TIME-SPACE-OPTIMAL STRING MATCHING
(preliminary report)

Zvi Galil and Joel Seiferas

TR 87
February 1981

Abstract. We design and analyse a linear-time string-matching algorithm which uses only a fixed number of local storage locations. This completely eliminates the need for the tabulated "failure function" in Knuth, Morris, and Pratt's linear-time algorithm. It makes possible a completely general implementation as a Fortran subroutine or even on a six-head finite automaton.

Part of this work was done while the first author visited the University of Tokyo. The work of the first author was supported in part by the National Science Foundation under grant MCS-25301 at the University of California at Berkeley, and the work of the second author was supported in part by the National Science Foundation under grant MCS79-05006.

Authors' addresses: Z. Galil, Department of Mathematical Sciences, Tel-Aviv University, Tel-Aviv, ISRAEL; J. I. Seiferas, Department of Computer Science, University of Rochester, Rochester, New York, U. S. A. 14627.

TIME-SPACE-OPTIMAL STRING MATCHING
(preliminary report)

Zvi Galil
Department of Mathematical Sciences
Tel-Aviv University
Tel-Aviv, ISRAEL

Joel Seiferas
Department of Computer Science
University of Rochester
Rochester, New York 14627

Introduction

The string-matching problem is to find all full instances of a "pattern" character string x as a subword (contiguous substring) in a "text" string y. While the naive algorithm (trying the pattern from scratch starting at each successive text position) requires time proportional to the product $|x| \cdot |y|$ of the string lengths in the worst case, Knuth, Morris, and Pratt [10] and Boyer and Moore [2, 10, 5] designed algorithms which require only linear time (proportional to $|x|+|y|$). Their algorithms, however, require numbers of local storage locations proportional to the length $|x|$ of the pattern in every case, making a general implementation impossible without plenty of dynamic storage allocation.

In [7] we designed linear-time algorithms requiring only $O(\log|x|)$ (at most some constant times $\log|x|$) local storage locations in the very worst case, and we designed *almost* linear-time algorithms requiring no dynamic storage allocation at all ($O(1)$ local storage locations). Both we [8] and Karp and Rabin [9] have subsequently developed linear-time algorithms requiring no dynamic storage allocation, but these algorithms require other special capabilities. The algorithms in [8] are just our earlier linear-time algorithms, modified to fill their relatively small dynamic storage needs by temporarily borrowing some of the space occupied by the input pattern. While extremely simple conceptually, the Karp-Rabin algorithm requires operations such as multiplication (by the alphabet size) and a source of random numbers. (The algorithm can err, but randomization keeps the probability of error small and independent of the input pattern and text.)

Part of this work was done while the first author visited the University of Tokyo. The work of the first author was supported in part by the National Science Foundation under grant MCS-25301 at the University of California at Berkeley, and the work of the second author was supported in part by the National Science Foundation under grant MCS79-05006.

In this paper we describe a new linear-time string-matching algorithm requiring neither dynamic storage allocation nor other high-level capabilities. The algorithm can be implemented to run in linear time even on a six-head two-way finite automaton. Moreover, the automaton requires only "$\{=,\neq\}$-branching" [1]. (Decisions depend on which of the six scanned pattern or text symbols and positions are the same, but not on the particular symbols or how many symbols there are. Hence the same algorithm works even for an infinite alphabet.) A "real-time" implementation is possible on such a multihead finite automaton with a few more heads.

Preliminaries

Throughout this paper, let k be some fixed, comfortably large integer. In retrospect, $k=4$ will have been large enough.

For $1 \leq i \leq |w|$, let $w(i)$ denote the i-th character of the character string w. For $0 \leq i \leq j \leq |w|$, let $[i,j]_w = w(i+1)\dots w(j)$.

Consider any nonnull string z. z is a *period* of the character string w if w is a prefix of the infinite string $z^\infty = zzz\dots$. Equivalently, z is a period of w if and only if w is a prefix of zw [10]. For each $p \leq |w|$, let $\text{reach}_w(p) =$

$$\begin{aligned}&\max\{\, q \leq |w| \mid [0,p]_w \text{ is a period of } [0,q]_w \,\}\\ &\quad = p + \max\{\, q \leq |w|-p \mid [0,q]_w = [p,p+q]_w \,\}.\end{aligned}$$

z is *basic* if it is not of the form z'^i for any integer $i > 1$. z is a *prefix period* of w if it is basic and z^k is a prefix of w. Equivalently, $[0,p]_w$ is a prefix period of w if it is basic and $\text{reach}_w(p) \geq kp$. (Since k is fixed, we do not bother to include it in the terminology.)

Examples. The string $ababab = (ab)^3$ is not basic, but the string $abababa$ is. Both strings have periods ab, $abab$, $ababab$, and even $abababa$. (Every extension of a string w is a (relatively uninteresting) period of w.) The string $w = (abababa)^k abab$ has $\text{reach}_w(1) = 1$, $\text{reach}_w(2) = 7$, and $\text{reach}_w(7) = |w| = 7k + 4$. If $k \geq 4$, then w has only the one prefix period $[0,7]_w = abababa$; if $k = 3$, then ab is also a prefix period.

Periodicity lemma [11, 10]. *If a string of length $p_1 + p_2$ has periods of lengths p_1 and p_2, then it has a period of length* $\gcd(p_1, p_2)$.

Proof: Note that it has a period of length $|p_1 - p_2|$, and cite Euclid's algorithm. ■

Corollary. *Distinct prefix periods of the same string differ in length by at least a factor of $k - 1$.*

Proof: Suppose, to the contrary, that w has prefix periods of lengths p_1 and p_2 with $p_1 < p_2 \leq (k-1)p_1$. Then $p_2 + p_1 \leq kp_1 \leq \text{reach}_w(p_1)$; so $[0, p_2 + p_1]_w$ has periods of both lengths, hence also one of length $\gcd(p_1, p_2)$. Therefore the prefix $[0, p_2]_w$ has a period of length $\gcd(p_1, p_2) \leq p_1 < p_2$ and is not basic, a contradiction. ■

Searching for a Fixed Pattern

Several earlier string-matching algorithms follow a single general scheme. That scheme considers positions p for the pattern in the text in increasing order, and it maintains the length $q \geq 0$ of a pattern prefix known to match the text starting following position p ($[0,q]_x = [p, p+q]_y$). For appropriately calculated $p' > p$ and q', then, the algorithms search as follows:

```
$(p,q) \leftarrow (0,0)$
ploop:
  while $y(p+q+1) = x(q+1)$ do $q \leftarrow q+1$
  $(p,q) \leftarrow (p',q')$
  goto ploop
```

Each time q reaches the pattern length $|x|$, a full instance of the pattern has been found following position p in the text ($x = [p, p + |x|]_y$); the search can be continued by dropping out of the while-loop. (We consider $y(p+q+1) = x(q+1)$ to be false whenever $p + q + 1 > |y|$ or $q + 1 > |x|$, so this will be automatic.) Of course the algorithms should halt when the end of the text is reached ($p = |y|$).

The earlier algorithms differ only in how they calculate p' and q'. The naive algorithm conservatively calculates $p' = p + 1$ and $q' = 0$. Since $[0,q]_x = [p, p+q]_y$, however, consideration of $p' = p + shift$ is futile unless $[0, q - shift]_x = [shift, q]_x$; so the Knuth-Morris-Pratt algorithm calculates $p' = p + \text{shift}_x(q)$, where $\text{shift}_x(q) =$

$$\min\{\, shift > 0 \mid [shift, q]_x = [0, q - shift]_x \,\},$$

and then can even salvage $q' = q - \text{shift}_x(q)$ if $q > 0$. To get by with a skimpier tabulation of the shift function, algorithms from [7] calculate $(p', q') =$

$$\begin{cases} (p + \text{shift}_x(q), q - \text{shift}_x(q)) & \text{if } \text{shift}_x(q) \leq q/k, \\ (p + \max(1, \lceil q/k \rceil), 0) & \text{otherwise.} \end{cases}$$

If k is large, the case $\text{shift}_x(q) \leq q/k$ above should be relatively rare. Our new algorithm below is inspired by the vain wish that such cases could be ignored *entirely*. Lemmas 1 and 2 below characterise such cases in terms of prefix periods of the pattern.

Lemma 1. *If $\text{shift}_x(q) \leq q/k$, then $[0, \text{shift}_x(q)]_x$ is a prefix period of x.*

Proof: By definition, $[0, \text{shift}_x(q)]_x$ is a (shortest) period of $[0,q]_x$. It appears k times because $q \geq k \cdot \text{shift}_x(q)$. If it were of the form z^i for some integer $i > 1$ (i.e., not basic), then z would be a shorter period of $[0,q]_x$. ■

Lemma 2. *If $[0, shift]_x$ is a prefix period of x, then*

$$shift = \text{shift}_x(q) \leq q/k \Leftrightarrow k \cdot shift \leq q \leq \text{reach}_x(shift).$$

Proof: Only the proof of the backward implication ($\Leftarrow$) requires a nontrivial observation: By the periodicity lemma (assuming $k \geq 2$), $\text{shift}_x(q) < shift$ would contradict the assumption that $[0, shift]_x$ is basic. ■

The following decomposition theorem, proved in the next section, now leads to an efficient algorithm to search for any fixed pattern x:

Decomposition theorem. *Each pattern x has a parse $x = uv$ such that v has at most one prefix period and $|u| = O(\text{shift}_v(|v|))$.*

(We cannot insist that v have *no* prefix period: For $x = a^n$, we would have to have $|u| > n - k$ and $\text{shift}_v(|v|) = 1$.) The efficient algorithm uses the general scheme discussed above to search for full instances of the pattern suffix v. If v has *no* prefix period, then the crucial values are always

$$(p', q') = (p + \max(1, \lceil q/k \rceil), 0).$$

If v has *one* prefix period, and its length is p_1, then the crucial values are $(p', q') =$

$$\begin{cases} (p + p_1, q - p_1) & \text{if } kp_1 \leq q \leq \text{reach}_v(p_1), \\ (p + \max(1, \lceil q/k \rceil), 0) & \text{otherwise.} \end{cases}$$

On a text y, the time to find all instances of v will be $O(|v|+|y|)$, because the nonnegative, nondecreasing integer quantity $(k+1)p+q = O(|v|+|y|)$ is bound to increase every $O(1)$ steps. The algorithm checks naively (in time $O(|u|)$) whether the pattern prefix u occurs to the immediate left of each discovered instance of v. Since v can occur at most $|y|/\operatorname{shift}_v(|v|)$ times in a text y, the total time for the naive checks will be $O(|u|)|y|/\operatorname{shift}_v(|v|) = O(|y|)$. So the total time is $O(|v|+|y|) = O(|x|+|y|)$, and the number of local storage locations is some small constant.

Proof of Decomposition Theorem

To prove the decomposition theorem, we need one more lemma.

Lemma 3. *For each basic string w, there is a parse $w = w_1w_2$ such that, no matter what w' is, $w_2w^{k-1}w'$ has no prefix period shorter than $|w|$.*

Proof: By the periodicity lemma, any offending prefix period would have to be shorter than $|w|/(k-1)$; so w' is irrelevant, and we may as well use the infinite suffix $w' = w^\infty$.

The obvious way to seek the parse is to start with w^∞ and repeatedly delete offending prefixes:

```
While the remainder has a prefix $z^k$ with $|z| < |w|$,
    delete a shortest such $z$.
```

If this terminates, then a satisfactory parse of (the last entered copy of) w has been found. Here is a termination argument: When z is deleted, z^{k-1} remains a prefix of the remainder. Therefore, the *next* deletion *z' cannot be shorter.* (Otherwise, by the periodicity lemma, z'^k would have to be a prefix of z^2, contradicting the previous choice of z as *smallest.*) Therefore, either $z' = z$, or $|z'| > |z|$. (In fact, $|z'| > |z|$ implies $|z'| \geq (k-2)|z|$, by the periodicity lemma; hence, z^{k-2} will be a prefix of *every* subsequent remainder.) But, since w is basic, the periodicity lemma implies that no same z continues to work forever. Therefore, the length eventually reaches $|w|$. ∎

Remark. A stronger claim can be made for the algorithm in the proof of Lemma 3 above: *Not even one full w gets deleted.*

Proof of remark: Suppose some deletion z ends at position $p \geq |w|$ in w^∞. (Our convention is that position p separates characters number p and $p+1$.) The algorithm would now loop (contradicting our termination argument) if some deletion z had started one period back, at position $p-|w|$. Therefore, position $p-|w|$ must have been *within* some deletion z_0 starting at some position p_0 ($p-|w|-p_0 < |z_0|$). By our observation in the proof above, ${z_0}^{k-2}$ occurs starting at position p. Therefore, it occurs at both positions p_0 and $p-|w|$. From this, it follows that ${z_0}^{k-2}$ (and hence ${z_0}^k$) has a period of length $p-|w|-p_0 < |z_0|$, contradicting the choice of z_0 as a shortest prefix period starting at position p_0. ∎

Proof of decomposition theorem: We obtain $v = [s,|x|]_x$ by deleting appropriate prefixes from the pattern until the remainder has at most one prefix period:

```
$s \leftarrow 0$
while $[s,|x|]_x$ has *more than* one prefix period do
  begin
    Let $p_2$ be the length of the second shortest
        prefix period.
    By appeal to Lemma 3, find $s' < s + p_2$
        such that $[s',|x|]_x$ has no prefix period
        shorter than $p_2$.
    Set $s \leftarrow s'$.
  end
```

It remains only to prove that $|u| = O(\operatorname{shift}_v(|v|))$ finally holds. For each s, let

$$\ell(s) = \text{length of the shortest period of } [s,|x|]_x,$$
$$p_1(s) = \text{length of the shortest } \textit{prefix} \text{ period of } [s,|x|]_x \text{ (if there is one).}$$

(If $p_1(s)$ exists, then $[s, s+p_1(s)]_x$ is the shortest period of some *prefix* of $[s,|x|]_x$, guaranteeing that $p_1(s) \leq \ell$.) By induction, we prove the loop invariant $s < 2\min(p_1(s), \ell(s))$:

$$\begin{aligned} s' &< s + p_2 \\ &< 2\min(p_1(s),\ell(s)) + p_2 \\ &= 2p_1(s) + p_2 \\ &\leq 2p_2/(k-1) + p_2 \\ &\leq 2p_2 \\ &\leq 2\min(p_1(s'),\ell(s')). \end{aligned}$$

(The periodicity lemma ensures that $p_2 \leq \ell(s')$.) Finally, therefore, $|u| = s < 2\ell(s) = 2\operatorname{shift}_v(|v|)$. ∎

Preprocessing a Pattern

Even the algorithm for *finding* the decomposition $x = uv = [0,s]_x[s,|x|]_x$ above can be implemented efficiently. First note that, by Lemma 3 and the subsequent remark, there is a very simple algorithm for finding s':

```
$s' \leftarrow s$
while $[s',|x|]_x$ has a prefix period shorter than $p_2$
  do Delete a *shortest* one.
```

Now an efficient implementation is natural in terms of efficient subroutines to find the one or two shortest prefix periods of a string.

The general scheme discussed above, and interpreted as in [7], provides an efficient algorithm to determine whether a string w has a prefix period and to find the shortest one if it does. The algorithm matches w against itself, starting with $(p, q) = (1, 0)$. Of course no full instance of the pattern will be found, but for each i we will have $\text{shift}_w(i) = p$ the first time $p + q = i$ holds. To see this, consider any i $(1 \leq i \leq |w|)$. The first time $p + q = i$ holds, no symbol beyond $w(i)$ has been examined, so $\text{shift}_w(i)$ is still a prospective position for the pattern; i.e., $p \leq \text{shift}_w(i)$. Since the algorithm guarantees that $[p, i]_w$ is a prefix of $[0, i]_w$ at this point, we cannot have $p < \text{shift}_w(i)$; so $p = \text{shift}_w(i)$, as claimed. The algorithm we want simply watches for the first i with $\text{shift}_w(i) \leq i/k$ $(p \leq (p+q)/k)$. (By Lemma 2, these inequalities will be *equalities*.) Until such an i is found, the calculation

$$(p', q') = (p + \max(1, \lceil q/k \rceil), 0)$$

will always be appropriate, since $\text{shift}_w(q) > q/k$ for all $q < i$. If the shortest prefix period exists and has length p_1, then the final values of p and q will be p_1 and $(k-1)p_1$, respectively. Therefore, the total time will be $O\big((k+1)p + q\big) =$

$$\begin{cases} O(p_1) & \text{if } p_1 \text{ exists,} \\ O(|w|) & \text{in any case.} \end{cases}$$

To determine whether w has a prefix period *shorter than some given* p_2, we can use the same algorithm until p reaches p_2; the running time for this variant will be

$$\begin{cases} O(p_1) & \text{if } p_1 < p_2 \text{ exists,} \\ O(p_2) & \text{in any case.} \end{cases}$$

Similarly, there is an efficient algorithm to determine whether a string w has *two* prefix periods and to find the *second* shortest one if it does. First, the algorithm seeks the shortest prefix period as above. If the shortest prefix period exists and has length p_1, then the algorithm straightforwardly determines $\text{reach}_w(p_1)$ in time $O(\text{reach}_w(p_1))$. Finally, the algorithm matches w against itself, starting with $(p, q) = (1, 0)$ as above, now watching for the first $i > \text{reach}_w(p_1)$ with $\text{shift}_w(i) = i/k$. Until such an i is found, the calculation $(p', q') =$

$$\begin{cases} (p + p_1, q - p_1) & \text{if } kp_1 \leq q \leq \text{reach}_w(p_1), \\ (p + \max(1, \lceil q/k \rceil), 0) & \text{otherwise} \end{cases}$$

will always be appropriate, since every $q < i$ will have either $\text{shift}_w(q) > q/k$ or $\text{shift}_w(q) = p_1$. If the second shortest prefix period exists, then its length p_2 will have to be at least $\text{reach}_w(p_1) - p_1$, by the periodicity lemma. By looking at the quantity $(k+1)p+q$ again, therefore, we see that the total time will now be

$$\begin{cases} O(p_1) + O(\text{reach}_w(p_1)) + O(p_2) & \text{if } p_2 \text{ exists,} \\ O(p_1) + O(\text{reach}_w(p_1)) + O(|w|) & \text{if only } p_1 \text{ exists,} \\ O(|w|) & \text{in any case} \end{cases}$$

$$= \begin{cases} O(p_2) & \text{if } p_2 \text{ exists,} \\ O(|w|) & \text{in any case.} \end{cases}$$

Now consider using these efficient subroutines to implement the outlined decomposition algorithm. The time for the one failed entry test for the outer loop will be $O(|v|)$. In terms of the current value of p_2, the time for finding s' will be $O(s' - s) + O(p_2) = O(p_2)$. Therefore, the time for the entire loop body, including the passed entry test, will be $O(p_2)$. By the periodicity lemma, each successive p_2 will be at least $k - 2 \geq 2$ times the preceding one. So the total decomposition time will be

$$O(|v|) + O\big(|v|(1 + 1/2 + 1/4 + \cdots)\big) = O(|v|) = O(|x|).$$

Combining this preprocessing algorithm with the searching algorithm described earlier, we finally get an algorithm which can find all full instances of an arbitrary pattern x in an arbitrary text y in time proportional to $|x| + |y|$, without dynamic storage allocation.

An Integrated Implementation

Having established the existence of our algorithm, we turn now to integrated and improved implementation. The following implementation of the entire algorithm will lead to a multihead finite automaton implementation in the next section.

```
$(p,q) \leftarrow (0,0)$
$(s,p_1,q_1) \leftarrow (0,1,0)$
$(p_2,q_2) \leftarrow (0,0)$
newp1:
  while $x(s+p_1+q_1+1) = x(s+q_1+1)$
      do $q_1 \leftarrow q_1+1$
  if $p_1+q_1 \geq kp_1$
      then $[(p_2,q_2) \leftarrow (q_1,0)$; goto newp2$]$
  if $s+p_1+q_1 = |x|$ then goto search
  $(p_1,q_1) \leftarrow (p_1+\max(1,\lceil q_1/k \rceil),0)$
  goto newp1
newp2:
  while $x(s+p_2+q_2+1) = x(s+q_2+1)$
      and $p_2+q_2 < kp_2$ do $q_2 \leftarrow q_2+1$
  if $p_2+q_2 = kp_2$ then goto parse
  if $s+p_2+q_2 = |x|$ then goto search
  if $q_2 = p_1+q_1$
    then $(p_2,q_2) \leftarrow (p_2+p_1,q_2-p_1)$
    else $(p_2,q_2) \leftarrow (p_2+\max(1,\lceil q_2/k \rceil),0)$
  goto newp2
parse:
  while $x(s+p_1+q_1+1) = x(s+q_1+1)$
      do $q_1 \leftarrow q_1+1$
  while $p_1+q_1 \geq kp_1$ do $(s,q_1) \leftarrow (s+p_1,q_1-p_1)$
  $(p_1,q_1) \leftarrow (p_1+\max(1,\lceil q_1/k \rceil),0)$
  if $p_1 < p_2$
    then goto parse
    else goto newp1
search:
  while $y(p+s+q+1) = x(s+q+1)$
      do $q \leftarrow q+1$
  if $q = |x|-s$ thenif $[p,p+s]_y = [0,s]_x$
    then announce instance of $x$ at text position $p$
  if $q = p_1+q_1$
    then $(p,q) \leftarrow (p+p_1,q-p_1)$
    else $(p,q) \leftarrow (p+\max(1,\lceil q/k \rceil),0)$
  if $p+s \leq |y|$ then goto search
```

(Note: The variables p and q are introduced only for clarity. By the time they are used (in the segment following *search*), p_2 and q_2 are free and could be reused instead.)

The main purposes of the program segments above are as follows:

newp1:
Find the shortest prefix period of $[s,|x|]_x$.
newp2:
Find the second shortest prefix period of $[s,|x|]_x$.
parse:
Increment s.
search:
Search the text for $x = [0,s]_x[s,|x|]_x$.

A more detailed direct analysis relies on the validity of the following assertions at the labeled checkpoints:

newp1:
$[s,|x|]_x$ has no prefix period shorter than p_1.
$[s+p_1,s+p_1+q_1]_x = [s,s+q_1]_x$
$p_2 \leq p_1$.
$p_2+q_2 = kp_2$.

newp2:
$[s,|x|]_x$ has shortest prefix period of length p_1.
$[s,s+p_1+q_1]_x$ has period of length p_1.
$[s,s+p_1+q_1+1]_x$ does not.
$p_2 \geq q_1$.
$[s,|x|]_x$ has only one prefix period shorter than p_2.
$[s+p_2,s+p_2+q_2]_x = [s,s+q_2]_x$.
$p_2+q_2 < kp_2$.

parse:
$[s,|x|]_x$ has no prefix period shorter than p_1.
$[s+p_1,s+p_1+q_1]_x = [s,s+q_1]_x$.
$p_1 < p_2$.
$p_2+q_2 = kp_2$.

search:
$[s,|x|]_x$ has at most one prefix period.
If $[s,|x|]_x$ does have a prefix period,
then its length is p_1.
$[s,s+p_1+q_1]_x$ has shortest period of length p_1.
$[s,s+p_1+q_1+1]_x$ does not have
period of length p_1.
All instances of x starting at text positions
before p have been announced.
$[p+s,p+s+q]_y = [s,s+q]_x$.
$s < 2p_1$.

First consider the very last assertion, which is crucial for the time analysis. Assuming all the other assertions hold, the integrated implementation increments s in the same way as the original algorithm. So the validity of the assertion follows from our proof of the decomposition theorem.

Verification of all the other assertions is routine, frequently by appeal to the periodicity lemma. Note that we have replaced the test $kp_1 \leq q_2 \leq p_1+q_1$ with the simplified test $q_2 = p_1+q_1$ in the segment following *newp2*. This is justified by the mismatch $x(s+p_2+q_2+1) \neq x(s+q_2+1)$ (and the periodicity lemma). Similarly, we have replaced the test $kp_1 \leq q \leq p_1+q_1$ with just $q = p_1+q_1$ in the segment following *search*. The last two assertions for *newp1* are included for their aid in the time analysis below.

For a direct time analysis, consider the expression

$$\begin{aligned} s2 &+ ((k+1)p_1 + q_1) \\ &+ ((k+1)p_2 + q_2) \\ &+ ((k+1)p + q). \end{aligned}$$

The value of this expression is always an integer $O(|x| + |y|)$. Its initial value is positive, and every assignment increases its value. (This is immediately clear for every assignment except $(p_2, q_2) \leftarrow (q_1, 0)$. Since that assignment occurs only when $p_1 + q_1 \geq kp_1$ (by the test) and $p_2 \leq p_1$ and $p_2 + q_2 = kp_2$ (by assertions at *newp*1), however,

$$\begin{aligned} (k+1)p_2 + q_2 &= kp_2 + (p_2 + q_2) \\ &= 2kp_2 \\ &\leq 2kp_1 \\ &\leq 2kq_1/(k-1) \\ &< (k+1)q_1; \end{aligned}$$

hence, the contribution by p_2 and q_2 is greater after the assignment than before.) Some such assignment is executed every $O(1)$ steps, except for tests $[p, p+s]_y = [0, s]_x$. But we have seen already that the total time for these tests is $O(|y|)$, because $s = O(p_1)$ holds at *search*. Therefore, the algorithm's total running time must be $O(|x| + |y|)$.

Multihead Finite Automaton

For an eleven-head finite automaton implementation, maintain text heads at positions $p + s + q$ and $p + s$, and pattern heads at positions $s + p_1 + q_1$, $s + q_1$, $s + kp_1$, $s + p_2 + q_2$, $s + q_2$, $s + kp_2$, $s + q$, s, and s again. (The values $s + kp_1$ and $s + kp_2$ might exceed $|x|$, but they will certainly be bounded by $(k+1)|x|$. The pattern head maintaining such a value can reverse direction whenever it reaches an endmarker, and the finite control can keep track of the net number of reversals for the current value.) With heads at these positions, each test $[p, p+s]_y = [0, s]_x$ requires only $O(s)$ steps, as before; every other test requires only $O(1)$ steps, provided the finite control keeps track of the order of the head positions; and each assignment requires at most a number of steps proportional to the resulting increase in the expression used in the time analysis above. Therefore, the total time remains $O(|x| + |y|)$.

We can save two pattern heads above by letting positions $s + kp_1$, $s + kp_2$, and $s + q$ share a single head. We let that head maintain position $s + kp_1$ in the segments following *newp*1 and *parse*, $s + kp_2$ in the segment following *newp*2, and $s + q$ in the segment following *search*. The time to relocate the head to position $s + q = s$ the one time control enters the segment following *search* is certainly $O(|x|)$. The time to relocate the head from position $s + kp_1$ (via position s) to position $s + kp_2 = s + kq_1$ when control enters the segment following *newp*2 is $O(p_1 + q_1) = O(q_1)$, but we were already allowing that long for the immediately preceding assignment $(p_2, q_2) \leftarrow (q_1, 0)$. The time to relocate the head from position $s + kp_2$ (via position s) to position $s + kp_1$ when control enters the segment following *parse* is $O(p_2 + p_1) = O(p_2)$, but p_2 will be at least $k - 2 \geq 2$ times as large the next time this is necessary. Therefore, the total time still remains $O(|x| + |y|)$.

To save one more pattern head, note that the head at position $s + p_2 + q_2$ is needed only in the segment following *newp*2, and that the second head at position s is needed only *outside* that segment (in fact, only for the assignment $(s, q_1) \leftarrow (s + p_1, q_1 - p_1)$). As above, there is time for one shared head to shift roles on entering and leaving the segment.

If both the pattern and the text are provided on the same input tape, then we can save two more heads. The two text heads are needed only after *search* is reached. At that point, however, it becomes unnecessary ever again to maintain pattern positions $s + p_2 + q_2$ and $s + q_2$; so the corresponding *pattern* heads can relocate to *text* position s and begin to serve as the *text* heads. The final result is the promised six-head finite automaton requiring only $\{=, \neq\}$-branching.

Real-Time Algorithms

In [12] we reported real-time Turing machine algorithms for string matching, for recognition of squares (strings of the form ww) and palindromes (strings which are their own reverses), and for a number of generalizations of these problems. Using our new algorithm as a building block, we can adapt all of these algorithms to run in real time even on a multihead finite automaton. In this context, "real time" means that, for some constant c, the input tape is extended by one symbol every c steps, and that the automaton must rule immediately on the acceptability of the extended input string. For the string-matching problem, the pattern (while it lasts) and the text are extended simultaneously, and each verdict must indicate whether an instance of the pattern-so-far ends at the current end of the text. (The problem would be much easier if the entire pattern preceded the entire text.)

Adaptation of the real-time algorithms from [12] is beyond the scope of this report, but the key is to use a variant of our new string matcher wherever in [12] we used the Fischer-Paterson string matcher [4]. The latter linear-time algorithm already required linear space on an off-line Turing machine; so for convenience in [12], we freely allowed ourselves the luxury of marking all the instances of x on a copy of y for examination in latter passes. In addition, we made use of the Fischer-Paterson algorithm to find not only *full* instances of x, but also "overhang" instances. An *overhang* instance of x occurs following position p in y ($|y| \geq |x|$) if either

$$-|x| \leq p \leq 0 \quad \text{and} \quad [0, p+|x|]_y = [-p, |x|]_x,$$

or

$$|y| - |x| \leq p \leq |y| \quad \text{and} \quad [p, |y|]_y = [0, |y| - p]_x.$$

In the first case, we call it a *left overhang* instance, and in the second, we call it a *right overhang* instance.

Careful examination of the algorithms in [12] and those in the preliminary report included in [6] reveals that, with one possible exception, it is never really necessary to record the instances of a pattern in a text for later examination. Instead, it suffices to be able to detect the instances one at a time, *in order*. The one possible exception is in the algorithm for Lemma 1.2 in [12], but an alternative algorithm is available from [6].

As described above, our multihead finite automaton algorithm already detects the *full* instances of x in y in order of their appearance. It remains only to modify the algorithm to detect *all* instances (both full and overhang) in order.

As a first step, we describe how the algorithm can detect all (left) overhang instances following positions p in the range $-|x|/2 \leq p \leq 0$, in time $O(|x|)$. Let $x = uv$, where $|u| = \lfloor |x|/2 \rfloor$. First the algorithm should find the length p_0 of the shortest period of v. (To do this in time $O(|v|)$, it should search as above for the second *full* instance of the pattern v in the text vv.) In the case that $p_0 \geq |v|/2$, the algorithm should search as above for *full* instances of v in $[0, |x|]_y$, and check naively (in time $O(|u|) = O(|x|)$) whether each discovered full instance of v extends to a left overhang instance of the entire pattern x. Since x can occur at most $|x|/p_0 = O(|x|/|v|) = O(|x|/|x|) = O(1)$ full times in y, the total time for the naive checks will be $O(|x|)$.

In the remaining case that $p_0 < |v|/2$, the algorithm should reparse x, in time $O(|x|)$, into $[0, i]_x[i, |x|]_x$ such that

$i \leq |x|/2$,
$[i, |x|]_x$ has (shortest) period of length p_0,
$[i-1, |x|]_x$ (if $i > 0$) does not.

Using the fact that $[i, |x|]_x$ has a period of length $p_0 < |v|$, the algorithm should search first for the left overhang instances which are left overhang instances of $[i, |x|]_x$. To do this, it should first determine $q_0 = \min(\text{reach}_y(p_0), |x| - i)$ in time $O(|x| - i) = O(|x|)$, and then search for full instances of v in $[0, |x| - i]_y$. A discovered instance of v extends to a left overhang instance of $[i, |x|]_x$ if and only if it *ends* at a position $p \leq q_0$. To find the left overhang instances of x which are *not* left overhang instances of $[i, |x|]_x$ (assuming $i > 0$, so that there might be some), the algorithm should search for full instances of $[i-1, |x|]_x$, and check naively whether each discovered full instance extends to a left overhang instance of the entire pattern. By the periodicity lemma, the length of the shortest period of $[i-1, |x|]_x$ must exceed $|v| - p_0 > |v| - |v|/2 = |v|/2$; so the total time for the naive checks will again be $O(|x|)$.

An algorithm to find *all* the nontrivial left overhang instances of x in order can simply apply the algorithm just described to the sequence of patterns $[|x|-2, |x|]_x, [|x|-4, |x|]_x, [|x|-8, |x|]_x, \ldots, [0, |x|]_x = x$. (If $|x|$ is not a power of 2, then x can be padded on the left out to the next such length, and the last few left overhang instances can be ignored.) The total time will be $O(2 + 4 + 8 + \cdots + |x|) = O(|x|)$. A similar algorithm, applied to the sequence of patterns $x = [0, |x|]_x, [0, |x|/2]_x, [0, |x|/4]_x, \ldots, [0, 2]_x$, can detect all *right* overhang instances of x in time $O(|x|)$. Combining results, then, we conclude that a multihead finite automaton, with only $\{=, \neq\}$-branching, can detect, in order, all instances (left overhang, full, and right overhang) of an arbitrary pattern x in an arbitrary text y in time $O(|x| + |y|)$.

Remaining Issues

We have refuted previously formulated versions of the conjecture that a two-way multihead finite automaton could not perform string matching efficiently [1, 8], but a number of questions remain, especially in retrospect. We list some of these below.

1. How much time and local storage are needed for a string matcher which cannot back up or reread the text? Our earlier algorithms [7, 8] had this property, but our new one sometimes has to reread some of the last $|x|$ many text characters. In terms of storage of the "restricted" variety, in other words, the new algorithm uses *nearly twice as much* as the earlier ones.

2. Can any *one-way* multihead finite automaton perform string matching at all? (Any such string matcher could not help running in linear time.)

3. How few heads suffice for a linear-time string matcher? For a real-time string matcher? For a real-time palindrome recognizer? Note that *two* heads suffice for the naive, *quadratic*-time string matcher. Ďuriš and Galil [3] have shown that a two-head finite

automaton cannot perform string matching at all if one of the heads is "blind" (can distinguish only endmarkers).

4. The instances of pattern x in text y can be characterized by a binary string of length $|x| + |y|$, the i-th bit indicating whether the pattern occurs following text position $i - |x|$. By the real-time result, a multihead finite automaton can simulate one-way access to this characterization in real time. How fast can a multihead finite automaton simulate *two-way* access to the characterization?

Acknowledgements. The authors thank Peter Gacs and Gary Peterson for their criticism of earlier versions of this paper.

References

1. A. B. Borodin, M. J. Fischer, D. G. Kirkpatrick, N. A. Lynch, and M. Tompa, *A time-space tradeoff for sorting on non-oblivious machines*, 20th Annual Symposium on Foundations of Computer Science (San Juan, Puerto Rico, 1979), IEEE Computer Society, Long Beach, California, 1979, pp. 319–327.

2. R. S. Boyer and J. S. Moore, *A fast string searching algorithm*, Communications of the ACM 20, 10 (October 1977), 762–772.

3. P. Ďuriš and Z. Galil, *Fooling a two way automaton or One pushdown store is better than one counter for two way machines (preliminary version)*, Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing (Milwaukee, Wisconsin, 1981), Association for Computing Machinery, New York, 1981 (to appear).

4. M. J. Fischer and M. S. Paterson, *String-matching and other products*, in: Complexity of Computation (SIAM-AMS Proceedings 7), R. M. Karp (Editor), American Mathematical Society, Providence, Rhode Island, 1974, pp. 113–125.

5. Z. Galil, *On improving the worst case running time of the Boyer-Moore string matching algorithm*, Communications of the ACM 22, 9 (September 1979), 505–508.

6. Z. Galil and J. Seiferas, *Recognizing certain repetitions and reversals within strings*, 17th Annual Symposium on Foundations of Computer Science (Houston, Texas, 1976), IEEE Computer Society, Long Beach, California, 1976, pp. 236–252.

7. Z. Galil and J. Seiferas, *Saving space in fast string-matching*, SIAM Journal on Computing 9, 2 (May 1980), 417–438.

8. Z. Galil and J. Seiferas, *Linear-time string-matching using only a fixed number of local storage locations*, Theoretical Computer Science (to appear).

9. R. M. Karp and M. O. Rabin, personal communication.

10. D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, *Fast pattern matching in strings*, SIAM Journal on Computing 6, 2 (June 1977), 323–350.

11. R. C. Lyndon and M. P. Schützenberger, *The equation $a^M = b^N c^P$ in a free group*, Michigan Mathematical Journal 9, 4 (1962), 289–298.

12. J. Seiferas and Z. Galil, *Real-time recognition of substring repetition and reversal*, Mathematical Systems Theory 11, 2 (1977), 111–146.