Algorithmica (1994) 12: 247–267

Algorithmica
© 1994 Springer-Verlag New York Inc.

Speeding Up Two String-Matching Algorithms[1]

M. Crochemore,[2] A. Czumaj,[3] L. Gasieniec,[3] S. Jarominek,[3] T. Lecroq,[2]
W. Plandowski,[3] and W. Rytter[3]

Abstract. We show how to speed up two string-matching algorithms: the Boyer–Moore algorithm (BM algorithm), and its version called here the reverse factor algorithm (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. The BM algorithm goes as far as the scanned segment (factor) is a suffix of the pattern. The RF algorithm scans while the segment is a factor of the pattern. Both algorithms 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 (represented by two pointers to the text) to speed up the RF algorithm considerably (to make a linear number of inspections of text symbols, with small coefficient), and to speed up the BM algorithm (to make at most $2 \cdot n$ comparisons). Only a constant additional memory is needed for the search phase. We give alternative versions of an accelerated RF algorithm: the first one is based on combinatorial properties of primitive words, and the other two use the power of suffix trees extensively. The paper demonstrates the techniques to transform algorithms, and also shows interesting new applications of data structures representing all subwords of the pattern in compact form.

Key Words. Analysis of algorithms, Pattern matching, String matching, Suffix tree, Suffix automaton, Combinatorial problems, Periods, Text processing, Data retrieval.

1. Introduction. The Boyer–Moore algorithm [BM] is one of the string-matching algorithms which is very fast on the average. However, it is successful mainly in the case of large alphabets. For small alphabets, its average complexity is $\Omega(n)$ (see [BR]) for the Boyer–Moore–Horspool version [H]. The reader can refer to [HS] for a discussion on practical fast string-searching algorithms. We discuss here a version of this algorithm, called the RF algorithm, which is much faster on the average, not only on large alphabets but also for small alphabets. If the alphabet is of size at least 2, then the average complexity of the new algorithm is $O(n \log(m)/m)$, and reaches the lower bound given in [Y]. The main feature of both algorithms is that they scan the text right-to-left from a supposed right position of the pattern. The BM algorithm goes as far as the scanned segment (also called the factor) is a suffix of the pattern, while the RF algorithm matches the text

[1] The work by M. Crochemore and T. Lecroq was partially supported by PRC "Mathématiques-Informatique," M. Crochemore was also partially supported by NATO Grant CRG 900293, and the work by A. Czumaj, L. Gasieniec, S. Jarominek, W. Plandowski, and W. Rytter was supported by KBN of the Polish Ministry of Education.

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

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

Received August 15, 1991; revised August 17, 1992, and November 11, 1992. Communicated by Alberto Apostolico.

against any factor of the pattern, traversing the factor graph or the suffix tree of the reverse pattern. Afterward, 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 algorithms: an additional constant memory is sufficient. We derive a version of the BM algorithm, called the Turbo_BM algorithm. One of the advantages of this algorithm with respect to the original BM algorithm is the simplicity of its analysis of complexity. At the same time, the Turbo_BM algorithm looks like a superficial modification of the BM algorithm. Only a few additional lines are inserted inside the search phase of the original algorithm, and two registers (constant memory to keep information about the last match) are added. The preprocessing phase is left unchanged. Recall that an algorithm remembering a linear number of previous matches has been given before by Apostolico and Giancarlo [AG] as a version of the BM algorithm. The Turbo_BM algorithm given here seems to be an efficient compromise between recording a linear-size history, as in the Apostolico–Giancarlo algorithm, and not recording the history of previous matches, as in the original BM algorithm.

Our method to speed up the BM and RF algorithms is an example of a general technique called the dynamic simulation in [BKR]—for a given algorithm A construct an algorithm A′ which works in the same way as A, but remembering part of the information that A is wasting; during the process this information is used to save part of the computation carried out by the original algorithm A. 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, but with the upper bound of $2 \cdot n$ on the number of comparisons, which improves slightly on the bound $3 \cdot n$ of the original algorithm. The derivation of this bound is also much simpler than the $3 \cdot n$ bound in [Co]. The previous bounds, established when the pattern does not occur in the text, are $7 \cdot n$ in [KMP] and $4 \cdot n$ in [GO]. It should be noted that a simple transformation of the BM algorithm to search for all occurrences of the pattern has quadratic-time complexity. Galil [G] has shown how to make it linear in this case.

Several 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 suffices 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 the BM and RF algorithms is shown in Figure 1.

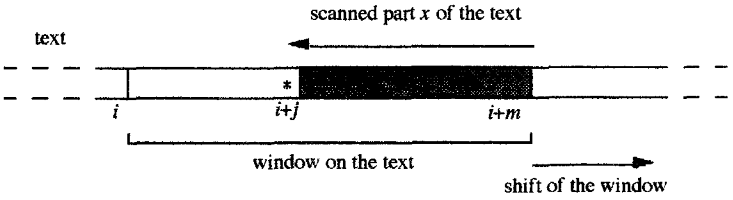


Fig. 1. One iteration of Algorithm 1. The algorithm scans right-to-left a segment (factor) x of the text.

```
**Algorithm 1**   /* common scheme for the BM and RF algorithms */
$i := 0$;
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 position $i+m$;
    let $x$ be the scanned part of the text;
    if $x = p$ then report a match at position $i$;
    compute the shift;
    $i := i + shift$; }
end.
```

In algorithms BM and RF we use the synonym x for the last-scanned segment $t[i+j..i+m]$ of the text t. This shortens the presentation. In one algorithm it is checked whether x is a suffix of p and, in the second algorithm, whether it is a factor of p. Shifts use a precomputed function on x. In fact, in the 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 reverse 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 the BM and RF algorithms can be viewed as instances of Algorithm 1. For a suffix x at position k, denote here by $BM\_shift[x]$ the match-shift $d2[k]$ defined in [KMP] for the BM algorithm (see also [Ah] or [R]). The value of $d2[k]$ is, roughly speaking, the minimal (nontrivial) shift of the pattern over itself such that the symbols aligned with the suffix x, except the first letter of x, agree. The symbol at the position, denoted by *, aligned with the first letter of x in Figure 2, is distinct if, in fact, any symbol aligns. The BM algorithm also uses heuristics on the alphabet. A second shift function serves to align the mismatch symbol in the text with an occurrence of it in the pattern. We mainly consider the BM algorithm without the heuristics. However, this feature is integrated in the final version of the Turbo_BM algorithm.

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

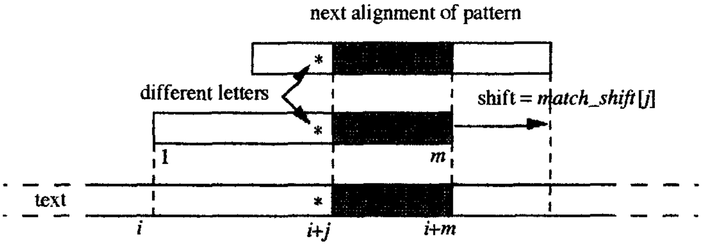


Fig. 2. One iteration in the BM algorithm.

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

The work which Algorithm 1 spends at one iteration is denoted here by *cost*, and the length of the shift is denoted by *shift*. In the BM algorithm *cost* is usually small but it gives a small shift. The strategy of the RF algorithm is more "optimal": the smaller the cost, the bigger the shift. In practice on the average, the match (and *cost*) at a given iteration is usually small; hence, the algorithm, whose shifts are inversely proportional to local matches, is close to optimal. The straightforward application of this strategy gives an RF algorithm that is very successful on the average. It is, however, quadratic in the worst case.

Algorithm RF makes essential use of a data structure representing the set $FACT(p)$. See [BBE$^+$] for the definition of directed acyclic word graphs (dawg's), see [Cr] for the definition of suffix automata, and see [Ap] for details on suffix trees. The graph $G = dawg(p^R)$ represents all subwords of p^R as labeled paths starting from the root of G. The factor z corresponds in a many-to-one fashion to a node $vert(z)$, such that the path from the root to that node "spells" z. Additionally, we add information to each node indicating whether all nodes 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 RF algorithm. Let x' be the longest word, which is a factor of p, found in a given iteration. When $x = p$, then $x' = x$; otherwise x' is obtained by cutting off the first letter of x (the mismatch symbol). The time spent scanning x is proportional to $|x|$. The multiplicative factor is constant if a matrix representation is used for transitions in the data structure. Otherwise, it is $O(\log |A|)$ (where A can be restricted to the pattern alphabet), which applies for arbitrary alphabets.

We now define the shift *RF_shift*, and describe how to compute it easily. Let u be the longest suffix of x' which is a proper prefix of the pattern p. We can assume

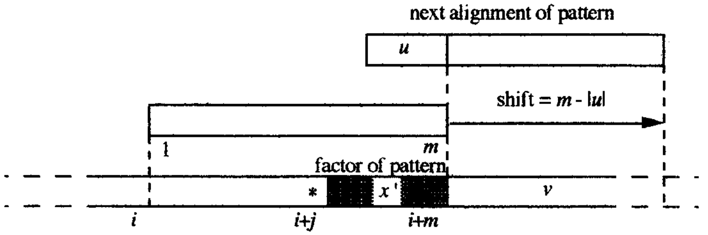


Fig. 3. One iteration of algorithm RF. Word u is the longest prefix of the pattern that is a suffix of x'.

that we always know the actual value of u, associated with the last node on the scanned path in G corresponding to a suffix of p^R. Then shift $RF\_shift[x] = m - |u|$ (see Figure 3).

The use of information about the previous match at a given iteration is the key to improvement. However, this application can be realized in many ways: we discuss three alternative transformations for RF. They lead to three versions of the RF algorithm, Turbo_RF, Turbo_RF′, and Turbo_RF″, that are presented in Sections 2 and 3. Algorithms Turbo_BM, Turbo_RF, Turbo_RF′, and Turbo_RF″ can be viewed as instances of Algorithm 2 presented below.

```
Algorithm 2  /* general scheme 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 $t[i+1..i+m]$ of the text;
   scan the text right-to-left from the position $i + m$, using memory to
   reduce number of inspections;
   let $x'$ be the part of the text scanned;
   if $x = p$ then report a match at position $i$;
   compute the shift $shift_i$ according to $x$ and $memory$;
   $i := i + shift_i$; update $memory$ using $x$; }
end.
```

2. Speeding up the Reverse Factor Algorithm. To speed up the RF algorithm we memorize the prefix u of size *m-shift* of the pattern (see Figure 6). The scan, between the part of the text to align with part v of the pattern, is done from right to left. When we arrive at the boundary between u and v in a successful scan (all comparisons positive), then we are at a *decision point*. Now, instead of scanning u until a mismatch is found, we can just scan (again) a part of u, due to the 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.

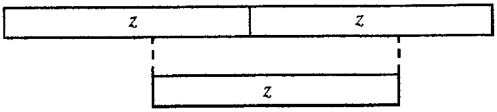


Fig. 4. If z is primitive, then such an overlap is impossible.

(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.

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).

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

```
Algorithm Turbo_RF
/* denote $t[i+j..i+m]$ by $x$; it is the last-scanned part of the text; we
   memorize the last prefix $u$ of the pattern; initially $u$ is the empty
   word; */
$i := 0$; $u :=$ empty;
while $i \leq n - m$ do
{  $j := m$; while $j > |u|$ and $x \in FACT(p)$ 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 $v \in SUF(p)$ then report a match at position $i$;
      else  { scan right-to-left up at most $per(u)$ symbols stopping at a
              mismatch;
              let $x$ be the successfully scanned text;
              if $|x| = m - |u| + per(u)$ then shift := $displ(x)$
              else $shift := RF\_shift(x)$; };
   else $shift := RF\_shift(x)$;
   $i := i + shift$; $u :=$ prefix of pattern of length $m - shift$; }
end.
```

Lemma 1 (Key Lemma). *Let u, v be as in Figure 3. Assume that u is periodic ($per(u) \leq |u|/2$.). Let z be the suffix of u of length $per(u)$ and let x be the longest*

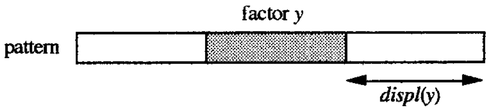


Fig. 5. Displacement of factor y in the pattern.

suffix of uv that belongs to $FACT(p)$. Then

$$zv \in FACT(p) \quad \textit{implies} \quad RF\_shift(x) = displ(zv).$$

PROOF. It follows from the definition of $per(u)$, as the smallest period of u, and the 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 $per(u)$. Hence, $displ(zv)$ should be a multiple of $per(u)$, and this easily implies that the smallest proper suffix of uv which is a prefix of p has size $|uv| - displ(zv)$. Hence, the next shift in the original RF algorithm is $shift = displ(zv)$. □

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

THEOREM 2. *On a text of length n, the Turbo_RF algorithm runs in time $O(n \cdot \log |A|)$ for arbitrary alphabets. It makes at most $2 \cdot n$ inspections of text symbols.*

PROOF. In the Turbo_BM algorithm at most $per(u)$ symbols of u are scanned. Let *extra_cost* be the number of symbols of u scanned at the actual stage. Since the next shift has length at least $per(u)$, $extra\_cost \leq next\_shift$. Hence, all extra inspections of symbols are amortized by th toal sum of shifts, which gives at most n inspections. The symbols in parts v are scanned for the first time in a given stage. They are disjoint in distinct phases. Hence, they also give at most n inspections. The work spent inside segments u, and inside segments v, is thus bounded by $2 \cdot n$. This completes the proof. □

3. Two Other Variations of Algorithm Turbo_RF Assume that we are at a decision point when we have just finished scanning v (see Figure 6). At this moment, we know that the part of the text immediately to the left is a prefix u of p of size $m - |v|$. Denote by $nextpref(v)$ the longest suffix of uv that is a proper prefix of p, and that is longer than v. If there is no such suffix, then denote the corresponding value by *nil*. The next *RF_shift* will be equal to $m - |nextpref(v)|$. The next value of u will be $nextpref(v)$. All these elements are uniquely determined by v. Hence, after suitable preprocessing of the pattern no symbols of u need to be read. The algorithm will make at most n inspections of text symbols against the pattern. However, the complexity is affected by the computation of $nextpref(v)$.

There are at least two possible approaches. One is to precompute a data structure which allows the computation at the kth iteration of the value of $nextpref(v)$ in a time $cost'_k$ such that the sum of all $cost'_k$'s is linear. The second

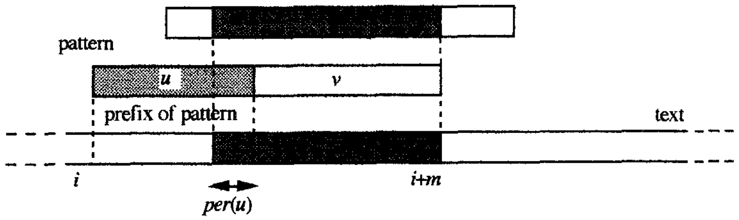
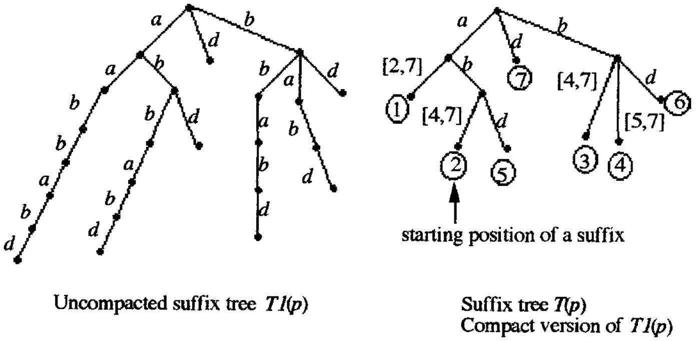


Fig. 6. We keep the prefix u of the pattern in memory. If u is periodic, then at most *per*(u) symbols of u are scanned again. however, this extra work is amortized by the next shift. The symbols of v are scanned for the first time.

possible solution is to preprocess the pattern in such a way that the value of *nextpref*(v) can be computed in constant time.

Technically, it is convenient to deal with suffixes. Denote $p' = p^R$. We look at the computation of *nextpref* from a "reverse" perspective. Let $nextsuf(v) = (nextpref(v))^R$. In other words, *nextsuf*(v) is the longest prefix of $v^R u^R$, which is a suffix of p', where u is the suffix of p' of length $m - |v|$.

In both approaches we present the set $FACT(p')$ by the suffix tree T. The edges of this tree are labeled by factors of the text represented by pairs (start-position, end-position). We take the compacted suffix tree for $p'\$$, in the sense of [Ap], then we cut off all edges labeled by \$. Afterward, each suffix of p' is represented by a node of T. Figure 7 shows an uncompacted suffix tree and a (compacted) suffix tree. The term "compacted" is omitted later. Call the factors of p', which correspond to nodes of T, the main factors. The tree T has only a linear number of nodes, hence, not all factors of p' are main. Nonmain factors correspond to a point on an edge of T. For a word v denote by *repr*(v) the node of T corresponding to the shortest word v' which is an extension of v (possibly $v = v'$). For example, the whole string p' is a main factor, and for $p' = aabbabd$ we have $repr(aa) = p'$.



Uncompacted suffix tree $TI(p)$

Suffix tree $T(p)$
Compact version of $TI(p)$

Fig. 7. The uncompacted tree and the suffix tree T for $p' = aabbabd$. Each factor corresponds to a node in the first tree. Call the factors corresponding to the nodes in the suffix tree *main* nodes. The representative *repr*(v) of the word v is the first descendant of v in the first uncompacted tree which is a node in the suffix tree.

The First Approach. Let P be the failure function of p, see [KMP]. The value $P(j)$ is the length of the longest proper suffix of $p[1..j]$ which is also a prefix of it (it is called a border). Assume that P is precomputed. Let $j = m - |v|$. Let $suf(k)$ denote the node corresponding to the suffix of size k. Then it is easy to prove the following fact:

$$|nextsuf(v)| = \mathrm{MAX}\{k/k = |v| + P^h(j) \text{ and } suf(k) \text{ is a descendant of } repr(v^R), \text{ for } h \geq 0\}.$$

If the set on the right side is empty, then $nextsuf(v) = nil$.

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 depth first search order. Then the nodes which are descendants of a given node form an interval of consecutively numbered nodes. Associate such an interval with each node. The question about descendants is then reduced to the inclusion of an integer in a known interval. This can be answered in a constant time. This gives the Turbo_RF′ algorithm presented above.

```
Algorithm Turbo_RF′
/* denote $t[i+j..i+m]$ by $x$, it is the last-scanned part of the text, we
   remember the last prefix $u$ of the pattern; initially $u$ is the empty
   word; */
$i := 0$; $u :=$ empty;
while $i \leq n - m$ do
{  $j := m$; while $j > |u|$ and $x \in FACT(p)$ 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 $v \in SUF(p)$ then report a match at position $i$;
      else  { compute MIN$\{k/k = |v| + P^h(j)$ and $suf(k)$ is a descendant
              of $repr(v^R)$, $h \geq 0\}$;
              if $k \neq nil$ then $shift := m - k$; else $shift := RF\_shift(v)$; }
   else $shift := RF\_shift(x)$;
   $i := i + shift$; $u :=$ prefix of pattern of length $m - shift$; }
end.
```

THEOREM 3. *The Turbo_RF′ algorithm finds all occurrences of a pattern of length m in a text of length n in $O(n)$ time. It makes at most n inspections of text symbols against the pattern. 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 considerably improve the complexity of the search phase of the algorithm. This increases the cost of the preprocessing phase which, however, remains linear. In the Turbo_RF′ algorithm, at a decision point, we sometimes have to spend 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 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 the main factors with their corresponding nodes. The computation is in a bottom-up manner on the tree T.

Case of a bottom node: v is a leaf.
$nextsuf(v) = nil$;
Case of an internal node:
assume v has sons $v_1, v_2, \ldots, v_q$, then a son v_j exists such that $nextsuf(v) = nextsuf(v_j)$ or, if v_j is a leaf, then $nextsuf(v) = v_j$.

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

The word v is a prefix of each of the candidates. What exactly does it mean for a word y, that has v as a prefix, 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 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 this situation in constant time.

It is enough to have a table *PREF* such that, for each position k in the pattern, the value $PREF[k]$ is the length of the longest prefix of p which starts at position k in p. This table can be easily computed in linear time as a side effect of the Knuth–Morris–Pratt string-matching algorithm. After the table is computed, we can check for each candidate in $O(1)$ time if it is good or not. For nodes which are not main, we set $nextpref(v) = y$, where $y = nextpref(repr(v^R))$, if y is a good candidate for v, i.e., $PREF[k] \geq |u| - k$, where $k = |u| - (|y| - |v|)$.

Hence, after preprocessing we keep a certain amount of additional data: the 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.

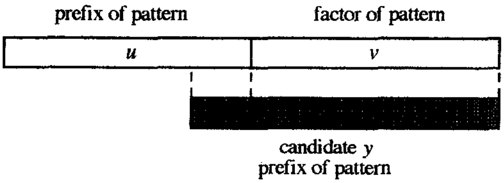


Fig. 8. The candidate y is *good* iff $PREF[k] \geq |u| - k$, where $k = |u| - (|y| - |v|)$.

THEOREM 4. *The pattern can be preprocessed in linear time in such a way that the computation of the RF_shift in the Turbo_RF algorithm can be accomplished in constant time whenever it is needed. The data strucure used in the preprocessing phase is read-only at the search phase. Only a constant read–write memory is used at search phase.*

Denote by Turbo_RF″ the version of the Turbo_RF algorithm in which the computation of the RF_shift is computed at decision points according to Theorem 4. The resulting Turbo_RF′ algorithm can be viewed as an automata-oriented string matching. We scan the text backward and change the state of the automaton. The shift is then specified by the state of the automaton where the scanning stops. Applying this idea directly gives a kind of Boyer–Moore automaton of polynomial (but not linear) size [L]. 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 r the size of the alphabet. Assume $r > 1$, and $\log m < 3 \cdot m/8$ (all logarithms are to base r). We consider the situation when the text is random. The probability of the occurrence of a specified letter on the ith position is $1/r$ and does not depend on letters on other positions.

THEOREM 5. *The expected time of the RF algorithm is $O(n \cdot \log(m)/m)$.*

PROOF. Let L_i be the length of shift in the ith iteration of the algorithm and let S_i be the length of substring of the pattern that is found in this iteration. Examine the first iteration of the algorithm. There are no more than m subwords of the pattern of length $2 \cdot \log m$ and there are $r^{2 \cdot \log m} = m^2$ possibilities of words that we read from the text which are all equally probable. Thus, with probability greater than (or equal to) $1 - 1/m$, $S_1 < 2 \cdot \log m$, so $L_1 > m - 2 \cdot \log m > m/4$. Let us call the ith shift *long* iff $L_i \geq m/4$ and *short* otherwise.

Divide computations into phases. Each phase ends on the first long shift. This means that there is exactly one long shift in each phase. It is obvious (by the definition of the long shift) that there are $O(n/m)$ phases in the algorithm. We now prove that an expected cost of each phase is $O(\log m)$.

CLAIM 1. *Assume that shifts i and $i + 1$ are both short. Then with probability more than $1/2$, the $(i + 2)$th shift is long.*

PROOF. If the ith and $(i + 1)$th shifts are both short, then the pattern is of the form $v(wv)^k sz$ where $k \geq 3$, $w, z \neq \varepsilon$, $|wv| = L_{i+1}$, and $|sz| = L_i$ (s may be equal to ε when $L_i < L_{i+1}$). Without loss of generality we can assume that wv is the minimal period of $v(wv)^k$ in such a sense that if there exists a word $w'v'$ such that $v(wv)^k = v'(w'v')^{k'}$ and $|w'v'| \leq |wv|$, then $w'v' = wv$. We can also assume (eventually changing wv and k) that $(wv)^k$ and sz do not have a common prefix. Now we have

$|wv| \leq L_{i+1}$ and $|sz| \leq L_i$. A suffix of the read part of the text is of the form $(wv)^k s$, and we have $A = \min(L_{i+1}, L_i)$ new symbols to read in the $(i + 2)$th iteration. Let y be a random word of length A to be read. Note that $|z| \leq |y|$. The question is what is the probability that $wvsy$ is a substring of the pattern. It is easy to see that if $wvsy$ is a substring of $v(wv)^k sz$, then y must be either equal to $pref(z)$ if $s \neq \varepsilon$, or $pref((wv)^l sz)$ otherwise, so there is at most A possibilities of such a y because the length of y is fixed. Thus the probability that reading A new symbols leads to a long (longer than $L_i + L_{i+1}$ which is less than $2 \cdot m/4$) substring of the pattern is not greater than $A/r^A \leq 1/2$. So, with probability $\geq 1/2$, $L_{i+2} = m - P_{i+2} \geq m - S_{i+2} \geq m/2 > m/4$. This completes the proof of the Claim 1. □

CLAIM 2. *The probability that the* kth *shift of the phase is short is* $\leq 1/2$ *for* $k \geq 3$.

PROOF. The assumptions say that the $(k - 1)$th and $(k - 2)$th shifts are also short, so by Claim 1 the kth shift is long with probability $\geq 1/2$. This completes the proof of Claim 2. □

We now end the proof of Theorem 5. Let X be the random variable which is the number of short shifts in the phase. What can we say about the distribution of X?

$$\begin{aligned}
&\Pr(X = 0) \geq 1 - 1/m,\\
&\Pr(X = 1) \leq 1/m,\\
&\Pr(X = 2) \leq 1/m,\\
&\Pr(X = 3) \leq 1/2 \cdot m,\\
&\quad\vdots\\
&\Pr(X = k) \leq 1/2^{k-2}m \qquad \text{for} \quad k \geq 2.
\end{aligned}$$

Let Y be the random variable which is a cost of the phase. Y is a function of the random variable X and $Y \leq 2 \cdot \log m + X \cdot m$.

$$\begin{aligned}
E(Y) &\leq 2 \cdot \log m \Pr(X = 0) + (2 \cdot \log m + m) \Pr(X = 1)\\
&\quad + \sum_{k=2} \Pr(X = k)(2 \cdot \log m + k \cdot m)\\
&\leq O(\log m) + 2 \cdot \log m/m \sum_{k=2} 1/2^{k-2} + \sum_{k=2} k/2^{k-2} = O(\log m).
\end{aligned}$$

This completes the proof of Theorem 5. □

5. Speeding up the Boyer–Moore Algorithm.

The linear-time complexity of the Boyer–Moore algorithm [BM] is quite nontrivial. The first proof of the linearity of the algorithm appeared in [KMP]. However, it needed more than a decade for full analysis. Cole has proved that the algorithm makes at most $3 \cdot n$ comparisons, see [Co], 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 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]) 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 the BM algorithm. Hence, all extra memory is of a constant size (two integers). The resulting Turbo_BM algorithm forgets all its history except the most recent one and its behavior again has a "mysterious" character. Despite that, the complexity is improved and the analysis is simple.

The main feature of the Turbo_BM algorithm is that during the search of the pattern, one factor of the pattern that matches the text at the current position is memorized (this factor can be empty). This has two advantages:

—It can lead to a jump over the memorized factor during the scanning phase.
—It allows the execution of what we call a *Turbo_shift*.

We now explain what a *Turbo_shift* is. Let x (**match**) be the longest suffix of p that matches the text at a given position. Also, let y (**memory**) be the memorized factor that matches the text at the same position. We assume that x and y do not overlap (for some nonempty word z, yzx is a suffix of p). For different letters a and b, ax is a suffix of p aligned with bx in the text (see Figure 9). The only interesting situation is when y is nonempty, which occurs only immediately after a BM_shift (a shift determined by the BM_shift function). Let *shift* be its length. A *Turbo_shift* can occur when x is shorter than y. In this situation ax is a suffix of y. Thus a and b occur at distance *shift* in the text. However, suffix yzx of p has period *shift* (by definition of the BM_shift), and thus it cannot overlap both occurrences of letters a and b in the text. As a consequence, the smallest valid shift of the pattern is $|y| - |x|$, which we call a *Turbo_shift*.

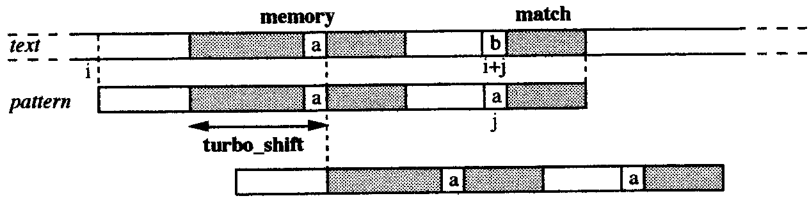


Fig. 9. *Turbo_shift* is *memory-match*. Distinct letters a and b in text are at distance h, and h is a period of the right part of pattern.

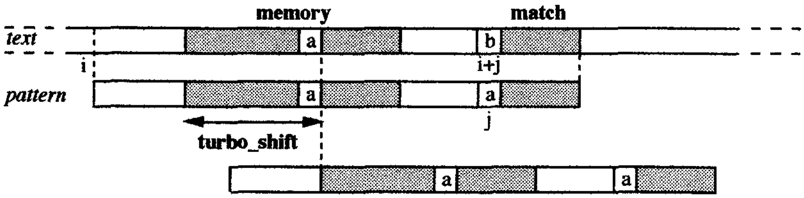


Fig. 10. *Turbo_shift* is *memory-match*. Distinct letters *a* and *b* in text are at distance *h*, and *h* is a period of the right part of pattern.

In the Turbo_BM algorithm the length of the shift is

$$shift = \max(BM\_shift(x),\ Turbo\_shift).$$

From the analysis of the BM algorithm itself, and the above computation of shifts, it is straightforward to derive a correctness proof of the Turbo_BM algorithm.

In the Turbo_BM algorithm below, the reader can note that, in case a BM_shift does not apply, the length of the actual shift is made greater than the length of the matched suffix x. The proof of correctness of this fact is similar to the above argument and is explained in Figure 10.

```
Algorithm Turbo_BM   /* reversed-suffix string matching */
$i := 0$; $memory := 0$;
while $i \le n - m$ do
{  $j := m$;
   while $j > 0$ and $x$ in $SUF(p)$ do
      if $memory \ne 0$ and $j = m - shift$ then $j := j - memory$   /* jump */
         else $j := j - 1$;
   if $x = p$ then report match at position $i$;
   $Turbo\_shift := memory - match$;   /* $match = |x| - 1$ */
   if $Turbo\_shift > BM\_shift[j]$ then
      { $i := i + \max(Turbo\_shift, match + 1)$; $memory := 0$; }
   else
      { $i := i + BM\_shift[j]$; $memory := \min(m - shift, match)$; }
}
end.
```

Theorem 6. *The search phase of the Turbo_BM algorithm is linear. It makes at most $2 \cdot n$ comparisons.*

Proof. We decompose the search 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 (in the case suf_k is not the p itself). We also call $shift_k$ the length of the shift done at stage k.

Consider three types of stages according to the nature of the scan and of the shift. We say that the shift at stage k is short if $2 \cdot shift_k < suf_k + 1$. The three types are:

(i) A stage followed by a stage with jump.
(ii) A stage with long shift, not followed by a stage with jump.
(iii) A stage with short shift, not followed by a stage with jump.

The idea of the proof is to amortize comparisons with shifts. We define $cost_k$ as follows:

—If stage k is of type (i), $cost_k = 1$.
—If stage k is of type (ii) or (iii), $cost_k = suf_k + 1$.

In the case of a type (i) stage, the cost corresponds to the only mismatch comparison. Other comparisons done during the same stage are reported to the cost of next stages. The total number of comparisons executed by the algorithm is the sum of the costs. We want to prove $\Sigma costs < 2 \cdot \Sigma shifts$. In the second Σ the length of the last shift is replaced by m. Even with this assumption, we have $\Sigma shifts \leq |t|$, and, if the above inequality holds, so is the result, $\Sigma costs < 2 \cdot |t|$.

For stage k of type (i), $cost_k$ $(= 1)$ is trivially less than $2 \cdot shift_k$, because $shift_k > 0$. For stage k of type (ii), $cost_k = suf_k + 1 \leq 2 \cdot shift_k$, by definition of long shifts.

It remains to consider stages of type (iii). Since in this situation we have $shift_k < suf_k$, the only possibility is that a BM_shift is applied at stage k. Memory is then set up. At the next sgage, $k + 1$, the memory is not empty, which leads to a potential turbo-shift. The situation at stage $k + 1$ is the general situation when a turbo-shift is possible (see Figure 11). Before continuing the proof, we first consider two cases and establish inequalities (on the cost of stage k) that are used later.

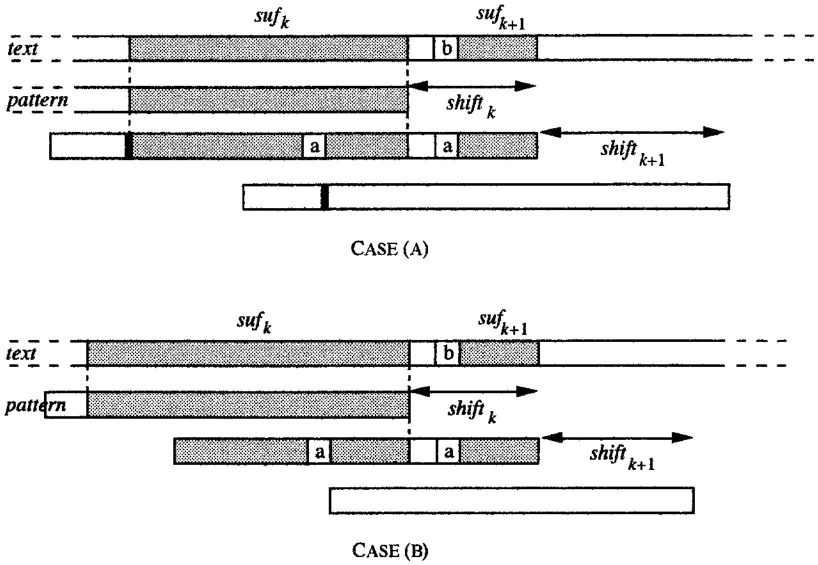
Case (a): $suf_k + shift_k \leq |p|$. By definition of the turbo-shift, we have $suf_k - suf_{k+1} \leq shift_{k+1}$. Thus,

$$cost_k = suf_k + 1 \leq suf_{k+1} + shift_{k+1} + 1 \leq shift_k + shift_{k+1}.$$

Case (b): $suf_k + shift_k > |p|$. By definition of the turbo-shift, we have $suf_{k+1} + shift_k + shift_{k+1} \geq m$. Then

$$cost_k \leq m \leq 2 \cdot shift_k - 1 + shift_{k+1}.$$

We can consider that at stage $k + 1$ Case (b) occurs, because this gives the higher bound on $cost_k$ (this is true if $shift_k \geq 2$; the case $shift_k = 1$ can be treated directly). If stage $k + 1$ is of type (i), then $cost_{k+1} = 1$, and then $cost_k + cost_{k+1} \leq 2 \cdot shift_k + shift_{k+1}$, an even better bound than expected. If at stage $k + 1$ we have $suf_{k+1} \leq shift_{k+1}$, then we get what was expected: $cost_k + cost_{k+1} \leq 2 \cdot shift_k + 2 \cdot shift_{k+1}$.



CASE (A)

CASE (B)

Fig. 11. Costs of stages k and $k + 1$ correspond to the shadowed areas plus mismatches. If $shift_k$ is small, then $shift_k + shift_{k+1}$ is big enough to amortize the costs partially.

The last situation to consider is when, at stage $k + 1$, we have $suf_{k+1} > shift_{k+1}$. This means, as previously mentioned, that a BM-shift is applied at stage $k + 1$. Thus, the above analysis also applies at stage $k + 1$, and, since only Case (a) can occur then, we get $cost_{k+1} \leq shift_{k+1} + shift_{k+2}$. We finally get $cost_k + cost_{k+1} \leq 2 \cdot shift_k + 2 \cdot shift_{k+1} + shift_{k+2}$.

The last argument proves the first step of an induction: if all stages k to $k + j$ are such that $suf_k > shift_k, \ldots, suf_{k+j} > shift_{k+j}$, then

$$cost_k + \cdots + cost_{k+j} \leq 2 \cdot shift_k + \cdots + 2 \cdot shift_{k+j} + shift_{k+j+1}.$$

Let k' be the first stage after stage k such that $suf_{k'} \leq shift_{k'}$. Integer k' exists because the contrary would produce an infinte sequence of shifts with decreasing lengths. We then get

$$cost_k + \cdots + cost_{k'} \leq 2 \cdot shift_k + \cdots + 2 \cdot shift_{k'}.$$

Which shows that $\Sigma cost_k \leq 2 \cdot \Sigma shift_k$, as expected. □

REMARK (On the Additional Application of Occurrence Shifts in the Turbo_BM Algorithm). In the Turbo_BM algorithm we deal only with match shifts of the BM algorithm. If the alphabet is binary, then occurrence shifts in the BM algorithm are useless. Generally, for small alphabets the occurrence heuristics have little effect. For bigger alphabets, we can include the occurrence shifts in the algorithm. The version of Turbo_BM including occurrence shifts is given below. In case an

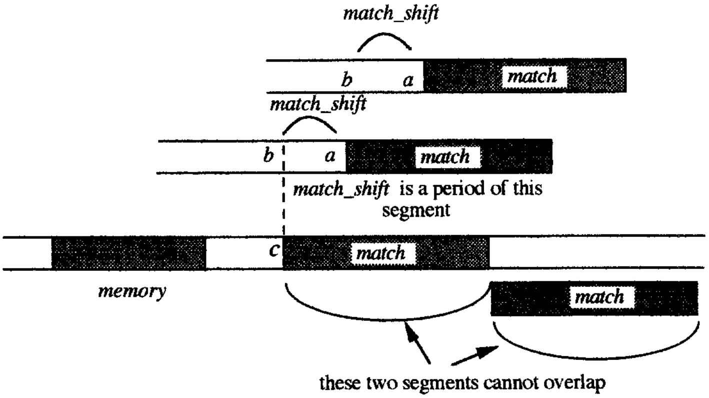

match_shift
b
a
match
match_shift
b
a
match
match_shift is a period of this segment
c
match
memory
match
these two segments cannot overlap

Fig. 12. If *occ_shift* > *match_shift* or if *Turbo_shift* > *match_shift*, then there is a symbol b in the pattern such that $a \neq b$. We then have an additional *Turbo_shift* of size *add-shift* = |*match*| = $|x| - 1$.

occurrence shift is possible, the length of the shift is made greater than *match*. This is similar to the case where a turbo-shift applies. The proof of correctness is also similar, and is explained in Figure 12.

The match shift of Boyer–Moore is a period of the segment *match* of the text (Figure 12). At the same time we have two distinct symbols a, b whose distance is the period of the segment. Hence we know that the shift is at least *match* + 1. This shows that Theorem 6 is still valid for the Turbo_BM algorithm with occurrence shifts.

```
Algorith Turbo_BM with occurrence shifts   /* reversed-suffix string matching */
$i := 0$; $memory := 0$;
while $i \le n - m$ do
{  $j := m$;
   while $j > 0$ and $x$ in $SUF(p)$ do
      if $memory \neq 0$ and $j = m - shift$ then $j := j - memory$   /* jump */
      else $j := j - 1$;
   if $x = p$ then report match at position $i$;
   $Turbo\_shift := memory\text{-}match$;   /* $match = |x| - 1$ */
   $shift := \max(BM\_shift[j], BM\text{-}occ\text{-}shift[j], Turbo\_shift)$;
   if $shift > BM\_shift$ then
      { $i := i + \max(shift, match + 1)$; $memory := 0$; }
   else
      { $i := shift$; $memory := \min(m - shift, match)$; }
}
end.
```

THEOREM 7. *The Turbo_BM algorithm with occurrence shifts makes at most* $2 \cdot n$ *comparisons.*

Table 1. Experiments on a binary alphabet.

m	BM	TRF
2	1.0014	0.9178
3	0.9728	0.8528
4	0.9236	0.8055
5	0.8589	0.7491
6	0.8002	0.6936
7	0.745	0.6397
8	0.6989	0.5901
9	0.6594	0.5446
10	0.6261	0.5049
20	0.4446	0.2932
30	0.3867	0.2142
40	0.35	0.168
50	0.3228	0.1403
60	0.2977	0.121
70	0.2781	0.1074
80	0.2652	0.0969
90	0.2587	0.0871
100	0.2481	0.0801

6. Experimental Results. In this section we present experiments done on the Turbo_RF algorithm versus the BM algorithm. This shows the good behavior of the Turbo_RF algorithm, that seems faster than the fastest practical algorithm.

Experiments to count the number of inspections are done on characters of the text. We have implemented both algorithms in C language. We use the binary alphabet $\{0, 1\}$. The text, randomly built on this alphabet, has length 15,000 (49.59% of 0 and 50.41% of 1). We search for all the patterns of length from 2 to 7. For longer patterns we search for 100 different patterns randomly chosen. The values given in Table 1 are the average numbers of inspections, c, for all symbols of the text, and all considered patterns. Table 1 is translated into Figures 13 and 14.

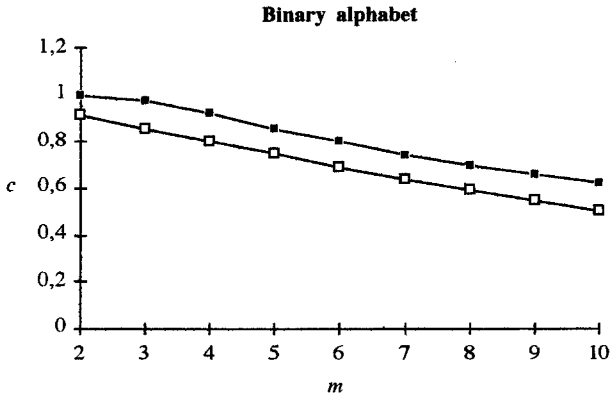


Fig. 13. Experiments for short patterns (■, the BM algorithm; □, the Turbo_RF algorithm).

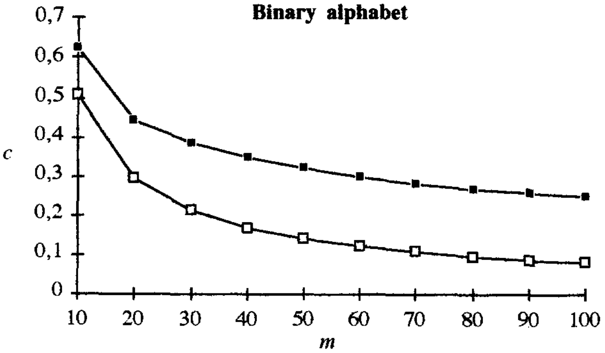


Fig. 14. Experiments for long patterns. (■, the BM algorithm; □, the Turbo_RF algorithm).

These results show that, for a binary alphabet, the Turbo_RF algorithm is always better than the BM algorithm. It even achieves a score less than $\frac{1}{10}$ for patterns longer than 80 characters. These experimental results confirm the theoretical properties of the Turbo_RF algorithm.

The Remark in Section 5 and the Appendix both show flexibility of the RF and Turbo_BM algorithms. Many changes and improvements are possible, probably the most interesting is the possibility of the extension for multipattern matching. We believe that the RF algorithm is a practical algorithm. It is usually extremely fast. The constant coefficient is small. A reasonable number of computer experiments performed so far support the claim.

Appendix. Computation of Periods in the Turbo_RF Algorithm. We show how the period of the prefix u of the pattern can be computed dynamically in the Turbo_RF algorithm. This means that the table of periods for all prefixes of the pattern need not be precomputed nor stored through the algorithm.

The general situation is as follows: We have to recognize a prefix u of the text and we are scanning the portion v of the text which consists of $m - |u|$ characters at the right of u. Let $u = (u_1u_2)^r u_1$ with $|u_1u_2| = per(u)$.

Case 1: A mismatch appears when scanning for u_2u_1v. In this situation the period of the new prefix can be computed with the difference of the two last positions where the pattern can start (see Figure 15).

Case 2: No mismatch appears when scanning for u_2u_1v.

Case 2.1: $|v| \leq per(u)$. In this situation the period of the new prefix is equal to $per(u)$ because this new prefix is also a prefix of u greater than $per(u)$ (see Figure 16).

Case 2.2: $|v| > per(u)$. Let L be the length of the path between the start state and the last final state encountered when scanning for u_2u_1v in the automaton.

Case 2.2.1: $L > |v|$. In this case L must be equal to $|v| + |u_1|$ because $L > |v|$ means that the beginning of the pattern is in u_2u_1 and by the minimality of

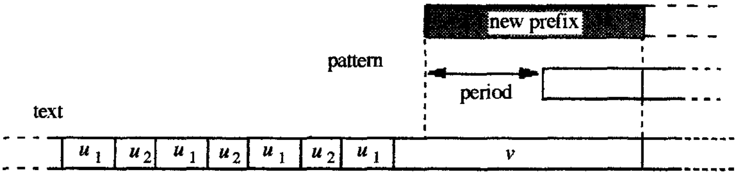


Fig. 15. A mismatch occurs while scanning u_2u_1v.

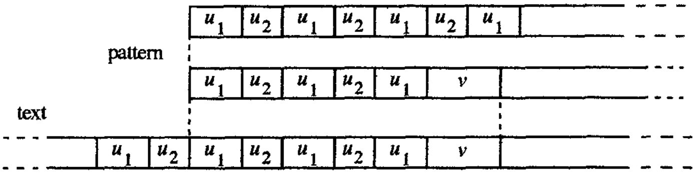

u_1 u_2 u_1 u_2 u_1 u_2 u_1
pattern
u_1 u_2 u_1 u_2 u_1 v
text
u_1 u_2 u_1 u_2 u_1 u_2 u_1 v

Fig. 16. $|v| \leq per(u)$.

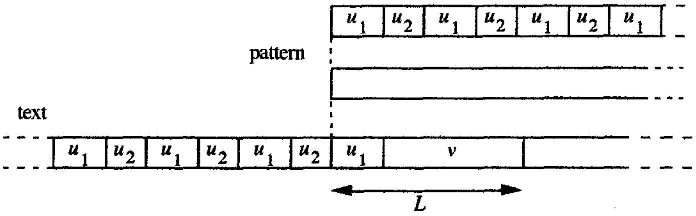

u_1 u_2 u_1 u_2 u_1 u_2 u_1
pattern
text
u_1 u_2 u_1 u_2 u_1 u_2 u_1 v
L

Fig. 17. $L > |v|$.

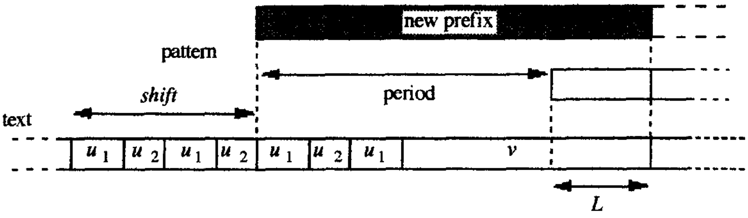


Fig. 18. $L < |v|$.

per$(u) = |u_2 u_1|$ the beginning of the pattern must match exactly $u_1 v$. Hence the period of the new prefix must be equal to *per*(u) (see Figure 17).

Case 2.2.2: $L < |v|$. In this situation the period of the new prefix is equal to *m-shift-L* (see Figure 18).

References

[Ah] A. V. Aho, Algorithms for finding patterns in strings, in *Handbook of Theoretical Computer Science*, vol. A (J. van Leeuwen, ed.), Elsevier, Amsterdam, 1990, pp. 255–300.

[Ap] A. Apostolico, The myriad virtues of suffix trees, in *Combinatorial Algorithms on Words* (A. Apostolico and Z. Galil, eds.), NATO Advanced Science Institutes, Series F, vol. 12, Springer-Verlag, Berlin, 1985, pp. 85–96.

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

[BR] R. A. Baeza-Yates and M. Régnier, Average running time of the Boyer–Moore–Horspool algorithm, *Theoret. Comput. Sci.* **92**(1) (1992), 19–31.

[BKR] L. Banachowski, A. Kreczmar, and W. Rytter, *Analysis of Algorithms and Data Structures*, Addison-Wesley, Reading, MA, 1991.

[BBE$^+$] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. T. Chen, and J. Seiferas, The smallest automaton recognizing the subwords of a text, *Theoret. Comput. Sci.* **40** (1985), 31–55.

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

[Co] R. Cole, Tight bounds on the complexity of the Boyer–Moore pattern matching algorithm, *Proceedings of the 2nd Annual ACM Symposium on Discrete Algorithms*, 1990, pp. 224–233.

[Cr] M. Crochemore, Transducers and repetitions, *Theoret. Comput. Sci.* **45** (1986), 63–86.

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

[GO] L. J. Guibas and A. M. Odlyzko, A new proof of the linearity of the Boyer–Moore string searching algorithm, *SIAM J. Comput.* **9** (1980), 672–682.

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

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

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

[L] T. Lecroq, A variation on Boyer–Moore algorithm, *Theoret. Comput. Sci.* **92** (1992), 119–144.

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

[Y] A. C. Yao, The complexity of pattern matching for a random string, *SIAM J. Comput.* **8** (1979), 368–387.