A Space-Efficient Implementation of the Good-Suffix Heuristic

Domenico Cantone, Salvatore Cristofaro, and Simone Faro

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

Abstract. We present an efficient variation of the *good-suffix* heuristic, firstly introduced in the well-known Boyer-Moore algorithm for the exact string matching problem. Our proposed variant uses only constant space, retaining much the same time efficiency of the original rule, as shown by extensive experimentation.

Keywords: string matching, experimental algorithms, text-processing, good-suffix rule, costant-space algorithms

1 Introduction

Given a text T and a pattern P (of length m) 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.

The most practical string matching algorithms show a sublinear behavior in practice, at the price of using extra memory of non-constant size to maintain auxiliary information. For instance, the Boyer-Moore algorithm [2] requires additional $\mathcal{O}(m+|\Sigma|)$-memory to compute two tables of shifts, which implement the well-known good-suffix and bad-character heuristics. Other efficient variants of the Boyer-Moore algorithm use additional $\mathcal{O}(m)$-space [7], or $\mathcal{O}(|\Sigma|)$-space [14,18], whereas, interestingly enough, two of the fastest algorithms require respectively $\mathcal{O}(|\Sigma|^2)$-space [1] and $\mathcal{O}(m \cdot |\Sigma|)$-space [5].

The first non-trivial constant-space string matching algorithm is due to Galil and Seiferas [10]. 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 [3,8,9,11,12]), and more recently two new constant-space algorithms have been presented, which have a sublinear average behavior though they are quadratic in the worst-case [6]. It is to be pointed out, though, that no constant-space algorithm which is competitive with the most efficient variants of the Boyer-Moore algorithm is known as yet.

Starting from the observation that most of the accesses to the good-suffix table are limited to very few locations, in this paper we propose a truncated good-suffix heuristic which require only constant-space and show by extensive experimentation that the Boyer-Moore algorithm and two of its more effective variants maintain much the same running times, when the truncated variant is used in place of the classical one.

Domenico Cantone, Salvatore Cristofaro, Simone Faro: A Space-Efficient Implementation of the Good-Suffix Heuristic, pp. 63–75 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

The paper is organized as follows. In Section 2 we give some preliminary notions. Then, in Section 3 we describe the preprocessing techniques introduced in the Boyer-Moore algorithm together with some efficient variants which make use of the same shift heuristics. In Section 4 we estimate the probability that a given entry of a good-suffix table is accessed and, based on such analysis, we come up with the proposal to memorize only a constant number of entries. We also show how such entries can be computed in constant-space. Subsequently, in Section 5 we present the experimental data obtained by running under various conditions the algorithms reviewed, and their modified versions. Such results confirm experimentally that by truncating the good-suffix table much the same running times are maintained. Finally, our conclusions are given in Section 6.

2 Preliminaries

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 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, where $0 \leq i \leq j < \text{length}(P)$. Moreover, for any $i, j \in \mathbb{Z}$, we put $P[i\,..\,j] = \varepsilon$ if $i > j$, and $P[i\,..\,j] = P[\max(i,0)\,..\,\min(j, \text{length}(P)-1)]$ otherwise.

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 denote by P^R the reverse of the string 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 $P[i]$ is aligned with $T[s+i]$, 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)
1.  Precompute_Globals(P)
2.  n := length(T)
3.  m := length(P)
4.  s := 0
5.  while s ≤ n − m do
6.      j := Check_Shift(s, P, T)
7.      s := s + Shift_Increment(s, P, T, j)
```

where

- the procedure *Precompute_Globals*(P) computes useful mappings, in the form of tables, which later may be accessed by the function *Shift_Increment*(s, P, T);
- the function *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 *Shift_Increment*(s, P, T, j) computes a *positive* shift increment according to the information tabulated by procedure *Precompute_Globals*(P) and to the position j of the last matched character in the pattern.

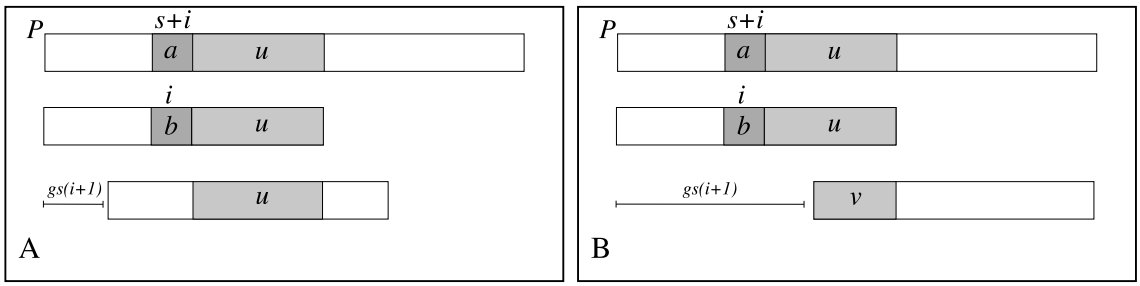

Figure 1. The good-suffix heuristic. Assuming that the suffix $u = P[i+1\,..\,m-1]$ of the pattern P has a match on the text T at shift s and that $P[i] \neq T[s+i]$, then the good-suffix heuristic attempts to align the substring $T[s+i+1\,..\,s+m-1] = P[i+1\,..\,m-1]$ with its rightmost occurrence in P preceded by a character different from $P[i]$ (see **(A)**). If this is not possible, the good-suffix heuristic suggests a shift increment corresponding to the match between the longest suffix of u with a prefix, v, of P (see **(B)**).

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) be *safe*, namely no valid shift may 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 *Check_Shift*(s, P, T) checks whether the current shift is valid by scanning the pattern from left to right, and the function *Shift_Increment*(s, P, T, j) always returns a unitary shift increment.

3 The good-suffix heuristic for preprocessing

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. In this section we focus our attention to the use of the good-suffix heuristic for preprocessing the pattern, introduced by Boyer and Moore in their celebrated algorithm [2].

The Boyer-Moore algorithm 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, it calls procedure *Boyer-Moore_Shift_Increment*(s, P, T, j) to compute the shift increment, where j is the position of last matched character in the pattern. Such procedure computes the shift increment as the maximum value suggested by the *good-suffix heuristic* and the *bad-character heuristic* below, using the functions gs_P and bc_P respectively, provided that both of them are applicable.

Boyer-Moore_Shift_Increment(s, P, T, j)
1. **if** $j > 0$ **then**
2. **return** $\max(gs_P(j), j - bc_P(T[s+j-1]) - 1)$
3. **return** $gs_P(0)$

Let us briefly review the shifting strategy of the good-suffix and the bad-character heuristics.

If the last matching character occurs at position j of the pattern P, the good-suffix heuristic suggests to align the substring $T[s+j\,..\,s+m-1] = P[j\,..\,m-1]$ with its rightmost occurrence in P (preceded by a character different from $P[j-1]$, provided that $j > 0$); this case is illustrated in Fig. 1A. If such an occurrence does not exist, the good-suffix heuristic suggests a shift increment which allows to match the longest suffix of $T[s+j\,..\,s+m-1]$ with a prefix of P; see Fig. 1B.

More formally, if the last matching character occurs at position j of the pattern P, the good-suffix heuristic states that the shift can be safely incremented by $gs_P(j)$ positions, where

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

for $i = 0, 1, \ldots, m$.

The bad-character heuristic states that if $c = T[s+j-1] \neq P[j-1]$ 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 $(s+j-1)$ in T. In the case in which c does not occur in P, then P can safely be shifted just past position $(s+j-1)$ in T. More formally, the shift increment suggested by the bad-character heuristic is given by the expression $(j - bc_P(T[s+j-1]) - 1)$, 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 in some situations the shift increment proposed by the bad-character heuristic may 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 $\mathcal{O}(m + |\Sigma|)$ and space $\mathcal{O}(m)$ and $\mathcal{O}(|\Sigma|)$, respectively, and that the overall worst-case running time of the Boyer-Moore algorithm, as described above, is linear (cf. [13]).

Due to the simplicity and ease of implementation of the bad-character heuristic, some variants of the Boyer-Moore algorithm have focused just around it and dropped the good-suffix heuristic. This is the case, for instance, of the Horspool algorithm [14], which computes shift advancements by aligning the rightmost character $T[s+m-1]$ with its rightmost occurrence on $P[0\,..\,m-2]$, if present; otherwise it shifts the pattern just past the current window.

Similarly the Quick-Search algorithm [18] 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 observation that the character $T[s+m]$ is always involved in testing for the next alignment, so that one can apply the bad-character heuristic to $T[s+m]$, rather than to the mismatching character, obtaining larger shift advancements.

A further example is given by the Berry-Ravindran algorithm [1], which extends the Quick-Search algorithm by using in the bad-character heuristic also the character $T[s+m+1]$ in addition to $T[s+m]$. In this case, the table used by the bad-character heuristic requires $\mathcal{O}(|\Sigma|^2)$-space and $\mathcal{O}(m + |\Sigma|^2)$-time complexity.

Experimental results show that the Berry-Ravindran algorithm is fast in practice and performs a low number of text/pattern character comparisons and that the Quick-Search algorithm is very fast especially for short patterns (cf. [16]).

The role of the good-suffix heuristic in practical string matching algorithms has recently been reappraised, also in consideration of the fact that often it is as effective as the bad-character heuristic, especially in the case of non-periodic patterns.

This is the case of the Fast-Search algorithm [4], a very simple, yet efficient, variant of the Boyer-Moore algorithm. The Fast-Search algorithm computes its shift increments by applying the bad-character heuristic if and only if a mismatch occurs during the first character comparison, namely, while comparing characters $P[m-1]$ and $T[s+m-1]$, where s is the current shift. In all other cases it uses the good-suffix heuristic. This translates in the following pseudo-code:

Fast-Search_Shift_Increment(s, P, T, j)
1. $m := \text{length}(P)$
2. **if** $j = m - 1$ **then**
3. **return** $bc_P(T[s+m-1])$
4. **else**
5. **return** $gs_P(j)$

A more effective implementation of the Fast-Search algorithm is obtained by iterating the bad-character heuristic until the last character $P[m-1]$ of the pattern is matched correctly against the text, at which point it is known that $T[s+m-1] = P[m-1]$, so that the subsequent matching phase can start with the $(m-2)$-*nd* character of the pattern. At the end of the matching phase the good-suffix heuristic is applied to compute the shift increment.

Another example is the Forward-Fast-Search algorithm [5], which maintains the same structure of the Fast-Search algorithm, but is based upon a modified version of the good-suffix heuristic, called *forward good-suffix* heuristic, which uses a look-ahead character to determine larger shift advancements. More precisely, if the last matching character occurs at position $j \leq m-1$ of the pattern P, the forward good-suffix heuristic suggests to align the substring $T[s+j \,..\, s+m-1]$ with its rightmost occurrence in P preceded by a character different from $P[j-1]$. If such an occurrence does not exist, the forward good-suffix heuristic proposes a shift increment which allows to match the longest suffix of $T[s+j \,..\, s+m-1]$ with a prefix of P. This corresponds to advance the shift s by $\overrightarrow{gs}_P(j, T[s+m])$ positions, where

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

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

The forward good-suffix heuristic requires a table of size $m \cdot |\Sigma|$ which can be constructed in time $\mathcal{O}(m \cdot \max(m, |\Sigma|))$.

Experimental results show that both the Fast-Search and the Forward-Fast-Search algorithms, though not linear, achieve very good results especially in the case of very short patterns or small alphabets.

4 Truncating the Good-Suffix Tables

Let us assume that we run the Boyer-Moore algorithm on a pattern P and a text T. Then, at the end of each matching phase, the Boyer-Moore algorithm accesses the entry at position $j > 0$ in the good-suffix table if and only if the last matched

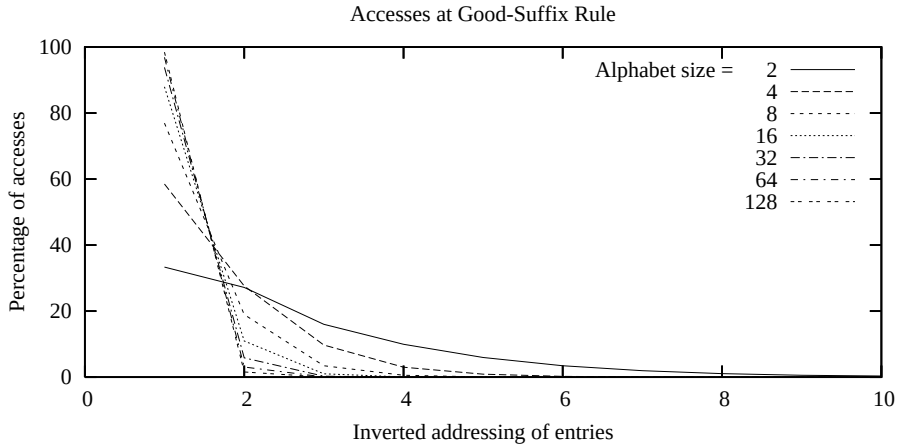

Figure 2. The percentage of accesses for each entry of the good-suffix heuristic, for different sizes of the alphabet. The values have been computed by running the Fast-Search algorithm with a set of 200 pattern, of length 40, and a 20Mb text buffer as input. The values in each curve f are relative to the inverted addressing of the entries, i.e. $f(j)$ is the percentage of accesses to the entry at position $m - j$.

character in the pattern occurs at position j of the pattern, i.e. if $P[j\,..\,m-1] = T[s+j\,..\,s+m-1]$ and $P[j-1] \neq T[s+j-1]$, where s is the current shift. Likewise, the Boyer-Moore algorithm accesses the entry at position $j = 0$ if and only if $P[0\,..\,m-1] = T[s\,..\,s+m-1]$, i.e. if and only if s is a valid shift.

Therefore, it is intuitively expected that the probability to access an entry at position j of the good-suffix table becomes higher as the value of j increases. In other words, it is expected that entries on the right-hand side of the good-suffix table have (much) higher probability to be accessed than entries on the left end side.

The above considerations, which will be formalized below under suitable simplifying hypotheses, suggest that the initial segment of the good-suffix tables can be dropped, without affecting very much the performance of the algorithm. In fact, we will see that in most cases, it is enough to maintain just a few entries of the good-suffix tables.

For the sake of simplicity, in the following analysis we will assume that the text T and pattern P are strings over a common alphabet Σ of size σ, randomly selected relatively to a uniform distribution.

Thus, for a shift $0 \leq s \leq n - m$ in T and a position $0 \leq j < m$ in P, the probability that $P[j] = T[s+j]$ is $1/\sigma$, whereas the probability that $P[j] \neq T[s+j]$ is $(\sigma - 1)/\sigma$.

Therefore, the probability p_j that j is the position of the last matched character in the pattern P, relatively to a shift s of the text, is given by

$$p_j = \begin{cases} \dfrac{\sigma - 1}{\sigma^{m-j+1}} & \text{if } 0 < j \leq m \\ \dfrac{1}{\sigma^m} & \text{if } j = 0\,. \end{cases}$$

Plainly, p_j is also the probability that location j of the good-suffix table is accessed.

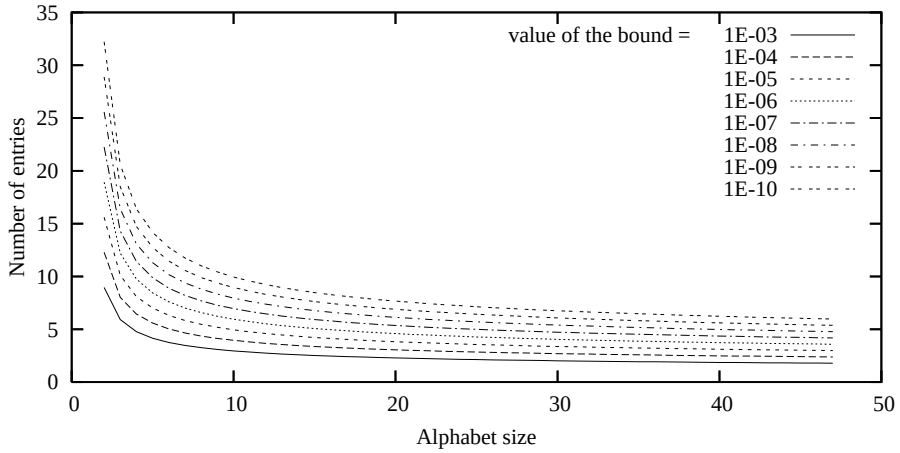

Figure 3. The function $\log_\sigma \frac{\sigma-1}{\beta} - 1$, for different values of the bound β. Note that if the bound β is greater or equal to 10^{-4}, then the number of entries accessed with probability greater than β is always no greater than 12 and in most cases no greater than 3.

As experimental evidence of the above analysis, we report in Fig. 2 the plots of the accesses to each entry of the good-suffix table, for different sizes of the alphabet, when running the Fast-Search algorithm with a set of 200 patterns of length 40 and a 20Mb text buffer as input. More precisely, for each function f in Fig. 2, $f(j)$ is the percentage of accesses to the entry at position $m-j$ in the good-suffix table. We can observe that, in general, only a very small number of entries is really used during a computation and, in particular, when the alphabet size is greater than or equal to 16 about 98% of the accesses are limited to the last three entries of the table.

We can readily evaluate the number $K_{\sigma,\beta}$ of entries of the good-suffix table which are accessed with probability greater than a fixed threshold $0 < \beta < 1$, for an alphabet of size σ. To begin with, notice that if $p_j > \beta$, then $\frac{\sigma-1}{\sigma^{m-j+1}} > \beta$, so that

$$j > m + 1 - \left\lceil \log_\sigma \frac{\sigma-1}{\beta} \right\rceil . \quad \text{and} \quad m_{\sigma,\beta} \leq \left\lceil \log_\sigma \frac{\sigma-1}{\beta} \right\rceil - 1 .$$

Observe that for $\bar{\beta} = 10^{-4}$, we have $K_{\sigma,\bar{\beta}} \leq 12$. Additionally, we have $K_{\sigma,\bar{\beta}} \leq 3$, for $14 \leq \sigma \leq 39$, and $K_{\sigma,\bar{\beta}} \leq 2$, for $\sigma \geq 40$. In other words, for alphabets of at least 14 characters, at most the last three entries of the good-suffix table are accessed with probability at least 10^{-4} (under the assumption of uniform distribution). Fig. 3 shows the shape of the function $\log_\sigma \frac{\sigma-1}{\beta} - 1$ for the following values of the bound $\beta = 10^{-3}, 10^{-4}, \ldots, 10^{-10}$. Note that if the bound β is greater or equal to 10^{-4}, then the number of entries accessed with probability greater than β is always no greater than 12 and in most cases no greater than 3.

4.1 The Bounded-Good-Suffix Heuristic

The above considerations justify the following *bounded good-suffix* heuristic. Let $\beta > 0$ be a fixed bound[1] and let $K = \left\lceil \log_\sigma \frac{\sigma-1}{\beta} \right\rceil - 1$, where, as usual, σ denotes the size of the alphabet. Then the bounded good-suffix heuristic works as follows.

[1] A good practical choice is $\beta = 10^{-4}$, as shown in Section 5.

During a matching phase, if the first mismatch occurs at position i of the pattern P and $i \geq m - K$, the bounded good-suffix heuristic suggests that the pattern is shifted $gs_P(i+1)$ positions to the right. Otherwise, if the first mismatch occurs at position i of the pattern P, with $i < m - K$, or if the pattern P matches the current window in the text, then the bounded good-suffix heuristic suggests that the pattern is shifted one position to the right.

More formally, if the first mismatch occurs at position i of the pattern P, the bounded good-suffix heuristic suggests that the shift s can be safely advanced $\beta gs_P(i-m+K)$ positions to the right, where, for $j = K - m - 1, \ldots, K - 1$, we have

$$\beta gs_P(j) =_{\text{Def}} \begin{cases} gs_P(j+m-K+1) & \text{if } j \geq 0 \\ 1 & \text{otherwise}. \end{cases}$$

Likewise, the *bounded forward good-suffix heuristic* suggests that when the first mismatch occurs at position i of the pattern P, then the shift s is advanced by $\overrightarrow{\beta gs}_P(i-m+k, T[s+m])$ positions to the right, where, for $j = K - m - 1, \ldots, K - 1$ and $c \in \Sigma$, we have

$$\overrightarrow{\beta gs}_P(j,c) =_{\text{Def}} \begin{cases} \overrightarrow{gs}_P(j+m-K+1,c) & \text{if } j \geq 0 \\ 1 & \text{otherwise}. \end{cases}$$

By way of example, when the bounded good-suffix heuristic is adopted in place of the good-suffix heuristic, the *Shift_Increment* procedure of the Boyer-Moore algorithm becomes:

```
βBoyer-Moore_Shift_Increment(s, P, T, j, σ, β)
1.   m := length(P)
2.   K := ⌈log_σ (σ−1)/β⌉ − 1
3.   if j ≥ m − K − 1 then
4.       if j > 0 then
5.           return max(βgs_P(j − m + K − 1), j − bc_P(T[s + j − 1]))
6.       else return βgs_P(0)
7.   else if j > 0 then
8.           return max(1, j − bc_P(T[s + j − 1]))
9.       else return 1
```

Next we discuss how the bounded good-suffix function βgs_P can be constructed (analogous remarks apply for the bounded forward good-suffix function). A first very natural way to compute the function βgs_P consists in computing a slightly modified version of the standard good-suffix function gs_P, and then keeping only the last K entries of the function. However such procedure, based on the one firstly given in [2] and later corrected in [17], has $\mathcal{O}(m)$-time and space complexity.

An alternative way to compute the bounded good-suffix function using only constant space, but still in $\mathcal{O}(m)$ worst-case time, is given by procedure *Precompute_βgs*, whose pseudo-code is presented below:

Precompute_βgs(P, σ, β)
1. $m := \text{length}(P)$
2. $K := \left\lceil \log_\sigma \frac{\sigma-1}{\beta} \right\rceil - 1$
3. **for** $\ell := 0$ **to** $K-1$ **do**
4. $\quad j := m-2$
5. $\quad$ **repeat**
6. $\quad\quad q := j - occur(P^R_{m-\ell..m-1}, P^R_{0..j})$
7. $\quad\quad j := q-1$
8. $\quad$ **until** $q < \ell$ **or** $P[m-\ell-1] \neq P[q-\ell]$
9. $\quad \beta gs_P(K-\ell-1) := m-q-1$
10. **return** βgs_P

First of all, we give the specification of the function *occur*, which is called by procedure *Precompute_βgs*. Given two strings X and Y, $occur(X, Y)$ computes the leftmost occurrence of X in Y, i.e.,

$$occur(X, Y) =_{\text{Def}} \min\{p \geq 0 \mid Y[\,p\,..\,p+|X|-1] = X\} \cup \{|Y|\}\,.$$

Observe that the function $occur(X, Y)$ can be computed by means of a linear-time string matching algorithm such as the Knuth-Morris-Pratt algorithm [15], thus requiring $\mathcal{O}(|X|+|Y|)$-time and $\mathcal{O}(|X|)$ additional space.

We are now ready to explain how the procedure *Precompute_βgs* works.

For $\ell = 0, 1, \ldots, K-1$, the ℓ-th iteration of the **for**-loop in line 3 finds the rightmost occurrence, $P[q-\ell+1\,..\,q]$, in P of its suffix of length ℓ preceded by a character different from $P[m-\ell-1]$. If such an occurrence does not exist, the ℓ-th iteration finds the rightmost position $q < \ell$ in the pattern such that $P[0\,..\,q] = P[m-q-1\,..\,m-1]$. More precisely, the search is performed within the **repeat**-loop in line 5, by means of repeated calls of type $occur((P[m-\ell\,..\,m-1])^R, (P[0\,..\,j])^R)$, each of which looks for the leftmost occurrence of the reverse of $P[m-\ell\,..\,m-1]$ in the reverse of $P[0\,..\,j]$. When such an occurrence is found at position q, so that $P[q-\ell+1\,..\,q]$ is a suffix of P, it is checked whether $q < \ell$ holds or whether the character $P[m-\ell-1]$ is different from $P[q-\ell]$. If any of such conditions is true, the **repeat**-loop stops, whereas if both conditions are false, another iteration is performed with $j = q-1$.

The value q, discovered during the ℓ-th iteration of the **for**-loop in line 3, is then used in line 9 to set the $(K-\ell-1)$-th entry of the βgs_P function to $m-q-1$.

Concerning the time and space analysis of the procedure *Precompute_βgs*, notice that each iteration of the **for**-loop, for $\ell = 0, 1, \ldots, K-1$, takes $\mathcal{O}(K+m)$-time, using only $\mathcal{O}(K)$-space. Indeed, each call $occur(P^R_{m-\ell..m-1}, P^R_{0..j})$ in the **repeat**-loop takes time proportional to $j-r$, where $r = occur(P^R_{m-\ell..m-1}, P^R_{0..j})$, and uses $\mathcal{O}(\ell)$ (reusable) space. Additionally, after each such call, the value of j is decreased by $r+1$. Hence, the overall running time of all calls to the function *occur* made in the **repeat**-loop is bounded by $\mathcal{O}(K+m)$, for each iteration of the **for**-loop.

Since the number of iterations in the **for**-loop is K, the overall running time of the procedure *Precompute_βgs* is $\mathcal{O}(K^2 + Km)$.

Notice that if we fix the value of $\beta = 10^{-4}$, then we have $K \leq 12$, as observed just before Section 4.1. Therefore, in such a case, the time and space complexity of the procedure *Precompute_βgs* are $\mathcal{O}(1)$ and $\mathcal{O}(m)$, respectively.[2]

[2] As will be shown in Section 5, the choice $\beta = 10^{-4}$ has very good practical results.

5 Experimental Results

To evaluate experimentally the impact of the bounded good-suffix heuristic, we have chosen to test it with the Boyer-Moore algorithm (in short, BM) and with two of its fastest variants in practice, namely the Fast-Search (FS) and the Forward-Fast-Search (FFS) algorithms. Their modified versions, obtained by using the bounded good-suffix heuristic in place of the good-suffix heuristic (in the case of the Boyer-Moore and the Fast-Search algorithms) and the bounded forward good-suffix heuristic in place of the forward good-suffix heuristic (in the case of the Forward-Fast-Search algorithm), are respectively denoted in short by βBM, βFS, and βFFS.

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, all algorithms have been tested on seven Randσ problems, for $\sigma = 2, 4, 8, 16, 32, 64, 128$, with patterns of length $m = 2, 4, 8, 10, 20, 40, 80$ and 160, and on two real data problems.

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 real data problems have been performed on a 180Kb natural language text file, containing the "Hamlet" by William Shakespeare (NL), and on a 2.4Mb file containing a protein sequence from the human genome. In both cases, the patterns to be searched for have been constructed by selecting 200 random substrings of length m from the files, for each $m = 2, 4, 8, 10, 20, 40, 80$ and 160.

For the implementation of the bounded versions of the (forward) good-suffix heuristic we have used the bound $\beta = 10^{-4}$.

With the exception of the last two tables in which running times are expressed in thousandths of seconds, all other running times in the remaining tables are expressed in hundredths of seconds.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
BM	46.82	39.77	31.51	25.89	21.23	19.93	19.56	17.56	15.64
βBM	47.34	40.33	31.94	26.10	21.64	20.24	19.82	17.95	15.93
FS	35.62	31.28	25.80	21.93	19.28	18.33	17.91	16.48	14.96
βFS	36.32	32.34	26.36	22.46	19.37	18.35	17.94	16.71	14.95
FFS	31.02	28.39	23.46	19.76	17.83	16.97	16.64	15.03	13.78
βFFS	37.27	32.44	25.83	21.71	19.30	18.53	18.18	16.33	14.95

Running times in hundredths of seconds for a Rand2 problem

$\sigma = 4$	2	4	6	8	10	20	40	80	160
BM	38.84	28.41	23.23	21.04	20.15	19.30	18.95	17.70	16.42
βBM	39.09	28.58	23.35	21.29	20.30	19.54	19.10	17.87	16.52
FS	26.08	21.15	18.95	18.14	17.64	17.07	16.70	15.91	14.84
βFS	26.56	21.49	19.17	18.30	17.65	17.12	16.72	16.02	14.93
FFS	25.14	20.58	18.58	17.34	16.52	16.11	15.90	14.32	13.29
βFFS	26.61	21.18	18.68	17.66	16.70	16.30	16.02	14.55	13.35

Running times in hundredths of seconds for a Rand4 problem

$\sigma = 8$	2	4	6	8	10	20	40	80	160
BM	33.24	23.16	18.97	17.86	17.36	17.12	17.00	16.42	15.83
βBM	33.01	23.09	19.14	17.93	17.36	17.14	17.09	16.47	15.91
FS	21.02	18.26	16.43	16.04	15.93	15.81	15.82	15.29	14.88
βFS	21.32	18.34	16.46	16.10	15.96	15.88	15.75	15.39	14.91
FFS	20.84	18.23	16.39	16.05	15.78	15.64	15.26	13.96	13.18
βFFS	21.12	18.36	16.43	16.05	15.82	15.67	15.31	14.00	13.02

Running times in hundredths of seconds for a Rand8 problem

$\sigma = 16$	2	4	6	8	10	20	40	80	160
BM	31.09	21.37	18.18	16.39	16.04	15.92	15.86	15.64	15.39
βBM	30.45	21.31	18.41	16.42	16.04	15.85	15.84	15.60	15.32
FS	19.14	16.77	15.94	15.66	15.40	15.32	15.28	15.12	14.91
βFS	19.29	16.89	16.05	15.61	15.45	15.39	15.25	15.09	14.92
FFS	19.19	16.84	15.90	15.65	15.44	15.35	15.11	13.98	13.26
βFFS	19.22	16.88	16.00	15.60	15.36	15.29	15.00	13.84	13.09

Running times in hundredths of seconds for a Rand16 problem

$\sigma = 32$	2	4	6	8	10	20	40	80	160
BM	29.96	20.38	17.49	16.03	15.78	15.47	15.25	15.15	15.02
βBM	29.44	19.97	17.63	16.12	15.79	15.47	15.19	15.11	15.03
FS	18.78	16.38	15.86	15.52	15.12	15.13	14.75	14.70	14.61
βFS	18.87	16.38	15.84	15.54	15.13	15.12	14.78	14.70	14.66
FFS	18.89	16.47	15.87	15.56	15.15	15.15	14.79	14.21	13.54
βFFS	18.84	16.32	15.89	15.52	15.12	15.10	14.65	14.05	13.35

Running times in hundredths of seconds for a Rand32 problem

$\sigma = 64$	2	4	6	8	10	20	40	80	160
BM	29.50	19.39	17.31	15.96	15.63	15.39	14.39	13.65	13.51
βBM	29.00	19.51	17.53	15.97	15.66	15.27	14.54	13.65	13.51
FS	18.60	16.24	15.75	15.61	14.96	15.06	14.22	13.63	13.32
βFS	18.71	16.35	15.81	15.51	14.94	15.10	14.20	13.63	13.47
FFS	18.63	16.30	15.78	15.50	14.98	15.16	14.29	13.51	13.44
βFFS	18.73	16.33	15.82	15.55	14.97	15.14	14.24	13.36	13.04

Running times in hundredths of seconds for a Rand64 problem

$\sigma = 128$	2	4	6	8	10	20	40	80	160
BM	29.36	19.29	17.13	15.96	15.59	15.27	14.00	12.42	11.90
βBM	28.84	19.40	17.40	15.95	15.64	15.24	13.93	12.38	11.96
FS	18.59	16.32	15.78	15.57	14.90	15.32	13.84	12.37	11.93
βFS	18.58	16.38	15.83	15.61	14.88	15.30	13.87	12.35	12.07
FFS	18.59	16.29	15.83	15.59	14.96	15.34	13.96	12.51	12.42
βFFS	18.56	16.28	15.90	15.60	14.95	15.37	13.86	12.34	11.87

Running times in hundredths of seconds for a Rand128 problem

NL	2	4	8	16	32	64	128	256	512
BM	5.56	3.35	3.46	2.75	2.65	2.70	2.30	1.45	2.30
βBM	5.57	3.11	2.56	2.25	2.41	2.60	2.35	1.30	2.05
FS	2.65	2.60	2.60	2.46	2.45	2.25	1.70	1.40	1.65
βFS	3.56	2.71	2.87	2.50	2.30	2.81	1.15	1.35	1.76
FFS	3.55	2.85	2.40	2.90	2.85	2.65	2.42	2.21	1.91
βFFS	2.45	2.75	2.30	2.46	2.41	2.55	1.40	1.25	1.26

Running times in thousandths of seconds for a natural language problem

Prot	2	4	8	16	32	64	128	256	512
BM	73.16	49.87	43.46	38.75	38.46	37.19	37.26	34.95	34.55
βBM	71.94	49.08	43.49	38.76	37.59	37.77	36.74	35.53	34.39
FS	45.81	40.01	38.06	36.85	35.97	36.34	35.66	33.77	33.34
βFS	45.38	39.65	37.91	36.99	36.41	36.10	35.24	33.79	33.86
FFS	45.26	39.71	37.61	37.50	37.43	36.45	36.21	33.90	36.40
βFFS	45.82	40.01	38.01	37.20	35.80	36.55	34.95	32.45	31.65

Running-times in thousandths of seconds for a protein sequence problem

The above experimental results show that the algorithms βBM, βFS, and βFFS have much the same running times of the algorithms BM, FS, and FFS. Only when the size of the alphabet is 2 the "bounded" versions have a slightly worse performance than their counterparts, especially for short patterns. On the other hand, as the size of the alphabet and pattern increases, often the "bounded" versions moderately outperform their counterpart. In particular, this behavior is more noticeable in the case of the Forward-Fast-Search algorithm and in the cases of the real data problems. The latter remark shows that our simplifying hypotheses in the analysis put forward in Section 4 do not lead to unrealistic results.

6 Conclusions

Space and time economy are essential features of any practical algorithm. However, they are often sacrificed in favor of asymptotic efficiency. This is the case of the most practical string matching algorithms which show in practice a sublinear behavior at the price of using extra memory of non-constant size to maintain auxiliary information. The Boyer-Moore algorithm, for instance, requires additional $\mathcal{O}(m)$ and $\mathcal{O}(|\Sigma|)$-space to compute the tables relative to the good-suffix and to the bad-character heuristics, respectively.

In this paper we have presented a practical modification of the good-suffix heuristic, called bounded good-suffix heuristic, which uses only constant space and can be computed in $\mathcal{O}(m)$-time and constant space.

Through an extensive collection of experimental tests on the Boyer-Moore algorithm and two of its most efficient variants (namely the algorithms Fast-Search and Forward-Fast-Search) we have shown that the "bounded" versions are comparable with their counterparts, which are often outperformed by them.

We are currently investigating the problem of finding an effective string matching algorithm which requires only extra constant space. To this purpose, we expect that the bad-character heuristic (which needs $\mathcal{O}(|\Sigma|)$-space) needs to be dropped and substituted by a heuristic of a different kind.

References

1. T. Berry and S. Ravindran: *A fast string matching algorithm and experimental results*, in Proceedings of the Prague Stringology Conference '99, J. Holub and M. Šimánek, eds., Czech Technical University in Prague, 1999, Collaborative Report DC–99–05.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) 1977, pp. 762–772.
3. D. Breslauer: *Saving comparisons in the Crochemore-Perrin string matching algorithm.* Theor. Comput. Sci., 158 1996.
4. D. Cantone and S. Faro: *Fast-Search: a new variant of the Boyer-Moore string matching algorithm*, in Proceedings of Second International Workshop on Experimental and Efficient Algorithms (WEA 2003), K. Jansen, M. Margraf, M. Mastrolilli, and J. Rolim, eds., vol. 2647 of Lecture Notes in Computer Science, Springer-Verlag, 2003.
5. D. Cantone and S. Faro: *Forward-Fast-Search: another fast variant of the Boyer-Moore string matching algorithm*, in Proceedings of the Prague Stringology Conference '03, M. Šimánek, ed., Czech Technical University in Prague, 2003.
6. D. Cantone and S. Faro: *Searching for a substring with constant extra-space complexity*, in Proceedings of Third International Conference on FUN with Algorithms (FUN 2004), P. Ferragina and R. Grossi, eds., Edizioni Plus, Università di Pisa, 2004.

7. 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) 1994.
8. M. Crochemore, L. Gąsieniec, and W. Rytter: *Constant-space string-matching in sublinear average time.* Theor. Comput. Sci., 218(1) 1999.
9. M. Crochemore and D. Perrin: *Two-way string-matching.* Journal of the ACM, 38(3) 1991.
10. Z. Galil and J. Seiferas: *Saving space in fast string-matching.* SIAM J. Comput., 9(2) 1980.
11. L. Gąsieniec, W. Plandowski, and W. Rytter: *Constant-space string matching with smaller number of comparisons: sequential sampling*, in Proc. 6th Symp. Combinatorial Pattern Matching, Z. Galil and E. Ukkonen, eds., vol. 937 of Lecture Notes in Computer Science, Springer-Verlag, 1995.
12. 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) 1995.
13. L. J. Guibas and A. M. Odlyzko: *A new proof of the linearity of the Boyer-Moore string searching algorithm.* SIAM J. Comput., 9(4) 1980.
14. R. N. Horspool: *Practical fast searching in strings.* Softw. Pract. Exp., 10(6) 1980.
15. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt: *Fast pattern matching in strings.* SIAM J. Comput., 6(2) 1977, pp. 323–350.
16. T. Lecroq: *New experimental results on exact string-matching*, Université de Rouen, France, 2000, Rapport LIFAR 2000.03.
17. W. Rytter: *A correct preprocessing algorithm for Boyer-Moore string searching.* SIAM J. Comput., 9 1980.
18. D. M. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) 1990.