A Failure Function for Multiple Two-dimensional Pattern Matching

Maxime Crochemore[1]*, Costas S. Iliopoulos[2,3]**,
Maureen Korda[2]***, and James F. Reid[2,4]†

[1] Institut Gaspard-Monge, Université de Marne-la-Vallée, Cité Descartes, 5 Bd Descartes, Champs-sur-Marne, F-77454 Marne-la-Vallée CEDEX 2, France.
[2] Dept. Computer Science, King's College London, London WC2R 2LS, UK.
[3] School of Computing, Curtin University of Technology, GPO Box 1987 U, Western Australia.
[4] Dipt. di Elettronica e Informatica, Università degli Studi di Padova, Via Gradenigo 6/A, Padova 35131, Italy.

Abstract. Given a two-dimensional text T and a set of patterns $\mathcal{D} = \{P_1, \ldots, P_k\}$ (the dictionary), the two-dimensional *dictionary matching* problem is to determine all the occurrences in T of the patterns $P_i \in \mathcal{D}$. The two-dimensional *dictionary prefix-matching* problem is to determine the longest prefix of any $P_i \in \mathcal{D}$ that occurs at each position in T. Given an alphabet Σ, an $n \times n$ text T and a dictionary $\mathcal{D} = \{P_1, \ldots, P_k\}$, we present an algorithm for solving the two-dimensional dictionary prefix-matching problem. Our algorithm requires $O(|T| + |\mathcal{D}|(\log m + \log|\Sigma|))$ units of time, where $m \times m$ is the size of the largest $P_i \in \mathcal{D}$. The algorithm presented here runs faster than the Amir and Farach [3] algorithm for the dictionary matching problem by an $O(\log k)$ factor. Furthermore, our algorithm improves the time bound that can be achieved using the Lsuffix tree of Giancarlo [6],[7] by an $O(k)$ factor.

Keywords: Two-dimensional string algorithms, dictionary prefix-matching, data structures, image processing.

---

* Partially supported by the C.N.R.S. Program "Génomes". mac@univ-mlv.fr.
** Partially supported by the EPSRC grant GR/J 17844. csi@dcs.kcl.ac.uk.
*** mo@dcs.kcl.ac.uk.
† Supported by a Marie Curie Fellowship of the European Commission Training and Mobility of Researchers (TMR) Programme. jfr@dcs.kcl.ac.uk.

1 Introduction

Given a string x of length n and a pattern p of length m, the *string prefix-matching* problem is that of computing the longest prefix of p which occurs at each position of x. Main and Lorentz introduced the notion of string prefix-matching in [15] and presented a linear time algorithm for it; their algorithm was an adaptation of the Knuth, Morris and Pratt algorithm [12].

In two dimensions, the *prefix-matching* problem is to compute the largest prefix of an $m \times m$ pattern P which occurs at each position of an $n \times n$ text T. For example if :

$$P = \begin{pmatrix} 0\,1\,1\,0 \\ 1\,1\,0\,1 \\ 0\,1\,0\,1 \\ 0\,0\,1\,1 \end{pmatrix} \text{ and } T = \begin{pmatrix} 1\,1\,0\,0\,1\,0\,0\,1 \\ 0\,\mathbf{0\,1\,1\,0}\,1\,1\,1 \\ 0\,\mathbf{1\,1\,0\,1}\,0\,1\,0 \\ 0\,\mathbf{0\,1\,0\,1}\,1\,0\,1 \\ 1\,\mathbf{0\,0\,1\,1}\,0\,0\,1 \\ 0\,1\,0\,\mathbf{0\,1}\,0\,0\,0 \\ 0\,\mathbf{0\,1\,1\,0}\,0\,1\,0 \\ 0\,\mathbf{1\,1\,0\,1}\,0\,1\,1 \end{pmatrix}$$

then the prefixes of the pattern P of size one are all the occurrences of the symbol 0 in the text T and the prefixes of size two, three and four are marked in bold.

The two-dimensional prefix-matching problem can be solved using the *LSuffix tree* data structure of Giancarlo [6], [7]. Giancarlo's data structure is a generalization to two dimensions of the McCreight[14] construction for one-dimensional strings. The LSuffix tree for an $n \times n$ input text T, defined over an alphabet Σ, takes $O(|T|(\log|\Sigma| + \log n))$ time to build, where $|T| = n^2$. All positions in T where an $m \times m$ pattern P occurs can be found in $O(|P|\log|\Sigma| + t)$ time, where $|P| = m^2$ and t is the total number of occurrences.

Let T be an $n \times n$ array called the *text* and let $\mathcal{D} = \{P_1, \ldots, P_k\}$ be a set of square patterns called the *dictionary*. Each $P_i \in \mathcal{D}$ is an $m_i \times m_i$ array such that the m_i's are not necessarily the same. We denote the total size of the dictionary by $|\mathcal{D}|$ where $|\mathcal{D}| = \sum_{i=1}^{k} m_i$ and let the largest pattern in $\mathcal{D}$ be of size $m \times m$.

Using the above notation, the two-dimensional *dictionary matching* problem (2DDM) is that of finding all the occurrences of the patterns in $\mathcal{D}$ in the text T. The two-dimensional *dictionary prefix-matching* problem (2DDPM) is to determine the longest prefix of at least one pattern in $\mathcal{D}$ that occurs at each position of the text T. A solution to the dictionary prefix-matching problem also solves the dictionary matching problem, but the reverse is not true. The two-dimensional *dictionary all prefix-matching* problem (2DDAPM) is to determine the longest prefix of all pat-

terns of $\mathcal{D}$ that occurs in each position of the text T. Here we present an algorithm for solving any of the three problems above that requires $O(|T| + |\mathcal{D}|(\log m + \log |\Sigma|))$ units of time.

Amir and Farach presented in [3] an $O((|T| + |\mathcal{D}|) \log k)$ time algorithm for the 2DDM problem. The algorithm presented here improves this bound by a factor of $O(\log k)$. One can use Giancarlo's Lsuffix tree for solving the 2DDPM problem in $O(|T| \log n + |\mathcal{D}|)$ time and the 2DDAPM problem in $O(|T| \log n + |\mathcal{D}| + t)$ units of time, where t is the number of occurrences of the patterns in the text. Since t can be $O(k|T|)$, the algorithm presented here is faster by at least a factor of $O(k)$.

In addition to its theoretical importance, dictionary matching problems have many practical applications in computer vision, data compression and visual databases. For example, in computer vision one is often interested in matching an enormous set of templates against a picture (see [3]). The 2DDAPM problem is of particular importance in occluded image analysis. In images composed from a given set of objects that may be partially occluded by each other, one needs to find prefixes of the objects occurring in the image (see [11], [9], [10]).

The paper is organised as follows. In the next section we present some basic definitions for strings in one and two dimensions. In Section 3 we describe the data structures needed for the algorithm: the common prefix tree and the Aho-Corasick automaton. In Section 4 we extend the notion of a failure function: we define the *dictionary diagonal failure function* and present an algorithm for computing it over a dictionary $\mathcal{D}$. In Section 5 we present the main algorithm. Finally in Section 6 we present our conclusions and open problems.

2 Background and basic string definitions

A *string* (or word) is a sequence of zero or more symbols drawn from an alphabet Σ, which consists of a finite number of symbols. The set of all strings over Σ is denoted by Σ^*. The string of length zero is the *empty string* ϵ and a string x of length $n > 0$ is represented by $x_1 x_2 \cdots x_n$, where $x_i \in \Sigma$ for $1 \leq i \leq n$. The string xy is a *concatenation* of two strings x and y. The concatenation of k copies of x is denoted by x^k. A string w is said to be *substring* of x if and only if $x = uwv$ for some $u, v \in \Sigma^*$. A string w is a *prefix* of x if and only if $x = wu$ for some $u \in \Sigma^*$; if u is not empty then w is a called a *proper prefix* of x. Similarly, w is a *suffix* of x if and only if $x = uw$ for some $u \in \Sigma^*$; if u is not empty then w is called a *proper suffix* of x. Additionally $prefix_k(x)$ denotes the first k symbols of x and $suffix_k(x)$ denotes the last k symbols of x.

A *two-dimensional string* is an $r \times s$ array (or matrix) of symbols drawn from Σ. We represent an $r \times s$ array X by $X[1..r, 1..s]$. A two-dimensional

$p \times q$ array Y is said to be a *sub-array* of X if the upper left corner of Y can be aligned with $X[i,j]$, i.e. $Y[1..p, 1..q] = X[i..i+p-1, j..j+q-1]$, for some $1 \leq i \leq r-p$ and $1 \leq j \leq s-q$. A square $m \times m$ sub-array Y is said to be a *prefix* of X if Y occurs at position $X[1..m, 1..m]$. Similarly, Y is said to a *suffix* of X if Y occurs at position $X[r-m+1..r, s-m+1..s]$.

In the sequel we let T be an $n \times n$ array called the *text* and $\mathcal{D} = \{P_1, \ldots, P_k\}$ be a set of square two-dimensional patterns P_i of size $m_i \times m_i$ for $i \in \{1, .., k\}$ called the *dictionary*. The m_i's are not necessarily the same. Furthermore we denote the total size of the dictionary by $|\mathcal{D}|$ where $|\mathcal{D}| = \sum_{i=1}^{k} m_i$.

We decompose our two-dimensional strings into L-shaped substrings according to the techniques first introduced in [3] and formally defined in [6, 7]. Specifically, we use the following notation from [6]: given an $n \times n$ array A, an Lcharacter is a string that is composed from the sub-row and sub-column of equal length that join in a position $A[i,j]$ and that occur to the left and to the top of $A[i,j]$. In this paper, we consider the positions of these substrings in reverse order: e.g, the Lcharacter that joins at position $A[i,i], 1 \leq i \leq n$ consists of the sub-row $A[i,i]A[i,i-1]..A[i,1]$ and the sub-column $A[i,i]A[i-1,i]..A[1,i]$. We call this the i-th Lcharacter of A. See Figure 1 (ii) for an example of an i-th Lcharacter and [7], [4] (Chap. 10) for further details and formalism on the linear representation of two-dimensional strings.

The *lower d-diagonal* of an $n \times n$ array T is $T[d,1], T[d+1,2], \ldots, T[n, n-d+1]$ for some $d \in \{1, .., n\}$. The *upper d-diagonal* of an $n \times n$ array T is $T[1,d], T[2,d+1], \ldots, T[n-d+1, n]$ for some $d \in \{1, .., n\}$.

Let T_s denote the prefix $T[1...s, 1...s]$ of T. We define the *diagonal failure function* $f(i)$ of T, for $1 \leq i \leq n$ to be equal to s, where s is the largest integer less than n such that $T[i-s+1...i, i-s+1...i] = T_s$; if there is no such s, then $f(i) = 0$ (see Figure 1 (i)). A linear procedure for computing the diagonal failure function was presented in [5]; a similar algorithm was presented in [1].

3 Common prefix tree and Aho-Corasick automaton

In this section we first introduce a simple data structure that stores all the common prefixes existing between patterns in $\mathcal{D}$. We then review the well known Aho-Corasick automaton that is used for dictionary matching in one-dimensional strings. Finally, we combine these techniques to create the data structure needed in the main algorithm.

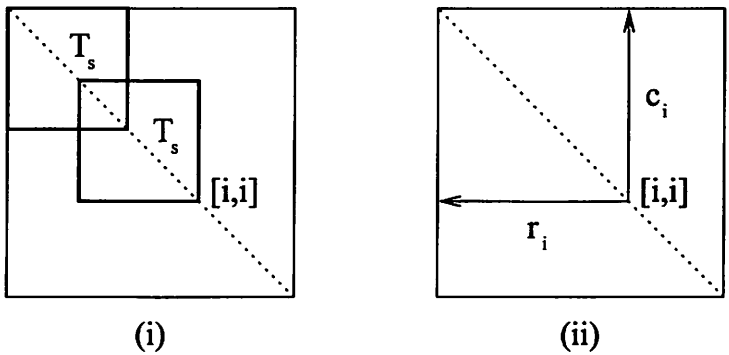

Fig. 1. (i) The diagonal failure function of a square matrix, (ii) The i-th Lstring of a square matrix.

3.1 Common prefix tree

The common prefix tree associated with a dictionary $\mathcal{D}$ of square two-dimensional patterns is a trie data structure such that the root node is at level 0 and the nodes at each level i represent all the distinct prefixes of size $i \times i$ that exist in $\mathcal{D}$. In this way, all common prefixes are identified and stored. It is used in the sequel for identifying subsets of $\mathcal{D}$ sharing common prefixes. Note that since the size of the patterns in $\mathcal{D}$ varies, one pattern may be contained within another one.

Definition 1. The *common prefix tree* for a dictionary $\mathcal{D} = \{P_1, \ldots, P_k\}$ of square patterns where P_i has size $m_i \times m_i$, is a rooted trie (digital search tree) with k leaves such that:

1. Each edge of the tree is labeled with the j-th Lcharacter of a pattern and is directed away from the root.
2. No two edges emanating from the same node have the same label.
3. The concatenation of m_i labels on the path from the root to a leaf is uniquely identified with a pattern P_i in $\mathcal{D}$.

We adopt the convention of padding each pattern P_i with an extra Lcharacter consisting of \$ symbols, where \$ is a symbol that does not appear in Σ. This ensures that each pattern P_i is uniquely identified with a leaf of the common prefix tree. Figure 2 represents a common prefix tree associated with a dictionary $\mathcal{D} = \{P_1, \ldots, P_6\}$.

Common prefix tree construction: The tree is constructed by simultaneously progressing along the main diagonal of each of the k patterns and

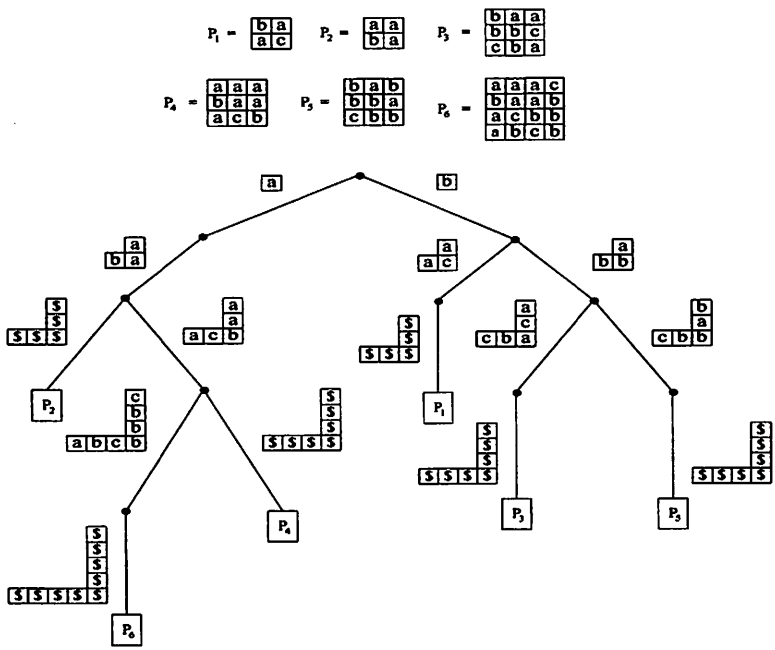

Fig. 2. The common prefix tree of the patterns in $\mathcal{D} = \{P_1, \ldots, P_6\}$.

performing substring comparisons amongst each resulting group of at most k Lcharacters. A brief description is as follows:

INITIALIZATION:

Firstly, using the patterns in $\mathcal{D}$, we build a suffix tree from the rows and a suffix tree from the columns. We then apply the linear time algorithm of [8] that preprocesses these trees for answering Lowest Common Ancestor (LCA) queries. This technique was first used in [13] for performing substring comparisons in constant time. We use the same method to compare sub-rows and sub-columns in constant time.

CONSTRUCTION:

At each step j, we compare the j-th Lcharacters (rows and columns) from each pattern of size at least $j \times j$: i.e the sub-rows $P_i[j, j..1]$ and the sub-columns $P_i[j..1, j]$. As soon as any two Lcharacters differ by at least one symbol, a new node is created in the tree.

After constructing the common prefix tree, we assign a unique index to each internal node such that each index represents a subset of patterns sharing a common prefix.

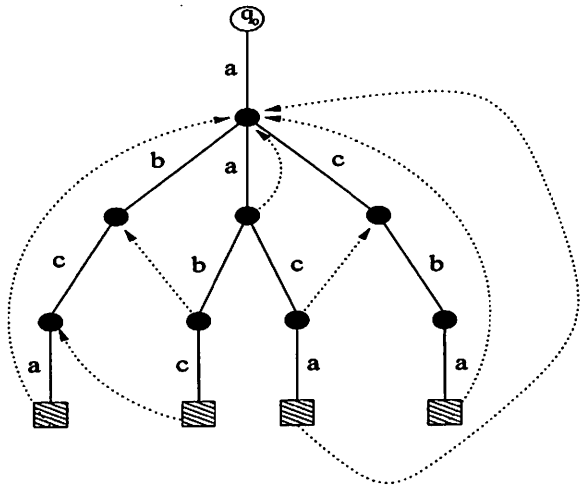

Fig. 3. The Aho-Corasick automaton for $\{abca, aabc, acba, aaca\}$. Non-trivial failure links are shown as dotted lines. Final states are shown as patterned squares.

Theorem 1. The common prefix tree of a dictionary of k square patterns of various sizes $\mathcal{D} = \{P_1, \ldots, P_k\}$ can be constructed in $O(km^2)$ time, where $m \times m$ is the size of the largest pattern in $\mathcal{D}$.

3.2 Aho-Corasick automaton

The Aho-Corasick automaton [2] was designed to solve the *multi-keyword pattern-matching* problem (one dimensional dictionary matching): given a set of keywords $\{r_1, \ldots, r_k\}$ and an input string t, test whether or not a keyword r_i occurs as a substring of t. The Aho-Corasick (AC) pattern matching automaton is a six-tuple $(Q, \Sigma, g, h, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $g : Q \times \Sigma \to Q \cup \{fail\}$ is the forward transition ($fail$ is a flag that is set when no forward transition can be made), $h : Q \to Q$ is the failure function (link), q_0 is the initial state and F is the set of final states (for details see [2]).

Informally, the automaton can be represented as a rooted labelled tree augmented with the failure links. The label of the path from the root (initial state q_0) to a state s is a prefix of one of the given keywords; we denote such a label by l_s. If s is a final state, then l_s is a keyword. There are no two sibling edges which have the same label. The failure link of a node s points to a node $h(s)$ such that the string $l_{h(s)}$ is the longest prefix of a keyword that is also a suffix of the string l_s (see Figure 3 for an example).

Theorem 2 ([2]). Given a set of keywords $\{r_1, \ldots, r_k\}$ and an input string t of length n, the Aho-Corasick automaton solves the multi-keyword pattern-matching problem in $O(n + km)$ time, where m is the length of the longest keyword.

In the sequel we shall need to perform substring comparisons using the Aho-Corasick automaton together with the following result.

Theorem 3 ([5]). Given the Aho-Corasick automaton for the set of keywords $r_1, \ldots, r_l$ and allowing linear time for preprocessing, the query of testing whether $\text{prefix}_s(r_{i,d}) = r_m$, where $r_{i,d}$ is the suffix of r_i which starts at the d-th position of r_i, requires constant time.

Next we describe our use of the Aho-Corasick automaton and how we establish links from it to the nodes of the common prefix tree. To aid the description we use of the following notation:
let r_i and c_i denote the rows and columns of the text T respectively, let $\tau_i^{(j)}$ denote the sub-rows $P_j[i,i]P_j[i,i-1]\ldots P_j[i,1]$, where $1 \leq i \leq m_j$ of each pattern P_j and let $\sigma_i^{(j)}$ denote the sub-columns $P_j[i,i]P_j[i-1,i]\ldots P_j[1,i]$, where $1 \leq i \leq m_j$ for each pattern P_j. We build an Aho-Corasick automaton using the following keywords:

$$r_1, \ldots, r_n, \quad \tau_1^{(1)}, \ldots, \tau_{m_1}^{(1)}, \quad \ldots, \quad \tau_1^{(k)}, \ldots, \tau_{m_k}^{(k)}$$

$$c_1, \ldots, c_n, \quad \sigma_1^{(1)}, \ldots, \sigma_{m_1}^{(1)}, \quad \ldots, \quad \sigma_1^{(k)}, \ldots, \sigma_{m_k}^{(k)}$$

From Theorem 3 it follows that, allowing linear time for preprocessing the keywords from the text and the patterns, the query of testing whether $\text{prefix}_s(\text{suffix}_d(r_i)) = \tau_s^{(j)}$ requires constant time. Note that several j's might satisfy this query. In order to identify whether a pattern P_j is one of those j's we label the edges of the common prefix tree using the final states of the AC automaton. Consider the edge between two nodes n_1, n_2 in the common prefix tree. If the Lcharacter that labels the edge (n_1, n_2) consists of the sub-row $P_j[i,i]\ldots P_j[i,1] = \tau_s^{(j)}$ and the sub-column $P_j[i,i]\ldots P_j[1,i] = \sigma_s^{(j)}$ for some $i \in \{1, .., m_j\}$, then we assign $LINK(n_1, h, g) = n_2$ where h and g are the final states in the AC automaton that are associated with $\tau_s^{(j)}$ and $\sigma_s^{(j)}$ respectively. We adopt the convention of padding the s-th row (column) of P_i, by the special symbol $\#_s \notin \Sigma$ for all s and i. This ensures that the s-th rows (columns) are uniquely identified for all s.

4 Dictionary diagonal failure function

The main algorithm makes use of the diagonals of the text matrix T (see Figure 4). Starting from the top of each d-diagonal and sliding downwards, we iteratively compute the maximum prefix, say $\hat{P}_i$, of some P_i at each point on the d-diagonal. At the next iteration step, we attempt to augment that occurrence by extending it by a sub-row and a sub-column, that is an $(i+1)$-th Lcharacter. If such an extension of the occurrence of the prefix of P_i is not possible, then we need to compute the largest prefix, say $\hat{P}_j$, of a pattern P_j, $j \in \{1,..,k\}$ that is also a suffix of $\hat{P}_i$. We then proceed to extend $\hat{P}_j$, if that is possible in turn. This technique is a generalization of the classical KMP failure function [12] to two-dimensional strings and multiple patterns.

Definition 2. Given a dictionary $\mathcal{D} = \{P_1, \ldots, P_k\}$, and a $p \times p$ prefix $\hat{P}_i$ of P_i, then we define the *dictionary failure function* $f(i,p) = (j,q)$ for all $1 \leq i,j \leq k$ where $1 \leq p \leq m_i$ and $1 \leq q \leq m_j$ if and only if the $q \times q$ prefix of the pattern P_j is the largest prefix of any pattern in $\mathcal{D}$ that is a suffix of $\hat{P}_i$.

STEP 1

Construct the generalised Lsuffix tree for $\mathcal{D} = \{P_1, \ldots, P_k\}$, where $m_i \times m_i$ is the size of P_i with $i \in \{1,..,k\}$. The construction is identical to the one used in [6] but uses only suffixes that occur along the main diagonals of the patterns. It requires $O(|\mathcal{D}|(\log m + \log|\Sigma|))$ units of time to build, where $m \times m$ is the size of the largest pattern in $\mathcal{D}$ and Σ is the alphabet.

STEP 2

Preprocess the Lsuffix tree to enable Lowest Common Ancestor (LCA) queries to be answered in constant time. The time for preprocessing is linear in the size of the tree, and uses the same techniques as those described in Section 3 for the construction of the common prefix tree.

STEP 3

Let i be the index of a pattern P_i, $i \in \{1,..,k\}$ and $s \in \{1,..,m_i\}$. Consider the $s \times s$ suffix of P_i, denoted as $\text{suffix}_s(P_i)$. This suffix is associated with a leaf of the Lsuffix tree. For each $j \in \{1,..,k\}$, we compute the Lowest Common Ancestor of P_j and $\text{suffix}_s(P_i)$. Let $d_{s,j}$ be the depth of the LCA. We perform this step for all patterns P_i and for all their suffixes. Since each LCA inquiry requires constant time, this step takes $O(|\mathcal{D}|)$ units of time.

STEP 4

The dictionary failure function is given by $f(i,p) = (j,q)$, where

$$q = \max\{d_{i,j} \mid p = d_{i,j} + m_i - s\}.$$

Note that there might be several patterns P_j that satisfy the maximal q, all having the same $q \times q$ prefix. Again this computation requires $O(|\mathcal{D}|)$ units of time.

The following theorem easily follows from above:

Theorem 4. The computation of the dictionary failure function requires $O(|\mathcal{D}|(\log m + \log|\Sigma|))$ units of time, where $m \times m$ is the size of the largest pattern in $\mathcal{D}$ and Σ is the alphabet.

5 Dictionary prefix-matching algorithm

The algorithm below makes use of the diagonals of the text matrix T (see Figure 4). Starting from the top of each d-diagonal and sliding downwards, we iteratively compute the maximum prefix of P_i for some $i \in \{1,..,k\}$ at each point of the d-diagonal. At the next iteration step, we attempt to augment that occurrence by extending it by a row and a column (in a manner similar to the L-character used by [6]); this is only possible when the relevant row and column of the text match the corresponding ones of the pattern. If such an extension of the occurrence of the prefix of P_i is not possible, then we make use of its diagonal failure function, and attempt to extend the prefix pointed to by the failure function. Analytically, the pseudo-code below computes the longest prefix of P_i for all $i \in \{1,..,k\}$ that occurs at every position in T; in order to simplify the exposition we only compute the maximum prefix of the patterns occurring at points below the main diagonal of the text.

Algorithm *2DAPPM*

input: $n \times n$ text $T, m_i \times m_i$ patterns P_i for $i \in \{1,..,k\}$, alphabet Σ.
output: longest prefix of each P_i in $\mathcal{D}$ for each location in T.

begin
$r_i \leftarrow T[i,n]T[i,n-1]\ldots T[i,1],\ 1 \le i \le n;$
$c_i \leftarrow T[n,i]T[n-1,i]\ldots T[1,i],\ 1 \le i \le n;$
comment r_i, c_i are the rows and columns of the text T reversed.
$\tau_i^{(j)} \leftarrow P_j[i,i]P_j[i,i-1]\ldots P_j[i,1],\ 1 \le i \le m_j,\ 1 \le j \le k;$
$\sigma_i^{(j)} \leftarrow P_j[i,i]P_j[i-1,i]\ldots P_j[1,i],\ 1 \le i \le m_j,\ 1 \le j \le k;$
comment $\tau_i^{(j)}, \sigma_i^{(j)}$ are the i-th Lcharacters of P_j.
$R \leftarrow \{r_1,\ldots,r_n,\tau_1^{(1)},\ldots,\tau_{m_1}^{(1)},\ldots,\tau_1^{(k)},\ldots,\tau_{m_k}^{(k)}\};$
$C \leftarrow \{c_1,\ldots,c_n,\sigma_1^{(1)},\ldots,\sigma_{m_1}^{(1)},\ldots,\sigma_1^{(k)},\ldots,\sigma_{m_k}^{(k)}\};$
Construct the Aho-Corasick automaton for R and C;
Compute the dictionary failure function $f(j,p)$ of P_j, for each
$1 \le j \le k$ and $1 \le p \le m_j$;
for $d = 1$ **to** n **do**
  Let $l_d = n - d + 1$ be the length of the diagonal;
  $r_{i,d} \leftarrow T[i,i-d+1]\cdots T[i,1],\ 1 \le i \le n;$
  $c_{i,d} \leftarrow T[i,i-d+1]\cdots T[d,i-d+1],\ 1 \le i \le n;$
  comment $r_{i,d}$ and $c_{i,d}$ are illustrated in Figure 4.
  $s \leftarrow 0;$
  for $j = 1$ **to** l_d **do**
    while $\text{prefix}_{s+1}(r_{j,d}) \neq \tau_{s+1}^{(q)}$ or $\text{prefix}_{s+1}(c_{j,d}) \neq \sigma_{s+1}^{(q)}$ **do**
      comment This is tested by using the list LINK (Theorem 3).
      $p(j-s,d) \leftarrow (q,s);$
      comment The integer $p(i,d)$ is the dimension of the largest prefix
        of the pattern P_q occurring at position $(i, i-d+1)$.
      $(q,s) \leftarrow f(q,s);$
    od
  $s \leftarrow s+1;$
  od
od
end

The test of the **while** loop condition in Algorithm *2DAPPM* is done using the list LINK. Suppose that the prefix that we have built so far is the label of the path from the root of the common prefix tree to the node n_1.

Also suppose that the Aho-Corasick automaton will terminate in a state say g for the row and state h for the column. The condition is met (the prefix can be extended) if $LINK(n_1, g, h)$ is defined.

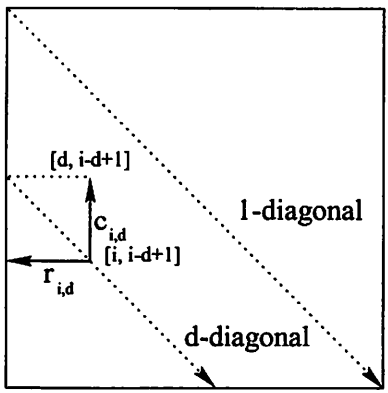

Fig. 4. A d-diagonal of the text T together with the sub-row $r_{i,d}$ and the sub-column $c_{i,d}$

Theorem 5. The algorithm (2DDAPM) computes the longest prefix of *all* square patterns from a given dictionary $\mathcal{D} = \{P_1, \ldots, P_k\}$ that occurs at each position of an $n \times n$ text T in $O(|T| + |\mathcal{D}|(\log m + \log|\Sigma|))$ time, where $m \times m$ is the size of the largest pattern in $\mathcal{D}$ for some $j \in \{1..k\}$.

Proof.

The computation of the dictionary failure function of all the patterns in $\mathcal{D}$ requires $O(|\mathcal{D}|(\log m + \log|\Sigma|))$ time. The computation of the Aho-Corasick automaton requires $O(|T| + |\mathcal{D}|)$ time. □

Corollary 1. *(2DDM)* There exists an algorithm that computes the occurrences of all square patterns P_j from a given dictionary $\mathcal{D} = \{P_1 \ldots, P_k\}$ in an $n \times n$ array T in $O(n^2 + |\mathcal{D}|(\log m + \log|\Sigma|))$ time.

Corollary 2. *(2DDPM)* There exists an algorithm that computes the occurrences of the longest prefix of any square pattern P_j from a given dictionary $\mathcal{D} = \{P_1, \ldots, P_k\}$ that occurs at each position of an $n \times n$ array T in $O(n^2 + |\mathcal{D}|(\log m + \log|\Sigma|)$ time.

6 Conclusion and open problems

Here we presented algorithms for three variants of the two-dimensional dictionary matching problem. The algorithm presented here runs faster than the Amir and Farach [3] algorithm for the dictionary matching problem by an $O(\log k)$ factor. Furthermore the same algorithm improves the time bound that can be achieved by using the Giancarlo's Lsuffix trees [6],[7] for the dictionary prefix-matching problem by a $O(k)$ factor.

An interesting open problem is to investigate whether the 2DDPM and 2DDAPM problems can be solved without the use of the Lsuffix trees. It will also be of interest to extend this work to rectangular arrays.

Due to the use of tree (indexing) data structures a large space overhead is incurred that detracts from the practicality of the algorithms given in this paper. Further investigation on designing more practical algorithms for the problems presented is needed. The new algorithms may be based on a different linearization technique (see [4]).

References

1. A. Amir, G. Benson and M. Farach, Alphabet independent two dimensional matching, In *Proc. 24th ACM Symposium on Theory of Computing*, (1992), 59–68.
2. A.V. Aho and M.J. Corasick, Efficient string matching: an aid to bibliographic search, *Comm. ACM*, (1975), 18(6), 333–340.
3. A. Amir and M. Farach, Two-dimensional dictionary matching, *Inform. Process. Lett.*, (1992), 44, 233–239.
4. A. Apostolico and Z. Galil, Pattern Matching Algorithms, *Oxford University Press*, (1997), New York.
5. M. Crochemore, C.S. Iliopoulos and M. Korda, Two-dimensional prefix string matching and covering on square matrices, *Algorithmica*, (1998), 20, 353–373.
6. R. Giancarlo, The suffix tree of a square matrix, with applications, In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms*, (1993), 402-411.
7. R. Giancarlo, A generalization of the suffix tree to square matrices, with applications, *SIAM J. Comput.*, (1995), 24(3), 520–562.
8. D. Harel and R.E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.*, (1984), 13(2), 338–355.
9. C.S. Iliopoulos and J.F. Reid, An optimal parallel algorithm for analysing occluded images, In *Proc. 4th Annual Australasian Conference on Parallel And Real-Time Systems*, (1997), University of Newcastle, Australia. N. Sharda and A. Tam (eds), Springer-Verlag, 104–113.
10. C.S. Iliopoulos and J.F. Reid, Validating and decomposing partially occluded two-dimensional images (extended abstract), In *Proc. 3rd Prague Stringology Club Workshop, PSCW98*, (1998), Dept. of Computer Science and Engineering, Czech Technical University, Prague, Czech Republic.

11. C.S. Iliopoulos and J. Simpson, On-line validation and analysis of occluded images, In *Proc. 8th Australasian Workshop on Combinatorial Algorithms*, (1997), Research on Combinatorial Algorithms, Queensland University of Technology, Australia, V. Estivill-Castro (ed), 25–36.
12. D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.*, (1977), 6, 322–350.
13. G. Landau and U. Vishkin, Fast parallel and serial approximate string matching, *J. of Algorithms*, (1989), 10, 157–169.
14. E.M. McCreight, A Space-economical suffix tree construction algorithm, *J. Assoc. Comput. Mach.*, (1976), 23, 262–272.
15. M.G. Main and R.J. Lorenz, An $O(n \log n)$ algorithm for finding all repetitions in a string, *J. of Algorithms*, (1984), 5, 422–432.
16. P. Weiner, Linear pattern matching algorithm, In *Proc. 14th IEEE Symp. on Switching and Automata Theory*, (1973), 1–11.