Efficient 2-dimensional Approximate Matching of Half-rectangular Figures*

Amihood Amir† Martin Farach‡

Georgia Tech DIMACS

July 23, 1993

Abstract

Efficient algorithms exist for the approximate two dimensional matching problem for rectangles. This is the problem of finding all occurrences of an $m \times m$ pattern in an $n \times n$ text with no more than k mismatch, insertion, and deletion errors. In computer vision it is important to generalize this problem to non-rectangular figures. We make progress towards this goal by defining *half-rectangular* figures of height m and area a. The approximate two dimensional matching problem for half-rectangular patterns can be solved using a dynamic programming approach in time $O(an^2)$. We show an $O(kn^2\sqrt{m \log m}\sqrt{k \log k} + k^2n^2)$ algorithm which combines convolutions with dynamic programming. Note that our algorithm is superior to previous known solutions for $k \leq m^{1/3}$.

At the heart of the algorithm are the *Smaller Matching Problem* and the *k-Aligned Ones with Location Problem.* These are interesting problems in their own right. Efficient algorithms to solve both these problems are presented.

---

*The results presented in this paper appeared in the proceedings of the Second Symposium on Discrete Algorithms [AF91].

†College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280; (404) 853-0083; amir@cc.gatech.edu; Partially supported by NSF grant IRI-9013055.

‡DIMACS, Box 1179, Rutgers University, Piscataway, NJ 08855; (908) 932-5928; farach@dimacs.rutgers.edu

1 Introduction

String matching, and its many generalizations, is a well studied area of computer science with applications in many fields. In its basic version - *exact string matching* - the problem is the following.

Input: Text string $T = T_1 T_2 \cdots T_n$ and pattern string $P = P_1 P_2 \cdots P_m$, where $T_i, P_i \in \Sigma$.

Output: All locations i in T where $T_i T_{i+1} \cdots T_{i+m-1} = P_1 P_2 \cdots P_m$.

Several techniques have been developed for efficient solutions of string matching problems.

In [FP74], Fischer and Paterson introduced *convolutions* (general linear products) as a means for solving string matching problems. The exact string matching problem was solved, for alphabet Σ in $O(n \log m \log |\Sigma|)$ word operations, where the words are of size $O(\log m)$. One of the strengths of the convolutions method is that it allows, with no time degradation, solving the string matching problem even when a *"don't care"* character (that matches every character) is added to the alphabet. Fischer and Paterson remarked that convolutions are likely to be useful in various other pattern matching problems. Indeed, it is possible, using convolutions, to solve the *string matching with mismatches problem* in time $O(|\Sigma| n \log m)$ [AL91]. In the string matching with mismatches problem we are interested in the number of mismatches that results from aligning the pattern against every position in the text. An exact match will yield zero mismatches. Abrahamson [Abr87] and, independently, Kosaraju [Kos87] used convolutions in a divide and conquer approach that solved the string matching with mismatches problem for infinite alphabets in time $O(n\sqrt{m} \log m)$.

The automaton method is a very efficient technique for exact string matching. It was used by [KMP77, BM77] for an $O(n)$ time exact string matching solution. In [AC75] it was extended to finding text locations where any of a given *set* of patterns match the text in time $O(n +$ total length of the patterns $+$ *output size*$)$. Here *output size* is the number of times a pair $(location, pattern)$ is output. As remarked by Aho and Corasick, this may be more than n, in case several patterns match the same text location. Other techniques for exact string matching are subword trees [Wei73], periodicity analysis [GS83], and sampling using a pattern "signature" [KR87, Vis89]. One drawback of all of these techniques is that they depend heavily on the *exactness* of the matching and do not seem immediately suitable for approximate string matching.

In the string matching with k-differences problem, 3 types of differences are distinguished:

(a) A pattern character is aligned corresponding to a different character in the text (*mismatch*).

(b) A text character is deleted (*deletion*).

(c) A pattern character is deleted (*insertion*).

The problem is to find all occurrences of the pattern in the text with at most k differences of type (a), (b) or (c).

Many approximate string matching problems can be solved in time $O(nm)$ by dynamic programming [Ukk85]. In [LV86], dynamic programming and suffix trees were combined to solve the *string matching with k-differences problem* in time $O(nk)$.

While the different string matching techniques solve various generalizations of string matching problems, there is still an interesting challenge. It seems hard to synthesize different techniques so that their strengths can be harnessed to efficiently solve composite problems. For example, convolutions can efficiently solve exact matching with don't cares, and dynamic programming with suffix trees efficiently solve matching with k-differences. However, there is no known efficient algorithm for matching with don't cares *and* mismatches, insertions and deletions. The problem this paper deals with is of a similar flavor, but in two-dimensional matching.

Consider problems of searching aerial photographs. The first phase in an abstraction into pure problems will be to classify the difficulties that arise into three major subclasses:

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

In [KS87] and [AL91], algorithms for some pure local errors problem were given. In [ALV90] a clean (discrete) version of scaling was solved in sublinear time. In this paper we solve a limited two-dimensional equivalent of the matching with don't cares, mismatches, insertions and deletions problem. The problem we solve will be precisely defined below.

The *two dimensional exact matching problem* is defined as follows:

Input: Text matrix $T[1,...,n;1,...,n]$ and pattern matrix $P[1,...,m;1,...m]$, where $T[i,j], P[i,j] \in \Sigma$.

Output: All locations $[i,j]$ in T where $T[i+k-1, j+l-1] = P[k,l], \quad k,l = 1,...,m$.

In [Bir77, Bak78] this problem was solved in time $O(n^2 \log |\Sigma|)$. Recently, some new solutions were developed [ABF92, BYR90, MR92, PG92].

The two-dimensional version of the k-differences problem was defined in [KS87] as follows:

Input: Text matrix $T[1, ..., n; 1, ..., n]$ and pattern matrix $P[1, ..., m; 1, ..., m]$.

Output: All locations $[i, j]$ in T where $P[l, 1] \cdots P[l, m]$ appears as a prefix of $T[i + l - 1, j] \cdots T[i + l - 1, n]$ with i_l differences, for $l = 1, ..., m$ and $\sum_{l=1}^{m} i_l \leq k$.

In words, find all text locations such that all rows of the pattern match the corresponding consecutive text rows, where the total number of differences allowed in all row matches is at most k. This is a reasonable definition where images are transmitted row by row and transmission errors may occur.

The problem was solved by [KS87] in time $O(kmn^2)$ and improved by [AL91] to $O(k^2n^2)$.

In both above versions of the two-dimensional matching, the pattern and the text are rectangular matrices. In reality, it is sometimes necessary to match *non-rectangular* shapes. In [AL91] it was shown that multi-dimensional matching can be reduced to string matching by appropriate padding with don't care characters. Such a padding allows solving the exact two-dimensional matching problem, or the k-mismatches problem (only mismatch errors are allowed) for *any shape* in time $O(|\Sigma|n^2 \log m)$. (Or for infinite alphabets in time $O(n^2\sqrt{m}\log m)$.) The only known way to solve the two-dimensional k-differences problem (including insertions and deletions within rows) for non-rectangular patterns is the straightforward dynamic programming approach which takes, for patterns of area a, $O(an^2)$. (This bound can be achieved by constructing an $n^2 \times m$ matrix that for each pattern row P_i and text location, gives the minimum number of differences that P_i incurs when it starts in that location. Such a matrix can be built in time $O(\sum_{i=0}^{m} |P_i|n^2)$, $O(|P_i|n^2)$ for each row P_i. Then for every location do a vertical scan that adds the total minimum number of differences.)

In this paper we make the first advance in the direction of efficiently solving the k-difference matching problem for non-rectangular patterns. We use a novel method that combines the power of convolutions, dynamic programming and subword trees. Our **main contribution** is solving the two-dimensional k-difference matching problem for *half-rectangular* patterns in time $O(kn^2\sqrt{m\log m}\sqrt{k\log k} + k^2n^2)$, where n^2 is the area of the text and m is the height of the pattern. This complexity assumes a word size of $O(\log m)$. Such an assumption is crucial in several recent string matching algorithms [KR87, AIL+88, KLP89].

Definition: A *(left) half-rectangular* pattern is a list of variable-length rows, $P_1, ..., P_m$. The pattern is represented by stacking each row P_i above row P_{i+1} with $P_{i,1}$ directly above $P_{i+1,1}$.

Intuitively, the leftmost border of the pattern is a vertical line, and every horizontal cut of the figure is a single segment. One may similarly define a right, top or bottom half-rectangle depending on whether the right, top or bottom border is a straight edge.

Our algorithm finds all appearances, with no more than k errors, of a half-rectangular pattern P in text T in time $O(kn^2\sqrt{m\log m}\sqrt{k\log k} + k^2n^2)$.

For $k \leq m^{\frac{1}{3}}$ our algorithm is the fastest known.

For $m^{\frac{1}{3}} < k$, the [LV86] algorithm can be modified to solve the problem in time $O((m+k^2)n^2)$, without use of convolutions.

For $k > \sqrt{a}$, where a is the area of P, the straightforward dynamic programming approach takes time $O(n^2 a)$.

To achieve the main result we needed some new tools. We provide efficient solutions to two problems that are interesting in their own right and which have already proven useful in other applications of convolutions [IS92]. These problems are the *smaller matching problem* and the *k-aligned ones with location problem.*

The smaller matching problem is:

Input: Text string $T = T_0, ..., T_{n-1}$ and pattern string $P = P_0, ..., P_{m-1}$ where $T_i, P_i \in N$ (the set of natural numbers).

Output: All locations i in T where $T_{i+k-1} \geq P_k, \quad k = 1, ..., m$.

In words, every matched element of the pattern is not greater than the corresponding text element. If the text and pattern are drawn schematically, we are interested in all position where the pattern lies completely below the text. See Fig. 1.

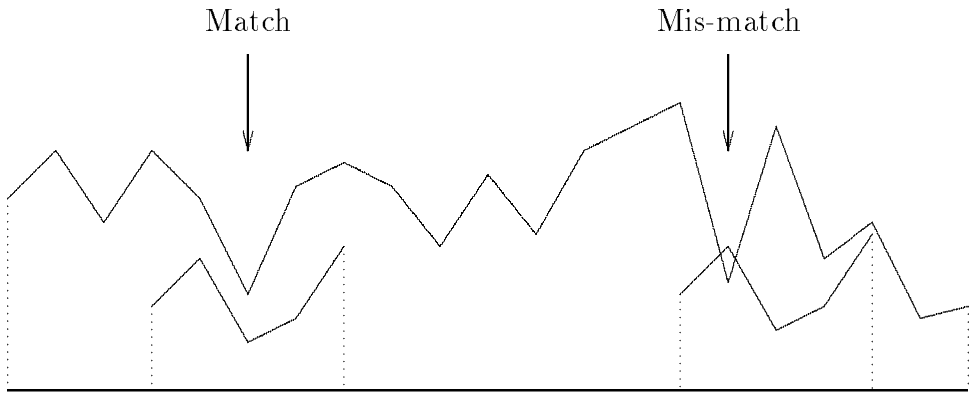

Figure 1: Example of Less-than Match and Mis-match

We defined the smaller matching problem in the case where the text and patterns are strings over the natural numbers. The problem may be generalized to strings over any partially ordered domain D. We need to solve the problem for a special type of partial order, the *forest partial order.* A forest partial order is induced by a forest, i.e. we are given a forest with a bijection between its nodes and the domain. For $a, b \in D, a > b$ iff a is a descendent of b in some tree. We solve the smaller matching problem for strings over the natural numbers and over domains with a forest partial order in time $O(n\sqrt{m}\log m)$. As defined here the problem is 1-dimensional. Our solution will be good for any dimension by a method similar to the one used in [AL91].

The motivation for the k-aligned ones with locations problem stems from the use of convolutions in pattern matching. The power behind all known convolution-based string matching algorithms is multiplication of polynomials with binary coefficients $(0,1)$. Polynomial multiplication can be done efficiently by using Fast Fourier Transform [AHU74]. The result of such a polynomial multiplication is the *number* of 1's in the pattern that are aligned with 1's in the text at each position. However, all information about the *location* of these aligned 1's is lost. We present a method of finding these locations in time $O(k^3 n\log m\log k)$.

Specifically, the k-aligned ones with location problem is:

Input: Text string $T = T_0, ..., T_{n-1}$ and pattern string $P_0, ..., P_{m-1}$ where $T_i, P_i \in \{0,1\}$.

Output: All locations i in T where $\sum_{l=0}^{m-1} T_{l+i}P_l \leq k$ and for each such i, all indices $i_1, ..., i_k$ where $P_{i_j} = T_{l+i_j} = 1$.

In other words, when aligning P in position i of T, there are at most k positions where there is a "1" both in P and T. All these indices of P are output.

Example: $T = 1001111011$; $P = 1101$

When aligning P in position 1 of T, two 1's are aligned with 1's: their locations are 1 and 4. When aligning P in position 4 of T, three 1's are aligned at locations 1,2 and 4.

We use efficient root finding techniques in a symmetric polynomial to solve the k-aligned ones with locations problem in time $O(k^3 n\log m\log k)$. We use this algorithm to find error locations in the smaller matching convolution, but the algorithm can be applied to find error locations in all convolutions (e.g. k-mismatches problem of [Abr87, Kos87]).

In section 2 we show the basic outline of the algorithm for the 2-dimensional matching with k-differences problem. In section 3 we present the smaller matching problem solution. In section 4 we give algorithms for handling a forest partial order. In section 5 we present the k-mismatch with error location problem and its solution. In section 6 we complete the algorithm for 2-dimensional matching of a non-rectangular figure with k differences. We conclude with open problems and future research.

2 Outline of the 2-dimensional Matching Problem

Input: Text matrix $T[1,...,n;1,...,n]$, pattern half-rectangle P, and a natural number k - the maximum number of allowed differences. The pattern is represented by the rows

$P[1,\ldots,m_1], P[2,\ldots,m_2],\ldots,P[m,\ldots,m_m]$.

1. **Preprocessing.** Construct tables and databases used by the algorithm. (Details appear in Section 6.) The preprocessing time is $O(|P|\log|\Sigma|)$.

2. Construct a new $n\times n$ matrix, T_R. T_R contains at each location the number of the longest row of P that starts at that location of T. This stage takes $O(n^2)$ time by using a slight modification of the [AC75] algorithm. (Details appear in Section 6.2).

3. Find all locations $[i,j]$ in T where for all but at most k of the rows P_l, $l\in\{1,...,m\}$, of P, row P_l is a *prefix* of the row represented by $T_R[i+l-1,j]$. Call the (at most k) rows where there is no match the *error rows*. This step is done in time $O(n^2\sqrt{m}\log m)$ by using the smaller matching algorithm described in sections 3 and 4. Moreover, the algorithm for the k-aligned ones with locations computes the actual position of each error row for each location in T in time $O(kn^2\sqrt{m\log m}\sqrt{k\log k})$. Note that this is not the time quoted for the "pure" solution of the k-aligned ones algorithm. This time bound results from a better balance between the number of blocks and the the block size in our divide-and-conquer approach. Full details are explained in Section 6.3.

   We now have all locations where all rows of P match the text with at most k error rows. The problem is that we don't know how many errors really exist in each error row.

4. For any particular alignment of the pattern with the text with fewer than k error rows, we process each error row using the one-dimensional algorithm presented in [LV86]. It was shown in [AL91] that such a dynamic programming algorithm requires processing time $O(e^2)$ for every error row with e errors. We proceed, error row by error row, until more than k total differences are found (no match) or until the error rows are exhausted (match). Thus, the use of this algorithm at this stage requires $O(k^2)$ per text location for a total of $O(k^2n^2)$.

Total Algorithm Time: $O(kn^2\sqrt{m\log m}\sqrt{k\log k}+k^2n^2)$. (Since $|P|<n^2$, the preprocessing time is subsumed in the O by the text scanning time.)

3 Smaller Matching Problem

In a manner similar to the divide and conquer ideas of Abrahamson [Abr87] and Kosaraju [Kos87] we will present two inefficient algorithms for solving the smaller matching problem, then show how a careful combination can yield a fast algorithm.

<u>Observation 1:</u> We will be assuming that $n\leq 2m$. The reason this does not limit generality is that if the smaller matching problem can be solved in time $f(m)$ for $n\leq 2m$, then it can

be solved in time $\frac{n}{m}f(m)$ for any n-length text. Simply divide the text into $2\frac{n}{2m}$ overlapping $2m$-length segments (see Fig. 2) and solve the matching problem separately for each. Clearly, if there is a match in any location of T it will appear in one of the segments.

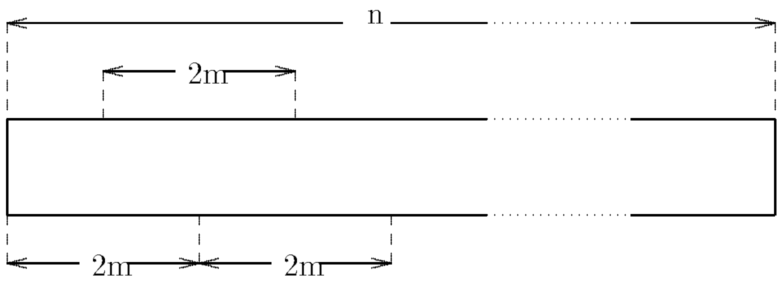

Figure 2: Method for Slicing Problem into Smaller Problems

3.1 The Two Algorithms

The following two inefficient algorithms solve the smaller matching problem. Each algorithm accepts $T = T_0, \ldots, T_{n-1}; P = P_0, \ldots, P_{m-1}$ as input and produces output $M[-(m-1) : n-1]$ where for $i = 0, ..., n-m$, $M[i]$ is the number of locations, j, where $P_j > T_{i+j}$. The values of $M[i]$ where $-(m-1) \leq i < 0$ and where $n - m < i \leq n-1$ are of no interest to us, but they are used by the FFT algorithm which is invoked.

3.1.1 The Brute Force Approach

Algorithm A

Input $T = T_0, \ldots, T_{n-1};\ P = P_0, \ldots, P_{m-1}$.

1. Initialize $M \leftarrow 0$
2. for $i = 0$ to $n-1$ do
   for $j = 0$ to $\min(i, m-1)$ do
      if $T_i < P_j$ then $M[i-j] \leftarrow M[i-j] + 1$

end Algorithm A

This algorithms takes each element of the text and compares it with each element of the pattern to find all the mismatches. Each time it finds one, it updates the vector M appropriately. It is clear that $M[i] = k$ iff there are k locations, j, where $P_j > T_{i+j}$.

Time: $O(mn)$

3.1.2 The Convolutions Approach

A preliminary is required for understanding algorithm B. We will be using convolutions for solving the smaller match problem.

Fischer and Paterson [FP74] observed that string matching is a special case of a generalized convolution.

Definition: Let $X = \langle x_0, ..., x_m \rangle$, $Y = \langle y_0, ..., y_n \rangle$ be two given vectors, $x_i, y_i \in D$. Let $\otimes$ and $\oplus$ be two given functions where

$$\otimes : D \times D \to E,$$

$$\oplus : E \times E \to E, \quad \oplus \text{ associative.}$$

Then the *convolution of X and Y with respect to $\otimes$ and $\oplus$* is:

$$X \langle \otimes, \oplus \rangle Y = \langle z_0, ..., z_{n+m} \rangle$$

where

$$z_k = \bigoplus_{i+j=k} x_i \otimes y_j \qquad \text{for} \quad k = 0, ..., m+n.$$

Examples:

Boolean Product: $\otimes$ is $\wedge$ and $\oplus$ is $\vee$.

Polynomial product: $\otimes$ is $\times$ and $\oplus$ is $+$.

Exact string matching: $\otimes$ is $=$ and $\oplus$ is $\wedge$ but the pattern is transposed.

For all matches of pattern $b\ a\ a$ in text $b\ a\ a\ b\ a$, do $b\ a\ a\ b\ a \langle =, \wedge \rangle b\ a\ a^R$.

$$\begin{array}{ccccccc}
 & & b & a & a & b & a \\
 & & & & a & a & b \\
- & - & - & - & - & - & - \\
 & & 1 & 0 & 0 & 1 & 0 \\
 & 0 & 1 & 1 & 0 & 1 & \\
0 & 1 & 1 & 0 & 1 & & \\
- & - & - & - & - & - & - \\
0 & 0 & 1 & 0 & 0 & 1 & 0 \\
 & & - & - & - & &
\end{array}$$

Note that $(X\langle=,\wedge\rangle Y)_k = 1$ iff $\langle x_{k-n},...,x_k\rangle = \langle y_n,...,y_0\rangle$ for $n \le k \le m$. We conclude that there is a match in position 2, i.e.

$$\begin{array}{cccccc} & b & a & a & b & a \\ & - & - & - & & \\ \langle & a & a & b & \rangle^R & \end{array}$$

Fischer and Patterson achieve such a convolution, using the FFT-based Schönhage-Strassen integer multiplication method, in time $O(\log|\Sigma|n\log m\log\log m)$ (bit complexity). For the word size of $O(\log m)$, which we assume, one can use the FFT for polynomial multiplication achieving the desired result in time $O(\log|\Sigma|n\log m)$. In [AL91] convolutions were used to find approximate matching for a finite alphabet Σ (the convolution $\langle\neq,+\rangle$) in time $O(|\Sigma|n\log n)$. Algorithm B uses a similar technique, but bear in mind that $|\Sigma| = m$ in the worst case.

Notation: For $\sigma, x \in \Re$ let

$$\chi_\sigma(x) = \begin{cases} 1 & \text{if } x = \sigma \\ 0 & \text{if } x \neq \sigma \end{cases}$$

$$\chi_{<\sigma}(x) = \begin{cases} 1 & \text{if } x < \sigma \\ 0 & \text{if } x \ge \sigma \end{cases}$$

If $X = x_1,\ldots,x_n$ then $\chi_\sigma(X) = \chi_\sigma(x_1),\ldots,\chi_\sigma(x_n)$. Similarly define $\chi_{<\sigma}(X)$.

We would like to know for each element of the pattern, where it is lined up with something less than it. In other words, for each σ in P, we can achieve this by computing $\chi_{<\sigma}(T)\otimes\chi_\sigma(P^R)$ (where $\otimes$ is polynomial multiplication), and considering all non-zero locations.

Let $\Sigma = \{\sigma_1,\sigma_2,\ldots,\sigma_k\}$ be the set of all distinct numbers appearing in P. Let $M_{\sigma_i} = \chi_{<\sigma_i}(T)\otimes\chi_{\sigma_i}(P^R)$ (where $\otimes$ is polynomial multiplication). Then by definition of convolution, M_{σ_i} is non-zero at position t iff there is a σ_i in the pattern matched with something smaller than σ_i when the pattern is lined up at t. These cases are exactly when we get a mismatch for the smaller matching problem, i.e. the text is less than the pattern symbol it is being compared to. If we let M be the sum of all the M_{σ_i}'s we get a non-zero if there was a mismatch caused by any $\sigma\in\Sigma$. However, calculating M would take $O(m)$ multiplications.

We assume that the polynomial multiplication result of vectors whose indices are $[0,\ldots,n-1]$ and $[0,\ldots,m-1]$ are returned in a vector whose indices range over $[-(m-1),\ldots,n-1]$. $T_0.\ldots T_n$ and $P_0,\ldots,P_m$ are returned in a vector $M_{-(m-1)},\ldots,M_0,\ldots,M_n$.

<u>Algorithm B:</u>

Input $T = T_0,\ldots,T_{n-1};\ P = P_0,\ldots,P_{m-1}$.

1. Let $\Sigma =$ the set of all different numbers appearing in P. $|\Sigma| \leq m$.

2. Init $M \leftarrow 0$

3. For all $\sigma \in \Sigma$ do
   $M \leftarrow M + \chi_{<\sigma}(T) \otimes \chi_{\sigma}(P^R)$
   end

end Algorithm B

$M[i] = k$ iff there is a smaller match of P in position i of T with k mismatches.

Time: $O(|\Sigma| n \log m)$, by computing $|\Sigma|$ polynomial multiplications using FFT. In the worst case this is $O(mn \log m)$.

3.2 A Fast Algorithm

Abrahamson [Abr87] and, independently, Kosaraju [Kos87] noted that approximate matching (convolution $\langle \neq, + \rangle$) can be done in time $O(n\sqrt{m} \log m)$ rather than $O(mn \log m)$ for infinite alphabets by using the multiplication technique on a limited alphabet of size $\sqrt{m}$ and then "fine tuning" by another method. We use the same idea for the smaller matching problem. The "fine tuning" algorithm is like Algorithm A and the limited alphabet algorithm is like Algorithm B.

Algorithm C

Input $T = T_0, \ldots, T_{n-1}$; $P = P_0, \ldots, P_{m-1}$.

Remember that $n \leq 2m$.

1. Consider $L = \langle T_0, 0, 0 \rangle$, $\langle T_1, 0, 1 \rangle$, $\ldots$, $\langle T_{n-1}, 0, n-1 \rangle$, $\langle P_0, 1, 0 \rangle$, $\langle P_1, 1, 1 \rangle$, $\ldots$, $\langle P_{m-1}, 1, m-1 \rangle$. [Every element is considered a triplet $\langle s, e, d \rangle$ where s is a number, e is 0 if the number is in T and 1 if the number is in P, and d is the location of the number in the array T or P.]

2. Sort L lexicographically. [There are at most $3m$ elements in L.] Call the sorted array L'.

3. Divide L' into b blocks, each containing no more than $3m/b$ elements.

4. For each block $B_i, i = 0, \ldots, b$ let b_i be the smallest (leftmost) element in the block; call b_i the *representative* of block B_i.

5. Let T' and P' be T and P such that every T_i and P_i is replaced by the representative of the block it is in. [Implemented by a sequential scan of L'.]

6. Call Algorithm B for all smallest matches of P' in T'.

   [P' and T' can be considered "flattened out" versions of P an T. When we apply Algorithm B to P' and T' we only detect the "large" mismatches, i.e., those between elements that are so different that they are in different blocks. However, mismatches between elements of the same block are undetected. At this stage we must "fine tune" our approximate solution. We can use a minor modification of Algorithm A for this purpose. The change to Algorithm A is that for every element T_i of T we only compare it to the (at most $3m/b$) elements of P that are in T_i's block.]

7. For $i = 0$ to $n-1$ do (remember that $n \le 2m$)
   Let B_{T_i} be the block of L' that T_i is in,
   Let $P^{B_{T_i}} \leftarrow \{\langle s,1,d\rangle | \langle s,1,d\rangle \in B_{T_i}\}$
   For every element $\langle s,1,d\rangle$ in $P^{B_{T_i}}$ (at most $3\sqrt{m}$ elements)
   if $T_i < s$ then $M[i-d] \leftarrow M[i-d]+1$
   end
   end

   [The vector M is now correct since the first part of the algorithm summed up exactly all the mismatches between elements in different blocks, and the last step found all the mismatches within blocks.]

end Algorithm C

Time:	$O(m \log m)$ for sorting
	$O(mb \log m)$ for Algorithm B
	$O(m\frac{m}{b})$ for Algorithm A
Total:	$O(mb \log m + m^2/b)$

By Observation 1, the time for a general text T of size n is

$$O(\frac{n}{m} mb \log m + \frac{n}{m}\frac{m^2}{b}) = O(nb \log m + \frac{nm}{b}).$$

Notice that for large m this can be a great saving over the naive $O(mn)$ algorithm, especially if b (the number of blocks) is chosen judiciously, e.g. $b = \sqrt{m}$. The algorithm construction is simple enough that there are no prohibitive constants lurking around. In fact, the polynomial multiplication can be done even faster using off-the-shelf FFT hardware.

4 A Forest Partial Order

4.1 A Forest Partial Order

The smaller match problem as defined in Section 2 assumes that the elements of T and P are numbers, i.e. are totally ordered. We can achieve the same results if the symbols are partially ordered in a forest.

Definition: Let $T = T_0, \ldots, T_{n-1}$; $P = P_0, \ldots, P_{m-1}$, where each T_i, P_j $i = 0, \ldots, n-1; j = 0, \ldots, m-1$ appears once in a forest of size $O(n)$ with roots $r_1, \ldots, r_l$, and the relation $a < b$ holds if a is an ancestor of b. The *smaller matching problem with a forest partial order* is that of finding all locations k, $0 \le k \le n-m$, where $P_i \le T_{k+i}; i = 0, \ldots, m-1$.

Algorithm D

1. For each tree rooted at $r_i; i = 1, \ldots, l$ assign a pair of numbers $\langle f(s), g(s) \rangle$ to every node s such that

   $f(a) < f(b)$ iff a is an ancestor of b or $\exists c, c$ an ancestor of both a and b and a is in a subtree of c that is to the right of the subtree b is in.

   $g(a) < g(b)$ iff b is an ancestor of a or $\exists c, c$ an ancestor of both b and a and a is in a subtree of c that is to the right of the subtree b is in.

   Note that in our partial order $a \le b$ iff $f(a) \le f(b)$ and $g(b) \le g(a)$.

2. Let

   T_r, P_r be T, P respectively where every element is replaced by the root of the tree it appears in,

   T_f, P_f be T, P respectively where every element s is replaced by $f(s)$,

   T_g, P_g be T, P respectively where every element s is replaced by $g(s)$.

3. Use any linear time algorithm to find all exact matches of P_r in T_r. [BM77, KMP77].

4. Use Algorithm C to find the smaller matches of P_f in T_f and a symmetric algorithm to find the greater matches of P_g in T_g.

5. The locations where there is an exact match of P_r in T_r (otherwise the elements are not comparable), a smaller match of P_f in T_f and a greater match of P_g in T_g (by definition of f and g) are exactly those where there is a smaller match with a forest partial order of T in P.

end Algorithm D

Implementation

Do a DFS with priority to the <u>rightmost child</u>. When backing up form node s, assign $g(s)$ starting from 1 and increasing to $size$, where $size$ is the size of the tree.

Do a DFS with priority to the <u>leftmost child</u>. When backing up form node s, assign $f(s)$ starting from $size$ and decreasing to 1.

Time:	$O(n)$ for steps 1, 3 and 5
	$O(n)$ for step 2
	$O(n\sqrt{m}\log m)$ for step 4
Total:	$O(n\sqrt{m}\log m)$

5 The k-Aligned Ones with Location Problem

5.1 Definition

Definition: Let $T = T_0, \ldots, T_{n-1}; P = P_0, \ldots, P_{m-1}; T_i, P_j \in \{0,1\}, i = 0, \ldots, n-1; j = 0, \ldots, m-1$. We say there is a $1 \cdot 1$ pair at index j of position i if $P_j = T_{i+j} = 1$. The *k-aligned ones with location problem* is that of finding for each i that has fewer than k $1 \cdot 1$ pairs, the set of j for which $P_j = T_{i+j} = 1$.

The polynomial multiplication of two sequences P and T^R computes the number of $1 \cdot 1$ pairs for each alignment of P and T. Replace every 1 in P with some other value; call the result P'. The polynomial multiplication P' and T^R computes the sum of values in P' that are aligned with a corresponding 1 in T. By carefully selecting the values that get substituted for the 1's in P, we can get information about where the $1 \cdot 1$ pairs are. We generate a set of k equations with k unknowns from which the actual locations of the $1 \cdot 1$ pairs are derived. The way we choose to construct these equations is by substituting for each 1 in P, its index in P raised to the j-th power. We do this k time, with j being 1 the first time, and incremented by one in each subsequent convolution. The exact details are given in the next section.

5.2 The Convolutions

For a given vector $V = \langle v_1, v_2, \ldots, v_m\rangle \in N^m$ and function $f : N \to N$, extend f to $f : N^m \to N^m$ as follows: $f(V) = \langle f(v_1), f(v_2), \ldots, f(v_m)\rangle$. For $1 \le i \le k$, let $f_i(v_j) = v_j j^i$ and let

$$C_i = C_{i,1}C_{i,2}\ldots C_{i,n+m} = f_i(P) \otimes T^R.$$

In other words, if P is lined up with T starting at position j, and the $1 \cdot 1$ pairs occur in locations $x_1, x_2, \ldots, x_k$ of P, we have:

$$
\begin{array}{rcl}
C_{1,m+j} & = & x_1 + x_2 + \cdots + x_k \\
C_{2,m+j} & = & x_1^2 + x_2^2 + \cdots + x_k^2 \\
& \vdots & \\
C_{k,m+j} & = & x_1^k + x_2^k + \cdots + x_k^k
\end{array}
$$

Since the $C_{i,j}$'s can grow to $O(km^k)$, they require $O(k \log m)$ bits each, or $O(k)$ words. Therefore, we cannot do multiplications or additions in constant time. For the sake of clarity, we will count the number of multiplications required for the computation assuming constant time per multiplication. However, our multiplications are "long multiplications", i.e. multiplications of numbers each of which ranges over up to k words. At the end of our discussion, we will multiply the complexity by the amount of time it takes to do one long multiplication, and that will produce the complexity of the algorithm.

Time: $O(kn \log m)$ multiplications, since we do k convolutions.

Given the $C_{i,l}$'s, we would like to be able to find the x_i's for all $m \le l \le n$.

5.3 Converting the System of Equations into a Polynomial

The above system of equations does not present an obvious solution. However, if we can use the $C_{i,l}$'s to calculate the coefficients of the polynomial:

$$g(x) = (x + x_1)(x + x_2) \dots (x + x_k) = 0 \tag{1}$$

then we can find the roots of this polynomial and negate them to solve for the x_i's. Let the expanded form of (1) be:

$$g(x) = s_0 x^k + s_1 x^{k-1} + \cdots + s_{k-1} x + s_k = 0.$$

Clearly $s_0 = 1$ and for $1 \le i \le k$,

$$s_i = \sum_{1 \le \alpha_1 < \alpha_2 < \cdots < \alpha_i \le k} x_{\alpha_1} x_{\alpha_2} \dots x_{\alpha_i}$$

It is easy to see that $s_1 = \sum_{1 \le i \le k} x_i = C_{1,l}$. A bit of inspection will also reveal that $s_2 = \sum_{1 \le i < j \le k} x_i x_j = (C_{1,l}^2 - C_{2,l})/2 = (s_1 C_{1,l} - s_0 C_{2,l})/2$. This suggests the following general result.

Lemma 1 *For* $1 \le i \le k$

$$s_i = \frac{\sum_{j=1}^{i} (-1)^{j-1} s_{i-j} C_{j,l}}{i}.$$

Proof: For $d, e \in N$ such that $0 \leq d - e < k$, let

$$h(d,e) = \{(a,B)|1 \leq a \leq k, B \subseteq \{1,\ldots,k\}, |B| = d-e, a \notin B\}$$

Let

$$\check{h}(d,e) = \sum_{(a,B)\in h(d,e)} x_a^e \prod_{b\in B} x_b.$$

Then clearly $s_i = \check{h}(i,1)/i$ since every product of i x_j's appears i times in the summation of $\check{h}(i,1)$ and $\check{h}(i,i) = C_{i,l}$.

Claim 1 *For* $1 \leq i, j < k$

$$s_i C_{j,l} = \check{h}(i+j,j) + \check{h}(i+j,j+1)$$

Proof: We know that $\check{h}(i,1) = is_i$ and $\check{h}(j,j) = C_{j,l}$. If we can prove that

$$\frac{\check{h}(i,1)}{i}\check{h}(j,j) = \check{h}(i+j,j) + \check{h}(i+j,j+1),$$

then we are done.

Let $(a,B) \in h(i+j,j)$. But $(a,\emptyset) \in h(j,j)$ and $\forall\alpha \in B, (\alpha, B - \{\alpha\}) \in h(i,1)$. Similarly, let $(a,B) \in h(i+j,j+1)$. $(a,\emptyset) \in h(j,j)$ and $\forall\alpha \in B\bigcup\{a\}, (\alpha, B\bigcup\{a\} - \{\alpha\}) \in h(i,1)$. So, for every term in $\check{h}(i+j,j) + \check{h}(i+j,j+1)$, there are i terms with the same value in $\check{h}(i,1)\check{h}(i,i)$. Conversely, for every $(c,B) \in h(i,1), (d,\emptyset) \in h(j,j)$ we have two cases. If $d \in B\bigcup\{c\}$ then $x_d^j x_c \prod_{b\in B} x_b = x_d^{j+1} \prod_{b\in B\bigcup\{c\}-\{d\}} x_b$ which is a term in $\check{h}(i+j,j+1)$. If $d \notin B\bigcup\{c\}$ then $x_c x_d^j \prod_{b\in B} x_b = x_d^j \prod_{b\in B\bigcup\{c\}} x_b$ which is a term in $\check{h}(i+j,j)$. Recall that there are i elements of $h(i,i)$ that contribute to the sum $x_d^j x_c \prod_{b\in B} x_b$. Therefore $\check{h}(i,1)\check{h}(i,i)/i = h(i+j,j) + \check{h}(i+j,j+1)$ and $s_i C_{j,l} = \check{h}(i+j,j) + \check{h}(i+j,j+1)$. $\square$

Claim 2 *For* $1 \leq y < x \leq k$,

$$\check{h}(x,y) = \sum_{j=y}^{x} (-1)^{j-y} s_{x-j} C_{j,l}.$$

Proof: For $y = x$ we have

$$\begin{aligned}\check{h}(x,x) &= \sum_{j=x}^{x} (-1)^{j-x} s_{x-j} C_{j,l} \\ &= s_0 C_{x,l}.\end{aligned}$$

Assume claim 2 holds for all $y, 1 \leq y_0 < y \leq x$. Then we know that

$$\begin{array}{rcl} \check{h}(x,y_0) & = & \check{h}(x,y_0) + \check{h}(x,y_0+1) - \sum_{j=y_0+1}^{x}(-1)^{j-y_0+1}s_{x-j}C_{j,l} \\ & = & s_{x-y_0}C_{y_0,l} - \sum_{j=y_0+1}^{x}(-1)^{j-y_0+1}s_{x-j}C_{j,l} \\ & = & \sum_{j=y_0}^{x}(-1)^{j-y_0}s_{x-j}C_{j,l}.\square \end{array}$$

By claim 2, we get that

$$\check{h}(i,1) = \sum_{j=1}^{i}(-1)^{j-1}s_{i-j}C_{j,l}.$$

Since $s_i = \check{h}(i,1)/i$,

$$s_i = \frac{\sum_{j=1}^{i}(-1)^{j-1}s_{i-j}C_{j,l}}{i}.\square$$

The time needed to convert from the $C_{i,l}$'s to the s_i's is clearly $O(k^2)$ multiplications. Note that the s_i's can grow as large as $O((km)^k)$ and thus can also be represented with $O(k)$ words. Note also that while in general such a method may need to handle round-off errors, in our case all s_i's are guaranteed to be natural numbers, since $x_i \in \{1,\ldots,,\},\ \ i = 1,\ldots,k$.

5.3.1 Finding the roots of the Polynomial

We now have a polynomial, $g(x)$ of degree k, all of whose roots are distinct integers between -1 and $-m$ (remember that we had to negate the roots of the polynomial to get the x_i's). Since g has k real roots, and g' (the derivative of g) has $k-1$ roots, we know that the roots of g' come in two varieties: they are either maxima of g above the x-axis, or minima of g below the x-axis. This follows from the fact that between any two roots of a polynomial there must be at least one root of its derivative, and since g has k real roots, there is exactly one extrema between every pair of adjacent roots of g. We show how to find one root of g. Once such a root (say $-x_i$) is found, repeat the same process to find a root of $g(x)/(x+x_i)$. Repeat the process until all roots are found.

To find a root of a $g(x)$, let us consider two cases.

Case 1: $g(x)$ has odd degree.

Then $g(0)g(-m-1) < 0$, that is, $g(x) > 0$ for all $x < -m$ and $g(x) < 0$ for all $x > -1$ or vice-versa (See Figure 3). Since we have a positive and a negative point, we know that we have bounded a root of g. If we evaluate g at the midpoint of this interval, we will either find a root, or we will have bounded a root in an interval half the size. We can continue with the binary search for at most $O(\log m)$ steps, and each step requires $O(k)$ multiplications to evaluate the k degree polynomial g. So the total time for finding a root of an odd degree polynomial is $O(k \log m)$ multiplications.

Case 2: $g(x)$ has even degree.

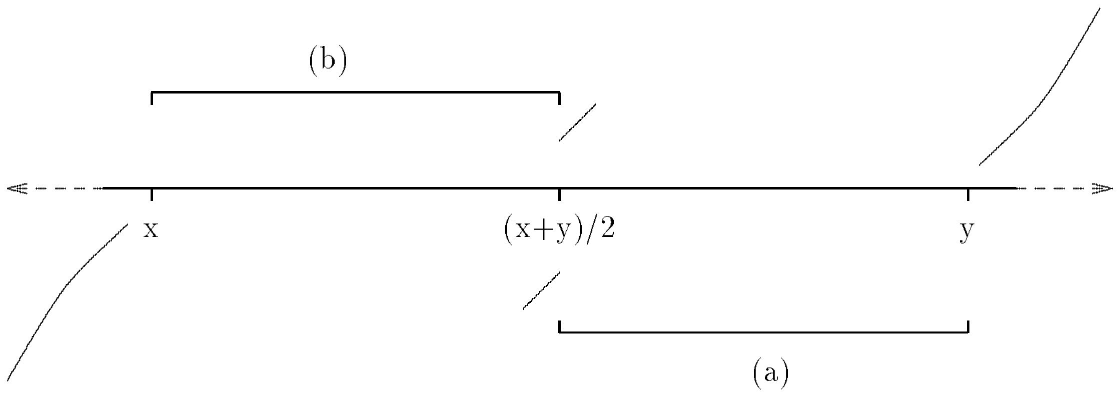

Figure 3: Case (a): $g(\frac{x+y}{2}) < 0$; Case (b): $g(\frac{x+y}{2}) > 0$

Then g' has odd degree. Let us assume, without loss of generality, that $g(0) > 0$ and $g(-m-1) > 0$. Then $g'(0) > 0$ and $g'(-m-1) < 0$. We have bounded a root of g' in the interval $[x, y]$, where initially, $x = -m$ and $y = -1$. Once again we proceed by binary search, but we have more work to do this time. If g is 0 at the midpoint, we have found a root. If it is negative, we can proceed with the binary search as in case 1, since we now have an interval bounded by a negative and positive point of g. If g is positive at the midpoint, then we evaluate g' at the midpoint. If g' is not 0, then we have the original condition on one of the two halves (that is, a positive derivative at one end, and a negative derivative at the other), so we simply proceed with the binary search on the appropriate half (See Figure 4) .

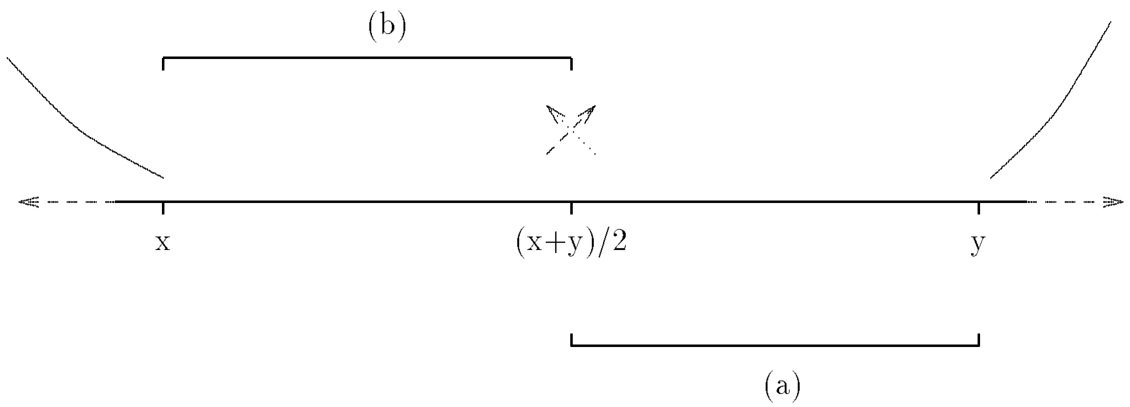

Figure 4: Case (a): $g'(\frac{x+y}{2}) < 0$ (dotted line) ; Case (b) $g'(\frac{x+y}{2}) > 0$ (dashed line)

If g' is 0, then it is a maximal point (See Figure 5). This means there is a root on each side.

We continue with the binary search on either side. Once again, we have $O(\log m)$ steps of the binary search, and each step takes $O(k)$ multiplications (to evaluate g and g') so the total time to find a root is $O(k \log m)$ multiplications.

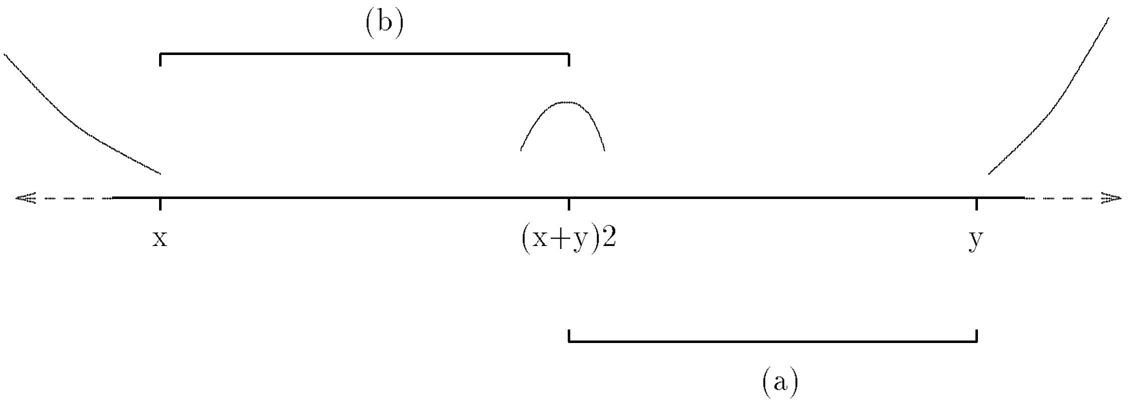

Figure 5: $g'(\frac{x+y}{2}) = 0$ so the search continues on either (a) or (b)

The process is repeated k times to find all k roots, so we have $O(k^2 \log m)$ multiplications. We now need to account for the fact that each multiplication is a long multiplication, i.e. a multiplication of two k-word numbers. Each such multiplication can be done by convolutions in time $O(k \log k)$. Thus the total time for computing the x_i's for each text location is $O(k^3 \log m \log k)$.

Since there are $O(n)$ locations for which we need to solve the system of equations, the total algorithm time is:

Time: $O(k^3 n \log m \log k)$.

6 Two-Dimensional Non-rectangular Matching

Recall that the algorithm proceeds in four phases: preprocessing; writing down the longest row that ends at a particular location; finding, for each alignment of the pattern, P, in the text, T, which rows don't match; and checking each mismatched row to add up the number of errors.

6.1 Preprocessing

We will use the *Smaller Matching Problem with a Forest Partial Order* in our algorithm, so we must first build the partial order. We now show how to construct a forest where each node represents a pattern row and where node a is an ancestor of node b iff the string represented by a is a prefix of the string represented by b.

Our idea is to construct a trie of pattern rows $P_1, \ldots, P_m$. We call a node where a pattern row ends, a *row node*, and associate it with the row that ends there.

Claim: The ancestor relation between row nodes in the trie is equivalent to the prefix relation.

Proof: By definition of a trie, the ancestor relation between nodes is the prefix relation between the strings ending in these nodes. In particular, this is true for row nodes. $\Box$

Now all we need to do is construct the forest resulting from the ancestor relation of row nodes in the trie.

Algorithm F

Input the rows of P $P_1, P_2, \ldots, P_m$

1. Construct the trie of $P_1, \ldots, P_m$ noting each row node with its row number.
2. Compact the trie to the forest where the parent of each row node is its lowest trie ancestor which is a row node.

end Algorithm F

Time: $O(|P| \log |\Sigma|)$, since the size of the trie is $O(|P|)$, and constructing the trie necessitates a binary search on the edges. The number of edges at each node may be as large as $|\Sigma|$.

6.2 Finding the Longest Matching Rows

We modify the algorithm of Aho and Corasick, [AC75]. We will modify the output table so that only the longest string is output at any stage. We will then use the modified table for the row scan to construct T_R as described in the outline. For simplicity of notation we assume that the text is given within a square array (although clearly the algorithm can handle rectangles).

Algorithm G

Input $T[0 : n-1; 0 : n-1]$, $P[0; 0 : m_0], P[1; 0 : m_1], \ldots, P[m-1; 0 : m_{m-1}]$.

1. Build an [AC75] DFA for P. Call it D with output table O.
2. Modify the output table to only output the longest string from each entry. Call the new table O'.
3. Create T_R as follows. Run D on the rows of T using the output table O'. If no string matches at $T[i;j]$ then $T_R[i;j] \leftarrow \$$, where \$ matches no string. If P_k is the longest row that matches at $T[i;j]$ then $T_R[i;j] \leftarrow P_k$.

   [At every position (i,j) of T_R we have the longest string which *starts* at $T[i;j]$]

<u>End Algorithm G</u>

Time: $O(n^2 \log |\Sigma|)$

6.3 Finding the Mismatched Rows

Input to this step: T_R. Each element of this array is a symbol representing the longest pattern row that matches that text location. We also have precomputed tree order relations for the "is-a-prefix-of" relation on the rows of P.

Processing: Viewing T_R in column major order, solve the smaller matching problem and choose only the locations where the number of mismatches in T_R is not greater than k. Use the k-aligned ones with locations algorithm to find the location of the error rows resulting from each alphabet letter in the convolution stage. Compute the location of the mismatches during the brute force stage. This is easily done since every location is checked.

Time: The k-aligned ones with location algorithm takes time $O(k^3 n^2 \log m \log k)$ for every alphabet symbol. If we choose the number of blocks b to be $\sqrt{m}$ then there are $O(\sqrt{m})$ symbols so the total time is $O(k^3 n^2 \sqrt{m} \log m \log k)$. The brute force stage takes time $O(n^2 \sqrt{m})$.

A better divide-and-conquer scheme:

For a more even distribution of labor, use

$$b = \frac{\sqrt{m}}{k\sqrt{\log m}\sqrt{k \log k}}.$$

New block size: $3k\sqrt{m \log m}\sqrt{k \log k}$.

Time: The k-aligned ones with locations algorithm now contributes $O(k^3 n^2 \log m \log k \frac{\sqrt{m}}{k\sqrt{\log m}\sqrt{k \log k}}) = O(kn^2 \sqrt{m \log m}\sqrt{k \log k})$. The brute force stage also takes time $O(kn^2 \sqrt{m \log m}\sqrt{k \log k})$.

6.4 Adding up the Number of Errors in the Mismatched Rows

We now have, for every location in the text, the rows of the pattern that have mismatches on when the pattern is aligned at that location of the text (for all locations where there are fewer than k errors).

For every text location with fewer than k error rows, we do the following. Starting with the first error row, find the differences in matching this row with the appropriate pattern row. If fewer than k differences are found, continue to the next error row. For each text location, stop processing when $k+1$ differences are found or after the last row is checked. If no more than k differences are found after checking the last column conclude that there is an occurrence of the pattern in the text. We find the differences between the error rows and their corresponding pattern rows by the algorithm given in [LV86]. The total time for this stage is $O(k^2n^2)$.

6.5 Complexity of the Algorithm

Finding the error rows takes time $O(kn^2\sqrt{m\log m}\sqrt{k\log k} + n^2k\log k)$. Finding the k differences within the error rows takes time $O(k^2n^2)$. All other steps run in linear time. The total is thus $O(kn^2\sqrt{m\log m}\sqrt{k\log k} + k^2n^2)$

For $k \leq m^{\frac{1}{3}}$ our algorithm is the fastest known.

For $m^{\frac{1}{3}} < k$, the [LV86] algorithm can be modified to solve the problem in time $O((m+k^2)n^2)$, without use of convolutions.

For $k > \sqrt{a}$, where a is the area of P, the straightforward dynamic programming approach takes time $O(n^2a)$.

7 Open Problems

In this paper we solved the two dimensional k-differences problem for vertical non-rectangular figures. It remains open to find efficient algorithms for non-vertical figures. This is related to the problem of string matching with k-differences and don't cares. Does an efficient smaller match algorithm exist for any partial order? We conjecture that convolutions such as $\langle \leq, + \rangle$, $\langle max, min \rangle$ or $\langle \neq, + \rangle$ can be solved in time $O(n\log m)$. There are almost no known lower bounds in this area.

8 Acknowledgments

The authors thank Professor S. Rao Kosaraju for bringing to our attention the methods for efficiently solving convolutions $\langle +, \neq \rangle$ and $\langle max, min \rangle$ and the Computer Science Department of the Johns Hopkins University for the use of its facilities. We also wish to warmly thank Rod Fontecilla, Ed Edmundson and Larry Washington.

References

[ABF92] A. Amir, G. Benson, and M. Farach. Alphabet independent two dimensional matching. *To appear, SIAM J. Comp.*, 1992.

[Abr87] K. Abrahamson. Generalized string matching. *SIAM J. Computing*, 16(6):1039–1051, 1987.

[AC75] A.V. Aho and M.J. Corasick. Efficient string matching. *C. ACM*, 18(6):333–340, 1975.

[AF91] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. *Proc. of 2nd Symoposium on Descrete Algorithms, San Fransisco, CA*, Jan 1991.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AIL+88] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347–365, 1988.

[AL91] A. Amir and G. Landau. Fast parallel and serial multidimensional approximate array matching. *Theoretical Computer Science*, 81:97–115, 1991.

[ALV90] A. Amir, G.M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *Proceedings of First Symposium on Discrete Algorithms, San Fransisco, CA*, 1990.

[Bak78] T.J. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comp*, 7:533–541, 1978.

[Bir77] R.S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977.

[BM77] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[BYR90] R. Baeza-Yates and M. Régnier. Fast algorithms for two dimensional and multiple pattern matching algorithms. *Proc. SWAT 90*, pages 332–347, 1990.

[FP74] M.J. Fischer and M.S. Paterson. String matching and other products. *Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.

[GS83] Z. Galil and J.I. Seiferas. Time-space-optimal string matching. *J. Computer and System Science*, 26:280–294, 1983.

[IS92] R.M. Idury and A.A Schäffer. Multiple matching of rectangular patterns. Technical Report TR-92-196, Rice University, Nov 1992.

[KLP89] Z. M. Kedem, G. M. Landau, and K. V. Palem. Optimal parallel suffix-prefix matching algorithm and application. *Proceedings of the Symposium on Parallel Algorithms and Architectures*, 6:388–398, 1989.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

[Kos87] S. Rao Kosaraju. Efficient string matching. Manuscript, 1987.

[KR87] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Res. and Dev.*, pages 249–260, 1987.

[KS87] K. Krithivansan and R. Sitalakshmi. Efficient two dimensional pattern matching in the presence of errors. *Information Sciences*, 13:169–184, 1987.

[LV86] G.M. Landau and U. Vishkin. Introducing efficient parallelism into approximate string matching. *Proc. 18th ACM Symposium on Theory of Computing*, pages 220–230, 1986.

[MR92] S. Muthukrishnan and H. Ramesh. A constant time optimal parallel algorithm for two dimensional matching. Manuscript, 1992.

[PG92] K. Park and Z. Galil. Truly alphabet-independent two-dimensional pattern matching. *Proc. 33rd IEEE FOCS*, pages 247–256, 1992.

[Ukk85] E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

[Vis89] U. Vishkin. Deterministic sampling for fast pattern matching. Manuscript, 1989.

[Wei73] P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.