Intern. J. Computer Math.,
Vol. 80, No. 7, July 2003, pp. 825–834

Taylor & Francis
Taylor & Francis Group

A NEW STRING MATCHING ALGORITHM

MUSTAQ AHMED[a,*], M. KAYKOBAD[a,†] and REZAUL ALAM CHOWDHURY[b,‡]

[a]*Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka-1000, Bangladesh;* [b]*Department of Computer Science, University of Texas at Austin, Texas, USA*

(Received 26 July 2002)

In this paper a new exact string-matching algorithm with sub-linear average case complexity has been presented. Unlike other sub-linear string-matching algorithms it never performs more than n text character comparisons while working on a text of length n. It requires only $O(m+\sigma)$ extra pre-processing time and space, where m is the length of the pattern and σ is the size of the alphabet.

Keywords: String matching; Complexity; Boyer–Moore algorithm

C. R. Categories: F.2.2

1 INTRODUCTION

The string matching problem that appears in many applications like word processing, information retrieval, bibliographic search, molecular biology, etc., consists in finding the first or all the occurrences of a pattern in a text, where the pattern and the text are strings over the same alphabet.

Many algorithms for solving this problem exist in literature [10, 20]. The naive (brute force) algorithm [7] locates all occurrences in time $O(nm)$, where m is the length of the pattern and n is the length of the text. Hashing provides a simple method [15] that runs in linear time in most practical situations. A minimal DFA recognizing the language $\Sigma^* x$, where $x[0{:}m-1]$ is the pattern over the alphabet Σ of size σ, can be used to locate all the occurrences of x in the text $y[0{:}n-1]$ by examining each text character exactly once [7]. But the construction of the DFA requires $O(\sigma+m)$ time and $O(\sigma m)$ space. Simon [14, 18, 19] showed that this DFA can be constructed in $O(m)$ time and space by introducing a delay bounded by $O(\sigma)$ per text character and his algorithm never performs more than $2n-1$ text character comparisons.

However, the first linear time string-matching algorithm was discovered by Morris and Pratt [17], and was later improved by Knuth *et al.* [16]. The search behaves like a recognition

* E-mail: mustaqahmed@yahoo.com
† Corresponding author. E-mail: kaykobad@cse.buet.ac.bd
‡ E-mail: shaikat@cs.utexas.edu

ISSN 0020-7160 print; ISSN 1029-0265 online
DOI: 10.1080/0020716031000087113

process by automation but the pre-processing time and space is reduced to $O(m)$. A text character is compared to a character of the pattern no more than $\log_\Phi(m+1)$ times (Φ is the golden ratio $(1+\sqrt{5})/2$), and there is no more than $2n-m$ text character comparisons on the whole. The Colussi [4–6, 12], Galil–Giancarlo [4, 12] and Apostolico–Crochemore [1, 13] algorithms are refinements of this algorithm and respectively require $1.5n$, $1.33n$ and $1.5n$ text character comparisons in the worst case.

The Knuth–Morris–Pratt algorithm scans the characters of the pattern from left to right. But choosing the reverse direction and introducing two heuristic functions, Boyer and Moore [3] have derived one of the most efficient string-matching algorithms in practice. The algorithm is sub-linear on the average but requires a quadratic ($O(nm)$) running time in the worst case. It also requires $O(\sigma+m)$ pre-processing time and space. However, the main drawback of this algorithm is that after a shift it forgets all the matches encountered so far. To remedy the situation the *prefix memorization* technique introduced by Galil [11] leads to an algorithm requiring only constant extra space and $14n$ text character comparisons in the worst case. Crochemore *et al.* [8] showed that *last-match memorization* yields an algorithm that never makes more than $2n$ comparisons. Apostolico and Giancarlo [2] designed yet another variant of the Boyer–Moore algorithm that remembers all the previous suffix matches of the pattern with the text at the cost of $O(m)$ extra pre-processing time and space. An upper bound of $1.5n$ text character comparisons has been proved for their algorithm [9].

In this paper, we present a new variant of the Boyer–Moore algorithm that remembers all the previous matches between the pattern and the text. The algorithm is sub-linear on the average and requires no more than n text character comparisons in the worst case. However, it requires $O(\sigma+m)$ extra pre-processing time and space.

2 THE NEW ALGORITHM

The string matching problem consists in finding the first occurrence or all occurrences of a pattern $x[0{:}m-1]$ in a text $y[0{:}n-1]$, or deciding that none exists. Both the pattern and the text are defined over the same alphabet Σ of size σ and are assumed to have length m and n respectively. Like other Boyer–Moore type algorithms, the algorithm presented in this paper solves this problem by sliding the pattern along the text from left to right. For each positioning that arises, it attempts to match the pattern against the text from right to left. If no mismatch occurs then an occurrence of the pattern is found. In case of a mismatch or a complete match it uses some pre-computed functions to shift the pattern to the right by such an amount that no potential match is missed. The algorithm always remembers the segments of the text within the current window of the pattern that were ever matched with segments of the pattern and, uses its pre-computed shift functions to obtain a shift that will align all segments remaining within the window after the shift to new matching positions within the pattern. During the right to left matching of the characters of the text and the pattern these matched segments are jumped over by the algorithm.

An example may reveal the idea behind the algorithm. Consider the situation depicted in Figure 1 where a mismatch occurs between the character $x[i]=b$ of the pattern and the character $y[j+i]=a$ of the text during an attempted match at position j of the text. Then $x[i+1, m-1]=y[j+i+1, j+m-1]=u_0$ and $x[i]\neq y[j+i]$. Note that it is possible to have $i=m-1$ and thus u_0 to be empty. u_1, u_2 and u_3 are the segments of the text within the current window of the pattern that were matched with the characters of the pattern in previous attempts. Now the pre-computed shift functions find the rightmost occurrence (if any) of the segment au_0 in x so that aligning this segment with the segment

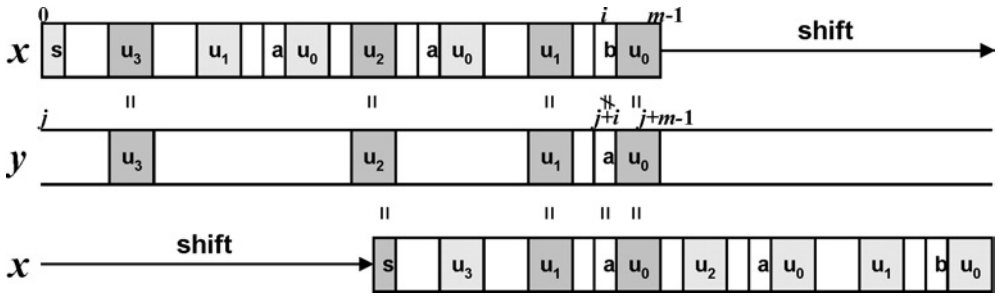

FIGURE 1 An example.

$y[j+i, m-1]$ of the text will also align each of the segments u_1, u_2 and u_3 of the text with corresponding matching position within x provided the entire segment or a portion of it still remains within the window of the pattern. If no such occurrence is found then the shift consists in aligning the longest suffix of $y[j+i+1, j+m-1]$ with a matching prefix of x. In Figure 1 a shift is found which causes u_1 to match completely and u_2 to match partially and leaves u_3 out of the window.

The pseudo code of this new string-matching algorithm is given in Figure 2 and explained in subsequent paragraphs.

The algorithm first performs some pre-processing. The *PRE_BC* function calculates the original Boyer–Moore *bad character shift* function, and stores it in a table *bc* of size σ. The *bc* function has the following definition, and can be calculated in $O(\sigma)$ time [3]. For each $a \in \Sigma$,

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

PRE_SUF calculates a table *suf* of size m. For $0 \le i < m$, *suf* is defined as follows:

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

It can be calculated in $O(m)$ time [2].

PRE_PRF computes a table *prf* of size $m+1$. *prf* is defined as follows:

$$prf[i] = \begin{cases} m - |u|, & \text{where } 0 \le i < m \text{ and } u \text{ is the longest suffix of } x[i, m-1] \\ & \quad \text{which is also a prefix of } x[0, m-1] \\ m, & \text{where } i = m \end{cases}$$

Figure 3 shows how to calculate *prf* from *suf* in $O(m)$ time.

PRE_LSP sets the variable *lsp* to $m - |u|$, where u is the longest proper prefix of x which is also a suffix of x. It is straightforward to calculate *lsp* from *suf* in $O(m)$ time.

PRE_BC_GS fills in two tables *bc_gs_ptr* and *bc_gs_val* using the pre-calculated *suf* table. *bc_gs_ptr* table contains σ entries, and each entry contains two values, namely *ptr* and *nt*. For each $a \in \Sigma$, $bc\_gs\_ptr[a] \cdot ptr$ contains an index (≥ 0) into the table *bc_gs_val* if there is any entry corresponding to a in that table, otherwise it contains -1. If $bc\_gs\_ptr[a] \cdot ptr \ne -1$, then the entries from index $bc\_gs\_ptr[a] \cdot ptr$ to $bc\_gs\_ptr[a] \cdot ptr + bc\_gs\_ptr[a] \cdot nt - 1$ in table *bc_gs_val* correspond to a. Each entry of the *bc_gs_val* table contains two entries: *loc* and *shift*. The entries (if any) corresponding

```
procedure MATCHER(x, m, y, n)
  integer i, j, k, l, lsp, m, n, suf [m], prf [m + 1]
  integer clink[m], slink[m], skip[m], bc[σ]
  record (integer nt, ptr) bc_gs_ptr[σ]
  record (integer loc, shift) bc_gs_val[m]
  //Preprocessing//
  call PRE_BC(x, m, bc)
  call PRE_SUF(x, m, suf)
  call PRE_PRF(m, suf, prf)
  call PRE_LSP(m, suf, lsp)
  call PRE_BC_GS(x, m, suf, bc_gs_ptr, bc_gs_val)
  call PRE_CLINK(x, m, clink)
  call PRE_SLINK(x, m, suf, prf, slink)
  //Searching//
  j ← 0, l ← 0, skip[0] ← m
  while j ≤ n − m do
    i ← m − 1, k ← skip[l]
    while i ≥ 0 do
      if k = 0 then
        l ← (l + m − 1) mod m, i ← i − skip[l]
        if i < 0 then break endif
        l ← (l + m − 1) mod m, k ← min(skip[l], i + 1)
      endif
      if x[i] = y[ j + i] then i ← i − 1 else break endif
    repeat
    if i < 0 then
      call OUTPUT(j)
      skip[l] ← m − lsp, l ← (l + 1) mod m, skip[l] ← lsp
    else
      skip[l] ← k, l ← (l + 1) mod m
      skip[l] ← m − i, l ← (l + 1) modm
      if i = m − 1 then
        skip[l] ← bc[y [ j + i]] − m + i + 1
      else
        skip[l] ← GET_BC_GS(y [ j + i], i + 1, bc_gs_ptr, bc_gs_val)
        if skip[l] = −1 then skip[l] ← prf [i + 1] endif
      endif
      call VALIDATE_SHIFT(m, clink, slink, l, skip)
    endif
    j ← j + skip[l]
  repeat
end MATCHER
```

FIGURE 2 Pseudo code of the new matcher.

```
procedure PRE_PRF(m, suf, prf)
  integer i, j, k, m, suf [m], prf [m + 1]
    k ← −1, prf [0] ← 0
    for i ← 0 to m − 1 by 1 do
      if suf[i] > i then
        for j ← k to i − 1 by 1 do
          prf[m − j − 1] ← m − k − 1
        repeat
        k ← i
      endif
    repeat
  end PRE_PRF
```

FIGURE 3 Pseudo code of *PRE_PRF*.

to a in that table are sorted on the increasing value of loc. For each value of i from $bc\_gs\_ptr[a] \cdot ptr$ to $bc\_gs\_ptr[a] \cdot ptr + bc\_gs\_ptr[a] \cdot nt - 1$, $bc\_gs\_val[i] \cdot shift$ has the following definition:

$$bc\_gs\_val[i] \cdot shift = \left\{ m - j - 1 \;\middle|\; \begin{array}{r} j \text{ is the rightmost occurrence of } ax[bc\_gs\_val[i]. \\ loc, m-1] \text{ in } x \\ \text{where } x[bc\_gs\_val[i] \cdot loc - 1] \neq a \end{array} \right\}$$

A size of $m - 2$ is sufficient for the table bc_gs_val. This is because, for each of the $m - 2$ values of $j (0 < j < m - 1)$, there can be at most one l ($l > 0$ and $j - l + 1 \geq 0$) such that $x[j - l + 1, j] = x[m - l, m - 1]$ and $x[j - l] \neq x[m - l - 1]$. If not, suppose, there exist two such values l_1 and l_2. Then,

(i) For l_1, $x[j - l_1 + 1, j] = x[m - l_1, m - 1]$ and $x[j - l_1] \neq x[m - l_1 - 1]$
(ii) For l_2, $x[j - l_2 + 1, j] = x[m - l_2, m - 1]$ and $x[j - l_2] \neq x[m - l_2 - 1]$

Now, if $l_1 < l_2$, then from (i), $x[j - l_1] \neq x[m - l_1 - 1]$, but from (ii), $x[j - l_1] = x[m - l_1 - 1]$ which is a contradiction. Again, if $l_1 > l_2$, then from (ii), $x[j - l_2] \neq x[m - l_2 - 1]$, but from (i), $x[j - l_2] = x[m - l_2 - 1]$ which is also a contradiction. Hence, $l_1 = l_2$.

Figure 4 shows how to construct the tables bc_gs_ptr and bc_gs_val from the pre-calculated table suf in $O(m + \sigma)$ time and space. The entries that will fill bc_gs_val are already available in the suf table, but in a different form and sorted on j (see the preceding paragraphs). Performing bucket sort twice – first on the values of $m - l$ and then on the values of $x[j - l]$, will put them in intended order.

The next pre-processing function *PRE_CLINK* fills in a table $clink$ of size $m \cdot clink[0]$ contains -1 and for $0 < i < m$, $clink[i]$ contains the index of the rightmost occurrence of $x[i]$ in $x[0, i - 1]$ if one exists, otherwise it contains -1. It is straightforward to build the $clink$ table directly from x in $O(m + \sigma)$ time.

PRE_SLINK is the last pre-processing function called. It uses the pre-calculated tables suf and prf to build the table $slink$ of size m. For $0 < i < m - 1$, $slink[i]$ has the following definition:

(i) $slink[i] = j$, if $x[0, i]$ is a suffix of x and j ($< i$) is the largest index such that $x[0, j]$ is also a suffix of x.

```
procedure PRE_BC_GS(x, m, suf, bc_gs_ptr, bc_gs_val)
integer f, i, k, l, m, link[m], loc[m], shift[2m], next[2m]
  char a, c[m]
  record (integer nt, ptr) bc_gs_ptr[σ]
  record (integer loc, shift) bc_gs_val[m]
  f ← 0, link[0: m − 1] ← −1
  for i ← 0 to m − 2 by 1 do
    k ← suf[i]
    if k > 0 and i ≥ k then
      c[f] ← x[i − k], shift[f] ← m − i − 1
      next[f] ← link[m − k], link[m − k] ← f, f ← f + 1
    endif
  repeat
for all a ∈ Σ do bc_gs_ptr[a] ← (0, −1) repeat
f ← m
for i ← m − 1 down to 1 by −1 do
    l ← link[i]
    while l ≠ −1 do
      a ← c[l], k ← bc_gs_ptr[a].ptr
      if k = −1 or loc[k − m] > i then
        loc[f − m] ← i, shift[f] ← shift[l], next[f] ← k,
         bc_gs_ptr[a] ← ( f, bc_gs_ptr[a].nt + 1 ), f ← f + 1
      endif
      l ← next[l]
    repeat
  repeat
  f ← 0
  for all a ∈ Σ do
    l ← bc_gs_ptr[a].ptr
    if l ≠ −1 then
      bc_gs_ptr[a].ptr ← f
      while l ≠ −1 do
        bc_gs_val[f] ← ( loc[l − m], shift[l] )
        f ← f + 1, l ← next[l]
      repeat
    endif
  repeat
end PRE_BC_GS
```

FIGURE 4 Pseudo code of *PRE_BC_GS*.

(ii) $slink[i] = k$, if $x[0, i]$ is not a suffix of x and there exists an l $(i \geq l > 0)$ such that $x[i - l + 1, i] = x[m - l, m - 1]$ and $x[i - l] \neq x[m - l - 1]$, and k $(< i)$ is the largest index such that $x[\max(0, k - l), k] = x[i - k + \max(0, k - l), i]$.

(iii) $slink[i] = -1$, otherwise.

Figure 5 shows how to implement *PRE_SLINK* in $O(m + \sigma)$ time and space. The implementation is somewhat similar to that of *PRE_BC_GS*.

```
procedure PRE_SLINK( x,, m, suf, prf, slink )
  integer m, suf[m], prf[m + 1], slink[m]
  integer f, i, k, l, link[m], len[m], owner[m]
  integer pre_loc[m], loc[2m], next[2m], cptr[σ]
  char a, c[m], x[m]
  l ← −1
  for i ← 0 to m − 1 by 1 do
    if suf[i] > i then slink[i] ← l, l ← i
    else slink[i] ← m − prf[m − suf[i]] − 1 endif
  repeat
  f ← 0, link[0:m − 1] ← −1
  for i ← 0 to m − 2 by 1 do
    k ← suf[i]
    if k > 0 and i ≥ k then
      c[f] ← x[i − k], loc[f] ← i, next[f] ← link[k]
      link[k] ← f, f ← f + 1
    endif
  repeat
    for all a ∈ ∑ do cptr[a] ← −1 repeat
    f ← m
     for i ← m − 1 down to 1 by −1 do
      l ← link[i]
      while l ≠ −1 do
        a ← c[l], loc[f] ← loc[l], len[f − m] ← I
        next[f] ← cptr[a]
        cptr[a] ← f, f ← f + 1, l ← next[l]
      repeat
    repeat
    owner[0:m − 1] ← −1
    for all a ∈ ∑ do
      l ← cptr[a]
      while l ≠ −1 do
        k ← len[l − m]
        if owner[k] = a then slink[loc[l]] ← pre_loc[k] endif
        pre_loc[k] ← loc[l], owner[k] ← a, l ← next[l]
    repeat
  repeat
end PRE_SLINK
```

FIGURE 5 Pseudo code of *PRE_SLINK*.

After pre-processing, the *MATCHER* routine enters the searching phase. It uses a circular table (*skip* table) of size m to remember the previously matched segments. Before entering the outer *while* loop, j (indicating the index into the text y to which the leftmost character of the pattern x is aligned) is initialized to 0, l (indicating the current index into the *skip* table) to 0 and $skip[l]$ to m. Before entering the inner while loop, i (the current index in the pattern x) is initialized to $m - 1$, that is comparison starts from the rightmost character of x. Let, $l_0 = l$ and for $0 < p < m$, $l_p = (l_{p-1} + m - 1) \bmod m$ At this moment, $skip[l_0]$ contains the length of the segment of unmatched characters of the text starting from index

$j+i$ (right to left), If $skip[l_0] < m$, then $skip[l_1]$ contains the length of the segment of matched characters (starting from index $j+i-skip[l_0]$ of the text) in the previous attempt. In general, if $skip[l_0]+skip[l_1]+\cdots+skip[l_{p-1}] < m$, then $skip[l_p]$ contains the length of the next segment of matched/unmatched (matched if p is odd, otherwise unmatched) characters of the text starting from index $j+i-(skip[l_0]+skip[l_1]+\cdots+skip[l_{p-1}])$. Note that, it may happen that $skip[l_0]+skip[l_1]+\cdots+skip[l_{p-1}] < m < skip[l_0]+skip[l_1]+\cdots+skip[l_p]$. In that case, the last segment length is taken to be $\min(m-skip[l_0]-skip[l_1]-\cdots-skip[l_{p-1}], skip[l_p])$. The inner while loop continues to match $x[i]$ to $y[j+i]$ for decreasing i until a mismatch occurs or the value of i falls below 0. In this process it jumps over the previously matched segments of the text and updates i and l accordingly.

If the inner *while* loop is exited with $i<0$, then a complete match is found and the algorithm updates (using *lsp*) the *skip* list and l in such a way that in the next attempt the longest proper suffix of $y[j, j+m-1]$ will be aligned with a matching prefix of x. On the other hand, if the algorithm exits the *while* loop with $i \geq 0$, then it first updates the *skip* list and l with the length of the matched segment during the current attempt. In this condition two cases may occur:

Case 1 ($i = M-1$) In this case, the *bc* table is used to find the rightmost occurrence of $y[j+i]$ in x and accordingly the *skip* table is updated to realize the proper shift in the next attempt.

Case 2 ($i < M-1$) In this case, a function *GET_BC_GS* is called to find the appropriate shift based on the rightmost occurrence of $y[j+i, j+m-1]$ in x. *GET_BC_GS* uses the *bc_gs_ptr* table with $y[j+i]$ to access the appropriate segment in the *bc_gs_val* table on which it performs a binary search in order to find the *loc* value $i+1$ and returns the value of the corresponding *shift* field. If it fails to find such a *loc* value, it returns -1. $skip[l]$ is updated with the return value of *GET_BC_GS*. If *GET_BC_GS* fails, then $skip[l]$ is updated with $prf[i+1]$ in order to align the longest suffix of $y[j+i+1, j+m-1]$ with a matching prefix of x.

After considering these two cases, *VALIDATE_SHIFT* (Fig. 6) is called which uses the *clink* and *slink* tables to check whether the current shift value aligns all the previously matched segments of the text that are still within the window of the pattern to new matching positions. If not, it finds the rightmost occurrence (even as a prefix) of $y[i, m-1]$ in x that leads to a valid shift and updates $skip[l]$ accordingly.

Last of all, the algorithm updates j with $j+skip[l]$ for the next attempt.

3 COMPLEXITY

This algorithm remembers each text segment it has ever matched with a segment of the pattern and always finds a shift value that never aligns them to mismatching positions. So, it compares each text character with a pattern character at most once, that is, it never makes more than n text-pattern comparisons.

The pre-processing stage has $O(m+\sigma)$ space and time complexity. This linear pre-processing leads to a delay of $O(\log_2 m)$ in the *GET_BC_GS* function. This delay can be reduced to $O(1)$ if direct access tables are used, in which case the pre-processing complexity will increase.

```
procedure VALIDATE_SHIFT( m, clink, slink, l, skip )
  integer i, j, k, l, m, p, loc[m], len[m], next[m], clink[m], slink[m], skip[m]
  if skip[l] ≥ m then return endif
  k ← 0, j ← l
  for i ← 0 to 1 by 1 do
    j ← (j + m − 1) mod m, k ← k + skip[j]
    if skip[l] + k ≥ m then return endif
  repeat
  p ← 0
  while skip[l] + k < m do
    j ← (j + m − 1) mod m, loc[p] ← m − k − 1
    len[p] ← skip[j]
    if skip[j] = 1 then next[p] ← clink[m − k − 1]
    else next[p] ← slink[m − k − 1] endif
    p ← p + 1, k ← k + skip[j]
    if skip[l] + k ≥ m then break endif
    j ← (j + m − 1) mod m, k ← k + skip[j]
  repeat
  loc[p] ← −1
  while skip[l] < m do
    i ← 0
    while loc[i] ≥ skip[l] do
      while next[i] > loc[i] − skip[l] do
        if len[i] = 1 then next[i] ← clink[next[i]]
              else next[i] ← slink[next[i]] endif
      repeat
      if next[i] < loc[i] − skip[l] then break endif
      i ← i + 1
    repeat
    if loc[i] < skip[l] then break endif
    if skip[(l + m − 1) mod m] = 1 then
      skip[l] ← m − 1 − clink[m − skip[l] − 1]
    else skip[l] ← m − 1 − slink[m − skip[l] − 1] endif
  repeat
end VALIDATE_SHIFT
```

FIGURE 6 Pseudo code of *VALIDATE_SHIFT*.

In the worst case, a call to the *VALIDATE_SHIFT* function may introduce an $O(m^2)$ delay, but in that case it will make an m character shift. Hence, for a text of length n, this function may introduce a total of $O(m^2 \times (n/m))$ or $O(mn)$ delay in the worst case.

4 CONCLUSION

In this paper, we have presented a new string-matching algorithm that makes the maximal use of all the information available to it. Like other Boyer–Moore type algorithms it is sub-linear on the average, but unlike those sub-linear algorithms it compares each text character to a pattern character at most once. Its pre-processing time and space is linear with respect to

pattern length and alphabet size. To the best of our knowledge it is the first string-searching algorithm with these properties.

References

[1] Apostolico, A. and Crochemore, M. (1991). Optimal canonization of all substrings of a string. *Inform. Comput.*, **95**(1), 76–95.

[2] Apostolico, A. and Giancarlo, R. (1986).The Boyer–Moore–Galil stringsearching strategy revisited. *SIAM J. Comput.*, **15**(1), 98–105.

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

[4] Breslauer, D. (1992). Efficient string algorithmics. *PhD thesis, Report CU-024-92*, Computer Science Department, Columbia University, New York, NY.

[5] Colussi, L. (1991). Correctness and efficiency of the pattern matching algorithms. *Inform. Comput.*, **95**(2), 225–251.

[6] Colussi, L., Galil, Z. and Giancarlo, R. (1990). On the exact complexity of string matching. *Proceedings of the 31st Annual Symposium on Foundations of Computer Science.* IEEE Computer Society Press, St. Louis, MO, pp. 135–144.

[7] Cormen, T. H., Leiserson, C. E. and Rivest, R. L.(1990). *Introduction to Algorithms*, MIT Press.

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

[9] Crochemore, M. and Leroq, T. (1997). Tight bounds on the complexity of the Apostolico–Giancarlo algorithm. *Information Processing Letters*, **63**(4), 195–203.

[10] Crochemore, M. and Rytter, W. (1994). *Text algorithms.* Oxford University Press, New York, Oxford.

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

[12] Galil, Z. and Giancarlo, R. (1992). On the exact complexity of string matching: Upper bounds. *SIAM J. Comput.*, **21**(3), 407–437.

[13] Hancart, C. (1993). Analyse exacte et en moyenne d'algorithmes de recherche d'un motifdans un texte, Thése de doctrat de l'Université de Paris 7, France.

[14] Hancart, C. (1993). On Simon's string matching algorithm. *Information Processing Letters*, **47**(2), 95–99.

[15] Karp, R. M. and Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, **31**(2), 249–260.

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

[17] Morris, J. H. Jr. and Pratt, V. R. (1970). A linear pattern-matching algorithm. *Technical Report 40*, University of California, Berkeley.

[18] Simon, I. (1993). String matching algorithms and automata. In: Baeza-Yates and Ziviani (Eds.), *First American Workshop on String Processing*, Universidade Federal de Minas Gerais, pp. 151–157.

[19] Simon, I. (1993). String matching algorithms and automata. In: Baeza-Yates and Ziviani (Eds.), *First American Workshop on String Processing*, Universidade Federal de Minas Gerais, pp. 151–157.

[20] Stephen, G. A. (1994). *String Searching Algorithms*, World Scientific Press.