Solving *MasterMind* using GAs and simulated annealing: a case of dynamic constraint optimization

J. L. Bernier, C. Ilia Herráiz, J. J. Merelo, S. Olmeda, A. Prieto

Department of Electronics and Computer Technology; University of Granada (Spain)
e-mail: jmerelo@kal-el.ugr.es, http://kal-el.ugr.es/jj/jj.html

Abstract. MasterMind is a game in which the player must find out, by making guesses, a hidden combination of colors set by the opponent. The player lays a combination of colors, and the opponent points out the number of positions the player has found out (black tokens) and the number of colors that are in a different place from the hidden combination (white tokens). This problem can be formulated in the following way: the target of the game is to find a string composed of l symbols, drawn from an alphabet of cardinality c, using as constraints hints that restrict the search space. The partial objective of the search is to find a string that meets all the constraints made so far the final objective being to find the hidden string.This problem can also be located within the class of *constrained optimization*, although in this case not all the constraints are known in advance; hence its dynamic nature.Three algorithms playing MasterMind have been evaluated with respect to the number of guesses made by each one and the number of combinations examined before finding the solution: a random-search-with-constraints algorithm, simulated annealing, and a genetic algorithm. The random search and genetic algorithm at each step plays the optimal solution i.e., the one that is consistent with the constraints made so far, while simulated annealing plays the best found within certain time constraints. This paper proves that the algorithms that follow the optimal strategy behave similarly, getting the correct combination in more or less the same number of guesses; between them, GA is better with respect to the number of combinations examined, and this difference increases with the size of the search space, while SA is much faster (around 2 orders of magnitude) and gives a good enough answer.

1 Introduction

MasterMind is a game in which the player must find out a hidden combination of colors set by the opponent. The player lays a combination of color pegs, and the opponent points out the number of pegs which appear in the same position in the guess and the hidden combination, (black tokens, or *bulls*) and the number of color pegs that appear in the hidden combination, but in a different place (white tokens, or *cows*). No hint is given about which of the pegs laid by the player are the correct ones. These hints placed by the opponent will be called *constraints*

or *rules*, since they restrict the search space. The problem of finding the secret combination can be formulated in the following way: the target is to find a string composed of l symbols, each one drawn from an alphabet of cardinality c, using as constraints hints that restrict the search space. During each step, the partial objective of the search is to find a string, which will not necessarily be the correct one, that meets all the constraints made so far, the final objective being to find the hidden string. The best player is the one that finds a solution in a minimal number of guesses, on average.

The optimal strategy to play *MasterMind* is then to play at each step a combination that is consistent with the constraints made so far; i. e., such that, if it was the secret combination, the *bulls* and *cows* (black and white tokens) set so far would be the same. An algorithm following this optimal strategy will try to find such a combination and play it at each step; suboptimal algorithms will try to find one that meets as many constraints as possible. Not all optimal strategies will behave in the same way, because at each step (except the last one) there are many combinations in the subset that meet all constraints, and some of them will restrict the remaining space more than others. For instance, it has been suggested that the best ones try to diversify as much as possible the number of colors [1]. However, if no attempt is made to further select the played combination, all optimal strategies, on average, should behave in the same way.

There are many possible ways of following this optimal strategy. They are all search algorithms that seek one of the optimal combinations at each step. The first obvious approach is *exhaustive search*. The space of all possible combinations is generated; each combination is examined in turn and played or ruled out. This approach is possible when the search space is small, compared with the typical computer memory available, but impossible when the search space is too big for the memory space in the computer. A *random search algorithm* can also be used; new combinations are generated randomly and played if they meet all current constraints; a variant of this algorithm would cache the number of constraints met by already examined combinations, so that the evaluation would be faster. Finally, a *heuristic approach* would try to play in the same way a human would do. A common strategy is to play at the beginning combinations with only one color, "to get the colors right" [2], until all the colors in the combination are found out; then different combinations with the colors already known are tried. This approach, while usually good for the standard *MasterMind* (4 pegs, 6 colors) fails in the generalized case, for instance, if there are many colors; there would be as many initial guesses as colors.

Different algorithms will consist, then, in different ways of searching the combination space for both the partial and the final solution. Solutions will be better if they are found in a minimal amount of guesses, and, if possible, examining a minimal subset of the search space, but for the generalized game, this might take too much time, and a suboptimal solution will often be good enough, if the time it takes is orders of magnitude smaller.

This problem has been approached from the AI, combinatorics and game theory fields.Several algorithms have been described in the literature that try

to follow an optimal strategy. Two of them have been published in the Journal of Recreational Mathematics [3, 4]. The first one manages to solve the standard game in, at most, 5 moves; the latter, using exhaustive search, proves that the optimal average number of moves is 4.340, if a maximum of 6 guesses are allowed, or 4.341, if only 5 guesses are allowed. This result has been slightly improved by Neuwirth [5], using several strategies taken from game theory. Viaud [6, 7] also presents a general framework to play *MasterMind*. A *Prolog* language program is also described in [8], which takes "an average of 4 to 6 guesses, with a maximum of 8"; other programs that play *MasterMind* using Prolog can be found in AI software repositories. V. Chvatal [9] (mentioned in [10]), proved that the strategy of selecting vectors at random is close to optimal in the general case.

A contest, with 5 positions and 7 numbers, is played in the Web site *Web-Meninges*. The best player so far is called `pc_test1`, and achieves an average of 5.08 moves. The algorithm is not known, although an educated guess would be that it uses exhaustive search (in a space of $7^5 = 16807$ combinations).

The paper is organized as follows: section 2 describes the genetic algorithm used for playing *MasterMind*, pointing out the problems it faces and the solutions proposed; section 3 presents the simulated annealing algorithm proposed to solve the same problem suboptimally. Section 4 is devoted to a comparison the previous two algorithms and a random search algorithm, and finally, these results are discussed in section 5, along with future lines of research.

2 A Genetic Algorithm playing Master Mind

As has been said, playing *MasterMind* is a restricted search problem, but it can also be approached as a constrained optimization problem; i. e., looking for the best solution within some constraints; in this case, the *fitness* of the solution is computed directly from the number of constraints it meets. Several genetic algorithms have been proposed to solve such problems. A popular approach involves a *penalty function*, substracted from the fitness, whose value increases with the badness of the constraint break [11]. Another solution proposed by Schoenauer et al. [12] is to change the fitness function each time a constraint is met, and then restart the population with a new fitness function to optimize another constraint.

On the other hand, a new constraint is added every time a combination is played, so that the genetic search itself develops in a changing environment. The main problem in this kind of environment, as pointed out by Cobb et al. [13], is to maintain diversity so that the changing fitness landscape can be explored efficiently. Grefenstette et al. [14] have proposed several strategies, mainly based on a randomization of the population: an operator called *hypermutation*, which mutates a part of the population with a high rate , *random replacement*, and simply a high mutation rate; a similar "restart" operator is presented by Maresky et al. in [15]. This sort of "frontal assault" approach to GAs is explicitly discarded by Goldberg in [16], but more natural approaches like de Jong's *crowding factor* [11] or Ryan's *pygmy algorithm* [17] keep diversity in a more natural way, while

yielding more or less the same results. In any case, a mechanism to maintain diversity must be used to properly solve *MasterMind*. Schoenauer et al.'s [12] approach skillfully reduces the problem mentioned in the first paragraph to this one; the solution proposed includes a *hypermutation* phase, each time a constraint is met; it also emphasizes the need to maintain diversity.

The two previous problems (which could be reduced to one, following Schoenauer et al. [12]), along with the fact that only one combination in the whole search space is correct, and the need to restrict the search space at each game step, call for an adaptation of one of the usual GAs, instead of a standard approach. Thus, a GA will be presented that tries to fiend a hidden combination following an optimal strategy; i.e., it plays a combination as soon as one suitable is found.

Nevertheless, the standard binary **representation** is used. The genetic algorithm acts on a population of binary strings representing combinations. The maximum number of colors allowed is 8, so that each color is represented in the chromosome string by 3 bits. Thus, the chromosome is 3*(length of the combination) long. If the number of colors is less than 8, the color of the combination played (phenotype) will be the remainder of dividing the gene value by the number of colors; this means it is a "redundant" coding, since two different bit strings can result in the same combination. Although this reprsetentation introduces a bias for a number of colors not a power of two, it has been chosen because it is more compact than others (integer or floating point), while being able to support the same kind of operations.

The **fitness function** must be obviously related to the optimal strategy mentioned in the introduction; i.e., to the number of rules a combination meets. However, instead of simply using that amount, as in previous approaches [18], more weight is given to black pegs (10) than to white pegs (1); thus, each time a rule is met, the fitness increases by 10*(number of black pegs) + (number of white pegs). This means that a rule with 0 blacks/0 whites adds 0 to fitness, but this is not usually a problem for the range of colors and length of combinations used (it could be, however, for longer combinations or more colors). This approach has the advantage over the previous one [18] of making some combinations better than others, if they meet "better" rules, even if the number of rules met is the same. Thus, solutions are built from the "best" rules, those with the most black pegs.

This election of fitness function increases the efficiency of the selection procedures over previous choices [18], but, nevertheless, it is not very efficient, since fitness increases by leaps, not gradually. This once again calls for an increase of "exploration", or diversity-creation, operators, instead of "exploitation" operators, since groups in the population will have exactly the same fitness.

The importance of maintaining a high diversity becomes apparent in the operator and parameter choices related to population. The theory with respect to **population size** usually indicates that it must grow with the chromosome size. In this case, the population size P is quite high compared with usual approaches: for a chromosome size of 18 bits (6 colors, 6 pegs), for instance, 500-chromosome

populations are used; and, besides, experiments show that P must grow faster than lineally to accomodate variations in the search space size, to the point that a 10000-chromosome population is used for length 10 combinations.

The **selection procedure** used is *steady state*; a proportion of the population is kept in each generation. New individuals are generated in 3 different ways:

- From two parents, using 2-point crossover, which does not consider gene (color) boundaries. This operator allows the mixture of combinations that meet two different rules to give a new one that meets both. Just as gene breaking has the effect of random mutation, this factor once again increases diversity. A low proportion X of the population P is created using this operator, $X = 0.15..0.30$; one of the parents is chosen from the $X \cdot P$ ranked best among the population.
- From only one parent, using mutation. This procedure is called *cloning with mutation*. The mutation operator performs bitflipping, and the rate is adjusted in such a way that at least a bit will be changed in each chromosome that undergoes cloning with mutation. Such a proportion C changes each time a new combination is played, and is proportional to the length of the combination less the number of black and white pegs for the *last* combination played. In this way, it goes down to 0 when all the colors in the combination are known, and only the correct position must be found out. An adaptive operator rate has already been used by Julstrom [19] to improve performance; however, in this case, it has been used to change the space that is being searched and the rate at which the search space is explored.

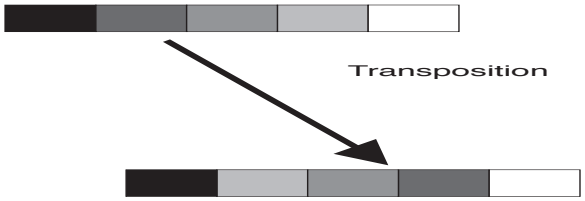

Fig. 1. Transposition operator used in this paper, in this case acting on 5-gene chromosomes. Genes 2 and 4 are interchanged.

- From only one parent, using *inversion* or *transposition* (*cloning with transposition*). This operator, which interchanges the values of whole genes in a chromosome (see Figure 1) is rather unusual in GAs, although mentioned by Holland [20] and Fogel [21]; however, it is specially suitable for this problem, since it can turn a combination with white pegs into another with color pegs in the correct position. That is the reason why the proportion of chromosomes cloned with inversion is proportional to the number of white pegs in the last combination played.

These last two operators are only applied to chromosomes whose fitness is not among the X best, ie, those ranked in the $P - 2X$ central positions. Among

these, $0.4(l - n_b - n_w)$ undergo cloning with mutation, and $0.4 \cdot n_w$ undergo cloning with transposition. These parameters have been found by tuning. The total population P and the percentage of population undergoing crossover X vary with the length of the combination; the other parameters stay the same through all tests made.

3 Simulated annealing

Simulated annealing (SA) has also been used for solving combinatorial problems [22, 23]. In a classical SA problem, a cost function to be minimized is defined and then, from an initial solution, different solutions are derived and compared with the retained solution; the solution with the least cost is kept, but retaining a solution with a greater cost is allowed with a certain probability. The probability of keeping a worse solution decreases with a parameter called temperature.

In the case of *MasterMind*, the objective is to find a combination that meets all the rules to make a new guess. Thus, SA is applied to discover a combination that meets the maximum number of rules within a limited time; if the stopping condition is to find a combination that meets all rules the procedure will become very similar to a random algorithm.

Each combination is compared with the previously played guesses, the number of different white and black pegs are computed and added to the cost, that is defined as $C = \sum_{i=1}^{n} abs(\Delta n_w^i) + abs(\Delta n_w^i + \Delta n_b^i)$ where n is the number of guesses played and Δn_w^i and Δn_b^i are the white and black peg differences between the target combination and the guess i. In this way, the cost increases with the number of unsatisfied rules, exactly the opposite of the fitness defined in section 2.

The SA algorithm is the usual, except that two operations are used to generate a combination from the previous one: permutation and mutation. While a permutation changes the order of two pegs, the mutation changes the color of a peg; in both cases the pegs are randomly chosen. The values of the parameters were fixed to the following ones: $T_f = 1e-30$, $\alpha = 0.98$. In this way, the combination payed meets a suboptimal number of rules in the worst case, but the time to find it is short. The number of guesses needed to obtain the final solution is usually acceptable. This algorithm finds the solution very quickly as the number of combinations to check is small.

4 Results

These two algorithms, along with a random search algorithm that plays the optimal strategy, have been tested in games with 6 colors, and 4 to 10 pegs. Using different randomly generated combinations to be guessed, 20 games were played for each color/peg combination for random and GA, and 100 for SA. Different parameter values were tested for the GA; results are presented for the best combination of parameters. The number of guesses made and the number of combinations evaluated were recorded.

A program using the previous algorithm [18] is also running on our Web server, using 6 colors and 6 pegs, and has so far obtained an average of 5.62 for 693 games played (GA) and 5.98 for 232 games played (SA). The current version has been evaluated with the *WebMeninges* setup, scoring an average of 5.45±0.67 guesses, and evaluating an average of 7755 ± 4286 combinations, placing it among the best 30. The other results are represented in figure 2, with averages and standard deviation. Times have not been measured accurately,

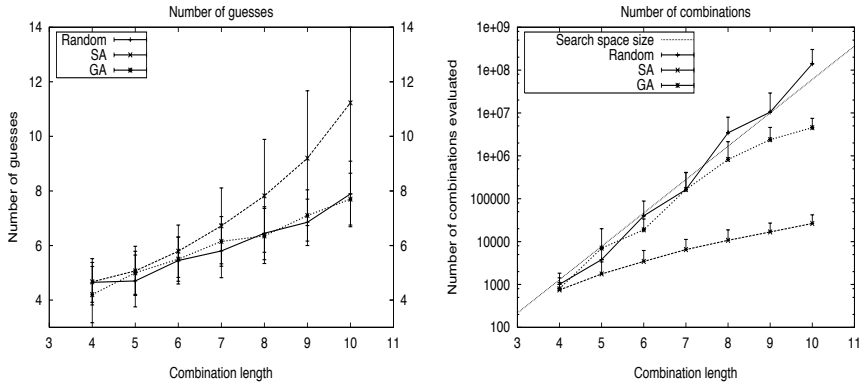

Fig. 2. Graph comparing the average number of guesses made and number of combinations evaluated, with standard deviation, by the three algorithms.

except for SA, only order of magnitude measures: SA takes from tens of milliseconds to 10 seconds, GA from seconds to hours, and the random algorithm from milliseconds to tens of hours (in fact, not all measures could be taken for length = 10). The algorithms following optimal strategy, GA and random, beat the SA in number of guesses, and are thus better players, but SA is much faster than either of them: at worst, only 25 seconds elapse before reaching a solution. It can be clearly seen that the GA and random algorithm are almost equal in average number of guesses made, and SA slightly worse, with the difference increasing with the combination length. This is due to the fact that SA does not follow the optimal strategy. However, the number of combinations examined by SA is orders of magnitude smaller than any of the other algorithms; the number of combinations evaluated by the random algorithm increases exponentially, almost at the same rate as the size of the search space, while it grows slower than exponentially for SA and GA. For the longest combination tested, l= 10, the number of combinations evaluated is two orders of magnitude bigger for SA than for GA, and 1 order of magnitude bigger for random than for GA, which explains the time taken by each one of them. In any case, the GA does not need to search the whole space to find the solution; for l = 10, only 4% of the search space is examined on average. Two other things must be taken into account: first, the number of actual combinations examined by the GA is, on average, 2/3 of the one shown, since the chromosome-combination mapping is not one

to one (see section 2. Relatively better results would be obtained for a representation in which genotype and phenotype were the same. Second, the worst result in combinations would be the number of guesses (the search space size) times 0.5, since, in any algorithm, the combinations already examined are not retained. This would be reduced to (the search space size) times 0.5 in the case of exhaustive search, but this possibility has not been examined.

How then, would a GA-endowed automated player fare against a SA-engine and a random challenger? To find this out, it must first be pointed out that it is not the same when a player always gets the hidden combination in 5 guesses, as when part of the time it is achieved in 4, and the rest in 6; while having similar averages, the latter would beat the former if it manages to score 4 once more than 6. Thus, the number of guesses each algorithm makes have been compared by plotting the proportion of times one algorithm would beat the other, and the number of draws. As can be seen in figure 3, the GA would beat SA more times than the other way round, even for small-length combinations; but for greater length combinations, these results improve to the point where it is almost impossible for the SA to win even one match. To enphasize the extent

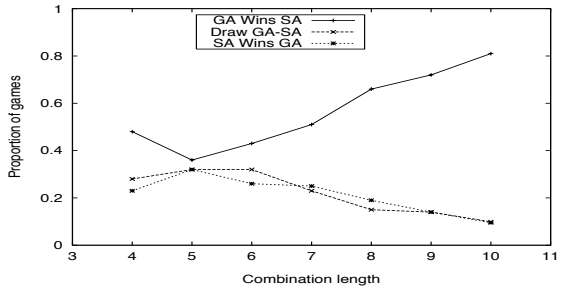

Fig. 3. The proportion of times each of GA and SA would beat the other, and the number of draws, is plotted for different combination length.

to which a GA is better than a random search algorithm, while averaging more or less the same number of guesses, figure 4 presents a *time-to-solution* graph for the extreme case: length 9. This graph represents the number of combinations evaluated in the x axis, and the accumulated proportion of runs that reach the solution in that number of guesses. On this extreme case, the GA always find the solutions in less than 10% of the time the random search algorithm needs.

5 Discussion

Examining the behavior of three algorithms on a combinatorial search problem, playing the game of Mastermind, we have found out that GA is a robust search algorithm, performing well through a wide range of problem sizes, and it is efficient, since it finds the solution with enough speed, while examining a small portion of the search space. Simulated annealing is very fast, but cannot play

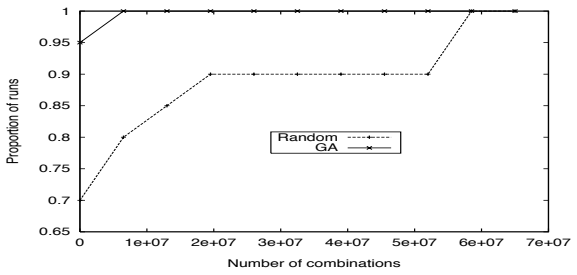

Fig. 4. Graph comparing the number of combinations a GA and a random search algorithm need to reach a solution for $l = 9$. The proportion of runs (over 20) is plotted on the y axis, and the number of combinations needed on the x axis.

optimally since it would reduce itself to a random search algorith; and the random search algorithm performs as expected, but is the slowest; it is good enough, nevertheless, for small search spaces. SA has the advantage that its optimality can be fixed in advance.

Several improvements can be made to the basic GA and SA algorithms to improve results. One of the lines we intend to pursue is hybridization of SA and GA, in the style suggested by Mahfoud et al. in [24], to improve SA performance and to make it possible to follow the optimal strategy. In any case, the current SA setup will be tested following the optimal strategy. The GA can also be improved. One interesting test would be to use integer representation, instead of binary; if crossover is used in this representation, genes will not be broken, and it will not have the effect of mutation. Other types of crossover, like 1-point crossover or uniform crossover, can also be tested. Besides, the adaptivity of operator rates can also be extended to crossover, so that the percentage of the population generated by crossover also changes through time. These modifications attempt to decrease the amount of invalid combinations generated. With respect to diversity, some other strategies can also be tested, like, for instance, the *pygmy* algorithm [17], or niching strategies like the one proposed originally by de Jong.

This procedure can also be used in other restricted combinatorial search problems. It could probably be used too in other games, like "Battleships" or "Scrabble"; in these cases, the search space is much bigger, but the techniques described in this paper could be adapted.

Acknowledgements I would like to acknowledge the help of Marc Schoenauer.

References

1. Melle Koning. Mastermind. newsgroup `comp.ai.games`, archived at `http://www.krl.caltech.edu/~brown/alife/news/ai-games-html/0370.html`.
2. Tim Cooper. Mastermind. newsgroup `comp.ai.games`, archived at `http://www.krl.caltech.edu/~brown/alife/news/ai-games-html/0313.html`.
3. Donald E. Knuth. The computer as Master Mind. *J. Recreational Mathematics*, (9):1–6, 1976-77.

4. Kenji Koyama; T. W. Lai. An optimal Mastermind strategy. *J. Recreational Mathematics*, 1994.
5. E. Neuwirth. Some strategies for mastermind. *Zeitschrift fur Operations Research. Serie B*, 26(8):B257–B278, 1982.
6. D. Viaud. Une strategie generale pour jouer au Mastermind. *RAIRO-Recherche Operationelle*, 21(1):87–100, 1987.
7. D. Viaud. Une formalisation du jeu de Mastermind. *RAIRO-Recherche Operationelle*, 13(3):307–321, 1979.
8. Leon Sterling; Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT press, 1986.
9. V. Chvatal. Mastermind. *Combinatorica*, 3(3-4):325–329, 1983.
10. Risto Widenius. Mastermind. newsgroup `comp.ai.games`, archived at `http://www.krl.caltech.edu/~brown/alife/news/ai-games-html/1311.html`.
11. Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution programs*. Springer-Verlag, 2nd edition edition, 1994.
12. Marc Schoenauer; Spyros Xanthakis. Constrained GA optimization. In Forrest [25], pages 573–580.
13. John J. Grefenstette. Genetic algorithms for changing environments. In R. Maenner; B. Manderick, editor, *Parallel Problem Solving from Nature, 2*, pages 137–144. Elsevier Science Publishers B. V., 1992.
14. Helen G. Cobb; John J. Grefenstette. Genetic algorithms for tracking changing environments. In Forrest [25], pages 523–530.
15. A. Barak J. Maresky; Y. Davidor; D. Gitler; G. Aharoni. Selective destructive re-start. In Eshelman [26], pages 144–150.
16. David E. Goldberg. Zen and the art of genetic algorithms. In J. David Schaffer, editor, *Procs. of the 3rd Int. Conf. on Genetic Algorithms, ICGA89*, pages 80–85. Morgan Kauffmann, 1989.
17. Conor Ryan. *Advances in Genetic Programming*, chapter Pygmies and Civil Servants. MIT press, 1992.
18. J. J. Merelo. Genetic Mastermind, a case of dynamic constraint optimization. GeNeura Technical Report G-96-1, Universidad de Granada, 1996.
19. Bryant A. Julstrom. What have you done for me lately? Adapting operator probabilities in a steady-state Genetic Algorithm. In Eshelman [26], pages 81–87.
20. John J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.
21. D. B. Fogel. *Evolutionary Computation: Towards a New Philosophy of Machine Intelligence*. IEEE press, 1995.
22. J. Aarts, E.; Korst. *Simulated Annealing and Boltzmann Machines*. John Wiley & Sons, 1989.
23. C. H. Papadimitriou; K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, New York,, 1982.
24. S. W. Mahfoud; D. E. Goldberg. Parallel recombinative simulated annealing: A genetic algorithm. Technical report, IlliGAL Report no. 92002, Dept. Of General Engineering, UIUC., 1992.
25. Stephanie Forrest, editor. *Proceedings of the 5th International Conference on Genetic Algorithms*. Morgan Kaufmann, 1993.
26. Larry J. Eshelman, editor. *Proceedings of the 6th International Conference on Genetic Algorithms*. Morgan Kaufmann, 1995.

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