Efficient randomized pattern-matching algorithms

by Richard M. Karp
Michael O. Rabin

We present randomized algorithms to solve the following string-matching problem and some of its generalizations: Given a string *X* of length *n* (the *pattern*) and a string *Y* (the *text*), find the first occurrence of *X* as a consecutive block within *Y*. The algorithms represent strings of length *n* by much shorter strings called *fingerprints*, and achieve their efficiency by manipulating fingerprints instead of longer strings. The algorithms require a constant number of storage locations, and essentially run in real time. They are conceptually simple and easy to implement. The method readily generalizes to higher-dimensional pattern-matching problems.

0. Introduction

Text-processing systems must allow their users to search for a given character string within a body of text. Database systems must be capable of searching for records with stated values in specified fields. Such problems are instances of the following string-matching problem: For a specified set $\{\langle X(i), Y(i)\rangle\}$ of pairs of strings, determine, if possible, an r such that $X(r) = Y(r)$. Usually the set is specified not by explicit enumeration of the pairs, but rather by a rule for computing the pairs $\langle X(i), Y(i)\rangle$ from some given data.

We present a randomized algorithm to solve this problem. The algorithm associates with each string X a fingerprint $\phi(X)$ which is much shorter than the string itself. The search for a match then compares short fingerprints instead of long strings.

The algorithm selects the *fingerprint function* ϕ at random from a family of easy-to-compute functions. No matter which input $\{\langle X(i), Y(i)\rangle\}$ is presented, the algorithm is unlikely to produce a *false match*, in which two fingerprints agree even though the original strings do not.

The most widely studied pattern-matching problem is the following: Given a *pattern* X of length n and a *text* Y of length $m \geq n$, find the first occurrence of X as a consecutive substring of Y. Several linear time algorithms have been given for this problem. The algorithms of Knuth, Morris, and Pratt [1] (KMP in the sequel) and of Boyer and Moore [2] require, for fast implementation, $O(n)$ registers to store a table of pointers. The characters of the text Y can come in a stream and require no storage. But for fast implementation it is useful to have portions of Y in main memory. The algorithm of Galil and Seiferas [3] requires only $O(\log n)$ registers. Recently Galil and Seiferas [4] have found a real-time algorithm using a constant number of registers.

Our method, based on fingerprint functions, runs essentially in real time; the exact meaning of this statement is spelled out in Theorem 4. It requires a constant number of registers and needs a substring of length n of the text in main memory. One version, described by Algorithm 3, runs strictly in real time but allows a provably minuscule probability of error.

A considerable advantage of our algorithms is that they produce the same theoretical time bounds as the deterministic algorithms and require a competitive or

smaller number of registers. At the same time they are conceptually very simple and consequently easy to program. For the classical linear string-matching case, they become practically competitive only for rather long patterns, say $n = 200$. For such long patterns it would seem that storing pointer tables in registers is not feasible, so the classical algorithms would slow down.

Our methods also apply when the pattern is a multidimensional rectangular array of symbols or even an irregularly shaped arrangement of symbols. In such applications our time bounds are superior to those previously known [5, 6]. Baker [5] has a linear-time algorithm for d-dimensional arrays, but it requires substantially more storage than our method. Bird [7] also has a linear-time algorithm for two-dimensional arrays with a storage requirement comparable to ours. Neither Baker's method nor Bird's applies to irregular shapes.

This work is a contribution to the growing body of literature on randomized algorithms [8] and provides further evidence of the efficacy of algorithms that flip coins.

1. An example

In order to familiarize the reader with our approach and demonstrate the extreme simplicity of the algorithm, let us describe one version for the case of linear string matching. Let the pattern and text be, respectively,

$$X = x_1 x_2 \cdots x_n, \qquad x_i \in \{0, 1\},$$

$$Y = y_1 y_2 \cdots y_m, \qquad y_j \in \{0, 1\}.$$

The restriction to the $\{0, 1\}$ alphabet is just for convenience. Let $Y(i) = y_i y_{i+1} \cdots y_{i+n-1}$; then a match occurs if $X = Y(i)$. Define

$$a = x_1 2^{n-1} + \cdots + x_n,$$

$$a(i) = y_i 2^{n-1} + \cdots + y_{i+n-1}, \qquad 1 \le i \le m - n + 1.$$

The occurrence of a match is obviously equivalent to $a = a(i)$. For integers b, c let $\mathrm{res}(b, c)$ denote the residue of b when divided by c. Note that if $0 \le r_1, r_2 < p$, then $\mathrm{res}(r_1 + r_2, p) = r_1 + r_2$ if $r_1 + r_2 < p$, and $\mathrm{res}(r_1 + r_2, p) = r_1 + r_2 - p$ otherwise.

Let p be a *randomly chosen* prime in the range $[1, nm^2]$. Denote $\mathrm{res}(b, p) = \bar{b}$, and let σ_p denote the operation $\sigma \bmod p$ for $\sigma = +, -, \cdot$. The algorithm starts by computing

$$\bar{a} = (x_1 \cdot_p 2 +_p x_2) \cdot_p 2 +_p x_3 \cdots$$

and $\overline{a(1)}$. The computation is done in real time with a fixed number of computer operations per bit of X and bit of Y as they are read in (two adds, two comparisons, and up to two subtractions per bit).

At the ith step, $1 \le i \le m - n$, we have $\bar{a}$ and $\overline{a(i)}$. If $\bar{a} \ne \overline{a(i)}$, compute $\overline{a(i+1)}$ by

$$\overline{a(i+1)} = (\overline{a(i)} -_p \overline{2^{n-1}} \cdot_p y_i) \cdot_p 2 +_p y_{i+n},$$

and test whether $\bar{a} = \bar{a}(i+1)$.

If, for some i, $\bar{a} = \overline{a(i)}$, then test whether $X = Y(i)$ using bit-by-bit comparison. If a match is found, stop. Otherwise choose a new random $p < nm^2$ and reinitialize the search at place $i + 1$ in the text.

It is shown that for *every* X and Y, this randomized algorithm runs in expected time $m + c$. It is "nearly real-time." Another version, employing simultaneously several randomly chosen primes, runs in real time.

We require auxiliary storage to keep p, $\bar{a}$, $\bar{a}(i)$, $\overline{2^{n-1}}$, i, where i is the pointer to the current place in Y.

We want to apply this basic idea to several pattern-matching problems and use a variety of fingerprint functions. It is therefore useful to develop a general framework into which the above and all the other algorithms will fit as special cases. This is done in the next section.

2. The general string-matching algorithm

In this section we establish a general framework that can be specialized to yield several particular string-matching problems and algorithms. An instance of the *general string-matching problem* is specified by

- Positive integers n and t.
- An index set R of cardinality t.
- For each $r \in R$, strings $X(r)$ and $Y(r)$ in $\{0, 1\}^n$.

The problem is to decide whether there exists an index τ such that $X(r) = Y(r)$ and, if so, to find one such index.

Particular string-matching problems lead to particular choices of R and particular rules for determining $X(r)$ and $Y(r)$ from the input data and the index r. We indicate two examples.

- *The linear pattern-matching problem*

This is the familiar problem treated in Section 1: Given two strings, a *pattern* X and a *text* Y, determine whether X occurs as a consecutive block within Y. Suppose $X = x_1 x_2 \cdots x_n$ and $Y = y_1 y_2 \cdots y_m$, where each x_i and each y_j is a 0 or a 1, and $m \ge n$. We take $t = m - n + 1$, $R = \{1, 2, \cdots, m - n + 1\}$, $X(r) = X$ for all r, and $Y(r) = y_r y_{r+1} \cdots y_{r+n-1}$.

- *Two-dimensional array matching*

This is the problem of determining whether a two-dimensional array of 0s and 1s occurs as a block within a larger array. For notational convenience we take the arrays to be square. Let $X = (x_{ij})$ be an $s \times s$ array of 0s and 1s, and let $Y = (y_{ij})$ be a $m \times m$ array of 0s and 1s, where $m \ge s$. The problem is to determine whether there is a pair $\langle k, l \rangle$ with $s \le k \le m$ and $s \le l \le m$ such that $x_{s-i,s-j} = y_{k-i,l-j}$ for all i and j such that $0 \le i \le s - 1$ and $0 \le j \le s - 1$. In other words, we are looking for an $s \times s$

subsquare of the text that exactly matches the pattern. To fit this problem within the general framework, we may take $n = s^2$, $t = (m - s + 1)^2$, and $R = \{\langle k, l\rangle \mid s \le k \le m$ and $s \le l \le m\}$. The string $X(\langle k, l\rangle)$ is obtained by concatenating together the rows of X, and $Y(\langle k, l\rangle)$ is obtained by concatenating together the rows of the $s \times s$ block within Y having its lower right-hand corner in the $\langle k, l\rangle$ position.

A simple, straightforward method of solving a string-matching problem is to impose a total ordering on the index set R, and then march through R, testing, for each index r, whether $X(r) = Y(r)$. This is the method actually used in many text-processing systems, and there are many situations in which it is the method of choice. If n, the length of the strings being compared, is very small, then the time to compare two strings is small, and the method is quite effective. If one can assume that the strings being compared are random strings of 0s and 1s, then it takes only two bit comparisons, on the average, to establish that $X(r) \neq Y(r)$, and so the method is again highly effective. But when n is large and we are unwilling to make any assumptions about the input data, the straightforward method may be unacceptable, since nt bit comparisons are required in the worst case.

We present a general approach to string matching which may sometimes have advantages over both the straightforward method mentioned above and some of the more sophisticated and theoretically efficient methods that have been proposed. Let S be a finite set and, for each $p \in S$, let $\phi_p(\cdot)$ be a function from $\{0, 1\}^n$ into a range D_p. The value $\phi_p(X)$ can be viewed as a "fingerprint" of the string X. The algorithm will compute one or more fingerprints of each string, and will compare $X(r)$ with $Y(r)$ only if the corresponding fingerprints of the two strings agree. The fingerprints thus serve as a preliminary filter which is highly likely to establish that $X(r) \neq Y(r)$, if indeed these two strings are unequal. Since the fingerprints are much shorter than the original strings, this screening process is likely to be advantageous.

The idea of using fingerprinting techniques for string-matching problems is not new. Many such techniques based on check sums and hash functions can be found in the literature. What is new is the particular way of choosing the fingerprinting functions at random at run time. This randomization technique permits us to establish very strong properties of our algorithms, even if the input data are chosen by an intelligent adversary who knows the nature of the algorithm.

We give three different randomizing algorithms based on the fingerprinting technique. For brevity, let $a_p(r)$ and $b_p(r)$ denote $\phi_p(X(r))$ and $\phi_p(Y(r))$, respectively. Assume that the index set R is totally ordered. Let α be the first element of R and let ω be an "end marker" that follows the last element of R. Finally, for $r \in R$, let r' denote the successor of r in $R \cup \{\omega\}$.

Algorithm 1

```
var match:boolean; r:member of R; k:positive integer;
begin
    for i := 1 to k do p_i := randomly chosen element of
    S;
    match := false; r := α;
    while match = false and r ≠ ω do
        begin
        if a_{p_i}(r) = b_{p_i}(r) for i = 1, 2, ···, k
        then match := true;
        r := r'
        end
end
```

Algorithm 2

```
var match:boolean; r:member of R;
begin
    p := randomly chosen element of S;
    match := false; r := α;
    while match = false and r ≠ ω do
        begin
        if a_p(r) = b_p(r)
        then if X(r) = Y(r) then match := true;
        r := r'
        end
end
```

Algorithm 3

```
var match:boolean; r:member of R;
begin
    p := randomly chosen element of S;
    match := false; r := α;
    while match = false and r ≠ ω do
        begin
        if a_p(r) = b_p(r)
        then   if X(r) = Y(r)
                   then match := true;
                   else p := randomly chosen element of S
               r := r'
        end
end
```

In comparing these algorithms, the concept of a *false match* is essential. A false match is said to occur in Algorithm 1 if, for some r such that $X(r) \neq Y(r)$, the algorithm determines that $a_{p_i}(r) = b_{p_i}(r)$ for all $i = 1, 2, \cdots, k$. Similarly, in Algorithms 2 and 3, a false match occurs if, for some r, the algorithm determines that $a_p(r) = b_p(r)$ but $X(r) \neq Y(r)$.

Algorithm 1 computes k fingerprinting functions for each string $X(r)$ or $Y(r)$. As soon as the fingerprinting functions indicate a match, the algorithm reports that a match has occurred and halts. Algorithms 2 and 3 compute only one

fingerprinting function. If the fingerprinting function indicates that $X(r)$ and $Y(r)$ may be equal, these algorithms then test whether $X(r)$ and $Y(r)$ are actually equal and, if not, continue scanning the input. Algorithm 3 has the additional feature that a new fingerprinting function is selected whenever a false match occurs.

Algorithm 1 never backs up over the data. Thus, it is the method of choice for a hardware implementation, or in any situation where the input data are streaming past an input terminal in an on-line fashion. Under reasonable assumptions it is a real-time algorithm (i.e., it dwells for a constant number of steps or each bit of its input), and it lends itself to parallel computation since the k fingerprinting functions can be computed independently. In this algorithm a false match, if it occurs, will go undetected, and thus the algorithm may erroneously report a match. However, because of the randomization in the choice of the fingerprinting functions, the probability of such an error can be reduced to a truly negligible level; moreover, this will be true uniformly, regardless of how the input data are chosen.

Algorithms 2 and 3 always give a correct result and, in the absence of a match or false match, they also run in real time. The time required to verify matches and to detect and recover from false matches also contributes to their running time. Since each of these algorithms makes a random choice of fingerprinting functions, the running time of each is a random variable even for a fixed input. We show that, uniformly for all inputs, each of these algorithms can be made to run in linear expected time. Moreover, we show that the probability of a catastrophe, in the form of an exceptionally long series of false matches, is negligible. Algorithm 3, which hedges against catastrophe by changing the fingerprinting function whenever a false match occurs, is especially safe in this respect. The advantages of such hedging are demonstrated in Section 5.

In support of the above claims, we show that, for certain classes of string-matching problems, the following three properties can be achieved simultaneously:

1. For all $p \in S$, $\log_2 |D_p| \ll n$, where n is the common length of the strings $X(r)$ and $Y(r)$, and D_p is the range of values of ϕ_p; i.e., the fingerprints of the strings in question can be represented much more compactly than the strings themselves.
2. For *every* particular problem instance, there is only a small probability that a false match will occur.
3. It is easy to compute $a_p(r')$ from $a_p(r)$ and $b_p(r')$ from $b_p(r)$; i.e., fingerprints are easy to update.

All three properties depend on the choice of a family of fingerprinting functions. Property 3 also depends on specific details of the string-matching problem being considered and on the total ordering of the index set R.

3. A family of modular fingerprint functions

A binary string $X = x_1x_2 \cdots x_n$ can be regarded as a binary representation of the integer

$$H(X) = \sum_{i=1}^{n} x_i 2^{n-i}.$$

For any integer p, the function $H_p(X) = H(X) \bmod p$ is a possible fingerprint function. Let M be a positive integer to be specified later. Define $S = \{p \mid p \text{ is prime and } p \le M\}$, and $\phi_p(X) = H_p(X)$ for all X.

A random prime in the range $[1, M]$ can be selected by repeatedly choosing random integers in that range, testing each for primality, and halting when a prime is found. The expected number of trials is approximately $\ln M$. The time to perform each primality test is $O((\log M)^2)$ if we use the probabilistic algorithms of Rabin [9] or Solovay and Strassen [10]. It is possible for these algorithms to incorrectly identify a composite number as prime, but the probability of such an error can be reduced to a completely negligible level. The effects of such a rare mishap are insignificant if we use Algorithm 3, which discards p as soon as a false match occurs.

To study the properties of the family $\{H_p\}$ of fingerprint functions based on primes, we require some number-theoretic definitions and lemmas. Let $\pi(u)$ denote the number of primes $\le u$.

Lemma 1
If $u \ge 29$, then the product of the primes $\le u$ is $> 2^u$.

Proof Theorem 18 of [11] states that the product of the primes $\le u$ is $> \exp(u - 2.05282u^{1/2})$. This inequality established the result for $u \ge 49$, and the result can be verified by direct computation for $29 \le u < 49$. □

Corollary 1
If $u \ge 29$ and $a \le 2^u$, then a has fewer than $\pi(u)$ different prime divisors.

Proof Suppose a has more than $\pi(u)$ prime divisors, and let these be $q_1 \cdots q_r$. We obtain the contradiction

$2^u \ge a \ge q_1q_2 \cdots q_r \ge$ the product of the first r primes $\ge$ the product of the first $\pi(u)$ primes = the product of the primes less than or equal to $u > 2^u$.

Lemma 2 (Rosser and Schoenfeld [11])
For all $u \ge 17$,

$$\frac{u}{\ln u} \le \pi(u) \le 1.25506 \frac{u}{\ln u}.$$

Theorem 3
If Algorithm 2 or Algorithm 3 is executed with $S = \{p \mid p \le M \text{ and } p \text{ prime}\}$, then, for every instance $\{\langle X(r), Y(r)\rangle, r \in R\}$, the probability that a false match

occurs is

$$\leq \frac{\pi(nt)}{\pi(M)}, \text{ provided } nt \geq 29.$$

Proof For a fixed input $\{\langle X(r), Y(r)\rangle, r \in R\}$ and any prime p, occurrence of a false match when Algorithm 2 or Algorithm 3 is executed with ϕ_p as the initially chosen fingerprint function is equivalent to each of the following statements:

1. For some $r \in R$, $a_p(r) = b_p(r)$ but $X(r) \neq Y(r)$.
2. For some $r \in R$ such that $X(r) \neq Y(r)$, $p \mid H(X(r)) - H(Y(r))$.
3. $p \mid \prod_{\{r \mid X(r) \neq Y(r)\}} \mid H(X(r)) - H(Y(r)) \mid$.

For each r, $|H(X(r)) - H(Y(r))| < 2^n$. Hence

$$\prod_{\{r \mid X(r) \neq Y(r)\}} |H(X(r)) - H(Y(r))| < 2^{nt}.$$

By Corollary 1, the product has at most $\pi(nt)$ prime divisors. Thus p is chosen at random from $\pi(M)$ primes, of which at most $\pi(nt)$ lead to a false match. It follows that, for a random choice of p, the probability of a false match is at most $(\pi(nt))/(\pi(M))$. □

Theorem 4
If Algorithm 1 is executed with $S = \{p \mid p \leq M \text{ and } p \text{ prime}\}$ and $\phi_p = H_p$, then, for every instance $\{\langle X(r), Y(r)\rangle, r \in R\}$, the probability that a false match occurs is

$$\leq \left(\frac{\pi(nt)}{\pi(M)}\right)^k \text{ provided } nt \geq 29$$

and

$$\leq t\left(\frac{\pi(n)}{\pi(M)}\right)^k \text{ provided } n \geq 29.$$

Proof A false match occurs only if each of the initially chosen primes $p_1, p_2, \cdots, p_k$ divides $|H(X(r)) - H(Y(r))|$ for some r such that $X(r) \neq Y(r)$. This implies that each of these primes divides

$$\prod_{\{r \mid X(r) \neq Y(r)\}} |H(X(r)) - H(Y(r))|.$$

Since this product is $<2^{nt}$, the number of primes that divide it is $\leq \pi(nt)$, provided $nt \geq 29$. Hence, the probability that p_1 divides this product is

$$\leq \frac{\pi(nt)}{\pi(M)},$$

and since the p_i are drawn independently at random from the primes dividing M, the probability that all k of the p_i divide this product is

$$\leq \left(\frac{\pi(nt)}{\pi(M)}\right)^k.$$

This proves the first inequality.

Since $|H(X(r)) - H(Y(r))| < 2^n$, the number of primes dividing $|H(X(r)) - H(Y(r))|$ is $\leq \pi(n)$, provided $X(r) \neq Y(r)$ and $n \geq 29$. Hence, the probability that the randomly chosen primes $p_1, p_2, \cdots, p_k$ all divide $|H(X(r)) - H(Y(r))|$ is

$$\leq \left(\frac{\pi(n)}{\pi(M)}\right)^k,$$

and the probability that this occurs for some $r \in R$ is

$$\leq t \cdot \left(\frac{\pi(n)}{\pi(M)}\right)^k.$$

This proves the second inequality. □

Corollary 4(a)
If Algorithm 2 or Algorithm 3 is executed with S equal to the set of primes $\leq nt^2$ and $\phi_p = H_p$, then, for every instance of the input data n, t, $\{\langle X(r), Y(r)\rangle, r \in R\}$ such that $nt \geq 29$, the probability that a false match occurs is $\leq 2.511/t$.

Proof Apply Lemma 2 to bound $\pi(nt)$ from above and $\pi(nt^2)$ from below. ■

Corollary 4(b)
If Algorithm 1 is executed with S equal to the set of primes $\leq nt^2$, then, for every instance of the input data n, t, $\{\langle X(r), Y(r)\rangle, r \in R\}$ such that $n \geq 29$ and for every choice of the parameter k, the probability that a false match occurs is $\leq (1.255)^k t^{-(2k-1)}(1 + 0.6 \ln t)^k$.

Proof The probability of a false match is bounded above by

$$t\left(\frac{\pi(n)}{\pi(M)}\right)^k.$$

Apply Lemma 2 and the inequality $n \geq 29$ to bound $\pi(n)$ from above and $\pi(nt^2)$ from below. □

Corollaries 4(a) and 4(b) establish that it is possible to achieve concise fingerprints that ensure a low probability of a false match. For example, suppose Algorithm 2 is run on an instance where $n = 250$, $t = 4000$, and $M = nt^2 = 4 \times 10^9$. Then, for any $p \leq M$, the range of the fingerprinting function H_p is $\{0, 1, \cdots, p-1\}$, where $p \leq 4 \times 10^9 < 2^{32}$. Hence each string of length 250 can be represented by a 32-bit fingerprint, and yet the probability that a false match occurs will be less than 10^{-3}. If Algorithm 1 with $k = 4$ is run on the same instance with the same set of fingerprinting functions, the probability of a false match is less than 2×10^{-22}.

4. Efficient updating for one-dimensional and higher-dimensional problems

In this section we investigate the storage requirements and execution times of our algorithms when the family $\{H_p\}$ of

fingerprint functions is used, where p is drawn from the set of primes $\leq nt^2$.

We assume that it requires constant time to fetch, store, compare, add, or subtract fingerprints. This is reasonable because a fingerprint is an integer in the range $[0, nt^2 - 1]$, so that the number of bits needed to represent a fingerprint is $\lceil \log_2 nt^2 \rceil$. This is of the same order of magnitude as the length of a pointer into the input data. In typical applications of our methods, the length of a fingerprint does not exceed the length of a register in the computer being used. Moreover, in the case of linear pattern matching or higher-dimensional array matching, the pattern of access to fingerprints is predetermined and regular, so that it is normally possible to fetch fingerprints from high-speed registers rather than from memory.

Since false matches are quite unlikely, the execution times of the algorithms are dominated by the updating operation, in which $a_p(r')$ is computed from $a_p(r)$ and $b_p(r')$ is computed from $b_p(r)$. We show that, in the case of linear pattern matching or higher-dimensional array matching, the time for each update is bounded by a constant. It follows that, in these cases, Algorithm 1 is a *real-time algorithm*. By this we mean that the algorithm makes a single pass through its input data, dwelling for a constant time on each bit, and then halts. Here we are assuming that the random primes $p_1, p_2, \cdots, p_k$ are chosen in a preprocessing step, before the input data arrive; this is valid only if the parameters n and t (or upper bounds on these parameters) are available in advance. Algorithms 2 and 3, which check for false matches, run in time $O(n + t)$ in the event that no false match occurs. We later investigate the probability distribution of the execution time of each of these algorithms, taking into account the effect of false matches.

- *The linear pattern-matching problem*

Let us recall how this problem, already treated briefly in Sections 1 and 2, fits into the general framework. We are given a pattern $X \in \{0, 1\}^n$ and a text $Y \in \{0, 1\}^m$, and wish to determine whether X occurs as a consecutive block within Y. Here $t = m - n + 1$, $R = \{1, 2, \cdots, m - n + 1\}$, $X(r) = X = x_1x_2 \cdots x_n$, $Y(r) = y_ry_{r+1} \cdots y_{r+n-1}$, $\alpha = 1$, and $r' = r + 1$.

We assume that the input is a string consisting of the pattern X followed by the text Y. As the input is scanned from left to right, the fingerprint of the pattern and the fingerprints of the blocks within the text are computed with a constant number of operations per bit of input.

Recall that $H(Y(r))$ denotes the integer represented by the string $Y(r)$. Then $H(Y(r + 1)) = (H(Y(r)) - 2^{n-1}y_r) \cdot 2 + y_{r+n}$. This gives the following formula for updating the fingerprint of a block of the text:

$$b_p(r + 1) = (b_p(r) + b_p(r) + \xi y_r + y_{r+n}) \bmod p, \quad (1)$$

where $\xi = -2^n \bmod p$. To initialize this computation, one pretends that the text is preceded by a string $y_{-(n-1)}y_{-(n-2)} \cdots y_0$ of n zeros. With this convention, we have

$$b_p(-n) = 0$$

and

$$b_p(r + 1) = (b_p(r) + b_p(r) + \xi y_r + y_{r+n}) \bmod p,$$

where r ranges from $-n$ to $m - n$, and $y_j = 0$ for $j < 0$. The fingerprint of the text X is computed in a similar manner:

$$a_p(-n) = 0$$

and

$$a_p(r + 1) = (a_p(r) + a_p(r) + \xi x_r + x_{r+n}) \bmod p,$$

where r ranges from $-n$ to 0, and $x_j = 0$ for $j < 0$. The fingerprint of X is $a_p(1)$.

Note that, if $0 \leq r_1 \leq p - 1$ and $0 \leq r_2 \leq p - 1$, then $r_1 + r_2 \bmod p$ is either $r_1 + r_2$ or $r_1 + r_2 - p$. It follows that updating can be performed with a constant number of operations. On a typical single-address computer, four fetches, three adds, three comparisons, two subtractions, and one store are sufficient for updating. Moreover, since the pattern of access to data is so simple and regular, it is possible to keep the constants ξ and p, the fingerprint of the pattern and the most recently computed fingerprint of a block of text, in fast registers, and to fetch bits from the pattern and text from memory into fast registers before they are needed, so that all operations take their operands from fast registers.

The storage requirements of Algorithms 1, 2, and 3 are modest. Algorithms 2 and 3 require six registers for data (to store the constants ξ and p, the fingerprint of the pattern, the most recently computed fingerprint of a block of text, and the two bits of input data needed for the current updating step) and two address registers which contain pointers into the input. Algorithm 1, which uses k fingerprinting functions at once, requires $4k + 2$ registers for data and two address registers.

Theorem 5

Algorithm 1 is a real-time algorithm. For every input consisting of a pattern of length n and a text of length m, the expected running time of Algorithm 2 or Algorithm 3 is $O(n + m)$.

Proof The proof that Algorithm 1 is real-time is given above. Algorithms 2 and 3 require $O(n + m)$ time for reading the input data and performing updating operations, $O(n)$ time to verify a match if one occurs, and $O(n)$ time to detect each false match that occurs. The probability of a false match is at most $2.511/(m - n + 1)$, and the maximum number of false matches possible is $m - n + 1$, so the expected time spent in detecting false matches is bounded above by

$$O(n)\cdot\frac{2.511}{m-n+1}\cdot(m-n+1)=O(n).$$

Thus the expected running time is $O(m+n)$. □

In Section 5 we make a further analysis of the probability distribution of the execution time of Algorithm 3, showing that Algorithm 3 rarely experiences a large deviation above the expected execution time.

• *Two-dimensional array matching*

In this subsection we sketch how Algorithms 1, 2, and 3 can be tailored to the two-dimensional array-matching problem introduced in Section 2. To do so, we impose a linear ordering on the index set R, and then show that fingerprints can be updated rapidly as the algorithm marches through this linear order.

Recall that $R = \{\langle k, l\rangle \mid s \le k \le m \text{ and } s \le l \le m\}$. We order R so that its first element is $\langle s, s\rangle$, its last element is $\langle m, m\rangle$, and the successor of $\langle k, l\rangle$ is given by

$$\langle k, l\rangle' = \text{if } k < m \text{ then } \langle k+1, l\rangle \text{ else (if } l < m \text{ then } \langle s, l+1\rangle).$$

Geometrically, the algorithm starts with the $s \times s$ block in the upper left-hand corner of the text, marches down until it reaches the last row, moves to the highest position one column to the right, marches down, etc.

Recall that, for any string X, $H(X)$ denotes the integer having X as its binary representation, and $H_p(X) = H(X) \bmod p$. For $k = 1, 2, \cdots, m$ and $l = s, s+1, \cdots, m$, let w_{kl} be the string $y_{k,l-s+1}y_{k,l-s+2}\cdots y_{kl}$ of length s. For $k = s, s+1, \cdots, m$ and $l = s, s+1, \cdots, m$, let z_{kl} be the string $w_{k-s+1,l}w_{k-s+2,l}\cdots w_{kl}$ of length s^2. Then z_{kl} is the bit pattern obtained by concatenating together the rows of the $s \times s$ block of array Y having position k, l in its lower right-hand corner.

Let

$$c_p(\langle k, l\rangle) = H_p(w_{kl})$$

and

$$b_p(\langle k, l\rangle) = H_p(z_{kl}).$$

Then $c_p(\langle k, l\rangle)$ is the fingerprint of the string of s bits in row k having rightmost position k, l, and $b_p(\langle k, l\rangle)$ is the fingerprint of the $s \times s$ block of Y with position k, l in its lower right-hand corner. The following update formulas are easily derived:

$$c_p(\langle k, l\rangle) = (2c_p(\langle k, l-1\rangle) + \xi y_{k,l-s} + y_{k,l}) \bmod p, \tag{2}$$

$$b_p(\langle k+1, l\rangle) = ((b_p(\langle k, l\rangle) + \vartheta\, c_p(\langle k-s+1, l\rangle))\lambda + c_p(\langle k+1, l\rangle)) \bmod p, \tag{3}$$

where $\xi = -2^s \bmod p$, $\lambda = 2^s \bmod p$, and $\vartheta = -2^{s(s-1)} \bmod p$. The right-hand side of (2) can be evaluated in constant time, and the right-hand side of (3) can be evaluated in constant time if we assume that multiplication mod p can be performed in constant time. This would be true, for example, if a hardware multiply/divide unit were available that delivered the remainder in the case of integer division.

Given these update formulas it is easy to work out the details of initialization and storage allocation for Algorithms 1, 2, and 3. Each of these algorithms requires $O(m)$ storage locations, and, in the absence of false matches, runs in time $O(m^2)$, since the entire computation is carried out in a single pass through X followed by a single pass through Y, with constant execution time per position.

A simple trick reduces the storage requirements from $O(m)$ to $O(s)$, at the cost of increasing the execution time by a constant factor. The idea is to cover the $m \times m$ text array with small subarrays, with the property that every $s \times s$ block in the text occurs as a block in one of the subarrays. The original algorithm can then be applied independently to each subarray. The reader will easily verify that, for each $w \ge s$, there exists a covering with

$$\frac{m^2}{w^2} + O\left(\frac{m}{w}\right)$$

subarrays, each of which is $(w+s-1) \times (w+s-1)$. The running time then becomes

$$O\left(\frac{m^2}{w^2}(w+s-1)^2\right),$$

and the storage requirement is $O(w+s-1)$. Choosing $w = O(s)$ gives time $O(m^2)$ and storage $O(s)$.

The algorithms generalize immediately to d-dimensional arrays, requiring $O(m^{d-1})$ storage to process a $m \times m \times \cdots \times m = m^d$ array. With the subarray-covering trick, the storage can be reduced to $O(s^{d-1})$, with expected running time $O(m^d)$.

Bird [7] has given an extension of the KMP algorithm to two-dimensional array matching, and his approach can also be applied to d-dimensional array matching. His method requires a fairly complex preliminary phase, in which the pattern is processed to give arrays of length $O(s^d)$ whose elements are pointers. Thus, our randomizing algorithm is simpler and equally fast and, in the version based on subarray covering, uses less storage.

5. The advantages of reinitializing

In this section we explore the properties of Algorithm 3, the version of the fingerprinting method which discards its current fingerprint function whenever a false match occurs. We show that this algorithm has two important advantages, which in some environments outweigh the overhead of reinitializing the fingerprint function after a false match:

1. Reinitializing is a hedge against catastrophe. It reduces to a completely negligible level the probability that a long series of false matches will occur.

2. The performance of the method remains good when fingerprinting is based on arbitrary moduli rather than primes.

• *Hedging against catastrophe*

Suppose we are presented with an instance of the linear pattern-matching problem in which the pattern X is the binary representation of a multiple of the prime p, and $Y = 0^m$. Then if Algorithm 1 or Algorithm 2 is executed with H_p as its fingerprint function, a false match occurs in each of the $m - n + 1$ possible positions where X is tested for an occurrence within Y. The following theorem shows that choosing Algorithm 3 renders the possibility of such a catastrophe remote regardless of how X and Y are chosen.

Theorem 6

Suppose Algorithm 3 is applied to an instance of the general string-matching problem specified by $\{\langle X(r), Y(r)\rangle, r \in R\}$, where $|R| = t$ and each string $X(r)$ or $Y(r)$ is of length n. If S is the set of primes less than or equal to $M = nt^2$ and $\phi_p = H_p$, then the probability that k or more false matches occur is $\leq (2.511/t)^k$.

Proof By Corollary 4(a), each time a new prime $p \in S$ is randomly chosen, the probability of a false match is at most $2.511/t$. □

• *Fingerprinting using arbitrary moduli*

We consider the behavior of Algorithm 3 when the fingerprinting process is based on arbitrary moduli, rather than primes; i.e., we take $S = \{1, 2, \cdots, M\}$, with $\phi_p = H_p$.

We require two number-theoretic lemmas.

Lemma 7 [11]

There is a constant B such that, for all positive integers x,

$$\ln\ln x + B - \frac{1}{2(\ln x)^2} < \sum_{\substack{p \text{ prime} \\ p \leq x}} \frac{1}{p} < \ln\ln x + B + \frac{1}{(\ln x)^2}.$$

Let M be a positive integer. Call an integer x *M-fat* if $1 \leq x \leq M$ and x has a prime divisor $p > \sqrt{M}$. Let $F(M)$ denote the number of M-fat integers.

Lemma 8

For $M \geq 9000$, $F(M) \geq M/2$.

Proof For any prime p the number of positive integers less than or equal to M and divisible by p equals $\lfloor M/p \rfloor$. If $x \leq M$ is M-fat, then exactly one prime $p > \sqrt{M}$ divides x. Thus

$$F(M) = \sum_{\substack{\sqrt{M} < p \leq M \\ p \text{ prime}}} \left\lfloor \frac{M}{p} \right\rfloor \geq M \sum_{\substack{\sqrt{M} < p \leq M \\ p \text{ prime}}} \frac{1}{p} - \pi(M).$$

Applying Lemma 2 and Lemma 11,

$$F(M) \geq M\left(\ln\ln M - \ln\ln\sqrt{M} - \frac{1}{2(\ln M)^2} - \frac{1}{(\ln\sqrt{M})^2}\right) - \frac{1.25506M}{\ln M}$$

$$= M\left(\ln 2 - \frac{9}{2(\ln M)^2} - \frac{1.25506}{\ln M}\right).$$

Now

$$M \geq 9000 \Rightarrow \ln M \geq 9.1 \Rightarrow F(M) \geq M\left(0.693 - \frac{9}{2\cdot(9.1)^2} - \frac{1.25506}{9.1}\right) > \frac{M}{2}. \square$$

We now estimate the probability of a false match when randomly chosen M-fat numbers are used for fingerprinting.

Lemma 9

Consider an instance of the general pattern-matching problem in which $t = n$. If we use Algorithm 3 with $S = \{p \mid p \text{ is } M\text{-fat}\}$, where

$$M \geq \max\left(\frac{6.25n^4}{(\ln n)^2}, 9000\right)$$

and $\phi_p(Y) = H(Y) \bmod p$, then the probability of a false match is $\leq 1/2$.

Proof The proof proceeds along the lines of Theorem 3 and Corollary 4(a). In this case $|R| = n$ and a false match occurs only if the M-fat integer p divides

$$P = \prod_{X(r) \neq Y(r)} |H(X(r)) - H(Y(r))| \leq 2^{n^2}.$$

Let L be the number of M-fat integers x that divide P. For each such x there is a prime $q > \sqrt{M}$ that divides x, and each such prime occurs in at most $\sqrt{M}$ different divisors of P. Thus P has at least $L/\sqrt{M}$ distinct prime divisors $> \sqrt{M}$, so $(\sqrt{M})^{L/\sqrt{M}} \leq P \leq 2^{n^2}$. Passing to logarithms,

$$L \leq \frac{n^2\sqrt{M}}{\log_2\sqrt{M}}.$$

Since the number of M-fat integers is $>(M/2)$, the probability that a randomly chosen M-fat integer triggers a false match is

$$\leq \frac{L}{M/2} \leq \frac{2n^2}{\sqrt{M}\log_2\sqrt{M}}.$$

For the indicated choice of M,

$$\frac{2n^2}{\sqrt{M}\log_2\sqrt{M}} < \frac{1}{2}. \square$$

Corollary 9

Suppose Algorithm 3 is applied to an instance of the general string-matching problem specified by $\{\langle X(r) < Y(r)\rangle, r \in R\}$, where $|R| = t$ and each string $X(r)$ or $Y(r)$ is of length n.

Assume that fingerprints can be updated in constant time, as in the case of linear pattern matching or two-dimensional array matching. If S is the set of positive integers $\leq M$, where

$$M = \max\left(\frac{6.25n^4}{(\ln n)^2}, 9000\right),$$

then the expected running time is $O(n + t)$.

Proof Apart from detecting false matches and restarting the fingerprinting process after a false match, the running time is $O(n + t)$. The time overhead associated with detecting a false match and resuming the computation is $O(n)$. With probability $>1/2$, the p chosen after a false match is M-fat. If p is M-fat, then, with probability $>1/2$, the computation will advance through at least n indices $r \in R$ before the next false match occurs. Hence, the expected number of false matches is $O(t/n)$, and the expected time spent in dealing with false matches is $O(t)$. □

6. A second family of fingerprint functions

In this section we present another interesting family $\{K_p\}$ of fingerprint functions. For each positive integer, p, K_p is a homomorphism from $\{0, 1\}^*$ into the group of 2×2 unimodular matrices with entries in Z_p, the ring of integer residues mod p.

Let λ denote the null string. Define a homomorphism K from $\{0, 1\}^*$ into 2×2 nonnegative integer unimodular matrices by

$$K(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad K(0) = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \quad K(1) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix},$$

and

$$K(X^*Y) = K(X) \cdot K(Y),$$

where * denotes concatenation of strings and $\cdot$ denotes matrix multiplication. For any positive integer p, the function K_p is defined in the same way, except that all matrix elements are regarded as elements of Z_p rather than as integers.

The function K has the following easily provable properties:

1. K is a monomorphism; i.e., $K(X) = K(Y) \Rightarrow X = Y$.
2. If $X \in \{0, 1\}^n$, then each element of $K(X)$ is less than or equal to F_n, the nth Fibonacci number ($F_0 = F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$, $n \geq 2$).

In comparison with the family $\{H_p\}$ of fingerprint functions, the family $\{K_p\}$ has the disadvantage that each fingerprint consists of four integers mod p, rather than one. We show that the two families are about equally effective in avoiding false matches, and that the use of $\{K_p\}$ leads to remarkably simple updating methods.

Theorem 10
If Algorithm 1 is executed with $S = \{p \mid p \leq M \text{ and } p \text{ is prime}\}$ and $\phi_p = K_p$, then, for every instance $\{\langle X(r), Y(r)\rangle, r \in R\}$, the probability that a false match occurs is

$$\leq \frac{\pi(\lceil 4t \log_2 F_n \rceil)}{\pi(M)}, \quad \text{provided } \lceil 4t \log_2 F_n \rceil \geq 29,$$

where $t = |R|$.

Proof The proof is similar to the proof of Theorem 4. If a false match occurs, then, for some $r \in R$ and some $1 \leq i, j \leq 2$,

$$K(X(r))_{i,j} \neq K(Y(r))_{i,j},$$

but

$$K_p(X(r))_{i,j} = K_p(Y(r))_{i,j}.$$

It follows that p divides the product of all the nonzero terms of the form $|K(X(r)_{i,j} - K(Y(r))_{i,j}|$, $r \in R$, $i \in \{1, 2\}$, $j \in \{1, 2\}$. This product is bounded above by F_n^{4t}, which in turn is $\leq 2^{\lceil 4t \log_2 F_n \rceil}$. By Corollary 1, the number of primes which divide this product is $\leq \pi(\lceil 4t \log_2 F_n \rceil)$. The result now follows, since p is chosen at random from a set of $\pi(M)$ primes. □

Corollary 10
If Algorithm 1 is executed with $S = \{p \mid p \leq nt^2 \text{ and } p \text{ prime}\}$ and $\phi_p = K_p$, then, for every instance of the input data $\{\langle X(r), Y(r)\rangle, r \in R\}$, the probability that a false match occurs is $\leq 6.971/t$.

Proof Apply Theorem 10, Lemma 2, and the fact that $\log_2 F_n \sim 0.694n$. □

We next demonstrate that, when Algorithm 1 is used with $\phi_p = K_p$, elegant updating methods result. For example, in the string-matching problem, the counterpart of Equation (1) is

$$a_p(r + 1) = A_p(y_r)^{-1} a_p(r) A_p(y_{r+n}).$$

Here all matrices are over Z_p,

$$A_p(0)^{-1} = \begin{bmatrix} 1 & 0 \\ p - 1 & 1 \end{bmatrix},$$

and

$$A_p(1)^{-1} = \begin{bmatrix} 1 & p - 1 \\ 0 & 1 \end{bmatrix},$$

and similarly for the two-dimensional array-matching problem, using appropriate counterparts to Equations (2) and (3).

7. Fingerprinting techniques for irregular shapes

In this section we demonstrate that randomized algorithms based on fingerprinting techniques can be applied not only

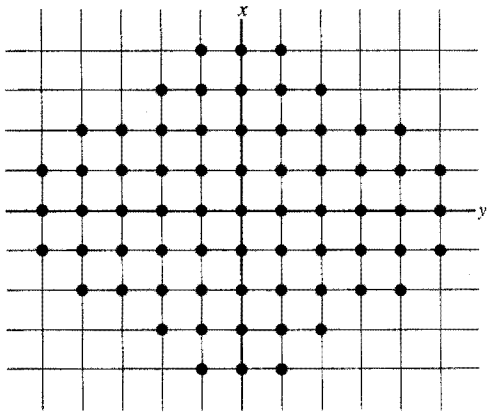

Figure 1

A shape S. As in arrays, the x coordinate designates rows and the y coordinate columns.

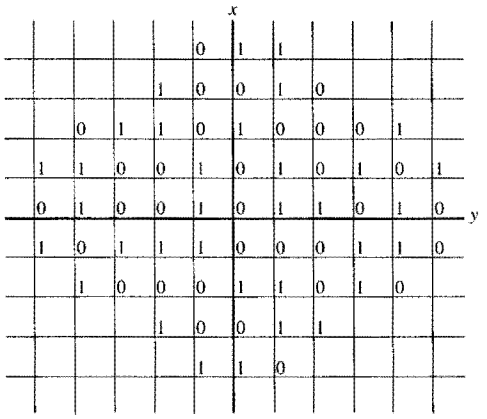

Figure 2

A pattern X of shape S.

to the matching of strings and arrays, but also to matching problems involving patterns of irregular shape. Let Z denote the set of integers. Define a *shape* S as a finite subset of $Z \times Z$ which includes $(0, 0)$. The *size* of S is by definition $|S| = n$. A *pattern* of shape S is a function $X: S \rightarrow \{0, 1\}$. See **Figures 1** and **2.**

Let S be a shape, and X a pattern of shape S. Let Y be an $m \times m$ array of 0s and 1s; more precisely, Y is a function from $\{1, 2, \cdots, m\}^2$ into $\{0, 1\}$. Define $S + (a, b) = \{(x + a, y + b) \mid (x, y) \in S\}$. We say that X *occurs in* Y *at* (a, b) if

$$S + (a, b) \subseteq \{1, 2, \cdots, m\} \times \{1, 2, \cdots, m\}$$

and

$$\text{for all } (x, y) \in S, \qquad X(x, y) = Y(x + a, y + b).$$

The pattern-matching problem in this case is the following. Given a two-dimensional array Y and a pattern X of shape S, does X occur in Y?

The straightforward algorithm requires, for an $m \times m$ array and for shapes of size n, about $m^2 n$ steps. Our general fingerprinting method in many cases reduces the number of steps to $m^2\sqrt{n} + n$.

Define a *horizontal segment* as a subset of $N \times N$ of the form $\{k\} \times \{y \in N \mid \ell \leq y \leq r\}$.

Given a shape S, we can decompose it into maximal horizontal segments and arrange these in some definite order $I_1, \cdots, I_c$. Our method is efficient, as compared with the straightforward method, whenever the number of segments satisfies $c \ll |S| = n$. For example, for shapes S which are circles, or ring-shaped with the interior radius half of the exterior radius, or equilateral triangles, we have $c = O(\sqrt{n})$.

Assume that we want to solve the pattern-matching problem for a two-dimensional $m \times m$ array and a pattern X of a favorable shape S of size n. In order to cast this problem into the general framework of Algorithm 1, decompose S into a disjoint union $I_1 \cup \cdots \cup I_c$ of horizontal segments where $I_j = \{k_j\} \times \{\ell_j \leq y \leq r_j\}$, $1 \leq j \leq c$. Let $\min(x) = \min k_j$, $\max(x) = \max k_j$, $\min(y) = \min \ell_j$, $\max(y) = \max r_j$. Note that since $(0, 0) \in S$, we have $\min(x), \min(y) \leq 0$. Define

$$R = \{(a, b) \mid 1 - \min(x) \leq a \leq m - \max(x), \quad 1 - \min(y) \leq b \leq m - \max(y)\}$$

and $|R| = t$. Thus $S + (a, b) \subseteq \{1, \cdots, m\}^2$ iff $(a, b) \in R$. We have $t \leq m^2$. Unravel X into a string by defining

$$X(I_j) = X(k_j, \ell_j)\, X(k_j, \ell_j + 1) \cdots X(k_j, r_j)$$

and

$$\bar{X} = X(I_1)\, X(I_2) \cdots X(I_c) \in \{0, 1\}^n.$$

Similarly, the 0-1 pattern formed in the array Y by the shape $S + (a, b)$ is unraveled into a string $\bar{Y}(a, b)$, $(a, b) \in R$. Thus we have the string-matching problem $\{\langle \bar{X}, \bar{Y}(a, b)\rangle, (a, b) \in R\}$, and the solution of this problem will tell us whether the pattern X occurs in the two-dimensional array Y.

It is most convenient to use the fingerprint functions K_p of Section 6. Choose a random prime $p \leq nt^2 \leq nm^4$. Then

$$K_p(\bar{X}) = \prod_{j=1}^{c} \prod_{y=\ell_j}^{r_j} K_p(X(k_j, y)).$$

Thus, calculating $K_p(\bar{X})$ requires $n - 1$ multiplications of 2×2 matrices in Z_p.

We want to calculate the $K_p(\bar{Y}(a, b))$ by $2c - 1$ operations per fingerprint. To this end, preprocess Y as follows. Associate with each position (k, ℓ), where $1 \le k, \ell \le m$, the cumulative product of matrices corresponding to the lowest ℓ bits in the kth row of Y, i.e., the matrix

$$f(k, \ell) = \prod_{j=1}^{\ell} K_p(Y(k, j)).$$

These matrices can be computed at the cost of $m(m-1)$ multiplications of 2×2 matrices over Z_p. The matrices $f(k, \ell)$ are stored in an appropriate array. Note that each $f(k, \ell)$ is a unimodular matrix, and for such a matrix

$$\begin{bmatrix} u & v \\ w & z \end{bmatrix}^{-1} = \begin{bmatrix} z & p - v \\ p - w & u \end{bmatrix}.$$

With the $f(k, \ell)$ available, we can calculate $K_p(\bar{Y}(a, b))$ for $(a, b) \in R$ by $2c - 1$ matrix multiplications,

$$K_p(\bar{Y}(a, b)) = \prod_{j=1}^{c} f^{-1}(k_j + a, \ell_j + b) f(k_j + a, r_j + b).$$

Summing up our results, we have the following direct corollary of Theorem 10 and Corollary 10.

Theorem 11
If Y is an $m \times m$ array and x is an S-shaped pattern where $|S| = n$ and S is the union of c horizontal segments, then testing whether X occurs in Y requires $n - 1 + m^2 + t \cdot (2c - 1)$ multiplications of 2×2 matrices in Z_p. The probability of a false match for a random choice of $p \le n \cdot m^4$ is smaller than $6.971/m^2$.

Remark The above method is advantageous for shapes S such that $c \ll |S|$. In many cases it has the effect of reducing the number of steps required to test whether X occurs in Y at position (a, b) from the area of S to the diameter of S, i.e., essentially from $|S|$ to $|S|^{1/2}$.

If the same array Y is repeatedly probed for the occurrence of patterns $X_1, X_2, \cdots$, then the computation of $f(k, \ell)$, $1 \le k, \ell \le m$, will serve for all these probes.

8. Parallel pattern matching

The randomized pattern-matching algorithms lend themselves in a convenient way to parallelization. We treat the string-matching problem and employ the fingerprinting functions K_p of Section 6.

Let $X = x_1 x_2 \cdots x_n$ be a bit pattern and $Y = y_1 y_2 \cdots y_m$ be a bit text. Let p be a fixed (randomly chosen) prime. Define

$$P(Y, k) = y_1 y_2 \cdots y_k, \quad 1 \le k \le n,$$

$$K_p(P(Y, k)) = K_p(y_1) \cdot K_p(y_2) \cdot \cdots \cdot K_p(y_k), \tag{4}$$

where the matrix multiplication is done for 2×2 matrices over Z_p.

Since matrix multiplication is associative, it follows from the parallel-prefix computation theorem that all the products (4) can be computed in parallel in time $O(\log m)$, using m processors. Similarly, $K_p(X)$ can be computed in time $O(\log n)$, using n processors. Denoting, as in Section 2, $Y(r) = y_{r+1} \cdots y_{r+n-1}$, $1 \le r \le m - n + 1$, the fingerprints

$$K_p(Y(r)) = K_p(P(Y, r-1))^{-1} \cdot K_p(P(Y, r + n - 1))$$

can be computed in parallel, using m processors, in constant time. Finally, the comparisons $K_p(X) = K_p(Y(r))$ can be done in parallel, using $m - n + 1$ processors, in constant time. Using Corollary 10, but choosing the prime in the range $[1, nm^k]$, we get the following.

Theorem 12
The string-matching problem for a pattern of length n and a text of length $m (n \le m)$, where we find all matches, can be solved by m processors in time $O(\log m)$ with probability of error smaller than $0.697/m^k$.

The same method produces optimally parallel algorithms for string matching when the number of processors is $< m/\log n$. A similar result was obtained by Vishkin [12], via a considerably more complicated deterministic algorithm.

Conclusion

We have seen that randomizing over a class of easily computable and easily updatable fingerprints produces very simple and efficient algorithms for a variety of one-dimensional and multidimensional pattern-matching problems. The salient point is that one can *prove* for these algorithms that they lead to short expected computation time or run in real time with a negligible probability of error, for every individual pattern/text pair.

The ideas and methods presented here have many variations and a wide range of additional applications. In particular, the second author has found another class of fingerprint functions employing polynomials over finite fields instead of integers [13].

Acknowledgments

Dr. Karp's work was supported by NSF Grant MCS77-09906 and by the Miller Institute for Basic Research in Science. Dr. Rabin's work was supported by NSF Grant MCS80-12716 at the University of California at Berkeley and by NSF Grant MCS81-21431 at Harvard University.

References

1. D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM J. Computing* **6**, 323–350 (1977).
2. R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Commun. ACM* **20**, 762–772 (1977).
3. Z. Galil and J. Seiferas, "Saving Space in Fast String Matching," *SIAM J. Computing* **9**, 417–438 (1980).
4. Z. Galil and J. Seiferas, "Time-Space Optimal String Matching," *Proc. 13th Annual ACM STOC*, 1981, pp. 106–113.

5. T. P. Baker, "A Technique for Extending Rapid Exact String Matching to Arrays of More than One Dimension," *SIAM J. Computing* **7**, 533–541 (1978).
6. R. M. Karp, R. E. Miller, and A. L. Rosenberg, "Rapid Identification of Repeated Patterns in Strings, Trees and Arrays," *Proc. 4th Annual ACM STOC*, 1972, pp. 125–136.
7. R. S. Bird, "Two Dimensional Pattern Matching," *Info. Proc. Lett.* **6**, 168–170 (1977).
8. M. O. Rabin, "Probabilistic Algorithms," *Algorithms and Complexity, Recent Results and New Directions*, J. F. Traub, Ed., Academic Press, Inc., New York, 1976, pp. 21–40.
9. M. O. Rabin, "Probabilistic Algorithm for Testing Primality," *J. Number Theor.* **12**, 128–138 (1980).
10. R. Solovay and V. Strassen, "A Fast Monte-Carlo Test for Primality," *SIAM J. Computing* **6**, 84–85 (1977).
11. J. B. Rosser and L. Schoenfeld, "Approximate Formulas for Some Functions of Prime Numbers," *Illinois J. Math.* **6**, 64–94 (1962).
12. U. Vishkin, "Optimal Parallel Pattern Matching in Strings," *Info. Control* **67**, 91–113 (1985).
13. M. O. Rabin, "Fingerprinting by Random Functions," *Report TR-15-81*, Center for Research in Computing Technology, Harvard University, Cambridge, MA, 1981.

Received December 17, 1986; accepted for publication January 6, 1987

Richard M. Karp *University of California, Berkeley, California 94720.* Dr. Karp received his Ph.D. in applied mathematics from Harvard University, Cambridge, Massachusetts, in 1959. He was a Research staff member in the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, from 1959 to 1968. In 1968 he became professor of computer science and operations research at the University of California. In 1980 he also became professor of mathematics at the University. Dr. Karp was co-chair of program and computational complexity at the Mathematical Sciences Research Institute from 1985 to 1986. He was the Miller Research Professor at the University from 1980 to 1981. He is a member of the American Academy of Arts and Sciences and the National Academy of Sciences. Dr. Karp received the Lanchester Prize in 1977, the Folkerson Prize in 1979, the ACM Touring Award in 1985, and the Distinguished Teaching Award in 1986. Areas of interest to him are combinatorial algorithms and computational complexity.

Michael O. Rabin *Harvard University, Cambridge, Massachusetts 02138.* Dr. Rabin is the first appointed Thomas J. Watson Sr. Professor of Computer Science at Harvard University. He received his M.Sc. degree from the Hebrew University, Jerusalem, Israel, in 1953, and his Ph.D. from Princeton University, New Jersey, in 1956. From 1956 to 1958 he was H. B. Fine Instructor in Mathematics at Princeton University, and he was a member of the Institute for Advanced Study in 1958. He became senior lecturer at the Hebrew University of Jerusalem in 1958, advancing to the rank of full professor in 1965. During his tenure at the Hebrew University he has been chairman of the Institute of Mathematics (1964–1966), chairman of the Computer Science Department (1970–1971), and rector (academic head) of the University (1972–1975); he was appointed the University's first Albert Einstein Professor (1980). In 1981 he was named Gordon McKay Professor of Computer Science at Harvard University and became Thomas J. Watson Sr. Professor in 1983. He currently serves on the faculties of both Harvard and the Hebrew University. Professor Rabin serves on the editorial boards of the *Journal of Computer and Systems Sciences*, the *Journal of Combinatorial Theory*, the *Journal of Theoretical Computer Science*, the *Journal of Algorithms*, and *Information and Control*. Among the awards he has received are the C. Weizmann Prize for Exact Sciences (1960), the Rothschild Prize in Mathematics (1974), the A. M. Turing Award in Computer Science (co-winner, 1976), and the Harvey Prize in Science and Technology (1980). He is also a foreign honorary member of the American Academy of Arts and Sciences (elected 1975), a member of the Israel Academy of the Sciences and Humanities (elected 1982), and a foreign associate of the National Academy of Sciences (elected 1984). His research interests include complexity of computations, efficient algorithms, randomizing algorithms, parallel and distributed computations, and computer security. Dr. Rabin is also interested in bringing traditional mathematical tools to bear on computer science problems of foundational as well as practical significance.