Tight bounds on the complexity of the Apostolico-Giancarlo algorithm

Maxime Crochemore, Thierry Lecroq

▶ To cite this version:

Maxime Crochemore, Thierry Lecroq. Tight bounds on the complexity of the Apostolico-Giancarlo algorithm. South American Workshop on String Processing (WSP 1996), 1996, France. pp.64-74. hal-00619989

HAL Id: hal-00619989
https://hal.science/hal-00619989

Submitted on 19 Mar 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Tight bounds on the complexity of the Apostolico-Giancarlo algorithm

Maxime Crochemore [*] Thierry Lecroq [†]

Abstract

The Apostolico-Giancarlo string-matching algorithm is analyzed precisely. We give a tight upper bound of $\frac{3}{2}n$ text characters comparisons when searching for a pattern in a text of length n. We exhibit a family of patterns and texts reaching this bound. We also provide a slightly improved version of the algorithm.

1 Introduction

The string-matching problem consists in finding all occurrences of a pattern in a text. It is a basic problem that occurs in information retrieval, bibliographic search and molecular biology, for example. It has been extensively studied and numerous techniques and algorithms have been designed to solve this problem (see [5] and [10]).

Basically, a string-matching algorithm applies the *sliding window mechanism* as follows. It first initializes the search by aligning the left ends of the pattern and the text. Then, it checks (or scans) if the pattern occurs in the text at the chosen position and eventually shifts the pattern to the right. Finally, it repeats the same operation until the right end of the pattern goes beyond the right end of the text.

Scan operations are composed of series of symbol comparisons made in a specific order. Boyer and Moore [2] have derived, from choosing the reverse order, (according to the direction of shifts) one of the most practically efficient algorithm. Scanning the characters of the pattern from right to left enables the algorithm to "jump" over some portions of the text and therefore to save symbol comparisons as well as running time. This assumes however that both text and pattern reside in main memory.

The main drawback of the algorithm is that after a shift it forgets all the matches encountered so far. As a consequence, the complexity analysis of the

*Institut Gaspard Monge, Université de Marne-la-Vallée, 2 rue de la Butte Verte, F-93160 NOISY-LE-GRAND, `mac@univ-mlv.fr`

†Laboratoire d'Informatique de Rouen, Université de Rouen, Faculté des Sciences et des Techniques, F-76128 MONT-SAINT-AIGNAN CEDEX, `lecroq@dir.univ-rouen.fr`

Memorization of	Extra space	Extra preprocessing time	Bound on the number of comparisons
prefix [6]	constant	none	$14n$
last match [4]	constant	none	$2n$
all matches [1]	$O(m)$	$O(m)$	$1.5n$

Figure 1: Features of variants of the Boyer-Moore algorithm.

Boyer-Moore algorithm is rather difficult to achieve. The worst-case time analysis is given by Cole [3] who proves a tight upper bound of $3n - n/m$ comparisons when looking for the first occurrence of a non-periodic pattern (where n is the length of the text and m is the length of the pattern).

Boyer-Moore algorithm has been primarily designed for discovering the first occurrence of the pattern in the text. When adapted in a straightforward way for searching for all the occurrences of the pattern, its worst-case running time is quadratic. The exact complexity in this situation is $O(n + rm)$, where r is the number of times the pattern occurs in the text (see [6]).

To remedy to the oblivious feature of the Boyer-Moore algorithm, several solutions have been proposed. Galil [6] introduces what can be called a *prefix memorization* in order to save comparisons after the localization of an occurrence of the pattern. This leads to a linear-time algorithm for the problem. Crochemore *et al.* [4] show that *last-match memorization*, which embodies the idea in [6], yields an algorithm that makes no more than $2n$ comparisons in the worst case. The algorithm called Turbo-BM uses the same preprocessing as Boyer-Moore algorithm, as well as Galil's algorithm, and requires only constant extra space (to store the last match or the prefix, respectively) at searching time.

Apostolico and Giancarlo [1] designed another variant of Boyer-Moore algorithm that remembers all previous matches between the pattern and the text. They proved an upper bound of $2n - m + 1$ text characters comparisons for their algorithm. Contrary to Galil's algorithm and Turbo-BM, this algorithm needs an additional preprocessing and requires $O(m)$ extra space to store all previous matches inside the current window on the text.

In this paper, we provide a worst-case analysis of the Apostolico-Giancarlo algorithm that proves an upper bound of $\frac{3}{2}n$ text characters comparisons at search phase. In some sense, this is a "reward" for the extra work necessary to implement this improvement on Boyer-Moore algorithm. Figure 1 summarizes the features of the three variants of Boyer-Moore algorithm. We also show that this bound is tight, by exhibiting a family of patterns and texts reaching this bound. Moreover, we reformulate the algorithm and design a slightly modified version of the algorithm that captures more information about previous matches. Although the improvement is hard to measure, the worst-case analysis still holds for the modified algorithm.

The paper is organized as follows. Section 2 recalls the elements introduced

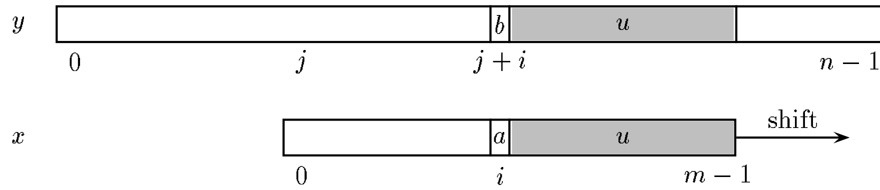

Figure 2: If $a \neq b$, algorithm BM applies a precomputed shift function.

by Apostolico and Giancarlo, and gives a presentation of their algorithm. In Section 3 we analyze its worst-case complexity in term of symbol comparisons. Finally, Section 4 presents the new version of the algorithm.

Throughout this paper the pattern is a word of length m denoted by x ($x = x[0 \ldots m-1]$). The text is denoted by y and has length n ($y = y[0 \ldots n-1]$). Both x and y are built over a finite alphabet Σ of size σ.

2 The Apostolico-Giancarlo algorithm

The Apostolico-Giancarlo algorithm (algorithm AG) is built upon the Boyer-Moore algorithm (algorithm BM) that we recall first.

For checking whether an occurrence of the pattern occurs at position j in the text, BM algorithm scans the characters from right to left beginning with the rightmost character of the pattern. 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 *occurrence shift* and the *match shift* functions. Figure 2 displays the situation, an attempt at position j, with a mismatch between character $x[i]$ of the pattern and character $y[j+i]$ of the text The suffix $u = x[i+1 \ldots m-1]$ of the pattern matches the segment $y[j+i+1 \ldots j+m-1]$ of the text.

The length of the shift computed by BM algorithm is the maximum value of the occurrence shift function and the match shift function with arguments defined by the situation. The occurrence shift consists in aligning the text character $y[j+i]$ with its rightmost occurrence in $x[0 \ldots m-2]$. The match shift consists in aligning the factor $u = y[j+i+1 \ldots j+m-1] = x[i+1 \ldots m-1]$ with its rightmost occurrence in x that is preceded by a character different from $x[i]$. If there exists no such factor, it consists in aligning the longest suffix of $y[j+i+1 \ldots j+m-1]$ with a matching prefix of x.

More formally, the two shift functions are defined as follows, with the notation of [7]. The occurrence shift is represented by a table d of size σ defined, for all $a \in \Sigma$, by:

$$d[a] = \min\{m\} \cup \{i \mid 0 < i < m \text{ and } x[m-1-i] = a\}.$$

```
BM (x, m, y, n)
  j ← 0
  while j ≤ n − m do
    i ← m − 1
    while i ≥ 0 and x[i] = y[j + i] do
      i ← i − 1
    if i < 0 then
      OUTPUT(match at position j)
      j ← j + dd'[0]
    else
      j ← j + max(dd'[i], d[y[j + i]] − m + i + 1)
```

Figure 3: The Boyer-Moore algorithm.

The match shift is stored in a table dd' of size $m+1$ defined, for all i, $0 \leq i < m$, by:

$$\begin{aligned} dd'[i] = \quad & \min\{s > 0 \mid (s > i \text{ or } x[i-s] \neq x[i]) \text{ and} \\ & (\text{for all } k, i < k < m, s > k \text{ or } x[k-s] = x[k])\}. \end{aligned}$$

Tables d et dd' can be precomputed in time $O(m + \sigma)$ before the search phase (see, for example, [5]). A presentation of algorithm BM based on these tables is depicted in Figure 3.

As noticed above, the drawback of algorithm BM is that after a shift it forgets completely what has been matched previously. Algorithm AG copes with this problem by remembering segments of the text already matched with suffixes of the pattern. At the end of each attempt, it keeps track of the length of the suffix matched during this attempt in a table called *skip*. It is exploited in conjunction with a table that stores the similar information related to the pattern itself. We call this table *suf* defined, for all i, $0 \leq i < m$, by:

$$\mathit{suf}[i] = \max\{|u| \mid u \text{ longest suffix of } x \text{ ending at } i \text{ in } x\}.$$

The table *suf* can be computed during the preprocessing phase required by the shift tables. This takes $O(m)$ extra time (see [1] or [5]).

We are now ready to explain the central idea of algorithm AG. Let us consider the situation displayed in Figure 4. The algorithm is scanning the text inside the window placed at position j_1, and the suffix $x[i+1 \ldots m-1]$ of the pattern matches the text. Let $k = \mathit{skip}[j_1 + i]$. If $k = 0$, the algorithm continues as algorithm BM does. Otherwise, three cases are considered.

Case 1: $k > \mathit{suf}[i]$ and $i+1 = \mathit{suf}[i]$. An occurrence of the pattern is detected.

Case 2: $k > \mathit{suf}[i]$ and $\mathit{suf}[i] \leq i$. The pattern does not occur at position j_1, and the algorithm executes a shift of length $dd[i+1]$ (see below).

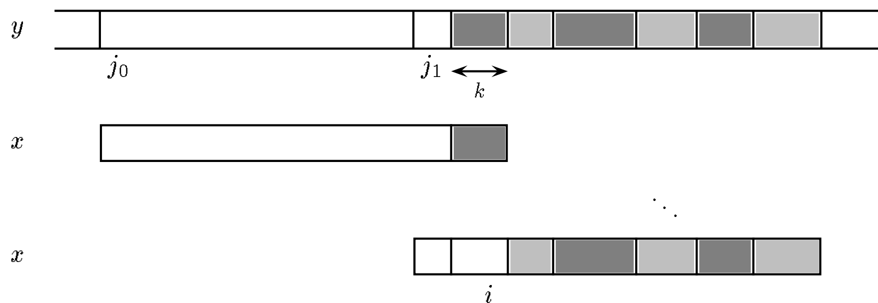

Figure 4: A typical situation during the Apostolico-Giancarlo algorithm: *jump* or *shift* ? Light gray areas correspond to factors of the text scanned during the current attempt, while dark gray areas correspond to factor scanned previously.

Case 3: $k \leq suf[i]$ then a "jump" is performed over the previous match of length k and the comparisons resume between characters $y[j_1 + i - k]$ and $x[i - k]$.

A presentation of algorithm AG is shown in Figure 5. Notice that the algorithm uses a weak version of the occurrence shift function represented by the table dd and defined as follows. For all i, $0 \leq i < m$:

$$dd[i] = \min\{s > 0 \mid \text{for all } k, i < k < m, s > k \text{ or } x[k - s] = x[k]\}.$$

The dd table does not take into account the mismatch character, and corresponds to the original table used by algorithm BM. The dd' table has been introduced by Knuth in [7].

The values of the table *skip*, initialized to 0, are computed during the searching procedure. The size of this table can be reduced to $O(m)$ since, during each attempt, no more than the m last values are necessary.

The proof of the next statement is in [1]. It does not include the preprocessing time that is linear in the length of the pattern.

Theorem 1 (Apostolico-Giancarlo) *Algorithm AG finds all the occurrences of a pattern of length m in a text of length n in time $O(n)$, and makes no more than $2n - m + 1$ character comparisons.*

3 Complexity analysis

In this section, we prove the tight bound of $1.5n$ on the number of symbol comparisons made by the Apostolico-Giancarlo algorithm. This refines the result of Theorem 1.

```
AG (y, x, n, m)
  j ← 0
  while j ≤ n − m do
    i ← m − 1
    while i ≥ 0 do
      if skip[j + i] = 0 then
        if x[i] = y[j + i] then i ← i − 1
        else break
      else if skip[j + i] > suf[i] then
        if suf[i] = i + 1 then i ← −1   /* Case 1 */
        break                           /* Case 2 */
      else  /* skip[j + i] ≤ suf[i] */
        i ← skip[j + i]                 /* Case 3 */
    skip[j + i] ← m − i − 1
    if i < 0 then
      OUTPUT(match at position j)
      j ← j + dd[0]
    else j ← j + max(dd[i + 1], d[y[j + i]] − m + i + 1)
```

Figure 5: A presentation of the Apostolico-Giancarlo algorithm.

y	=	a	a	a	a	**a**	a	b	a	b	a	b
x	=	a	a	b	a	b	a	b				
		x	=	a	a	b	a	b	a	b		
				x	=	a	a	b	a	b	a	b

Figure 6: Algorithm AG can make several mismatches on a symbol of the text (3 on the symbol in bold). Underlined characters are compared positively once.

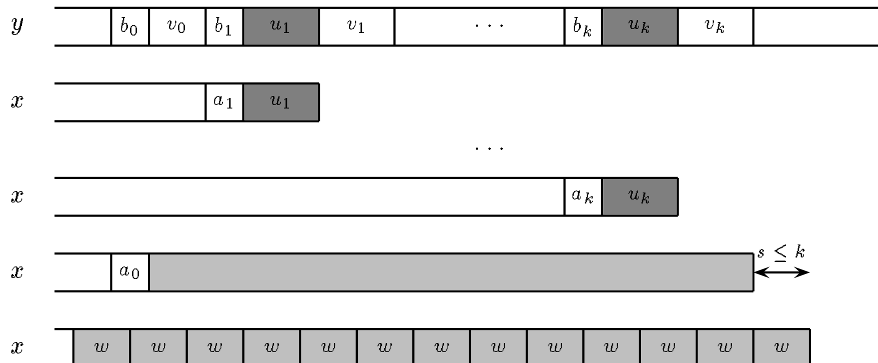

Figure 7: Impossible: k positive re-comparisons on $a_1, \ldots, a_k$, but a subsequent shift of length $s \leq k$.

During the execution of the algorithm, if a symbol of the text is found to match a symbol of the pattern, then never again the symbol of the text is compared. The total number of such positive comparisons may be n, for instance, if a^m is searched for in a^{em} ($n = em$). Therefore we focus our attention on mismatch symbols of the text. These symbols can be compared several times as shown by the example of Figure 6. The strategy to prove the result is to amortize the number of re-comparisons on lengths of shifts. This is the aim of Lemma 1, which consequence, given in Lemma 2 is that the total number of all re-comparisons is bounded by $n/2$. The bound on the number of symbol comparisons in algorithm AG follows immediately (Theorem 2).

Lemma 1 *If, during an attempt of algorithm AG, k positive symbol comparisons are made on text characters previously compared then the subsequent shift realized by the algorithm has length at least $k + 1$.*

Sketch of the proof Assume that A is an attempt during which k comparisons are made on text characters that have already been compared. Figure 7 displays the situation. Characters that are compared again at the present step were mismatch characters during previous attempts. Since a mismatch in the algorithm implies an immediate shift, the characters correspond to segment of the text in the form $b_\ell u_\ell$ ($1 \leq \ell \leq k$). All these segment are disjoint (or possibly adjacent) inside the text. Thus, the scanned part of the text is in the form $b_0 v_0 b_1 u_1 v_1 b_2 u_2 v_2 \ldots b_k u_k v_k$ (see Figure 7) where:

- b_0 is the mismatch characters, and $v_0 b_1 u_1 v_1 b_2 u_2 v_2 \ldots b_k u_k v_k$ is a suffix of the pattern,
- the b_ℓ's ($1 \leq \ell \leq k$) are the k text characters that are re-compared (positively) during the attempt,

- the u_ℓ's ($1 \leq \ell \leq k$) are the corresponding suffixes of x matched during k previous attempts (they are jumped over during the present step A),
- the v_ℓ's ($1 \leq \ell \leq k$) are the segments of the text that are effectively scanned during the attempt ($|v_\ell| \geq 0$) or they are previously matched suffixes that are jumped over during the present attempt.

Note that, by definition of the above elements, the words $b_\ell u_\ell$'s are not suffixes of the pattern, and that $|u_\ell| \geq 0$ ($1 \leq \ell \leq k$).

Assume, *ab absurdo*, that the match shift of length s realized at the end of attempt A is no longer than k. Let w be the suffix of length s of x. By definition of the function dd, $v_0 b_1 u_1 v_1 b_2 u_2 v_2 \ldots b_k u_k v_k w$ is a suffix of x and has period $s = |w|$.

For two different indices ℓ' and ℓ'', $u_{\ell'}$ and $u_{\ell''}$ fall at the same position within the factor w because there are only $k-1$ possible positions. Therefore, we have $b_{\ell'} u_{\ell'} = b_{\ell''} u_{\ell''}$, which implies that the shifts at the corresponding attempts are of the same length. Whence, they have produced the same situations implying $u_{\ell'+1} = u_{\ell''+1}$. We get a contradiction with the fact that $b_k u_k$ is different from other $b_\ell u_\ell$.

So, the length of the match shift at the attempt A is greater than k. Since the length of the actual shift is at least the length of the match shift, algorithm AG performs a shift of length at least $k+1$, as announced.

Lemma 2 *The Apostolico-Giancarlo algorithm makes at most $n/2$ comparisons on text characters previously compared.*

Proof Let us group all the attempts performed by the algorithm: two attempts are in the same group if they perform a comparison on a common text character. A group g of attempts in which are performed k_g positive comparisons on text characters previously compared contains at least $k+1$ attempts (as in Figure 7). Among the corresponding shifts, k_g of them are of length at least 1, and one shift has length at least $k_g + 1$, by Lemma 1. Thus, the total length of shifts involved in the group is at least $2k_g + 1$.

If the symbol b_0 of Figure 7 is not re-compared at that step, the total number of re-comparisons is k_g, which is no more than half the total length of shifts. If the symbol b_0 of Figure 7 is re-compared at that step, there are $k_g + 1$ shifts of length at least 1 (instead of k_g). Then, the total number of re-comparisons is $k_g + 1$, which is again no more than half the total length of shifts, $2k_g + 2$.

Finally, since the sum of all shifts is no more the n, the total number of re-comparisons is no more than $n/2$, which ends the proof.

Theorem 2 *The Apostolico-Giancarlo algorithm performs no more $\frac{3}{2}n$ character comparisons. There are texts of length n and associated patterns of length $2m+1$ for which the algorithm makes $\frac{3m+1}{2m+1}n - m$ character comparisons.*

Proof The result is a direct consequence of Lemma 2. The bound is tight: for $x = a^{m-1}ba^m b$ and $y = (a^{m-1}ba^m b)^e$ ($m, e > 0$, $n = (2m+1)e$), the algorithm makes exactly $2m + 1 + (3m+1)e$, that is, $\frac{3m+1}{2m+1}n - m$ character comparisons.

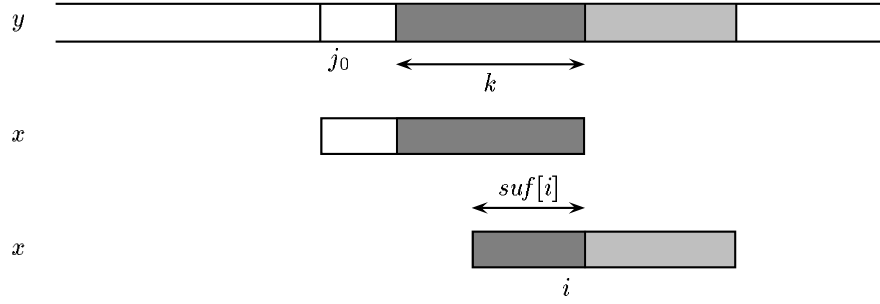

Figure 8: Case 1', $k > suf[i]$ and $suf[i] = i + 1$, an occurrence of x is found.

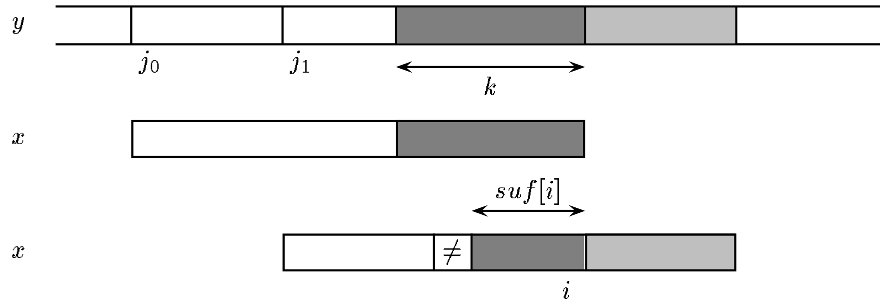

Figure 9: Case 2', $k > suf[i]$ and $suf[i] \leq i$, a mismatch occurs between $y[j_1 + i - suf[i]]$ and $x[i - suf[i]]$.

4 Improving the Apostolico-Giancarlo algorithm

In this section we refine the analysis that leads to algorithm AG. Let us look more closely at the general situation depicted in Figure 4 ($k = skip[j_1 + i]$). Actually four different cases can be considered:

Case 1': (identical to Case 1) $k > suf[i]$ and $suf[i] = i + 1$. An occurrence of x is found at position j (see Figure 8).

Case 2': $k > suf[i]$ and $suf[i] \leq i$. A mismatch occurs between characters $y[j_1 + i - suf[i]]$ and $x[i - suf[i]]$ (see Figure 9). Thus, as already mentioned in [8], the length of the shift is computed as in algorithm BM (*i.e.*, using $dd'[i - suf[i] + 1]$ and $d[y[j_1 + i - suf[i]]]$).

Case 3': $k < suf[i]$. A mismatch occurs between characters $y[j_1 + i - k]$ and $x[i - k]$ (see Figure 10). Thus a shift can be performed using $dd'[i - k + 1]$

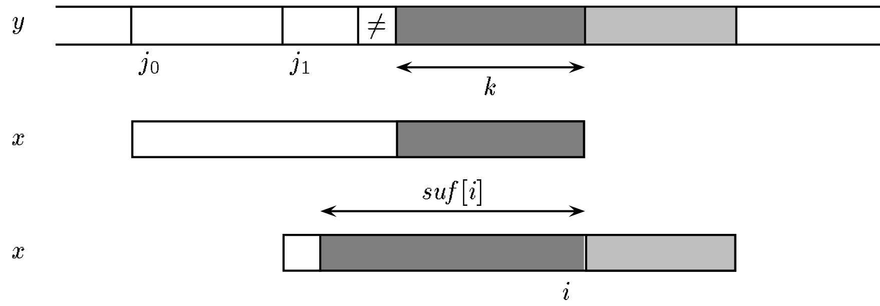

Figure 10: Case 3', $k < suf[i]$ a mismatch occurs between $y[j_1 + i - k]$ and $x[i - k]$.

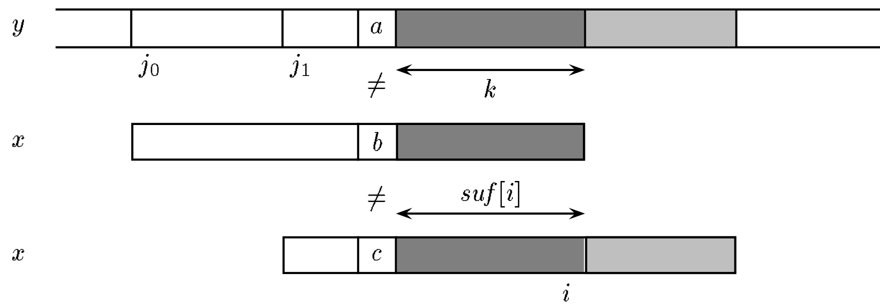

Figure 11: Case 4', $k = suf[i], a \neq b$ and $b \neq c$.

```
AG' (y, x, n, m)
  j ← 0
  while j ≤ n − m do
    i ← m − 1
    while i ≥ 0 do
      if skip'[j + i] = 0 then
        if x[i] = y[j + i] then i ← i − 1
        else break
      else if skip'[j + i] > suf[i] then
        i ← i − suf[i]                    /* Case 1' and 2' */
        break
      else if skip'[j + i] < suf[i] then
        i ← i − skip'[j + i]              /* Case 3' */
        break
      else  /* skip'[j + i] < suf[i] */
        i ← i − skip'[j + i]              /* Case 4' */
    skip'[j + i] ← m − i − 1
    if i < 0 then
      OUTPUT(match at position j)
      j ← j + dd'[0]
    else j ← j + max(dd'[i + 1], d[y[j + i]] − m + i + 1)
```

Figure 12: The Apostolico-Giancarlo algorithm revisited.

and $d[y[j_1 + i - k]]$.

Case 4': $k = suf[i]$. This is the only case where a "jump" has to be done in order to resume the comparisons between characters $y[j_1 + i - k]$ and $x[i - k]$ (see Figure 11).

Following the four cases we obtain the version AG' of the Apostolico-Giancarlo algorithm presented in Figure 12. Note that it uses the match shift function dd', which provides longer shift on the average. The worst-case analysis of Section 3 is still valid for algorithm AG'.

The table *skip'* computed by algorithm AG' satisfies the property: if $skip'[i] > 0$, $skip'[i]$ is the length of the longest suffix of x ending at position i in the text. The table *skip* of algorithm AG does not share the same property.

5 Conclusion

We have presented an analysis on the number of symbol comparisons of the Apostolico-Giancarlo algorithm proving a tight $1.5n$ bound. Following it we have designed a new version of the algorithm that fully used all the information considered by the algorithm to take local decisions. The complete analysis of

the new algorithm remains to be done. In particular, it is worth evaluating how behaves the new algorithm with respect to total number of comparisons. This number is known to be bounded by $11n$ for algorithm AG (see [1]).

References

[1] A. Apostolico, R. Giancarlo, The Boyer-Moore-Galil string-searching strategy revisited, *SIAM J. Comput.* **15**(1) (1984) 98–105.

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

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

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

[5] M. Crochemore, W. Rytter, *Text algorithms*, Oxford University Press, New York, Oxford, 1994.

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

[7] D.E. Knuth, J.H. Morris Jr, V.R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6**(2) (1977) 323–350.

[8] T. Lecroq, Experimental results on string matching algorithms, *Software–Practice & Experience* **27**(5) (1995) 727–765.

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

[10] G.A. Stephen, *String Searching Algorithms*, World Scientific Press, 1994.