Which patterns are hard to find?

Richard Cole * *Ramesh Hariharan* * *Michael S. Paterson* † *Uri Zwick* ‡

Abstract

The paper considers the exact number of character comparisons needed to find all occurrences of a pattern of length m in a text of length n using on-line and general algorithms. For on-line algorithms, a lower bound of about $(1 + \frac{9}{4(m+1)}) \cdot n$ character comparisons is obtained. For general algorithms, a lower bound of about $(1 + \frac{2}{m+3}) \cdot n$ character comparisons is obtained. These lower bounds complement an on-line upper bound of about $(1 + \frac{8}{3(m+1)}) \cdot n$ comparisons obtained recently by Cole and Hariharan. The lower bounds are obtained by finding patterns with interesting combinatorial properties (these are the hard to find patterns). It is also shown that for some patterns off-line algorithms can be more efficient than on-line algorithms.

1 Introduction

The classical *string matching* problem is the problem of finding all occurrences of a pattern $w[1 \ldots m]$ in a text $t[1 \ldots n]$. String matching is among the most extensively studied problems in computer science. A survey of the various algorithms devised for it can be found in [Ah90].

Among the most efficient algorithms devised for string matching are algorithms that gain information about the pattern and text only by performing comparisons between pattern and text characters. Such algorithms need not have any prior knowledge of the (possibly infinite) alphabet from which the pattern and text are drawn. We investigate the exact comparison complexity of string matching in this model and obtain lower bounds on the number of comparisons required (in the worst case). These lower bounds allow the algorithms to preprocess the pattern (but not the text). The lower bounds remain valid even if the algorithms do know the alphabet in advance provided that the alphabet contains a character not appearing in the pattern.

Two kinds of comparison based algorithms have been studied. An *on-line* algorithm is an algorithm that examines text characters only in a window of size m sliding monotonically to the right; furthermore, the window can slide to the right only when all matching pattern instances to the left of the window or aligned with the window have been discovered. A general (or *off-line*) algorithm is an algorithm that can access both the pattern and the text in an unrestricted manner.

Perhaps the most widely known linear time algorithms for string matching are the Knuth-Morris-Pratt [KMP77] and Boyer-Moore [BM77] algorithms. We refer to them as the KMP and BM algorithms, respectively. The KMP algorithm makes at most $2n-m$ comparisons and this bound is tight. Cole [Cole91] has recently shown that the BM algorithm performs at most $3n$ comparisons. Recently, Galil and Giancarlo [CGG90],[GG92] analyzed and modified a string matching algorithm designed by Colussi [Coll91] and showed that it makes at most $\frac{4}{3}n$ comparisons. Galil

*Courant Institute, New York University, New York 10012. The first two authors were supported in part by NSF grants CCR-8902221, CCR-8906949, CCR-9202900 and CCR-8901484.

†Department of Computer Science, University of Warwick, Coventry CV4 7AL, England. This author was supported in part by the ESPRIT II BRA Programme of the EC under contracts #3075 (ALCOM) and #7141 (ALCOM II). A part of this work was carried out while this author was visiting Tel Aviv University.

‡Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. A part of this work was carried out while this author was visiting the University of Warwick.

and Giancarlo [GG91] have also shown that any *on-line* algorithm for string matching must perform at least $\frac{4}{3}n - O(1)$ comparisons for some strings (the string `aba` is an example). We show here that this lower bound also applies to general algorithms, if only pattern-text comparisons are allowed.

The algorithm of Galil and Giancarlo [GG92] is efficient for relatively short patterns. It may become inefficient for longer patterns. Breslauer and Galil [BG92] and Cole and Hariharan [CH92] have shown that the string matching problem becomes easier as the length of the pattern increases. Breslauer and Galil [BG92] developed an algorithm that performs at most $(1+O(\frac{\log m}{m}))\cdot n$ character comparisons for texts of length n and patterns of length m. Cole and Hariharan [CH92] improved this by obtaining an algorithm that performs at most $(1+\frac{8}{3(m+1)})\cdot n$ comparisons for patterns of length m. This as we shall see is almost tight.

Galil and Giancarlo [GG91] showed that any on-line algorithm must perform at least $(1+\frac{2}{m+3})\cdot n - O(1)$ comparisons for some patterns of odd length m, and that any (general) algorithm must perform at least $(1+\frac{1}{2m})\cdot n - O(1)$ comparisons for some patterns of length m. We improve these lower bounds and obtain a lower bound of $(1+\frac{16}{7m+27})\cdot n - O(1)$ comparisons for on-line algorithms that do not use text-text comparisons, a lower bound of $(1+\frac{9}{4(m+1)})\cdot n - O(1)$ for on-line algorithms that are allowed to use text-text comparisons, and a lower bound of $(1+\frac{2}{m+3})\cdot n - O(1)$ comparisons for off-line algorithms that are allowed to use text-text comparisons. We suspect that text-text comparisons can improve the efficiency of string matching algorithms for some patterns.

In the next section we explain, in more detail, the rules of the string matching game in the comparison model setting. In section 3 we describe the adversary arguments that lie at the heart of our lower bound proofs. The off-line lower bounds presented in Section 4 follow almost immediately from the arguments of Section 3. A specific lower bound is obtained for every pattern. This lower bound depends on the first and second periods of the pattern (see next section). These off-line lower bounds are shown to be tight for an interesting family of patterns. Exploiting the additional restrictions placed on on-line algorithms we obtain, in Sections 5 and 6, improved on-line lower bounds. The lower bound of Section 5 depends again on the first and second periods of the patterns. Additional periods and more complicated combinatorial structures are used in Section 6. Finally, in Section 8 we exhibit a pattern (`abaa`) for which off-line (or even on-line algorithms with a small look-ahead) may perform better than on-line algorithms.

Preliminary versions of the on-line and off-line lower bounds that depend on the first and second periods of the patterns appeared in [ZP91]. Lower bounds of about $(1+\frac{11}{5(m+1)})\cdot n$ comparisons for on-line algorithms and about $(1+\frac{2}{m+3})\cdot n$ comparisons for off-line algorithms were obtained in [CH92] for a model in which only pattern-text comparisons are allowed. The lower bounds presented here are sharper for many patterns than those given in [CH92].

2 Preliminaries

The algorithms we consider are allowed to access the text and the pattern only through queries of the form "$t[i] = w[j]$?" or "$t[i] = t[j]$?". To each such query the algorithm is supplied with a 'yes' or 'no' answer. An algorithm is charged only for the queries it makes; all other computations are free of charge. Algorithms may adaptively choose their queries depending on the answers to earlier queries.

For a string w, let $c(w)$ denote the minimal constant for which there exists a string matching algorithm that finds all occurrences of the pattern w in a text of length n using at most $c(w)\cdot n + o(n)$ comparisons (between text and pattern characters and between pairs of text characters). A variant of $c(w)$ is $c^*(w)$ in which the algorithm is not allowed to compare pairs of text characters. Obviously $c(w) \leq c^*(w)$.

In the definition of $c(w)$ and $c^*(w)$, we allow unrestricted *off-line* algorithms that have random access to all the characters of the text. In contrast, we define $c_k(w)$ and $c_k^*(w)$ to be the corresponding minimal constants when the algorithms have access to the text only through a sliding window of size $|w|+k$, where $|w|$ denotes the length of w. Furthermore, the algorithm is only allowed to slide the window past a text position when it has already reported whether an occurrence of the pattern starts at that text position. Algorithms using a sliding window of size $|w|$ (i.e., $k=0$) are traditionally called *on-line* algorithms. We call algorithms that use larger windows *finite look-ahead* or *window* algorithms. Clearly $c(w) \leq c_k(w) \leq c_0(w)$ for any $k \geq 0$. We show in Section 8 that, for some w and some k, $c_k(w) < c_0(w)$. More specifically, we show there that $c_4(\texttt{abaa}) < c_0(\texttt{abaa})$. This means that, for some patterns, algorithms that use larger windows may be more efficient than all algorithms that use smaller windows. It is still an open problem whether there exists a string w for which $c(w) < c_k(w)$ for every $k \geq 0$. That is, it is not known whether there exist strings for which an optimal off-line algorithm is better than any finite look-ahead algorithm. It is easy to see however that $c_k(w)$ is non-increasing in k and that $\lim_{k\to\infty} c_k(w) = c(w)$.

Let w be a string of length m. We say that z, $1 \leq z \leq m$, is *a period* of w if and only if $w[i] = w[i+z]$ for every i, $1 \leq i \leq m-z$. Let z_1 be the minimal period of w (m is always a period of w so a minimal period exists). Let z_2 be the minimal period of w which is not divisible by z_1. If such a second period does not exist we set $z_2 = \infty$. We call z_1 *the period* of w and z_2 *the second period* of w. Periodicity properties play a major role in the sequel.

It is well known (see, e.g., [KMP77]) that if z_1 and z_2 are periods of w and if $z_1 + z_2 \leq |w| + \gcd(z_1, z_2)$ then $\gcd(z_1, z_2)$ is also a period of w. If z_1 and z_2 are the first and second periods of w then $\gcd(z_1, z_2)$ is not a period of w and, as a consequence, $z_1 + z_2 \geq |w| + 2$.

3 Adversary arguments

Our lower bounds are derived using an adversary that fills in the text while answering the algorithm's queries. The adversary always 'tiles' the text with (overlapping) occurrences of the pattern. Every character of the text eventually becomes part of an occurrence, which the algorithm must find. Consequently the algorithm must establish the identity of each text character and it can achieve this only by getting at least one 'yes' answer for each position. The adversary tries to avoid giving 'yes' answers whenever possible. It gives a 'yes' answer only when a 'no' answer would either contradict a previous answer or prevent it from completely tiling the text. The arguments of this section are generalizations of arguments of Galil and Giancarlo [GG91].

Lemma 3.1 *A comparison based algorithm can be certain about the identity of s text characters in a text t only after receiving at least s 'yes' answers.*

Proof : The Lemma follows from the fact that the inequality relation is not transitive. The exact details are omitted. □

Next we describe a scheme using which the adversary can give any algorithm a relatively large number of 'no' answers. We begin with a definition.

Definition 3.2 Let w be a string. A family $\mathcal{F} = \{t_v : v \in \{0,1\}^r\}$ of texts of length n is said to be *r-separating* for w if there exist indices $u_1^0, \ldots, u_r^0$ and $u_1^1, \ldots, u_r^1$ such that

1. The text t_v, for every $v = (v_1, \ldots, v_r) \in \{0,1\}^r$, contains occurrences of w starting at positions $u_1^{v_1}, \ldots, u_r^{v_r}$, but not at positions $u_1^{\overline{v_1}}, \ldots, u_r^{\overline{v_r}}$ (where $\overline{x} = 1 - x$ is the complement of x).

2. The answer to any query of the form '$w[i] = t[j]$?' is either 'yes' for all texts $t_v \in \mathcal{F}$, or 'no' for all texts $t_v \in \mathcal{F}$, or 'yes' for a text $t_v \in \mathcal{F}$ if and only if $v_k = \varepsilon$, for some fixed k and fixed $\varepsilon \in \{0,1\}$.

A very simple example of a separating family for aba may be obtained as follows. Let $n = 3r + 1$ for some $r \geq 1$. Place a's in positions $3j$ of all texts t_v for $0 \leq j < r$. In positions $3j+1, 3j+2$ of t_v, put ba if $v_j = 0$ and ab if $v_j = 1$ (for simplicity, we number the positions here from 0). This family may be depicted schematically as:

$$\ldots \; \mathtt{a} \begin{matrix}\mathtt{ba}\\ \mathtt{ab}\end{matrix} \mathtt{a} \begin{matrix}\mathtt{ba}\\ \mathtt{ab}\end{matrix} \mathtt{a} \begin{matrix}\mathtt{ba}\\ \mathtt{ab}\end{matrix} \mathtt{a} \; \ldots$$

The following Lemma is easily established. Its proof is omitted.

Lemma 3.3 *If $\mathcal{F} = \{t_v : v \in \{0,1\}^r\}$ is an r-separating family then the answer to a text-text query '$t[i] = t[j]$?' is either 'yes' in every text $t_v \in \mathcal{F}$, or 'no' in every text $t_v \in \mathcal{F}$, or 'yes' in t_v if and only if $v_{k_1} = \varepsilon_1$, or 'yes' in t_v if and only if $v_{k_1} \oplus v_{k_2} = \varepsilon_1$, or 'yes' in t_v if and only if $v_{k_1} = \varepsilon_1$ and $v_{k_2} = \varepsilon_2$, for some fixed k_1, k_2 and fixed $\varepsilon_1, \varepsilon_2 \in \{0,1\}$.*

In the example given after Definition 3.2, there is no text-text query whose answer is 'yes' if and only if $v_{k_1} = \varepsilon_1$ and $v_{k_2} = \varepsilon_2$, for some fixed k_1, k_2, $k_1 \neq k_2$, and $\varepsilon_1, \varepsilon_2 \in \{0,1\}$. Such a situation may arise however for patterns w that contain more than two distinct characters.

We are now ready to prove:

Lemma 3.4 *If $\mathcal{F}$ is an r-separating family for w, then for any comparison-based algorithm for w there exists a text $t_v \in \mathcal{F}$ for which the algorithm receives at least r 'no' answers before being able to locate all the occurrences of w in t_v.*

Proof : The adversary maintains a set E of linear equations, over the binary field GF(2), in the variables $v_1, \ldots, v_r$. At any stage, there is at least one vector $v \in \{0,1\}^r$ that satisfies all the equations in E, and if a vector $v \in \{0,1\}^r$ satisfies all the equations in E, then the text t_v is consistent with all answers given so far by the adversary. Further, the number of equations in E is at most the number of 'no' answers given by the adversary. At the beginning, $E = \phi$ and, as no query has been made, all texts are still possible. This is how the adversary responds to a new query:

If the answer to the query is the same for all texts t_v for which v is a solution of E, the adversary responds with this common answer. The set E remains unchanged and all the invariants remain satisfied.

Otherwise, the adversary answers with a 'no'. It then adds an equation to E in the following way. As the answer to the current query is not the same for all the texts in $\mathcal{F}$, there exists, by Definition 3.2 and by Lemma 3.3, either a single equation e_1 or two equations e_1 and e_2 such that the answer to the query is 'yes' in t_v if and only if v satisfies e_1, or both e_1 and e_2.

If the answer to the query, according to t_v, is 'yes' if and only if e_1 is satisfied, then $\overline{e_1}$, the equation obtained from e_1 by complementing its free coefficient, is added to E. If the answer to the query, according to t_v, is 'yes' if and only if both e_1 and e_2 are satisfied, then at least one of e_1 and e_2 is independent of the equations of E, as otherwise the answer would have been the same for all surviving texts. If e_1 does not depend on E then the equation $\overline{e_1}$ is added to E, otherwise $\overline{e_2}$ is added. It is easy to verify that all the required invariants are still satisfied.

The algorithm's task is finished only when there is a unique solution to E. This happens only when the set E contains at least r equations. An equation is added to E only as a result of a 'no' answer. The adversary can therefore give the algorithm at least r 'no' answers. □

Lemmas 3.1 and 3.4 can be combined to give a lower bound of $n + r$ if, in every text t_v of the separating family $\mathcal{F}$ used in Lemma 3.4, every text position is covered by an occurrence of the pattern w. Such separating families will be constructed in the next section.

4 Off-line lower bounds

Theorem 4.1 *If w is a string and z_1, z_2 are its first and second periods then $c(w) \geq 1 + \frac{1}{z_1 + z_2}$.*

Proof : Assume, without loss of generality, that $n = r(z_1 + z_2) + |w|$ for some $r \geq 1$. For every $v \in \{0,1\}^r$ construct a text t_v of length n in which occurrences of w start at $j(z_1+z_2)$, and either at $j(z_1+z_2)+z_1$ if $v_j = 0$, or at $j(z_1+z_2)+z_2$ if $v_j = 1$, for every j, $0 \leq j < r$. It is easy to verify that $\mathcal{F} = \{t_v : v \in \{0,1\}^r\}$ is an r-separating family for w where $u_j^0 = j(z_1+z_2)+z_1$ and $u_j^1 = j(z_1+z_2)+z_2$, for every $0 \leq j < r$. This construction is depicted in Figure 1 (note that $z_1 + z_2 \geq |w| + 2$). Consider now a comparison-based algorithm A that finds all occurrences of w in a string of length n. According to Lemma 3.4, A gets at least r 'no' answers for at least one text t_{v_0}, where $v_0 \in \{0,1\}^r$. Every text t_v, and in particular t_{v_0}, is completely covered with occurrences of w. Thus, according to Lemma 3.1, A must also get at least n 'yes' answers on t_{v_0}. Thus, in total, A must make, in the worst case, at least $n + r = (1 + \frac{1}{z_1+z_2})n - \frac{|w|}{z_1+z_2}$ comparisons for a text of length n. □

As an example, note that for the string `aba`, $z_1 = 2$ and $z_2 = 3$ and therefore $c(\texttt{aba}) \geq \frac{6}{5}$. The separating family used to obtain this lower bound may schematically be depicted as:

$$\ldots \texttt{aba} \begin{matrix}\texttt{ba}\\\texttt{ab}\end{matrix} \texttt{aba} \begin{matrix}\texttt{ba}\\\texttt{ab}\end{matrix} \texttt{aba} \begin{matrix}\texttt{ba}\\\texttt{ab}\end{matrix} \texttt{aba} \ldots$$

This family has the property that in every text of the family, every position is covered by an occurrence of `aba`. The separating family given after Definition 3.2 did not have this property.

As a further example, note that for the string `abaa` we have $z_1 = 3$ and $z_2 = 4$, and therefore $c(\texttt{abaa}) \geq \frac{8}{7}$. In Section 8 it will be shown that this bound is tight, i.e., $c(\texttt{abaa}) = \frac{8}{7}$. We will see however (Theorem 7.1) that $c_0(\texttt{abaa}) = \frac{5}{4}$. This gives us an example of a string for which off-line algorithms can be more efficient than on-line algorithms.

Theorem 4.2 *If w is a string, z_1, z_2 are its first and second periods, and $2z_2 - z_1 \leq |w|$ then $c(w) \geq 1 + \frac{1}{z_2}$.*

Proof : The proof is very similar to the proof of Theorem 4.1. A separating family, in which every text is almost completely tiled with occurrences of w, may be obtained this time without using occurrences of w that are common to all the texts of the family. We leave Figure 2 as a hint and omit the details. □

Theorem 4.3 $c^*(\texttt{aba}) = \frac{4}{3}$.

Proof : The upper bound will follow from Theorem 7.1. The lower bound does not follow from Theorem 4.2 as the condition $2z_2 - z_1 \leq |w|$ is not satisfied. A specialized argument is needed in this case. The argument given here assumes that only pattern-text comparisons are allowed. It does not seem to extend in a simple manner to the case in which both pattern-text and text-text comparisons are allowed. The lower bound is obtained using the separating family for `aba` given after Definition 3.2. A complication arises however as texts in this family are not completely covered by occurrences of `aba`.

Assume that $n = 3r + 1$ for some $r \geq 1$. The adversary starts by putting `a`'s in all text positions $3j$, for $0 \leq j < r$. It will set positions $3j+1, 3j+2$ to either `ab` or `ba` only after replying to at least one query concerning these positions with a 'no'. The adversary answers the queries of the algorithm in the following way. If the queried text position is already set by the adversary, the answer consistent with this setting is returned. If the query is '$t[3j+k] = \texttt{a}$?' or '$t[3j+k] = \texttt{b}$?' where $k = 1, 2$, and position $3j + k$ is not yet set, the adversary responds with a 'no'. It then sets positions $3j + 1, 3j + 2$ to either `ab` or `ba`, whichever is consistent with its 'no' answer.

All text positions of the form $3j + 1$ and $3j + 2$ will eventually be covered by occurrences of `aba`. The adversary therefore forces at least one 'no' answer and two 'yes' answers for each such pair. Positions of the form $3j$ might not be covered by occurrences of `aba`. But if position $3j$ is not covered by such an occurrence then positions $3j - 2, 3j - 1$ are set to `ba` and positions $3j + 1, 3j + 2$ are set to `ab`. Any algorithm must still query position $3j$ at least once, in such a case, to either

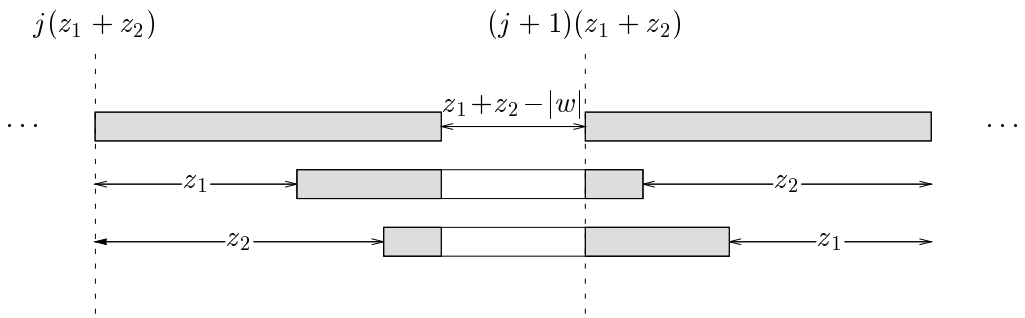


Figure 1: The configuration used to prove that $c(w) \geq 1 + \frac{1}{z_1+z_2}$

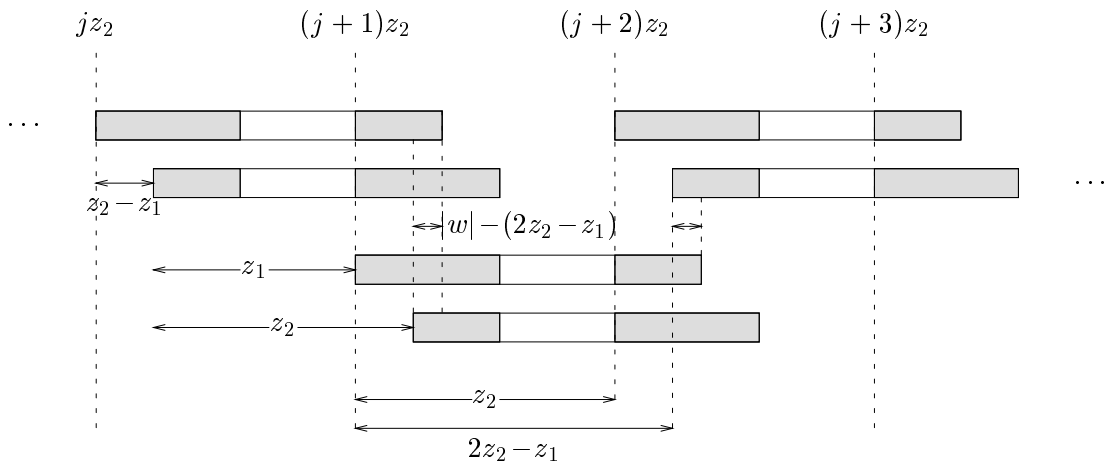


Figure 2: A configuration used to show that $c(w) \geq 1 + \frac{1}{z_2}$

verify or rule out an occurrence of aba starting at position $3j-1$. This completes the proof. □

As a corollary to Theorem 4.2 we get

Corollary 4.4 *For $k, \ell \geq 2$ we have $c(\mathtt{a}^k\mathtt{ba}^\ell) \geq 1 + \frac{1}{\max\{k,\ell\}+2)}$.*

Proof : It is easy to check that the first and second periods of $w = \mathtt{a}^k\mathtt{ba}^\ell$ are $z_1 = \max\{k,\ell\}+1$ and $z_2 = \max\{k,\ell\}+2$ and that $2z_2 - z_1 = \max\{k,\ell\}+3 \leq |w| = k+\ell+1$. The claim therefore follows immediately from Theorem 4.2. □

In Section 7 we show that the bounds given in Corollary 4.4 are tight. They can even be matched using on-line algorithms. As a further Corollary to Theorem 4.2 (or Corollary 4.4) we get

Corollary 4.5 *For every $m = 2k+1$, where $k \geq 2$, there exists a string $w_m (= \mathtt{a}^k\mathtt{ba}^k)$ of length m such that any algorithm that finds all the occurrences of w_m in a text of length n must make at least $\left(1 + \frac{2}{m+3}\right) n - O(1)$ comparisons in the worst case.*

We know (see the last paragraph of Section 2) that if z_1 and z_2 are the first and second periods

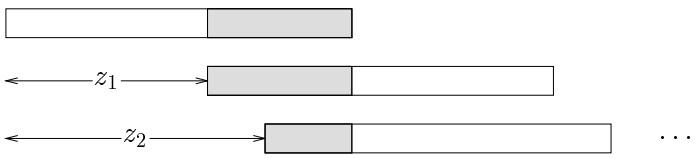


Figure 3: The configuration used to prove that $c_0(w) \geq 1 + \frac{1}{z_2}$.

of w then $z_1 + z_2 \geq |w| + 2$. As $z_2 \geq z_1 + 1$ we get that $z_2 \geq \lceil \frac{|w|+3}{2} \rceil$. Corollary 4.5 is therefore the strongest result of its kind implied by Theorem 4.2.

5 On-line lower bounds - I

In this short section we show that for on-line algorithms $c_0(w) \geq 1 + \frac{1}{z_2}$, even if the condition $2z_2 - z_1 \leq |w|$ does not hold.

Theorem 5.1 *If w is a string and z_2 is its second period then $c_0(w) \geq 1 + \frac{1}{z_2}$.*

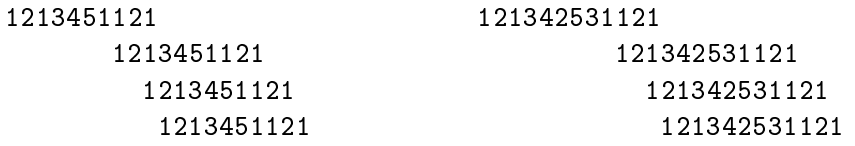
Proof : Suppose that an on-line algorithm has just found an occurrence of w in the text. The window will now be slid by at most z_1 positions to the right. Place two copies of w shifted by z_1 and z_2 positions below w, as shown in Figure 3. Denote these copies by w' and w''. Since $z_2 - z_1$ is not a period of w, the two copies w' and w'' must disagree in at least one position after the end of the found occurrence of w. The adversary will extend the found occurrence of w by either w' or w'' in a way that will force the algorithm to get at least one 'no' answer. If the algorithm makes a query whose answer is identical under both continuations, the adversary gives the algorithm this common answer. At some stage the algorithm has to make a query that distinguishes between the two incompatible continuations. No matter what this query is, the adversary answers it by 'no'. The adversary now chooses the continuation consistent with this 'no' and answers all further questions accordingly, until the algorithm finds the chosen occurrence. By then the algorithm has either made at least $z_1 + 1$ queries and can slide the window by only z_1 positions, or has made at least $z_2 + 1$ queries and can slide the window by only z_2 positions. Note that to verify an occurrence of the pattern in the text, the algorithm must get at least one 'yes' answer for each character of this occurrence. This process will be repeated again and again forcing the algorithm to make at least $(1 + \frac{1}{z_2}) \cdot n - O(1)$ queries on a text of length n. □

In the next section we use more complicated arguments to obtain better lower bounds for on-line algorithms.

6 On-line lower bounds - II

In (the proof of) Theorem 5.1 it was shown that, for every non-periodic pattern, the adversary can force any algorithm to make at least *one* mistake (i.e., get at least one 'no' answer) for each occurrence of the pattern used in the tiling of the text. Now we show that for certain patterns the adversary can force any algorithm to make at least *two* mistakes for each such occurrence. For all strings, the algorithm of Cole and Hariharan [CH92] makes at most two mistakes for each such occurrence, so no adversary can force all algorithms to make at least three mistakes per occurrence.

Theorem 6.1 *Let w be a string of length m and let $z_1 < z_2 < \ldots < z_k$ be periods of w such that for every $1 \leq i < j \leq k$, $z_j - z_i$ is not a period of w.*



(a) The set up $\langle w_{10}; 7, 9, 10\rangle$ (b) The set up $\langle w_{12}; 9, 11, 12\rangle$

Figure 4: Two simple setups in which Theorem 6.1 can be applied

(i) *If none of the multisets* $\{w[m+i-z_j] : 1 \leq j \leq k\}$, *for* $1 \leq i \leq z_1$, *contains a character exactly* $k-1$ *times, then* $c_0^*(w) \geq 1 + \frac{2}{z_k}$.

(ii) *If, in addition, none of the multisets* $\{(w[m+i_1-z_j], w[m+i_2-z_j]) : 1 \leq j \leq k\}$, *for* $1 \leq i_1 < i_2 \leq z_1$, *contains exactly* $k-1$ *equal pairs, then* $c_0(w) \geq 1 + \frac{2}{z_k}$.

Before proceeding with the proof of this theorem, we try to clarify the conditions appearing in it. Consider $k+1$ copies of w, positioned in an array of $k+1$ rows numbered $0, 1, \ldots, k$, and $m + z_k$ columns numbered $1, \ldots, m + z_k$, where the copy in the i-th row is shifted z_i positions to the right with respect to the copy in the 0-th row. Such arrays are depicted in Figure 4(a) for the string $w_{10} =$ `1213451121` with $z_1 = 7$, $z_2 = 9$ and $z_3 = 10$, and in Figure 4(b) for the string $w_{12} =$ `121342531121` with $z_1 = 9$, $z_2 = 11$ and $z_3 = 12$. A multiset $\{w[m+i-z_j] : 1 \leq j \leq k\}$ contains the k characters appearing in column $m+i$ of rows $1, \ldots, k$ in the array corresponding to w. The requirement in clause (i) above is that in each column that lies after the end of the copy of the 0-th row, but at or before the end of any of the other copies, no character appears in all but one of the rows. It is easy to check that in both cases depicted in Figure 4 this condition is satisfied. Note that when $k = 3$ this condition requires that the three characters in such a column will either be all equal or all distinct.

To check the condition of clause (ii) above, one needs to look at pairs of such columns and compare the pair of characters appearing in each row. The number j of equal pairs should satisfy $j \neq k-1$. It is easily verified that this condition is satisfied in the array of w_{10} but not in the array of w_{12}. Thus for w_{10} we obtain $c_0(w_{10}) \geq \frac{6}{5}$, while for w_{12} we can only infer $c_0^*(w_{12}) \geq \frac{7}{6}$.

Proof : The proof (of both statements) is a simple extension of the proof of Theorem 5.1. Suppose that an on-line algorithm has just found an occurrence of w in the text. The window can be slid at most z_1 positions to the right. Below w, place k copies of w shifted by $z_1, z_2, \ldots, z_k$ positions respectively (the reader may refer to Figure 3 imagining that k, instead of just two, copies appear there). Since none of $z_j - z_i$ is a period of w, each pair of copies must disagree in at least one position after the end of the found occurrence of w. The adversary will extend the found occurrence of w by one of the k copies in a way that will force the algorithm to get at least two 'no' answers. If the algorithm asks a question whose answer under the above k continuations is the same, the adversary gives the algorithm this common answer. At some stage the algorithm has to make a query to which the answer is 'yes', according to some of the continuations, and 'no' according to the rest of them. The adversary will answer this query with a 'no'. Conditions (i) and (ii) imply that at least two continuations are consistent with this 'no' answer. The adversary now gives the common answers to all queries that do not distinguish between the remaining continuations. At some stage the algorithm has to make another query to which both answers are possible. The adversary answers this again with a 'no'. At least one continuation is consistent with all the replies given by the adversary. The adversary chooses one of them and answers all subsequent queries accordingly, until the algorithm finds the next occurrence of w. By then the algorithm has

made at least $z_i + 2$ queries, for some $1 \leq i \leq k$, and it can slide the window by only z_i positions. □

We will henceforth say that $\langle w; z_1, z_2, \ldots, z_k \rangle$ is a *setup* if w is a string, $z_1 < z_2 < \ldots < z_k$ are periods of w, and none of $z_j - z_i$, for $i \neq j$, is a period of w. The string w_{10} is the shortest string for which a setup satisfying the conditions of Theorem 6.1 can be obtained. The string w_{12} is the shortest string for which a setup that satisfies condition (i), but not condition (ii), of Theorem 6.1 can be obtained. This was verified using a computer search.
Let

$$v_0 = \texttt{12121121213412156781479121212112121}$$

and $v_k = pad(v_0, 0^k)$ for $k > 0$ where $pad(v, u)$ denotes the string obtained by placing a copy of u before and after each character of v.
Let

$$u_k = \left(\left((\texttt{12})^k\,\texttt{1}\right)^2\,\texttt{3}^3\right)^2\,\texttt{12}\,\left((\texttt{12})^k\,\texttt{1}\right)^2\,\texttt{3}^3\,\left((\texttt{12})^k\,\texttt{1}\right)^2 \quad .$$

The following Lemma is easily verified.

Lemma 6.2 *(i) The setups $\langle v_k; 25(k+1), 30(k+1), 32(k+1)\rangle$, for $k \geq 0$, satisfy the conditions of Theorem 6.1.*
(ii) The setups $\langle u_k; 8k+12, 12k+17, 14k+18, 14k+20\rangle$, for $k \geq 1$, satisfy the the first condition of Theorem 6.1.

The setups $\langle v_0; 25, 30, 32\rangle$ and $\langle u_1; 20, 29, 32, 34\rangle$ are shown, as examples, in Figures 5 and 6. As $|v_k| = 36k + 35$ and $|u_k| = 16k + 19$ we get as a corollary:

Corollary 6.3 *(i) For every $m = 36k + 35$, where $k \geq 0$, there exists a string $w_m (= v_k)$ of length m such that any on-line algorithm that finds all the occurrences of w_m in a text of length n must make at least $\left(1 + \frac{9}{4(m+1)}\right) n - O(1)$ comparisons in the worst case.*
(ii) For every $m = 16k + 19$, where $k \geq 1$, there exists a string $w_m (= u_k)$ of length m such that any on-line algorithm that uses only pattern-text comparisons to find all the occurrences of w_m in a text of length n must make at least $\left(1 + \frac{16}{7m+27}\right) n - O(1)$ comparisons in the worst case.

We have verified using a computer search that no better setups with three, four or five overlaps can be obtained using strings of length at most 250. We believe that if $\langle w; z_1, \ldots, z_k\rangle$ is a setup satisfying the conditions of Theorem 6.1 then $|z_k| \geq \frac{7}{8}|w|$. If this is true, than the results of Corollary 6.3 are essentially the best that can be obtained using our methods.

7 On-line upper bounds

The next Theorem exhibits an interesting family of strings for which Theorem 5.1 is tight.

Theorem 7.1 *For every $k, \ell \geq 1$ we have $c_0(\texttt{a}^k\texttt{ba}^\ell) = c_0^*(\texttt{a}^k\texttt{ba}^\ell) = 1 + \frac{1}{\max\{k,\ell\}+2}$.*

Proof : The lower bound is a corollary of Theorem 5.1. A matching upper bound is fairly straightforward for the case $k \leq \ell$, but needs more care when $k > \ell$. We describe an algorithm that works in both cases.

Algorithm for $\texttt{a}^k\texttt{ba}^\ell$

The algorithm is described as a sequence of steps, in each of which a text character is compared to the aligned pattern character. In the case of a mismatch or if an occurrence of the pattern has been verified, the window is shifted along to the next position at which a pattern occurrence is possible. We represent the state of the algorithm before each step by an *information string* uxv describing (part of) the current knowledge the algorithm has about the text characters in the window. Here, $u \in \{0, \texttt{a}\}^k$, $v \in \{0, \texttt{a}\}^\ell$, and $x \in \{0, \texttt{A}, \texttt{b}\}$. A '0' in the information string indicates that no information is available on the corresponding position. An '`a`' (or a '`b`') indicates that the character in that position is known to be an `a` (or a `b`). An '`A`' (or a '`B`') indicates that the

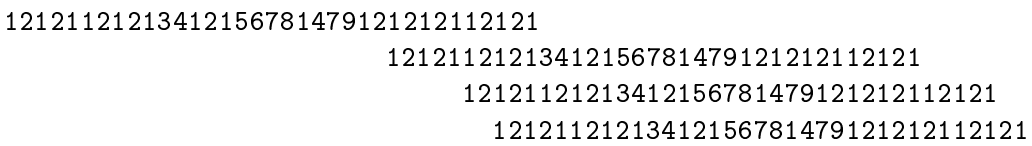


Figure 5: The setup $\langle v_0; 25, 30, 32\rangle$.

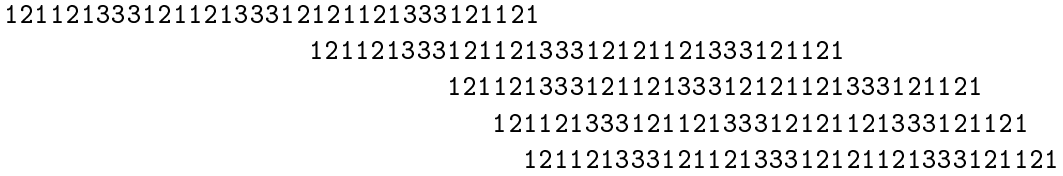


Figure 6: The setup $\langle u_1; 20, 29, 32, 34\rangle$.

character in the corresponding position is known not to be an `a` (or a `b`). The state can be written in the specified form because, after any necessary window shift, the information string must be consistent with the pattern. Our algorithm makes only '`a`?' and '`b`?' queries, and we *choose to forget* any negative information represented by '`B`'. We shall call the $(k+1)$-st position in the window the `b`-*position* and all the others `a`-*positions*. An `a`-position is always queried for an `a`. A `b`-position is always queried for a `b`.

In terms of the information strings, the algorithm is simply described.

```
IF 0 ∈ information string
    THEN query the rightmost 0 ;
    ELSE {x = A} query the b-position
;
```

This procedure is repeated until the text string is exhausted. To prove the upper bound we first establish the following pair of invariants.

(i) If $x = \mathtt{b}$ then $v = \mathtt{a}^{\ell}$.

(ii) If $x = 0$ then $\mathtt{a}^{\ell+1}$ is not a subword of u.

Invariant (i) holds because x can only become `b` after an information string of the form $u0\mathtt{a}^{\ell}$, and following any shift we again have $x \neq \mathtt{b}$.

For Invariant (ii), while $x = 0$ no tests in u are made, and the only `a`'s shifted into u come from v. These are separated from the previous contents of u by the 'x' of the previous information string.

Nearly all comparisons can be associated with text positions in the following way. Any query made at an `a`-position is associated with the corresponding text position. When $x = 0$ and the `b`-position is queried, a `b` result is associated with that text position. If the result is `B` then care is needed since, if $\ell < k$, this result will be represented as a 0 in u. However, in this case a shift of size $\ell + 1$ will be made and, by Invariant (ii), at least one 0 will be shifted out from the information string. The query is associated with the text character corresponding to any one such 0. The remaining case is when $x = \mathtt{A}$ and the `b`-position is queried. Such a query is not associated with any text position. We note that after such a query a shift of $1 + \max\{k, \ell\}$ is always made, and that the resulting information string will have $x = 0$. Since a window shift is made in any step which changes $x = 0$ to $x = \mathtt{A}$, clearly an accumulative shift of at least $2+\max\{k, \ell\}$ positions must occur between any two such 'extra' queries. The upper bound follows. □

As a corollary we get that Theorem 5.1 is also tight for all members of the $\mathtt{a}^k\mathtt{b}\mathtt{a}^{\ell}$ family to which it can be applied.

Corollary 7.2 *For* $k, \ell \geq 2$ *we have* $c(\mathtt{a}^k\mathtt{b}\mathtt{a}^{\ell}) = c_0^*(\mathtt{a}^k\mathtt{b}\mathtt{a}^{\ell}) = 1 + \frac{1}{\max\{k,\ell\}+2}$.

8 Look-ahead is useful

In the full version of the paper we present an optimal algorithm specifically tailored for the string `abaa`. The algorithm uses a window of size eight and its performance matches the general lower bound obtained for `abaa` using Theorem 4.1. Thus, $\frac{8}{7} = c(\texttt{abaa}) = c_4(\texttt{abaa}) < c_0(\texttt{abaa}) = \frac{5}{4}$ and `abaa` is therefore a string for which look-ahead is useful.

9 Concluding remarks

What is the hardest string to find? The, perhaps disappointing, answer is `aba` (or `mum` and `dad` and so on). We know that $c_0^*(\texttt{aba}) = \frac{4}{3}$ and that $c_0^*(w) \leq \frac{4}{3}$ for any string w ([GG92]). We believe that $c_0^*(w) < \frac{4}{3}$ whenever $|w| \geq 4$ but this has been established only for $|w| \geq 8$ [CH92]. This would imply that `aba`, and its like, are strictly the hardest strings to find. It is interesting to note that while we have shown here that $c^*(\texttt{aba}) = c_0^*(\texttt{aba}) = c_0(\texttt{aba}) = \frac{4}{3}$, the exact value of $c(\texttt{aba})$ is not known yet. We only know that $\frac{6}{5} \leq c(\texttt{aba}) \leq \frac{4}{3}$.

The task of computing $c_0^*(w)$ (or any of the other three variants) exactly, for a given pattern w, seems at the moment to be a very hard task. The constants $c_0^*(w), c_0(w)$ can in principle be computed algorithmically as there is only a finite, though huge, number of different on-line algorithms for every specific w. The task of finding $c(w)$ and $c^*(w)$ may be even harder. We do not know at present whether $c(w)$ and $c^*(w)$ are always rational (although it would seem odd if they were not).

A small gap still remains between our lower bounds and the upper bounds of Cole and Hariharan [CH92]. While closing this gap will have no practical value, we think that it may reveal many interesting properties of strings and string matching algorithms.

References

[Ah90] A.V. Aho, *Algorithms for finding patterns in strings,* Handbook of Theoretical Computer Science, Vol.A, ed. J. van Leeuwen, Elsevier Science Publishers B.V. (1990), pp. 257–297.

[AG86] A. Apostolico and R. Giancarlo, *The Boyer-Moore-Galil string searching strategies revisited,* SIAM Journal on Computing, 15 (1986), pp. 98-105.

[AC89] A. Apostolico and M. Crochemore, *Optimal canonization of all substrings of a string,* Technical Report No. TR 89-75, LITP, Université Paris 7, FRANCE, (1989).

[BGR90] R. Baeza-Yates, G.H. Gonnet, M. Regnier, *Analysis of Boyer-Moore type string searching algorithms,* Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, (1990), pp. 328-343.

[BG92] D. Breslauer and Z. Galil, *Efficient comparison based string matching,* Report CS-R9249 CWI, Amsterdam.

[BM77] R. Boyer and S. Moore, *A fast string matching algorithm,* CACM, 20 (1977), pp. 762-772.

[Cole91] R. Cole, *Tight bounds on the complexity of the Boyer-Moore algorithm,* Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, (1991); to appear in SIAM J. Computing.

[Coll91] L. Colussi, *Correctness and efficiency of pattern matching algorithms,* Information and Computation, 95 (1991), pp. 225-251.

[CH92] R. Cole and R. Hariharan, *Tighter upper bounds on the comparison complexity of string matching,* in preparation. A preliminary version appeared

in Proceedings of the 33rd Annual IEEE Symposium on the Foundations of Computer Science, (1992), pp. 600-609.

[CGG90] L. Colussi, Z. Galil, R. Giancarlo, *On the Exact Complexity of String Matching,* Proceedings of the 31st Annual IEEE Symposium on the Foundations of Computer Science, (1990), pp. 135-143.

[CCG92] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter, *Speeding Up Two String-Matching Algorithms,* Proceedings of STACS 1992.

[GG91] Z. Galil and R. Giancarlo, *On the Exact Complexity of String Matching: Lower Bounds,* SIAM Journal on Computing, 6 (1991), pp. 1008-1020.

[GG92] Z. Galil and R. Giancarlo. *On the Exact Complexity of String Matching: Upper Bounds,* SIAM Journal on Computing, 3 (1992), pp. 407-437.

[GO80] L.J. Guibas and A.M. Odlyzko, *A new proof of the linearity of the Boyer-Moore string searching algorithm,* SIAM Journal on Computing, 9 (1980), pp. 672-682.

[HS91] A. Hume, D. Sunday, *Fast string searching,* Software – Practice and Experience, 21 (1991), PP. 1221-1248.

[KMP77] D.E. Knuth, J. Morris, V. Pratt, *Fast pattern matching in strings,* SIAM Journal on Computing, 6 (1973), pp. 323-350.

[ZP91] U.
Zwick and M. Paterson, *Lower bounds for string matching in the sequential comparison model,* Manuscript.