

SPEEDING UP TWO STRING-MATCHING ALGORITHMS

Maxime CROCHEMORE^{1,2}, Thierry LECROQ¹

LITP, Institut Blaise Pascal, Université Paris 7, 2 Place Jussieu, 75251 Paris Cedex 05, France

Artur CZUMAJ, Leszek GASIENIEC, Stefan JAROMINEK, Wojciech PLANDOWSKI, Wojciech RYTTER

Institute of Informatics, Warsaw University, ul. Banacha 2, 00-913 Warsaw 59, Poland

Abstract

We show how to speed up two string-matching algorithms : the Boyer-Moore algorithm (BM algorithm) and its version called here the reversed-factor algorithm (the RF algorithm). The RF algorithm is based on factor graphs for the reverse of the pattern. The main feature of both algorithms is that they scan the text right-to-left from the supposed right position of the pattern, BM algorithm goes as far as the scanned segment is a suffix of the pattern, while the RF algorithm is scanning while it is a factor of the pattern. Then they make a shift of the pattern, forget the history and start again. The RF algorithm usually makes bigger shifts than BM, but is quadratic in the worst case. We show that it is enough to remember the last matched segment to speed up considerably the RF algorithm (to make linear number of comparisons with small coefficient) and to speed up BM algorithm with match-shifts (to make at most $2.n$ comparisons). Only a constant additional memory is needed for the search phase. We give alternative versions of an accelerated algorithm RF: the first one is based on combinatorial properties of primitive words, and two others use extensively the power of suffix trees.

1. INTRODUCTION

The Boyer-Moore algorithm [BM 77] is one of the string-matching algorithms very fast on average. However it is successful mainly for the case of big alphabets. For small alphabets its average complexity is $\Omega(n)$, see [BR 91]. We discuss a version of this algorithm, named here the RF algorithm, which is much faster on average, also for small alphabets. If the alphabet is of size at least 2 then the average complexity of the new is $O(n \log(m)/m)$, and reaches the lower bound given in [Yao 79]. The main feature of both algorithms is that they scan the text right-to-left from the supposed right position of the pattern. BM algorithm goes as far as the scanned segment (also called a factor) is a suffix of the pattern, while the RF algorithm matches the text against any factor of the pattern, traversing the factor graph or the suffix tree of the reversed pattern. Afterwards, both algorithms make a shift of the pattern to the right, forget the history and start again. We show that it is enough to remember the last matched segment to speed up the algorithm: an additional constant memory is sufficient.

We derive a version of BM algorithm named here the algorithm Turbo_BM. One of the advantages of this algorithm with respect to the original BM algorithm is the simplicity of the complexity analysis. At the same time the algorithm Turbo_BM looks as a superficial modification of BM algorithm. Only few additional lines are inserted inside the search phase of the original algorithm and two registers (constant memory for the last match) are added. The preprocessing phase is left unchanged. An algorithm remembering a linear number of previous matches has been given before by Apostolico and Giancarlo [AG 86] as a version of BM algorithm. The algorithm Turbo_BM given here seems to be an efficient compromise between the recording of a linear size history as in the Apostolico-Giancarlo algorithm, and no recording of any history about previous matches in the original BM algorithm.

Our method to speed up the BM and RF algorithms is an example of a general technique called in [BKR 91] the dynamic simulation - for a given algorithm A construct the algorithm A' which works in the same way as A but remembering a part of the information A is wasting; during the process such an information is used to save on a part of the computation the original algorithm A does. In our case the additional information is the constant size information about the last match. The transformation of the Boyer-Moore algorithm gives an algorithm of the same simplicity as the original Boyer-Moore algorithm and with the upper bound of $2.n$ on the number of comparisons, which improves slightly on the bound

¹ Work by these authors is partially supported by PRC "Mathématiques-Informatique".

² Work by this author is partially supported by NATO Grant CRG 900293

$3.n$ of the original algorithm. The derivation of this bound is also much simpler than the $3.n$ bound in [Co 89]. The transformations of the RF algorithm show the applicability of data structures representing succinctly the set of all subwords of a pattern p of length m . We denote this set by $FACT(p)$. The set of all suffixes of p is denoted by $SUF(p)$. For simplicity of presentation we assume that the size of the alphabet is constant.

The general structure of BM and RF algorithms looks as in Figure 1.

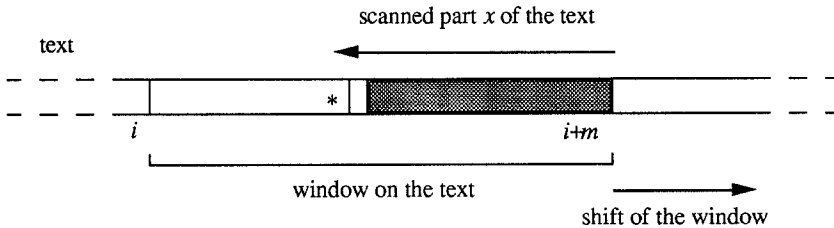


Figure 1. One iteration of Algorithm 1.

The algorithm scans right-to-left a segment (factor) x of the text.

Algorithm 1 /* common schema for algorithms BM and RF */;

$i:=0$;

while $i \leq n-m$ **do**

{ align pattern with positions $t[i+1..i+m]$ of the text;
 scan the text right-to-left from the position $i+m$;
 let x be the scanned part of the text;
if match found ($x = \text{pattern}$) **then** report it;
 compute the shift;
 $i:=i + \text{shift}$; }

end.

In Algorithms BM and RF we use the synonym x for the lastly scanned segment $t[i+j..i+m]$ of the text t . This will shorten the presentation. In one algorithm we check if x is a suffix, and in the second algorithm we check if it is a factor of t . The shift uses a pre-computed function on x . In fact in BM algorithm x is identified with a position j on the pattern, while in the RF algorithm x is identified with a node corresponding to x^R in a data structure representing $FACT(p^R)$. We use the reversed pattern because we scan right-to-left, while most data structures for the set of factors are oriented to left-to-right scanning of the pattern. These orders are equivalent after reversing the pattern. In both cases a constant size memory is sufficient to identify x .

Both algorithms BM and RF can be viewed as instances of Algorithm 1. For a suffix x of length j denote here by $BM_shift[x]$ the match-shift $d2[j-1]$ defined in [KMP 77] (see also [Ry 80] and [Ah 90]). The value of $d2[j-1]$ is, roughly speaking, the minimal shift (>0) of the pattern on itself such that the symbols aligned with the suffix x , except the first letter of x , agree. The symbol at the position aligned with the first letter of x , denoted by $*$ in Figure 2, is distinct, if there is any symbol aligned (see Figure 2).

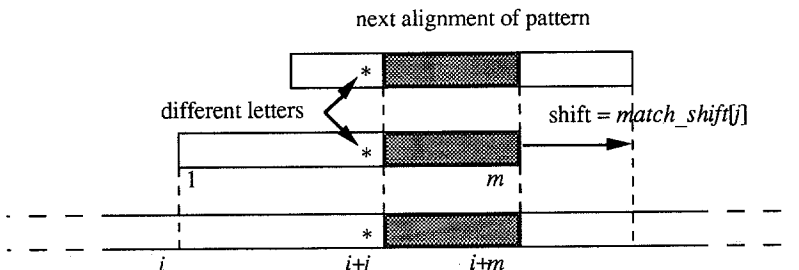


Figure 2. One iteration in BM algorithm.

```

Algorithm BM; /* reversed-suffix string matching */
i:=0; /* denote  $t[i+j..i+m]$  by  $x$ , it is the lastly scanned part of the text */
while  $i \leq n-m$  do
{    $j:=m$ ; while  $j > 1$  &  $x \in \text{SUF}(p)$  do  $j:=j-1$ ;
    if  $x = \text{pattern}$  then report match;
     $\text{shift} := \text{BM\_shift}[j]$ ;
     $i := i + \text{shift}$ ; }
end.

```

The work which Algorithm 1 spends at one iteration is denoted here by *cost*, the shift is denoted by *shift*. In BM algorithm a small cost gives usually a small shift. The strategy of the RF algorithm is more optimal: the smaller is the cost the bigger is the shift. The bigger shifts speed up the algorithm better. In practice *cost*_{*i*} and the match at a given iteration is usually very small, hence the algorithm whose shifts are reversely proportional to the local matches is closer to optimal. The straightforward application of this strategy gives algorithm RF that is very successful on average, unfortunately it is quadratic in the worst case.

```

Algorithm RF; /* reversed-factor string-matching */
i:=0; /* denote  $t[i+j..i+m]$  by  $x$ , it is the lastly scanned part of the text */
while  $i \leq n-m$  do
{    $j:=m$ ; while  $j > 1$  &  $x \in \text{FACT}(p)$  do  $j:=j-1$ ;
/* in fact, we check the equivalent condition  $x^R \in \text{FACT}(p^R)$  */
     $x = t[i+j..i+m]$  is the scanned part of the text;
    if  $x = \text{pattern}$  then report match;
     $\text{shift} := \text{RF\_shift}[x]$ ;  $i := i + \text{shift}$ ; }
end.

```

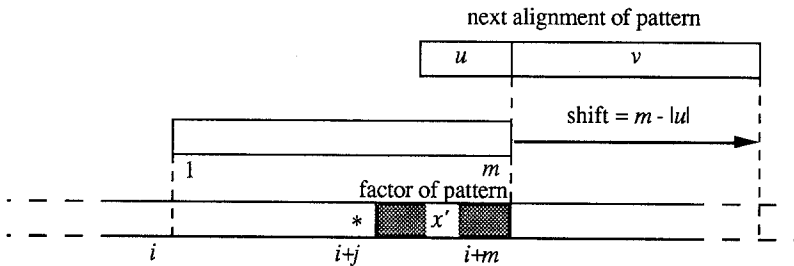


Figure 3. One iteration of Algorithm RF.
Word u is the longest prefix of the pattern that is a suffix of x' .

The algorithm RF makes an essential use of a data structure representing the set $\text{FACT}(p)$, see [BBEHCS 85] for the definition of directed acyclic word graphs (dawg's), [Cr 86] for the definition of suffix automata and see [Ap 85] for details on suffix trees. The graph $G = \text{dawg}(p^R)$ represents all subwords of p^R as labelled paths starting from the root of G . The factor z corresponds in a many-to-one fashion to a node $\text{vert}(z)$ such that the path from the root to that node "spells" z . Additionally we add to each node an information telling whether all paths corresponding to that node are suffixes of the reversed pattern p^R (prefixes of p). We traverse this graph when scanning the text right-to-left in the algorithm RF. Let x' be the longest word which is a factor of p found in a given iteration. When $x=p$ $x'=p$; otherwise x' is obtained by cutting off the first letter of x (the mismatch symbol).

We define the shift RF_shift , and describe how to compute it easily. Let u be the longest suffix of x' which is a prefix of the pattern. It should be a proper suffix of x' iff $x'=x=p$. We can assume that we always know the actual value of u , it is the last node on the scanned path in G corresponding to a suffix of p^R . Then $\text{shift } \text{RF_shift}[x] = m - |u|$ (see Figure 3).

The use of information about the previous match in the next iteration is the key to an improvement. However this application can be realized in many ways: we discuss three alternative transformations for RF. This leads to three versions Turbo_RF, Turbo_RF', and Turbo_RF'' of the RF algorithm that are presented in Section 3.

Algorithms Turbo_BM and Turbo_RF, Turbo_RF', Turbo_RF'' can be viewed as instances of Algorithm 2 presented below.

```

Algorithm 2 /* general schema for algorithms Turbo_RF, Turbo_RF', Turbo_RF'' and Turbo_BM: a
version of Algorithm 1 with an additional memory */;
i:=0; memory:=nil;
while  $i \leq n-m$  do
{
  align pattern with positions  $i+1..i+m$  of the text;
  scan the text right-to-left from the position  $i+m$ ,
      use memory to reduce number of inspections;
  let  $x'$  be the part of scanned text; /*  $x'$  is here usually smaller than  $x$  in Algorithm 1 */
  if match found then report it;
  compute the shift  $shift_i$ ; depending on  $x$  and memory;  $i:=i+shift_i$ ;
  update memory using the information about  $x$ ;
}
end.
    
```

2. SPEEDING UP THE REVERSED-FACTOR ALGORITHM

To speed up algorithm RF we memorize the prefix u of size $m-shift$ of the pattern. We have a situation depicted in Figure 3. We then scan the part of the text align with the part v of the pattern right-to-left. When we arrive at the boundary between u and v in a successful scan (all comparisons positive) then we are in a *decision point*. Now instead of scanning u until a mismatch is found, we just can scan a part of u , due to combinatorial properties of *primitive words*. A word is primitive iff it is not a proper power of a smaller word. We denote by $per(u)$ the length of the smallest period of u . Primitive words have the following useful properties:

- a) the prefix of u of size $per(u)$ is primitive;
- b) a cyclic shift of a primitive word is also primitive, hence the suffix z of u of size $per(u)$ is primitive;
- c) if z is primitive then the situation presented in the Figure 4 is impossible.

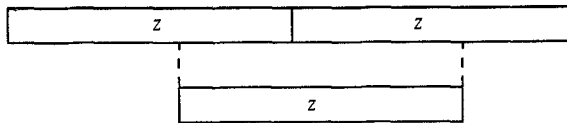


Figure 4. If z is primitive then such an overlap is impossible.

If $y \in FACT(p)$ we denote by $displ(y)$ the least integer d such that $y = p[m-d-|y|+1..m-d]$ (see Figure 5).

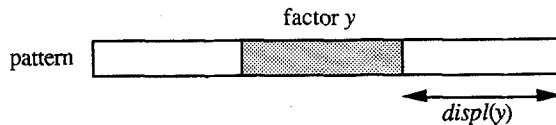


Figure 5. Displacement of factor y in the pattern.

The crucial point is that if we scan successfully v and the suffix of u of size $per(u)$ then we know the shift without further calculations: many comparisons are saved and the algorithm RF can be sped up in this moment. In terms of the next lemma we save $|x|-|zv|$, in the situation when $|x| \geq |zv|$.

Algorithm Turbo_RF ;

/* denote $t[i+j..i+m]$ by x ; it is the lastly scanned part of the text; we memorize the last prefix u of the pattern; initially u is the empty word; */

$i:=0$; $u:=\text{empty}$;

while $i \leq n-m$ **do**

{ $j:=m$; **while** $j > |u|$ & $x \in \text{FACT}(p)$ **do** $j:=j-1$;

if $j=|u|$ **then**

/* we are at the decision point between u and v , after v has been successfully scanned */

if $v \in \text{SUF}(p)$ **then** report a match;

else { scan right-to-left up at most $\text{per}(u)$ symbols stopping at a mismatch;

 let x be the successfully scanned text;

if $|x|=m-|u|+\text{per}(u)$ **then** $\text{shift}:=\text{displ}(x)$ **else** $\text{shift}:=\text{RF_shift}(x)$; }

else $\text{shift}:=\text{RF_shift}(x)$;

$i:=i+\text{shift}$; $u:=\text{prefix of pattern of length } m-\text{shift}$; }

end.

Lemma 1. (key lemma)

Let u, v be as in Figure 3. Assume that u is periodic ($\text{per}(u) \leq |u|/2$). Let z be the suffix of u of length $\text{per}(u)$ and let x be the longest suffix of uv that belongs to $\text{FACT}(p)$. Then

$$zv \in \text{FACT}(p) \text{ implies } \text{RF_shift}(x) = \text{displ}(zv).$$

Proof.

It follows from the definition of $\text{per}(u)$, as the smallest period of u , and periodicity of u that z is a primitive word. The primitivity of z implies that occurrences of z can appear from the end of u only at distances which are multiples of $\text{per}(u)$. Hence $\text{displ}(zv)$ should be a multiple of $\text{per}(u)$, and this easily implies that the smallest proper suffix of uv which is a prefix of p has size $|u|-\text{displ}(zv)$. Hence the next shift in the original algorithm RF is $\text{shift}=\text{displ}(zv)$. ♦

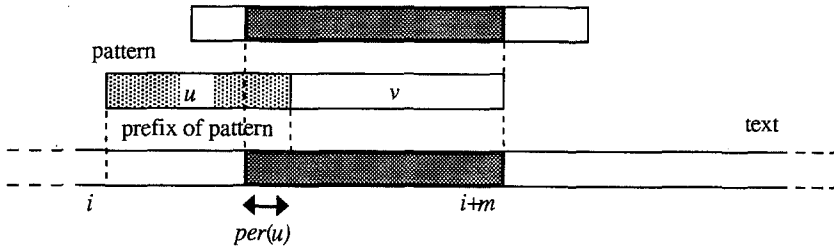


Figure 6. We keep in memory the prefix u of pattern. If u is periodic then at most $\text{per}(u)$ symbols of u are scanned again. However, this extra work is amortized by next shift.

The symbols of v are scanned for the first time.

We need to explain how we add only a constant memory to the algorithm RF. The variables u and x need only pointers to the text. The values of displacements are in the data structure which represents $\text{FACT}(p)$, similarly the values of periods $\text{per}(u)$ for all prefixes of the pattern are pre-computed with the representation of $\text{FACT}(p)$ and read-only in the algorithm. In fact it is possible to remove the table of periods and values of $\text{per}(u)$ can be computed dynamically inside the algorithm Turbo_RF using constant additional memory.

Theorem 2.

The algorithm Turbo_RF makes at most $2.n$ symbol comparisons.

Proof.

If u is periodic then there are scanned again at most $\text{per}(u)$ symbols of u . If u is not periodic then we scan again at most half of u . Let extra_cost be the numbers of symbols inside u scanned in the actual stage. In each case $\text{extra_cost} \leq \text{next_shift}$. Hence it is amortized by the next shift, this gives together at most n comparisons. The symbols in parts v are scanned for the first time in a given stage. They are disjoint in distinct phases. Hence they give together at most n comparisons. The work spent inside segments u and inside segments v is thus bounded by $2.n$. This completes the proof. ♦

We can check whether $suf(k)$ is a descendant of $repr(v^R)$ in a constant time, after preprocessing the suffix tree T . We can number the nodes of the tree in a DFS order. Then the nodes which are descendants of a given node form an interval of consecutively numbered nodes. Associate with each node such an interval. Then the question about descendants is reduced to an inclusion of an integer in a known interval. This can be answered in a constant time. Altogether this gives the algorithm Turbo_RF'.

Theorem 3.

The algorithm Turbo_RF' makes at most n symbol comparisons of pattern versus text t . The total number of iterations $P^h(j)$ done by the algorithm does not exceed n . The preprocessing time is also linear.

Proof.

We have already discussed the preprocessing phase. Each time we make an iteration of type $P^h(j)$ the pattern is shifted to the right of the text by at least one position, hence there are at most n such iterations. This completes the proof. ♦

The second approach

Here we improve the complexity of the search phase of the algorithm considerably. This increases the cost of the preprocessing phase that however is still linear. In the algorithm Turbo_RF' at a decision point we have sometimes to spend a linear time to make many iterations of type $P^h(j)$. In this new version, we compute the shift in constant time. It is enough to show how to preprocess the suffix tree T for p' to compute $nextsuf(v)$ for any factor v of p' in a constant time whenever it is needed.

First we show how to compute $nextsuf(v)$ for main factors, i.e. factors corresponding to nodes of T . Let us identify main factors with their corresponding nodes. The computation is in a bottom-up manner on a tree T .

Case of a bottom node: v is a leaf.

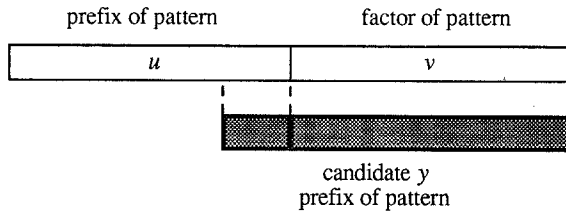
$nextsuf(v) = nil$;

Case of an internal node:

assume v has sons v_1, v_2, \dots, v_q , then there exists a son v_j such that $nextsuf(v) = nextsuf(v_j)$ or, if v_j is a leaf then $nextsuf(v) = v_j$

We scan the sons v_j of v , and for each of them check if $nextsuf(v_j)$ or v_j is a good candidate for $nextsuf(v)$. We choose the longest good candidate, if there is any. Otherwise the result is nil .

The word v is prefix of each of the candidates. What does it exactly mean for a word y , whose prefix is v , to be a good candidate? Let u be the prefix of the pattern p of length $m-|v|$. The candidate y is good iff the prefix of y of length $|y|-|v|$ is a suffix of u , see the Figure 8. This means that the prefix of the pattern which starts at position $|u|-(|y|-|v|)$ continues to the end of $|u|$. We have to be able to check it in constant time.



Hence after preprocessing we keep a certain amount of additional data: suffix tree, the table of $nextpref(v)$ for all main nodes of this tree and the table $PREF$. Anyway, altogether this needs only linear size memory and is later accessed in a read-only way.

Theorem 4.

The pattern can be preprocessed in linear time in such a way that the computation of the RF_shift in the algorithm Turbo_RF can be accomplished in a constant time whenever it is needed. The data structure used in the preprocessing phase is read-only at search phase. Only a constant read-write memory is used at search phase.

Denote by Turbo_RF" the version of the algorithm Turbo_RF in which the computation of the RF_shift is computed at decision points according to Theorem 4. The resulting algorithm Turbo_RF" can be viewed as an automata-oriented string-matching. We scan the text backwards and change the state of the automaton. The shift is then specified by state of the automaton where the scanning stops. This idea applied directly gives a kind of Boyer-Moore automaton of polynomial (but not linear) size [Le 91]. However it is enough to keep in memory only a linear part of such an automaton (implied by the preprocessing referred in the theorem).

4. THE ANALYSIS OF THE AVERAGE CASE COMPLEXITY OF THE ALGORITHM RF AND ITS VERSIONS.

Denote by A the alphabet and by r the size of A . Assume $r > 1$.

Assume that the input alphabet has $r > 1$ letters. We consider the situation when the text is random. The probability of the occurrence of a specified letter on the i -th position is $1/r$, and these probabilities are independent (for distinct positions).

Theorem 5.

The expected time of the algorithm RF is $O(n \log_r(m)/m)$.

Proof.

Let $cost_i$ be the number of comparisons made in the i -th iteration of the algorithm. The shift computed in this iteration is denoted by $shift_i$. If $cost_i \leq 4 \log_r m$ then $shift_i \geq m - 4 \log_r m$ and $shift_i$ is called *long*. It is called *short* otherwise. For a m which is big enough each long shift satisfies : $shift_i \geq m/2$.

The proof relies on two claims that we first establish :

Claim -i-.

The probability that $shift_i$ is short is less than $1/m^2$.

Proof of Claim -i-.

There exists less than m^2 different factors of length $4 \log_r m$ in the pattern and there may be $r^{4 \log_r m} = m^4$ different strings of this length in the scanned text. If we divide these numbers we obtain the required result. This completes the proof of the claim.

Let us partition the text t into disjoint segments of length $m/2$. The sequence of the iterations of the algorithm is called the k -th *phase* iff it consists of all iterations of the algorithm with the supposed end of the pattern placed in the k -th segment. Hence there are at most $2n/m$ phases in the algorithm. Now we study the expected cost of one phase. Let X_k be the random variable, whose value is the cost spent by the algorithm in the k -th phase.

Claim -ii-.

The expected cost of X_k is $\text{ave}(X_k) = O(\log_r m)$.

Proof of Claim -ii-.

Let us estimate separately the cost of the first iteration and the expected cost of other iterations in the k -th phase. The probability that the first shift in the k -th phase is short is less than $1/m^2$, due to Claim -i-. If this shift is long then the cost is logarithmic. Otherwise, the cost of all other iterations in the k -th phase does not exceed m^2 . However the probability that we start the second iteration in the phase is less than $1/m^2$. Hence all these iterations contribute together $O(m^2 \cdot 1/m^2)$ average cost. Altogether we have $O(\log m + 1)$, which is of a logarithmic order. This completes the proof of the claim.

The cost of the whole algorithm is :

$$\text{ave}(X_1 + X_2 + \dots + X_{2n/m}) = \text{ave}(X_1) + \text{ave}(X_2) + \dots + \text{ave}(X_{2n/m}) = O(n \cdot \log(m)/m).$$

This completes the proof of the theorem. ♦

5. SPEEDING UP THE BOYER-MOORE ALGORITHM

The linear time complexity of the Boyer-Moore algorithm is quite nontrivial. The first proof of the linearity of the algorithm appeared in [KMP 77]. Other authors have worked on it (see [Ga 79] and [GO 80]), however it was needed more than a decade for the full analysis. R. Cole has proved that the algorithm makes at most $3 \cdot n$ comparisons, see [Co 90], and that this bound is tight. The “mysterious” behavior of the algorithm is due to the fact that it forgets the history and the same part of the text can be scanned an unbounded (at most logarithmic) number of times. The whole “mystery” disappears when the whole history is memorized and additional $O(m)$ size memory is used. Then in successful comparisons each position of the text is inspected at most once. The resulting algorithm is an elegant string-matching algorithm (see [AG 86]) with a straightforward analysis of the text searching phase. However it requires more preprocessing and more tables than the original BM algorithm. In our approach no extra preprocessing is needed and the only table we keep is the original table of shifts used in BM algorithm. Hence all extra memory is of a constant size (two integers). The resulting algorithm Turbo_BM forgets all its history except the most recent one and its behavior has again a “mysterious” character. Despite that, the complexity is improved and the analysis is simple.

The main feature of the algorithm Turbo_BM is that during the process the factor of the pattern that matched the text during the last attempt is memorized. This has two advantages: it can lead to both a jump over the factor during the scanning phase and to, what is called, a *Turbo_shift*.

We now explain what is a *Turbo_shift*. Let x be the longest suffix of p that matches the text at a given position. Let also $fact$ be the stored factor that matches the text at the same position. For different letters a and b , ax is a suffix of p aligned with bx in the text (see Figure 9). A *Turbo_shift* can occur when x is shorter than $fact$. In this situation the suffix ax of the pattern is aligned with the factor bx of the text and a, b are different letters. Since x is shorter than $fact$, ax is a suffix of $fact$. Thus a and b occur at distance $m-g$ in the text. But, since suffix $p[f+1..m]$ of p has period $m-g$ it cannot overlap both letters a and b . As a consequence the smallest shift of the pattern is then $(m-f)-(m-j)-(m-g)$, that is, $j+g-f-m$ (see Figure 10). This proves that if Algorithm BM is correct so is Algorithm Turbo_BM.

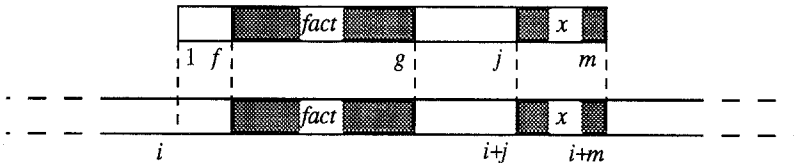


Figure 9. Variables i, j, f and g in Turbo_BM (matching parts in grey).

```

Algorithm Turbo_BM; /* reversed-suffix string matching */
i:=0; /* denote  $t[i+j..i+m]$  by  $x$ , it is the lastly scanned part of the text */
f:=0; g:=0; /* factor  $p[f+1..g]$  of  $p$  matches the text  $t[i+f+1..i+g]$  */
while  $i \leq n-m$  do
{
  j:=m; while  $j > 0$  &  $x \in SUF(p)$  do if  $j=g+1$  then  $j:=f$  /* jump */ else  $j:=j-1$ ;
  if  $x = \text{pattern}$  then report match;
  if  $g-f > m-j$  then Turbo_shift :=  $g+j-m-f$  else Turbo_shift := 0;
  if  $BM\_shift[j] \geq Turbo\_shift$  then
  {
     $\bar{i} = i + BM\_shift[j]$ ;
     $f = \max(\bar{0}, j - BM\_shift[j]$ ;  $g = m - BM\_shift[j]$ ; }
  else
  {
     $i = i + Turbo\_shift$ ;
     $f = 0$ ;  $g = 0$ ; }
}
end.

```

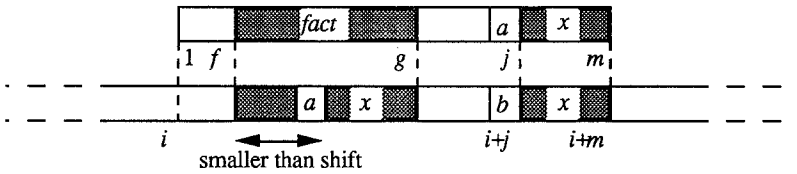


Figure 10. Turbo_shift.

Theorem 6.

The algorithm Turbo_BM makes at most $2.n$ comparisons.

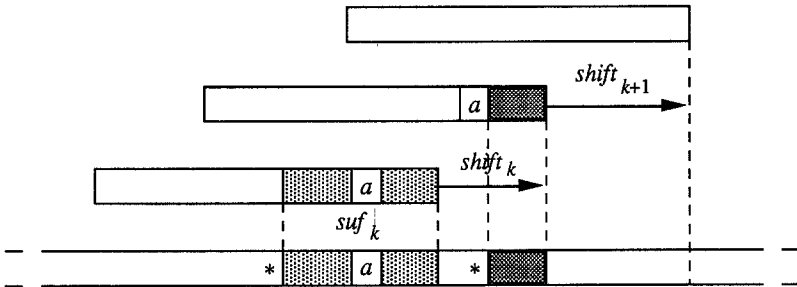
Proof.

We decompose the searching phase into stages. Each stage is itself divided into the two operations: scan and shift. At stage k we call Suf_k the suffix of the pattern that matches the text and suf_k its length. It is preceded by a letter that does not match the aligned letter in the text. At the end of stage k , suffix Suf_k is stored to avoid scanning it again. At next stage, we say that there is a (true) jump if the letter of the text which precedes the occurrence of Suf_k is tested again. We also call $shift_k$ the length of the shift done at stage k . Let $cost_k$ be the number of comparisons done at this stage.

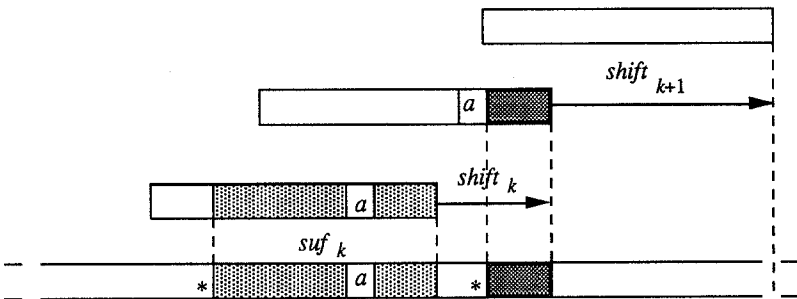
Consider three types of stages according to the natures of the scan and of the shift:

- (i) stage followed by a stage with jump,
- (ii) no type (i) stage with long shift.
- (iii) no type (i) stage with short shift,

We say that a shift is short if $2 \cdot shift_k < (suf_k + 1)$.



Case (a)



Case (b)

Figure 11. The costs of stages k and $k+1$ correspond to shadowed areas plus mismatches denoted by stars (together at most $suf_k + shift_{k+1} + 1$ comparisons). If $shift_k$ is small then $shift_{k+1}$ is big enough (together with $shift_k$) to amortize the costs.

To prove the theorem it is enough to show that the sum of all costs is less than twice the sum of all shifts: $\Sigma cost_k \leq 2 \cdot \Sigma shift_k$.

We count separately the number of comparisons done during stages. We consider a stage k .

If stage k is of type (i) with no jump its cost is suf_k+1 . The comparison with mismatch is one, obviously less than $2 \cdot shift_k$. The rest of the cost, suf_k , is accounted to the next stage. If stage k is of type (i) with a jump its cost is also suf_k+1 by the previous rule. Then the same applies.

If stage k is of type (ii), $cost_k = suf_k+1 \leq 2 \cdot shift_k$.

It remains to consider stages of type (iii). The situation is displayed in the Figure 11. This case is itself divided into two subcases. In case (a), $suf_k+shift_k \leq m$ and the contrary holds in case (b).

Case (a). $suf_k+shift_k \leq m$

Since stage $k+1$ makes no jump, a mismatch should occur inside the suffix of p of length $shift_k$. The *Turbo_shift* then implies $shift_{k+1} \geq suf_k - cost_{k+1} + 1$.

Therefore, $cost_k + cost_{k+1} \leq suf_k + 1 + shift_k \leq 2 \cdot shift_k + shift_{k+1} + 1 \leq 2 \cdot (shift_k + shift_{k+1})$.

Case (b). $suf_k+shift_k > m$

If no mismatch happens at stage $k+1$, $cost_k + cost_{k+1} \leq m + shift_k$. An occurrence of p is found in the text. Then we count $cost_k$ as $shift_k$ and $cost_{k+1}$ as m . In other words, $m - shift_k$ comparisons are reported from stage k to stage $k+1$.

If a mismatch happens at stage $k+1$, as in case (a), the *Turbo_shift* implies $shift_{k+1} \geq m - 2 \cdot shift_k + 1$. Then $cost_k + cost_{k+1} \leq m + cost_{k+1} \leq shift_{k+1} + 2 \cdot shift_k - 1 + cost_{k+1}$.

If $cost_{k+1} \leq shift_{k+1}$ we get the result. The assumption holds in particular if stage $k+1$ is of type (i) because then its cost is 1.

It remains to look at the case where $shift_{k+1} < cost_{k+1}$ and stage $k+1$ is not of type (i). The *Turbo_shift* at stage $k+2$ implies that $shift_k + shift_{k+1} + shift_{k+2} > m$. On the other hand, $cost_k + cost_{k+1} + cost_{k+2} \leq m + shift_k + shift_{k+1} < 2 \cdot m$ (because $shift_k + shift_{k+1} < m$). This shows that $cost_k + cost_{k+1} + cost_{k+2} \leq 2 \cdot (shift_k + shift_{k+1} + shift_{k+2})$.

This ends case (b) and the whole proof. ♦

6. FINAL REMARKS

Remark 1 (on the algorithm Turbo_BM)

In the algorithm Turbo_BM we deal only with match shifts of the Boyer-Moore algorithm. If the alphabet is binary then occurrence shifts in Boyer-Moore are useless. Generally for small alphabets the occurrence heuristics have little effect. For bigger alphabets we can include in the algorithm the occurrence shifts in many possible ways. The simplest way to do it is to choose the match_shift if both match_shift and occ_shift are small (at most $suf_k/2$). In other words, the occurrence shift is considered only when it is reasonably big. Then the same analysis applies.

Another alternative is to consider all occurrence shifts, but this leads to an algorithm which uses only additional memory dynamically updated during the text-search phase. However, it seemingly requires additional linear preprocessing and a linear size table which after preprocessing will be read only.

Remark 2 (on the fast multi-pattern string-matching)

One can easily extend the RF algorithm to the multi-pattern string matching. Let p_1, p_2, \dots, p_k be a set of patterns and $FACT(p_1, p_2, \dots, p_k)$ be the set of all factors of the pattern.

Algorithm multi-RF; /*reversed-factor string-matching for patterns p_1, p_2, \dots, p_k */

m :=length of the shortest pattern

i :=0; /* denote $t[i+j..i+m]$ by x , it is the lastly scanned part of the text */

while $i \leq n-m$ **do**

{ j := m ; **while** $j > 1$ & $x \in FACT(p_1, p_2, \dots, p_k)$ **do** j := $j-1$;

$x = t[i+j..i+m]$ is the scanned part of the text;

if $x = p_r$ for some r in $[1..k]$ **then** report match;

$shift$:= $multi-RF_shift[x]$; i := $i+shift$; }

end.

The definition of the table $RF_shift[x]$ can be extended to many patterns in a natural way. When defining $shift\ RF_shift[x] = m - |u|$, we allow uv to be a prefix of one of p_1, p_2, \dots, p_k , see Figure 3.

The algorithm multi-RF is also fast on average, however similarly as RF it takes quadratic time in pessimistic case. We are able to make an accelerated version of this algorithm similar to Turbo-RF'. The accelerated algorithm Turbo-multi-RF has $O(n \log(m))$ time complexity, or it can have $O(n)$ time complexity if we use a table of $O(m^2)$ size. This table does not need to be initialized and only a linear sized part of it is used.

REFERENCES

- [Ah 90] A.V. AHO, Algorithms for finding patterns in strings, in: (J. VAN LEEUWEN, editor, *Handbook of Theoretical Computer Science*, vol A, *Algorithms and complexity*, Elsevier, Amsterdam, 1990) 255-300.
- [Ap 85] A. APOSTOLICO, The myriad virtues of suffix trees, in: (A. APOSTOLICO, Z. GALIL, editors, *Combinatorial Algorithms on Words*, NATO Advanced Science Institutes, Series F, vol. 12, Springer-Verlag, Berlin, 1985) 85-96.
- [AG 86] A. APOSTOLICO, R. GIANCARLO, The Boyer-Moore-Galil string searching strategies revisited, *SIAM J.Comput.* 15 (1986) 98-105.
- [BR 91] R.A. BAEZA-YATES, M. RÉGNIER, Average running time of the Boyer-Moore-Horspool algorithm, *Theoret. Comput. Sci.* (1991) to appear.
- [BBEHCS 85] A. BLUMER, J. BLUMER, A. EHRENFUCHT, D. HAUSSLER, M.T. CHEN, J. SEIFERAS, The smallest automaton recognizing the subwords of a text, *Theoret. Comput. Sci.* 40 (1985) 31-55.
- [BKR 91] L. BANACHOWSKI, A. KRECZMAR, W. RYTTER, *Analysis of algorithms and data structures*, Addison Wesley, 1991.
- [BM 77] R.S. BOYER, J.S. MOORE, A fast string searching algorithm, *Comm. ACM* 20 (1977) 762-772.
- [Co 90] R. COLE, Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm, in: (*2nd annual ACM Symp. on Discrete Algorithms*, 1991) 224-233
- [Cr 86] M. CROCHEMORE, Transducers and repetitions, *Theoret. Comput. Sci.* 45 (1986) 63-86.
- [Ga 79] Z. GALIL, On improving the worst case running time of the Boyer-Moore string searching algorithm, *Comm. ACM* 22 (1979) 505-508.
- [GO 80] L.J. GUIBAS, A.M. ODLYZKO, A new proof of the linearity of the Boyer-Moore string searching algorithm, *SIAM J.Comput.* 9 (1980) 672-682.
- [KMP 77] D.E. KNUTH, J.H. MORRIS Jr, V.R. PRATT, Fast pattern matching in strings, *SIAM J.Comput.* 6 (1977) 323-350.
- [Le 91] T. LECROQ, A variation on Boyer-Moore algorithm, *Theoret. Comput. Sci.* (1991) to appear.
- [Ry 80] W. RYTTER, A correct preprocessing algorithm for Boyer-Moore string searching, *SIAM J.Comput.* 9 (1980) 509-512.
- [Ya 79] A.C. YAO, The complexity of pattern matching for a random string, *SIAM J.Comput.* 8 (1979) 368-387.