

Improved Approximation Algorithm for the Number of Queries Necessary to Identify a Permutation

Mourad El Ouali Volkmar Sauerland

Department of Computer Science
Christian-Albrechts-Universität zu Kiel
Kiel, Germany

`<meo,vsa>@informatik.uni-kiel.de`

November 1, 2018

Abstract

In the past three decades, deductive games have become interesting from the algorithmic point of view. Deductive games are two players zero sum games of imperfect information. The first player, called "codemaker", chooses a secret code and the second player, called "codebreaker", tries to break the secret code by making as few guesses as possible, exploiting information that is given by the codemaker after each guess. A well known deductive game is the famous Mastermind game. In this paper, we consider the so called Black-Peg variant of Mastermind, where the only information concerning a guess is the number of positions in which the guess coincides with the secret code. More precisely, we deal with a special version of the Black-Peg game with n holes and $k \geq n$ colors where no repetition of colors is allowed. We present a strategy that identifies the secret code in $\mathcal{O}(n \log_2 n)$ queries. Our algorithm improves the previous result of Ker-I Ko and Shia-Chung Teng (1985) by almost a factor of 2 for the case $k = n$. To our knowledge there is no previous work dealing with the case $k > n$.

Keywords: Mastermind; combinatorial problems; permutations; algorithms

1 Introduction

In the past three decades, deductive games have become interesting from the algorithmic point of view. In this kind of games, two players are involved. They are called the codemaker and the codebreaker, respectively. One of the most famous games of this kind is Mastermind.

1.1 Problem Description

Mastermind is a two players board game invented in 1970 by the postmaster and telecommunication expert Mordecai Meirowitz. The original version of Mastermind consists of a board with twelve (or ten, or eight) rows containing four holes and pegs of six different colors. The idea of the game is that the codemaker, say "Carole", chooses a secret color combination of four pegs from the six possible colors and the codebreaker, say "Paul", has to identify the code by a sequence of queries and corresponding information that is provided by Carole. All queries are also color combinations of four pegs. Information is given about the number of correctly positioned colors and further correct colors, respectively. Mathematically, we have two arbitrary positive integers, n and k . Carole selects a vector $y \in \{1, \dots, k\}^n$ and Paul gives in each iteration a query in form of a vector $x \in \{1, \dots, k\}^n$. Carole replies with a pair of two numbers:

- A black information $\text{black}(x, y)$, which is the number of positions in which both vectors x and y coincide:

$$\text{black}(x, y) = |\{i \in \{1, \dots, n\}; x(i) = y(i)\}|.$$

- A white information $\text{white}(x, y)$, which is the number of additional pegs with a right color but a wrong position:

$$\text{white}(x, y) = \max_{\sigma \in S_n} |\{i \in \{1, \dots, n\}; y(i) = x(\sigma(i))\}| - \text{black}(x, y).$$

The Black-Peg game is a special version of Mastermind, where Carole answers only with the black information. A further version is the so-called AB game in which all colors within a code must be distinct. In this paper, we deal with a special combination of the Black-Peg game and the AB game, where both the secret vector and the guesses are permutations ($k \geq n$) and the answers are given by the black information, only.

1.2 Related Works

It is not only the playful nature of Mastermind that has attracted the attention of computer scientists, but more importantly its relation to information-theoretic and fundamental complexity questions. In 1963, several years before the invention of Mastermind as a commercial board game, Erdős and Rényi [6] analyzed the same problem with two colors. One of the earliest analysis of this game after its commercialization dealing with the case of 4 pegs and 6 colors was done by Knuth [14]. He presented a strategy that identifies the secret code in at most 5 guesses. Ever since the work of Knuth the general case of arbitrary many pegs and colors has been intensively investigated in combinatorics and computer science literature.

In the field of complexity, Stuckman and Zhang [18] showed that it is \mathcal{NP} -complete to determine if a sequence of queries and answers is satisfiable. Concerning the approximation aspect, there are many works regarding different methods [1, 2, 3, 5, 7, 8, 9, 10, 12, 16, 17, 18, 19].

The Black-Peg game was first introduced by Chvátal for the case $k = n$. He gave a deterministic adaptive strategy that uses $2n\lceil\log_2 k\rceil + 4n$ guesses. Later, Goodrich [11] improved the result of Chvátal for arbitrary n and k to $n\lceil\log_2 k\rceil + \lceil(2 - 1/k)n\rceil + k$ guesses. Moreover, he proved in same paper that this kind of game is \mathcal{NP} -complete. A further improvement to $n\lceil\log_2 n\rceil + k - n + 1$ for $k > n$ and $n\lceil\log_2 n\rceil + k$ for $k \leq n$ was done by Jäger and Peczarski [13]. Recently, Doerr et al [4] improved the result obtained by Chvátal to $\mathcal{O}(n \log \log n)$ and also showed that this asymptotic order even holds for up to $n^2 \log \log n$ colors, if both black and white information is allowed.

Another variant of Mastermind is the so-called AB game, also known as "Bulls and Cows" game. Here, all pegs in the secret code as well as in each query must have distinct colors.

Concerning the combination of both variants, Black-Peg game and AB game, there is only one work due to Ker-I Ko and Shia-Chung Teng [15] for the case $k = n$. They presented a strategy that identifies the secret permutation in at most $2n \log_2 n + 7n$ guesses and proved that the corresponding counting problem is $\#\mathcal{P}$ -complete. To our knowledge there is no result for the case $k > n$, yet.

1.3 Our Contribution

In this paper we consider the Black-Peg game without color repetition. We present a polynomial-time algorithm that identifies the secret permutation in less than $n \log_2 n + \lambda n$ queries in the case $k = n$ and in less than $n \log_2 n + k + 2n$ queries in the case $k > n$. Our performance in the case $k = n$ is an improvement of the result of Ker-I Ko and Shia-Chung Teng [15] by almost a factor of 2. Note, that additional difficulty compared to the case considered by Doerr et al is given here by the fact that color repetition is not only forbidden for the secret code but also for the guesses.

2 An Algorithm for Permutation-Mastermind

We first consider the case $k = n$ and forbidden color repetition, meaning that codes are one-line representations of permutations in S_n . For convenience, we will use the term permutation for both, a mapping in S_n and its one-line representation as a vector. Our algorithm for finding the secret permutation $y \in S_n$ includes two main phases which are based on two ideas. In the first phase we guess an initial sequence of n permutations that has a predefined structure. In the second phase, the structure of the initial sequence and the corresponding information by the codemaker enable us to identify correct components y_i of the secret code one after another, each by using a binary search. Recall, that for two codes $w = (w_1, \dots, w_n)$ and $x = (x_1, \dots, x_n)$, we denote by $\text{black}(w, x)$ the number $|\{i \in \{1, \dots, n\} \mid w_i = x_i\}|$ of components in which w and x are equal. We denote the mapping x restricted to the set $\{s, \dots, l\}$ with $(x_i)_{i=s}^l$, $s, l \in \{1, \dots, n\}$.

2.1 Phase 1

Consider the n permutations, $\sigma^1, \dots, \sigma^n$, that are defined as follows: σ^1 corresponds to the identity map and for $j \in \{1, \dots, n-1\}$, we obtain σ^{j+1}

from σ^j by a circular shift to the right, i.e. we set

$$\begin{aligned}
\sigma^1 &= (1, 2, \dots, n), \\
\sigma^2 &= (n, 1, 2, \dots, n-1), \\
\sigma^3 &= (n-1, n, 1, 2, \dots, n-2), \\
&\dots, \\
\sigma^{n-1} &= (3, 4, \dots, n, 1, 2), \\
\sigma^n &= (2, 3, \dots, n, 1).
\end{aligned} \tag{1}$$

Within those n permutations, every color appears exactly once at every position and, thus, we have

$$\sum_{j=1}^n \text{black}(\sigma^j, y) = n. \tag{2}$$

We guess $\sigma^1, \dots, \sigma^{n-1}$ and obtain the additional information $\text{black}(\sigma^n, y)$ from (2).

2.2 Phase 2

The strategy of the second phase identifies the values of y one after another. This is done by using two binary search routines, called `FINDFIRST` and `FINDNEXT`, respectively. The idea behind both binary search routines is to exploit the information that for $1 \leq i, j \leq n-1$ we have $\sigma_i^j = \sigma_{i+1}^{j+1}$, $\sigma_i^n = \sigma_{i+1}^1$, $\sigma_n^j = \sigma_1^{j+1}$ and $\sigma_n^n = \sigma_1^1$. While, except for an unfrequent special case, `FINDFIRST` is used to identify the first correct component of the secret code, `FINDNEXT` identifies the remaining components in the main loop of the algorithm. Actually, `FINDFIRST` would also be able to find the remaining components but requires more guesses than `FINDNEXT` (twice as much in the worst case). On the other hand, `FINDNEXT` does only work if at least one value of y is already known such that we have to identify the value of one secret code component in advance.

2.2.1 Identifying the First Component

Equation (2) implies that either $\text{black}(\sigma^j, y) = 1$ holds for all $j \in \{1, \dots, n\}$ or that we can find a $j \in \{1, \dots, n\}$ with $\text{black}(\sigma^j, y) = 0$.

In the first case, which is unfrequent, we can find one correct value of y by guessing at most $\frac{n}{2} + 1$ modified versions of some initial guess, say σ^1 . Namely, if we define a guess σ by swapping a pair of components of σ^1 , we will obtain $\text{black}(\sigma, y) = 0$, if and only if one of the swapped components has the correct value in σ^1 .

In the frequent second case, we find the first component by `FINDFIRST` in at most $2\lceil \log_2 n \rceil$ guesses. The routine `FINDFIRST` is outlined as Algorithm 1 and works as follows: In the given case, we can either find a $j \in \{1, \dots, n-1\}$ with $\text{black}(\sigma^j, y) > 0$ but $\text{black}(\sigma^{j+1}, y) = 0$ and set $r := j + 1$, or we have $\text{black}(\sigma^n, y) > 0$ but $\text{black}(\sigma^1, y) = 0$ and set $j := n$ and $r := 1$. We call such an index j an *active* index. Now, for every $l \in \{2, 3, \dots, n\}$ we define the code

$$\sigma^{j,l} := ((\sigma_i^j)_{i=1}^{l-1}, \sigma_1^r, (\sigma_i^r)_{i=l+1}^n),$$

and call the peg at position l in $\sigma^{j,l}$ the pivot peg. From the information $\sigma_i^j = \sigma_{i+1}^r$ for $1 \leq i \leq n-1$ we conclude that $\sigma^{j,l}$ is actually a new permutation as required. The fact that $\text{black}(\sigma^r, y) = 0$ implies that the number of correct pegs up to position $l-1$ in σ^j is either $\text{black}(\sigma^{j,l}, y)$ (if $y_l \neq \sigma_1^r$) or $\text{black}(\sigma^{j,l}, y) - 1$ (if $y_l = \sigma_1^r$). For our algorithm, we will only need to know if there exist one correct peg in σ^j up to position $l-1$. The question is cleared up, if $\text{black}(\sigma^{j,l}, y) \neq 1$. On the other hand, if $\text{black}(\sigma^{j,l}, y) = 1$, we can define a new guess $\rho^{j,l}$ by swapping the pivot peg with a wrong peg in $\sigma^{j,l}$. We define

$$\rho^{j,l} := \begin{cases} ((\sigma_i^j)_{i=1}^l, \sigma_1^r, (\sigma_i^r)_{i=l+2}^n) & \text{if } l < n \\ (\sigma_1^r, (\sigma_i^j)_{i=2}^{n-1}, \sigma_1^j) & \text{if } l = n \end{cases}$$

assuming for the case $l = n$, that we know that $\sigma_1^j \neq y_1$. We will obtain $\text{black}(\rho^{j,l}, y) > 0$, if and only if the pivot peg had a wrong color before, meaning that there is one correct peg in σ^j in the first $l-1$ places. Thus, we can find the position m of the left most correct peg in σ^j by a binary search as outlined in Algorithm 1.

2.2.2 Identifying a Further Component

For the implementation of `FINDNEXT` we deal with a partial solution vector x that satisfies $x_i \in \{0, y_i\}$ for all $i \in \{1, \dots, n\}$. We call the (indices of the) non-zero components of the partial solution *fixed*. They indicate the components of the secret code that have already been identified. The (indices

Algorithm 1: Function FINDFIRST

input : Code y and an active index $j \in \{1, \dots, n\}$
output: Position m of the left most correct peg in σ^j

```
1 if  $j = n$  then  $r := 1$  else  $r := j + 1$ ;  
2  $a := 1$ ;  
3  $b := n$ ;  
4  $m := n$ ; // correct position in  $\sigma^j$  to be determined  
5 while  $b > a$  do  
6    $l := \lceil \frac{a+b}{2} \rceil$ ; // current pivot position  
7   Guess  $\sigma^{j,l} := ((\sigma_i^j)_{i=1}^{l-1}, \sigma_1^r, (\sigma_i^r)_{i=l+1}^n)$ ;  
8    $s := \text{black}(\sigma^{j,l}, y)$ ;  
9   if  $s = 1$  then  
10    if  $l < n$  then  $\rho^{j,l} := ((\sigma_i^j)_{i=1}^l, \sigma_1^r, (\sigma_i^r)_{i=l+2}^n)$ ;  
11    else  $\rho^{j,l} := (\sigma_1^r, (\sigma_i^j)_{i=2}^{n-1}, \sigma_1^j)$ ;  
12    Guess  $\rho^{j,l}$ ;  
13     $s := \text{black}(\rho^{j,l}, y)$ ;  
14   if  $s > 0$  then  
15      $b := l - 1$ ;  
16     if  $b < m$  then  $m := b$ ;  
17   else  $a := l$ ;  
18 Return  $m$ ;
```

of the) zero components are called *open*. Whenever FINDNEXT makes a guess σ , it requires to know the number of open components in which the guess coincides with the secret code, i.e. the number

$$\text{black}(\sigma, y, x) := \text{black}(\sigma, y) - \text{black}(\sigma, x).$$

Note, that the term $\text{black}(\sigma, x)$ is known by the codebreaker. After the first component of y has been found and fixed in x , there exists a $j \in \{1, \dots, n\}$ such that $\text{black}(\sigma^j, y, x) = 0$. As long as we have open components in x , we can either find a $j \in \{1, \dots, n-1\}$ with $\text{black}(\sigma^j, y, x) > 0$ but $\text{black}(\sigma^{j+1}, y, x) = 0$ and set $r := j+1$, or we have $\text{black}(\sigma^n, y, x) > 0$ but $\text{black}(\sigma^1, y, x) = 0$ and set $j := n$ and $r := 1$. Again, we call such an index j an *active* index.

Let j be an active index and r its related index. Let c be the color of some component of y that is already identified and fixed in the partial solution x . With l_j and l_r we denote the position of color c in σ^j and σ^r respectively. The peg with color c serves as a pivot peg for identifying a correct position m in σ^j that is not fixed, yet. There are two possible modes for the binary search that depend on the fact if $m \leq l_j$. The mode is indicated by a boolean variable `leftSearch` and determined by lines 4 to 8 of FINDNEXT. Clearly, $m \leq l_j$ if $l_j = n$. Otherwise, we guess

$$\sigma^{j,0} := \left(c, (\sigma_i^j)_{i=1}^{l_j-1}, (\sigma_i^j)_{i=l_j+1}^n \right),$$

By the information $\sigma_i^j = \sigma_{i+1}^r$ we obtain that $(\sigma_i^j)_{i=1}^{l_j-1} \equiv (\sigma_i^r)_{i=2}^{l_j}$. We further know that every open color has a wrong position in σ^r . For that reason, $\text{black}(\sigma^{j,0}, y, x) = 0$ implies that $m \leq l_j$. The binary search for the exact value of m is done in the interval $[a, b]$, where m is initialized as n and $[a, b]$ as

$$[a, b] := \begin{cases} [1, l_j] & \text{if } \text{leftSearch} = \text{true} \\ [l_r, n] & \text{else} \end{cases}$$

(lines 9 to 11 of FINDNEXT). In order to determine if there is an open correct component on the left side of the current center l of $[a, b]$ in σ^j we can define a case dependent permutation:

$$\sigma^{j,l} := \begin{cases} \left((\sigma_i^j)_{i=1}^{l-1}, c, (\sigma_i^j)_{i=l}^{l_j-1}, (\sigma_i^j)_{i=l_j+1}^n \right) & \text{if } \text{leftSearch} = \text{true} \\ \left((\sigma_i^r)_{i=1}^{l_r-1}, (\sigma_i^r)_{i=l_r+1}^l, c, (\sigma_i^r)_{i=l+1}^n \right) & \text{else} \end{cases}$$

Algorithm 2: Function FINDNEXT

input : Code y , partial solution $x \neq 0$ and an active index
 $j \in \{1, \dots, n\}$
output: Position m of a correct open component in σ^j

1 **if** $j = n$ **then** $r := 1$ **else** $r := j + 1$;
2 Choose a color c with identified position (a value c of some non-zero component of x);
3 Let l_j and l_r be the positions with color c in σ^j and σ^r , respectively;
4 **if** $l_j = n$ **then** leftSearch := true **else**
5 Guess $\sigma^{j,0} := \left(c, (\sigma_i^j)_{i=1}^{l_j-1}, (\sigma_i^j)_{i=l_j+1}^n\right)$;
6 $s := \text{black}(\sigma^{j,0}, y, x)$;
7 **if** $s = 0$ **then** leftSearch := true;
8 **else** leftSearch := false;
9 **if** leftSearch **then** let $a := 1$ and $b := l_j$;
10 **else** let $a := l_r$ and $b := n$;
11 $m := n$; // correct position in σ^j to be determined
12 **while** $b > a$ **do**
13 $l := \lceil \frac{a+b}{2} \rceil$; // current position for peg c
14 **if** leftSearch **then** $\sigma^{j,l} := \left((\sigma_i^j)_{i=1}^{l-1}, c, (\sigma_i^j)_{i=l}^{l_j-1}, (\sigma_i^j)_{i=l_j+1}^n\right)$;
15 **else** $\sigma^{j,l} := \left((\sigma_i^r)_{i=1}^{l_r-1}, (\sigma_i^r)_{i=l_r+1}^l, c, (\sigma_i^r)_{i=l+1}^n\right)$;
16 Guess $\sigma^{j,l}$;
17 $s := \text{black}(\sigma^{j,l}, y, x)$;
18 **if** $s > 0$ **then**
19 $b := l - 1$;
20 **if** $b < m$ **then** let $m := b$;
21 **else** $a := l$;
22 **Return** m ;

In the first case, the first $l - 1$ components of $\sigma^{j,l}$ coincide with those of σ^j . The remaining components of $\sigma^{j,l}$ cannot coincide with the corresponding components of the secret code if they have not been fixed, yet. This is because the l -th component of $\sigma^{j,l}$ has the already fixed value c , components $l + 1$ to l_j coincide with the corresponding components of σ^r which satisfies $\text{black}(\sigma^r, y, x) = 0$ and the remaining components have been checked to be wrong in this case. Thus, there is a correct open component on the left side of l in σ^j , if and only if $\text{black}(\sigma^{j,l}, y, x) \neq 0$. In the second case, the same holds for similar arguments. Now, if there is a correct open component to the left of l , we update the binary search interval $[a, b]$ by $[a, l - 1]$ and set $m := \min(m, l - 1)$. Otherwise, we update $[a, b]$ by $[l, b]$.

2.3 The Main Algorithm

The main algorithm is outlined as Algorithm 3. It starts with an empty partial solution and finds the components of the secret code y one-by-one. Herein, the vector v does keep record about the number of open components in which the permutations $\sigma^1, \dots, \sigma^n$ equal y and is, thus, initialized by $v_i := \text{black}(\sigma^i, y)$, $i \in \{1, \dots, n - 1\}$ and $v_n := n - \sum_{i=1}^{n-1} v_i$. As mentioned above, the main loop always requires an active index. For that reason, if $v = \mathbf{1}_n$ in the beginning, we fix one solution peg in σ^1 and update x and v , correspondingly. Every call of `FINDNEXT` in the main loop augments x by a correct solution value. Since one call of `findNext` requires at most $1 + \lceil \log_2 n \rceil$ guesses, Algorithm 3 does not need more than $(n - 3)\lceil \log_2 n \rceil + \frac{5}{2}n - 1$ queries (inclusive at most $\frac{n}{2} + 1$ initial and 2 final queries, respectively) to break the secret code.

3 More Colors than Components

Now, we consider the case $k > n$ and forbidden color repetition. Let $y = (y_1, \dots, y_n)$ be the code that must be found. We use the same notations as above.

3.1 Phase 1

Consider the k permutations $\bar{\sigma}^1, \dots, \bar{\sigma}^k$, where $\bar{\sigma}^1$ corresponds to the identity map on $\{1, \dots, k\}$ and for $j \in \{1, \dots, k - 1\}$, we obtain $\bar{\sigma}^{j+1}$ from $\bar{\sigma}^j$ by a

Algorithm 3: Mastermind Algorithm for Permutations

```
1 Let  $y$  be the secret code and set  $x := (0, 0, \dots, 0)$ ;
2 Guess the permutations  $\sigma^i$ ,  $i \in \{1, \dots, n-1\}$  defined by (1);
3 Initialize  $v \in \{0, 1, \dots, n\}^n$  by  $v_i := \text{black}(\sigma^i, y)$ ,  $i \in \{1, \dots, n-1\}$ ,
    $v_n := n - \sum_{i=1}^{n-1} v_i$ ;
4 if  $v = \mathbf{1}_n$  then
5    $j := 1$ ;
6   Find the position  $m$  of the correct peg in  $\sigma^1$  by at most  $\frac{n}{2} + 1$ 
   further guesses;
7 else
8   Choose an active index  $j \in \{1, \dots, n\}$  and call FINDFIRST to find
   the position of the correct peg in  $\sigma^j$  by at most  $2\lceil \log_2 n \rceil$  further
   guesses;
9  $x_m := \sigma_m^j$ ;
10  $v_j := v_j - 1$ ;
11 while  $|\{i \in \{1, \dots, n\} \mid x_i = 0\}| > 2$  do
12   Choose an active index  $j \in \{1, \dots, n\}$ ;
13    $m := \text{FINDNEXT}(y, x, j)$ ;
14    $x_m := \sigma_m^j$ ;
15    $v_j := v_j - 1$ ;
16 Make at most two more guesses that are obtained from  $x$  by assigning
   its two remaining zero-components to the two unidentified colors;
```

circular shift to the right. We define k codes $\sigma^1, \dots, \sigma^k$ by $\sigma^j = (\bar{\sigma}_i^j)_{i=1}^n$, $j \in \{1, \dots, k\}$, i.e. we set

$$\begin{aligned}\sigma^1 &= (1, 2, \dots, n), \\ \sigma^2 &= (2, \dots, n-1, n, n+1), \\ &\dots, \\ \sigma^{k-n+1} &= (k-n+1, k-n+2, \dots, k-1, k), \\ \sigma^{k-n+2} &= (k-n+2, k-n+3, \dots, k, 1). \\ &\dots, \\ \sigma^{k-1} &= (k-1, k, 1, \dots, n-3, n-2), \\ \sigma^k &= (k, 1, \dots, n-1).\end{aligned}$$

Within those k codes, every color appears exactly once at every position and, thus, we have

$$\sum_{j=1}^k \text{black}(\sigma^j, y) = n,$$

similar to (2). Since $k > n$, this implies that

Lemma 1. *There is a $j \in \{1, \dots, k\}$ such that $\text{black}(\sigma^j, y) = 0$.*

3.2 The Main Algorithm

We are able to apply the routines described in Section 2 by replacing the $n-1$ initial queries with the new $k-1$ queries $\sigma^1, \dots, \sigma^{k-1}$ and initializing the vector v correspondingly (lines 2 and 3 of Algorithm 3). By Lemma 1, the case $v = \mathbb{1}$ will not appear and we can always apply FINDFIRST to identify the first correct value. For the required number of queries to break the secret code we have: The initial $k-1$ guesses, one call of FINDFIRST to detect the first correct position (at most $2\lceil \log_2 n \rceil$ guesses), a call of FINDNEXT for every other but the last two positions (at most $1 + \lceil \log_2 n \rceil$ guesses per position) and one or two final guesses. This yields

Theorem 2. *The modified Mastermind Algorithm for Permutations breaks the secret code in at most $(n-1)\lceil \log_2 n \rceil + k + n - 2$ queries.*

4 Conclusions and Further Work

In this paper we presented a deterministic algorithm for the identification of a secret code in "Permutation Mastermind". A challenge of Permutation Mastermind is that no color repetition is allowed for a query while most strategies for other Mastermind variants exploit the property of color repetition. Further, concerning Permutation Mastermind, we consider for the first time the case that the number of colors is greater than the code length. The provided Algorithms were implemented in Matlab and tested for $n, k \leq 1000$.

In [15] it is mentioned that a trivial lower bound on the worst case number of queries for Permutation Mastermind is n , but the authors conjecture that this number is actually $\Omega(n \log n)$, a proof of which would close the gap to the upper bound. The authors show that the search space reduces at most by a factor of e after the first guess and point out that a similar assertion for the whole sequence of queries would yield the desired lower bound due to Stirling's formula for $n!$. Experimentally, we checked the worst case search space reduction after the first two queries to be less than e^2 for $n \leq 9$. Finding general theoretical formulas w.r.t. this matter remains a very ambiguous task. However, further experiments may not only increase evidence about the lower bound conjecture but probably will even give ideas concerning expedient analysis steps.

[SODA 2013] Benjamin Doerr, Reto Spöhel, Henning Thomas, and Carola Winzen. Playing Mastermind with Many Colors In: Proc. of ACM-SIAM Symposium on Discrete Algorithms (SODA 2013), pages 695-704, SIAM Society for Industrial and Applied Mathematics, 2013. arXiv version.

References

- [1] L. Berghman, D. Goossens, and R. Leus. Efficient solution for mastermind using genetic algorithms. *Computer Research*, 36(6):1880–1885, 2009.
- [2] Z. Chen, C. Cunha, and S. Homer. Finding a hidden code by asking questions. In *Proc. of the 22nd Conference on Computing and Combinatorics (COCOON'96)*, pages 50–56. Springer, 1996.
- [3] V. Chvátal. Mastermind. *Combinatorica*, 3:325–329, 1983.

- [4] B. Doerr, R. Spöhel, H. Thomas, and C. Winzen. Playing Mastermind with Many Colors. *In: Proc. of ACM-SIAM Symposium on Discrete Algorithms (SODA 2013)*, pages 695–704. SIAM Society for Industrial and Applied Mathematics, 2013.
- [5] B. Doerr, and C. Winzen. Playing Mastermind with constant-size memory. *In Proc. of the symposium on Theoretical Aspects of Computer Science (STACS’12)*, pages 441–452. Schloss Dagstuhl-leibniz-Zentrum für Informatik, 2012.
- [6] P. Erdős and C. Rényi. On Two Problems in Information Theory. *Publications of the Mathematical Institute of the Hungarian Academy of Science*, 8:229–242, 1963.
- [7] R. Focardi and F.I. Luccio. Cracking bank pins by playing Mastermind. *In proc. of the 5th international conference on Fun with algorithms (FUN’10)*, pages 202–213. Springer, 2010.
- [8] J. J. M. Guervós, C. Cotta, and A. M. Gacia. Improving and scaling evolutionary approaches to the Mastermind problem. *In proc. of Applications of Evolutionary Computation (EvoApplications’11)*, pages 103–112. Springer, 2011.
- [9] J. J. M. Guervós, A. M. Mora, and C. Cotta. Optimizing worst case scenario in evolutionary solution to the Mastermind puzzle. *In Proc. of IEEE Congress on Evolutionary Computation (CEC’11)*, pages 2669–2676. IEEE, 2011.
- [10] M. T. Goodrich. The Mastermind attack on geonomic data. *In Proc. of the 2009 30th IEEE Symposium on Security and Privacy (SP’09)*, pages 204–218. IEEE, 2009.
- [11] M. T. Goodrich. On the algorithmic complexity of the Mastermind game with black-peg results. *Information Processing Letters*, 109:675–678, 2009.
- [12] G. Jäger and M. Peczarski. The number of pessimistic guesses in generalized Mastermind. *Information Processing Letters*, 109:635–641, 2009.

- [13] G. Jäger and M. Peczarski. The number of pessimistic guesses in generalized black-peg Mastermind. *Information Processing Letters*, 111:933–940, 2011.
- [14] D. E. Knuth. The computer as a master mind. *Journal of recreational Mathematics*, 9:1–5, 1977.
- [15] K. Ko and S. Teng. On the Number of queries necessary to identify a permutation. *Journal of Algorithms*, 7:449–462, 1986.
- [16] T. Kalisker and D. Camens. An optimal Mastermind using genetic algorithms. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'03)*, pages 1590–1591. ACM, 2003.
- [17] K. Koyama and T. W. Lai. An optimal Mastermind strategy. *Journal of Recreational Mathematics*, 25:251–256, 1993.
- [18] J. Stuckman and G. Zhang. Mastermind is \mathcal{NP} -complete. *INFOCOMP Journal of Computer Science*, 5:25–28, 2006.
- [19] A. Temporel and T. Kovacs. A heuristics hill climbing algorithm for Mastermind. In *proc. of the 2003 UK Workshop on Computational Intelligence (UKCT'03)*, pages 189–196. Springer, 2003.
- [20] G. Viglietta. Hardness of Mastermind. In *Proc. of the 6th International Conference on Fun with Algorithms (FUN'12)*, pages 368–378. Springer, 2012.