JOURNAL OF ALGORITHMS 13, 2–32 (1992)

Efficient Pattern Matching with Scaling

AMIHOOD AMIR*

College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280

GAD M. LANDAU†

Department of Computer Science, Polytechnic University, 333 Jay Street, Brooklyn, New York 11201

AND

UZI VISHKIN‡

Institute for Advanced Computer Studies and Department of Electrical Engineering, University of Maryland, College Park, Maryland 20742 and Department of Computer Science, School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel 69978

Received April 17, 1990; revised May 23, 1991

The problem of *pattern matching with scaling* is defined. The input for the two-dimensional version of the problem consists of an $n \times n$ "text" matrix and an $m \times m$ "pattern" matrix. We want to find all occurrences of the pattern in the text, *scaled* to all natural multiples. That is, for every natural number i, $1 \leq i \leq \lfloor n/m \rfloor$ we seek all occurrences of the pattern in the text, where each character of the pattern corresponds to an $i \times i$ square in the text. This problem is useful for some tasks in computer vision. Our main contribution is a linear time algorithm for the problem. We also consider situations where the text is provided in a less redundant form. For instance, suppose that a repeating character is compressed into one character, along with the number of repetitions. We show how to enhance our algorithm so that its running time may become sublinear with respect to the original redundant input representation. Our algorithms are based on a new algorithmic approach to two-dimensional string matching. Unlike existing approaches, the new approach does not work by reducing a two-dimensional problem into an one-dimensional problem.

*Partially supported by NSF Grant IRI-9013055.

†Partially supported by NSF Grant CCR-8908286 and the New York State Science and Technology Foundation, Center for Advanced Technology in Telecommunications, Polytechnic University, Brooklyn, NY.

‡Partially supported by NSF Grant CCR-8906949.

0196-6774/92 $3.00

1. Introduction

Elegant and efficient algorithms exist for exact string matching (e.g., [W-73, KMP-77, AC-77, BM-77, GS-83, V-85, V-91]), as well as efficient extensions to two dimensions [B-77, Ba-78]. Considerable attention has been given lately to approximate string matching [U-85, LV-89, GP-90]. In [LV-89] it was shown that all occurrences of a pattern string of length m in a text string of length n with no more than k errors (mismatches, addition, and deletion of characters) can be found in time $O(kn)$. This result was extended to two dimensions in [KS-87] and improved by [AL-90].

One of the roles of theoretical computer science is to develop an algorithmic theory for various application domains. We can go about developing such theory by way of abstracting practical algorithmic problems to "pure" form. (A single practical problem may lead to several pure problems.) This should be followed by designing algorithms for the specific pure problem(s). Finally, the knowledge base, consisting of these algorithms, will be used for composing an algorithm for the original practical problem. This paper is a modest part of such treatment. Consider problems of searching aerial photographs. The first phase in an abstraction into pure problems will be to classify the difficulties that arise into three major subclasses:

- local errors; caused by occlusion and varying level of detail.
- scaling (or calibration of size); caused by the distance (and to some extent, the angle) of the camera.
- rotation; caused by the orientation of the camera in relation to the object.

In [AL-90] and [KS-87] algorithms for some pure local errors problem were given. In the present paper, we tackle a clean (discrete) version of scaling.

We present here an efficient algorithm for finding all *scaled* occurrences of a given pattern in a text. As an example of a two-dimensional scaling problem, consider reading a text with characters of differing sizes (e.g., newspapers).

For didactic reasons, we start with a definition of the one-dimensional version of our newly defined scaling problem. A linear time algorithm for this problem was easily derived from the known linear time algorithms for the exact string matching problem. Encouraged by this success we tried to extend known algorithms for two-dimensional exact string matching to our two-dimensional problem (i.e., two-dimensional string matching with scaling). Unfortunately, standard techniques failed to offer efficient results. So, we had to develop an approach that is different from known two-

dimensional algorithms (e.g., [B-77, Ba-78]). Recall that these algorithms work by reducing the two-dimensional input arrays into one-dimensional strings and then applying one-dimensional techniques. Interestingly, our approach is *inherently two-dimensional* since it adheres to the two-dimensional structure of the problem. Suggesting algorithmic techniques for coping with the two-dimensionalmatching problem with scaling is the primary concern of the present paper. An offspring of our research is a "sublinear" time algorithm; i.e., one whose complexity is, in some sense, a function of the number of *blocks* of repeating symbols in the images, rather than the number of symbols (pixels).

String Matching with Scaling

We define one- and two-dimensional versions of our problem. The string $aa \cdots a$, where the symbol a is repeated k times (to be denoted a^k), is referred to as *scaling of the singleton string a by multiplicative factor* k, or simply as *a scaled to* k. Similarly, consider a string $A = a_1 \cdots a_l$. A *scaled to* k (A^k) is the string $a_1^k, \ldots, a_l^k$.

The problem of *one-dimensional string matching with scaling*. *Input*: Pattern $P = p_1 \cdots p_m$ and text $T = t_1 \cdots t_n$, where $n > m$. *Output*: All positions in T, where an occurrence of P scaled to k starts, for any $k = 1, \ldots, \lfloor n/m \rfloor$.

A simple linear time algorithm in Eilam-Tzoreff and Vishkin [EV-88] can be adapted for the one-dimensional problem. Under some reasonable assumptions this algorithm runs in sublinear (i.e., $o(n + m)$) time). Specifically, if $P = x_1^{k_1}, \ldots, x_f^{k_f}$ and $T = y_1^{l_1}, \ldots, y_g^{l_g}$ then the algorithm runs in $O(f + g)$ time for inputs that are given in this compressed representation. This compressed representation together with the sublinearity result presents a challenging question for the two-dimensional problem that is not fully resolved in the present paper. It is not clear how to define an analogous compression for the two-dimensional problem, or whether a unique definition of compression even exists. Section 3 copes with this issue.

Let $P[m \times m] = (p_{i,j})_{i=1,\ldots,m;\, j=1,\ldots,m}$ be a two-dimensional matrix over a finite alphabet Σ. Then *P scaled to* k (P^k) is the $km \times km$ matrix where every symbol $P[i, j]$ of P is replaced by a $k \times k$ matrix whose elements all equal the symbol in $P[i, j]$. More precisely,

$$P^k[i, j] = P\left[\left\lceil \frac{i}{k} \right\rceil, \left\lceil \frac{j}{k} \right\rceil\right].$$

For an example, see Fig. 1.

$$P = \begin{matrix} O & O & O \\ O & I & O \\ O & O & O \end{matrix} \qquad P^2 = \begin{matrix} O & O & O & O & O & O \\ O & O & O & O & O & O \\ O & O & I & I & O & O \\ O & O & I & I & O & O \\ O & O & O & O & O & O \\ O & O & O & O & O & O \end{matrix}$$

FIGURE 1

The problem of *two-dimensional pattern matching with scaling*. *Input*: Pattern matrix $P[i, j]$, $i = 1, \ldots m$; $j = 1, \ldots, m$ and text matrix $T[i, j]$, $i = 1, \ldots, n$; $j = 1, \ldots, n$, where $n > m$. *Output*: All locations in T, where occurrence of P scaled to k (a *k-occurrence*) starts, for any $k = 1, \ldots, \lfloor n/m \rfloor$.

We defined the scaled matching problem on square texts and patterns for the sake of simplicity only. All our results are also true for rectangular texts and patterns. A more interesting generalization is of *rectangular scaling*, where every character is replaced by a rectangle of size $i \times j$ with i being the *row scale* and j being the *column scale*. Our techniques allow for finding all rectangular scale occurrences of the pattern with just a polylog slowdown compared to the square scales. It will be interesting to find out whether rectangular scaling can also be detected in linear time.

Bird and, independently, Baker [B-77, Ba-78] showed that all exact matches of two-dimensional pattern P in text T can be found in time $O(n^2)$ (linear in the size of the text) when the size of the alphabet is fixed. Using Bird's algorithm separately for each scale $k = 1, \ldots, \lfloor n/m \rfloor$, the two-dimensional pattern matching with scaling problem can be solved in time $O(n^3/m)$. Our *main contribution* in the present paper is a linear time algorithm for this problem.

From a remote enough perspective our algorithmic design strategy can be viewed as realizing the following (very vague) approach: For each scale k, try to select only a fraction of $1/k$ among the n columns and seek k-occurrences only in these columns. Since each selected column intersects n rows, this leads to consideration of $O(n^2/k)$ elements. Summing over all scales, we get $O(n^2)$ multiplied by the harmonic sum $\sum_{i=1}^{n/m}(1/i)$, whose limit is $\log(n/m)$ making the total number of elements scanned $O(n^2 \log(n/m))$. In all our algorithms, time bounds are proportional to the total number of elements scanned. So this approach had to be enhanced in order to obtain a linear time algorithm. A final intuitive step was to select also a $1/k$ fraction of the rows. Since $\sum_{i=1}^{n/m}(1/i^2)$ is bounded by a constant, the number of elements decreases now to

$$O\left(n^2 \sum_{i=1}^{n/m} \frac{1}{i^2}\right) = O(n^2).$$

We consider situations where the text is provided in a less redundant form. For instance, suppose that a repeating character is compressed into one character, along with the number of repetitions. We show how to enhance our algorithm so that its running time may become sublinear with respect to the original redundant input representation. Sublinearity is a difficult concept, with different meanings depending on the context. There are a few works, in the literature, whose concern is to show that some string matching algorithms run in sublinear time under some assumptions about the source of the input. It is easy to justify such concern on practical grounds. To the best of our knowledge, however, theory has not yet developed a widely agreeable concept of how a "typical" input looks like. *The nature of our scaling problem suggests the concept of sublinearity that we adopted.* A scaled occurrence in the text implies blocks of repeating symbols. The number of such repeating blocks is less than the total number of symbols. Exploiting these blocks in a way that the entire text need not be scanned is the source of sublinear processing time that we have in mind. We caution that even a "natural" definition of a repeating block is not clear. We present here an initial attempt to grapple with this problem.

The paper is organized as follows. In Section 2, we present an easy and elegant solution to the one-dimensional version of the scaled matching problem. In an attempt to give a self-contained presentation, Section 2 also summarizes some known techniques that are used in the paper; Section 2.2 reviews suffix trees and lowest common ancestors in trees. Section 2.3 reviews range minimum queries, the all-nearest smaller value problem, and Cartesian trees. In Section 3, we consider a relatively simple problem for warm-up. The problem is that of finding the exact match of a two-dimensional $m \times m$ pattern in a two-dimensional $n \times n$ text when they are given in a compressed form. We present a new algorithm whose running time is linear in a naive compressed form (rather than the $O(n^2)$ time Baker–Bird algorithm). Armed with the notions and techniques of Section 3, we produce a linear-time ($O(n^2)$) algorithm for scaled two-dimensional matching in Section 4. Finally, in Section 5, we show that our algorithm actually achieves sublinear time utilizing the compressed form of the input.

2. Preliminaries

2.1. *Scaled Matching: The One-Dimensional Case*

Recall the one-dimensional version of the scaled matching problem. *Input*: Pattern $P = p_1 \cdots p_m$ and text $T = t_1 \cdots t_n$, where $n > m$. *Output:* All positions in T, where an occurrence of P scaled to k starts, for

any $k = 1, \ldots, \lfloor n/m \rfloor$. The natural source of sublinearity in scaled string matching is compression of repeating symbols in the pattern and the text.

DEFINITION 1. Let $S = \sigma_1\sigma_2 \cdots \sigma_n$ be a string over some alphabet Σ. The *factorized representation* of string S is the string $S' = \sigma_1'^{r_1}\sigma_2'^{r_2} \cdots \sigma_{\hat{n}}'^{r_{\hat{n}}}$ such that: (1) $\sigma_i' \neq \sigma_{i+1}'$ for $1 \leq i < \hat{n}$; and (2) S can be described as concatenation of the symbol σ_1' repeated r_1 times, the symbol σ_2' repeated r_2 times, ..., and the symbol $\sigma_{\hat{n}}'$ repeated $r_{\hat{n}}$ times.

We denote by $S^\Sigma = \sigma_1'\sigma_2' \cdots \sigma_{\hat{n}}'$, the *symbol part* of S, and by $S^\#$, the vector of natural numbers $r_1, r_2, \ldots, r_{\hat{n}}$, the *exponent part* of S.

EXAMPLE 1. For $S = AAABABBCCACAAAA$, $S' = A^3B^1A^1B^2C^2A^1C^1A^4$, $S^\Sigma = ABABCACA$, and $S^\# = [3, 1, 1, 2, 2, 1, 1, 4]$.

2.1.1. *Algorithm for the One-Dimensional Scaled Matching Problem*

Step 1. Derive the symbol string T^Σ and the exponent string $T^\#$ $(= t_1, \ldots, t_{\hat{n}})$ from the text string T. Similarly, derive P^Σ and $P^\#$ $(= p_1, \ldots, p_{\hat{m}})$ from the pattern string P.

Observation 1. Finding all occurrences of P in T scaled to k is equivalent to finding all locations i that satisfy conditions A and B below:

Condition A. There is an exact occurrence of P^Σ in location i of T^Σ.

Condition B.1. $t_i \geq kp_1$.

Condition B.2. $t_{i+1} = kp_2, \ldots, t_{i+\hat{m}-2} = kp_{\hat{m}-1}$.

Condition B.3. $t_{i+\hat{m}-1} \geq kp_{\hat{m}}$.

Step 2. Suppose $\hat{m} \geq 3$. Derive the *quotient* string $T' = t_2/t_1, t_3/t_2, \ldots, t_{\hat{n}}/t_{\hat{n}-1}$ from $T^\#$ and the quotient string $P' = p_3/p_2, p_4/p_3, \ldots, p_{\hat{m}-1}/p_{\hat{m}-2}$ from $P^\#$.

Observation 2. Suppose $\hat{m} \geq 3$. Condition B.2 from Observation 1 is satisfied for $k = t_2/p_2$ if and only if an occurrence of string P' starts at location $i + 1$ of string T'.

Step 3.1. Find all occurrences of string P^Σ in the string T^Σ and all occurrences of the string P' in the string T'. This is done by applying any linear time string matching algorithm.

Step 3.2. For each location i in T^Σ, such that P^Σ starts at i and P' starts at location $i + 1$ of T', check whether Conditions B.1 and B.3 extend to locations i and $i + \hat{m} - 1$ in T'. This will take $O(1)$ time per location i.

Comments. (1) Extension to the case where $\hat{m} < 3$ within the same complexity bound as claimed below is trivial. (2) Step 2 and Observation 2 follow Eilam-Tzoreff and Vishkin [EV-88].

Time. Step 1 needs $O(|P| + |T|)$ time. All other steps need $O(|P^{\Sigma}| + |T^{\Sigma}|)$ time. If the input is already provided in the factorized form, then the running time of the algorithm will be linear in the length of the input, and possibly sublinear in $|P| + |T|$.

2.2. *Longest Common Prefix of Two Suffixes*

Given a string $C = c_1, \ldots, c_l$ we would like to preprocess it so that the following query can be processed in constant time. *Longest common prefix* (LCP) *query*. Given two suffixes find their longest common prefix. Namely, given a suffix $C_i = c_i, \ldots, c_l$, and another suffix $C_j = c_j, \ldots, c_l$, we want to find the largest f such that $c_i, \ldots, c_{i+f} = c_j, \ldots, c_{j+f}$. Following [LV-89], we will do the following:

(1) Construct the classic suffix tree data-structure for string C. It turns out that any such LCP query can be presented as a lowest common ancestor query with respect to the suffix tree. So,

(2) Preprocess the tree so that queries requesting the lowest common ancestor of two nodes in the tree can be processed in constant time.

In the rest of Section 2.2, we elaborate on these two steps and explain how to perform them in linear time. Any LCP query will then be processed in constant time.

2.2.1. *Suffix Tree*

PROBLEM DEFINITION. Let $C = c_1, \ldots, c_l$ be a string, where $c_l = \$$ and the symbol \$ does not appear elsewhere in the string C. We define the *suffix tree* of C as follows:

(1) It is a tree in which all the edges of the tree are directed away from the root. The out-degree of each node of the tree is either zero (if the node is a leaf) or ≥ 2.

(2) *Leaves of the suffix tree.* Each suffix $C_i = c_i, \ldots, c_l$ of the string C defines a *leaf* of the tree. (The tree has l leaves.)

(3) *Internal nodes of the suffix tree.* Let C_i and C_j be any two suffixes. Suppose $c_i, \ldots, c_{i+f}$ is their longest common prefix. That is, $c_i, \ldots, c_{i+f}$ equals to $c_j, \ldots, c_{j+f}$ and $c_{i+f+1} \neq c_{j+f+1}$. Then, $c_i, \ldots, c_{i+f}$ defines an *internal node* (i.e., a node that is not a leaf) of the tree.

(4) *Edges of the suffix tree.* Let D be a contiguous substring of the string C. Let B be a proper prefix of D. Suppose also that both D and B define nodes of the tree. Then there is an *edge* connecting the nodes of D and B if there is no contiguous substring F of the string C such that the

following three conditions hold at once: F is a proper prefix of D, B is a proper prefix of F, and F defines a node of the tree.

Implementation Remark. For each node v of the tree, a contiguous substring $c_i, \ldots, c_{i+f}$ that defines it will be stored as follows: START(v) := i and LENGTH(v) := f.

Weiner [W-73] (and McCreight [Mc-76]) compute the suffix tree in $O(n)$ time when the size of the alphabet is fixed. If the alphabet of the pattern contains x letters then it is easy to adapt the algorithm of [W-73] to run in time $O(n \log x)$.

2.2.2. *Lowest Common Ancestor*

Recall that our goal is to find the largest f for which $c_i, \ldots, c_{i+f} = c_j, \ldots, c_{j+f}$. Let $\mathrm{LCA}_{i,j}$ be the lowest common ancestor (in short LCA) of the leaves of the suffixes $c_i, \ldots, c_l$ and $c_j, \ldots, c_l$ in the suffix tree. The desired f is simply LENGTH($\mathrm{LCA}_{i,j}$). Thus, the problem of finding this f is reduced to finding $\mathrm{LCA}_{i,j}$. We use the algorithm of [HT-84] (or the simpler algorithm of [SV-88]) for the purpose of computing LCAs in the suffix tree, whenever we need to find such an f throughout the algorithms.

Using the classification of [HT-84] we are interested in the *static lowest common ancestors* problem; where the tree is static, but queries for lowest common ancestors of pair of vertices are given on line. That is, each query must be answered before the next one is known. The suffix tree has $O(n)$ nodes. The algorithm of [HT-84] (or [SV-88]) preprocesses the suffix tree in $O(n)$ time. Then, given an LCA query it responds in $O(1)$ time.

2.3. *Range Minimum, All-Nearest Smaller Values, and Cartesian Trees*

Let $L = [l_1, \ldots, l_n]$ be an array of n numbers. All three definitions below relate to this array:

A *range minimum* query is of the form:

> Given a range of indices $[i, \ldots, j]$, where $1 \le i \le j \le n$,
>
> return an index k, $i \le k \le j$, such that $l_k = \min\{l_i, \ldots, l_j\}$.

The *all-nearest smaller value* (ANSV) problem is of the form: For every i, $1 \le i \le n$, find the largest index $l < i$ for which $l_l < l_i$, if such index exists, and the smallest index $s > i$ for which $l_s < l_i$, if such index exits.

A *Cartesian tree* of L is a rooted binary tree defined recursively as follows: Let $l_{\text{root}} = \min\{l_1, \ldots, l_n\}$ (if $\min\{l_1, \ldots, l_n\}$ is obtained at more than one index then *root* is the minimum such index). Then,

l_{root} is the *root* of the Cartesian tree
the *left child* of the root is the Cartesian tree of $[l_1, \ldots, l_{\text{root}-1}]$
the *right child* of the root is the Cartesian tree of $[l_{\text{root}+1}, \ldots, l_n]$.

In [GBT-84] it was shown that, following a linear time preprocessing of L, each range minimum query can be processed in $O(1)$ time and that the Cartesian tree of L can be constructed in time $O(n)$ quite easily, using range minimum queries. We will be using this idea in Algorithms B and C for slightly modified Cartesian trees. In [BBGSV-88], it was shown how to use the output of an ANSV computation for constructing a Cartesian tree. The idea is that the parent of node i in the Cartesian tree, for each $1 \leq i \leq n$, is the larger among its left and right nearest smaller neighbors. If one (left or right) of the nearest neighbors does not exist, then the parent is the existing node. If both left and right neighbors do not exist then node i is the root of the Cartesian tree. Relations among the lowest common ancestor, range minima, Cartesian tree, and ANSV problems in the context of parallel computation are discussed in [BSV-88, BV-89].

3. Two-Dimensional Sublinear Matching

Our main contribution is Algorithm C for the two-dimensional scaled matching problem. It is given in the next section. Algorithms A and B below gradually prepare the reader for the presentation of Algorithm C. They are both designed for a simpler problem: the *exact two-dimensional matching problem* with a finite alphabet. Algorithm A runs in time $O(n^2)$. This very simple algorithm provides a framework that guides our more advanced algorithms. Algorithm B is a sublinear algorithm for the same problem. All our algorithms will assume a finite alphabet. For general alphabet, the time complexity is multiplied by $\log m$. We note that the exact two-dimensional algorithms of [Ba-78, B-77] also become $O(n^2 \log m)$ for the general alphabet.

3.1. The First Algorithm

Algorithm A has three steps. The first step (Step A.1) is similar to Section 2.2. We construct data structures, based on the pattern matrix P and the text matrix T, that support $O(1)$ time retrieval of longest common prefix queries. The second and third steps are similar to many string matching algorithms. In Step A.2, we build a table based on analysis of the

pattern. In the final step (Step A.3) we scan the text based on the table of Step A.2 and find all the occurrences of the pattern in the text. During Steps A.2 and A.3 several kinds of queries arise. Each of these queries is answered in $O(1)$ time using the data structures constructed in Step A.1. In order to motivate the presentation we present the steps of the algorithm in reverse order. We start with Step A.3. This motivates Step A.2. Queries that may arise in these two steps motivate Step A.1. Incidentally, this line of presentation offers a top down description of Algorithm A.

ALGORITHM A.
Input: $n \times n$ text matrix T and $m \times m$ pattern matrix P.

A.3. *Text Analysis*

Scan each *column* j of T separately, as if it was a one-dimensional string. The scan is done in a similar way to the classical string matching algorithm of [KMP-77] with one major modification regarding comparison of elements of the pattern and the text: The symbol in position $[i, j]$ is taken to be the *whole row* of length m, $T[i, j]T[i, j+1]\cdots T[i, j+m-1]$, and it will be compared to a full row of the pattern. Routine COMPARE, given later, actually performs such comparisons. We will assume familiarity with the KMP algorithm and especially the concept of a failure array. Specifically,

Input for Step A.3. The data structure built for P and T, the pattern and the text matrices, in Step A.1; FAIL, the array of failure links that is set up in Step A.2.

Output: All indices of T where a copy of P begins.

```
var
   j, i, l: index
   {j indexes text columns; i indexes text rows;
   l indexes the pattern rows and FAIL array.}
for column j := 1 to n − m + 1 do
   i := 1; l := 1
   While i ≤ n do
      if COMPARE(T[i; j, j + 1, ..., j + m − 1], P[l; 1, ..., m])
      then i := i + 1; l := l + 1
           if l = m + 1
           then Output: there is a match at [i − m, j];
                l := FAIL(l)
      else l := FAIL(l)
   end {while}
end {for}
```

Routine COMPARE($T[i; j, j+1, \ldots, j+m-1], P[l; 1, \ldots, m]$)

A query requesting the comparison

$$T[i; j, j+1, \ldots, j+m-1] = ?P[l; 1, \ldots, m]$$

is answered in $O(1)$ time based on the data structure of Step A.1.

A.2. *Pattern Analysis*

Input for Step A.2. The data structure built for P, the pattern matrix, in Step A.1.

Output. The failure array FAIL for the pattern string $PAT = P_1, P_2, \ldots, P_m$, whose symbols are the rows of P, the pattern matrix; specifically, $P_i = P[i; 1, \ldots, m]$.

Computation. Apply the pattern analysis step of the KMP algorithm to PAT with one adjustment: Comparing two symbols $p_i = p_j$ translates into a query

$$P[i; 1, 2, \ldots, m] = ?P[j; 1, \ldots, m].$$

Such a query needs $O(1)$ time based on the data structure of Step A.1.

A.1. *Constructing Data Structures* (*Same as Section* 2.2)

A.1.1. Form a long string C as follows: Concatenate all rows of T and append to them a concatenation of the rows of P. Construct a suffix tree ST of C.

A.1.2. Apply to the tree ST the preprocessing stage of the lowest common ancestor algorithm of [HT-84 or SV-88].

Retrieving queries. The queries that arise in the pattern analysis and text analysis (Steps A.2 and A.3) have the following form: *Are two contiguous substrings of length* m, $c_i, c_{i+1} \ldots c_{i+m-1}$ *and* $c_j, c_{j+1} \ldots c_{j+m-1}$, *of the long string* C *equal*? This translates trivially into the following query: *Find whether the length of the longest common prefix of the two suffixes* $c_i, c_{i+1}, \ldots$ *and* $c_j, c_{j+1}, \ldots$ *in* C *is larger than* $m - 1$.

Finally, recall that Section 2.2.2 explains how to process longest common prefix queries using lowest common ancestor queries (with respect to the tree ST).

Comment. The reader may wonder why we did not simply replace every set of identical rows in the pattern by the same symbol, following Step A.1.1. This complication will serve the presentation of our more advanced algorithms.

end Algorithm A

Time.

Step A.1. The suffix tree can be computed in time $O(n^2 + m^2)$ [W-73]. We also preprocess this tree in time $O(n^2 + m^2)$ to enable processing LCA queries for any pair of nodes in constant time [HT-84, SV-88].

Step A.2 takes $O(m)$ time. This follows from the linearity of the pattern analysis in KMP and the constant time processing of an LCA query.

Step A.3 takes $O(n^2)$ time, by [KMP-77], and [HT-84] for the LCA.

3.2. The Second Algorithm

We describe now a "sublinear" exact two-dimensional algorithm (Algorithm B). As indicated in the Introduction, the goal and definitions of sublinearity are strongly biased towards presentation of the algorithms for the scaled two-dimensional string matching problem. The algorithm will achieve sublinear time if the following two ways for compression of the input are "sufficiently" effective:

- Compression of equal successive symbols within each row of the pattern and the text. This is similar to the one dimensional case (strings), as demonstrated in the preliminaries.
- Compression of equal successive subrows of the text or the pattern. For an illustration, see Fig. 2.

DEFINITION 2. A submatrix $T' = T[i_1, \ldots, i_2; j_1, \ldots, j_1 + m - 1]$ of T is a block in position i_1 of column j_1 if all rows of T' are equal and if no row can be added to T' without disrupting the above condition.

The efficiency of the algorithm is achieved by grouping the n columns of the text into sets of m successive columns each, as follows $\{1, \ldots, m\}$, $\{m + 1, \ldots, 2m\}, \ldots, \{(\lfloor n/m \rfloor - 1)m + 1, \ldots, \lfloor n/m \rfloor m\}$ (possibly followed

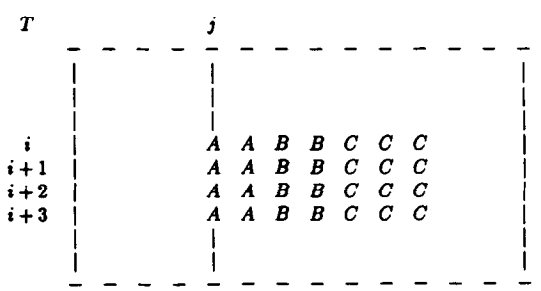

FIGURE 2

by a final set of less than m elements $(\{\lfloor n/m \rfloor m + 1, \ldots, n\})$. Instead of handling each of the columns separately, we combine the effort for processing all m columns in a set. The key player in each such set is its rightmost column, called *power column*. (Namely, columns m, $2m, \ldots, \lfloor n/m \rfloor m$ are the *power columns*.)

The components of Algorithm B are similar to Algorithm A. Algorithm B has three steps. Step B.1 constructs data structures that support constant time retrieval of longest common prefixes. A key addition is a data structure that allows for constructing all blocks of each column in time proportional to the actual number of blocks. Step B.2 builds a table based on pattern analysis. Step B.3 (the text analysis) will have two substeps. For each column, Step B.3.1 will find all its blocks. Finally, Step B.3.2 will scan the text by advancing through the blocks of each column.

ALGORITHM B.

Input: *FT* the factorized representation of the text matrix T and *FP* the factorized representation of the pattern matrix P. Both *FT* and *FP* are given in a factorized representation of the rows.

B.1. *Constructing Data Structures*

B.1.1. Form a long string C as follows. Concatenate all rows of *FT* and append to them all rows of *FP*. Construct a suffix tree *ST* of C.

Implementation Comment. Formally, the characters of a factorized representation are ordered pairs of the form (symbol, exponent). For instance, a row $a^3b^4c^5a^2b^5$ becomes $(a, 3)(b, 4)(c, 5)(a, 2)(b, 5)$. This means introduction of three new symbols, "(", ")" and "," to help distinguish between elements of the pairs.

B.1.2. Apply to the tree *ST* the preprocessing stage of a lowest common ancestor algorithm.

The above steps enable retrieval of longest common prefix queries with respect to suffixes of C. Step B.3 needs an additional data structure that enables retrieval of the following queries: *Given a power column* c *and another column* $j \geq c$ (alternatively, $j < c$), *find all rows* $1 \leq i \leq n - 1$ *such that rows* i *and* $i + 1$ *differ between columns* c *and* j (alternatively, between columns j and $c - 1$). *Formally*, $T[i; c, \ldots, j] \neq T[i + 1; c, \ldots, j]$ (alternatively, $T[i; j, j + 1, \ldots, c - 1] \neq T[i + 1; j, j + 1, \ldots, c - 1]$).

We would like the time to process this query to be proportional to the length of the output (that is, the number of such rows i). Steps B.1.3–B.1.4 construct such a data structure.

Let $[i, c]$ be a position on a power column c in T. Let $B_r[i, c]$ be the largest integer k for which the two strings $T[i; c, c + 1, \ldots, c + k - 1]$

and $T[i+1; c, c+1, \ldots, c+k-1]$ are equal. Similarly, let $B_l[i, c]$ be the largest integer k for which the two strings $T[i; c-k, c-(k-1), \ldots, c-1]$ and $T[i+1; c-k, c-(k-1), \ldots, c-1]$ are equal. In other words, $B_r[i, c]$ gives the longest common prefix of rows i and $i+1$ starting at column c and $B_l[i, c]$ gives the longest common suffix of rows i and $i+1$ ending at column $c-1$.

B.1.3. Trace each pair of rows, i and $i+1$, in FT, from right to left, and whenever a power column c is encountered, set $B_r[i, c]$.

(The number of operations is proportional to the sum of the factorized length of rows i and $i+1$ and the number of power columns, $\lfloor n/m \rfloor$.)

Similarly, derive $B_l[i, c]$ for all power columns c by tracing these rows from left to right.

EXAMPLE 2. Let row i in FT be $a^3b^4c^5a^2b^5$ and row $i+1$ be $c^2a^1b^4c^1a^1c^3b^1a^1b^5$. Suppose columns 5, 10 and 15 are power columns. The figure below illustrates rows i and $i+1$ in T, as well as the power columns:

$$\begin{array}{c} a\,a\,a\,b\,b\,b\,b\,c\,c\,c\,c\,c\,a\,a\,b\,b\,b\,b\,b \\ \qquad\quad | \qquad\quad | \qquad\quad | \\ c\,c\,a\,b\,b\,b\,b\,c\,a\,c\,c\,c\,b\,a\,b\,b\,b\,b\,b \end{array}$$

Then $B_r[i, 5] = 4$, $B_r[i, 10] = 3$, $B_r[i, 15] = 5$ and $B_l[i, 5] = 2$, $B_l[i, 10] = 0$, $B_l[i, 15] = 1$.

For each power column c, consider the arrays $B_r[1, \ldots, n-1; c]$ and $B_l[1, \ldots, n-1; c]$.

B.1.4. For each power column c, construct the Cartesian tree $CT_r[c]$ for array $B_r[1, \ldots, n-1; c]$ and another Cartesian tree $CT_l[c]$ for array $B_l[1, \ldots, n-1; c]$.

EXAMPLE 3. An example where $m = 10$ and $n = 36$ is given in Fig. 3. We relate to power column 10. In the figure we already extracted the values of $B_l[1, \ldots, n-1]$ and $B_r[1, \ldots n-1]$. The Cartesian tree $CT_l[10]$ (whose root is marked by the value 0) is laid on the left side of the power column. The Cartesian tree CT_r [10] (whose root is marked by the value 1) is laid on the right side of the power column.

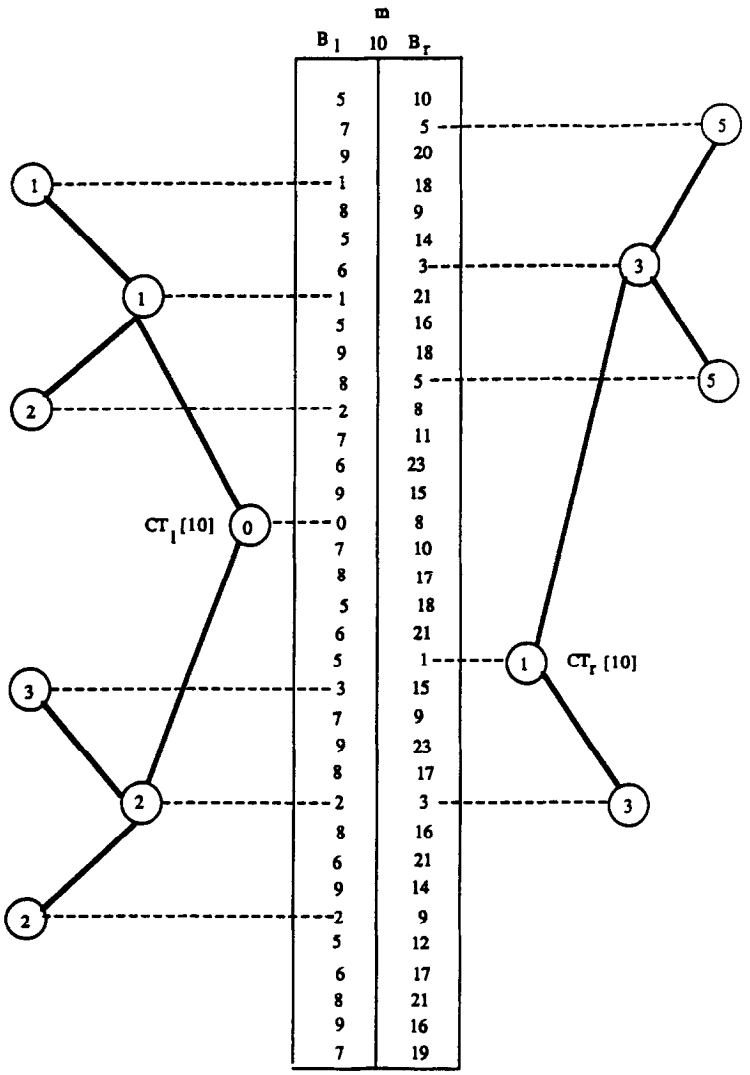

FIGURE 3

B.2. *Pattern Analysis*

Let $P_1, P_2, \ldots, P_m$ be the rows of the pattern matrix P and let $FP_1, FP_2, \ldots, FP_m$, respectively, be their factorized representation. FP can be described as row FP_{l_1} repeating r_1 times (the block at row l_1), followed by row $FP_{l_2}(\neq FP_{l_1})$ repeating r_2 times, followed by additional blocks until finally we have row $FP_{l_\alpha}(\neq FP_{l_{\alpha-1}})$ repeating r_α times; where $l_1 = 1$,

$l_k = r_1 + \cdots + r_{k-1} + 1$ and $r_1 + r_2 + \cdots + r_\alpha = m$. The above description is a factorized representation of the sequence $FP_1, FP_2, \ldots, FP_m$ when viewed as a one-dimensional string R with m symbols. Formally, a pair (FP_{l_k}, r_k) is a block of r_k rows at row l_k.

B.2.1. Compress the string R into its factorized representation, which is denoted FR.

(This will take time proportional to $|FP_1| + |FP_2| + \cdots + |FP_m|$.)

In the remainder of this presentation we assume that $\alpha > 2$. We leave it to the interested reader to see why there is a solution within our claimed bound for the case $\alpha \leq 2$. Let $\hat{FR}$ be the subsequence of FR that starts at its second element and ends at the predecessor of the last element. That is, $\hat{FR} = (FP_{l_2}, r_2) \ldots (FP_{l_{\alpha-1}}, r_{\alpha-1})$.

B.2.2. Compute the failure array FAIL for string $\hat{FR}$ using the suffix tree ST that was constructed in Step B.1 (as in Algorithm A).

B.3. *Text Analysis*

We restrict our explanation to columns $j = 1, \ldots, m$ and their representative power column m only. Other columns and power columns are treated similarly. Consider such column j.

The output of Step B.3.1 is a linked list, denoted S_j, whose elements are the blocks of column j. The key observation is that a block at column j ends at a row i if and only if at least one of the following two conditions is satisfied: (1) $B_l[i, m] < m - j$, or (2) $B_r[i, m] < j$. Recall that each number $B_l[i, m]$ defines a node of the Cartesian tree $CT_l[m]$ and each number $B_r[i, m]$ defines a node of the Cartesian tree $CT_r[m]$.

B.3.1.1. Get the list of all rows i (in increasing order of i) such that $B_l[i, m] < m - j$, from the Cartesian tree $CT_l[m]$.

B.3.1.2. Get the list of all rows i such that $B_r[i, m] < j$, from the Cartesian tree $CT_r[m]$.

B.3.1.3. Derive list S_j by merging the above two lists.

EXAMPLE 3 (Continued). In Fig. 3 we relate to column $j = 6$. The rows that we obtained in Steps B.3.1.1 and B.3.1.2 are marked.

Comment. For column j, Step B.3.1 can be implemented in time proportional to $|S_j|$, the number of blocks at column j.

B.3.2. *Additional preprocessing.* While scanning the text in Step B.3.3 we compare pattern rows and text substrings. For implementing such

comparisons, some preprocessing is needed. It is described in the context of routine COMPARE later.

B.3.3. *Scanning the text.* Scan each *column* j of T separately, as if it was a one-dimensional string. The scan is done in a similar way to Step A.3, with a few modifications. Most notable is the fact that instead of advancing from a row i to its successive row $i + 1$ in column j, our advancement is guided by the list S_j. Another modification is due to the fact that comparisons between pattern and text symbols are now comparisons between blocks.

Step B.3.3.1. *Output of Step* B.3.3.1. All indices of T where a copy of rows $P_{l_2} \ldots P_{l_\alpha - 1}$ of P begins.

```
var
   j, i, k: index
   {j indexes text columns; i indexes text rows;
   k indexes subindices of FR̂ and FAIL array.}
for column j := 1 to n − m + 1 do
   i := l_2 (start of second block in S_j); k := 2
   While i ≤ n do
      if COMPARE(block at T[i, j], block at row l_k of P)
      then i := i + r_k (start of next block in S_j);
      k := k + 1
         if k = α
         then
              Output: there is a match at [i − (m − l_α) + 1, j];
              k := FAIL(k)
      else k := FAIL(k)
   end {while}
end {for}
```

Step B.3.3.2. For every occurrence of $\hat{FR}$, check whether it extends into an occurrence of the whole pattern.

ROUTINE COMPARE.

Input. (1) Block at $T[i, j]$. (The algorithm guarantees that row i is in the list S_j and the number of rows in the block is $\text{succ}(i) - i$, where $\text{succ}(i)$ is the successor of row i in S_j.) (2) Block at row l_k of P (represented by the pair (FP_{l_k}, r_k)).

Output. Determine in $O(1)$ time whether $T[i; j, j + 1, \ldots, j + m - 1] = P_{l_k}$ and $\text{succ}(i) - i = r_k$.

The main tool in comparing the strings $T[i; j, j + 1, \ldots, j + m - 1]$ and P_{l_k} will be an LCA query with respect to the suffix tree ST. However, there is one difficulty. If $T[i, j - 1] = T[i, j]$, then the symbol in position

$[i, j]$ may not be represented in FT. As a result, the suffix starting at $T[i, j]$ will be missing from the suffix tree ST.

Following the precomputation of Step B.3.2 (see below), we have for each such location $[i, j]$, the smallest $j_1 \geq j$ for which the suffix starting at $T[i, j_1]$ appears in ST. So,

if $j_1 - j$ equals the first exponent in FP_{l_k}
then find whether the longest common prefix of the following two suffixes has at least
$m - (j_1 - j)$ characters:
(a) $T[i; j_1, \ldots]$
(b) $P_{l_k}[j_1 - j + 1, \ldots]$
(Both suffixes must be in ST)
and verify that $T[i, j] = P[l_k, 1]$

Implementation Remark. The longest common prefix query is implemented as an LCA query with respect to ST. The answer is in term of factorized length and special care is needed for interpretation of inequality at the last character of FP_{l_k}.

It remains to specify Step B.3.2.

B.3.2. *Details of additional preprocessing (continued).* Lists S_j contain all locations $[i, j]$ in the text to which Routine COMPARE has to be applied. Note that these lists are actually sorted lexicographically with j first and i second.

B.3.2.1. Sort these locations lexicographically with i first and j second. (This is done using bucket sort and results in a separate increasing list for each i.)

B.3.2.2. Scan row i of FT and form an increasing list of all locations $[i, j]$ in T for which the suffix $T[i; j, \ldots]$ appears.

B.3.2.3. For each i, merge the list of step (1) with the list of step (2) and thereby derive for each location $[i, j]$ of the first list the smallest $j_1 \geq j$ such that $[i, j_1]$ is in the second list.

end Algorithm B

Time. Step B.1.1. The length of string C is $|FT| + |FP|$. It was shown in the preliminaries that the suffix tree of string S can be constructed in time $O(|S|)$. However, this construction assumes a finite alphabet. In our case, the characters of C are pairs (σ, r), where σ is a symbol from a finite alphabet, but $1 \leq r \leq m$. Note, however, that r is the number of repetitions of σ in the uncompressed representation. The binary representation of r has $\log r \leq r$ bits. Let C_b be the concatenation of sequences $(\sigma, r_1, r_2, \ldots, r_{\log r})$, where r_i is the ith bit in the binary representation of r, FT_b be the part of C_b induced by FT and FP_b the part of C_b induced by FP.

Clearly, $|C| \leq |C_b| = |FP_b| + |FT_b| \leq (|P| + |T|)$. C_b is a string over a finite alphabet, so the suffix tree for C_b can be constructed in time $O(|C_b|)$. In fact, the suffix tree construction is the only place where an unbounded alphabet would be handled differently. For unbounded alphabet the time bound for Steps B.1.1 and B.1.2 becomes $O(|C_b| \log m)$.

Step B.1.1. $O(|C_b|)$.

Step B.1.2. $O(|C_b|)$ by [HT-84] or [SV-88].

Step B.1.3. $O(\lfloor n^2/m \rfloor + |FT| + |FP|)$.

Step B.1.4. $O(\lfloor n^2/m \rfloor)$ by [GBT-84].

Step B.2.1. $O(|FP|)$.

Step B.2.2. $O(|\hat{FR}|) \leq O(m)$ by [KMP-77].

Step B.3.1. $O(\Sigma_{j=1}^{n}$ number of blocks in column $j) = O(\Sigma_{j=1}^{n} |S_j|)$.

Step B.3.2. $O(\Sigma_{j=1}^{n} |S_j| + |FT|)$.

Step B.3.3. $O(\Sigma_{j=1}^{n} |S_j|)$.

Total time.

$$O\left(\left\lfloor \frac{n^2}{m} \right\rfloor + |C_b| + \sum_{j=1}^{n} |S_j|\right).$$

We explain each additive term in the above time bound and argue that under appropriate circumstances, the above complexity is sublinear. $\lfloor n^2/m \rfloor$ is less than n^2. As mentioned earlier, $|C_b| \leq |T| + |P|$. For example, if the entire text and pattern consist of a single repeating symbol then $|C_b| = (n + m)\log n < n^2$. $\Sigma|S_j|$ is the sum of all blocks in all columns. In the above example $\Sigma|S_j| = n < n^2$.

4. Scaled Matching in Linear Time

In this section we consider the two-dimensional scaled matching problem. Algorithm C, the main result in the present paper, solves this problem in time $O(n^2)$. In Section 5 we slightly modify Algorithm C and achieve sublinear time under some circumstances. Algorithm C is a generalization of Algorithm B. The main changes are in the text analysis (Step C.3). An overview of these changes is given as part of presenting this step.

The following definition generalizes the *block* concept:

k-blocks. Let k be a positive integer. A *k-block* at *position* i_1 *of column* j_1 of T is a submatrix $T' = T[i_1, \ldots, i_2; j_1, \ldots, j_1 + km - 1]$ that

satisfies the following:

(1) all rows of T' are equal;

(2) no rows can be added to T' without disrupting condition (1) above (formally, substring $T[i_2; j_1, \ldots, j_1 + km - 1]$ is not equal to substring $T[i_2 + 1; j_1, \ldots, j_1 + km - 1]$ and substring $T[i_1; j_1, \ldots, j_1 + km - 1]$ is not equal to substring $T[i_1 - 1; j_1, \ldots, j_1 + km - 1]$).

The *height* of a submatrix is its number of rows. The height of such k-block is $i_2 - i_1 + 1$. If a k-block is contained in a k-occurrence then its height must be at least k (because each pattern row must appear k successive times).

ALGORITHM C.

Input: FT the factorized representation of the text matrix T and FP the factorized representation of the pattern matrix P.

C.1. *Constructing Data Structures*

C.1.1. Form a long string C as follows. Concatenate all rows of FT and append to them the following $\lfloor n/m \rfloor$ strings:

1. A concatenation of the rows of FP.

2. A concatenation of the rows of P, where each row is scaled to 2 and given in a factorized representation. (These are the rows of FP where each exponent is simply doubled.)

3. For k, $3 \le k \le \lfloor n/m \rfloor$, a concatenation of the rows of P, where each row is scaled to k, and given in a factorized representation.

Similar to the time analysis of Step B.1.1, we have to actually compute the suffix tree of a string C_b, which is composed of FT_b followed by $(FP_j^k)_b$ for all rows j, $1 \le j \le m$ and $k, 1 \le k \le \lfloor n/m \rfloor$. The length of string C_b is equal to

$$|FT_b| + \sum_{j=1}^{m} \sum_{k=1}^{\lfloor n/m \rfloor} \left|\left(FP_j^k\right)_b\right|$$

which is $o(n^2)$, since

$$|FT_b| = o(n^2)$$

and

$$\sum_{j=1}^{m} \sum_{k=1}^{\lfloor n/m \rfloor} \left| \left(FP_j^k \right)_b \right| \leq m \sum_{k=1}^{\lfloor n/m \rfloor} m(\log k + 1) \leq nm \log \left\lfloor \frac{n}{m} \right\rfloor = o(n^2).$$

Construct a suffix tree ST of C.

C.1.2. Apply to the tree ST the preprocessing stage of a lowest common ancestor algorithm.

The above step will enable constant time comparison between a text substring and a row of P^k for any $1 \leq k \leq n/m$.

C.1.3. Construct $B_r[i, c]$ and $B_l[i, c]$, $i = 1, \ldots, n$; $c = m, 2m, \ldots, \lfloor n/m \rfloor$ as in Step B.1.3.

In Algorithm B, a Cartesian tree guided the division into blocks. In Algorithm C we need to divide into k-blocks for every scale k; $1 \leq k \leq \lfloor n/m \rfloor$. The Cartesian tree has actually more information than we need as it enables division into blocks with height $< k$. Since this is too costly, we modify the definition of a Cartesian tree so that blocks with small heights are not even considered.

A *k-height Cartesian tree* of L is a rooted binary tree defined recursively as follows:

Let $l_{\text{root}} = \min\{l_1, \ldots, l_n\}$ (if $\min\{l_1, \ldots, l_n\}$ is obtained at more than one index then *root* is the minimum such index). Then,

l_{root} is the *root* of the k-height Cartesian tree

the *left child* of the root is the k-height Cartesian tree of $[l_1, \ldots, l_{\text{root}-k}]$

the *right child* of the root is the k-height Cartesian tree of $[l_{\text{root}+k}, \ldots, l_n]$.

For efficiency reasons we will generate as much of each k-height tree as needed during the text analysis, so that the number of operations is proportional to the number of nodes that are actually needed in each k-height tree. Implementation of this approach needs constant-time retrieval of range minimum queries. For this, we perform the preprocessing algorithm of [GBT-84] in Step C.1.4 and use their $O(1)$ time retrieval of range minimum queries in the text analysis.

C.1.4. For each power column c, preprocess $B_r[i = 1, \ldots, n; c]$ and $B_l[i = 1, \ldots, n; c]$ for range minimum queries.

Some additional preprocessing is needed. The factorized representation easily allows construction of the following arrays. For each power column c:

1. $D_r[i, c]$ = the number of successive repetitions of the symbol in $T[i, c]$ starting from location $[i, c]$ and to the right. (Formally, $T[i, c] = T[i, c + 1] = \cdots = T[i, c + D_r[i, c]] \neq T[i, c + D_r[i, c] + 1]$.)

2. $D_l[i, c]$ = the number of preceding repetitions of the symbol in $T[i, c]$ starting from location $[i, c]$ and to the left.

C.1.5. For each power column c, preprocess $D_r[i = 1, \ldots, n; c]$ and $D_l[i = 1, \ldots, n; c]$ for range minimum queries.

C.2. *Pattern Analysis* (*Same as* B.2)

C.3. *Text Analysis*

An overview of the ideas of Algorithm C is delayed until Case 1.1 below. This case captures the main difficulties that we cope with. The other cases are needed to show that our algorithm is complete and capable of handling any anomaly that may come up.

Recall that in Step B.2.1 (of Algorithm B) the following assumption was made. There are at least two pattern rows i_1 and i_2, where $P[i_1; 1, \ldots, m] \neq P[i_1 + 1; 1, \ldots, m]$ and $P[i_2; 1, \ldots, m] \neq P[i_2 + 1; 1, \ldots, m]$. This assumption is not limiting since the scaling problem has an easy solution for the excluded case. Furthermore, this easy solution enables us to assume also that: *At least one row of the pattern has factorized length* > 1. If each pattern row consists of a single repeating symbol, then we view the pattern in column major order (a 90° rotation) and thereby get a pattern with no row changes. The aforementioned easy solution will take care of this. We consider two cases:

Case A. In *any* $m/2$ consecutive pattern rows there is *at least one* row of factorized length > 1.

Case B. Otherwise. That is, there are at least $m/2$ consecutive rows of factorized length 1.

Case A is treated in Algorithm C. Case B is treated in Algorithm S at the end of this section.

C.3.1.1. *Getting intervals of candidate scaled occurrences.* In order to achieve the claimed complexity results we treat each possible scale k separately one at a time. Different procedures are being used for scales that are smaller or equal than m (Case 1) and scales that are bigger than m (Case 2).

Case 1. $k \leq m$. Divide the rows of each power column c into *row groups* of $km/2$ successive rows each. (The last one may be smaller.)

For each group $[w_1, w_2]$ (where w_1 is the starting row and w_2 is the ending row) let r_1 be the minimum $D_r[i, c]$ over all i in $[w_1, w_2]$ and let i_1 be a row in $[w_1, w_2]$ for which this minimum is achieved (i.e., $D_r[i_1, c] = r_1$).

(*Comment*. Location $[i_1, c + r_1]$ is called an *anchor*.)

Case 1.1. $r_1 \leq (k - 1)m$. In terms of ideas, this is the main subcase. Below we present an overview of how this subcase is handled throughout the *entire* algorithm. (This is not just an overview of Step C.3.1.1.)

Overview of the text analysis step for Case 1.1. For power column c and its associated columns $c - m + 1, \ldots, c$, we explain how to find all occurrences of P^k that start in any row of these columns. For simplicity, we will assume in this overview that all row groups of power column c satisfy Case 1.1. For each row group $[w_1, w_2]$ we find in Step C.3.1.1 an anchor $[i_1, c + r_1]$, where $(c + r_1) \leq c + (k - 1)m$ and row i_1 is in the row group.

Observation. Any occurrence of P^k in columns $c - m + 1, \ldots, c$ of the text that contains location $[i_1, c + r_1]$ must start in a column j, where $c + r_1 - j$ is an integer multiple of k. This is true since each character of P repeats itself k times in each row of P^k. Specifically, such an anchor for row group $[w_1, w_2]$ enables us to restrict candidate occurrences starting in the *working area* defined below.

COROLLARY. *Consider the subarray consisting of rows* $w_1 - km/2, \ldots, w_1 - 1$ *and columns* $c - m + 1, \ldots, c$. *Then candidate* k*-occurrences can start only in columns* $c - m + 1 \leq j \leq c$, *for which* $(c + r_1 - j) \pmod{k} = 0$.

For implementing the restriction that is implied by the corollary we include an interval $[w_1 - km/2 \ldots w_1 - 1]$ in a *list of intervals* I_j^k for each such column j. We emphasize this connection between row groups and working areas. It helps in showing that all possible occurrences are considered. Let $[i, j]$ be a (starting location of a) candidate k-occurrence. (Formally, $[i, j]$ will be included in an interval of I_j^k.) We say that a k-occurrence at $[i, j]$ *extends* into rows $i, i + 1, \ldots, i + km - 1$ of column j and these rows define a *segment* $[i \ldots i + km - 1]$ on column j. Such segments will delimit the scope of our computation below. Our computation will not generate these segments explicitly but will rather derive them on-the-fly from the lists of intervals I_j^k. Step C.3.1.2 finds locations $[i, j]$ in which k-block starts only within segments of candidate k-occurrences. This procedure for finding k-blocks uses a k-height Cartesian tree. This implies the following problem: not all k-blocks are actually found. How-

ever, notice that we are only interested in k-blocks whose height is at least k. We are guaranteed that all k-blocks in a sequence of consecutive k-blocks of height $\geq k$ will be found by the k-height Cartesian tree, with the possible exception of the first k-block of the sequence. An S_j^k list connects all starting locations of k-blocks that were found. (These lists are similar to the S_j list of Algorithm B.) Finally, Step C.3.3 runs the one-dimensional version of a KMP-like scan on each S_j^k list. The KMP-like scan needs to be extended to cope with three additional problems: (a) The appearance of the first k-block will be verified separately, after the rest of the pattern is found. When comparing a k-block starting in position $[i, j]$ to a pattern block. (b) Verify that i and its predecessor in the list S_j^k belong in the same segment of a candidate k-occurrence on column j. Otherwise there is a gap between them where there cannot be any occurrence of P^k. (c) Verify that there are no hidden k-blocks between i and $i + k$. Namely, the k-block that starts at $[i, j]$ includes rows $i, \ldots, i + k$. If one of these conditions does not hold then the KMP-like scan will advance to check whether a k-occurrence starts at the next block in the S_j^k list. This concludes the overview.

We now proceed with a concrete presentation of Step C.3.1.1 for Case 1.1 and power column c. Case 1.1 implies that in any possible k-block starting in columns $c - m + 1, \ldots, c$ and rows $w_1 - km/2, \ldots, w_1 - 1$ there is a change of symbol in position $c + r_1$ and a k-block starting at one of the columns $c - m + 1, \ldots, c$ has to start in a position that is consistent with the column in which a change of symbol occurs. If $r_1 = ak + b$ for $1 \leq b \leq k$ then the columns in which P^k can start are $j = (c + b) - k$, $(c + b) - 2k, \ldots, (c + b) - \lfloor (m + b - 1)/k \rfloor k$. Therefore,

$$\text{for } j = (c + b) - k, (c + b) - 2k, \ldots, (c + b) - \left\lfloor \frac{m + b - 1}{k} \right\rfloor k$$

$$\text{Add interval } [w_1 - \frac{km}{2}, w_1 - 1] \text{ to } I_j^k,$$

end

Case 1.2. $(k - 1)m < r_1 \leq km$. (Note that r_1 is not relevant to any column j such that $c - m + 1 \leq j \leq c + r_1 - km$. For these columns use the symbol change to the left; see Case 1.3.)

$$\text{For all } j = (c + b) - k, (c + b) - 2k, \ldots, c + r_1 - km + 1$$

$$\text{Add interval } \left[w_1 - \frac{km}{2}, w_1 - 1 \right] \text{ to } I_j^k,$$

end

Case 1.3. $r_1 > km$. Let l_2 be the minimum $D_l[i, c]$ over all i in $[w_1, w_2]$ and let i_1 be a row in $[w_1, w_2]$ for which this minimum is achieved. Take $l_1 = l_2 - 1$.

If $l_1 < m$ then for all $j = (c - l_1), c - l_1 - k, c - l_1 - 2k, \ldots, (c - l_1) - \left\lfloor \frac{m - l_1 - 1}{k} \right\rfloor k$

Add interval $\left[w_1 - \frac{km}{2}, w_1 - 1 \right]$ to I_j^k,

end

If $l_1 > m$ then there is no symbol change in this work area and it cannot contribute to any I_j^k. To justify Step C.3.1.1 for Case 1 we explain why we cannot miss any k-occurrence. Given a k-occurrence at some location $[i, j]$, take the leftmost power column that intersects it. One of its row groups ($km/2$ consecutive rows) must be contained in the occurrence. In Case A, every $m/2$ consecutive pattern rows have at least one row of factorized length > 1. Such a row will provide an anchor. The anchor will imply adding an interval into I_j^k. This interval must include i.

Time for Step C.3.1.1 *on Case* 1. Consider the work per power column and scale k. There are $O(n/km)$ working areas and for each we may need to update at most $(m/k)I_j^k$ lists. Computing r_1 and l_1 takes constant time per working area using range minimum queries on the D_r and D_l arrays (see Section 2.3). The time is $O(\sum_{k=1}^{m}(n/km)(m/k) = O(n)$ per power column, and since there are n/m power columns, the total time for constructing the I_j^k lists for all $k \leq m$ is $O(n^2/m)$.

Remark. We could not give exactly the same treatment to $k > m$ and still get the same, or even linear, time bound. Note that the number of row groups per power column is $O(n/km)$. Each such group needs at least constant time. The total time is $O(\sum_{k=1}^{n/m}(n/km) = O(n \log n/m)$. Summing over all n/m power columns we get $O(n^2 \log n/m^2)$ time. For $m^2 = o(\log n)$ this becomes non-linear.

Case 2. $k > m$. The main reason for choosing the power columns at a distance of m from each other was so that every k-block will intersect a power column, even for $k = 1$. However, for larger scales, larger distances between each pair of power columns suffice for constructing the I_j^k lists. For $k \geq m$ this is done in precisely the same manner as for $k < m$ but using fewer power columns. Each row below gives a range of scales and the set of power columns needed for them.

Scale	Power columns
$k = 1, 2, \ldots, m-1$	$m, 2m, 3m, \ldots$
$k = m, m+1, \ldots, m^2-1$	$m^2, 2m^2, 3m^2, \ldots$
$k = m^2, m^2+1, \ldots, m^3-1$	$m^3, 2m^3, 3m^3, \ldots$
$\vdots$	$\vdots$

There are $\log n/\log m$ ranges of scales above. We already showed that for the first range (i.e., $k = 1, \ldots, m-1$) the time is $O(n^2/m)$. Let us analyze the second range. For each power column and scale $k = m, \ldots, m^2-1$, we have n/km row groups and for each we may need to update at most $(m^2/k)I_j^k$ lists. We bound the time per power column by $O(\sum_{k=m}^{m^2-1}(n/km)(m^2/k)) = O(nm\sum_{k=m}^{m^2-1}1/k^2) = O(nm(1/m)) = O(n)$. (To see that $O(\sum_{k=m}^{\infty}1/k^2) = O(1/m)$ integrate the function $1/x^2$). Since there are n/m^2 power columns, the total time for constructing the I_j^k lists for all $m \le k \le m^2$ is $O(n^2/m^2)$. Similarly, the third range of scales $k = m^2, \ldots, m^3-1$ takes $O(n^2/m^3)$ time, etc. Over all ranges of scales we obtain $O(n^2/m + n^2/m^2 + \ldots)$ which is $O(n^2/m)$ time.

C.3.1.2. *Getting k-blocks within intervals.* We now identify first rows of k-blocks within the segments that are to be scanned. The linked list connecting first rows of k blocks at column j is called S_j^k. The S_j^k lists are the output of Step C.3.1.2. We derive the segments to be scanned from the I_j^k list in a straightforward manner.

For each segment $[v, w]$ of column j and scale k: Construct the k-height Cartesian tree of $B_l[v, v+1, \ldots, w; c]$ and the k-height Cartesian tree of $B_r[v, v+1, \ldots, w; c]$ (where c is the power column of j).

These trees are produced in time $O((w-v)/k)$ by range minimum queries, each query providing a first row of a k-block.

Merge the above two k-height Cartesian trees into an S_j^k list.

Time. For a fixed scale k, the number of first rows of k-blocks that were actually found is $O(n^2/k^2)$.

S_j^k is now composed exactly of the k-blocks that are used by the scanning part of the algorithm.

C.3.2. *Additional preprocessing.* Same as B.3.2, but sort lexicographically with i first, j second, and k last.

C.3.3. *Scanning the text.* Same as B.3.3, but scanning is *separate* for every scale k. S_j^k thus replaces S_j as the list of next text block to scan. In addition to the change described in the above overview of Case 1.1, there is also a necessary change in COMPARE (block at $T[i, j]$, block at row l_s of P). The block at row l_s of P is a pair (FP_{l_s}, r_s). For scale k the block is actually $(FP_{l_s}^k, kr_s)$. This does not change the failure function of the pattern, but the comparison is different. The string comparison can be done in constant time since all rows of FP^k for all scales k appear in suffix tree ST. The numerical comparison is trivial.

end Algorithm C

Time complexity

Steps C.1.1 and C.1.2. $o(n^2)$.

Steps C.1.3–C.1.5. $O(n^2/m)$.

Step C.2. $O(|FP|)$.

Step C.3.1.1. $O(n^2/m)$.

Step C.3.1.2. For each scale k the total length of segments that are derived from the I_j^k lists cannot exceed $O(n^2/k)$. In each interval, only k-blocks are accessed; i.e., the processing time for each interval is at most interval length$/k$. Therefore, the processing time for each scale k is at most $O(n^2/k^2)$. Summing over all k we obtain

$$O\left(\sum_{k=1}^{\lfloor n/m \rfloor} \frac{n^2}{k^2}\right) = O(n^2).$$

Steps C.3.2 and C.3.3. $O(\text{size of } S_j^k \text{ lists}) \leq O(n^2)$.

In order to finish Section 4, we still need to consider Case B, where the pattern has $m/2$ consecutive rows of factorized length 1. In addition, we may assume that we also have at least $m/2$ consecutive pattern columns, whose factorized length is 1—otherwise apply Algorithm C to a rotation by 90° of the text and pattern. We observe that these (at least) $m/2$ consecutive rows must all be equal and these (at least) $m/2$ consecutive columns must also all be equal. Therefore, we call this collection of rows, the *repeating symbol block*. We consider two subcases:

Case B.1. There is at least one pattern row of factorized length > 2. Algorithm S treats this case. Case B2 is discussed later.

ALGORITHM S.

S.1 *Preprocessing*. Same as Algorithm C. This special case does not require pattern preprocessing. We are now ready to find all scaled appearances of P. Let P_i, row i of P, be the lowest row of factorized length > 2 that appears above the repeating symbol block. (The case where this row is below the repeating symbol block is handled symmetrically.) Row P_i has factorized length > 2 and say that rows $i + d + 1, \ldots, i + d + l$, $l > m/2$ have factorized length 1. (Rows $i + 1, \ldots, i + d$ have factorized length 2.)

S.2. Find all scaled occurrences of P_i in T. This can be done in time $O(|FT|)$ by the algorithm for one dimensional scaled matching presented in Section 2. Since only one P_i^k can appear in a given location, there are at most $|FT|$ such locations. Note that only one P_i^k can start at a given text

location. The reason is as follows: Let a^r be the smallest internal run of P_i (i.e., ba^rc appears in P_i, where $b, c \neq a$, and r is the smallest repetition where such a situation occurs). Then a^{rk} is the smallest internal run of P_i^k, which precludes $P_i^{k'}$, $k' \neq k$, from starting at the same location.

S.3. Discard every location $[l_1, l_2]$ where the following two conditions do not both hold:

S.3.1. There is a k-block of height k starting at $[l_1, l_2]$ whose rows all equal P_i^k.

S.3.2. There are $kl(> km/2)$ equal subrows of length km and factorized length 1, starting at position $[l_1 + k(d+1), l_2]$.

S.3.1 can be verified in constant time by range minimum queries to B_r and B_l; and LCA queries with respect to ST. S.3.2 can be verified in constant time by range minimum queries to D_r, D_l, B_r, and B_l.

Assume location $[l_1, l_2]$ passed verifications S.3.1 and S.3.2. Without loss of generality we assume that the first symbol's change in P_i appears in the first half of row P_i. This means that it is impossible for any of the $k^2m^2/4$ locations $[l, l']$, $l = l_1 + k(d+1), \ldots, l_1 + k(d+1) + km/2$; $l' = l_2, \ldots, l_2 + km/2$ to satisfy condition S.3.1. Therefore, the total number of remaining locations after Step S.3 is at most $4n^2/k^2m^2$, for a given k. Each remaining location $[l_1, l_2]$ defines one candidate k-occurrence in location $[s_1, s_2] = [l_1 - k(i-1), l_2]$.

S.4. For every scale k and potential starting point $[s_1, s_2]$ do
For $e = 0$ to $m - 1$ do
S.4.1 verify that $T[s_1 + ek; s_2, \ldots, s_2 + km - 1] = P_{e+1}^k$.
S.4.2 verify that all rows $T[s_1 + ek; s_2, \ldots, s_2 + km - 1], \ldots,$
$T[s_1 + ek + k - 1; s_2, \ldots, s_2 + km - 1]$ are equal.
end

All locations that passed the verification of Step S.4 are starting occurrences of P^k. Step S.4.1 can be done in constant time in a manner similar to subroutine COMPARE of Algorithm C. Step S.4.2 can be done in constant time by range minimum queries on B_l and B_r. Therefore, for any given k and potential starting point $[s_1, s_2]$ an occurrence of P^k can be verified in time $O(m)$.

end Algorithm S

Total time for Algorithm S:

$$O(|FT|) + O\left(\sum_{k=1}^{\lfloor n/m \rfloor} \frac{n^2}{k^2m^2} m\right) = O(|FT|) + O\left(\frac{n^2}{m}\right).$$

Case B2. Each pattern row (and column) has factorized length ≤ 2. In this case, the pattern consists of at most two repeating symbols. Suppose the pattern consists of two symbols a and b; and $T[1, 1] = a$, then each row (or column) that has two symbols starts with a's and ends with b's. Also given a location $T[i, j] = a$ we must have that for all $x \leq i$ and $y \leq j$, $T[x, y] = a$; and given a location $T[i, j] = b$ we must have that for all $x \geq i$ and $y \geq j$, $T[x, y] = b$. Algorithm S can handle this special case with some minor changes. (*Hint*: In Algorithm S a row P_i of factorized ≥ 3 is needed for deriving the time complexity of $O(|FT|)$, since only one P_i^k can appear in a given location of FT. Instead, in Case B2 we will find a pair of successive rows whose change of letters occur in different columns. These two columns, when using the pair of rows together, provide the same property. If there are no such columns the situation is even simpler.)

5. Concluding Remarks and Open Problems

When scaled matchings are being searched, it is quite reasonable to assume that the text has large blocks of repeating characters. These repetitions were assumed to be reflected in the input representation, yielding a shorter input. For Algorithm B we mentioned two types of repetitions: (i) a character repeating itself successively within the same row; (ii) a row (of the pattern) that repeats itself successively. Algorithm C took advantage of these two types of repetitions.

In the following three implementation remarks, we discuss ways for reducing the linear running time of our algorithm to sublinear time. These ways are particularly effective around areas where k-occurrences, for relatively large k, are discovered.

Implementation Remark 1. Steps C.1.1 and C.1.2 (in Algorithm C) require $O(|FT_b| + \sum_{j=1}^{m} \sum_{k=1}^{\lfloor n/m \rfloor} |(FP_j^k)_b|)$ time, which is $o(n^2)$. There, we compute a suffix tree of the rows of the pattern in all scales, as well as of rows of the text. Alternatively, we can compute two suffix trees: (i) A suffix tree for the symbol part of the rows of the pattern and the text. (ii) For each row of the pattern and the text get the quotient string of the row (see Section 2.1). The second suffix tree will be for these quotient strings. These two new suffix trees provide the same information as the old one in Algorithm C, but require only $O((|FT_b| + |FP_b|))$ computation time.

Implementation Remark 2. Even if there are no large height k-blocks at all, Step C.3.1.2 may still enforce a running time of $O(n^2)$. This can be avoided by a divide-and-conquer construction of the k-height trees that, by means of lookahead, discards large chunks of rows where there are no m successive k-blocks of height $\geq k$.

Implementation Remark 3. Step C.3 treated each scale k separately. Consider treating the scales in decreasing order, from the largest possible, which is $\lfloor n/m \rfloor$, down to 1. While investigating occurrences of P^k for a relatively large k we may discover contiguous k-blocks whose heights are each integer multiples of k. If the number of such contiguous blocks is large enough we may conclude that l-occurrences (for $l < k$) that are contained in these contiguous k-blocks, are impossible. Hence, an implementation idea is to discard from consideration containment of l-occurrences by excluding appropriate sections from the lists of intervals I_j^l. When later, the scale l is treated, we simply skip over the excluded sections.

We have barely scratched the surface on the subject of sublinearity. The compressed representation we used was a concatenation of the factorized representation of the rows. Clearly this is an inherently one-dimensional approach. Even here our algorithm is probably not optimal. One would like the time complexity to be linear in the size of the factorized representation.

Several inherently two-dimensional compressions such as medial axis [RK-82] and quadtrees [S-90] are known. To our knowledge, no known compression scheme is particularly tailored for pattern searches in the compressed representation. We conclude with the following intriguing open problem. Is there a two-dimensional compression where exact matching or perhaps even scaled matching, can be done in time linear (or close to linear) in the size of that compression?

REFERENCES

[AC-75] A. V. AHO AND M. J. CORASICK, "Efficient string matching," *Comm. ACM* **18**, No. 6, (1975), 333–340.

[AILSV-88] A. APOSTOLICO, C. ILIOPOULOS, G. M. LANDAU, B. SCHIEBER, AND U. VISHKIN, Parallel construction of a suffix tree with applications, *Algorithmica* **3** (1988), 347–365.

[AL-90] A. AMIR AND G. LANDAU, Fast parallel and serial multidimensional approximate array matching, "Sequences: Combinatorics, Compression, Security and Transmission" (R. Capocelli, Ed.), New York/Berlin, Springer-Verlag, pp. 3–24, 1990; *Theoret. Comp. Sci.* **81**, No. 1 (1991), 97–115.

[Ba-78] T. J. BAKER, A technique for extending rapid exact-match string matching to arrays of more than one dimension, *SIAM J. Comput.* **7** (1978), 533–541.

[B-77] R. S. BIRD, Two dimensional pattern matching, *Inform. Process. Lett.* **6**, No. 5 (1977), 168–170.

[BBGSV-89] O. BERKMAN, D. BRESLAUER, Z. GALIL, B. SCHIEBER, AND U. VISHKIN, Highly parallelizable problems, in "Proceedings, 21st ACM Symposium on Theory of Computing, 1989," pp. 309–319. (This conference version contains results from [BSV-88].

[BSV-88] O. BERKMAN, B. SCHIEBER, AND U. VISHKIN, "Some Doubly Logarithmic Optimal Parallel Algorithms Based on Finding All Nearest Smaller Values," UMIACS-TR-88-79, University of Maryland, College Park, MD 20742, October 1988.

[BV-89] O. BERKMAN AND U. VISHKIN, Recursive *-tree parallel data structure, *in* "Proceedings, 30th IEEE Symp. on Foundations of Computer Science, 1989," pp. 196–202; *SIAM J. Comput.*, to appear.

[BM-77] R. S. BOYER AND J. S. MOORE, A fast string searching algorithm, *Comm. ACM* **20** (1977), 762–772.

[EV-88] T. EILAM-TZOREFF AND U. VISHKIN, Matching patterns in a string subject to multi-linear transformations, *Theoret. Comput. Sci.* **60** (1988), 231–254.

[FP-74] M. J. FISCHER AND M. S. PATERSON, String matching and other products, *in* "Complexity of Computation" (R. M. Karp, Ed.), SIAM-AMS Proceedings, Vol. 7, pp. 113–125, Amer. Math. Soc., Providence, RI, 1974.

[GBT-84] H. N. GABOW, J. L. BENTLEY, AND R. E. TARJAN, Scaling and related techniques for geometry problems, *in* "Proceedings 16th ACM Symposium on Theory of Computing, 1984," pp. 135–143.

[GP-90] Z. GALIL AND K. PARK, An improved algorithm for approximate string matching, *SIAM J. Comput.* **19** (1990), 989–999.

[GS-83] Z. GALIL AND J. I. SEIFERAS, Time-space-optimal string matching, *J. Comput. System Sci.* **26** (1983), 280–294.

[HT-84] D. HAREL AND R. E. TARJAN, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.* **13** (1984), 338–355.

[KS-87] K. KRITHIVASAN AND R. SITALAKSHMI, Efficient two dimensional pattern matching in the presence of errors, *Inform. Sci.* **43** (1987), 169–184.

[KMP-77] D. E. KNUTH, J. H. MORRIS, AND V. R. PRATT, Fast pattern matching in strings, *SIAM J. Comput.*, **6** (1977), 323–350.

[LV-89] G. M. LANDAU AND U. VISHKIN, Fast parallel and serial approximate string matching, *J. Algorithms* **10**, No. 2 (1989), 157–169.

[Mc-76] E. M. MCCREIGHT, A space-economical suffix tree construction algorithm, *J. Assoc. Comput. Mach.* **23** (1976) 262–272.

[RK-82] A. ROSENFELD AND A. C. KAK, "Digital Picture Processing," 2nd ed., Academic Press, New York, 1982.

[S-90] H. SAMET, "The Design and Analysis of Spatial Data Structures," Addison-Wesley, Reading, MA, 1990.

[SV-88] B. SCHIEBER AND U. VISHKIN, On finding lowest common ancestors: Simplification and parallelization, *SIAM J. Comput.*, **17** (1988), 1253–1262.

[U-85] E. UKKONEN, Finding approximate pattern in strings, *J. Algorithms* **6** (1985), 132–137.

[V-85] U. VISHKIN, Optimal parallel pattern matching in strings, *Inform. Control* **67** (1985), 91–113.

[V-91] U. VISHKIN, Deterministic sampling—A new technique for fast pattern matching, *SIAM J. Comput.* **20**, No. 1 (1991), 22–40.

[W-73] P. WEINER, Linear pattern matching algorithm, *in* "Proceedings, 14th IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.