Linear-Time String-Matching Using Only a Fixed Number of Local Storage Locations

by
Zvi Galil and Joel Seiferas

TR60
September 1979

Author's Addresses:  J.I. Seiferas
Computer Science Department
University of Rochester
Rochester, NY 14627

Z. Galil
Department of Mathematical Sciences
Computer Science Division
Tel-Aviv University
Tel-Aviv, Israel

Part of this work was done while the first author was visiting the IBM San Jose Research Laboratory, San Jose, California. The work of the first author was supported in part by the Israel Commission for Basic Research, and the work of the second author was supported in part by the National Science Foundation under Grant No. MCS79-05006.
Abstract. We report a linear-time string-matching algorithm for a random-access machine without dynamic storage allocation. To do this, we tell how to adapt a cited algorithm to fill its dynamic storage needs by temporarily borrowing some of the space occupied by the input pattern. In automata-theoretic terms, we tell how to adapt the cited algorithm to run on a writing multihead finite automaton with a restricted writing alphabet.

Key words. string-matching, pattern-matching, text-searching, multihead finite automaton, writing multihead finite automaton
Introduction

In an earlier paper [3], we asked whether any variant of the linear-time string-matching algorithm of Knuth, Morris, and Pratt [4] could be implemented as a FORTRAN subroutine. Although a FORTRAN subroutine can use a variable-length array if it receives the array and its length as arguments, every local storage location must be allocated when the subroutine is compiled; i.e., there is no provision for storage allocation "dynamically", during execution. The Knuth-Morris-Pratt algorithm, however, uses a number of local storage locations which grows with the size of the input to the algorithm. This rules out any straightforward implementation which is completely general (at least in principle). Linear-time algorithms we developed [3, Theorem 1] use numbers of local storage locations which grow much more slowly, but which do still grow. In this note, we show that a fixed number of local storage locations does suffice, at least for an implementation which is slightly less "straightforward".

The string-matching problem is to find all instances of a "pattern" character string \( x \) as a subword (contiguous substring) in a "text" string \( y \). In [3], we decreed that a "straightforward implementation" would neither change the contents of the \(|x| + |y|\) memory locations initially containing \( x \) and \( y \) nor store more than \( O(\log(|x| + |y|)) \) bits (enough for an arbitrary reference into \( x \) or \( y \)) in a single memory location. (We use \(|w|\) to denote the length of the character string \( w \), and "big-O" to denote "at most some constant times".) In this note, we relax the first constraint, but we assume each of the
$|x| + |y|$ memory locations involved is large enough to hold only one symbol from the pattern's and text's alphabet. In the case that the alphabet is binary, for example, the pattern and text are supplied in an array of $|x| + |y|$ one-bit registers.

The algorithm we describe will never change the contents of text registers, and it will change the contents of pattern registers only before the first text character is read. (As we point out later, the original pattern is not lost. An additional linear-time postprocessing phase, after the entire text has been read, could restore the pattern registers to their original state.) The only significant local storage locations will hold a pointer into the text and a fixed number of pointers into the pattern. Moreover, the only necessary pointer arithmetic will be "add 1" and "subtract 1", the latter only for pattern pointers. In other words, we will describe a somewhat restricted writing multihead finite automaton.

In [3], although we did not explicitly formulate the machine model, we described a string-matching algorithm which can run in linear time on a multihead finite automaton with a single, one-way, read-only text input head; a fixed number of two-way, read-only pattern input heads; and the following primitive instructions:

1. Branch according to the next text character.
2. Announce the end of a pattern instance in the text.
3. Branch according to the pattern character or endmarker scanned by pattern head $i$.
4. Shift pattern head $i$ right or left (but not beyond an endmarker).
5. Permanently store the position currently occupied by pattern
(6) Branch according to whether the position currently occupied by pattern head \( i \) has ever been stored. Moreover, in the terminology of [3] (reviewed below), the sequence of stored pattern positions is \( \text{GATE}(0), \text{GATE}(1), \ldots, \text{GATE}(2\ell) \), where \( \ell = O(\log |x|) \); and all these positions are stored before the first text character is read. The algorithm, together with an implementation for the particular instances of instructions (5) and (6) which arise, is described in the proof of Theorem 5 in [3]. In this note we describe an alternative implementation for these same instances. For this implementation, we allow the pattern heads to write, but only characters already in the pattern alphabet. (Without this restriction, the automaton could simply implement M. Fischer and M. Paterson's linear-time string-matching Turing machine [1] on a "second track" of the pattern input tape.) The total additional time for the implementation is proportional to the pattern length. Hence, we show that instructions (5) and (6) in the result stated above can be replaced by

(7) Change the pattern character scanned by pattern head \( i \) to the pattern character \( \sigma \) (from the same alphabet).

The GATE Sequence

It is nice properties of the GATE sequence that make our algorithm possible. To record a representation of that sequence, our algorithm could overwrite some long enough redundant segment of the pattern, if there were one. This works because of the fact that longer GATE sequences can arise only from more redundant patterns. The guaranteed redundancy
is a prefix of the form \( \text{www} \), with \( w \) exponentially longer than the
GATE sequence (see Lemmas 3 and 4 below). Our algorithm will write over
the third instance of \( w \) in such a prefix, and move its notes to longer
such prefixes as the GATE sequence grows.

Consider any fixed pattern \( x \). For each \( q : |x| \), define
\[
KMP_3(q) = \min\{p : |0,p|_x \text{ has prefix } |p,q|_x \text{ & } 0 < p \leq q/3\},
\]
where \([i,j]_w\) denotes that substring of the character string \( w \) which
starts at position \( i+1 \) and ends at position \( j \). (We will use \([i,j]\)
(without the subscript) to refer to the sequence of positions.) The GATE
sequence will be a succinct representation of the partial function \( \text{KMP}_3 \).

Let \( \text{VAL}(1) < \ldots < \text{VAL}(\ell) \) be the sequence of distinct values \( \text{KMP}_3 \)
assumes. Take \( \text{GATE}(0) = 0 \) and, for each \( r : 1 \leq r \leq \ell \),
\[
\text{GATE}(2r - 1) = \min\{q : \text{KMP}_3(q) = \text{VAL}(r)\},
\]
\[
\text{GATE}(2r) = 1 + \max\{q : \text{KMP}_3(q) = \text{VAL}(r)\}.
\]
The following four lemmas summarize properties of the GATE sequence.
For proofs, consult [3].

**Lemma 1.** \( \text{GATE}(0) < \text{GATE}(1) < \ldots < \text{GATE}(2\ell) \).

**Lemma 2.** For each \( q : |x| \), define \( \text{LOC}(q) = \max\{r : q \geq \text{GATE}(r)\} \). Then
\[
\text{KMP}_3(q) = \begin{cases} 
\text{undefined} & \text{if } \text{LOC}(q) \text{ is even}, \\
\text{VAL}(\lfloor \text{LOC}(q) + 1/2 \rfloor) & \text{if } \text{LOC}(q) \text{ is odd}.
\end{cases}
\]

**Lemma 3.** For each \( r : 1 \leq r \leq \ell \), \( \text{VAL}(r) = \text{GATE}(2r - 1)/3 \). (Hence,
\([0, \text{GATE}(2r - 1)]_x \) is of the form \( \text{www} \).

**Lemma 4.** For each \( r : 1 \leq r < \ell \), \( \text{GATE}(2r + 1) \geq 2 \cdot \text{GATE}(2r - 1) \). (Hence,
\( r = O(\log \text{GATE}(2r - 1)) \).
The Implementation

Our implementation of the instances of instructions (5) and (6) will make use of one more lemma proved in [3]:

**Lemma 5.** For each increasing sequence of integers $0 = c_0 < c_1 < c_2 < \ldots < c_m = B$ ("landmarks"), consider a $B$-bounded counter which can be incremented by 1, decremented by 1, tested for 0, tested for $B$, and also tested for (anonymous) membership in $\{c_0, c_1, c_2, \ldots, c_m\}$. Any such counter can be described by the marked concatenation of the binary representations of the successive differences $c_1 - c_0, c_2 - c_1, \ldots, c_m - c_{m-1}$. There is a fixed multihead Turing machine which, given any such description on its tape, can simulate the corresponding counter in real time and in space proportional to the length of the description.

The "original" pattern heads, those used in the algorithm with instructions (5) and (6), will be present in the modified version as well. In addition, there will be some fixed number of additional "auxiliary" pattern heads. Two such heads will be used to mark the two most recently "stored" pattern positions; since $\mathrm{GATE}(0) < \mathrm{GATE}(1) < \ldots$, these heads will have to shift at most $|x|$ times each. Additionally, assuming the most recent pattern position to be "stored" was either $\mathrm{GATE}(2r-1)$ or $\mathrm{GATE}(2r)$, it will suffice somehow to maintain for each original head a counter with landmarks $\mathrm{GATE}(0), \ldots, \mathrm{GATE}(2r-1)$. By Lemma 5, these counters can be simulated in real time using $O(r \cdot \log \mathrm{GATE}(2r-1)) = O(\log^2 \mathrm{GATE}(2r-1))$ bits of space on a multihead Turing machine tape, if we also maintain the marked concatenation of the binary representations.
of $GATE(1) - GATE(0)$, ..., $GATE(2r - 1) - GATE(2r - 2)$. Let $c$ be so large that this tape unit and the marked concatenation require at most $c \cdot \log_2 GATE(2r - 1)$ bits of storage. If $GATE(2r - 1)$ is so small that $c \cdot \log_2 GATE(2r - 1) > GATE(2r - 1)/3$, then the tape unit can be simulated in finite control; otherwise, it will suffice to "borrow" $GATE(2r - 1)/3$ positions from the pattern and to simulate the tape unit with additional auxiliary heads there. (We assume the pattern alphabet has at least two symbols; otherwise, there is a trivial linear-time algorithm.)

The borrowed segment will be $[2 \cdot GATE(2r - 1)/3, GATE(2r - 1)]$. By definition, $[0, GATE(2r - 1)]_x$ has prefix $[VAL(r), GATE(2r - 1)]_x = [GATE(2r - 1)/3, GATE(2r - 1)]_x$; so the correct contents of the borrowed segment will still be available in the segment $[GATE(2r - 1)/3, 2 \cdot GATE(2r - 1)/3]$. An auxiliary head will trail $GATE(2r - 1)/3$ positions behind each original head (reversing direction at the left endmarker, as if it were a "fold", when necessary), to read the correct pattern symbol when the original head is in the borrowed segment. To watch for the latter situation, we station an auxiliary head at each end of the borrowed segment.

Unless $c \cdot \log_2 GATE(2r + 1) > GATE(2r + 1)/3$, some written adjustment is necessary when $GATE(2r + 1)$ has to be "stored"; but time $O(GATE(2r + 1))$ will suffice. At most three pattern segments are involved:

- real = $[GATE(2r - 1)/3, 2 \cdot GATE(2r - 1)/3]$,  
- brw = $[2 \cdot GATE(2r - 1)/3, GATE(2r - 1)]$,  
- brw' = $[2 \cdot GATE(2r + 1)/3, GATE(2r + 1)]$.

(By Lemma 4, these segments do not overlap.) The marked concatenation of
the binary representations of \( GATE(1) - GATE(0) \), \( \ldots \), \( GATE(2r - 1) - GATE(2r - 2) \) must be copied from \( brw \) (or from finite control) into \( brw' \). The binary representations of \( GATE(2r) - GATE(2r - 1) \) and \( GATE(2r + 1) - GATE(2r) \) must be calculated and appended to the marked concatenation. Each counter maintained in \( brw \) (or in finite control) must be emptied into a corresponding new counter to be maintained in \( brw' \). The auxiliary head trailing each original head must be repositioned \( GATE(2r + 1)/3 \) positions behind that head, and the auxiliary heads delimiting \( brw \) must be moved to delimit \( brw' \). Finally, the original contents of \( brw \) must be restored from \( real \). Since the time for this adjustment is \( O(GATE(2r + 1)) \), the total time for all such adjustment is \( O(GATE(1)) + O(GATE(3)) + \ldots + O(GATE(2\ell - 1)) \). By Lemma 4, this sum is \( O(GATE(2\ell - 1)) = O(|x|) \). Q. E. D.

Remark. While the text is being read above, only the pattern segment \( [2 \cdot GATE(2\ell - 1)/3, GATE(2\ell - 1)] \) might not contain its original contents. An additional postprocessing phase, after the entire text has been read, could restore the original contents of this segment from \( [GATE(2\ell - 1)/3, 2 \cdot GATE(2\ell - 1)/3] \) in additional time \( O(|x|) \).

Concluding Remarks

For a number of reasons, the significance of the algorithm described above is strictly theoretical. For all practical purposes, linear-time algorithms described in [3] will never actually require dynamic storage allocation (see [3] for a discussion). Furthermore, our algorithm has embedded in it at least two very complicated Turing machine algorithms.
via Lemma 5 (see [3]). The algorithm was designed to keep its
description simple, not to keep the literal algorithm simple or to keep
the constant multiple in its linear running time small.

It remains open whether a FORTRAN subroutine can (in principle)
perform string-matching in linear time without temporarily modifying its
arguments. We conjecture that a nonwriting multihead finite automaton
cannot perform string-matching in linear time.

REFERENCES

1. M. J. FISCHER and M. S. PATerson, String-matching and other products,
in Complexity of Computation (SIAM-AMS Proceedings, Vol. 7), R. M.
Karp, Ed., American Mathematical Society, Providence, Rhode Island,
1974, pp. 113-125.

2. M. J. FISCHER and A. L. ROSENBERG, Real-time solutions of the
origin-crossing problem, Mathematical Systems Theory, Vol. 2, No. 3,
September 1968, pp. 257-263.

3. Z. GALIL and J. I. SEIFERAS, Saving space in fast string-matching,

matching in strings, SIAM Journal on Computing, Vol. 6, No. 2, June

5. B. L. LEONG and J. I. SEIFERAS, New real-time simulations of
multihead tape units, Journal of the Association for Computing
Machinery, to appear.