Constant-space string-matching in sublinear average time

Maxime Crochemore\textsuperscript{a}, Leszek Gąsieniec\textsuperscript{b,\ast}, Wojciech Rytter\textsuperscript{b,c}

\textsuperscript{a} Institut Gaspard Monge, Universit\'e de Marne-la-Vall\'ee, France
\textsuperscript{b} Department of Computer Science, University of Liverpool, UK
\textsuperscript{c} Institute of Informatics, Warsaw University, Poland

Abstract

Given two strings: pattern $P$ of length $m$ and text $T$ of length $n$. The string-matching problem is to find all occurrences of the pattern $P$ in the text $T$. We present a string-matching algorithms which works in $o(n)$ average time and constant additional space for one-dimensional texts and two-dimensional arrays. This is a first attempt to the small-space string-matching problem in which sublinear time algorithms are achieved. We show that all occurrences of one- or two-dimensional patterns can be found in $O(n/r)$ average time with constant memory, where $r$ is the repetition size of $P$ (size of the longest repeated subword of $P$). \copyright 1999 Elsevier Science B.V. All rights reserved.

1. Introduction

The string-matching problem is defined as follows: Assume we are given two strings: pattern $P$ of length $m$ and text $T$ of length $n$. The pattern occurs at position $i$ in text $T$ iff $P = T[i \ldots i + m - 1]$. We consider algorithms that determine all occurrences of the pattern $P$ in the text $T$. The complexity of the string matching algorithm is measured by the number of symbol comparisons of pattern and text symbols. The algorithms solving string-matching problem in linear time and constant space are perhaps the most interesting ones among all designed for the entire problem. The first algorithm which uses a constant amount of additional memory was proposed by Galil and Seiferas in [9]. Later Crochemore and Perrin in [4] have presented an algorithm that achieves a smaller (at most $2n$) number of comparisons while preserving the small amount of memory. Then, another improvement ($\frac{3}{2}$) on the number of comparisons was presented by Breslauer in [2]. Two alternative algorithms were introduced by Gąsieniec et al. in [10] ($2 + \varepsilon$) and [11] ($1 + \varepsilon$).

\footnotesize{\textsuperscript{\ast} Corresponding author. E-mail: leszek@csc.liv.ac.uk. Research supported in part by NUF-NAL (The Nuffield Foundation Awards to Newly Appointed Lecturers) award.}

0304-3975/99/$-$ see front matter \copyright 1999 Elsevier Science B.V. All rights reserved.

PII: S0304-3975(98)00259-X
There are known algorithms which make a sublinear number of comparisons on the average. The first such method was proposed in [12] for strings. An attempt to 2d-dimensional pattern matching fast on the average is due to Baeza-Yates and Régnier in [1]. However, all known sublinear average time algorithms use a linear-size additional memory to maintain a table of shifts as in the Boyer-Moore algorithm, (see e.g. [12,8]), or for the representation of a directed subword graph or equivalent data structures (see e.g. [3,6]). The latter algorithms have the best-possible $O((n \log m)/m)$ average time complexity due to lower bound of Yao [13].

One can try to find a trade-off between small space and good average time applying techniques from [3] to the subwords of the pattern $P$. This may lead to an algorithm which uses $O(s)$ space (size of the preprocessed subwords) and has $O((n \log s)/s)$ average time. Until now there was no algorithm both performing an average sublinear number of comparisons and using only constant memory space.

In this paper we present the novel idea of such an algorithm for one-dimensional strings as well as for two-dimensional arrays. The idea of the algorithms is based on the use of subword repetitions.

We assume that all strings considered in the paper are built over a binary alphabet $\Sigma = \{a, b\}$. We use notation $v[i]$ to express the $i$th symbol of word $v$ and $|v|$ for its length. The index of all strings starts from 1. We say that string $v$ has a period $p$ iff $v[i] = v[i + p]$, for all $i = 1, \ldots, |v| - p$. Moreover, we say that word $v$ is periodic if it has a period of length at most $|v|/2$, otherwise $v$ is called nonperiodic.

2. One-dimensional patterns

We do not consider separate cases of periodic and nonperiodic patterns separately, due to the fact, that the number of logarithmic-size subwords in any pattern is large enough to guarantee that at least one of them is repeated. Denote by $\text{rep.size}(P)$ the length of a largest subword of $P$ which has two disjoint occurrences in $P$. The following holds.

Lemma 1. Assume the size of the alphabet is constant. Then for any pattern $P$ of size $m$ $\text{rep.size}(P) = \Omega(\log m)$.

Let $r = \text{rep.size}(P)$, and $w$ be the longest repeated subword. Assume

$$P[p \ldots p + r - 1] = P[q \ldots q + r - 1], \quad p + r - 1 < q.$$ 

The repetition description of $P$ is given by a 4-tuple $\text{REPET}(P) = (w, r, p, q)$, where $w$ stands for the longest repeated subword, $r$ for the repetition size, and $p$, $q$ are positions of $w$ occurrences in $P$.

Example 1. Let $P = \text{abbaababbaabbaababb}$ (longest repetition in bold), then:

- $\text{rep.size}(P) = 9$, and
- $\text{REPET}(P) = (w, r, p, q) = (\text{babbaabab}, 9, 2, 14)$. 

the same symbol in P at positions $p''$, $q''$

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
</tr>
</tbody>
</table>

pattern P

word w

the same word w

pattern P

Fig. 1. Testing occurrences of $P$ starting in the window $W = [i - r/2..i]$. The case when a mismatch at positions $p', q'$ is found in the checking area so there is occurrence starting in the window $W$.

We call a window any interval of consecutive $r/2$ positions $[i - r/2..i]$ in $T$, for $i > r/2$.

Assume that after a suitable preprocessing 4-tuple $(w, r, p, q)$ has been computed. In this paper we do not consider the complexity of the preprocessing.

For a position $i$ in $T$ denote by $\mathcal{C}_i =$ CheckingArea($i$) the union of two intervals

$$\mathcal{C}_i = [i + p...i + p + r/2 - 1] \cup [i + q...i + q + r/2 - 1].$$

A mismatch in CheckingArea($i$) is any pair $p', q'$ of positions in $\mathcal{C}_i$ such that

$$T[p'] \neq T[q'] \text{ and } p' - p = q' - q.$$

Denote by Leftmost_Mismatch($i$) the procedure that finds the first (from the left) mismatch position in a given checking area $\mathcal{C}_i$. If there is no such a mismatch position then a special value nil is returned.

**Lemma 2.** If Leftmost_Mismatch($i$) $\neq$ nil then no occurrence of the pattern starts in the window $[i - r/2, i]$.

**Proof.** Assume there is an occurrence of $P$ which starts in the window. Let positions $p'', q''$ be the positions (within the pattern) in this occurrence of $P$ which should match positions $p'$ and $q'$ in $T$. The positions $p'', q''$ are contained within two occurrences of the repeated subword $w$, hence we should have $P[p''] = P[q'']$, see Fig. 1. This contradicts $T[p'] = T[q']$. $\square$

Denote by Naive_Check($i$) the procedure that tests a possible occurrence of $P$ starting at a given position $i$ in $T$ and that tests the equality of corresponding symbols from left to right.
In the worst case, $m$ comparisons can be performed, but we show that for random binary texts $T$ the average time is very small. We assume that symbols of the text are uniformly distributed.

**Lemma 3.** On random texts each of the procedures Naive_Check and Leftmost_Mismatch makes on the average less than two comparisons.

**Proof.** The probability that the algorithm performs $i$ comparisons is $1/2^i$. Hence the average number of comparisons is given by the sum $\sum i/2^i$. The sum is bounded by 2. This completes the proof. $\square$

**Algorithm ID_Pattern_Searching;**

\[ r := \text{rep.size}(P); \{ r = \Omega(\log m) \} \]
\[ i := r/2 + 1; \]
\[ \text{while } r/2 < i \leq n - m \text{ do} \]
\[ \{ \text{testing occurrences starting in } [i - \frac{r}{2} \ldots i] \} \]
\[ \begin{align*}
& i_0 := \text{Leftmost_Mismatch}(i) \\
& \text{if } i_0 = \text{nil} \text{ then} \\
& \quad \text{for each } i_0 \in [i - \frac{r}{2} \ldots i] \text{ do} \\
& \quad \quad \text{Naive_Check}(i_0); \\
& i := i + r/2; \\
& \end{align*} \]

**Theorem 4.** For a random text $T$, we can find all the occurrences of $P$ in $T$ in $\mathcal{O}(n/\text{rep.size}(P))$, which is $\mathcal{O}(n/(\log m))$, average time using constant additional memory.

**Proof.** There are $\mathcal{O}(n/r)$ iterations in the algorithm ID_Pattern_Searching. Each iteration uses $\mathcal{O}(1)$ comparisons on average to compute the leftmost mismatch. The value of the leftmost mismatch is $\text{nil}$ with the probability $1/2^{r/2}$.

The total cost of employing naive check in a window of size $O(r)$ costs $O(r)$ time on average if it is activated, due to Lemma 3, since we have $O(r)$ calls to naive check, each one costing on average $O(1)$. However, the probability of activating the naive check is at most $1/2^{r/2}$, hence the average number of comparisons performed in the naive check is bounded by $r/2^{r/2} = O(1)$.

The comparisons done during different iterations can be dependent on each other, but the independence is not needed due to the following well-known fact.

**Claim (Feller [7]).** The average value of a sum of random variables is the sum of their average values.

The average cost of one iteration is $O(1)$. Therefore the algorithm makes altogether at most $O(n/r)$ comparisons on the average. Due to Lemma 1 $r = \Omega(\log n)$, hence the algorithm performs on average $O(n/\log(n))$ comparisons. $\square$
3. Two-dimensional pattern-matching

In this section we show that also for the 2d-matching problem the efficiency of a search depends on the repetition size.

Assume the pattern $P$ and the text $T$ are $m \times m$ and $n \times n$ symbol arrays, respectively. Denote $N = n^2$, $M = m^2$.

We say that the pattern occurs in $T$ at position $(i, j)$ iff $P[x, y] = T[i+x-1, j+y-1]$ for all integers $1 \leq x, y \leq m$.

A two-dimensional pattern $P$ has a period $[a, b]$ if $P[i,j] = P[i + a, j + b]$, for all $1 \leq i \leq m - a$ and $1 \leq j \leq m - b$.

If pattern $P$ has a period $[a, b]$ such that $\max\{a, b\} \leq m/2$ then it is called periodic.

Denote by row_rep_size($P$) the maximum repetition size of a row of $P$.

**Theorem 5.** Assume $P$ and $T$ are two-dimensional texts. For a random two-dimensional text $T$ there is an algorithm that finds all the occurrences of $P$ in $T$ in the average time $O(N/\text{row_rep_size}(P))$, which is $O(N/\log M)$, using constant additional memory. If pattern $P$ contains a periodic row then the algorithm performs only $O(N/m)$ comparisons.

**Proof.** We construct a two-dimensional version of the algorithm 1D_Pattern_Searching.

In the case where all rows of the pattern are nonperiodic, the algorithm takes the first row of the pattern and looks for it scanning each row of $T$ partitioned into windows of size row_rep_size($P$). For each window at least one position involves a test for an occurrence of the whole pattern. Instead of $\text{Naive\_Check}(i_0)$, a version for two dimensions $\text{2d\_Naive\_Check}(i_0, j_0)$ is used. The text is divided into $N/\text{row_rep_size}(P)$ windows, and in each of them the average number of comparisons is constant. Hence, the total number of comparisons is $O(N/\text{row_rep_size}(P))$, which is $O(N/(\log M))$ since row_rep_size($P$) = $\Omega(\log M)$, due to Lemma 1.

In the case where pattern $P$ has at least one periodic row, the algorithm chooses one such row and then searches for it the 2d-text, row by row. Each row of $T$ is partitioned into large windows of repetition size, i.e. $\geq m/2$). There are $O(N/m)$ such windows, and in each of them the algorithm makes a constant number of comparisons on the average. Hence the total number of comparisons is $O(N/m)$, which completes the proof. $\Box$

In the case of a periodic pattern $P$ the text search can be done faster.

**Theorem 6.** If the pattern $P$ is periodic the search for it in $T$ can be done in time $O(N/M)$.

**Proof.** Since the pattern $P$ is periodic it has two repeated subrectangles of size at least $m/2 \times m/2$ (see Fig. 2, and the shaded areas named $A$), which defines a set of pairs
of equal symbols of size $\Omega(M)$. We consider right bottom quadrants $D$ and $E$ of these rectangles. The two-dimensional sampling is using this set as follows. Assume that there is a pair of different symbols $(x, y)$ in the text $T$ whose positions differ exactly by a vector that is a short period in $P$. Let symbol $x$ belong to square $D$ and let $y$ belong to $E$. Then there is no any occurrence of pattern $P$ in the window $C$. Using the latter observation the text $T$ is divided into windows of size at least $m/4 \times m/4 = \Omega(M)$ (corresponding to first quadrant of $A$). The search in every window starts from the test of equality of symbols in pairs between windows $E$ and $D$. Since the text is random the algorithm makes only a constant number of tests on the average in every window, and this finally gives the $O(2/M)$ desired bound.  

Define two-dimensional repetition size of 2d-pattern $P$ ($2drep\text{-}size(P)$, in short) as the largest repeated subsquare area of $P$. Similarly to one-dimensional case the following holds.

**Theorem 7.** For a random two-dimensional text $T$ there is an algorithm that finds all the occurrences of $P$ in $T$ in $O([N/2drep\text{-}size](P))$ average time using constant additional memory.

4. Conclusion

The main result of the paper is a constant space algorithm that performs $O(n/ \log(m))$ comparisons on the average for one-dimensional as well as for two-dimensional texts. In the case of periodic patterns the average behavior of the algorithm is even better, reaching the asymptotic bound of $O(n/m)$.

Our paper initiates a discussion about pattern matching algorithms using small space and that are fast on the average. In this paper we have done some steps towards the
goal but we think that the most interesting problem is still open: what is the exact
average complexity of constant-space string matching? Or respectively: what is the
space bound needed by any algorithm making $O((n/m) \log(m))$ comparisons on the
average.

References

2nd Scandinavian Workshop on Algorithm Theory, SWAT’90, Lecture Notes in Computer Science, vol.
447, pp. 332–347.

European Symp. on Algorithms, 1993, pp. 61–72.


[8] Z. Galil, On improving the worst case running time of the Boyer–Moore string searching algorithm,
CACM 22 (1979) 505–508.


[11] L. Gasieniec, W. Plandowski, W. Rytter, Constant-space string matching with smaller number of
comparisons: sequential sampling, 6th Symposium on Combinatorial Pattern Matching CPM’95, Helsinki
1995, LNCS 937, pp. 78–89.

322–350.

368–387.