A unifying look at the Apostolico-Giancarlo string-matching algorithm

Maxime Crochemore, Christophe Hancart, Thierry Lecroq

A Unifying Look at the Apostolico–Giancarlo String-Matching Algorithm

MAXIME CROCHEMORE, *IGM (Institut Gaspard-Monge), Université de Marne-la-Vallée, 77454 Marne-la-Vallée CEDEX 2, France. E-mail: mac@univ-mlv.fr, URL: http://www-igm.univ-mlv.fr/~mac*

CHRISTOPHE HANCART, *LIFAR (Laboratoire d'Informatique Fondamentale et Appliquée de Rouen), Faculté des Sciences et des Techniques, Université de Rouen, 76821 Mont-Saint-Aignan Cedex, France. E-mail: hancart@dir.univ-rouen.fr*

THIERRY LECROQ, *LIFAR (Laboratoire d'Informatique Fondamentale et Appliquée de Rouen), and ABISS (Atelier Biologie Informatique Statistiques Sociolinguistique), Faculté des Sciences et des Techniques, Université de Rouen, 76821 Mont-Saint-Aignan Cedex, France. E-mail: lecroq@dir.univ-rouen.fr, URL: http://www-igm.univ-mlv.fr/~lecroq*

ABSTRACT: String matching is the problem of finding all the occurrences of a pattern in a text. We present a new method to compute the combinatorial shift function ("matching shift") of the well-known Boyer–Moore string matching algorithm. This method implies the computation of the length of the longest suffixes of the pattern ending at each position in this pattern. These values constituted an extra-preprocessing for a variant of the Boyer-Moore algorithm designed by Apostolico and Giancarlo. We give here a new presentation of this algorithm that avoids extra preprocessing together with a tight bound of $1.5n$ character comparisons (where n is the length of the text).

Keywords: string matching, analysis of algorithm

1 Introduction

The string matching problem consists in finding one or more usually all the occurrences of a pattern x of length m in a text y of length n. It can occur in information retrieval, bibliographic search and most recently it has some applications in molecular biology. It has been extensively studied and numerous techniques and algorithms have been de-

signed to solve this problem (see [7], [19] and [3]). We are interested here in the problem where the pattern is given first and can then be searched in various texts. Thus a preprocessing phase is allowed on the pattern.

Basically a string-matching algorithm uses a window to scan the text. The size of this window is equal to the length of the pattern. It first aligns the left ends of the window and the text. Then it checks if the pattern occurs in the window (this specific work is called an *attempt*) and *shifts* the window to the right. It repeats the same procedure again until the right end of the window goes beyond the right end of the text. One of the most famous string matching algorithm was given in 1977 by Boyer and Moore [2]. Its main feature is that at each attempt it scans the characters of the pattern from right to left which enables it to "jump" over some portions of the text and therefore to save some comparisons. Its main drawback is that after a shift, it forgets all the characters it has previously matched. This make the complexity analysis of the Boyer-Moore algorithm very difficult. Cole [4] proved, a long time after the design of the algorithm, the tight bound of $3n - n/m$ comparisons to locate a non-periodic pattern. When searching for all the occurrences of the pattern in the text, the Boyer-Moore algorithm has a quadratic worst-case time complexity. The exact complexity is $O(n + rm)$ where r is the number of occurrences of the pattern in the text (see [14]). A major difficulty when one wants to implement the Boyer-Moore algorithm is to understand the computation of the "matching shift" which is one of the two shift functions usually used by the algorithm. We give a new method to compute this function. This method uses values needed by the Apostolico-Giancarlo algorithm. To remedy the oblivious feature of the Boyer-Moore algorithm, Apostolico and Giancarlo [1] gave in 1986 an algorithm which remembers at each position of the text previously aligned with the right end of the pattern, the length of the longest suffix of the pattern ending at this position. This technique leads to an upper bound of $2n - m + 1$ text character comparisons. Actually remembering only the last suffix of the pattern matched in the text also leads to an upper bound of $2n$ comparisons. The Turbo-BM algorithm [5] applies this strategy and reaches this bound. In analyzing more in detail the Apostolico-Giancarlo algorithm, we are able to give an upper bound of $\frac{3}{2}n$ text characters comparisons. We show that this bound is tight by exhibiting a family of patterns and texts reaching this bound. Moreover we reformulate the algorithm in order to save other kinds of comparisons and to improve the length of the shifts.

This paper is organized as follows: Section 2 recalls briefly the Boyer–Moore algorithm; in Section 3 we give an history of the Boyer–Moore algorithm and its variants; in Section 4 we give a method to compute the matching shift function of the Boyer–Moore algorithm and in Section 5 we describe a new version of the Apostolico–Giancarlo algorithm; a new tight bound of $1.5n$ text character comparisons is proved in the same section. Throughout this paper the pattern is denoted by a word x of length m, $x = x[0..m-1]$. The text is denoted by a word y of length n, $y = y[0..n-1]$. Both x and y are built over a finite alphabet Σ of size σ.

FIG. 1: Typical situation during the Boyer–Moore algorithm: a suffix u of the pattern is found and a mismatch occurs between a character a in the pattern x and a character b in the text y.

2 Boyer–Moore string-matching algorithm

The Boyer-Moore algorithm is considered as the most efficient string matching algorithm in usual applications. A simplified version of it or the entire algorithm is often implemented in a text editor for the "search" and "substitute" commands.

The algorithm scans the characters of the pattern from right to left beginning with the rightmost symbol. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the pattern to the right. These two shift functions are called the *matching shift* and the *occurrence shift*.

Assume that a suffix u of x has been matched and a mismatch occurs between the character $x[i] = a$ of the pattern and the character $y[i+j] = b$ of the text during an attempt where x is aligned with $y[j..j+m-1]$. Then, $x[i+1..m-1] = y[i+j+1..j+m-1] = u$ and $a = x[i] \neq y[i+j] = b$ (see Fig. 1).

The matching shift consists in aligning the substring $u = x[i+1..m-1] = y[i+j+1..j+m-1]$ with one of its reoccurrences in x. Informally, let us distinguish three matching shift cases on the grounds of the restrictions imposed on the character c preceding this reoccurrence:

weak matching shift :
there is no condition on the character c preceding u, it is then possible that $c = a$ (see Fig. 2).

strong matching shift :
the character c must be different from the character a (see Fig. 3).

best matching shift :
the character c must be equal to b (see Fig. 4).

It is not too difficult to see that the following inequality holds:

$$|\text{weak matching shift}| \leq |\text{strong matching shift}| \leq |\text{best matching shift}|$$

where the absolute value of a shift denotes the length of the shift.

If there exists no other occurrence of u, the matching shift consists in aligning the longest suffix v of $y[i+j+1..j+m-1]$ with a matching prefix of x (see Fig. 5).

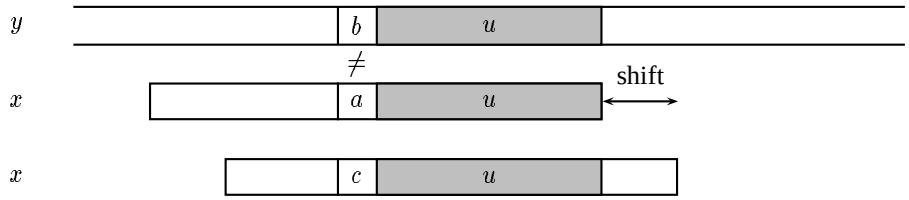

FIG. 2. Weak matching shift: c can be equal to a.

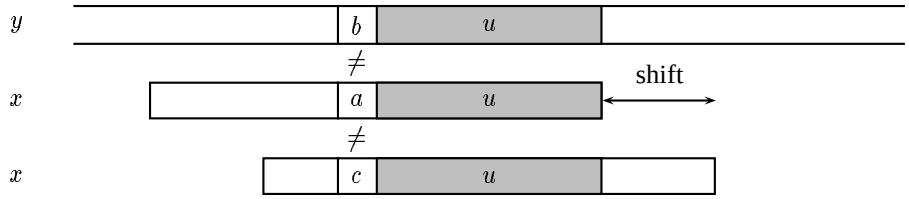

FIG. 3. Strong matching shift: $c \neq a$.

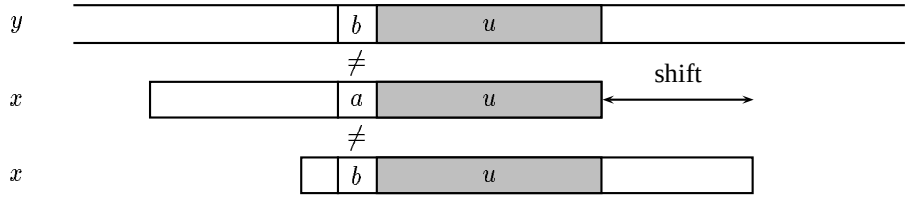

FIG. 4. Best matching shift.

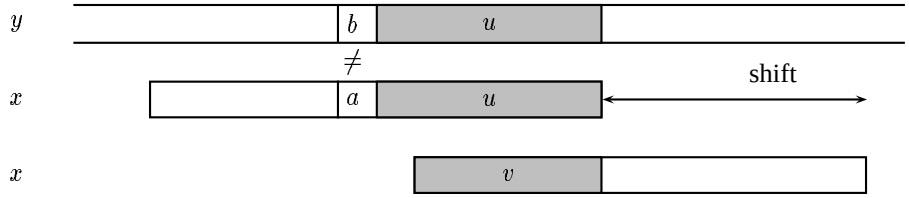

FIG. 5. Matching shift, only a prefix of u reappears in x.

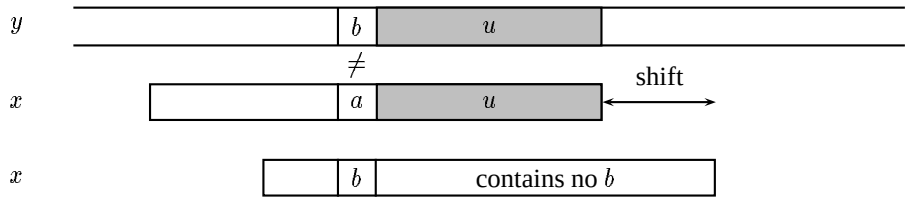

FIG. 6. Occurrence shift, b appears in x.

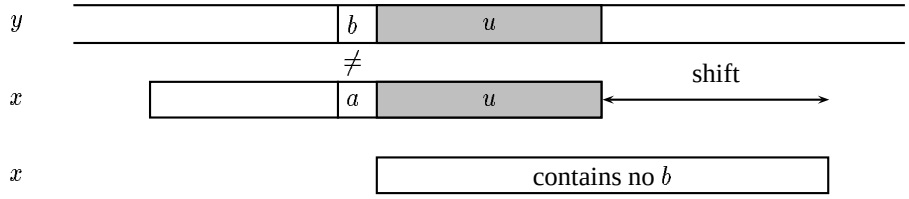

FIG. 7. Occurrence shift, b does not appear in x.

The occurrence shift consists in aligning the text character $y[i+j]$ with its rightmost occurrence in $x[0..m-2]$ (see Fig. 6). If $y[i+j]$ does not appear in the pattern x, no occurrence of x in y can include $y[i+j]$, and the left end of the pattern is aligned with the character immediately after $y[i+j]$, namely $y[i+j+1]$ (see Fig. 7).

The three shift functions will be denoted by the variables *wMatch*, *sMatch*, and *bMatch*. We will define these three variables with the aid of the condition functions *Cs*, *Cos* and *Cob*:

For $0 \le i \le m-1$, $1 \le s \le m$ and $a \in \Sigma$, let us define the following conditions.

- The condition of suffix *Cs* is defined for a position i and a shift s:

$$Cs(i,s) = \begin{cases} 0 < s \le i \text{ and } x[i-s+1\,..\,m-s-1] \text{ is a suffix of } x \\ \text{or} \\ s > i \text{ and } x[0\,..\,m-s-1] \text{ is a suffix of } x \end{cases}$$

- The strong condition of occurrence *Cos* is defined for a position i and a shift s:

$$Cos(i,s) = \begin{cases} 0 \le s \le i \text{ and } x[i-s] \ne x[i] \\ \text{or} \\ s > i \end{cases}$$

- The best condition of occurrence *Cob* is defined for a position i, a character a and a shift s:

$$Cob(i,a,s) = \begin{cases} 0 \le s \le i \text{ and } x[i-s] = a \\ \text{or} \\ s > i \end{cases}$$

```
BOYER-MOORE(x, m, y, n)
1  j ← 0
2  while j ≤ n − m
3      do i ← m − 1
4         while i ≥ 0 and x[i] = y[i + j]
5             do i ← i − 1
6         if i < 0
7           then REPORT(j)
8                j ← j + MATCH(0, 0)
9           else j ← j + max(MATCH(i, j), occ[y[i + j]] − m + i + 1)
```

FIG. 8. The Boyer–Moore string matching algorithm.

Then, for $0 \leq i \leq m-1$ and $0 \leq j \leq n-1$:

- the weak matching shift is defined by:

$$wMatch[i,j] = \min\{s > 0 \mid Cs(i,s) \text{ holds}\}$$

- the strong matching shift is defined by:

$$sMatch[i,j] = \min\{s > 0 \mid Cs(i,s) \text{ and } Cos(i,s) \text{ hold}\}$$

- the best matching shift is defined by:

$$bMatch[i,j] = \min\{s > 0 \mid Cs(i,s) \text{ and } Cob(i, y[i+j], s) \text{ hold}\}$$

Remark: $wMatch[0] = sMatch[0] = bMatch[0,j]$ is equal to the period of x for all $0 \leq j \leq n-1$.

The occurrence shift is defined as follows. For $a \in \Sigma$:

$$occ[a] = \begin{cases} \min\{i \mid 1 \leq i \leq m-1 \text{ and } x[m-1-i] = a\} & \text{if } a \text{ appears in } x, \\ m & \text{otherwise.} \end{cases}$$

The Boyer-Moore algorithm is shown in Fig. 8. The function MATCH(i,j) can return either $wMatch[i]$, $sMatch[i]$ or $bMatch[i,j]$. In the three cases the algorithm will locate all the occurrences of x in y. When shifting the pattern, it applies the maximum between the occurrence shift and the matching shift.

3 A brief history

In April 1974, Robert S. Boyer (Stanford Research Institute) and J. Strother Moore (Xerox Palo Alto Research Center) designed a string-matching algorithm with the following features: right-to-left comparisons, occurrence shift, weak matching shift and a fast loop [2]. At the same period and independently R. W. Gosper (Stanford University) discovered the right-to-left comparisons system and the occurrence shift. In December

1975, Ben Kuipers (Artificial Intelligence Laboratory, MIT) communicated to Boyer and Moore the idea of the strong matching shift. And at the same period Boyer and Moore introduced the best matching shift in $O(m \times \sigma)$ time and space complexities. In January 1976, Donald E. Knuth (Stanford University) showed that the strong matching shift is enough for the linearity of the algorithm when the pattern is not present in the text giving a bound of $7n$ character comparisons. He also gave a general bound of $O(n + rm)$ character comparisons where r is the number of occurrences of the pattern in the text. He finally introduced the Boyer-Moore automaton which conceptualizes an algorithm that remembers all the matched text characters among the m last scanned [14]. In 1979, Zvi Galil (Tel Aviv University) published a linear algorithm for finding all occurrences of the pattern in the text [8] using prefix memorization. In 1980, Wojciech Rytter (Warsaw University) gave the first published correct version of the preprocessing of the strong matching shift [17]. This same year Leo J. Guibas (Xerox Corporation, Palo Alto Research Center) and Andrew M. Odlyzko (Bell Telephone Laboratories) gave a proof of a $4n$ bound and conjectured that the right bound was $2n$ [9]. Still in 1980 R. Nigel Horspool (McGill University) designed a practical algorithm using only the occurrence shift based on the rightmost character of the window [12]. In 1986 Alberto Apostolico (Purdue University) and Raffaele Giancarlo (Salerno University) presented an algorithm that they proved performs $2n$ character comparisons in the worst case for finding all the occurrences of the pattern in the text using $O(m)$ extra space [1]. In 1987, Zhu Rui Feng and Tadao Takaoka (Ibaraki University) presented an algorithm using a two-dimensional occurrence shift [21]. In 1988, R. Schaback (Göttingen University) published a study on the expected sublinearity of the Boyer–Moore algorithm [18]. In 1990, Richard Cole (Courant Institute, New York University) gave a simple proof of a $4n$ bound and a tight bound of $3n$ character comparisons [4]. The same year Daniel Sunday (Johns Hopkins University) designed the Quick Search algorithm (using the occurrence shift with the text character immediately to the right of the window) [20]. In 1991, Andrew Hume (AT&T Bell Laboratories) and Daniel Sunday (Johns Hopkins University) published a study on practical string matching algorithms where they gave the Tuned Boyer–Moore algorithm which consists of a fast loop with three consecutive occurrence shifts [13]. In 1992, Maxime Crochemore (LITP, University Paris 7), Artur Czumaj (Warsaw University), Leszek Gąsieniec (Warsaw University), Stefan Jarominek (Warsaw University), Thierry Lecroq (LITP, University of Orleans), Wojciech Plandowski (Warsaw University) and Wojciech Rytter (Warsaw University) designed the Turbo–BM algorithm which has a bound of $2n$ character comparisons in the worst case when searching for all the occurrences of the pattern in the text with a constant extra-space [5] using last match memorization. In 1993, Christophe Hancart (LITP, University Paris 7) computed the best matching shift in $O(m)$ [11]. In 1996, Maxime Crochemore (IGM, University of Marne-la-Vallée) and Thierry Lecroq (LIR, University of Rouen) gave a new presentation of the Apostolico–Giancarlo algorithm and a tight bound of $1.5n$ character comparisons [6].

It is worth noting that the Boyer–Moore string-matching algorithm has been introduced to the wide public in the *PC Magazine* and *Dr. Dobbs Journal* by Costas Menico in 1989 [15] and Jeff Prosise in 1996 [16] respectively.

```
SUFFIXES(x, m)
 1  suf[m − 1] ← m
 2  g ← m − 1
 3  for i ← m − 2 downto 0
 4      do if i > g and suf[i + m − 1 − f] < i − g
 5            then suf[i] ← suf[i + m − 1 − f]
 6            else g ← min{g, i}
 7                 f ← i
 8                 while g ≥ 0 and x[g] = x[g + m − 1 − f]
 9                     do g ← g − 1
10                 suf[i] ← f − g
11  return suf
```

FIG. 9. Algorithm SUFFIXES.

4 Computing the strong matching shift

Since Knuth showed that the strong matching shift is sufficient to have a linear algorithm when looking for the first occurrence of the pattern ([14]), the strong matching shift is then the shift generally used when one implements the Boyer–Moore algorithm. The first correct computation of the strong matching is due to Rytter [17] but it is quite difficult to understand. We will give here a simpler version based on the computation of the longest suffixes of x ending at each position in x. The lengths of these suffixes greatly help the computation of the matching shift.

4.1 Computing the longest suffixes ending at each position in the pattern

Let us first present the computation of the longest suffixes of x ending at each position in x. It can be viewed as an application from right to left of the fundamental preprocessing (or Z algorithm) given by Gusfield [10]. For $0 \leq i \leq m-1$ we denote by $suf[i]$ the length of the longest suffix of x ending at position i in x. Let us denote by $lcsuf(u, v)$ the longest common suffix of two words u and v.

The computation of the table *suf* is done by the algorithm SUFFIXES presented in Figure 9. Figure 10 depicts the variables and the invariants of the main loop of algorithm SUFFIXES. The values of *suf* are computed for each position i in x in decreasing order. The algorithm uses two variables f and g which satisfy:

- $g = \min\{j - suf[j] \mid i < j < m-1\}$
- f is a position j such that $i < j < m-1$ and $j - suf[j] = g$

In order to prove the correctness of algorithm SUFFIXES we will first show an intermediate lemma.

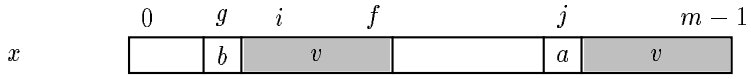

FIG. 10: Variables i, f, g of algorithm SUFFIXES. The main loop has invariants: $v = \mathit{lcsuf}(x, x[g+1\,..\,f])$ and $a \neq b$ ($a, b \in \Sigma$), $j = g + m - 1 - f$, and $i < f$. The picture corresponds to the case where $g < i$.

LEMMA 4.1
If $i > g$, we have

$$\mathit{suf}[i] = \begin{cases} \mathit{suf}[i+m-1-f] & \text{if } \mathit{suf}[i+m-1-f] < i-g \\ i-g+\ell & \text{otherwise} \end{cases}$$

where $\ell = |\mathit{lcsuf}(x[0\,..\,g], x[0\,..\,m-1-i+g])|$.

Proof: Let $v = x[g+1\,..\,f]$. This word is a suffix of x by definition of f and g. Let $k = \mathit{suf}[i+m-1-f]$. By definition of *suf*, the word $x[i+m-1-f-k\,..\,i+m-1-f]$ is a suffix of x but $x[i+m-f-k\,..\,i+m-1-f]$ is not a suffix of x.

In the first case ($i > g$ and $\mathit{suf}[i+m-1-f] < i-g$), the word $[x+m-f-k\,..\,i+m-1-f]$ occurs in v ending at position $i+m-1-f$. Thus it also occurs ending at position i in x which shows that $x[i+m-1-f-k\,..\,i+m-1-f]$ is the longest suffix of x ending at position i. Thus $\mathit{suf}[i] = k = \mathit{suf}[i+m-1-f]$.

In the second case, the word $x[g+1\,..\,i]$, which is a prefix of v, is a suffix of $x[i+m-1-f-k\,..\,i+m-1-f]$ and thus of x. It is easy to see that $\mathit{suf}[i] = i-g+\ell$. □

THEOREM 4.2
Algorithm SUFFIXES computes correctly the table *suf*.

Proof: The variables f and g satisfy the definition given before Lemma 4.1 before each execution of the main loop of the algorithm. Then for a given i such that $i > g$ the algorithm applies the relation given by Lemma 4.1 which gives a correct value. It remains to check that the computation is correct when $i \leq g$. In this situation the instructions from line 8 to line 9 compute $|\mathit{lcsuf}(x[0\,..\,i], x)| = |x[g+1\,..\,f]|$ which is by definition the correct value for $\mathit{suf}[i]$. Therefore algorithm computes correctly the table *suf*. □

We will now give the time complexity of algorithm SUFFIXES.

```
STRONG-MATCHING(x, m)
1  j ← 0
2  for i ← m − 1 downto −1
3      do if i = −1 or suf[i] = i + 1
4           then while j < m − 1 − i
5                  do sMatch[j] ← m − 1 − i
6                     j ← j + 1
7  for i ← 0 to m − 2
8      do sMatch[m − 1 − suf[i]] ← m − 1 − i
9  return sMatch
```

FIG. 11. Algorithm STRONG-MATCHING.

THEOREM 4.3
Algorithm SUFFIXES runs in time $O(m)$. Less than $2m$ character comparisons are performed.

Proof: The character comparisons are performed on line 8. Each comparison between two equal characters leads to decrementing the variable g that never increases. As g goes from $m-1$ to -1, it gives a maximum of m positive comparisons. Each negative comparison leads to move to the next step of the main loop of the algorithm. There are thus a maximum of $m-1$ such comparisons. It gives us overall $2m-1$ character comparisons.

This shows that the total time of all the runs of the loop from line 8 to line 9 is $O(m)$. The other instructions of the loop from line 3 to line 10 are executed in contant time. Thus the whole algorithm is in $O(m)$. □

4.2 Computing the strong matching shift

We are now able to give, in Fig.. 11, the algorithm STRONG-MATCHING which computes the table *sMatch* using the table *suf*.

The invariants of the second loop of algorithm STRONG-MATCHING are presented in Fig. 12.

We will now show that algorithm STRONG-MATCHING computes correctly table *sMatch.* We first begin by proving two intermediate lemmas.

LEMMA 4.4
For $0 \leq i < m$, if $suf[i] = i+1$ then, for $0 \leq j < m-1-i$, $sMatch[j] \leq m-1-i$.

Proof: The assumption $suf[i] = i+1$ is equivalent to the assumption that $x[0\,..\,i]$ is a suffix of x. Thus $m - suf[i] = m-1-i$ is a period of x. Let j be a position such that $0 \leq j < m-1-i$. Condition $Cs(j, m-1-i)$ is satisfied since $m-1-i > j$ and $x[0\,..\,m-(m-1-i)-1] = x[0\,..\,i]$ is a suffix of x. Condition $Cos(j, m-1-i)$ is also satisfied since $m-1-i > j$. Then by definition of *suf*, $sMatch[j] \leq m-1-i$. □

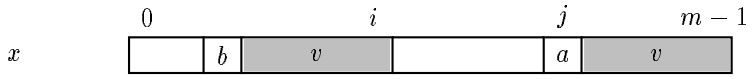

FIG. 12: Variable i of algorithm STRONG-MATCHING. Situation where $suf[i] < i+1$. The loop of lines 7-8 has the following invariants: $v = lcsuf(x, x[0\,..\,i])$ and $a \neq b$ ($a, b \in \Sigma$) and $suf[i] = |v|$. Thus $sMatch[j] \leq m-1-i$ with $j = m-1-suf[i]$.

LEMMA 4.5
For $0 \leq i \leq m-2$ we have $sMatch[m-1-suf[i]] \leq m-1-i$.

Proof: If $suf[i] < i+1$, condition $Cs(m-1-suf[i], m-1-i)$ is satisfied since we have both $m-1-i \leq m-1-suf[i]$ and $x[i-suf[i]+1\,..\,i] = x[m-1-suf[i]+1\,..\,m-1]$. Moreover condition $Cos(m-1-suf[i], m-1-i)$ is also satisfied since $x[i-suf[i]] \neq x[m-1-suf[i]]$ by definition of suf. Thus $sMatch[m-1-suf[i]] \leq m-1-i$.

If $suf[i] = i+1$, by lemma 4.4 we have for $j = m-1-suf[i] = m-i-2$, $sMatch[j] \leq m-1-i$. □

THEOREM 4.6
Algorithm STRONG-MATCHING computes correctly the table *sMatch*.

Proof: We have to show that for each j, $0 \leq j < m$, the final value s given to $sMatch[j]$ by algorithm STRONG-MATCHING is the minimum value which satisfies $Cs(j, s)$ and $Cos(j, s)$.

Let us assume first that s results from an assignment in the loop from lines 2 to 6. Then the first part of condition *Cs* is not satisfied. By lemma 4.4 we verify that s is the minimum value that satisfies the second part of condition $Cs(j, s)$. In this case, $s = m-1-i$ for a value i such that $suf[i] = i+1$ and $j < m-1-i$. This last inequality shows that condition $Cos(j, s)$ is also satisfied. Thus $s = sMatch[j]$.

Let us assume now that s results from an assignment in the loop from lines 7 to 8. Thus $j = m-1-suf[i]$ and $s = m-1-i$, and, by lemma 4.5, $sMatch[j] \leq s$. We also have $0 < s \leq i$, which shows that the second parts of conditions $Cs(j, s)$ and $Cos(j, s)$ cannot be satisfied. Since the values of $m-1-i$ are considered in decreasing order during the execution of the loop, s is the smallest value of $m-1-i$ for which $j = m-1-suf[i]$. Thus $s = sMatch[j]$. This ends the proof. □

THEOREM 4.7
Algorithm STRONG-MATCHING computes the table *sMatch* for a word of length m in

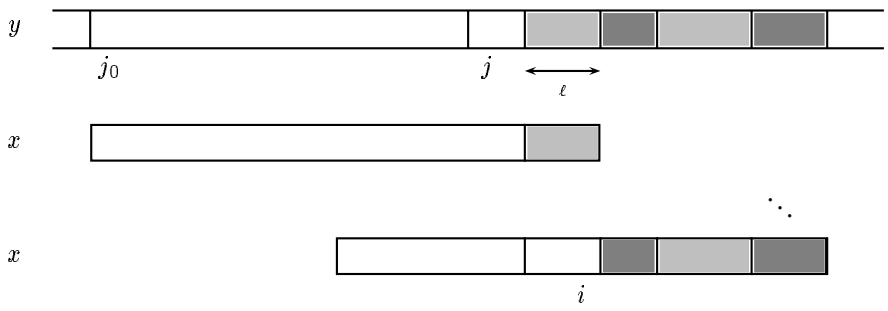

FIG. 13: A typical situation during the Apostolico-Giancarlo algorithm: *jump* or *shift* ? Dark gray areas correspond to factors that have been compared during the current attempt while light gray areas correspond to factor that have been jumped.

time $O(m)$ (including the computation of the table *suf*) and requires $O(m)$ extra space.

Proof: The extra space needed for the computation (excluding the word x and the table *sMatch*) is constituted by the table *suf* and some variables, thus $O(m)$.

The loop from line 2 to line 6 executes in $O(m)$ as each instruction executes in constant time for variables i and j which take $m+1$ values.

The loop from line 7 to line 8 executes also in $O(m)$ which gives the result. The computation of the table *suf* has the same complexity by Theorem 4.3. □

5 The Apostolico–Giancarlo algorithm

The main drawback of the Boyer–Moore algorithm is that after a shift it forgets completely what it has previously matched. Apostolico–Giancarlo algorithm remedies this. It remembers at the end of each attempt the length of the suffix of the pattern matched during this attempt. Matches so memorized are possibly used to avoid comparisons and compute shifts.

We are now going to see how the algorithm scans the characters. Assume that during an attempt where the pattern is aligned with the text characters $y[j_0..j_0+m-1]$, a suffix of length ℓ of the pattern has been found i.e. $x[m-\ell..m-1] = y[j_0+m-\ell..j_0+m-1]$ and $x[m-\ell-1] \neq y[j_0+m-\ell-1]$.

If during a later attempt where the pattern is aligned with the text characters $y[j..j+m-1]$ with $j_0 < j$, a match is found between characters $x[i+1..m-1]$ and $y[j_0+m-\ell..j+m-1]$ where $i = m-(j-j_0)-1$ (see Fig. 13).

Actually four different cases can arise: they are illustrated by Figures 14 to 17.

Case 1 :

skip$[i+j] >$ *suf*$[i]$ and $i+1 =$ *suf*$[i]$: then an occurrence of x is found at position j (see Fig. 14). A shift of length *sMatch*$[0]$ is performed and *skip*$[j+m]$ is set to

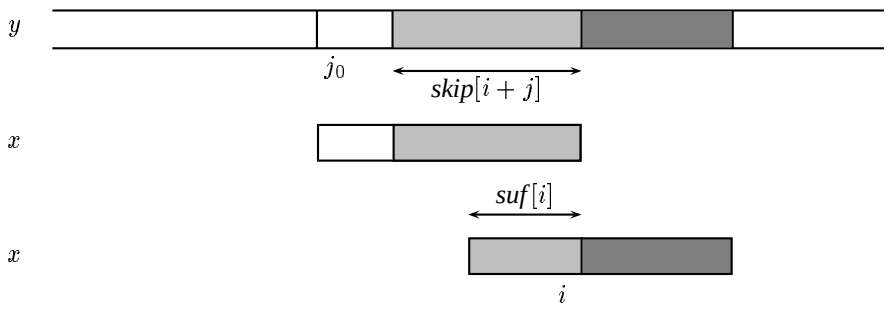

FIG. 14: Case 1, *skip*$[i+j] >$ *suf*$[i]$ and *suf*$[i] = i+1$ then an occurrence of x is found.

m.

Case 2 :
skip$[i+j] >$ *suf*$[i]$ and *suf*$[i] \leq i$: then a mismatch occurs between characters $x[i-$*suf*$[i]]$ and $y[j+i-$*suf*$[i]]$ (see Fig. 15). Thus a shift can be performed using *sMatch*$[i-$*suf*$[i]]$ and *occ*$[y[j+i-$*suf*$[i]]]$ and *skip*$[j+m]$ is set to $m-i-$*suf*$[i]-1$.

Case 3 :
skip$[i+j] <$ *suf*$[i]$: then a mismatch occurs between characters $x[i-$ *skip*$[i+j]]$ and $y[j+i-$ *skip*$[i+j]]$ (see Fig. 16). Thus a shift can be performed using *sMatch*$[i-$ *skip*$[i+j]]$ and *occ*$[y[j+i-$ *skip*$[i+j]]]$ and *skip*$[j+m]$ is set to $m-i-$ *skip*$[i+j]-1$.

Case 4 :
skip$[i+j] =$ *suf*$[i]$: then this is the only case where a "jump" has to be performed in order to resume the comparisons between characters $x[i-$*suf*$[i]]$ and $y[j+i-$*suf*$[i]]$ (see Fig. 17).

Following these four cases we are now able to formulate the Apostolico–Giancarlo algorithm (see Fig. 18).

6 The complexity of the Apostolico–Giancarlo algorithm

We are first going to show that comparing the same k characters twice causes a right shift of x of length greater than k. A text character can be compared again only if the previous comparisons it was involved in were mismatches.

LEMMA 6.1 ([6])
If an attempt performs k comparisons with text characters already previously compared. Then the shift following this attempt is of length at least $k+1$.

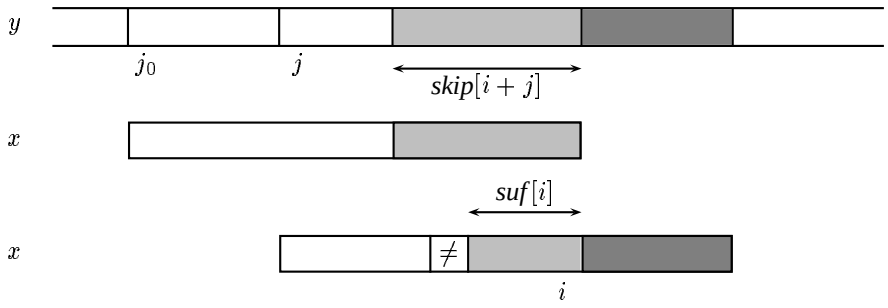

FIG. 15: Case 2, *skip*$[i+j] >$ *suf*$[i]$ and *suf*$[i] \leq i$ then a mismatch occurs between $x[i -$ *suf*$[i]]$ and $y[j + i -$ *suf*$[i]]$.

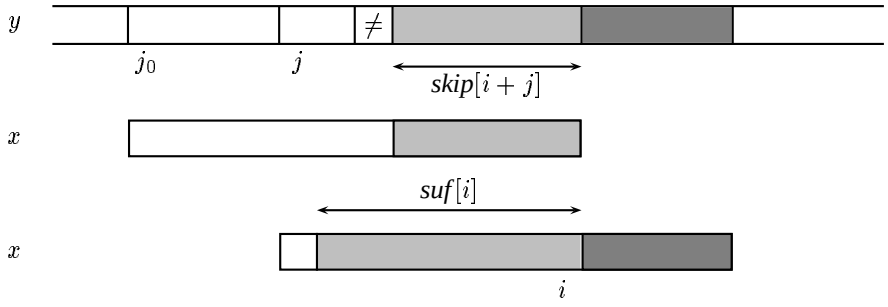

FIG. 16: Case 3, *skip*$[i+j] <$ *suf*$[i]$ then a mismatch occurs between $x[i-$*skip*$[i+j]]$ and $y[j + i -$ *skip*$[i + j]]$.

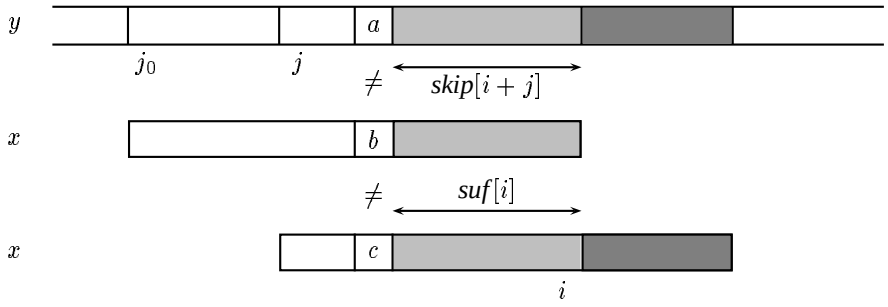

FIG. 17. Case 4, *skip*$[i + j] =$ *suf*$[i], a \neq b$ and $b \neq c$.

```
APOSTOLICO-GIANCARLO(x, m, y, n)
 1  j ← 0
 2  while j ≤ n − m
 3      do i ← m − 1
 4         while i ≥ 0
 5             do if skip[i + j] = 0
 6                   then if x[i] = y[i + j]
 7                           then i ← i − 1
 8                           else BREAK()
 9                elseif skip[i + j] > suf[i]
10                   then ▷ Cases 1 and 2
11                        i ← i − suf[i]
12                        BREAK()
13                elseif skip[i + j] < suf[i]
14                   then ▷ Case 3
15                        i ← i − skip[i + j]
16                        BREAK()
17                else     ▷ Case 4
18                        i ← i − suf[i]
19         skip[i + j] ← m − i − 1
20         if i < 0
21           then REPORT(j)
22                j ← j + sMatch[0]
23           else j ← j + max(sMatch[i], occ[y[i + j]] − m + i + 1)
```

FIG. 18. The Apostolico–Giancarlo algorithm revisited.

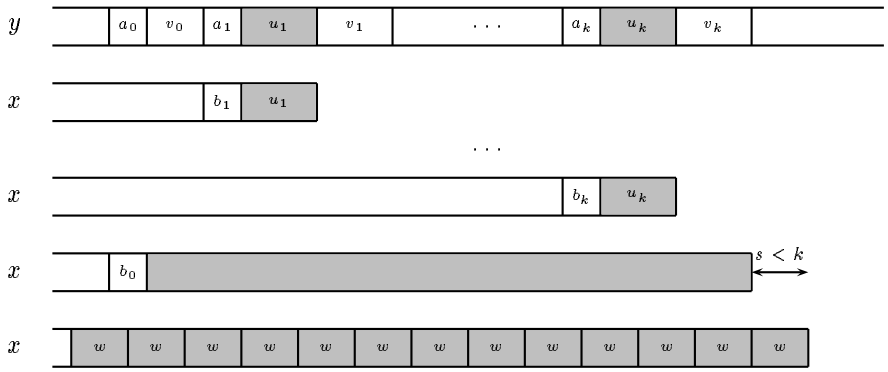

FIG. 19: An attempt performing k comparisons with text characters that have already been compared.

Sketch of the proof: Assume that A is an attempt that performs k comparisons with text characters that have already been compared then the recognized suffix of x in y is equal to $v_0 a_1 u_1 v_1 a_2 u_2 v_2 \cdots a_k u_k v_k$ (see Fig. 19) where:

- the a_i's for $1 \leq i \leq k$ are the text characters that have already been compared during k previous attempts,
- the u_i's are the recognized suffixes of x in y during these k previous attempts (thus they are jumped during attempt A, and the $a_i u_i$'s are not suffixes of x), $|u_i| \geq 0$,
- the v_i's have not been compared previously, $|v_i| \geq 0$.

Assume that the matching shift s following attempt A is shorter than k.
Then $v_0 a_1 u_1 v_1 a_2 u_2 v_2 \cdots a_k u_k v_k$ is a suffix of w^r with $|w| = s < k$.
Then two u_i's cannot be aligned with the same character within a factor w thus $|w| \geq k$ but $|w|$ cannot be equal to k because no $a_i u_i$ is a suffix of x thus $s = |w| > k$.
So the length of the matching shift following attempt A is greater than k. As the length of the actual shift is greater or equal to the length of the matching shift, the actual shift performed after attempt A is strictly longer than k. □

We are now going to give an upper bound on the number of comparisons performed with text characters already compared.

LEMMA 6.2 ([6])
The Apostolico–Giancarlo algorithm performs at most $\frac{n}{2}$ comparisons with text characters that have already been compared.

Proof: Let us divide all the attempts performed by the algorithm in several groups. Two attempts are in the same group if they perform a comparison on a common text character.
A group G of attempts that performs k comparisons with text characters that have already been compared contains at least $k+1$ attempts and implies k shifts of length at least 1 and one shift of length at least $k+1$ (by lemma 6.1). Thus it implies a sum of shifts of total length at least $2k+1$.
Let c_k for $0 \leq k \leq m-1$ be the number of groups of attempts performing k comparisons with text characters that have already been compared.
Then the total number of groups of attempts is $\sum_{k=0}^{m-1} c_k$.
The sum of all the shift lengths must be less than n (including the shift after the last attempt):

$$\sum_{k=1}^{m-1} (2k+1)c_k + c_0 < n$$

which implies:

$$\sum_{k=1}^{m-1} k c_k \leq \frac{n}{2}$$

□

We are now able to give the maximal number of text character comparisons performed by the Apostolico–Giancarlo algorithm.

THEOREM 6.3 ([6])
The Apostolico–Giancarlo algorithm performs at most $1.5n$ text characters comparisons and this bound is tight.

Proof: Each text character can be compared positively at most once and the algorithm can perform at most $\frac{n}{2}$ comparisons with text characters that have already been compared (by lemma 6.2).
This bound is tight: for $x = a^{m-1}ba^mb$ and $y = (a^{m-1}ba^mb)^n$ the algorithm performs $\frac{3m-1}{2m-1}n$ text characters comparisons. □

7 Conclusion

We gave a new method to compute the strong matching shift of the Boyer–Moore algorithm. This method is simpler than the previous published methods. It computes and uses a table storing the length of the longest suffix of the pattern ending at each position in the pattern. This table is extensively used in the new version of the Apostolico–Giancarlo algorithm, which performs a maximum number of comparisons that is half the maximum of the Boyer–Moore algorithm in the worst case.

References

[1] A. Apostolico and R. Giancarlo. The Boyer–Moore–Galil string searching strategies revisited. *SIAM J. Comput.*, 15(1):98–105, 1986.

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

[3] C. Charras and T. Lecroq. Exact string matching algorithms. Available at URL: `http://www-igm.univ-mlv.fr/~lecroq/string/`.

[4] R. Cole. Tight bounds on the complexity of the Boyer–Moore string matching algorithm. *SIAM J. Comput.*, 23(5):1075–1091, 1994.

[5] 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.

[6] M. Crochemore and T. Lecroq. Tight bounds on the complexity of the Apostolico–Giancarlo algorithm. *Inform. Process. Lett.*, 63(4):195–203, 1997.

[7] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[8] Z. Galil. On improving the worst case running time of the Boyer–Moore string searching algorithm. *Comm. ACM*, 22(9):505–508, 1979.

[9] 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):672–682, 1980.

[10] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[11] C. Hancart. *Analyse exacte et en moyenne d'algorithmes de recherche d'un motif dans un texte*. Thèse de doctorat, Université Paris 7, 1993.

[12] R. N. Horspool. Practical fast searching in strings. *Software–Practice and Experience*, 10(6):501–506, 1980.

[13] A. Hume and D. M. Sunday. Fast string searching. *Software–Practice and Experience*, 21(11):1221–1248, 1991.

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

[15] C. Menico. Faster string searches. *Dr. Dobbs J.*, pages 74–75, 1989.

[16] J. Prosise. High-performance string searching: How a smart algorithm can make a big difference. *PC Magazine*, October 10, 1996.

[17] W. Rytter. A correct preprocessing algorithm for Boyer–Moore string searching. *SIAM J. Comput.*, 9(3):509–512, 1980.

[18] R. Schaback. On the expected sublinearity of the Boyer–Moore algorithm. *SIAM J. Comput.*, 17(4):648–658, 1988.

[19] G. A. Stephen. *String searching algorithms*. World Scientific Press, 1994.

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

[21] R. F. Zhu and T. Takaoka. On improving the average case of the Boyer–Moore string matching algorithm. *J. Inform. Process.*, 10(3):173–177, 1987.

Received