Constant-space string matching in sublinear average time

Maxime Crochemore, Leszek Gąsieniec, Wojciech Rytter

Constant-Space String-Matching in Sublinear Average Time

(Extended Abstract)

MAXIME CROCHEMORE*
Universitè de Marne-la-Vallèe

LESZEK GĄSIENIEC†
Max-Planck Institut für Informatik

WOJCIECH RYTTER‡
Warsaw University
and
University of Liverpool

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 simple string-matching algorithms which works in average $o(n)$ time with constant additional space for one-dimensional texts and two-dimensional arrays. This is the first attempt to the small-space string-matching problem in which sublinear time algorithms are delivered. More precisely we show that all occurrences of one- or two-dimensional patterns can be found in $O(\frac{n}{r})$ average time with constant memory, where r is the *repetition size* (size of the longest repeated subword) of P.

*Institut Gaspard Monge, Universitè de Marne-la-Vallèe, France (mac@univ-mlv.fr).

†Max-Planck Institut für Informatik, Im Stadtwald, D–66123 Saarbrücken, Germany (leszek@mpi-sb.mpg.de).

‡Institute of Informatics, Warsaw University, Poland and Department of Computer Science, University of Liverpool, U.K. Supported by the grant KBN 8T11C01208 (rytter@mimuw.edu.pl).

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 $P = T[i..i+m-1]$. 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 [8]. 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]. In the meantime, alternative algorithms were introduced by Gąsieniec, Plandowski and Rytter in [9] $(2+\varepsilon)$ and [10] $(1+\varepsilon)$.

Besides there are known algorithms which make a sublinear number of comparisons on the average. The first such method was proposed in [11] 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 keep a table of shifts as in the Boyer-Moore algorithm, (see e.g. [11], [7]), or for the representation of a directed subword graph or equivalent data structures (see e.g. [3] and [6]). The latter algorithms have the best possible $O(\frac{n \log m}{m})$ average time complexity due to lower bound of Yao [12].

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 might lead to an algorithm which uses $O(s)$ space (size of the preprocessed subwords) and has $O(\frac{n \log s}{s})$ 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.

For the simplicity of the presentation we assume that all strings considered in the paper are built over a binary alphabet $\Sigma = \{\mathrm{a}, \mathrm{b}\}$.

We say that the word $w \in \Sigma^*$ has *a period* q $(0 < q \leq |w|)$ if $w[i] = w[i+q]$ for

all positions $1 \leq i \leq |w| - q$.

The shortest period of w is called *the period* of w. If it satisfies $q \leq |w|/2$, then the word w is called *periodic*; otherwise, w is called *nonperiodic*.

2 Nonperiodic one-dimensional patterns

In this section we assume that the pattern P is nonperiodic.

Let us denote by $rep\_size(P)$ the size of the length of a largest subword of P.

Example 1.

The repeated subword in an example text given below is indicated here in bold.

$$rep\_size(\text{a}\mathbf{babbaabab}\text{aaa}\mathbf{babbaabab}\text{ba}) = 9.$$

The number of logarithmic-size subwords of a text is large enough to guarantee that at least one of them repeats. This implies easily the following fact.

Lemma 1

For each pattern P of size m $rep\_size(P) = \Omega(\log m)$.

Denote $r = rep\_size(P)$, and let w be a longest repeated subword. Assume

$$P[p-r..p-1] = P[q-r..q-1], \quad p \leq q-r \quad \text{and} \quad P[p] \neq P[q].$$

In Example 1 we have

$$(w, r, p, q) = (babbaabab, 9, 11, 23).$$

The positions p, q are mismatches w.r.t. the repetition of the word w. In general if there are no mismatch positions based on repetition w to the right of two copies of w then we try to find them to the left reversing the string-matching process.

In case no mismatch is found neither to the right nor to the left it means that the repetition occurs at the borders of the pattern. This case is handled similarly to the periodic case discussed in the next section.

We say that a position i in T is a *mismatch position* iff $T[i+p-1] \neq T[i+q-1]$.

We call a *window* any interval of positions $[i..i+r-1]$ on the T, for $1 \leq i \leq n-r+1$.

Assume w.l.o.g. that we already know the 4-tuple (w, r, p, q).

Denote by *Leftmost_Mismatch*(W) the procedure that finds the first (from the left) mismatch position in a given window W. If there is no such a mismatch position then a special value *nil* is returned.

Lemma 2
(1). If Leftmost_Mismatch(W) $=$ *nil , no position of* P *in* T *is in* W,
(2). Otherwise, no position of P *in* T *is in* $W - \{$*Leftmost_Mismatch*$(W)\}$.

Proof:
The mismatch is used as a constant-size deterministic sample. □

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 are done, but for random binary texts T the average time is really 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 2 comparisons.

Proof: The sum $\Sigma \frac{i}{2^i}$ is bounded by 2. □

Lemma 4
Assume that pattern P *is nonperiodic. Then, for a random text* T*, we can find all the occurrences of* P *in* T *in* $O(\frac{n}{\text{rep\_size}(P)})$*, which is* $O(\frac{n}{\log m})$*, average time using constant additional memory. The worst-case running time of the algorithm is* $O(n)$.

Proof:
There are $O(n/r)$ iterations in the algorithm *Nonperiodic_Pattern_Searching* below. Each iteration uses at most 4 comparisons on the average both for execution of *Naive_Check* and *Leftmost_Mismatch*, due to Lemma 3.

The comparisons done during different iterations can be dependent on each other, but the independence is not needed according to the fact that the average value of a sum of random variables is the sum of their average value.

Therefore the algorithm makes altogether at most $O(n/r)$ comparisons on the average.

```
ALGORITHM Nonperiodic_Pattern_Searching;
{ nonperiodic pattern };
 i:= 1;
 r:= rep_size(P);
 while i ≤ n − m do
    begin
      W := [i..i + r − 1];
      i_0 := Leftmost_Mismatch(W)
      if i_0 ≠ nil then
        if Naive_Check(i_0) then
            report match at i_0;
      i:= i + r;
    end
```

Similarly to the algorithm presented in [10] we can guarantee the linear worst-case time of the algorithm *Nonperiodic_Pattern_Searching* since the shifts are based on a longest repeated subword of the pattern. This completes the proof. □

3 Periodic one-dimensional patterns

Assume now that P is periodic, so obviously its repetition size is large.

Lemma 5
If P is periodic then rep_size$(P) \geq \frac{m}{2}$.

In this situation we cannot use the approach based on 4-tuples (w, r, p, q). Thus we derive a slightly different algorithm, which is even more efficient than the one used in nonperiodic case.

Lemma 6
Assume P is periodic. Then for a random text T we can find all occurrences of P in T in $O(\frac{n}{m})$ average time using constant additional memory. The worst-case time of the algorithm is linear.

Proof:
Assume p is the period of P, where $p \leq |P|/2$. We can partition the positions in

T into disjoint consecutive *large windows*; each window consists of $m/2$ consecutive positions of T (the last one can be smaller). The first large window is $[1..m/2]$.

The algorithm makes $\frac{n}{m/2}$ iterations. We process each large window as follows. Assume that the current window is $[i+1..i+m/2]$.

Phase 1. find the rightmost mismatch in T according to the period p in the segment $[i+1..i+m]$. If a mismatch is found then switch to the next window $[i+m/2+1..i+m]$ and execute Phase 1 again, otherwise

Phase 2. search naively for an occurrence of P starting in the current window

The probability that we do not have a mismatch in Phase 1 is exponentially small, so the expected cost of the second phase is very small even if we search for the occurence naively. The expected time to find a mismatch in the first phase is $O(1)$. There are $O(n/m)$ iterations, so the total cost is as required. This completes the proof. □

The algorithm for the nonperiodic case when repetition is placed on borders is handled in the same way but with windows of size $O(r)$.

Lemma 4 and Lemma 6 imply the following result.

Theorem 7
For a random text T we can find all occurrences of P in T in $O(\frac{n}{rep\_size(P)})$ average time (which is $O(\frac{n}{\log m})$) using constant additional memory. The worst-case time of the algorithm is linear.

4 Two-dimensional pattern-matching

In this section we show that also for the 2d-pattern 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 $1 \le x, y \le m$.

A 2-dimensional pattern P has a period $[a,b]$ if $P[i,j] = P[i+a, j+b]$, for all $1 \le i \le m-a$ and $1 \le j \le m-b$.

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

Denote by $1rep\_size(P)$ the maximum repetition size of a row of P.

Theorem 8

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 time $O(\frac{N}{1rep\_size(P)})$, which is $O(\frac{N}{\log M})$), average time using constant additional memory. If P contains a periodic row then the algorithm performs only $O(\frac{N}{m})$ comparisons.

Proof:

Similarly as in 1-dimensional case we consider periodic and nonperiodic case separately. The algorithm is almost the same as for one dimension. We can construct a 2-dimensional version of the algorithm *Nonperiodic_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 $1rep\_size(P)$. For each window at least one position involves a test for an occurrence of the whole pattern. Instead of *Naive_Check*(i_0), a version for 2 dimensions *2d-Naive_Check*(i_0, j_0) is used. According to lemma 1 we have altogether $N/1rep\_size(P)$ windows, and in each of them the average number of comparisons is constant. Hence the total number of comparisons is $O(N/1rep\_size(P))$, which is $O(\frac{N}{\log M})$ since $1rep\_size(P) = \Omega(\log M)$.

In the case where pattern P has at least one periodic row, the algorithm chooses one such row and then proceeds in a similar way as in 1-dimensional case. Each row of T is partitioned into *large windows*. There are $O(\frac{N}{m})$ 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 $O(\frac{N}{m})$. This completes the proof. □

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

Theorem 9

If the pattern P is periodic the search for it in T can be done in time $O(\frac{N}{M})$.

Proof:

Since the pattern P is periodic it has two repeated subrectangles of size at least $\frac{m}{2} \times \frac{m}{2}$ (see fig. 1, 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 2-dimensional sampling is using this set as follows. Assume that there

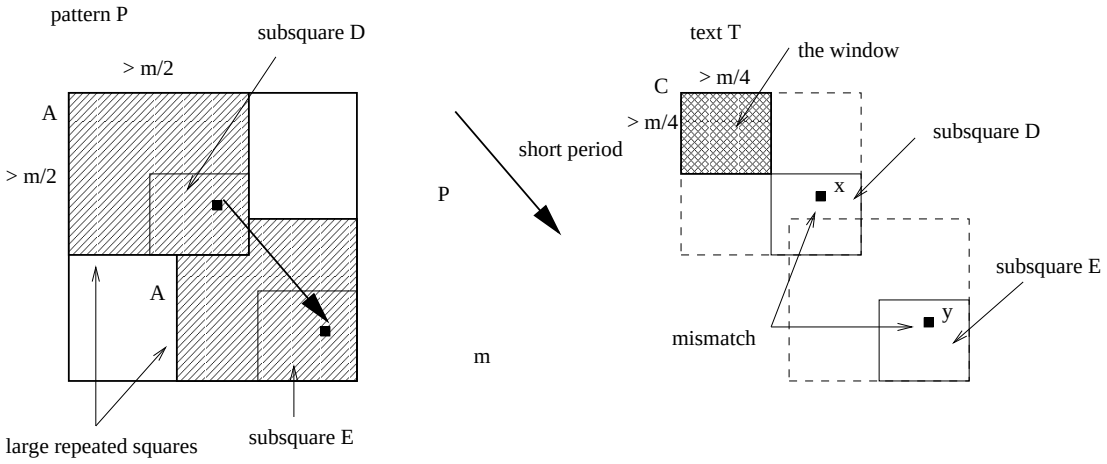

Figure 1: Sampling in 2-dimensions, if there is mismatch between position x and y then there is no occurrence of P starting in the indicated window.

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 B. Using the latter observation the text T is divided into windows of size at least $\frac{m}{4} \times \frac{m}{4} = \Omega(M)$ (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 $O(\frac{N}{M})$ desired bound. □

We can define 2-dimensional repetition size of 2d-pattern P (2drep_size(P), in short) as the largest repeated subsquare area of P. Similarly to 1-dimensional case we can prove that.

Theorem 10
For a random two-dimensional text T there is an algorithm that finds all the occurrences of P in T in $O(\frac{N}{2drep\_size(P)})$ average time using constant additional memory.

5 Summary

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 $O(\frac{n}{m})$.

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 $O(\frac{n}{m} \cdot \log(m))$ comparisons on the average.

References

[1] R. Baeza-Yates and M. Régnier, Fast Algorithms for two-dimensional and Multiple Pattern Matching, In Proc. of *2nd Scandinavian Workshop on Algorithm Theory*, SWAT'90, LNCS 447, pp. 332-347.

[2] D. Breslauer, Saving Comparisons in the Crochemore–Perrin String Matching Algorithm. In Proc. of *1st European Symp. on Algorithms*, p. 61–72, 1993.

[3] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms, *Algorithmica* (1994) 12, pp.247–267.

[4] M. Crochemore and D. Perrin, Two-way string-matching. *J. Assoc. Comput. Mach.*, 38(3), p. 651–675, 1991.

[5] M. Crochemore and W. Rytter, Periodic Prefixes in Texts. In Proc. of *Sequences'91 Workshop Sequences II: Methods in Communication, Security and Computer Science*, p. 153–165, Springer–Verlag, 1993.

[6] M. Crochemore and W. Rytter, Text algorithms. *Oxford University Press*

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

[8] Z. Galil and J. Seiferas, Time-space-optimal string matching. *J. Comput. System Sci.*, 26, p. 280–294, 1983.

[9] L. Gąsieniec, W. Plandowski and W. Rytter, The zooming method: a recursive approach to time-space efficient string-matching. *Theoret. Comput. Sci. 1996*

[10] L. Gąsieniec, W. Plandowski and W. Rytter, Sequential sampling: a new approach to constant space pattern-matching. *CPM 1995*

[11] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings. *SIAM J. Comput.*, 6, p. 322–350, 1977.

[12] A.C. Yao, The Complexity of Pattern Matching for a Random String, *SIAM Journal on Computing*, 8(3), pp. 368–387, August 1979.