INFORMATION AND COMPUTATION 117, 78–90 (1995)

Multiple Matching of Rectangular Patterns

RAMANA M. IDURY*,† AND ALEJANDRO A. SCHÄFFER‡

Rice University, Houston, Texas 77251
E-mail: idury@hto.usc.edu Schaffer@cs.rice.edu

We describe the first worst-case efficient algorithm for simultaneously matching multiple rectangular patterns of varying sizes and aspect ratios in a rectangular text. Efficient means significantly more efficient asymptotically than applying known algorithms that handle one height (or width or aspect ratio) at a time for each height. Our algorithm features an interesting use of multidimensional range searching, as well as new adaptations of several known techniques for two-dimensional string matching. We also extend our algorithm to a dynamic setting where the set of patterns can change over time.

1. INTRODUCTION

Searching for a fixed pattern in a text is one of the most basic problems in string matching. We define this problem as:

• *Fixed pattern matching* (FPM). Given a *fixed* pattern P over an alphabet Σ, preprocess it so as to be able to find all its occurrences in a *query* text T.

There are several solutions (e.g., Knuth, Morris, and Pratt, 1977; Boyer and Moore, 1977) that preprocess the pattern in time $O(p)$, where p is the pattern length, and search a text of length t in time $O(t)$.

In this paper we study two important extensions to the basic FPM problem: multiple patterns and two-dimensional rectangular strings (both as text and as patterns). Our main result is the first efficient algorithm for matching multiple rectangular patterns in a rectangular text. In particular, our algorithm handles patterns of arbitrary sizes and aspect ratios. Our result addresses a longstanding open problem posed by Baker (1978). The natural extension of FPM to multiple one-dimensional patterns is defined as:

• *Multiple pattern matching* (MPM). Given a *fixed* set of patterns $P_1, ..., P_n$ over an alphabet Σ, preprocess it so as to be able to search for all occurrences of all the patterns in a *query* text T. The set of patterns is also called a *dictionary*, denoted by D. The size (i.e., sum of lengths of all patterns) of the dictionary is denoted by d.

Aho and Corasick, or AC for short, solved this problem with linear preprocessing and searching times (Aho and Corasick, 1975). The salient feature of their algorithm is the extension of the Knuth–Morris–Pratt (1977) (henceforth KMP) algorithm for FPM to multiple patterns. The AC algorithm preprocesses the patterns in time $O(d \log \sigma)$ and searches a text in time $O(t \log \sigma + \text{tocc})$, where σ is the number of characters that occur in all patterns, and tocc is the total number of pattern occurrences. The AC algorithm can be trivially modified to report just the longest pattern that matches (ending) at each position of the text in time $O(t \log \sigma)$.

To extend FPM to two dimensions we make both the text and the pattern rectangular arrays. Define:

• *Two-dimensional fixed pattern matching* (TFPM). Given a *fixed* rectangular pattern P, preprocess it so as to be able to find all its occurrences in a rectangular *query* text T.

The TFPM problem was originally solved by Bird (1977) and independently by Baker (1978). The basic idea in their algorithms is to *linearize* the pattern by treating each column as a single character in a new derived alphabet. Bird and Baker use the AC algorithm as a subroutine to transform the columns of pattern and text into these special characters and then run the KMP algorithm to search the modified text. The preprocessing time is $O(p \log \sigma)$ and the search time is $O(t \log \sigma)$. Other algorithms that seem to perform better on random patterns have been described in (Zhu and Takaoka, 1989; Baeza-Yates and Régnier, 1990). Recent papers by Amir, Benson, and Farach (1992) and Galil and Park (1992) give new algorithms whose running times are linear without depending on the alphabet size.

Since he used an algorithm for MPM to solve the TFPM problem, it is not surprising that Baker (1978) observed that it is natural to combine the two paradigms. Define:

• *Two-dimensional multiple pattern matching* (TMPM). Given a *fixed* set of rectangular patterns $P_1, ..., P_n$ over an alphabet Σ, preprocess it so as to be able to search for all

* Present address: Department of Mathematics, University of Southern California, Los Angeles, CA 90089-1113.
† Partially supported by a grant from the W. M. Keck Foundation.
‡ Partially supported by NSF Grant CCR-9010534.

0890-5401/95 $6.00

occurrences of all the patterns in a rectangular *query* text T. The set of patterns is also called a *dictionary*, denoted by D. The size (i.e., sum of areas of all patterns) of the dictionary is denoted by d.

Template-based computer vision is one natural motivation for TMPM. We can think of a seeing being as having a mental dictionary of known templates (patterns) that are quickly matched against a new scene (text). For other applications of two-dimensional matching and a general survey with lots of references see Section 7.3.2 of (Gonnet and Baeza-Yates, 1991). Much of that section focuses on a "dual" problem where there are multiple texts and one pattern; Giancarlo (1992) has some recent improvements to those results.

If all the patterns in an instance of TMPM are of fixed height, then Baker (1978) noted that his algorithm can be adapted to solve TMPM simply by replacing the use of the KMP algorithm for text scanning with a use of the AC algorithm. The search time for one height or width becomes $O(t \log(n+\sigma) + \text{tocc})$. When the number of different heights, widths, and aspect ratios is large, we would like an algorithm whose performance depends as little as possible on the number of different pattern sizes. Baker explicitly left open this question of how to solve TMPM efficiently when the patterns have varying heights and widths (Baker, 1978).

Amir and Farach (1992) found an efficient algorithm for TMPM in the special case where all the patterns are square; a similar algorithm was given in (Giancarlo, 1993). These algorithms use $O(d \log(n+\sigma))$ time for pattern preprocessing and $O(t \log(n+\sigma))$ time for searching. Square patterns have the special property that they can be aligned at a corner. Amir and Farach used this fact to linearize the text down the diagonals (instead of across rows or down columns, as Bird and Baker did) and apply a generalization of the AC algorithm to the linearized text.

We use a different approach from (Amir and Farach, 1992; Giancarlo, 1993) to solve TMPM for rectangular patterns. Rectangular patterns of different heights, widths, and aspect ratios cannot be aligned at a corner. Thus there is no natural way to define the "biggest suffix" of the text that matches a "prefix" of some pattern.

Our first basic idea is to split the patterns into two subpatterns each, so that the number of different subpattern heights and widths is small. In the text scanning phase we recognize occurrences of the subpatterns and then combine occurrences of subpatterns that represent an occurrence of a full pattern. The most interesting feature of our algorithm is a connection between two-dimensional pattern matching and some multidimensional range searching problems. Our TMPM algorithm also includes new adaptations of three techniques from algorithms for *other* two-dimensional string matching problems: splitting patterns into two overlapping pieces of fixed height (Gonnet, 1992), focusing on text columns that are (numbered) 0 mod q for some appropriate q (Amir, Landau, and Vishkin, 1990; Baeza-Yates and Régnier, 1990), and the smaller matching with tree partial order paradigm, which was defined and studied in (Amir and Farach, 1991a). For other, different uses of computational geometry in string matching see (Manber and Baeza-Yates, 1991; Amir, Landau, and Vishkin, 1990).

The preprocessing time, text scanning time, and space of our TMPM algorithm depend in complicated ways on the distribution of patterns among the different sizes because this affects which known range-searching algorithm is best to use. Following the general trend in string matching, we restrict ourselves to variations of our algorithm that use linear, i.e., $O(d)$, space, and get the best time possible using linear space. We do not count the $O(t)$ disk space needed to store the input text.

Define the *linear size* $b(P)$ of a rectangular pattern P to be the *smaller* of its height and width. Let B be the biggest linear size of any pattern. To express the possible running times of our algorithm we give one definition.

DEFINITION 1.1. We call a dictionary *size − diverse* if for some fixed $k > 1$, the number of patterns of any fixed linear size $b \geq \log d$ is $O(b^k)$.

For size-diverse dictionaries our algorithm achieves the following time bounds using linear space:

Dictionary preprocessing time. $O(d \log(n+\sigma))$.

Text scanning time. $O(t \log d \log(B+n+\sigma) + \text{tocc})$.

There are many dictionaries that are not size-diverse for which the above time bounds still hold. For any dictionary, we can achieve the combination:

Dictionary preprocessing time. $O(d \log(n+\sigma))$.

Text scanning time. $O(t \log^2 d \log(B+n+\sigma) + \text{tocc})$.

It is surprising that our algorithm may achieve better times for size-diverse dictionaries because this looks like the hardest case from the perspective of the Bird-Baker algorithm.

The rest of this paper is organized as follows. In Section 2 we give basic definitions and an overview of our method. In Section 3 we present the connection between pattern matching and computational geometry showing how to use the connection to preprocess the patterns and do the final stage of matching against the text. In Section 4 we give more details of the text scanning algorithm and prove the resource bounds. In Section 5 we show how our algorithm can be extended to a dynamic setting where the set of patterns can change over time.

2. DEFINITIONS AND ALGORITHMS OVERVIEW

In this section we give some basic definitions, introduce some auxiliary data structures, and give an overview of the

text scanning algorithm. The algorithm begins by making two classifications into cases. First, we assume that all patterns have height $\leqslant$ width; the opposite case can be handled in the same way after transposing all patterns and the text. Second, we partition the set of patterns into $\lfloor \log B \rfloor$ sets $\mathscr{P}_0, \mathscr{P}_1, \ldots$, such that:

DEFINITION 2.1. The patterns in set $\mathscr{P}_g$ have a height h in the range $2^g < h \leqslant 2^{g+1}$.

For each g, we search the text for just the patterns in $\mathscr{P}_g$ in one phase. Since there are at most $\lfloor \log B \rfloor$ sets of patterns, this contributes a multiplicative factor of $O(\log B)$ to the text scanning time, but it does not change the asymptotic preprocessing time or space.

From now on we focus on one specific $\mathscr{P}_g$ and assume that we are just interested in the patterns in that set. We use the following example to illustrate various definitions.

EXAMPLE 2.2. Consider a dictionary of two 3×4 patterns. Here, $g = 1$ and $h = 3$:

$$\begin{array}{cccc|cccc} a & b & c & a & b & c & a & b \\ c & a & a & b & a & a & b & c \\ a & b & b & a & b & b & a & b \end{array}$$

Our first key idea is to divide each pattern into two pieces, such that each piece has height 2^g and full width. For a pattern of height h, the top piece includes rows $1, \ldots, 2^g$, and the bottom piece includes rows $h - 2^g + 1, \ldots, h$. The two pieces share $2^{g+1} - h$ rows and will overlap unless $h = 2^{g+1}$.

DEFINITION 2.3. We call the pattern pieces *half patterns*; the top pieces are called *upper half patterns*, and the bottom pieces are called *lower half patterns*.

In Example 2.2, the upper half patterns are

$$\begin{array}{cccc|cccc} a & b & c & a & b & c & a & b \\ c & a & a & b & a & a & b & c \end{array}$$

And the lower half patterns are

$$\begin{array}{cccc|cccc} c & a & a & b & a & a & b & c \\ a & b & b & a & b & b & a & b \end{array}$$

DEFINITION 2.4. Let C be the set of columns that occur in some half pattern

As in the Bird–Baker algorithm, each column is assigned a number (memory address) that we think of as a character, and then we view each half pattern as a one-dimensional string of column characters. In Example 2.2, the set of columns is $C = \{ab, ac, ba, bc, ca, cb\}$, where each column is written as a string written top-to-bottom. For convenience, we write a two-dimensional pattern as a one-dimensional string of columns, where each column is separated by a vertical bar. With this notation, we can write the first pattern of Example 2.2 as $aca \mid bab \mid cab \mid aba$, its upper half pattern as $ac \mid ba \mid ca \mid ab$, and its lower half pattern as $ca \mid ab \mid ab \mid ba$.

Since all half patterns are of the same height, we could use the Bird–Baker algorithm to recognize all occurrences of half patterns. To do the efficient synthesis of half patterns into full patterns, we need extra preprocessing.

DEFINITION 2.5. A *prefix* (*suffix*) of a rectangular pattern is a prefix (suffix) of its string-of-column-character representation. The *reverse* of a rectangular pattern is the reverse of its string-of-column-character representation (i.e., the pattern read right-to-left)

In Example 2.2, $bab \mid cab \mid aba$ is a prefix of the second pattern and a suffix of the first pattern, whereas $bcb \mid aba \mid cab \mid bab$ is the reverse of the second pattern. We maintain two extra dictionaries of upper half patterns.

DEFINITION 2.6. The dictionary $H^{f,u}$ (H for half, f for forward, u for upper) contains the 2^g *longest* prefixes of each upper half pattern. The dictionary $H^{b,u}$ (b for backward) contains the 2^g *smallest* non-empty prefixes of the reverse of each upper half pattern; in other words, $H^{b,u}$ contains the last 2^g non-empty suffixes of each upper half pattern, with the prefixes stored in reverse fashion. We also maintain two similar dictionaries $H^{f,l}$ and $H^{b,l}$ for lower half patterns.

In Example 2.2, $H^{f,u} = \{ac \mid ba \mid ca \mid ab, ac \mid ba \mid ca, ba \mid ca \mid ab \mid bc, ba \mid ca \mid ab\}$, $H^{f,l} = \{ca \mid ab \mid ab \mid ba, ca \mid ab \mid ab, ab \mid ab \mid ba \mid cb, ab \mid ab \mid ba\}$, $H^{b,u} = \{ab, ab \mid ca, bc, bc \mid ab\}$, and $H^{b,l} = \{ba, ba \mid ab, cb, cb \mid ba\}$.

The reason for storing the prefixes and reverse prefixes is that our final scanning pass will look only in text columns that are numbered $0 \bmod 2^g$. Following (Amir, Landau, and Vishkin, 1990), we call these *power columns*. If there is a match spanning columns $[c_1 \cdots c_2]$ of the text, we will find it in the rightmost power column $c \leqslant c_2$. The pattern occurrence must include at least one power column because its width is assumed to be at least as big as its height, which is at least $2^g + 1$. The part of the match in text columns $c_2, c_2 - 1, \ldots, c$ (read from right to left) will have its upper half in $H^{b,u}$ and its lower half in $H^{b,l}$. The part of the match in text columns $c_1, c_1 + 1, \ldots, c$ (read from left to right) will have its upper half in $H^{f,u}$ and its lower half in $H^{f,l}$.

Let P be a pattern of width w. Since we cannot predict where in P an occurrence will first intersect a power column, we define:

DEFINITION 2.7. An *avatar*[1] of pattern P is an ordered pair (P_f, P_b) such that P_f is a prefix of P of length at least $w - 2^g + 1$ and P_b is the reverse of a suffix of P of length at most 2^g. The strings P_f, P_b overlap in exactly one column

[1] The word avatar comes from Sanskrit and originally means an incarnation of a Hindu deity. In English, avatar also can mean a variant phase of continuing basic entity (Webster's Dictionary).

character. We say that P_f is the *forward projection* of the avatar and P_b is the *backward projection* of the avatar.

Our intention is that a match of pattern P can be viewed as that avatar, (P_f, P_b), where P_f is the part to the left of and including the power column and P_b is the part to the right of and including the power column. Each pattern has 2^g avatars. In Example 2.2, the avatars of the first pattern are $(aca\,|\,bab\,|\,cab\,|\,aba, aba)$ and $(aca\,|\,bab\,|\,cab, aba\,|\,cab)$. The avatars of the second pattern are $(bab\,|\,cab\,|\,aba\,|\,bcb, bcb)$ and $(bab\,|\,cab\,|\,aba, bcb\,|\,aba)$.

For every h, $2^g < h \leqslant 2^{g+1}$, we maintain 2 one-dimensional dictionaries on the avatars of all patterns of height h.

DEFINITION 2.8. The dictionary F_h contains the forward projections of the avatars of all the patterns. Similarly, B_h contains all the backward projections of the avatars of all the patterns

In Example 2.2, $F_3 = \{aca\,|\,bab\,|\,cab\,|\,aba, aca\,|\,bab\,|\,cab, bab\,|\,cab\,|\,aba\,|\,bcb, bab\,|\,cab\,|\,aba\}$, and $B_3 = \{aba, aba\,|\,cab, bcb, bcb\,|\,aba\}$.

We also extend the definitions of avatar and projections to half patterns. Note that we have set up the auxiliary dictionaries so that the forward projection of any avatar of any upper half pattern is in $H^{f,u}$, the backward projection of any avatar of any upper half pattern is in $H^{b,u}$, and similarly for lower half patterns.

Our text scanning algorithm has two basic parts, a text preprocessing step and a matching step. In the preprocessing step our goal is to find for text location $T[i, j]$ in power column j the widest elements of $H^{f,u}$ and $H^{f,l}$ that match the text when their upper *right* corner is placed at $T[i, j]$. The answers are placed in the auxiliary arrays FU, FL. Similarly, we seek the widest elements in $H^{b,u}$ and $H^{b,l}$ that match the text when their upper *left* corner is placed at $T[i, j]$. These answers are stored in the arrays BU, BL.

In the text scanning phase, we ask for each location $T[i, j]$ in a power column and each height $h \in (2^g, 2^{g+1}]$: are there any matches of patterns such that the top row is i, the rightmost power column is j, and the height of the pattern matched is h? It is important to see that we ask as many as 2^g queries at each power column position, but there are at most $\lfloor t/2^g \rfloor$ power column positions, so the total number of queries is $O(t)$.

To answer a query of height h at $T[i, j]$ we use the information in $FU[i, j]$, $FL[i, j]$, $BU[i+h-2^g, j]$, and $BL[i+h-2^g, j]$. In the next section we show how to transform our queries into geometric problems.

3. PREPROCESSING AND MATCHING PATTERNS

In this section we investigate how to synthesize projections and subpattern matches into full pattern matches. Our synthesis uses a geometric approach. We describe how to use our geometric pattern representations. The synthesis from projections and subpattern matches to full pattern matches is carried out in three steps as follows:

Step 1: For a text location $T[i, j]$ on a power column j, we compute $w_f[i, j]$ which is the widest forward projection of a pattern of height h matching at $T[i, j]$. For this we take the values $FU[i, j]$ and $FL[i+h-2^g, j]$, which are pointers into $H^{f,u}$ and $H^{f,l}$, respectively, and transform them into $w_f[i, j]$ which is a pointer in F_h.

Step 2: We do a similar procedure on $BU[i, j]$ and $BL[i+h-2^g, j]$ and obtain the widest backward projection $w_b[i, j]$ of a pattern of height h matching at $T[i, j]$.

Step 3: We use $w_f[i, j]$ and $w_b[i, j]$ to report all the matching patterns at the subrow $T[i, j]$ through $T[i, j+2^g-1]$.

We now define a computational geometric problem and one variant of it. We reduce each of the above steps to the variants.

- *Rectangle enclosure reoprting* (RER) (Edelsbrunner and Overmars, 1982). Given a set V of rectangles in the plane and another such query rectangle Q, the *rectangle enclosure reporting* problem asks for reporting *all* rectangles in V that enclose Q.

- *Nested rectangle enclosure searching* (NRES). We guarantee that the set V is *nested* which means that for $v_1, v_2 \in V$, if v_1 and v_2 intersect then one of them encloses the other. In this case, we report the *smallest* rectangle v in V that encloses Q. Since V is nested the choice of v is unique.

The rectangle enclosure reporting problems can be solved efficiently using multidimensional range searching algorithms (Lee and Wong, 1981; Edelsbrunner and Overmars, 1982; Gabow, Bentley, and Tarjan, 1984). In the rest of the section, we show the reductions from steps 1, 2, 3 to RER or NRES. To perform the reductions we define a *dominance* relation on strings as follows.

DEFINITION 3.1. For any two strings u_1 and u_2, u_2 *dominates* u_1 or u_1 is *dominated by* u_2, denoted $u_1 \leqslant_s u_2$, if u_1 is a suffix of u_2. We also extend the definition of dominance to a pair of strings as follows: $(u_1, v_1) \leqslant_s (u_2, v_2)$ if $u_1 \leqslant_s u_2$ and $v_1 \leqslant_s v_2$.

For example, $cbc \leqslant_s abcbc$ and $bab\,|\,cab\,|\,aba \leqslant_s aca\,|\,bab\,|\,cab\,|\,aba$. Similarly, $(cbc, aba) \leqslant_s (abcbc, bcaba)$ and $(bab\,|\,cab\,|\,aba, aba) \leqslant_s (aca\,|\,bab\,|\,cab\,|\,aba, bcb\,|\,aba)$.

DEFINITION 3.2. Let D be a one dimensional dictionary automaton constructed using the AC algorithm (Aho and Corasick, 1975). For any prefix u in D, define fail(u) as the longest proper suffix v of u such that v is also a prefix in D. The fail link of u points to v.

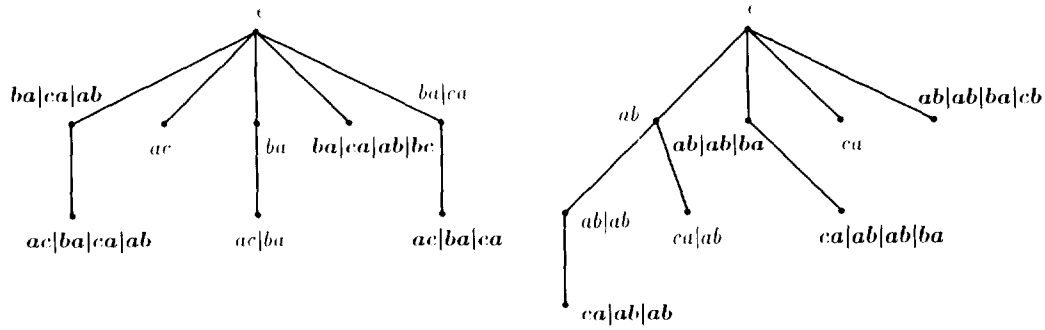

FIG. 1. Prefixes of failtree($H^{f,u}$) and failtree($H^{f,l}$).

Fact 3.3. The *fail* links of a dictionary D form a tree which we call the *fail tree* of the dictionary, denoted by failtree(D).

Figure 1 shows failtree($H^{f,u}$) (left) and failtree($H^{f,l}$) (right) of Example 2.2. Patterns are shown in bold font. The *fail* of a node is its parent in the fail tree. We can conclude from Definition 3.2 and Fact 3.3 that:

Fact 3.4. (Aho and Corasick, 1975; Amir and Farach, 1991a). If u_1 and u_2 are prefixes of some patterns in a dictionary D, then $u_1 \leqslant_s u_2$ if and only if u_1 is an ancestor of u_2 in failtree(D).

Fact 3.5. (Idury and Schäffer, 1992; Amir, Farach, Idury, La Poutré, and Schäffer, 1993), When we perform an Euler tour of a tree starting at the root we visit each node twice, once in the preorder and once in the postorder. If we replace the first visit by a left parenthesis and the second visit by a right parenthesis we get a list of balanced parentheses.

We use Fact 3.5 to transform the fail tree of a one-dimensional dictionary D with m prefixes to an equivalent list of $2m$ balanced parentheses by performing an Euler tour of failtree(D). Each matching pair of parentheses corresponds to a node in the tree which also corresponds to a prefix of some pattern. If we do a bijective mapping of the list of $2m$ parentheses to the integer points $0 \ldots m-1$ on the real line by assigning to each parenthesis its rank in the list, then each prefix of a pattern corresponds to an interval on the real line, with endpoints corresponding to the ranks of its associated parenthesis pair. It may be noted that these intervals are nesting because the corresponding parentheses are balanced. We call this scheme the *line-mapping* of a dictionary D on to the real line, wherein we map failtree(D) as described above.

DEFINITION 3.6. For a pattern $P \in D$, we use line(P) to denote the interval to which P is mapped. More generally, if u is a prefix of some pattern in D, then line(u) denotes the interval to which u is mapped. Similarly, we use line(D) to denote the set of intervals formed.

Figure 2 shows the corresponding intervals of the trees in Fig. 1. Once again, intervals of patterns are shown in bold font. The following lemma follows from the above definition:

LEMMA 3.7. *Suppose* $u, v \in D$. *Then* $u \leqslant_s v$ *if and only if* line(u) *encloses* line(v).

If u_1, v_1 are prefixes in a dictionary D_1 and u_2, v_2 are prefixes in a dictionary D_2, then $(u_1, u_2) \leqslant_s (v_1, v_2)$ if and only if u_1 is ancestor of v_1 in failtree(D_1) and u_2 is an ancestor of v_2 in failtree(D_2). The idea of examining the position of the pattern in multiple trees is inspired by the *smaller matching problem* studied in (Amir and Farach,

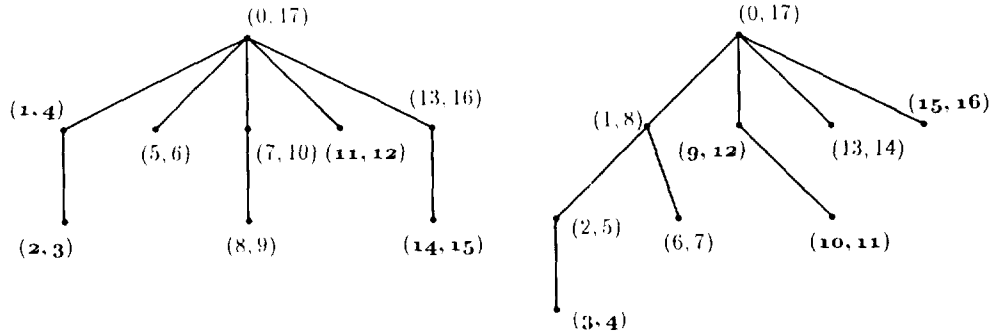

FIG. 2. Intervals of failtree($H^{f,u}$) and failtree($H^{f,l}$).

1991a). In that context, a pattern matches at a text position if each pattern character is less than or equal to the corresponding text character when evaluated in some partial order on the alphabet. Here we are just comparing two character strings, but the characters are themselves pattern prefixes.

If we line-map D_1 on x-axis and D_2 on y-axis, then each pair (u_1, u_2) is mapped to two intervals—line(u_1) on x-axis and line(u_2) on y-axis—which induce a rectangle. We call this scheme the *rectangle-mapping* of a pair of dictionaries (D_1, D_2) on to the plane.

DEFINITION 3.8. For a pair of patterns $P_1 \in D_1$ and $P_2 \in D_2$, we use rect(P_1, P_2) to denote the rectangle to which (P_1, P_2) is mapped. More generally, if u_1 is a prefix in D_1 and u_2 is a prefix in D_2 then rect(u_1, u_2) denotes the rectangle to which (u_1, u_2) is mapped. Similarly we use rect(D_1, D_2) to denote the set of rectangles formed.

In Example 2.2, if we line-map $H^{f,u}$ along the x-axis, and $H^{f,l}$ along the y-axis, the patterns of F_3 are rectangle-mapped as shown in Fig. 3. Note that the rectangles in this case are nesting. The following lemma follows from the above definition.

LEMMA 3.9. *Suppose* $u_1, v_1 \in D_1$ *and* $u_2, v_2 \in D_2$. *Then* $(u_1, u_2) \leqslant_s (v_1, v_2)$ *if and only if* rect(u_1, u_2) *encloses* rect(v_1, v_2).

We are now ready to show the reductions:

THEOREM 3.10. *We can reduce both Step* 1 *and Step* 2 *to NRES.*

Proof. We first reduce *Step* 1 to NRES. For every forward projection P_f of a pattern of height h in F_h, its upper half pattern P_f^u is in $H^{f,u}$ and its lower half pattern P_f^l is in $H^{f,l}$. Under the rectangle-mapping of $(H^{f,u}, H^{f,l})$, $P_f = (P_f^u, P_f^l)$ is transformed into a rectangle in the plane. Let $V_h^f = \{\text{rect}(P_f^u, P_f^l) \mid P_f \in F_h\}$ be the set of rectangles corresponding to the patterns in F_h. We want to set $w_f[i, j]$ to the widest P_f such that $(P_f^u, P_f^l) \leqslant_s (FU[i, j], FL[i+h-2^g, j])$. Since $(FU[i, j], FL[i+h-2^g, j])$ is also transformed into another rectangle in the plane, we have that rect(P_f^u, P_f^l) is the *smallest* rectangle in V_h^f that encloses rect$(FU[i, j], FL[i+h-2^g, j])$ by Lemma 3.9.

Recall that F_h contains the forward projections of patterns of height h. Since we linearize the patterns along their columns, F_h is actually a one-dimensional dictionary. Therefore the choice of the widest pattern P_f in F_h is unique. We can infer from this unique choice that the set of rectangles rect$(H^{f,u}, H^{f,l})$ is nesting. Therefore the choice of the widest pattern P_f in F_h, or equivalently the smallest rectangle rect(P_f^u, P_f^l) in V_h^f is unique.

We can reduce *Step* 2 to NRES using a similar procedure to the above. We define V_h^b similarly. We set $w_b[i, j]$ to the widest $P_b = (P_b^u, P_b^l)$ in B_h such that $(P_b^u, P_b^l) \leqslant_s (BU[i, j], BL[i+h-2^g, j])$. ∎

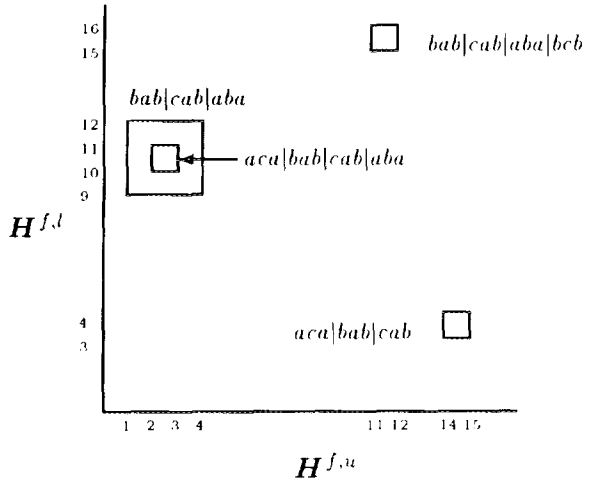

FIG. 3. Rectangle-mapping of patterns in F_3.

THEOREM 3.11. *We can reduce Step* 3 *to RER.*

Proof. The reduction is similar to the one used in Theorem 3.10. Let D_h be the set of patterns of height h in $\mathscr{P}_g$, where $2^g < h \leqslant 2^{g+1}$. Consider a pattern $P \in D_h$. P has 2^g avatars with each avatar having its forward projection in F_h and its backward projection in B_h. Under the rectangle-mapping of (F_h, B_h), each avatar of $P = (P_f, P_b)$ is mapped into a rectangle in the plane. Let $V_h = \{\text{rect}(P_f, P_b) \mid P \in D_h\}$ be the set of rectangles corresponding to the patterns in D_h. Similarly, $(w_f[i, j], w_b[i, j])$ is also mapped into a rectangle. A pattern P is matched at some location in the subrow $T[i, j]$ through $T[i, j+2^g-1]$ if and only if it has an avatar (P_f, P_b) such that $(P_f, P_b) \leqslant_s (w_f[i, j], w_b[i, j])$. It follows from the rectangle-mapping of (F_h, B_h) that rect(P_f, P_b) encloses rect$(w_f[i, j], w_b[i, j])$ by Lemma 3.9. To find all the matches with their top row in text row i and j as their rightmost power column, we perform a query of reporting *all* enclosing rectangles of rect$(w_f[i, j], w_b[i, j])$ in V_h. ∎

We now express the time and space required to preprocess patterns of height h in $\mathscr{P}_g$ as a function of the time and space needed for RER and NRES.

LEMMA 3.12. *Let* n_h *be the number of patterns in* D_h. *Similarly, let* d_h *be the total size of all patterns in* D_h. *Suppose that the preprocessing time and space bounds for either RER or NRES with* m *rectangles are* $O(m \cdot T_{\text{rect}}(m))$ *and* $O(m \cdot S_{\text{rect}}(m))$, *respectively. Given* C, *we can build* F_h, B_h,

V_h^f, V_h^b, V_h in time $O(d_h \log n_h + h \cdot n_h \cdot T_{\text{rect}}(h \cdot n_h))$, and space $O(d_h + h \cdot n_h \cdot S_{\text{rect}}(h \cdot n_h))$.

Proof. Consider a pattern P of height h in D_h. Let $(P_f^1, P_b^1), \ldots, (P_f^{2g}, P_b^{2g})$ be the avatars of P, such that $P_f^1 = P$. From the definition of an avatar, P_f^i is a prefix of P_f^{i-1} for $1 < i \leqslant 2^g$. Similarly P_b^i is a prefix of P_b^{i+1} for $1 \leqslant i < 2^g$. Since any dictionary automaton containing a pattern P contains all the prefixes of P, there is no penalty in terms of time and space to include a prefix of P as a new pattern. Following this reasoning, the avatars of P can be added in F_h or B_h without any extra cost. Therefore the resource bounds for building F_h and B_h are $O(d_h \log(n_h + \sigma))$ time and $O(d_h)$ space, respectively, which follow from the bounds of the AC algorithm. The reason the time is $O(d_h \log n_h)$ is that in an AC goto tree, each state can have only n_h outgoing edges.

Let us bound the number of rectangles in V_h^f, V_h^b, and V_h. The number of avatars of P are 2^g. Since $2^g < h$, the total number of avatars is bounded by the sum of heights of the patterns in D_h which is $h \cdot n_h$. Since the size of the image of any mapping is at most equal to the size of the domain of the mapping, it follows that the number of rectangles generated in any reduction is at most equal to the number of avatars. The time and space bounds to reduce each step to either RER or NRES follow. ∎

We are now ready to present the pseudocode for the preprocessing of patterns.

ALGORITHM 1. Code for preprocessing patterns.

```
PREPROCESS(P_g)
NOTE ∀p ∈ P_g, height(p) ⩽ width(p) and 2^g < height(p) ⩽ 2^(g+1)
 1  Build the dictionary C of columns of height 2^g
 2  Build H^(f,u), H^(f,l), H^(b,u), H^(b,l)
 3  For each h, 2^g < h ⩽ 2^(g+1)
 4       Build the dictionary of columns of height h
 5       Let D_h ⊆ P_g be the set of patterns with height h
 6       Assume w.l.o.g. that D_h is non-empty. Build D_h
 7       For each pattern P ∈ D_h
 8            For each avatar (P_f, P_b) of P
 9                 Add rect(P_f^u, P_f^l) to V_h^f
10                 Add rect(P_b^u, P_b^l) to V_h^b
11                 Add rect(P_f, P_b) to V_h
12       Build the data structure needed to solve NRES on V_h^f and V_h^b
13       Build that data structure needed to solve RER on V_h
```

We are now ready to prove the time and space bounds for the preprocessing of patterns.

THEOREM 3.13. *Let p_g denote the total size of all pattterns in $\mathscr{P}_g$. Suppose that the preprocessing time and space for either RER or NRES with m rectangles are $O(m \cdot T_{\text{rect}}(m))$ and $O(m \cdot S_{\text{rect}}(m))$, respectively. Let $\pi_g = \sum_h (h \cdot n_h)$ be the total number of avatars generated by patterns in $\mathscr{P}_g$. We can preprocess $\mathscr{P}_g$ in time $O(p_g \log(n + \sigma) + \pi_g \cdot T_{\text{rect}}(\pi_g))$, and space $O(p_g + \pi_g \cdot S_{\text{rect}}(\pi_g))$.*

Proof. Using analysis similar to that in the previous proof, it follows that the dictionaries C, $H^{f,u}$, $H^{f,l}$, $H^{b,u}$, $H^{b,l}$ can be built in $O(p_g \log(n + \sigma))$ time and $O(p_g)$ space using the AC algorithm. From Lemma 3.12, steps 3–13 take time $O(\sum_h (h \cdot n_h \cdot T_{\text{rect}}(h \cdot n_h)))$. From this we conclude that steps 1–13 take time $O(p_g \log(n + \sigma) + \pi_g \cdot T_{\text{rect}}(\pi_g))$. Similarly we can prove that steps 1–13 take space $O(p_g + \pi_g \cdot S_{\text{rect}}(\pi_g))$. Therefore the preprocessing time and space bounds follow. ∎

4. PREPROCESSING AND SEARCHING THE TEXT

In this section we give some more details of the scanning algorithm outlined in the previous section, summarize the scanning algorithm in pseudocode and prove the time bounds we claimed in Section 1.

The pseudocode for the scanning algorithm is given below. We discuss three details of it that are not apparent from the description in Section 3.

First, our use of the AC algorithm to report all half-pattern columns occurring in a text column is a little unusual. We use the AC algorithm scanning down each column seeking to report at each location $T[i, j]$ whether a half-pattern column matches starting at $T[i, j]$ and if so, which column matches. Normally, the AC algorithm records its matches at the end of the match, where they are detected. In our application, each half pattern column is of height 2^g. Thus there can be at most one match ending (starting) at a given position in a text column and the length of the match is known. Therefore, we can modify the AC algorithm to record the column matches at the starting (highest) position, by subtracting $2^g - 1$ from the row number where the match ends.

Second, although it is possible to use our geometric approach for any distribution of patterns, we get better bounds *with linear space* by using the Bird–Baker algorithm for small pattern heights. Specifically, we define a height $h_{\min}$ such that for patterns of height $\leqslant h_{\min}$ we use the Bird–Baker algorithm for each separate height. This takes time $O(h_{\min} \cdot t \log(n + \sigma) + \text{tocc})$. We choose suitable values of $h_{\min}$ later.

Third, from the description of the algorithm it may appear that we use $O(t)$ extra space to store the arrays T_c, *FU*, *FL*, *BU*, *BL* because they store one item per text character. There is a standard trick to keep the space to

$O(d)$, when $d \ll t$. Split the text into overlapping patches of size $2d \times 2d$ and run the algorithm separately on each patch. Each text position occurs in at most four patches.

EXAMPLE 4.1. We use the following section of a text T to explain the scanning process. We use the dictionary of Example 2.2 for the purpose. Later we show how we match patterns for locations $T[i, j]$ through $T[i, j+1]$. Here j is a power column.

	$j-3$	$j-2$	$j-1$	j	$j+1$
i	a	b	c	a	b
$i+1$	c	a	a	b	c
$i+2$	a	b	b	a	b

We are now ready to give the pseudocode for the scanning algorithm:

ALGORITHM 2. Code for scanning a text with $\mathscr{P}_g$.

```
SCAN(T)
 0  Suppose g > h_min. Otherwise run Bird-Baker algorithm directly
 1  Scan each column of T top-to-bottom with C, giving T_c
 2  Scan each row of T left-to-right with H^{f,u}, giving FU
 3  Scan each row of T left-to-right with H^{f,l}, giving FL
 4  Scan each row of T right-to-left with H^{b,u}, giving BU
 5  Scan each row of T right-to-left with H^{b,l}, giving BL
 6  For each h, 2^g < h ≤ 2^{g+1}
 7       For each power column j (i.e., j mod 2^g = 0) of T
 8            Set w_f[i, j] to the smallest rectangle in V_h^f enclosing rect(FU[i, j], FL[i + h - 2^g, j])
 9            Set w_b[i, j] to the smallest rectangle in V_h^b enclosing rect(BU[i, j], BL[i + h - 2^g, j])
10            Let M ∈ V_h be the set of rectangles enclosing rect(w_f[i, j], w_b[i, j])
11            Report the corresponding patterns of M
```

We now illustrate the scanning algorithm, under the assumption that $h_{\min} = 0$, using Example 4.1. Since $g = 1$ in this case, we are operating in the range $2 < h \leq 4$. We show how we find all the matching patterns for the subrow $T[i, j]$ through $T[i, j+1]$ with a *single* query at location $T[i, j]$.

After step 5, we have $FU[i, j] = ac \mid ba \mid ca \mid ab$, $FL[i+1, j] = ca \mid ab \mid ab \mid ba$, $BU[i, j] = bc \mid ab$, and $BL[i, j] = cb \mid ba$. At step 6, we deal with the case $h = 3$ since the patterns in Example 2.2 are of height 3. At steps 8 and 9, we set $w_f[i, j] = aca \mid bab \mid cab \mid aba$ and $w_b[i, j] = bcb \mid aba$. Finally at steps 10 and 11, we report the actual matches as follows. The first pattern $aca \mid bab \mid cab \mid aba$ is matched at location $T[i, j]$ since it has an avatar $(aca \mid bab \mid cab \mid aba, aba)$ dominated by $(w_f[i, j], w_b[i, j])$. From Theorem 3.11, it follows that the rectangle of this avatar is in M at step 10. Similarly, the second pattern $bab \mid cab \mid aba \mid bcb$ is matched at location $T[i, j+1]$ since it has an avatar $(bab \mid cab \mid aba, bcb \mid aba)$ dominated by $(w_f[i, j], w_b[i, j])$. Accordingly, its rectangle is also in M at step 10.

The correctness of the algorithm follows from Section 3. It remains to prove the running time bounds and make sure that we use only $O(d)$ space. We first state two important results from previous papers.

LEMMA 4.2. (Lee and Wong, 1981; Edelsbrunner and Overmars 1982). *The RER and NRES problems can be reduced to four-dimensional range searching problem.*

Proof. Let each rectangle R have extreme x values $(x_{\min}(R), x_{\max}(R))$ and extreme y values $(y_{\min}(R), y_{\max(R)})$. Then rectangle R encloses the query rectangle Q if and only if:

$$x_{\min}(R) \leq x_{\min}(Q)$$
$$x_{\max}(R) \geq x_{\max}(Q)$$
$$y_{\min}(R) \leq y_{\min}(Q)$$
$$y_{\max}(R) \geq y_{\max}(Q)$$

Thus if we represent each rectangle by its four extreme coordinates, we seek in the RER problem to report all rectangles R that satisfy the four constraints above with respect to Q. To solve *NRES* we seek the enclosing rectangle with maximum $x_{\min}$ and $y_{\min}$. ∎

In fact, the problems are equivalent in some sense (Edelsbrunner and Overmars, 1982). Those readers familiar with range searching will note that in the above reduction the four constraint intervals for Q are unbounded on one side. This enables us to use a result from (Gabow, Bentley, and Tarjan, 1984) who already noted its applicability to the counting variant of RER.

LEMMA 4.3 (Gabow, Bentley, and Tarjan 1984). *There is a data structure that enables us to store m rectangles in two dimensions with $S_{\text{rect}} = T_{\text{rect}} = O(\log^2 m)$ and answer RER queries in time $O(\log^2 m + \text{nrect})$, where* nrect *is the number of rectangles reported. For the NRES problem, then additive term* nrect *is dropped.*

We now give our first result on time bounds:

THEOREM 4.4. *We can solve the TMPM problem using $O(d)$ space and times*:

Preprocessing. $O(d \log(n+\sigma))$.

Text scanning. $O(t(\log^2 d \log(B+n+\sigma)) + \text{tocc})$.

Proof. We analyze the algorithm SCAN. The multiplicative factor of $\log(B+n+\sigma)$ includes the log B term for the different sizes when we use our algorithm and the $\log(n+\sigma)$ term from the Bird–Baker algorithm. There is an extra multiplicative factor of 2 to handle the symmetric case where width < height.

Regardless of whether the dictionary is size-diverse or not, steps 1–5 take time $O(t \log d)$ using the AC algorithm. Note that this is counted as part of the scanning time, not the dictionary preprocessing time.

We take $h_{\min} = \log^2 d$. This means that the use of the Bird–Baker algorithm in step 0 (over all heights) takes time $O(t \log^2 d \log(n+\sigma) + \text{tocc})$.

Since there are 2^g possible heights h and and $\lfloor t/2^g \rfloor$ power columns, the number of instances of RER and NRES queries that we need to solve in the loop at steps 6–11 is $O(t)$. Since each query takes $O(\log^2 d)$ time by Lemma 4.3, the scanning time bound follows.

We now prove the time and space bounds for the preprocessing. Recall from the last section that the number of avatars inserted is π_g, which is also the number of points inserted in any geometric structure. From Lemma 4.3, the space and time used by the Gabow–Bentley–Tarjan structures is $O(m \log^2 m)$ for m patterns. To show that this amount is $O(d)$, it suffices to show that $\pi_g = O(d/(\log^2 d))$. The key point is that we do not use the avatars of short patterns with height below $h_{\min}$. Thus $h \geq h_{\min} \geq \log^2 d$ by our choice. Therefore the following inequalities hold:

$$\pi_g \leq \sum_{h \geq h_{\min}} (h \cdot n_h) \leq \left(\sum_{h \geq h_{\min}} (h^2 \cdot n_h) \right) \Big/ h_{\min} = d/(\log^2 d).$$

From the above argument and Theorem 3.13, the time and space bounds for the preprocessing follow. ∎

We now give our second result on time bounds:

THEOREM 4.5. *For size-diverse dictionaries we can improve the bounds to*:

Preprocessing. $O(d \log(n+\sigma))$

Text scanning. $O(t(\log d \log(B+n+\sigma)) + \text{tocc})$

Proof. In the case where the patterns are size-diverse, we replace the Gabow–Bentley–Tarjan structure for range searching with a data structure of Bentley and Maurer (1980) that achieves query time $O(\log m)$ at the expense of having $S_{\text{rect}} = T_{\text{rect}} = O(m^{1+\varepsilon})$, for $\varepsilon > 0$. In the size-diverse case we use $h_{\min} = \log d$. The improved scanning time bound follows as in the general case. To prove the space and preprocessing time bounds, it suffices to show that if the patterns are size-diverse, $\pi_g^{1+\varepsilon} = O(d)$.

We choose ε small enough, so that the patterns are size-diverse with exponent $k = (1-\varepsilon)/\varepsilon$. By the definition of size-diverse, we have $n_h \leq h^{(1-\varepsilon)/\varepsilon}$ and $n_h^{\varepsilon} \leq h^{1-\varepsilon}$. Refining the above inequalities one more time, we get:

$$(\pi_g)^{(1+\varepsilon)} \leq \sum_{h \geq h_{\min}} (h \cdot n_h)^{(1+\varepsilon)} \leq \sum_{h \geq h_{\min}} (h \cdot h^{\varepsilon} \cdot h^{1-\varepsilon} \cdot n_h)$$

$$= \left(\sum_{h \geq h_{\min}} (h^2 \cdot n_h) \right) = O(d). \quad \blacksquare$$

5. DYNAMIC DICTIONARY MATCHING OF RECTANGLE PATTERNS

One of the secondary themes of this paper is that combining string matching paradigms (in particular, multiple matching and two-dimensional matching) can be difficult. Thus, it is interesting to ask: to what extent can our algorithm for multiple matching of rectangular patterns be extended to encompass other paradigms? In this section we extend our algorithm to further combine it with the *dynamic dictionary* paradigm. For one dimensional strings this paradigm is defined as follows:

- *Dynamic dictionary matching* (DDM). Given a *dictionary* of patterns $P_1, \ldots, P_n$ over an alphabet Σ, preprocess it so as to be able to search for all occurrences of all the patterns in a query text T. In addition, make the dictionary *dynamic* by supporting *insertion* and *deletion* of individual patterns from the dictionary.

A semi-dynamic version of DDM, allowing only insertions, was proposed by Meyer (1985). Amir and Farach (1991b) introduced the full DDM problem and got the first interesting bounds. The DDM problem for one-dimensional strings was studied further in (Amir, Farach, Galil, Giancarlo, and Park, 1994; Idury and Schäffer, 1992; Amir, Farach, Idury, La Poutré, and Schäffer, 1993), and we summarize the best bounds known for the case of an alphabet of arbitrary size:

Preprocessing. $O(d \log \sigma)$;

Insertion/deletion. $O(p \log d)$, where p is the length of the pattern;

Text scanning. $O((t + \text{tocc}) \log d)$.

In our use of DDM here, we do not report all matches at a given position, just the longest match, in which case the text scanning time is $O(t \log d)$. Other search/update time combinations are achievable (Idury and Schäffer 1992). The one-dimensional DDM algorithms were extended to handle two dimensional square patterns in (Amir, Farach, Idury, LaPoutré, and Schäffer, 1993).

We will describe a modification and augmentation of the approach in Sections 3 and 4 that proves:

THEOREM 5.1. *Dynamic dictionary matching of rectangular patterns in a rectangular text can be solved in the following time bounds*:

Preprocessing. $O(d \log^4 d)$;

Insertion/delation. $O(p \log^4 d)$, *where p is the area of the pattern*;

Text searching. $O(t \log^4 d \log B + \text{tocc})$.

Our dynamic algorithm retains most of the structure of the static algorithm presented in Sections 3 and 4. We summarize the features of the dynamic algorithm shared by the static algorithm along with the modifications.

First, the pattern representation and the searching algorithm for the dynamic case are very similar to the static case. We maintain Bird–Baker dictionary structures on the top of the patterns, one for each distinct height. We use them to run separately for each height below a threshold height $h_{\min}$ as in the static case. Of course, these dictionary structures are now dynamic. Similarly, the columns of patterns are inserted into a one dimensional dynamic dictionary using the algorithm of (Idury and Schäffer, 1992).

Second, we maintain the Euler tour information for the pattern trees as done in the static case. This information is needed to map pattern prefixes to rectangles using the reduction described in Section 3. However, we cannot use the rank order of the associated parentheses as the rank of a parenthesis keeps changing in a dynamic setting. We therefore assign a *different* set of numbers to parentheses such that the sorted order of these numbers is the same as the rank order of the parentheses.

To achieve this, we use the data structure of Dietz and Sleator (1987), which we call a DS-list. The Dietz–Sleator list assigns to each vertex visit in the Euler tour a number such that the order of the numbers is the same as the order in the Euler tour. After a vertex is inserted or deleted, the Euler tour numbers can be updated in $O(1)$ time. In particular, the insertion or deletion of a vertex causes only a constant number of the other list numbers to change.

Third, as one might expect, the avatars need to be made dynamic. This means that we insert avatars as points into a dynamic range searching scheme. To dynamize the range searching we use a data structure of Willard and Lueker (1985) instead of either the Bentley–Maurer structure or the Gabow–Bentley–Tarjan structure that were preferable for the static case. The Bentley–Maurer structure cannot be easily used because the validity of the assumption that the dictionary is size-diverse may change over time. The Gabow–Bentley–Tarjan structure cannot be used because it relies on fast processing of least-common-ancestor queries in a static tree, and hence, appears hard to dynamize. The following fact follows from using the data structure of Willard and Lueker:

Fact 5.2. As described above, the two-range-searching problems, RER and NRES, are equivalent to four-dimensional range-searching problems. The Willard–Lueker data structure on d points uses $O(d \log^3 d)$ space and allows us to solve RER in time $O(\log^4 d + \text{nrect})$ and NRES in time $O(\log^4 d)$. Insertions and delations take time $O(\log^4 d)$.

Finally, we need to use a special technique in the spirit of (Overmars, 1983) to keep the space $O(d)$, as d changes. These details will be explained later.

Everything else in the text-searching algorithm stays the same, except that we may alter $h_{\min}$ (the height at which we switch from Bird–Baker to our algorithm) to reduce the space requirement. Recall that in the static case we set $h_{\min} = \log^2 d$, so that the space required by the Gabow–Bentley–Tarjan range searching structure would be $O(d)$. Using the Willard–Lueker data structure, the (tempting) natural analog is to put $h_{\min} = \log^3 d$, but d is changing over time. In fact, we will make $h_{\min} = O(\log^3 d)$ at all times but the exact value of $h_{\min}$ might vary. With this choice, the use of the Bird–Baker algorithm takes time $O(t \log^4 d + \text{tocc})$. The $\log n$ term from the static time is replaced by $\log d$ in the dynamic case because we are using *dynamic* one dimensional dictionaries. We call the data structure that stores the avatars of patterns of height h the *Willard–Lueker structure* for height h.

In the remainder of the section, we describe our insertion and deletion algorithms. Assume for the rest of the section that all logarithms are base 2. At any given time define $c = 2^{\lceil \log \log d \rceil}$. That is, c is a power of 2 and d is in the range $[2^c, 2^{2c})$. Assume also that $d \geq 4$ and therefore, $c \geq 2$. We define a *threshold* point for each range, which is set to 2^{2c-1} for the range $[2^c, 2^{2c})$. Now classify possible pattern heights by:

- Lower domain, $[1, c^3]$
- Middle domain, $(c^3, 8c^3]$
- Upper domain, $(8c^3, \infty)$.

The main motivation for defining the above domains is to maintain the overall space to be linear. We build Willard–Leuker structures *completely* for the upper domain. We *do not* build them for the lower domain. We keep the structures *partially* built for the middle domain. The middle domain serves as a buffer for making smooth transitions from one range to the other.

Our idea is to keep the Willard–Lueker structures for the middle range *partially* built, making progress towards completion whenever a pattern deletion occurs and towards depletion whenever a pattern insertion occurs. By "partially built" we mean that for some patterns, all the avatars have been inserted as points in the structure, for some patterns no avatars have been inserted, and for at most one pattern some avatars are in and others are out. When the dictionary

size is above the threshold, we keep deleting the avatars of the patterns in the middle domain in anticipation of a shift from the present range to the next range. We cannot afford to use any space for Willard–Leuker structures for the avatars in the current middle domain once we shift as they will belong to the lower domain in the next range. Similarly, when the dictionary size is below the threshold, we keep inserting the avatars of the patterns in the middle domain.

We maintain the following invariants in our insertion/deletion algorithms:

1. For every h, all patterns of height h are kept in a Bird–Baker dictionary for height h, which is used in searching for heights that are smaller than the current value of $h_{\min}$.

2. For every h, all patterns of height h in the upper domain are kept in a Willard–Lueker structure for height h.

3. No patterns of heights in the lower domain are kept in a Willard–Lueker structure.

4. $h_{\min}$ is the largest height in the middle domain; i.e., $8c^3 = \theta(\log^3 d)$

As a result of invariants 3 and 4, we can show similarly to Theorem 4.4, that the amount of space needed for the Willard–Lueker structures is $O(d)$ (because they only store information for patterns whose height is $\Omega(\log^3 d)$). A key point is that, although we cannot afford the space to keep the Willard–Lueker structures for heights in the lower domain, we *can* afford the space for the middle domain.

We also assume, without loss of generality, that no single pattern is so big that its insertion or deletion causes c to change by more than a factor of 2. If a pattern is big enough so that its insertion or deletion causes c to change by more than a factor of 2, then its area A must be $\Omega(d)$. Therefore, we can rebuild the entire dictionary from scratch, having known the final value of d in advance. The cost of doing so can be paid by the pattern being inserted or deleted.

For reasons explained later, we define L as a *queue* of avatars of patterns in the middle domain which are not inserted as points into any Willard–Lueker structure. In the following algorithm description we do not distinguish between a pattern avatar (P_f, P_b), the rectangle that the avatar is mapped to, and the four-dimensional point that describes the extreme coordinates of that rectangle.

We now give the pseudocode for insertion and deletion of a pattern:

ALGORITHM 3. Pseudocode for inserting a pattern.

```
INSERT(P)
 0  Let h and A be the height and area of P,
      respectively
 1  d ← d + A
 2  If d ≥ 2^(2c) then
 3       c ← 2c
 4       Set h_min ← 8c^3, and change the lower,
           middle, and upper domains
           accordingly
 5  Insert P into D_h
 6  For each avatar (P_f, P_b) of P
 7       Insert P_f^u, P_f^l, P_b^u, P_b^l into H^{f,u}, H^{f,l},
           H^{b,u}, H^{b,l}
 8       If P is the upper domain add (P_f, P_b)
           to Willard-Lueker structure
 9       If P is in middle domain add (P_f, P_b)
           to L.
10  If d ≥ 2^(2c-1) (threshold) then
11       Delete up to 2A avatars of patterns
           in middle domain from
12         Willard-Lueker structures and
             add them to L
```

ALGORITHM 4. Pseudocode for deleting a pattern.

```
DELETE(P)
 0  Let h and A be the height and area of P,
      respecively
 1  d ← d − A
 2  If d < 2^c then
 3       c ← c/2
 4       Set h_min ← 8c^3, and change the lower,
           middle, and upper domains
           accordingly
 5  Delete P from D_h
 6  For each avatar (P_f, P_b) of P
 7       Delete P_f^u, P_f^l, P_b^u, P_b^l from H^{f,u}, H^{f,l},
           H^{b,u}, H^{b,l}
 8       Delete the avatars of P (if any)
           from Willard-Lueker structure
           and L
 9  If d < 2^(2c-1) (threshold) then
10       Insert up to 2A avatars of patterns
           in L into
11         Willard-Leuker structures and
             delete them from L
```

The above invariants imply that the search and update bounds in Theorem 5.1 can be achieved using the insertion and deletion algorithms above, as long as the domain boundaries stay the same. If the domain boundaries change, this causes $h_{\min}$ to change. We consider separately the cases where the shift in boundaries is caused by an increase or decrease in d.

An insertion of a pattern can cause the dictionary size to increase, so that the new lower domain becomes $[1, 8c^3]$, the new middle domain becomes $(8c^3, 64c^3]$, and the new upper domain becomes $(64c^3, \infty)$. Invariants 1 and 2 are preserved by the insertion algorithm. To preserve invariant 4, we change $h_{\min}$ to $64c^3$. To preserve invariant 3, we must have deleted all Willard–Lueker structures for heights

$(c^3, 8c^3]$, which used to be the middle domain and is now part of the lower domain. Note that we keep the Willard–Lueker structures for the new middle domain; this is important because we will need them immediately if a pattern deletion causes the domains to shift back to previous values.

A deletion of a pattern can cause the dictionary size to decrease, so that the new lower domain becomes $[1, c^3/8]$, the new middle domain becomes $(c^3/8, c^3]$, and the new upper domain becomes (c^3, ∞). Again, invariant 1 is preserved by the deletion algorithm, invariant 3 is preserved because the lower domain decreases in size, and we preserve invariant 4 by adjusting $h_{\min}$ to c^3. To preserve invariant 2, we must have Willard–Lueker structures ready for heights $(c^3, 8c^3]$, which used to be the middle domain and is now part of the upper domain. Note that we do not have any Willard–Lueker structures for the new middle domain; this is important because if a pattern insertion causes the domains to shift back to previous values, this domain will become part of the lower domain.

Our strategy for building the middle domain Willard–Lueker structures is as follows. We do extra work in the insertion (deletion) algorithm, so that whenever a pattern is inserted (deleted) and the dictionary size goes above (below) the threshold value, we also make progress towards deleting (inserting) from (into) the Willard–Lueker structures the avatars of middle domain patterns in L. To delete (insert) a pattern of area A, we are allowing time $O(A \log^4 d)$. Recall that each pattern of area A has at most A avatars. To delete (insert) these avatars from (into) the Willard–Lueker structures also takes time $O(A \log^4 d)$. Therefore, when a pattern of area A is inserted (deleted) and the dictionary size goes above (below) the threshold, we delete (insert) up to $2A$ avatars of patterns in L with no asymptotic time penalty. The following lemma shows that invariant 2 can be preserved under deletions.

LEMMA 5.3. *Suppose we use the above strategy of rebuilding during insertions and delations. When a deletion causes the middle domain to drop from* $(c^3, 8c^3]$ *to* $(c^3/8, c^3]$, *there will be no more avatars of patterns in L with heights in the old middle domain after that deletion is complete. That is, the Willard–Lueker structures for heights* $(c^3, 8c^3]$ *will be complete when* $(c^3, 8c^3]$ *is shifted to the upper domain.*

Proof. Suppose that the dictionary size never exceeded the threshold for the old range. Whenever we enter a range from the range below it (as a result of some insertion), the Willard–Leuker structures for its middle domain will be fully built because its middle domain was part of the upper domain for the range below it. We could not have deleted any avatars from the Willard-Leuker structures during the insertions of patterns because the dictionary size never exceeded the threshold. Therefore, its L list will continue to be empty as desired.

Suppose that the dictionary size exceeded the threshold for the old range. Consider the last time α that the dictionary size is above the threshold for the old range, and let $d \geq 2^{2c-1}$ be the size of the dictionary at time α. Let d' be the size of the dictionary just after the middle domain drops to $(c^3/8, c^3]$ at time β. By the definitions of c and threshold, and by construction of the domains, $d \geq 2d'$. Let $S = x + y$ be the total area of patterns that are both inserted (with total area x) and deleted (with total area y) during the period $[\alpha, \beta]$. Thus the total area of the deleted patterns during this period is $y = d + x - d'$, which is at least $x + d'$. During those deletions we could insert into the Willard–Lueker structures all the avatars of patterns whose areas total $2(x + d')$. During this period we could have deleted at most $2x$ avatars from the Willard–Leuker structures because of the insertions of patterns. Therefore, the amount we need to insert is at most $d' + 2x \leq 2(x + d')$. ∎

Similarly, we can show that invariant 3 can be preserved under insertions. This completes the proof of Theorem 5.1.

ACKNOWLEDGMENTS

We thank Amihood Amir for suggesting this problem to us, Martin Farach and Raffaele Giancarlo for helpful discussions, and Gaston Gonnet for sending us the transparencies of (Gonnet, 1992).

Received December 3, 1992; final manuscript received May 28, 1993

REFERENCES

Aho, A. V., and Corasick, M. J. (1975), Efficient string matching: An aid to bibliographic search, *Com ACM* **18**, 333–340.

Amir, A., Benson, G., and Farach, M. (1992), Alphabet independent two-dimensional matching, *in* "Proceedings, 24th ACM Symposium on Theory of Computing," pp. 59–68; full version, *SIAM J. Comp.* **23** (1994), 313–323.

Amir, A., and Farach, M. (1991a), Efficient 2-dimensional approximate matching of non-rectangular figures, *in* "Proceedings, 2nd ACM–SIAM Symposium on Discrete Algorithms," pp. 212–223.

Amir, A., and Farach, M. (1991b), Adaptive dictionary matching, *in* "Proceedings, 32nd IEEE Symposium on Foundations of Computer Science," pp. 760–766.

Amir, A., and Farach, M. (1992), Two-dimensional dictionary matching, *Inform. Process. Lett.* **44**, 233–239.

Amir, A., Farach, M., Galil, Z., Giancarlo, R., and Park, K. (1994), Dynamic dictionary matching, *J. Comput. Systems Sci.* **49**, 208–222.

Amir, A., Farach, M., Idury, R. M., La Poutré, J. A., and Schäffer, A. A. (1993), Improved dynamic dictionary matching, *in* "Proceedings 4th Annual ACM-SIAM Symposium on Discrete Algorithms," pp. 392–401; full version, *Inform. and Comp.*, to appear.

Amir, A., Landau, G. M., and Vishkin, U. (1992), Efficient pattern matching with scaling, *J. Algorithms* **13**, 2–32.

Baeza-Yates, R., and Régnier, M. (1990), Fast algorithms for two dimensional and multiple pattern matching, *in* "Proceedings, 2nd Scandinavian Workshop on Algorithm Theory," Lecture Notes in Computer Science, Vol. 447, pp. 332–347, Springer-Verlag, New York/Berlin; full version, *Inform. Process. Lett.* **45** (1993), 51–57.

Baker, T. P. (1978), A technique for extending rapid exact-match string matching to arrays of more than one dimension, *SIAM J. Comput.* **7**, 533–554.

Bentley, J. L., and Maurer, H. A. (1980), Efficient worst-case data structures for range searching, *Acta inform.* **13**, 155–168.

Bird, R. S. (1977), Two dimensional pattern matching, *Inform. Process. Lett.* **6**, 168–170.

Boyer, R. S., and Moore, J.S. (1977), A fast string searching algorithm, *Comm. ACM* **20**, 762–772.

Dietz, P., and Sleator, D. D. (1987), Two algorithms for maintaining order in a list, *in* "Proceedings, 19th ACM Symposium on Theory of Computing," pp. 365–372; *J. Comput. System Sci.*, to appear.

Edelsbrunner, H., and Overmars, M. H. (1982), On the equivalence of some rectangle problems, *Inform. Process. Lett.* **14**, 124–127.

Gabow, H. N., Bentley, J. L., and Tarjan, R. E. (1984), Scaling and related techniques for geometry problems, *in* "Proceedings, 16th ACM Symposium on Theory of Computing," pp. 135–143.

Galil, Z., and Park, K. (1992), Truly alphabet-independent two-dimensional pattern matching, *in* "Proceedings, 33rd IEEE Symposium on Foundations of Computer Science," pp. 247–256.

Giancarlo, R. (1992), personal communication.

Giancarlo, R. (1993), The suffix tree of a square matrix with applications, *in* "Proceedings, 4th ACM-SIAM Symposium on Discrete Algorithms," pp. 402–411.

Gonnet, G. (1992), Efficient two-dimensional searching, *in* "Proceedings, 3rd Scandinavian Workshop on Algorithm Theory," Lecture Notes in Computer Science, Vol. 621, p. 317; abstract of an invited talk.

Gonnet G. H., and Baeza-Yates, R. (1991), "Handbook of Algorithms and Data Structures," 2nd ed., Addison-Wesley, Reading, MA.

Idury, R. M., and Schäffer, A. A. (1992), Dynamic dictionary matching with failure functions, *in* "Proceedings, 3rd Symposium on Combinatorial Pattern Matching," Lecture Notes in Computer Science, Vol. 644, pp. 276–287, Springer-Verlag, New York/Berlin; full version, *Theoret. Comput. Sci.* **131** (1994), 295–310.

Knuth, D. E., Morris, J. H., and Pratt, V. R. (1977), Fast pattern matching in strings, *SIAM J. Comput.* **6**, 323–350.

Lee, D. T., and Wong, C. K. (1981), Finding intersections of rectangles by range search, *J. Algorithms* **2**, 337–347.

Manber, U., and Baeza-Yates, R. (1991), An algorithm for string matching with a sequence of don't cares, *Inform. Process. Lett.* **37**, 133–136.

Meyer, B. (1985), Incremental string matching, *Inform. Process. Lett.* **21**, 219–227.

Overmars, M. H. (1983), "The Design of Dynamic Data Structures," Lecture Notes in Computer Science, Vol. 156, Springer-Verlag, New York/Berlin.

Willard, D. E., and Lueker, G. S. (1985). Adding range restriction capabilities to dynamic data structures, *J. Assoc. Comput. Mach.* **32**, 597–617.

Zhu, R. F., and Takaoka, T. (1989), A technique for two-dimensional pattern matching, *Comm. ACM* **32**, 1110–1120.