in the world (all four games ended in a draw). In 2007, HIARCS won the 17th international Paderborn computer chess championship.

As mentioned earlier, EVOL* is an experimental descendant of the program FALCON [15], [18], which successfully participated in three WCCCs. During the 2008 world computer chess championship [16], FALCON used an earlier version [21] of the evolutionary approach described in this paper. Competing with an average laptop against nine strong chess programs (eight of which ran on fast multicore machines ranging from 4 to 40 cores), the GA-based version of FALCON reached second place in the world computer speed chess championship and sixth place in the WCCC. These highly surprising results, especially in light of the huge hardware handicap of FALCON relative to its competitors, demonstrate the capabilities of the GA-based approach.

Table V provides the results of EVOL* against the top commercial programs JUNIOR, FRITZ, and HIARCS. Again, all matches consisted of 300 games at a time control of 5 minutes per game for each side.

To further examine the performance of EVOL* under long time control conditions, we conducted a series of matches between EVOL* and the abovementioned top tournament-playing programs at a time control of 1 hour per game for each side. Each match consisted of 100 games. Table VI provides the results.

The results show that the performance of a genetically evolved program is on a par with that of the top commercial chess programs. The superior performance by JUNIOR is well noted and should serve as a source of inspiration. In addition, Table VII compares the tactical performance of our evolved organism against these three commercial programs. The results show the number of ECM positions solved by each program. A similar trend emerges, i.e., the evolved organism is on a par with these top commercial programs.

The results of the abovementioned tests establish that even though the parameters of our program are evolved from

covariance matrix adaptation evolution strategy (CMA-ES) [29], differential evolution (DE) [43], and particle swarm optimization (PSO) [36]].

Appendix A
Experimental Setup

Our experimental setup consisted of the following resources.

1) CRAFTY, JUNIOR, FRITZ, and HIARCS chess programs running as native ChessBase engines.
2) Encyclopedia of chess middlegames (ECM) test suite, consisting of 879 positions.
3) FRITZ 9 interface for automatic running of matches, using SHREDDER opening book.
4) Long time control matches (see Table VI) were conducted on an Intel Core 2 Quad Q8300 with 8.0 GB RAM. All the other experiments were conducted on an Intel Core 2 Duo T8100 with 3.0 GB RAM.

Appendix B
Elo Rating System

The Elo rating system, developed by Arpad E. Elo [23], is the official system for calculating the relative skill levels of players in chess. A grandmaster is generally associated with rating values above 2500 Elo and a novice player with rating values below 1400 Elo.

Given the rating difference (RD) of two players, the following formula calculates the expected winning rate (W, between zero and one) of the player:

$$W = \frac{1}{10^{-RD/400} + 1}.$$

Given the winning rate of a player, as is the case in our experiments, the expected rating difference can be derived from the above formula

$$RD = -400 \log_{10}(\frac{1}{W} - 1).$$

References

[1] P. Aksenov, "Genetic algorithms for optimising chess position scoring," M.Sc. Thesis, Dept. Comput. Sci., Univ. Joensuu, Joensuu, Finland, 2004.

[2] T. S. Anantharaman, "Extension heuristics," *ICCA J.*, vol. 14, no. 2, pp. 47–65, 1991.

[3] J. Baxter, A. Tridgell, and L. Weaver, "Learning to play chess using temporal-differences," *Mach. Learn.*, vol. 40, no. 3, pp. 243–263, 2000.

[4] D. F. Beal, "Experiments with the null move" in *Advances in Computer Chess 5*. Amsterdam, The Netherlands: Elsevier Science, 1989, pp. 65–79.

[5] D. F. Beal and M. C. Smith, "Quantification of search extension benefits," *ICCA J.*, vol. 18, no. 4, pp. 205–218, 1995.

[6] Y. Björnsson and T. A. Marsland, "Multi-cut pruning in alpha-beta search," in *Proc. 1st Int. Conf. Comput. Games*, 1998, pp. 15–24.

[7] Y. Björnsson and T. A. Marsland, "Multi-cut alpha-beta-pruning in game-tree search," *Theoretical Comput. Sci.*, vol. 252, nos. 1–2, pp. 177–196, 2001.

[8] Y. Björnsson and T. A. Marsland, "Learning control of search extensions," in *Proc. 6th Joint Conf. Inf. Sci.*, 2002, pp. 446–449.

[9] J. R. Capablanca, *Chess Fundamentals*, N. de Firmian, Ed. New York, NY, USA: Random House, 2006.

[10] T. Cazenave, "Generation of patterns with external conditions for the game of Go," *Advances in Computer Games 9*, H. J. van den Herik and B. Monien, Eds. Maastricht, The Netherlands: Springer, 2001, pp. 275–293.

[11] K. Chellapilla and D. B. Fogel, "Evolving neural networks to play checkers without expert knowledge," *IEEE Trans. Neural Netw.*, vol. 10, no. 6, pp. 1382–1391, Nov. 1999.

[12] K. Chellapilla and D. B. Fogel, "Evolution, neural networks, games, and intelligence," *Proc. IEEE*, vol. 87, no. 9, pp. 1471–1496, Sep. 1999.

[13] K. Chellapilla and D. B. Fogel, "Evolving an expert checkers playing program without using human expertise," *IEEE Trans. Evol. Comput.*, vol. 5, no. 4, pp. 422–428, Aug. 2001.

[14] S. Chinchalkar, "An upper bound for the number of reachable positions," *ICCA J.*, vol. 19, no. 3, pp. 181–183, 1996.

[15] O. E. David, "Search and knowledge-based innovations in computer chess," Master's Thesis, Dept. Comput. Sci., Bar-Ilan Univ., Ramat-Gan, Israel, 2005.

[16] O. E. David, "The 16th world computer-chess championship," *ICGA J.*, vol. 31, no. 3, pp. 166–171, 2008.

[17] O. E. David and N. S. Netanyahu, "Extended null-move reductions," in *Proc. Int. Conf. Comput. Games*, 2008, pp. 205–216.

[18] O. E. David, "Genetic algorithms based learning for evolving intelligent organisms," Ph.D. dissertation, Dept. Comput. Sci., Bar-Ilan Univ., Ramat-Gan, Israel, 2009.

[19] O. E. David, H. J. van den Herik, M. Koppel, and N. S. Netanyahu, "Simulating human grandmasters: Evolution and coevolution of evaluation functions," in *Proc. ACM Genetic Evol. Comput. Conf.*, 2009, pp. 1483–1489.

[20] O. E. David, M. Koppel, and N. S. Netanyahu, "Genetic algorithms for automatic search tuning," *ICGA J.*, vol. 33, no. 2, pp. 67–79, 2010.

[21] O. E. David, M. Koppel, and N. S. Netanyahu, "Expert-driven genetic algorithms for simulating evaluation functions," *Genetic Program. Evolv. Mach.*, vol. 12, no. 1, pp. 5–22, 2011.

[22] C. Donninger, "Null move and deep search: Selective search heuristics for obtuse chess programs," *ICCA J.*, vol. 16, no. 3, pp. 137–143, 1993.

[23] A. E. Elo. A. E. Elo. *The Rating of Chessplayers, Past and Present*. London, UK: B. T. Batsford Ltd., 1978.

[24] D. B. Fogel, *Blondie24: Playing at the Edge of AI*. San Francisco, CA, USA: Morgan Kaufmann, 2002.

[25] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon, "A self-learning evolutionary chess program," *Proc. IEEE*, vol. 92, no. 12, pp. 1947–1954, Dec. 2004.

[26] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon, "Further evolution of a self-learning chess program," in *Proc. IEEE Symp. Comput. Intell. Games*, 2005, pp. 73–77.

[27] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon, "The Blondie25 chess program competes against Fritz 8.0 and a human chess master," in *Proc. IEEE Symp. Comput. Intell. Games*, 2006, pp. 230–235.

[28] R. Gross, K. Albrecht, W. Kantschik, and W. Banzhaf, "Evolving chess playing programs," in *Proc. Genetic Evol. Comput. Conf.*, 2002, pp. 740–747.

[29] N. Hansen and A. Ostermeier, "Completely derandomized self-adaptation in evolution strategies," *Evol. Comput.*, vol. 9, no. 2, pp. 159–195, 2001.

[30] A. Hauptman and M. Sipper, "Using genetic programming to evolve chess endgame players," in *Proc. Eur. Conf. Genetic Programm.*, 2005, pp. 120–131.

[31] A. Hauptman and M. Sipper, "Evolution of an efficient search algorithm for the Mate-in-N problem in chess," in *Proc. Eur. Conf. Genetic Programm.*, 2007, pp. 78–89.

[32] E. A. Heinz, "Extended futility pruning," *ICCA J.*, vol. 21, no. 2, pp. 75–83, 1998.

[33] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI, USA: Univ. Michigan Press, 1975.

[34] R. M. Hyatt, A. E. Gower, and H. L. Nelson, "Cray blitz," *Computers, Chess, and Cognition*, T.A. Marsland and J. Schaeffer, Eds. New York, NY, USA: Springer-Verlag, 1990, pp. 227–237.

[35] G. Kendall and G. Whitwell, "An evolutionary approach for the tuning of a chess evaluation function using population dynamics," in *Proc. Congr. Evol. Comput.*, 2001, pp. 995–1002.

[36] J. Kennedy and R. Eberhart, "An evolutionary approach for the tuning of a chess evaluation particle swarm optimization," in *Proc. IEEE Int. Conf. Neural Netw.*, 1995, pp. 1942–1948.

[37] D. E. Moriarty and R. Miikkulainen, "Evolving neural networks to focus minimax search," in *Proc. 12th Nat. Conf. Artif. Intell.*, 1994, pp. 1371–1377.

[38] T. P. Runarsson and S. M. Lucas, "Imitating play from game trajectories: Temporal difference learning versus preference learning," in *Proc. IEEE Conf. Comput. Intell. Games*, 2012, pp. 79–82.

[39] S. Samothrakis, S. M. Lucas, T. P. Runarsson, and D. Robles, "Coevolving game-playing agents: Measuring performance and intransitivities," *IEEE Trans. Evol. Comput.*, vol. 17, no. 2, pp. 213–226, Apr. 2013.

[40] J. Schaeffer, M. Hlynka, and V. Jussila, "Temporal difference learning applied to a high-performance game-playing program," in *Proc. Int. Joint Conf. Artif. Intell.*, 2001, pp. 529–534.

[41] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, and R. Lake, "Checkers is solved," *Science*, vol. 317, pp. 1518–1522, Sep. 2007.

[42] C. E. Shannon, "Programming a computer for playing chess," *Philosoph. Mag.*, vol. 41, no. 314, pp. 256–275, Mar. 1950.

[43] R. Storn and K. Price, "Differential evolution—A simple and efficient heuristic for global optimization over continuous spaces," *J. Global Optimiz.*, vol. 11, no. 4, pp. 341–359, 1997.

[44] G. Tesauro, "Practical issues in temporal difference learning," *Machine Learning*, vol. 8, nos. 3–4, pp. 257–277, 1992.

[45] W. Tunstall-Pedoe, "Genetic algorithms optimising evaluation functions," *ICCA J.*, vol. 14, no. 3, pp. 119–128, 1991.

[46] A. M. Turing, C. Strachey, M. A. Bates, and B. V. Bowden, "Digital computers applied to games," in *Faster Than Thought: A Symposium on Digital Computing Machines*, B. V. Bowden, Ed. London, U.K.: Pitman, 1953, pp. 286–310.

[47] M. A. Wiering, "TD learning of game evaluation functions with hierarchical neural architectures," Master's Thesis, Dept. Comput. Sci., Univ. Amsterdam, Amsterdam, The Netherlands, 1995.

[48] M. Wüllenweber, F. Friedel, and M. Feist, "Kramnik vs. DEEP FRITZ 10," *ICGA J.*, vol. 29, no. 4, pp. 208–213, 2006.

Omid E. David received the Ph.D. degree for his work on automatic learning of the parameters of a chess program's evaluation function and selective search mechanism through the use of genetic algorithms from Bar-Ilan University, Ramat-Gan, Israel, in 2009.

He is with Bar-Ilan University, where he teaches courses on evolutionary computation. He is also a Researcher and Consultant in machine learning and data mining (specifically, machine learning algorithms for big data). He is the developer of FALCON, a grandmaster-level chess playing program. His research interests include genetic algorithms, genetic programming, neural networks, computer games, machine learning for big data, image processing, and computer vision.

H. Jaap van den Herik studied mathematics at Vrije Universiteit, Amsterdam, The Netherlands, and received the Ph.D. degree from Delft University of Technology, Delft, The Netherlands, in 1983.

He is a Professor of computer science with Tilburg University, Tilburg, The Netherlands, and Professor of Law and Computer Science with the Faculty of Law, Universiteit Leiden, Leiden, The Netherlands. His research interests include computer games, serious games, adaptive agents, neural networks, information retrieval, big data, e-humanities, intelligent crowd sourcing, and intelligent systems for law applications.

Dr. Herik is a member of the TWINS and the Royal Holland Society of Sciences and Humanities. He has been an ECCAI Fellow since 2003.

Moshe Koppel received the Ph.D. degree for his work in logic and number theory from New York University, New York, NY, USA.

He is with Bar-Ilan University, Ramat-Gan, Israel, where he teaches computer science. His research interests include machine learning applications, especially in the area of text categorization.

Nathan S. Netanyahu is an Associate Professor with the Department of Computer Science, Bar-Ilan University, Ramat-Gan, Israel, and is also affiliated with the Gonda Brain Research Center, Bar-Ilan University, and the Center for Automation Research/UMIACS, University of Maryland, College Park, MD, USA. His research interests include algorithm design and analysis, computational geometry, evolutionary computation, image processing, pattern recognition, remote sensing, and robust statistical estimation. He has co-authored over 80 refereed papers. He is also the co-editor of the books *Computer and Games, 4th International Conference* (Springer Verlag, 2006) and *Image Registration for Remote Sensing* (Cambridge University Press, 2011).

Dr. Netanyahu has served as an Associate Editor for *Pattern Recognition*.