Finding Better Solutions to the Mastermind Puzzle Using Evolutionary Algorithms

Juan J. Merelo-Guervós[2] and Thomas Philip Runarsson[1]

[1] School of Engineering and Natural Sciences, University of Iceland
tpr@hi.is
[2] Department of Architecture and Computer Technology, ETSIIT, University of Granada, Spain
jmerelo@geneura.ugr.es

Abstract. The art of solving the Mastermind puzzle was initiated by Donald Knuth and is already more than thirty years old; despite that, it still receives much attention in operational research and computer games journals, not to mention the nature-inspired stochastic algorithm literature. In this paper we revisit the application of evolutionary algorithms to solving it and trying some recently-found results to an evolutionary algorithm. The most parts heuristic is used to select guesses found by the evolutionary algorithms in an attempt to find solutions that are closer to those found by exhaustive search algorithms, but at the same time, possibly have better scaling properties when the size of the puzzle increases.

1 Introduction

Mastermind in its current version is a board game that was introduced by the telecommunications expert Mordecai Merowitz [12] and sold to the company Invicta Plastics, who renamed it to its actual name; in fact, Mastermind is a version of a traditional puzzle called *bulls and cows* that dates back to the Middle Ages. In any case, Mastermind is 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 $\kappa = 6$, 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.
- The codebreaker then tries to guess this secret code by producing a combination.
- 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 until the secret code is found.

For instance, a game could go like this: The codemaker sets the secret code *ABBC*, and the rest of the game is shown in Table 1.

Different variations of the game include giving information on which position has been guessed correctly, avoiding repeated symbols in the secret combination (*bulls and*

C. Di Chio et al. (Eds.): EvoApplications 2010, Part I, LNCS 6024, pp. 121–130, 2010.

Table 1. Progress in a Mastermind game that tries to guess the secret combination *ABBC*. The player here is not particularly clueful, playing a fourth combination that is not *consistent* with the first one, not coinciding, for instance, in two positions and one color (corresponding to the 2 black/1 white response given by the codemaker) with it. The rest of the combinations would effectively be *consistent*; for instance, ABBD coincides in two places (first A and third B) and one "color" (B) with the first, and two positions (A and B in the first and second position) with the second combination.

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

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).

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 the 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, since the codebreaker is not really matching his skills against the codemaker, but facing a problem that must be solved with the help of hints, the implication being 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 search problem with constraints.

What makes this problem interesting is its relation to other, generally called *oracle* problems such as circuit and program testing, differential cryptanalysis and other puzzle games (these similarities were reviewed in our previous paper [8]) is the fact that it has been proved to be NP-complete [11,5] and that there are several open issues, namely, what is the lowest average number of guesses you can achieve, 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 first issue.

This NP completeness implies that it is difficult to find algorithms that solve the problem in a reasonable amount of time, and that is why initial papers [8,2] introduced stochastic evolutionary and simulated annealing algorithms that solved the Mastermind puzzle in the general case, finding solutions in a reasonable amount of time that scaled roughly logarithmically with problem size. The strategy followed to play the game was optimal in the sense that is was guaranteed to find a solution after a finite number of combinations; however, there was no additional selection on the combination played other than the fact that it was consistent with the responses given so far.

In this paper, after reviewing how the state of the art in solving this puzzle has evolved in the last few years and showing how different evolutionary algorithms fare against each other, we try to apply some kind of selection to consistent solutions found by the evolutionary algorithm, so that the combination played is most likely to shorten the search time. In this we follow a path initiated by Berghman et al. [1], but taking into account our own results [10] on the number of combinations that are needed to obtain a reasonable result. The main idea driving this line of research is to try and obtain results that are comparable with the exhaustive search methods, but without the need to *see* all possible combinations at the same time. This will allow to create a method that can work, in principle, for any problem size, and scale reasonably unlike exhaustive search whose scaling is exponential in time *and* in memory.

The rest of the paper is organized as follows: next we establish terminology and examine the state of the art; then the new evolutionary algorithms we introduce in this paper are presented in section 3 and the experimental results in 4; finally, conclusions are drawn in the closing section 5.

2 State of the Art

Before presenting the state of the art, a few definitions are needed. We will use the term *response* for the return code of 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} iff

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

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

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

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.

One of the earliest strategies, by Knuth [6], is perhaps the most intuitive for Mastermind. In this strategy the player selects the guess that reduces the number of remaining consistent guesses and the opponent the return code leading to the maximum number of guesses. Using a complete minimax search Knuth shows that a maximum of 5 guesses are needed to solve the game using this strategy. This type of strategy is still the most widely used today: most algorithms for Mastermind start by searching for a consistent combination to play.

In some cases once a single consistent guess is found it is immediately played, in which case the object is to find a consistent guess as fast as possible. For example, in [8] 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 a consistent combinations

eventually produces a number of guesses that uniquely determine the code. However, the maximum, and average, number of combinations needed is usually high. Hence, some bias must be introduced in the way 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 alternative to discovering a single consistent guess is to collect a set of consistent guesses and select among them the best alternative. For this a number of heuristics have been developed over the years. Typically these heuristics require all consistent guesses to be first found. The algorithms then use some kind of search over the space of consistent combinations, so that only the guess that 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. To each combination corresponds a partition of the rest of 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: 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), etc. Some partitions may also be empty, or contain a single element (4b, 0w will contain just AABB, obviously). For a more exhaustive explanation see [7]. 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.

The path leading to the most successful strategies to date include using the *worst case*, *expected case*, *entropy* [9,3] and *most parts* [7] strategies. The *entropy* strategy selects the guess with the highest entropy. The entropy is 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 [9]. The *worst case* is a one-ply version of Knuth's approach, but Irving [4] suggested using the *expected case* rather than the worst case. Kooi [7] 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 the number of them. Computing the number of them is, however, faster than determining their size. For this reason the *most parts* strategy has a computational advantage.

Following a tip in one of the papers that tackled Mastermind, recently Berghman et al. [1] proposed an evolutionary algorithm which finds a number of consistent guesses and then uses a strategy to select which one of these should be played. The strategy they apply is similar 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 played against every different secret. The return codes are then used to compute the size of the set of remaining consistent guesses in the set. 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.

The heuristic strategies described above use some form 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 random, but no more. However, it has been shown [10] 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 sufficient in order to find solutions that are statistically indistinguishable from the best solutions found. This is the path explored in this paper.

3 Description of the Method

Essentially, the method described is an hybrid between an evolutionary algorithm and the exhaustive partition-based methods described above. Instead of using exhaustive search to find a set of consistent combinations, which then are compared to see the way they partition that set, we use evolutionary algorithms to find a set of consistent combinations and compute then the partitions they yield. The size of the set is fixed according to our previous results [10], 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. By using this approach we are assuming that the set of consistent combinations found by the evolutionary algorithm is random; since evolutionary search introduces a bias in search space sampling, this need not be the case.

The evolutionary algorithms used are similar to the Estimation of Distribution Algorithm (EDA) that was also shown in our previous paper. The fitness function is similar to the one proposed by Berghman et al. [1], except for the term proportional to the number of positions; that is,

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

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. 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. In the previous paper [10] a *local entropy* was used to bias search, but in this paper we found this was harmful, so we have left it out. However, the first combination was fixed to *ACBA*, same as before.

While in [10] the EDA presented proceeded until a consistent solution was found, and then played the one with the highest local entropy (since they were ranked by local entropy), in this work we will proceed in the following way:

- 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 15 generations, reset the population substituting it by a random one. Again, this was a number considered high enough to imply stagnation of search and at the same time give the algorithm a chance to find solutions.

If any of the conditions above hold (either a big enough set is found, or the algorithm has not found any more solutions for 3 generations), a set of consistent solutions is obtained; for each solution in this set, the way they partition the rest of the set is computed. Then, the solution which leads to the maximum number of non-zero partitions (*most parts*) is computed, in a similar way as proposed by [7] and differently from Berghman [1], which use some form of expected size strategy. As shown by [10], *most parts* and the entropy method are the best, however, *most parts* is faster to compute. We wanted also to find out whether this strategy obtained better solutions than the plain vanilla evolutionary algorithm.

Several evolutionary algorithms were also tested. A rank-based steady state algorithm obtained generally good results, but there were occasions (around one in several thousands) where it could not find it even with restarts, so it was eliminated. Finally we settled for an EDA and a canonical genetic algorithm. In both cases letter strings where used for representing the solutions; that means that when mutation is used, a string mutation (change a letter for another in the alphabet) is used. Source code and all parameter files used for these experiments are available as GPL'ed code from `http://sl.ugr.es/alg_mm/`. The parameters used for both algorithms are shown in table 2. No exhaustive tweaking of them has been made.

Results obtained with these algorithms are shown in the next section.

Table 2. Parameter values in the evolutionary algorithms used in this paper

Parameter	EDA	CGA
Operators	N/A	Mutation, 1-point crossover
Population	300	256
Replacement Rate	50%	40%

4 Experimental Results

In the same way as was done by ourselves [10] and Berghman [1], we have tested the algorithms on every possible combination, but instead of doing it 3 times, we have run the algorithm ten times for each combination from $AAAA$ to $FFFF$ to obtain a higher statistical certainty. The whole 10K runs last several hours on an Intel quad-core computer. The number of combinations obtained by both methods are shown in table 3 and compared with others, results for *most parts* and *entropy* are taken from our former paper, and Berghman's from his paper.

Table 3. Comparison of results obtained for different Mastermind-solving algorithms. Entropy and *Most parts* are taken from [10]; μ is the maximum size of the set of consistent solutions used to find the solution. The EDA algorithm shown is the one that uses local entropy in fitness, as explained in the text. Berghman uses different sizes, and we include only the smallest and biggest; when the value has not been published, an empty slot is shown. The last two lines show the results obtained by the algorithms presented in this paper.

Strategy	*min*	*mean*	*median*	*max*	*st.dev.*	*max guesses*
Entropy $\mu = \infty$	4.383	4.408	4.408	4.424	0.012	6
Most parts $\mu = \infty$	4.383	4.410	4.412	4.430	0.013	7
Entropy $\mu = 20$	4.438	4.468	4.476	4.483	0.016	7
Most parts $\mu = 20$	4.429	4.454	4.454	4.477	0.016	7
EDA $\mu = 1$	4.524	4.571	4.580	4.600	0.026	7
Berghman $\mu = 30$	-	4.430	-	-	-	7
Berghman $\mu = 60$	-	4.390	-	-	-	-
CGA $\mu = 20$	4.402	4.434	4.433	4.471	0.018	7
EDA $\mu = 20$	4.419	4.445	4.448	4.467	0.017	7

The results obtained by these new methods, CGA and EDA, are competitive with the exhaustive search methods with $\mu = 20$, that is, they are all statistically equivalent. All statistical tests were performed using the Wilcoxon rank sum test for equal medians at a 0:05 significance level. When $\mu = 30$ they become also statistically equivalent to using the whole set of consistent guesses, i.e. $\mu = \infty$, resulting in a method comparative to exhaustive search. The results also show that sampling of the combination set, as performed by the evolutionary algorithm, is actually random, since it behaves in the same way as shown by the random sampling algorithm. In fact, this should be expected, at least for EDA, since the results obtained by a plain-vanilla, play-as-find the solution go method, were statistically indistinguishable from a random algorithm. Besides, EDA and CGA need roughly the same number of evaluations to find those solutions, although, on average, EDA is better, since the maximum number of evaluations needed by the CGA is six times bigger than the worst result obtained by the EDA.

On the other hand, and all things being equal[1] the results appear very similar to those obtained by Berghman with the smallest set size μ. We cannot say if they are statistically indistinguishable, but it is quite likely that they actually are. We say this, since the *expected size* strategy is worse than the *most parts* and *entropy* methods [10]. The running time, for an interpreted (not compiled) code which has not been optimized in any way, is around one second per game, which is comparable to those published by them[2].

How do these algorithms actually work to obtain these results? Since the initial pool has a good amount of combinations, they usually include a big enough pool of consistent combinations, sufficient for the second play and on occasions also the third. The

[1] That is, assuming the average has been computed in the same way as we did.

[2] But please note that machines are quite different too. That only means that running times are on the same order of magnitude.

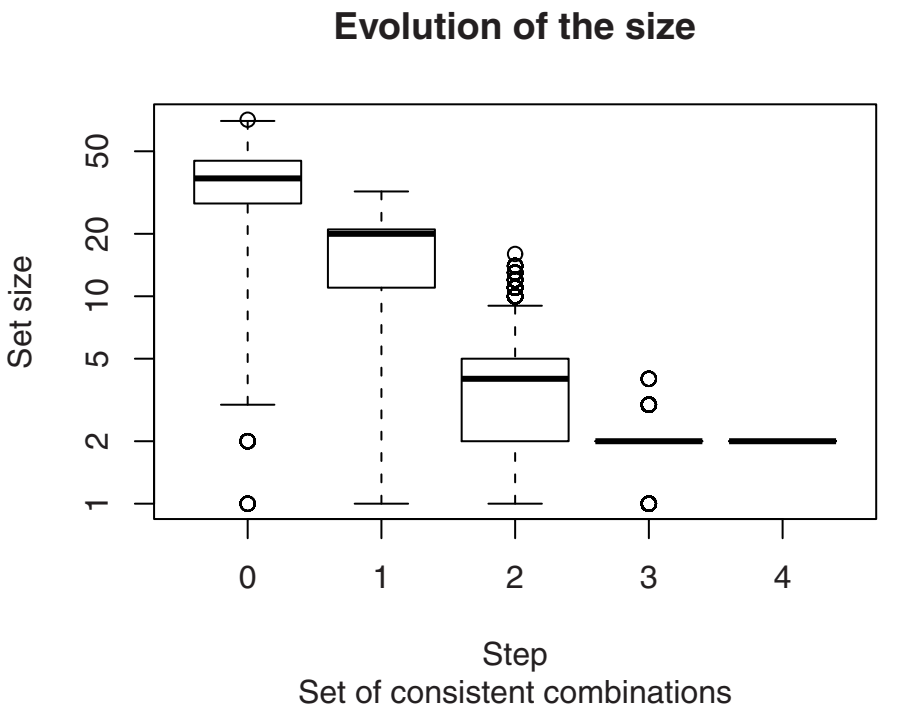

Fig. 1. Boxplot of the evolution of set size with the game, horizontal line in the box indicates the average, the box upper and lower side the quartiles, and dots outside it outliers; the x axis (labeled *Step* represents the number of combinations played after the first one, with the y axes representing the set size; thus, at $x = 0$ the set size after the first combination has been played has been represented. The y axis has been scaled logarithmically

remaining draws, notably including the secret code, must be found by the evolutionary algorithm. In fact, the average number of elements in the set of consistent solutions evolves with combinations played as shown in figure 1.

In that figure, the first set has a size close to 50, and the second is pegged at approximately 20, indicating that the evolutionary algorithm has probably intervened to find a few consistent solutions. For the rest of the combinations it is quite unusual to find a set of that size, at least with the constraints we have set; eventually the 4th and 5th draws are drawn from a single set. The figure also shows an idea of how set sizes should scale for bigger sizes of the Mastermind problem; however, this hunch should be proved statistically.

5 Conclusion and Future Work

In this paper we have introduced a new evolutionary algorithm that tries to find the solution to the game of Mastermind, in as few games as possible, by finding a set of

possible solutions and ranking them according to the most part heuristic. This approach is as good as other systems that use exhaustive heuristic search, comparable to other methods that represent the state of the art in evolutionary solution of this game, and does not add too much overhead in terms of computing or memory, using time and memory roughly in the same order of magnitude as the simple evolutionary algorithm.

This also proves that exhaustive heuristic search strategies can be successfully translated to evolutionary algorithms, provided that the size of the sample is tested in advance. This might prove a bit more difficult as the size increases, but a systematic exploration of the problem in different sizes might give us a clue about this. Our hunch right now is that it will scale as the logarithm of the problem size, which would be a boon for using evolutionary algorithms for bigger sizes.

This is probably an avenue we will pursue in the future: compute the size set of consistent guesses needed to find the correct solution in a minimal number of draws, and apply it to evolutionary algorithms of different shapes. One of these could be to find a way of including the score of a consistent solution in the evolutionary algorithm by using this as a fitness, and taking into account its consistency only as a constraint, not as the actual fitness. This will prove increasingly necessary as the size of the consistent solution sets increases with problem space size.

The space of EAs has not been explored exhaustively in this paper either. Although it is rather clear that different parameter settings will not have an influence on the quality of play, it remains to be seen how they will impinge on the number of evaluations needed to find the consistent combinations. Different combinations of operators and operator rates will have to be tested. And as the complexity of the system goes up with the combination length and number of colors, distributed approaches will have to be tested, which could have an influence not only on the running time, but also on the number of evaluations needed to find it. These will be explored in the future.

Acknowledgements

This paper has been funded in part by the Spanish MICYT projects NoHNES (Spanish Ministerio de Educación y Ciencia - TIN2007-68083) and TIN2008-06491-C04-01 and the Junta de Andalucía P06-TIC-02025 and P07-TIC-03044.

References

1. Berghman, L., Goossens, D., Leus, R.: Efficient solutions for Mastermind using genetic algorithms. Computers and Operations Research 36(6), 1880–1885 (2009), http://www.scopus.com/inward/record.url?eid=2-s2.0-56549123376
2. Bernier, J.L., Herráiz, C.I., Merelo-Guervós, J.J., Olmeda, S., Prieto, A.: Solving mastermind using GAs and simulated annealing: a case of dynamic constraint optimization. In: Ebeling, W., Rechenberg, I., Voigt, H.-M., Schwefel, H.-P. (eds.) PPSN 1996. LNCS, vol. 1141, pp. 554–563. Springer, Heidelberg (1996), http://citeseer.nj.nec.com/context/1245314/0

3. Bestavros, A., Belal, A.: Mastermind, a game of diagnosis strategies. Bulletin of the Faculty of Engineering, Alexandria University (December 1986), http://citeseer.ist.psu.edu/bestavros86mastermind.html, available from http://www.cs.bu.edu/fac/best/res/papers/alybull86.ps
4. Irving, R.W.: Towards an optimum mastermind strategy. Journal of Recreational Mathematics 11(2), 81–87 (1978–1979)
5. Kendall, G., Parkes, A., Spoerer, K.: A survey of NP-complete puzzles. ICGA Journal 31(1), 13–34 (2008), http://www.scopus.com/inward/record.url?eid=2-s2.0-42949163946
6. Knuth, D.E.: The computer as Master Mind. J. Recreational Mathematics 9(1), 1–6 (1976–1977)
7. Kooi, B.: Yet another Mastermind strategy. ICGA Journal 28(1), 13–20 (2005), http://www.scopus.com/inward/record.url?eid=2-s2.0-33646756877
8. Merelo-Guervós, J.J., Castillo, P., Rivas, V.: Finding a needle in a haystack using hints and evolutionary computation: the case of evolutionary MasterMind. Applied Soft Computing 6(2), 170–179 (2006), http://www.sciencedirect.com/science/article/B6W86-4FH0D6P-1/2/40a99afa8e9c7734baae340abecc113a, http://dx.doi.org/10.1016/j.asoc.2004.09.003
9. Neuwirth, E.: Some strategies for Mastermind. Zeitschrift fur Operations Research. Serie B 26(8), B257–B278 (1982)
10. Runarsson, T.P., Merelo, J.J.: Adapting heuristic Mastermind strategies to evolutionary algorithms. In: NICSO 2010 Proceedings. LNCS. Springer, Heidelberg (2010) (to be published) ArXiV: http://arxiv.org/abs/0912.2415v1
11. Stuckman, J., Zhang, G.Q.: Mastermind is NP-complete. CoRR abs/cs/0512049 (2005)
12. Wikipedia: Mastermind (board game) — Wikipedia, The Free Encyclopedia (2009), http://en.wikipedia.org/w/index.php?title=Mastermind_board_game&oldid=317686771 (Online; accessed 9-October-2009)