Efficient solutions for Mastermind using genetic algorithms

Lotte Berghman, Dries Goossens, Roel Leus*
ORSTAT, K.U.Leuven, Naamsestraat 69, 3000 Leuven, Belgium

Abstract

We present a new genetic algorithm for playing the game of Mastermind. The algorithm requires low run-times and results in a low expected number of guesses. Its performance is comparable to that of other meta-heuristics for the standard setting with four positions and six colors, while it outperforms the existing algorithms when more colors and positions are examined. The central idea underlying the algorithm is the creation of a large set of eligible guesses collected throughout the different generations of the genetic algorithm, the quality of each of which is subsequently determined based on a comparison with a selection of elements of the set.

Keywords: Mastermind, genetic algorithms, games.

1 Introduction

Mastermind is a game played by two players, the *codemaker* or *encoder* and the *codebreaker* or *decoder*. The game begins with the codemaker selecting a secret code, a sequence of P colors (digits, pegs or other symbols) $\mathbf{s} = (s_1, s_2, \ldots, s_P) \in C \equiv \{1, 2, \ldots, N\}^P$ chosen from a set of N colors (with repetitions allowed); we represent the different colors by the numbers $1, 2, \ldots, N$. The codebreaker will then try to guess the code by means of a number of guesses $\mathbf{g}_i = (g_{i1}, g_{i2}, \ldots, g_{iP}) \in C$, $i = 1, 2, \ldots$. After each guess, the codemaker responds with two numbers X_i and Y_i. The value X_i is the number of exact matches, counting the number of times $s_p = g_{ip}$, $p = 1, \ldots, P$. The second, the number of "near matches", is the number of digits of the guess which are the right color but in the wrong position. In the real game, X_i is represented by black or red pegs, and Y_i by white pegs. At any stage throughout the game, we call a code $\mathbf{c}$ *eligible* or *feasible* if it results in the same values for X_k and Y_k for all guesses k that have been

*Corresponding author. E-mail: Roel.Leus@econ.kuleuven.be.

played up till that stage, if $\mathbf{c}$ were the secret code; E_i is the set of eligible codes after guess $(i-1)$ (with $E_1 \equiv C$). The game ends when $X_i = P$.

It was the Israeli postmaster and telecommunications expert Mordecai Meirowitz who invented Mastermind in 1970. The English company Invicta Plastics Ltd. bought all the intellectual property rights and brought Mastermind on the market in 1972. Since then, over 55 million copies of the popular game have been sold in more than 80 countries. Presently, a commercial board game exists in two variants: "classic" Mastermind with four positions and six colors, and "super" Mastermind with five positions and eight colors. Moreover, a lot of internet and computer versions are available and the game can also be played simply with pen and paper. Applications in various disciplines, mostly related to artificial intelligence, are cited in Chen et al. (1996), Goddard (2003) and Merelo-Guervós et al. (2006), including adaptive search and computational learning with queries.

In this article we present a new genetic algorithm (GA) to play Mastermind. The prime objective function considered is the average number of guesses needed to find the secret code, where the average is computed over a number of games. The central idea underlying the algorithm is the creation of a large set of eligible guesses collected throughout the different generations of the GA, the quality of each of which is subsequently determined based on a comparison with a selection of elements of the eligible set. Note that when $N = 6$ and $P = 4$, the entire dataset of all $6^4 = 1296$ secret codes is used for evaluation, while for higher values of N and P, a representative dataset is usually constructed by random sampling.

The remainder of this article is structured as follows. First, we survey the existing literature on algorithmic approaches to Mastermind in Section 2. Subsequently, we describe a new GA in Section 3, and we compare its performance with some of the existing algorithms (Section 4). Finally, some conclusions are presented in Section 5.

2 Literature review

We have subdivided the literature on algorithmic approaches to Mastermind into three categories: full enumeration, rules of thumb and meta-heuristics.

2.1 Full enumeration

For each guess, full enumeration investigates all codes in C and retains the code with the highest score on some objective function. With Knuth's (1977) "strategically optimal" algorithm, the decoder can always succeed in five moves or less (4.478 on average) when $P = 4$ and $N = 6$. For each guess, the decoder chooses the combination that minimizes the maximum number of remaining eligible codes. Irving (1979) based his algorithm on Knuth's but minimizes the expected number of remaining codes; he needs

4.369 guesses on average. Norvig (1984) also proposes an algorithm similar to that of Knuth, but only plays *eligible* guesses; he needs 4.47 guesses on average.

Neuwirth (1982) develops a strategy for bringing down the number of guesses to 4.3642, in which the next guess to be made depends on the remaining number of eligible codes. Koyoma & Lai (1994) use full enumeration and reach an average of 4.34 moves (six in the worst case) to find the secret code when playing with four positions and six colors. The authors are able to reduce the search space by considering only one of a number of so-called "case-equivalent" combinations. An obvious disadvantage of this algorithm, shared with the other full-enumeration algorithms, is the run-time (needed to enumerate all codes). Rosu (1999) also describes a full-enumeration algorithm that is extremely time-intensive: when played with $P = 4, N = 6$, the algorithm needs 15 hours and results in an average of 4.34 guesses. Kooi (2005) proposes the "Most Parts Strategy", which needs on average 4.373 guesses for $P = 4$ and $N = 6$. For each guess, this strategy chooses the code that has the largest number of possible responses (X_i, Y_i), given the set E_i.

Bestavros & Belal (1986) combine full enumeration with information theory for the choice of the objective function and reach an average of 3.860 and 3.835 guesses for the strategies called "MaxMin" and "MaxEnt", respectively – but only for a sample of 500 games.

The recent article by Chen et al. (2007) proposes a variant on full enumeration via depth-first search, following the philosophy of *beam search* (Russell & Norvig 2003), in which only a limited number of branches (the most promising ones) is explored in each node of the search tree. For low to high run-times (dependent on the number of explored branches), the average number of guesses ranges from 4.385 to 4.359, with corresponding average run-times of 1.25 and 993.2 seconds.

2.2 Rules of thumb

The aim of what we have called *rules of thumb* is different from the algorithms discussed in the previous paragraph: the goal is now to obtain good results in a short time. The simplest algorithm is probably the one by Shapiro (1983), where the decoder constructs a list of the elements of C in random order. For each guess, he starts from the last examined position in the previous guess and continues the list until he finds an eligible code, which will be his next guess. Swaszek (2000) proceeds similarly, but adopts a list ordered in increasing index of the colors. He uses a fixed first guess and needs 4.758 guesses on average for four positions and six colors.

Rosu (1999) also describes a rule of thumb using 4.64 trials on average when played with $P = 4$ and $N = 6$, which only plays eligible codes. A random combination is chosen for the first guess. For all subsequent guesses, random codes are generated until a first eligible code is found, which is the

next guess. The average number of guesses grows linearly with N and P, while the computation time grows exponentially.

Temporel & Kovacs (2003) propose a new algorithm that minimizes not only the average number of guesses but also the number of evaluated codes. When played with $P = 4$ and $N = 6$, the algorithm needs an average of 4.64 trials and evaluates only 41.2 combinations on average, which is not even four percent of all 1296 possible combinations.

2.3 Meta-heuristics

Bernier et al. (1996), Bento et al. (1999) and Kalisker & Camens (2003) all propose a GA that uses a fitness value that reflects the eligibility. Bernier et al. (1996) need an average of 5.62 guesses over 693 games when played with $N = P = 6$. Bento et al. (1999) reach an average of 6.866 guesses when played with $P = 5$ and $N = 8$. Their initial population contains 300 random codes with P different colors. A certain fraction of the best elements of the parent population replaces the corresponding number of worst elements of the child population. Kalisker & Camens (2003) need 4.75 guesses on average when played with $P = 4$ and $N = 6$, and 6.39 guesses when $P = 5, N = 8$.

Merelo-Guervós et al. (2006) develop an evolutionary algorithm that needs 4.13 trials for $P = 4$ and $N = 6$, averaged over "in some cases, up to 1000, in other cases, only 100" runs, "dependent on N and P". The first guess always contains four different colors. Each generation contains 400 codes. The 50% worst combinations in each generation are eliminated and substituted by the offspring resulting from the application of the genetic operators to the remaining members of the population. When 15 generations are created without eligible code, the population is killed and a new one generated. The average number of evaluated combinations is only 279, which makes this a relatively fast method. Moreover, although the size of the search space grows exponentially with N, the number of combinations evaluated grows only linearly. When a feasible solution is found, it is immediately played.

Ugurdag et al. (2006) present a population-based algorithm inspired by the algorithms of Shapiro, Knuth and Kooi. The first two guesses are fixed. For all subsequent guesses, only eligible codes are played. For each guess, a population $\widehat{E}_i \subseteq E_i$ is created, with the intention that the population $\widehat{E}_i$ be a good representation of E_i. Every $\mathbf{c} \in \widehat{E}_i$ is a candidate for the next guess; for each such candidate, it is checked how it would partition the population $\widehat{E}_i$ if played as guess, each subset of the partition corresponding with one answer (X_i, Y_i). To this aim, each code in the population $\widehat{E}_i$ is assumed to be the secret code and the candidate is scored against it. The number of non-empty subsets in the partition is the fitness value; a candidate $\mathbf{c} \in \widehat{E}_i$ with highest fitness is selected as next guess. The average number of guesses is 4.345 for $P = 4$ and $N = 6$, 6.169 for $P = 5$ and $N = 8$ and 7.079 for

$P = 6$ and $N = 9$.

Singley (2005) starts with a number of fixed guesses to discover the number of occurrences of each color in the secret code, and then continues with a tabu search algorithm. When played with $P = 4$ and $N = 6$, this algorithm requires on average 5.736 guesses. This number increases linearly with the size of the problem, while the search time increases exponentially. The fitness score of a code is a function of the responses for the previous guesses. The search for a new guess starts from the last-played guess, in which two randomly chosen positions are swapped and added to the tabu list. If a chosen pair is already in the list and the new combination is eligible, then the score after making the swap is compared to the best score obtained so far and replaces the latter when the new score is better. A pair that is not yet in the tabu list is swapped, whether it is an improvement over the current combination or not. This procedure continues until an eligible code is obtained.

3 A new genetic algorithm

A number of well-performing algorithms, e.g. the one by Merelo-Guervós et al. (2006), simply play any feasible code as soon as one is found. Merelo-Guervós et al. (2006) point out that an interesting avenue for further research would be to evaluate each code not only by considering feasibility but also by its predictive power. It is exactly such an improvement that we investigate in this article.

In this section, we present our new algorithm with an embedded GA, where a large set of eligible codes is collected throughout the different generations. The quality of each of these codes is determined based on a comparison with a selection of elements of the eligible set. We present an overview of the game and the global steps of the GA as Algorithm 1, and provide details on some key design and implementation issues in the remainder of this section.

As initial guess $\mathbf{g}_1$, we use a fixed code. For the case with four positions and six colors, we obtain the best results with $\mathbf{g}_1 = (1, 1, 2, 3)$, although the influence of this choice on the performance is limited. Nevertheless, this initial guess reduces the average number of guesses required to break the code by 0.02 compared to $\mathbf{g}_1 = (1, 1, 2, 2)$, which is suggested by Knuth (1977) and used by Merelo-Guervós et al. (2006).

From the second move onwards, the guess that is played is determined by the actual GA. Throughout this article, we use a population of size 150. This population is initialized randomly, taking into account that every code in the population should be distinct. Subsequent generations of the population are created through 1-point or 2-point crossover (each with a probability of 0.5) from two parents of the previous generation. With a probability of

crossover

Algorithm 1 A new genetic algorithm

```
Set i = 1;
Play fixed initial guess g_1;
Get response X_1 and Y_1;
while X_i ≠ P do
    i = i + 1;
    Set Ê_i = {} and h = 1;
    Initialize population;
    while (h ⩽ maxgen AND |Ê_i| ⩽ maxsize) do
        Generate new population using crossover, mutation, inversion and permutation;
        Calculate fitness;
        Add eligible combinations to Ê_i (if not yet contained in Ê_i);
        h = h + 1;
    end while
    Play guess g_i ∈ Ê_i;
    Get response X_i and Y_i;
end while
```

g1 = (1, 1, 2, 3)

maxgen = 100, maxsize = 60

gi is the one with least remaining eligible codes

mutation 0.03

inversion 0.02

if code is already in the set

fitness -- to selet the parent

0.03, the crossover is followed by a mutation that replaces the color of one randomly chosen position by a random other color. Furthermore, there is a 0.03 chance of permutation, where the colors of two random positions are switched. Finally, there is a 0.02 chance of inversion, in which case two positions are randomly picked, and the sequence of colors between these positions is inverted. When these procedures lead to a code that is already present in the population, it is replaced by a completely new solution that is constructed in a way that aims to improve the diversity of the population: each digit receives a value from the set $\{1, 2, \ldots, N\}$, where each value has the same probability $\frac{1}{N}$ of being selected.

The probability for a code to be selected as a parent is proportional to its fitness value. In order to compute the fitness value of a chromosome $\mathbf{c}$, we compare it with every previous guess $\mathbf{g}_q$ by determining the number of black pins $X'_q(\mathbf{c})$ and the number of white pins $Y'_q(\mathbf{c})$ that the code $\mathbf{c}$ would score if the previous guess $\mathbf{g}_q$ were the secret code. The difference between X'_q and X_q and between Y'_q and Y_q is an indication of the quality of the code $\mathbf{c}$; if these differences are zero for each previous guess $\mathbf{g}_q$ then the code is eligible.

For the fitness value, it would seem logical to give a larger relative weight a to the difference in black pins than to the difference in white pins, since a black pin is a better result than a white one. Our experiments show, however, that the best result is obtained when black and white pins receive the same weight (see Table 1). The reason for this could be that our algorithm plays an eligible code at each turn. When infeasible codes could be played as well, it would probably be better to associate a higher weight with the black pins, since in this way the codes that are closer to an eligible code obtain a higher fitness value. In order to slick the values of the fitness function, a con-

$a = 1$	4.607	$b = 1$	4.605
$a = 2$	4.608	$b = 2$	4.600
$a = 3$	4.652	$b = 3$	4.606

Table 1: Average number of guesses for different values for a and b.

stant is added, proportional to the number of positions P and the number of played turns i. This term is multiplied by a weight b, which determines the relative importance of this term compared to the number of black and white pins. Experimental results show that the best result is obtained for $b = 2$ (see Table 1). This results in the following fitness function f:

b = 2 , a = 1

$$f(\mathbf{c}; i) = a \left(\sum_{q=1}^{i} |X'_q(\mathbf{c}) - X_q| \right) + \sum_{q=1}^{i} |Y'_q(\mathbf{c}) - Y_q| + bP(i-1).$$

For each guess to be made, at most *maxgen* generations are created by the GA. The eligible codes in each generation are added to the set $\widehat{E}_i$ (if they are not yet contained in that set); the next guess will be chosen out of $\widehat{E}_i$. In line with Ugurdag et al. (2006), we assume that the set $\widehat{E}_i$ is a good representation of E_i. Our approach differs from that of Ugurdag et al. in that we use a GA for creating the "sample" $\widehat{E}_i$, whereas Ugurdag et al. resort to the selection of one feasible solution (using a variant of Shapiro's algorithm) from each subset of a partition of C. This partition is such that each of the subsets contains solutions that are conveniently spread over C (so that different values occur for each of the digits).

Although harmful to the objective function, we limit *maxgen* and the size of $\widehat{E}_i$ in order to restrict the computation time. We find that a good balance is struck with $maxgen = 100$ and $|\widehat{E}_i| = 60$ (see Table 2). Without the cardinality restriction, $|\widehat{E}_2|$ is about 250 on average, which decreases to 55 for $|\widehat{E}_3|$, only 12 for $|\widehat{E}_4|$, and even less for possible subsequent guesses. We conclude that our restriction on the cardinality of $\widehat{E}_i$ tends to constrain the number of candidate eligible codes only for the second guess.

Finally, we need to decide which of the eligible codes in $\widehat{E}_i$ will be our next guess. Ideally, one would want to play the eligible code that leads to the least remaining eligible codes after it has been played. However, as the search space increases, finding this code becomes quite time-intensive, and we therefore resort to alternatives that demand less computational effort.

$\|\widehat{E}_i\|$=50	4.656
$\|\widehat{E}_i\|$=60	4.653
$\|\widehat{E}_i\|$=70	4.650

Table 2: Average number of guesses for different values of $|\widehat{E}_i|$.

	Average number of guesses	Computation time
Random	4.59	0.762 sec
Similar	4.47	0.741 sec
Different	5.12	0.895 sec

Table 3: Average number of guesses for three selection rules.

similarity

First, we look into the influence of playing a code that is most similar to other codes in $\widehat{E}_i$, compared to playing that code that differs most from all other codes in $\widehat{E}_i$, or just playing a randomly selected element from that set. The underlying idea is that similarity to other eligible codes is a proxy for the number of eligible secret codes that will remain after playing that code. More specifically, for each $\mathbf{c} \in \widehat{E}_i$, we determine the number of black and white pins if $\mathbf{c}$ were the guess and another $\mathbf{c}^* \in \widehat{E}_i$ the secret code, for each $\mathbf{c}^* \in \widehat{E}_i \setminus \{\mathbf{c}\}$. These values are summed over all $\mathbf{c}^* \in \widehat{E}_i \setminus \{\mathbf{c}\}$, and the black and white pins receive equal weight. The resulting total score measures to what extent each code resembles the other codes in $\widehat{E}_i$. Our experiments show that better results are obtained when the code that is the most similar to the others is played (see Table 3).

Some algorithms discussed in Section 2 base their score on the possible responses (X_i, Y_i), on the probability that these responses will be obtained and/or on the number of eligible codes that remain after playing a code. As a second alternative, we adopt this global viewpoint as follows: the quality of an eligible code $\mathbf{c} \in \widehat{E}_i$ is evaluated by means of an estimate of the number of remaining eligible codes if $\mathbf{c}$ is the next guess. This estimate is obtained by considering only a random selection $S \subseteq \widehat{E}_i$: for each candidate guess $\mathbf{c} \in \widehat{E}_i$ and each $\mathbf{c}^* \in S \setminus \{\mathbf{c}\}$ functioning as possible secret code, it is checked how many other codes in $S \setminus \{\mathbf{c}, \mathbf{c}^*\}$ would remain eligible. The candidate $\mathbf{c}$ with minimum average number of remaining eligible codes is picked. Our experiments show that even for small selections S, the approach is valuable, in that the obtained result is better than the result after using a random selection. It is obvious that as $|S|$ increases, the solution quality also rises, but so do the computation times. Table 4 shows that the highest-quality results are obtained when the selection S equals $\widehat{E}_i$.

$\lvert S\rvert$	Average number of guesses	Computation time
30	4.431	0.596 sec
40	4.42	0.597 sec
50	4.417	0.61 sec
60	4.39	0.614 sec

Table 4: Average number of guesses and computation time for various sizes of the set S.

Algorithm	Average	Maximum number of guesses
Knuth	4.478	5
Norvig	4.47	6
Chen et al. (2007) (1)	4.385	6
Kooi	4.373	6
Irving	4.369	6
Neuwirth	4.3642	6
Chen et al. (2007) (20)	4.359	6
Koyama and Lai	4.34	5
Rosu	4.34	-
Bestavros and Belal (MaxMin)	3.86*	-
Bestavros and Belal (MaxEnt)	3.835*	-
New GA	4.39	7

Table 5: Results for full enumeration with $P = 4$ and $N = 6$; an asterisk '*' indicates that only a subset of the 6^4 codes was considered for obtaining this result.

Both of the foregoing proposals for choosing a guess from $\widehat{E}_i$ obtain better results than a random selection. It turns out that the second method (pursuing the least remaining eligible codes) does better than the approach based on similarity. Consequently, the second alternative is retained and implemented in the final version of the algorithm, which is used throughout Section 4.

4 A comparison with existing algorithms

In this section, a number of the algorithms available in the literature are compared with our new algorithm, and this for different datasets (different combinations of values for P and N). Our algorithm has been coded[1] in Borland Delphi 6 using Microsoft Visual Studio 2005. The experiments were run on a Dell Inspiron 1150 with an Intel Pentium-4 2.8-GHz processor and 512 MB RAM, equipped with Windows XP Professional.

4.1 $P = 4$ and $N = 6$

For this dataset we report the average number of guesses as well as the maximum number of guesses, for the algorithms that we have implemented ourselves and for those for which the reference text mentions this value. To evaluate the performance of the different algorithms for larger instances (higher values of P and/or N), we perform sampling instead of averaging

[1] Our source code is available on the webpage http://www.econ.kuleuven.be/public/NDBAC96/mastermind.htm.

Algorithm	Average	Maximum number of guesses
Shapiro	4.78	8
Swaszek	4.758	9
Rosu	4.64	8
Temporel and Kovacs	4.64	-
New GA	4.39	7

Table 6: Results for rules of thumb with $P = 4$ and $N = 6$.

Algorithm	Average number of guesses
Singley	5.736
Kalisker and Camens	4.75
Ugurdag et al.	4.345
Merelo-Guervós et al.	4.13*
New GA	4.39

Table 7: Results for meta-heuristics with $P = 4$ and $N = 6$; an asterisk '*' indicates that only a subset of the 6^4 codes was considered for obtaining this result.

over the entire dataset (mainly because the dataset simply contains too many instances). For these datasets, the statistic "maximum number of guesses" loses most of its information, since the more samples are drawn and examined, the higher the probability of obtaining extreme values. We have consequently included the second statistic only for the dataset with $P = 4$ and $N = 6$.

Table 5 shows that all full-enumeration algorithms obtain good results, comparable to the algorithm proposed in this text. When the number of positions and colors increases, however, the computation time of these methods will increase more than linearly because all combinations are evaluated one by one. Consequently, this type of algorithms will certainly not be preferred for larger problems. The algorithm by Chen et al. (2007) takes the number of branches explored in each node of the search tree as input parameter, which is indicated between round brackets. Bestavros & Belal (1986) seem to obtain the best performance of all the algorithms appearing in the literature. The results of the new GA are obtained as the average number of guesses after playing each of the 1296 games three times. All other results in this section are copied from the literature.

From Table 6 we observe that the results of the rules of thumb proposed by Shapiro (1983), Swaszek (2000), Rosu (1999) and Temporel & Kovacs (2003) are also comparable to our new GA. Finally, Table 7 contains the outcomes of a number of meta-heuristics. Merelo-Guervós et al. (2006) report the best results in this category, but a direct comparison may not be appropriate since their average was not computed over the same dataset.

Algorithm	Average number of guesses
Bento et al.	6.866
Kalisker and Camens	6.39
Ugurdag et al.	6.169
Merelo-Guervós et al.	5.904
New GA	5.618

Table 8: Results for meta-heuristics with $P = 5$ and $N = 8$.

Singley (2005) needs significantly more guesses on average, probably because of the number of fixed guesses needed to discover the number of occurrences of each color. None of these references mentions the maximum number of guesses needed.

4.2 $P = 5$ and $N = 8$

A number of meta-heuristics in the literature present results for the dataset corresponding with $P = 5$ and $N = 8$; a summary is depicted in Table 8. For the new GA, our first guess is $\mathbf{g}_1 = (1, 1, 2, 3, 4)$. The best results are obtained by our new GA.

The reader should note that for this dataset as well as for all those discussed infra, all averages are computed by considering only a representative sample of the entire dataset. The results of the new GA are obtained as the average number of guesses after playing 500 games with randomly generated secret codes. All other results are copied from the literature. All articles that clearly state the number of games played for objective-function evaluation also opt for 500 games.

4.3 $P = 6$ and $N = 9$

For this dataset we have adopted a first guess of $\mathbf{g}_1 = (1, 1, 2, 2, 3, 4)$ and increased $|\widehat{E}_i|$ and $|S|$ so that $|\widehat{E}_i| = |S| = 80$. No further parameter tuning is performed, so all other parameters and settings remain the same as for the standard dataset with $P = 4, N = 6$. We have implemented a few algorithms to be able to compare the results of the new GA for this dataset

Algorithm	Average number of guesses	Computation time
Swaszek (first move random)	7.41	1.866 sec
Ugurdag et al.	7.079	
Shapiro	6.76	1.087 sec
Rosu	6.67	1.282 sec
New GA	6.475	1.284 sec

Table 9: Results for $P = 6$ and $N = 9$.

Algorithm	Average number of guesses	Computation time
Swaszek (first move random)	9.2	30 min 31 sec
Rosu	8.577	25 min 44 sec
New GA	8.366	20.571 sec

Table 10: Results for $P = 8$ and $N = 12$.

to more than only the algorithm of Ugurdag et al. (2006). The averages for the different algorithms are quite close to one another (see Table 9). The average computation time needed by Shapiro's (1983) algorithm seems to be a bit shorter, but that is a consequence only of the fact that in this implementation, the generation of a list with all possible combinations in a random order is done beforehand.

The result of Ugurdag et al. (2006) is copied from the reference. All other results are obtained as the average number of guesses after playing 500 games with randomly generated secret codes.

4.4 $P = 8$ and $N = 12$

A few slight changes are made for this dataset: we adapt the first guess to $\mathbf{g}_1 = (1, 1, 2, 2, 3, 3, 4, 5)$ and increase the parameters $|\widehat{E}_i|$, $|S|$ and $maxgen$ so that $|\widehat{E}_i| = |S| = 100$ and $maxgen = 200$. Table 10 shows that Rosu (1999) does quite well as far as the average number of guesses is concerned, but this at expense of very high run-times. The result for the algorithm by Swaszek (2000) is a bit worse for the number of guesses as well as for the calculation time. Both of these algorithms are outperformed by our new GA with respect to the number of guesses, and this with considerably lower computation times.

The results for Rosu and Swaszek and the result of the new GA are obtained as the average number of guesses after playing respectively 50 and 100 games with randomly generated secret codes. For Shapiro's algorithm the entire list of elements of C needs to be stored in memory before the start of the actual game, and this would result in an inordinately large occupation of computer memory for $P = 8, N = 12$, which is the reason why we have not applied the algorithm to this dataset.

4.5 Conclusion

By way of conclusion, the same three algorithms that were considered for $P = 8$ and $N = 12$ are compared in Tables 11 and 12 over the different datasets. The ranking of the algorithms based on the average number of guesses is the same for all the datasets. On the other hand, the run-times of Rosu (1999) and Swaszek (2000) exhibit a considerably faster growth rate than the new GA. Our new algorithm is more time-consuming for low

Algorithm	4;6	6;9	8;12
Swaszek (first move random instead of fixed)	4.768	7.41	9.2
Rosu	4.64	6.67	8.577
New genetic algorithm	4.39	6.475	8.366

Table 11: Average number of guesses for different datasets.

Algorithm	4;6	6;9	8;12
Swaszek (first move random)	0.001 sec	1.866 sec	30 min 31 sec
Rosu	0.001 sec	1.282 sec	25 min 44 sec
New genetic algorithm	0.614 sec	1.284 sec	20.571 sec

Table 12: Average CPU-time for different datasets.

values of P and N, but the computational effort rises only slightly for larger instances, while the other algorithms are clearly less preferable for higher P and N.

5 Summary

In this article we have presented a new genetic algorithm for playing the game of Mastermind, which requires low run-times and results in a low expected number of guesses. A number of well-performing algorithms, e.g. the one by Merelo-Guervós et al. (2006), simply play any feasible code as soon as one is found. Merelo-Guervós et al. point out that an interesting avenue for further research would be to evaluate each code not only by considering feasibility but also by its predictive power. It is exactly such an improvement that we investigate in this article. We do need to point out to the reader that since we use a meta-heuristic that does not perform a systematic search of the solution space, (unnecessarily) long run-times cannot entirely be excluded, although we are confident that our algorithm will only very rarely exhibit such undesirable behavior; this is confirmed by our computational experiments.

The central idea underlying our solution procedure is the creation of a large set of eligible guesses collected throughout the different generations of the genetic algorithm, the quality of each of which is subsequently determined based on a comparison with a selection of elements of the set. We have evaluated the algorithmic performance by means of comparisons with a number of existing algorithms and observe that its performance is comparable to that of other meta-heuristics for the standard setting with four positions and six colors, while it outperforms the existing algorithms when more colors and positions are considered.

References

Bento, L., Pereira, L. & Rosa, A. (1999). Mastermind by Evolutionary Algorithms, *in SAC '99: Proceedings of the Sixth Annual Workshop on Selected Areas in Cryptography, Kingston, Ontario, Canada*, pp. 307–311.

Bernier, J., Herráiz, C., Merelo-Guervós, J., Olmeda, S. & Prieto, A. (1996). Solving MasterMind using GAs and simulated annealing: A case of dynamic constraint optimization, *in PPSN IV: Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, Springer-Verlag, London, UK, pp. 554–563.

Bestavros, A. & Belal, A. (1986). MasterMind: A game of diagnosis strategies, *Bulletin of the Faculty of Engineering, Alexandria University*, Alexandria, Egypt.

Chen, S., Lin, S., Huang, L. & Hsu, S. (2007). Strategy optimization for deductive games, *European Journal of Operational Research* **183**: 757–766.

Chen, Z., Cunha, C. & Homer, S. (1996). Finding a hidden code by asking questions, *in COCOON '96: Computing and Combinatorics, Proceedings of the Second Annual International Conference, Hong Kong*, pp. 50–55.

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

Irving, W. (1979). Towards an optimum Mastermind strategy, *Journal of Recreational Mathematics* **11**(2): 81–87.

Kalisker, T. & Camens, D. (2003). Solving Mastermind using genetic algorithms, *in GECCO '03: Proceedings of the Genetic and Evolutionary Computation Conference, Chicago, USA*, pp. 1590–1591.

Knuth, E. (1977). The computer as Master Mind, *Journal of Recreational Mathematics* **9**: 1–6.

Kooi, B. (2005). Yet another mastermind strategy, *ICGA Journal* **28**(1): 13–20.

Koyoma, K. & Lai, W. (1994). An optimal Mastermind strategy, *Journal of Recreational Mathematics* **25**: 251–256.

Merelo-Guervós, J., Castillo, P. & Rivas, V. (2006). Finding a needle in a haystack using hints and evolutionary computation: the case of evolutionary MasterMind, *Applied Soft Computing* **6**(2): 170–179.

Neuwirth, E. (1982). Some strategies for MasterMind, *Zeitschrift für Operations Research* **26**: 257–278.

Norvig, P. (1984). Playing Mastermind optimally, *SIGART Bulletin* (90): 33–34.

Rosu, R. (1999). *Mastermind*, Master's thesis, North Carolina State University, Raleigh, North Carolina.

Russell, S. & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*, Prentice Hall.

Shapiro, E. (1983). Playing Mastermind logically, *SIGART Bulletin* (85): 28–29.

Singley, A. (2005). *Heuristic solution methods for the 1-dimensional and 2-dimensional Mastermind problem*, Master's thesis, University of Florida.

Swaszek, P. (2000). The mastermind novice, *Journal of Recreational Mathematics* **30**: 193–198.

Temporel, A. & Kovacs, T. (2003). A heuristic hill climbing algorithm for Mastermind, *in UKCI '03: Proceedings of the 2003 UK Workshop on Computational Intelligence, Bristol, United Kingdom*, pp. 189–196.

Ugurdag, H., Sahin, Y. & Baskirt, O. (2006). Population-based FPGA solution to Mastermind game, *AHS '06: Proceedings of the First NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 237–246.