Improving evolutionary solutions to the game of MasterMind using an entropy-based scoring method

Kuthrapalli, Wollowitz, Hofstadter
Caltech
Pasadena
k,w,h@geekysit.com

Amy Martin
Neurobiology Institute
Los Angeles
amy@globalsolutio.ns

ABSTRACT

Solving the MasterMind puzzle, that is, finding out a hidden combination by using hints that tell you how close some strings are to that one is a combinatorial optimization problem that becomes increasingly difficult with string size and the number of symbols used in it. Since it does not have an exact solution, heuristic methods have been traditionally used to solve it; these methods scored each combination using a heuristic function that depends on comparing all possible solutions with each other. In this paper we first optimize the implementation of previous evolutionary methods used for the game of mastermind, obtaining up to a 40% speed improvement over them. Then we study the behavior of an entropy-based score, which has previously been used but not checked exhaustively and compared with previous solutions. The combination of these two strategies obtain solutions to the game of Mastermind that are competitive, and in some cases beat, the best solutions obtained so far. All data and programs have also been published under an open source license.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; G.1.6 [**Mathematics of Computing**]: NUMERICAL ANALYSIS—*Optimization*

General Terms

Algorithms

Keywords

games, evolutionary algorithms, optimization, puzzles

1. INTRODUCTION AND STATE OF THE ART

Mastermind [42, 24, 1, 36] is a puzzle in which one player A hides a combination of κ symbols and length ℓ that the other player B has to discover by playing combinations coded in the same alphabet and with the same length. The answers from player A to every combination include the number of symbols in the combination that are in the correct position (usually represented in the actual board game as *black* pins) and the number of symbols that have been guessed correctly but are in a different position (*white* pins). Player B then makes a new guess and obtains another answer; this process continues until the hidden one is played by B; B then receives as score the number of guesses made; obviously, higher *score* is worse. A and B then interchange their positions. Eventually, the scores are tallied and the one with the smaller number of guesses made wins. In the actual board game, symbols in the alphabet take the shape of colored pins; that is why we usually talk about *colors* instead of symbols. Historically, the game which is actually played by humans [42] uses $\kappa = 6$ symbols (or colors) and length $\ell = 4$, with *Super Mastermind* versions with up to $\ell = 5$ pegs and $\kappa = 8$ different colors.

Solving this kind of puzzle is an exciting endeavor, because it is a game in which almost any computer strategy can beat a human, which use a very different strategy [26, 21] to play it. But the interesting part is that solutions to Mastermind can be applied easily to other games such as MineSweeper [27] or Hangman since the structure of the game is the same: probing a part of the search space by making a guess and getting a response from whoever holds the secret. But its applications are not reduced to other puzzles or games; solutions to Mastermind have also been used in the following applications:

- Focardi [13] describes a method to break ATMs guessing PINs via making requests to the application programming interface of the so-called HSM (hardware security modules) which are in charge of encryption in several phases of ATM transactions.
- Gagneur and coauthors [15] describe its application in a procedure called *selective phenotyping*. Since it's impossible to find out the phenotype of a single gene, the only way of doing it is via phenotyping of several groups of genes that include the one we are interested in. The procedure used to find this groups of combinations of genes is in fact identical to the solution of the game of Mastermind.
- Mastermind can be used as a testbed for search algorithms, as claims the title of a paper by OǴeran et al. [38]. Strategies developed for Mastermind can be applied to other search algorithms as well.

GECCO'13, July 6-10, 2013, Amsterdam, The Netherlands.

- Querying player B for how close is the played combination to the hidden one and getting an answer is an example of *information leakage* [6], since A reveals some info about its structure which can be eventually used to find it when enough information is available. This technique has been used to find out information about particular DNA codes belonging to a particular person, in what has been named *Mastermind* attack on genomic data [20, 19].

Being able to come up with an effective solution that can be applied to all these problems is a good enough reason, but the generalized Mastermind problem is also interesting from a theoretical point of view, with several studies pointing to the fact that it is NP-Hard problem [41]. The issue of finding bounds to the number of guesses needed is still open, although Doerr and Winzen have lately improved in previously found (and rather loose, [7]) bounds [9, 10]. However, experimental methods are very far from those bounds, obtaining the solution in a much lower number of guesses; they are valuable, however, to give a hint on the scaling of the solution complexity with the number of symbols and the length.

In the absence of exact methods, heuristic solutions such as the one studied in this paper are, for the time being, the best approaches to solve this problem; they are also the ones that present a better scalability and are able to solve the problem to the largest problem size. These solutions fall mainly into two different types: exhaustive (which can only solve versions of the game with a small number of symbols and length due to memory limitations) and approximate, which do not use (or at least try not to do it in the general case) the whole search space. Any of them [38, 14, 4, 39] seek first combinations that match the answers made to B, and thus could possible be the hidden code. These combinations are called *eligible*, *possible* or *consistent*. Those solutions are guaranteed to reduce the search space at least by one, but obviously every one will yield a different response from A, with one being possible optimal (since the set is bound to include the hidden combination). The problem is that knowing which one is the best is, a priori, impossible, and in fact how well any eligible combination will do can only be guessed.

How do we measure, them, how well a certain combination is going to behave? By assigning a *score* to every eligible combination (or every combination the method knows of) which reflects usually how well it will be able to reduce the search space once played, with the rationale that a scoring method that reduces quickly the search space will eventually yield a single eligible combination, which in the case of exhaustive search will be the hidden code and in approximate methods will be at least a good candidate. This score will have to be computed using the only information available to the method: the (possibly partial) set of all eligible combinations. This set, besides possibly (surely, in case of exhaustive search) including the hidden combination, has all the information available to the method on the hidden combination.

So, to compute these scores, every combination is compared in turn with the rest of the combinations in the set; the number of combinations that get every response (there is a limited amount of possible responses, from no blacks/no whites to $\ell-1$ blacks through all combinations of $\ell-p$ blacks and p whites) is noted. Eventually this results in a series of *partitions* in which the set of consistent combinations is divided by its *distance* (in terms of common positions and colors) to every other. So, a *partition* is a subset of the set of eligible combinations. All combinations in the eligible set are then defined by a list of the number of combinations that would obtain every possible answer, and are scored according to this list. However, there will be combinations that will have exactly the same list of values (and thus be in the same *partition*), they will obviously have the same score since they *partition* the set of eligible combinations in the same way. Whatever scoring function we choose, there will be a non-empty set of combinations with the best score; one of the combinations of this set is chosen deterministically (using lexicographical order, for instance) or randomly to be played.

Please note that these are heuristic methods based on expected outcome. On average, they will work as expected, but even combinations with the same score will obtain a different result depending, of course, on which is the actual secret combination. Finding the best one for a particular size, or even a particular set of combinations, is an open issue which leaves room for research such as the one performed in this paper.

These partitions are used to assign a score to every combination; several scores have been proposed:

- *Worst case* was proposed by Knuth [24] as the first rigorous strategy to play Mastermind; combinations are scored according to the size of the biggest partition; the bigger, the worse. The combination to play is extracted from the set of those with the smallest biggest combination. Ties are broken by using the first in lexicographical order.

- *Most parts*, proposed in [25], takes into account only the number of non-zero partitions. This was the option used by authors such as Merelo et al. [29] in the area of evolutionary algorithms. It is fast to compute but it is not clear what is the rationale behind it, other than the fact that unlikely combinations (such as those with a single symbol) usually have low scores, besides, this score is a first approximation of

- *Entropy* which has been used in works such as [37, 5, 8], computes the entropy of partitions, and tries to maximize it. Combinations with a high entropy also have a high amount of information, which is related to the fact that a guess must try and extract the maximum information from the hidden guess.

- *Expected size* used by [4, 22] tries to minimize the expected size; this one is related to the Worst Case mentioned in the first item and follows the same rationale: by playing the combinations that minimize the expected (as opposed to worst, in the case of Knuth) size, the set of eligible combinations will be cut down eventually to one.

However, scores are heuristic, so, for the time being and there is no rigorous way of choosing the best score. Experiments so far show that the best strategies are Most parts and Entropy, with no distinct advantage, for all problem sizes, of any one of them over the others [39]; and even so, the best results published so far use a the Expected Size strategy [4]. In fact, combinations of them are possible, according to

[33], but have not, so far, been proved in problems where a sample of the consistent set is used (as opposed to the whole set); besides, combining the above mentioned scores is, in fact, a score, and nothing bars somebody with coming up with a score different from the set mentioned above with the constraint that it will still use the same information contained in the consistent set (or a subset thereof).. Most strategies, however, concentrate on finding the right size for minimization of the number of turns. The complexity of the solving algorithm, however, increases quadratically with the set size, since it involves comparing every combination with the rest, which implies that the time needed to find the solution and also number of evaluations increase too fast with the search space size, resulting in a bad scaling behavior, so using bounds on the set size has a big impact on the behavior of the algorithm.

If any method wants to improve over the state of the art, it has got to be compared with the paper by Berghman et al. in [4]. They obtained a system that is able to find the solution in an average number of moves that is, for all sizes tested, better than any solution previously published; the number of evaluations was not published, but time was; this last is difficult to compare (since software and hardware environment is different) but it will give at least an order of magnitude (sub-second, seconds, minutes) with which to compare new methods. Besides being the best solutions found so far, they were very fast. As a point of fact, some researchers have tried to break the search space with a high dimension barrier by searching for fast solutions: Khalifa and Yampolsky [23] were able to solve the length 8, 12 colors ($\ell = 8, \kappa = 12$) problem in just 8 seconds, but with an average number of guesses equal to 25 (vs 20.571 seconds and 8.366 average number of guesses). This probes than there are other ways to solve the problem if you are not looking for the best solution, but it is not the venue we will be pursuing in this paper, where we will try to beat, at least from the point of view of guesses played.

We will do so by looking at the problems with this approach: there were many parameters that had to be set for each size, starting with the first guess and the size of the consistent set, as well as population size and other evolutionary algorithm parameters; some parameters are studied, but mainly for the smallest size, while others, like the size of the sample of the eligible set or the initial combination, are just mentioned for sizes bigger than baseline. Changing the eligible set size might lead to better (and possibly slower) solutions. Besides, as mentioned above, Expected Size does not obtain good values in exhaustive search methods, so we will use Entropy to score solutions. And finally, we will try to optimize our method by profiling the application to identify bottlenecks and iron them out of the implementation. The Entropy scoring method has resulted in good results for the smaller sizes [33] and [40] but needs a certain amount of parameter seeking to obtain good results. We will show that our results are competitive and in some cases better than those considered state of the art and published by Berghman et al. [4]. This is a small improvement, but significant, and since we cannot be sure in advance of what is the optimal value, it is in fact enough to prove this improvement even if it is a few percentage points.

The rest of the paper is organized as follows: next we outline the evolutionary method used to play MasterMind in Section 2. Improvements obtained via profiling are shown in Section 3 followed by the results of using Entropy as scoring method in Section 4. Finally, we draw some conclusions in the final Section 5.

2. AN EVOLUTIONARY METHOD FOR PLAYING MASTERMIND

This paper uses the method called, simply, *Evo* [30, 8, 29, 31, 32] (as in EvoMastermind). This method, which was released as open source code at CPAN (the Comprehensive Perl Archive Network, available at a mirror near you) (`http://search.cpan.org/dist/Algorithm-Mastermind/`), is an evolutionary algorithm that has been optimized for speed and number of evaluations. Evolutionary algorithms [18, 35, 12] are a Nature-inspired search and optimization method that, modeling natural evolution and its molecular base, uses a population of solutions encoded into a data structure (usually a string, but currently any data structure is used) to find the optimal one. Candidate solutions are scored according to its closeness to the optimal solution (called *fitness*) and the whole population evolved by discarding solutions with the lowest fitness and making those with the highest fitness reproduce via combination (crossover) and random change (mutation). It is proved that these algorithms are able, given enough time and resources, to find *good enough* and even optimal solutions to several problems in engineering [16] and other fields such as economic forecasting [2].

Evo, which is explained extensively in [32], iterates the evolutionary algorithm until a prefixed amount of consistent combinations has been found; in this paper and following [28] that number has been fixed to ten. It uses Most Parts [25] to score consistent combinations, and the *distance to consistency* for non-consistent ones, so that the fitness directs search towards finding consistent solutions and then towards higher-scoring solutions. If the number of consistent solutions is below the threshold, different from and stays so for several generations (fixed to three), the best combination is played; if no feasible combination is found for a fixed amount of generations (fixed to 50) the population resets. This is a last-ditch solution which leads to a high number of evaluations and, in some cases, to an endless loop or resets which shows that, for that particular configuration, the evolutionary algorithm is unable to find a solution. These problems are in part overcome via endgames [17, 32], which streamline the search for solution via different techniques.

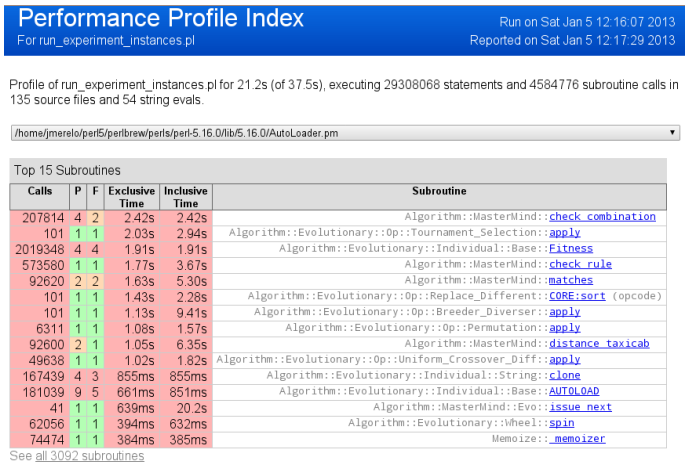
If we keep the consistent set size fixed we still have some parameters to set, mainly two related ones: population and tournament size; this size indicates how many combinations will be compared with each other before selecting a single one to go to the reproduction pool and selective pressure increases with its value; with a value such as eight the probability of a low-fitness combination to be selected is very low, while with a value equal to 2 (the minimum) there is a certain probability that those combinations with low rankings will be selected if they *joust* in the tournament with each other. The relationship among both quantities has been well established by Bäck and coworkers in [3].

In this paper we have made several improvements over the published algorithm. We have profiled the application to improve speed, which has resulted also in a change of the operators used in the experiments. Second, we have studied several parameter configurations to improve at the same time the number of evaluations (which results also in a

speed improvement) and the number of games played. And, finally, we have used a different scoring method over previous papers in a attempt to beat, or at least equal, the already published results.

3. PROFILING FOR PERFORMANCE ENHANCEMENT

One of the first questions one always has to ask about any new algorithm or method proposed is whether its implementation is, in fact, the most efficient possible. Implementation matters, as the authors of [34, 11] have proved. Implementation affects not only same-method speed, but also gives some insight on what is important and what is not in the method, allowing a more deep understanding of it.


Performance Profile Index
For run_experiment_instances.pl
Run on Sat Jan 5 12:16:07 2013
Reported on Sat Jan 5 12:17:29 2013

Profile of run_experiment_instances.pl for 21.2s (of 37.5s), executing 29308068 statements and 4584776 subroutine calls in 135 source files and 54 string evals.

/home/jmerelo/perl5/perlbrew/perls/perl-5.16.0/lib/5.16.0/AutoLoader.pm

Top 15 Subroutines

Calls	P	F	Exclusive Time	Inclusive Time	Subroutine
207814	4	2	2.42s	2.42s	Algorithm::MasterMind::check_combination
101	1	1	2.03s	2.94s	Algorithm::Evolutionary::Op::Tournament_Selection::apply
2019348	4	4	1.91s	1.91s	Algorithm::Evolutionary::Individual::Base::Fitness
573580	1	1	1.77s	3.67s	Algorithm::MasterMind::check_rule
92620	2	2	1.63s	5.30s	Algorithm::MasterMind::matches
101	1	1	1.43s	2.28s	Algorithm::Evolutionary::Op::Replace_Different::CORE:sort (opcode)
101	1	1	1.13s	9.41s	Algorithm::Evolutionary::Op::Breeder_Diverser::apply
6311	1	1	1.08s	1.57s	Algorithm::Evolutionary::Op::Permutation::apply
92600	2	1	1.05s	6.35s	Algorithm::MasterMind::distance_taxicab
49638	1	1	1.02s	1.82s	Algorithm::Evolutionary::Op::Uniform_Crossover_Diff::apply
167439	4	3	855ms	855ms	Algorithm::Evolutionary::Individual::String::clone
181039	9	5	661ms	851ms	Algorithm::Evolutionary::Individual::Base::AUTOLOAD
41	1	1	639ms	20.2s	Algorithm::MasterMind::Evo::issue_next
62056	1	1	394ms	632ms	Algorithm::Evolutionary::Wheel::spin
74474	1	1	384ms	385ms	Memoize::_memoizer

See all 3092 subroutines

Figure 1: Output of `Devel::NYTProf` before profiling. Please note the place where the algorithm spends most of the time, `check_combination` and, in fact, the top ten functions taking most of the time.

Since the code for the methods presented here is written in Perl, we have used `Devel::NYTProf` a source profiler originally developed for the New York times. We tested it over 10 runs of the algorithm with different hidden combinations. Please take into account that, since evolutionary algorithms are stochastic, there is a certain amount of randomness in the result. This profiler outputs a file [1] that can be analyzed and output to a set of web pages. The first page of the report has been captured and is shown in Figure 1. Prior to this profiling another analysis was made, it showed that the *permutation* operator that converted a combination in a permutation was extremely slow (since it had to generate a representation of the combination and proceed to the next permutation until it reached a random number). That is why it this operator was eliminated altogether; however, since it is a major component of the algorithm its effect on performance will be examined in the next Section.

This report shows the usual output in profilers, including the number of calls to a function, exclusive (time spent exclusively in that function) and inclusive (which includes calls to functions from the one listed). We usually have to look at the first item, which a function called `check_combination`. This is the function that compares two combinations, be it the hidden combination with the one played, or obviously more frequently, the combinations in the eligible set among them or the combinations in the population with the already played ones to compute fitness. Even as this function had been optimized in previous versions of the algorithm, it eventually results that it takes 12μs per call, which is a lot considering that it is the core of the algorithm. This makes this function a target for optimization; however, it is almost impossible to go further using Perl. It gets worse, however, since this function is called mainly from the 4th in the list, `check_rule`, which compares a string with already played guesses. That adds a few seconds to the total tally and it gets even worse since this one is called mainly from `distance_taxicab`, 9th in the list, where the program spends almost 70μs per call, being called almost a hundred thousand times during the 10 games.

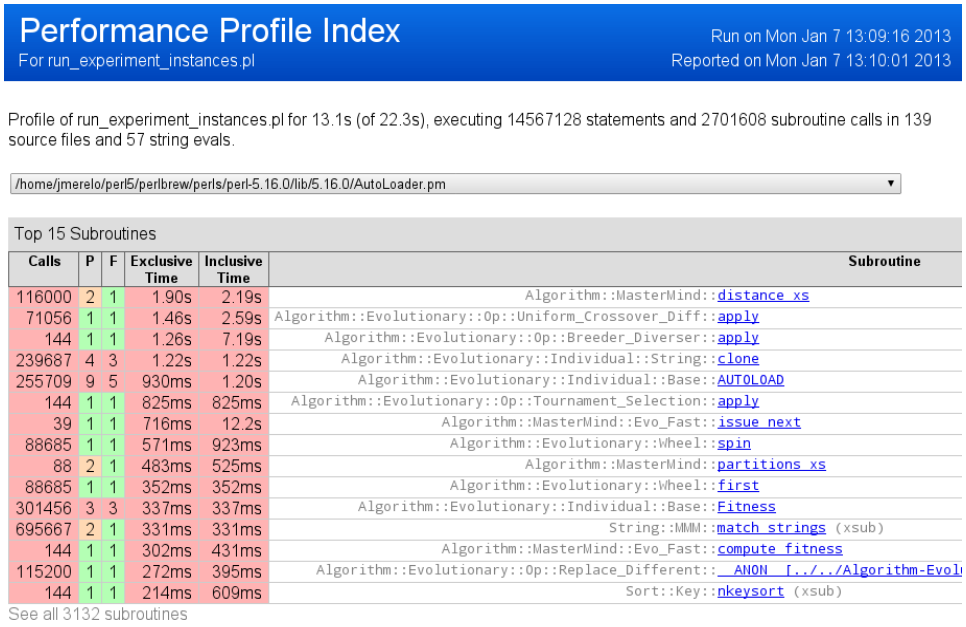
That is good news, also, since we have a clear target for optimization, string comparison. This is the first and most important optimization performed: `check_combination` was programmed in C, and the several levels of indirection used for comparing strings were reduced when possible. The substitute for `distance_taxicab` is called `distance_xs` (XS is the method for including compiled C functions inside Perl programs) and is shown at the top of the new version in Figure 2. This new function takes 19μs per call, less than 1/3 what it did before. Taking into account that previously it took around 1/3 of the total time, the total improvement in performance will amount to around 10%. That is not dramatic, but since usual experiments take up to several hours, it might mean a much better turnaround for performing them.

Besides, the optimization does not stop here. The second function is also an important target for optimization, because it takes a lot of time and is only called 100 times, one for each generation (around 10 generations per game), and a whooping 29 ms. per call. Since this function uses lots of complicated data structures it is not such an easy target for optimization, but still some improvements can be made. Since most of the time is spent comparing fitnesses, it is just a matter of substituting a function call (`Fitness()`) by reading a value (`->'_fitness'`). It is not a good practice (implementation might vary), but we trade that for a good increment in speed (to 5.73 ms. per call); that is basically achieved since that particular line is called almost a million times during the run. That can also be seen in the third function in the list, which is Fitness. By suppressing most calls to that function we inmediately eliminate its 1.91s from the running time (or at least most of them, since you actually have to retrieve the fitness).

Finally, as it happens with most evolutionary algorithms, we find a sorting function in the sixth place, which is called from many places. That is specific from the evolutionary algorithm (not MasterMind) but can be eliminated anyways by using `Sort::Key`, the fastest sort function available in the Perl module repository (called CPAN).

As we proceed further down the list we could find more convenient optimizations. However, the effort needed will not result in a big improvement in speed. Final result for the algorithms eventually used in this paper are shown in Figure 2, which now needs 22.5s to run instead of the previous 37.5, a better than 40% improvement in speed. Please note than, even so, the speed of interpreted languages can be slower than compiled languages. However, we have proved that

[1] All result files are available under request



Performance Profile Index
For run_experiment_instances.pl
Run on Mon Jan 7 13:09:16 2013
Reported on Mon Jan 7 13:10:01 2013

Profile of run_experiment_instances.pl for 13.1s (of 22.3s), executing 14567128 statements and 2701608 subroutine calls in 139 source files and 57 string evals.

/home/jmerelo/perl5/perlbrew/perls/perl-5.16.0/lib/5.16.0/AutoLoader.pm

Top 15 Subroutines

Calls	P	F	Exclusive Time	Inclusive Time	Subroutine
116000	2	1	1.90s	2.19s	Algorithm::MasterMind::distance_xs
71056	1	1	1.46s	2.59s	Algorithm::Evolutionary::Op::Uniform_Crossover_Diff::apply
144	1	1	1.26s	7.19s	Algorithm::Evolutionary::Op::Breeder_Diverser::apply
239687	4	3	1.22s	1.22s	Algorithm::Evolutionary::Individual::String::clone
255709	9	5	930ms	1.20s	Algorithm::Evolutionary::Individual::Base::AUTOLOAD
144	1	1	825ms	825ms	Algorithm::Evolutionary::Op::Tournament_Selection::apply
39	1	1	716ms	12.2s	Algorithm::MasterMind::Evo_Fast::issue_next
88685	1	1	571ms	923ms	Algorithm::Evolutionary::Wheel::spin
88	2	1	483ms	525ms	Algorithm::MasterMind::partitions_xs
88685	1	1	352ms	352ms	Algorithm::Evolutionary::Wheel::first
301456	3	3	337ms	337ms	Algorithm::Evolutionary::Individual::Base::Fitness
695667	2	1	331ms	331ms	String::MMM::match_strings (xsub)
144	1	1	302ms	431ms	Algorithm::MasterMind::Evo_Fast::compute_fitness
115200	1	1	272ms	395ms	Algorithm::Evolutionary::Op::Replace_Different::__ANON__ [../../Algorithm-Evol
144	1	1	214ms	609ms	Sort::Key::nkeysort (xsub)

See all 3132 subroutines

Figure 2: Output of `Devel::NYTProf` after the last optimization profiling. The first one now is the new function created, and all previously time-hogging functions have disappeared.

Table 1: Values for the Evo parameters that obtain the best result. Permutation, crossover and mutation are *priorities*; they are normalized to one to convert them to application rates. In practice, crossover will be applied to 80% and mutation and permutation to 10% of the newly generated combinations each.

Parameter	Value
Crossover	8
Mutation	1
Permutation	1
Replacement rate	0.75

big speed improvements can be made by using a systematic procedure; these speeds can be in many cases much bigger than those possibly obtained by tweaking or improvement of the algorithm itself.

4. USING ENTROPY FOR COMBINATION SCORING IN EVOLUTIONARY MASTERMIND

In this Section we will perform experiments pursuing the two objectives of this paper: obtain a running time that is competitive with the state of the art and obtaining better results than those published in [4]. The new version is called FastEvoEntropy since its scoring method is Entropy and it is faster than previously published versions. Two versions of this last method have been tested: one includes a new permutation operator with a different implementation that chooses randomly a random number of characters in the string that shuffles it; the second one substitutes permutation by two different operators, Shift and Swap; this last one has been traditionally used by other methods. The first one shifts with carry a string left or right with the eliminated character being inserted at the other side; swap is a simple permutation that swaps the place of two characters (previously checking that they are different).

Results for number of guesses (Table 3a), evaluations (Table 3c) and time (Table 3b) are shown. All results indicate the population and consistent set limit size in parenthesis, average and error of the mean. Evo++ results are included for the sake of comparison and were published in [40]; Berghman's results were published in [4] when shown (number of evaluations and, in one case, average time to solution were not published. In all cases 5000 different games were played; Berghman et al. cites 500 games in their paper.

Just two sizes were used for this comparison; they pose a difficult enough problem for being challenging and, besides, these are the sizes for which results have been published and can thus be compared against this new method. The common parameters for all runs are shown in Table 1. The rows in each table is correlative, that is, they correspond to the same experiment. Time measures were not taken previously so this new Evo version cannot be compared with the previous one (but the previous Section has sufficiently proved the former was slower).

The tables above show several results. First, the canonical implementation of the permutation, albeit slow, is better than the new one, results for Evo++ (EN) are (significantly using Wilcox T-Test) better than any result published in this paper; this consistently for the two sizes studied. This issue will have to be pursued further, but it was not the main objective of this paper, which was to establish a new state of the art in MasterMind results.

The results shown in Table 3a show that, effectively, this new algorithm is better at solving the game than Berghman's (incidentally, [40] too). The difference is small but significant using Wilcoxon t-test, and that is so for all methods that use 256 as consistent set limit size; for $\ell = 6, \kappa = 9$, even

Table 2: Comparison among approaches in this paper: Evo and Berghman et al. [?]. Previously obtained results published in [40] are also included for the sake of comparison. Those labeled EN used Entropy as scoring method, while MP use Most Parts.

	$\ell = 5, \kappa = 8$	$\ell = 6, \kappa = 9$
Berghman et al.	5.618	6.475
Evo++ (EN)	(1000,200) 5.555 ± 0.011	(2000,200) 6.373 ± 0.011
Evo++ (MP)	(1000,80) 5.602 ± 0.012	(2000,200) 6.436 ± 0.012
FastEvoEntropy	(768,32) 5.6148 ± 0.011	(1024,32) 6.433 ± 0.012
	(768,256) 5.59 ± 0.012	(1024,256) 6.44 ± 0.012
	(1024,256) 5.577 ± 0.011	(2048,256) 6.42 ± 0.012
FastEvoEntropy S+SH	(800, 256) 5.5762 ± 0.012	(1024,256) 6.43 ± 0.12

(a) Mean number of guesses with standard deviation; the quantities in parentheses indicate population and consistent set size (in the case of Evo++).

	$\ell = 5, \kappa = 8$	$\ell = 6, \kappa = 9$
Berghman et al.		1.284
FastEvoEntropy	(768,32) 0.709 ± 0.004	(1024,32) 2.31 ± 0.93
	(768,256) 3.03 ± 0.02	(1024,256) 5.42 ± 0.034
	(1024,256) 3.26 ± 0.02	(2048,256) 6.88 ± 0.04
FastEvoEntropy S+SH	(800, 256) 2.93 ± 0.02	(1024,256) 5.41 ± 0.06

(b) Mean time to solution with error of the mean; differences are always significant.

	$\ell = 5, \kappa = 8$	$\ell = 6, \kappa = 9$
Evo++ (EN)	26098 ± 144	67521 ± 384
Evo++ (MP)	20120 ± 114	68860 ± 406
FastEvoEntropy	(768,32) 12679 ± 73	(1024,32) 38306 ± 14098
	(768,256) 25167 ± 141	(1024,256) 49492 ± 288
	(1024,256) 29279 ± 161	(2048,256) 76277 ± 443
FastEvoEntropy S+SH	(800, 256) 25488 ± 144	(1024,256) 49663 ± 779

(c) Mean number of evaluations with error of the mean; differences are always significant.

when the smaller size (consistent set size CS=32) is used. This new version of the method, as is usual, achieves the best results with a sizable population and CS=256. However, increasing more the population (to 2048) does not result in a noticeable (in fact, not significantly better) decrease in the number of games needed to win. We can also notice in the results that the new permutation operator is not significantly different from the new swap+shift (S+SH) operators; as already pointed out, both are worse than the canonical permutation operator. On the other hand, the main conclusion drawn from the time-to-solution table (3b is that, in fact, it is difficult to obtain a solution faster than Berghman's et al. did. You can obtain a better solution, but not a faster solution. We should have to improve the faster solution around 50% to obtain it as fast as Berghman's. However even if it is a bit slower, it is better. By reducing even further the CS size and population it it's not impossible to achieve that target. That, for the moment, is left as future work.

The third table with the average number of evaluations show that these new methods, even achieving worse results, do so using a comparable number of evaluations (for $\ell = 5$) or considerably less (for $\ell = 6$). This is due to the change in the operators used, and also shows that it is possible that exploration of search space changes considerably with them and they are unable to find the best solutions so efficiently. There is also a rough correlation between the number of evaluations and the time needed to find the solution and some hope in the fact that, despite the search space for $\ell = 6, \kappa = 9$ being roughly 30 times larger than $\ell = 5, \kappa = 8$, since we will be able to pursue bigger problem spaces with a probably logarithmic increase in the time.

Finally using S+SH is a bit faster than using (the new) permutation operator, as shown in the last row of results. In any case, results are good enough so the new operators introduced in this work have proved its worth as alternatives to permutation.

5. CONCLUSIONS

In this paper we have tried to establish a new state of the art with a method that is able to obtain better solutions to the game of MasterMind than those published and do it in less time. However, we have not managed to achieve both objectives: our results are better than those published but considerably slower. There is some room for improvement in both aspects, but that is left as future work. The achieved results, however, show a big improvement in the number of evaluations needed and the time needed to do them, thanks to the optimization done after profiling the method. This profiling has led to a change in the algorithm by eliminating a slow operator which, however, has resulted in worse results than previously obtained. As indicated above, this is a result that will have to be pursued further.

The main result, thus, in this paper, which is obtaining a better player at Mastermind than the state of the art, has

been achieved mainly by using Entropy as a scoring method. Previously [40] it proved to be better than most parts and both are known to be better than Worst Case [24]. However, so far Expected Case (used by Berghman) obtained the best results. This is no longer so, and it will have to be explored whether entropy effectively continues being a good scoring method for eligible combinations when the size of the search space increases.

It should be noted also that all results, configuration files and data are available under an open source licence (address hidden for double blind revision).

Several possible lines of work have been pointed out above. Profiling is a methodology that is obviously valid for a particular problem size. When size increases new functions may start to be the bottleneck, so size-specific optimizations are also possible. Another option is to code in C other critical parts of the program, but none so simply have been found, so other speedup methods (including complete factorization) should be sought.

An interesting option that would be worth exploring is using multi-threaded applications to simulate parallel execution within a single computer. This will speed up the application, but it will have an important impact on the algorithm so it will have to be examined carefully looking at different alternatives and ways of interaction of the different threads. We think that it is going to be difficult to obtain much improvement over the average number of games obtained here (or, for that matter, in other papers pursuing this research issue), even using exhaustive search when a sufficiently powerful machine is found, so any improvement on the algorithm will have to be done on the number of evaluations needed or the time needed to reach them.

6. ACKNOWLEDGMENTS

Hidden for double-blind revision.

7. REFERENCES

[1] Marc Aldebert and Risto Widenius. Mastermind. Newsgroup `comp.ai.games`, archived at `https://groups.google.com/d/topic/comp.ai.games/FVOEABY4gbQ/discussion`, 1995.

[2] E. Alfaro-Cid, P. A. Castillo, A. Esparcia, K. Sharman, J.J. Merelo, A. Prieto, A.M. Mora, and J.L.J. Laredo. Comparing multiobjective evolutionary ensembles for minimizing type I and II errors for bankruptcy prediction. In *WCCI 2008 Proceedings*, pages 2907–2913. IEEE Press, 2008.

[3] Thomas Bāck. Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In *In Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 57–62. IEEE Press, 1994.

[4] L. Berghman, D. Goossens, and R. Leus. Efficient solutions for Mastermind using genetic algorithms. *Computers and Operations Research*, 36(6):1880–1885, 2009.

[5] Azer Bestavros and Ahmed Belal. Mastermind, a game of diagnosis strategies. *Bulletin of the Faculty of Engineering, Alexandria University*, December 1986.

[6] Y. Chen and D. Evans. Auditing information leakage for distance metrics. In *Privacy, security, risk and trust (PASSAT), 2011 ieee third international conference on and 2011 ieee third international conference on social computing (socialcom)*, pages 1131–1140. IEEE, 2011.

[7] V. Chvatal. Mastermind. *Combinatorica*, 3(3-4):325–329, 1983.

[8] Carlos Cotta, Juan J. Merelo Guervós, Antonio Mora García, and Thomas Runarsson. Entropy-driven evolutionary approaches to the Mastermind problem. In Robert Schaefer, Carlos Cotta, Joanna Kolodziej, and Günter Rudolph, editors, *Parallel Problem Solving from Nature PPSN XI*, volume 6239 of *Lecture Notes in Computer Science*, pages 421–431. Springer Berlin / Heidelberg, 2010.

[9] B. Doerr, R. Spöhel, H. Thomas, and C. Winzen. Playing mastermind with many colors. *arXiv preprint arXiv:1207.0773*, 2012.

[10] B. Doerr and C. Winzen. Ranking-based black-box complexity. *Algorithmica*, pages 1–39, 2012. Article in Press.

[11] J.A. Durlak and E.P. DuPre. Implementation matters: A review of research on the influence of implementation on program outcomes and the factors affecting implementation. *American journal of community psychology*, 41(3):327–350, 2008.

[12] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

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

[14] J. Francis. Strategies for playing MOO, or "Bulls and Cows". `http://www.jfwaf.com/Bulls%20and%20Cows.pdf`.

[15] J. Gagneur, M.C. Elze, and A. Tresch. Selective phenotyping, entropy reduction, and the mastermind game. *BMC bioinformatics*, 12(1):406, 2011. `http://www.biomedcentral.com/1471-2105/12/406`.

[16] P. García-Sánchez, J.J. Merelo, J.P. Sevilla, J.L.J. Laredo, A.M. Mora, and P.A. Castillo. Automatic generation of XSLT stylesheets using evolutionary algorithms. In *Genetic and Evolutionary Computation Conference, GECCO2008*, pages 1701–1702, New York, NY, USA, 2008. ACM Press.

[17] W. Goddard. A computer/human Mastermind player using grids. *South African Computer Journal*, 30:3–8, 2003.

[18] David E. Goldberg. Zen and the art of genetic algorithms. In J. David Schaffer, editor, *ICGA*, pages 80–85. Morgan Kaufmann, 1989.

[19] M. T. Goodrich. Learning character strings via mastermind queries, with a case study involving mtdna. *IEEE Transactions on Information Theory*, 58(11):6726–6736, 2012.

[20] M.T. Goodrich. The mastermind attack on genomic data. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 204–218. IEEE, 2009.

[21] S. Hubalovsky. Modeling and computer simulation of real process–solution of mastermind board game. *International Journal of Mathematics and Computers in Simulation*, pages 107–118, 2012.

[22] R. W. Irving. Towards an optimum Mastermind

strategy. *Journal of Recreational Mathematics*, 11(2):81–87, 1978-79.

[23] A.B. Khalifa and R.V. Yampolskiy. GA with Wisdom of Artificial Crowds for Solving Mastermind Satisfiability Problem. *International Journal of Intelligent Games & Simulation*, 6(2):6, 2011.

[24] Donald E. Knuth. The computer as Master Mind. *J. Recreational Mathematics*, 9(1):1–6, 1976-77.

[25] B. Kooi. Yet another Mastermind strategy. *ICGA Journal*, 28(1):13–20, 2005.

[26] P.R. Laughlin, R. Lange, and J. Adamopoulos. Selection strategies for "mastermind" problems. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 8(5):475, 1982.

[27] M. Legendre, K. Hollard, O. Buffet, A. Dutech, et al. MineSweeper: Where to probe? Technical Report RR-8041, INRIA, 2012. `http://hal.inria.fr/hal-00723550`.

[28] J. J. Merelo, A. M. Mora, and C. Cotta. Finding fast solutions to the game of Mastermind. In Carlos Cotta Antonio Fernández-Leiva and Raúl Lara, editors, *GAME-ON 2012, 13th International Conference on Intelligent Games and Simulation*, pages 25–28. Eurosis, 2012.

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

[30] Juan-Julián Merelo and Thomas P. Runarsson. Finding better solutions to the Mastermind puzzle using evolutionary algorithms. In Cecilia di Chio et al., editor, *Applications of Evolutionary Computing, Part I*, volume 6024 of *Lecture Notes in Computer Science*, pages 120–129, Istanbul, Turkey, 7 - 9 April 2010. Springer-Verlag.

[31] Juan-J. Merelo-Guervós, Carlos Cotta, and Antonio Mora. Improving and Scaling Evolutionary Approaches to the MasterMind Problem. In Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna Esparcia-Alcázar, Juan Julián Merelo Guervós, Ferrante Neri, Mike Preuss, Hendrik Richter, Julian Togelius, and Georgios N. Yannakakis, editors, *EvoApplications (1)*, volume 6624 of *Lecture Notes in Computer Science*, pages 103–112. Springer, 2011.

[32] Juan-J. Merelo-Guervós, Antonio-Miguel Mora, and Carlos Cotta. Optimizing worst-case scenario in evolutionary solutions to the MasterMind puzzle. In *IEEE Congress on Evolutionary Computation*, pages 2669–2676. IEEE, 2011.

[33] Juan-Julián Merelo-Guervós, Antonio-Miguel Mora, Carlos Cotta, and Thomas-Philip Runarsson. An experimental study of exhaustive solutions for the mastermind puzzle. *CoRR*, abs/1207.1315, 2012.

[34] Juan-Julián Merelo-Guervós, Gustavo Romero, Maribel García-Arenas, Pedro A. Castillo, Antonio-Miguel Mora, and Juan-Luís Jiménez-Laredo. Implementation matters: Programming best practices for evolutionary algorithms. In Joan Cabestany, Ignacio Rojas, and Gonzalo Joya Caparrós, editors, *IWANN (2)*, volume 6692 of *Lecture Notes in Computer Science*, pages 333–340. Springer, 2011.

[35] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution programs*. Springer-Verlag, 2nd edition, 1994.

[36] G. Montgomery. Mastermind: Improving the search. *AI Expert*, 7(4):40–47, 1992.

[37] E. Neuwirth. Some strategies for Mastermind. *Zeitschrift fur Operations Research. Serie B*, 26(8):B257–B278, 1982.

[38] JH O´Geran, HP Wynn, and AA Zhigljavsky. Mastermind as a test-bed for search algorithms. *Chance*, 6:31–37, 1993.

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

[40] A. U. Thor-Hidden. Title hidden. In *Accepted, to be published in EvoApps proceedings 2013*, page 10pp, 2013.

[41] Giovanni Viglietta. Hardness of mastermind. *CoRR*, abs/1111.6922, 2011.

[42] Wikipedia. Mastermind (board game) — Wikipedia, The Free Encyclopedia, 2009.