Fast-Search: a New Efficient Variant of the Boyer-Moore String Matching Algorithm

Domenico Cantone and Simone Faro

Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
{cantone | faro}@dmi.unict.it

Abstract. We present a new variant of the Boyer-Moore string matching algorithm which, though not linear, is very fast in practice.
We compare our algorithm with the Horspool, Quick Search, Tuned Boyer-Moore, and Reverse Factor algorithms, which are among the fastest string matching algorithms for practical uses. It turns out that our algorithm achieve very good results in terms of both time efficiency and number of character inspections, especially in the cases in which the patterns are very short.

Key words: string matching, experimental algorithms, text processing.

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 several areas such as text processing, information retrieval, and computational biology.

A very comprehensive description of the existing string matching algorithms and a fairly complete bibliography can be found respectively at the following URLs

- `http://www-igm.univ-mlv.fr/~lecroq/string/`
- `http:// liinwww.ira.uka.de/bibliography/Theory/tq.html`.

We first introduce the notation and terminology used in the paper. We denote the empty string by ε. 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 $0 \leq i \leq j < \text{length}(P)$, we denote by $P[i..j]$ the substring of P contained between the $(i+1)$-st and the $(j+1)$-st characters of P. Moreover, for any i and j, we put

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

If P and P' are two strings, 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 $0 \leq i \leq \text{length}(P)$. Similarly we

write $P' \sqsubset P$ to indicate that P' is a prefix of P, i.e., $P' = P[0..i-1]$, for some $0 \leq i \leq \text{length}(P)$.

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
        s = s + Shift_Increment(s, P, T)
```

where

- the procedure $Precompute\_Globals(P)$ computes useful mappings, in the form of tables, which may be later accessed by the function $Shift\_Increment(s, P, T)$;
- the function $Shift\_Increment(s, P, T)$ checks whether s is a valid shift and computes a *positive* shift increment.

Observe that for the correctness of procedure $Generic\_String\_Matcher$, it is plainly necessary that the shift increment Δs computed by $Shift\_Increment(s, P, T)$ 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 and the function $Shift\_Increment(s, P, T)$ always returns a unitary shift increment, after checking whether the current shift is valid. The latter can be instantiated as follows:

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

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

Information gathered during the execution of the $Shift\_Increment(s, P, T)$ 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

Shift_Increment(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 the character $T[s+m-1]$ does not occur in P in any other position; then the shift can safely be incremented 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 $\mathcal{O}(n/m)$.

1.1 The Boyer-Moore Algorithm

The Boyer-Moore algorithm (cf. [BM77]) is a progenitor of several algorithmic variants which aim at efficiently computing shift increments close to optimal. Specifically, the Boyer-Moore algorithm can be characterized by the following function *BM_Shift_Increment*(s, P, T) which, as in the previous example, scans the pattern P from right to left. *BM_Shift_Increment*(s, P, T) computes the shift increment as the maximum value suggested by the *good suffix rule* and the *bad character rule* below, via the functions gs_P and bc_P respectively, provided that both of them are applicable.

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

If a mismatch occurs at position i of the pattern P, while it is scanned from right to left, the good suffix rule suggests to align the substring $T[s+i+1 \ldots s+m-1] = P[i+1 \ldots m-1]$ with its rightmost occurrence in P preceded by a character different by $P[i]$. If such an occurrence does not exist, the good suffix rule suggests a shift increment which allows to match the longest suffix of $T[s+i+1 \ldots s+m-1]$ 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 be safely 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 be safely shifted in such a way that its rightmost occurrence of c, if present, is aligned with position $(s+i)$ of T. In the case in which c does not occur in P, then P can be safely shifted just past position $(s+i)$ of T. More formally, the

shift increment suggested by the bad character rule is given by the expression $(i - bc_P(T[s+i]))$, where

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

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. [GO80]).

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.

Several variants of the Boyer-Moore algorithm have been proposed over the years. In particular, we mention Horspool, Quick Search, Tuned Boyer-Moore, and the Reverse Factor algorithms, which are among the fastest variants in practice (cf. [Hor80], [Sun90], [HS91], and [CCG+94], respectively). In Sect. 3, we will compare them with our proposed variant of the Boyer-Moore algorithm.

1.2 The Horspool Algorithm

Horspool suggested a simplification to the original Boyer-Moore algorithm, defining a new variant which, though quadratic, performed better in practical cases (cf. [Hor80]). He just dropped the good suffix rule and based the calculation of the shift increments only on the following variation of the bad character rule. Specifically, he observed that when the first mismatch between the window $T[s..s+m-1]$ and the pattern P occurs at position $0 \leq i < m$ and the rightmost occurrence of the character $T[s+i]$ in P is at position $j > i$, then the bad character rule would shift the pattern backwards. Thus, he proposed to compute the shift advancement 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 (notice that the character $P[m-1]$ has been left out); otherwise the pattern is advanced just past the window. This corresponds to advance the shift by $hbc_P(T[s+m-1])$ positions, where

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

It turns out that the resulting algorithm performs well in practice and can be immediately translated into programming code (see Baeza-Yates and Régnier [BYR92] for a simple implementation in the **C** programming language).

1.3 The Quick-Search Algorithm

The Quick-Search algorithm, presented in [Sun90], 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 corresponds to advance the shift by $qbc_P(T[s+m])$ positions, where

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

Experimental tests have shown that that the Quick-Search algorithm is very fast especially for short patterns (cf. [Lec00]).

1.4 The Tuned Boyer-Moore Algorithm

The Tuned Boyer-Moore algorithm (cf. [HS91]) can be seen as an efficient implementation of the Horspool algorithm. Again, let P be a pattern of length m. 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 tries then 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, in order to compute the last shifts correctly, the algorithm in the first place adds m copies of $P[m-1]$ at the end of the text, as a sentinel.

The fact that the blind shifts require no checks is at the heart of the very good practical behavior of the Tuned Boyer-Moore, despite its quadratic worst-case time complexity (cf. [Lec00]).

1.5 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 accomplished by means of the smallest suffix automaton of the reverse of the pattern, while scanning the text and pattern from right to left (for a complete description see [CCG+94]).

The Reverse Factor algorithm has a quadratic worst-case time complexity, but it is very fast in practice (cf. [Lec00]). Moreover, it has been shown that on the average it inspects $\mathcal{O}(n \log(m)/m)$ text characters, reaching the best bound shown by Yao in 1979 (cf. [Yao79]).

2 Fast-Search: a New Efficient Variant of the Boyer-Moore Algorithm

We present now a new efficient variant of the Boyer-Moore algorithm, called Fast-Search, which will use the *Fast_Search_Shift_Increment* procedure to be given below as shift increment function. 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. The main observation upon which our Fast-Search algorithm is based is the following:

> 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 $P[m-1] \neq T[s+m-1]$.

The above observation, which will be proved later in Sect. 2.1, suggests at once that the following shift increment rule should 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:

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

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. 2.2.

Experimental data which will be presented in Sect. 3 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).

2.1 The Horspool Bad Character Rule versus the Good Suffix Rule

We will show in Proposition 1 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:

Proposition 1. *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). □

```
Fast-Search(P, T)
1.    n = length(T)
2.    m = length(P)
3.    T' = T.P
4.    bc_P = precompute-bad-character(P)
5.    gs_P = precompute-good-suffix(P)
7.    s = 0
8.    while bc_P(T'[s + m − 1]) > 0 do s = s + bc_P(T'[s + m − 1])
9.    while s ≤ n − m do
10.       j = m − 2
11.       while j ≥ 0 and P[j] = T'[s + j] do j = j − 1
12.       if j < 0 then print(s)
13.       s = s + gs_P(j + 1)
14.       while bc_P(T'[s + m − 1]) > 0 do s = s + bc_P(T'[s + m − 1])
```

Fig. 1. The Fast-Search algorithm.

2.2 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 $(m-2)$-nd 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.

3 Experimental Results

In this section we present experimental data which allow to compare the running times and number of character inspections of the following string matching algo-

rithms in various conditions: Fast-Search (FS), Horspool (HOR), Quick-Search (QS), Tuned Boyer-Moore (TBM), and Reverse Factor(RF).

All five 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 three Randσ problems, for $\sigma = 2, 8, 20$, and on a natural language text buffer.

A Randσ problem consisted in searching a set of 200 random patterns over an alphabet Σ of size σ, for each assigned value of the pattern length, in a 20Mb random text over the same alphabet Σ. We have performed our tests with patterns of length $2, 4, 6, 8, 10, 20, 40, 80$, and 160.

The tests on a natural language text buffer have been performed on a 3.13Mb file obtained from the WinEdt spelling dictionary by discarding non-alphabetic characters. All words in the text buffer have been searched for.

In the following tables, running times are expressed in hundredths of seconds. Concerning the number of character inspections, these have been obtained by taking the average of 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 total number of characters in the text buffer.

Experimental results show that the Fast-Search algorithm obtains the best runtime performances in most cases and, sporadically, it is second only to the Tuned Boyer-Moore algorithm.

Concerning the number of text character inspections, it turns out that the Fast-Search algorithm is quite close to the Reverse Factor algorithm, which generally shows the best behaviour. We notice, though, that in the case of very short patterns the Fast-Search algorithm reaches the lowest number of character accesses.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
HOR	46.05	44.75	44.77	45.12	44.83	42.10	41.23	40.83	42.13
QS	38.13	40.59	42.11	41.27	41.13	38.97	38.09	37.04	37.54
TBM	**36.27**	36.26	38.42	38.87	38.69	37.75	37.81	37.36	38.44
RF	268.38	197.88	149.83	120.14	100.02	60.37	37.91	28.40	22.63
FS	38.38	**32.96**	**30.19**	**27.35**	**25.40**	**21.04**	**18.90**	**18.16**	**17.39**

Running times for a Rand2 problem.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
HOR	1.83	1.72	1.66	1.66	1.64	1.59	1.64	1.61	1.68
QS	1.54	1.65	1.69	1.64	1.63	1.64	1.67	1.60	1.63
TBM	1.23	1.35	1.42	1.45	1.45	1.42	1.46	2.43	2.49
RF	1.43	1.06	**.78**	**.62**	**.51**	**.29**	**.16**	**.09**	**.05**
FS	**1.00**	**.92**	.80	.70	.63	.45	.34	.26	.22

Number of text character inspections for a Rand2 problem.

$\sigma = 8$	2	4	6	8	10	20	40	80	160
HOR	30.22	21.99	21.85	18.62	18.04	17.27	17.24	17.11	17.38
QS	22.41	20.43	19.48	17.63	17.41	16.93	16.86	16.82	16.94
TBM	23.14	**19.51**	18.95	17.34	17.07	16.79	16.78	16.73	16.97
RF	120.5	74.29	63.99	48.61	42.84	29.16	22.23	19.71	16.48
FS	**22.06**	**19.51**	**18.77**	**17.11**	**16.96**	**16.65**	**16.64**	**16.54**	**16.47**

Running times for a Rand8 problem.

$\sigma = 8$	2	4	6	8	10	20	40	80	160
HOR	1.191	.680	.507	.422	.374	.294	.282	.275	.281
QS	.842	.575	.456	.393	.358	.291	.282	.278	.285
TBM	.663	.386	.291	.245	.218	.174	.168	.164	.167
RF	.674	.381	.278	.225	**.191**	**.112**	**.063**	**.360**	**.020**
FS	**.600**	**.348**	**.260**	**.217**	.193	.150	.137	.126	.120

Number of text character inspections for a Rand8 problem.

$\sigma = 20$	2	4	6	8	10	20	40	80	160
HOR	24.51	18.56	17.03	16.39	16.01	15.19	14.78	14.84	14.98
QS	19.16	17.16	16.19	15.77	15.51	14.93	14.70	14.67	14.69
TBM	19.12	16.68	15.80	15.48	**15.25**	14.79	14.64	14.57	14.79
RF	96.16	56.63	43.32	36.69	32.29	23.43	19.46	17.83	14.62
FS	**19.11**	**16.67**	**15.78**	**15.43**	15.26	**14.74**	**14.58**	**14.55**	**14.51**

Running times for a Rand20 problem.

$\sigma = 20$	2	4	6	8	10	20	40	80	160
HOR	1.075	.566	.395	.311	.259	.161	.119	.106	.103
QS	.735	.463	.346	.282	.241	.156	.118	.107	.103
TBM	.563	.297	.208	.164	.137	.086	.064	.057	.055
RF	.565	.302	.214	.171	.143	.084	**.049**	**.027**	**.014**
FS	**.538**	**.284**	**.198**	**.156**	**.131**	**.082**	.060	.054	.051

Number of text character inspections for a Rand20 problem.

NL	2	4	6	8	10	20	40	80	160
HOR	3.56	2.71	2.48	2.39	2.32	2.18	2.17	2.15	2.01
QS	2.77	2.48	2.38	2.33	2.23	2.19	2.16	2.14	1.99
TBM	**2.81**	2.47	2.32	**2.27**	2.23	2.21	2.15	2.19	**1.91**
RF	14.44	8.69	6.67	5.69	4.97	3.47	2.84	2.77	5.41
FS	2.85	**2.39**	**2.27**	**2.27**	**2.20**	**2.15**	**2.13**	**2.12**	1.93

Running times for a natural language problem.

NL	2	4	6	8	10	20	40	80	160
HOR	1.094	.590	.418	.337	.282	.172	.111	.077	.059
QS	.759	.489	.375	.309	.261	.175	.125	.086	.069
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	.028

Number of text character inspections for a natural language problem.

4 Conclusion

We have presented a new efficient variant of the Boyer-Moore string matching algorithm, named Fast-Search, based on the classical bad character and good suffix rules to compute shift advancements, as other variations of the Boyer-Moore algorithm.

Rather than computing the shift advancement as the larger of the values suggested by the bad character and good suffix rules, our algorithm applies repeatedly the bad character rule until the last character of the pattern is matched correctly, and then, at the end of each matching phase, it executes one application of the good suffix rule.

It turns out that, though quadratic in the worst-case, the Fast-Search algorithm is very fast in practice and compares well with other fast variants of the Boyer-Moore algorithm, as the Horspool, Quick Search, Tuned Boyer-Moore, and Reverse Factor algorithms, in terms of both running time and number of character inspections.

References

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

[BYR92] R. A. Baeza-Yates and M. Régnier. Average running time of the Boyer-Moore-Horspool algorithm. *Theor. Comput. Sci.*, 92(1):19–31, 1992.

[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.

[GO80] 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.

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

[HS91] A. Hume and D. M. Sunday. Fast string searching. *Softw. Pract. Exp.*, 21(11):1221–1248, 1991.

[Lec00] T. Lecroq. New experimental results on exact string-matching. Rapport LIFAR 2000.03, Université de Rouen, France, 2000.

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

[Yao79] A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.