SIAM J. COMPUT.
Vol. 16, No. 6, December 1987

006

GENERALIZED STRING MATCHING*

KARL ABRAHAMSON†

Abstract. Given a pattern string of length n and an object string of length m, the string matching problem asks for the positions of all occurrences of the pattern in the object string. This paper investigates a generalization of string matching, in which the pattern is a sequence of pattern elements, each compatible with a set of symbols. The alphabet of symbols is infinite, with its members encoded in a finite alphabet. In contrast to standard string matching, which can be solved in simultaneous linear time and constant space, it is shown that generalized string matching requires a time-space product of $\Omega(n^2/\log n)$ on a powerful model of computation, when the alphabet is restricted to n symbols. Our proof uses a method of Borodin. The obvious algorithm for generalized string matching requires time $O(NM)$, where N is the length of the encoding of the pattern, and M is that of the object string. We describe an algorithm which solves generalized string matching in time $O(N+M+mN^{1/2}\,\text{polylog}\,(n))$.

Key words. string matching, regular expressions, time-space tradeoff

AMS(MOS) subject classifications. 68Q25, 68Q05

1. Introduction.

One of the goals of research in the analysis of algorithms is the characterization of subproblems of a general problem according to their complexity. In particular, one would like to know the influence of certain natural features of the input on the complexity of the problem.

This paper investigates the influence of a natural feature of patterns on the complexity of pattern matching in strings. The investigation involves an application of the time-space tradeoff proof technique of Borodin et al. [5], [4], and an application of Fischer and Paterson's fast pattern matching ideas [7].

1.1. Pattern matching in strings.

In abstract form, a pattern matching problem consists of a representation of a set of strings, called the *pattern*, and a single string, called the *file*. The goal is to find all substrings of the file which are members of the set represented by the pattern.

Sets of strings can be described in many ways. For example, one could indicate by a given string the set of strings within some fixed distance (under some measure) of the given string. Such implicit, or approximate, string matching problems are studied in Ukkonen [11] and Landau and Vishkin [10]. Here, we consider more explicitly given sets.

Two common representations for patterns are regular expressions [3] and simple strings (representing singleton sets). This paper considers a class of patterns intermediate between those two. Let Σ be the alphabet used in the file, and Δ be a set of *pattern elements*. Let $\sim \subseteq \Delta \times \Sigma$ be a given relation. If $\delta \sim \sigma$, say that δ and σ are *compatible*.

A *generalized string pattern* is a finite sequence of pattern elements. Pattern $p = p_1 \cdots p_n \in \Delta^*$ represents the set $\{x_1 \cdots x_n \in \Sigma^* : p_i \sim x_i, \text{ for } i = 1, \cdots, n\}$. Simple string patterns are the special case of generalized string patterns where $\Delta = \Sigma$ and $\sim$ is equality.

We consider a specific set Δ, consisting of two kinds of pattern elements: positive and negative. Let $k \geqq 0$ and $\sigma_1, \cdots, \sigma_k \in \Sigma$. A positive pattern element has the form $\langle \sigma_1 \cdots \sigma_k \rangle$, and is compatible with each of $\sigma_1, \cdots, \sigma_k$, and nothing else. A negative pattern element has the form $[\sigma_1 \cdots \sigma_k]$ and is compatible with every member of Σ

* Received by the editors August 26, 1985; accepted for publication (in revised form) January 23, 1987.
† Department of Computer Science, University of British Columbia, Vancouver, British Columbia, Canada.

other than $\sigma_1, \cdots, \sigma_k$. For example, pattern $\langle ab\rangle[\ ][a]$ matches any string of length 3 beginning with a or b and not ending with a.

As a convenience, Σ is presumed to contain a distinguished symbol ϕ, which is compatible with every pattern element. Symbol ϕ will not explicitly be written in pattern elements.

It is an important property of both simple string and regular expression patterns that arbitrarily large alphabets can be dealt with, by encoding into a fixed alphabet. Unwanted matches which begin in the middle of an encoded symbol can be avoided by a suitable choice of encoding. Such encoding is possible for the simple reason that equality is a congruence relation.

But $\sim$ is not necessarily a congruence relation. In order to permit encoding of arbitrarily large alphabets, it is necessary to treat the file alphabet Σ not as a primitive alphabet, but as a set of strings which encode its members. Throughout this paper, Σ is presumed to be the infinite set $\{\phi, a_1, a_2, \cdots\}$, where a_i is represented by the string $\#\bar{\imath}$, and $\bar{\imath}$ is the binary representation of i, without leading zeros. Symbol ϕ is represented by itself.

Some terminology will aid in the discussion of string matching. The word *symbol* will be reserved for members of Σ. A member of Δ is called a pattern element. If symbol σ occurs in pattern element δ, write $\sigma \in \delta$. Relation $\in$ should not be confused with relation $\sim$. For example, $a \in [ab]$. A *character* is a member of the finite alphabet used for encoding symbols and pattern elements, including $\{\#, 0, 1, \phi\}$ and brackets.

Throughout, the pattern is $p = p_1 \cdots p_n$ and the file is $f = f_1 \cdots f_m$. $\Sigma_p \subseteq \Sigma$ is the set of symbols occurring in p. Both p and f have more than one associated length measure, depending on the level of detail at which they are viewed. Throughout, n denotes the number of pattern elements in p, N the number of characters in p, and $\hat{N}$ the number of symbols in p. For example, if $p = \langle \#1\#10\rangle\langle \#11\rangle$, then $n = 2$, $\hat{N} = 3$, and $N = 12$. Similarly, $m \geqq n$ denotes the number of symbols in f, and M denotes the number of characters in f.

An *alignment* consists of n consecutive symbols of f, and is referred to by its final symbol position in f. A *match*, or *matching alignment*, is an alignment which is in the set represented by p.

1.2. The model of computation. The standard models for string matching are finite state machines, multi-tape Turing machines, and unit and logarithmic cost random access machines [3]. Each has its drawbacks. Finite state machines are too weak. Turing machines do not permit efficient search structures, such as binary search trees, or efficient representation of graphs. Unit cost RAMs permit algorithms which build up large numbers, unrealistically performing arithmetic on them at unit cost. Logarithmic cost RAMs introduce a logarithmic factor where it is unreasonable to have one, needlessly obscuring analysis. For example, the obvious algorithm to count the length of the input takes $O(n \log n)$ time on a length n input. A model is introduced below which circumvents these problems.

Our algorithms are for a version of random access machines, called time-bounded-magnitude random access machines (TBM-RAMs). Such a machine is a unit cost RAM with one modification. No cell, including the accumulator, may contain a number which exceeds the number of steps executed so far. If a program attempts to generate a number in excess of the time counter, the machine spins its wheels until a time is reached when the number may be stored. Any polynomial function of time would serve equally well as a bound on the magnitude of cells. But typical RAM programs use less space than time, so the TBM-RAM model is suitable.

The input to a TBM-RAM is presumed to be given in a separate, read-only memory, one character per cell. The output is written into a special write-only output memory. Input and output space are not counted in space complexity.

TBM-RAMs provide a compromise between the unit cost and logarithmic cost RAMs, permitting the simplicity and realism of the unit cost RAM for reasonable programs, without permitting abuse of the privilege of assigning unit cost to each instruction. TBM-RAMs simulate, with at most a constant factor loss, the models usually used for string matching, including multitape Turing machines, and random access machines as they are typically employed. Hence, they provide a suitable framework for string matching algorithms.

Some of the numbers generated by our algorithms are very large. Such numbers are not intended to be stored in a single cell; doing so would cause the algorithms to take exponential time on a TBM-RAM. Instead, large numbers are stored in many cells, one bit per cell. It will be apparent, without need for comment, which numbers are large enough to warrant special treatment.

1.3. Summary of results. Pattern matching with simple string patterns is referred to as standard string matching. Standard string matching is solvable in linear time (in $N+M$) [3], [6], [8], [9]. Galil and Seiferas [8] show that standard string matching can, in fact, be solved in simultaneous linear time and constant space.

We have no nontrivial lower time bound for generalized string matching. It is nevertheless possible to prove that, in terms of simultaneous time and space complexity, generalized string matching is harder than standard string matching. Our first result is an $\Omega(n^2/\log n)$ lower bound on the product of time and space complexities of any algorithm for a small subproblem of generalized string matching, for which $m=2n$. For $\log n \leqq S \leqq n$, the subproblem admits space $O(S)$, time $O(n^2 \log (n)/S)$ algorithms.

Regular expression matching is known to be solvable in time $O(NM)$ [3], and no faster algorithm is known. Our second result is an $O(N+M+m\hat{N}^{1/2}\,\text{polylog}\,(n))$ time algorithm for generalized string matching. The existence of such an algorithm is strong circumstantial evidence that generalized string matching is easier than regular expression matching. In the absence of good lower bounds on regular expression matching, no more can be said.

The remainder of this paper is organized as follows. The time-space tradeoff result is proved in § 2. In §§ 3–6, three algorithms for generalized string matching are developed, each building on the ideas of the previous one. Algorithm A is a practical, but quadratic time, algorithm. Algorithm B runs in close to linear time when Σ is finite, but takes more than quadratic time for infinite Σ. Algorithm C is the promised sub-quadratic time algorithm.

2. A time-space tradeoff. This section is concerned with a subproblem of generalized string matching, called the all-out-of-place problem. For the size n problem, the pattern is fixed as $[a_1][a_2]\cdots[a_n]$. The file has $2n$ symbols, which must be among $\{a_1, \cdots, a_n\}$.

We will show that any time T, space S algorithm for the all-out-of-place problem must satisfy $T \cdot S = \Omega(n^2/\log n)$. The algorithm of Galil and Seiferas solves standard string matching over the same alphabet $\{a_1, \cdots, a_n\}$ (suitably encoded) in time $O(N+M) = O(n \log n)$ and constant space; that is, a time-space product of $O(n \log n)$. So, in terms of time-space product, generalized string matching is more difficult than standard string matching.

The all-out-of-place problem actually exhibits a tradeoff. For every $\log n \leqq S \leqq n$, there is a space $O(S)$, time $O(n^2 \log (n)/S)$ algorithm.

2.1. An algorithm demonstrating the tradeoff. We want to find all matching alignments of pattern $[a_1] \cdots [a_n]$ in $f_1 f_2 \cdots f_{2n}$. Notice that each symbol in f is incompatible with exactly one of the pattern elements. In fact, knowing that $f_i = a_j$ is enough to determine one mismatching alignment, namely the position ending at f_{i+n-j}. Suppose we are willing to use $O(n)$ space. Then keep a bit vector MISMATCH (k), for $k = n, \cdots, 2n$. Initialize the vector to all zeros, and then scan the input $f_1 \cdots f_{2n}$, setting MISMATCH $(i+n-j) = 1$ for each $f_i = a_j$ and $0 \leqq i-j \leqq n$. Then the matching alignments are those k such that MISMATCH (k) is 0 at the end.

Suppose we have only $S < n$ bits of space available for the MISMATCH vector. Then simply make $\lceil n/S \rceil$ passes over the input, storing a different section of the MISMATCH vector during each pass. At the end of each pass, output the match positions found during that pass. Each pass requires $O(n \log n)$ time to read f, character by character. The total time required is $O(n^2 \log n/S)$.

We now turn to the lower bound. The lower bound proof technique was introduced by Borodin et al. [5], and has been applied to sorting [5], [4] and other problems [1], [13].

2.2. A model of computation. The lower bound proof involves a nonuniform model of computation, similar to the familiar comparison tree model of sorting programs, and the string matching model of Yao [12]. Following Borodin et al., algorithms in the nonuniform model are called *branching programs.* For the all-out-of-place problem, the pattern is built into the program, and the input is the file, of fixed length $2n$.

The model provides powerful queries. Each node contains a number i, $1 \leqq i \leqq 2n$, representing the query "what is the ith symbol of the input?" There are n arcs exiting the node, each labeled by a symbol. See Fig. 1. The ability of a branching program to extract, in a single query, the value of an entire symbol partially explains the discrepancy between the upper bound of $O(n^2 \log n)$ and the lower bound of $\Omega(n^2/\log n)$.

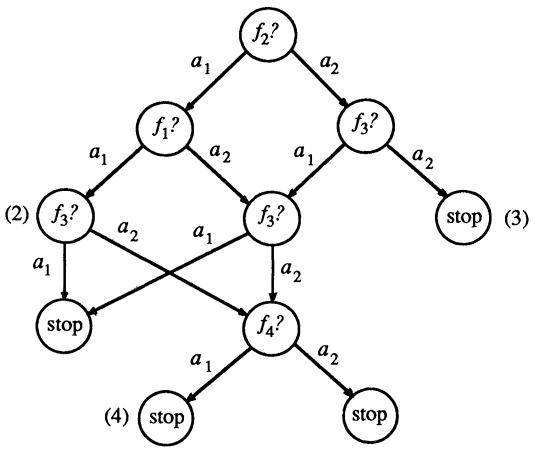

FIG. 1. *A branching program for pattern* $[a_1][a_2]$, $m = 4$.

At various points, an algorithm may conclude that there is a match at some particular alignment. The model provides a means for reporting matching alignments. Each node may be labeled by a collection of output numbers, representing the ending locations of matches. In Fig. 1, outputs are in parentheses beside nodes. Output of the selected matching alignments occurs when the program enters the node. There is

no provision for giving mismatch answers. Instead, the mismatch answers are implicit. A program is correct if, for each input, it outputs all of the matching alignments.

Programs are not restricted to trees, but may be arbitrary directed graphs. One of the nodes is designated the start node, and some are designated stop nodes. The time complexity of a program is the number of queries in the longest computation.

The analogue to space complexity of a branching program is its *capacity.* The capacity of a branching program is the log (base 2) of the number of nodes in the program. Notice that, as is commonly done, we are not counting read-only input space or write-only output space as part of an algorithm's space requirement. We are interested in the amount of temporary storage that an algorithm needs.

Space on TBM-RAMs and capacity of branching programs are related by the following lemma, whose proof is elementary.

LEMMA 2.1. *Suppose there exists a TBM-RAM program P which solves every size n instance of a string matching problem X in time at most T and space at most S. Then there is a branching program for the size n instances of X of time complexity at most T and capacity at most* $S \log T + c$, *where c depends only on P, not on S, T or n.*

Sketch of proof. Let the nodes of the branching program correspond to instantaneous descriptions of the TBM-RAM. The factor of $\log T$ in the capacity accounts for the capacities of individual cells. □

It is convenient to make some assumptions about the structure of branching programs. Say that a branching program is in *normal form* if (1) its nodes can be divided into *levels*, such that the start node is in level 0, and each arc goes from a node in level i to one in level $i+1$, for some i; and (2) all of the stop nodes are in the same level.

Note that a normal form program must be acyclic, and its time complexity is just the largest level number. Any branching program can be put into normal form, by the conceptual addition of a clock which is included in the state space and is incremented at each step. Doing so adds $\log T$ to the capacity of a program of time complexity T. For our purposes, T is polynomial in n, so the added capacity is $O(\log n)$. From now on, we will assume that all branching programs are in normal form. Our results only hold for capacities of at least $\log n$.

Tree programs are a useful restriction of branching programs. Note that a tree program is in normal form if all of its leaves are at the same level. The *depth* of a tree program is its time complexity. Say that a tree program is *nonredundant* if no computation asks the same query twice. Given a depth $q \leqq n$ tree program for a size n problem, one can easily find an equivalent nonredundant, normal form, depth q tree program by simply eliminating redundant queries and then adding worthless but nonredundant queries to pad all paths to the same length.

2.3. Two technical lemmas. Before proving the main theorem, we need two technical lemmas. The first concerns (directed) paths in branching programs. We are only concerned with programs for the all-out-of-place problem. Paths are defined in the usual way; a path of length q has q edges and $q+1$ nodes.

The edges along a path of length q are labeled by the responses to the q queries contained in all but the last node in the path. Say that an input string f is *query-consistent* with a path π provided f is consistent with the responses in π. For example, if π contains the response "a_3" to query "f_9?," then f is query-consistent with π only if $f_9 = a_3$.

The nodes on a path π may have outputs associated with them. Say that an input f is *output-consistent* with π provided the outputs given by π are correct for input f.

For example, if π contains the output 34, then f is output-consistent with π only if f matches $[a_1]\cdots[a_n]$ ending on f_{34}.

The *success set* of a path π is the set of inputs with which π is both query-consistent and output-consistent. Say that π is a *k-path* if it gives at least k outputs. A path is *nonredundant* if it does not contain two nodes with the same query.

LEMMA 2.2. *Any nonredundant k-path of length q has a success set of size at most* $(n-k(n-q)/(2n-q))^{2n-q}$.

Proof. The path makes exactly q distinct queries, and receives the values of q input symbols. There remain $2n-q$ *unqueried* input symbols.

Each output declares a match of $[a_1]\cdots[a_n]$ at some alignment in the input. At least $n-q$ of the n matched positions of each output must be unqueried. Therefore, each output restricts the success set in at least $n-q$ unqueried positions. Specifically, one symbol is forbidden in each position.

Notice that two overlapping matches restrict the input in disjoint ways. For example, a match beginning at the third input symbol f_3 requires that f_6 not be a_4, while a match beginning at f_5 requires that f_6 not be a_2. Altogether, the k outputs must forbid at least $k(n-q)$ symbol occurrences, where a symbol occurrence is a pair (symbol, input position).

A product of positive numbers $x_1x_2\cdots x_r$ is maximized subject to $x_1+\cdots+x_r\leqq c$ where the x_i are all equal. It follows that the success set has maximum size when the $k(n-q)$ forbidden symbol occurrences are distributed uniformly among the $2n-q$ unqueried positions. In that case, there are at most $n-k(n-q)/(2n-q)$ allowable symbols in each unqueried position. There is only one allowable symbol in each queried position, its value determined by the response to the query. The total number of inputs which are both query-consistent and output-consistent with the path is therefore at most $(n-k(n-q)/(2n-q))^{2n-q}$. □

The second technical lemma concerns the frequency of inputs which have many matching alignments. One would expect matches of the pattern $[a_1]\cdots[a_n]$ to be common. The following lemma meets that expectation. Say that a length $2n$ input string is *admissible* if it matches $[a_1]\cdots[a_n]$ in at least $n/10$ alignments.

LEMMA 2.3. *There are at least* $n^{2n}/6$ *admissible inputs, for all* $n\geqq 2$.

Proof. The probability that a randomly chosen input matches $[a_1]\cdots[a_n]$ at a given alignment is $((n-1)/n)^n$, which is at least $1/4$ for $n\geqq 2$. Therefore, the total number of matches, summed over all inputs and all alignments, is at least $(n/4)n^{2n}$. Suppose a fraction r of the inputs match $[a_1]\cdots[a_n]$ in less than $n/10$ alignments. The remaining fraction $1-r$ match in at most n alignments, since a little thought reveals that there must be at least one mismatch for every input (consider the value of f_n). Thus $(rn/10+(1-r)n)n^{2n}\geqq(n/4)n^{2n}$, from which it follows that $r\leqq 5/6$. So at least $1/6$ of the n^{2n} inputs match in $n/10$ or more alignments. □

2.4. A time capacity tradeoff.

THEOREM 2.4. *There is a constant* $c>0$ *such that, for* $S\geqq\log n\geqq 1$, *every time T and capacity S branching program for the all-out-of-place problem satisfies* $S\cdot T\geqq cn^2$.

Proof. Consider a normal form branching program P which solves the all-out-of-place problem in time T and capacity S. Divide P into disjoint *stages*, each containing $q+1<T$ consecutive levels. See Fig. 2. If necessary, pad $T+1$ to a multiple of $q+1$; q will be chosen sufficiently small that the padding is insignificant. Obtain a new program P' by unwinding P within each stage, duplicating shared nodes, so that each stage becomes a collection of disjoint depth q trees. By a leaf of such a tree, we mean a node at the deepest level in a stage. Modify the trees so that each is nonredundant, and every path from a root to a leaf has length exactly q.

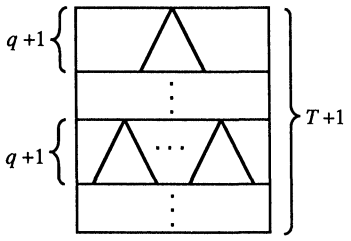

FIG. 2. *A staged branching program.*

In what follows, by a path in P', we mean a path from the root to a leaf of one of the trees just constructed. Each tree has exactly n^q paths, all nonredundant. There are at most 2^S trees in P', since the root of each tree corresponds to a distinct node in P. Hence, there are at most $2^S n^q$ paths in P'.

Imagine running P' on an admissible input. P' must give at least $n/10$ outputs, and there are only $(T+1)/(q+1) < T/q$ stages, so some stage must give at least $nq/10T$ outputs. Hence, one of the paths in one of the stages is an $(nq/10T)$-path. Indeed, for every admissible input, there is an $(nq/10T)$-path which is followed during computation on that input. The paths need not all be distinct, since one path may serve many inputs. Let $h_1, \cdots, h_r$ be all of the $(nq/10T)$-paths in P'.

Let s_i be the success set of h_i, for $i = 1, \cdots, r$. Since each admissible input must have an associated h_i, Lemma 2.3 ensures that

$$|s_1| + \cdots + |s_r| \geqq \frac{n^{2n}}{6}. \tag{2.1}$$

But each h_i is a nonredundant $(nq/10T)$-path of length q. By Lemma 2.2, for $i = 1, \cdots, r$,

$$|s_i| \leqq \left(n - \frac{nq(n-q)}{10T(2n-q)}\right)^{2n-q}. \tag{2.2}$$

Combine and simplify inequalities (2.1) and (2.2), to obtain

$$rn^{-q}\left(1 - \frac{q(n-q)}{10T(2n-q)}\right)^{2n-q} \geqq \frac{1}{6}.$$

But $r \leqq 2^S n^q$, since there are only that many paths in P'. Let $q = n/2$. Then

$$2^S\left(1 - \frac{n}{60T}\right)^{3n/2} \geqq \frac{1}{6}.$$

Taking logs and applying the inequalities $\log(1-x) \leqq -x \log e$ (for $0 \leqq x < 1$) and $T \geqq n$ gives

$$S - \frac{cn^2}{T} \geqq -\log 6 \tag{2.3}$$

for some constant $c > 0$. There are two cases. If $2T \log 6 \geqq cn^2$ then T itself is sufficiently large, and we are done. If $2T \log 6 < cn^2$, then inequality (2.3) implies $ST \geqq cn^2/2$. □

COROLLARY 2.5. *Any TBM-RAM program for the all-out-of-place problem which solves every size n instance in time at most $T(n)$ and space at most $S(n)$ satisfies $S(n) \cdot T(n) = \Omega(n^2/\log n)$.*

3. Overhanging matches. Until now, we have considered only full matches. It is convenient to find *overhanging* matches as well. A left-overhanging match is a suffix of the pattern which matches a prefix of the file. A right-overhanging match is a prefix of the pattern which matches a suffix of the file.

There are two advantages to finding overhanging matches: (a) They provide a means of understanding what is going on at early stages of string matching algorithms, which tend to begin by looking for left-overhanging matches; (b) The file can be broken up into blocks of n symbols and the string matching algorithm run independently on each block. Right-overhanging matches of block b_i are easily combined with left-overhanging matches of block b_{i+1} to obtain full matches for $b_i b_{i+1}$.

Our algorithms find both left- and right-overhanging matches. In order to include overhanging matches in the definition of matching, extend file $f_1 \cdots f_m$ to $f_{1-n} \cdots f_{m+n-1}$, where, for $i<1$ and $i>m$, f_i is presumed to be ϕ.

4. Algorithm A. This section presents some basic definitions and lemmas, leading to an algorithm for generalized string matching which, although of quadratic time complexity, is quite practical and includes some of the basic ideas of later algorithms. This section contains no proofs, due to the simplicity of the results.

Algorithm A is based on the ideas underlying the Knuth, Morris and Pratt [9] string matching algorithm, without the failure function computation. Extend the relation $\sim$ to strings in the obvious way. Define the collection of sets

$$R_k = \{j\colon 1 \leqq j \leqq n \text{ and } p_1 \cdots p_j \sim f_{k-j+1} \cdots f_k\},$$

for $k = 0, \cdots, m$. Inspection of the definition of R_k reveals that the alignment ending on f_k matches p if and only if either $0 \leqq k \leqq m$ and $n \in R_k$ or $m < k \leqq m+n-1$ and $m+n-k \in R_m$. Thus, it suffices to compute the sets R_k for $k = 0, \cdots, m$. The following definitions and proposition provide a means of computing R_k. For $\sigma \in \Sigma$, and S a set of natural numbers let

$$S+1 = \{x+1\colon x \in S\},$$

$$M(\sigma) = \{j\colon 1 \leqq j \leqq n \text{ and } p_j \sim \sigma\}.$$

PROPOSITION 4.1.

(a) $R_0 = \{1, \cdots, n\}$,

(b) $R_k = ((R_{k-1}+1) \cup \{1\}) \cap M(f_k)$ *for* $k > 0$.

The sets R_k must be represented somehow. A natural representation of a set $S \subseteq \{1, \cdots, n\}$ is the n-bit binary integer $\text{rep}(S) = \sum_{i \in S} 2^{i-1}$. Let $r_k = \text{rep}(R_k)$ and $\mu(\sigma) = \text{rep}(M(\sigma))$. Then Proposition 4.1 implies the following.

PROPOSITION 4.2.

(a) $r_0 = 2^n - 1$,

(b) $r_k = (2 \cdot r_{k-1}+1)\ \&\ \mu(f_k)$ *for* $k > 0$,

where & *is the bitwise Boolean* <u>*and*</u> *operation on binary representations.*

Algorithm A computes r_k, for $k = 0, \cdots, m$, by the recurrences of Proposition 4.2. Doing so requires $O(m)$ arithmetic and Boolean operations on n bit numbers, plus the operations required to compute $\mu(f_k)$, for $k = 1, \cdots, m$.

Compute the μ values as follows. Construct a search structure for $\sigma \in \Sigma_p$, such that σ can be found in time proportional to the length of its encoding. In one pass over p, construct representations, for each $\sigma \in \Sigma_p$, of the sets

$$P^- = \{j\colon p_j \text{ is a negative element}\},$$

$$M^+(\sigma) = \{j\colon \sigma \in p_j \text{ and } p_j \text{ is a positive element}\},$$

$$M^-(\sigma) = \{j\colon \sigma \in p_j \text{ and } p_j \text{ is a negative element}\}.$$

Then compute $\mu(\sigma)$, for each $\sigma \in \Sigma_p$, according to

$$\mu(\sigma) = \text{rep}\,(M^+(\sigma)) \cup (P^- \cap \overline{M^-(\sigma)}).$$

The total time required is $O(N + n|\Sigma_p|)$. Whenever $\mu(\sigma)$ is needed, it is looked up in the search structure. If σ is not found, then $\sigma \notin \Sigma_p$, and $\mu(\sigma) = \text{rep}\,(P^-)$.

On a TBM-RAM, Algorithm A takes time $O(M + N + nm + n|\Sigma_p|)$. In subsequent sections, asymptotically faster Algorithms B and C are developed. But, in spite of its quadratic time complexity, Algorithm A is the most practical of the three algorithms for the following reason.

In typical applications, patterns containing more than a few dozen pattern elements are rare. But the numbers involved in Algorithm A are just n bits long. Since arithmetic and Boolean operations on 32 bit numbers can typically be done in a single fast instruction, it is reasonable to presume that arithmetic and Boolean operations take constant time. In that case, Algorithm A runs in time $O(N+M)$, with a reasonably small constant hidden in the "O."

5. Algorithm B. Fischer and Paterson [7] describe an algorithm for a subproblem of generalized string matching: string matching with don't cares. In our notation, their algorithm permits patterns of two forms, $\langle\sigma\rangle$ and [ ], for $\sigma \in \Sigma$. This section applies Fischer and Paterson's ideas to generalized string matching.

Just as Algorithm A is based on the sets R_k, Algorithm B is based on a collection of functions. For $i = n+1, \cdots, n+m-1$, define p_i to be [ ]. For $k = 0, \cdots, m$ and $i = 1, \cdots, m+n-1$, let

$$G_k(i) = |\{j\colon 1 \leqq j \leqq i \text{ and } p_j \not\sim f_{k+j-i}\}|,$$

$$\nu_k(i) = \begin{cases} 0 & \text{if } p_i \sim f_k, \\ 1 & \text{if } p_i \not\sim f_k. \end{cases}$$

$G_k(i)$ counts the number of incompatible pairs of symbols when $p_1 \cdots p_i$ is aligned with $f_{k-i+1} \cdots f_k$. An analogue to Proposition 4.1 is easily obtained for G_k.

PROPOSITION 5.1.

(a) $G_0(i) = 0$ *for* $i = 1, \cdots, m+n-1$,

(b) $G_k(1) = \nu_k(1)$ *for* $k = 0, \cdots, m$,

(c) $G_k(i) = G_{k-1}(i-1) + \nu_k(i)$ *for* $k > 0$ *and* $i > 1$.

G_k can be viewed as an alternate representation of R_k, since, for $1 \leqq i \leqq n$, $G_k(i) = 0$ if and only if $i \in R_k$. G_k can itself be represented by an integer g_k. Let d be the smallest power of 2 larger than n, and let

$$g_k = \sum_{i=1}^{m+n-1} G_k(i) d^{i-1}, \qquad \mu_k = \sum_{i=1}^{n} \nu_k(i) d^{i-1}.$$

The value μ_k is merely an encoding, in base d, of the set $M(f_k)$ of § 4. Although the encoding is different from that of § 4, notably in polarity, both are named μ. An analogue to Proposition 4.2 follows from Proposition 5.1.

PROPOSITION 5.2.

(a) $g_0 = 0$,

(b) $g_k = d \cdot g_{k-1} + \mu_k$ *for* $k > 0$.

Note that g_k encodes a vector of integers, and the integer addition in part (b) simulates component-wise vector addition. The choice of $d > n$ ensures that carries cannot cross component boundaries.

An obvious algorithm computes g_k, for each k, according to Proposition 5.2. It turns out, however, that it is enough to compute the single quantity g_m. Notice that

G_m counts incompatible pairs when a prefix of p is aligned with a suffix of f; that is, G_m looks for right-overhanging matches. But *every* match of $p_1 \cdots p_n$ in $f_1 \cdots f_m$ is a right-overhanging match of $p_1 \cdots p_n[\ ] \cdots [\ ]$ in $f_{1-n} \cdots f_m$. So p matches the alignment ending on f_k, for $k = 1, \cdots, m+n-1$, if and only if $G_m(m+n-k) = 0$. Thus, all of the desired information can be extracted from g_m. From Proposition 5.2,

$$g_m = \mu_m + \mu_{m-1}d + \mu_{m-2}d^2 + \cdots + \mu_1 d^{m-1}. \tag{5.1}$$

The value of μ_k depends only on the value of f_k. If $f_k = \sigma$, say that $\mu_k = \mu_\sigma$. Any two symbols σ, $\tau \notin \Sigma_p$ are equivalent w.r.t. pattern p, so $\mu_\sigma = \mu_\tau$. Define μ^- to be the value of μ_σ for $\sigma \notin \Sigma_p$. If a symbol σ occurs more than once in f, then μ_σ occurs more than once in the r.h.s. of equation (5.1). Let $I_\sigma = \{k: f_k = \sigma\}$ and $I^- = \{k: f_k \notin \Sigma_p\}$. Collect like coefficients together to obtain

$$g_m = \sum_{\sigma \in \Sigma_p} \mu_\sigma \sum_{k \in I_\sigma} d^{m-k} + \mu^- \sum_{k \in I^-} d^{m-k}. \tag{5.2}$$

The procedure described in § 4 can be used to compute μ^- and μ_σ, for all $\sigma \in \Sigma_p$. Because μ_σ is an $n \log d \approx n \log n$ bit number, the computation time climbs from $O(N + |\Sigma_p| n)$ to $O(N + |\Sigma_p| n \log n)$.

Equation (5.2) includes $|\Sigma_p| + 1$ terms of the form $\Sigma_k\, d^{n-k}$. Each represents an $m \log d \approx m \log n$ bit number. They can all be computed, in a single pass over f, in time $O(M + (|\Sigma_p| + 1)m \log n)$.

Now equation (5.2) involves $|\Sigma_p| + 1$ multiplications of $n \log n$ by $m \log n$ bit numbers. Algorithm B uses the Schönhage–Strassen integer multiplication algorithm [2], [9], which multiplies an r bit number by an s bit number, $r \leqq s$, in time $O(s \log r \log \log r)$. The time required to carry out the $|\Sigma_p| + 1$ multiplications is therefore $O((|\Sigma_p| + 1)m \log^2 n \log \log n)$.

THEOREM 5.3. *Algorithm* B *solves generalized string matching in time* $O(N + M + (|\Sigma_p| + 1)m \log^2 n \log \log n)$.

For fixed finite Σ, Algorithm B runs in time $O(m \log^2 n \log \log n)$. But for infinite Σ, $|\Sigma_p|$ may be as large as $\hat{N}$, the number of symbol occurrences in p. In that case, Algorithm B is worse than Algorithm A.

6. Algorithm C. A common technique in algorithm design is to break a problem into a few subproblems of the same kind, each exhibiting different characteristics, and to apply to each subproblem an algorithm tailored to that subproblem's characteristics. That is the approach taken in designing Algorithm C.

Two methods of breaking a generalized string matching problem into subproblems are used. The first breaks up strings according to alphabets. For $f \in \Sigma^*$, $p \in \Delta^*$, and $\Gamma \subseteq \Sigma$, let $f(\Gamma)$ be the string obtained from f by replacing each symbol in $\Sigma - \Gamma$ by ϕ and let $p(\Gamma)$ be the pattern obtained by erasing every occurrence of every symbol in $\Sigma - \Gamma$ from p. For example, $\langle a_1 a_2 a_3 \rangle [a_1 a_4](\{a_1, a_2\}) = \langle a_1 a_2 \rangle [a_1]$. Then the matching alignments of p in f are just those alignments where $p(\Gamma)$ matches $f(\Gamma)$ and $p(\Sigma - \Gamma)$ matches $f(\Sigma - \Gamma)$.

A second breakdown divides the pattern into positive and negative parts. Let p^+ be the pattern obtained from p by replacing each negative pattern element by $[\ ]$. Similarly, obtain p^- from p by replacing each positive element by $[\ ]$. Then the matching alignments of p in f are precisely those alignments where both p^+ and p^- match.

Algorithm B performs badly when $|\Sigma_p|$ is close to $\hat{N}$. But in that case, most of the symbols must occur infrequently in p. An obvious idea is to break the problem up according to frequency of occurrence of symbols in p. Let z be a function of $\hat{N}$ and

n, to be determined later. Define

$$\Sigma' = \{\sigma \in \Sigma_p : \sigma \text{ occurs at most } z \text{ times in } p\},$$
$$\Sigma'' = \Sigma - \Sigma'.$$

The matching alignments of p in f are just those where $p(\Sigma'')$ matches $f(\Sigma'')$, $p^-(\Sigma')$ matches $f(\Sigma')$ and $p^+(\Sigma')$ matches $f(\Sigma')$. Each subproblem is solved separately.

(1) To find the matches of $p(\Sigma'')$ in $f(\Sigma'')$, use Algorithm B. The time required is $O(M+N+(|\Sigma''|+1)m \log^2 n \log\log n)$. But each member of Σ'' occurs at least z times in p. Since p contains only $\hat{N}$ symbols, $|\Sigma''| \leqq \hat{N}/z$. So part (1) takes time $O(M+N+(\hat{N}/z)m \log^2 n \log\log n)$.

(2) Pattern $p^-(\Sigma')$ contains only negative pattern elements. Each occurrence of a symbol σ in $f(\Sigma')$ is responsible for at most z mismatches, one for each occurrence of σ in p. In time $O(N)$, a list of occurrence positions of every member of Σ' in p can be constructed. Then, in time $O(M+zm)$, all of the mismatch location can be recorded, in a bit array.

(3) For brevity, let p^+ denote $p^+(\Sigma')$ and f denote $f(\Sigma')$. The matching alignments of p^+ in f can be found by counting, for each $k = 1, \cdots, m+n-1$, the size of the set $\{j: p_j^+ \not\sim f_{k-n+j}\}$; a count of zero indicates a match.

Notice that incompatible pairs can only occur when a positive element of p^+ is aligned with a non-ϕ symbol of f. Let $\pi(k,j)$ stand for the statement "p_j^+ is positive and $f_{k-n+j} \neq \phi$," and $\alpha(k,j)$ stand for "$p_j^+ \sim f_{k-n+j}$." Then

$$\begin{aligned}|\{j: p_j^+ \not\sim f_{k-n+j}\}| &= |\{j: \neg\alpha(k,j) \text{ and } \pi(k,j)\}| \\ &= |\{j: \pi(k,j)\}| - |\{j: \alpha(k,j) \text{ and } \pi(k,j)\}|.\end{aligned}$$

The term $|\{j: \alpha(k,j) \text{ and } \pi(k,j)\}|$ can be computed by a method very similar to that used for subproblem 2. Where the algorithm for p^- records a mismatch, the algorithm for p^+ increments a counter. There are at most zm pairs (k,j) satisfying $\alpha(k,j)$ and $\pi(k,j)$, so the time required is $O(N+M+zm)$.

Term $|\{j: \pi(k,j)\}|$ can be computed as follows. Form pattern q from p^+ by replacing each positive element by $\langle\ \rangle$. Note that q has only two distinct elements, $\langle\ \rangle$ and $[\ ]$. Form file g by replacing each non-ϕ symbol in f by some fixed symbol $c \neq \phi$. Then run Algorithm B on pattern q and file g. The output is a count, for each alignment of q in g, of the number of incompatible pairs. But that is precisely $|\{j: \pi(k,j)\}|$. Since $|\Sigma_q| = 0$, Algorithm B takes time $O(N+M+m \log^2 n \log\log m)$, which is low order compared to the cost of the rest of the algorithm.

Putting all of the parts together, Algorithm C runs in time $O(N+M+zm+(\hat{N}/z)m \log^2 n \log\log n)$.

THEOREM 6.1. *When* $z = \hat{N}^{1/2} \log n \log\log^{1/2} n$, *Algorithm* C *solves generalized string matching in time*

$$O(N+M+m\hat{N}^{1/2} \log n \log\log^{1/2} n) = O(N+MN^{1/2} \log N \log\log^{1/2} N).$$

Algorithm C is far from being practical. Its purpose is merely to show that generalized string matching does not require $\Omega(NM)$ time. Both Algorithms A and B achieve less than order NM time for some classes of inputs: Algorithm A is better for $n \ll N$ or $m \ll M$, and Algorithm B is better when $|\Sigma_p|$ is small. But Algorithm C is faster than order NM for all sufficiently large N and M, without assumptions about the input.

7. String matching with don't cares. In the case where patterns are restricted to $[\ ]$ and $\langle\sigma\rangle$, for $\sigma \in \Sigma$, there is a faster algorithm than Algorithm C. Suppose all symbols

have the same encoded length K, and suppose the code for σ is $x_1 \cdots x_K$. Then $\langle\sigma\rangle$ can be replaced by $\langle x_1\rangle \cdots \langle x_K\rangle$, [ ] can be replaced by [ ] $\cdots$ [ ] (K times), and matching can be done at the level of characters. This is essentially the technique employed by Fischer and Paterson [7]. The time required to perform the matching, by Algorithm B, is $O(Km \log^2 (Kn) \log\log (Kn))$.

As the string matching problem has been formulated, the input is not given in a constant length code. But, in time $O(M+N)$, it is possible to convert p and f to a constant length code. By breaking f into blocks of length n, as discussed in § 3, and using a different code for each block, it is possible to get by with a coded length of $K = \lceil \log n \rceil$. The total time, including recoding and matching, is $O(M+N+m \log^3 n \log\log n)$.

8. Conclusion and open problems. Our time-space tradeoff result provides a separation between standard and generalized string matching, on the basis of time-space product complexity. The time-space product lower bound is the best lower bound known for generalized string matching. A nonlinear lower bound on time would represent a more clear separation between standard and generalized string matching. Alternatively, a linear or nearly linear time algorithm would greatly increase the significance of the time-space tradeoff. The existence of a linear time algorithm for generalized string matching remains open.

Generalized string patterns represent a particular feature of patterns, namely, that they consist of a concatenation of sets of individual symbols. Do patterns with less strict features admit subquadratic time algorithms? Regular expressions with the equivalent of negative pattern elements represent an obvious generalization of generalized string patterns. Other obvious ones are: (a) patterns representing sets of uniform length strings; (b) star-free regular expressions.

Generalized string patterns with only positive pattern elements can be viewed as star-free regular expressions, in which concatenation operations may not occur within the scope of union operators. Aho and Corasick [2] describe a linear time algorithm for matching with star-free expressions in which union operators may not occur within the right scope of concatenation operators. Although our algorithm and theirs are incompatible, the existence of both algorithms suggests that there might be a subquadratic time algorithm for pattern matching with arbitrary star-free regular expressions.

REFERENCES

[1] K. ABRAHAMSON, *Time-space tradeoffs for branching programs contrasted with those for straight-line programs*, in Proc. 27th Annual IEEE Symposium Foundations of Computer Science, 1986, pp. 402-409.

[2] A. V. AHO AND M. J. CORASICK, *Efficient string matching: an aid to bibliographic search*, Comm. Assoc. Comput. Mach., 18 (1975), pp. 333-340.

[3] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[4] A. BORODIN AND S. COOK, *A time-space tradeoff for sorting on a general sequential model of computation*, in Proc. 12th Annual ACM Symposium Theory of Computing, 1980, pp. 294-301.

[5] A. BORODIN, M. J. FISCHER, D. G. KIRKPATRICK, N. A. LYNCH AND M. TOMPA, *A time-space tradeoff for sorting on non-oblivious machines*, J. Comput. System. Sci., 22 (1981), pp. 351-364.

[6] R. S. BOYER AND J. S. MOORE, *A fast string searching algorithm*, Comm. Assoc. Comput. Mach., 20 (1977), pp. 762-772.

[7] M. J. FISCHER AND M. S. PATERSON, *String matching and other products*, in Complexity of Computation, R. M. Karp, ed., (SIAM-AMS Proceedings 7), American Mathematical Society, Providence, RI, 1974, pp. 113-125.

[8] Z. GALIL AND J. SEIFERAS, *Time-space optimal string matching*, in Proc. 13th Annual ACM Symposium Theory of Computing, 1981, pp. 106–113.

[9] D. E. KNUTH, J. H. MORRIS, JR. AND V. R. PRATT, *Fast pattern matching in strings*, this Journal, 6 (1977), pp. 323–350.

[10] G. M. LANDAU AND U. VISHKIN, *Efficient string matching with k mismatches*, Theoret. Comput. Sci., 43 (1986), pp. 239–249.

[11] E. UKKONEN, *Finding approximate patterns in strings*, J. Algorithms, 6 (1985), pp. 132–137.

[12] A. C. YAO, *The complexity of pattern matching for a random string*, Technical Report STAN-CS-77-629, Dept. Computer Science, Stanford University, Stanford, CA, 1977.

[13] Y. YESHA, *Time-space tradeoffs for matrix multiplication and discrete Fourier transform on any general sequential random access computer*, J. Comput. System Sci., 29 (1984), pp. 183–197.