Edgar H. Sibley
Panel Editor

By reducing an array matching problem to a string matching problem in a natural way, it is shown that efficient string matching algorithms can be applied to arrays, assuming that a linear preprocessing is made on the text.

A Technique for Two-Dimensional Pattern Matching

Rui Feng Zhu and Tadao Takaoka

Three major efficient pattern matching algorithms were proposed in the past decade, one by Knuth, Morris, and Pratt (KMP algorithm) [8], one by Boyer and Moore (BM algorithm) [5], and one by Rabin and Karp (RK algorithm) [6].

A natural generalization of the pattern matching problem is the two-dimensional pattern matching problem where the pattern is a two-dimensional array of characters `PATTERN[1 .. P1, 1 .. P2]` and the text is a second array `TEXT[1 .. N1, 1 .. N2]`. In this case the problem is to determine where, if anywhere, the pattern occurs as a subarray of the text.

In this article we first present an efficient pattern matching algorithm for the two-dimensional case, one which is a combination of the KMP and RK algorithms. Assuming, in the worst case, that the RK algorithm runs linearly for one-dimensional array matching, this algorithm takes time proportional to $O(N1 * N2 + P1 * P2)$ in the worst and average cases, which is clearly optimal since both the pattern and the text have to be read and this takes $O(N1 * N2 + P1 * P2)$ time. The algorithm needs $O(N1 * N2 + P1 * P2)$ space, and also works on-line (assuming the text is read row by row). Second, we describe another approach, which takes $O(N1 * N2 * \log(P2)/P2 + P1 * P2)$ time on average and also runs faster when the row length of the pattern increases.

Computer experiments show that for various pattern sizes the average cost for either of our algorithms is much less than that of the algorithm proposed by Bird (B algorithm) [4].

THE KMP ALGORITHM

The KMP algorithm [8] solves the problem by comparing the characters of the pattern with those of the text from the left end of the pattern. To illustrate this approach, consider an example, searching for `pattern = 'ababb'` in `text = 'babbababacababb'`.

```
pattern:  ababb
text:     babbababacababb
```

Since the text pointer points to the character `b`, which does not match `'a'` in the pattern, the pattern is shifted one place to the right and the next text character is inspected.

```
pattern:   ababb
text:     babbababacababb
```

There is a match, so the pattern remains where it is while the next several characters are scanned until we come to a mismatch.

```
pattern:   ababb
text:     babbababacababb
```

The first two pattern characters are matched, but there is a mismatch on the third. Thus it is known that the last three characters of the text have been `ab?`, where `'?'` is not equal to `'a'`. Next the pattern can be shifted three more places to the right, since `'?'` is not equal to `'a'` and a shift of one or two positions could not lead to a match. Thus the situation appears as:

```
pattern:      ababb
text:     babbababacababb
```

The characters are scanned again and another partial match occurs. A mismatch occurs on the fifth pattern character.

```
pattern:     ababb
text:    babbababacababb
```

Similiarly we know that the last five text characters were abab?, where ? is not equal to 'b' but might be equal to 'a'. Thus the pattern should be shifted two places to the right.

```
pattern:         ababb
text:      babbababacababb
```

Comparing the two characters we get a mismatch, so we shift the pattern further four places.

```
pattern:             ababb
text:      babbababacababb
```

Scanning the following characters, the full pattern is found. The pattern match has succeeded.

```
pattern:             ababb
text:      babbababacababb
```

This example shows that by knowing the characters in the pattern and the position where a mismatch occurs, it can be determined where in the pattern to continue to search for a match without moving backward in the text. It also tells us that the process of pattern matching will run much more efficiently if we have an auxiliary table which will provide us with the information of how far to slide the pattern when a mismatch has been detected at a certain position in the pattern. The following algorithm summarizes the behavior of the KMP algorithm.

```
ALGORITHM 1: KMP Algorithm.
INPUT.
  The text(pattern) is stored in the
  array TEXT[1 .. N] (PT[1 .. M]), h is
  the auxiliary function, p and q are
  pointers of pattern and text,
  respectively.
OUTPUT.
  Location at which the pattern is
  found.
METHOD.
  begin
    compute-h;
    p := 1; q := 1;
    while (p ≤ M) and (q ≤ N) do
    begin
      while (p > 0) and (PT[p]
      ≠ TEXT[q]) do p := h[p];
        p := p + 1; q := q + 1
    end;
    if p ≤ M then writeln('not found')
        else writeln
          ('The left end of the pattern
            is found at', q - p + 1)
  end.
procedure compute-h;
begin
  t := 0; h[1] := 0;
  for p := 2 to M do
    begin
    while (t > 0) and (PT[p - 1]
    ≠ PT[t]) do t := h[t];
    t := t + 1;
    if PT[p] ≠ PT[t] then h[p] := t
    else h[p] := h[t]
  end
end.
Note:
  The definition of the h function is:
    h[p] = max{s | (s = 0)
    or ((PT[1 .. s - 1] = PT[p - s
       + 1 .. p - 1]
    and PT[s] ≠ PT[p] and (1 ≤ s < p))},
      h[1] = 0.
```

The value of $h[p]$ gives the position at which matching starts again after a mismatch occurs. The detailed reasoning of this computation of h is given in [8]. For our example of PT = 'ababb' h is computed as (0, 1, 0, 1, 3).

THE RK ALGORITHM

The RK algorithm [6] solves the problem by treating each possible M-character section (where M is the pattern length) of the text as a key in a hash table, then computing the hash function of it and checking to see if it is equal to the hash function of the pattern. Here the hash function is defined as follows: $h(k) = k \bmod q$, where q is a large prime. The use of such a large value of q makes it extremely unlikely that a collision will occur. We translate the M-character to numbers by packing them together in a computer word, which we then treat as an integer—k in the above. This corresponds to writing the characters as numbers in a radix-d number system, where d is the number of possible characters. The number k corresponding to the M-character section TEXT[i] ... TEXT[i + M - 1] is:

$$k = \text{ord(TEXT[i])} * d^{M-1} + \text{ord(TEXT[i + 1])} * d^{M-2} + \ldots + \text{ord(TEXT[i + M - 1])}$$

where ord(x) is the order of character x. Shifting one position to the right in the text simply corresponds to replacing k by

$$(k - \text{ord(TEXT[i])} * d^{M-1}) * d + \text{ord(TEXT[i + M])}.$$

This leads to the very simple pattern matching algorithm implemented below.

```
ALGORITHM 2: RK Algorithm.
INPUT:
  The text(pattern) is stored in the
  array TEXT[1 .. N] (PT[1 .. M]); dm is
  used as a constant equal to d^(M-1).
```

```
OUTPUT:
  Location in the text which has the
  same hash value as the pattern.
METHOD:
  begin
    dm := 1; for i := 1 to M - 1
      do dm := (d * dm) mod q;
    h1 := 0; for i := 1 to M
      do h1 := (h1 * d + ord(PT[i]))
                mod q;
    h2 := 0; for i := 1 to M
      do h2 := (h2 * d + ord(TEXT[i]))
                mod q;
    i := 1;
    while (h1 ≠ h2) and (i ≤ N - M) do
      begin
        h2 := (h2 + d * q - ord(TEXT[i])
              * dm) mod q;
        h2 := (h2 * d + ord(TEXT[i + M]))
              mod q;
        i := i + 1;
      end;
    if h1 = h2 then writeln('found at',
                i)
                else writeln ('not
                found')
  end.
```

The algorithm only finds a position in the text which has the same hash value as the pattern. To be sure, we should do a direct comparison of characters in the text with those in the pattern. Obviously, this algorithm takes time proportional to $M + N$, assuming that we can ignore the probability of collision, that is, the case where two different strings have the same hash values.

THE ALGORITHM

The general scheme of our algorithm is to use the hash function method proposed in the RK algorithm vertically. We first translate the two-dimensional arrays of characters of TEXT and PATTERN to one-dimensional numbers of TEXT' and PATTERN', respectively; we then search TEXT' for the occurrences of the PATTERN' using the KMP technique row by row. The algorithm is as follows:

```
ALGORITHM 3.
INPUT:
  The text and pattern are stored in
  the TEXT[1 .. N1, 1 .. N2] and PATTERN
  [1 .. P1, 1 .. P2], respectively.
OUTPUT:
  Location at which pattern occurs.
METHOD:
   1. begin
   2.    texthash; row := P1;
   3.    dm := 1; for j := 1 to P1 - 1
           do dm := (d * dm) mod q;
   4.    read(PATTERN);
   5.    pathash;
   6.    repeat KMP' (PATTERN',
           TEXT', found, column);
   7.      if (found = false) and (row
           < N1) then change(row);
   8.    row := row + 1;
   9.    until (found = true) or (row
         > N1);
  10.    if found = true
  11.      then writeln
             ('The beginning of the
             pattern is found at', row -
             P1 + 1, column - P2 + 1)
  12.      else writeln
             ('not found')
  13. end.

procedure pathash;
begin
for j := 1 to P2 do
  begin
    PATTERN'[j] := 0;
    for i := 1 to P1 do
    PATTERN'[j] := (PATTERN' [j] * d
      + ord(PATTERN[i, j])) mod q
  end
end;

procedure texthash;
begin
for j := 1 to N2 do
  begin
    TEXT'[j] := 0;
    for i := 1 to P1 do
    TEXT'[j] := (TEXT'[j] * d + ord
    (TEXT[i, j])) mod q
  end
end;
```

In algorithm 3 we first compute two vectors of PATTERN' and TEXT' from PATTERN and TEXT. For example, suppose PATTERN is:

1	2	3	4	5	6	7	8
a	b	c	d	e	a	p	a
a	a	a	a	a	a	a	a
a	c	a	c	c	c	c	m
e	n	k	t	r	e	f	w
s	n	t	o	b	h	t	e

FIGURE 1. An Example of *PATTERN*

Assume that ord('a') = 0, d = 32, q = 33,554,393, and the distance between two adjacent alphabetic characters is 1, so PATTERN' is:

1	2	3	4	5	6	7	8
146	105086	2097491	3148398	196897	2183	15730867	12996

FIGURE 2. An Example of *PATTERN'*

And the TEXT' can then also be computed in the same way.

Second, the KMP' procedure is used to match PATTERN' with TEXT'. If the matching process fails at "row-th" row, that is, found = false, another procedure change(row) is used to change the value of TEXT' from row to row + 1. The change(row) procedure is obviously optimal since shifting one row down in the TEXT simply corresponds to replacing TEXT'[j] by:

```
TEXT'[j] - ord(TEXT[row - P1 + 1, j])
                * dm)) * d
         + ord(TEXT[row + 1, j]).
```

Such operations are executed repeatedly, until found = true or row > N1. Procedure change(row) is as follows:

```
procedure change(row);
begin
for j := 1 to N2 do
TEXT'[j]
  := (TEXT'[j] + d * q - ord(TEXT[row
     - P1 + 1, j]) * dm) * d
     + ord(TEXT[row + 1, j]) mod q
end;
```

The KMP' procedure is the same as the KMP algorithm except that it returns the truth value and position in "found" and "column" if the pattern is found and Procedure directcompare(found, row, column) is inserted at the end of the KMP algorithm.

```
procedure directcompare(found, row,
column);
begin
  found := true; i := 1; j := 1;
  while (i ≤ P1) and (found = true) do
    begin
      while (j ≤ P2) and (found = true)
      do
        begin
        if PATTERN[i, j]
          ≠ TEXT[row - P1 + i,
                 column - P2 + j]
           then found := false;
        j := j + 1;
        end;
        i := i + 1
    end
end;
```

Notice that procedure KMP' only works because the KMP algorithm is independent of the size of the character alphabet, which can consequently be arbitrarily large.

The pattern matching algorithms are mainly used for text editing. From the user's point of view, the response time is from the time the pattern is input to the time the pattern is found. To do the job of change(row) in advance, we can modify our algorithm in the following way.

First, we expand TEXT' to an array [P1 .. N1, 1 .. N2] and replace procedure texthash with the following code:

```
procedure texthash';
begin
  for j := 1 to N2 do
  begin
    TEXT'[P1, j] := 0;
  for i := 1 to P1 do
  TEXT'[P1, j] := (TEXT'[P1, j] * d
    + ord(TEXT[i, j])) mod q;
  end;
  for i := P1 + 1 to N1 do
  for j := 1 to N2 do
  TEXT'[i, j] := (TEXT'[i - 1, j] + d * q
    - ord(TEXT[i - P1]) * dm) * d
    + ord(TEXT[i, j]) mod q
end;
```

Then the seventh line and procedure change(row) of algorithm 3 are deleted. We measure the running time after the pattern is read, that is, from the fifth line of algorithm 3, and the B algorithm [4] is also measured in the same way. (The computation of TEXT' is considered to be preprocessing.) The results are shown in the first and second columns of Table I and indicate that for various sizes of patterns the average cost of our algorithm is about one-seventh of that of the B algorithm. The summary of algorithm B is listed in Appendix A. The advantages of this algorithm are:

1. Time complexity is $O(N1 * N2 + P1 * P2)$ for either of the worst case and average case since the number of multiplications during the preprocessing stage for the hash function calculation is $O(N1 * N2 + P1 * P2)$ and the largest number of comparisons made in KMP' procedure is $O((N1 - P1) * N2)$.
2. The actual running time is much less than the B algorithm. We measure running time after the pattern is read.
3. Space complexity is $O(N1 * N2 + P1 * P2)$, including the auxiliary array TEXT'.
4. The algorithm is simpler than other algorithms proposed in the past [2, 3].
5. Its on-line nature, that is, the input is scanned only once, and after scanning the character at any position of the input, before scanning further, it is possible to answer YES or NO to whether the pattern matches at that position.

Another Approach

In this approach, as the strategy proposed in the BM algorithm [5] is used, we first present the BM algorithm in the following section.

TABLE I. Average Costs of the Three Algorithms

Pattern size	Average Cost		
	B Algorithm	Algorithm 3	Algorithm 4
[1 . . 5, 1 . . 5]	41774	4755	14860
[1 . . 10, 1 . . 10]	32910	6730	9060
[1 . . 20, 1 . . 20]	38020	6390	6460
[1 . . 30, 1 . . 30]	38490	6157	5460
[1 . . 40, 1 . . 40]	40730	6140	4750
[1 . . 50, 1 . . 50]	39952	5670	4390
[1 . . 60, 1 . . 60]	41260	5280	3810
[1 . . 70, 1 . . 70]	41910	5360	3810
[1 . . 80, 1 . . 80]	43520	5120	3430
[1 . . 90, 1 . . 90]	43650	4780	3380
[1 . . 100, 1 . . 100]	42910	4500	3310

The size of text is [1 . . 1000, 1 . . 1000]. Text and pattern are derived from random text composed of 24 characters. Costs are measured by computing time in milliseconds.

The BM Algorithm

Unlike the KMP algorithm and the straightforward algorithm, the BM algorithm compares the pattern with the text from the right end of the pattern. The performance of this algorithm is quite good in the average case, where it performs in $O(N/M)$ time. Here N is the length of text, while M is the length of pattern. Several variations were proposed to improve the number of character comparisons in the worst case of the BM algorithm. They are: Knuth [8] which achieved linear time in the worst case but lost the linear time in the preprocessing; Galil [7] which made the worst cost bounded by $14n$; and recently published by Apostolico and Giancarlo [2] which had the worst case bounded by $2n$. The authors have presented a method [9] to improve the average performance of the BM algorithm.

The BM algorithm solves the pattern matching problem by repeatedly positioning the pattern over the text and attempting to match it. For each positioning that occurs, the algorithm starts matching the pattern against the text from the right end of the pattern. If no mismatch occurs, then the pattern has been found, otherwise the algorithm takes a shift that is an amount by which the pattern will be moved to the right before a new matching attempt is undertaken.

```
ALGORITHM 4: BM Algorithm.
INPUT:
  The text(pattern) is stored in the
  array TEXT[1 .. N] (PATTERN[1 .. M]);
  p(q) points to the current characters
  of the PATTERN(TEXT); D[ch] and DD[i]
  are the auxiliary shift functions.
OUTPUT:
  Location at which the pattern is
  found.
METHOD:
  begin
    q := M;
    repeat p := M;
      while (p > 0) and (PATTERN[p]
        = TEXT[q] do
        begin p := p - 1; q := q - 1 end;
      if p ≠ 0 then q := q
         + max(D[TEXT[q]], DD[p])
    until (p = 0) or (q > N)
  end.
Note:
  The definitions of the D shift
  function and the DD shift function
  are:
  D[ch] = min{s | s = M
    or (0 < s < M and PATTERN[M - s]
      = ch)};
  DD[p] = min{s + M - p | (s ≥ 1) and
  (s ≥ p
    or PATTERN[p - s] ≠ PATTERN[p]
    and (s ≥ i or PATTERN[i - s]
  = PATTERN[i], p < i ≤ M)}.
```

Here D and DD are often referred to as shift functions. The DD shift function is based on the idea that when the pattern is moved right, it has to (1) match over all the characters previously matched, and (2) bring a different character over the character of the text that caused the mismatch. Figure 3 illustrates the definition of the DD shift function. The D shift function uses the fact that we must bring over ch = text[q] (the character that caused mismatch), the first occurrence of 'ch' in the pattern viewed from the right end. The definition of the D shift function is shown in Figure 4. Both shift functions can be obtained from precomputed tables based solely on the pattern and the alphabet used. The DD shift function requires a table of length equal to the pattern, while the D shift function requires a table of size equal to the alphabet size. Given the two values of the two shift functions, the BM algorithm chooses the larger one. For reasons of space the details regarding computations of the D shift function and the DD shift functions are omitted. (Details are given in [5] and [8].)

The Algorithm

Examining the example in Figure 2, we find that the values of PATTERN' and TEXT' vary over a great range. This means that for most cases the comparisons between PATTERN' and TEXT' are mismatches. Based on this observation, we modify our algorithm in the following way. First, using any efficient sorting method, we sort PATTERN' and store the sorted PATTERN' and its indices value into PH1[1 .. 2, 1 .. P2]. The PH1 values of the example in Figure 2 are given in Figure 5.

Second, incorporating the DD shift function of the BM algorithm into our situation, we replace procedure KMP' of algorithm 3 by procedure SEARCH in which

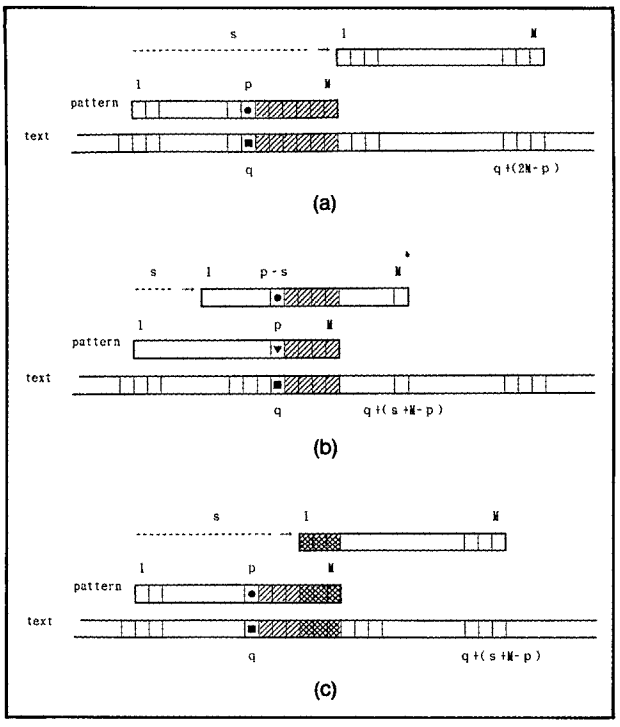

FIGURE 3. The Definition of *DD Shift* Function. (a) $s = M$, (b) $s < p$, (c) $s \geq p$

we begin comparison at the end of PATTERN'. The algorithm is summarized as follows.

```
ALGORITHM 5.
INPUT:
  The text and pattern are stored in
  the TEXT[1 .. N1, 1 .. N2] and
  PATTERN[1 .. P1, 1 .. P2],
  respectively.
OUTPUT:
  Location at which pattern occurs.
METHOD:
   1. begin
   2.    texthash;
   3.    for row := P1 to (N1 - 1)
           do change(row);
   4.    row := P1;
   5.    dm := 1; for j := 1 to P1 - 1
           do dm := (d * dm) mod q;
   6.    read(PATTERN);
   7.    pathash;
   8.    repeat SEARCH(PATTERN',
            TEXT', found);
   9.      row := row + 1;
  10.    until (found = true)or
            (row > N1);
  12.    if found = true
  13.      then writeln
             ('The beginning of the
             pattern is found at',
             row - P1 + 1, column + 1)
  14.      else writeln ('not found')
  15. end.
```

```
procedure search (PATTERN', TEXT'
  found);
begin
  j := P2;
  repeat
    i := P2;
  while TEXT'[row, j] = PATTERN'[i]
        do
    begin j := j - 1; i := i - 1 end;
  if i := 0
  then
  begin directcompare
           (found, row, j + P2);
    column := j; end
    else
    begin
      y := binarysearch(TEXT'[row, j]);
        if y = P2 + 1 then j := j + P2
          else
          begin
            k := P2 - PH1[2, y];
            if DD[i] > k
              then j := j + DD[i]
              else j := j + k
          end
    end
  until (j > N2) or (found = true)
end;

function binarysearch(v: integer):
integer;
begin
  l := 1; r := P2;
    repeat
       x := (l + r) div 2;
       if v < PH1[1, x]
          then r := x - 1
          else l := x + 1
  until (v = PH1[1, x]) or (l > r);
if v = PH1[1, x] then binarysearch := x
  else binarysearch := P2 + 1
end;
```

Notice that the mismatch occurs at the right end of PATTERN' on almost every occasion when PATTERN' is positioned over TEXT', and this will make the BM

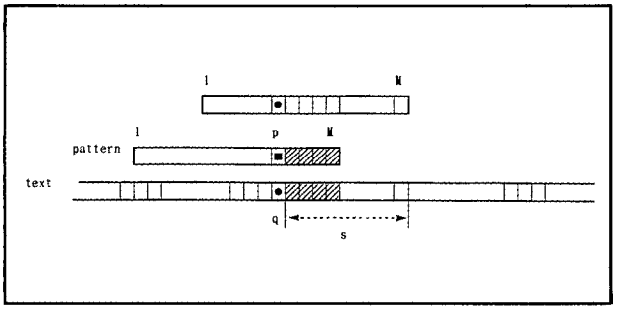

FIGURE 4. The Definition of the *D Shift* Function

	1	2	3	4	5	6	7	8
1	146	2183	12996	105086	196897	2097491	3148398	15730867
2	1	6	8	2	5	3	4	7

FIGURE 5. An Example of *PH1*

strategy work efficiently.

As the elements of TEXT' are large numbers bounded by q, the size of the character alphabet is equal to q, and also the definition of the D shift function in the BM algorithm is based on the size of the character alphabet; use of the D shift function is not appropriate in this context. Instead, we use function binarysearch to examine whether the value of TEXT' at position j is the same as any of PH1[1, k], $1 \leq k \leq P2$ and give the position in PH1 if the same value exists. As discussed above, in most cases the answer is NO, which means that j is increased by $P2$, that is, PATTERN' is slid to the right $P2$ places. On the other hand, occasionally the answer may be YES. In this case, comparing the two amounts of shifts made by the DD shift function and binarysearch, we choose the larger one. The amount k of shift made by binarysearch is the distance from the position kk where PATTERN'[kk] = TEXT'[row, j] to $P2$. As for the DD shift function, refer to [8].

For each binarysearch, log($P2$) steps are needed; so the average time of this approach is $O(N1 * N2 * \log(P2)/P2 + P1 * P2)$. Obviously, it takes less time when $P2$ increases, that is, the row length of the pattern gets longer. Refer to the third column of Table I. We can find that this approach works best as $P1$ and $P2$ are larger than 30. The advantages of this approach are similar to those listed in the previous section dealing with algorithm 3.

TWO-DIMENSIONAL TEXT MODIFICATION

The main aim of pattern matching is to modify a portion of the TEXT at positions where the pattern is found. In the one-dimensional case, the portion is often lengthened or shortened after modification. But, in the two-dimensional case, the modified portion generally does not change in size. This can be defined formally as follows. Assuming the pattern is found at TEXT[k1, k2], that is:

```
PATTERN[i, j]
    = TEXT[k1 + i - P1, k2 + j - P2]
                    1 ≤ i ≤ P1, 1 ≤ j ≤ P2
```

and

```
TEXT[k1 + i - P1, k2 + j - P2]
                  (1 ≤ i ≤ P1, 1 ≤ j ≤ P2)
```

is replaced by

```
PATTERN'[1 .. P1, 1 .. P2];
```

correspondingly, a portion of TEXT' should be substituted. The range of such substitution in TEXT' is

```
[k1 - P1 + i, k2 - P2 + j]
        (1 ≤ i ≤ 2 * P1 - 1, 1 ≤ j ≤ P2)
```

and is shown in the shaded part in Figure 6. The algorithm of modifying TEXT and TEXT' is summarized as follows:

```
procedure MODIFYTEXT';
begin
for i := 1 to P1 do
for j := 1 to P2 do
  TEXT[k1 - P1 + i, k2 - P2 + j]
    := PATTERN'[i, j];
  for i := k1 - P1 + 1 to k1 + P1 - 1 do
  for j := k2 - P2 + 1 to k2 dc
  TEXT'[i, j] := (TEXT'[i - 1, j] - TEXT
    [i - P1, j] * dm) * d + ord(TEXT
    [1, j])
end;
```

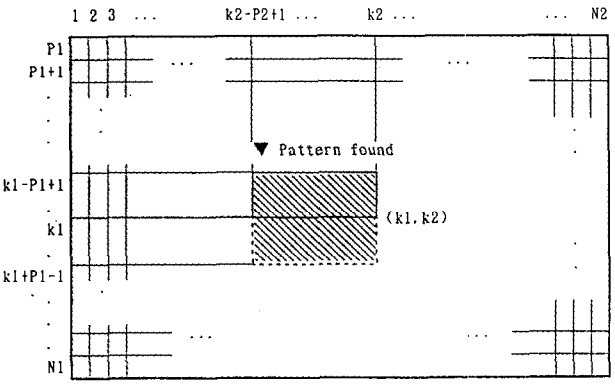

FIGURE 6. The Range of Substitution in TEXT'

Procedure MODIFYTEXT' takes $O(P1 * P2)$ time. This means that we can use TEXT' many times without spending much time to modify TEXT' when a portion of TEXT is replaced.

APPENDIX A

THE B ALGORITHM

The B algorithm invented by Bird [4] is briefly described as follows. The text[1 .. N, 1 .. N] is read from the upper left corner to the bottom right corner in the order of text[1, 1], ..., text[1, N], text[2, 1], ..., text[2, N], ..., text[N, 1], ..., text[N, N]. The pattern pat[1 .. M, 1 .. M] is divided into M rows of R_1 = pat[1, 1 .. M], ..., R_M = pat[M, 1 .. M]. Note that some of $R_1, \ldots, R_M$ may be equal with each other. Each time text[i, j] is read, it is determined whether the portion text[i, j - M + 1 .. j] = R matches any of $R_1, \ldots, R_M$ (horizontal matching). See Figure 1 and Figure 2.

We use a one-dimensional array a[1 .. N] such

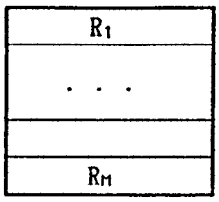

FIGURE 1. An Example of Pattern

that `a[j] = k` means that `k - 1` rows R_1, ..., R_{k-1} match the portion `text[i - k + 1 .. i - 1, j - M + 1 .. j]`. If $R = R_k$, `a[j]` is increased by one. If not, we find maximum s such that:

$$R = R_s,\ R_1 = R_{k-s+1},\ \ldots,\ R_{s-1} = R_{k-1} \qquad (1)$$

using the KMP method vertically, and set `a[j]` to $s + 1$. If there is no such s, `a[j]` is set to 0. If `a[j] = M + 1`, we can say that the pattern has been found.

This vertical use of the KMP method is done by regarding the sequences of rows R_1, ..., R_M as a linear pattern `r[1 .. M]`. The elements of `r[1 .. M]` can be up to M different symbols or integers for row patterns. The condition (1) above is illustrated by Figure 3.

An example of `pat[1 .. 5, 1 .. 5]` and `r[1 .. 5]` is given in Figure 4; we do not distinguish between rows and their corresponding integer values. While `R ≠ r[k]`, we perform `k ← h[k]` where h is the `h` function used in the KMP method. We repeat this until condition (1) is true. For the above r, we have the following `h` function:

	1	2	3	4	5
r	1	2	3	1	3
h	0	1	1	0	2

Now we go back to the problem of the horizontal matching. The solution is well explained by using the above example. We construct a finite automaton for

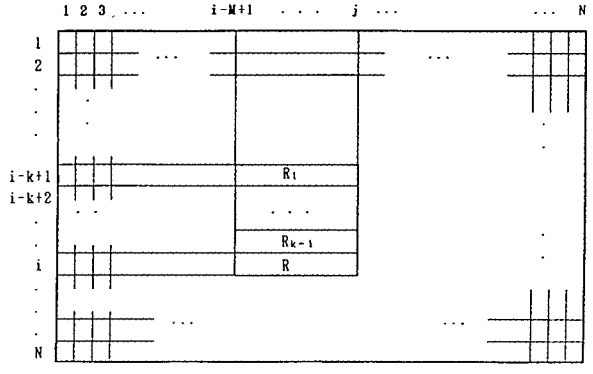

FIGURE 2. An Example of Text

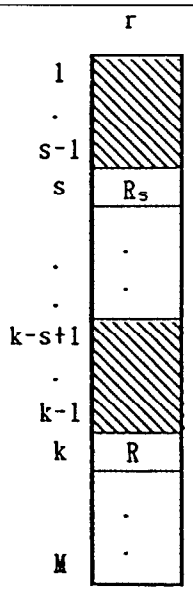

FIGURE 3. Illustration of Condition (1)

multiple pattern matching invented by Aho and Corasick [1] as shown in Figure 5.

In this machine, 0 is the initial state and 5, 8, and 12 are final states shown by double circles. The final states output integer values for corresponding rows. Using the output function `out`, these are specified as:

```
out(5) = 1 = r[1] = r[4],
out(8) = 2 = r[2],
out(12) = 3 = r[3] = r[5].
```

	1	2	3	4	5
1	a	a	b	b	a
2	a	a	a	b	b
3	a	b	a	b	a
4	a	a	b	b	a
5	a	b	a	b	a

Pat

1	1
2	2
3	3
4	1
5	3

r

FIGURE 4. An Example of Pat [1 .. 5, 1 .. 5] and Its Corresponding r [1 .. 5]

The **goto** function `g` is shown in Figure 5, for example, `g(2, b) = 3, g(2, a) = 6` and so on. If there is no definition of `g` in the machine, we set the value of `g` to "fail," for example, `g(6, a) = fail`. If the value of `g` is fail, the failure function `f` is consulted, which tells us from which state the processing should resume. For the above example, `f` becomes as follows:

j	1	2	3	4	5	6	7	8	9	10	11	12
f(j)	0	1	9	0	1	2	3	4	0	1	9	10

The B algorithm is given below, where `i` and `j` are expressed by `row` and `column`.

```
(a). Algorithm B.
Input: A text array T[1 .. N1, 1 .. N2] of string and a pattern matching machine
       constructed from pattern PT[1 .. P1, 1 .. P2]. The machine is composed
       of goto function g, failure function f, and output function out.
Output: locations in T at which PT occurs.
Method:
  begin
    for row := 1 until N1 do
      begin
        state := 0;
          for column := 1 until N2 do
            begin
              while g(state, T[row, column]) = fail do
                state := f(state);
              state := g(state, T[row, column]);
              if out(state) ≠ 0
                then
                  begin
                    k := a[column];
                    while (k > 0) and (PT[k, 1 .. P2] ≠ out(state)) do
                      k := h(k);
                    a[column] := k + 1;
                    if k = P1
                      then print('PT is found at (row - P1, column - P2)')
                  end
                else a[column] := 0
            end
      end
  end.
```

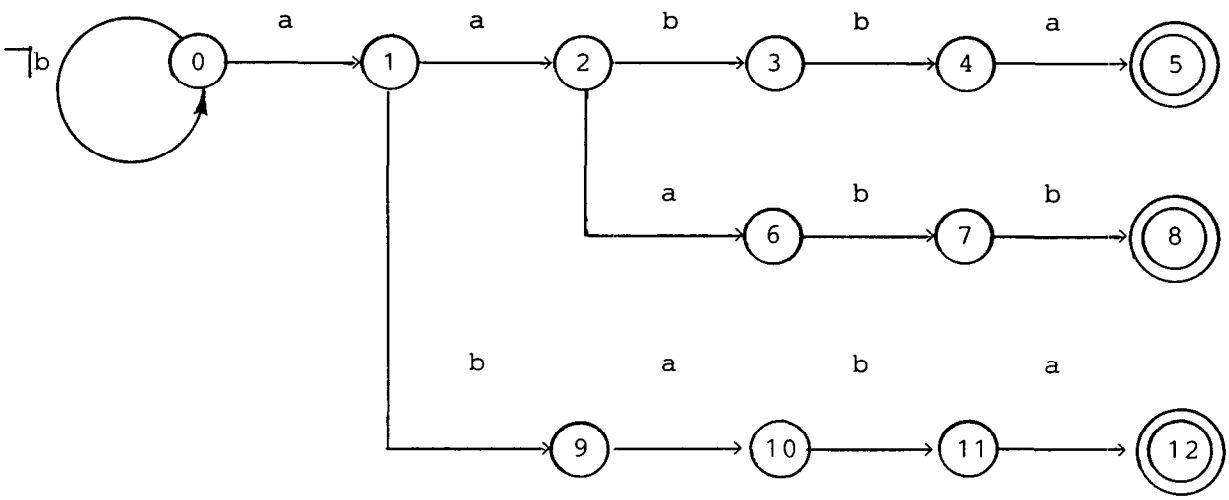

FIGURE 5. Multiple Pattern Matching Machine

```
(b). Construction of the goto function.
Input: Pattern PT[1 .. P1, 1 .. P2].
Output: Goto function g and output function out.
Method: We assume out(s) = 0 when state s is first created, and g(s, a) =
        fail if 'a' is undefined or if g(s, a) has not yet been defined. The
        procedure enter(PT) inserts into the goto graph a path that spells
        out PT. The elements of r[1 .. P1] represent each unique row of PT.
  begin
    newstate := 0; out(0) := 0;
    for i := 1 until P1 do
      enter(r[i], PT[i, 1 .. P2]);
    for all 'a' such that g(0, a) = fail do g(0, a) := 0
  end.

procedure enter(i1, A1 A2 ... Am);
  begin
    state := 0; j := 1;
    while g(state, Aj) ≠ fail do
      begin
        state := g(state, Aj);
        j := j + 1
      end;
    for p := j until m do
      begin
        newstate := newstate + 1;
        g(state, Ap) := newstate;
        state := newstate;
        out(state) := 0
      end;
    out(state) := i1
  end.
(c). Construction of the failure function.
Input: Goto function g and output function out.
Output: Failure function f.
Other variables: queue is a first-in-first-out data structure.
Method:
  begin
    queue := empty;
    for each 'a' such that g(0, a) = s ≠ 0 do
      begin
        queue := queue.s; {add s to the end of queue}
        f(s) := 0
      end;
    while queue ≠ empty do
      begin
        let r be the head of queue, that is, queue = r.tail;
        queue := tail;
        for each 'a' such that g(r, a) = s ≠ fail do
          begin
            queue := queue.s;
            state := f(r);
            while g(state, a) = fail do state := f(state);
            f(s) := g(state, a);
          end
      end
  end.

(d). Computation of h function.
Input: Pattern PT[1 .. P1, 1 .. P2].
```

```
Output: Shift function h.
Method:
  begin
    t := 0; h[1] := 0;
    for p := 2 to P1 do
      begin
        while (t > 0) and (PT[p - 1, 1 .. P2] ≠ PT[t, 1 .. P2]) do t := h[t];
        t := t + 1;
        if PT[p, 1 .. P2] ≠ PT[t, 1 .. P2] then h[p] := t else h[p] := h[t]
      end
  end.
```

Acknowledgments. The authors would like to thank Mr. Mark F. Villa of the University of Alabama at Birmingham for correcting the English of the first manuscript.

REFERENCES

1. Aho, A.V., and Corasick, M.J. Efficient string matching: An aid to bibliographic search. *Commun. ACM 18*, 6 (June 1975), 333–340.
2. Apostolico, A., and Giancarlo, R. The Boyer–Moore–Galli string searching strategies revisited. *SIAM J. Comput. 15*, 1 (Feb. 1986), 98–105.
3. Baker, T.P. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comput. 7*, 4 (Nov. 1978), 533–541.
4. Bird, R.S. Two dimensional pattern matching. *Info. Process. Lett. 6*, 5 (Oct. 1977), 168–170.
5. Boyer, R.S., and Moore, J.S. A fast matching algorithm. *Commun. ACM 20*, 10 (Oct. 1977), 762–772.
6. Davies, G., and Bowsher, S. Algorithms for pattern matching. *Softw. Pract. Exper. 16*, 6 (June 1986), 575–601.
7. Galil, Z. On improving the worst case running time of the Boyer–Moore string matching algorithm. *Commun. ACM 22*, 9 (Sept. 1979), 505–508.
8. Knuth, D.E., Morris, J.H., and Pratt, V.R. Fast pattern matching in strings. *SIAM J. Comput. 6*, 2 (June 1977), 323–350.
9. Zhu, R. F., and Takaoka, T. On improving the average case of the Boyer–Moore string matching algorithm. *J. Inf. Process. 10*, 3 (Mar. 1987), 173–177.

CR Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*pattern matching*; I.5.4 [**Pattern Recognition**]: Applications—*text processing*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Analysis of algorithms, information retrieval, pattern matching, pattern recognition, string searching, text editing

ABOUT THE AUTHORS:

RUI FENG ZHU is a graduate student of computer science at the University of Ibaraki, Japan.

TADAO TAKAOKA is a professor of computer science at the University of Ibaraki, Japan. His research interests include analysis of algorithms and program verification, and he is co-author of *Fundamental Algorithms*. Authors' Present Address: Department of Computer Science, University of Ibaraki, Hitachi, Japan 316.

Abstracts (*continued from p. 1135*)

structing dynamic Huffman codes that is described and analyzed in a companion paper [3]. The program runs in real time; that is, the processing time for each letter of the message is proportional to the length of its codeword. The number of bits used to encode a message of t letters is less than t bits more than that used by the well-known two-pass algorithm. This is best possible for any one-pass Huffman scheme. In practice, it uses fewer bits than all other Huffman schemes. The algorithm has applications in file compression and network transmission.

For Correspondence: Department of Computer Science, Brown University, Providence, R.I. 02912.

ALGORITHM 674
FORTRAN Codes for Estimating the One-Norm of a Real or Complex Matrix, with Applications to Condition Estimation

Nicholas J. Higham

FORTRAN 77 codes SONEST and CONEST are presented for estimating the 1-norm (or the ∞-norm) of a real or complex matrix, respectively. The codes are of wide applicability in condition estimation since explicit access to the matrix, A, is not required; instead, matrix-vector products Ax and A^Tx are computed by the calling program via a reverse communication interface. The algorithms are based on a convex optimization method for estimating the 1-norm of a real matrix devised by Hager [Condition estimates, *SIAM J. Sci. Stat. Comput. 5* (1984), 311–316]. We derive new results concerning the behavior of Hager's method, extend it to complex matrices, and make several algorithmic modifications in order to improve the reliability and efficiency.

For Correspondence: Department of Mathematics, University of Manchester, Manchester M13 9PL, England.