Playing Mastermind With Constant-Size Memory

Benjamin Doerr, Carola Winzen

Playing Mastermind With Constant-Size Memory

Benjamin Doerr[1] and Carola Winzen[2]

1 **Max-Planck-Institut für Informatik**
Saarbrücken, Germany
2 **Max-Planck-Institut für Informatik**
Saarbrücken, Germany

Abstract

We analyze the classic board game of Mastermind with n holes and a constant number of colors. The classic result of Chvátal (Combinatorica 3 (1983), 325–329) states that the codebreaker can find the secret code with $\Theta(n/\log n)$ questions. We show that this bound remains valid if the codebreaker may only store a constant number of guesses and answers. In addition to an intrinsic interest in this question, our result also disproves a conjecture of Droste, Jansen, and Wegener (Theory of Computing Systems 39 (2006), 525–544) on the memory-restricted black-box complexity of the OneMax function class.

1998 ACM Subject Classification F.2.2 [Analysis of algorithm and problem complexity]: Non-numerical Algorithms and Problems

Keywords and phrases Algorithms, Mastermind, black-box complexity, memory-restricted algorithms, query complexity

Digital Object Identifier 10.4230/LIPIcs.STACS.2012.441

1 Introduction

The original *Mastermind* game is a board game for two players invented in the seventies by Meirowitz. It has pegs of six different colors. The goal of the *codebreaker*, for brevity called *Paul* here, is to find a color combination made up by *codemaker* (called *Carole* in the following). He does so by guessing color combinations and receiving information on how close this guess is to Carole's secret code. Paul's aim is to use as few guesses as possible.

For a more precise description, let us call the colors 1 to 6. Write $[n] := \{1, \dots, n\}$ for any $n \in \mathbb{N}$. Carole's secret code is a length-4 string of colors, that is, a $z \in [6]^4$. In each iteration, Paul guesses a string $x \in [6]^4$ and Carole replies with a pair $(\mathrm{eq}(z,x), \pi(z,x))$ of numbers. The first number, $\mathrm{eq}(z,x)$, which is usually indicated via black answer-pegs, is the number of positions, in which Paul's and Carole's string coincide. The other number, $\pi(z,x)$, usually indicated by white answer-pegs, is the number of additional pegs having the right color, but being in the wrong position. Formally $\mathrm{eq}(z,x) := |\{i \in [4] \mid z_i = x_i\}|$ and $\pi(z,x) := \max_{\rho \in S_4} |\{i \in [4] \mid z_i = x_{\rho(i)}\}| - \mathrm{eq}(z,x)$, where S_4 denotes the set of all permutations of [4]. Paul "wins" the game if he guesses Carole's string, that is, if Carole's answer is $(4,0)$.

We are interested in strategies for Paul that guarantee him to find the secret code with few questions. We thus adopt a worst-case view with respect to Carole's secret code. This is equivalent to assuming that Carole may change her hidden string at any time as long as it remains consistent with all previous answers (*devil's strategy*).

Previous results. Mathematics and computer science literature produces a plethora of results on the Mastermind problem. For the original game with 6 colors and 4 positions,

29th Symposium on Theoretical Aspects of Computer Science (STACS'12).
Editors: Christoph Dürr, Thomas Wilke; pp. 441–452
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Knuth [8] showed that Paul needs at most four queries until being able to identify Carole's string (which he may query in the fifth iteration to win the game).

Chvátal [3] studies a general version of this game with k colors and n positions, that is, the secret code is a length-n string $z \in [k]^n$. Denote by $d(n,k)$ the minimum number of guesses that enable Paul to win the game for any secret code. Chvátal proves that for $k < n^{1-\varepsilon}$, $\varepsilon > 0$ an arbitrarily small constant, we have $d(n,k) = O(\frac{n \log k}{\log n - \log k})$. More precisely, he shows that for any $\varepsilon > 0$ and n sufficiently large, $(2+\varepsilon)\frac{n(1+2\log k)}{\log n - \log k}$ guesses chosen from $[k]^n$ independently and uniformly at random, with high probability, suffice to distinguish between all possible codes (that is, each secret code leads to a different sequence of answers). Therefore, the secret code can be determined after that many guesses. This remains true if Carole replies only with black answer-pegs, that is, if for any of Paul's guesses x she reveals to him only $\mathrm{eq}(z,x)$, the number of bits, in which her and Paul's string coincide.

For larger values of k, the following is known. For $n \le k \le n^2$, Chvátal proves $d(n,k) \le 2n \log k + 4n$ and for $k = \omega(n^2 \log n)$ he shows $(k-1)/n \le d(n,k) \le \lceil k/n \rceil + d(n,n^2)$. These results have subsequently been improved. Chen, Cunha, and Homer [2] show that $d(n,k) \le 2n\lceil \log n \rceil + 2n + \lceil k/n \rceil + 2$ for $k \ge n$. Goodrich [7] proves $d(n,k) \le n\lceil \log k \rceil + \lceil (2-1/k)n \rceil + k$ for arbitrary k.

For $k = 2$ colors, the Mastermind problem is related to the well-studied coin weighing problem. For this reason, first results on this problem date back to years as early as 1963, when Erdős and Rényi [6] show that $d(n,2) = \Theta(n/\log n)$.

Concerning the computational complexity, Stuckman and Zhang [9] show that it is NP-hard to decide whether a given sequence $(x^{(i)}, (\mathrm{eq}^{(i)}, \pi^{(i)}))_{i=1}^t$ of queries $x^{(i)}$ and answers $(\mathrm{eq}^{(i)}, \pi^{(i)})$ of black and white pegs has a secret code leading to these answers, i.e., whether there exists a string $z \in [k]^n$ such that $\mathrm{eq}(z, x^{(i)}) = \mathrm{eq}^{(i)}$ and $\pi(z, x^{(i)}) = \pi^{(i)}$ for all $i \in [t]$. Goodrich [7] proves that this is already NP-hard if we only ask for consistence with the black answer-peg replies $\mathrm{eq}^{(i)}$.

Our results. Originally motivated by a conjecture on black-box complexities (cf. Section 2), we study a memory-restricted version of the Mastermind problem. Since this original motivation asks for the case of two colors only, we restrict ourselves to the number k of colors being constant, though clearly our methods can also be used to analyze larger numbers of colors.

The memory-restriction can be briefly described as follows. Given a memory of size $m \in \mathbb{N}$, Paul can store up to m guesses and Carole's corresponding replies. Based *only* on this information, Paul decides on his next guess. After receiving Carole's reply, based only on the content of the memory, the current guess, and the current answer, he decides which m out of the $m+1$ strings and answers he keeps in the memory. Note that our memory restriction means that Paul truly has no other memory, in particular, no iteration counters, no experience that certain colors are not used, and so one. So formally Paul's strategy consists of a guessing strategy, which can be fully described by a mapping from m-sets of guesses and answers to strings $x \in [k]^n$, and a forgetting strategy, which maps $(m+1)$-sets of guesses and answers to m-subsets thereof.

Clearly, a memory-restriction makes Paul's life not easier. The $O(n/\log n)$ strategies by Erdős and Rényi [6] and by Chvátal [3] do use the full history of guesses and answers and thus only work with a memory of size $\Theta(n/\log n)$. Surprisingly, this amount of memory is not necessary. In fact, one single memory cell is sufficient.

▶ **Theorem 1.** *Let $k \in \mathbb{N}$. For all $n \in \mathbb{N}$, Paul has a size-one memory strategy winning the Mastermind game with k colors and n positions in $O(n/\log n)$ guesses. This remains true if we allow Carole to play a devil's strategy and if Carole only reveals the number of fully correct pegs $\mathrm{eq}(x,z)$ ("black answer-peg version of Mastermind").*

The bound in Theorem 1 is asymptotically tight. A lower bound of $\Omega(n/\log n)$ is already true without memory restrictions. This follows easily from an information theoretic argument, cf. [6] or [3]. Our result disproves a conjecture of Droste, Jansen, and Wegener [5], who believed that a lower bound of $\Omega(n \log n)$ should hold for the 2-color black answer-peg Mastermind problem with memory-restriction one.

The proof of Theorem 1 is quite technical. For a clearer presentation of the ideas, we first consider the size-two memory-restricted model, cf. Section 3. The proof of Theorem 1 is given in Section 4. Before going into the proofs, in the following section we sketch the connection between Mastermind games and black-box complexities.

2 Mastermind and Black-Box Complexities

In this section, we describe the connection between the Mastermind game and black-box complexity. The reader only interested in the Mastermind result may skip this section without loss.

Roughly speaking, the *black-box complexity* of a set of functions is the number of function evaluations needed to find the optimum of an unknown member from that set. Since problem-unspecific search heuristics such as randomized hill-climbers, evolutionary algorithms, simulated annealing etc. do optimize by repeatedly generating new search points and evaluating their objective values ("fitness"), the black-box complexity is a lower bound on the efficiency of such general-purpose heuristics [5].

Black-box complexity. Let $\mathcal{S}$ be a finite set. A (randomized) algorithm following the scheme of Algorithm 1 is called black-box optimization algorithm for functions $\mathcal{S} \to \mathbb{R}$.

Algorithm 1: Scheme of a black-box algorithm for optimizing $f : \mathcal{S} \to \mathbb{R}$

```
Initialization: Sample x^(0) according to some probability distribution p^(0) on S;
Query f(x^(0));
for t = 1, 2, 3, ... do
    Depending on ((x^(0), f(x^(0))), ..., (x^(t-1), f(x^(t-1)))) choose a probability
    distribution p^(t) on S and sample x^(t) according to p^(t);
    Query f(x^(t));
```

For such an algorithm A and a function $f : \mathcal{S} \to \mathbb{R}$, let $T(A, f) \in \mathbb{R} \cup \{\infty\}$ be the expected number of fitness evaluations until A queries for the first time some $x \in \arg\max f$. We call $T(A, f)$ the *runtime of A for f*. For a class $\mathcal{F}$ of functions $\mathcal{S} \to \mathbb{R}$, the *$A$-black-box complexity of $\mathcal{F}$* is $T(A, \mathcal{F}) := \sup_{f \in \mathcal{F}} T(A, f)$, the worst-case runtime of A on $\mathcal{F}$. Let $\mathcal{A}$ be a class of black-box algorithms for functions $\mathcal{S} \to \mathbb{R}$. Then the *$\mathcal{A}$-black-box complexity of $\mathcal{F}$* is $T(\mathcal{A}, \mathcal{F}) := \inf_{A \in \mathcal{A}} T(A, \mathcal{F})$. If $\mathcal{A}$ is the class of all black-box algorithms, we also call $T(\mathcal{A}, \mathcal{F})$ the *unrestricted black-box complexity* of $\mathcal{F}$.

As said, the unrestricted black-box complexity is a lower bound for the efficiency of randomized search heuristics optimizing $\mathcal{F}$. Unfortunately, often this lower bound is not very useful. For example, Droste, Jansen, and Wegener [5] observe that the NP-complete MaxClique problem on graphs of n vertices has a black-box complexity of only $O(n^2)$.

Black-box algorithms with bounded memory. As a possible solution to this dilemma, Droste, Jansen, and Wegener suggest to restrict the class of algorithms considered from all black-box optimization algorithms to a reasonably large subset. A natural restriction is to forbid the algorithm to exploit the whole history of search points evaluated.

This is motivated by the fact that many heuristics, e.g., evolutionary algorithms, only store a bounded size *population* of search points. Simple hill-climbers or the Metropolis algorithm even store only one single search point.

Algorithm 2 is the scheme of a black-box algorithm with bounded memory of size μ. It is important to note that a black-box algorithm with bounded memory is not allowed to access any other information than the one stored in the μ pairs $(x^{(1)}, f(x^{(1)})), \dots, (x^{(\mu)}, f(x^{(\mu)}))$, which are currently stored in the memory and, in the selection step, also the information provided by $(x^{(\mu+1)}, f(x^{(\mu+1)}))$. In particular, the algorithm does not have access to an iteration counter.

Algorithm 2: Scheme of a black-box algorithm with memory of size μ for optimizing function $f : \mathcal{S} \to \mathbb{R}$

```
Initialization: M ← ∅;
for t = 1, 2, ... do
    Depending (only) on M choose a probability distribution p on S and sample x^(μ+1)
    according to p; //variation step
    Query f(x^(μ+1));
    Select M ⊆ M ∪ {(x^(μ+1), f(x^(μ+1)))} of size |M| ≤ μ; //selection step
```

Mastermind and the OneMax function class. A test function often regarded to analyze how the randomized search heuristic under investigation progresses in easy parts of the search space, is the simple OneMax function $\textsc{OneMax} : \{0,1\}^n \to \mathbb{R}, x \mapsto \sum_{i=1}^n x_i$. Note that $\textsc{OneMax}(x) = \mathrm{eq}((1,\dots,1), x)$ for all $x \in \{0,1\}^n$. In fact, for any $z \in \{0,1\}^n$, $\mathrm{eq}(z, \cdot)$ yields an equivalent optimization problem. Let us denote by $\textsc{OneMax}_n := \{\mathrm{eq}(z, \cdot) \mid z \in \{0,1\}^n\}$ the class of all these functions.

Due to a coupon collector effect, many classical randomized search heuristics like randomized local search or the $(\mu + \lambda)$ evolutionary algorithm (with μ, λ constants) need $\Theta(n \log n)$ function evaluations to optimize $\textsc{OneMax}_n$.

As a moments thought reveals, black-box algorithms optimizing $\textsc{OneMax}_n$ correspond to strategies for Paul in the Mastermind game (without memory restriction) with two colors and only black answer-pegs used. Hence the unrestricted black-box complexity of $\textsc{OneMax}_n$ is $\Theta(n/\log n)$ by the results of Erdős and Rényi [6] and Chvátal [3].

This connection was seemingly overlooked so far in the randomized search heuristics community, where Droste, Jansen, and Wegener [5] prove an upper bound of $O(n)$ and later Anil and Wiegand [1] prove the asymptotically correct bound of $O(n/\log n)$. Since already the first bound is lower than what many randomized search heuristics achieve, Droste, Jansen, and Wegener suggest to investigate the memory-restricted black-box complexity of $\textsc{OneMax}_n$. They conjecture in [5, Section 6] that a memory restriction of size one leads to a black-box complexity of order $\Theta(n \log n)$.

Again, clearly, the memory-restricted black-box complexity of $\textsc{OneMax}_n$ and optimal strategies for Mastermind with two colors, black answer-pegs only, and a corresponding memory restriction are equivalent questions. Consequently, our result can be rephrased to saying that the black-box complexity of $\textsc{OneMax}_n$ even with the memory restricted to one is $\Theta(n/\log n)$, disproving the conjecture of Droste, Jansen, and Wegener.

3 The Mastermind Game with Memory of Size Two

Since the proof of Theorem 1 is quite technical, we give in this section a simpler proof showing that with a memory of size two Paul can win the game using only $O(n/\log n)$ guesses. Already this proof contains many ingredients needed to prove Theorem 1, e.g., the use of the random guessing strategy with limited memory, the block-wise determination of the secret code, and the simulation of iteration counters in the memory.

Let $k \geq 2$ be the number of colors used. In particular for $k = 2$, it will be convenient to label the colors from 0 to $k-1$. Let us denote the set of colors by $\mathcal{C} := [0..k-1] := \{0, 1, \dots, k-1\}$. We assume that k is a constant and that the number n of positions in the string is large, that is, all asymptotic notation is with respect to n.

▶ **Theorem 2.** *Paul has a size-two memory strategy winning the black answer-peg only Mastermind game with k colors and n positions in $O(n/\log n)$ guesses. This remains true if we allow Carole to play a devil's strategy.*

As many previous works, the proof of Theorem 2 heavily relies on *random guessing*. For the case of $k = 2$ colors, already Erdős and Rényi [6] showed that there is a $t \in \Theta(n/\log n)$ such that t guesses $x^{(1)}, \dots, x^{(t)}$ chosen from $\{0,1\}^n$ independently and uniformly at random, together with Carole's black answer-peg answers, uniquely define the hidden code. This was generalized by Chvátal [3] to the following result.

▶ **Theorem 3** (from [3]). *Let $\varepsilon > 0$, let $n > n(\varepsilon)$ be sufficiently large and let $k < n^{1-\varepsilon}$. Let $x^{(1)}, \dots, x^{(t)}$ be $t \geq (2+\varepsilon)\frac{n(1+2\log k)}{\log n - \log k}$ samples chosen from $\mathcal{C}^n$ independently and uniformly at random. Then for all $z \in \mathcal{C}^n$, the set*

$$\mathcal{S}^{consistent} := \{y \in \mathcal{C}^n \mid \forall i \in [t] : \mathrm{eq}(y, x^{(i)}) = \mathrm{eq}(z, x^{(i)})\}$$

satisfies $\mathrm{E}[|\mathcal{S}^{consistent}|] \leq 1 + 1/n$.

Since the strategy implicit in Theorem 3 needs a memory of size $\Theta(n/\log n)$, we cannot apply it directly in our setting. We can, however, adapt it to work on smaller portions ("blocks") of the secret code, and this with much less memory.

Let $y \in \mathcal{C}^n$ and let $B \subseteq [n]$ be a block (i.e., an interval) of size $s := \lceil \sqrt{n} \rceil$. As we shall see, by $t \in O(s/\log s)$ times guessing a string obtained from y by replacing the colors in B by randomly chosen ones (and guessing k additional *reference strings*), we can determine $z_{|B}$, the part of the secret code z in block B.

We can do so with a memory of size two only. We store the string obtained from y by altering it on B (*sampling string*) in one cell. Note that we do not need to remember y, as we only need to ensure that our guesses agree in the positions $[n] \setminus B$. We use the other memory cell (*storage string*, in the following typically denoted by x) to store the random substrings of length s substituted into y at B, and Carole's answers. Note that each such answer can be encoded in binary using $\ell_n \in O(\log n)$ entries of the string. Hence the t guesses and answers can be memorized using a total number of $t(s + \ell_n) = O(n/\log n)$ positions.

This approach allows us to determine s positions of z using $t = O(s/\log s)$ guesses. Hence we can determine the secret code z with $t\lceil n/s \rceil = O(n/\log n)$ guesses as desired.

In Algorithm 3 (notation used will be introduced below) we make this strategy more precise by giving it in pseudo-code. Note, however, that this algorithm does not fully satisfy the size-two memory restriction. The reason is that the queries do not only depend on the current state of the memory, but also on iteration counters and, e.g. in lines 9 and 11, on the program counter. Further below, in Algorithm 4 we shall remove this shortcoming with a few additional technicalities, which we are happy to spare for the moment.

Algorithm 3: An almost size-two memory-restricted algorithm winning the k-color black answer-peg only Mastermind game in $O(n/\log n)$ guesses. **Remark:** x denotes the unique string in $\mathcal{M}$ with $x_n = 1$ and y denotes the unique string in $\mathcal{M}$ with $y_n = 0$.

1 **Initialization:** $y \leftarrow [0 \dots 0]$;
2 Query $\mathrm{eq}(z, y)$ and update $\mathcal{M} \leftarrow \{(y, \mathrm{eq}(z, y))\}$;
3 **for** $i = 1$ **to** $\lceil (n-1)/s \rceil$ **do**
4 $x \leftarrow [0 \dots 0|1]$; //initialization of x
5 Query $\mathrm{eq}(z, x)$ and update $\mathcal{M}$ by adding $(i = 1)$ or replacing $(i > 1)$ $(x, \mathrm{eq}(z, x))$ in $\mathcal{M}$;
6 **for** $q = 0$ **to** $t + k - 1$ **do**
7 **if** $q < k$ **then** $y \leftarrow \mathtt{substitute}(y, B_i, [q \dots q])$; //reference string
8 **else** $y \leftarrow \mathtt{substitute}(y, B_i, r)$ where $r \in \mathcal{C}^{|B_i|}$ u.a.r.; //random guess
9 Query $\mathrm{eq}(z, y)$ and update $\mathcal{M}$ by replacing $(y, \mathrm{eq}(z, y))$;
10 $x \leftarrow [x_1 \dots x_{p_1(x)} | \mathtt{BLOCK}_i(y) | \mathtt{binary}_{\ell_n}(\mathrm{eq}(z, y)) | 1 | 0 \dots 0 | 1]$; //add y's info to x
11 Query $\mathrm{eq}(z, x)$ and update $\mathcal{M}$ by replacing $(x, \mathrm{eq}(z, x))$;
12 **while** $\Delta_i(y) < |B_i|$ **do**
13 $y \leftarrow \mathtt{substitute}(y, B_i, w)$, where $w \in \mathcal{S}_i^{\text{consistent}}$ u.a.r.;
14 Query $\mathrm{eq}(z, y)$ and update $\mathcal{M}$ by replacing $(y, \mathrm{eq}(z, y))$;
15 **while** $\mathrm{eq}(z, y) < n$ **do** $y \leftarrow \mathtt{substitute}(y, \{n\}, c)$, where $c \in \mathcal{C}$ u.a.r., and query $\mathrm{eq}(z, y)$;

Before we argue for the correctness of Algorithm 3, let us fix the notation. For any string $x \in \mathcal{C}^n$ we also write $x = [x_1 \dots x_n]$. To ease reading, we allow ourselves to indicate different structural components of x by vertical bars, e.g., $x = [x_1 \dots x_p | x_{p+1} \dots x_n]$. For $i \in [\lceil (n-1)/s \rceil]$ let $B_i := \{(i-1)s + 1, \dots, is\} \cap [n-1]$, the positions of the i-th block. Set

$$\mathtt{BLOCK}_i(x) := x_{|B_i} := [x_{(i-1)s+1} \dots x_{\min\{is, n-1\}}] ,$$

the i-th block of x. For any string $r \in \mathcal{C}^{|B_i|}$ we define

$$\mathtt{substitute}(x, B_i, r) := [x_1 \dots x_{(i-1)s} | r | x_{\min\{is, n-1\}+1} \dots x_n] ,$$

the string with the i-th block substituted by r. Similarly, let $\mathtt{substitute}(y, \{n\}, c) := [y_1 \dots y_{n-1} | c]$. Note that we do not assign the n-th position to any of the blocks. We do so because in Algorithms 3 and 4 we shall use that position to indicate, which one of the two strings in the memory $\mathcal{M}$ is the storage string (the unique $x \in \mathcal{M}$ with $x_n = 1$) and which one is the sampling string (the unique string $y \in \mathcal{M}$ with $y_n = 0$).

Let $p_1(x) := \max\{i \in [n-1] \mid x_i = 1\}$, the largest position $i < n$ of x with entry "1". As mentioned above, we encode Carole's answers $\mathrm{eq}(z, y) \in [0..n]$ in binary, using $\ell_n := \lceil \log n \rceil + 1$ positions, and we denote this binary encoding of length ℓ_n by $\mathtt{binary}_{\ell_n}(\mathrm{eq}(z, y))$. By $\Delta_i(y)$ we denote the contribution of the i-th block to the value $\mathrm{eq}(z, y)$, i.e., $\Delta_i(y)$ is the number of positions in the i-th block, in which Paul's guess y and Carole's secret code z coincide. Formally, $\Delta_i(y) := \mathrm{eq}(z_{|B_i}, y_{|B_i})$. Lastly, let $\mathcal{S}_i^{\text{consistent}}$ be the set of strings w of length $|B_i|$ such that $\mathtt{substitute}(z, B_i, w)$ is consistent with all of Carole's replies (formal definition follows). We shall see below that both $\Delta_i(y)$ and $\mathcal{S}_i^{\text{consistent}}$ can be computed solely from the content of the memory cells (lines 12–14).

We now argue for the correctness of Algorithm 3. Let us consider the state of the memory after having sampled all t random samples for the i-th block (that is, we are in lines 12–14).

We show that based on the information given in the memory, we can restore the full history of guesses for the i-th block. To this end, first note that for any guess y done in line 9, we used $s + \ell_n + 1$ positions in x for storing its information (line 10; we add the additional "1" at the end to ease determining via $p_1(x)$ the positions in x, which have not yet been used for storing information). In lines 6–11 we first asked and stored k non-random guesses $x^c = \texttt{substitute}(y, B_i, [c \dots c])$ and we stored these *reference strings* together with Carole's replies $\mathrm{eq}(z, x^c) = \sum_{h=1}^{\ell_n} 2^{h-1} x_{c(s+\ell_n+1)-h}$, $c \in [0..k-1]$. Therefore, for $j \in [t]$, the j-th random sample is $r^{(j)} = [x_{(k+j-1)(s+\ell_n+1)+1} \dots x_{(k+j-1)(s+\ell_n+1)+|B_i|}]$ and the corresponding query was $y^{(j)} = \texttt{substitute}(y, B_i, r^{(j)})$. We have stored Carole's reply to this guess in binary, and we can infer $\mathrm{eq}(z, y^{(j)}) = \sum_{h=1}^{\ell_n} 2^{h-1} x_{(k+j)(s+\ell_n+1)-h}$. This shows how to regain the full guessing history.

Next we show how to compute the contributions $\Delta_i(y^{(j)})$ of the entries in the i-th block. To this end, note that the constant substrings $[c \dots c]$ in the reference strings x^c in total contribute exactly $|B_i|$ to the sum $\mathrm{eq}(z, x^0) + \dots + \mathrm{eq}(z, x^k)$. Formally, $\sum_{c=0}^{k-1} \mathrm{eq}([z_{(i-1)s+1} \dots z_{\min\{is, n-1\}}], [c \dots c]) = |B_i|$. Since all other positions of the sampling string y are not changed during the phase, in which we determine the i-th block, we infer that

$$\Delta_i(y^{(j)}) = \mathrm{eq}(z, y^{(j)}) - (\mathrm{eq}(z, x^0) + \dots + \mathrm{eq}(z, x^k) - |B_i|)/k\,.$$

Consequently, in lines 12–14, the algorithm can compute $\Delta_i(y^{(j)})$ for all $j \in [t]$. From this it can infer

$$\mathcal{S}_i^{\text{consistent}} := \{\tilde{z} \in \mathcal{C}^{|B_i|} \mid \forall j \in [t] : \mathrm{eq}(\tilde{z}, \texttt{BLOCK}_i(y^{(j)})) = \Delta_i(y^{(j)})\},$$

the set of possible code segments in B_i. By Theorem 3, the expected size of $\mathcal{S}_i^{\text{consistent}}$ is bounded from above by $1 + 1/|B_i|$. Thus, in lines 12–14 we need an expected number of $1 + 1/|B_i|$ samples w chosen from $\mathcal{S}_i^{\text{consistent}}$ uniformly at random until we find a $y = \texttt{substitute}(y, B_i, w)$ with $\Delta_i(y) = s$ (which implies that the i-th block of y coincides with Carole's secret code). This shows how we determine the entries of the i-th block in an expected total number of $t = O(s/\log s)$ guesses.

When Algorithm 3 executes line 15, all but the last entry of y coincide with Carole's secret code. Hence trying random colors in the n-th position finds the hidden code z with an additional expected number of $k = \Theta(1)$ guesses.

To turn Algorithm 3 into a size-two memory-restricted one, we use the first ℓ_n entries of x to store in binary the iteration counter i, which indicates the index of the block currently being under consideration. This will move the storage space for the guesses and answers by ℓ_n positions to the right. Formally, we define $i(x) := \sum_{h=0}^{\ell_n - 1} 2^h x_{\ell_n - h}$. The inner for loop needs no additional memory to be simulated, because we can learn from $p_1(x)$ how many guesses $q(x)$ have been queried already. More precisely, since storing each guess requires $s + \ell_n + 1$ positions and the first ℓ_n positions are used for indicating the number of already determined entries, we have $q(x) := (p_1(x) - \ell_n)/(s + \ell_n + 1)$.

Lastly, we need to replace the sequential queries in lines 9 and 11 of Algorithm 3 (as this exploits information stored in the program counter). Fortunately, again we can deduce from the memory where we stand. We define a function $\texttt{Part}(y, x)$, which equals 1 if the information of y has been added to the storage string x already and which equals 0 otherwise.

Algorithm 4: A size-two memory-restricted algorithm winning the k-color black answer-peg only Mastermind game in $O(n/\log n)$ guesses. **Remark:** x denotes the unique string in $\mathcal{M}$ with $x_n = 1$ and y denotes the unique string in $\mathcal{M}$ with $y_n = 0$.

1 **Initialization:** Let $\mathcal{M} \leftarrow \emptyset$; // clear memory
2 **if** $\mathcal{M} = \emptyset$ **then**
3 $\quad y \leftarrow [0...0]$; //first reference string
4 $\quad$ Query $\mathrm{eq}(z, y)$ and update $\mathcal{M} \leftarrow \{(y, \mathrm{eq}(z, y))\}$;
5 **else if** $|\mathcal{M}| = 1$ **then**
6 $\quad x \leftarrow [0...0|1]$; //initialization of storage string
7 $\quad$ Query $\mathrm{eq}(z, x)$ and update $\mathcal{M} \leftarrow \mathcal{M} \cup \{(x, \mathrm{eq}(z, x))\}$;
8 **else if** $i(x) < \lceil (n-1)/s \rceil$ **then**
9 $\quad$ **if** $x = [0 \dots 0|1]$ **or** $\Delta_{i(x)}(y) = |B_{i(x)}|$ **then**
10 $\quad\quad x \leftarrow [\mathtt{binary}_{\ell_n}(i(x)+1)|\mathtt{BLOCK}_{i(x)+1}(y)|\mathtt{binary}_{\ell_n}(\mathrm{eq}(z, y))|1|0 \dots 0|1]$; //clear storage string and add first reference string
11 $\quad\quad$ Query $\mathrm{eq}(z, x)$ and update $\mathcal{M}$ by replacing $(x, \mathrm{eq}(z, x))$;
12 $\quad$ **else if** $\mathit{Part}(y, x) = 1$ **and** $q(x) < t + k$ **then**
13 $\quad\quad$ **if** $q(x) < k$ **then** $y \leftarrow \mathtt{substitute}(y, B_{i(x)}, [q(x) \dots q(x)])$; //reference string
14 $\quad\quad$ **else** $y \leftarrow \mathtt{substitute}(y, B_{i(x)}, r)$ where $r \in \mathcal{C}^{|B_{i(x)}|}$ u.a.r.; //random guess
15 $\quad\quad$ Query $\mathrm{eq}(z, y)$ and update $\mathcal{M}$ by replacing $(y, \mathrm{eq}(z, y))$;
16 $\quad$ **else if** $\mathit{Part}(y, x) = 0$ **and** $\Delta_{i(x)}(y) < |B_{i(x)}|$ **then**
17 $\quad\quad x \leftarrow [x_1 \dots x_{p_1(x)}|\mathtt{BLOCK}_{i(x)}(y)|\mathtt{binary}_{\ell_n}(\mathrm{eq}(z, y))|1|0 \dots 0|1]$; //add y's info to x
18 $\quad\quad$ Query $\mathrm{eq}(z, x)$ and update $\mathcal{M}$ by replacing $(x, \mathrm{eq}(z, x))$;
19 $\quad$ **else if** $\mathit{Part}(y, x) = 1$ **and** $q(x) = t + k$ **then**
20 $\quad\quad y \leftarrow \mathtt{substitute}(y, B_{i(x)}, w)$ where $w \in \mathcal{S}^{\text{consistent}}_{i(x)}$ chosen u.a.r.;
21 $\quad\quad$ Query $\mathrm{eq}(z, y)$;
22 $\quad\quad$ **if** $\Delta_{i(x)}(y) = |B_{i(x)}|$ **then** Update $\mathcal{M}$ by replacing $(y, \mathrm{eq}(z, y))$;
23 **else if** $i(x) = \lceil (n-1)/s \rceil$ **then**
24 $\quad y \leftarrow \mathtt{substitute}(y, \{n\}, c)$ where $c \in \mathcal{C} \backslash \{y_n\}$ u.a.r.;
25 $\quad$ Query $\mathrm{eq}(z, y)$;
26 Go to line 2;

That is, we set

$$\mathtt{Part}(y, x) = \begin{cases} 1, & \text{if } \sum_{i=1}^{\ell_n} 2^{i-1} x_{p_1(x)-i} = \mathrm{eq}(z, y) \\ & \quad \text{and } \mathtt{BLOCK}_{i(x)}(y) = [x_{p_1(x)-\ell_n-|B_{i(x)}|} \dots x_{p_1(x)-\ell_n-1}] \\ 0, & \text{otherwise}. \end{cases}$$

Note that $\mathtt{Part}(y, x) = 1$ indicates that the information of y has been stored in x also in the case that our current sample equals the previous one. This is no problem as then the current guess does not give any new information. Hence the use of `Part` modifies the algorithm to sample t random guesses without immediate repitition. Note that the probability to sample the same string $r \in \mathcal{C}^{|B_{i(x)}|}$ twice in a row is at most $1/2$ (if the last block consists only of one position and $k = 2$) and is typically much smaller. Hence, occurrences of this event have no influence on the asymptotic number of guesses needed to win the game.

With these modifications, Algorithm 3 becomes the truly size-two memory-restricted Algorithm 4.

4 Memory of Size One: Proof of Theorem 1

Compared to the situation in Section 3, Paul faces two additional challenges in the size-one memory-restricted setting. The obvious one is that he has less memory available, in particular, after a large part of the code has been determined and needs to be stored. The more subtle one is that he cannot any longer query a search point and then store whatever is worth storing in the second memory cell. With one memory cell, all he can do is to guess a new string and keep or forget it.

Before we give a proof of Theorem 1, let us discuss a linear time winning strategy, i.e., a strategy that allows Paul to find Carole's secret code in a linear expected number of guesses using one memory cell only. This linear time strategy will be used in the proof of Theorem 1 to determine the last $\Theta(n/\log n)$ entries of the secret code.

The basic idea of the linear time strategy is to test each position one by one, from left to right. Since we have just one memory cell, we need to indicate in this one string, which entries have been determined already. We do so by keeping all not yet determined entries at one identical value different from the one of the entry determined last. To this end, let us for all $x \in \mathcal{C}^n$ define

$$\mathrm{tn}(x) := \min\{i \in [n] \mid \forall j \in \{i, \dots, n\} : x_j = x_i\},$$

the *tail number of* x. The following lemma describes the linear time strategy.

▶ **Lemma 4.** *Let $x \in \mathcal{C}^n$. Furthermore, let us denote Carole's secret code by $z \in \mathcal{C}^n$. Let us assume that the first $\mathrm{tn}(x) - 1$ entries of z have been determined (i.e., Carole can no longer change the entries of $[z_1 \dots z_{\mathrm{tn}(x)-1}]$). Further assume that $x_i = z_i$ for all $i < \mathrm{tn}(x)$ and that $\mathcal{M} = \{(x, \mathrm{eq}(z, x))\}$ is the current content of the memory cell.*

There is a size-one memory-restricted guessing procedure `LinAlg` *that—even if Carole plays a devil's strategy—after an expected constant number of successive calls modifies the memory such that the string y now in the memory satisfies $y_i = z_i$ for all $i \leq \mathrm{tn}(x)$ and $\mathrm{tn}(y) = \mathrm{tn}(x) + 1$. Every call of* `LinAlg` *requires only one guess.*

Interestingly, for the definition of `LinAlg`, we need to distinguish between the cases of $k = 2$ and $k \geq 3$ colors, as certain arguments exploit particular properties of these cases. We claim that Algorithm 5 certifies Lemma 4 for $k = 2$ colors. Here we denote, for all $i \in [n]$, by e_i^n the i-th unit vector of length n.

Algorithm 5: Routine `LinAlg` for $k = 2$ colors

Assumption: The string $x \in \{0,1\}^n$ in the memory satisfies $\mathrm{tn}(x) < n$ and $x_i = z_i$ for all $i < \mathrm{tn}(x)$;
Sample $y \in \{x \oplus e^n_{\mathrm{tn}(x)}, x \oplus \sum_{i=\mathrm{tn}(x)+1}^{n} e_i^n\}$ uniformly at random;
Query $\mathrm{eq}(z, y)$;
if $y = x \oplus e^n_{\mathrm{tn}(x)}$ **then**
  if $\mathrm{eq}(z, y) > \mathrm{eq}(z, x)$ **then** $\mathcal{M} \leftarrow \{(y, \mathrm{eq}(z, y))\}$;
else
  if $\mathrm{eq}(z, x) + \mathrm{eq}(z, y) = n + \mathrm{tn}(x)$ **then** $\mathcal{M} \leftarrow \{(y, \mathrm{eq}(z, y))\}$;

Algorithm 6: A size-one memory algorithm winning the k-color Mastermind game in $O(n/\log n)$ guesses.

1 **Initialization:** Let $\mathcal{M} \leftarrow \emptyset$;
2 **if** $\mathcal{M} = \emptyset$ **then**
3 $\quad x \leftarrow [0 \dots 0]$;
4 $\quad$ Query $\mathrm{eq}(z, x)$ and update $\mathcal{M} \leftarrow \{(x, \mathrm{eq}(z, x))\}$;
5 **if** $\exists c \in \mathcal{C} : \mathit{suffix}(x) = [cc] \wedge \mathrm{tn}(x) \leq \ell$ **then**
6 $\quad$ `LinAlg`; //find the first ℓ entries $[z_1 \dots z_\ell]$
7 **else if** $\exists c \in \mathcal{C} : \mathit{suffix}(x) = [cc] \wedge \mathrm{tn}(x) = \ell + 1$ **then**
8 $\quad x \leftarrow [\underbrace{0 \dots 0}_{\ell} | \underbrace{0 \dots 0}_{bs} | x_1 \dots x_\ell | \underbrace{0 \quad \dots \quad 0}_{n-(2\ell+bs+\ell_s+2)} | \mathtt{binary}_{\ell_s}(1) | 01]$; //copy prefix (which coincides with the hidden code)
9 $\quad$ Query $\mathrm{eq}(z, x)$ and update $\mathcal{M}$ by replacing $(x, \mathrm{eq}(z, x))$;
10 **else if** $\mathit{suffix}(x) = [01] \wedge i(x) \leq b \wedge q(x) < t + k$ **then**
11 $\quad$ Apply `Sampling`;
12 **else if** $\mathit{suffix}(x) = [01] \wedge i(x) \leq b \wedge q(x) = t + k$ **then**
13 $\quad$ Apply `OptimizeBlock`;
14 **else if** $\mathit{suffix}(x) = [01] \wedge i(x) = b + 1$ **then**
15 $\quad x \leftarrow [x_{\ell+bs+s+1} \dots x_{2\ell+bs+s+1} | x_{\ell+1} \dots x_{\ell+bs} | c \dots c]$ with $c \in \mathcal{C} \backslash \{x_{\ell+bs}\}$ u.a.r.;
16 $\quad$ Query $\mathrm{eq}(z, x)$ and update $\mathcal{M}$ by replacing $(x, \mathrm{eq}(z, x))$; //prepares x for `LinAlg`
17 **else if** $\exists c \in \mathcal{C} : \mathit{suffix}(x) = [cc] \wedge \ell + bs < \mathrm{tn}(x) \leq n - 2$ **then**
18 $\quad$ `LinAlg`;
19 **else if** $\exists c \in \mathcal{C} : \mathit{suffix}(x) = [cc] \wedge \mathrm{tn}(x) = n - 1$ **then**
20 $\quad$ Sample $y \in \{[x_1 \dots x_{n-2} | p] \mid p \in \mathcal{C}^2\} \backslash \{x\}$ uniformly at random;
21 $\quad$ Query $\mathrm{eq}(z, y)$;
22 $\quad$ **if** $\mathrm{eq}(z, y) = n$ **then** $\mathcal{M} \leftarrow \{(y, \mathrm{eq}(z, y))\}$; //hidden code found
23 Go to line 2;

A proof of Lemma 4 and the algorithm `LinAlg` for $k \geq 3$ colors can be found in [4]. Building on `LinAlg`, we can now present Paul's strategy for the one-memory setting and thus prove Theorem 1.

The very rough overview of Paul's strategy is the following. He determines the first $n - \Theta(n/\log n)$ positions using random guessing, where he manages to store the random substrings and Carole's answers in the yet undetermined part of his one string in the memory. As in the proof of Theorem 2, he does so by iteratively determining blocks of length $s := \lceil \sqrt{n} \rceil$. Then, using the linear time strategy from Lemma 4, he determines the missing entries in $O(n/\log n)$ guesses.

To distinguish between the sampling and the linear time phase, Paul uses the last two entries $\mathtt{suffix}(x) := [x_{n-1}x_n]$ of his string x. He has $\mathtt{suffix}(x) = [01]$, when he is in the random guessing phase, and he uses $\mathtt{suffix}(x) = [cc]$ for some $c \in \mathcal{C}$ to indicate that he applies calls to `LinAlg`. Once Paul has determined all but the last two entries (visible from $\mathrm{tn}(x) = n - 1$), he simply needs to sample uniformly at random from the set of all $k^2 - 1$ remaining possible strings. This clearly finds z in a constant expected number of queries.

The total expected number of guesses can be bounded by

$$\overbrace{\tfrac{n-2}{s}(1-\Theta(\log^{-1} n))}^{\text{number of blocks determined in phase 1}} \overbrace{O(\tfrac{s}{\log s})}^{\text{queries needed to determine any such block}} + \overbrace{O(\tfrac{n}{\log n})}^{\text{queries needed in the 2nd phase}} + \overbrace{O(1)}^{\text{phase 3}} = O(\tfrac{n}{\log n}).$$

A non-trivial part is the random guessing phase. As in the proof of Theorem 2, after guessing $t+k$ strings, we want to be able to regain the full guessing history. If we simply stored the random substring and Carole's reply in some unused part of x, then this changed memory would influence Carole's next answer and we would be unable to deduce information on the next guess from it. We solve this difficulty as follows. We store Carole's latest reply (i.e., value $\mathrm{eq}(z,x)$ currently in the memory) and we sample new (random) substrings for the current block at the same time. Here we store the value $\mathrm{eq}(z,x)$ in a part of x for which we know the entries of Carole's hidden code. By this, we can separate in Carole's next answer the influence of the just stored information from the one of the random guess. The precise description of this `Sampling` substrategy can be found in [4].

To gain this storage space where we know the hidden code, we start with another phase, Phase 0, in which we apply the `LinAlg` procedure $O(\log n)$ times until we found the first $\ell := \ell_n + 1$ positions of z (cf. Lemma 4).

The pseudo-code for the size-one memory-restricted strategy winning the Mastermind game with k colors in $O(n/\log n)$ guesses is given in Algorithm 6. Similar to the notation in the proof of Theorem 2, we denote for any $h \in [0..n]$ the binary encoding of length ℓ_n by $\texttt{binary}_{\ell_n}(h)$ and we denote the binary encoding of length $\ell_s := \lceil \log s \rceil + 1$ by $\texttt{binary}_{\ell_s}(h)$. The current block of interest $i(x)$ is encoded in positions $\{n-\ell_s-1,\dots,n-2\}$, i.e., we have $i(x) := \sum_{h=0}^{\ell_s-1} 2^h x_{n-2-h}$ and $B_{i(x)} := \{\ell + (i(x)-1)s + 1, \dots, \ell + i(x)s\}$. The total number of blocks which we determine via random guessing is $b := \lfloor \frac{n-2}{s}(1 - \frac{K}{\log n}) \rfloor$ for some suitable large constant K. The number of random guesses for each block is $t := \lceil (2+\varepsilon)\frac{s(1+2\log k)}{\log s - \log k} \rceil$ for some arbitrarily small constant $\varepsilon > 0$. Lastly, the actual number of already sampled guesses for block $B_{i(x)}$ is denoted by $q(x)$. As discussed in the proof of Theorem 2, $q(x)$ can be computed via the largest position $p_1 < n-2-\ell_s$ with $x_{p_1} = 1$. Note that the `Sampling` routine described above implicitly updates the counter $q(x)$ by changing the p_1 value. The `OptimizeBlock` routine determines $\texttt{BLOCK}_{i(x)}(z)$, stores it in $B_{i(x)}$ and increases the block counter $i(x)$ by one.

To end this proof sketch, let us show that our memory cell has enough storage capacity to store all $t+k$ substrings, the values $\texttt{binary}_{\ell_n}(\mathrm{eq}(z,x))$, and the values $\texttt{binary}_{\ell_s}(\Delta_{i(v)}(v))$. We have $n-2-(2\ell+bs) = n - n(1-K/\log n) - O(\log n) = Kn/\log n - O(\log n)$ positions for storing information and the total number of positions needed for storing the $t+k$ sample informations is

$$(t+k)(s+O(\log n)) = \Theta(n/\log n) + o(n/\log n) < Kn/\log n - O(\log n)$$

for constant, but sufficiently large K.

Acknowledgment

Carola Winzen is a recipient of the Google Europe Fellowship in Randomized Algorithms. This research is supported in part by this Google Fellowship.

References

1 Gautham Anil and R. Paul Wiegand. Black-box search by elimination of fitness functions. In *Proceedings of the 10th ACM Workshop on Foundations of Genetic Algorithms (FOGA'09)*, pages 67–78. ACM, 2009.

2 Zhixiang Chen, Carlos Cunha, and Steven Homer. Finding a hidden code by asking questions. In *Proceedings of the 2nd Annual International Conference on Computing and Combinatorics (COCOON'96)*, pages 50–55. Springer, 1996.

3 Vasek Chvátal. Mastermind. *Combinatorica*, 3:325–329, 1983.

4 Benjamin Doerr and Carola Winzen. Playing mastermind with constant-size memory. *CoRR*, abs/1110.3619, 2011.

5 Stefan Droste, Thomas Jansen, and Ingo Wegener. Upper and lower bounds for randomized search heuristics in black-box optimization. *Theory of Computing Systems*, 39:525–544, 2006.

6 Paul Erdős and Alfréd Rényi. On two problems of information theory. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 8:229–243, 1963.

7 Michael T. Goodrich. On the algorithmic complexity of the mastermind game with black-peg results. *Information Processing Letters*, 109:675–678, 2009.

8 Donald E. Knuth. The computer as a master mind. *Journal of Recreational Mathematics*, 9:1–5, 1977.

9 Jeff Stuckman and Guo-Qiang Zhang. Mastermind is NP-complete. *INFOCOMP Journal of Computer Science*, 5:25–28, 2006.