# Theoretical Issues of Searching Aerial Photographs: A Bird's Eye View*

Amihood Amir

Bar-Ilan University and Georgia Tech
Department of Computer Science
52900 Ramat-Gan
ISRAEL

e-mail: amir@cs.biu.ac.il

**Abstract.** We review some pattern matching algorithms and techniques motivated by the discrete theory of image processing.

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

The issues we discuss are digitization, local errors, rotation and scaling.

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

**Keywords:** multidimensional pattern matching, local errors, rotations, scaling, digitization, discrete image processing.

## 1 Motivation

String Matching is one of the most widely studied problems in computer science [35]. Part of its appeal is in its direct applicability to "real world" problems. Some variation of the Boyer-Moore [21] algorithm is directly implemented in the search command of practically all text editors. The longest common subsequence dynamic programming algorithm [22] is implemented in the *UNIX* "`diff`" command. The largest overlap heuristic for finding the shortest common superstring [46] 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 and techniques that were motivated by image processing. Covering all aspects of the problem is clearly a mammoth undertaking. We concentrate on serial

*Partially supported by NSF grant CCR-01-04494 and ISF grant 82/01.

deterministic algorithms. The reader should be aware that there is also a wide body of work on probabilistic, randomized, and 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.

There are other issues of interest in two dimensional matching. Among them 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. jpeg). 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. We will not delve into compression issues in this paper, although many of the techniques mentioned here have been used in compressed matching algorithms.

**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. As in the case of compressed matching, we will not have the opportunity to say much of this exciting area of research, although here too many algorithms use techniques that will be addressed herein.

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

1. Motivation
2. Exact Two Dimensional Matching
    2.1 Linear Reductions
        2.1.1 Automata Methods
        2.1.2 Suffix Tree Methods
    2.2 Convolutions
    2.3 Periodicity Analysis

2.3.1 Two Dimensional Periodicity
2.3.2 The Dueling Method
3. Approximate Matching of Rectangular Patterns
4. Approximate Matching of Nonrectangular Patterns
4.1 Mismatches
4.2 Mismatches, Insertions and Deletions
5. Scaled Matching
6. The Geometric Model
6.1 Scaling
6.2 Rotation
7. Conclusions and Open Problems

# 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 $\Sigma$ 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. $T[i + k, j + l] = P[k + 1, l + 1] \;\; 0 \leq k, l \leq n - 1$.

## 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 the early solutions to the two dimensional exact matching problem use exact string matching algorithms in one way or another.

The Knuth-Morris-Pratt [39] 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 $O(n^2m)$, since this is a basic KMP on the text, but every comparison takes time $O(m)$. It is

an improvement over the naive $O(n^2m^2)$ algorithm, but not good enough. What is needed 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 [20] and, independently, Baker [18] proposed the following solution. Preprocess 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 [2] 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. $\{s_n,\ s_{n-1}s_n,\ s_{n-2}s_{n-1}s_n,\ ...,\ s_2s_3 \cdots s_n,\ s_1s_2 \cdots s_n\}$) where every path of single outdegree node is compressed to a single node.

Many different methods for constructing suffix trees and suffix arrays have been developed [49, 44, 26, 47]. The importance of suffix trees for our purpose is that they 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 [15]. 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 $j_0 \leq j \leq j_0 + k - 1$.
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 $m$.

The suffix tree for the concatenation of the text and pattern rows can be constructed in time $O(n^2 \log\log|\Sigma|)$. 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 [41], that the Harel and Tarjan [37] 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 [15].

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. $O(n^2 \log\log|\Sigma|)$ 2. $O(n^2)$.

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

It should be stressed that more modern and direct methods for solving this problem exist, using suffix arrays [38]. Also, other algorithms for computing the LCA in a tree exist [45, 19, 24].

## 2.2 Convolutions

Convolutions were officially introduced to the field of pattern matching Fischer and Paterson [27]. Denote by $A \otimes B$ the *convolution* of arrays $A$ and $B$. A convolution uses two initial functions, $A$ and $B$, to produce a third function $A \otimes B$. We formally define a discrete convolution.

**Definition:** Let $A$ be a real-valued function whose domain is $\{0, ..., n\}$ and $B$ a real-valued function whose domain is $\{0, ..., m\}$. We may view $A$ and $B$ as arrays of

numbers, whose lengths are $n+1$ and $m+1$, respectively. The *discrete convolution of $A$ and $B$* is the polynomial multiplication

$$A \otimes B[j] = \sum_{i=1}^{m} A[j+i]B[i], \quad j = 0, ..., n-m.$$

In the general case, the convolution can be computed by using the Fast Fourier Transform (FFT) [25]. This can be done in time $O(n \log m)$, in a computational model with word size $O(\log m)$. Fischer and Paterson used convolutions for solving exact string matching with "don't cares" (a special character that matches all symbols) in time $O(\log |\Sigma| n \log m)$.

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 [13] ).

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 $O(mn)$. 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. A uniquely two dimensional approach to pattern matching was introduced [4]. This technique analyzes 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 also proved useful in compressed matching [7], and in developing optimal parallel algorithms for two dimensional matching [8, 23].

**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 [4] 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 [4] 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 [4]).

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 $\Sigma$. 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.

It should be noted that later applications required some finer grained characterizations of periodicity [36, 14].

### 2.3.2 The Dueling Method

Dueling was first used by Vishkin for efficient parallel string matching algorithms [48]. The idea is to provide, in constant time, a method that eliminates one of two competing candidates for pattern occurrence. This elimination is based on identifying locations where the two candidates expect conflicting symbols. Vishkin used string periodicity properties to guarantee that such locations exist for every two overlapping candidates.

The dueling idea was extended to two dimensions by amir, Benson and Farach [6, 5]. It turned out that even where there is no periodicity, a judicious use of dueling can provide a simple and alphabet-independent $O(n^2)$ algorithm for two dimensional exact matching.

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 [6].

**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 [6, 11].

# 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 [43]:

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 [1] used a divide-and-conquer approach in conjunction with the convolutions method to solve the string mismatches problem in time $O(n\sqrt{m}\log m)$. His 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 [41] 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 $\Sigma$, 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 [40] and Amir and Landau [13] 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 [40] 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 ([13]).

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 [12] 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 [3]. 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 were found in time $O(k^3 n \log m \log k)$ in [12].

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

Recently, using superimposed codes, the $k$-aligned ones with locations problem has been solved $O(kn \log m \log k)$ [17].

# 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 [15]:

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 $\Sigma$. 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 [15] 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

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

A simpler, alphabet-independent algorithm, that can be generalized to dictionary scaled matching was presented in [11].

A key technique in all discrete scaling algorithms is the *Range Minimum Problem.* Defined as follows:

**Definition:** Let $L = [l_1, ..., l_n]$ be an array of $n$ numbers. A *Range Minimum* query is of the form:

> Given a range of indices $[i, ..., j]$, where $1 \leq i \leq j \leq n$, return an index $k \quad i \leq k \leq j$ such that $l_k = \min\{l_i, ..., l_j\}$.

In [34] it was shown that a list of length $n$ can be preprocessed in time $O(n)$ such that subsequent range minimum queries can be answered in constant time.

In scaled matching we are naturally interested in locations where there are many consecutive rows (columns) with the same elements. The range-minimum queries allow finding these areas in constant time. This can be achieved by preprocessing for every text location the longest common prefix of it, and the subrow immediately above it.

# 6 The Geometric Model

Everything we have seen so far suffers greatly from the encounter with "real-life problems". There is some justification for dealing with discrete scales in a combinatorial

sense, since it is not clear what is a fraction of a pixel. However, in reality an object may appear in non-discrete scales. It is necessary to, both, define the combinatorial meaning of such scaling, and present efficient algorithms for the problem's solution. The rotation problem, presents similar challenges. What is the discrete meaning of a rotated pattern? The answer to both above problems involves a *Geometric Model* for two-dimensional matching.

Until now, we considered the text and pattern to be matrices of alphabet symbols. The new idea, first proposed by Landau and Vishkin [42] is to consider the text and pattern as large rectangles composed of *unit squares*. These unit squares are "colored" by a picture of reality. For the sake of scaling and rotation, we consider the color of the center of a unit square as the color of the square. We will define the meaning of this geometric model for scaling and rotation in more detail in sections 6.1 and 6.2. However, for historical reasons we mention that Landau and Vishkin's motivation for defining the geometric model was the digitization process. For all intents the granularity of the world is so fine as to be considered continuous. Nevertheless, when a photo is taken, the image is projected onto a pixel map with much coarser granularity. Landau and Vishkin viewed the process as sampling unit squares and assigning an image pixel the color of its sampled center. This idea is used for the geometric definition of rotation and scaling to sizes that are not natural numbers.

## 6.1 Scaling

Amir, Butman, Lewenstein and Porat [10] present a definition for *scaled pattern matching* with arbitrary real scales. The definition is pleasing in a "real-world" sense. Below see "lenna" scaled to non-discrete scales by this definition and the results look natural (see Figure 1). This definition was inspired by the idea of digitizing analog signals by sampling, however, it does not assume an underlying continuous function thus stays on the combinatorial pattern matching field. This seems to be the natural way to define combinatorially the meaning of scaling in the signal processing sense.

Figure 1: An original image, scaled by 1.3 and scaled by 2, using the combinatorial definition of scaling.

This definition, that had been sought by researchers in pattern matching since at least 1990, captures *scaling* as it occurs in images, yet has the necessary combinatorial

features that allows developing deterministic algorithms and analysing their worst-case complexity. Indeed Amir, Butman, Lewenstein and Porat [10] present a two dimensional efficient algorithm for real scaled pattern matching.

The definition of two-dimensional scaled matching is an extension of the one dimensional definition.

**Definition** Let $T$ be a two-dimensional $n \times n$ array over some finite alphabet $\Sigma$.

1. The *unit pixels array* for $T$ ($T^{1X}$) consists of $n^2$ unit squares, called *pixels* in the real plane $R^2$. The corners of the pixel $T[i,j]$ are $(i-1,j-1),(i,j-1),(i-1,j)$, and $(i,j)$. Hence the pixels of $T$ form a regular $n \times n$ array covering the area between $(0,0),(n,0),(0,n)$, and $(n,n)$. Point $(0,0)$ is the *origin* of the unit pixel array. The *center* of each pixel is the geometric center point of its square location. Each pixel $T[i,j]$ is identified with the value from $\Sigma$ that the original array $T$ had in that position. We say that the pixel has a *color* from $\Sigma$. See figure 2 for an example of the grid and pixel centers of a $7 \times 7$ array.

2. Let $r \in \Re,\ r > 1$. The *r-ary pixels array* for $T$ ($T^{rX}$) consists of $n^2$ *r-squares*, each of dimension $r \times r$ whose *origin* is $(0,0)$ and covering the area between $(0,0),(nr,0),(0,nr)$, and $(nr,nr)$. The corners of the pixel $T[i,j]$ are $((i-1)r,(j-1)r),(ir,(j-1)r),((i-1)r,jr)$, and $(ir,jr)$. The *center* of each pixel is the geometric center point of its square location.

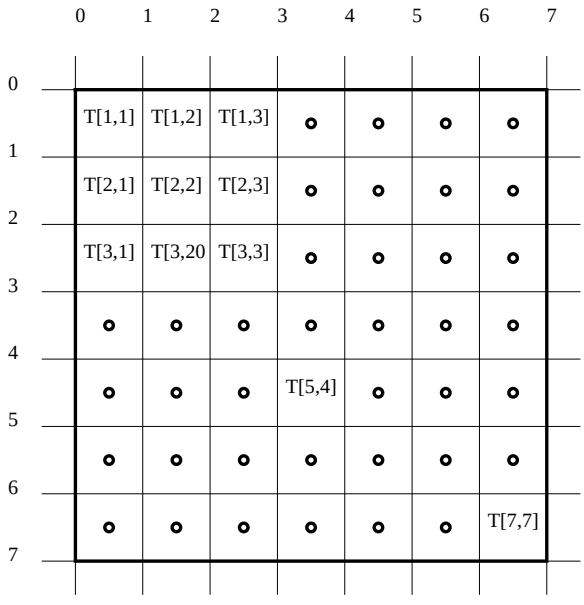


Figure 2: The grid and pixel centers of a unit pixel array for a $7 \times 7$ array.

**Notation:** Let $r \in \Re$. $\|r\|$ denotes the *rounding* of $r$, i.e.

$$\|r\| = \begin{cases} \lfloor r \rfloor & \text{if the fraction part of } r \text{ is less than .5;} \\ \lceil r \rceil & \text{otherwise.} \end{cases}$$

In this definition we round 0.5 up. There may be cases where we need to round 0.5 down. For this we denote:

$$||\lfloor r \rfloor|| = \begin{cases} \lfloor r \rfloor & \text{if the fraction part of } r \text{ is not more than .5;} \\ \lceil r \rceil & \text{otherwise.} \end{cases}$$

**Definition** Let $T$ be an $n \times n$ text array and $P$ be an $m \times m$ pattern array over alphabet $\Sigma$. Let $r \in \Re,\ 1 \leq r \leq \frac{n}{m}$.

We say that there is an *occurrence of $P$ scaled to $r$* at text location $[i, j]$ if the following condition hold:

Let $T^{1X}$ be the unit pixels array of $T$ and $P^{rX}$ be the r-ary pixel arrays of $P$. Translate $P^{rX}$ onto $T^{1X}$ in a manner that the origin of $P^{rX}$ coincides with location $(i-1, j-1)$ of $T^{1X}$. Every center of a pixel in $T^{1X}$ which is within the area covered by $(i-1, j-1), (i-1, j-1+mr), (i-1+mr, j-1)$ and $(i-1+mr, j-1+mr)$ has the same color as the $r$-square of $P^{rX}$ in which it falls.

The colors of the centers of the pixels in $T^{1X}$ which are within the area covered by $(i-1, j-1), (i-1, j-1+mr), (i-1+mr, j-1)$ and $(i-1+mr, j-1+mr)$ define a $\|mr\| \times \|mr\|$ array over $\Sigma$. This array is denoted by $P^r$ and called *$P$ scaled to $r$*.

It is possible to find all scaled occurrences of an $m \times m$ pattern in an $n \times n$ text in time $O(n^2m^2)$. Such an algorithm, while not trivial, is nonetheless achievable with known techniques. In [10] an $O(nm^3 + n^2 m \log m)$ algorithm was presented. Suitable trade-offs lead to an algorithm whose running time is $O(n^{1.5}m^2\sqrt{\log m})$.

The efficiency of the algorithm results from the properties of scaling. The scaling definition needs to accommodate a conflict between two notions, the continuous (represented by the real-number scale), and the discrete (represented by the array representation of the images). Understanding, and properly using, the shift from the continuous to the discrete and back are key to the efficiency of the algorithms.

## 6.2 Rotation

The pattern matching with rotation problem is that of finding all occurrences of a two dimensional pattern in a text, in all possible rotations. An efficient solution to the problem proved elusive even though many researchers were thinking about it for over a decade. Part of the problem was lack of a rigorous definition to capture the concept of rotation in a discrete pattern.

The major breakthrough came when Fredriksson and Ukkonen [31] resorted to a geometric interpretation of text and pattern and provided the following definition.

Let $P$ be a two-dimensional $m \times m$ array and $T$ be a two-dimensional $n \times n$ array over some finite alphabet $\Sigma$. As in the previous section, the array of *unit pixels* for $T$ consists of $n^2$ unit squares, called *pixels* in the real plane $R^2$. The corners of the pixel for $T[i, j]$ are $(i-1, j-1)$, $(i, j-1)$, $(i-1, j)$, and $(i, j)$. Hence the pixels for $T$ form a regular $n \times n$ array covering the area between $(0, 0)$, $(n, 0)$, $(0, n)$, and $(n, n)$. The *center* of each pixel is the geometric center point of the pixel. Each pixel $T[i, j]$ is identified with the value from $\Sigma$ that the original text had in that position. We say that the pixel has a *color* from $\Sigma$.

The array of pixels for pattern $P$ is defined similarly. A different treatment is necessary for patterns with odd sizes and for patterns with even sizes. For simplicity's sake we assume throughout the rest of this paper that the pattern is of size $m \times m$ and $m$ is even. The *rotation pivot* of the pattern is its exact center, the point $(\frac{m}{2}, \frac{m}{2}) \in R^2$. See Figure 3 for an example of the rotation pivot of a $4 \times 4$ pattern $P$.

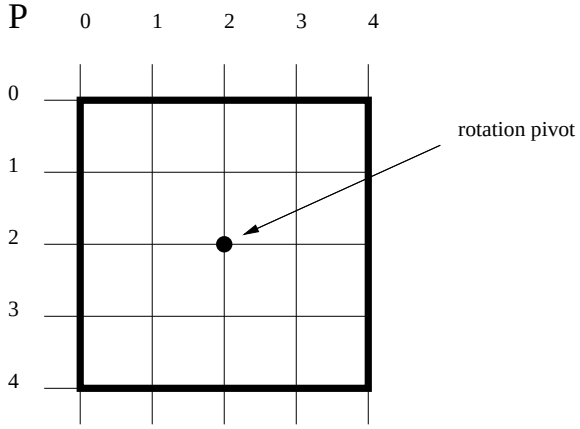


Figure 3: The rotation pivot of a $4 \times 4$ pattern $P$.

Consider now a rigid motion (translation and rotation) that moves $P$ on top of $T$. Consider the special case where the translation moves the grid of $P$ precisely on top of the grid of $T$, such that the grid lines coincide.

Assume that the rotation pivot of $P$ is at location $(i, j)$ on the text grid, and that the pattern lies *under* the text. The pattern is now rotated, centered at $(i, j)$, creating an angle $\alpha$ between the $x$-axes of $T$ and $P$. $P$ is said to be at *location* $((i, j), \alpha)$ *under* $T$. Pattern $P$ is said to have an *occurrence* at location $((i, j), \alpha)$ if the *center* of each pixel in $T$ has the same color as the pixel of $P$ under it, if there is such a pixel. When the center of a text pixel is exactly over a vertical (horizontal) border between text pixels, the color of the pattern pixel left (below) to the border is chosen. Consider some occurrence of $P$ at location $((i_0, j_0), \alpha)$. This occurrence defines a non-rectangular substring of $T$ that consists of all the pixel of $T$ whose centers are inside pixels of $P$. We call this string $P$ *rotated by* $\alpha$, and denote it by $P^\alpha$. Note that there is an occurrence of $P$ at location $((i, j), \alpha)$ if and only if $P^\alpha$ occurs at $(i, j)$.

Fredriksson, Navarro and Ukkonen [29] give two possible definitions for rotation. One is as described above and the second is, in some way, the opposite. $P$ is placed *over* the text $T$. More precisely, assume that the rotation pivot of $P$ is on top of location $(i, j)$ on the text grid. The pattern is now rotated, centered at $(i, j)$, creating an angle $\alpha$ between the $x$-axes of $T$ and $P$. $P$ is said to be at *location* $((i, j), \alpha)$ *over* $T$. Pattern $P$ is said to have an *occurrence* at location $((i, j), \alpha)$ if the center of each pixel in $P$ has the same color as the pixel of $T$ under it.

While the two definitions of rotation, "over" and "under", seem to be quite similar, they are not identical. For example, in the "pattern over text" model there exist angles for which two pattern pixel centers may find themselves in the same text pixel. Alternately, there are angles where there are "holes" in the rotated pattern, namely there is a text pixel that does not have in it a center of a pattern pixel, but all text pixels around it have centers of pattern pixels. See Figure 4 for an example.

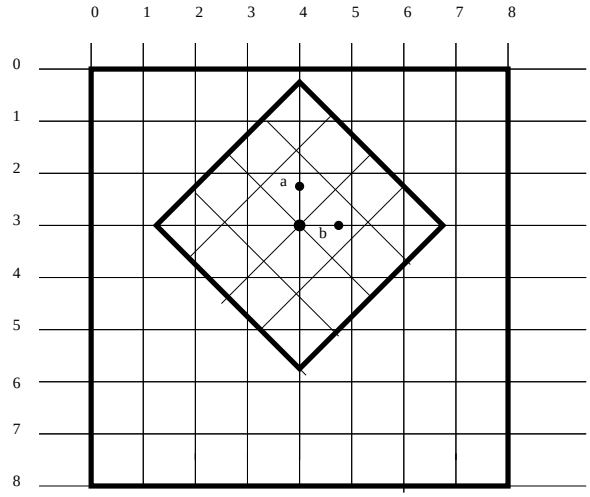


Figure 4: An example of a "hole" in the pattern. Text pixel $T[3,5]$ has no pattern pixel over it, but the pixels $T[2,5]$ and $T[3,6]$ have pattern pixel centers.

The challenges of "discretizing" a continuous image are not simple. In the Image Processing field, stochastic and probabilistic tools need to be used because the images are "smoothed" to compensate for the fact that the image is presented in a far coarser granularity than in reality. The aim of the the pattern matching community has been to fully discretize the process, thus our different definitions. However, this puts us in a situation where some "gut" decisions need to be made regarding the model that best represents "reality". It is our feeling that in this context the "pattern under text" model is more intuitive since it does not allow anomalies such as having two pattern centers over the same text pixel (a contradiction) nor does it create "holes" in the rotated pattern For examples of the rotated patterns in the two models see Figure 5.

Most of the algorithms for rotated matching are filtering algorithms that behave well on average but that have a bad worst case complexity (e.g. [32, 29, 33]). In three papers ([30, 9, 28]), there is a $O(n^2m^3)$ worst case algorithm for rotated matching. All worst-case algorithms basically work by enumerating all possible rotated patterns and solving a two dimensional dictionary matching problem on the text. In [9] it was proven that there are $\Theta(m^3)$ such rotated patterns. The high complexity results from the fact that the dictionary patterns have "don't care" symbols in them and thus, essentially, every pattern needs to be sought separately.

In [16], Amir, Kapah and Tsur present the first rotated matching algorithms whose time is better than $O(n^2m^3)$. The scanning time of their algorithms is $O(n^2m^2)$. These results are achieved by identifying monotonicity properties on the rotated patterns. These properties allow using *transitivity*-based dictionary matching algorithms, cutting the worst-case time by an $m$ factor.

# 7 Conclusions and Open Problems

We have scanned some of the problems and techniques in two dimensional matching. Clearly, we are still far from our motivation of actually finding a given template in an

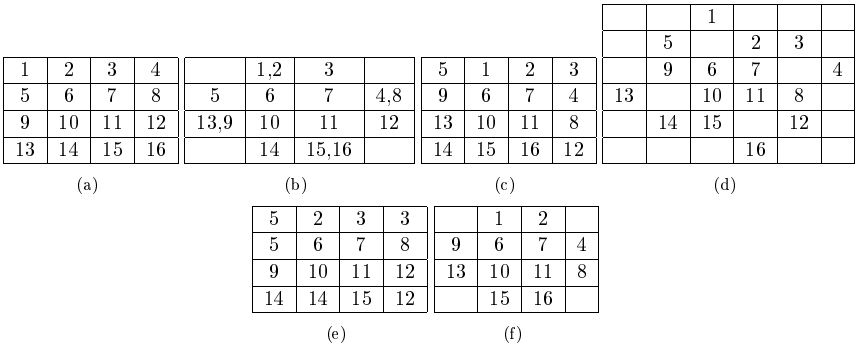


Figure 5: An example of some possible 2-dimensional arrays that represent one pattern. Fig (a) — the original pattern. Figures (b)–(d) are computed in the "pattern over the text" model. Fig (b) — a representation of the pattern rotated by $19^0$. Fig (c) — Pattern rotated by $21^0$. Fig (d) — Pattern rotated by $26^0$. Figures (e)–(f) are computed in the "pattern under the text" model. Fig (e) — Pattern rotated by $17^0$. Fig (f) — Pattern rotated by $26^0$.

image. What we have are 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 a real-time suffix tree construction algorithm, approximate indexing and dictionary matching algorihtms, and even more efficient algorithms for rotations. Other problems are new methods for general convolutions, multidimensional 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

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

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

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

[4] 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.

[5] A. Amir, G. Benson, and M. Farach. The truth, the whole truth, and nothing but the truth: Alphabet independent two dimensional witness table construction. Technical Report GIT-CC-92/52, Georgia Institute of Technology, August 1992.

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

[7] A. Amir, G. Benson, and M. Farach. Optimal two-dimensional compressed matching. *Proc. ICALP 94*, pages 215–226, 1994.

[8] A. Amir, G. Benson, and M. Farach. Optimal parallel two dimensional text searching on a crew pram. *Information and Computation*, 144(1):1–17, July 1998.

[9] A. Amir, A. Butman, M. Crocehmore, G.M. Landau, and M. Schaps. Two-dimensional pattern matching with rotations. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, number 2676 in LNCS, pages 17–31. Springer, 2003.

[10] A. Amir, A. Butman, M. Lewenstein, and E. Porat. Real two dimensional scaled matching. In *Proc. 8th Workshop on Algorithms and Data Structures (WADS)*, pages 353–364, 2003.

[11] A. Amir and G. Calinescu. Alphabet independent and dictionary scaled matching. *J. of Algorithms*, 36:34–62, 2000.

[12] 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.

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

[14] A. Amir, G. Landau, and D. Sokol. Inplace run-length 2d compressed search. *Theoretical Computer Science*, 290(3):1361–1383, 2003.

[15] 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.

[16] A. Amir, D. Tsur, and O. Kapah. Faster two dimensional pattern matching with rotations. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2004. to appear.

[17] Y. Aumann, M. Lewenstein, N. Lewenstein, and D. Tsur. Pealing codes with applications to finding witnesses. submitted for publication, 2004.

[18] 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.

[19] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. *Proc. 21st ACM Symposium on Theory of Computation*, pages 309–319, 1989.

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

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

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

[23] R. Cole, M. Crochemore, Z. Galil, L. Gąsieniec, R. Harihan, S. Muthukrishnan, K. Park, and W. Rytter. Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions. *Proc. 34th IEEE FOCS*, pages 248–258, 1993.

[24] R. Cole and R. Hariharan. Dynamic lca queries in trees. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 235–244, 1999.

[25] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1992.

[26] M. Farach. Optimal suffix tree construction with large alphabets. *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pages 137–143, 1997.

[27] 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.

[28] K. Fredriksson, V. Mäkinen, and G. Navarro. Rotation and lighting invariant template matching. In *Proceedings of the 6th Latin American Symposium on Theoretical Informatics (LATIN'04)*, LNCS, 2004. To appear. Available at `http://www.dcc.uchile.cl/ gnavarro/ps/latin04.ps.gzx`.

[29] K. Fredriksson, G. Navarro, and E. Ukkonen. An index for two dimensional string matching allowing rotations. In *Prof. IFIP International Conference on Theoretical Computer Science (IFIP TCS)*, volume 1872 of *LNCS*, pages 59–75. Springer, 2000.

[30] K. Fredriksson, G. Navarro, and E. Ukkonen. Optimal exact and fast approximate two dimensional pattern matching allowing rotations. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 2373 of *LNCS*, pages 235–248. Springer, 2002.

[31] K. Fredriksson and E. Ukkonen. A rotation invariant filter for two-dimensional string matching. In *Proc. 9th Annual Symposium on Combinatorial Pattern Matching (CPM 98)*, pages 118–125. Springer, LNCS 1448, 1998.

[32] K. Fredriksson and E. Ukkonen. A rotation invariant filter for two-dimensional string matching. In *Proc. 9th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1448 of *LNCS*, pages 118–125. Springer, 1998.

[33] K. Fredriksson and E. Ukkonen. Combinatorial methods for approximate pattern matching under rotations and translations in 3d arrays. In *Proc. 7th Symposium on String Processing and Information Retrieval (SPIRE'2000)*, pages 96–104. IEEE CS Press, 2000.

[34] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. *Proc. 16th ACM Symposium on Theory of Computing*, 67:135–143, 1984.

[35] 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.

[36] Z. Galil and K. Park. Alphabet-independent two-dimensional witness computation. *SIAM J. Comp.*, 25(5):907–935, October 1996.

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

[38] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 03)*, pages 943–955, 2003. LNCS 2719.

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

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

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

[42] G. M. Landau and U. Vishkin. Pattern matching in a digitized image. *Algorithmica*, 12(3/4):375–408, 1994.

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

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

[45] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comp.*, 17:1253–1262, 1988.

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

[47] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[48] U. Vishkin. Optimal parallel pattern matching in strings. *Proc.* 12*th ICALP*, pages 91–113, 1985.

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