MIT/LCS/TM-41

STRING-MATCHING AND OTHER PRODUCTS

Michael J. Fischer

Michael S. Paterson

January 1974

MAC TECHNICAL MEMORANDUM 41

STRING-MATCHING AND OTHER PRODUCTS

Michael J. Fischer

Michael S. Paterson

January 1974

This research was supported in part by the National Science Foundation under research grant GJ-34671; in part by the Artificial Intelligence Laboratory supported by the Advanced Research Projects Agency of the Department of Defense which was monitored by ONR Contract No. N00014-70-A-0362-0003, and in part by the Science Research Council of the U. K.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

STRING-MATCHING AND OTHER PRODUCTS[†]

Michael J. Fischer
Massachusetts Institute of Technology, Cambridge, Massachusetts

and

Michael S. Paterson
University of Warwick, Coventry, England

ABSTRACT

The string-matching problem considered here is to find all occurrences of a given pattern as a substring of another longer string. When the pattern is simply a given string of symbols, there is an algorithm due to Morris, Knuth and Pratt which has a running time proportional to the total length of the pattern and long string together. This time may be achieved even on a Turing machine. The more difficult case where either string may have "don't care" symbols which are deemed to match with all symbols is also considered. By exploiting the formal similarity of string-matching with integer multiplication, a new algorithm has been obtained with a running time which is only slightly worse than linear.

[†]Much of this work was carried out at the University of Warwick with partial support from the Science Research Council of the U.K. It was also supported in part by the National Science Foundation under research grant GJ-34671 to MIT Project MAC, and in part by the Artificial Intelligence Laboratory, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research contract number N00014-70-A-0362-0003.

1. Introduction.

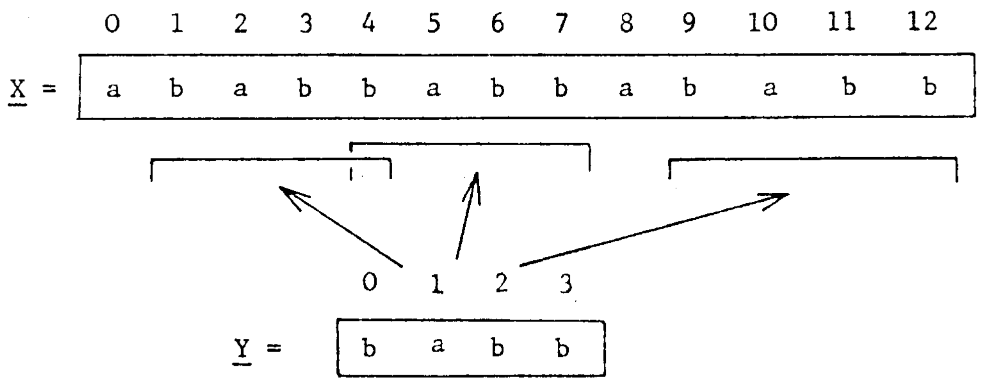
We consider several problems concerned with the matching of strings of symbols. A typical practical problem is that we are given a (long) symbol string $\underline{X} = X_0X_1X_2 \ldots X_m$, the "text", and another (short) string $\underline{Y} = Y_0Y_1 \ldots Y_n$, the "pattern", over the same finite alphabet Σ. The task is to find all occurrences of the pattern as a consecutive substring in the text, that is, to find all i, $n \leq i \leq m$, such that:

$$\underline{Y} = [X_{i-n} \ldots X_i]$$

The obvious naive algorithm tries each i in turn and compares Y_j with X_{i-n+j} for $j = 0,1,\ldots$ as far as necessary, and is represented by the following informal program.

```
        FOR $i = n$ STEP 1 UNTIL $m$
            FOR $j = 0$ STEP 1 UNTIL $n$
                IF $Y_j \neq X_{i-n+j}$ GOTO L
            REPEAT
            PRINT(i)
    L: REPEAT
```

For example with the following strings the desired outputs would be 4, 7, 12.



An upper bound on the computation time for this algorithm is $O(m.n)$ and the matching of $a^m b$ with $a^n b$ shows that this bound is realistic.

2. Morris-Knuth-Pratt algorithm.

A considerable improvement on the naive procedure described above is afforded by an algorithm due to J. H. Morris, D. E. Knuth and V. R. Pratt [4], which has a running time which is $O(m + n)$. The essential idea is that if we have successfully matched a segment of the string $\underline{X}$ with an initial segment of $\underline{Y}$ before reaching an inequality, then it is unnecessary and wasteful to read those symbols of $\underline{X}$ again since they are *the same as the* $\underline{Y}$*-segment*. A better procedure is to carry out the first comparisons for the next relative position of the pattern $\underline{Y}$, by comparing $\underline{Y}$ with a segment of *itself*, and of course the comparisons can be pre-computed once and for all at the beginning. The pre-computation required is very quick and has the same general form as the main computation itself.

We shall describe a "theoreticians' version" of the Morris-Knuth-Pratt algorithm to simplify the presentation and analysis. For a symbol string $\underline{Z} = Z_0 \ldots Z_n$, define the function P for $i = 0,\ldots,n$ by:

$$P(i) = \max \{t \mid \bigwedge_{0 \le r \le t} Z_{i-r} = Z_{t-r} \text{ and } -1 \le t < i\}$$

Provided we consider $\bigwedge_{0 \le r \le -1}$ to be identically <u>true</u>, $P(i)$ is always well-defined. It is not difficult to verify that:

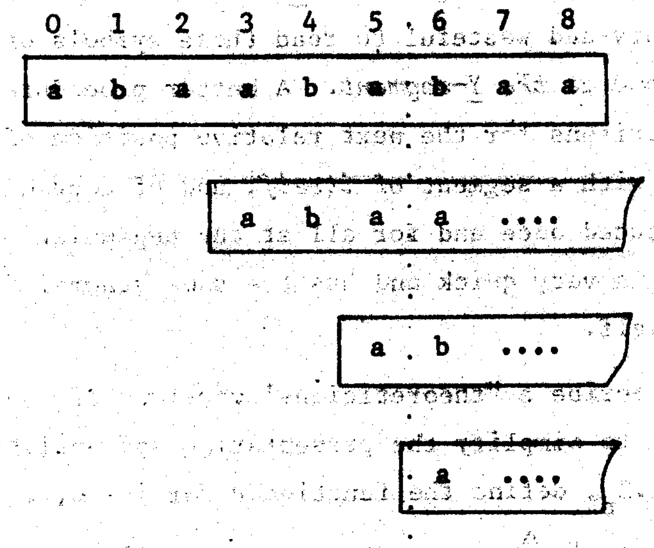
$$P^{(k)}(i) = k^{th} \text{ largest } t \text{ such that } \bigwedge_{0 \le r \le t} Z_{i-r} = Z_{t-r} \text{ and } -1 \le t < i$$
$$\text{if this is defined.}$$

($P^{(k)}(i)$ denotes the composition of P with itself k times, so $P^{(0)}(i) = i$ and $P^{(k+1)}(i) = P(P^{(k)}(i))$.) The usefulness of P results from the following recursive definition.

$$P(0) = -1$$

$$P(i+1) = \begin{cases} P^{(k)}(i) + 1 \text{ for } 0 \le i < n \\ \text{where } k \text{ is the least positive integer such that} \\ \quad Z[P^{(k)}(i) + 1] = Z_{i+1} \\ -1 \text{ if there is no such } k \end{cases}$$

An example is illustrated below.



$P(5) = 2, \qquad P(P(5)) = 0, \qquad P^{(3)}(5) = -1$

$P(6) = P(P(5)) + 1 = 1$

i	0	1	2	3	4	5	6	7	8
P(i)	-1	-1	0	0	1	2	1	2	3

Provided that the string $\underline{Z}$ and the values of $P(j)$ for $j \leq i$ are readily accessible, the value of $P(i+1)$ may be computed in time less than

$$c.(P(i) - P(i+1)+2)$$

for some constant c, independent of i and n. This is because

$$P(j+1)-1 \leq P(j) < j \quad \text{for all } j$$

and hence the k of the recursive definition satisfies:

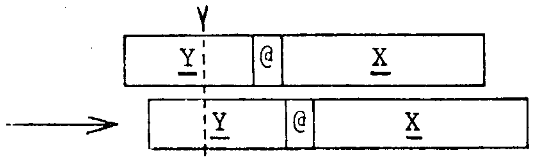
$$P(i) - P(i+1) + 2 \geq k \geq 1$$

Therefore the total running time is bounded by:

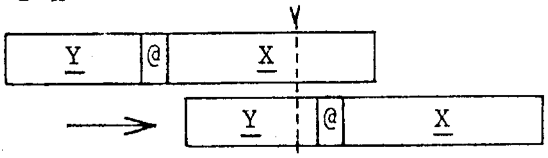
$$c.(P(0) - P(n)) + 2c(n+1) = O(n).$$

To solve our original problem we concatenate $\underline{Y}$, a new symbol @, and $\underline{X}$ in that order and compute the values of P for the string $\underline{Y}$ @ $\underline{X}$ in time $O(m+n)$. Because of the @, P can never take a value greater than $n = |\underline{Y}| - 1$. The values of i for which $P(i) = n$, mark the positions where $\underline{Y}$ matches a substring of $\underline{X}$, or more precisely:

$$P(n+2+i) = n \iff \bigwedge_{0 \leq r \leq n} X_{i-r} = Y_{n-r}$$



Pre-processing phase



Output phase

3. A Turing machine implementation.

The linear time bound obtained in Section 2 for the Morris-Knuth-Pratt algorithm seems to depend not only on the use of a random access machine, but also on the assignment of unit cost to a memory access, for just the P array alone contains $O(n \log n)$ bits when represented as a sequence of binary integers. This makes a linear time Turing machine implementation somewhat surprising.

The central economy results from representing the P array by a table Δ of differences. Define $P(-1) = -1$ and let

$$\Delta(i) = 1 + P(i) - P(i+1), \quad -1 \leq i < n.$$

Then

$$P(i) = i - \sum_{j=-1}^{i-1} \Delta(j)$$

and

$$\sum_{j=-1}^{n-1} \Delta(j) = n - P(n) \leq n+1,$$

so the Δ array can be represented in linear space, even using unary notation.

We may expand the recursive definition of P in Section 2 as follows:

<u>Algorithm X</u>.

Stage (0): Set $P(0) \leftarrow -1$. Go to stage (1,1).

```
Stage $(i+1,\ k)$:
    1. If $Z_{[P^{(k)}(i)+1]} = Z_{i+1}$, set $P(i+1) \leftarrow P^{(k)}(i) + 1$ and go to stage $(i+2,\ 1)$.
    2. If $P^{(k)}(i) = -1$, set $P(i+1) \leftarrow -1$ and go to stage $(i+2,\ 1)$.
    3. Otherwise, go to stage $(i+1,\ k+1)$.
```

Algorithm X may be rewritten without explicit reference to P by using the Δ array and three new variables p, s and d. Inductively, at the beginning of stage (i+1, k), the variables will satisfy:

$$p = P^{(k)}(i) \tag{3.1}$$

$$s = P^{(k)}(i) - P^{(k+1)}(i) \tag{3.2}$$

$$d = P(i) - P^{(k)}(i) \tag{3.3}$$

Algorithm Y maintains these conditions and computes the Δ array. (The column vector notation denotes simultaneous assignment.)

<u>Algorithm</u> <u>Y</u>.

```
Stage $(0)$:
    $\Delta(-1) \leftarrow 1;\ \begin{bmatrix} p \\ s \\ d \end{bmatrix} \leftarrow \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix};$ go to stage $(1,\ 1)$.

Stage $(i+1,\ k)$:
    1. If $Z_{p+1} = Z_{i+1}$, then begin $\Delta(i) \leftarrow d;\ \begin{bmatrix} p \\ s \\ d \end{bmatrix} \leftarrow \begin{bmatrix} p+1 \\ s+\Delta(p) \\ 0 \end{bmatrix};$
       go to stage $(i+2,\ 1)$.
```

2. If $p = -1$, then begin $\Delta(i) \leftarrow d+1$; $\begin{bmatrix} p \\ s \\ d \end{bmatrix} \leftarrow \begin{bmatrix} p \\ s \\ 0 \end{bmatrix}$;

   go to stage $(i+2, 1)$; end.

3. Otherwise, begin

$$\begin{bmatrix} p \\ s \\ d \end{bmatrix} \leftarrow \begin{bmatrix} p - s \\ s - \sum_{j=p-s}^{p-1} \Delta_j \\ d + s \end{bmatrix};$$ go to stage $(i+1, k+1)$; end.

It may be readily verified that conditions (3.1)-(3.3) hold after stage (0), are preserved by the remaining stages, and the Δ's are computed correctly.

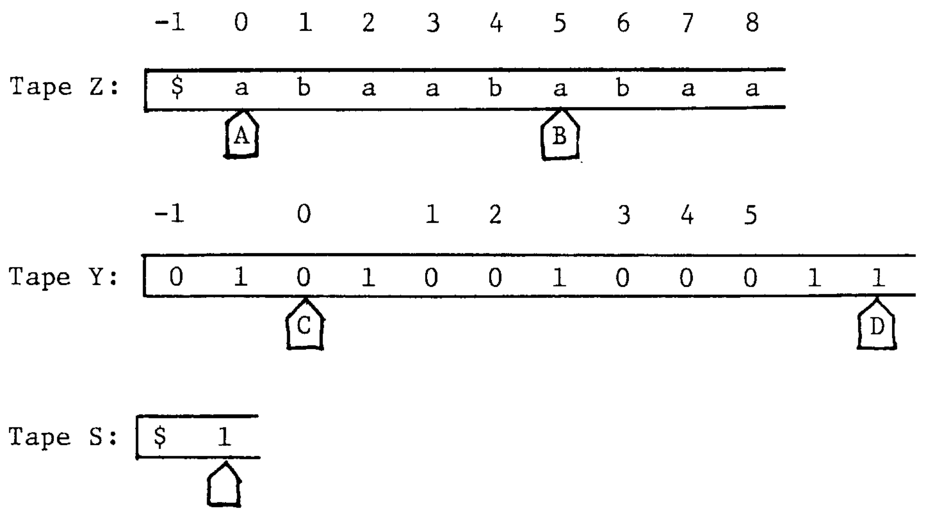
The Turing machine to implement Algorithm Y has four tapes, each one-way infinite to the right. The input tape Z has two heads A and B. Tape Y has two heads C and D and holds the Δ's and d. p is represented by the positions of heads A and C. Tape S is used as a counter and holds s. Tape T is a scratch tape. The tapes with two heads may be replaced without time loss by several tapes with only one head per tape [3].

At the start of stage $(i+1, k)$, head A is scanning Z_p and head B is scanning Z_i. Tape Y contains the binary word

$$01^{\Delta(-1)}01^{\Delta(0)}01^{\Delta(1)}0\ldots01^{\Delta(i-1)}01^{d}$$

head C is on the "0" immediately preceeding the block $1^{\Delta(p)}$ (the $p+2^{nd}$ "0" from the left), and head D is on the rightmost non-blank square. Finally, the counter S contains the number s.

Below are the Turing machine tapes of the example of Section 2 at the beginning of stage (6,2).



We now examine the operations that might be required in stage (i+1, k). The first test, "$Z_{p+1} = Z_{i+1}$", is accomplished in three Turing machine steps, for heads A and B are only one square away from the symbols Z_{p+1} and Z_{i+1}, respectively. Similarly, the test "p = -1" becomes a test to see if head A is scanning the left endmarker, "$".

The updating in case (1) is accomplished by shifting D right and printing a "0", shifting heads A and B right one square, and moving C right to the next "0". As C is advanced, S is incremented once for each "1" that C passes over.

The updating in case (2) is even easier. A and C are left alone, B moves right one square, and D moves right two squares, printing a "1" followed by a "0".

To accomplish case (3), head C is moved left over s zeros. For each "0" passed over by C, head A moves one square to the left and head D moves one square to the right and prints a "1". For each "1" passed over by C, S is decremented. Since the counter S is modified by this process, its contents are first copied into the temporary counter T which is then used to control the iteration.

We total up separately the time spent in each of the three cases. For each i, case (2) is executed during at most one of the stages (i+1, k). Each such execution takes a constant amount of time, so the total over all stages is clearly O(n).

When case (3) is executed at stage (i+1, k), it takes time cs for some constant c, where $s = P^{(k)}(i) - P^{(k+1)}(i)$ is the value of S at the start of the stage, for $\sum_{j=p-s}^{p-1} \Delta_j \leq s$. Let k_i be the largest value of k for which a stage (i+1, k) is executed. Then stages (i+1, 1), ..., (i+1, k_i-1) all execute case (3) and stage (i+1, k_i) executes case (1) or (2). Hence, the total time spent in case (3) from the start of stage (i+1, 1) to the start of stage (i+2, 1) is

$$\sum_{k=1}^{k_i-1} c(P^{(k)}(i) - P^{(k+1)}(i)) = c(P(i) - P^{(k_i)}(i)) \leq c(P(i) - P(i+1) + 1).$$

Summing over all i, the total time in case (3) is O(n).

Finally, the time spent in case (1) is bounded by the number of times C is shifted right. But this is at most the eventual length ℓ_Y of tape Y plus the number of times C is shifted left. The latter occurs only in case (3) and hence is bounded by 0(n). Since

$$\ell_Y = n + 2 + \sum_{j=-1}^{n} \Delta_j < 2n + 3 = 0(n),$$

the total time spent in case (1) is also 0(n).

It follows that the total time of the Turing machine is 0(n).

4. "Don't care" symbols.

An interesting extension of this simple string-matching problem which has practical applications results from the introduction of a "don't care" symbol, ϕ, into the alphabet. ϕ has the property of "matching" with any symbol. We shall write "$\equiv$" for this matching, so

$$\phi \equiv x \quad \text{for all } x \in \Sigma \cup \{\phi\}$$

The Morris-Knuth-Pratt algorithm breaks down in this situation, basically because "$\equiv$" is not a transitive relation, that is:

$$x \equiv y \quad \wedge \quad y \equiv z \;\not\rightarrow\; x \equiv z$$

The above implication is valid only if $y \neq \phi$. Transitivity was assumed implicitly in deriving the recursive relation used to compute P. The naive algorithm given initially works just as before, with "$\equiv$" in place of "=". The ostensible aim of this paper is to produce a more efficient algorithm for string-matching with "don't care" symbols. This is achieved only for the case when Σ is a finite alphabet.

5. Generalised linear products.

Both of the string-matching problems described so far can be regarded as special cases of a very general "linear product". Given two vectors of elements, $\underline{X} = X_0,\ldots,X_m$ and $\underline{Y} = Y_0,\ldots,Y_n$, the *linear product with respect to* $\otimes$ and $\oplus$, written $\underline{X}\boxed{\substack{\otimes\\ \oplus}}\underline{Y}$, is a vector $\underline{Z} = Z_0,\ldots,Z_{m+n}$ where:

$$Z_k = \bigoplus_{i+j=k} X_i \otimes Y_j \qquad \text{for } k = 0,\ldots, m+n$$

For this to be meaningful, X_i, $Y_j \in D$, $Z_k \in E$, for some sets D,E, and $\otimes$ $\oplus$ are functions,

$$\otimes : D \times D \rightarrow E$$
$$\oplus : E \times E \rightarrow E, \quad \oplus \text{ associative}$$

If $\oplus$ is $\wedge$, and $\otimes$ is $=$ or $\equiv$, the middle m-n+1 truth values of the linear product give the information required in matching the text $\underline{X}$ against the *reversal* of $\underline{Y}$, that is $Y_n \ldots Y_0$, since

$$\left(\underline{X}\boxed{\substack{\equiv\\ \wedge}}\underline{Y}\right)_k = \underline{\text{true}} \leftrightarrow [X_{k-n}\ldots X_k] \equiv [Y_n \ldots Y_0]$$

for $n \leq k \leq m$. The reason for introducing general linear products here lies in the following two cases.

(i) *Boolean product* where $\oplus$ is $\vee$ and $\otimes$ is $\wedge$

and (ii) *polynomial product* where $\oplus$ is $+$ and $\otimes$ is $\times$.

The polynomial product is of course the ordinary multiplication of polynomials. The four products with which we are principally concerned are illustrated in Figure 1.

6. Algorithms for linear products.

For the simple string products the Morris-Knuth-Pratt algorithm can be extended to yield the complete linear product. If we append a string of n ϕ's to the end of the text $\underline{X}$, then the same algorithm correctly computes the last n truth-values of the linear product. We are thus computing the values of P for the string $\underline{Y}^R$ @ $\underline{X}$ ϕ^n. For the *first* n elements of the product, we know of no better method than to reverse both strings and use the same procedure.

For strings over a finite alphabet with "don't cares", we follow an indirect course, showing first that the computation time for string product is of the same order as that for Boolean product. If σ, τ are two distinct symbols of Σ, and $\underline{X}$ contains only σ's and ϕ's while $\underline{Y}$ contains only τ's and ϕ's, then the string product of $\underline{X}$ and $\underline{Y}$ is precisely the negation of the Boolean product of the strings $\hat{\underline{X}}$ and $\hat{\underline{Y}}$, where

Continued on Page 15

LINEAR PRODUCT $\underline{Z} = \underline{X} \boxed{\substack{\otimes \\ \oplus}} \underline{Y}$; $Z_k = \bigoplus_{i+j=k} X_i \otimes Y_j$

Examples: with the convention that 1,0 represent <u>true</u>,<u>false</u> respectively.

(1)

$$\begin{array}{ccccccc|c}
 & b & a & a & b & a & & \boxed{\substack{\equiv \\ \wedge}} \\
 & & b & a & a & & & \\
\hline
 & & 0 & 1 & 1 & 0 & 1 & \\
 & 0 & 1 & 1 & 0 & 1 & & \wedge \\
1 & 0 & 0 & 1 & 0 & & & \\
\hline
1 & 0 & 0 & 1 & 0 & 0 & 1 &
\end{array}$$

(2)

$$\begin{array}{ccccccc|c}
 & a & b & \phi & \phi & a & & \boxed{\substack{\equiv \\ \wedge}} \\
 & & a & \phi & b & & & \\
\hline
 & & 0 & 1 & 1 & 1 & 0 & \\
 & 1 & 1 & 1 & 1 & 1 & & \wedge \\
1 & 0 & 1 & 1 & 1 & & & \\
\hline
1 & 0 & 0 & 1 & 1 & 1 & 0 &
\end{array}$$

(3)

$$\begin{array}{cccccccccccc|c}
 & & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & & & \boxed{\substack{\wedge \\ \vee}} \\
 & & & & 1 & 0 & 1 & 0 & 1 & & & & \\
\hline
1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 &
\end{array}$$

(4)

$$\begin{array}{cccccccccccc|c}
 & & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & & & \boxed{\substack{\times \\ +}} \\
 & & & & 1 & 0 & 1 & 0 & 1 & & & & \\
\hline
1 & 0 & 1 & 1 & 1 & 2 & 0 & 3 & 0 & 2 & 0 & 1 &
\end{array}$$

Figure 1.

$$\hat{X}_i = \underline{\text{true}} = 1 \leftrightarrow X_i = \sigma$$
$$\hat{Y}_i = \underline{\text{true}} = 1 \leftrightarrow Y_i = \tau$$

since

$$\bigwedge_{i+j=k} X_i \equiv Y_j \leftrightarrow \bigwedge_{i+j=k} \neg \underline{\hat{X}}_i \vee \neg \underline{\hat{Y}}_j \leftrightarrow \neg \bigvee_{i+j=k} \underline{\hat{X}}_i \wedge \underline{\hat{Y}}_j$$

Thus Boolean product is no harder than ϕ-string product. On the other hand, let H_ρ be the predicate on $\Sigma \cup \{\phi\}$ defined by:

$$\begin{aligned} H_\rho(x) &= 1 \quad \text{if } x = \rho \\ &= 0 \quad \text{if } x \neq \rho \text{ (or } x = \phi) \end{aligned}$$

and extend H_ρ to strings in the obvious way. Then

$$\underline{Z} = \underline{X} \boxed{\substack{\equiv \\ \wedge}} \underline{Y} = \neg \bigvee_{\substack{\sigma \neq \tau \\ \sigma,\tau \in \Sigma}} H_\sigma(\underline{X}) \boxed{\substack{\wedge \\ \vee}} H_\tau(\underline{Y})$$

Informally, this equation states that $\underline{X}$ and $\underline{Y}^R$ match in a given relative position if and only if there is no pair of distinct symbols $\sigma, \tau \in \Sigma$ which clash. Hence the ϕ-string product takes the same time as the Boolean product to within a constant factor, independent of m and n.

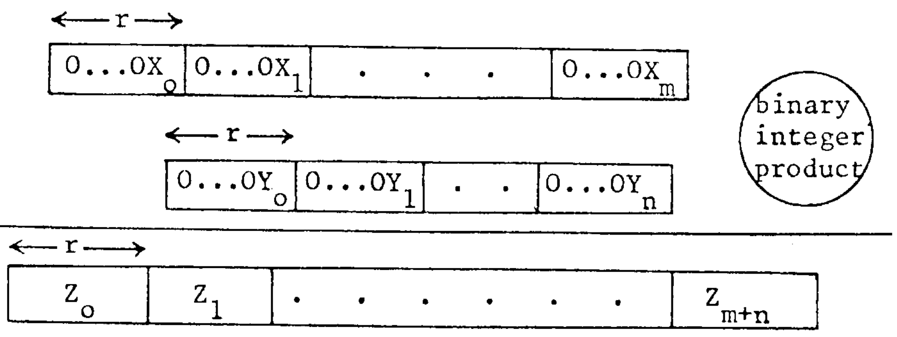
There is a considerable similarity between the Boolean product and the polynomial product over the integers, as is shown in the example above. When 1 and 0 are identified with true and false respectively, the Boolean product can be obtained by performing the polynomial product and then by replacing any non-zero element by 1. This idea of embedding a Boolean algebra in a ring for computational purposes has been exploited to achieve a fast Boolean matrix multiplication and transitive closure algorithm [1].

One very convenient way to compute the polynomial product is to embed the product in a single large integer multiplication, for which there are a variety of well-known efficient algorithms. For the polynomial product of the $\{0,1\}$-strings $X_0,\dots,X_m$ and $Y_0,\dots,Y_n$, where $m \geq n$, the maximum possible coefficient in the product is $n + 1$. If we choose r so that $2^r > n + 1$, compute the integers

$$X(2^r) = \sum_{i=0}^{m} X_i.2^{ri} \text{ and } Y(2^r) = \sum_{j=0}^{n} Y_j.2^{rj}$$

and then multiply $X(2^r)$ by $Y(2^r)$, the result will be the product polynomial Z, evaluated at 2^r. Successive blocks of length r in the binary representation of $Z(2^r)$ will give the coefficients of Z, and by replacing non-zero coefficients by 1 we obtain the elements of the Boolean product. This is illustrated below.

$r > \log_2(n+1)$, $m \geqslant n$



where $\underline{Z} = \underline{X} \boxed{\substack{\times\\+}} \underline{Y}$. $\boxed{\substack{\times\\+}}$ is polynomial product.

The operations required to construct $X(2^r)$ and $Y(2^r)$, and to pick out the coefficients of Z are very easy and efficient on a binary computer. On most computers there is fast special-purpose hardware for multiplication of integers up to a certain size, and efficient routines for multiplying larger integers. These may be used to yield a good practical program for the Boolean product of strings of moderate length, which however has a running time that is still proportional to mn.

For truly large integers, the Schönhage-Strassen algorithm [5] multiplies M-digit numbers by N-digit numbers in a time which is $O(M\cdot\log N \cdot \log\log N)$ for $M \geq N$, using a multi-tape Turing machine. For our application, $M = mr = O(m \log n)$ and $N = nr = O(n \log n)$. Hence,

<u>Result</u>. For a finite alphabet, the "don't care" product of strings of lengths m and n, $(m \geq n)$, can be computed with a multitape Turing machine in time $O(m\cdot(\log n)^2\cdot \log\log n)$.

7. Large alphabets and numbers of comparisons.

The algorithm for ϕ-product described so far has the disadvantage that the running time increases rapidly with the size of the alphabet Σ. It is approximately proportional to $|\Sigma|^2$. By coding the symbols of Σ into a binary alphabet we can use just two Boolean products for strings of length $m.\log|\Sigma|$ and $n.\log|\Sigma|$. Provided $|\Sigma|$ is bounded by a power of n, this introduces a factor of just $\log|\Sigma|$ into the running time.

It is interesting to observe that the Morris-Knuth-Pratt algorithm works for an infinite alphabet, provided we take the predicate "=" as a basic operation. Our algorithm for ϕ-product is not of this form and we may ask whether there is any algorithm, with access to the strings only through the predicate "$\equiv$", which has a computation time better than the obvious $O(m.n)$. Under such a strict limitation the answer is "no", and this is easily seen by considering the product of the two strings $\underline{X} = \phi^{m+1}$ and $\underline{Y} = \phi^{n+1}$. All $\equiv$-tests have the result <u>true</u>, but suppose that during the execution of some algorithm there is some test "$X_i \equiv Y_j$?" which is never made. The computation and output would be indistinguishable from that for the pair of strings $\phi^i \sigma \phi^{m-i}$ and $\phi^j \tau \phi^{n-j}$, where $\sigma, \tau \in \Sigma$ and $\sigma \neq \tau$, and therefore the algorithm cannot correctly compute the string product. Hence any ϕ-product algorithm of this class must sometimes make at least $(m+1)(n+1)$ tests.

The above restriction is perhaps a little severe, even if we consider the case of infinite Σ, so let us allow in addition an explicit test for the "don't care" symbol, that is "$X_i = \phi$?" or "$Y_j = \phi$?". The lower bound on the number of tests is now radically different, for we can show that $O(m+n)$ are sufficient. Unfortunately we still know of no algorithm with a *total* running time less than that of the naive algorithm for the ϕ-product over an infinite alphabet.

8. Algorithm for ϕ-product using $O(m+n)$ tests.

We have to evaluate the $(m+n+1)$ conjunctions

$$Z_k = \bigwedge_{i+j=k} X_i \equiv Y_j \quad \text{for } k = 0,\ldots, m+n$$

Firstly we determine all occurrences of ϕ in $\underline{X}$ and $\underline{Y}$, and replace by <u>true</u> any equivalence involving ϕ. Possibly some of the Z_k's may thus be determined. So far as we know the remaining symbols in $\underline{X}$ and $\underline{Y}$ may be completely distinct. At each stage of the algorithm we shall maintain an equivalence relation on these symbols, such that we have determined that all symbols in the same equivalence class are identical. We can always choose "$X_i \equiv Y_j$?" for our next comparison, where X_i and Y_j are in distinct equivalence classes, and Z_{i+j} has yet to be determined. If $X_i \equiv Y_j$, then the equivalence classes of X_i and Y_j can be united, whereas if $X_i \neq Y_j$ then Z_{i+j} can be determined as <u>false</u>. If during the course of the algorithm the former case occurs (m+n+1) times then only one equivalence class remains, and if the latter case occurs (m+n+1) times then all the Z_k's have been determined. Either way, no further comparisons are required. Hence at most 2(m+n+1) equality tests and m+n+2 ϕ-tests are needed, giving a total which is O(m+n).

9. On-line palindromes.

A computation is performed *on-line* if the i^{th} output symbol is produced before the $i+1^{st}$ input symbol is read. Let $Z_i = 1$ if $X_0 \ldots X_i$ is a palindrome (i.e. if $X_0 \ldots X_i = X_i \ldots X_0$), and $Z_i = 0$ otherwise. Then

$$\underline{Z} = \underline{X} \boxed{\begin{array}{c} = \\ \wedge \end{array}} \underline{X}$$

so $\underline{Z}$ can be computed in time O(n), even on a Turing machine, as outlined in Sections 3 and 6 using the Morris-Knuth-Pratt algorithm.

Fischer and Stockmeyer [2] present a general procedure for converting any off-line multiplication algorithm which runs in time T(n) to an on-line method taking time O(T(n) log n) when T satisfies $T(2n) \geq 2T(n)$. Their construction applies to any generalised linear product, so in particular, the time O(n) method above for computing $\underline{Z}$ can be converted to an on-line Turing machine program that runs in time O(n log n).

10. Conclusions and open problems.

We have considered string-matching problems with and without a "don't care" symbol. In both cases a naive procedure, based directly on the problem definition, takes time proportional to $m \times n$ where m, n are the lengths of the two strings to be matched. The Morris-Knuth-Pratt algorithm provides a practical and elegant way to compute the former problem in $O(m+n)$ time, but there seems to be no obvious extension of their algorithm to the "don't care" case. This is partially explained by our lower bound result which shows that $O(mn)$ is the best possible bound unless more information is allowed than the mere results of comparisons between pairs of symbols. With a further basic test which explicitly detects the "don't care" symbol, this lower bound collapses and there is at least the possibility of a faster algorithm. Provided that the symbol alphabet is finite, we have demonstrated an algorithm with a running time which is $O(m \cdot \log n \cdot \log \log n)$. The method is indirect and not of practical value except for very large m and n, however it shows the feasibility of algorithms which are faster than the naive procedure for "don't care" matching.

We have not treated at all the superficially similar problem, where a "don't care" symbol can "match" an arbitrary *string* of symbols. A good algorithm for this would have obvious practical applications.

We have only begun to compare and contrast the computational complexity of generalised linear products for various $\otimes$ and $\oplus$. There are several more, interesting, structures for which the linear product is a natural operation. A study of algorithms for linear products, based on the axiomatic properties of $\otimes$ and $\oplus$, may provide valuable insight into why some products are easier than others.

References.

[1] M.J. Fischer and A.R. Meyer. "Boolean matrix multiplication and transitive closure", 12th IEEE Symposium on Switching and Automata Theory (1971), 129-131.

[2] M.J. Fischer and L.J. Stockmeyer. "Fast on-line integer multiplication", 5th ACM Symp. on Theory of Computing (1973), 67-72.

[3] P.C. Fischer, A.R. Meyer and A.L. Rosenberg. "Real-time simulation of multihead tape units", Jour. Assoc. Comp. Mach. 19, 4 (October 1972), 590-607.

[4] J.H. Morris and V.R. Pratt. "A linear pattern-matching algorithm", TR-40, Computer Center, Univ. of Calif. Berkeley (June 1970).

[5] A. Schönhage and V. Strassen. "Schnelle Multiplikation grosser Zahlen", Computing 7 (1971), 281-292.

BIBLIOGRAPHIC DATA SHEET	1. Report No.s: GJ-34671 + N00014-70-A-0362-0003 MAC TM-41	2.	3. Recipient's Accession No.

4. Title and Subtitle
String-Matching and Other Products

5. Report Date : Issued January 1974

6.

7. Author(s)
Michael J. Fischer and Michael S. Paterson

8. Performing Organization Rept. No. MAC TM-41

9. Performing Organization Name and Address
PROJECT MAC; MASSACUSETTS INSTITUTE OF TECHNOLOGY:
545 Technology Square, Cambridge, Massachusetts 02139

10. Project/Task/Work Unit No.

11. Contract/Grant Nos: GJ34671 + N00014-70-A-0362-0003

12. Sponsoring Organization Name and Address

Office of Naval Research	Associate Program Director
Department of the Navy	Office of Computing Activities
Information Systems Program	National Science Foundation
Arlington, Va 22217	Washington, D. C. 20550

13. Type of Report & Period Covered : Interim Scientific Report

14.

15. Supplementary Notes
Much of this work was carried out at the University of Warwick with partial support from the Science Research Council of the U. K.

16. Abstracts
The string-matching problem considered here is to find all occurrences of a given pattern as a substring of another longer string. When the pattern is simply a given string of symbols, there is an algorithm due to Morris, Knuth and Pratt which has a running time proportional to the total length of the pattern and long string together. This time may be achieved even on a Turing machine. The more difficult case where either string may have "don't care" symbols which are deemed to match with all symbols is also considered. By exploiting the formal similarity of string-matching with integer multiplication, a new algorithm has been obtained with a running time which is only slightly worse than linear.

17. Key Words and Document Analysis. 17a. Descriptors

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement
Unlimited Distribution
Write Project MAC Publications

19. Security Class (This Report) UNCLASSIFIED

20. Security Class (This Page UNCLASSIFIED

21. No. of Pages 22

22. Price