Assessing efficiency of different evolutionary strategies playing MasterMind

Juan J. Merelo, Antonio M. Mora, Thomas P. Runarsson and Carlos Cotta

***Abstract*— A MasterMind player must find out a secret combination (set by another player) by playing others of the same kind and using the hints obtained as a response (which reveal how close the played combination is to the secret one) to produce new combinations. Despite having been researched for a number of years, there are still many open issues: finding a strategy to select the next combination to play that is able to consistently obtain good results at any problem size, and also doing so in as little time as possible. In this paper we cast the solution of MasterMind as a constrained optimization problem, introducing a new fitness function for evolutionary algorithms that takes that fact into account, and compare it to other approaches (exhaustive/heuristic and evolutionary), finding that it is able to obtain consistently good solutions, and in as little as 30% less time than previous evolutionary algorithms.**

I. Introduction

In its current version, MasterMind [1] is a board game that was designed by the telecommunications expert Mordecai Merowitz [2] and sold to the company Invicta Plastics, who renamed it to its current name to commercialize it in the early 70s, when it enjoyed a wide popularity. However, this current game is a version of a traditional puzzle called *bulls and cows* [3], and apparently AB in Asia [4], that dates back to the Middle Ages. In any case, MasterMind can be considered a puzzle (rather than a game) in which two persons, the *codemaker* and *codebreaker* try to outsmart each other in the following way:

- The codemaker sets a length ℓ combination of κ symbols. In the classical version, $\ell = 4$ and $\kappa = 6$ (with the space of secret combinations having, then, 1296 combinations), and color pegs are used as symbols over a board with rows of $\ell = 4$ holes; however, in this paper we will use uppercase letters starting with A instead of colours. Versions with different values of ℓ and κ receive different names: $\ell = 5$ and $\kappa = 8$ is called Logik [5], Challenge or Super-MasterMind.
- The codebreaker then tries to guess this secret code by producing a combination with the same length and using the same set of symbols as the secret code.
- The codemaker gives a response consisting on the number of symbols guessed in the right position (usually represented as black pegs) and the number of symbols in an incorrect position (usually represented as white pegs).
- The codebreaker then, using that information as a hint, produces a new combination.
- This process is repeated until the secret code is found.

For instance, a game could go like this: The codemaker sets the secret code *ABBC*, the codebreaker plays *AABB*, getting 2 black pegs (for *A-B-*) and a white peg (for - - - *B*). And the rest of the game proceeds as shown in Table I, needing a total of five combinations to find the correct answer (although, in this case, it could have needed many more).

Combination	*Response*
AABB	2 black, 1 white
ABFE	2 black
ABBD	3 black
BBBE	2 black
ABBC	4 black

TABLE I

PROGRESS IN A MASTERMIND GAME THAT TRIES TO GUESS THE SECRET COMBINATION *ABBC*. 2ND AND 4TH COMBINATIONS ARE NOT *consistent* WITH THE FIRST ONE, NOT COINCIDING IN TWO POSITIONS AND ONE COLOR WITH IT.

Different variations of the game include, for instance, giving information on which position has been guessed correctly, avoiding repeated symbols in the secret combination (*bulls and cows* is actually this way), or allowing the codemaker to change the code during the game (but only if this does not make responses made so far false), or even try to find out what is the minimum number of combinations whose answer would allow to determine the secret code uniquely; this version is called *static* MasterMind [6].

In any case, the codebreaker is allowed to make a maximum number of combinations (usually fifteen, or more for larger values of κ and ℓ), and its score corresponds to the number of combinations needed to find the secret code; after repeating the game a number of times with codemaker and codebreaker changing sides, the one with the lower score wins.

Since MasterMind is asymmetric, in the sense that the position of one of the players after setting the secret code is almost completely passive, and limited to give hints as a response to the guesses of the codebreaker, it is rather a puzzle than a game: the codebreaker is not really matching her skills against the codemaker, but facing a problem that must be solved with the help of hints, the implication being

J. J. Merelo and Antonio M. Mora are with the Dept. of Architecture and Computer Technology, ETSIIT, University of Granada, email: jj@merelo.net, amorag@geneura.ugr.es

Thomas P. Runarsson is with the School of Engineering and Natural Sciences, University of Iceland, email: tpr@hi.is

Carlos Cotta is with the Departament of Languages and Computer Sciences at the University of Málaga, email: ccottap@lcc.uma.es

978-1-4244-6297-1/10/$26.00

that playing MasterMind is more similar to solving a Sudoku than to a game of chess. Thus, the solution to MasterMind, unless in a very particular situation (always playing with an opponent who has a particular bias for choosing codes, or maybe playing the dynamic code version), is a constrained combinatorial search, and not a learning problem.

Solving MasterMind is interesting by itself; historically has been applied a good amount of strategies, and that is why, specifically, the solution of MasterMind using evolutionary algorithms is used in some artificial intelligence classes or even as part of computer-supported cooperative learning experiments [7]. However, it is also interesting for its relationship to other generally called *oracle* problems such as circuit and program testing, differential cryptanalysis, uniquely identifying a person from queries to a genetic database [8] and other puzzle-like games and problems (the interested reader is referred to [9] for a more complete, although currently outdated, review), the fact that it has been proved to be NP-complete [10], [11] and that there are several open issues, namely, what is the lowest average number of guesses you can achieve (solved only for the smallest sizes), how to minimize the number of evaluations needed to find them (and thus the run-time of the algorithm), and obviously, how it scales when increasing κ and ℓ. This paper will concentrate on the second issue, namely, minimizing the number of evaluations needed to find a good solution, but without forgetting the fact that the objective of the game is to find the secret combination in a minimum number of guesses.

That is why, within our ultimate target of providing an evolutionary algorithm for solving MasterMind with a good scaling behavior with problem size, in this paper, after reviewing how the state of the art in solving this puzzle has evolved in the last years and showing how different evolutionary algorithms fare against each other, we build on previous work [12] by introducing a new fitness function designed to minimize the number of evaluations (and guesses), thus making the method faster, and compare results obtained with EAs using other fitness functions (and other methods, evolutionary and heuristics based on exhaustive search, found in the literature). At the same time, our target is to explore the EA parameter space in order to find a set that minimizes the number of evaluations, making then the algorithm as fast as possible. Finally, we try to explain the reasons for the different results found with the current and former fitness functions by looking at several features in the evolutionary run.

The rest of the paper is organized as follows: next we establish the terminology and examine the state of the art in algorithms that solve MasterMind (including evolutionary algorithms); then the new version of the evolutionary algorithms that we introduce in this paper are presented in Section III and the experimental results in IV; finally, conclusions are drawn in the closing Section V.

II. Definitions and state of the art

Before presenting the state of the art, some definitions are needed. We will use the term *response* for the codes returned by the codemaker to a played combination, c_{played}. A response is therefore a function of the combination, c_{played} and the secret combination c_{secret}. Let the response be denoted by $h(c_{played}, c_{secret})$. A combination c is *consistent with* c_{played} if, and only if,

$$h(c_{played}, c_{secret}) = h(c_{played}, c) \tag{1}$$

that is, if the combination has as many black and white pegs with respect to the played combination as the played combination with respect to the secret combination. Furthermore, a combination c is *consistent* if, and only if,

$$h(c_i, c) = h(c_i, c_{secret}) \ \forall \ i = 1..n \tag{2}$$

where n is the number of combinations, c_i, played so far; that is, c is *consistent with* all guesses made so far. A combination that is consistent is a candidate solution.

The concept of consistent combination will be important for characterizing different approaches to the game of MasterMind. The naive approach is to play a combination as soon as it is found, in which case the object is to find a consistent guess as fast as possible. For example, in [9] an evolutionary algorithm is described for this purpose. These strategies are fast and do not need to examine a big part of the space. Playing consistent combinations eventually produces a number of guesses that uniquely determine the secret code. However, both the maximum and mean number of combinations that need to be examined are usually high, even as large as the search space itself (when there is a single consistent combination left –last draw– on average the whole search space will have to be tested to find it). Hence, some bias must be introduced in the way how combinations are searched; if not, the guesses will be no better than a purely random approach, as solutions found (and played) are a random sample of the space of consistent guesses.

However, this strategy is not optimal from the point of view of game playing since, after each combination is played, every consistent combination in the set will yield a different result, and a different reduction in search space size, once played. That is why most algorithms rely on finding out which combination outside the consistent set is expected to maximally reduce the set of remaining combinations. For this a number of heuristics have been developed over the years; typically they require all consistent guesses to be found first, and then use some kind of search over the space of consistent combinations, so that only the guess that (in expected value) extracts the most information from the secret code is issued, or else the one that reduces as much as possible the set of remaining consistent combinations. However, this is obviously not known in advance, so a strategy must be devised to find this out, and this strategy is based on the concept of *partitions* (also called Hash Collision Groups, HCG [4]), which are created in the following way: to each combination corresponds a partition of the rest of

Combination	Number of combinations in the partition with response									
	0b-2w	0b-3w	0b-4w	1b-1w	1b-2w	1b-3w	2b-0w	2b-1w	2b-2w	3b-0w
AABA	0	0	0	14	8	0	13	1	0	3
AACC	8	0	0	10	5	0	8	4	1	3
AACD	6	2	0	11	6	1	4	5	1	3
AACE	6	2	0	11	6	1	4	5	1	3
AACF	6	2	0	11	6	1	4	5	1	3
ABAB	8	0	0	10	5	0	8	4	1	3
ABAD	6	2	0	11	6	1	4	5	1	3
ABAE	6	2	0	11	6	1	4	5	1	3
ABAF	6	2	0	11	6	1	4	5	1	3
ABBC	4	4	0	10	8	0	8	1	1	3
ABDC	3	4	1	11	9	1	4	2	1	3
ABEC	3	4	1	11	9	1	4	2	1	3
ABFC	3	4	1	11	9	1	4	2	1	3
ACAA	0	0	0	14	8	0	13	1	0	3
ACCB	4	4	0	10	8	0	8	1	1	3
ACDA	0	0	0	16	10	2	5	3	0	3
BBAA	8	0	0	10	5	0	8	4	1	3
BCCA	4	4	0	10	8	0	8	1	1	3
BDCA	3	4	1	11	9	1	4	2	1	3
BECA	3	4	1	11	9	1	4	2	1	3
BFCA	3	4	1	11	9	1	4	2	1	3
CACA	8	0	0	10	5	0	8	4	1	3
CBBA	4	4	0	10	8	0	8	1	1	3
CBDA	3	4	1	11	9	1	4	2	1	3
CBEA	3	4	1	11	9	1	4	2	1	3
CBFA	3	4	1	11	9	1	4	2	1	3
DACA	6	2	0	11	6	1	4	5	1	3
EBAA	6	2	0	11	6	1	4	5	1	3
FACA	6	2	0	11	6	1	4	5	1	3
FBAA	6	2	0	11	6	1	4	5	1	3

TABLE II

TABLE OF PARTITIONS AFTER TWO COMBINATIONS HAVE BEEN PLAYED; THIS TABLE IS THE RESULT OF COMPARING EACH COMBINATION AGAINST ALL THE REST OF THE SET, WHICH IS THE SET OF CONSISTENT COMBINATIONS IN A GAME AFTER TWO COMBINATIONS HAVE ALREADY BEEN PLAYED. IN BOLDFACE, THE COMBINATIONS WHICH HAVE THE MINIMAL WORST SET SIZE (WHICH HAPPEN TO BE IN THE 1B-1W COLUMN, BUT IT COULD BE ANY ONE); IN THIS CASE, EQUAL TO TEN. A STRATEGY THAT TRIES TO MINIMIZE WORST CASE WOULD PLAY ONE OF THOSE COMBINATIONS. THE COLUMN 0B-1W WITH ALL VALUES EQUAL TO 0 HAS BEEN SUPPRESSED; COLUMN FOR COMBINATION 3B-1W, BEING IMPOSSIBLE, IS NOT SHOWN EITHER. SOME ROWS HAVE ALSO BEEN ELIMINATED FOR LACK OF SPACE.

the space, according to their match (the number of blacks and white pegs that would be the response when matched with each other). Let us consider the first combination in Table I: if the combination considered is AABB, there will be 256 combinations whose response will be 0b, 0w (those with other colors), 256 with 0b, 1w (those with either an A or a B), and so on. Some partitions may also be empty, or contain a single element (4b, 0w will contain just AABB, obviously). For a more exhaustive explanation see [13]; the whole partition set for an advanced stage of the game is shown in Table II. Each combination is thus characterized by the features of these partitions: the number of non-empty ones, the average number of combinations in them, the maximum, and other characteristics one may think of. For instance, in Table II combination ABDC (and others) would have a maximum of non-zero partitions (none of them has zero elements), while AACC (and others, shown in boldface) have a minimum worst-case partition size (equal to ten).

Most heuristic strategies rely on this concept, which was introduced by Knuth [14]. Knuth's algorithm tries to minimize the worst case by following the strategy of minimizing the worst expected set size; for instance, in the set in Table II this algorithm would play one of the combinations shown in boldface (the first one in lexicographical order, since it is a deterministic algorithm). Using a complete minimax search Knuth shows that a maximum of 5 guesses are needed to solve the game using this strategy.

The path leading to the most successful heuristic strategies to date include the minimization of the *worst case* [14] or *expected case* [15], or maximization of *entropy* [16], [17] or *number pf partitions* [13]. Among these, the *entropy* strategy selects the guess with the highest entropy, computed as follows: for each possible response i for a particular consistent guess, the number of remaining consistent guesses is found. The ratio of reduction in the number of guesses is also the *a priori* probability, p_i, of the secret code being in the corresponding partition. The entropy is then computed as $\sum_{i=1}^{n} p_i \log_2(1/p_i)$, where $\log_2(1/p_i)$ is the information in bit(s) per partition, and can be used to select the next combination to play in MasterMind [16]. The *worst case*

is a one-ply version of Knuth's approach, but Irving [15] suggested using the *expected case* rather than the worst case. Kooi [13] noted, however, that the size of the partitions is irrelevant and that rather the number of non empty partitions created, n, was important. This strategy is called *most parts*. The strategies above require one-ply look-ahead and either determining the size of resulting partitions and/or its number. Computing its number is, however, faster than determining their expected size or entropy. For this reason the *most parts* strategy has a computational advantage, and has been used by us in our previous work [12].

The evolutionary algorithms that try to solve this problem have also historically proceeded roughly in the same way. After using naive strategies that played the first combination found, [18], using suboptimal strategies with the objective of avoiding the search from getting stuck [19], or even playing the best guess each generation in a policy that resulted in a fast and very bad solution to the puzzle [20], [21], better solutions were found, either from the point of view of the evolutionary search or from the game-playing one [9].

However, it was not until recently when Berghman et al. [22] adopted the method of partitions to an evolutionary algorithm. The strategy which they apply is similar to the *expected size* strategy. However, it differs in some fundamental ways. In their approach each consistent guess is assumed to be the secret in turn, and each guess is played against every different secret. The return codes are then used to compute the size of the subset of remaining consistent guesses in the set, assuming that either similarity or dissimilarity (represented by the average number of black plus white pegs) will be indicative of that size; eventually the number is maximized after proving that using the most similar (highest number of pegs) obtains the lowest average number of guesses. An average is then taken over the size of these sets. Here, the key difference between the *expected size* method is that only a subset of all possible consistent guesses is used and some return codes may not be considered (or considered more frequently than once), which might lead to a bias in the result. Indeed they remark that their approach is computationally intensive which leads them to reduce the size of this subset further (by taking only a small sample); but their results, either for the basic game or for higher-dimensional varieties, are quite good and comparable with those obtained considering heuristic strategies, with the main difference that they are not using an exhaustive search algorithm which can then scale better than them.

The heuristic strategies described above use some kind of look-ahead, which is computationally expensive. If no look-ahead is used to guide the search, a guess is selected purely at *random*, and any other way of ranking solutions might find a solution that is slightly better than this approach, but no more. However, it has been shown in [23] that in order to get the benefit of using look-ahead methods, an exhaustive set of all consistent combinations is not needed; a 10% fraction is enough to find solutions that are statistically indistinguishable from the best solutions found. This amount was statistically established, and then tested in an evolutionary algorithm in [12], where the *most-partitions* strategy was used over a set of consistent guesses whose size was computed statistically, and which had been found using evolutionary algorithms (estimation of distribution and canonical genetic algorithms), to obtain solutions that were quite competitive, for the basic case, with the exhaustive search strategies, and significantly better than random search; they were also similar to the results obtained by Berghman et al., but using a smaller set size and a computationally simpler strategy.

What we will do in this paper is try and improve on those results above, by using a new fitness function that takes into account the partitioning of search space by each consistent solution, and by trying to find a set of parameters that combine good results in game play and a low number of evaluations to find the solution.

III. Description of the method

Essentially, the method used in this paper is an hybrid between an evolutionary algorithm and the exhaustive partition-based methods described above, as has been mentioned. Instead of using exhaustive search to find a set of consistent combinations, which are then compared on how they divide that set (in partitions), we use evolutionary algorithms to find a set of consistent combinations and compute then the partitions they yield. The size of the set is established according to our previous results [23], which show that a set of size 20 is enough to obtain results that are statistically indistinguishable from using the whole set of consistent combinations.

In a previous paper [12] we used as fitness function one similar to the one used by previous evolutionary algorithms [22], [9], except for the term proportional to the number of positions, that is:

$$f(c_{guess}) = -\sum_{i=1}^{n} |h(c_i, c_{guess}) - h(c_i, c_{secret})| \quad (3)$$

which is the number of black and white peg changes needed to make the current combination c_{guess} consistent; this number is computed via the absolute difference between the number of black and white pegs h the combination c_i has had with respect to the secret code c_{secret} (which we know, since we have already played it) and what c_{guess} obtains when matched with c_i; being h a vectorial function, $||$ is then equivalent to the taxicab distance or L_1. For instance, if the played combination $ABBB$ has obtained as result $2w, 1b$ and our c_{guess} $CCBA$ gets $1w, 1b$ with respect to it, this difference will be $|2-1|w + |1-1|b = 1$. This operation is repeated over all the combinations c_i that have been played. There was a problem with this fitness: if a combination is consistent, its fitness was equal to zero, and also equal to all the other consistent partitions, creating a neutral evolution landscape which impeded the progress of evolution for them. Please note that this is only one possible distance; instead of L_1 another distance such as the Chebyshev distance or L_∞ could be used; this distance takes the maximum distance

of any dimension; in this case, the maximum difference either in black or white pegs. In preliminary tests done using this distance as fitness, we have found that there was no significative difference with the one we have used. In fact, other distances (L_2 or Euclidean, for instance) could also be considered; although they will have an impact in the fitness landscape, they will probably have little, if any, influence in the solution.

However, this distance is not the only factor: as we have seen in the previous section, not all combinations have the same ability to solve the puzzle, so it is sensible to include whatever score we are using to rank them also within the fitness function. That is what we introduce in this paper: initially, the fitness of non-consistent solutions is computed as $f(c_{guess})$. Let us call this score g, which is then defined as:

$$g(c_{guess}) = \begin{cases} f(c_{guess}) & f(c_{guess}) > 0 \\ P(c_{guess}) & f(c_{guess}) = 0 \end{cases} \quad (4)$$

That is, for a consistent solution fitness is the number of non-empty partitions (noted with P), resulting in negative fitness for non-consistent and positive for consistent-solutions. This fitness g is then lineally transformed (for using it in fitness-proportional selection methods) by adding $1-(Minf(c_{guess}))$, so that the worst non-consistent solution will have fitness 1 and the best consistent solution its initial number of partitions plus one plus the minimum negative distance to consistency, ensuring also that consistent solutions are always better than non-consistent ones, and also different depending on their score, which can then be used by evolution to improve the population.

The rest of the algorithm is also a canonical genetic algorithm with 1-point crossover, single-character mutation and fitness-proportional selection, the same as the one we have used as a baseline. The first combination has fixed to $ABCA$ (as used in most papers; in any case, the selection of the first combination does not have a big impact in the result), same as before, working as follows:

- Continue evolutionary search until at least 20 consistent solutions are found.
- If a set of 20 solutions is not found, continue until the number of consistent solutions does not change for three generations. This low number was chosen to avoid stagnation of the population.
- If at least a single consistent solution is not found after 50 (in previous papers we used 15) generations, reset the population substituting it by a random one. Again, this was a number considered high enough to imply stagnation of search, giving at the same time the algorithm a chance to find solutions.

This means that, when the evolutionary loop exits, we always have a non-zero set of consistent solutions; if it contains a single individual, it is played; if it contains several, one is randomly chosen among those with the highest number of non-zero partitions.

In principle, any kind of evolutionary algorithm can be used internally to evolve new solutions; however, we used a canonical evolutionary algorithm (CGA) with one-point crossover and single-character mutation. Solutions were represented directly, without binary codification, with crossover and mutation taking thus place over the characters. Evolutionary parameters are shown in Table III in the column labelled EvoRank (the name of our approach).

Parameter	EvoRank	CGA
Population	128, 256, 400	400
Replacement Rate	0.25, 0.4, 0.5, 0.75,	0.4
Generations to reset	50	15

TABLE III
PARAMETER VALUES IN THE EVOLUTIONARY ALGORITHMS USED IN THIS PAPER.

Together with this algorithm, we tested the one that obtained the best results in our previous paper [12], a canonical evolutionary algorithm; the main difference is that it considers as fitness only the distance to consistency $(1/(1+f(c_{guess})))$, using the partition method to select, from all the consistent solutions, the combination that is going to be played. Parameters are also shown in Table III, in the column labelled CGA [1].

We will compare these two algorithms from the dual point of view of average combinations played and number of combinations evaluated; that is why we will also include in the comparison a random *naive* algorithm that plays as soon as a consistent solution is found. All experiments have been made by applying the algorithm 10 times over the whole set of 1296 combinations.

IV. EXPERIMENTAL RESULTS

The results for the experiments performed over several variants of the method are shown in Table IV. When EvoRank (the method introduced in this paper) uses a population of 400 and a default replacement rate of 40%, its results are statistically indistinguishable from the exhaustive search strategy that uses the same method to select the consistent combination to play (which is labelled with *Most Parts*). Besides, these results are better than a virtually identical algorithm (but for the new fitness function) that uses a canonical GA to find a set of consistent solutions, but not the score of its partitions in the fitness (CGA). Besides, these results are comparable to those published by Berghman et al.'s [22]. It must be stressed that just three runs of the whole combination space were made in [22], and not ten, as we have done in this case, and therefore it is difficult to assess the significance of those results. In fact, [22] does not even include data on standard deviations.

Let us look at the second objective of this paper: to find out if the number of evaluations needed to find those results is also better in the case of the algorithm with the new fitness function. Several parameter settings for the canonical GA with partitions have been tested, along with several

[1]source code and all parameter files used for these experiments are available as GPL'ed code from `http://sl.ugr.es/alg_mm/`

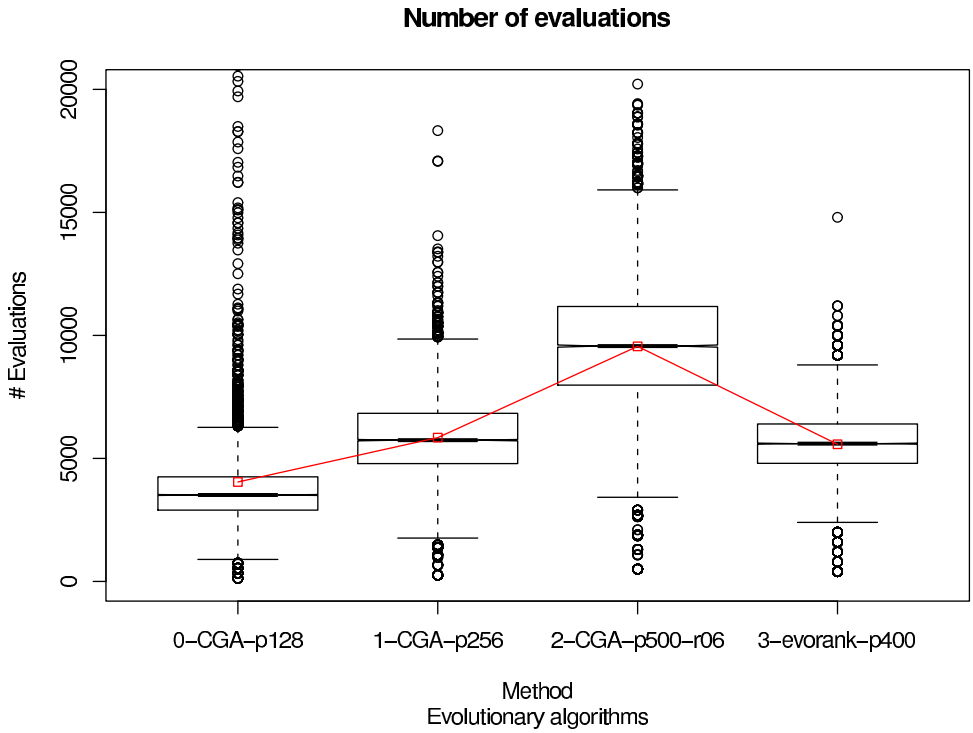

Fig. 1. Boxplot of the number of evaluations for several settings of the canonical CGA algorithm, and the new EvoRank algorithm, proposed here. The points indicate the average, which is joined by a line for comparison purposes. The results for EvoRank with population=400 are, in fact, very similar to those obtained by CGA with other settings, but there are less outliers; the minimum average number of evaluations corresponds to CGA with $p = 128$, but with too many outliers.

population sizes and replacement rates, always running 10 times over every combination in the space; results are plotted in Figure 1. In general, it has been observed that the number of evaluations grows with population size, at the same time that the average number of combinations decreases. However, the number of evaluations for EvoRank grows less fast than for the CGA; and, in fact the number of evaluations yielded for a particular population size (for instance, 400), is less for EvoRank than for CGA, around 1/3rd less; as can be seen in Figure 1. The number of evaluations for EvoRank and $p = 400$ is on average, similar to the number of evaluations for the CGA with $p = 256$. In fact, if, for all experiments made, we plot the number of evaluations vs. the average number of combinations played, we obtain what we see in Figure 2, that is, the CGA method is able to achieve a good game-playing score, but only at the expense of using a higher average number of evaluations, and for a comparable number of evaluations, EvoRank is able to find the solution with a lower average number of combinations; see for instance the *row* with 4000 and 6000 evaluations. This difference accounts for a lower running time, obviously; all in all, the complete simulation takes a few hours in a dual-core AMD Phenom II, with less than one second for each game.

Once it has been established that the new fitness function obtains better results, it remains to try and find out why it does so. In principle, the fact that it needs less evaluations

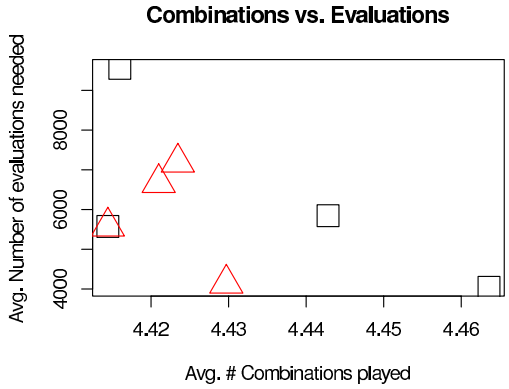

Fig. 2. Number of evaluations plotted against the average number of combinations to solution for different parameter settings in the EvoRank method (triangles) and a CGA (squares). The best non-dominated solution indicated with both, a square and a triangle, and corresponds to EvoRank with $p = 400$, but please note that the Evorank solution at 4.43 and the CGA solution at 4.46 are also non dominated; however, they are worse in the most important objective, the number of draws.

implies that it is better at finding the best combinations, and if it, besides, plays better, it means that the sampling of the space of consistent combinations is biased towards finding better solutions. In part, it should be expected since the fitness function is designed to behave that way, but, does

Strategy	*min*	*mean*	*median*	*max*	*st.dev.*
Berghman (60)	-	4.390	-	-	-
Entropy	4.383	4.408	4.408	4.424	0.012
Most parts	4.383	4.410	4.412	4.430	0.013
evorank-p400	4.392	4.414	4.413	4.448	0.018
evorank-p256-r05	4.408	4.430	4.433	4.444	0.013
Berghman (30)	-	4.430	-	-	-
CGA	4.402	4.434	4.433	4.471	0.018
Expected size	4.447	4.470	4.468	4.490	0.015
Worst case	4.461	4.479	4.473	4.506	0.016
evorank-p128	4.414	4.457	4.460	4.498	0.023
evorank-p128-r025	4.443	4.476	4.483	4.502	0.020
evorank-p128-r075	4.478	4.498	4.497	4.522	0.017
Random	4.566	4.608	4.608	4.646	0.026

TABLE IV

COMPARISON OF RESULTS FOR DIFFERENT MASTERMIND-SOLVING STRATEGIES, INCLUDING THE ONES INTRODUCED IN THIS PAPER AND LABELLED evorank AND SHOWN IN serif FONT. THE p SUFFIX CORRESPONDS TO POPULATION, AND r TO REPLACEMENT RATE; DEFAULT REPLACEMENT RATE IS 0.4. HORIZONTAL LINES SEPARATE RESULTS THAT ARE STATISTICALLY DIFFERENT, BUT FOR BERGHMAN ET AL.'S RESULTS, ON WHICH NOT ENOUGH INFORMATION IS AVAILABLE. BERGHMAN'S RESULTS ARE TAKEN FROM [22]; ENTROPY, MOST PARTS, EXPECTED SIZE AND WORST CASE ARE HEURISTIC, EXHAUSTIVE SEARCH METHODS, AND VALUES WERE COMPUTED IN [23]. RESULTS FOR THE CANONICAL GENETIC ALGORITHM ARE TAKEN FROM [12].

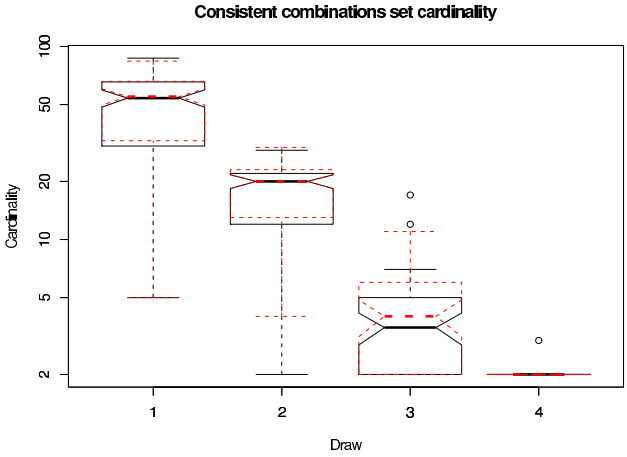

Fig. 3. Cardinality of the set of consistent combinations after each move (or draw) for EvoRank (black, solid boxes) and CGA (red, dashed boxes). Except for the slightly different situation after the third move, the values are virtually the same, as shown by the overlap of boxplot notches.

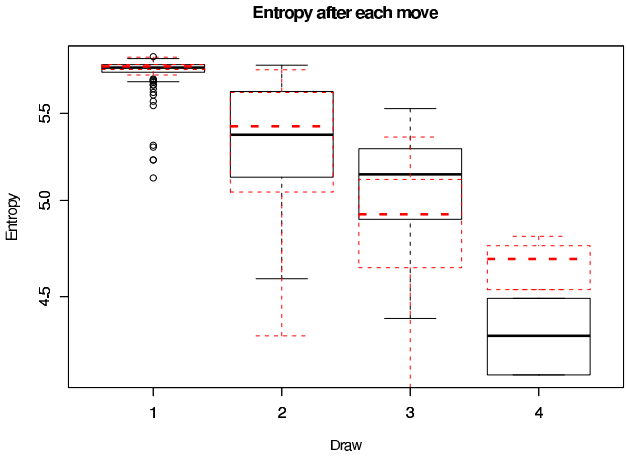

Fig. 4. Entropy of the set of consistent combinations after each move (or draw) for EvoRank (black, solid boxes) and CGA (red, dashed boxes).

it have any other effect on the evolutionary algorithm? For instance, does it find better solutions because it creates bigger sets of consistent combinations? Or because, since it creates a rougher fitness landscape, it is able to maintain diversity better?

In order to check this, we have performed an independent experiment: we have run the algorithm 100 times, with the same setup (population = 400, replacement rate=0.4), and, as mentioned above, the same underlying evolutionary algorithm, and measured the cardinality of the set of consistent combinations and genotypic entropy after playing each combination. The results are shown in Figures 3 and 4, the first of which shows that the cardinality of the sets is virtually the same; in fact, the cardinality of the first moves is entirely determined by the random creation of the population, and later on by the cutoff size, which is equal to twenty and the same for both. From this it can be concluded that the better solutions found by EvoRank is due to the selection of *better* consistent solutions, because the sample used has got the same size.

However, are these better solutions found because the fitness function introduces more diversity, since it has got many more different values? In CGA, all consistent solutions have got the same fitness (equal to one, as explained above); in EvoRank, its fitness will depend to its score. Well, in fact this does not happen, as shown in Figure 4. Entropy is indeed different for both methods, but there is no clear trend: it seems higher after the second and fourth move (remember that this is an average of 100 games) for the canonical GA, but worse after the third move. This might imply that the exploration/exploitation balance behave in different ways in the two methods: exploration seems to be better in the determinant third move for EvoRank, but it switches to exploitation in the fourth move, thus being able to find the solution faster; but, in fact, this higher diversity in the CGA case (at least for the last move) can be explained by the creation of a neutral fitness landscape by all combinations with the same fitness value, something that is not so frequent in EvoRank.

From these experiments we can then conclude that EvoRank, a Canonical Genetic Algorithm with a fitness that includes the partition score (in this case, the number of partitions) is a better choice for solving MasterMind, and that this result is due to the better sampling of the solution space that is done by this fitness function.

V. Conclusion and future work

In this paper we have introduced a new fitness function for solving the game of MasterMind using evolutionary algorithms, and found that not only it finds better solutions, but it does so in less evaluations (and thus time). Besides, its results are comparable to those found using exhaustive search algorithms, which holds some promise when solving problems with higher dimensions. This new fitness function is a result of seeing the problem from a different and novel point of view: what we are seeking are the best consistent solutions; the fact that they are consistent or not is actually a constraint, which, incidentally, takes us back to the first papers on the subject, which considered MasterMind a constrained optimization problem [18]. This result can also be generalized to any type of evolutionary algorithm: always take into account heuristics when either designing the fitness function or creating operators.

It remains to be seen also how this strategy scales to higher sizes; the first move would be, similarly to what was done in [23], to compute the size of the set whose results are statistically indistinguishable from using the whole set of consistent combinations for bigger sizes, and maybe get an estimate of how it scales with problem size; then use that set size in evolutionary algorithms to obtain performance, and once again see how the solutions scale with problem size, to check whether they scale in the same way as exhaustive search algorithms or have a better behavior. There is an additional problem with bigger sizes: it becomes impossible to test exhaustively the whole search space, so finding a complete set of combinations on which to obtain accurate results is also a challenge. For the time being, besides, we have only used standard evolutionary operators. Using string permutation, and maybe other specially designed operators will reduce even further the amount of evaluations needed to find the solution.

Finally, as was done in previous approaches [9] to MasterMind with good results, it would be interesting to test whether heuristics tricks, such as using *endgames*, that is, deterministic or exhaustive search methods, when certain situations arise (a combination gets all blacks but one, or all whites, or zero blacks/whites) would enhance the algorithms, and in which way (number of evaluations and average number of combinations played).

Acknowledgments

This paper has been funded in part by the Spanish MICYT projects NoHNES (Spanish Ministerio de Educación y Ciencia - TIN2007-68083), Spanish MICINN project NEMESIS (Spanish Ministerio de Ciencia e Innovación - TIN2008-05941) and the Junta de Andalucía P06-TIC-02025 and P07-TIC-03044.

References

[1] E. W. Weisstein, "Mastermind." From MathWorld–A Wolfram Web Resource. [Online]. Available: http://mathworld.wolfram.com/Mastermind.html

[2] Wikipedia, "Mastermind (board game) — Wikipedia, The Free Encyclopedia," 2009. [Online]. Available: http://sl.ugr.es/001X

[3] ——, "Bulls and cows — Wikipedia, the free encyclopedia," 2009. [Online]. Available: http://sl.ugr.es/001W

[4] S.-T. Chen, S.-S. Lin, and L.-T. Huang, "A two-phase optimization algorithm for mastermind," *Computer Journal*, vol. 50, no. 4, pp. 435–443, 2007.

[5] A. Heeffer and H. Heeffer, "Near-optimal strategies for the game of Logik," Gent University, Tech. Rep., 2007. [Online]. Available: http://logica.ugent.be/albrecht/thesis/Logik.pdf

[6] W. Goddard, "Static mastermind," *Journal of Combinatorial Mathematics and Combinatorial Computing*, vol. 47, pp. 225–236, 2003.

[7] A. Monteserin, S. Schiaffino, and A. Amandi, "Assisting students with argumentation plans when solving problems in CSCL," *Computers and Education*, vol. 54, no. 2, pp. 416 – 426, 2010. [Online]. Available: http://http://dx.doi.org/10.1016/j.compedu.2009.08.025

[8] M. Goodrich, "On the algorithmic complexity of the Mastermind game with black-peg results," *Information Processing Letters*, vol. 109, no. 13, pp. 675–678, 2009.

[9] J. J. Merelo-Guervós, P. Castillo, and V. Rivas, "Finding a needle in a haystack using hints and evolutionary computation: the case of evolutionary MasterMind," *Applied Soft Computing*, vol. 6, no. 2, pp. 170–179, January 2006. [Online]. Available: http://dx.doi.org/10.1016/j.asoc.2004.09.003

[10] J. Stuckman and G.-Q. Zhang, "Mastermind is NP-complete," *INFOCOMP J. Comput. Sci*, vol. 5, pp. 25–28, 2006. [Online]. Available: http://arxiv.org/abs/cs/0512049

[11] G. Kendall, A. Parkes, and K. Spoerer, "A survey of NP-complete puzzles," *ICGA Journal*, vol. 31, no. 1, pp. 13–34, 2008.

[12] J.-J. Merelo and T. P. Runarsson, "Finding better solutions to the mastermind puzzle using evolutionary algorithms," ser. Lecture Notes in Computer Science, C. D. Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. I. Esparcia-Alcázar, C.-K. Goh, J. Merelo, F. Neri, M. Preuss, J. Togelius, and G. N. Yannakakis, Eds., vol. 6024, no. 6024. Istanbul, Turkey: Springer-Verlag, 7 - 9 Apr. 2010, pp. 120–129, evoApplications2010 to be held in conjunction with EuroGP'2010, EvoCOP2010 and EvoBIO2010. To appear.

[13] B. Kooi, "Yet another Mastermind strategy," *ICGA Journal*, vol. 28, no. 1, pp. 13–20, 2005.

[14] D. E. Knuth, "The computer as Master Mind," *J. Recreational Mathematics*, vol. 9, no. 1, pp. 1–6, 1976-77.

[15] R. W. Irving, "Towards an optimum Mastermind strategy," *Journal of Recreational Mathematics*, vol. 11, no. 2, pp. 81–87, 1978-79.

[16] E. Neuwirth, "Some strategies for Mastermind," *Zeitschrift fur Operations Research. Serie B*, vol. 26, no. 8, pp. B257–B278, 1982.

[17] A. Bestavros and A. Belal, "Mastermind, a game of diagnosis strategies," *Bulletin of the Faculty of Engineering, Alexandria University*, December 1986. [Online]. Available: http://citeseer.ist.psu.edu/bestavros86mastermind.html

[18] J. J. Merelo, "Genetic Mastermind, a case of dynamic constraint optimization," 1996, GeNeura Technical Report G-96-1, Universidad de Granada.

[19] J. L. Bernier, C.-I. Herráiz, J.-J. Merelo-Guervós, S. Olmeda, and A. Prieto, "Solving *mastermind* using GAs and simulated annealing: a case of dynamic constraint optimization," in *Proceedings PPSN, Parallel Problem Solving from Nature IV*, ser. Lecture Notes in Computer Science, no. 1141. Springer-Verlag, 1996, pp. 553–563.

[20] L. Bento, L. Pereira, and A. Rosa, "Mastermind by evolutionary algorithms," in *Procs SAC 99*, 1999, pp. 307–311.

[21] T. Kalisker and D. Camens, "Solving Mastermind using genetic algorithms," in *GECCO 2003, Genetic and Evolutionary Computation Conference*, ser. Lecture Notes in Computer Science, E. C.-P. et al., Ed., no. 2724. Springer Verlag, 2003, pp. 1590–1591.

[22] L. Berghman, D. Goossens, and R. Leus, "Efficient solutions for Mastermind using genetic algorithms," *Computers and Operations Research*, vol. 36, no. 6, pp. 1880–1885, 2009.

[23] T. P. Runarsson and J. J. Merelo, "Adapting heuristic Mastermind strategies to evolutionary algorithms," in *NICSO'10 Proceedings*, ser. LNCS. Springer-Verlag, 2010, to be published, also available from ArXiV: http://arxiv.org/abs/0912.2415v1.