“ IT’S ECONOMY, STUPID! ” : SEARCHING FOR A SUBSTRING WITH CONSTANT EXTRA SPACE COMPLEXITY

Domenico Cantone and Simone Faro
Università di Catania, Dipartimento di Matematica e Informatica
$\{$*cantone, faro*$\}$*@dmi.unict.it*

Abstract

Space and time economy are essential features of any practical algorithm. However, they are often sacrificed in favor of asymptotic efficiency. In this paper, we conduct an extensive study of the string-matching problem when no extra space is allowed. After reviewing the most relevant constant-space string-matching algorithms present in literature, we propose two new algorithms and compare their behavior with existing ones in terms of average running-time and number of character comparisons. From our experimental results it turns out that sometimes *economical* solutions are more efficient than *unrestricted* ones . . . “it’s economy, stupid!”

1. *Introduction*

Given a text T of length n and a pattern P of length m over an alphabet Σ, the *string-matching problem* consists in finding *all* occurrences of the pattern P in the text T. It is a very extensively studied problem in computer science, mainly due to its direct applications to several areas such as text processing, information retrieval, and computational biology.

The most practical string-matching algorithms show in practice a sublinear behavior at the price of using extra memory of non-constant size for auxiliary information. For instance, the Boyer-Moore algorithm [BM77] requires additional $\mathcal{O}(m + |\Sigma|)$-memory to compute two tables of shifts. Other more efficient variants use instead additional $\mathcal{O}(m)$-space [CCG$^+$94], or $\mathcal{O}(|\Sigma|)$-space [Hor80, Sun90], while, interestingly enough, two of the fastest algorithms require respectively $\mathcal{O}(|\Sigma|^2)$-space [BR99] and $\mathcal{O}(m \times |\Sigma|)$-space [CF03].

The first non-trivial constant-space string-mathching algorithm is due to Galil and Seiferas [GS77]. Their algorithm, though linear in the worst-case, was too complicated to be of any practical interest. Slightly more efficient constant-space algorithms have been subsequently reported in the literature (see [CP91, Bre93, GPR95a, GPR95b]; we will review them later).

The quite recent algorithm by Crochemore *et al.* [CGR99] is linear in the worst-case and yet has a sublinear average behavior. On the other hand, the so-called Not-So-Naive algorithm [Han92] is quite fast in practice, especially for very short patterns, despite its quadratic worst-case complexity.

In this paper we propose two new constant-space algorithms for the string-matching problem which, though quadratic, are very efficient in practice. We compare them in terms of running-time and average number of character comparisons with existing constant-space algorithms and with the Horspool algorithm [Hor80], one of the more practical variants of the Boyer-Moore algorithm, which uses non-constant extra memory. Quite surprisingly, it turns out that sometimes constant-space algorithms may be faster than those which have no memory restrictions!

The paper is organized as follows. In Section 2 we survey the known string-matching algorithms with constant extra space. Next, in Section 3 we present two new constant-space string-matching algorithms. Experimental data obtained by running under various conditions all the algorithms reviewed are presented and compared in Section 4. Finally, we draw our conclusions in Section 5.

1.1. *Preliminaries*

We introduce the notations and terminology used in the paper. A string P of length m is represented as an array $P[0..m-1]$. Thus, $P[i]$ will denote the $(i+1)$-st character of P, for $i = 0, \ldots, m-1$. For $0 \leq i \leq j <$ length(P), we denote by $P[i..j]$ the substring of P contained between the $(i+1)$-st and the $(j+1)$-st characters of P. We say that a pattern P has a *period* of length $0 < q \leq |P|$ if $P[i] = P[i+q]$ for all positions $1 \leq i \leq |P| - q$. The shortest period of P is called *the period* of P and it is denoted by $per(P)$. If $per(P) \leq |P|/2$, then the pattern P is said to be *periodic*, otherwise P is *nonperiodic*.

Next, let T be a text of length n. If the character $P[0]$ is aligned with the character $T[s]$ of the text, so that the character $P[i]$ is aligned with the character $T[s+i]$, for $i = 0, \ldots, m-1$, we say that the pattern P has *shift* s in T. In this case the substring $T[s..s+m-1]$ is called the *current window* of the text. If $T[s..s+m-1] = P$, we say that the shift s is *valid*.

Most string-matching algorithms perform a preprocessing of the pattern in order to compute useful mappings, in form of tables, which may be later accessed to compute the shift increments. Starting from the shift $s = 0$, the searching phase consists in checking whether s is a valid shift and then repeatedly computing a *positive* shift increment Δs such that no valid shift can belong to the interval $\{s+1, \ldots, s+\Delta s-1\}$.

The Naive string-matching algorithm, for instance, performs no preprocessing of the pattern P. It starts by aligning the left ends of the pattern and text. Then, for each value of the shift $s = 0, 1, \ldots, n-m$, it checks whether $P[0..m-1]$ is equal to $T[s..s+m-1]$ by simply comparing each character of the pattern with its correspondant character in the text, proceeding from left to right. At the end of the matching phase, the shift is advanced by one position to the right. In the worst-case, the Naive algorithm requires $\mathcal{O}(mn)$ character comparisons. Notice also that the Naive algorithm uses only constant extra space.

2. *Matching with constant extra space complexity*

In this section we briefly review the known string-matching algorithms which make use of only constant extra space. We will follow chronological order.

2.1. The Galil-Seiferas algorithm

The first linear-time algorithm that used a constant amount of additional space was proposed by Galil and Seiferas in [GS77]. Their algorithm requires a preprocessing phase of $\mathcal{O}(m)$-time complexity and it can be shown that its subsequent searching phase performs at most $5n$ text characters comparisons.

The Galil-Seiferas algorithm is based on the concept of *prefix period* of a string and the value of a suitable constant k (see [GS77] for details). Galil and Seiferas suggested that practically the constant k could be taken equal to 4. The preprocessing phase of the Galil-Seiferas algorithm consists in finding a *perfect factorization* $U.V$ of the pattern P, i.e. a decomposition of P such that V has at most one prefix period, say Z, and $|U| = \mathcal{O}(|Z|)$. Thus V is of the form $Z^l.Z'.a.Z''$, with Z' a prefix of Z and $Z'.a$ not a prefix of Z. The searching phase of the Galil-Seiferas algorithm consists in scanning the text T for each occurrence of V. When an occurrence of V is found, the algorithm checks naively if U occurs just before it in T.

Suppose that $|U| = u$, $|Z| = p_1$, and $|Z^l.Z'| = p_1 + q_1$. If a mismatch is found between characters $P[u+j]$ and $T[s+j]$ and $j = p_1 + q_1$ holds, then a shift of length p_1 can be performed and the comparison is resumed with character $P[u+q_1]$. Otherwise, if $j \neq p_1 + q_1$ then a shift of length $\lfloor q/k+1 \rfloor$ can be performed and the comparison is resumed with character $P[u]$.

2.2. The Two-Way algorithm

Crochemore and Perrin presented in [CP91] a constant-space string-matching algorithm which performs only $2n$ character comparisons. Their algorithm, called Two-

Way algorithm, runs in $\mathcal{O}(n)$ worst-case time complexity but requires an ordered alphabet and an $\mathcal{O}(m)$-time preprocessing phase.

In the preprocessing phase, the Two-Way algorithm factorizes the pattern P in two parts P_l and P_r in a suitable manner, so that one has $P = P_l.P_r$. Then the searching phase of the Two-Way algorithm consists in first comparing the character of P_r from left to right, then the character of P_l from right to left. If a mismatch occurs while scanning the k-th character of P_r, then a shift of length k is performed. If a mismatch occurs while scanning P_l, then a shift of length $per(P)$ is performed. The same shift of length $per(P)$ is also applied when an occurrence of the pattern is found. The length of the matching prefix of the pattern (namely $m - per(P)$) is memorized to avoid to rescan such a prefix again during the subsequent attempt.

Later, Breslauer designed in [Bre93] a variation of the Two-Way algorithm which performs less than $2n$ comparisons still using constant space. In particular he designed a $(\frac{3}{2} + \varepsilon)n$-comparisons constant-space algorithm.

2.3. The Not-So-Naive algorithm

The Not-So-Naive algorithm [Han92] is a very simple variation of the Naive algorithm that turns out to be quite efficient in some practical cases. As in the case of the Naive algorithm, the searching phase is performed by scanning the text and pattern from left to right. However, the Not-So-Naive algorithm identifies two cases in which at the end of each matching phase the shift can be advanced by two positions to the right, rather than by one as in the Naive algorithm.

Let us first assume that $P[0] \neq P[1]$. If $P[0] = T[s]$ and $P[1] = T[s+1]$, then at the end of the matching phase the shift s can be safely advanced by 2 positions, since $P[0] \neq P[1] = T[s+1]$. Let us now suppose that $P[0] = P[1]$. If $P[0] = T[s]$ but $P[1] \neq T[s+1]$, then again the shift s can be safely advanced by 2 positions.

Plainly, the needed preprocessing phase can be performed in constant space and time. Though in the worst-case the Not-So-Naive algorithm can execute $\mathcal{O}(mn)$ character comparisons during the searching phase, it turns out from empirical results that it performs quite well in practice.

2.4. The Sequential-Sampling algorithm

An alternative algorithm, called Sequential-Sampling, which performs $(2 + \varepsilon)n$ character comparisons in the worst-case, was proposed by Gąsieniec, Plandowsky, and Rytter in [GPR95a]. They later improved it in [GPR95b], by reducing the number of character comparisons to $(1 + \varepsilon)n$.

The Sequential-Sampling algorithm is based on the powerful idea of sampling, originally introduced in [Vis91]. Assume that a nonperiodic pattern P has a periodic prefix and denote by π the longest such periodic prefix. Let $q-1$ be the length of π, let per be the length of the shortest period of π, and let $p = q - per$. In the matching phase, the Sequential-Sampling algorithm first compares the characters of P at positions p and q with their corresponding characters in the text T, and then, if no mismatch is found, it applies a constant-space version of the Knuth-Morris-Pratt algorithm [KMP77].

It turns out that the Sequential-Sampling algorithm runs in $\mathcal{O}(n)$-time and makes $(1+\varepsilon)n + \mathcal{O}(\frac{n}{m})$ character comparisons in the worst-case, whereas its preprocessing phase takes $\mathcal{O}(m)$-time and makes $(1+\varepsilon)m + \mathcal{O}(\frac{1}{\varepsilon})$ comparisons.

2.5. The Dogaru algorithm

In [Dog98] a very simple string-matching algorithm was presented by Dogaru. The Dogaru algorithm does not preprocess the pattern in any way and it has an $\mathcal{O}(nm)$ worst-case time complexity.

As in the case of the Naive algorithm, during the searching phase the Dogaru algorithm scans the text and patterns from left to right. However, if a mismatch is found between characters $P[j]$ and $T[s+j]$, search continues by looking for occurrences of the character $P[j]$ which caused the mismatch within the substring $T[s+j+1..n-m+j]$. If $P[j]$ is not found, then the algorithm terminates. On the other hand, if an occurrence of character $P[j]$ is found, say at position s' of T, then the algorithm naively checks whether an occurrence of P begins at position $s'-j$ in T. The search is then resumed from position $s'+1$ of the text.

2.6. The CGR algorithm

Crochemore, Gąsieniec, and Rytter presented in [CGR99] a string-matching algorithm, here called CGR, which runs in average $o(n)$-time, using only constant additional space. This can be regarded as the first attempt to the small-space string-matching problem in which a sublinear time algorithm is delivered.

Roughly speaking, the CGR algorithm is based on the following idea. Let r be the size of the longest repeated subword of P: hence, there exist two positions p and q in P such that $P[p-r..p-1] = P[q-r..q-1]$, with $p \leq q-r$ and $P[p] \neq P[q]$. As a bit of terminology, any interval $[s..s+r-1] \subseteq [0..n-1]$ is called an r-*window* of T; in addition, we say that a position i in T is a *mismatch position* if $T[i+p-1] \neq T[i+q-1]$. Given an r-window W in T, if there is no mismatch position in W, then no occurrence of P in T is in W. Otherwise, if j is the leftmost mismatch position of W, then no occurrence of P in T is in $W - \{j\}$.

It can be shown that the CGR algorithm finds all occurrences of a pattern P in $\mathcal{O}(\frac{n}{r})$ average-time using only constant additional memory. The worst-case running-time of the CGR algorithm is $\mathcal{O}(n)$. Moreover, if the pattern P is periodic, so that $r \geq \frac{m}{2}$, it can be proved that for a random text T all occurrences of P in T can be found in $\mathcal{O}(\frac{n}{m})$ average-time using constant additional space.

3. *Two new constant-space algorithms*

In this section we present two new simple string-matching algorithms which achieve very good results in practical cases, though both of them have an $\mathcal{O}(nm)$ worst-case time complexity.

The first algorithm, called Quite-Naive, is an improvement of the Not-So-Naive algorithm and requires a preprocessing phase of $\mathcal{O}(m)$-time complexity. The second algorithm, called Tailed-Substring, does not require any preprocessing phase and performs better in most cases, especially for longer patterns.

3.1. *The* Quite-Naive *algorithm*

The Quite-Naive algorithm requires a linear-time preprocessing of the pattern in constant-space complexity and finds all occurrences of a pattern P in a text T in quadratic worst-case time. In practical cases, it performs slightly better than the Not-So-Naive algorithm, of which it is a variation.

Given a pattern P of length m, we define the following values δ and γ:

$$\delta = \min\{1 \leq j < m : P[m-1-j] = P[m-1]\} \cup \{m\}$$

$$\gamma = \min\{1 \leq j < m : P[m-1-j] \neq P[m-1]\} \cup \{m\} .$$

Such values are precomputed by the Quite-Naive algorithm. Notice that if $\delta > 1$ then $\gamma = 1$. Likewise, if $\gamma > 1$ then $\delta = 1$. Thus the preprocessing phase inspects at most $m + 1$ characters and, plainly, requires only constant-space.

The matching phase of the Quite-Naive algorithm differs from the one of the Not-So-Naive algorithm in the following two points. Firstly, as in a Boyer-Moore type algorithm [BM77], the pattern and text are scanned from right to left. Secondly, the following two cases are identificed in which the shift can be advanced by possibly more than two positions. Let us suppose that, for a particular value of the shift, the character $P[0]$ of the pattern is aligned with the character $T[s]$ of the text. Then:

```
Quite-Naive(P, T)
    n = length(T)
    m = length(P)

Preprocessing:
        γ = 1
        δ = 1
        while δ < m and P[m − 1] ≠ P[m − 1 − δ] do
            δ = δ + 1
        while γ < m and P[m − 1] = P[m − 1 − γ] do
            γ = γ + 1

Searching Phase
    s = 0
    while s ≤ n − m do
        if P[m − 1] ≠ T[s + m − 1] then s = s + γ
        else
            j = m − 2
            while j ≥ 0 and P[j] = T[s + j] do j = j − 1
            if j < 0 then print(s)
            s = s + δ
```

Figure 1: The Not-Naive algorithm

- if a mismatch occurs during the first comparison, namely if $P[m-1] \neq T[s+m-1]$, the pattern is advanced by γ positions; otherwise,

- if character $P[m-1]$ matches its corresponding character, namely if $P[m-1] = T[s+m-1]$, at the end of the matching phase the pattern is advanced by δ positions.

The code of the Quite-Naive algorithm is presented in Figure 1.

3.2. *The Tailed-Substring algorithm*

Our second constant-space algorithm, called Tailed-Substring, performs its preprocessing in parallel with the searching phase. Despite its $\mathcal{O}(nm)$-time worst-case complexity, it is very fast in practice.

The Tailed-Substring algorithm is based on the following notion of maximal tailed-substring of P. We say that a substring S of P is a *tailed-substring* if its last character is not repeated elsewhere in S. Then a *maximal tailed-substring* of P is a tailed-substring of P of maximal length.

```
    Tailed-Substring(P, T)
1.      n = length(T)
2.      m = length(P)

    Searching Phase 1:
3       s = 0
4       δ = 1
5       i = k = m − 1
6       while s ≤ n − m and i − δ ≥ 0 do
7           if P[i] ≠ T[s + i] then s = s + 1
8           else
9               j = 0
10              while j < m and P[j] = T[s + j] do j = j + 1
11              if j = m then print(s)
12              h = i − 1
13              while h ≥ 0 and P[h] ≠ P[i] do h = h − 1
14              if δ < i − h then
15                  δ = i − h
16                  k = i
17              s = s + i − h
18              i = i − 1

    Searching Phase 2:
19      while s ≤ n − m do
20          if P[k] ≠ T[s + k] then s = s + 1
21          else
22              j = 0
28              while j < m and P[j] = T[s + j] do j = j + 1
23              if j = m then print(s)
24              s = s + δ
```

Figure 2: The Tailed-Substring algorithm

In the following, given a maximal tailed-substring S of a pattern P, we associate to S its length δ and a natural number $\delta - 1 \leq k < m$ such that $S = P[k - \delta + 1..k]$.

The Tailed-Substring algorithm searches for a pattern P in a text T in two subsequent phases. During the first phase, while it searches for occurrences of P in T, the Tailed-Substring algorithm also computes values of δ and k such that $S = P[k - \delta + 1..k]$ is a maximal tailed-substring of P. During the second phase, it just uses the values for δ and k computed in the first phase to speed up the search of the remaining occurrences of P in T. The code of the Tailed-Substring algorithm is presented in Figure 2.

The first searching phase (lines 3-18) works as follows. Initially, the value of δ is set to 1 and the values of i and k are set to $m - 1$. Next, the following steps are repeated until $\delta \geq i$. The first value of the shift s such that $P[i] = T[s + i]$ is looked for (lines 6-7) and then it is checked whether $P = T[s..s + m - 1]$, proceeding from left to right (lines 9-11). At this point, the rightmost occurrence h of $P[i]$ in $P[0..i-1]$ is searched for (lines 13). If such an occurrence is found (i.e., $h \geq 0$), the algorithms aligns it with character $s + i$ of the text; otherwise, the shift is advanced by $i + 1$ positions (in this case $h = -1$). Then, if the condition $i - h \geq \delta$ holds, δ is set to $i - h$, k is set to i, and the value of i is decreased by 1. It can be shown that at the end of the first searching phase $P[k - \delta + 1..k]$ is a maximal tailed-substring of P.

In the second searching phase (lines 19-24), the algorithm looks for an occurrence of character $P[k]$ in the text. When a value s such that $P[k] = T[s + k]$ is found, it is checked whether $P = T[s..s + m - 1]$, proceeding from left to right, and then the shift is advanced by δ positions to the right. The preceding steps are repeated until all occurrences of P in T have been found.

The resulting algorithm runs in $\mathcal{O}(nm)$ worst-case time complexity but it turns out that it achieves very good results in practical cases, especially when the length of the pattern increases.

4. *Experimental results*

In this section we present and comment some experimental data relative to the following selection of string-matching algorithms discussed in the preceding sections: the Naive algorithm (NAIVE), the Two-Way algorithm (TW), the Not-So-Naive algorithm (NSN), the Dogaru algorithm (OD), the CGR algorithm (CGR), the Quite-Naive algorithm (QN), and the Tailed-Substring algorithm (TS). Experimental results for the Galil-Seiferas and Sequential-Sampling algorithms have not been reported, since they do not have good performances in practical cases. All the above algorithms have been compared in terms of their running-time and average number of character comparisons.

We have also included experimental results relative to the Horspool algorithm (HOR) [Hor80] which, though quadratic, is one of the most efficient variant of the Boyer-Moore algorithm. We recall that the Horspool algorithm uses additional memory of size $\mathcal{O}(|\Sigma|)$.

All algorithms have been implemented in the **C** programming language and tested to search for the same set of strings in large fixed text buffers on a PC with AMD Athlon processor of 1.19GHz. In particular, all algorithms have been run on four Randσ problems, for $\sigma = 2, 4, 8, 20$, and on two real world problems, NL (natural language) and Prot (protein sequence), with patterns of length $m = 2$, 4, 6, 8, 10, 20, 40, 80, and 160. We recall that each Randσ problem consists in searching a set of 200 random patterns of a given length in a 20Mb random text over a common alphabet of size σ. The tests on the natural language text buffer NL have been performed on a 180Kb file containing the english text "Hamlet" by William Shakespeare while tests on a protein sequence Prot have been performed on a 2.4Mb file containing a sequence from human genome. For real world problems the patterns to be searched for have been constructed by selecting 200 random substrings of length m from the text, for each $m = 2, 4, 6, 8, 10, 20, 40, 80, 160$.

4.1. *Running-times*

Experimental results show that the Not-So-Naive algorithm atteins the best run-time performances in the case of very small patterns. For patterns of length greater than 10, the Quite-Naive and the Tailed-Substring algorithms have better performances. In particular the Tailed-Substring algorithm achieves very good results for long patterns. In addition, we observe that (a) the Quite-Naive algorithm achieves always the second best results, (b) it is faster than the Not-So-Naive algorithm for long patterns, and (c) it is faster than the Tailed-Substring algorithm for short patterns.

We notice also that the CGR algorithm obtains the best results when it is run with very long patterns and the size of the alphabet is very small. In fact, for long random patterns, the size r of the longest repeated subword turns out to be large enough.

It is quite interesting to observe that when the alphabet is small the constant-space algorithms perform better than the Horspool algorithm. The latter achieves slightly better results when the alphabet is large and the pattern is not very short.

In the following tables, running-times are expressed in hundredths of seconds.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
NAIVE	51.43	63.62	67.07	67.74	68.26	62.20	53.85	53.17	53.08
NSN	**31.70**	**37.09**	38.51	40.04	39.84	39.06	37.57	37.81	37.23
OD	53.19	67.31	72.42	74.25	73.63	70.92	67.82	68.18	67.90
TW	59.66	50.75	45.37	43.97	41.75	38.62	38.64	38.03	37.68
CGR	58.82	70.38	59.14	51.46	45.98	31.97	24.77	**21.39**	**19.79**
QN	36.83	40.55	42.17	42.66	42.52	40.79	38.76	38.12	38.96
TS	44.06	41.38	**36.55**	**33.98**	**31.13**	**26.68**	**24.08**	22.64	21.56
HOR	43.97	44.33	45.77	45.53	45.29	42.08	40.79	39.58	40.50

Running-times for a Rand2 problem

$\sigma = 4$	2	4	6	8	10	20	40	80	160
NAIVE	44.15	45.83	45.77	45.77	44.91	41.56	41.49	41.49	41.61
NSN	**27.32**	28.16	28.65	28.39	28.56	28.23	27.77	27.87	28.11
OD	39.97	42.89	42.68	42.59	42.57	42.20	42.33	41.94	42.11
TW	46.61	40.34	38.24	37.06	37.03	35.98	34.79	34.89	34.65
CGR	53.82	64.72	59.51	50.27	45.02	32.59	26.29	22.91	20.77
QN	32.12	**27.14**	**25.98**	25.43	25.72	25.31	25.24	24.92	25.12
TS	35.75	30.07	26.36	**23.79**	**22.23**	**19.30**	**18.30**	**17.74**	**17.25**
HOR	36.54	27.15	23.42	22.03	21.45	20.43	20.67	20.22	20.89

Running-times for a Rand4 problem

$\sigma = 8$	2	4	6	8	10	20	40	80	160
NAIVE	35.90	35.99	35.94	31.56	30.71	29.75	29.02	29.04	29.01
NSN	**23.63**	23.73	24.07	23.43	23.69	23.74	23.63	23.49	23.62
OD	30.61	31.03	31.08	30.97	30.87	30.78	30.87	30.96	30.77
TW	37.97	34.71	33.57	33.16	32.78	32.18	31.18	30.99	30.72
CGR	48.90	55.22	55.21	52.20	48.72	35.20	28.53	24.30	22.21
QN	25.64	**23.32**	**21.87**	**21.57**	21.27	20.86	20.83	21.01	20.55
TS	29.50	26.11	23.98	22.23	**20.85**	**18.73**	**17.41**	**16.69**	**16.30**
HOR	28.98	20.94	18.73	17.88	17.27	16.45	16.40	16.38	16.30

Running-times for a Rand8 problem

$\sigma = 20$	2	4	6	8	10	20	40	80	160
NAIVE	31.93	31.84	27.20	26.94	25.38	25.87	25.30	26.45	25.23
NSN	**21.39**	**21.15**	**22.34**	**21.82**	21.48	21.62	21.41	21.18	21.26
OD	25.77	25.83	26.82	26.06	25.93	25.90	25.79	25.80	25.65
TW	33.78	32.50	32.93	30.98	30.72	30.11	29.14	29.09	28.86
CGR	44.41	50.29	53.85	51.86	51.92	47.22	34.30	29.31	25.56
QN	24.12	23.56	23.23	**21.82**	**21.18**	20.62	19.62	19.96	19.66
TS	26.57	24.98	24.93	23.12	23.14	**20.15**	**18.14**	**17.09**	**16.44**
HOR	24.00	18.73	17.80	16.40	16.29	15.95	15.40	15.45	15.46

Running-times for a Rand20 problem

NL	2	4	6	8	10	20	40	80	160
NAIVE	0.21	0.22	0.26	0.22	0.23	0.21	0.26	0.20	0.22
NSN	0.18	0.18	0.16	0.16	0.17	0.14	0.13	0.20	0.14
OD	0.18	0.15	0.14	0.18	0.18	0.20	0.14	0.15	0.19
TW	0.25	0.19	0.18	0.24	0.19	0.22	0.24	0.24	0.20
CGR	0.38	0.34	0.45	0.37	0.40	0.30	0.20	0.12	0.08
QN	**0.14**	**0.12**	**0.11**	**0.10**	**0.08**	**0.08**	0.12	**0.08**	0.13
TS	0.15	0.17	0.16	0.18	0.14	0.10	**0.06**	0.11	**0.11**
HOR	0.17	0.10	0.04	0.06	0.04	0.04	0.03	0.04	0.03

Running-times for a natural language problem

Prot	2	4	6	8	10	20	40	80	160
NAIVE	4.02	3.96	3.80	3.77	3.76	3.77	3.74	3.65	3.76
NSN	**2.75**	2.77	2.73	2.69	2.70	2.73	2.74	2.73	2.72
OD	3.11	3.12	3.13	3.11	3.11	3.14	3.17	3.12	3.14
TW	3.88	3.77	3.68	3.68	3.66	3.60	3.59	3.67	3.57
CGR	5.77	6.26	6.38	6.42	6.28	5.15	3.91	3.42	2.85
QN	2.77	**2.60**	**2.59**	**2.52**	**2.43**	**2.38**	2.30	2.34	2.38
TS	3.26	3.06	2.92	2.87	2.75	2.42	**2.25**	**2.18**	**2.22**
HOR	2.89	2.26	2.08	1.96	1.95	1.87	1.83	1.82	1.78

Running-times for a a protein sequence problem

4.2. *Average Number of Comparisons*

For each test, the average number of character comparisons has been obtained by taking the total number of times a text character is compared with a character in the pattern and dividing it by the total number of characters in the text buffer.

It turns out that the Quite-Naive and the Tailed-Substring algorithms achieve always very good results. In particular the Tailed-Substring algorithm achieves the best result in most cases and it performs better than the Horspool algorithm in the case of small alphabets. Notice also that when the size of the alphabet is very small the Not-So-Naive algorithm obtains the best results for short patterns whereas the CGR algorithm performs better for long patterns.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
NAIVE	1.500	1.875	1.968	1.992	1.998	2.000	2.000	2.000	2.000
NSN	1.000	1.375	1.468	1.492	1.498	1.500	1.500	1.500	1.500
OD	1.614	2.071	2.157	2.159	2.179	2.157	2.168	2.169	2.165
TW	**.9550**	**1.115**	1.088	1.070	1.026	.9433	.9743	.9695	.9713
CGR	1.766	1.816	1.548	1.331	1.184	.7544	**.5288**	**.3980**	**.3248**
QN	1.000	1.262	1.358	1.368	1.392	1.373	1.385	1.356	1.395
TS	1.480	1.308	**1.086**	**.9502**	**.8498**	**.6634**	.5526	.4877	.4412
HOR	1.166	1.171	1.153	1.113	1.117	1.073	1.101	1.066	1.099

Average number of comparisons for a Rand2 problem.

$\sigma = 4$	2	4	6	8	10	20	40	80	160
NAIVE	1.250	1.328	1.333	1.333	1.333	1.333	1.333	1.333	1.333
NSN	.9329	.9881	1.025	1.010	1.020	1.016	1.009	.9995	1.019
OD	1.295	1.377	1.382	1.383	1.382	1.385	1.383	1.381	1.383
TW	**.8948**	.9393	.9402	.9305	.9375	.9598	.9637	.9870	.9919
CGR	1.945	1.935	1.756	1.467	1.305	.8929	.6568	.5193	.4195
QN	.9329	**.8565**	.8128	.7865	.7935	.7776	.7740	.7520	.7642
TS	1.121	.8863	**.7352**	**.6214**	**.5491**	**.3943**	**.3156**	**.2765**	**.2378**
HOR	.8214	.5537	.4481	.4002	.3812	.3533	.3679	.3453	.3715

Average number of comparisons for a Rand4 problem.

$\sigma = 8$	2	4	6	8	10	20	40	80	160
NAIVE	1.125	1.142	1.142	1.142	1.142	1.142	1.142	1.142	1.142
NSN	.9540	.9716	.9819	.9639	.9739	.9739	.9659	.9579	.9679
OD	1.138	1.156	1.156	1.156	1.156	1.156	1.157	1.156	1.156
TW	**.9155**	.9344	.9346	.9381	.9399	.9678	.9849	.9905	.9932
CGR	1.987	1.978	1.909	1.790	1.654	1.125	.8547	.6654	.5615
QN	.9540	**.8421**	.7583	.7228	.7056	.6642	.6593	.6496	.6365
TS	1.030	.8680	**.7504**	**.6641**	**.5890**	**.4114**	**.2911**	**.2324**	**.1914**
HOR	.6583	.3789	.2800	.2306	.2034	.1578	.1509	.1467	.1499

Average number of comparisons for a Rand8 problem.

$\sigma = 20$	2	4	6	8	10	20	40	80	160
NAIVE	1.050	1.052	1.052	1.052	1.052	1.052	1.052	1.052	1.052
NSN	.9723	.9703	.9796	.9796	.9796	.9842	.9703	.9703	.9842
OD	1.052	1.055	1.055	1.055	1.054	1.055	1.055	1.055	1.055
TW	**.9576**	.9587	.9516	.9557	.9568	.9706	.9839	.9989	.9931
CGR	1.998	2.002	1.992	1.981	1.957	1.730	1.184	.9629	.7758
QN	.9723	**.9160**	**.8525**	.8070	.7574	.6778	.6063	.6418	.6169
TS	1.005	.9205	.8532	**.7917**	**.7380**	**.5603**	**.3835**	**.2557**	**.1906**
HOR	.5628	.2965	.2064	.1626	.1359	.0842	.0610	.0540	.0535

Average number of comparisons for a Rand20 problem.

NL	2	4	6	8	10	20	40	80	160
NAIVE	1.059	1.066	1.072	1.061	1.063	1.071	1.061	1.063	1.069
NSN	.9954	.9968	.9974	.9958	.9939	.9931	.9964	.9992	.9944
OD	1.053	1.066	1.064	1.068	1.064	1.066	1.059	1.066	1.066
TW	**.9534**	.9431	.9506	.9514	.9647	.9647	.9810	.9849	.9945
CGR	1.999	2.013	2.006	1.946	1.886	1.504	.9386	.5454	.3658
QN	.9954	**.9024**	**.8470**	.8052	**.7802**	.7079	.6644	.6515	.6125
TS	1.006	.9102	.8597	**.8032**	.7829	.6641	**.6010**	**.6050**	**.5796**
HOR	.5745	.3126	.2227	.1716	.1421	.0879	.0586	.0429	.0345

Average number of comparisons for a natural language problem

Prot	2	4	6	8	10	20	40	80	160
NAIVE	1.055	1.058	1.056	1.060	1.058	1.057	1.058	1.058	1.057
NSN	.9707	.9696	.9716	.9675	.9783	.9678	.9786	.9761	.9763
OD	1.056	1.059	1.059	1.060	1.060	1.062	1.060	1.059	1.065
TW	**.9562**	.9623	.9536	.9487	.9597	.9694	.9822	.9948	.9956
CGR	1.997	1.993	1.969	1.928	1.861	1.466	1.058	.8773	.6499
QN	.9707	.8929	.8460	.8114	.7817	.7145	.6418	.6343	.6496
TS	1.007	**.9210**	.8590	**.8000**	.7564	**.6049**	**.4610**	**.3541**	**.3629**
HOR	.5701	.3016	.2099	.1650	.1388	.0853	.0603	.0500	.0449

Average number of comparisons for a protein sequnce problem

5. *Conclusion*

After having surveyed the state-of-the-art of constant-space string-matching algorithms, we have presented two new string-matching algorithms with constant extra space complexity that, despite their quadratic worst-case time complexity, have very good performances in practice. In fact we have shown that in some cases one of our proposed algorithms has a better behavior than other string-matching algorithms which are allowed non-constant extra space, as is the case of the Horspool algorithm, one of the fastest variant of the Boyer-Moore algorithm: sometimes *economical* solutions are more efficient than *unrestricted* ones . . . “It’s economy, stupid!”

References

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[BR99] T. Berry and S. Ravindran. A fast string matching algorithm and experimental results. In J. Holub and M. Šimánek, editors, *Proceedings of the Prague Stringology Club Workshop ’99*, pages 16–28, Czech Technical University, Prague, Czech Republic, 1999. Collaborative Report DC–99–05.

[Bre93] Dany Breslauer. Saving comparisons in the Crochemore-Perrin string matching algorithm. In Thomas Lengauer, editor, *Algorithms—ESA ’93, First Annual European Symposium*, volume 726 of *Lecture Notes in Computer Science*, pages 61–72, Bad Honnef, Germany, 30 September–2 October 1993. Springer.

[CCG+94] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

[CF03] D. Cantone and S. Faro. Forward-Fast-Search: Another fast variant of the Boyer-Moore string matching algorithm. In M. Šimánek, editor, *Proc. of the Prague Stringology Conference (PSC 2003)*, pages 10–24, Czech Technical University, Prague, Czech Republic, 2003.

[CGR99] Maxime Crochemore, Leszek Gaşieniec, and Wojciech Rytter. Constant-space string-matching in sublinear average time. *Theoretical Computer Science*, 218(1):197–203, 1999.

[CP91] M. Crochemore and D. Perrin. Two-way string-matching. *Journal of the ACM*, 38(3):561–675, July 1991.

[Dog98] O. C. Dogaru. On the all accorrences of a word in a text. In *Proc. of The Prague Stringology Conference*, 1998.

[GPR95a] L. Gąsieniec, W. Plandowski, and W. Rytter. Constant-space string matching with smaller number of comparisons: sequential sampling. In Z. Galil and E. Ukkonen, editors, *Proc. of the 6th Annual Symposium on Combinatorial Pattern Matching*, number 937 in Lecture Notes in Computer Science, pages 78–89, Espoo, Finland, 1995. Springer-Verlag, Berlin.

[GPR95b] L. Gąsieniec, W. Plandowski, and W. Rytter. The zooming method: a recursive approach to time-space efficient string-matching. *Theor. Comput. Sci.*, 147(1–2):19–30, 1995.

[GS77] Z. Galil and J. Seiferas. Saving space in fast string-matching. In *Proc. of the 18th IEEE Annual Symposium on Foundations of Computer Science*, pages 179–188, Providence, Rhode Island, 1977. IEEE Computer Society Press.

[Han92] C. Hancart. Une analyse en moyenne de l'algorithme de Morris et Pratt et de ses raffinements. *des Automates et Applications, Actes des 2e Journes Franco-Belges*, 1992.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.

[KMP77] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.

[Sun90] D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.

[Vis91] U. Vishkin. Deterministic sampling - A new technique for fast pattern matching. *SIAM J. Comput.*, 20(1):22–40, 1991.