1985

The Boyer-Moore-Galil String Searching Strategies Revisited

Alberto Apostolicao

Raffaele Giancarlo

Report Number:

85-539

THE BOYER-MOORE-GALIL STRING SEARCHING STRATEGIES REVISITED

Alberto Apostolico

Raffaele Giancarlo
The University of Salerno

CSD-TR-539
October 1985

THE BOYER-MOORE-GALIL STRING SEARCHING STRATEGIES REVISITED*

ALBERTO APOSTOLICO
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907 - U.S.A.

and

RAFFAELE GIANCARLO
Dipartimento di Informatica ed Applicazioni
The University of Salerno
I84100 Salerno - ITALY

June 1984

ABSTRACT

Based on the Boyer-Moore-Galil approach, a new algorithm is proposed which requires a number of character comparisons bounded by 2n, regardless of the number of occurrences of the pattern in the textstring. Preprocessing is only slightly more involved and still requires a time linear in the pattern size.

Keywords and Phrases:
String Searching, Pattern Matching, Shift Functions, Text Editing, Analysis of Algorithms.

1. Introduction

The string searching problem is to find all occurrences of a given *pattern* y in a given *text* x, both y and x being strings over a finite alphabet.

Letting $|x|=n$ and $|y|=m$, brute force procedures that involve $\Omega(nm)$ comparisons in the worst case can be quickly developed. However, as the copious literature [1-8] devoted to this subject over the past decade shows, the bound can be lowered to $O(n)$, provided some preprocessing of the pattern is allowed. As pointed out by Boyer and Moore [2], the time spent in the preprocessing plays generally a secondary role in the overall design perspective. However, it is fortunate that all

* This work was supported in part by the Italian Ministry of Education. Additional support was provided by the Italian National Council for Research and by N.A.T.O. under research grant no. 039.82. An extended abstract related to this paper has been presented at the 20th Annual Allerton Conference on Communication, Control, and Computing - Monticello, Illinois, October 6-8, 1982.

preprocessing strategies set up so far perform in time $O(m)$.

As is well known, one of the first string searching algorithms was proposed in [2]. Unlike the Knuth-Morris-Pratt algorithm [6], it compares y with x starting from the right end of y. The performance of this algorithm is quite good on the average case, where it performs in $O(n/m)$. On the other hand, it displays a worst case running time $\Omega(n^2)$.

Improving over the Boyer-Moore algorithm (hereafter, '*BM*' for short) Knuth, Morris and Pratt [6] also set up a modified version of it that performs at most $6n$ character comparisons, if the pattern does not appear in the text. More recently, Guibas and Odlyzko [5] narrowed that bound to $4n$ and conjectured it is $2n$. Zvi Galil [3] presented a new version of the modified *BM* algorithm and, by using the Guibas and Odlyzko result, showed a $14n$ character comparison worst case running time for his algorithm. This version is obtainable by the former one in a straightforward manner, even though it is not straightforward to prove its correctness.

As pointed out in [6], the analysis of the *BM* procedure is not simple. This is due to the fact that, when the *BM* algorithm shifts the pattern to the right, it does not retain any information about characters already matched. Based on this observation, Knuth, Morris and Pratt [6] suggested that the algorithm be made less oblivious by arranging the various situations that could arise in the course of the pattern matching process into a suitable table of "states". Problem is that the number of "states" in such a generalization of the *BM* strategy can be quite large (the obvious upper bound is 2^m, but it is not known how tight a bound this is). Thus the work involved in preparing that table is prohibitive in practice. There is room to suspect that a good portion of the table is unneeded in general. Galil's algorithm can be regarded in fact as a nonoblivious version of the *BM* strategy which only exploits two "states".

We present here still another upgrade of the *BM* that keeps track of which substrings of the pattern matched which substrings of the text during previous alignments, and exploits such recordings later in the matching process. If we allow for at most one recording per shift, then the number of such states is obviously bounded by $n-m+1$. The resulting algorithm works in linear time and displays three interesting features:

(1) It performs at most $2n-m+1$ character comparisons.

(2) The proof of linearity is very simple.

(3) *dd* heuristics (in the sense of [6]) can be used instead of *dd'*, not affecting (1-2).

The first feature conveys in our view the most interesting result of this paper: indeed it is seen to follow from the even stronger finding that no character of the text needs to be accessed more than twice. The inspection of text characters is the main (and obviously unavoidable) means by which information is acquired during any pattern matching process, so that the number of character comparisons performed is customarily considered especially significant. We shall show, however, that even taking into account the other comparisons (with the exception of those hidden in the control structure) yields the palatable bound of $11n$.

This paper is organized as follows. In section 2 we review briefly the salient features of the *BM* and some of its derivations. Section 3 is devoted to the exposition of our method, under the assumption that the information conveyed by preprocessing is already available. This latter problem is addressed in section 4.

2. The Boyer-Moore Approach to Pattern Matching

We will assume that the input x (y) is stored into the array $text[1:n]$ ($pattern[1:m]$).

The obvious way to locate all occurrences of y in x is by repeated aligning and checking from left to right. One innovative feature of the *BM* strategy is in that, for each alignment of the two strings, character comparisons are performed from right to left, starting at the right end of the pattern. As is well known, this contributes a significant speed-up in cases of mismatch (cfr. [6]), even though it leads to a quadratic worst case behavior. A compact presentation of the *BM* algorithm is given in

[3]. We report it below for the convenience of the reader.

```
Procedure BM    * $i$ ($j$) points to the current character              *
                * of the pattern (text); $s$[character,$i$] is the auxiliary  *
                * 'shift' function.                                      *

j:=m;
do while $j \le n$
      begin
            do $i$ := $m$ to 0 by -1 until $pattern[i] \ne text[j-m+i]$
            if $i$=0 then begin output (match at $j-m+1$) end
                  else $j$ := $j+s[text[j-m+i],i]$
      end
```

Tables such as s are usually referred to as *shift* functions. In [3,6] s is formally defined as follows:

s[character,i] = max {s.match[i], s.occ[character,i]}

where:

s.match[i] = min {t / $t \ge 1$ *and* ($t \ge i$ *or* $pattern[i-t] \ne pattern[i]$) *and* (($t \ge k$ *or* $pattern[k-t] = pattern[k]$) *for* $i < k \le m$ }

(this is called dd' in [6])

and

s.occ[character,i] = min {$t-m+i$ / $t = m$ *or* ($0 \le t < m$ *and* $pattern[m-t]$ = character)

The s.match portion ensures that (1) when moved to the right the pattern will match all previously matched characters, and (2) the character of the text that causes the mismatch will be aligned with a different character of the pattern.

The s.occ heuristics causes $text[j-m+i]$ (i.e. the mismatching character) to be aligned with the closest matching character of the pattern.

The shift function s', originally introduced in [2], neglects the (2) heuristics. Instead of s.match[i], we have there (cfr. the dd function in [6]):

s'.match[(i] = min {t / $t \ge 1$ *and* ($t \ge k$ *or* $pattern[k] = pattern[k-t]$) for $i < k \le m$ }

and, correspondingly:

s'.[character,i] = max {s'.match[i], s.occ[character,i]}

Both s and s' can be computed in $O(m)$ steps. The reader is referred to [6,7] for the details of such constructions.

It is convenient to extend s (s') to deal with the case $i=0$, as follows:

s[character,0] = s'[character,0] = min $\{t \,/\, t \geq 1$ *and* $pattern[k] = pattern[k+t]$, for
$i \leq k \leq m-t$

This helps resuming efficiently the pattern matching process following the detection of an occurrence of the pattern.

One more improvement is derived from the observation that if the pattern is *periodic* (i.e. $y=u^k u'$, with $k>1$ and u' a prefix of u), consecutive overlapping occurrences can be detected at once. Indeed, let u be the shortest string such that $y=u^k u'$ and let $k>1$. Let also t denote the length of u and take $l_0=m-t+1$. By combining the above observations, Z. Galil set up the following modified procedure BM' [3]:

```
Procedore $BM'$
j := $m$; $l$ := 0;
do while $j \leq n$
        begin
                do $i$ := $m$ to $l$ by -1 until $pattern[i] \neq text[j-m+i]$
                if $i = l$ then
                        begin
                                output (match at $j-m+1$); $l$ := $l_0$; $t$ := 0
                        end
                        else $l$ := 0
                $j$ := $j + s[text[j-m+i], i]$
        end
```

The BM' takes linear time even in the worst case. For a periodic pattern in the form $u^k u'$, the shift following a complete match must lead the prefix $u^{(k-1)}u'$ of the pattern to be aligned with the position of the text previously matched against the suffix of the pattern of the same form. This corresponds to singling out and exploiting exactly one of the many possible 'states' described in [6] (all other configurations could be thought of at this point as funneled into a single 'superstate'.

3. The Algorithm

It is convenient to give first an informal outline of our approach. To start with, consider the situation of Fig. 1 below which depicts one possible 'instantaneous description' of the pattern matching process: the pattern has undergone, say, s shifts, and $m-i$ successful character matches have been performed.

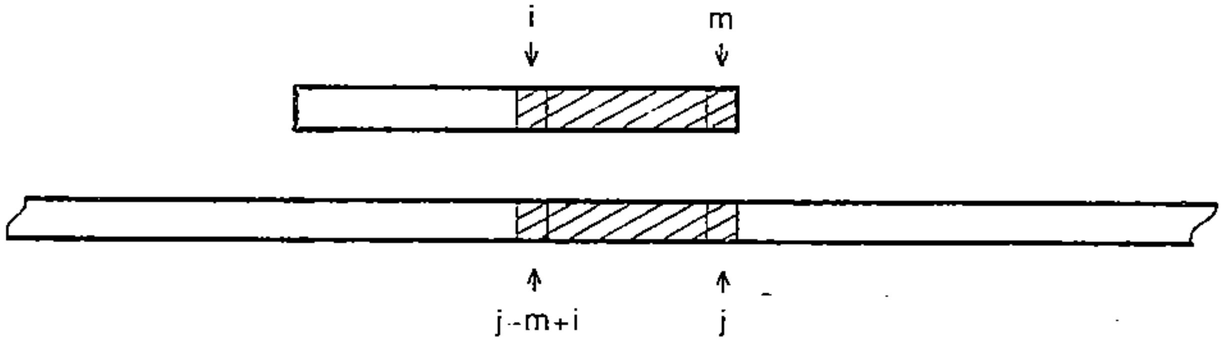

Fig. 1

According to *BM* (*BM'*), if $text[j-m+i] \neq pattern[i]$ the pattern will undergo one more shift as prescribed by the s function. Letting the value of s be $s=k$, Fig. 2 displays the situation that would arise if k more successful matches are performed.

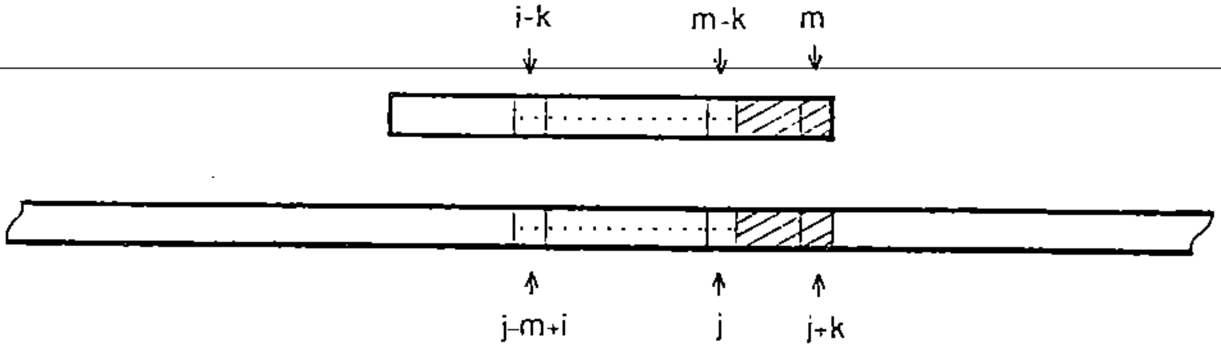

Fig. 2

Plainly stated, *BM* (*BM'*) would keep trying to extend the matched region to the left. In view of the matches achieved during the stage of Fig. 1 (dotted region in the text), however, it is immediately seen that two possibilities are open at this point:

A) The dotted portion of the pattern is also a suffix of the pattern. In principle, this region could be skipped at once, resuming comparisons at the two characters immediately preceding the dotted areas.

B) The dotted portion of the pattern is not a suffix of the pattern, in which case one more shift could be imposed right away.

Thus, if track is kept of past matched segments of the text, and if the structure of the pattern is a-priori known, then the characters falling within these segments need not be reaccessed at subsequent stages. It should be pointed out that, unlike case (A) above, the segments of the text to be skipped at some stage may be more than one, in general. However, we show in this paper that the simple observation above does in fact contribute a substantial saving on the number of character comparisons needed in the process.

In order to proceed to a more formal description of our algorithm we need some means to keep track of which segments of the text matched some suffix of the pattern. In addition, we have to devise a tool - based on the structure of the pattern - that shall enable us to exploit such recorded information in a fast way.

To simplify our description at this stage, we will solve the first problem via the auxiliary array $skip[1:n]$ initialized to 0 and such that whenever in the course of the matching it turns out that, say, $text[l-k+1:l] = pattern[m-k+1:m]$ then $skip[l]$ is set to k. We will show later that a much more space efficient implementation of this bookkeeping is possible, as the reader might already suspect.

The second problem calls for the introduction of the boolean function $Q:\{1,2,\ldots,m\} \times \{1,2,\ldots,m\} \rightarrow \{\text{true}, \text{false}\}$ defined as follows:

$$Q[i,k] = \begin{cases} \text{true} & \text{iff } (k \le i \text{ and } pattern[l-k+1:i] \ne pattern[m-k+1:m]) \\ & or\ (k > i \text{ and } pattern[1:i] \ne pattern[m-i+1:m]) \\ \text{false} & \text{otherwise} \end{cases}$$

We defer to section 4 the actual construction of Q.

The role of the above two implements is transparent. Indeed assume that $skip[l] > 0$ and $Q[i,skip[l]]$ is false. Then either $text[l-skip(l)+1:l] = pattern[i-skip[l]+1:i]$ and $i > skip[l]$, or $text[l-i+1:l] = pattern[l:i]$ and $i \le skip[l]$. In the first case a text segment has been bumped into, which falls entirely within the pattern and which is known to match the pattern in its current position. Otherwise an occurrence of the entire pattern has been detected. We shall see that the management of this latter case embodies the ideas in [3].

The listing of the procedure *BM*", which is given below, features the function s' in place of s. This has to do with the computations of the shifts that have to follow the detection of the condition $Q[i,skip[l]]$ = true. In this case it is known that an already visited segment of the text does not match the substring w of the pattern currently aligned with it, yet it is not known where exactly a character mismatch is located. On the other hand, the function s (s') takes characters and not substrings as one of its arguments. We stipulate in this case to impose a shift based on the value returned by s' in correspondence with the rightmost character of the string w. Notice that this extension of the function s' cannot result in a longer shift, compared to that based on the character that actually causes mismatch. We leave it as an exercise for the reader to show that, in unorthodox circumstances such as above, s could not consistently handle the shift. Although one could envision to use both tables, we elect here to give up the more informative shift function s (*dd*' in [6]), in favor of the conceptually simpler version s' (*dd* in [6]). Fortunately, this has no influence on the upper bound on the number of character comparisons for our strategy. The construct *andif* in the listing of *BM*" is assumed not to check the second condition if the first is false.

```
Procedure BM"

j := m;

do while j ≤ n
     begin
          do i := m to 0 by-max(1,skip[j-m+i])
               until Q[i,skip[j-m+i]] or ((skip[j-m+i] = 0 ) andif (pattern[i] =
               text[j-m+i]))
          if i ≤ 0 then begin output (match at j-m+1); i := 0 end

          skip[j] := m-i; j := j+s'[text[j-m+i],i]
     end
```

As mentioned, the *BM*" turns out to embody the ideas in [3]. In fact it behaves like *BM*', soon after detecting an occurrence of a periodic pattern of the form $u^k u'$ $(k>1)$. In the case *skip*$[j]$ is set to m, resulting in a shift of length $t = |u|$. Since $Q[m-t,m]$ is false, *BM*" will detect a new occurrence of the pattern after only t more successful matches.

Theorem 1: *BM*" detects all occurrences of *pattern*[1:m] in *text*[1,n] by performing at most $2n-m+1$ character comparisons.

Proof: The preceding discussion and the listing of *BM*" establish that all the occurrences of the pattern in the text are indeed detected. The construct *andif* does not check the second condition if the first is false. Each comparison between a character of the text and a character of the pattern may result in either a match or mismatch. If they match, then the text character will be skipped later, whence each text character can be involved in a matching comparison at most once. It is easily seen that the overall number of mismatching comparisons cannot exceed $n-m+1$. Indeed, each time a mismatch is detected this causes a shift to be performed, and there are at most $n-m+1$ shifts. Thus the number of character comparisons performed by *BM*" is at most $2n-m+1$.

□

Theorem 1 conveys the main result of this paper. Such gain in efficiency in terms of character comparisons is largely traded in exchange for a somewhat more complicated preprocessing. The reader might also suspect that the savings on character comparisons boosts the number of the other comparisons, some of which could be taken as surrogates for the former ones. Thus, it is of interest to account for the comparisons needed to check *skip* and Q. The condition *skip*$[j-m+i]$ = 0 is obviously detected in one comparison. We will show later that it takes two comparisons to check that $Q[i,$*skip*$[j-m+i]]$ is true. Both conditions are tested exactly each time a character comparison is performed, plus each time *skip*$[j-m+i] > 0$. Since this latter circumstance can occur at most $n-m+1$ times, we derive that the checks of Q and *skip* cannot exceed a total of $3n-2m+2$, which yields $3(3n-2m+2)=9n-6m+6$ comparisons. Thus the number of both character and noncharacter comparisons is bounded by $11n-7m+7$, which is still slightly better than the $14n$ in [3]. Such figure can be lowered further, at the expense of a more involved construction. This task, however, goes beyond the scope of this paper.

The auxiliary array *skip*[1:m] could be substituted by a circular array of size m in a straightforward way. An even better approach is to make use of a doubly linked list, as follows. Let *text*$[j]$ be currently aligned with *pattern*$[m]$. Whenever a mismatch occurs following $k \geq 1$ successful matches (possibly both of characters and string segments) the right end of the list is updated by appending a new record that stores the values of j and k. Those records that account for the segments falling within the span (k) of the newcomer record are disposed of. Finally, the leftmost record is released whenever the total number of records exceeds m. The details of this construction are quite standard and we leave them for the reader as an exercise. Having stored the value of the text index j each time a record is created makes it also trivial to check later as to whether or not the information stored in it is consistent with the current alignment of *pattern* and *text*. One nice feature of this implementation is its payoff in terms of space occupancy. In absolute terms, this latter is obviously bounded by $O(m)$. We notice, however, that a new entry is appended to the list only following at last one successful character match. The number cc of such matches can be very small, yielding an $O(cc)$ bound that might, in some instances, be better than the former.

4. Preprocessing

The analysis following Theorem 1 relies on the assumption that the truth value of $Q[i,k]$ can be retrieved in exactly two comparisons. We show now how this is made possible by a suitable preprocessing of the pattern.

Let v be a generic string of m characters. For simplicity, we will denote $v[i+1:j]$

shortly as $[i,j]_v$. Recall that a string u is a *period* of v if v is a prefix of u^k, with $k>1$. For each $i \leq m$ let [4]:

$$reach_v[i] = \max\{j \leq m / [0,1]_v \text{ is a period of } [0,j]_v\} =$$
$$= i + \max\{j \leq m-i / [0,j]_v = [i,i+j]_v\}$$

Letting $v = v^R$, the *reverse* string of v, we associate with each position i in v the position $i = m-i+1$ in w. We call i the *conjugate* of i. Let now $revpat = pattern^R$.

Lemma 1: $Q[i,k] =$ true iff $reach_{revpat}\ [i'-1] < \min(m, i'+k-1)$

Proof: Assume that $Q[i,k]=$true. By definition, either (case 1) $0 \leq k \leq i$ and $pattern[i-k+1:i] \neq pattern[m-k+1,m]$ or (case 2) $i<k$ and $pattern[1,i] \neq pattern[m-i+1,m]$. Case 1 implies that $revpat[1:k] \neq revpat[m-i+1:m-i+k]$, that is to say $[0,k]_{revpat} \neq [i'-1,m-i+k]_{revpat}$. Thus the largest q such that $[0,q]_{revpat} = [m-i,m-i+q]_{revpat}$ must be less than k. It follows that $reach_{revpat}[i'-1] = i'-1+q < i'-1+k = m-i+k \leq m$. Case 2 implies that $m < i+k-1$, whence we again need to prove that $reach_{revpat}[i'-1] < m$. This is easily accomplished by an argument analogous to that of case 1. Conversely, assume that $reach_{revpat}\ [i'-1] < \min(m, i'+k-1) = \min(m, m-i+k)$. Now $reach_{revpat}[m-1] = m-i+q$, where q is the largest integer such that $[0,q]_{revpat} = [m-i,m-i+q]_{revpat}$. Consider the case where $k>i$. Then $m < m-i+k$, whence $reach_{revpat}[m-i] < m$. Since $m = m-i+i$, it follows from the definition of *reach* that $[0,i]_{revpat} \neq [m-i,m]_{revpat}$, which implies that $Q[i,k] =$ true. An analogous argument holds for the case where $0 \leq k \leq i$.

□

The information needed for the table *reach* could be collected in liner time as a byproduct of the Knuth-Morris-Pratt algorithm [6]. A more explicit construction is the following. Let $d_1, d_2, \ldots, d_p$ be the sequence of all differences between consecutive occurrences of $w[1]$ in $w[1:m]$. We can put $reach_w[i] = i + prefix_w[i]$, where $prefix_w(i)$ is the longest prefix of w that starts at position $i+1$. This enables to reason in terms of the more handy table $prefix_w$. To simplify matters even further, we extend $w[1{:}m]$ by appending one 'sentinel' location to its right end. In other words, we have now an array $w[1{:}m+1]$ and we assume that $w[m+1]$ contains a symbol not appearing in $w[1{:}m]$. The array $prefix_w[1{:}m]$ is now filled in care of the following procedure.

```
Procedure Prefix
1. for $i$ := 1 to $n$ do $prefix_w[i]$ := 0  *initialize*
2. $i$ := $d_1-1$; $k$ := 0;
3. repeat $k$ := $k+1$ until $w[k] \neq w[k+i]$ *compute first nontrivial entry*
4. $prefix_w[i]$ := $k-1$
5. for $f$ := 2 to $p$ do  *compute all other nontrivial entries*
6.     begin
7.     $j$ := $i$
8.     $i$ := $i+d_f$
9.     if $prefix_w[d_f-1] < k-d_f$ then $prefix_w[i]$ := $prefix_w[d_f]$
10.         else begin $k$ := max(0,$k-d_f$)
11.                 repeat $k$ := $k+1$ until $w[k] \neq w[i+1+k]$
12.                 $prefix_w[i]$ := $k$
13.                 end
14.    end
```

Theorem 2: The procedure *Prefix* correctly computes $prefix_w$ [1:m].

Proof: $prefix_w[1{:}d_1-1]$ is filled with zero's by initialization, and it is easy to check that lines 2 -4 compute $prefix_w[d_1-1]$. Assume now that $prefix_w$ has been correctly computed up to a certain position i such that $w[i+1]=w[1]$ and let $prefix_w[i]$ be equal to some integer $p \geq 1$. Let also d be the smallest integer such that $w[i+d+1]=w[1]$. Let $j=i+d$. The *repeat* loop of line 11 clearly computes $prefix_w[j]$ in the case $k-d_f \leq 0$ (recall that all entries of $prefix_w$ are non negative). It remains to show that also the case $k-d_f > 0$ is dealt with consistently. This case splits in two subcases, both of which exploit the circumstance that position j falls within a replica of a prefix of w starting at i. The value of $prefix_w[i]$ has simply to be recopied from $prefix_w[d-1]$ if this latter is less than $k-d_f$ (line 9). Otherwise, $prefix_w[i]$ is at least $k-d_f$ and we need only check the following characters in an attempt to lengthen it.

□

Theorem 3: The procedure ***Prefix*** takes $O(m)$ time.

Proof: The total work for accessing positions i such that $w[i]=w[1]$ is obviously bounded by m. We can charge the work involved in comparing $w[k]$ and $w[i+1+k]$ to position $i+1+k$. Each such position cannot be charged more than one matching comparison. Mismatching comparisons cannot exceed $p \leq m$, which concludes our proof.

□

The procedure ***Prefix***, once applied to $w = revpat$, makes readily available the desired table $reach_{revpat}$.

5. Concluding Remarks

We have shown that the Boyer-Moore-Galil approach to pattern matching can be upgraded by keeping track of the segments of the pattern successfully matched with the text at each stage. Combining such recordings with a-priori knowledge about the structure of the pattern yields an algorithm which accesses each text character at most twice.

From the standpoint of algorithmic combinatorics, this result is of some merit. Moreover, the increase in terms both of control structure and preprocessing overhead seems to be tolerable. Thus, the overall strategy compares rather favorably with other nontrivial ones, also in the practical perspective.

Acknowledgement

We are indebted to Z. Galil for many helpful comments and suggestions. We are also indebted to the referees for their excellent work in the revision of a preliminary version of this paper. In particular, we gratefully acknowledge the contribution conveyed by one of them, whose selfless profusion of punctual and thoroughly expert advice was of invaluable help in improving the presentation of our ideas.

References

[1] A.V. Aho and M.J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search", *Comm. ACM 18* (1975), 333-340.

[2] R.S. Boyer and J.S. Moore, "A Fast String Searching Algorithm", *Comm. ACM 20* (1977), 262-272.

[3] Z. Galil, "On Improving the Worst Case Running Time of the Boyer-Moore String Searching Algorithm", *Comm. ACM 22* (1979), 505-508.

[4] Z. Galil and J. Seiferas, "Time Space Optimal String Matching", *Journal of Computer and System Sciences 26* (1983), 280-294.

[5] L.J. Guibas and A.M. Odlyzko, "A New Proof of the Linearity of the Boyer-Moore String Searching Algorithm", Proc. 18th Annual IEEE Symposium on Foundations of Computer Science (1977), 189-195.

[6] D.E. Knuth, J.H. Morris and V.B. Pratt, "Fast Pattern Matching in Strings", *SIAM J. on Computing 6* (1977), 189-195.

[7] W. Rytter, "A Correct Preprocessing Algorithm for Boyer-Moore String Searching", *SIAM J. on Computing 9* (1980), 509-512.

[8] A.C.C. Yao, "The Complexity of Pattern Matching for a Random String", Technical Report, Computer Science Department, Stanford University, Stanford, CA (1977)..