Journal of Automata, Languages and Combinatorics **u** (v) w, x–y

FAST-SEARCH ALGORITHMS: NEW EFFICIENT VARIANTS OF THE BOYER-MOORE PATTERN-MATCHING ALGORITHM [1]

Domenico Cantone and Simone Faro
Department of Mathematics and Computer Science, University of Catania
Viale A.Doria n.6 95125, Catania, Italy
e-mail: {cantone, faro}@dmi.unict.it

ABSTRACT

We present two variants of the Boyer-Moore string matching algorithm, named Fast-Search and Forward-Fast-Search, and compare them with some of the most effective string matching algorithms, such as Horspool, Quick Search, Tuned Boyer-Moore, Reverse Factor, and Berry-Ravindran. All algorithms are compared in terms of their run-time efficiency, number of text character inspections, and number of character comparisons.

It turns out that the new proposed variants, though not linear, achieve very good results especially in the case of very short patterns or small alphabets.

Keywords: string matching, experimental algorithms, text processing, comparisons, efficient algorithms, text searching.

1. Introduction

Given a text T and a pattern P over some 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 such diverse areas as text, image and signal processing, speech analysis and recognition, information retrieval, computational biology and chemistry, etc.

Several string matching algorithms have been proposed over the years. The Boyer-Moore algorithm [1] deserves a special mention, since it has been particularly successful and has inspired much work. It is based upon three simple ideas: right-to-left scanning, bad character and good suffix heuristics. We will review it at length in Section 2.1. Many subsequent algorithms have been based on variations on how to apply the two mentioned heuristics.

Before entering into details, we need a bit of notations and terminology. A string P is represented as a finite array $P[0\,..\,m-1]$, with $m \geq 0$. In such a case we say that P has length m and write $\text{length}(P) = m$. In particular, for $m = 0$ we obtain

[1] This paper is based on results preliminarily presented in [4, 5].

the empty string, also denoted by ε. By $P[i]$ we denote the $(i+1)$-st character of P, for $0 \leq i < \text{length}(P)$. Likewise, by $P[i\,..\,j]$ we denote the substring of P contained between the $(i+1)$-st and the $(j+1)$-st characters of P, for $0 \leq i \leq j < \text{length}(P)$. Moreover, for any $i, j \in \mathbb{Z}$, we put

$$P[i\,..\,j] = \begin{cases} \varepsilon & \text{if } i > j \\ P[\max(i,0), \min(j, \text{length}(P)-1)] & \text{otherwise.} \end{cases}$$

For any two strings P and P', we write $P' \sqsupset P$ to indicate that P' is a suffix of P, i.e., $P' = P[i\,..\,\text{length}(P)-1]$, for some $0 \leq i < \text{length}(P)$. Similarly, we write $P' \sqsubset P$ to indicate that P' is a prefix of P, i.e., $P' = P[0\,..\,i-1]$, for some $0 \leq i \leq \text{length}(P)$. In addition, we write $P.P'$ to denote the concatenation of P and P'.

Let T be a text of length n and let P be a pattern of length m. 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 have the following general structure:

```
Generic_String_Matcher(T, P)
    Precompute_Globals(P)
    n = length(T)
    m = length(P)
    s = 0
    while s ≤ n − m do
        j = Check_Shift(s, P, T)
        s = s + Shift_Increment(s, P, T, j)
```

where

- the procedure $\mathit{Precompute\_Globals}(P)$ computes useful mappings, in the form of tables, which later may be accessed by the function $\mathit{Shift\_Increment}(s, P, T)$;
- the function $\mathit{Check\_Shift}(s, P, T)$ checks whether s is a valid shift and returns the position j of the last matched character in the pattern;
- the function $\mathit{Shift\_Increment}(s, P, T, j)$ computes a *positive* shift increment according to the information obtained by procedures $\mathit{Precompute\_Globals}(P)$ and $\mathit{Check\_Shift}(s, P, T)$.

Observe that for the correctness of procedure *Generic_String_Matcher*, it is plainly necessary that the shift increment Δs computed by $\mathit{Shift\_Increment}(s, P, T)$ is *safe*, namely no valid shift can belong to the interval $\{s+1, \ldots, s+\Delta s-1\}$.

In the case of the naive string matching algorithm, for instance, the procedure *Precompute_Globals* is just dropped, procedure $\mathit{Check\_Shift}(s, P, T)$ checks whether the current shift is valid by scanning the pattern from left to right, and the function $\mathit{Shift\_Increment}(s, P, T, j)$ always returns a unitary shift increment. Then, such procedures can be instantiated as follows:

```
Naive_Check_Shift(s, P, T)
    for i = 0 to length(P) − 1 do
        if P[i] ≠ T[s + i] then
            return i − 1
    print(s)
    return m − 1

Naive_Shift_Increment(s, P, T, j)
    return 1
```

Therefore, in the worst case, the naive algorithm requires $\mathcal{O}(mn)$ character comparisons.

Information gathered during the execution of the *Shift_Increment*(s, P, T, j) function, in combination with the knowledge of P as suitably extracted by procedure *Precompute_Globals*(P), can yield shift increments larger than 1 and ultimately lead to more efficient algorithms. Consider for instance the case in which *Check_Shift*(s, P, T) processes P from right to left and finds immediately a mismatch between $P[m-1]$ and $T[s+m-1]$, where additionally it is known that the character $T[s+m-1]$ does not occur in P in any other position; then *Shift_Increment* can safely increment the shift by m. In the best case, when the above case occurs repeatedly, it can be verified that the text T does not contain any occurrence of P in sublinear time $\mathcal{O}(n/m)$.

The paper is organized as follows. In Section 2 we survey some of the most effective string matching algorithms. In Section 3 we describe a new variant of the Boyer-Moore algorithm, called Fast-Search, and prove some elementary technical properties. Next, in Section 4, we present a variation of the Fast-Search algorithm, called Forward-Fast-Search. Experimental data obtained by running all the reviewed algorithms under various conditions are presented and analyzed in Section 5. Finally, we draw our conclusions in Section 6.

2. Some Very Fast String Matching Algorithms

In this section we briefly review the Boyer-Moore algorithm and some of its most efficient variants that have been proposed over the years. In particular, we consider the Horspool [8], Tuned Boyer-Moore [9], Quick-Search [13], and Berry-Ravindran [2] algorithms. We also review the Reverse Factor algorithm [6], which is based on the smallest suffix automaton of the reverse pattern.

2.1. The Boyer-Moore Algorithm

The Boyer-Moore algorithm [1] is the progenitor of several algorithmic variants which aim at computing close to optimal shift increments very efficiently. Specifically, the Boyer-Moore algorithm checks whether s is a valid shift by scanning the pattern P from right to left and, at the end of the matching phase, computes the shift increment

as the maximum value suggested by the *good suffix rule* and the *bad character rule* below, using the functions gs_P and bc_P respectively, provided that both of them are applicable.

```
BM_Check_Shift(s, P, T)
    for i = length(P) − 1 downto 0 do
        if P[i] ≠ T[s + i] then
            return i + 1
    print(s)
    return 0

BM_Shift_Increment(s, P, T, j)
    if j > 0 then
        return max(gs_P(j), i − bc_P(T[s + j − 1]))
    return gs_P(0)
```

If the first mismatch occurs at position i of the pattern P, the good suffix rule suggests to align the substring $T[s+i+1\,..\,s+m-1] = P[i+1\,..\,m-1]$ with its rightmost occurrence in P preceded by a character different from $P[i]$. If such an occurrence does not exist, the good suffix rule suggests a shift increment which allows to match the longest suffix of $T[s+i+1\,..\,s+m-1]$ with a prefix of P.

More formally, if the first mismatch occurs at position i of the pattern P, the good suffix rule states that the shift can safely be incremented by $gs_P(i+1)$ positions, where

$$gs_P(j) =_{\text{Def}} \min\{0 < k \leq m \mid P[j-k\,..\,m-k-1] \sqsupset P \text{ and } (k \leq j-1 \rightarrow P[j-1] \neq P[j-1-k])\} \ ,$$

for $j = 0, 1, \ldots, m$. (The situation in which an occurrence of the pattern P is found can be regarded as a mismatch at position -1.)

The bad character rule states that if $c = T[s+i] \neq P[i]$ is the first mismatching character, while scanning P and T from right to left with shift s, then P can safely be shifted in such a way that its rightmost occurrence of c, if present, is aligned with position $(s+i)$ in T. In the case in which c does not occur in P, then P can safely be shifted just past position $(s+i)$ in T. More formally, the shift increment suggested by the bad character rule is given by the expression $(i - bc_P(T[s+i]))$, where

$$bc_P(c) =_{\text{Def}} \max(\{0 \leq k < m \mid P[k] = c\} \cup \{-1\}) \ ,$$

for $c \in \Sigma$, and where we recall that Σ is the alphabet of the pattern P and text T. Notice that there are situations in which the shift increment given by the bad character rule can be negative.

It turns out that the functions gs_P and bc_P can be computed during the preprocessing phase in time $\mathcal{O}(m)$ and $\mathcal{O}(m+|\Sigma|)$, respectively, and that the overall worst-case running time of the Boyer-Moore algorithm, as described above, is linear (cf. [7]).

For the sake of completeness, we notice that originally the Boyer-Moore algorithm made use of a good suffix rule based on the following simpler function

$$gs'_P(j) =_{\text{Def}} \min\{0 < k \leq m | P[j-k..m-k-1] \sqsupset P\} \ ,$$

for $j = 0, 1, \ldots, m$, which led to a non-linear worst-case running time.

2.2. The Horspool Algorithm

Horspool suggested a simplification of the original Boyer-Moore algorithm, defining a new variant which, though quadratic, performed better in practical cases (cf. [8]).

In the Horspool algorithm, the good suffix rule is just dropped and the shift advancement is computed in such a way that the rightmost character $T[s+m-1]$ is aligned with its rightmost occurrence on $P[0\,..\,m-2]$, if present; otherwise the pattern is advanced just past the window. This corresponds to advance the shift by $hbc_P(T[s+m-1])$ positions, where

$$hbc_P(c) =_{\text{Def}} \min(\{1 \leq k < m \mid P[m-1-k] = c\} \cup \{m\}) \ .$$

This translates into the following pseudo-code:

```
Horspool_Shift_Increment(s, P, T, j)
    return hbc_P(T[s + m − 1])
```

The resulting algorithm performs well in practice and can immediately be translated into programming code (see [3] for a simple implementation in the **C** programming language).

2.3. The Tuned Boyer-Moore Algorithm

The Tuned Boyer-Moore algorithm can be seen as an efficient implementation of the Horspool algorithm. Again, let P be a pattern of length m (cf. [9]). Each iteration of the Tuned Boyer-Moore algorithm can be divided into two phases: *last character localization* and *matching phase*. The first phase searches for a match of $P[m-1]$, by applying rounds of three blind shifts (based on the classical bad character rule) until needed. The matching phase then tries to match the rest of the pattern $P[0\,..\,m-2]$ with the corresponding characters of the text, proceeding from right to left. At the end of the matching phase, the shift advancement is computed according to the Horspool bad character rule. Moreover, to begin with, the algorithm adds m copies of $P[m-1]$ at the end of the text, as a sentinel, to compute the last shifts correctly.

The fact that the blind shifts require no comparison is at the heart of the very good practical behavior of the Tuned Boyer-Moore, despite its quadratic worst-case time complexity (cf. [11]).

2.4. The Quick-Search Algorithm

The Quick-Search algorithm, presented in [13], uses a modification of the original heuristics of the Boyer-Moore algorithm, much along the same lines of the Horspool algorithm. Specifically, it is based on the following observation: when a mismatch character is encountered, the pattern is always shifted to the right by at least one character, but never by more than m characters. Thus, the character $T[s+m]$ is always involved in testing for the next alignment. So, one can apply the bad character rule to $T[s+m]$, rather than to the mismatching character, obtaining larger shift advancements. This amounts to advance the shift by $qbc_P(T[s+m])$ positions, where

$$qbc_P(c) =_{\text{Def}} \min(\{0 < k \leq m \mid P[m-k] = c\} \cup \{m+1\}) \ .$$

The corresponding *Shift_Increment* function is very similar to the one of the Horspool algorithm, using the text character at position $s+m$ for shifting:

```
QS_Shift_Increment(s, P, T, j)
    return qbc_P(T[s + m])
```

Experimental tests have shown that the Quick-Search algorithm is very fast especially for short patterns (cf. [11]).

2.5. The Berry-Ravindran Algorithm

The Berry-Ravindran algorithm extends the Quick-Search algorithm by using in the bad character rule the two characters $T[s+m]$ and $T[s+m+1]$ rather than just the last character $T[s+m]$ of the window, where m is the size of the pattern P (cf. [2]). Thus, at the end of each matching phase with shift s, the Berry-Ravindran algorithm advances the pattern so that the substring of the text $T[s+m\,..\,s+m+1]$ is aligned with its rightmost occurrence in P. This amounts to advance the shift by $BRbc_P(T[s+m], T[s+m+1])$ positions, where

$$\begin{aligned} BRbc_P(c_1, c_2) =_{\text{Def}} \min \ & (\{0 < k < m \mid P[m-k-1] = c_1 \text{ and } P[m-k] = c_2\} \\ & \cup \ \{k \mid k = m \text{ and } P[0] = c_2\} \cup \{m+1\}) \ . \end{aligned}$$

Hence, we obtain the following pseudo-code of the *Shift_Increment* function for the Berry-Ravindran algorithm:

```
BR_Shift_Increment(s, P, T, j)
    return BRbc_P(T[s + m], T[s + m + 1])
```

The precomputation of the table used by the bad character rule requires $\mathcal{O}(|\Sigma|^2)$-space and $\mathcal{O}(m + |\Sigma|^2)$-time complexity, where Σ is the alphabet of the text and pattern. Experimental results show that the Berry-Ravindran algorithm is fast in practice and performs a low number of text/pattern character comparisons (cf. [2]).

2.6. The Reverse Factor Algorithm

Unlike the variants of the Boyer-Moore algorithm summarized above, the Reverse Factor algorithm computes shifts which match prefixes of the pattern, rather than suffixes. This is made possible by the smallest suffix automaton of the reverse of the pattern P, which is a deterministic finite automaton $S(P)$ whose accepted language is the set of suffixes of P (for a complete description see [6]).

The Reverse Factor algorithm has a quadratic worst-case time complexity, but it is very fast in practice (cf. [11]). Moreover, it has been shown that on the average it inspects $\mathcal{O}(n \log(m)/m)$ text characters, reaching the best bound shown in [14].

3. The Fast-Search Algorithm

In this section we present the first of our proposed variants of the Boyer-Moore algorithm, namely the Fast-Search algorithm.

As before, let P be a pattern of length m and let T be a text of length n over a finite alphabet Σ; also, let $0 \leq s \leq m - n$ be a shift. We make the following observation, which will be proved later in Sect. 3.1:

> the Horspool bad character rule leads to larger shift increments than the good suffix rule if and only if a mismatch occurs immediately, while comparing the pattern P with the window $T[s..s + m - 1]$, namely when $P[m - 1] \neq T[s + m - 1]$.

Such an observation suggests that the following shift increment rule may lead to a faster algorithm than the Horspool one:

> to compute the shift increment use the Horspool bad character rule, if a mismatch occurs during the first character comparison; otherwise use the good suffix rule.

This translates into the following pseudo-code of the *Shift_Increment* function, which is at the heart of the Fast-Search algorithm (cf. [4]):

```
Fast_Search_Shift_Increment(s, P, T, j)
    if j = m then
        return hbc_P(T[s + m − 1])
    else
        return gs_P(j)
```

Notice that $hbc_P(a) = bc_P(a)$, whenever $a \neq P[m - 1]$, so that the term $hbc_P(T[s + m-1])$ can be substituted by $bc_P(T[s+m-1])$ in the above procedure, as will be done in the efficient implementation of the Fast-Search algorithm to be given in Sect. 3.2.

Experimental data which will be presented in Sect. 5 confirm that the Fast-Search algorithm is faster than the Horspool algorithm. In fact, we will see that, though not linear, Fast-Search compares well with the fastest string matching algorithms, especially in the case of short patterns. We also notice that the functions hbc_P and

gs_P can be precomputed in time $\mathcal{O}(m)$ and $\mathcal{O}(m + |\Sigma|)$, respectively, by *Precompute_Globals*(P).

3.1. The Horspool Bad Character Rule versus the Good Suffix Rule

We will show in Theorem 2 that the Horspool bad character rule wins against the good suffix rule only when a mismatch is found during the first character comparison. To this purpose we first prove the following technical lemma.

Lemma 1 *Let P be a pattern of length $m \geq 1$ over an alphabet Σ. Then the following inequalities hold:*

(a) $gs_P(m) \leq hbc_P(c)$, *for* $c \in \Sigma \setminus \{P[m-1]\}$;

(b) $gs_P(j) \geq hbc_P(P[m-1])$, *for* $j = 0, 1, \ldots, m-1$.

Proof. Concerning (a), let $c \in \Sigma \setminus \{P[m-1]\}$ and let $\bar{k} = hbc_P(c)$. If $\bar{k} = m$, then $gs_P(m) \leq hbc_P(c)$ follows at once. On the other hand, if $\bar{k} < m$, then we have $P[m-1-\bar{k}] = c$, so that $P[m-1-\bar{k}] \neq P[m-1]$, since by assumption $P[m-1] \neq c$ holds. Therefore $gs_P(m) \leq \bar{k} = hbc_P(c)$, proving (a).

Next, let $0 \leq j < m$ and let $0 < k \leq m$ be such that

- $P[j-k..m-k-1] \sqsupset P$, and
- $P[j-1] \neq P[j-1-k]$, provided that $k \leq j-1$,

so that $gs_P(j) \leq k$. If $k < m$, then $P[m-k-1] = P[m-1]$, and therefore $hbc_P(P[m-1]) \leq k$. On the other hand, if $k = m$, then we plainly have $hbc_P(P[m-1]) \leq k$. Thus, in any case, $gs_P(j) \geq hbc_P(P[m-1])$, proving (b). □

Then we have:

Theorem 2 *Let P and T be two nonempty strings over an alphabet Σ and let $m = |P|$. Let us also assume that we are comparing P with the window $T[s..s+m-1]$ of T with shift s, scanning P from right to left. Then*

(a) if the first mismatch occurs at position $(m-1)$ of the pattern P, then

$$gs_P(m) \leq hbc_P(T[s+m-1]) \ ;$$

(b) if the first mismatch occurs at position $0 \leq i < m-1$ of the pattern P, then

$$gs_P(i+1) \geq hbc_P(T[s+m-1]) \ ;$$

(c) if no mismatch occurs, then

$$gs_P(0) \geq hbc_P(T[s+m-1]) \ .$$

Proof. Let us first assume that $P[m-1] \neq T[s+m-1]$, i.e., the first mismatch occurs at position $(m-1)$ of the pattern P, while comparing P with $T[s..s+m-1]$ from right to left. Then by Lemma 1(a) we have $gs_P(m) \leq hbc_P(T[s+m-1])$, yielding (a).

On the other hand, if $P[m-1] = T[s+m-1]$, i.e., the first mismatch occurs at position $0 \leq i < m-1$ or no mismatch occurs, then Lemma 1(b) implies immediately (b) and (c). □

3.2. An Efficient Implementation

A more effective implementation of the Fast-Search algorithm can be obtained much along the same lines of the Tuned Boyer-Moore algorithm. The main idea consists in iterating the bad character rule until the last character $P[m-1]$ of the pattern is matched correctly against the text, and then applying the good suffix rule, at the end of the matching phase. More precisely, starting from a shift position s, if we denote by j_i the total shift advancement after the i-th iteration of the bad character rule, then we have the following recurrence:

$$j_i = j_{i-1} + bc_P(T[s + j_{i-1} + m - 1]) \ .$$

Therefore, starting from a given shift s, the bad character rule is applied k times in row, where $k = \min\{i \mid T[s + j_i + m - 1] = P[m-1]\}$, with a resulting shift advancement of j_k. At this point it is known that $T[s + j_k + m - 1] = P[m-1]$, so that the subsequent matching phase can start with the $(m-1)$-st character of the pattern.

As in the case of the Tuned Boyer-Moore algorithm, the Fast-Search algorithm benefits from the introduction of an external sentinel, which allows to compute correctly the last shifts with no extra checks. For this purpose, we have chosen to add a copy of the pattern P at the end of the text T, obtaining a new text $T' = T.P$. Plainly, all the valid shifts of P in T are the valid shifts s of P in T' such that $s \leq n - m$, where, as usual, n and m denote respectively the lengths of T and P.

The code of the Fast-Search algorithm is presented in Fig. 1.

4. The Forward-Fast-Search Algorithm

In this section we present the second of our proposed variants of the Boyer-Moore algorithm, obtained by modifying the Fast-Search algorithm discussed in Section 3.

The new algorithmic variant, that we call Forward-Fast-Search, mantains the same structure of the Fast-Search algorithm, but is based upon a modified version of the good suffix rule, called *forward good suffix* rule, which uses a look-ahead character to determine larger shift advancements (cf. [5]).

The forward good suffix rule requires a precomputed table of size $m \cdot |\Sigma|$, where m is the length of the pattern and Σ is the alphabet of the text and pattern.

Concerning the running time, the forward good suffix rule could be precomputed by $|\Sigma|$ iterations of the standard linear precomputation of the Boyer-Moore good suffix rule, yielding a $\mathcal{O}(m \cdot |\Sigma|)$ time complexity. Nevertheless, we propose an alternative,

```
Fast-Search(P, T)
    n = length(T)
    m = length(P)
    T' = T.P
    bc_P = precompute-bad-character(P)
    gs_P = precompute-good-suffix(P)
    s = 0
    while bc_P(T'[s + m − 1]) > 0 do
        s = s + bc_P(T'[s + m − 1])
    while s ≤ n − m do
        j = m − 2
        while j ≥ 0 and P[j] = T'[s + j] do
            j = j − 1
        if j < 0 then
            print(s)
        s = s + gs_P(j + 1)
        while bc_P(T'[s + m − 1]) > 0 do
            s = s + bc_P(T'[s + m − 1])
```

Figure 1: The Fast-Search algorithm.

more direct approach which behaves very well in practice, though it requires $\mathcal{O}(m \cdot \max(m, |\Sigma|))$ time in the worst case.

4.1. Strengthening the Good Suffix Rule

4.1.1. The Backward Good Suffix Rule

A first natural way to strengthen the good suffix rule, which yields the *backward good suffix* rule, can be obtained by merging it with the bad character rule as follows. As usual, let us assume that we are comparing a pattern P of length m with the window $T[s\,..\,s+m-1]$ at shift s of a given text T, scanning it from right to left. If the first mismatch occurs at position i of the pattern P, i.e. $P[i+1\,..\,m-1] = T[s+i+1\,..\,s+m-1]$ and $P[i] \neq T[s+i]$, then the backward good suffix rule proposes to align the substring $T[s+i+1\,..\,s+m-1]$ with its rightmost occurrence in P preceded by the *backward* character $T[s+i]$. If such an occurrence does not exist, the backward good suffix rule proposes a shift increment which allows to match the longest suffix of $T[s+i+1\,..\,s+m-1]$ with a prefix of P. More formally, this

corresponds to increment the shift s by $\overleftarrow{gs}_P(i+1, T[s+i])$, where

$$\overleftarrow{gs}_P(j,c) =_{\text{Def}} \min\{0 < k \leq m \mid P[j-k \ldots m-k-1] \sqsupset P \\ \text{and } (k \leq j-1 \rightarrow P[j-1] = c)\} \ ,$$

for $j = 0, 1, \ldots, m$ and $c \in \Sigma$.

4.1.2. The Forward Good Suffix Rule

As observed by Sunday [13], after a matching phase with shift s, the *forward* character $T[s+m]$ is always involved in the subsequent matching phase. Thus, another possible variant of the good suffix rule, which we call *forward good suffix* rule, consists in matching the forward character $T[s+m]$, rather than the mismatched character $T[s+i]$. More precisely, if as above the first mismatch occurs at position i of the pattern P, the forward good suffix rule suggests to align the substring $T[s+i+1\,..\,s+m]$ with its rightmost occurrence in P preceded by a character different from $P[i]$. If such an occurrence does not exist, the forward good suffix rule proposes a shift increment which allows to match the longest suffix of $T[s+i+1\,..\,s+m]$ with a prefix of P. This corresponds to advance the shift s by $\overrightarrow{gs}_P(i+1, T[s+m])$ positions, where

$$\overrightarrow{gs}_P(j,c) =_{\text{Def}} \min(\{0 < k \leq m \ \mid \ P[j-k\,..\,m-k-1] \sqsupset P \\ \text{and } (k \leq j-1 \rightarrow P[j-1] \neq P[j-1-k]) \\ \text{and } P[m-k] = c\} \ \cup \ \{m+1\}) \ ,$$

for $j = 0, 1, \ldots, m$ and $c \in \Sigma$.

4.1.3. Comparing the Good Suffix Rule with its Variants

We computed the average shift advancement suggested by the good suffix rule and its backward and forward variants on four Randσ problems, for $\sigma = 2, 4, 8, 20$, with pattern lengths $2, 4, 6, 8, 10, 20, 40, 80$, and 160, where a Randσ problem consists in searching, for each assigned value of the pattern length, a set of 200 random patterns over an alphabet Σ of size σ in a 20Mb random text over the same alphabet Σ.

Experimental results, presented in the tables below, show that the forward and backward good suffix rules propose on the average much larger shift advancements than the standard good suffix rule (up to 400% better). In addition, the forward good suffix rule shows always a slightly better behavior than the backward one, which becomes more sensible in the case of very small alphabets. This is partly due to the fact that the forward character is always used by the forward good suffix rule to compute shift advancements, whereas there are cases in which the backward good suffix rule does not exploit the backward character.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
gs	1.540	2.762	3.869	4.765	5.468	8.464	12.254	16.137	21.807
$\overleftarrow{gs}$	1.540	2.762	3.869	4.765	5.468	8.464	12.254	16.137	21.807
$\overrightarrow{gs}$	**2.269**	**3.642**	**5.026**	**6.310**	**7.394**	**12.21**	**18.200**	**25.586**	**34.798**
$\sigma = 4$	2	4	6	8	10	20	40	80	160
gs	1.750	3.062	4.334	5.196	6.079	8.697	12.382	16.857	22.645
$\overleftarrow{gs}$	1.750	3.540	5.170	6.691	8.097	13.62	21.604	30.540	42.891
$\overrightarrow{gs}$	**2.687**	**4.407**	**6.114**	**7.696**	**9.245**	**15.55**	**25.149**	**36.584**	**51.398**
$\sigma = 8$	2	4	6	8	10	20	40	80	160
gs	1.880	3.453	4.833	5.399	6.656	10.05	13.613	19.510	25.807
$\overleftarrow{gs}$	1.880	3.857	5.692	7.441	9.294	17.63	31.570	51.010	75.734
$\overrightarrow{gs}$	**2.860**	**4.775**	**6.671**	**8.399**	**10.24**	**18.72**	**33.225**	**54.825**	**81.334**
$\sigma = 20$	2	4	6	8	10	20	40	80	160
gs	1.930	3.714	5.238	6.684	8.512	12.81	19.078	25.169	33.975
$\overleftarrow{gs}$	1.930	3.956	5.892	7.919	9.867	19.47	38.167	72.950	136.45
$\overrightarrow{gs}$	**2.946**	**4.929**	**6.896**	**8.868**	**10.85**	**20.44**	**39.206**	**74.084**	**138.22**

Average advancements for some Randσ problems

4.1.4. Implementing the Forward Good Suffix Rule

Given a pattern P of length m over an alphabet Σ, we have plainly

$$\overrightarrow{gs}_P(j, c) = gs_{P.c}(j) \ ,$$

for $j = 0, 1, \ldots, m$ and $c \in \Sigma$, where $P.c$ is the string obtained by concatenating the character c at the end of P. Thus, a natural way to compute the forward good suffix function $\overrightarrow{gs}_P$ consists in computing the standard good suffix functions $gs_{P.c}$, for all $c \in \Sigma$, by means of the $\mathcal{O}(m)$ tricky algorithm firstly given in [10] and then corrected in [12].

Such a procedure is asymptotically optimal, as it has $\mathcal{O}(m \cdot |\Sigma|)$ space and time complexity.

In Figure 2 we propose an alternative procedure to compute the forward good suffix function which, despite its $\mathcal{O}(m \cdot \max(m, |\Sigma|))$ worst-case time complexity, turns out to be very efficient in practice, even for large values of m.

After an initialization phase which takes $\mathcal{O}(m \cdot |\Sigma|)$ space and time complexity, the precompute-forward-good-suffix procedure carries out m iterations of its main **for**-loop, starting at line 7. During the k-th iteration, for $k = 1, 2, \ldots, m$, it computes the sequence $\mathcal{S}_k(P)$ of all occurrences in P of the suffix $P[m-k \,..\, m-1]$ of length k, implicitly represented by means of the array $next$:

$$\begin{aligned} \mathcal{S}_k(P) = \langle\ & P[next[m-1]-k+1 \,..\, next[m-1]] \ , \\ & P[next^{(2)}[m-1]-k+1 \,..\, next^{(2)}[m-1]], \\ & \ldots\ldots \\ & P[next^{(r_k)}[m-1]-k+1 \,..\, next^{(r_k)}[m-1]] \ \rangle \ , \end{aligned} \tag{1}$$

```
precompute-forward-good-suffix(P)
Initialization:
 1.    m = length(P)
 2.    for i = 0 to m do
 3.        for c ∈ Σ do
 4.            gs→[i, c] = m + 1
 5.    for i = 0 to m − 1 do
 6.        next[i] = i − 1
Computation:
 7.    for slen = 0 to m − 1 do
 8.        last = m − 1
 9.        i = next[last]
10.        while i ≥ 0 do
11.            if gs→[m − slen, P[i + 1]] > m − 1 − i then
12.                if  (i − slen < 0 or
13                        (i − slen ≥ 0 and P[i − slen] ≠ P[m − 1 − slen])) then
14.                    gs→[m − slen, P[i + 1]] = m − 1 − i
15.            if (i − slen ≥ 0 and P[i − slen] = P[last − slen]) or
16.               (i − slen < 0) then
17.                next[last] = i
18.                last = i
19.            i = next[i]
20.        if gs→[m − slen, P[0]] > m then
21.            gs→[m − slen, P[0]] = m
22.        next[last] = −1
23.    return gs→
```

Figure 2: The function for computing forward good suffixes

where r_k is such that $next^{(r_k+1)}[m-1] = -1$. For that purpose, lines 15-18 implement the recurrence

$$\mathcal{S}_k(P) = \langle P[j-k+1\,..\,j] \,|\, P[j-k+2\,..\,j] \in \mathcal{S}_{k-1}(P) \text{ and } P[j-k+1] = P[m-k] \rangle\,,$$

where $\mathcal{S}_0(P)$ is also formally given by (1), thanks to the way the array *next* is initialized in lines 5-6. Moreover, during the k-th iteration of the **for**-loop, for each $P[j-k+1\,..\,j] \in \mathcal{S}_k(P)$, the procedure updates, if necessary, the value $\overrightarrow{gs}(m-k-1, P[j+1])$ by setting it to $(m-1-j)$ (lines 11-14).

Plainly, the procedure in Figure 2 requires $\mathcal{O}(m \cdot |\Sigma|)$ space. To compute its

time complexity, it is enough to observe that the k-th execution of the **while**-loop in lines 10-19, for $k = 1, 2, \ldots, m$, takes $\mathcal{O}(|\mathcal{S}_{k-1}(P)|)$ time, giving a total of $\mathcal{O}(\sum_{j=0}^{m-1} |\mathcal{S}_j(P)|) = \mathcal{O}(m^2)$ time in the worst case. This leads to an overall $\mathcal{O}(m \cdot \max(m, |\Sigma|))$ worst-case time complexity, taking into account also the initizialization phase.

Experimental results show that the sum $\sum_{j=0}^{m-1} |\mathcal{S}_j(P)|$ has on the average an almost linear behavior. For instance, the following tables report the average of the sum $\sum_{j=0}^{m-1} |\mathcal{S}_j(P)|$ computed for $100,000$ random patterns of size m over an alphabet of size σ, for $\sigma = 2, 4, 8, 20$ and $m = 2, 4, 6, 8, 10, 20, 40, 80, 160$. The tests relative to a natural language buffer NL have been computed by randomly selecting $100,000$ substrings for each given pattern length over the 3.13Mb file obtained by discarding the nonalphabetic characters from the WinEdt spelling dictionary.

m	2	4	6	8	10	20	40	80	160
m^2 (worst case)	4	16	36	64	100	400	1600	6400	25600
Average for $\sigma = 2$	2.50	7.38	13.07	19.01	25.02	55.09	114.89	234.98	474.57
Average for $\sigma = 4$	2.24	5.46	8.76	12.10	15.45	32.09	65.34	132.06	264.98
Average for $\sigma = 8$	2.12	4.67	7.23	9.81	12.40	25.24	50.93	102.45	204.98
Average for $\sigma = 20$	2.04	4.25	6.46	8.68	10.89	21.96	44.00	88.21	176.63
Average on NL	2.04	4.23	6.47	8.84	11.99	28.57	57.97	111.61	208.00

For the same set of random tests, we also computed the *total* time taken to construct the forward good suffix function $\overrightarrow{gs}$, using the two implementations described earlier, namely the one which has a $\mathcal{O}(m \cdot |\Sigma|)$ worst-case time and space complexity and the procedure precompute-forward-good-suffix. Such implementations are denoted respectively "$\overrightarrow{gs}$ (I)" and "$\overrightarrow{gs}$ (II)" in the tables below, where experimental results are expressed in hundredths of seconds.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
$\overrightarrow{gs}$ (I)	58.1	60.1	63.1	66.1	68.1	81.1	103.2	149.2	**239.3**
$\overrightarrow{gs}$ (II)	**3.0**	**6.0**	**11.0**	**15.1**	**18.0**	**37.0**	**74.1**	**145.3**	288.4

$\sigma = 4$	2	4	6	8	10	20	40	80	160
$\overrightarrow{gs}$ (I)	113.2	117.1	121.2	124.2	128.2	142.2	174.2	235.4	357.5
$\overrightarrow{gs}$ (II)	**3.0**	**6.0**	**10.0**	**13.0**	**16.0**	**33.1**	**64.1**	**126.2**	**250.3**

$\sigma = 8$	2	4	6	8	10	20	40	80	160
$\overrightarrow{gs}$ (I)	225.3	230.4	237.3	240.4	243.3	268.4	313.4	401.6	577.9
$\overrightarrow{gs}$ (II)	**4.0**	**7.0**	**11.0**	**14.0**	**19.0**	**36.1**	**72.1**	**141.2**	**289.4**

$\sigma = 20$	2	4	6	8	10	20	40	80	160
$\overrightarrow{gs}$ (I)	558.8	573.9	580.8	589.8	598.9	642.9	733.1	905.3	1250.8
$\overrightarrow{gs}$ (II)	**5.0**	**11.0**	**16.0**	**20.1**	**26.0**	**50.1**	**98.1**	**195.3**	**394.6**

NL	2	4	6	8	10	20	40	80	160
$\overrightarrow{gs}$ (I)	553.8	565.8	573.8	583.8	592.8	636.9	725.0	895.3	1238.8
$\overrightarrow{gs}$ (II)	**5.0**	**10.0**	**16.0**	**19.0**	**23.1**	**48.1**	**95.1**	**189.3**	**379.5**

```
Forward-Fast-Search(P, T)
    n = length(T)
    m = length(P)
    T' = T.P[m − 1]^{m+1}
    bc = precompute-bad-character(P)
    →gs = precompute-forward-good-suffix(P)
    s = 0
    while bc[T'[s + m − 1]] > 0 do
        s = s + bc[T'[s + m − 1]]
    while s ≤ n − m do
        j = m − 2
        while j ≥ 0 and P[j] = T'[s + j] do
            j = j − 1
        if j < 0 then
            print(s)
        s = s + →gs[j + 1, T[s + m]]
        while bc[T'[s + m − 1]] > 0 do
            s = s + bc[T'[s + m − 1]]
```

Figure 3: The Forward-Fast-Search algorithm

The above experimental results show that for alphabets of size at least 4 the procedure precompute-forward-good-suffix is on the average always faster than the implementation of the forward good suffix function described at the beginning of the present section.

4.2. Building up the Forward-Fast-Search Algorithm

The implementation of the Forward-Fast-Search algorithm can be obtained along the same lines of the Fast-Search and the Tuned Boyer-Moore algorithms.

In the first phase, called *character localization* phase, the algorithm iterates the bad character rule until the last character $P[m-1]$ of the pattern is matched correctly against the text. More precisely, starting from a shift position s, if we denote by j_i the total shift advancement after the i-th iteration of the bad character rule, then we have the following recurrence:

$$j_i = j_{i-1} + bc_P(T[s + j_{i-1} + m - 1]) \ .$$

Therefore, the bad character rule is applied k times in a row, where $k = \min\{i \mid T[s + j_i + m - 1] = P[m-1]\}$, with an overall shift advancement of j_k.

At this point we have that $T[s + j_k + m - 1] = P[m-1]$, so that the subsequent *matching* phase can test for an occurrence of the pattern by comparing only the

remaining $(m-1)$ characters of the pattern. At the end of the *matching* phase the algorithm applies the forward good suffix rule instead of the traditional good suffix rule.

As in the case of the Fast-Search and Tuned Boyer-Moore algorithms, the Forward-Fast-Search algorithm benefits from the introduction of an external sentinel: since the forward good suffix rule looks at the character $T[s+m]$ just after the current window, $m+1$ copies of the character $P[m-1]$ are added at the end of the text T, obtaining a new text $T' = T.P[m-1]^{m+1}$. This allows to compute correctly the last shifts with no extra checks. Plainly, all the valid shifts of P in T are the valid shifts s of P in T' such that $s \leq n-m$, where, as usual, n and m denote respectively the lengths of T and P. The code of the Forward-Fast-Search algorithm is presented in Figure 3.

5. Experimental Results

In this section we present and comment experimental data relative to the string matching algorithms which have been discussed in the preceding sections, namely: Horspool (HOR), Quick-Search (QS), Barry-Ravidran (BR), Tuned Boyer-Moore (TBM), Reverse Factor (RF), Fast-Search (FS), and Forward-Fast-Search (FFS). We have chosen to compare the algorithms in terms of their running time, number of text character inspections, and number of character comparisons.

All algorithms have been implemented in the **C** programming language and were used to search for the same strings in large fixed text buffers on a PC with AMD Athlon processor of 1.19GHz. In particular, the algorithms have been tested on four Randσ problems, for $\sigma = 2, 4, 8, 20$, and on a natural language text buffer NL 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 3.13Mb file obtained by discarding the nonalphabetic characters from the WinEdt spelling dictionary. Patterns to be searched for have been constructed by selecting 200 random substrings of length m, for each $m = 2, 4, 6, 8, 10, 20, 40, 80, 160$.

5.1. Running Times

Experimental results show that the Fast-Search and the Forward-Fast-Search algorithms attain the best run-time performances in most cases, expecially in the case of small alphabets. Additionally it turns out that the Forward-Fast-Search algorithm performs better in most cases and, sporadically, it is second only to the Fast-Search algorithm, in the case of natural language texts and long patterns, and to the Berry-Ravindran algorithm, in the case of large alphabets and patterns.

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

$\sigma = 2$	2	4	6	8	10	20	40	80	160
HOR	42.01	44.18	42.86	42.02	46.57	40.24	39.51	38.83	39.95
QS	34.33	41.12	38.35	39.30	42.80	37.42	36.77	36.42	36.54
BR	44.84	49.36	44.42	43.48	47.69	40.66	40.70	40.74	40.54
TBM	33.96	36.54	36.88	36.65	40.53	35.98	36.05	35.54	36.30
RF	249.2	200.0	145.9	114.2	107.3	57.95	36.84	27.95	22.36
FS	41.79	35.36	28.72	25.32	26.15	20.40	18.40	17.99	17.31
FFS	**31.08**	**28.87**	**25.28**	**22.37**	**23.15**	**18.05**	**16.78**	**16.62**	**15.82**

Running times for a Rand2 problem

$\sigma = 4$	2	4	6	8	10	20	40	80	160
HOR	34.66	25.57	22.05	20.76	20.27	19.68	20.05	19.54	20.20
QS	26.49	22.10	19.87	19.35	18.98	18.58	19.05	18.73	19.04
BR	32.20	25.68	22.08	20.31	19.24	17.29	16.66	16.36	16.51
TBM	25.53	20.68	19.15	18.85	18.76	18.50	18.81	18.38	18.78
RF	156.1	98.60	74.84	62.28	53.79	34.73	24.26	20.34	16.67
FS	28.60	20.58	18.91	18.26	17.86	17.22	16.53	16.18	15.82
FFS	**24.87**	**20.06**	**18.35**	**17.65**	**17.22**	**16.23**	**15.61**	**15.33**	**14.40**

Running times for a Rand4 problem

$\sigma = 8$	2	4	6	8	10	20	40	80	160
HOR	27.71	20.19	18.40	17.43	16.84	15.70	15.56	15.62	15.71
QS	20.91	18.27	17.17	16.59	16.25	15.36	15.22	15.23	15.35
BR	25.19	20.55	18.77	17.74	17.02	15.33	14.55	14.55	13.96
TBM	21.09	17.78	16.78	16.77	16.22	15.14	15.11	15.05	15.18
RF	114.8	70.75	54.97	46.27	40.62	27.26	20.58	18.17	15.01
FS	20.66	17.75	16.75	16.41	16.01	15.02	14.89	14.80	14.81
FFS	**20.20**	**17.58**	**16.60**	**16.17**	**15.82**	**14.87**	**14.54**	**14.52**	**13.92**

Running times for a Rand8 problem

$\sigma = 20$	2	4	6	8	10	20	40	80	160
HOR	23.45	18.17	16.58	16.21	15.89	15.21	14.90	14.84	14.98
QS	18.67	16.84	15.78	15.69	15.49	14.98	14.74	14.73	14.79
BR	21.83	18.88	17.32	16.89	16.47	15.47	14.90	**14.42**	**12.60**
TBM	**18.76**	16.78	15.64	**15.44**	15.39	14.85	14.82	14.65	14.65
RF	92.44	54.83	41.67	35.57	31.61	23.12	19.25	17.69	14.72
FS	19.11	16.59	15.57	15.49	**15.24**	**14.81**	14.66	14.65	14.58
FFS	**18.76**	**16.51**	**15.51**	**15.44**	**15.24**	14.83	**14.64**	14.65	14.35

Running times for a Rand20 problem

NL	2	4	6	8	10	20	40	80	160
HOR	3.40	2.65	2.45	2.36	2.36	2.22	2.15	2.11	1.98
QS	2.73	2.42	2.35	**2.24**	2.20	**2.14**	2.09	**2.09**	2.01
BR	3.28	2.87	2.66	2.59	2.47	2.33	2.25	2.21	1.95
TBM	2.77	2.39	2.27	2.25	**2.18**	2.19	2.09	2.12	1.93
RF	13.94	8.33	6.48	5.46	4.87	3.35	2.79	2.68	4.67
FS	2.79	2.45	**2.22**	**2.24**	2.19	**2.14**	**2.06**	**2.09**	**1.91**
FFS	**2.70**	**2.35**	2.26	2.26	**2.18**	2.15	2.13	2.11	2.24

5.2. Average Number of Text Character Inspections

For each test, the average number of character inspections has been obtained by taking the total number of times a text character is accessed (either to perform a comparison with a pattern character, or to perform a shift, or to compute a transition in an automaton) and dividing it by the length of the text buffer.

It turns out that the Forward-Fast-Search algorithm is always very close to the best results which are generally obtained by the Fast-Search algorithm, for short patterns, and by Reverse-Factor algorithm, for long patterns. We notice, however, that the Forward-Fast-Search algorithm attains in most cases the second best result and it is better than Reverse-Factor, for short patterns, and Fast-Search, for long patterns.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
HOR	**1.00**	1.15	1.26	1.26	1.28	1.24	1.27	1.23	1.27
QS	1.54	1.67	1.63	1.67	1.64	1.61	1.65	1.61	1.60
BR	1.28	1.25	1.20	1.20	1.19	1.19	1.19	1.18	1.16
TBM	1.23	1.35	1.46	1.46	1.47	1.43	1.46	1.42	1.46
RF	1.43	1.06	**.799**	**.615**	**.519**	**.294**	**.169**	**.096**	**.054**
FS	**1.00**	**.929**	.806	.698	.632	.460	.348	.270	.213
FFS	1.15	.993	.833	.703	.621	.410	.289	.210	.161

Average number of text character inspections for a Rand2 problem.

$\sigma = 4$	2	4	6	8	10	20	40	80	160
HOR	**.714**	.510	.435	.404	.392	.373	.389	.365	.392
QS	1.03	.817	.700	.675	.645	.610	.650	.622	.633
BR	.949	.713	.569	.488	.429	.307	.264	.244	.251
TBM	.841	.591	.504	.468	.454	.432	.450	.422	.446
RF	.886	.528	**.387**	**.316**	**.264**	**.154**	**.089**	**.051**	**.028**
FS	**.714**	**.489**	.398	.356	.330	.273	.239	.200	.177
FFS	.768	.526	.418	.367	.330	.241	.182	.136	.105

Average number of text character inspections for a Rand4 problem.

$\sigma = 8$	2	4	6	8	10	20	40	80	160
HOR	**.600**	.350	.263	.222	.198	.158	.153	.149	.152
QS	.842	.575	.456	.393	.358	.291	.282	.278	.277
BR	.844	.582	.443	.360	.305	.179	.109	.072	.057
TBM	.663	.386	.291	.245	.218	.174	.168	.164	.167
RF	.674	.381	.278	.225	**.191**	**.112**	**.063**	**.036**	**.020**
FS	**.600**	**.348**	**.260**	**.217**	.193	.150	.137	.126	.117
FFS	.627	.368	.274	.227	.201	.146	.117	.093	.075

Average number of text character inspections for a Rand8 problem.

$\sigma = 20$	2	4	6	8	10	20	40	80	160
HOR	**.538**	.285	.199	.157	.132	.083	.061	.054	.053
QS	.734	.463	.346	.282	.242	.157	.118	.104	.104
BR	.787	.528	.397	.318	.266	.146	.078	.042	.023
TBM	.563	.297	.208	.164	.137	.086	.063	.056	.056
RF	.565	.302	.214	.170	.143	.084	**.049**	**.027**	**.014**
FS	**.538**	**.284**	**.198**	**.156**	**.131**	**.082**	.060	.053	.052
FFS	.550	.293	.205	.161	.135	**.082**	.060	.049	.043

Average number of text character inspections for a Rand20 problem.

NL	2	4	6	8	10	20	40	80	160
HOR	**.550**	.300	**.211**	**.171**	.144	.091	.059	.042	.032
QS	.759	.489	.375	.309	.261	.175	.125	.086	.066
BR	.795	.538	.411	.335	.278	.155	.085	.050	.028
TBM	.584	.318	.226	.182	.153	.096	.062	.044	.034
RF	.588	.321	.231	.185	.153	**.084**	**.045**	**.024**	**.013**
FS	**.550**	**.299**	**.211**	**.171**	**.143**	.087	.055	.038	.027
FFS	.565	.312	.220	.180	.152	.088	.054	.036	.026

Average number of text character inspections for a natural language problem

5.3. 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 Forward-Fast-Search algorithm achieves the best results in most cases and, sporadically, it is second only to the Berry-Ravindran algorithm, which obtains very good results for short patterns and small alphabets. Moreover, we observe that Tuned Boyer-Moore, Fast-Search and Forward-Fast-Search algorithms perform a very low number of characters comparisons in the case of large alphabets.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
HOR	1.000	1.159	1.260	1.269	1.281	1.244	1.272	1.235	1.270
QS	.9588	1.109	1.088	1.119	1.095	1.073	1.104	1.079	1.080
BR	**.2631**	**.3766**	.3916	.3989	.3962	.3973	.3969	.3940	.3893
TBM	.3333	.6044	.6995	.7154	.7249	.7082	.7215	.7024	.7205
FS	.3333	.4767	.4466	.3925	.3573	.2609	.1967	.1530	.1248
FFS	.3076	.4224	**.3875**	**.3324**	**.2962**	**.1964**	**.1377**	**.1003**	**.0766**

Average number of comparisons for a Rand2 problem.

$\sigma = 4$	2	4	6	8	10	20	40	80	160
HOR	.7143	.5100	.4356	.4041	.3922	.3732	.3890	.3652	.3928
QS	.6053	.4864	.4109	.3908	.3716	.3491	.3719	.3556	.3742
BR	.2747	.2353	.1898	.1628	.1432	.1025	.0883	.0813	.0837
TBM	.1429	.1445	.1264	.1175	.1140	.1085	.1131	.1062	.1141
FS	.1429	.1373	.1141	.1024	.0949	.0784	.0690	.0577	.0526
FFS	**.1323**	**.1272**	**.1041**	**.0913**	**.0822**	**.0601**	**.0454**	**.0341**	**.0263**

Average number of comparisons for a Rand4 problem.

$\sigma = 8$	2	4	6	8	10	20	40	80	160
HOR	.6000	.3501	.2639	.2222	.1985	.1586	.1531	.1490	.1522
QS	.4631	.3189	.2505	.2139	.1943	.1559	.1504	.1487	.1524
BR	.2711	.1940	.1479	.1202	.1018	.0598	.0364	.0243	.0190
TBM	.0667	.0482	.0365	.0307	.0274	.0219	.0212	.0206	.0210
FS	.0667	.0477	.0359	.0300	.0267	.0207	.0190	.0175	.0167
FFS	**.0634**	**.0459**	**.0345**	**.0287**	**.0252**	**.0184**	**.0148**	**.0117**	**.0095**

Average number of comparisons for a Rand8 problem.

$\sigma = 20$	2	4	6	8	10	20	40	80	160
HOR	.5385	.2844	.1991	.1569	.1316	.0828	.0608	.0541	.0537
QS	.3837	.2427	.1805	.1476	.1263	.0817	.0607	.0538	.0534
BR	.2608	.1760	.1323	.1061	.0887	.0490	.0263	.0141	.0079
TBM	.0256	.0149	.0104	.0082	.0069	.0043	.0032	.0028	.0028
FS	.0256	.0149	.0104	.0082	.0069	.0043	.0032	.0028	.0027
FFS	**.0251**	**.0147**	**.0103**	**.0081**	**.0068**	**.0042**	**.0030**	**.0025**	**.0022**

Average number of comparisons for a Rand20 problem.

NL	2	4	6	8	10	20	40	80	160
HOR	.5501	.3000	.2117	.1716	.1445	.0913	.0595	.0420	.0329
QS	.4031	.2605	.2002	.1646	.1393	.0914	.0654	.0455	.0364
BR	.2599	.1794	.1371	.1118	.0927	.0519	.0286	.0168	.0094
TBM	.0345	.0245	.0171	.0142	.0123	.0089	.0061	.0046	.0042
FS	.0345	.0245	.0171	**.0141**	**.0121**	.0066	.0043	.0030	.0025
FFS	**.0333**	**.0244**	**.0168**	.0153	.0140	**.0058**	**.0032**	**.0020**	**.0014**

Average number of comparisons for a natural language problem

6. Conclusion

We presented two new efficient variants of the Boyer-Moore string matching algorithm, named Fast-Search and Forward-Fast-Search. Both algorithms scan the pattern from right to left and repeatedly apply the bad character rule until the last character of the pattern is matched correctly. At the end of the matching phase, shift advancements are computed as a function of the matched suffix of the pattern (in the case of the Fast-Search algorithm) or as a function of the matched suffix of the pattern plus the first character of the text past the current window (in the case of the Forward-Fast-Search algorithm).

It turns out that, in spite of their quadratic worst-case time complexity and, in the case of the Forward-Fast-Search algorithm, of the $\mathcal{O}(m \cdot |\Sigma|)$-space and $\mathcal{O}(m \cdot \max(m, |\Sigma|))$-time worst-case complexity to precompute the forward good suffix function, both algorithms are very fast in practice and compare well with other fast variants of the Boyer-Moore algorithm.

We plan to evaluate theoretically the average time complexity of both Fast-Search and Forward-Fast-Search algorithms and to adapt them to scanning strategies which depend on character frequencies.

References

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

[2] T. Berry and S. Ravindran. A fast string matching algorithm and experimental results. *Proc. of the Prague Stringology Club Workshop '99* Czech Technical University, Prague, Czech Republic, Collaborative Report DC–99–05, pp. 16–28, 1999.

[3] R. A. Baeza-Yates and M. Régnier. Average running time of the Boyer-Moore-Horspool algorithm. *Theor. Comput. Sci.*, 92(1):19–31, 1992.

[4] D. Cantone and S. Faro. Fast-Search: a new variant of the Boyer-Moore string matching algorithm. In K. Jansen et al. (Eds.), *Proc. of WEA 2003*, LNCS 2647, pp. 47–58, 2003.

[5] D. Cantone and S. Faro. Forward-Fast-Search: another fast variant of the Boyer-Moore string matching algorithm. In M. Šimánek (Ed.), *Proc. of the Prague Stringology Conference '03*, Czech Technical University, Prague, Czech Republic, pp. 10–24, 2003.

[6] 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.

[7] L. J. Guibas and A. M. Odiyzko. A new proof of the linearity of the Boyer-Moore string searching algorithm. *SIAM J. Comput.*, 9(4):672–682, 1980.

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

[9] A. Hume and D. M. Sunday. Fast string searching. *Softw. Pract. Exp.*, 21(11):1221–1248, 1991.

[10] D. E. Knuth, J. H. Morris, and V. B. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:323-350, 1977.

[11] T. Lecroq. New experimental results on exact string-matching. Rapport LIFAR 2000.03, Université de Rouen, France, 2000.

[12] W. Rytter. A correct preprocessing algorithm for Boyer-Moore string searching. *SIAM J. Comput.*, 9:509-512, 1980.

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

[14] A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.