Optimizing search via diversity enhancement in evolutionary MasterMind

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

***Abstract*— A MasterMind player must discover a secret combination by making guesses using the hints obtained as a response to the previous ones. Finding a general strategy that scales well with problem size is still an open issue, despite having been approached from different angles, including evolutionary algorithms. In previous papers we have tested different approaches to the evolutionary MasterMind and having found out that diversity is essential in this kind of combinatorial optimization problems, in this paper we try to tune the search methods to keep a high diversity level and thus obtain solutions to the puzzle in less average evaluations, and, if possible, in less number of combinations played. This will allow us to get improvements in the time that will be used to explore problems of bigger size.**

I. Introduction

MasterMind [1] is a two-player code-breaking game, or in some sense a single-player puzzle, where one of the players –the *codemaker* (CM)– has no other role in the game than setting a hidden combination, and automatically providing hints on how close the other player –the *codebreaker* (CB)– has come to correctly guess this combination. More precisely, the flow of the game is as follows:

- The CM sets and hides a length ℓ combination of κ symbols. Therefore, the CB is faced with κ^ℓ candidates for the hidden combination, which is typically represented by an array of pegs of different colors (but can also be represented using any digits or letter strings) and hidden from the CB.
- The CB tries to guess this secret code by producing a combination with the same length, and using the same set of symbols. As a response to that move, the CM acting as an *oracle* (which explains the inclusion of this game in the category called *oracle* games) provides information on the number of symbols guessed in the right position (black pegs in the physical board game), and the number of symbols with the correct color, but in an incorrect position (white pegs); this is illustrated in Table I.
- The CB uses (or not, depending on the strategy he is following) this information to produce a new combination, that is assessed in the same way. If he correctly guesses the hidden combination in at most N attempts, the CB wins. Otherwise, the CM takes the game. N usually corresponds to the physical number of rows in the game board, which is equal to fifteen in the first commercial version.
- CM and CB are then interchanged, and several rounds of the game are played. The player that is able to obtain the minimal amount of attempts wins. This is a part of the game we do not consider.

This puzzle is, in fact, a quite interesting combinatorial problem, as it relates to other *oracle* problems such as the hacking of the PIN codes used in bank ATMs [2] or uniquely identifying a person from queries to a genetic database [3]. Several issues remain open, such as what is the lowest average number of guesses needed to solve the problem for any given κ and ℓ. Associated to this, there arises the issue of coming up with an efficient mechanism for finding the hidden combination independently of the problem size, or at least, a method that scales gracefully when the problem size increases.

This paper is mainly concerned with tuning some search parameters and methods in the evolutionary algorithm to find the set that produces the best solutions in the least number of combinations examined. In our previous paper in this line of research [4] we introduced several mechanisms that decreased the number of combinations that were needed to find the solution mainly through *endgames*, i.e. heuristics that abandoned the evolutionary algorithm or reduced the search space when certain conditions were met. In this paper we will mainly look at the evolutionary algorithm itself, introducing operators that boost diversity so that search is enhanced and, at the same time, better solutions can be found. Our main intention is, once again, to find rules that are as general as possible and that can be applied to a wide range of problem sizes, eventually creating a solving method that can be used to find the solution in real time for that range.

The rest of the paper is organized as follows: next section presents the state of the resolution of the MasterMind puzzle and its evolution, having special emphasis in evolutionary solutions; we will explain then (in Section III) the general characteristics of the evolutionary algorithm used to solve MasterMind in this work. Results obtained with this setup will be presented in Section IV, and we will finish the paper with the conclusions that derive from them (commented in Section V).

II. Background

As mentioned in Section I, a MasterMind problem instance is characterized by two parameters: the number of colors κ and the number of pegs ℓ. Let $\mathbb{N}_\kappa = \{1, 2, \cdots \kappa\}$ be the set of symbols used to denote the colors. Subsequently, any

JJM and AMM are with the Dept. of Architecture and Computer Technology, ETSIIT, University of Granada, email: jmerelo,amorag@geneura.ugr.es
TPR is with the School of Engineering and Natural Sciences, U. of Iceland, email: tpr@hi.is
CC is with the Dept. of Languages and Computer Sciences at the U. of Málaga, email: ccottap@lcc.uma.es

combination, either the hidden one or one played by the CB, is a string $c \in \mathbb{N}_\kappa^\ell$. Whenever the CB plays a combination c_p, a *response* $h(c_p, c_h) \in \mathbb{N}^2$ is obtained from the CM, where c_h is the hidden combination. A response $\langle b, w \rangle$ indicates that the c_p matches c_h in b positions, and there exist other w symbols in c_p present in c_h but in different positions.

Most strategies use what are called *consistent* combinations; a combination c is *consistent* with another played previously c_p if, and only if, $h(c, c_p) = h(c_p, c_h)$, i.e., if c has as many black and white pegs with respect to the c_p as c_p has with respect to the hidden combination. Intuitively, this captures the fact that c might be a potential candidate for hidden combination in light of the outcome of playing c_p. We can easily extend this notion and denote a combination c as consistent (or feasible) if, and only if, it is consistent with all the combinations played so far, i.e., $h(c, c_p^i) = h(c_p^i, c_h)$ for $1 \leqslant i \leqslant n$, where n is the number of combinations played so far, and c_p^i is the $i-$th combination played. Any consistent combination is a candidate solution.

It is straightforward to see that the number of feasible solutions decreases with each guess made by the CB (as long as she always plays feasible solutions, and unlike it is done in figure I; otherwise no reduction at all might happen, which does not implies that, in the longer term, it might be a better strategy). For the same reason, feasibility is a dynamic property that all the solutions initially have, and eventually loose at some point (depending on the feedback from the CM), except for the hidden combination that always remains feasible. This transient nature of feasibility (or in other words, the decreasing size of the space of potential solutions) turns out to be a central feature in the strategies devised to play MasterMind; in fact, most strategies seek to reduce search space as much as possible, or at least to minimize the possibility of a minimal reduction.

Thus, a very naïve approach is to play a consistent combination as soon as it is found, in which case the objective is to find a consistent guess as fast as possible. For example, in [5] an evolutionary algorithm is described for this purpose. These strategies are fast for small spaces and do not need to examine a big part of the space, at least in the average case. 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. 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.

The main problem of the naïve strategy is the fact that every consistent combination in the set will, once played, yield a different result (or one in a class of different results) and a different reduction in search space size. Not paying attention to these possible outcomes when selecting the combination to be played, is, in fact, naïve. For this reason a sensible algorithm should try to find out which combination within

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.

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

the consistent set is expected to maximally reduce the set of remaining combinations; other possibilities (for instance, playing a fixed –non-sequential– set of combinations until enough information for the hidden combination is gathered) were already laid out by OǴeran et al. in [6] together with the one mentioned above, which is called *first consistent* by them. This leads to a generic framework for defining MasterMind strategies endowed with

1) a procedure for finding a large set (even a complete one) Φ of feasible combinations, and
2) a decision-making procedure to select which combination $c \in \Phi$ will be played.

Regarding the first item, Φ needs not be the full set of feasible solutions at a certain step: as proved in our previous paper [7] a fraction of around 1/6 (for the *classic* mastermind) is enough to find solutions that are statistically indistinguishable from the best solutions found. This was experimentally established and then implemented by the authors in an EA termed *EvoRank* [8].

The procedure mentioned in the second item should minimize the losses of the CB, i.e., reducing the number of feasible solutions in the next step as much as possible. A popular option to achieve this is to rely on the idea of Hash Collision Groups, HCG [9] or *partitions*. These are maximal sets of solutions that will remain feasible given a certain feedback provided by the CM. Since the goal of the CB is to find the hidden combination in as few steps as possible, she is interested in obtaining as small a partition as possible. Therefore, it makes sense to focus on the size of these partitions (which one will be the remaining feasible partition is obviously not known in advance, so some heuristic reasoning is required). As an example, let us consider the first combination in Table I: if the hidden 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 [10]. 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 related to the reduction of the size of the search space that might distinguish them.

To formalize these ideas, let $\vec{\Xi} = \{\Xi_{ibw}\}$ be a three-dimensional matrix that estimates the number Ξ_{ibw} of combinations that will remain feasible after combination c_i is played and response $\langle b, w\rangle$ is obtained from the CM. Then, the potential strategies for the CB are:

1) Minimizing the worst-case partition [11]: pick $c_i = \arg\min_i\{\max_{b,w}(\Xi_{ibw})\}$.
2) Minimizing the average-case partition [12]: pick $c_i = \arg\min_i\{\sum_{b,w} p_{bw}\Xi_{ibw}\}$, where p_{bw} is the prior probability of obtaining a particular outcome. If for instance we compute $p_{bw} = \sum_i \Xi_{ibw} / \sum_{i,b,w} \Xi_{ibw}$.
3) Maximizing the number of potential partitions [10]: pick $c_i = \arg\max_i\{|\{\Xi_{ibw} > 0\}|\}$, where $|C|$ isthe cardinality of set C. This strategy is also called *most parts*.
4) Maximizing the information gained [13], [14]: pick $c_i = \arg\max_i\{H_{b,w}(\Xi_{ibw})\}$, where $H_{b,w}(\Xi_{i[\cdot][\cdot]})$ is the entropy of the corresponding sub-matrix.

On the other hand, the strategy defined by Kooi [10] is based on the assumption that the size of the partitions is irrelevant and that rather the number of non empty partitions created, n, was important. This is an advantageous assumption since computing the number of partitions is faster than determining their expected size or entropy. For this reason the *most parts* strategy has a certain computational advantage, and has been used by us in our previous work [15] and in this one too.

Another major component of the algorithm used in this paper are *endgames* [16], which were added to EvoRank to create the *EvoRank-EG* method; in this context endgames refer to exhaustive search algorithms once that, due to special combination of the constraints (the answers given by the CM), the search space has been quite reduced. These endgames were used in two particular cases: when the answer is *all whites*, which means that the solution is a permutation of the played combination, and when it is *no whites or blacks*, effectively reducing the number of colors in the combination to those that are *not* in the combination played. The general framework will be explained in detail just next.

III. A NEW VERSION OF EVO, THE EVOLUTIONARY ALGORITHM FOR PLAYING MASTERMIND

Previous instances of the MasterMind-solving EA used particular evolutionary algorithms: Estimation of Distribution Algorithms [7] or a Canonical GA [8]. In this paper we have developed the more general evolutionary algorithm introduced in [16] and made some changes designed to keep diversity high, so that we can explore all the possibilities to obtain the required exploitation/exploration balance. We will explain the different parts of this algorithm, and the changes, in the next paragraphs.

The fitness function [15] has two different factors. The first one is the number of black and white peg changes needed to make the current combination c_{guess} consistent [15]:

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

This number is computed via the absolute difference between the number of black and white pegs h that 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.

The second factor was included (in EvoRank [16]) to avoid having all consistent combinations with the same fitness (zero). This created a neutral evolution landscape which impeded the progress of evolution; besides, 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. This was introduced in Initially, the fitness of non-consistent solutions is computed as $f(c_{guess})$. Let us call this score g_{raw}, which is then defined as:

$$g_{\text{raw}}(c_{guess}) = \begin{cases} f(c_{guess}) & f(c_{guess}) < 0 \\ P(c_{guess}) & f(c_{guess}) = 0 \end{cases} \quad (2)$$

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 (in case it is used in fitness-proportional selection methods) by making

$$g(c_{guess}) = g_{\text{raw}}(c_{guess}) + 1 - \min_c\{f(c)\}$$

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, additionally ensuring 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. Most-parts was chosen instead of partition entropy, which is the best option when using the whole consistent set, since, as proved in [7], when only a part of the set is considered as we do in Evo, its results are not statistically different from the most-parts strategy we use here. The first combination is the usual one proposed by Knuth [11]; for $\kappa = 6$ it would be $ABCA$.

The rest of the Evo algorithm was presented in [4]; it includes mechanisms to prevent population stagnation by resetting it after several generations have lapsed without any improvement; the basic idea is to *fill* the set of consistent solutions until it is no longer possible and then play the best combination found. The changes over the previous algorithm [16] brought by this one included using tournament selection (instead of roulette wheel) for increasing selective pressure if needed, a heuristic crossover operator that ensured offspring was different from parents, and new selection operators that selected also parents that where always different.

However, even if this proved an improvement, there was still some room for improving performance even further. Still the average number of combinations examined for reaching a solution was higher than search space size, at least for the smaller sizes; for instance, for $\kappa = 8, \ell = 4$ search space size is $8^4 = 4096$ and average number of combinations examined were 6949. Even as this was not the case for bigger space sizes (the algorithm examined about half the search space), the obvious implication is that combinations were examined several times, or even that there were several copies of some combinations in the population. These were problems that led to stagnation, a high degree of exploration, and eventually worse results than should be expected.

The changes introduced in this paper basically try to overcome those problems by avoiding as much as possible the introduction of repeated combinations, and, at the same time, fixes some bugs that were present in the last version. Our intention with these changes are basically to reduce the number of combinations needed to find the solution so that it scales better and can be carried reasonably to bigger MasterMind problems. These improvements could, in principle, improve the number of moves needed to find the solution, but in advance this is not what we are shooting at. At the same time, in the previous paper [4] we needed to set the evolutionary algorithm parameters in a certain way so that the best results were obtained for a particular problem size. We will try to find a combination that is robust for all sizes, so that we minimize the number of parameters needed to find a good solution in a reasonable amount of time, and reduce them only to population and consistent set size.

The main changes presented in this paper are included in the `Breeder_Diverser` operator included in the `Algorithm::Evolutionary` library [17]. We avoid the introduction of repeated combinations by checking that the offspring of the genetic operators is always different to the parents. We do not check that it is not already in the population, since this would lead to an additional problem; thus, we will not avoid the reintroduction of combinations in the population, just one of the causes: offspring and parents being the same.

At the same time, since this version of the algorithm has more parameters, we perform a partial exploration of parameter space in order to find out the type of influence they have on the outcome (moves and number of combinations examined).

The implementation of this algorithm is available at the writing of this paper at *http://goo.gl/ky4Ge* and will be later on part of the *Algorithm::MasterMind* Perl module published in CPAN (*http://goo.gl/WAU0D*).

IV. Experimental results and discussion

We will test the changes introduced in this paper first by exploring the influence of different parameter values on the outcome, and then by comparing the best results with other published so far. All tests will be made on a base configuration that has $\ell = 4, \kappa = 8$. The MasterMind case is too small to make any differences among configurations,

TABLE II

Values for the Evo parameters that obtain the best result. Permutation, crossover and mutation are *priorities*; they are normalized to 1 to convert them to rates.

Parameter	Value
Crossover	1
Mutation	1
Permutation	1
Replacement rate	0.75
Population	400
Tournament size	2

and this one is large enough to find differences among configurations, but small enough to perform experiments in a reasonable amount of time. All experiments used the code and instance sets uploaded to the repository; in this case the `instancias4_8.txt` which is at *http://goo.gl/6yu16*. A single run over these set of instances, including 5000 combinations (which means that some of them will be repeated) were made. Unless said otherwise, the values for the parameters are as shown in table II.

Evo parameters fall in two different categories, those related to the evolutionary algorithm and those that are problem-specific, such as combination scoring and size of the consistent set sample. We will begin by examining the latter. As we have said, even if most-parts and other strategies for scoring consistent combinations do not have difference for certain consistent-set size, they could have for other problem and consistent set sizes; however, it is very likely that for the testbed we have chosen there is no difference, so we concentrate on checking the influence of the size of the consistent set (which we will call N_{cs} from now on). In order to do that, we will use four different sizes: 25, 30, 35 and 40. The computed size for the smaller problem ($\ell = 4, \kappa = 6$) was 20, so we have started with a size that is slightly bigger to end with twice the size. First result is that there is no significant difference among the average number of moves, which is around 5.16. It is slightly better for $N_{cs} = 30$, but the difference with the worst is not significant using Wilcoxon test. It is also slightly worse than the best value published before (5.148), but difference is not significant.

However, there is a difference among the number of combinations played to find the solution, as shown in figure 1. The average number of evaluations increases with the set size, which is an expected outcome since the algorithm must keep trying to *fill* the set to the desired size. This is rather useless at the latest stages of search, because after the third move the consistent set size is almost always smaller than that size, so this deeper search does not lead to better results, and might even make then worse. Please note that all differences are significant, even as median values are the same for some of them. Best average, at 6412 combinations, is found for $N_{cs} = 30$, which coincides with the best average moves value. Please note that this value is also, already, significantly better than the best value found previously [4] of 6949 combinations. It is, however, still bigger than the search

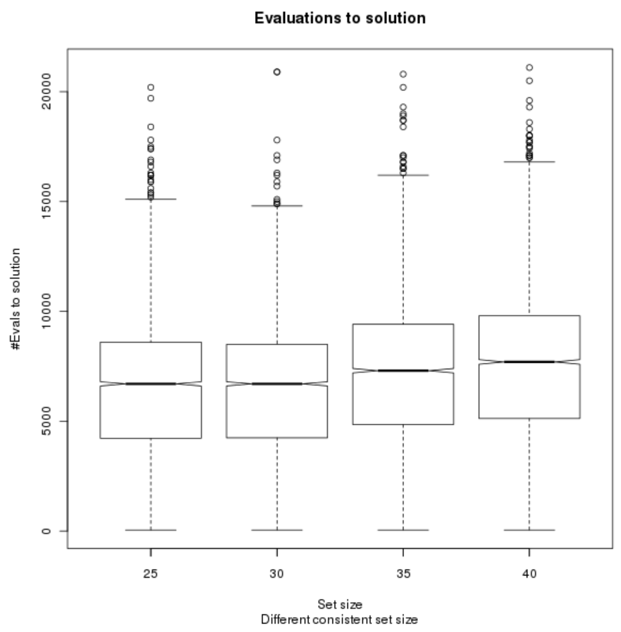

Fig. 1. Boxplots comparing the number of evaluations for different sizes of the consistent set, all other parameters being the same.

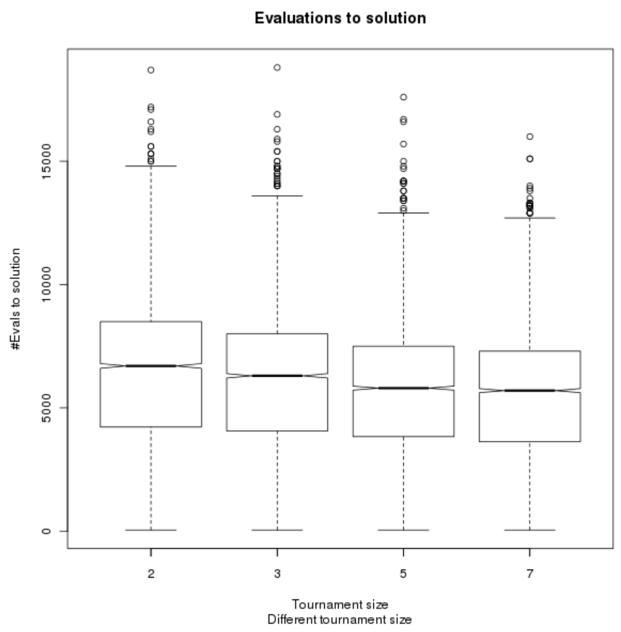

Fig. 2. Boxplots comparing the number of evaluations for different sizes of the tournament used for selection, all other parameters being the same. Crossover priority has been changed to 8; results for crossover 1 were not significantly different from baseline.

space (4096). Our conclusion from this is that this parameter does not have a big influence in the number of moves, but it is convenient to keep it as small as possible to be able to reach solutions fast. The rule of thumb here will be to increase consistent set size with the logarithm of the search space size, or set it to around 10% of the population (which, following evolutionary algorithm lore, must also increase with search space size).

The first evolutionary parameter we will test is the tournament set size. Tournament selection [18] picks T random individuals from the population, and deterministically chooses the best to incorporate it into the pool of individuals that will reproduce. Selective pressure increases with size, because it makes much more unlikely that the worse individuals will be selected. A tournament size of two makes possible that not-so-good individuals make it to the reproductive pool when confronted with others in the same tier; however, with bigger tournament size this probability is diminished. Three sizes besides baseline have been tested: 3, 5 and 7. The results obtained on the number of evaluations are shown in figure 2. In this case, we changed crossover priority to 8, which changed its application rate to 80% (as opposed to 33% before). We performed several experiments with the baseline value, and there was not enough change in either evaluations or moves. It makes sense, since crossover is an exploitative operator and it makes sense to enhance a higher selective pressure with more exploitation.

In this case, the number of moves did have statistically different values for different tournament sizes. $T = 7$ and $T = 5$ were not different, but the former was different from the rest and obtained the best result so far with an average number of moves equal to 5.13, However, this value is only different at an 80% level from the previous best value obtained with the baseline configuration. The number of combinations examined, however, were pairwise statistically different and show an improvement with tournament size, down to an average of 5550 for $T = 7$.

From this we can conclude that the evolutionary algorithm searches efficiently the space via its exploitative mechanisms. It is convenient to tip the exploitation/exploration balance towards the former, improving thus moves and, at the same time, the number of evaluations. Besides, we have performed several experiments changing the value of permutation and mutation rate, as well as population, with results that proved that their influence on the outcome is not too important.

So, eventually, using the default combination shown above, we have performed experiments over a bigger range of the solution space. Even as the best results were obtained for $T = 7$, good enough results were obtained for $T = 2$, which is the default value and the one we use here.

The main result of the experiments above is that the resetting mechanism is never activated; this was one of the main reasons why Evo yielded a high number of evaluations. Besides, the change in other parameters (crossover, mutation and permutation priority) did not have impact either on the number of evaluations or the number of moves. So we settled for the parameters in table II and repeated the experiments for several problem sizes, including one not approached (due to the time it needed) before. The average number of moves is virtually the same, but using a smaller population, as in the [4], although for the first time we have managed to go up to $\ell = 6, \kappa = 9$ obtaining 6.479 ± 0.89 moves.

Berghman does not publish the number of evaluations,

TABLE III

MEAN NUMBER OF EVALUATIONS AND THE STANDARD ERROR OF THE MEAN.

	$\ell = 4$	$\ell = 5$		$\ell = 6$
	$\kappa = 8$	$\kappa = 8$	$\kappa = 9$	$\kappa = 9$
EVO++	6412 ± 3014	14911 ± 6120	25323 ± 9972	46483 ± 17031
EVO	6949 ± 48	19758 ± 556	36485 ± 413	

so in III we compare with our own. New algorithm yields always less evaluations, with difference increasing with problem size. We also set a baseline for the bigger problem size; $\ell = 6, \kappa = 9$. It should be noted that standard error is bigger, but even so differences are statistically significant. The number of evaluations made are a decreasing portion of the search space, up to 8% for the bigger problem but is, still, higher than our own [5]. However, it should be noted that it is relatively easy to obtain solutions using more moves and less evaluations by decreasing population and consistent set size.

V. CONCLUSION AND FUTURE WORK

In this paper we have advanced the research into MasterMind solution strategies initiated seventeen years ago [19] by exploring the space of parameters to find out that one of them, the size of the consistent set that is used to compute the score of consistent combinations, does not have a big influence in the resulting number of moves, so it can be reduced to improve performance. Increasing diversity also boosts speed by reducing the number of repeated combinations in the population and avoiding the hyper mutation phases that plagued previous versions of the algorithm; this also allows to increase selective pressure via the tournament set size which results in a better exploitation of the search space and eventually better results in both fronts: number of guesses and evaluations.

We have also proved that, by itself, the increase in diversity combined with the decrease in size of consistent set and population is robust enough to yield a lower number of evaluations throughout a wide range of sizes, making bigger problems approachable. However, the outcome of using a higher selective pressure has not been checked. This is left as future work, together with the examination of the contributions of diversity, consistent set construction and application of endgames which were introduced previously to the overall result.

ACKNOWLEDGMENTS

This work is supported by grants TIN2011-28627-C04-02, -01 and P08-TIC-03903.

REFERENCES

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

[2] R. Focardi and F. Luccio, "Guessing bank pins by winning a mastermind game," *Theory of Computing Systems*, pp. 1–20, 2011.

[3] 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.

[4] J.-J. Merelo-Guervós, A.-M. Mora, and C. Cotta, "Optimizing worst-case scenario in evolutionary solutions to the MasterMind puzzle," in *IEEE Congress on Evolutionary Computation*. IEEE, 2011, pp. 2669–2676.

[5] 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, http://dx.doi.org/10.1016/j.asoc.2004.09.003.

[6] J. O´Geran, H. Wynn, and A. Zhigljavsky, "Mastermind as a test-bed for search algorithms," *Chance*, vol. 6, pp. 31–37, 1993.

[7] T. P. Runarsson and J. J. Merelo, "Adapting heuristic Mastermind strategies to evolutionary algorithms," in *NICSO'10 Proceedings*, ser. Studies in Computational Intelligence. Springer-Verlag, 2010, pp. 255–267, also available from ArXiV: http://arxiv.org/abs/0912.2415v1.

[8] J. Merelo, A. Mora, T. Runarsson, and C. Cotta, "Assessing efficiency of different evolutionary strategies playing mastermind," in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, August 2010, pp. 38–45.

[9] 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.

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

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

[12] 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.

[13] 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

[14] C. Cotta, J. Merelo Guervós, A. Mora García, and T. Runarsson, "Entropy-driven evolutionary approaches to the Mastermind problem," in *Parallel Problem Solving from Nature PPSN XI*, ser. Lecture Notes in Computer Science, R. Schaefer et al. Eds., vol. 6239. Springer Berlin / Heidelberg, 2010, pp. 421–431. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15871-1_43

[15] J.-J. Merelo and T. P. Runarsson, "Finding better solutions to the Mastermind puzzle using evolutionary algorithms," in *Applications of Evolutionary Computing, Part I*, ser. Lecture Notes in Computer Science, C. di Chio et al., Ed., vol. 6024. Istanbul, Turkey: Springer-Verlag, 7 - 9 Apr. 2010, pp. 120–129.

[16] J.-J. Merelo-Guervós, C. Cotta, and A. Mora, "Improving and Scaling Evolutionary Approaches to the MasterMind Problem," in *EvoApplications (1)*, ser. Lecture Notes in Computer Science, C. D. Chio, S. et al, Eds., vol. 6624. Springer, 2011, pp. 103–112.

[17] J.-J. Merelo-Guervós, P.-A. Castillo, and E. Alba, "`Algorithm::Evolutionary`, a flexible Perl module for evolutionary computation," *Soft Computing*, vol. 14, no. 10, pp. 1091–1109, 2010, accesible at http://sl.ugr.es/000K. [Online]. Available: http://www.springerlink.com/content/8h025g83j0q68270/fulltext.pdf

[18] D. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," *Foundations of genetic algorithms*, vol. 1, pp. 69–93, 1991.

[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. [Online]. Available: http://www.springerlink.com/content/78j7430828t2867g