

Playing Mastermind with Constant-Size Memory

Benjamin Doerr · Carola Winzen

Published online: 11 November 2012
© Springer Science+Business Media New York 2012

Abstract We analyze the classic board game of Mastermind with n holes and a constant number of colors. The classic result of Chvátal (Combinatorica 3:325–329, 1983) 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 Comput. Syst. 39:525–544, 2006) on the memory-restricted black-box complexity of the OneMax function class.

Keywords Mastermind · Query complexity · Memory-restricted algorithms

1 Introduction

The original *Mastermind* game is a board game for two players invented in the seventies by Mordecai Meierowitz. 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 the *codemaker*, called *Carole* in the following. He does so by guessing color combinations and receiving information on how close his 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 $(\text{eq}(z, x), \pi(z, x))$ of numbers. The first number, $\text{eq}(z, x)$, which is usually indicated by black answer-pegs, is the number of positions in which Paul's and Carole's

This is the full version of [6].

B. Doerr · C. Winzen

D1: Algorithms and Complexity, Max Planck Institute for Informatics, Campus E1 4,
66123 Saarbrücken, Germany

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 $\text{eq}(z, x) := |\{i \in [4] \mid z_i = x_i\}|$ and $\pi(z, x) := \max_{\rho \in S_4} |\{i \in [4] \mid z_i = x_{\rho(i)}\}| - \text{eq}(z, x)$, where S_4 denotes the set of all permutations of the set $[4]$. 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 has produced a plethora of results on the Mastermind problem. For the original game with 6 colors and 4 positions, Knuth [10] 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] studied 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 proved that for any arbitrarily small constant $\varepsilon > 0$ and for $k < n^{1-\varepsilon}$ colors we have $d(n, k) = O(\frac{n \log k}{\log n - \log k})$. More precisely, he showed that for any $\varepsilon > 0$, n sufficiently large and for any $k \leq n^{1-\varepsilon}$, a number of $(2 + \varepsilon) \frac{n(1+2 \log k)}{\log n - \log k}$ guesses chosen from $[k]^n$ independently and uniformly at random, with high probability, leads to a different sequence of answers for each possible code, which implies that the answers uniquely determine the secret code. In particular, this probabilistic construction shows the existence of such a sequence of guesses, and thus, of a strategy determining the secret code with $(2 + \varepsilon) \frac{n(1+2 \log k)}{\log n - \log k}$ 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 $\text{eq}(z, x)$; the number of bits in which her and Paul’s string coincide.

For larger values of k , the following is known. For $n \leq k \leq n^2$, Chvátal proved $d(n, k) \leq 2n \log k + 4n$ and for $k = \omega(n^2 \log n)$ he showed $(k - 1)/n \leq d(n, k) \leq \lceil k/n \rceil + d(n, n^2)$. These results have subsequently been improved. Chen, Cunha, and Homer [2] showed that $d(n, k) \leq 2n \lceil \log n \rceil + 2n + \lceil k/n \rceil + 2$ for $k \geq n$. Goodrich [8] proved $d(n, k) \leq n \lceil \log k \rceil + \lceil (2 - 1/k)n \rceil + k$ for arbitrary k . This was again improved by Jäger and Peczarski [9], who showed an upper bound of $n \lceil \log n \rceil - n + k + 1$ for the case $k > n$ and $n \lceil \log k \rceil + k$ for the case $k \leq n$.

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 [7] showed that $d(n, 2) = \Theta(n/\log n)$.

Concerning the computational complexity, Stuckman and Zhang [12] showed that it is \mathcal{NP} -hard to decide whether a given sequence $(x^{(i)}, (\text{eq}^{(i)}, \pi^{(i)}))_{i=1}^t$ of queries $x^{(i)}$ and answers $(\text{eq}^{(i)}, \pi^{(i)})$ of black and white answer-pegs has a secret code leading to these answers, i.e., whether there exists a string $z \in [k]^n$ such that $\text{eq}(z, x^{(i)}) = \text{eq}^{(i)}$ and $\pi(z, x^{(i)}) = \pi^{(i)}$ for all $i \in [t]$. Goodrich [8] proved that this is already \mathcal{NP} -hard if we only ask for consistence with the black answer-peg replies $\text{eq}^{(i)}$.

Our Results Originally motivated by a conjecture on black-box complexities (cf. Sect. 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.

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 separate iteration counters and boolean variables (possibly hidden as case distinctions) may be used. 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 does not make Paul's life easier. The $O(n/\log n)$ strategies by Erdős and Rényi [7] 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 suffices.

Theorem 1 *Let $k \in \mathbb{N}_{\geq 2}$. 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 she only reveals the number of fully correct pegs $\text{eq}(x, z)$ ("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. [7] or [3]. Our result disproves a conjecture of Droste, Jansen, and Wegener [4], 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. Sect. 3. The proof of Theorem 1 is given in Sect. 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,

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

- 1 **Initialization:** Sample $x^{(0)}$ according to some probability distribution $p^{(0)}$ on \mathcal{S} ;
 - 2 Query $f(x^{(0)})$;
 - 3 **for** $t = 1, 2, 3, \dots$ **do**
 - 4 Depending on $((x^{(0)}, f(x^{(0)})), \dots, (x^{(t-1)}, f(x^{(t-1)})))$ choose a probability distribution $p^{(t)}$ on \mathcal{S} and sample $x^{(t)}$ according to $p^{(t)}$;
 - 5 Query $f(x^{(t)})$;
-

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 [4].

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 $\mathcal{S} \rightarrow \mathbb{R}$.

For such an algorithm A and a function $f : \mathcal{S} \rightarrow \mathbb{R}$, 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 $\mathcal{S} \rightarrow \mathbb{R}$, 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 $\mathcal{S} \rightarrow \mathbb{R}$. 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} . In fact, it is a lower bound for the efficiency of any randomized black-box algorithm. Unfortunately, often this lower bound is not very useful. For example, Droste, Jansen, and Wegener [4] observed that the \mathcal{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 suggested 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, the Metropolis algorithm or simulated annealing (see, e.g., [13] and the references therein) 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 information other 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 $(x^{(\mu+1)}, f(x^{(\mu+1)}))$.

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

```

1 Initialization:  $\mathcal{M} \leftarrow \emptyset$ ;
2 for  $t = 1, 2, \dots$  do
3   Depending (only) on  $\mathcal{M}$  choose a probability distribution  $p$  on  $\mathcal{S}$  and
   sample  $x^{(\mu+1)}$  according to  $p$ ; //variation step
4   Query  $f(x^{(\mu+1)})$ ;
5   Select  $\mathcal{M} \subseteq \mathcal{M} \cup \{(x^{(\mu+1)}, f(x^{(\mu+1)}))\}$  of size  $|\mathcal{M}| \leq \mu$ ; //selection step

```

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

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 ONEMAX_n . This stems from the fact in a typical run of such an algorithm, new solutions are obtained from flipping mostly a constant number of bits in an existing solution. By the coupon collector theorem (see, e.g., [11]), $\Omega(n \log n)$ such modifications are necessary to ensure that each bit is touched at most once.

As a moments thought reveals, black-box algorithms optimizing 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 ONEMAX_n is $\Theta(n/\log n)$ by the results of Erdős and Rényi [7] and Chvátal [3].

This connection was apparently overlooked so far in the randomized search heuristics community, where Droste, Jansen, and Wegener [4] proved an upper bound of $O(n)$ and later Anil and Wiegand [1] proved 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 suggested to investigate the memory-restricted black-box complexity of ONEMAX_n . They conjectured in [4, Sect. 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 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 ONEMAX_n even with the memory restricted to one is $\Theta(n/\log n)$. This disproves the conjecture of Droste, Jansen, and Wegener.

From the view-point of building a useful complexity theory for randomized search heuristics, Theorem 1 indicates that a memory restriction alone does not suffice to overcome the drawbacks of the unrestricted black-box model.

3 The Mastermind Game with Memory of Size Two

Since the proof of Theorem 1 is quite technical, we shall 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 $C := [0..k-1] := \{0, 1, \dots, k-1\}$. 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 version of 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.*

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 [7] showed that there is a $t = \Theta(n/\log n)$ 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 C^n independently and uniformly at random. Then for all $z \in C^n$, the set*

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

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

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 $y \in C^n$ and let $B \subseteq [n]$ be a block (i.e., an interval) of size $s := \lceil \sqrt{n} \rceil$. As we shall see, by $t = O(s/\log s)$ 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 $\ell_n = O(\log n)$ entries of the string. Hence the t guesses and answers can be memorized using a

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

```

total number of $t(s + \ell_n) = O(n/\log n)$ positions. Let us remark that we do need to query the scores $\text{eq}(z, x)$ of the storage strings only in order to be able to add the pair $(x, \text{eq}(z, x))$ to the memory \mathcal{M} (here and henceforth we denote by \mathcal{M} the memory, which is a set containing pairs $(x, \text{eq}(z, x))$ of a previous query x and its answer $\text{eq}(z, x)$). The answers $\text{eq}(z, x)$ themselves, however, are irrelevant to us.

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.

Before we argue for the correctness of Algorithm 3, let us fix the notation. For a 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

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

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

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

the string x with the i th block substituted by r . Similarly, let $\text{substitute}(y, \{n\}, c) := [y_1 \dots y_{n-1} |c]$. 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 the n th 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 $\text{eq}(z, y) \in [0..n]$ in binary, using $\ell_n := \lceil \log n \rceil + 1$ positions, and we denote this binary encoding of length ℓ_n by $\text{binary}_{\ell_n}(\text{eq}(z, y))$. By $\Delta_i(y)$ we denote the contribution of the i th block to the value $\text{eq}(z, y)$; 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, $\Delta_i(y) := \text{eq}(z|_{B_i}, y|_{B_i})$. Lastly, let $\mathcal{S}_i^{\text{consistent}}$ be the set of strings w of length $|B_i|$ such that $\text{substitute}(z, B_i, w)$ 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 $x^c = \text{substitute}(y, B_i, [c \dots c])$ and we stored these *reference strings* together with Carole’s replies $\text{eq}(z, x^c) = \sum_{h=1}^{\ell_n} 2^{h-1} x_{c(s+\ell_n+1)-h}$, $c \in [0..k - 1]$. 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 $y^{(j)} = \text{substitute}(y, B_i, r^{(j)})$. We have stored Carole’s reply to this guess in binary, and we can infer $\text{eq}(z, y^{(j)}) = \sum_{h=1}^{\ell_n} 2^{h-1} x_{(k+j)(s+\ell_n+1)-h}$. 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 $\text{eq}(z, x^0) + \dots + \text{eq}(z, x^k)$. Formally, $\sum_{c=0}^{k-1} \text{eq}([z_{(i-1)s+1} \dots z_{\min\{is,n-1\}}], [c \dots c]) = |B_i|$. Since all other positions of the sampling string y are not changed during the phase in which we determine the i th block, each of them either contributes 0 to the sum $\sum_{c=0}^{k-1} \text{eq}(z, x^c)$, or it contributes k . We thus infer that

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

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

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

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 $y = \text{substitute}(y, B_i, w)$ 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 alone the state of the algorithm. We define a function $\text{Part}(y, x)$ which equals 1 if the information of y has been added to the storage string x already and which equals 0 otherwise. That is, we set

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

Note that $\text{Part}(y, x) = 1$ 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 repetition. 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.

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

```

4 Reducing the Memory to Size One

Compared to the situation in Sect. 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.

4.1 Linear Time Strategies

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 we set, for all $x \in \mathcal{C}^n$,

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

the *tail number* of x . The way to make progress from x is to randomly change the $\text{tn}(x)$ th entry of x to a random other value, or to change the entries in positions $\text{tn}(x) + 1$ to n to the same random other value. The latter is a successful move if $x_{\text{tn}(x)}$ is already correct, but is not marked as determined by being different from the other tail entries.

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 $\text{tn}(x) - 1$ entries of z have been determined (i.e., Carole can no longer change the entries of $[z_1 \dots z_{\text{tn}(x)-1}]$). Further assume that $x_i = z_i$ for all $i < \text{tn}(x)$ and that $\mathcal{M} = \{(x, \text{eq}(z, x))\}$ is the current content of the memory cell.*

There is a size-one memory-restricted guessing procedure `NextEntry` 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 $i \leq \text{tn}(x)$ and $\text{tn}(y) = \text{tn}(x) + 1$. Every call of `NextEntry` requires only one guess.

Interestingly, for the definition of `NextEntry`, we need to distinguish between the cases of $k = 2$ and $k \geq 3$ colors, as certain arguments exploit particular properties of these cases. For $k = 2$ colors and Carole not playing a devil's strategy but choosing a random secret $z \in \{0, 1\}^n$, we have analyzed this algorithm already in [5].

4.1.1 The Case of $k = 2$ Colors $\mathcal{C} = \{0, 1\}$

In this section we prove that for $k = 2$ colors $\mathcal{C} = \{0, 1\}$, `NextEntry` is a procedure that requires, in expectation, three calls to modify the memory content by replacing the current string x that is assumed to satisfy the conditions of Lemma 4, by a string y with $\text{tn}(y) = \text{tn}(x) + 1$ and $y_i = z_i$ for all $i \leq \text{tn}(x)$.

For all $i \in [n]$ let e_i^n be the i th unit vector of length n .

Algorithm 5: Routine NextEntry for $k = 2$ colors

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

Proposition 5 For $k = 2$ colors, Algorithm 5 verifies Lemma 4. In expectation, three calls to routine NextEntry suffice.

Proof Let $x \in \{0, 1\}^n$ be a bit string with $\text{tn}(x) < n$ and $x_i = z_i$ for all $i < \text{tn}(x)$.

Algorithm 5 samples with probability 1/2 the string $y = x \oplus e_{\text{tn}(x)}^n$, and with probability 1/2 it samples $y = x \oplus \sum_{i=\text{tn}(x)+1}^n e_i^n$. That is, either it flips only the $\text{tn}(x)$ th bit of x or it flips all “tail bits” but the $\text{tn}(x)$ th one.

If $y = x \oplus e_{\text{tn}(x)}^n$, clearly we have $z_{\text{tn}(x)} = y_{\text{tn}(x)}$ if and only if $\text{eq}(z, y) > \text{eq}(z, x)$.

Therefore, let us assume that Algorithm 5 samples $y = x \oplus \sum_{i=\text{tn}(x)+1}^n e_i^n$. We show that $z_{\text{tn}(x)} = y_{\text{tn}(x)} (= x_{\text{tn}(x)})$ holds if and only if $\text{eq}(z, x) + \text{eq}(z, y) = n + \text{tn}(x)$. By definition we have $y_i = x_i = z_i$ for all $i < \text{tn}(x)$. Thus, the first $\text{tn}(x) - 1$ bits of x and y contribute $2(\text{tn}(x) - 1)$ to the sum $\text{eq}(z, x) + \text{eq}(z, y)$; formally,

$$\begin{aligned} & \text{eq}([z_1 \dots z_{\text{tn}(x)-1}], [x_1 \dots x_{\text{tn}(x)-1}]) + \text{eq}([z_1 \dots z_{\text{tn}(x)-1}], [y_1 \dots y_{\text{tn}(x)-1}]) \\ &= 2(\text{tn}(x) - 1). \end{aligned}$$

On the other hand, for all $i > \text{tn}(x)$ either have $z_i = x_i$ or $z_i = 1 - x_i = y_i$. Thus, the last $n - \text{tn}(x)$ bits of x and y contribute exactly $n - \text{tn}(x)$ to the sum $\text{eq}(z, x) + \text{eq}(z, y)$; formally,

$$\begin{aligned} & \text{eq}([z_{\text{tn}(x)+1} \dots z_n], [x_{\text{tn}(x)+1} \dots x_n]) + \text{eq}([z_{\text{tn}(x)+1} \dots z_n], [y_{\text{tn}(x)+1} \dots y_n]) \\ &= n - \text{tn}(x). \end{aligned}$$

By definition we also have $y_{\text{tn}(x)} = x_{\text{tn}(x)}$ and, thus,

$$\begin{aligned} & \text{eq}(z, x) + \text{eq}(z, y) \\ &= \text{eq}([z_1 \dots z_{\text{tn}(x)-1}], [x_1 \dots x_{\text{tn}(x)-1}]) \\ &\quad + \text{eq}(z_{\text{tn}(x)}, x_{\text{tn}(x)}) + \text{eq}([z_{\text{tn}(x)+1} \dots z_n], [x_{\text{tn}(x)+1} \dots x_n]) \\ &\quad + \text{eq}([z_1 \dots z_{\text{tn}(x)-1}], [y_1 \dots y_{\text{tn}(x)-1}]) \\ &\quad + \text{eq}(z_{\text{tn}(x)}, x_{\text{tn}(x)}) + \text{eq}([z_{\text{tn}(x)+1} \dots z_n], [y_{\text{tn}(x)+1} \dots y_n]) \\ &= 2(\text{tn}(x) - 1) + n - \text{tn}(x) + 2 \text{eq}(z_{\text{tn}(x)}, x_{\text{tn}(x)}) \\ &= n + \text{tn}(x) + 2 \text{eq}(z_{\text{tn}(x)}, x_{\text{tn}(x)}) - 2 \end{aligned}$$

This shows that $\text{eq}(z, x) + \text{eq}(z, y) = n + \text{tn}(x)$ if and only if $\text{eq}(z_{\text{tn}(x)}, x_{\text{tn}(x)}) = 1$; i.e., if and only if $z_{\text{tn}(x)} = x_{\text{tn}(x)} (= y_{\text{tn}(x)})$.

It is immediate that for a secret code z taken from $\{0, 1\}^n$ uniformly at random, the probability to obtain, in one call of `NextEntry`, a string y with $\text{tn}(y) = \text{tn}(x) + 1$ and $y_i = z_i$ for all $i < \text{tn}(y)$ is $1/2$. This shows that, if Carole does not play a devil’s strategy and if her string is taken from $\{0, 1\}^n$ uniformly at random, we need, on average, two successive calls to procedure `NextEntry` until we obtain a string y as desired.

Proposition 5 follows from the easy observation that it takes, on average, three iterations until both $y = x \oplus e''_{\text{tn}(x)}$, and $y = x \oplus \sum_{i=\text{tn}(x)+1}^n e_i^n$ have been sampled. That is, even if Carole plays a devil’s strategy, three calls of Algorithm 5, on average, force her to accept one entry $z_{\text{tn}(x)} \in \{0, 1\}$. □

To win the two-color Mastermind game in a linear number of guesses, Paul may just start with querying the string $[0, \dots, 0]$ and then calling Algorithm 5 sufficiently often.

4.1.2 The Case of $k \geq 3$ Colors $\mathcal{C} = [0..k - 1]$

The main argument of Proposition 5, namely that $\sum_{c=0}^{k-1} \text{eq}(z, [c \dots c]) = n$, seems hard to extend to more than two colors with no additional memory. However, having more than two colors can be exploited in a different way as it gives more than one possibility to mark the *tail* $[x_{\text{tn}(x)} \dots x_n]$ of a search point x .

Proposition 6 *For $k \geq 3$ colors, Algorithm 6 satisfies the claims of Lemma 4.*

Proof Let $x \in \mathcal{C}^n$ with $\text{tn}(x) < n$ and $x_i = z_i$ for all $i < \text{tn}(x)$. If $y = [x_1 \dots x_{\text{tn}(x)-1} | j | x_{\text{tn}(x)+1} \dots x_n]$, then clearly we have $\text{eq}(z, y) > \text{eq}(z, x)$ if and only if $y_{\text{tn}(x)} = j = z_{\text{tn}(x)}$. Therefore, all we need to show is that, using the strategy of Algorithm 6, it takes a constant number of guesses until for each $j \in \mathcal{C}$ there exists an $i_j \in \mathcal{C} \setminus \{j\}$ such that

Algorithm 6: Routine `NextEntry` for $k \geq 3$ colors

- 1 **Assumption:** The string $x \in \mathcal{C}^n$ in the memory satisfies $\text{tn}(x) < n$ and $x_i = z_i$ for all $i < \text{tn}(x)$;
 - 2 With probability $(k - 1)/k$ sample $y \in \{[x_1 \dots x_{\text{tn}(x)-1} | j | x_{\text{tn}(x)+1} \dots x_n] \mid j \in \mathcal{C} \setminus \{x_{\text{tn}(x)}\}\}$ uniformly at random and with probability $1/k$ sample $y \in \{[x_1 \dots x_{\text{tn}(x)-1} | j \dots j] \mid j \in \mathcal{C} \setminus \{x_{\text{tn}(x)-1}\}\}$ uniformly at random;
 - 3 Query $\text{eq}(z, y)$;
 - 4 **if** $y = [x_1 \dots x_{\text{tn}(x)-1} | j | x_{\text{tn}(x)+1} \dots x_n]$, $j \neq x_{\text{tn}(x)}$ **then**
 - 5 **if** $\text{eq}(z, y) > \text{eq}(z, x)$ **then** $\mathcal{M} \leftarrow \{(y, \text{eq}(z, y))\}$; $//z_{\text{tn}(x)} = j$
 - 6 **else**
 - 7 $\mathcal{M} \leftarrow \{(y, \text{eq}(z, y))\}$;
-

we have queried both $x = [z_1 \dots z_{\text{tn}(x)-1} | i_j \dots i_j]$ and $y = [z_1 \dots z_{\text{tn}(x)-1} | j | i_j \dots i_j]$ in two subsequent guesses. This follows essentially from the fact that k is constant.

More precisely—regardless of the current search point x —for any bitstring $y = [z_1 \dots z_{\text{tn}(x)-1} | j | i_j \dots i_j]$ the probability to sample y in the second of two subsequent calls to Algorithm 6 is constant. Therefore, the expected number of calls to Algorithm 6 until y is sampled is constant. The claim follows by the linearity of expectation. \square

4.2 Proof of Theorem 1

Building on NextEntry and the block-wise random guessing strategy introduced in Sect. 3, we can now prove Theorem 1. That is, we present Paul’s $O(n/\log n)$ winning strategy for the setting with one single memory cell.

Proof of Theorem 1 The structure of this proof is as follows. First we sketch the main ideas and give a high-level pseudo-code for the size-one memory-restricted strategy winning the black answer-peg only Mastermind game with k colors in $O(n/\log n)$ guesses. After fixing some notation, we then present more details for the different phases, in particular for the random guessing phase, which is the most critical part of this proof. We present here the details of Paul’s strategy for the case of $k = 2$ colors. The generalization to $k \geq 3$ colors is pretty much straightforward. Some remarks on the differences between the case of $k = 2$ and $k \geq 3$ colors are given at the end of this proof.

Let us begin with the rough overview of Paul’s strategy. 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 $s := \lceil \sqrt{n} \rceil$. The number of blocks determined this way will be $b := \lfloor \frac{n-2}{s} (1 - \frac{K}{\log n}) \rfloor$, where K is a suitably large constant.

This is the first phase of the strategy. In the second phase, using the linear time strategy from Lemma 4, he determines the missing $\Theta(n/\log n)$ entries in $O(n/\log n)$ guesses.

To distinguish between the sampling and the linear time phase, Paul uses the last two entries $\text{suffix}(x) := [x_{n-1}x_n]$ of his string x . He has $\text{suffix}(x) = [01]$ when he is in the random guessing phase, and he uses $\text{suffix}(x) = [cc]$ for some $c \in \mathcal{C}$ to indicate that he applies calls to NextEntry. Once Paul has determined all but the last two entries (visible from $\text{tn}(x) = n - 1$), he simply needs to sample uniformly at random from the set of all $k^2 - 1$ remaining possible strings. This clearly determines z in a constant expected number of additional queries (phase 3).

The total expected number of guesses can be bounded by

$$\underbrace{\frac{n-2}{s} (1 - \Theta(\log^{-1} n))}_{\text{number of blocks determined in phase 1}} \quad \underbrace{O\left(\frac{s}{\log s}\right)}_{\text{queries needed to determine any such block}} \quad + \quad \underbrace{O\left(\frac{n}{\log n}\right)}_{\text{queries needed in phase 2}} \quad + \quad \underbrace{O(1)}_{\text{queries needed in phase 3}} \quad = O\left(\frac{n}{\log n}\right).$$

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 $\text{eq}(z, x)$ currently in the memory) and we sample new (random) substrings for the current block at the same time. Here we store the value $\text{eq}(z, x)$ 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 strategy is presented below.

To gain the storage space, for which we know the hidden code, we need to add another phase, phase 0, in which we apply $O(\log n)$ calls to the `NextEntry` procedure until we have determined the first $\ell := \ell_n + 1$ positions of z (cf. Lemma 4). This does not change the overall asymptotic number of queries Paul needs to win the game.

The pseudo-code for this size-one memory-restricted strategy is given in Algorithm 7. Similar to the notation in the proof of Theorem 2, we denote for any $h \in [0..n]$ its binary encoding of length ℓ_n by $\text{binary}_{\ell_n}(h)$ and for $h \in [0..s]$ we denote its binary encoding of length $\ell_s := \lceil \log s \rceil + 1$ by $\text{binary}_{\ell_s}(h)$. 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 $B_{i(x)} := \{\ell + (i(x) - 1)s + 1, \dots, \ell + i(x)s\}$, and, consequently, $\text{BLOCK}_{i(x)}(x) := [x_{\ell+(i(x)-1)s+1} \dots x_{\ell+i(x)s}]$. The number of random guesses for each block is $t := \lceil (2 + \varepsilon) \frac{s(1+2\log k)}{\log s - \log k} \rceil$ where $\varepsilon > 0$ is an arbitrarily small constant. Lastly, the actual number of already sampled guesses for block $B_{i(x)}$ is denoted by $q(x)$. As in the proof of Theorem 2, $q(x)$ can be computed via $p_1(x) := \max\{i \in [n - \ell_s - 3] \mid x_i = 1\}$, the largest position $i < n - 2 - \ell_s$ with entry $x_i = 1$. Details on how $q(x)$ can be computed are given in the description of the `OptimizeBlock` routine, which, after t random samples have been sampled via the `Sampling` routine, determines $\text{BLOCK}_{i(x)}(z)$, stores it in $B_{i(x)}$, and increases the block counter $i(x)$ by one.

Let us now present a more detailed description of Algorithm 7.

As in the proof of Theorem 2 let us assume that Carole has chosen a fix code $z \in \mathcal{C}^n$, which she does not change during the game; i.e., to any of Paul’s guesses x she replies $\text{eq}(z, x)$. By adopting a worst-case view below, we implicitly still allow Carole to change z as long as the new choice is consistent with all previous replies.

If in any iteration we find an x with $\text{eq}(z, x) = n$, we have $x = z$ and all we need to do is to output x . Thus, in what follows we always assume $\text{eq}(z, x) < n$.

Initialization of Algorithm 7, Lines 1–4 For initialization, Paul picks a $c \in \mathcal{C}$ uniformly at random and guesses the all-“ c ”s string of length n , $x = [c \dots c]$. He updates the memory $\mathcal{M} \leftarrow \{(x, \text{eq}(z, x))\}$ accordingly. This memory satisfies all conditions of line 1 of routine `NextEntry` (Algorithms 5 and 6) with $\text{tn}(x) = 1$.

Phase 0 of Algorithm 7, Lines 5–6 To this string, Paul applies successive calls to the routine `NextEntry`. By Lemma 4 he finds a string $y \in \mathcal{C}^n$ with $y_i = z_i, i \leq \ell$, and $\text{tn}(y) = \ell + 1$ in an expected number of $O(\ell)$ guesses. As mentioned above, Paul

Algorithm 7: A size-one memory-restricted algorithm winning the k -color black answer-peg only Mastermind game in $O(n/\log n)$ guesses.

```

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

```

runs this first phase until he has determined the first

$$\ell = \lceil \log n \rceil + 2 = \ell_n + 1$$

entries. This is the number

$$\ell_n = \lceil \log n \rceil + 1$$

of entries needed to store in binary any integer value $h \in [0..n]$ plus 1. These bits shall be used in phase 1 of Algorithm 7 to indicate the status of the Sampling routine (first position) and for storing Carole’s latest reply $\text{eq}(z, x) \in [0..n]$ (positions $\{2, \dots, \ell\}$). We shall describe this in more detail below.

First Phase of Algorithm 7, Lines 7–13 After Paul has determined the first ℓ entries, he first needs to prepare the string for the random guessing phase. This is done in lines 7–9. Since we want to use the first ℓ entries to store reference values, we need to make a copy of the prefix (which, by construction, coincides with Carole’s hidden code). To this end, we query in line 8 the string

$$y = \left[\underbrace{0 \dots 0}_{\ell + bs \text{ entries}} \mid x_1 \dots x_\ell \mid \underbrace{0 \dots 0}_{n - (2\ell + bs + \ell_s + 2) \text{ entries}} \mid \underbrace{\text{binary}_{\ell_s}(1) \mid 01}_{\ell_s \text{ entries}} \right],$$

where x is the string that is currently in the memory (i.e., the string we obtained through phase 0)¹ and $bs = n - \Theta(n/\log n)$ is the number of positions Paul determines using random guessing. As mentioned in the overview, the last two entries $\text{suffix}(x) = [x_{n-1}x_n] = [01]$ indicate that we are entering the second phase. In positions $\{n - \ell_s - 1, \dots, n - 2\}$ we indicate in binary the block which we are currently trying to determine. That is, whenever x is the current string in the memory with $\text{suffix}(x) = [01]$, then the block currently of interest is $B_{i(x)}$ with $i(x) = \sum_{i=0}^{\ell_s-1} 2^i x_{n-2-i}$. We initialize $i(x) = 1$.

After guessing y and updating the memory by replacing the current one with $\{(y, \text{eq}(z, y))\}$, Paul enters the first phase (as indicated by $\text{suffix}(x)$). The overall expected number of queries needed until this point is $O(\ell) = O(\log n)$.

After lines 7–9 have been executed, Paul determines all but $n - \Theta(n/\log n)$ entries by iteratively determining blocks of length $s = \lceil \sqrt{n} \rceil$ via random guessing. In total, he determines $b = \lfloor \frac{n}{s} (1 - \frac{K}{\log n}) \rfloor$ such blocks in this phase. The description of the routines `Sampling` (in which k reference strings and t random samples are queried for the $i(x)$ th block $B_{i(x)}$) and `OptimizeBlock` (in which we use the reference strings and the random guesses to determine $\text{BLOCK}_{i(x)}(z)$, the $i(x)$ th block of the secret code z) is quite technical. We present the details after the description of the remaining phases.

Second Phase of Algorithm 7, Lines 14–18 In the second phase of Algorithm 7 we again apply successive calls to routine `NextEntry` to determine all but the last two remaining entries. To this end, we first need to prepare the string. This is done in lines 14–16 of Algorithm 7. It follows from the correctness of the first phase that the string x queried in line 16 satisfies $x_i = z_i$ for all $1 \leq i \leq \ell + bs$. And, by definition, it also satisfies $x_{\text{tn}(x)-1} \neq x_{\text{tn}(x)}$ with $\text{tn}(x) = \ell + bs + 1$.

From Lemma 4 we infer that via routine `NextEntry` we find a string x with $\text{tn}(x) = n - 1$ and $x_i = z_i$ for all $1 \leq i \leq n - 2$ in an expected number of $O(n - 2 - (\ell + bs)) = O(n/\log n)$ queries. These are lines 17 and 18 of Algorithm 7.

Third Phase of Algorithm 7, Lines 19–22 Similarly to the last step of Algorithm 4, all we need to do in the last phase of Algorithm 7 is to determine the last two entries. This is done by sampling y uniformly at random from the set of possible target strings

$$\{[x_1 \dots x_{n-2} \mid p] \mid p \in \{0, 1\}^2 \setminus \{x\}\}$$

¹That is, we have $\text{tn}(x) = \ell + 1, \forall i \leq \ell : x_i = z_i$, and $\exists c \in \mathcal{C} \setminus \{x_\ell\} \forall i \geq \text{tn}(x) : x_i = c$.

and we find $y = z$ after a constant expected number of queries. This phase is recognized by the algorithm by the fact that $\text{tn}(x) = n - 1$. Note that we have $\text{tn}(x) \leq n - 2$ in the `NextEntry` phases—phases 0 and 2—and that we have $\text{tn}(x) = n$ in phase 1.

Summing up the expected number of queries needed for each phase, we have shown that Paul needs, on average,

$$O(\log n) + O(n/\log n) + O(n/\log n) + O(1) = O(n/\log n)$$

queries until he has identified Carole’s hidden code z .

In the remainder of this proof we present the details of the first phase of Algorithm 7, the random sampling routine `Sampling`, and the `OptimizeBlock` routine. As mentioned above, this description requires some technicalities. Therefore, we split it into the following parts:

In Part I we present the general structure of the guesses Paul queries in the sampling phase. Here, we shall also show that the n positions are indeed sufficient to store, for any of the b blocks of length s , all necessary information about the samples.

Part II provides further notation used in the pseudo-code of Algorithm 8.

In Part III we show how the contributions $\Delta_{i(x)}(r) \in [0..s]$ of the random samples $r \in \mathcal{C}^s$ can be computed solely from the content in the memory. This also shows that indeed after sampling the t random guesses for the current block of interest, it is possible to regain the full query history using only the information that has been stored in the memory.

We conclude the description of phase 1 in Part IV, where we explain how the memory is being updated once the entries $\text{BLOCK}_{i(x)}(z)$ of the secret code z in the $i(x)$ th block have been determined.

Part I The general structure of a random query x for determining block $B_{i(x)}$ is the following.

$$\begin{aligned}
 x = & \left[\underbrace{x_1}_{(1)} \mid \underbrace{\text{binary}_{\ell_n}(\text{eq}(z, y))}_{(2)} \mid \underbrace{\text{opt}(B_1) \dots \text{opt}(B_{i(x)-1})}_{(3)} \mid \underbrace{r}_{(4)} \mid \underbrace{0 \dots 0}_{(5)} \mid \underbrace{z_1 \dots z_\ell}_{(6)} \right. \\
 & \underbrace{\text{binary}_{\ell_n}(\text{eq}(z, x^0)) \mid \text{binary}_{\ell_n}(\text{eq}(z, x^1)) \mid 1}_{(7)} \\
 & \left. \underbrace{\text{binary}_{\ell_n}(\text{eq}(z, \text{ref}^{(1)})) \mid r^{(1)} \mid \Delta_{i(x)}(r^{(1)}) \mid 1 \dots \text{binary}_{\ell_n}(\text{eq}(z, \text{ref}^{(t')}) \mid r^{(t')} \mid \Delta_{i(x)}(r^{(t')}) \mid 1}_{(8)} \right] \\
 & \left. \underbrace{0 \dots 0}_{(9)} \mid \underbrace{\text{binary}_{\ell_s}(i(x))}_{(9)} \mid \underbrace{01}_{(10)} \right], \tag{1}
 \end{aligned}$$

where we use

- (1) the first entry $x_1 \in \{0, 1\}$ to indicate whether we are sampling a new random substring ($x_1 = 1$) or whether we are doing a storage operation by which we add

to x all necessary information from the previous guess ($x_1 = 0$). An explanation of these operations follows below;

- (2) ℓ_n entries for encoding the value $\text{eq}(z, y) \in [0..n]$ of the string y that is currently stored in the memory cell (serves as reference value),
- (3) $(i(x) - 1)s$ entries for the already determined blocks $B_1, \dots, B_{i(x)-1}$,
- (4) s entries for the current block $B_{i(x)}$ of interest. If we are sampling new information (i.e., if $x_1 = 1$), then the substring r is a string taken from \mathcal{C}^s uniformly at random and r is the all-zeros string $[0, \dots, 0]$ of length s otherwise;
- (5) $(b - i(x))s$ zeros for the yet untouched blocks $B_{b'}$ with $i(x) < b' \leq b$,
- (6) ℓ entries for storing the length- ℓ prefix that coincides with Carole's hidden code (obtained through phase 0),
- (7) $2\ell_n + 1$ entries for storing the values $\text{eq}(z, x^0)$ and $\text{eq}(z, x^1)$ of the two reference strings x^0 and x^1 (explanation follows),
- (8) $t'(\ell_n + s + \ell_s + 1)$, $t' \leq t$, entries for storing, for each random sample,
 - (i) the value $\text{eq}(z, r \in \mathcal{E})$ for a reference string $r \in \mathcal{E}$ (in binary, requires ℓ_n positions),
 - (ii) the random sample $r \in \mathcal{C}^s$ itself,
 - (iii) its contribution $\Delta_{i(x)}(r) \in [0..s]$ to Carole's reply (in binary, requires ℓ_s positions), and
 - (iv) one additional "1" (to ease the computation of the number of guesses $q(x)$ via $p_1(x)$; details follow),
- (9) ℓ_s entries for encoding in binary which block we are currently trying to determine, and
- (10) the last two entries, $\text{suffix}(x)$, for indicating the current phase of the algorithm.

Clearly, one critical part is the limited storage capacity. For this reason, let us show that we have enough positions to store all the information needed to compute $\mathcal{S}_{i(x)}^{\text{consistent}}$, the set of all strings consistent with Carole's replies for the random guesses in the $i(x)$ th block $B_{i(x)}$.

Recall that, by Theorem 3, for determining the $i(x)$ th block $\text{BLOCK}_{i(x)}(z)$ of z , we need $t = (2 + \varepsilon) \frac{s(1+2\log k)}{\log s - \log k} = \Theta(s/\log n)$ random guesses ($\varepsilon > 0$ being an arbitrarily small constant). In addition, equivalently to the proof of Theorem 2, we need again 2 reference strings x^0 and x^1 (reference (7) in Eq. (1)). These two reference strings will be needed to infer the contributions $\Delta_{i(x)}(r)$ of the random samples $r \in \mathcal{C}^s$ in the $i(x)$ th block.

From the structure of the guesses presented in Eq. (1) above, we infer that the total storage requirement can be bounded from above by

$$\begin{aligned} & 1 + \ell_n + bs + \ell + 2\ell_n + 1 + t(\ell_n + s + \ell_s + 1) + \ell_s + 2 \\ & = bs + ts + o(n/\log n) \leq n(1 - K/\log n) + \Theta(n/\log n) + o(n/\log n) < n \end{aligned}$$

for sufficiently large, but constant K and sufficiently large n . This shows that, for sufficiently large n , Paul indeed can store all information needed to compute $\mathcal{S}_{i(x)}^{\text{consistent}}$ in one single string of length n .

Part II Let us now fix the notation used in the pseudo-code of routine `Sampling` (Algorithm 8). For all $b' < i(x)$ we set

$$\text{opt}(B_{b'}) := [x_{\ell+(b'-1)s+1} \dots x_{\ell+b's}],$$

the entries of x in the b' th block. The notation “`opt`” is justified by the fact that we shall have $\text{opt}(B_{b'}) = \text{BLOCK}_{b'}(z)$ for all $b' < i(x)$. Furthermore, let

`AddReferenceStringInfo(x)`

$$:= \left[\underbrace{0 \dots 0}_{(1),(2)} \mid \underbrace{\text{opt}(B_1) \mid \dots \mid \text{opt}(B_{i(x)-1})}_{(3)} \mid \underbrace{0 \dots 0}_{(4)} \mid \underbrace{x_{\ell+i(x)s+1} \dots x_{2\ell+bs}}_{(5),(6)} \mid \underbrace{x_2 \dots x_\ell \mid \text{binary}_{\ell_n}(\text{eq}(z, x)) \mid 1}_{(7)} \mid \underbrace{x_{2\ell+bs+2\ell_n+2} \dots x_n}_{(*)} \right],$$

where the references in the expression above are the same as the ones used in Eq. (1) and where (*) is simply a copy of the last entries of x . That is, the `AddReferenceStringInfo(x)` operation adds to x the values $\text{eq}(z, x^0)$ and $\text{eq}(z, x^1)$ and each of these values is stored in binary notation of length ℓ_n . Lastly, we denote by `Add(eq(z, x))` the operation

`Add(eq(z, x))`

$$:= \left[\underbrace{0 \dots 0}_{(1),(2)} \mid \underbrace{\text{opt}(B_1) \mid \dots \mid \text{opt}(B_{i(x)-1})}_{(3)} \mid \underbrace{0 \dots 0}_{(4)} \mid \underbrace{x_{\ell+i(x)s+1} \dots x_{p_1(x)}}_{(5),(6),(7),(8)} \mid \underbrace{x_2 \dots x_\ell \mid \text{BLOCK}_{i(x)}(x) \mid \text{binary}_{\ell_s}(\Delta_{i(x)}(\text{BLOCK}_{i(x)}(x))) \mid 1}_{(\dagger)} \mid \underbrace{x_{p_1(x)+\ell_n+s+\ell_s+2} \dots x_n}_{(*)} \right], \tag{2}$$

which adds (in substring (\dagger)) to the memory

- a copy of the value $\text{eq}(z, \text{ref})$ of a reference string ref (which was previously stored in positions $\{2, \dots, \ell\}$),
- the random sample $\text{BLOCK}_{i(x)}(x)$ of the last guess,
- the contribution $\Delta_{i(x)}(\text{BLOCK}_{i(x)}(x))$ of the random sample $\text{BLOCK}_{i(x)}(x)$ to $\text{eq}(z, x)$, and
- the one additional “1” that shall ease the computation of $q(x)$, the number of already queried samples.

All but the first ℓ entries (which are set to zero) are copied from x . The references in Eq. (2) are again the same as in Eq. (1).

Part III Let us now show in detail how to infer the contributions $\Delta_{i(x)}(r)$ of the random guesses. For clarity, we show how to do this for the first block; i.e., for the

Algorithm 8: The Sampling routine for $k = 2$ colors.

- 1 **Assumption:** Memory $\mathcal{M} = \{(x, \text{eq}(z, x))\}$ satisfies $\text{eq}(z, x) < n$,
 $\text{suffix}(x) = [01]$, $i(x) \leq b$, and $q(x) < t + 2$;
 - 2 **if** $q(x) = 0 \wedge x_1 = 0$ **then**
 - 3 $x \leftarrow [1|\text{binary}_{\ell_n}(\text{eq}(z, x))|\text{opt}(B_1)| \dots$
 $|\text{opt}(B_{i(x)-1})|1 \dots 1|x_{\ell+i(x)s+1} \dots x_n]$;
 - 4 Query $\text{eq}(z, x)$ and update \mathcal{M} by replacing $(x, \text{eq}(z, x))$;
 - 5 **else if** $q(x) = 0 \wedge x_1 = 1$ **then**
 - 6 $x \leftarrow \text{AddReferenceStringInfo}(x)$;
 - 7 Query $\text{eq}(z, x)$ and update \mathcal{M} by replacing $(x, \text{eq}(z, x))$;
 - 8 **else if** $2 \leq q(x) < t + 2 \wedge x_1 = 0$ **then**
 - 9 $x \leftarrow [1|\text{binary}_{\ell_n}(\text{eq}(z, x))|\text{opt}(B_1)| \dots$
 $|\text{opt}(B_{i(x)-1})|r|x_{\ell+i(x)s+1} \dots x_n]$ for $r \in \mathcal{C}^s$ chosen u.a.r.;
 - 10 Query $\text{eq}(z, x)$ and update \mathcal{M} by replacing $(x, \text{eq}(z, x))$;
 - 11 **else if** $2 \leq q(x) < t + 2 \wedge x_1 = 1$ **then**
 - 12 $x \leftarrow \text{Add}(\text{eq}(z, x))$;
 - 13 Query $\text{eq}(z, x)$ and update \mathcal{M} by replacing $(x, \text{eq}(z, x))$;
-

positions $\{\ell + 1, \dots, \ell + s\}$. The procedure is similar for all other blocks and we shall comment on this case at the end of this part.

First note that after executing lines 7 to 9 of Algorithm 7, Paul enters the routine Sampling with $\mathcal{M} = \{(x^0, \text{eq}(z, x^0))\}$ where

$$x^0 = \left[\underbrace{0 \dots 0}_{\ell+bs \text{ entries}} | z_1 \dots z_\ell | \underbrace{0 \dots 0}_{n-(2\ell+bs+\ell_s+2) \text{ entries}} \mid \underbrace{|\text{binary}_{\ell_s}(1)|01}_{\ell_s \text{ entries}} \right],$$

and he queries in the first sampling iteration (lines 2–4 of Algorithm 8)

$$x^1 = \left[\underbrace{1|\text{binary}_{\ell_n}(\text{eq}(z, x^0))|}_{1+\ell_n=\ell \text{ entries}} \mid \underbrace{1 \dots 1}_s \mid \underbrace{0 \dots 0}_{(b-1)s \text{ entries}} \mid \right. \\ \left. z_1 \dots z_\ell \mid \underbrace{0 \dots 0}_{n-(2\ell+bs+\ell_s+2) \text{ entries}} \mid \text{binary}_{\ell_s}(1)|01 \right]$$

with the entries in the first block replaced by the all-ones substring $[1, \dots, 1]$ and the first ℓ entries updated. We can compute the contribution of the first ℓ entries $[1|\text{binary}_{\ell_n}(\text{eq}(z, x^0))]$ to the value $\text{eq}(z, x^1)$ via

$$\begin{aligned} \tilde{f}(x^1) &:= \text{eq}([z_1 \dots z_\ell], [1|\text{binary}_{\ell_n}(\text{eq}(z, x^0))]) \\ &= \text{eq}([x_{\ell+bs+1}^1 \dots x_{2\ell+bs}^1], [x_1^1 \dots x_\ell^1]), \end{aligned}$$

and, by the same reasoning, the contribution of the first ℓ entries in x^0 to $\text{eq}(z, x^0)$ via $\tilde{f}(x^0) = \text{eq}([x_{\ell+bs+1}^0 \dots x_{2\ell+bs}^0], [0 \dots 0])$. Let us, for a moment, assume that we now had $\mathcal{M} = \{x^1, \text{eq}(z, x^1)\}$ and that we had another string

$$y = \left[z_1 \dots z_\ell \mid \underbrace{r}_{s \text{ entries}} \mid \underbrace{0 \dots 0}_{(b-1)s \text{ entries}} \mid z_1 \dots z_\ell \mid \underbrace{0 \dots 0}_{n-(2\ell+bs+\ell_s+2) \text{ entries}} \mid \text{binary}_{\ell_s}(1)01 \right]$$

for some random substring $r \in \mathcal{C}^s$. Then we could compute the contribution of the random entries r in the first block B_1 of y via

$$\tilde{\Delta}_1(r) = \text{eq}(z, y) - \frac{\text{eq}(z, x^0) + \text{eq}(z, x^1) + (\ell - \tilde{f}(x^0)) + (\ell - \tilde{f}(x^1)) - s}{2}.$$

Key to this equality is the fact that the first ℓ entries of y contribute ℓ to Carole’s response $\text{eq}(z, y)$ to guess y , whereas the first ℓ entries of x^0 and x^1 contribute $\tilde{f}(x^0) + \tilde{f}(x^1)$ to the sum $\text{eq}(z, x^0) + \text{eq}(z, x^1)$, and the fact that the entries in the first block— $[0 \dots 0]$ and $[1 \dots 1]$, respectively—contribute in total s towards $\text{eq}(z, x^0) + \text{eq}(z, x^1)$. All other entries $x_i, y_i, i > \ell + s$ contribute either 2 or 0 to the sum $\text{eq}(z, x^0) + \text{eq}(z, x^1)$ and every entry contributes 2 if and only if it contributes 1 to the value $\text{eq}(z, y)$.

Note, however, that we would now have to choose which of the strings to keep in the memory and we would eventually loose the information $\text{eq}(z, x^1)$. Therefore, in lines 6 and 7 in Algorithm 8, we first query the reference string

$$\begin{aligned} x^2 &= \text{AddReferenceStringInfo}(x^1) \\ &= \left[\underbrace{0 \dots 0}_{\ell \text{ entries}} \mid \underbrace{0 \dots 0}_{s \text{ entries}} \mid \underbrace{0 \dots 0}_{(b-1)s} \mid z_1 \dots z_\ell \mid \right. \\ &\quad \left. \underbrace{\text{binary}_{\ell_n}(\text{eq}(z, x^0)) \mid \text{binary}_{\ell_n}(\text{eq}(z, x^1)) \mid 1}_{2\ell_n+1 \text{ entries}} \right] \\ &\quad \left[\underbrace{0 \dots 0}_{n-(2\ell+bs+2\ell_n+1+\ell_s+2) \text{ entries}} \mid \text{binary}_{\ell_s}(1)01 \right]. \end{aligned}$$

This query is needed only to store the values $\text{eq}(z, x^0)$ and $\text{eq}(z, x^1)$ of both reference strings. Since adding the substring $[\text{binary}_{\ell_n}(\text{eq}(z, x^0)) \mid \text{binary}_{\ell_n}(\text{eq}(z, x^1)) \mid 1]$ to the memory string again changes the number of positions in which the guess and Carole’s hidden string coincide, we need to store this information in the next query as well. More precisely, we have that x^0 and x^2 differ in exactly the substring $[\text{binary}_{\ell_n}(\text{eq}(z, x^0)) \mid \text{binary}_{\ell_n}(\text{eq}(z, x^1)) \mid 1]$, and the contribution of this substring (compared to the all-zeros substring which it replaces) is $\text{eq}(z, x^2) - \text{eq}(z, x^0)$.

Furthermore, we need to indicate that we are sampling a new random substring. This is the first position in the string and the next $\ell - 1$ entries are needed to encode in binary the value $\text{eq}(z, x^2)$. That is, instead of querying y as above we query (lines 9 and 10 in Algorithm 8)

$$\begin{aligned} x^3 &= [1 \mid \text{binary}_{\ell_n}(\text{eq}(z, x^2)) \mid r^{(1)} \mid 0 \dots 0 \mid z_1 \dots z_\ell \mid \text{binary}_{\ell_n}(\text{eq}(z, x^0)) \mid \\ &\quad \text{binary}_{\ell_n}(\text{eq}(z, x^1)) \mid 1 \mid 0 \dots 0 \mid \text{binary}_{\ell_s}(1)01], \end{aligned}$$

where the substring $r^{(1)} \in \mathcal{C}^s$ in B_1 is taken uniformly at random. The number of zeros in the first all-zeros substring is again $(b - 1)s$ and in the second all-zeros substring it is $n - (2\ell + bs + 2\ell_n + 1 + \ell_s + 2)$. Now, in the same fashion as above, we can compute the contribution $\Delta_1(r^{(1)}) = \text{eq}([z_{\ell+1} \dots z_{\ell+s}])$ of the substring $r^{(1)} \in \mathcal{C}^s$ via

$$\Delta_1(r^{(1)}) = \text{eq}(z, x^3) - \left(\frac{\text{eq}(z, x^0) + \text{eq}(z, x^1) + (\ell - \tilde{f}(x^1)) + (\ell - \tilde{f}(x^0)) - s}{2} + (\text{eq}(z, x^2) - \text{eq}(z, x^0)) - (\ell - \tilde{f}(x^3)) \right). \tag{3}$$

Note that all the information needed for this computation is contained in the string x^3 itself.

Since later we want to be able to regain the full guessing history, in the next guess we store both the reference value $\text{eq}(z, x^2)$ as well as the contribution $\Delta_1(r^{(1)})$. And, of course, we also need to store the random guess $r^{(1)} = \text{BLOCK}_1(x^3)$ itself. Therefore, we query (lines 12 and 13 in Algorithm 8) in the next iteration of Algorithm 7

$$x^4 = \left[\underbrace{0 \dots 0}_{\ell} \mid \underbrace{0 \dots 0}_s \mid \underbrace{0 \dots 0}_{(b-1)s} \mid z_1 \dots z_{\ell} \mid \text{binary}_{\ell_n}(\text{eq}(z, x^0)) \mid \text{binary}_{\ell_n}(\text{eq}(z, x^1)) \mid 1 \mid \underbrace{\text{binary}_{\ell_n}(\text{eq}(z, x^2))}_{=[x_2^3 \dots x_{\ell}^3]} \mid \underbrace{\text{BLOCK}_1(x^3)}_{=r^{(1)}} \mid \underbrace{\text{binary}_{\ell_s}(\Delta_1(r^{(1)}))}_{\text{see Eq. (3)}} \mid 1 \mid 0 \dots 0 \mid \text{binary}_{\ell_s}(1) \mid 01 \right].$$

Note that, since $\Delta_1(r^{(1)}) \in [0 \dots s]$, we can encode this value using ℓ_s positions only.

By continuing like this we are able to compute, in any iteration of the first phase, the contributions $\Delta_{i(x)}(r)$ of the random substrings $r \in \mathcal{C}^s$.

As in the proof of Theorem 2 we need to be able to compute how many random guesses have been queried already for the current block of interest. As indicated above, this can be derived from $p_1(x)$ as follows. For any random guess $r \in \mathcal{C}^s$ we use $\ell_n + s + \ell_s + 1$ entries for storing all information that will be needed later to regain the full guessing history. Furthermore, we used $2\ell_n + 1$ entries for storing the values $\text{eq}(z, x^0)$ and $\text{eq}(z, x^1)$ of the two reference strings x^0 and x^1 , and we store information only in positions $i > 2\ell + bs$. Hence, the number of guesses for block $B_{i(x)}$ can be computed as

$$q(x) = \begin{cases} 0, & \text{if } p_1(x) \leq 2\ell + bs \text{ and } x_1 = 0 \\ 1, & \text{if } p_1(x) \leq 2\ell + bs \text{ and } x_1 = 1 \\ 2 + \frac{p_1(x) - (2\ell + bs + 2\ell_n + 1)}{\ell_n + s + \ell_s + 1}, & \text{otherwise.} \end{cases}$$

After querying t random guesses (i.e., after querying a total number of $t + k$ guesses) for the first block, we regain the full guessing history from the string x currently in the memory as follows. The i th random sample $r^{(i)} \in \mathcal{C}^s$ which we guessed

for the first block is

$$r^{(i)} := [x_{2\ell+bs+2\ell_n+1+(i-1)(\ell_n+s+\ell_s+1)+\ell_n+1} \cdots x_{2\ell+bs+2\ell_n+1+(i-1)(\ell_n+s+\ell_s+1)+\ell_n+s}],$$

and the corresponding query was

$$y^{(i)} := [1|x_{2\ell+bs+2\ell_n+1+(i-1)(\ell_n+s+\ell_s+1)+1} \cdots x_{2\ell+bs+2\ell_n+1+(i-1)(\ell_n+s+\ell_s+1)+\ell_n} | \\ r^{(i)}|x_{\ell+s+1} \cdots x_{2\ell+bs+2\ell_n+1+(i-1)(\ell_n+s+\ell_s+1)}|0 \dots 0|x_{n-\ell_s-1} \cdots x_n].$$

We have stored in binary the contribution $\Delta_1(r^{(i)})$ of $r^{(i)}$ to the overall function value $\text{eq}(z, y^{(i)})$ in positions

$$\{2\ell + bs + 2\ell_n + 1 + (i - 1)(\ell_n + s + \ell_s + 1) + \ell_n + s + 1, \dots, \\ 2\ell + bs + 2\ell_n + 1 + (i - 1)(\ell_n + s + \ell_s + 1) + \ell_n + s + \ell_s\}$$

and thus we have

$$\Delta_1(r^{(i)}) = \sum_{i=0}^{\ell_s-1} 2^i x_{2\ell+bs+2\ell_n+1+(i-1)(\ell_n+s+\ell_s+1)+\ell_n+s+\ell_s-i}.$$

By Theorem 3, the expected size of

$$\mathcal{S}_1^{\text{consistent}} := \{w \in \{0, 1\}^s \mid \forall i \leq t : \text{eq}(y, r^{(i)}) = \Delta_1(r^{(i)})\}$$

is bounded from above by $1 + 1/s$. That is, we can now identify $\text{BLOCK}_1(z)$ in a constant number of guesses. These are lines 3–5 of routine `OptimizeBlock` (Algorithm 9).

As mentioned above, determining the other blocks $2, \dots, b$ is similar. In these iterations, the $(i(x) - 1)s$ entries in positions $\{\ell + 1, \dots, \ell + (i(x) - 1)s\}$ are already optimized, that is, they coincide with Carole’s hidden string z . Thus, they are not changed in any further iteration of Algorithm 9.

Algorithm 9: The `OptimizeBlock` routine.

- 1 **Assumption:** Memory $\mathcal{M} = \{(x, \text{eq}(z, x))\}$ satisfies $\text{eq}(z, x) < n$, $\text{suffix}(x) = [01]$, $i(x) \leq b$, and $q(x) = t + 2$;
- 2 **if** $x_1 = 0$ **then**
- 3 $y \leftarrow [1 \dots 1 | \text{opt}(B_1) | \dots | \text{opt}(B_{i(x)-1}) | w | x_{\ell+i(x)s+1} \dots x_n]$ for $w \in \mathcal{S}_{i(x)}^{\text{consistent}}$ chosen u.a.r.;
- 4 Query $\text{eq}(z, y)$;
- 5 **if** $\Delta_{i(x)}(\text{BLOCK}_{i(x)}(y)) = s$ **then** $\mathcal{M} \leftarrow \{(y, \text{eq}(z, y))\}$;
 $//w = \text{BLOCK}_{i(x)}(z)$
- 6 **else**
- 7 $x \leftarrow \text{Update}(x)$;
- 8 Query $\text{eq}(z, x)$ and update \mathcal{M} by replacing $(x, \text{eq}(z, x))$; $//$ string prepared for determining the next block

Part IV Once $\text{BLOCK}_{i(x)}(z) = [z_{\ell+(i(x)-1)s+1} \dots z_{\ell+i(x)s}]$ has been determined, we need to update the memory such that we can start determining the entries of the next block. These are lines 7 and 8 in Algorithm 9. Here we abbreviate

$$\text{Update}(x) := \left[\underbrace{0 \dots 0}_{(a)} \mid \underbrace{\text{opt}(B_1) \mid \dots \mid \text{opt}(B_{i(x)})}_{(b)} \mid \underbrace{0 \dots 0}_{(c)} \mid \underbrace{x_{\ell+bs+1} \dots x_{2\ell+bs}}_{(d)} \right. \\ \left. \underbrace{0 \dots 0}_{(e)} \mid \underbrace{\text{binary}_{\ell_s}(i(x) + 1)}_{(f)} \mid \underbrace{01}_{(g)} \right],$$

where

- (a) the first ℓ entries are set to zero,
- (b) we now have $i(x)$ already determined blocks,
- (c) the new block of interest, block $i(x) + 1$, as well as all blocks $b' > i(x) + 1$ are (still) set to zero,
- (d) we keep the copy of the prefix $[z_1 \dots z_\ell]$ in positions $\{\ell + bs + 1, \dots, 2\ell + bs\}$,
- (e) all information that we have used in the previous query to determine block $B_{i(x)}$ is removed (and set to zero),
- (f) the index for the current block of interest is increased by one, and
- (g) the last two entries still indicate the second phase.

The Case of $k \geq 3$ Colors For the general case, the main strategy as given by Algorithm 7 remains the same. What needs to be changed is the Sampling routine where instead of sampling only two reference strings x^0 and x^1 , we need to sample k reference strings x^0, x^1, \dots, x^{k-1} with $\text{BLOCK}_{i(x)}(x^c) = [c \dots c]$ for all $c \in [0..k - 1]$.

Algorithm 10 shows the generalized sampling routine. Here we define

AddReferenceStringInfo₂(x)

$$:= \left[\underbrace{0 \dots 0}_{(1), (2)} \mid \underbrace{\text{opt}(B_1) \mid \dots \mid \text{opt}(B_{i(x)-1})}_{(3)} \mid \underbrace{0 \dots 0}_{(4)} \mid \underbrace{x_{\ell+i(x)s+1} \dots x_{p_1(x)}}_{(5), (6), (7), (7')} \right. \\ \left. \underbrace{x_2 \dots x_\ell \mid \text{binary}_{\ell_n}(\text{eq}(z, x)) \mid 1}_{(\dagger)} \mid \underbrace{x_{p_1(x)+\ell_n+s+\ell_s+2} \dots x_n}_{(*)} \right],$$

where

- (1)–(7) are the same references as in Eq. (1),
- (7') are the additional positions needed for storing the values $\text{eq}(z, x^j)$ of the already queried reference strings $x^2, \dots, x^{q(x)-1}$ (requiring $2\ell_n + 1$ positions each),
- (†) we add the information of the $q(x)$ th reference string $x^{q(x)}$ to the memory (again requiring $2\ell_n + 1$ positions), and
- (*) is simply a copy of the last entries of the previous guess.

The substring $[x_2 \dots x_\ell]$ is needed again to infer the contribution of the positions in which we added the information of the previous reference string $x^{q(x)-1}$. The reasoning is the same as in the case of $k = 2$ colors.

Algorithm 10: The Sampling routine for $k \geq 3$ colors.

- 1 **Assumption:** Memory $\mathcal{M} = \{(x, \text{eq}(z, x))\}$ satisfies $\text{eq}(z, x) < n$, $\text{suffix}(x) = [01]$, $i(x) \leq b$, and $q(x) < t + k$;
 - 2 **if** $q(x) = 0 \wedge x_1 = 0$ **then**
 - 3 $x \leftarrow [1|\text{binary}_{\ell_n}(\text{eq}(z, x))|\text{opt}(B_1)|\dots|\text{opt}(B_{i(x)-1})|1\dots 1$
 $|x_{\ell+i(x)s+1}\dots x_n]$;
 - 4 Query $\text{eq}(z, x)$ and update \mathcal{M} by replacing $(x, \text{eq}(z, x))$;
 - 5 **else if** $q(x) = 0 \wedge x_1 = 1$ **then**
 - 6 $x \leftarrow \text{AddReferenceStringInfo}(x)$;
 - 7 Query $\text{eq}(z, x)$ and update \mathcal{M} by replacing $(x, \text{eq}(z, x))$;
 - 8 **else if** $2 \leq q(x) < k \wedge x_1 = 0$ **then**
 - 9 $x \leftarrow [1|\text{binary}_{\ell_n}(\text{eq}(z, x))|\text{opt}(B_1)|\dots|\text{opt}(B_{i(x)-1})|q(x)\dots q(x)$
 $|x_{\ell+i(x)s+1}\dots x_n]$;
 - 10 Query $\text{eq}(z, x)$ and update \mathcal{M} by replacing $(x, \text{eq}(z, x))$;
 - 11 **else if** $2 \leq q(x) < k \wedge x_1 = 1$ **then**
 - 12 $x \leftarrow \text{AddReferenceStringInfo}_2(x)$;
 - 13 Query $\text{eq}(z, x)$ and update \mathcal{M} by replacing $(x, \text{eq}(z, x))$;
 - 14 **else if** $k \leq q(x) < t + k \wedge x_1 = 0$ **then**
 - 15 $x \leftarrow [1|\text{binary}_{\ell_n}(\text{eq}(z, x))|\text{opt}(B_1)|\dots|\text{opt}(B_{i(x)-1})|r|x_{\ell+i(x)s+1}\dots x_n]$
 for $r \in \mathcal{C}^s$ chosen u.a.r.;
 - 16 Query $\text{eq}(z, x)$ and update \mathcal{M} by replacing $(x, \text{eq}(z, x))$;
 - 17 **else if** $k \leq q(x) < t + k \wedge x_1 = 1$ **then**
 - 18 $x \leftarrow \text{Add}(\text{eq}(z, x))$;
 - 19 Query $\text{eq}(z, x)$ and update \mathcal{M} by replacing $(x, \text{eq}(z, x))$;
-

Since we added more reference string information, we need to adjust the definition of $q(x)$ accordingly. Since we need $2\ell_n + 1$ additional bits for each reference string x^j , $2 \leq j \leq k - 1$, and we use $\ell_n + s + \ell_s + 1$ entries for storing the information of each random guess, we have

$$q(x) := \begin{cases} 0, & \text{if } p_1(x) \leq 2\ell + bs \text{ and } x_1 = 0 \\ 1, & \text{if } p_1(x) \leq 2\ell + bs \text{ and } x_1 = 1 \\ j, & \text{if } p_1(x) = 2\ell + bs + 2\ell_n + 1 + (j - 2)(2\ell_n + 1) \text{ and } 2 \leq j < k \\ k + \frac{p_1(x) - (2\ell + bs + (k-1)(2\ell_n + 1))}{\ell_n + s + \ell_s + 1}, & \text{otherwise.} \end{cases}$$

It is easily verified that all statements made in the above proof for $k = 2$ colors remain correct if we consider the general case of $k \geq 3$ colors. Only the computation of the contributions $\Delta_{i(x)}(\text{BLOCK}_{i(x)}(x))$, Eq. (3), becomes a bit more tedious. However, all calculations are straightforward. □

Acknowledgements 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. Anil, G., Wiegand, R.P.: Black-box search by elimination of fitness functions. In: Proc. of Foundations of Genetic Algorithms (FOGA'09), pp. 67–78. ACM, New York (2009)
2. Chen, Z., Cunha, C., Homer, S.: Finding a hidden code by asking questions. In: Proc. of the Second Annual International Conference on Computing and Combinatorics (COCOON'96). Lecture Notes in Computer Science, vol. 1090, pp. 50–55. Springer, Berlin (1996)
3. Vasek, C.: Mastermind. *Combinatorica* **3**, 325–329 (1983)
4. Droste, S., Jansen, T., Wegener, I.: Upper and lower bounds for randomized search heuristics in black-box optimization. *Theory Comput. Syst.* **39**, 525–544 (2006)
5. Doerr, B., Winzen, C.: Memory-restricted black-box complexity of OneMax. *Inf. Process. Lett.* **112**, 32–34 (2012)
6. Doerr, B., Winzen, C.: Playing Mastermind with constant-size memory. In: Proc. of the Symposium on Theoretical Aspects of Computer Science (STACS'12), LIPIcs, vol. 14, pp. 441–452 (2012). Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik
7. Erdős, P., Rényi, A.: On two problems of information theory. *Magy. Tud. Akad. Mat. Kut. Intéz. Közl.* **8**, 229–243 (1963)
8. Goodrich, M.T.: On the algorithmic complexity of the mastermind game with black-peg results. *Inf. Process. Lett.* **109**, 675–678 (2009)
9. Jäger, G., Peczarski, M.: The number of pessimistic guesses in generalized black-peg Mastermind. *Inf. Process. Lett.* **111**, 933–940 (2011)
10. Knuth, D.E.: The computer as a master mind. *J. Recreat. Math.* **9**, 1–5 (1977)
11. Motwani, R., Raghavan, P.: *Randomized Algorithms*. Cambridge University Press, Cambridge (1995)
12. Stuckman, J., Zhang, G.-Q.: Mastermind is NP-complete. *INFOCOMP J. Comput. Sci.* **5**, 25–28 (2006)
13. Wegener, I.: Simulated annealing beats metropolis in combinatorial optimization. In: Proc. of the 32nd International Colloquium on Automata, Languages and Programming (ICALP'05). Lecture Notes in Computer Science, vol. 3580, pp. 589–601. Springer, Berlin (2005)