A genetic algorithm to improve an Othello program

Jean-Marc Alliot, Nicolas Durand

ENAC*, CENA**

Introduction

Let a computer program learn to play a game has been for long a subject of studies. It probably began in 1959 with A. Samuel's program [Sam59], and similar methods are still used on the most advanced computer programs, such as Deep Thought ; however, in chess, the different methods used are mainly linear regression on parameters of the evaluation function.

Othello was, from the start, a game where learning proved to be very useful. The BILL program used Bayesian learning [LM90] to improve parameters of the evaluation function. Similar methods are still applied for the best Othello programs (such as `LOGISTELLO`[Bur94]). However, these methods require large databases of games, and are very efficient only on Othello game playing, because the game always terminates in a fixed number of moves, with perfect end-games up to 15 moves. Such methods would be very difficult to use on chess playing algorithms.

Other techniques used include co-evolution [SG94], use of neural networks trained by GA to focus minimax-search [MM94], evolving strategies with GA [MM93]. However, [SG94] did not lead to World class Othello program, [MM94] was unable to improve search as soon as the evaluation function was good enough to correctly predict the move, and [MM93] was apparently not able to improve the classical (positional+mobility strategy).

In this article, we show how genetic algorithms can be used to evolve the parameters of the evaluation function of an Othello program. We must stress that our method can be used on any algorithm using an evaluation function, for any two players game. In the first part of the article, we explain the structure of the Othello program. In the second part, we explain the structure of our genetic algorithm, the operators (crossover, mutation) used, and our method to compute fitness. In the third part, results are presented and, in the last part, possible improvements and generalization of this method are discussed.

1 The Othello program

This work is based on an Othello program developed by the author a few years ago. This program was tested against other public domain programs and by a good French player (rated 10th in France). The program performed very correctly against these opponents.

The main advantage of this program is its very simple structure. The evaluation function has very few parameters. First, a static evaluation is obtained by using a static valuation of each part of the Othello board. For each disc on one square of the board, the value shown on table 1 is added to the current evaluation value if the disc belongs to the computer or subtracted if the disc is owned by its opponent.

Of course, this evaluation function has to be slightly enhanced. First, when one corner of the board is already occupied, the values of the three squares next to it are set to 0. On figure 1, the values of square G8, G7 and H7 are set to 0, instead of -150, -250, -150. Second, also when one corner is occupied, all squares on the edge which are connected to this corner and of the same color get a bonus (called thereafter the *bonus value*). On figure 1, three white discs get a bonus.

Then a second important factor has to be taken into account : the *liberty score*. The number of liberties of a disc is the number of empty squares close to the disc. The number of liberties of

* Ecole Nationale de l'Aviation Civile, 7 Avenue Edouard Belin, 31055 Toulouse CEDEX, France. e-mail : alliot@dgac.fr

** Centre d'Etudes de la Navigation Aérienne, e-mail : durand@cenatls.cena.dgac.fr

500	-150	30	10	10	30	-150	500
-150	-250	0	0	0	0	-250	-150
30	0	1	2	2	1	0	30
10	0	2	16	16	2	0	10
10	0	2	16	16	2	0	10
30	0	1	2	2	1	0	30
-150	-250	0	0	0	0	-250	-150
500	-150	30	10	10	30	-150	500

Table 1. Static square values

a player is the sum of all the liberties of all his discs. For example, on figure 1, black discs have 8 liberties. The *liberty score* is the difference between the liberties of the computer and the liberties of its opponent. This number is multiplied by a *penalty coefficient* and the result is subtracted to the evaluation function computed above.

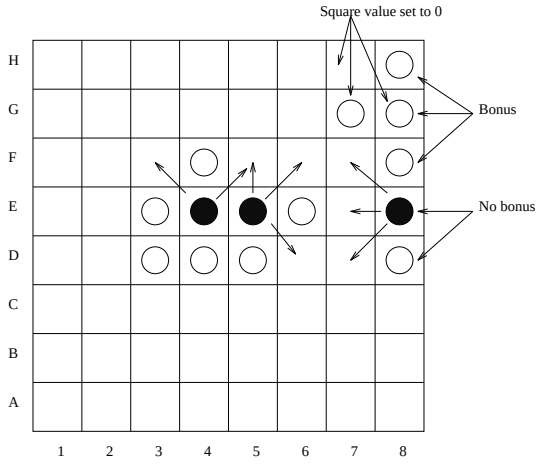

Fig. 1. Evaluation function principles

So the evaluation function has three terms : the static evaluation of the position, a bonus given for connected discs on the edge and a penalty value for the liberties of the discs. Given a board position, the function can be computed with only 12 parameters : 10 (given the symmetries) for the static evaluation of the board, one for the value of the bonus coefficient for each connected disc on the border, and one for the value of the penalty coefficient for each liberty of each disc. On the original program, the values of the 10 coefficients where [500, -150, -250, 30, 0, 1, 10, 0, 2, 16] (that can be easily deduced from table 1). The bonus value was set to 30 and the liberty penalty set to 8.

The rest of the program was a classical α-β iterative deepening algorithm, with an exhaustive search 11 to 14 plies before the end of the game.

As simple as it might look, the program was quite efficient, as the evaluation function was fast to compute and quite good. It was tested against different opponents, such as other public domain Othello programs, and human players, and always obtained good results. This is not very surprising, as almost all good Othello playing programs use such functions, starting with Rosenbloom's `IAGO` program [Ros82].

2 Optimizing the evaluation function parameters

Tuning the parameters of the evaluation function is a problem very difficult to solve. The values shown above are the results of a few interviews of good players which were also programmers, with no other justification. Using local climbing hill methods (modifying slightly one coefficient, while freezing the others) never gave conclusive results. The process was unstable : no stable maximum was ever reached, and coefficients were modified in one direction, then in the other. Classical optimization methods fail to solve this kind of problem.

Moreover, it is always difficult to evaluate if a program is performing well or badly. Even if the modified program wins 3 games in a row against the original one, this does not mean it is better. It could just mean it was lucky (of course, if it loses 3 in a row, it could also mean it was unlucky). To evaluate if a program is better or worst than another, a more complex methodology must be designed.

2.1 Principles

Classical Genetic Algorithms and Evolutionary Computation principles such as those described in the literature [Gol89, Mic92] are used; Figure 2 describe the main steps of GAs.

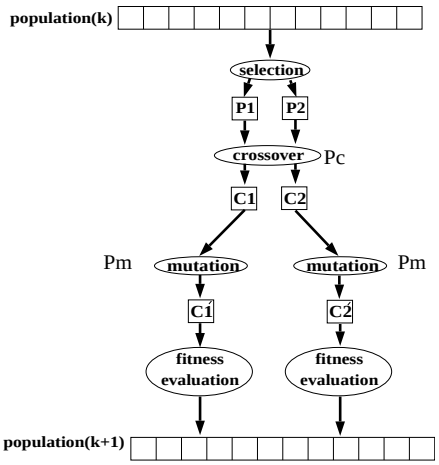

Fig. 2. GA principle

First a population of points in the state space is randomly generated. Then, the value of the function to optimize, called *fitness*, is computed for each population element. In a second step, elements are selected[3] according to their fitness. Afterward, classical operators of crossover and mutation are applied to diversify the population (they are applied with respective probabilities P_c and P_m). At this step a new population has been created and the process is iterated.

This GA has been improved by including different enhancements, described in different papers [DASF94, DAAS94]. We shortly present them again here (a full description is available in [All94]).

First a Simulated Annealing process is used after applying the operators [MG92]. For example, after applying the crossover operator, there are four individuals (two parents $P1$,$P2$ and two children $C1$,$C2$) with their respective fitness. Afterward, those four individuals compete in a tournament. The two winners are then inserted in the next generation. The selection process of the

[3] Selection aims at reproducing better individuals according to their fitness. We tried two kinds of selection process, Roulette Wheel Selection" and "Stochastic Remainder Without Replacement Selection", the last one always gave better results.

winners is the following: if $C1$ is better than $P1$ then $C1$ is selected. Else $C1$ will be selected according to a probability which decreases with the generation number. At the beginning of the simulation, $C1$ has a probability of 0.5 to be selected even if its fitness is worse than the fitness of $P1$ and this probability decreases to 0.01 at the end of the process. A description of this algorithm is given on figure 3.

Tournament selection brings some convergence theorems from the Simulated Annealing theory. On the other hand, as for Simulated Annealing, the (stochastic) convergence is ensured only when the fitness probability distribution law is stationary in each state point [AK89].

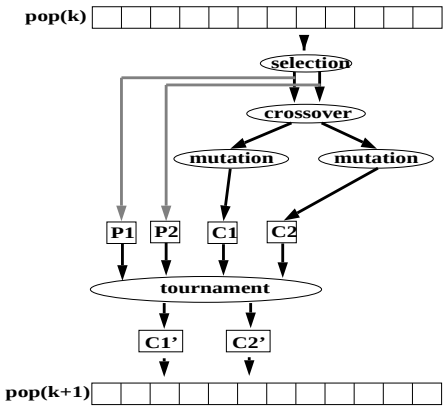

Fig. 3. GA and SA mixed up

Another enhancement was to use sharing as described in [YG]. This prevents the GA to fall too early in local minima. The method described in [YG] has the advantage of having a complexity in $n \log n$ instead of n^2 but is not perfect, as it does not always compute real clusters, but only approximates them. The algorithm has been modified in order to adapt dynamically the number of clusters during computation.

Last, parallelism is used to reduce time in computation. This method is mandatory on this problem as the time needed to compute one fitness is very long. The model of parallelism used is "island parallelism" : on a network of machines, each machine runs one genetic algorithm with its own population (one island). From time to time (every 10 generations for example), each island sends "missionaries" (some elements of its population, around 10%) to another island. Elements are picked at random in the origin population and replace elements taken also at random in the destination population (an elitist strategy is used to prevent destruction of best element, or best clusters element).

2.2 Chromosome structure

The chromosome is a very simple one. It contains 12 integers defining the evaluation function of the Othello program. It contains also the fitness computed at the previous generation, as it will be needed to compute the fitness at the next generation.

2.3 Computing fitness

To compute fitness, the idea is to let one population element play against a reference program (the original one) and use the result as fitness. However, this is not satisfactory. First of all, results are probabilistic. To have a good estimation, a large number of games would be required. This is not possible because of the time required. It takes 0.080s to play one game with a depth of 0 ply (only

using the evaluation function), 0.190s to play one game at depth 1, 0.807s at depth 2 and 3.56s at depth 3.

An other problem is determinism: a computer program playing against another computer program at the same depth will always play the same moves. It is useless to play many games with the same depth or the same starting position. These problems are classical when trying to evaluate two computer programs playing against each other, and some of the solutions presented here were already proposed in [LM90].

The following methodology is used; for each element of the population:

- 4 different starting positions are randomly generated. In each of these positions there are at least 4 discs (the 4 center discs) and at most 14 (there can be 10 more discs randomly placed in the inner 4×4 board).
- the new program then plays for each of these positions once with white and once with black. For each of this two games, the number $s = nw - nb$ is computed where nw stands for the number of white discs at the end of the game, and nb the number of black discs; then the difference $s1 - s2$ is computed. If the result is positive, we say that it is a victory for the new program, if the result is negative we say it is a loss, if the result is zero, it is a draw. For example, if the new program plays with white and wins $50 - 14$, then plays with black on the same position and loses $40 - 24$, we add $50 - 14 = 36$ to $24 - 40 = -16$, which gives 20, a victory for the new program.
- This operation is repeated with 3 different depths of evaluation, 0, 1 and 2.
- As 4 different positions are played at 3 different depths, the number of victories v and the number of draw d are computed. The number of discs of the program minus the number of disk of its opponent at the end of each game are also computed: we will call it c. Then we the current fitness f_c is :

$$f_c = v + d/2 + c/1000$$

- The real fitness is f computed with :

$$f = (f_c + f_p)/2$$

  where f_c is the current fitness and f_p is the fitness computed at the previous generation. This way, a program, if it is not modified, can accumulate results as time goes by. Of course, mutation and crossover must be correctly handled. This will be discussed in the next section.

This methodology was not chosen at random. In the first time, only one depth was used for searching. The new program was very efficient at this given depth, but was easily defeated at all other depths. With only one starting position (always the same) instead of five different randomly chosen at each generation, the program was very efficient for that given position, but played poorly with other starting positions. Last, having a fitness which uses past experience helped a lot to have stable results.

However, the results of the program were disappointing; it is easy to understand why: a program winning 12 games out of 12 has a better adaptation than a program winning 46 games out of 48. However, it is clear that, with "luck", it is "easier" to win 12/12 than 46/48. It is necessary to introduce also this factor when computing the fitness.

From a mathematical point of view, the probability of winning m games out of n, if the probability of winning one game is p, is given by the following formula:

$$P(p, m, n) = \binom{n}{m} p^m (1-p)^{n-m}$$

If n games are played, there are $n+1$ different possible results (only wins or losses are considered). They have all the same probability. So, we have:

$$\int_0^1 P(m, n, p)dp = 1/(n+1)$$

Let's consider the following function : $h(p,m,n) = (n+1)P(p,m,n)$. The fitness $f(m,n)$ of an element which has won m points over n games played will be given by the following implicit equation (we choose a 95% confidence value):

$$\int_{f(m,n)}^{1} h(p,m,n)\,dp = 0.95$$

This seriously modifies fitness values. For example $f(12,12) = 0.79$ while $f(23,24) = 0.83$, $f(44,48) = 0.82$ and $f(65,72) = 0.83$). All tables for n=12,24,36,...were pre-computed with Maple and included in the program.

2.4 The crossover operator

Different operators to cross programs were tried. A stochastic barycentric crossover was finally chosen.

When we cross 2 elements, a number α is picked randomly in the range $[-0.5, 1.5]$ and a number i in the range $[1, 11]$. Then the ith element of each of the two chromosomes is crossed. The new ith component of each chromosome is:

$$c'_{1i} = \alpha c_{1i} + (1-\alpha)c_{2i}$$

$$c'_{2i} = (1-\alpha)c_{1i} + \alpha c_{2i}$$

There is a 13th element in a chromosome, which is the fitness computed at the last generation. The fitness of the new elements is the mean of the fitness of the two parents. This is of course not perfect, but gives quite good results.

2.5 The mutation operator

A number i is picked in the range $[1, 11]$. Then a Gaussian noise centered on 0 is added, with a deviation of 50 if it is one of the 10 first components (static values of squares), and 20 if it is the 11th or the 12th.

The fitness of the new element is the same as the fitness of the parent.

2.6 Computing a distance for the sharing operator

To use sharing, distance between two elements has to be defined. A classical Euclidean distance could be used: $\sum_{i=1}^{12}(c_{1i}-c_{2i})^2$. But, here, it is not a very good idea. As the evaluation function is partly linear regarding its parameters, multiplying the tenth first parameters by 2 gives a program which will play exactly as the original one ; however, regarding Euclidean distance, they are very far from each other... To solve this problem, the value of the first element of the evaluation function is set to 500, and the GA is not allowed modify this value on any element of the population. This is not completely a neutral choice. By choosing this value we prevent the GA to evolve programs that might have a negative value for the first coefficient (all programs with a positive value for the first coefficient are equivalent). Half of the search space disappears. However, we strongly believe that corners must have a positive value...

2.7 Parameters settings for the GA

It is very long to run a genetic algorithm on such a problem. Evaluating one element of population takes more than 10s. Running a GA with 100 elements and 100 generations would require 30 hours. That's why a parallel GA was used. Each machine ran a GA with 30 elements of populations for 150 generations. Due to workload and speed differences between the 30 workstations (HP-720 and HP-730), the test ran for 12 hours instead of theoretically 10 hours.

We used a crossover probability of 60% and a mutation probability of 15%. Sharing, evolutive fitness, simulated annealing and an elitist strategy were also used.

3 Results

The evolution of fitness on the 8 best machines (those which gave the best results in the end) is presented on figure 4. Fitness evolves and can decrease, as it is not a static value but depends on

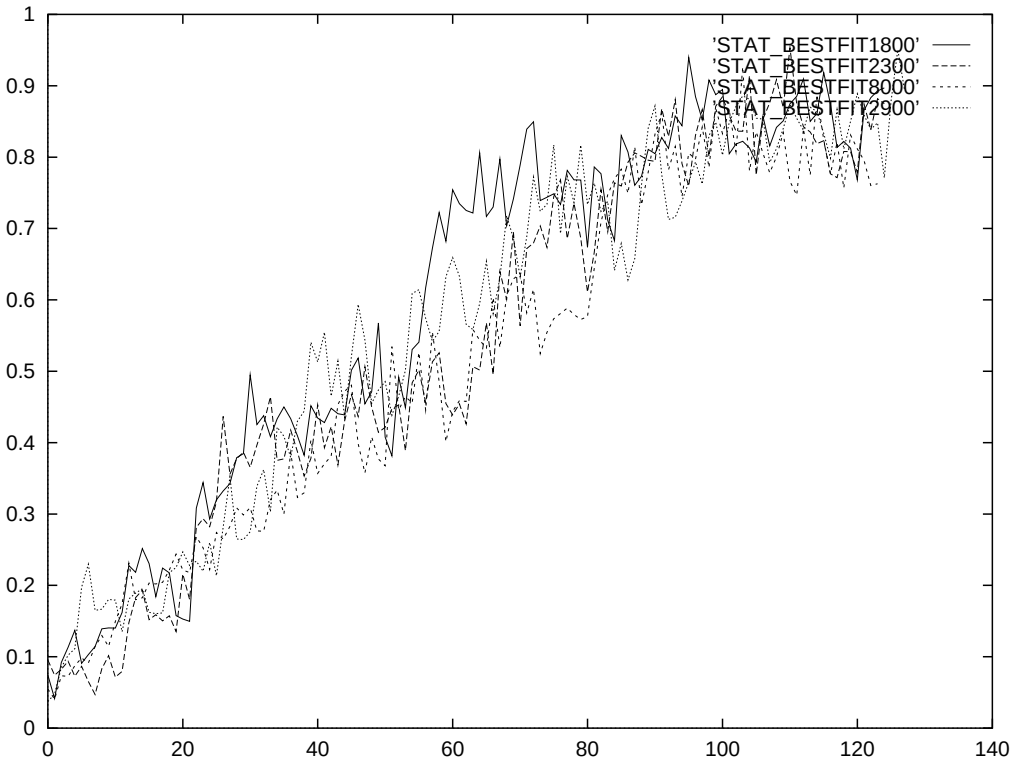

Fig. 4. Fitness evolution

the starting positions which are generated randomly at each turn. However, having a value which is always above 0.8 is good.

We chose the program which looked best. The static values are shown in table 2. The bonus value was set to 26 and the liberty penalty set to 104.

500	-86	96	26	26	96	-86	500
-86	-1219	-6	0	0	-6	-1219	-86
96	-6	52	15	15	52	-6	96
26	0	15	-17	-17	15	0	26
26	0	15	-17	-17	15	0	26
96	-6	52	15	15	52	-6	96
-86	-1219	-6	0	0	-6	-1219	-86
500	-86	96	26	26	96	-86	500

Table 2. Static square values

The program was intensively tested against the reference program. the new program was created by running some sequences of test, with 4 positions played and a depth going from 0 to 2. So it

was tested at depths ranging from 0 to 6, by playing 400 positions at each depth, except at depth 6 where we only played 100 positions. Results are given in table 3. The program wins 68% to 77% of points (1 point is given for a win one half point for a draw and zero for a loss), which is a definite improvement. At depths 3, 4, 5 and 6 which were not used to "train" the program, results are even better than at depth 0, 1 and 2.

Depth	Wins	Losses	Draw	% points	Disc difference
0	282	98	20	73	+28
1	252	101	47	68	+24
2	278	78	44	75	+20
3	281	75	44	75	+22
4	280	75	45	75	+22
5	286	68	46	77	+25
6	72	20	8	76	+23

Table 3. Results of the stage-1 program against the original

We might fear that the new program is very efficient against our test program, but will be easily defeated against other programs or human players. The new program (called the stage-1 program) was tested against human players, but it was very inconclusive, as both program always won against all the players we could find. So, the stage-1 program was used as the reference program to evolve a stage-2 program. Again, the GA was able to improve the program; The static values are shown in table 4. The bonus value was set to 18 and the liberty penalty set to 312.

500	-240	85	69	69	85	-240	500
-240	-130	49	23	23	49	-130	-240
85	49	1	9	9	1	49	85
69	23	9	32	32	9	23	69
69	23	9	32	32	9	23	69
85	49	1	9	9	1	49	85
-240	-130	49	23	23	49	-130	-240
500	-240	85	69	69	85	-240	500

Table 4. Static square values

Then, similar tests were ran again, having the stage-2 program playing against the stage-1 program. Only tests for depths 0,1,2,3 and 4 were ran, due to lack of time. Results are very good (see table 5), except at depth 0. We can only guess that the stage-1 program is already excellent at that depth and very difficult to improve. At depth 1 and 2, which were also used to train the program, results are above 75%, and are still excellent at depth 3 and 4.

Is the stage-2 program still better that the original one? The same tests (see table 6) show that results are excellent at every depth, as good as the results of the stage-1 program.

4 Remaining problems and improvements

It would of course be interesting to build a stage-3, a stage-4 program, etc, and see how long the program can be improved, or if, in the long time, the resulting program becomes less efficient against the original program, etc. This will be done in the near future.

Depth	Wins	Losses	Draw	% points	Disc difference
0	188	194	18	49	-3
1	285	82	33	75	+27
2	303	55	42	81	+28
3	298	50	52	78	+37
4	320	46	34	85	+36
5	329	40	31	86	+38

Table 5. Results of the stage-2 program against the stage-1 program

Depth	Wins	Losses	Draw	% points	Disc difference
0	274	107	19	71	+26
1	266	93	41	72	+21
2	243	120	37	65	+20
3	292	61	47	79	+27
4	277	82	41	74	+22

Table 6. Results of the stage-2 program against the original program

We have stressed the fact that our methodology could be extended to almost any two-players game which rely on an evaluation function. The only problem would be the time required to play the game. It would be very interesting to try to evolve parameters of a chess program with our algorithm. However, this would require a tremendous power in calculation: it is much longer to evaluate a chess position and the branching factor in chess is higher than in Othello. Moreover, an evaluation function in chess needs much more parameters. It would probably require weeks to get results, even with many workstations working together.

Building the evaluation function by evolving the structure of the evaluation function itself with Genetic Programming, instead of evolving only the parameters was inconclusive. The problem was to find the right terminal functions. Using functions too basic gave no result at all, and using too macroscopic functions (already including most of the structure) was not very different from evolving only the parameters of the evaluation function.

We had previously worked on evolving neural nets with Genetic Algorithms. This gives very good results on some test problems such as car parking [SR92], or aircraft conflict resolution. These examples are quite simple as there are few inputs (6 for the second example) to the network, and the GA has only a hundred parameters to evolve. We are currently working on the following program: the base idea is to have an evaluation function computed by a neural network, and to build this neural network with a GA. However, a direct approach with random initialization of all weights in the network gave no result. The GA was never able to build a network that could even defeat the reference program once in 20 games. So, we now train a very simple neural network (64 inputs, 8 neuronal units in the hidden layer, and one output) with our reference program. We then have a program playing Othello with a neural network instead of a classical evaluation function. This program plays quite good Othello, and sometimes defeat the reference program. Now, we are going to evolve the neural net with a GA, starting with this NN as the base for all elements. This may give interesting results.

It must be noted that this approach is very slow, as the neural networks we used are based on classical floating-point functions. Results might be obtained by using ALN networks [ALLR91] instead of classical feed forward networks : in Othello, inputs of the network are almost binaries, and ALN are much faster and easy to evolve.

References

[AK89] Emile Aarts and Jan Korst. *Simulated annealing and Boltzmann machines.* Wiley and sons, 1989. ISBN: 0-471-92146-7.

[All94] Jean-Marc Alliot. A parametrable parallel genetic algorithm with simulated annealing and adaptive sharing. Technical report, Ecole Nationale de l'Aviation Civile, 1994.

[ALLR91] William Armstrong, Jiandong Liang, Dekang Lin, and Scott Reynolds. Experience using adaptative logic networks. In *Proceedings of the IASTED International Symposium on Computers, electronics, communication and control*, 1991.

[Bur94] Michael Buro. *Techniken fur die bewertung von Speilsituationen anhand von beispielen.* PhD thesis, Universitat GH Paderborn, 1994.

[DAAS94] Nicolas Durand, Nicolas Alech, Jean-Marc Alliot, and Marc Schoenauer. Genetic algorithms for conflict resolution in air traffic. In *Proceedings of the Second Singapore Conference on Intelligent Systems.* SPICIS, 1994.

[DASF94] Daniel Delahaye, Jean-Marc Alliot, Marc Schoenauer, and Jean-Loup Farges. Genetic algorithms for air traffic assignment. In *Proceedings of the EuropeanConference on Artificial Intelligence.* ECAI, 1994.

[Gol89] David Goldberg. *Genetic Algorithms.* Addison Wesley, 1989. ISBN: 0-201-15767-5.

[LM90] Kai-Fu Lee and Sanjoy Mahajan. The development of a world class othello program. *Artificial Intelligence*, 43, 1990.

[MG92] Samir W. Mahfoud and David E. Goldberg. Parallel recombinative simulated annealing: a genetic algorithm. IlliGAL Report 92002, University of Illinois at Urbana-Champaign, 104 South Mathews Avenue Urbana IL 61801, April 1992.

[Mic92] Zbigniew Michalewicz. *Genetic algorithms+data structures=evolution programs.* Springer-Verlag, 1992. ISBN: 0-387-55387-.

[MM93] David Moriarty and Risto Miikulainen. Evolving complex othello startegies using marker-based genetic encoding of neural networks. Technical Report AI93-206, University of Texas, Austin, TX 78712-1188, September 1993.

[MM94] David Moriarty and Risto Miikulainen. Evolving neural networks to focus minimax search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence AAAI-94*, Seattle, WA, 1994.

[Ros82] P. Rosenbloom. A world championship-level othello program. *Artificial Intelligence*, 19:279–320, 1982.

[Sam59] Arthur Samuel. Some studies in machine learning using the game of checkers. *IBM journal of research development*, 3(3):210–229, 1959.

[SG94] Robert Smith and Brian Gray. Co-adaptive genetic algorithms: An example in othello strategy. In *Proceedings of the Florida Artificial Intelligence Symposium*, 1994.

[SR92] Marc Schoenauer and Edmund Ronald. Evolving neural nets for control. Technical report, Centre de mathématiques appliquées de l'Ecole Polytechnique, 1992.

[YG] X. Yin and N. Germay. A fast genetic algorithm with sharing scheme using cluster analysis methods in multimodal function optimization. Technical report, Laboratoire d'Electronique et d'Instrumentation, Catholic University of Louvain.

This article was processed using the LaTeX macro package with LLNCS style