Multidimensional Pattern Matching: A Survey

Amihood Amir*

GIT–CC–92/29

July 1992

Abstract

We review some recent algorithms motivated by computer vision.

The problem inspiring this research is that of searching an aerial photograph for all appearances of some object.

The issues we discuss are local errors, scaling, compression and dictionary matching.

We review deterministic serial techniques that are used for multidimensional pattern matching and discuss their strengths and weaknesses.

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

*Paritally supported by NSF grant IRI-9013055.

1 Motivation

String Matching is one of the most widely studied problems in computer science [Gal85]. Part of its appeal is in its direct applicability to "real world" problems. The Knuth-Morris-Pratt [KMP77] algorithm is directly implemented in the *emacs* "`s`" and *UNIX* "`grep`" commands. The longest common subsequence dynamic programming algorithm [CKK72] is implemented in the *UNIX* "`diff`" command. The largest overlap heuristic for finding the shortest common superstring [Ukk90] is used in DNA sequencing. In this respect string matching is somewhat of an anomaly in theoretical computer science. Theoretical algorithms can not often be used as "off-the-shelf" solutions for practical problems.

We consider one of the important roles of theoretical computer science, that of providing an algorithmic theory for various application domains. Usually that process starts with abstracting the practical problem into several "pure form" combinatorial problems. It is hoped that understanding the solution of these problems will aid in an efficient solution of the original application. In this paper we review some of the algorithms that were motivated by two dimensional computer vision. Covering all aspects of the problem is clearly a mammoth undertaking. We concentrate on serial deterministic algorithms. The reader should be aware that there is also wide work on probabilistic as well as parallel approaches to the problem.

The main practical motivation for this survey is the problem of **searching aerial photographs**. The (ambitious) practical goal of this application is to input an aerial photograph and a template of some object (a pattern). The output is all locations on the aerial photograph where the template object appears. In reality we need to grapple with several problems:

1. Local errors. These may arise from atmospheric distortions, transmission noise, level of detail, the digitization process, or occlusion of pattern parts by other objects.

2. Scaling. The size of the object in the input aerial photo may differ from that in the template.

3. Rotation. The object may be facing different directions in the aerial photograph and in the template.

Further issues of interest are:

Compressed Matching: Digitized images are known to be extremely space consuming. However, regularities in the images can be exploited to reduce the necessary storage area. Thus we find that many systems store images in a compressed form (e.g. quadtrees [Sam90]). If searching for appearances of a pattern in an image can be done *without decompressing* then compression becomes a **time saving** tool in addition to its being a space saving device.

Dictionary Matching: The aerial photograph model we described is by no means the only possible vision paradigm. While it may be important to find an object in a given image, biological vision is somewhat an "inverse" of that process. Rather than searching for a small template in a large image, we have in our minds a tremendous database of objects we have seen (or imagined). When presented with an image we recognize it with amazing speed. Thus it is interesting to come up with efficient algorithms for quickly recognizing objects from a preprocessed dictionary, in a given image.

For ease of perusal we enclose a table of contents for this paper:

2 Exact Matching

The most restrictive possible abstraction of the aerial photograph problem is that of seeking an *exact matching* of the pattern in the image, where both pattern and image are *rectangles*. Throughout this paper we define our problems in terms of *squares* rather than *rectangles*. In almost all cases the reason is simply for convenience of notation, and the results directly generalize. We explicitly mention those results that only apply to squares.

The *Exact Two Dimensional Matching Problem* is defined as follows: Let Σ be an alphabet. *INPUT:* *Text* array $T[n \times n]$ and *pattern* array $P[m \times m]$. *OUTPUT:* All locations $[i, j]$ in T where there is an *occurrence* of the pattern, i.e. $T[i+k, j+l] = P[k+1, l+1] \quad 0 \leq k, l \leq n-1$.

2.1 Linear Reductions

A natural way of solving any generalized problem is by reducing it to a special case whose solution is known. It is not surprising that most solutions to the two dimensional exact matching problem use exact string matching algorithms in one way or another.

The Knuth-Morris-Pratt [KMP77] algorithm basically follows the idea of matching the text and pattern character by character until a mismatch occurs. Then the pattern is slid forward for the greatest overlap with the old pattern position, and the comparison resumes from there. This idea can be generalized in the following way:

Starting from the leftmost column and moving to the right, proceed down the columns and compare a *pattern row* with a *length-m text subrow* starting at the scanned text location. Proceed in this fashion until a mismatch occurs. Upon a mismatch, slide the pattern *down* for the greatest overlap with the old pattern position and resume comparisons from there.

The idea is obvious, but its straightforward implementation would take time $O(n^2 m)$, since this is a basic KMP on the text, but every comparison takes time $O(m)$. It is an improvement over the naive $O(n^2 m^2)$ algorithm, but not good enough. What we need is a method for *quick* comparison of a pattern row and a length-m text subrow. Two solutions are possible.

2.1.1 Automata Methods

Bird [Bir77] and, independently, Baker [Bak78] proposed the following solution. Preproces the text and identify all occurrences of all pattern rows. Represent each different row by a new symbol and place this symbol at the text location where the row occurs. The problem is now exactly that of string matching, where we are seeking all occurrences of the string composed of the new symbols in the order that their respective rows appear in the pattern. The string matching part can be done in time $O(n^2)$. The only question is how to efficiently identify the occurrences of all pattern rows.

This was done by using the Aho and Corasick [AC75] dictionary matching algorithm. The *Dictionary Matching Problem* is the following: Preprocess a *dictionary* of patterns $\{P_1 = p_{11} \cdots p_{1m_1}, P_2 = p_{21} \cdots p_{2m_2}, ..., P_k = p_{k1} \cdots p_{km_k}\}$. Subsequently, for every *INPUT:* Text $T = t_1 \cdots t_n$. *OUTPUT:* All locations in the text where there is a match with any dictionary pattern.

Aho and Corasick preprocess the dictionary in time $O(\sum_{i=1}^{k} m_i \log|\Sigma|)$ and subsequently search an input text in time $O(n \log|\Sigma| + occ)$, where occ = number of patterns that occur in the text. If all patterns are of the same length then only a single pattern can end at any text location. The time then becomes $O(n \log|\Sigma|)$.

Returning to two dimensional matching. We may view each distinct pattern row as a separate pattern in a dictionary. The result is a dictionary matching problem where all dictionary patterns have the same length. Thus the Bird-Baker solution is the following:

1. Find all occurrences of all pattern rows in the text. Mark the end of each distinct row's occurrence with a new special symbol.
2. Scan the text down the columns, from left to right. Run the Knuth-Morris-Pratt (KMP) algorithm searching for the *string* composed of the new symbols representing the distinct pattern rows.

Time: 1. Using Aho and Corasick, $O(n^2 \log|\Sigma|)$. 2. $O(n^2)$.
Total Time: $O(n^2 \log|\Sigma|)$.

2.1.2 Suffix Tree Methods

Recall that our aim is to use the KMP algorithm for solving the exact two dimensional matching problem. What we seek is a method for constant time comparison of pattern rows with length-m text subrows (and with each other). Bird and Baker solved this problem by performing the comparisons in advance. In this section we will see a method where this comparison can be done while scanning the text.

Definition: Let $S = s_1 \cdots s_n$ be a string. A *suffix tree* of S is a trie of all suffixes of S (i.e. $\{s_n,\ s_{n-1}s_n,\ s_{n-2}s_{n-1}s_n,\ \ldots,\ s_2s_3\cdots s_n,\ s_1s_2\cdots s_n\}$) where every path of single outdegree node is compressed to a single node.

Weiner [Wei73] and McCreight [McC76] showed different methods of constructing the suffix tree of $S = s_1 \cdots s_n$ in time $O(n \log|\Sigma|)$. The importance of suffix trees for our purpose is that it has the following properties: 1) The leaves represent exactly the suffixes of S, and 2) The *lowest common ancestor* (LCA) of any two nodes is the *longest common prefix* of the strings they represent.

We can thus make the following observation [ALV90]. Let S be a string composed of the concatenation of all text rows followed by all pattern rows. The following queries are equivalent:

1. Pattern row P_{i_0} equals text subrow $T_{i_1 j}$, where $j_0 \le j \le j_0 + k - 1$.
2. The length of the longest common prefix of the suffixes of S starting at P_{i_0} and $T_{i_1 j_1}$ is at least m.
3. The length of the LCA in S's suffix tree of the nodes representing the suffixes that start at P_{i_0} and $T_{i_1 j_1}$ is at least m.

Using Weiner's or McCreight's algorithm, the suffix tree for the concatenation of the text and pattern rows can be constructed in time $O(n^2 \log|\Sigma|)$. All we need now is a method for finding the lowest common ancestor of two nodes in a tree in constant time.

It was pointed out by Landau and Vishkin [LV89], that the Harel and Tarjan [HT84] algorithm does precisely that. Harel and Tarjan showed that given any n-node tree, one can preprocess the tree in time $O(n)$ in a manner that allows subsequent LCA queries in time $O(1)$.

We now have all the components for a different exact two dimensional matching algorithm [ALV90].

1. Construct the text and pattern suffix tree and preprocess for LCA.

2. Scan the text down the columns, from left to right. Run the KMP algorithm modified in a way that symbol comparison is replaced by checking if the LCA length is at least m.

Time: 1. $O(n^2 \log|\Sigma|)$ using Weiner (or McCreight) and Harel-Tarjan. 2. $O(n^2)$.
Total Time: $O(n^2 \log|\Sigma|)$.

2.2 Convolutions

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.

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

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

Exact string matching with "don't cares": Again $\otimes$ is $=$ but with the following slightly different definition:

$$a = b \ =^{def} \begin{cases} 1, & \text{if } a = b \text{ or } a = \phi \text{ or } b = \phi; \\ 0, & \text{otherwise.} \end{cases}$$

$\oplus$ is $\wedge$ and the pattern is transposed.

In [FP74] it was proven that exact string matching with don't cares can be solved using convolutions in the same way as exact string matching. The time is then $O(\log|\Sigma| n \log m)$.

We can use the string matching with don't care problem to solve the two dimensional matching problem as follows (actually the same idea can be used for d-dimensional matching [AL91]).

Without loss of generality, we may assume that the text is of size $2m \times 2m$. This can be achieved by dividing the text into four overlapping grids of $2m \times 2m$ matrices. In the following discussion we assume,then, that $n = 2m$.

Let $T_{n\times n}$ be the text matrix, $P_{m\times m}$ be the pattern matrix. Let $L_{[1:n^2]}$ be the linear representation of T in row major order, and let $M_{[1:(m-1)n+m]}$ be the vector:

$$M_i = \begin{cases} P_{1,i} & \text{for } i = 1 \text{ to } m \\ \phi & \text{for } i = m+1 \text{ to } n \\ P_{2,i-n} & \text{for } i = n+1 \text{ to } m+n \\ \phi & \text{for } i = n+m+1 \text{ to } 2n \\ . & \\ . & \\ . & \\ P_{m,i-(m-1)n} & \text{for } i = (m-1)n+1 \text{ to } (m-1)n+m \end{cases}$$

Matching M in L (and taking care of boundary conditions) is equivalent to matching P in T. What is actually being done is padding the pattern with wildcards up to the size of the text dimension. The boundary condition can then be handled on line.

Time: The reduction is to string matching with don't cares with text of size n^2 and pattern of size $O(mn)$ (however, the maximum number of possible mismatches is still m^2 per location). This is solved by the convolutions method in time $O(|\log\Sigma|n^2\log m)$, but since $n = 2m$ the time is $O(|\log\Sigma|m^2\log m)$. For a general n the time is $O(|\log\Sigma|n^2\log m)$.

2.3 Periodicity Analysis

All the previously discussed two dimensional matching algorithms are reductions of the problem into one dimension. These reductions all cost at least an additional $\log|\Sigma|$ factor. Recently, a uniquely two dimensional approach to pattern matching was introduced [?]. This technique analyses the two dimensional structure of the pattern and makes use of it in the text scanning step. We will see that this allows an alphabet independent text scanning. This technique will also prove useful in compressed matching, as will be seen in section 6. We feel that periodicity is an important tool in multidimensional matching and that full understanding of this idea will offer efficient algorithms for many multidimensional matching problems.

The two-dimensional periodicity idea: A periodic pattern may contain locations, other than the origin, where the pattern can be superimposed on itself without mismatch. Suppose our pattern is *non periodic*, i.e. there are no such locations, other than the origin. We could then narrow down the number of *potential candidates* for a pattern appearance in the text in a fashion that insures that all such candidates are "sufficiently far" from each other. Verification of a candidate could then be done in the naive character-by-character comparison, but the time would still be linear because the candidates do not overlap.

In the next sections we will look more closely into two dimensional periodicity and its application to exact matching.

2.3.1 Two Dimensional Periodicity

In a periodic string, a smallest period can be found whose concatenation generates the entire string. In two dimensions, if an array were to extend infinitely so as to cover the plane, the one-dimensional notion of a period could be generalized to a unit cell of a lattice. But, a rectangular array is not infinite and may cut a unit cell in many different ways at its edges.

Instead of defining two-dimensional periodicity on the basis of some subunit of the array, Amir and Benson [?] use the idea of *self-overlap*. This idea applies also to strings. A string w is periodic if the longest prefix p of w that is also a suffix of w is at least half the length of w. For example, if $w = abcabcabcab$, then $p = abcabcab$ and since it is over half as long as w, w is periodic. This definition implies that w may overlap itself starting in the fourth position.

The preceding idea easily generalizes to two-dimensions as illustrated by the following preliminary definitions. Let A be an array. A *prefix* of A is a rectangular subarray that contains one corner element of A. A *suffix* is a rectangular subarray that contains the diagonally opposite corner. A is *periodic* if the largest prefix that is also a suffix has dimensions greater than half those of A. Again, this implies that A may overlap itself if the prefix of one copy of A is aligned with the suffix of a second copy of A.

Notice that the choice of the corner in which to put the prefix is arbitrary. Because of the symmetry, the prefix may be assigned to either the upper left or lower left corners of A. This clearly gives us two directions in which A can be periodic. Following [?] we classify the type of periodicity of A based on whether it has periodicity in either or both of these directions. To simplify the discussion, we describe square arrays. The results can be extended to all rectangular arrays (see [?]).

We begin with some formal definitions of two-dimensional periodicity and related concepts. Let $A[0 \ldots m-1, 0 \ldots m-1]$ be an $m \times m$ square array. Each element of A contains a symbol from an alphabet Σ. We can divide the array into four *quadrants*, labeled in a counterclockwise direction from upper left, quadrants *I,II,III*, and *IV*. Given two copies of A, one directly on top of the other. The two copies are said to be *in register* because some (in this case all) of the elements overlap, and overlapping elements contain the same symbol. If we can slide the upper copy over the lower copy to a point where the copies are again in register, then at least one of the corner elements of the upper array will overlap an element of the lower array. The element in the lower copy that is under this corner is the *source*. We say that the array is *quadrant I symmetric* if an overlapping corner is element $A(0,0)$. The element in the lower copy is a *quadrant I source*. Quadrants II, III and IV symmetry and sources are similarly defined.

Let the array be quadrant I symmetric and let the upper and lower copies be in register when element $A(0,0)$ overlaps element $A(r,c)$, the source. Then there exists a *quadrant I symmetry vector* $\vec{v}_I = r\vec{y} + c\vec{x}$ where $\vec{x}$ is the horizontal unit vector in the direction of increasing column index and $\vec{y}$ is the vertical unit vector in the direction of increasing row index. If the array is quadrant II symmetric, then the upper and lower copies are in register when $A(m-1,0)$ overlaps $A(r,c)$. The *quadrant II symmetry vector* is $v_{II} = (r-(m-1))\vec{y} + c\vec{x}$. Note that the coefficient on $\vec{y}$ is negative for quadrant II. The quadrants III and IV symmetry vectors are defined similarly.

The *length* of a symmetry vector is the maximum of the absolute values of its coefficients. If the length of a symmetry vector is $< \frac{m}{2}$, then the vector is *periodic*.

For the classification scheme, we need to pick the shortest symmetry vector for each of quadrants I and II. But, there may be several shortest vectors in a given quadrant. Also, the same orthogonal vector may be shortest in both quadrants. Let B_I be the set of shortest non-vertical vectors in quadrant I and let B_{II} be the set of shortest non-horizontal vectors in quadrant II. The *basis vectors for array* A are vector $\vec{v}_1 \in B_I$ (if any) with smallest r value and the vector $\vec{v}_2 \in B_{II}$ (if any) with smallest c value. In other words, $\vec{v}_1$ is the closest to horizontal in B_I and $\vec{v}_2$ is the closest to vertical in B_{II}.

The four categories of two-dimensional periodicity are :

- **Non-periodic**—A has no periodic vectors.
- **Lattice periodic**—Both quadrants I and II of A have a periodic basis vector. All quadrant I sources which occur in quadrant I fall on the nodes of a lattice which is defined by these vectors. The same is true for quadrant II sources in quadrant II. Specifically, let $\vec{v}_1$ and $\vec{v}_2$ be the periodic basis vectors in quadrants I and II respectively. Then, for all integers i,j such that $(0,0) + i\vec{v}_1 + j\vec{v}_2$ is an element of quadrant I, that element is a quadrant I source and no other

elements in quadrant I are quadrant I sources. Similarly, for all $\hat{\imath}, \hat{\jmath}$ such that $(m-1, 0) + \hat{\imath}\vec{v}_1 + \hat{\jmath}\vec{v}_2$ is an element of quadrant II, that element is a quadrant II source and no others.

- **Line periodic**—One quadrant of A has a periodic vector and one does not. The sources in the quadrant with the periodic vector all fall on one line. Specifically, if quadrant I is the quadrant with the periodic basis vector $\vec{v}_1$, then for all i such that $(0, 0) + i\vec{v}_1$ is an element in quadrant I, that element is a quadrant I source and no others.

- **Radiant periodic**—This category is identical to the line periodic category, except that in the quadrant with the periodic vector, the sources fall on several lines which all radiate from the quadrant's corner. We do not describe the exact location of the sources because these depend on the specific array, except we note that none is a linear combination of both basis vectors for the array.

Having categorized the different classes of periodicity, we need to find out, for any given matrix, what is its periodicity type. We can actually do more than that. We construct two WITNESS arrays. For each location $A[i, j]$, TOP-WITNESS$[i, j] = (m, m)$ if A is in register with itself when element $A[0, 0]$ overlaps element $A[i, j]$. Otherwise, TOP-WITNESS$[i, j] = (r, c)$ where (r, c) identifies some mismatch. Specifically $A[r, c] \neq A[i + r, j + c]$. BOTTOM-WITNESS$[i, j]$ is filled out similarly except element $A[m - 1, 0]$ overlaps element $A[i, j]$.

Using suffix trees with constant time LCA queries, and using some KMP variations described by Main and Lorentz [ML84], it is possible to construct the WITNESS arrays in time $O(m^2 \log |\Sigma|)$ (see [?] for details).

2.3.2 Alphabet Independent Search

The WITNESS tables for the pattern provide some important understanding of its two dimensional structure. This knowledge can be exploited [ABF94] to provide an alphabet independent $O(n^2)$ text scanning phase.

Text processing is accomplished in two stages: Candidate Consistency and Candidate verification. A *candidate* is a location in the text where the pattern may occur. We denote a candidate starting at text location $T[r, c]$ by (r, c). We say that two candidates (r, c) and (x, y) are *consistent* if they expect the same text characters in their region of overlap (two candidates with no overlap are trivially consistent).

Initially, we have no information about the text and therefore all text locations are candidates. However, not all text locations are consistent. During the candidate consistency phase, we eliminate candidates until all remaining candidates are pairwise consistent. During the candidate verification phase, we check the candidates against the text to see which candidates represent actual occurrences of patterns. We exploit the consistency of the surviving candidates to rule out large sets of candidates with single text comparisons (since all consistent candidates expect the same text character).

Candidate Consistency: This is done with two sweeps of the text. The first eliminates inconsistent candidates within each column, and the second eliminates all inconsistent candidates in the text. The result of these two sweeps are potential sources, none of which can conflict with any other. This means that if we verify that one of these potential sources is indeed the source of a pattern occurrence then all potential sources *within* the verified area are *guaranteed* to overlap the verified area. Thus, verification need not ever backtrack. The details of the $O(n^2)$ candidate consistency algorithms can be found in [ABF94].

Candidate Verification: As mentioned above, we are guaranteed that all consistent candidate sources overlap consistently with the pattern. We only need to verify that they are indeed pattern sources. This can be done in linear time by the time-tested sport cheer - *the wave*.

The idea of the wave is for each element to jump up and wave a pair $\langle i, j\rangle$ whose meaning is that this element has to be tested against $P[i, j]$. There may be several such options for some locations, but any will do because the candidate sources are now all compatible. The waved pair, in addition to knowledge of candidate sources, causes the element immediately below the waving location to wave its own pair. When all column waves are done we do row waves and every text element now needs a single comparison. For further details on the wave, see [ABF94].

3 Approximate Matching of Rectangular Arrays

One possible string matching generalization that has been researched is *approximate string matching* - finding all occurrences of a pattern in a text where *differences* are allowed.

Three types of *differences* were distinguished [Lev66]:

1. A pattern character corresponds to a different character in the text (*mismatch*).

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

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

Two problems were considered in the one dimensional case: The *string matching with k mismatches problem (the k mismatches problem)* - find all occurrences of the pattern in the text with at most k type-1 differences. The *string matching with k differences problem (the k differences problem)* - find all occurrences of the pattern in the text with at most k differences of type 1. 2. or 3.

Abrahamson [Abr87] and, independently Kosaraju [Kos87] used a divide-and-conquer approach in conjunction with the convolutions method to solve the string mismatches problem in time $O(n\sqrt{m}\log m)$. Their algorithm writes, for every text location, the number of mismatches that will occur it the pattern is compared with the text starting at that location.

Landau and Vishkin [LV89] gave a $O(nk)$ algorithm for the k-differences problem.

In this section we consider approximate pattern matching where both text and pattern are rectangles.

INPUT: Text array $T[n \times n]$ and $P[m \times m]$ where all elements of P and T are in alphabet Σ, and integer k. *OUTPUT:* All occurrences of the pattern in the text with at most k differences.

The definition of insertion and deletion in multidimensions need clarification. The effect of insertion and deletion may be different depending on the implementation. We illustrate with a two dimensional example. If a matrix is transmitted serially a deleted character means an appropriate shift of the entire array. However, it may be the case that the array is transmitted column by column with an EOD indication between them. In that case, a deletion or insertion affects only the column it appears in. Following Krithivasan and Sitalakshmi [KS87] and Amir and Landau [AL91] we assume the latter situation. It is clear that the case where a deletion or insertion affects only the row it appears in can be handled in a similar manner.

Krithivasan and Sitalakshmi [KS87] solved this problem in time $O(n^2mk)$. This was improved by Amir and Landau to $O(n^2k^2)$ ($O(n^2k)$ if only mismatch errors are allowed).

The idea was using dynamic programming to handle the insertion and deletion problems, and suffix trees to identify runs of matching pattern and text substrings.

4 Approximate Matching of Non Rectangular Patterns

The approximate two dimensional problem we saw in section 3 was defined with both the pattern and text being rectangular matrices. In reality, it is usually necessary to match *non-rectangular* shapes. The techniques presented in section 3 seem inadequate in dealing with nonrectangular arrays.

4.1 Mismatches

In section 2.2 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 for *any shape* in time $O(|\Sigma|n^2 \log m)$. We simply pad the matrix appropriately so only the given shape is matched.

The $|\Sigma|$ factor in the complexity results from the fact that we need to do $|\Sigma|$ convolutions. In each one we count the number of pattern mismatches for a different alphabet symbol ([AL91]).

This method is clearly efficient for bounded small alphabets. For unbounded alphabets we may use the Abrahamson-Kosaraju divide-and-conquer technique to achieve time $O(n^2\sqrt{m}\log m)$.

4.2 Mismatches, Insertions and Deletions

Pattern matching provides many examples of powerful techniques that solves various different problems. However, when some criteria are combined, there is no ready solution. For example, convolutions solve the "don't care" problem, and the mismatches problem, but can not be used when insertions and deletions are introduced. Automata methods or suffix trees work mainly for exact matching. But if presented with the problem of matching with differences **and** don't cares then there is no known efficient method that can solve it.

Amir and Farach [AF91b] made the first advance in the direction of efficiently solving the k-difference matching problem for non-rectangular patterns. A novel method was used, that combines the power of convolutions, dynamic programming and subword trees. It proved effective in 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.

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.

This algorithm is efficient for any pattern that can be split into a "small" number of half-rectangular shapes. An example is any convex shape in an orientation where the longest diameter is vertical. We are searching for all locations where a half-rectangular pattern matches the text allowing no more than k mismatches, insertions (in rows) and deletions (in rows) errors.

To achieve this result some new tools were needed. Efficient solutions to two problems were provided. 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 element.

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. Both these problems can be solved in time $O(n\sqrt{m}\log m)$.

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. These locations can be found in time $O(k^3 n \log m \log k)$ [AF91b].

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 $P_{i_j} = T_{l+i_j} = 1$.

5 Scaled Matching

All the problems we have seen so far were useful mainly in solving matching with "local errors" prolems. We mentioned that in reality we may be interested in matching patterns whose occurrence in the text is of different scale than provided by the pattern. For example, reading a newspaper one encounters letters of the alphabet in various sizes.

A "clean" version of the problem may be defined as follows [ALV90]:

The string $aa...a$ where the symbol a is repeated k times (to be denoted a^k), is referred to as *a scaled to k*. Similarly, consider a string $A = a_1 \cdots a_l$. *A scaled to k* (A^k) is the string $a_1^k,...,a_l^k$.

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

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

The problem of *two-dimensional pattern matching with scaling* is defined as follows: *INPUT:* Pattern matrix $P[i,j] \quad i = 1,...m; j = 1,...,m$ and Text matrix $T[i,j] \quad i = 1,...,n; j = 1,...,n$ where $n > m$. *OUTPUT:* all locations in T where an occurrence of P scaled to k starts, for any $k = 1,...,\lfloor \frac{n}{m} \rfloor$.

The basic algorithmic design strategy of Amir-Landau-Vishkin [ALV90] can be viewed as realizing the following approach: For each scale k, try to select only a fraction of $\frac{1}{k}$ among the n columns and seek k-occurrences only in these columns. Since each selected column intersects n rows, this leads to consideration of $O(\frac{n^2}{k})$ elements. Summing over all scales, we get $O(n^2)$ multiplied by the harmonic sum $\sum_{i=1}^{\frac{n}{m}} \frac{1}{i}$, whose limit is $\log \frac{n}{m}$ making the total number of elements scanned $O(n^2 \log \frac{n}{m})$.

A final intuitive step was to select also a $\frac{1}{k}$ fraction of the rows. Since $\sum_{i=1}^{\frac{n}{m}} \frac{1}{i^2}$ is bounded by a constant, the number of elements decreases now to

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

6 Compressed Matching

The two most common motivations for the study of data compression are:

- **Data storage:** Data compression can allow more data to be stored on a given device.
- **Data Communications:** Communication links, be they cables, phone lines or satellite channels, are both limited and, usually, slower than the data gathering and data processing ends. Compresing the data prior to transmission, and decompressing on the receiving end can increase the speed (and thus volume) of the transmission process.

The main thrust in the study of data compression has been in achieving compression that is efficient in the success it has in packing while also being practical in time and space. This is a very reasonable direction, especially in light of the relatively faster technology of data processing compared to data storage or transmission. We are interested in a new paradigm of this problem. We would like the compression to also have the property that allows **pattern matching** in the **compressed data**. We see a realistic motivation for this property that arises from both the data storage and data communication realms.

It has been noted that in reality digitized images are space consuming, on the one hand, but also very "compressible". If a pattern can be found without the need to decompress the data, noticeable savings in time will arise. On the communications arena, systems of small networked processors could enjoy a boost in efficiency if data could be both **transmitted** and **analysed** in a compressed form, especially if the process requires transmission and analysis of the same data in many nodes. It is with this in mind that the compressed matching model was defined [?].

The *compressed pattern matching problem* is the following: Let c be a given compression algorithm, let $c(D)$ be the result of c compressing data D. *INPUT:* Compressed text $c(T)$ and compressed pattern $c(P)$. *OUTPUT:* All locations in T where pattern P occurs.

A compressed matching algorithm is *optimal* if its time complexity is $O(|c(T)|)$. It is *efficient* if for T such that $|c(T)| << |T|$ its time complexity is $o(|T|)$.

The above definition of *efficient* may seem somewhat strange. It is motivated by the following observation. The size of the compressed data will vary depending on the internal structure of the data. In the worst case it may be $\Omega(|T|)$. Any algorithm that is not optimal would then be worse than decompression and then matching. However, the concept of compression is implicitely assuming that compression is beneficial, *i.e.* $|c(D)| << |D|$. This leads to our definition of efficient compressed matching. In particular, we consider algorithms with complexity $O(|c(T)|polylog(|c(T)|)$ to be efficient. We call such algorithms *almost optimal*.

In one dimension (strings), an optimal compressed matching algorithm is readily apparent for the run-length compression [ALV90]. The run-length compression is defined as follows:

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

Again in [ALV90], a simple two-dimensional generalization of the run-length representation was considered, namely, the concatenation of the run-length compression of all the matrix rows (or columns). For simplicity's sake we will refer to this compression as the *two-dimensional run-length compression*. In [ALV90], an efficient compressed matching algorithm was presented for the two-dimensional run-length compression. However, the time complexity of that algorithm is $\Omega(|T|/\sqrt{|P|})$. For a large but very compressible T, and a small P, this result, while efficient, is not very attractive.

A $O(|c(T)|\log|P| + |P|)$ time algorithm for the two-dimensional run-length compression is shown in [?]. The main tools for that algorithm are the periodicity analysis of the pattern, coupled with efficient candidate consistency and verification in the compressed text (see section 2.3.1).

7 Dictionary Matching

We have remarked that dictionary matching is a natural and important model of vision. In reality, we would like to extend all the results we discussed and apply them to two dimensions. Unfortunately, this seems very hard. In this section we will see a solution for *exact dictionary matching* where all the dictionary patterns are **squares**. There is currently no known efficient method for dictionary matching where the dictionary patterns are **rectangles**.

Definition: The *Two-dimensional Dictionary matching problem* is defined as follows:

INPUT: Text matrix $T[1,...,n;1,...,n]$ and pattern matrices $P_1[1,...,m;1,...,m]$, $P_2[1,...,m;1,...,m]$, $\ldots$, $P_k[1,...,m;1,...,m]$.

OUTPUT: At each location $[x,y]$ in T the i, $1 \leq i \leq k$ such that $P_i[l,1]\cdots P_i[l,m]$ appears in $T[x+l-1,y]\cdots T[x+l-1,y+m-1]$ for $l = 1,...,m$ (and a 0 at $[x,y]$ if no such i exists). In other words, at each location of the text, we would like to know which pattern matches exactly starting at that position of the text.

This problem cam be solved by extending the automaton techniques used for two dimensional exact matching of a single pattern that were described in section 2.1.

Definition: The *Two-dimensional Dictionary matching with different size patterns problem* is defined as follows:

INPUT: Text matrix $T[1,...,n;1,...,n]$ and pattern matrices $P_1[1,...,m_1;1,...,m_1]$, $P_2[1,...,m_2;1,...,m_2]$, $\ldots$, $P_k[1,...,m_k;1,...,m_k]$.

OUTPUT: At each location $[x,y]$ in T the i's, $1 \leq i \leq k$ such that $P_i[l,1]\cdots P_i[l,m_i]$ appears in $T[x+l-1,y]\cdots T[x+l-1,y+m_i-1]$ for $l=1,...,m_i$ (and a 0 at $[x,y]$ if no such i exists). In other words, at each location of the text, we would like to know which patterns match exactly starting at that position of the text. An equivalent problem is to output the i at each location with the maximum m_i over all those P_i's that match. Then any prefix of P_i would also match.

This problem is solved by linearization [AF91b]. We view each pattern matrix as a string. Each symbol of this string is a character of the **diagonal** together with the subrow from this diagonal character to the left edge of the pattern , and the subcolumn from this diagonal symbol to the top of the pattern.

We can construct an Aho-Corasick automaton for these strings in a manner similar to the KMP construction described in section 2.1. The dictionary matching problem can then be solved with the following time complexity: Preprocessing time $O(A\log A)$ and text processing time $O(n^2\log A)$ where $A=\sum_{i=1}^{k} m_j^2$. In words, A is the total area of all the patterns.

Note that this algorithm is not adaptive. Introduction of a new dictionary pattern requires reprocessing of the entire dictionary. This deficiency arises from the fact that the Aho-Corasick one dimensional dictionary matching algorithm is not adaptive. See [AF91a] for an adaptive one dimensional dictionary matching algorithm.

8 Conclusions and Open Problems

We have scanned some of the recent results in two dimensional matching. Clearly, we are far from our motivation of actually finding a given template in an image. A major problem is that of rotation. There is still no efficient algorithm for finding all rotated occurrences of a pattern in a text even if we restrict our alphabet to $\{0,1\}$.

However, even solving the rotation problem will only be a beginning. What we will then have is various techniques for solving different subproblems, but we need one method to solve them all. We need a scaled-rotated-approximate-dictionary matching of nonrectangular patterns and that seems a great challenge indeed.

There are many technical open problems that were left in the wake of the results described in this survey, such as an adaptive two dimensional dictionary matching algorithm, or a dictionary matching algorithm for rectangular patterns. Other problems are faster methods for general convolutions, multi-dimensional extensions that are dimension-independent, dictionary matching with "don't cares". But the grand inspiration continues to be integration of these solutions to a general matching algorithm.

We may never reach that goal, but the way sure is exciting...

References

[AB92] A. Amir and G. Benson. Two-dimensional periodicity and its application. *Proc. of 3rd Symposium on Discrete Algorithms, Orlando, FL*, pages 440–452, Jan 1992.

[ABF94] A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two dimensional pattern matching. *SIAM J. Comp.*, 23(2):313–323, 1994.

[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. *Comm. ACM*, 18(6):333–340, 1975.

[AF91a] A. Amir and M. Farach. Adaptive dictionary matching. *Proc. 32nd IEEE FOCS*, pages 760–766, 1991.

[AF91b] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. *Proc. of 2nd Symposium on Discrete Algorithms, San Francisco, CA*, pages 212–223, 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*, pages 344–357, 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.

[CKK72] V. Chvatal, D.A. Klarner, and D.E. Knuth. Selected combinatorial research problems. Technical Report STAN-CS-72-292, Stanford University, 1972.

[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.

[Gal85] Z. Galil. Open problems in stringology. In Z. Galil A. Apostolico, editor, *Combinatorial Algorithms on Words*, volume 12, pages 1–8. NATO ASI Series F, 1985.

[HT84] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestor. *Computer and System Science*, 13:338–355, 1984.

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

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

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

[Lev66] V. I. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.

[LV89] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[ML84] M.G. Main and R.J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. of Algorithms*, pages 422–432, 1984.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[Ukk90] E. Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5:313–323, 1990.

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