Evolutionary-based heuristic generators for checkers and give-away checkers

Jacek Mańdziuk*, Magdalena Kusiak and Karol Walędzik

Faculty of Mathematics and Information Science,
Warsaw University of Technology,
Plac Politechniki 1, 00-661 Warsaw, Poland

Abstract

Two methods of genetic evolution of linear and non-linear heuristic evaluation functions for the game of checkers and give-away checkers are presented in the paper. The first method is based on the simplistic assumption that a relation 'close' to partial order can be defined over the set of the evaluation functions. Hence explicit fitness function is not necessary in this case and direct comparison between heuristics (a tournament) can be used instead. In the other approach heuristic is developed step-by-step based on the set of training games. First, the end-game positions are considered and then the method gradually moves "backwards" in the game tree up to the starting position and at each step the best fitted specimen from the previous step (previous game tree depth) is used as the heuristic evaluation function in the alpha-beta search for the current step.

Experimental results confirm that both approaches lead to quite strong heuristics and give hope that more sophisticated and more problem-oriented evolutionary process might ultimately provide heuristics of quality comparable to those of commercial programs.

Keywords: game playing, heuristic generators, checkers, give-away checkers, evaluation function, evolutionary computation.

Preliminary version of the paper. The revised version was published in
EXPERT SYSTEMS, 24(4), 189-211, 2007, Blackwell Publishing.
The final version is available on request.

*Corresponding author. E-mail:mandziuk@mini.pw.edu.pl. This work was supported by the Warsaw University of Technology grant no. 504G 1120 0008 000

1 Introduction

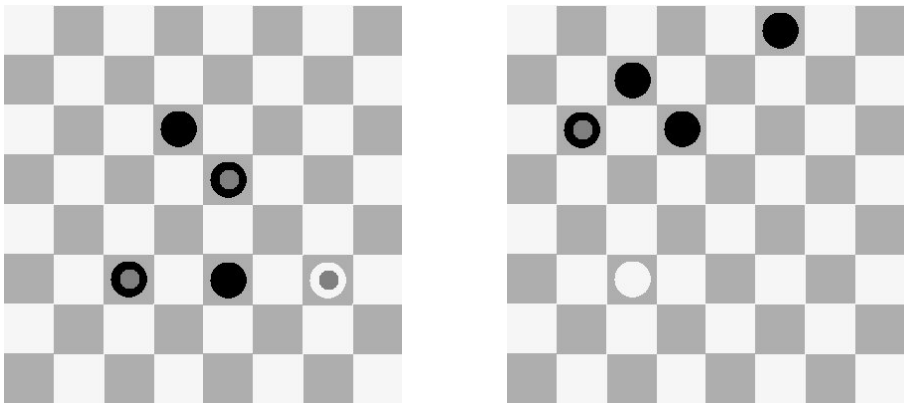
Games, from a purely scientific point of view, provide cheap, reproducible environments suitable for testing new search algorithms, pattern-based evaluation methods or learning concepts.

Since the seminal papers devoted to programming chess [39, 43, 24] and checkers [31] in the 1950s., games remained through decades an interesting topic for both classical AI and Computational Intelligence (CI) based approaches. In 1965 chess was even announced as "*the Drosophila of Artificial Intelligence*" by the Russian mathematician Alexander Kronrod.

Although most examples of application of CI methods to game playing make use of either reinforcement learning method or neural networks, there are also some papers devoted to evolutionary or hybrid neuro-genetic solutions, e.g. in chess [19, 14], in checkers [8, 2], in Othello [23], in Rummy [20], in tic-tac-toe game [8], in Iterated Prisoner's Dilemma [11, 8, 38], in Backgammon [30], in Poker [5, 6], and others.

The main idea of this paper is to test the efficacy of "pure" evolutionary approach to the problem of building evaluation functions in US checkers and US give-away checkers (GAC)[1].

The game of checkers is widely known and does not need any introduction. For those who are unfamiliar with this game and its rules we would recommend consulting one of the internet checkers playing sites. The game of GAC is played according to the same rules as checkers. Only the ultimate goals of both games differ. In order to win a game of GAC player has to lose all his/her pieces or be unable to perform a valid move [3]. Despite simplicity of the rules GAC is not a trivial game and results of brute-force algorithm using trivial strategy of losing pieces as quickly as possible are unsatisfactory. One of the reasons for unsuitability of this simple algorithm is the fact that a single piece is barely mobile and has very restricted choice of possible moves. Fig. 1 presents two situations in which white loses despite having only one piece left.



(a) Black to play and win. (b) White to play and lose.

Figure 1: Examples of inefficiency of greedy heuristics. White move bottom-up. Open circles denote kings.

[1]Since within the whole paper the US version of the game of checkers and the game of give-away checkers is considered henceforth we'll simply write *checkers* or *give-away checkers*, skipping *US*.

In the left figure the winning sequence for Black is the following (see Fig. 2 for board description):

1.	...	23-27	2.	24-31 (27)	22-26
3.	31-22 (26)	15-18	4.	22-6 (18,10)	0:1

In the right figure White has two choices of the first move, but in either case loses:

1.	22-17	10-14	1.	22-18	9-14
2.	17-1 (14,6,K)	9-6	2.	18-2 (14,6,K)	3-7
3.	1-10 (6)	3-7	3.	2-11 (7)	10-15
4.	10-3 (7)	0:1	4.	11-18 (15)	0:1

Certainly, the game of GAC is not as popular as checkers, but still there has been some interest in this game too [3, 27, 21, 37]. An example of a GAC playing program, which became the main opponent for heuristics developed in this paper, is described in more detail in section 7.1.

As opposed to GAC, there have been quite a lot of papers concerning checkers (or draughts) published in the literature (e.g. [31, 17, 1, 26, 40, 36, 2, 13]), but since they are not directly related to our work we'll not discuss them any further, except for a brief mention of the three major achievements in this field.

First of all, the history of checkers playing programs begins with the seminal papers of Arthur L. Samuel [31, 32]. Samuel's program was able to develop the evaluation function on its own. More precisely it was equipped with an a priori defined set of expert features potentially relevant for building board evaluation function and during the self-play phase was able to successfully define the essential subset of features and their weights in order to form polynomial evaluation function. Having spent enough time in the self-play stage the program achieved an expert level of play.

Another 'grand AI achievement' in the game of US checkers is Chinook program developed by Jonathan Schaeffer [34, 35, 36] which was able to successfully compete with the human absolute checkers genius - Marion Tinsley. Chinook uses a very well tuned hand-crafted evaluation function and is equipped with an end-game database covering all possible board positions of 8 or fewer pieces. Also, Chinook's search mechanisms are very sophisticated. Unlike Samuel's program, Chinook does not include any learning mechanism.

The third undisputable achievement in the field of checkers is Anaconda (also known as Blondie24) developed by Kumar Chellapilla and David Fogel [9, 10, 13]. Carefully designed evolutionary process over the ensemble of neural networks allowed Blondie24 to achieve the rating of an expert player on the internet checkers playing site *without any built-in human expert knowledge*. It is worth to note that in order to generate A-class level playing network (just below the expert level) only about 250 evolutions was required. After the

next 600 evolutions an expert level of play was achieved. Anaconda (Blondie24) is an apparent, successful example of learning from scratch using Computational Intelligence techniques.

Unlike in the above mentioned 'grand AI achievements' the piece of research described in this paper is not intended to create a program capable of competing against professional GAC programs (in fact the authors are unaware of any such software) or commercial checkers applications, but to test usefulness of "generic" genetic approach for generating evaluation functions in these two game domains. The underlying assumption is the simplicity and generality of proposed solutions. Consequently, fitness function is defined in a problem independent manner and straightforward genetic operators are applied. Moreover, a plain alpha-beta algorithm, with no particular optimization for speed, is used and a shallow search ($4-6$ plies) is implemented. No opening books or end-game databases are used.

There are a few reasons for considering these two particular games (checkers and give-away checkers). First of all, checkers is a very popular and widely known game. Next, the rules of both games are simple and at the same time none of the games has been solved and both require nontrivial heuristics. Finally, the possibility of checking the effectiveness of the proposed solutions to the problems defined over the same set of constraints (rules of the game) but with completely "opposite" goal functions (goals of the game) seems to be tempting and may possibly allow for some general (game independent) observations.

Hence, in order to prove the efficacy of evolutionary methods in game playing domain we have applied them to both games and in each case to both linear and non-linear evaluation functions. Evolved evaluation functions were tested against one another and also against some commercial and non-commercial software available on the internet or obtained via private communication.

For each type of heuristics (linear vs non-linear) two heuristic generators were used and compared. The first one relies on the simplistic assumption that the "winning relation" defined over the set of all heuristics is transitive. Therefore instead of using the extended fitness function a direct comparison between evolved heuristics is used.

In the other approach heuristic is developed gradually starting from the end-game positions. These positions are either leaves or nodes close to the leaves in the game tree and therefore can be estimated based on the game outcomes. This makes it possible to build fitness function for that phase of the game, and the best fitted specimen in that phase is used to define the fitness function for positions which are a few moves closer to the starting position (i.e. a few moves up in the game tree). Following this scheme the algorithm reaches the beginning of the game.

In order to provide justified and convincing conclusions we have decided to include in this paper several low-level implementation details. Some technical issues are thoroughly discussed especially when particular parameters seem to have an important impact on the quality of evolutionary process.

Even though it is hard to expect that "blind" evolutionary process together with a shallow game tree search may lead to construction of a grandmaster-level heuristic, and despite undisputable achievements of hard AI approaches in the domain of checkers, e.g. [36], the results presented in this paper strongly support the claim that *due to their biological soundness and self-improvement capabilities evolutionary methods deserve strong consideration in game playing domain.*

It is also reasonable to say that there is still some room for quality improvement of the evolved heuristics in case more sophisticated and more problem-oriented evolutionary process is applied.

The paper is organized as follows. In the next section the components of the evaluation functions are described. In section 3 two heuristic generators used to genetically develop the evaluation functions are presented and discussed. These are Heuristic Generator (HG) presented in sect. 3.1 and Simple Heuristic Generator (SHG) described in sect. 3.2. Next, in section 4 linear (sect. 4.1) and nonlinear (sect. 4.2) heuristics are described in detail with particular emphasis put on the way they are composed of simple components listed in sect. 2. Several low-level implementation details concerning evolutionary processes implemented in both heuristic generators as well as initial assessment of their performance are outlined in sections 5 and 6, resp. Numerical results, along with their in-depth discussion are presented in section 7, respectively for give-away checkers (in sect. 7.1) and checkers (in sect. 7.2). The last section is devoted to conclusions and directions for future development of this research.

2 Components of the Evaluation Functions

Heuristics described in this paper are relatively simple and make use of some or all of the following parameters, calculated separately for each player:

1. Number of pawns;
2. Number of kings;
3. Number of safe pawns (i.e. adjacent to the edge of the board);
4. Number of safe kings;
5. Number of moveable pawns (i.e. able to perform a move other than capturing).
6. Number of moveable kings. Parameters 5 and 6 are calculated taking no notice of capturing priority;
7. Aggregated distance of the pawns to promotion line;
8. Number of unoccupied fields on promotion line.

Heuristics could also consider sums of or differences in respective parameters for both players rather than raw numbers for each player separately. Once heuristics using the straightforward parameters listed above had been generated and tested, it was decided that it would be desirable to add more sophisticated parameters characterizing layout of the pieces on the board. The following parameters were, therefore, introduced:

9. Number of defender pieces[2] (i.e. the ones situated in two lowermost rows);
10. Number of attacking pawns (i.e. positioned in three topmost rows);
11. Number of centrally positioned pawns (i.e. situated on the eight central squares of the board);
12. Number of centrally positioned kings;
13. Number of pawns positioned on the main diagonal;
14. Number of kings positioned on the main diagonal;
15. Number of pawns situated on double diagonal;
16. Number of kings situated on double diagonal;
17. Number of loner pawns. Loner piece is defined as the one not adjacent to any other

[2]Please note, that within the whole paper the term "pieces" will denote "pawns and king together".

piece;
18. Number of loner kings;
19. Number of holes, i.e. empty squares adjacent to at least three pieces of the same color.

Apart from the above parameters six patterns were defined. They are described below using common notation presented in Fig. 2. Since only one instance of each pattern can exist for any of the players at the same time, features $20-25$ can take only boolean values.

20. Presence of a Triangle pattern(see Fig. 3(a)).
21. Presence of an Oreo pattern (see Fig. 3(b)).
22. Presence of a Bridge pattern (see Fig. 3(c)).
23. Presence of a Dog pattern (see Fig. 3(d)).
24. Presence of a pawn in corner (i.e. White (Black) pawn on square 29 (4), resp.);
25. Presence of a king in corner (i.e. White (Black) king on square 4 (29), resp.);

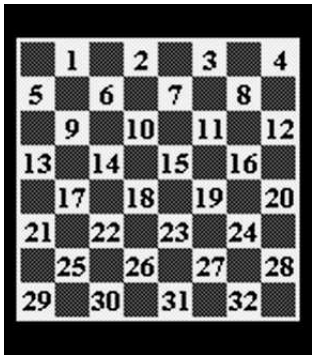


Figure 2: Notation used for describing patterns. White pawns are at the bottom.

Generally two types of heuristics were considered. Linear heuristics consisted of linear combination of the parameters listed above:

$$LinearCombinationOfParameters = a_1 \cdot param_1 + a_2 \cdot param_2 + \ldots + a_j \cdot param_j, \quad (1)$$

where $1 \leq j \leq 18$ and $param_1, \ldots, param_j$ were freely chosen from the above 18 parameters (being either raw numbers or sums or differences calculated for both players).

Nonlinear heuristics were composed of a small number (in our tests 3 or 5) of nonlinear components of the following form:

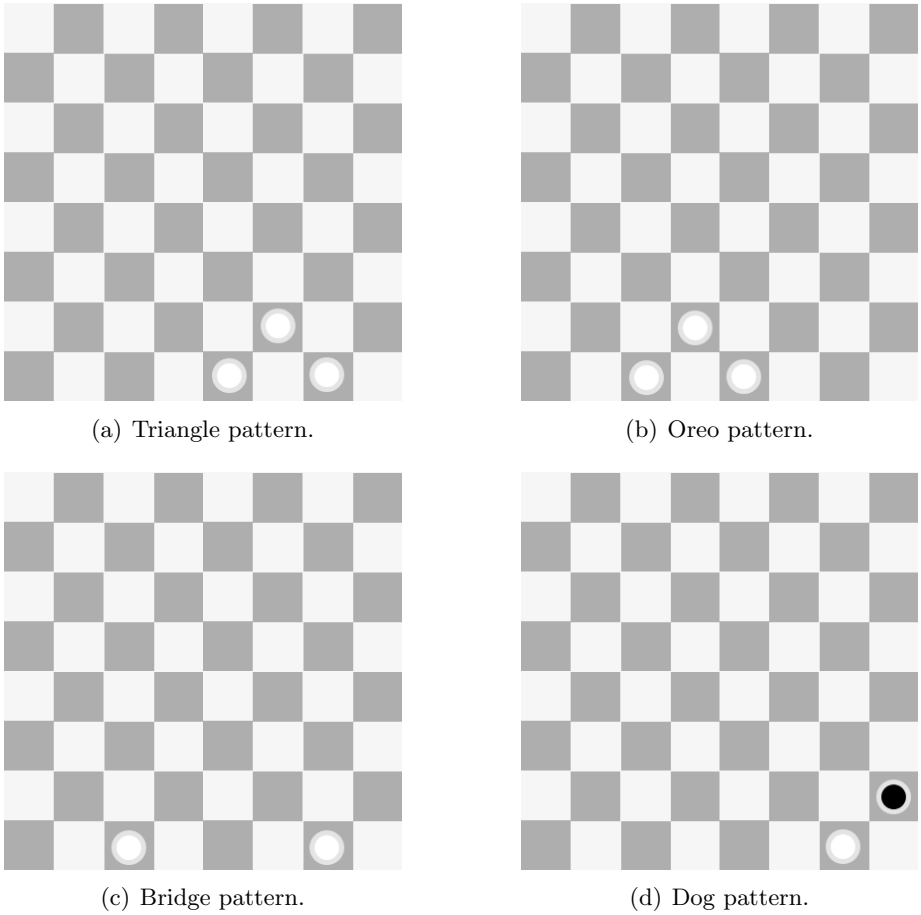
$$\begin{aligned} \text{IF[NOT]}\ (&min_1 \leq par_1 \leq max_1 \\ &\text{AND/OR}\ min_2 \leq par_2 \leq max_2 \\ &\text{AND/OR}\ \ldots min_i \leq par_i \leq max_i) \\ \text{THEN}\ &Heuristic\_Result+ = LinearCombinationOfParameters \end{aligned} \quad (2)$$

where again $1 \leq i, j \leq 18$ and $par_1, \ldots, par_i$ and $param_1, \ldots, param_j$ were defined over the set of the above mentioned parameters and $LinearCombinationOfParameters$ was defined as in (1). As it was the case with linear heuristics sums or differences of particular parameters calculated for both players could be used instead of raw numbers.

The final sets of parameters and the respective limits $min_k, max_k; 1 \leq k \leq i$ were chosen manually basing on preliminary tests.

In order to restrict the nonlinear condition space two additional constraints were imposed on (2), i.e. only one type of operator (AND or OR) could be used in a single condition (2) and negation could be applied to the whole condition only.

For either type of heuristics coefficients $a_1, \ldots, a_j$ in (1) were optimized in the evolutionary process described in the next section.



(a) Triangle pattern.

(b) Oreo pattern.

(c) Bridge pattern.

(d) Dog pattern.

Figure 3: Four out of six patterns used as heuristics' components. All patterns are described from the White player side.

3 Evolutionary Algorithms

Genetic algorithms were used to generate of coefficients $a_1, \ldots, a_j$ for parameters $param_1$,..., $param_j$ in (1). All variable parameters of a heuristic were represented as a vector of real numbers where each number denoted a single gene. Conditions that nonlinear components consisted of were not modified by the process of evolution. Two different approaches implemented are described in sect. 3.1 and sect. 3.2, resp.

3.1 Heuristic Generator (HG)

One of the difficulties encountered while designing the genetic algorithm was defining fitness function for the heuristics. *Unless an efficient evaluation function is given* any game position can be assessed correctly only after the whole game is played (i.e. the final result is known) and both sides play perfectly. But, since evolution of such function is actually our goal and the basic assumption that we made in this work is to use *zero expert knowledge approach*, we obviously can not posses such external 'efficient evaluation function". The general idea of how to solve this problem was described in [7]. According to [7] the game was partitioned into several disjoint stages based on the number of moves already performed. In the first phase of the algorithm a heuristic able to assess correctly situations close to the end of the game was obtained.

In order to achieve this, alpha-beta algorithm with no heuristic was used to assess a number of randomly generated positions close to the leaves of the full game tree (i.e. close to the end of the game). All positions beyond the depth of alpha-beta algorithm were considered a draw. Subsequently, each specimen assessed the same positions and its fitness was calculated according to the formulae:

$$\frac{n}{\sum (h_i - a_i)^2}, \tag{3}$$

where n denotes the number of test situations, h_i assessment of the i-th test situation by the heuristic specimen and a_i by the alpha-beta algorithm.

Additionally, when during the particular run of the algorithm speciation encouragement feature was enabled, fitness value of each of the specimens was at this point divided by the sum of sharing function values for all the specimens in the population. Sharing function was defined as $exp(-d^2)$, where d is Euclidean distance between the genotypes of two specimens. This was done as a mean to encourage speciation which in turn might lead to improved exploration of the problem space [15, 16]. Impact of sharing function was however ignored when selecting the best specimen in the population.

Since the games of checkers and give-away checkers have only three possible outcomes and the values of heuristics were expected to fall into a continuous interval, the depth of a leaf in the game tree was taken into consideration while assessing it. At first, the win value was defined as max_win_value and loss value as its opposite, where max_win_value was defined to be at least twice as big as the maximal depth (calculated from the root of the tree) of a game tree node examined by alpha-beta. Assuming that the current position was searched to depth g, the final assessment was equal to $max\_win\_value - g$ in case of a win or $g - max\_win\_value$ in case of a loss, or $zero$ in case of a draw. Hence, the absolute value of the assessment of any position other than a draw would always belong to the interval $(0.5 \cdot max\_win\_value,\ max\_win\_value)$. This method of including the depth of the searched tree in position assessment correlates with the observation that winning in shorter time (or loosing after longer time) should be assessed better than winning after more moves (loosing quickly)[3].

Once the initial stage had ended, worst fitted fraction (usually between 0.7 and 0.8) of the population was replaced by new random specimens. New board situations closer to the root of the game tree were generated and they were assessed by the alpha-beta

[3] A similar idea is used in Reinforcement Learning algorithms, where the reward/punishment is scaled by a discount factor exponentially decreasing in time [18, 42].

algorithm with the fittest specimen of the previous phase used as its heuristic function. The process continued until the root of the game tree was reached.

In each phase a constant fraction (between 0.35 and 0.4) of all the test boards came from the stage of the game closest to the beginning (i.e. the one considered at the moment). Depending on the settings of the algorithm, the positions closer to the leaves of the game tree were either regenerated and reassessed by the newest heuristic or once generated they were used throughout all subsequent phases.

Standard genetic operators: selection, mutation and crossover were implemented in the following way.

Selection – selection was done by means of tournaments. For each tournament a number (exact count depended on the algorithm settings - usually between 2 and 4) of specimens were randomly chosen (with return) from the population. Their fitness assessments were compared in order to determine the winner of the tournament. Winners of two such tournaments were coupled and went on to crossbreed.

Crossover – each pair of respective linear combinations in a heuristic (i.e. each non-linear component and base heuristic function) was crossed over independently. The genotype of each linear combination was randomly partitioned into two. The offspring[4] inherited values of each part of the genotype from one parent. The value of the gene on which the division was placed was randomly chosen from the interval defined by the values of this gene in parent specimens.

The weakest specimen in the population was to be replaced by the newly created descendant. However, in default configuration of the algorithm used throughout the tests carried out, the replacement would happen only if the new specimen's fitness evaluation was greater than that of the one to be replaced (with sharing function considered as usual).

Mutation – three kinds of mutation could occur in each of the genes of every newly created specimen: multiplying a value of a gene by two, dividing it by two or changing its sign. Multiplying or dividing a value by two were twice as probable as changing the sign[5]. Each gene of a specimen mutated independently. The overall probability of mutation for each gene was within the range $(5 \cdot 10^{-4}, 11 \cdot 10^{-4})$ for HG and $(6 \cdot 10^{-4}, 2 \cdot 10^{-3})$ for the other generator (SHG) - see the next section.

3.2 Simple Heuristic Generator (SHG)

The other approach to evolution of heuristic functions considered in this paper is based on a simplistic assumption that results of games played by pairs of specimens define a relation close to partial order. In order to compare the fitness of two specimens, they would play one or two (with sides swap) games against each other, depending on the algorithm settings. In order to strike the balance between speed and accuracy of such comparisons, the games would be played with a depth limit for alpha-beta of 3 (for simpler heuristics) or 4 (for more complex heuristic functions taking into considerations more parameters). If a draw occurred, a tie breaker heuristic could be used; however, it was switched off during all the tests. If, in case of two-game comparisons, each heuristic would score one win, the one winning in fewer moves would be considered the ultimate winner.

[4]Please note, that there was only ONE offspring from each pair.

[5]Instead of multiplication/division of gene's value also addition/subtraction within some range was tested, but results were poorer in that case.

Basing on the relation described above, it was relatively easy to compare and sort specimens within a small set and, subsequently, perform a tournament selection without any fitness function whatsoever.

The genetic operators used in the SHG algorithm bore great resemblance to those used in HG algorithm implementation. The only significant difference was the necessity to normalize specimens' genotypes (since there was no external factor withholding the linear combinations of weights from exploding). Two specimens to crossbreed were chosen by means of tournaments and additional tournament was held to determine the weakest specimen to be replaced by the offspring. As it was the case in HG, the new specimen survived only if it was fitter than the one to be replaced.

4 Types of Heuristics

Based on preliminary tests we have decided to inspect several types of heuristics in more detail. Some of them were linear and some consisted only of nonlinear components described above. Most of the heuristics were generated twice: once using HG and once with SHG.

Additional tests for the game of GAC using SHG were also carried out in order to find out whether values of respective parameters for both players in linear heuristics exhibited symmetry (i.e. evolved to approximately opposite values). Pawns counts and kings counts weights were found to be the most asymmetrical. The other parameters tended to evolve symmetrical weights. It was therefore recommended that for these parameters differences in respective values instead of raw values be used in order to decrease the number of genes. As for the two asymmetrical parameters, a hypothesis was proposed that the asymmetry was a result of dependencies between parameters and therefore using differences instead of individual values would affect the quality of heuristics in a negative way. On the other hand, it might also be the case that simple evaluation functions do not properly assess the quality of a position on the board and the asymmetry results from conflicting win criteria and mobility issues mentioned earlier. After thorough exploration of this issue we leaned towards the latter hypothesis. Further simulations confirmed the efficiency of using raw numbers instead of differences for these two above mentioned parameters, but only in case of the simplest heuristics (see comparison between 8F and 10F heuristics presented in the following sections). Generally, we concluded that using more parameters (i.e. more sophisticated heuristics) and first of all dividing the entire game into stages (each with dedicated evaluation function) is more advisable than using raw material values (see sect. 4.2). Hence in all heuristics described in this paper, except for 10F ones, all parameters are defined as differences in respective values calculated for both players.

4.1 Linear Heuristics

8 Factors (8F). This heuristic took into consideration differences in values of eight parameters, namely numbers of (1) pawns, (2) kings, (3) safe pawns, (4) safe kings, (5) moveable pawns, (6) moveable kings, (7) unoccupied fields on promotion line and (8) aggregated distance to promotion line.

10 Factors (10F). This heuristic took into account the same parameters as 8F the only exception being the fact that it considered raw numbers of pawns and kings for each player

separately (instead of differences).
15 Factors (15F). All factors of this heuristic were defined as differences of respective parameters calculated for both sides. Note, that this heuristic was defined differently for each game (the detailed description is presented in Table 1).

feature name	15F checkers	15F GAC	19F both	25F both	E3Ph checkers	E3Ph GAC
pawns	X	X	X	X	X	X
kings	X	X	X	X	X	X
aggregated distance to promotion line		X	X	X		X
save pawns	X	X	X	X	X	X
save kings	X		X	X		
attacking pawns	X	X	X	X	X	X
centrally positioned pawns	X	X	X	X	X	
centrally positioned kings	X	X	X	X	X	X
movable pawns		X	X	X		X
movable kings			X	X		
unoccupied fields on promotion line			X	X		
defender pieces	X	X	X	X		X
pawns on main diagonal			X	X		
kings on main diagonal		X	X	X		X
pawns on double diagonal			X	X		
kings on double diagonal		X	X	X	X	X
loner pawns		X	X	X		X
loner kings			X	X		
surrounded empty fields	X	X	X	X		X
Bridge patterns	X			X	X	
Oreo patterns	X			X	X	
Triangle patterns	X			X	X	
Dog patterns	X	X		X	X	X
'Man in corner' patterns	X	X		X		
'King in corner' patterns	X			X	X	

Table 1: List of factors defining **15F**, **19F**, **25F** and **E3Ph** heuristics generated for checkers and give-away checkers. A sign "X" in any field denotes inclusion of the respective feature in the heuristic function. 'Both' denotes that heuristic definition is identical for both games. Please note, that *all features are calculated as differences* (not the raw values).

19 Factors (19F). Again, all components of this heuristic were in the form of differences between the respective values calculated for both players. It took into account all the parameters considered by 8F heuristic. Apart from that it used some parameters characterizing layout of pieces on the board. It should be noticed that patterns (Dog, Bridge, Oreo, ... etc.) were not included in this heuristics' definition (see Table 1).
25 Factors (25F). This heuristic took into consideration differences in values of all available parameters described in sect. 2.

4.2 Nonlinear Heuristics

The general idea of nonlinear heuristics was based on the fact that it is advantageous to divide the entire game into several disjoint stages and to use different heuristics for different stages [36]. Two crucial moments requiring change of the heuristic evaluation function were identified. Firstly, presence of kings clearly indicates that the game has entered an advanced stage. Moreover, due to the mobility issues mentioned earlier, end-game positions also require defining and applying different heuristic.

Preliminary tests showed that introducing nonlinear components with conditions that were not disjoint hindered the genetic algorithm significantly. This was particularly the case for HG and resulted from the fact that having several overlapping conditions made it possible to achieve very similar results in many ways, each time with very different values of parameters.

The following non-linear heuristics were developed:

3Phase (3Ph). In this heuristic the game was divided into three stages:
Beginning − each player has more than 3 pawns and no kings are present on the board.
Kings − both opponents have more than 3 pieces and at least one king is present.
Ending − one of the players has at most 3 pieces left.

Nonlinear components corresponding to each stage were defined. Linear heuristics in each component took into consideration the differences in exactly the same 8 parameters as in 8F heuristic (parameters concerning kings were, of course, only considered if kings were present on the board). For example phase *Beginning* was encoded by the following pseudo-algorithm:

IF ABS(Players_pieces_count - 10.0) < 6.5 AND
ABS(Opponent's_pieces_count - 10.0) < 6.5 AND
ABS(Total_kings_count) < 0.5 THEN
Heuritic_Value $+= C_1 * Diff(1) + C_2 * Diff(2) + C_3 * Diff(3) + C_4 * Diff(4) + C_5 * Diff(5) + C_6 * Diff(6) + C_7 * Diff(7) + C_8 * Diff(8)$,

where $C_1, \ldots, C_8$ are evolvable coefficients, and $Diff(n)$ denotes the value equal to the difference of feature n (listed in sect. 2) between player and his opponent.

Expert 3 Phase (E3Ph). This heuristic was generated only using HG algorithm. The game was divided into stages in the same way as it was the case for 3Ph; however, apart from differences in numbers of pawns and kings, linear heuristics used in each of the stages took into consideration some parameters describing layout of pieces on the board (see Table 1 for detailed description). Again, parameters involving kings were not considered in the *Beginning* phase when no kings existed. Please note that this 'expert' heuristic was defined separately for checkers and GAC. For each game the respective set of contributing features was chosen basing on results obtained in previous runs of generators by means of choosing parameters which proved to be the most significant.

5 Phase (5Ph). In the last heuristic the division of the game was similar to the one used in 3Ph. The second stage (Kings) was subdivided into three stages: MyKings, HisKings and BothKings, depending on which player was in possession of kings. Linear heuristics used in each phase considered the same parameters as those in 3Ph.

5 Algorithm Settings

In the HG, the initial depth in the game tree was between 81 and 87 for GAC and between 82 and 88 for checkers. The intervals were determined basing on preliminary tests devoted to calculating the average number of moves necessary to finish a game performing random moves. The difference in depths between subsequent phases was set to 6 since it was agreed that alpha-beta search with depth limit of 6 was still reasonably fast.

In most tests the number of test boards ranged from 3 000 to 4 500 when reusing of boards was disabled and was equal to about 6 000 when this feature was enabled.

While fewer situations were assessed in each phase during the generator runs without reusing test boards, the total number of situations considered was greater which resulted in better exploration of the problem space. On the other hand, evaluating as many as 6 000 boards during each phase minimized the chance of considering too few situations belonging to certain categories and propagating the error upwards. On the other hand yet again, if for some reason an erroneous heuristic was generated at an initial stage the chance of gradually recovering from the error was relatively small, because boards assessed by the faulty heuristic were reused in subsequent phases.

Test populations consisted of 350 specimens. In each phase the weakest 80% of the population were regenerated. Depending on the number of genes of each specimen the size of a tournament was set between 2 and 5. The greater the number of specimens participating in a tournament, the quicker the convergence was. However, if the algorithm converged too quickly, evolution of more sophisticated heuristics could be seriously hindered.

For SHG each test population consisted of 100-150 specimens. Populations had to be smaller because of the way specimens were compared with each other. Comparisons were symmetrical, i.e. two specimens played two games against each other swapping sides after the first game. Depending on the complexity of a heuristic (i.e. on the number of genes in the longest linear combination) search depth in games played for comparison purposes was set to 3 (for simpler heuristics) or 4 (for more complicated ones). The latter led to the necessity to further reduce the number of specimens to $60 - 80$. In order to compensate for decreasing size of the population mutation probability was increased. Significant improvement in the quality of heuristics generated was noticed. However, despite substantially reducing size of the population the algorithm was still considerably slower when search depth was set to 4 than when it operated at the depth of 3.

6 Evolutionary Process

It was observed that HG tended to exhibit quicker convergence than SHG and that heuristics generated by the former were generally of slightly better quality. However, HG turned out to be very sensitive to changes in algorithm parameters. For instance, evolution of more complex heuristic used to converge too quickly if the size of tournament was too big. This resulted in the whole population reaching a local minimum which in turn led to heuristics of relatively poorer quality.

Preliminary tests showed that using speciation for nonlinear heuristics led to worse results. The probable reason for this was the fact that the impact of the speciation on a specimen's fitness could prevail over the influence of one of the categories not adequately represented in the test boards set. This might in turn lead to improper values of one of

the nonlinear components which might be propagated upwards.

Tests also showed that HG genetic algorithm could be hindered significantly by introducing nonlinear components with conditions that were not disjoint. The reason for that was that overlapping conditions allowed for obtaining very similar assessments (and, therefore, fitness function values) with very different genotypes.

Mutations seem to have had virtually no influence on the results of evolution of simple linear heuristics using HG. Significantly less than 1% of the fittest specimens (logged by the generator every 50 generations) turned out to have experienced mutation. The same was the case for more sophisticated linear heuristics taking into consideration some parameters concerning layout of pieces on the board.

As for nonlinear heuristics generated using HG, the fraction of mutated specimens logged was substantially higher than for linear ones but still did not exceed 10%. Such a significant difference might result from the fact that in initial stages of the algorithm some genes were not applicable to the situations evaluated and therefore their mutation had no influence on the overall fitness of the specimen.

As far as SHG is concerned, hardly any mutated specimens were logged. However, this probably stemmed from the fact that the fittest specimen was only chosen every 1 000 crossbreedings due to high cost of comparisons. When analyzing changes in intervals to which values of different genes belonged, it may be noticed that mutations quite frequently led to improved fitness. This was particularly the case for more sophisticated heuristics with larger numbers of genes and smaller populations.

Since in all tests newly created specimen would replace the weakest specimen in the population (in SHG the specimen to be replaced was chosen by means of a tournament) only if it was fitter than the specimen to be replaced, the ratio of *effective* crossovers (i.e. the ones in which a resulting specimen was actually added to the population) to *potential* crossovers was also investigated. It turned out that the fraction remained fairly stable throughout the process. About 80%-90% of all crossovers were effective in HG. The stability resulted from the fact that in the initial stages of the algorithm convergence was comparatively quick and therefore descendants tended to be fitter than specimens from previous generations. On the other hand, in the final stages vast majority of the specimens were almost identical and there were virtually no difference in fitness between ancestors and descendants in which case new specimens were preferred and added to the population.

As for SHG the ratio in question was around 60%-70% when search depth in games played for the purpose of comparison was set to 3 and 80%-90% when it was set to 4. This confirms the intuition that comparisons at the depth of 4 are more accurate, provide more meaningful results and minimize the impact of randomness.

The convergence of the evolution is clearly illustrated by changes in lengths of intervals for different parameters as well as by distinct declines in their variances. For most parameters variances dropped by more than a thousand times in the course of evolution. In particular, in HG initial values of parameters were randomly selected from the interval $(-100, 100)$ and after the evolution (i.e. in the final population) most of them were between -1 and 1 with standard deviation between 10^{-30} and 10^{-26}. Similarly, for SHG initial values were located between $(-0.5, -0.3)$ and $(0.3, 0.5)$ - due to normalization procedure with standard deviation about 0.1. After the evolutionary process was completed standard deviation dropped to about 10^{-5}.

7 Results

This section provides numerical results of several tests performed in order to estimate the quality of HG and SHG approaches. First, the results for GAC are presented in sect. 7.1. The results for checkers are placed in sect. 7.2.

7.1 Give-Away Checkers

Due to the lack of GAC playing programs reliable estimation of the quality of heuristics developed in our experiment is not an obvious task. The heuristics were first tested against the TD-GAC program available to the authors. The results of this comparison are presented in section 7.1.1. Another way to estimate heuristics' quality is their comparison with the null heuristic as suggested by Jonathan Schaeffer [37]. This approach is discussed in sect. 7.1.2.

7.1.1 Comparison with TD-GAC program

In order to check the quality of heuristics generated for the game of GAC, they were tested against TD-GAC program [28, 22, 27, 29]. TD-GAC is based on temporal difference approach [41] and learns from the games played. Starting from scratch several TD players were trained in a three-step procedure (with subsequently stronger opponents - trainers) with the set of features following Samuel's checkers approach [31, 32]. The strongest of the above players became the opponent for heuristics discussed here.

Each heuristic played 40 games against the program, in half of them playing white and in the other half black stones. During the tests alpha-beta search depth was set to 6 and search depth of TD-GAC was set to 4 since it made use of much more sophisticated parameters, including indirect exploration of the game tree one ply further. TD-GAC learning function was switched off during the tests. Due to some randomness in searching the game tree implemented in alpha-beta, games played between any particular heuristic and TD-GAC were not identical.

Most heuristics proved to perform well against the program with frequent wins and few losses. Results of the heuristics' tournaments against TD-GAC are shown in Fig. 4. Each heuristic was assigned 1 point for a game won and $\frac{1}{2}$ point for a draw.

As it can be seen in the figure, nonlinear heuristics (in particular E3Ph and 3Ph) tended to outperform the linear ones. Similarly, heuristics generated using HG performed generally better than those evolved using SHG, which suggests that non-transitiveness of SHG specimens' order relation as well as its smaller population size may hinder the evolutionary process. Poor outcomes of the 5Ph heuristics might result from their greater complexity which might require much bigger population, more sophisticated design of the evolutionary process and better tuning of algorithm parameters.

7.1.2 Comparison with null heuristic

Another possible way of checking the efficacy of evolved heuristics is to compare them with a null heuristic. This approach was suggested by Jonathan Schaeffer who did a similar experiment and observed that for deep search levels it is extremely hard to beat a null heuristic in this particular game [37].

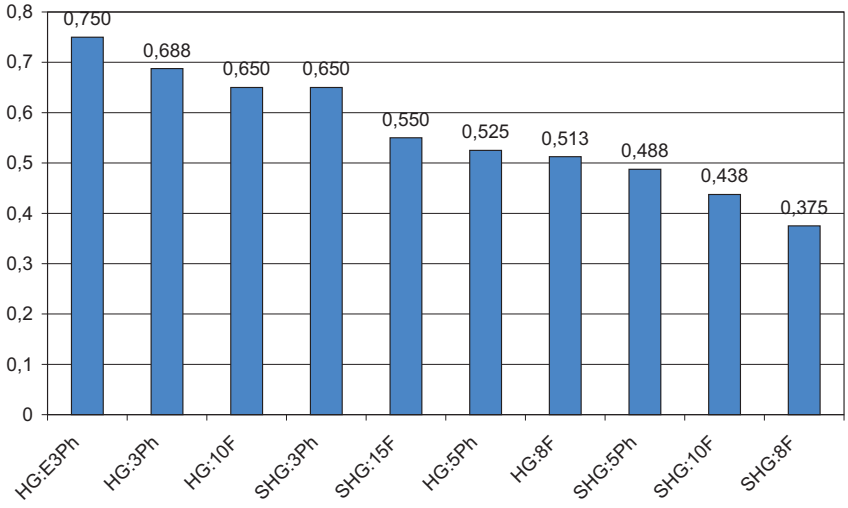


Figure 4: Performance of the top-10 heuristics against TD-GAC. The y-axis represents the points scored by each heuristic as a fraction of maximum possible score.

There are several ways to implement a null-heuristic approach in GAC. The most obvious one is to use a regular alpha-beta approach with a random estimation of the nodes[6]. This approach however may lead to several random alpha-beta cut-offs, among which good continuations could be lost. Hence, we have taken a different approach: in each node we examined the outgoing branches in random order. Whenever search depth limit was reached without encountering a terminal position, node was assessed as a draw (i.e. value 0). That way, as long as no leaves were encountered, absolutely no cut-offs were performed and the last (i.e. random - considering sorting procedure) branch was chosen in each of the nodes expanded.

The first comparison was made for the search depth equal to 10 (for both evolved and null heuristics)[7]. The results were very encouraging (see Fig. 5): 9 out of 15 tested heuristics scored at least 75% of all available points in a 20-game match against the null heuristics playing half of the games with white stones and half of them with black ones. None of the heuristics was outperformed by the null one (the weakest heuristic achieved the result of 50%).

In the next step we compared the heuristics playing at depth 6 with the null heuristic remaining at depth search equal to 10. 4-ply difference in search is of course a significant bias towards null heuristic but, surprisingly enough, the null heuristic was able to defeat only two evolved ones and achieved a draw with one heuristic. The remaining 12 of our heuristics scored between 55% and 77.5% points - again in a 20-game match with swapping sides after each game.

The above results suggest that the quality of evolved heuristics is relatively high. Considering the 4-ply depth search difference in the last experiment it may also be suggested

[6]Certainly, if a node represents the position of the end of a game (win, loss, draw) then the real (not random) value should be returned

[7]Please recall that as previously explained all tested heuristics were developed at search depth equal to 4 and application search depth was increased only during the experiment against null heuristic.

that evolved heuristics have potential for generalization and therefore can be successfully applied to different search depths.

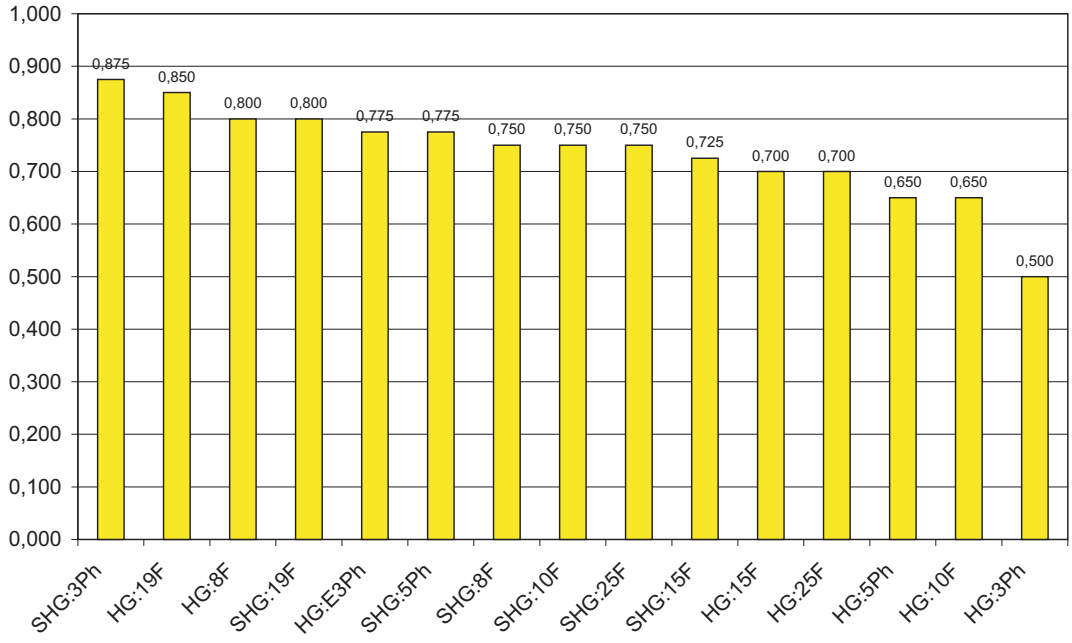


Figure 5: Comparison of evolved heuristics and the null heuristic in 20-game matches with depth search equal to 10. Bars denote percentage of points scored by each heuristic.

7.2 Checkers

In the game of checkers some initial tests were carried out to measure performance of the simple alpha-beta algorithm implemented. Depending on heuristic, 55 000 to 100 000 nodes were analyzed per second and search with depth limit set to 6 would take on average 40 to 200 ms. However, this average was calculated basing on games that were mainly considered ties after 140 plies performed and therefore many of the searches were performed during end-games when branching factor of the game tree was significantly higher than on average. Therefore alpha-beta might be expected to produce results up to 5 times faster for mid-game positions.

Another factor examined was alpha-beta algorithm's pruning efficiency, defined as $1 - \frac{n}{s}$, where n is the actual number of nodes analyzed and s denotes theoretical size of the analyzed game subtree calculated basing on reported average branching factor and search depth. With search depth limit of 6, this parameter would usually fall into a range of 0.7 to 0.85, and up to 0.99 for searches with higher depth limits.

7.2.1 Comparison of heuristics against one another

In order to determine relative quality of heuristics a tournament was held. During the tournament each heuristic played 10 games with each other with sides swap after each

game. Since 15 heuristics were tested, 1 050 games were played in total. Analogously to the case of GAC, due to some randomness in searching the game tree implemented in alpha-beta, games played between any two heuristics were pairwise different.

Two classifications were used. The first one was based on the total number of games won, tied and lost by each heuristic. The other one took into consideration results of whole 10-game clashes between heuristics. In either case, heuristic would gain 1 point for a win and $\frac{1}{2}$ point for a draw. In order to balance the influence of heuristic and random factor, while at the same time stay within reasonable time constraints all the games were played with alpha-beta's search depth limit of 6. The results are presented in Fig. 6. Outcomes of the tournament show clearly that heuristics generated by HG outperformed those produced by SHG algorithm, regardless of the comparison method (individual games or clashes). The probable reason was the fact that partial order relation defined for the SHG application does not necessarily reflect the quality of the specimens and, what may be even more important, is in many cases non-transitive. Moreover, HG algorithm, thanks to its speed, is able to operate on 3 times bigger populations than SHG given the same amount of time.

As might be expected the E3Ph heuristic proved to perform better than any other, no matter which comparison criteria were considered. More advanced heuristics, considering more sophisticated parameters, also tended to perform well in the tournament which again was in line with the expectations. Surprisingly poor score of the SHG 5Ph heuristic (located below the top-10 players) is a clear sign of SHG generator's inefficiencies when dealing with more complicated specimens.

7.2.2 Comparison with public domain and commercial programs

As it was mentioned before, our application was not meant to compete against commercial checkers programs. Its main purpose was to assess credibility of evolutionary approach to heuristic generation. It was developed with flexibility rather than speed in mind and as a result it assessed between 55 000 and 100 000 nodes per second whereas professional checkers applications were sometimes more than one order of magnitude faster. This is why the quality of the application should not be directly evaluated basing only on the results of games played against commercial applications. Nevertheless, in order to assess quality of the heuristics some comparison games were played. In order to make them fair, most of the features of the commercial programs (opening book, ending databases, hashtables, killer moves identification etc.) had to be disabled or reduced as much as possible. Seven best heuristics were chosen for those tests: HG:E3Ph, HG:15F, HG:25F, HG:19F, HG:10F, HG:3Ph and SHG:15F.

The first checkers engine that was tested against the generated heuristics was Simple Checkers [12]. Although it is a relatively simple program with publicly available source code, it is at the same time one of the best engines among those not making use of opening books and end-game databases and it surpasses many commercial applications [25]. Since Simple Checkers is open-source program which compiles to a *dll* file, it was possible to automate test games between our heuristics and Simple Checkers. In order to make them fair, Simple Checkers algorithm was slightly modified so that it would always perform search exactly to the depth of 6 plies, i.e. the same depth as our alpha-beta algorithm. Simple Checkers generally appeared to be too strong opponent and HG:E3Ph turned out to be the only heuristic capable of winning at least one of 50 games played, but on the

other hand it was capable of drawing distinctly more than half of the rest, achieving the final score of 37%, i.e. 1 win, 35 ties, 14 loses. The next two heuristics were SHG:15F and HG:15F scoring 10% and 9%, resp. It can therefore be stated that, although none of the heuristics managed to surpass Simple Checkers, the best one of them, being able to at least draw more than 70% of the games, may be regarded as one of comparable quality. It is reasonable to hope that well-guided evolutionary process could lead to generation of even better performing heuristics, capable of reaching Simple Checkers' level of play.

In the next tests HG:E3Ph heuristic, the undoubtedly best of all heuristics generated, played against shareware version of commercial application Actual Checkers 2000A [4]. Actual Checkers program was set to the highest possible difficulty level and was expected to perform a move in 3 seconds on average, which resulted in searches of depth 12 to 18. Our alpha-beta algorithm could analyze up to 8 million nodes but the actual number of nodes analyzed was usually well below 1 million per move, which was equivalent to search depth of approximately 9 plies during most of the mid-game but significantly less during end-game. In spite of lower search depth HG:E3Ph managed to draw 3 of 4 games played. Single games played with search depths of 9 and 10 ended in draws as well. During the tests it was observable that the HG:E3Ph heuristic would perform noticeably worse during end-games, often failing to take full advantage of its upper-hand or defend a draw well enough. This probably stemmed from the fact that Actual Checkers' hashtables proved especially useful in this phase of the game.

Finally, in order to once again confirm quality of the HG:E3Ph heuristic, games against another shareware edition of commercial checkers program Mad Checkers [33] were played. This time alpha-beta's depth limit was set to 8 as usual, but Mad Checkers engine, being less acclaimed application, was given 0.5s for its move, which was not significantly less than it usually took our program to assess mid-game situations. In order to make the competition even easier for Mad Checkers, alpha-beta was switched to depth limit of 7 (or in some cases even 6) whenever it happened to perform too slowly during endgames. Nevertheless, our heuristic managed to win 3 out of 4 games with one being a draw because of position repetitions, although Mad Checkers was left with 3 pieces against our heuristic's 5. For the second and final 4-game tournament against Mad Checkers our alpha-beta algorithm was switched to depth limit of 6 plies whilst Mad Checkers was still admitted 0.5s for a move (i.e. on average more than two times longer than our program would require for a move). This time our heuristic managed to win 1 game and draw 3 others (with one of the draws resulting, again, from heuristic's inability to make proper use of the fact that it outnumbers opponent by 2 kings).

8 Conclusions

The focus of this paper is to test usefulness of pure genetic approach for generating evaluation functions in two game domains: GAC and checkers. The underlying assumption is simplicity and generality of proposed solutions which results in using straightforward genetic operators and shallow game tree search with plain alpha-beta algorithm. In particular, no alpha-beta search enhancements (transposition tables, history heuristics, killer moves, etc.) and no opening books or end-game databases are used.

Under the above assumptions one cannot expect to create a master playing program capable of competing with strong humans or dedicated commercial software. What could

however be expected is a decent level of play and possibility of drawing several conclusions concerning generalization abilities (due to shallow search level) and efficacy of proposed genetic approaches.

Unlike the most of top-level playing programs *both generators proposed in this paper are problem independent* and therefore the observations attained for GAC and checkers may possibly be valid to a wide range of two-player perfect information board games. In our opinion, especially the Heuristic Generator (HG) deserves recommendation and further community interest, in particular in *zero-knowledge approaches*.

Generally, the results support the following four conclusions:

Firstly, *HG is definitely a more efficient generator* compared to SHG. For the game of checkers the top-5 best heuristics were produced by HG (see Fig. 6). Comparisons against commercial programs were also more favorable for HG. In give-away checkers the three winning heuristics against TD-GAC program were HG:E3Ph, HG:3Ph and HG:10F. Only comparison against null heuristic was 'inconclusive' with a little favor for SHG. Generally, the results support the claim that careful and tuned implementation of HG may lead to high quality heuristic evaluation functions for both games.

Secondly, for both games *it is advantageous to divide the entire game into phases* and develop separate heuristics for each part of the game. This is certainly a well-known recommendation in case of checkers and other board 'mind' games, but in our particular genetic implementation it was observed that game phases need to partition board positions space into *disjoint sets*. Otherwise the genetic process may have difficulties in assigning coefficients for the features shared by two or more game phases.

Thirdly, the general guideline is *to use differences of respective parameters calculated for both sides* rather than raw numbers separately for both players. This observation was fully confirmed, with the only exception being the case of relatively simple heuristics, where it is recommended that the number of pawns and the number of kings be input as raw values (c.f. comparison between 8F and 10F heuristics).

Fourthly, one of potential weaknesses of the two proposed generators might have been a shallow search depth used during the evolution phase (usually equal to 4), but the results of comparison with null heuristic in GAC and comparisons with several checkers programs suggest that, on the contrary, even though heuristics were generated based on 4-ply searches, they were capable of efficient generalization at greater search depths (from 6 to 10 in our tests).

Several other more specific conclusions can also be drawn from numerical results and implementation details. We hope that detailed presentation in this paper of several 'technical' issues may be helpful to other researchers working in the area of games and computational intelligence paradigms.

To summarize, achieved outcomes suggest that usage of genetic algorithms may be a credible way of producing heuristic functions for two-player games. Although heuristics described in this document did not surpass or even reach the level of play of those created during years of checkers programs development, they are still a challenge for at least some of the commercially available programs. This indicates that evolutionary approach, tuned by a careful choice of settings and meta-rules can prove very useful and successful, in particular for games, for which not enough explicit expert knowledge exists.

References

[1] Akl, S.: Checkers-playing programs. In Shapiro, S.C., ed.: Encyclopedia of Artificial Intelligence. Volume 1. John Wiley & Sons, New York (1987) 88–93

[2] Aleksander, I.: Neural networks - evolutionary checkers. Nature **402** (1999) 857

[3] Alemanni, J.B. http://perso.wanadoo.fr/alemanni/ give_away.html (1993)

[4] Atlant_Software_Inc.: Actual checkers 2000a program (2005)

[5] Barone, L., While, L.: An adaptive learning model for simplified poker using evolutionary algorithms. In: Proceedings of the Congress of Evolutionary Computation (GECCO-1999). (1999) 153–160

[6] Barone, L., While, L.: Adaptive learning for poker. In: Proceedings of the Genetic and Evolutionary Computation Conference. (2000) 566–573

[7] Borkowski, M.: Analysis of algorithms for two-player games. M.Sc. Thesis, Warsaw University of Technology (in Polish) (2000)

[8] Chellapilla, K., Fogel, D.: Evolution, neural networks, games, and intelligence. Proceedings of the IEEE **87** (1999) 1471–1496

[9] Chellapilla, K., Fogel, D.: Anaconda defeats Hoyle 6-0: A case study competing an evolved checkers program against commercially available software. In: Congress on Evolutionary Computation, La Jolla, CA, USA. (2000) 857–863

[10] Chellapilla, K., Fogel, D.: Evolving a neural network to play checkers without human expertise. In Baba, N., Jain, L., eds.: Computational Intelligence in Games. Volume 62. Springer Verlag, Berlin (2001) 39–56

[11] Darwen, P., Yao, X.: On evolving robust strategies for iterated prisoner's dilemma. Volume 956 of LNCS., Springer (1995) 276–292

[12] Fierz, M.: Simple checkers program (2005)

[13] Fogel, D.B.: Blondie24: Playing at the Edge of Artificial Intelligence. Morgan Kaufmann (2001)

[14] Fogel, D., Hays, T., Hahn, S., Quon, J.: A self-learning evolutionary chess program. Proceedings of the IEEE **92** (2004) 1947–1954

[15] Goldberg, D.E., Richardson, J.: Genetic algorithms with sharing for multimodal function optimisation. In Grefenstette, J.J., ed.: Proceedings of the 2nd International Conference on Genetic Algorithms, Cambridge, MA, USA, Lawrence Erlbaum Associates (1987) 41–49

[16] Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Pub. Co. (1989)

[17] Griffith, K.: A comparison and evaluation of three machine learning procedures as applied to the game of checkers. Artificial Intelligence **5** (1974) 137–148

[18] Kaelbling, L., Littman, M., A.W.Moore: Reinforcement learning: A survey. Journal of Artificial Intelligence Research **4** (1996) 237–285

[19] Kendall, G., Whitwell, G.: An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In: Proceedings of the 2001 Congress on Evolutionary Computation CEC2001, IEEE Press (2001) 995–1002

[20] Kotnik, C., Kalita, J.K.: The significance of temporal-difference learning in self-play training td-rummy versus evo-rummy. In Fawcett, T., Mishra, N., eds.: Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), Washington, DC, USA, AAAI Press (2003) 369–375

[21] Kusiak, M., Walędzik, K., Mańdziuk, J.: Evolution of heuristics for give-away checkers. In Duch, W., et al., eds.: Artificial Neural Networks: Formal Models and Their Applications - Proc. ICANN 2005, Part 2, Warszawa, Poland. Volume 3697 of LNCS., Springer (2005) 981–987

[22] Mańdziuk, J., Osman, D.: Temporal difference approach to playing give-away checkers. In Rutkowski, L., et al., eds.: 7th Int. Conf. on Art. Intell. and Soft Comp. (ICAISC 2004), Zakopane, Poland. Volume 3070 of LNAI., Springer (2004) 909–914

[23] Moriarty, D.E., Miikkulainen, R.: Discovering complex othello strategies through evolutionary neural systems. Connection Science **7** (1995) 195–209

[24] Newell, A., Shaw, J., Simon, H.: Chess-playing programs and the problem of complexity. IBM Journal of Research and Development **2** (1958) 320–335

[25] Newell, B.: Checkers programs repository (2005)

[26] Oldbury, D.: AI (Any Interest) for draughts? In Levy, D., Beal, D., eds.: Heuristic Programming in Artificial Intelligence: the first computer olympiad. Ellis Horwood, Chichester, UK (1989) 165–175

[27] Osman, D., Mańdziuk, J.: Comparison of tdleaf(λ) and td(λ) learning in game playing domain. In Pal, N.R., et al., eds.: 11th Int. Conf. on Neural Inf. Proc. (ICONIP 2004), Calcutta, India. Volume 3316 of LNCS., Springer (2004) 549–554

[28] Osman, D., Mańdziuk, J.: TD-GAC. http://gac-arena.gt.pl/ (2004)

[29] Osman, D., Mańdziuk, J.: TD-GAC: Machine Learning experiment with give-away checkers. In Dramiński, M., Grzegorzewski, P., Trojanowski, K., Zadrożny, S., eds.: Issues in Intelligent Systems. Models and Techniques. Exit (2005) 131–145

[30] Pollack, J.B., Blair, A.D., Land, M.: Coevolution of a backgammon player. In Langton, C.G., Shimokara, K., eds.: Proceedings of the Fifth Artificial Life Conference, MIT Press (1997) 92–98

[31] Samuel, A.L.: Some studies in machine learning using the game of checkers. IBM Journal of Research and Development **3** (1959) 210–229

[32] Samuel, A.L.: Some studies in machine learning using the game of checkers II - recent progress. IBM Journal of Research and Development **11** (1967) 601–617

[33] Sapphire_Games: Mad checkers program (2005)

[34] Schaeffer, J., Culberson, J.C., Treloar, N., Knight, B., Lu, P., Szafron, D.: A world championship caliber checkers program. Artificial Intelligence **53** (1992) 273–289

[35] Schaeffer, J., Lake, R., Lu, P., Bryant, M.: Chinook: The world man-machine checkers champion. AI Magazine **17** (1996) 21–29

[36] Schaeffer, J.: One Jump Ahead: Challenging Human Supremacy in Checkers. New York: Springer-Verlag (1997)

[37] Schaeffer, J. *Private communication* (2005)

[38] Seo, Y.G., Cho, S.B., Yao, X.: Exploiting coalition in co-evolutionary learning. In: Proceedings of the 2000 Congress on Evolutionary Computation. Volume 2., IEEE Press (2000) 1268–1275

[39] Shannon, C.E.: Programming a computer for playing chess. Philosophical Magazine **41 (7th series)** (1950) 256–275

[40] Smeets, J., Putter, G.: Some experience with a self-learning computer program for playing draughts. In Levy, D., Beal, D., eds.: Heuristic Programming in Artificial Intelligence: the first computer olympiad. Ellis Horwood, Chichester, UK (1989) 176–194

[41] Sutton, R.: Learning to predict by the method of temporal differences. Machine Learning **3** (1988) 9–44

[42] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA (1998)

[43] Turing, A.M.: Digital computers applied to games. In Bowden, B.V., ed.: Faster than thought: a symposium on digital computing machines. Pitman, London, UK (1953)

Appendix 1. Example game of HG:E3Ph vs Actual Checkers

An example of a game played by **HG:E3Ph (Black)** against **Actual Checkers (White)**. HG:E3Ph searches 10 plies, Actual Checkers nominally makes a move in 3s, which is equivalent to searches between 12 and 18 plies. HG:E3Ph is making the first move.

1.	11-16	24-19	2.	9-13	28-24
3.	16-20	22-18	4.	8-11	18-14
5.	10-17 (14)	21-14 (17)	6.	6-10	25-21
7.	10-17 (14)	21-14 (17)	8.	1-6	29-25
9.	6-10	25-21	10.	10-17 (14)	21-14 (17)
11.	2-6	30-25	12.	6-10	25-21
13.	10-17 (14)	21-14 (17)	14.	13-17	26-22
15.	17-26 (22)	31-22 (26)	16.	7-10	14-7 (10)
17.	3-10 (7)	22-17	18.	11-16	23-18
19.	16-23 (19)	18-14	20.	23-26	14-7 (10)
21.	26-11 (K)	17-14	22.	31-26	7-3 (K)
23.	26-22	14-10	24.	22-18	10-6
25.	18-15	6-1 (K)	26.	15-10	24-19
27.	4-8	27-23	28.	8-11	23-18
29.	5-9	1-5	30.	9-14	18-9 (14)
31.	11-16	19-15	32.	10-19 (15)	3-7
33.	20-24	7-11	34.	16-20	9-6
35.	19-23	5-9	36.	24-28	6-1 (K)
37.	20-24	9-14	38.	23-26	14-10
39.	24-27	32-23 (27)	40.	26-19 (23)	11-15
41.	19-23	1-6	42.	28-32 (K)	6-9
43.	32-28	9-14	44.	23-19	15-24 (19)
45.	28-19 (24)	10-7	46.	12-16	7-11
47.	16-20	14-10	48.	20-24	11-15
49.	19-23	10-7	50.	23-19	7-10
51.	19-23	10-7	52.	23-19	7-10
53.	19-23	$\frac{1}{2}$:$\frac{1}{2}$			

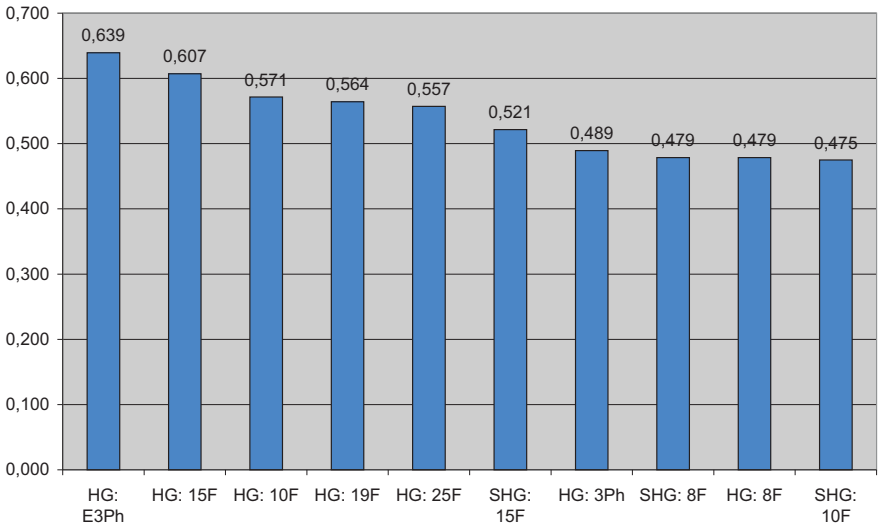
Three-move repetition draw in equal position.

Appendix 2. Example game of HG:E3Ph vs Mad Checkers

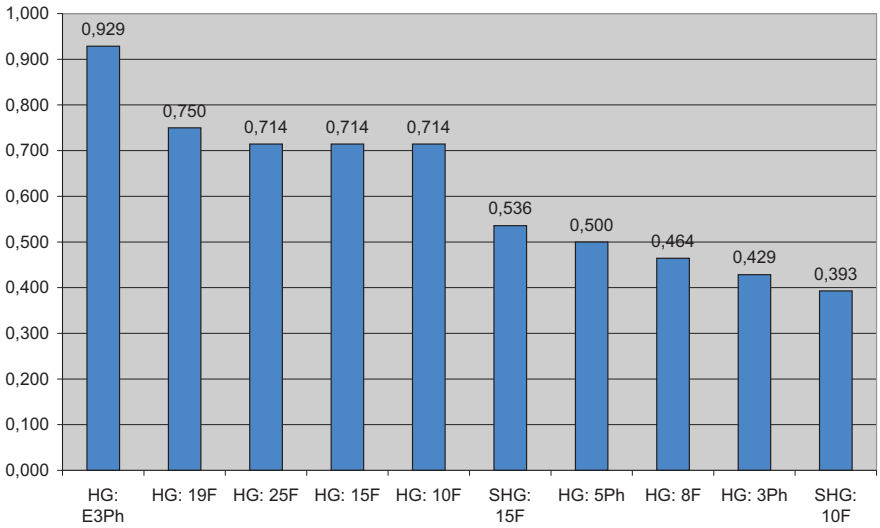
An example of a game played by **HG:E3Ph (Black)** against **Mad Checkers (White)**. Both sides use in average 0.5s per move. Mad Checkers is making the first move.

1.	23-19	9-13	2.	22-18	11-16
3.	18-14	10-17 (14)	4.	21-14 (17)	16-23 (19)
5.	27-18 (23)	8-11	6.	25-22	11-16
7.	22-17	13-22 (17)	8.	26-17 (22)	16-20
9.	32-27	6-9	10.	30-26	1-6
11.	29-25	9-13	12.	25-22	12-16
13.	26-23	4-8	14.	31-26	6-10
15.	24-19	8-12	16.	28-24	3-8
17.	18-15	7-11	18.	14-7 (10,K)	11-25 (15,22)
19.	17-14	2-11 (7)	20.	26-22	25-29 (K)
21.	22-18	13-17	22.	14-9	5-14 (9)
23.	18-9 (14)	17-21	24.	9-6	29-25
25.	6-2 (K)	25-22	26.	19-15	11-18 (15)
27.	23-14 (18)	22-18	28.	14-9	8-11
29.	9-6	11-15	30.	27-23	18-27 (23)
31.	2-7	27-32	32.	7-11	20-27 (24)
33.	11-18 (15)	16-19	34.	18-15	19-24
35.	6-2 (K)	24-28	36.	2-6	27-31 (K)
37.	15-18	32-27	38.	18-22	28-32 (K)
39.	6-9	12-16	40.	9-6	27-24
41.	6-9	24-28	42.	9-6	28-24
43.	6-9	24-28	44.	9-13	16-19
45.	22-17	19-23	46.	17-22	28-24
47.	13-9	23-27	48.	9-14	32-28
49.	14-9	24-19	50.	9-5	27-32 (K)
51.	5-9	32-27	52.	9-6	19-15
53.	6-9	27-23	54.	9-5	31-27
55.	5-9	15-11	56.	9-5	11-15
57.	5-9	15-10	58.	9-5	28-24
59.	22-17	24-19	60.	5-1	27-24
61.	1-5	23-18	62.	5-1	24-28
63.	1-5	18-14	64.	17-13	19-23
65.	5-1	23-18	66.	13-17	14-9
67.	17-13	9-5	68.	13-9	5-14 (9)
69.	1-5	28-24	70.	5-1	14-9
71.	1-5	18-14	72.	5-1	9-5
73.	1-6	10-1 (6)	0:1		

HG:E3Ph finally wins after having some trouble with taking advantage of its material superiority in the end-game.



(a) Comparison based on individual games.



(b) Comparison based on 10-game clashes.

Figure 6: Performance of the top-10 checkers heuristics in the tournament. The y-axis in each figure represents the fraction of points scored by each heuristic.