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

Amihood Amir† Martin Farach‡
Georgia Tech DIMACS

September 9, 1991

Abstract

Finding all occurrences of a non-rectangular pattern of height m and area a in an $n \times n$ text with no more than k mismatch, insertion, and deletion errors is an important problem in computer vision. It can be solved using a dynamic programming approach in time $O(an^2)$. We show a $O(kn^2\sqrt{m \log m}\sqrt{k \log k} + k^2n^2)$ algorithm which combines convolutions with dynamic programming. At the heart of the algorithm are the *Smaller Matching Problem* and the *k-Aligned Ones with Location Problem*. Efficient algorithms to solve both these problems are presented.

*The results presented in this paper appeared in the proceedings of the Second Symposium on Descrete 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 $T = T_1T_2 \cdots T_n$ and pattern string $P = P_1P_2 \cdots P_m$, where $T_i, P_i \in \Sigma$. *Output:* All locations i in T where $T_iT_{i+1} \cdots T_{i+m-1} = P_1P_2 \cdots P_m$. 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 $O(\log|\Sigma| n \log m)$ word operations, where the words are of size $O(\log n)$. 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 $O(|\Sigma| n \log m)$. 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 a $O(n)$ 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 $O(n + \text{total length of the patterns} + \textit{output size})$. Here *output size* is the number of times a pair $(\textit{location}, \textit{pattern})$ 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.

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 $O(nk)$. 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 is the text (*mismatch*).

(b) A text pattern 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).

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.

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

- local errors: caused by occlusion and varying 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 *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 $O(n^2)$.

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 in $T[i+l-1,j] \cdots T[i+l-1, j+m-1]$ 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, $O(an^2)$.

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 $O(\log m)$ as is standard for string matching algorithms.

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 is efficient for any pattern that can be split into a "small" number of half-rectangular shapes. In this paper we illustrate it with *vertical patterns* - that can be split vertically into two half rectangles (see Fig. 1). An example is any convex shape in an orientation where the longest diameter is vertical. We are searching for all locations where a vertical pattern matches the text allowing no more than k mismatches, insertions (in rows) and deletions (in rows) errors.

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 may prove useful in other applications of convolutions. 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 $T_i, P_i \in N$. *Output:* All locations i in T where $T_{i+k-1} \geq P_k k = 1, ..., m$. In words, every matched element of the pattern is not greater than the corresponding text

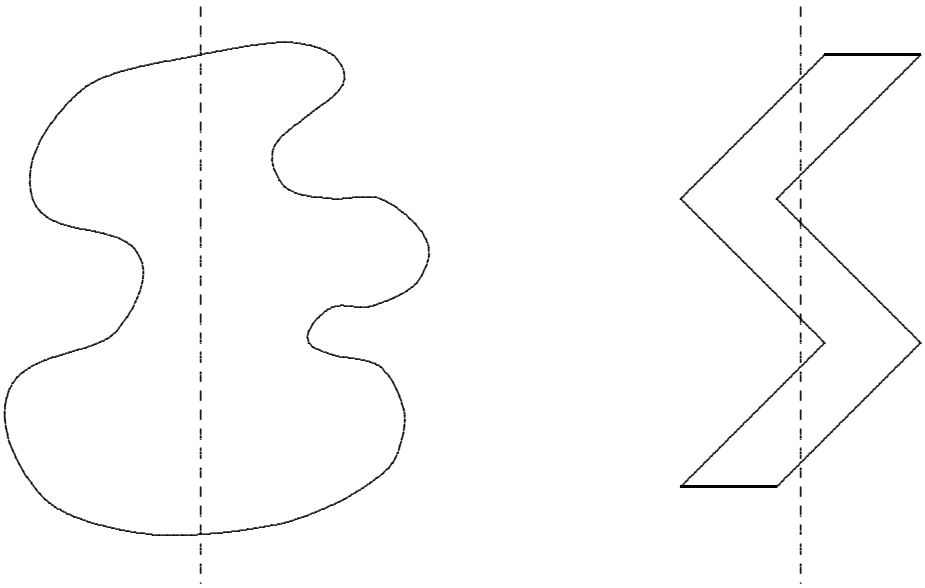

Figure 1: Examples of Figures which Can and Cannot be Cut Properly

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. 2.

The smaller matching problem with a forest partial order is defined similarly with the exception that the order relation is that induced by a given forest. We solve both these problems 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

$$\sum_{l=0}^{m} T_{l+i} P_l \leq k$$

and for each such i, all indices $i_1, ..., i_k$ where $P_{i_j} = T_{l+i_j} = 1$.

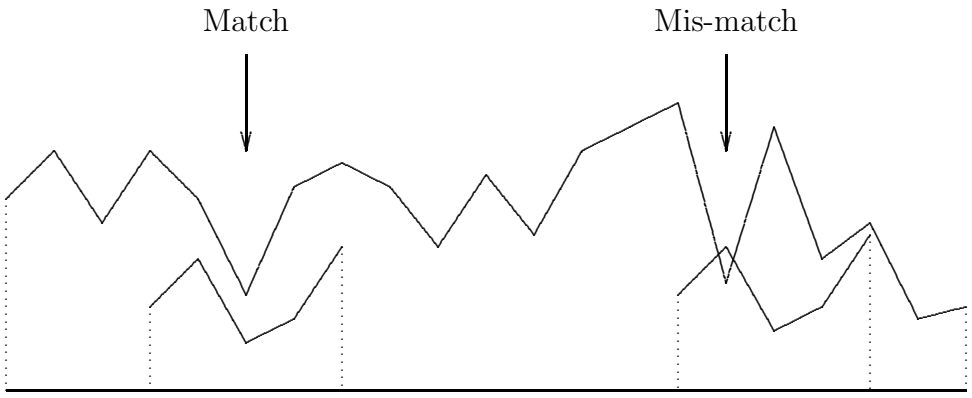

Figure 2: Example of Less-than Match and Mis-match

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: $T = 1001111011; \qquad P = 1101$

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]).

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 definition and 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

The idea of the algorithm is to split the pattern along the verticle line into two parts, P_L and P_R. Next, find all locations where each of the halves P_L and P_R match with no more than k differences. Subsequently choose only the locations in the intersection that total at most k differences.

Input: Text matrix $T[1,...,n;1,...,n]$, pattern image P, and a natural number k - the maximum number of allowed differences. The pattern may be represented as a matrix, with don't care characters padding the borders, or row by row, with the location of the vertical cut indicated at each row.

1. **Preprocessing.** Split pattern P into P_R and P_L along the vertical cut, where P_R is the right part and P_L is the left part. Construct tables and databases used by the algorithm. (Details appear in Section 6.)

2. Construct two new $n \times n$ matrices, T_L and T_R. T_R contains at each location the number of the longest row of P_R that starts at that location of T. T_L contains at each location the number of the longest row of P_L that ends 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 $P_{R,l}$, $l \in \{1,...,m\}$, of P_R, row $P_{R,l}$ is a *prefix* of the row represented by $T_R[i+l-1,j]$. Similarly, find all locations $[i,j]$ in T where for all but at most k of the rows $P_{L,l}$, $l \in \{1,...,m\}$, of P_L, row $P_{L,l}$ is a *suffix* of the row represented by $T_L[i+l-1,j]$. 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 $O(kn^2\sqrt{m\log m}\sqrt{k\log k})$.

   By considering the intersection of the locations where P_L and P_R match in at least $m-k$ rows, 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: $O(kn^2\sqrt{m\log m}\sqrt{k\log k} + k^2n^2)$.

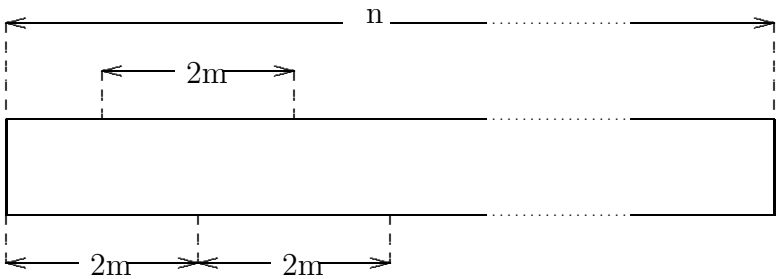

Figure 3: Method for Slicing Problem into Smaller Problems

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.

Observation 1: 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. 3) and solve the matching problem separately for each.

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 $M[-(m-1) : n]$ where for $i = 0, ..., n-m$, $M[i]$ is the number of locations, j, where $P_j > T_{i+j}$.

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

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

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

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 $n \leq k \leq m$. We conclude that there is a match in position 2, i.e.

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

Fischer and Patterson achieve such a convolution, using the FFT-based Schönhage-Strassen integer multiplication method, in time $O(\log|\Sigma| n \log m)$. 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

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

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

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 different numbers appearing in P. Let $M_i = \chi_{<\sigma_i}(T) \otimes \chi_{\sigma_i}(P^R)$ (where $\otimes$ is polynomial multiplication). Then M_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. If we let M be the sum of all the M_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 $T_0 \ldots T_n$ and $P_0, \ldots, P_m$ are returned in a vector $M_{-(m-1)}, \ldots, M_0, \ldots, M_n$.

Algorithm B:

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 errors.

Time: $O(|\Sigma| n \log m)$. 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 l 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 $\sqrt{m}$ blocks, each containing no more than $3\sqrt{m}$ elements.

4. For each block $B_i, i = 0, \ldots, \sqrt{m}$ 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 $\sqrt{m}$) elements of P that are in T_i's block.]

7. For $i = 0$ to $n - 1$ do (remember that $n = 2m$)
   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 included all the errors between blocks and the last part found all the errors within a block.]

end Algorithm C

Time:	$O(m \log m)$ for sorting
	$O(m\sqrt{m} \log m)$ for Algorithm B
	$O(m\sqrt{m})$ for Algorithm A
Total:	$O(m\sqrt{m} \log m)$

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

$$O(\frac{n}{m} m\sqrt{m} \log m) = O(n\sqrt{m} \log m).$$

Notice that for large m this can be a great saving over the naive $O(mn)$ algorithm. 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, $0 \leq k \leq n-m$ where $P_i \leq T_{k+i}; i = 0, \ldots, m-1$.

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. 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. a is in a subtree of c, that is, to the right of the subtree b is in.

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

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, a smaller match of P_f in T_f and a greater match of P_g in T_g are exactly those where there is a smaller match with a forest partial order of T in P.

end Algorithm D

Implementation

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

2. Sort all elements of T, P in some lexicographic order of their (binary) encoding. As an element is assigned a number in the DFS, assign that number to the sorted list. By binary search T_f, P_f, T_g, P_g can be constructed.

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

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

$$O(\frac{n}{m}m\sqrt{m} \log m) = O(n\sqrt{m} \log m)$$

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 $1-1$ 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 $1-1$ 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 $1-1$ 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 $1-1$ pairs are. We generate a set of k equations with k unknowns from which the actual locations of the $1-1$ pairs are derived.

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 $1 \leq i \leq k$, 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_1 is lined up with T_j and the $1-1$ pairs occur in locations $x_1, x_2, \ldots, x_k$, we have:

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

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 and we will multiply this by the time needed to compute each multiplication at the end.

Time: $O(kn \log m)$ multiplications.

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

5.3 Converting the System of Equations into a Polynomial

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

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

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 $1 \leq i \leq k$,

$$s_i = \sum_{1 \leq \alpha_1 < \alpha_2 < \cdots < \alpha_i \leq k} x_{\alpha_1} x_{\alpha_2} \ldots x_{\alpha_i}$$

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

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

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

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: See Appendix.

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: See Appendix.

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 represendted with $O(k)$ words.

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.

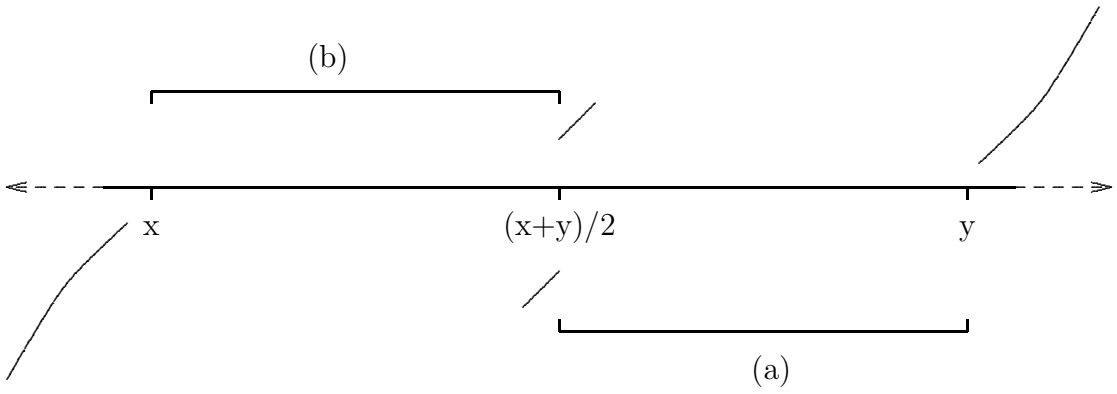

Figure 4: Case (a): $g(\frac{x+y}{2}) < 0$; Case (b): $g(\frac{x+y}{2}) > 0$

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 4). 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

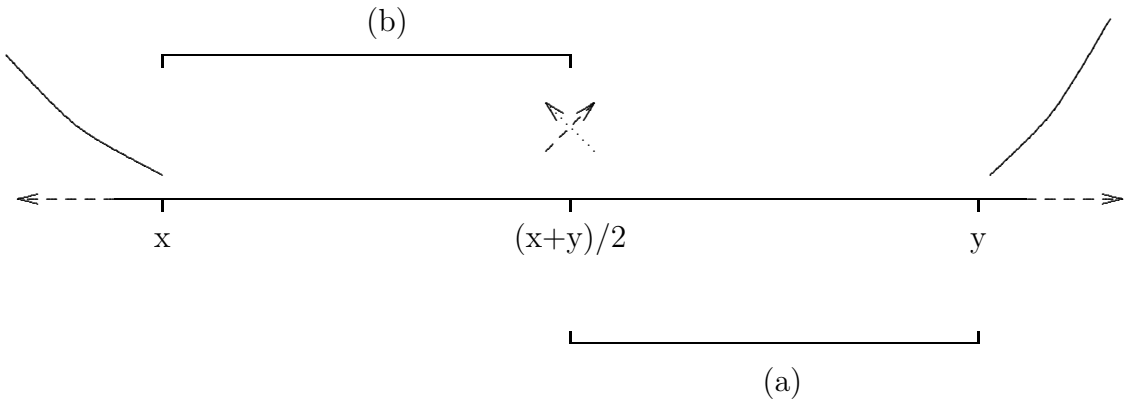

Figure 5: Case (a): $g'(\frac{x+y}{2}) < 0$ (dotted line) ; Case (b) $g'(\frac{x+y}{2}) > 0$ (dashed line)

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.

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 $[x, y]$, were 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 5) .

If g' is 0, then it is a maximal point (See Figure 6). 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.

The process is repeated k times to find all k roots. To multiply two numbers with $O(k)$ words each requires $O(k \log k)$ time, thus:

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

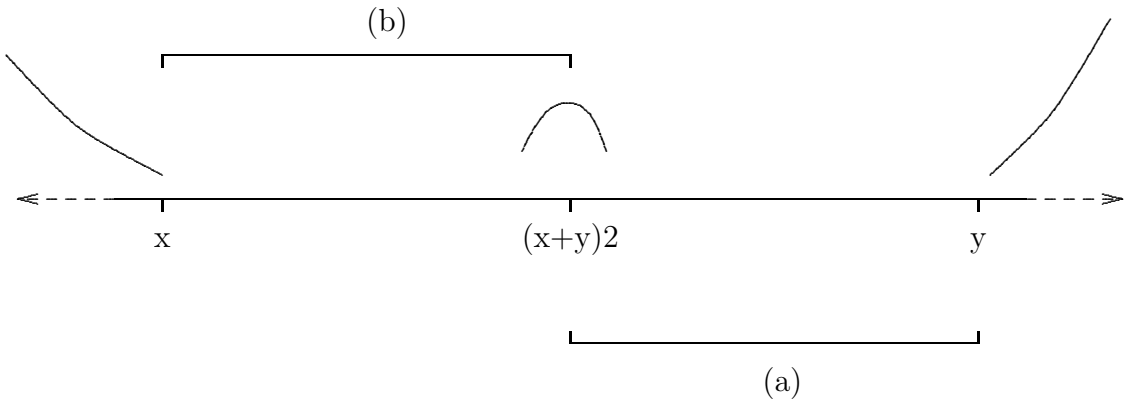

Figure 6: $g'(\frac{x+y}{2}) = 0$ so the search continues on either (a) or (b)

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. Aho and Corasick [AC75] solved the following problem. Given a set of strings, $R_1, R_2, \ldots, R_i$, and a text, find all occurrences of any of the strings in the text. They solved this problem by building a DFA-like structure as in [KMP77]. However, several of the nodes in the DFA correspond to "output nodes" each of which corresponds to an entry in an "output table". Each entry in the output table is a set of pointers to strings rather than a single string because any section of text which matches a string R_j would also match all the R_k's that are suffixes of R_j. For example, an entry that matches "she" would also match "he". Thus, once we have the output table built, we can think of each entry as a longest string and all of its suffixes within the set $R_1, R_2, \ldots, R_i$. The construction is linear in the sum of the lengths of the patterns. It is easy to see that the table takes no more storage than the strings themselves. Consider each string. For each string there will be one entry in the table for which that string is the longest. If the string is of length x it can have at most $x-1$ suffixes and so the entry in the table will be of at most size x pointers to strings. Thus the table is linear in the size of the strings. We will use the

output table to construct a forest partial order under the suffix relation for the rows of P_L.

Algorithm F

Input the rows of P_L $P_{L,1}, P_{L,2}, \ldots, P_{L,m}$

1. Associate with each string a field which contains its length
2. Build an [AC75] pattern matching machine from $P_{L,1}, P_{L,2}, \ldots, P_{L,m}$
3. Bin-sort the output table by size of the set of strings at each entry
4. Let $MSET$ be the size of the largest set in the output table
5. Make each element in the sets of size 1 a root and mark each node
6. For $i = 2$ to $MSET$ do
   For each set s of size i do
   Let l be the (unique) unmarked string in s
   Let m be the longest marked string in s
   Mark l and add l to the forest as a child of m
   end
   end

end Algorithm F

Clearly there will always be exactly one unmarked string in any given set and that string will be the longest. Thus we know that we must insert the string as a leaf in the tree as a child of the longest marked string.

Time: Steps (a)-(c) are linear in the size of the input. In steps (e) and (f), each set in the output table is scanned only once. Since the total size of the output table is no more than the size of the input, the total preprocessing time is $O(|P|)$.

A symmetric algorithm constructs the forest of the prefix relation of the rows of P_R (suffixes of the reversed rows).

6.2 Finding the Longest Matching Rows

Once again, we modify the algorithm of Aho and Corasik, [AC75] We will modify the 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 and T_L as described in the outline. For simplicity of notation

we assume that the text and pattern are given within square arrays (although clearly the algorithm can handle rectangles).

Algorithm G

Input $T[0:n-1;0:n-1]$, $P[0:m-1;0:m-1]$

1. Split pattern, P into its two parts, P_L and P_R.
2. Build an [AC75] DFA for P_L. Call it D_L with output table O_L.
3. Modify the output table to only output the longest string from each entry. Call the new table O'_L.
4. Create T_L as follows. Run D_L on the rows of T using the output table O'_L. If no string matches at $T[i;j]$ then $T_L[i;j] \leftarrow \$$, where \$ matches no string. If $P_{L,k}$ is the longest row that matches at $T[i;j]$ then $T_L[i;j] \leftarrow P_{L,k}$.

   [At every position (i,j) of T_L we now have the longest string which ends at $T[i;j]$]
5. Repeat the process symmetrically for P_R.

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

End Algorithm G

Time: $O(n^2)$

6.3 Finding the Mismatched Rows

Input to this step: T_L and T_R. Each element of these arrays 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_R and the "is-a-suffix-of" relation on the rows of P_L.

Processing: Viewing T_L and T_R in column major order, solve the smaller matching problem and choose only the locations where the total number of mismatches in both T_R and T_L 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 $O(k^3n^2 \log m \log k)$ for every alphabet symbol. There are $O(\sqrt{m})$ symbols so the total time is $O(k^3n^2\sqrt{m} \log m \log k)$. The brute force stage takes time $O(n^2\sqrt{m})$.

A better divide-and-conquer scheme:

Recall that the divide-and-conquer of section 3 divided the text into blocks of size $\sqrt{m}$. The treatment of elements within the blocks was by the brute force method, while the global treatment was by convolutions with an alphabet (the group representatives) of size $O(\sqrt{m})$. However, finding the error locations is much harder in the convolution stage than in the brute force stage, and this leads to the difference between their time complexities. For a more even distribution of labor, we change the block size.

New block size: $k\sqrt{m \log m}\sqrt{k \log k}$. The alphabet for the convolution stage is now of size $O(\frac{\sqrt{m}}{k\sqrt{\log m}\sqrt{k \log k}})$.

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

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. Can the smaller match problem be solved in linear time?

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 the numerical analysts we harassed: Howard Elman, Rod Fontecilla, Dianne O'Leary, and Pete Stewart, as well as Ed Edmundson and Larry Washington.

Appendix

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

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

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

References

[Abr87] K. Abrahamson. Genrealized string matching. *SIAM*, 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.

[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.

[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.

[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.

[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.