Multiple Matching of Rectangular Patterns (Extended Abstract)

Ramana M. Idury [*]
Rice University

Alejandro A. Schäffer [†]
Rice University

Abstract

We describe the first efficient algorithm for simultaneously matching multiple rectangular patterns of varying sizes and aspect ratios in a rectangular text. Efficient means significantly better asymptotically than known algorithms that handle one height, width, or aspect ratio at a time. 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., [23, 13]) that preprocess the pattern in time $O(p)$, where p is the pattern length, and search a text of length t in time $O(t)$.

[*]Department of Computer Science, Rice University, Houston, TX 77251-1892. idury@cs.rice.edu; Partially supported by a grant from the W. M. Keck Foundation.

[†]Department of Computer Science, Rice University, Houston, TX 77251-1892. schaffer@cs.rice.edu; Partially supported by NSF grant CCR-9010534.

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 T. P. Baker in 1978 [10].

The natural extension of FPM to multiple one dimensional patterns is defined as:

- **Multiple Pattern Matching (MPM):** Given a *fixed* set of patterns $P_1, \ldots, P_n$ 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. The set of patterns is also called a *dictionary.*

Aho and Corasick, or AC for short, solved MPM [1]. The salient feature of their algorithm is the extension of the Knuth-Morris-Pratt [23] (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 $O(t \log \sigma + tocc)$, where d is the total size of all patterns, σ is the number of characters that occur in some pattern, 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 [12] and independently by Baker [10]. 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 $O(t \log \sigma)$, where p and t are the *area* of the pattern and text. Other algorithms that seem to perform better on random patterns have been described in [29, 9]. Recent papers give new algorithms whose running times are linear without depending on the alphabet size [2, 17].

Since he used an algorithm for MPM to solve the TFPM problem, it is not surprising that Baker [10] observed that it is natural to combine the two paradigms. Define:

- **Two Dimensional Multiple Pattern Matching (TMPM):** Given a *fixed* set of rectangular patterns $P_1, \ldots, P_n$ over an alphabet Σ, preprocess it so as to be able to search for all occurrences of all the patterns in a rectangular *query* text T. The set of patterns is also called a *dictionary*.

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 [21]. Much of that section focuses on a "dual" problem where there are multiple texts to be preprocessed and one query pattern; Giancarlo [18] has some recent improvements to those results.

If all the patterns in an instance of TMPM are of fixed height, then Baker [10] noted that his algorithm can be adapted to solve TMPM by replacing the use of the KMP algorithm with a use of the AC algorithm. The search time for one height or width becomes $O(t \log(n + \sigma) + tocc)$. 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 [10].

Amir and Farach [3] found an efficient algorithm for TMPM in the special case where all the patterns are square; a similar algorithm was given by Giancarlo [19]. 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 [3, 19] to solve TMPM for rectangular patterns. Rectangular patterns of different heights, widths, and aspect ratios *cannot be aligned at a corner.* There is no natural way to define the "biggest suffix" of the text that matches a "prefix" of some 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 [20], focusing on text columns that are (numbered) 0 mod q for some appropriate q (see [8, 9]), and the smaller matching with tree partial order paradigm, which was defined and studied in [5]. For other, different uses of computational geometry in string matching see [25, 8].

The time and space of our TMPM algorithm depend 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.

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: $O(t \log d \log(B + n + \sigma) + tocc)$

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: $O(t \log^2 d \log(B + n + \sigma) + tocc)$

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 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 sketch how our algorithm can be extended to a dynamic setting where the set of patterns can change over time.

2 Definitions and Algorithm 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 $\leq$ width; the opposite case can be handled symmetrically. Second, we partition the set of patterns into $\lfloor \log B \rfloor$ sets $\mathcal{P}_0, \mathcal{P}_1, \ldots$ such that:

Definition 2.1 *The patterns in set $\mathcal{P}_g$ have a height h in the range $2^g < h \leq 2^{g+1}$.*

For each g, we search the text for just the patterns in $\mathcal{P}_g$ 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 does not change the asymptotic preprocessing time or space.

From now on we focus on one specific $\mathcal{P}_g$ and assume 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 $h = 3$.*

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

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:

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

And the lower half patterns are:

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

Definition 2.4 *Let C be the set of text 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 text columns is $C = \{ab, ac, ba, bc, ca, cb\}$, where each column is written as a string where left-to-right in the string corresponds to top-to-bottom in the column. 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 $aca|bab|cab|aba$, its upper half pattern as $ac|ba|ca|ab$, and its lower half pattern as $ca|ab|ab|ba$.

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, $bab|cab|aba$ is a prefix of the second pattern and a suffix of the first pattern, whereas $bcb|aba|cab|bab$ 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, $H^{f,u} = \{ac|ba|ca|ab, ac|ba|ca, ba|ca|ab|bc, ba|ca|ab\}$, $H^{f,l} = \{ca|ab|ab|ba, ca|ab|ab, ab|ab|ba|cb, ab|ab|ba\}$, $H^{b,u} = \{ab, ab|ca, bc, bc|ab\}$, and $H^{b,l} = \{ba, ba|ab, cb, cb|ba\}$.

The reason for storing the prefixes and reverse prefixes is that our final scanning pass will look only in text columns numbered 0 mod 2^g. Following [8], we call these *power columns.* If there is a match spanning columns $[c_1 \ldots c_2]$ of the text, we will find it in the rightmost power column $c \leq c_2$. 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 character. We say the 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 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, $2^g < h \leq 2^{g+1}$, we maintain two 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 extend the definitions of avatar and projections to half patterns. 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, we find for text location $T[i,j]$ in power column j, the widest elements of $H^{f,u}$ and $H^{f,l}$ that match 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.

[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 a continuing basic entity (Webster's Dictionary).

In the text scanning phase, we ask for each power column location $T[i,j]$ 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? We ask 2^g queries at each power column position, but there are at most $\lfloor t/2^g \rfloor$ power columns, so the total number of queries is $O(t)$.

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 build (preprocess) and use (search) 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 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 Reporting (RER) [15]:** Given a set V of rectangles in the plane and another 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 [24, 15, 16]. 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 $u_1 \leq_s u_2$, if u_1 is a suffix of u_2. We also extend the definition of dominance to a pair of strings as follows: $(u_1, v_1) \leq_s (u_2, v_2)$ if $u_1 \leq_s u_2$ and $v_1 \leq_s v_2$.*

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

Definition 3.2 *Let D be a one dimensional dictionary automaton constructed using the AC algorithm [1]. 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.*

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

We can conclude from Definition 3.2 and Fact 3.3 that:

Fact 3.4 *[1, 5] If u_1 and u_2 are prefixes of some patterns in a dictionary D, then $u_1 \leq_s u_2$ if and only if u_1 is an ancestor of u_2 in $failtree(D)$.*

We use Fact 3.4 to transform each prefix of a pattern in a one dimensional dictionary D with m prefixes to a line interval on the real line as follows. Perform an Euler tour of $failtree(D)$ and use:

Fact 3.5 *[22, 7] 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.*

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 $0 \ldots 2m-1$ on the real line in the same order, then each prefix of a pattern corresponds to an interval on the real line. 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.

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.*

The following lemma follows from the above definition:

Lemma 3.7 *Suppose $u, v \in D$. Then $u \leq_s v$ 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 $(u_1, u_2) \leq_s (v_1, v_2)$ 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 $failtree(D_2)$. 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.*

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 $(u_1, u_2) \leq_s (v_1, v_2)$ 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 $V_h^f = \{rect(P_f^u, P_f^l) \mid P_f \in F_h\}$ 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 $(P_f^u, P_f^l) \leq_s (FU[i,j], FL[i+h-2^g, j])$. 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. Since F_h itself is a one dimensional dictionary, 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 $(P_b^u, P_b^l) \leq_s (BU[i,j], BL[i+h-2^g, j])$. ▮

Theorem 3.11 *We can similarly reduce* **Step 3** *to RER.*

We now express the time and space required to preprocess patterns of height h in $\mathcal{P}_g$ 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 the preprocessing time and space bounds for either RER or NRES with m rectangles are $O(m \cdot T_{rect}(m))$ and $O(m \cdot S_{rect}(m))$ respectively. Given C, we can build $F_h, B_h, V_h^f, V_h^b, V_h$ in time $O(d_h \log n_h + h \cdot n_h \cdot T_{rect}(h \cdot n_h))$, and space $O(d_h + h \cdot n_h \cdot S_{rect}(h \cdot n_h))$.*

Proof: Consider a pattern P of height h in D_h. Let $(P_f^1, P_b^1), \ldots, (P_f^{2^g}, P_b^{2^g})$ 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 $1 < i \leq 2^g$. Similarly P_b^i is a prefix of P_b^{i+1} for $1 \leq i < 2^g$. 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 $O(d_h \log n_h)$ 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 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 state the time and space bounds for the preprocessing of patterns. Pseudocode for the preprocessing of patterns is given in the Appendix.

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

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 $O(\sum_h (h \cdot n_h \cdot T_{rect}(h \cdot n_h)))$. From this we conclude that steps 1–13 take time $O(p_g \log(n + \sigma) + \pi_g \cdot T_{rect}(\pi_g))$. Similarly we can prove that steps 1–13 take space $O(p_g + \pi_g \cdot S_{rect}(\pi_g))$. 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 in the Appendix. 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. 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 time 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 $\leq h_{\min}$ we use the Bird-Baker algorithm for each separate height. This takes time $O(h_{\min} \cdot t \log(n + \sigma) + tocc)$. 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 4 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 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 $FU[i,j] = ac|ba|ca|ab$, $FL[i+1,j] = ca|ab|ab|ba$, $BU[i,j] = bc|ab$, and $BL[i,j] = cb|ba$. 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 $w_f[i,j] = aca|bab|cab|aba$, and $w_b[i,j] = bcb|aba$. Finally at steps 10 and 11, we report the actual matches as follows. The first pattern $aca|bab|cab|aba$ is matched at location $T[i,j]$ since it has an avatar $(aca|bab|cab|aba, aba)$ 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 $bab|cab|aba|bcb$ is matched at location $T[i,j+1]$ since it has an avatar $(bab|cab|aba, bcb|aba)$ 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 *[24, 15] RER and NRES can be reduced to 4-dimensional range searching problems.*

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: $x_{\min}(R) \leq x_{\min}(Q)$, $x_{\max}(R) \geq x_{\max}(Q)$, $y_{\min}(R) \leq y_{\min}(Q)$, and $y_{\max}(R) \geq y_{\max}(Q)$. 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 inequalities 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 [15]. Those readers familiar with range searching will notice that in the above reduction the four constraint intervals for Q are unbounded on one side. This enables us to use a result of Gabow-Bentley-Tarjan [16] who already noted its applicability to the counting variant of RER.

Lemma 4.3 *[16] There is a data structure that enables us to store m points in four dimensions with $S_{rect} = T_{rect} = O(\log^2 m)$ and answer RER queries in time $O(\log^2 m + nrect)$, where nrect is the number of rectangles reported. For the NRES problem, the additive term nrect is dropped.*

We now give two results 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: $O(t(\log^2 d \log(B + n + \sigma)) + tocc)$

Proof: We analyze the algorithm SCAN whose pseudocode description is given in the Appendix. One multiplicative factor of $\log B$ in the scanning time, when we do not use the Bird-Baker algorithm, arises because we call SCAN once for each height range. Regardless of whether the dictionary is size-diverse or not, steps 1–5 take time $O(t \log d)$ using the AC algorithm.

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 $O(t \log^2 d \log(n + \sigma) + tocc)$. 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 Theorem 3.13 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: $\pi_g \leq \sum_{h \geq h_{\min}} (h \cdot n_h) \leq (\sum_{h \geq h_{\min}} (h^2 \cdot n_h))/h_{\min} = d/(\log^2 d)$.
From the above argument and Theorem 3.13, the time and space bounds follow. ▮

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

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

Text Scanning: $O(t(\log d \log(B + n + \sigma)) + tocc)$

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 [11] that achieves query time $O(\log m)$ at the expense of having $S_{rect} = T_{rect} = O(m^{1+\epsilon})$, for $\epsilon > 0$. 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, $\pi_g^{1+\epsilon} = O(d)$.

We choose ϵ small enough, so that the patterns are size diverse with exponent $k = (1 - \epsilon)/\epsilon$. By the definition of size-diverse, we have $n_h \leq h^{(1-\epsilon)/\epsilon}$ and $n_h^\epsilon \leq h^{1-\epsilon}$. Refining the above inequalities one more time, we get: $(\pi_g)^{(1+\epsilon)} \leq \sum_{h \geq h_{min}} (h \cdot n_h)^{(1+\epsilon)} \leq \sum_{h \geq h_{min}} (h \cdot h^\epsilon \cdot h^{1-\epsilon} \cdot n_h) = (\sum_{h \geq h_{min}} (h^2 \cdot n_h)) = O(d)$. ∎

5 Dynamic Dictionary Matching of Rectangular 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 briefly summarize how we can extend our algorithm to further combine it with the *dynamic dictionary* paradigm. The details are given in our full paper. For one dimensional strings this paradigm is defined as follows:

- **Dynamic Dictionary Matching(DDM):** Maintain a dictionary of patterns under the operations *insert* a pattern, *delete* a pattern, and *search* for all occurrences of patterns currently in the dictionary in a query text.

A semi-dynamic version of DDM, allowing only insertions, was proposed by Meyer [26]. Amir and Farach [4] introduced the full DDM problem and got the first interesting bounds. The DDM problem for one dimensional strings was studied further in [6, 22, 7]. The one dimensional DDM algorithms were extended to handle two dimensional square patterns in [7].

Our main result on dynamic dictionary matching for rectangular patterns is:

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

Text Searching: $O(t \log^4 d \log B + tocc)$.

At a high level the pattern representation and the searching algorithm for the dynamic case are similar to the approach in Sections 3 and 4. We briefly summarize the main new ideas. First, as one might expect, all our one dimensional dictionaries of columns and avatars are made dynamic. Second there are two other principal changes in data structures. Third, we need to use a dynamization technique in the spirit of [27] to keep the space $O(d)$, as d changes. The two other changes in data structures are:

1. To dynamically maintain the Euler tour information for the pattern trees, we use a list data structure of Dietz and Sleator [14], which we call a DS-list.

2. To dynamize our range searching we use a data structure of Willard and Lueker [28] 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.

Acknowledgements 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 [20].

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18:333-340, 1975.

[2] A. Amir, G. Benson, and M. Farach. Alphabet independent two-dimensional matching. *Proc. of the 24th Ann. ACM Symp. on Theory of Computing*, pages 59-68, 1992.

[3] A. Amir and M. Farach. Two dimensional dictionary matching. *Information Processing Letters.* To appear.

[4] A. Amir and M. Farach. Adaptive dictionary matching. *Proc. of the 32nd IEEE Annual Symp. on Foundation of Computer Science*, pages 760-766, 1991.

[5] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. *Proc. of the Second Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 212-223, 1991.

[6] A. Amir, M. Farach, R. Giancarlo, Z. Galil, and K. Park. Dynamic dictionary matching. Manuscript, 1991.

[7] A. Amir, M. Farach, R. M. Idury, J. A. La Poutré, and A. A. Schäffer. Improved dynamic dictionary matching. *Proc. of the Fourth Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 392-401, 1993. Full version available as DIMACS Tech. Report 92-33.

[8] A. Amir, G. M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *J. Algorithms*, 13:2–32, 1992.

[9] R. Baeza-Yates and M. Régnier. Fast algorithms for two dimensional and multiple pattern matching. *Proc. of the 2nd Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science 447*, pages 332–347, 1990.

[10] T. P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comp.*, 7:533–541, 1978.

[11] J. L. Bentley and H. A. Maurer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13:155–168, 1980.

[12] R. S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6:168–170, 1977.

[13] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[14] P. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. of the 19th Ann. ACM Symp. on Theory of Computing*, pages 365–372, 1987. To appear in *J. Comp. Syst. Sci.*

[15] H. Edelsbrunner and M. H. Overmars. On the equivalence of some rectangle problems. *Information Processing Letters*, 14:124–127, 1982.

[16] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. *Proc. of the 16th Ann. ACM Symp. on Theory of Computing*, pages 135–143, 1984.

[17] Z. Galil and K. Park. Truly alphabet-independent two-dimensional pattern matching. *Proc. of the 33rd IEEE Annual Symp. on Foundation of Computer Science*, pages 247–256, 1992.

[18] R. Giancarlo. Personal Communication, 1992.

[19] R. Giancarlo. The suffix tree of a square matrix with applications. *Proc. of the Fourth Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 402–411, 1993.

[20] G. Gonnet. Efficient two-dimensional searching. *Proc. of the 3rd Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science 621*, page 317, 1992. Abstract of an invited talk.

[21] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 2nd edition, 1991.

[22] R. M. Idury and A. A. Schäffer. Dynamic dictionary matching with failure functions. In *Proc. of the Third Symp. on Combinatorial Pattern Matching*, 1992. Full version submitted to a journal.

[23] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

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

[25] U. Manber and R. Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Inf. Proc. Let.*, 37:133–136, 1991.

[26] B. Meyer. Incremental string matching. *Information Processing Letters*, 21, 1985.

[27] M. H. Overmars. *The Design of Dynamic Data Structures.* Springer-Verlag Lecture Notes in Computer Science 156, 1983.

[28] D. E. Willard and G. S. Lueker. Adding range restriction capabilities to dynamic data structures. *Journal of the ACM*, 32:597–617, 1985.

[29] R. F. Zhu and T. Takaoka. A technique for two-dimensional pattern matching. *Communications of the ACM*, 32:1110–1120, 1989.

A Pseudocode for Pattern Preprocessing and Text Scanning

Algorithm 1 *Code for preprocessing patterns.*

```
PREPROCESS($\mathcal{P}_g$)
NOTE $\forall p \in \mathcal{P}_g$, $height(p) \leq width(p)$ and $2^g < height(p) \leq 2^{g+1}$
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 $h$, $2^g < h \leq 2^{g+1}$
4       Build the dictionary of columns of height $h$
5       Let $D_h \subseteq \mathcal{P}_g$ 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 $P \in D_h$
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 the data structure needed to solve RER on $V_h$
```

Algorithm 2 *Code for scanning a text with $\mathcal{P}_g$.*

```
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 $h$, $2^g < h \leq 2^{g+1}$
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 $M \in V_h$ be the set of rectangles enclosing $rect(w_f[i,j], w_b[i,j])$
11          Report the corresponding patterns of $M$
```