Two-dimensional prefix string matching and covering on square matrices

Maxime Crochemore, Costas S. Iliopoulos, Maureen Korda

Two-dimensional Prefix String Matching and Covering on Square Matrices

Maxime Crochemore* Costas S. Iliopoulos† Maureen Korda‡

August 1996

Abstract

Two linear time algorithms are presented. One for determining, for every position in a given square matrix, the longest prefix of a given pattern (also a square matrix) that occurs at that position and one for computing all square covers of a given two-dimensional square matrix.

* Institut Gaspard Monge, Universite de Marne-la-Vallee, 2, rue de la Butte Verte, F-93160 Noisy-le-Grand , France. Email: mac@univ-mlv.fr.

† Department of Computer Science, King's College London, Strand, London, England and School of Computing, Curtin University, Perth, WA, Australia. Email: csi@dcs.kcl.ac.uk. Partially supported by SERC grants GR/F 00898 and GR/J 17844, NATO grant CRG 900293, ESPRIT BRA grant 7131 for ALCOM II, and MRC grant G 9115730.

‡ Department of Computer Science, King's College London, Strand, London, U.K. Email: mo@dcs.kcl.ac.uk. Supported by a Medical Research Council Studentship.

1. Introduction

In recent studies of repetitive structures of strings, generalized notions of periods have been introduced. A typical regularity, the period u of a given string x, grasps the repetitiveness of x since x is a prefix of a string constructed by concatenations of u. A substring w of x is called a *cover* of x if x can be constructed by concatenations and superpositions of w. A substring w of x is called a *seed* of x if there exists a superstring of x which is constructed by concatenations and superpositions of w. For example, abc is a period of $abcabcabca$, $abca$ is a cover of $abcabcaabca$, and $abca$ is a seed of $abcabcaabc$. The notions "cover" and "seed" are generalizations of periods in the sense that superpositions as well as concatenations are considered to define them, whereas only concatenations are considered for periods.

Given a string x of length n and a pattern p of length m, the *prefix string matching* problem is that of determining, for every position in x, the longest prefix of p that occurs in that position. Main and Lorentz introduced the notion of prefix string matching in [ML84] and presented a linear time algorithm for it. In two dimensions, the *prefix string matching* problem is that of determining, for every position in a given $n \times n$ text matrix T, the longest prefix of a given $m \times m$ pattern matrix P that occurs in that position. The two-dimensional prefix string matching problem can be solved using the *LSuffix tree* construction of Giancarlo (see [G93], [G95]). The LSuffix tree for T, defined over an alphabet Σ, takes $O(n^2(\log|\Sigma|+\log n))$ time to build. Giancarlo's construction is the two-dimensional analog of the suffix tree ([W]). Two-dimensional dictionary matching and two-dimensional pattern retrieval (see [G95]) are among the applications of the Lsuffix tree; one can use the Lsuffix tree to derive $O(n^2(\log|\Sigma|+\log n))$ time for the two-dimensional prefix string matching problem. Here we present an optimal linear time algorithm for the two-dimensional prefix string matching problem that makes use of the notion of a two-dimensional failure function (also used in [ABF92]) as well as the Aho-Corasick automaton ([AC95]) in order to reduce the number of substring comparisons to linear. Both constructions are dependent on the alphabet. In the case of fixed alphabets, the algorithm presented here is faster than the one presented in [G95] by a factor of $O(logn)$.

In computation of covers, two problems have been considered in the literature. The *shortest-cover* problem is that of computing the shortest cover of a given string of length n, and the *all-covers* problem is that of computing all the covers of a given string. Apostolico, Farach and Iliopoulos [AFI91] introduced the notion of covers and gave a linear time algorithm for the shortest-cover problem. Breslauer [Br92] presented a linear time on-line algorithm for the same problem. Moore and Smyth [MS94] presented a linear time algorithm for the all-covers problem. In parallel computation, Breslauer

[Br95] gave an optimal $O(\alpha(n)\log\log n)$-time algorithm for the shortest cover, where $\alpha(n)$ is the inverse Ackermann function. Iliopoulos and Park [IP94] gave an optimal $O(\log\log n)$-time (thus work-time optimal) algorithm for the shortest-cover problem.

Iliopoulos, Moore and Park [IMP93] introduced the notion of seeds and gave an $O(n\log n)$-time algorithm for computing all the seeds of a given string of length n. For the same problem Ben-Amram, Berkman, Iliopoulos and Park [BBIP94] presented a parallel algorithm that requires $O(\log n)$ time and $O(n\log n)$ work. Apostolico and Ehrenfeucht [AE93] considered yet another problem related to covers.

In this paper we generalize the all-covers problem to two-dimensions and we present an optimal linear time algorithm for the problem. Let P be a square submatrix of a square matrix T; we say that P *covers* T (or equivalently P is a *cover* of T), if every point of T is within an occurrence of P. The *two-dimensional all-covers* problem is as follows: given a two-dimensional square matrix T, compute all square submatrices P that cover T. While the algorithms for the shortest-cover problem [AFI91,Br92,Br95,IP94] rely mostly on string properties, our algorithm for the two-dimensional all-covers problem is based on the Aho-Corasick Automaton and "gap" monitoring techniques.

A variant of the covering problem (see [DS]) defined above, was shown to have applications to DNA sequencing by hybridization using oligonucleotide probes: given string x, compute a minimal set of strings of fixed length k that cover x; the strings of the minimal set are said to be the k-covers of x. The k-covering problem is closely related to the string prefix matching problem, dictionary matching problem and the Aho-Corasick Automata; techniques employed here as well as the ones used by Giancarlo in [G95], could lead to efficient sequential and parallel solutions for the k-covering problem.

The paper is organized as follows: in the next section we present some definitions and results used in the sequel. In section 3 we present a linear time algorithm for computing all the borders of a square matrix. In section 4 we present a linear time algorithm for computing the diagonal failure function of a square matrix (an algorithm similar to the one in [ABF92]). In section 5 we present a linear time algorithm for the two-dimensional prefix string matching problem. And finally in section 6, we present a linear time algorithm for computing all the covers of square matrices.

2. Preliminaries

A *string* is a sequence (concatenation) of zero or more symbols from an alphabet Σ. The set of all strings over the alphabet Σ is denoted by Σ^*. The string xy is a *concatenation* of two strings x and y. The concatenations of k copies of x is denoted by x^k. A non-empty string x of length n is represented by $x_1x_2\cdots x_n$, where $x_i \in \Sigma$

for $1 \leq i \leq n$. A string w is a *substring* of x if $x = uwv$ for $u, v \in \Sigma^*$; we say that w occurs at position j of x if and only if $x_j \cdots x_{j+l-1} = w_1 ... w_l$, where l is the length of w. A string w is a *prefix* of x if $x = wu$ for $u \in \Sigma^*$; furthermore we define $\text{prefix}_k(x) = x_1 x_2 \ldots x_k$. Similarly, w is a *suffix* of x if $x = uw$ for $u \in \Sigma^*$; furthermore we define $\text{suffix}_k(x) = x_n x_{n-1} \ldots x_{n-k+1}$. A prefix $x_1 ... x_p$, for some $1 \leq p < n$ of x is a *period* of x, if $x_i = x_{i+p}$ for all $1 \leq i \leq n - p$. *The period* of a string x is the shortest period of x.

A two-dimensional string is an $n_1 \times n_2$ matrix drawn from Σ. In this paper, we deal exclusively with the special case $n = n_1 = n_2$, where the matrix is square. The $n \times n$ square matrix T can be represented by $T[1 \cdots n, 1 \cdots n]$. An $m \times m$ matrix P is a *submatrix* of T, if the upper left corner of P can be aligned with an element $T[i, j]$, $1 \leq i, j \leq n - m + 1$ and $P[1 \cdots m, 1 \cdots m] = T[i \cdots i + m - 1, j \cdots j + m - 1]$. In this case, the submatrix P is said to *occur* at position $[i, j]$ of T. A submatrix P is said to be a *prefix* of T if P occurs at position $[1, 1]$ of T. Similarly, a submatrix P is a *suffix* of T if P occurs at position $[n - m + 1, n - m + 1]$ of T.

A string b is a *border* of x if b is a prefix and a suffix of x. The empty string and x itself are *trivial* borders of x. An $m \times m$ submatrix P is a *border* of T, if P occurs at positions $[1, 1]$, $[n - m + 1, 1]$, $[1, n - m + 1]$ and $[n - m + 1, n - m + 1]$ of T. The empty matrix and the matrix T itself are *trivial* borders of T.

Fact 1. A string u is a period of $x = ub$ if and only if b is a non-trivial border of x.

Proof. It follows immediately from the definitions of period and border. □

Fact 2. A cover of string x is also its border. A cover of matrix T is also its border.

Proof. A cover of a string (matrix) occurs as both a prefix and a suffix and therefore it is a border. □

The Aho-Corasick Automaton [AC75] was designed to solve the *multi-keyword pattern-matching* problem: given a set of keywords $\{r_1, r_2 ..., r_h\}$ and an input string t of length n, determine for every keyword r_i whether or not it occurs as a substring of t. The Aho-Corasick 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 input, $g : Q \times \Sigma \rightarrow Q \cup fail$ is the forward transition, $f : Q \rightarrow Q$ is the failure function (link), q_0 is the initial state and F is the set of final states (for details see [AC75]).

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) to a state s is a

prefix of one of the given keywords ; we denote such 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 $f(s)$ such that the string $l_{f(s)}$ is the longest suffix of l_s that is a prefix of another keyword (see Figure 1 for an example.) The following theorem can be found in [AC95].

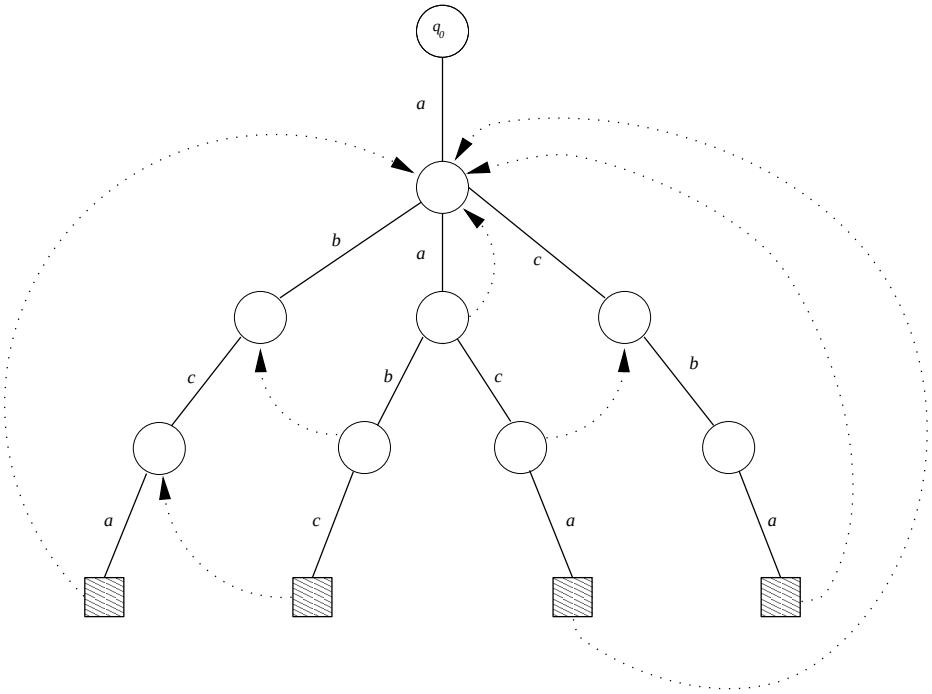


Figure 1

The Aho-Corasick Automaton for $\{abca, aabc, acba, aaca\}$. Trivial failure links, i.e., failure links to the initial state, are not shown. Non-trivial failure links are shown as dotted lines. Final states are shown as patterned squares.

Theorem 2.1 The Aho-Corasick automaton solves the multi-keyword pattern-matching problem in $O(\sum_{i=1}^{h} |r_i| + n)$ time. □

In the sequel we shall need to perform string comparisons using the Aho-Corasick automaton and the following results by Harel and Tarjan [HT84], Scheiber and Vishkin [SV88] and Berkman and Vishkin [BV94] will be used in guiding us within the automaton:

Theorem 2.2 Let $\mathcal{T}$ be a rooted tree with n nodes and let u, v be nodes of the tree. One can preprocess $\mathcal{T}$ in linear time, so that the following queries can be answered in constant time:

(i) Find the lowest common ancestor of u and v.

(ii) Find the k-th level ancestor node on the path from node u to the root where the first ancestor of u is the parent of u. □

Using the above theorem one can derive the following two useful corollaries:

Corollary 2.3 Given the Aho-Corasick automaton for keywords $r_1, r_2, ..., r_h$ and allowing linear time for preprocessing, the query of testing whether $\text{prefix}_k(r_i) = r_m$ requires constant time.

Proof. We preprocess the automaton as required by the Berkman-Vishkin algorithm. We can identify the ancestor node, s, of the leaf r_i, which corresponds to the k-th prefix of r_i in constant time. This node is the $(|r_i| - k)$-th ancestor on the path from the leaf r_i (final state) to the root (initial state). The equality holds only when s is also the leaf (final state) r_m. □

Corollary 2.4 Given the Aho-Corasick automaton for keywords $r_1, r_2, ..., r_h$ and allowing linear time for preprocessing, the query of testing whether $\text{prefix}_k(r_{i,d}) = r_m$ requires constant time, where $r_{i,d} = r_i[d..|r_i|]$.

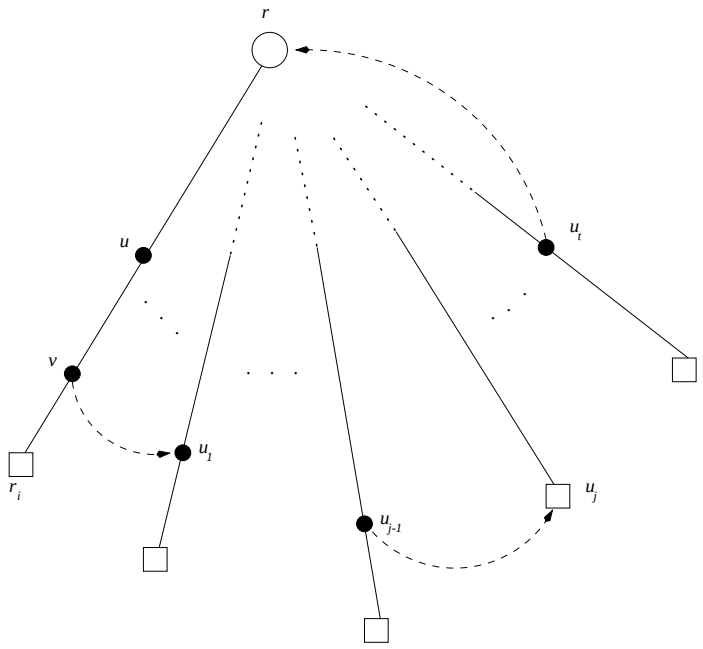


Figure 2

Proof. Let $r_i = \rho_1\rho_2\cdots\rho_q$ and $r_{i,d} = \rho_d\rho_{d+1}\cdots\rho_q$. Then we have $\text{prefix}_k(r_{i,d}) = \rho_d\rho_{d+1}\cdots\rho_{d+k-1}$. Let v be the $d+k$-th node on the path from the root to the leaf r_i. Using the failure links it is possible to construct a path $v, u_1..., u_t, r$ from the node v to the root r. It is not difficult to see that the label of the path from the root r to u_j is $s := \rho_{d+k-l}\rho_{d+k-l+1}\cdots\rho_{d+k}$, where l is the distance of u_j from the root r (see Figure 2). One can see that $\text{prefix}_k(r_{i,d})$ is a label of a path from the root to a node u, if and only if u is one of the u_j's. One can say that equality of the query holds if u_j, for some j, is the leaf r_m.

Given the automaton one can answer the query in constant time as follows. Consider the following tree $\mathcal{T}$: root is the initial state of the Aho-Corasick automaton, nodes are the nodes of the automaton and edges the failure links of the automaton. Let $\text{prefix}_k(r_{i,d})$ be associated with a node v as above. It is not difficult to see that $\text{prefix}_k(r_{i,d}) = r_m$ if and only if the Lowest Common Ancestor of r_m and v in T is r_m - a condition that can be checked in constant time by the [BV94] algorithm. □

By Fact 2, all borders of a string x or matrix T are candidates for covers. Our algorithm for the all-covers problem starts by computing all the borders of T and finds covers among the borders.

The algorithm is subdivided in the following three steps:

1. Compute all the borders $B_1, B_2, \cdots B_k$ of the matrix T and derive the candidates for covers. Here we use the fact that a cover of T is also a border. Let B_k be the largest candidate border.
2. For every position $[i, j]$ of T, compute the longest prefix of B_k that occurs at that position. We next "round down" these occurrences to the nearest border size, thus finding the largest border that occurs in that position.
3. We begin a gap monitoring program starting with B_k. We consider all occurrences of B_k in T and we check whether there are any positions of T that are not covered by an occurrence of B_k –we call these positions "gaps" (see section 6 for a formal definition). If there is a gap, then B_k is not a cover. We proceed by considering whether B_{i-1} covers T for all $i = k-1, ..., 1$, again by monitoring their gaps.

3. Computing All Borders and Candidates

Here we describe a linear time algorithm for computing all the square borders of an $n \times n$ matrix T. The algorithm makes use of two auxiliary $n \times n$ matrices C and R, where $C[i, j] := 1$, if and only if the j-th column of T has a border of length

i and $R[i,j] := 1$, if and only if the j-th row of T has a border of length i; otherwise $C[i,j] = R[i,j] = 0$. Formally

$$C[i,j] = \begin{cases} 1, & T[1..i,j] \text{ is a border of } T[1..n,j]; \\ 0, & \text{otherwise.} \end{cases}$$

$$R[i,j] = \begin{cases} 1, & T[j,1..i] \text{ is a border of } T[j,1..n]; \\ 0, & \text{otherwise.} \end{cases}$$

We compute all the borders of every row and column of T using the Knuth, Morris and Pratt ([KMP77]) algorithm. Furthermore we also use matrix M, such that:

$$M[i] = \begin{cases} 1, & R[i,j] = C[i,j] = 1, \ \forall\ 1 \le j \le i \text{ AND } R[i,l] = 1, \ \forall\ n-i+1 \le l \le n; \\ 0, & \text{otherwise.} \end{cases}$$

Lemma 3.1. The square submatrix $T[1..i,1..i]$ is a border of T if and only if $M[i] = 1$.

Proof. Assume that $M[i] = 1$. From the definition of $M[i]$ it follows that $R[i,j] = 1$, for all $1 \le j \le i$, which in turn implies that rows $r_1, r_2, ..., r_i$ all have borders of length i. Therefore $T[1..i,1..i]$ occurs at position $[1, n-i+1]$ of T. Similarly, from the fact that $C[i,j] = 1$, for all $1 \le j \le i$ the columns $c_1, c_2, ..., c_i$ all have borders of length i. Therefore $T[1..i,1..i]$ occurs at position $[n-i+1, 1]$ of T. From the fact that $R[i,j] = 1$, for all $n-i+1 \le j \le n$ it follows that rows $r_{n-i+1}, r_{n-i+2}, ..., r_n$ all have borders of length i, which in turn implies that $T[1..i,1..i]$ occurs at position $[n-i+1, n-i+1]$ of T.

The converse follows similarly. □

Based on this lemma, the following algorithm is proposed to compute the borders of T:

begin
  Compute $C[i,j]$, $1 \le i,j \le n$;
  Compute $R[i,j]$, $1 \le i,j \le n$;
  comment Use the KMP algorithm.
  for $1 \le i \le n$ **do**
    if $M[i] = 1$ **then**
      return $T[1..i,1..i]$ as a border of T;
      comment It follows from Lemma 3.1.
  od
end □

Algorithm 3.1

Theorem 3.2 Algorithm 3.1 computes all the borders of an $n \times n$ matrix T in $O(n^2)$ time.

Proof. The computation of the borders of each row and column takes $O(n)$ time using the KMP algorithm, $O(n^2)$ in total. Border verification of each diagonal candidate takes time $O(i)$, $O(n^2)$ in total. □

Although here we make use of the borders as candidates for covers, one can obtain a (perhaps) smaller set of candidates by modifying the matrices C and R as follows:

$$C[i,j] = \begin{cases} 1, & T[1..i,j] \text{ is a cover of } T[1..n,j]; \\ 0, & \text{otherwise.} \end{cases}$$

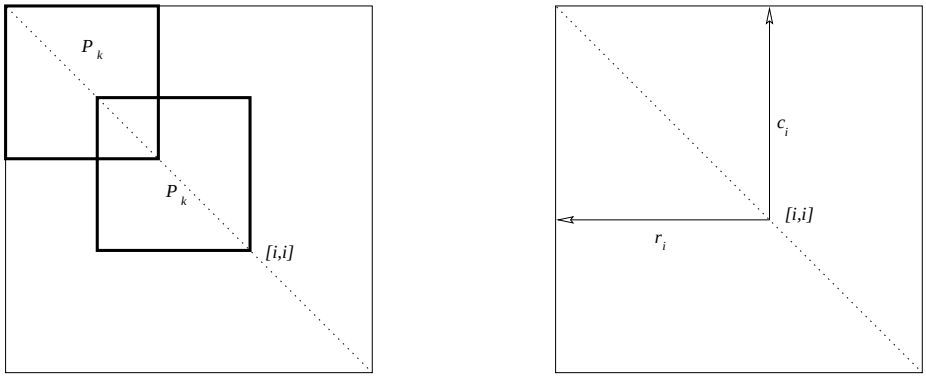
$$R[i,j] = \begin{cases} 1, & T[j,1..i] \text{ is a cover of } T[j,1..n]; \\ 0, & \text{otherwise.} \end{cases}$$

One can compute all the covers of every row and column of T using the [MS94] algorithm. The verification of the above set of candidates can be similarly done in linear time. These candidates may lead to a more efficient algorithm (as they may be fewer than the borders) but it does not change the asymptotic complexity.

4. Computing the Diagonal Failure Function of a Square Matrix

An $m \times m$ submatrix P is said to be a *diagonal border* of an $n \times n$ matrix T, if P occurs at positions $[1,1]$ and $[n-m+1, n-m+1]$ of T. We define the *diagonal failure function* $f(i)$, of T, for $1 \leq i \leq n$ to be equal to k, where $T[1..k, 1..k]$ is the largest diagonal border of $T[1..i, 1..i]$; if there is no such k, then $f(i) = 0$ (see Figure 3-(i)). Below we present a linear time procedure for computing the diagonal failure function; a similar algorithm was presented in [ABF92] and [ABF94].

The computation of the diagonal failure function shadows the computation of the failure function given in the Knuth-Morris-Pratt algorithm, with the exception that character comparisons are now substring comparisons. The algorithm makes use of the Aho-Corasick multi-word automaton to perform these comparisons in constant time. First we construct the Aho-Corasick multi-word automaton for the set of words $r_1, r_2, ..., r_n, c_1, c_2, ..., c_n$, where $r_i = T[i,i]T[i,i-1]\cdots T[i,1]$, $1 \leq i \leq n$ and $c_i = T[i,i]T[i-1,i]\cdots T[1,i]$, $1 \leq i \leq n$ (see Figure 3-(ii)). Secondly, at iteration i, we have computed $f(i) = k$ and we proceed to compute $f(i+1)$ by comparing r_{k+1} and c_{k+1} with



(i) (ii)

Figure 3

$\text{prefix}_{k+1}(r_{i+1})$ and $\text{prefix}_{k+1}(c_{i+1})$ respectively (Recall that $\text{prefix}_k(x) = x_1x_2\cdots x_k$ and $\text{suffix}_k(x) = x_kx_{k+1}\cdots x_n$ with $|x| = n$.) Clearly, if both match, then we have $f(i+1) = k+1$. Otherwise, as in KMP, we recursively (on k) compare r_{k+1} and c_{k+1} with $\text{prefix}_{k+1}(r_{i+1})$ and $\text{prefix}_{k+1}(c_{i+1})$, with $k := f(k)$, until both strings match.

Formally, the algorithm is as follows:

begin
  $r_i \leftarrow T[i,i]T[i,i-1]\cdots T[i,1], \quad 1 \le i \le n;$
  $c_i \leftarrow T[i,i]T[i-1,i]\cdots T[1,i], \quad 1 \le i \le n;$
  $R \leftarrow \{r_1, r_2, ..., r_n\};$
  $C \leftarrow \{c_1, c_2, ..., c_n\};$
  Construct the Aho-Corasick automaton for R and C;
  Preprocess the automaton for LCA queries;
  $f(1) \leftarrow 0; \quad k \leftarrow f(1);$
  for $i = 2$ **to** n **do**
    while $\text{prefix}_{k+1}(r_i) \neq r_{k+1}$ or $\text{prefix}_{k+1}(c_i) \neq c_{k+1}$ **do**
      comment The condition is tested as in Corollary 2.3.
      $k \leftarrow f(k);$
    od
    $k \leftarrow k+1;$
    $f(i) \leftarrow k;$
  od
end □

Algorithm 4.1

Theorem 4.1 Algorithm 4.1 computes the diagonal failure function of an $n \times n$ matrix in $O(n^2)$ time.

Proof. The computation of the Aho-Corasick automaton and their preprocessing require $O(n^2)$ time. The condition of the while loop can be computed in constant time as in Corollary 2.3. □

5. Prefix String Matching in Two Dimensions

Let P (the "Pattern"), T (the "Text") be $m \times m$, and $n \times n$ matrices respectively. The two-dimensional problem of prefix string matching is that of determining, for every position in the text matrix T, the longest prefix of P that occurs in that position.

The algorithm below shadows the Main-Lorentz ([ML84]) algorithm on the diagonals of the text matrix T (see Figure 4) with the exception that character comparisons are now substring comparisons. We use the Aho-Corasick automata to perform string comparisons in constant time. In order to simplify the exposition, we only compute the maximum prefix of the pattern occurring at points below the main diagonal of the text. The main diagonal is said to be the *1-diagonal*, and the diagonal starting at position $[d, 1]$ is said to be the *d-diagonal*.

In Algorithm 5.1 below, we first construct the Aho-Corasick multi-word automaton for the set of rows $r_1, r_2, ..., r_n$ and $r'_1, r'_2, ..., r'_m$ of T and P. We also construct the Aho-Corasick multi-word automaton for the set of columns $c_1, c_2, ..., c_n$ and $c'_1, c'_2, ..., c'_m$ of T and P. Starting from the top of each d-diagonal, and sliding downwards (on the d-diagonal), we iteratively compute the maximum prefix of P at each point of the d-diagonal as follows: at the end of the j-th iteration, we have computed the largest prefix of P that is a suffix of $T[d..d+j-1, 1..j]$ for all $1, ..., j$. Next, 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 Giancarlo in [G93] and [G95]); 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 is not possible, then we make use of the diagonal failure link, and next we attempt to extend the prefix pointed to by the link. Analytically, the pseudo-code for solving the prefix string matching problem is presented below.

begin
$r_i \leftarrow T[i,n]T[i,n-1]\cdots T[i,1], \quad 1 \le i \le n;$
$c_i \leftarrow T[n,i]T[n-1,i]\cdots T[1,i], \quad 1 \le i \le n;$
comment The strings r_i, c_i are the rows and columns of the text T reversed.
$r'_i \leftarrow P[i,i]P[i,i-1]\cdots P[i,1], \quad 1 \le i \le m;$
$c'_i \leftarrow P[i,i]P[i-1,i]\cdots P[1,i], \quad 1 \le i \le m;$
comment The strings r'_i, c'_i are similar to the ones in Figure 3-(ii).

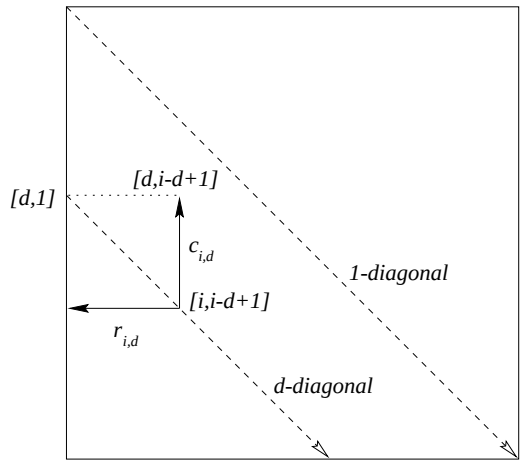


Figure 4

$R \leftarrow \{r_1, r_2, ..., r_n, r'_1, r'_2, ..., r'_m\}$;
$C \leftarrow \{c_1, r_2, ..., c_n, c'_1, c'_2, ..., c'_m\}$;
Construct the Aho-Corasick automata for R and C;
Compute the diagonal failure function f of P;
for $d = 1$ **to** n **do**
  $k \leftarrow 0$;
  for $i = d$ **to** n **do**
    $r_{i,d} \leftarrow T[i, i-d+1] \ \cdots \ T[i, 1]$;
    $c_{i,d} \leftarrow T[d, i-d+1] \ \cdots \ T[d, i-d+1]$;
    comment See Figure 4 for illustration of $r_{i,d}$ and $c_{i,d}$.
    while $\text{prefix}_{k+1}(r_{i,d}) \neq r'_{k+1}$ or $\text{prefix}_{k+1}(c_{i,d}) \neq c'_{k+1}$ **do**
      comment The condition is tested as in corollary 2.4.
      $p(i-k, d) \leftarrow k$;
      comment The integer $p(i-k, d)$ is the length of the largest prefix of the pattern occurring at position $[i-k, i-k-d+1]$.
      $k \leftarrow f(k)$;
    od
  $k \leftarrow k+1$;
  od
od
end □

Algorithm 5.1

Theorem 5.1 Algorithm 5.1 computes the longest prefix of the $m \times m$ matrix P occurring at every position of an $n \times n$ matrix T in $O(n^2 + m^2)$ time.

Proof. The computation of the diagonal failure function of P requires $O(m^2)$ time. The

computation of the Aho-Corasick automata requires $O(n^2 + m^2)$ time. The theorem follows from corollaries 2.3 and 2.4. □

6. Computing the Gaps

In this section we focus on the covering problem: given a square matrix T compute all sub-matrices S that cover T. Recall that a sub-matrix S covers T if every position of T is within an occurrence of S. The linear time algorithm below is based on gap monitoring techniques.

The algorithm makes use of the fact that a cover of T is also a border. We first compute all the borders of T, let B_k be the largest one. Next we compute the largest prefix of B_k that occurs at every position of T. We "round down" these occurrences to the nearest border size. Then we begin a gap monitoring program starting with B_k. We consider all occurrences of B_k in T and we check whether there are any positions that are not covered by an occurrence of B_k – these "uncovered" positions are called gaps. If there is a gap, then B_k is not a cover. We proceed by considering whether B_{k-1} covers T. Note that B_{k-1} occurs in all positions of B_k, thus we have to "add" some positions to the occurrences of B_k to obtain the set of occurrences of B_{k-1} in T. In fact every insertion of a new position may "reduce" previous gaps. Also due to the fact that we now consider a smaller border, B_{k-1}, some positions previously covered by B_k may now be left "uncovered", and therefore the gap size between previous gaps may "increase". We monitor all these changes and if all gaps are closed then the border is a cover. Analytically the steps of the *Gap Monitoring Algorithm* are as follows:

Algorithm 6.1

STEP 1. Compute all the square $b_t \times b_t$ borders B_t, $1 \leq t \leq k$, of the input matrix T; Let B_k be the largest border and without loss of generality $b_t < b_{t+1}$, $1 \leq t < k$. For $1 \leq i \leq n$, let $\mathcal{F}(i) = \{t \ : \ b_t \leq i < b_{t+1}\}$ and let $\mathcal{C}(i) = \{t \ : \ b_{t-1} < i \leq b_t\}$. The "floor" $\mathcal{F}$ function will be used for rounding down the maximum border prefixes computed in the next step and the "ceiling" $\mathcal{C}$ function will be used for rounding up the gap length in step 5.

STEP 2. For every position $[i, j]$ of T compute the length $d(i, j)$ of the maximum prefix of B_k that occurs in that position by using Algorithm 5.1. Let $P[i, j] = \mathcal{F}(d(i, j))$, i.e. the index of the largest border that occurs at position $[i, j]$ of T. Here we round down the occurrence (of the prefix of B_k) to the nearest border size, because only borders are candidates for covers (see Fact 2).

STEP 3. Let D_t denote the lexicographicaly ordered list of positions $[e, j]$ such that

(i) $P[i, j] = t$,

(ii) $i \leq e < i + b_t$.

The list D_t contains all positions of T which belong to the first column of an occurrence of B_t (see shaded area of the j-column of T in Figure 5-(i)). Furthermore each position $[p, q]$ of T is associated with a *range* $[p, q, l, r]$ if and only if $[p, q] \in D_t$, $\forall\ t\ :\ l \leq t \leq r$, $[p, q] \notin D_{l-1}$ and $[p, q] \notin D_{r+1}$ (see Figure 5-(ii)). In other words, a position $[p, q]$ is associated with a range $[p, q, l, r]$, if and only if, the position $[p, q]$ is within the first column of an occurrence of all of the following borders $B_l, B_{l+1}, ..., B_r$, but the position $[p, q]$ is not within the first column of an occurrence of either B_{l-1} or B_{r+1}. Note that a position may be associated with more than one range, e.g., in Figure 5-(ii), assuming that $\alpha < l - 1$, we have $[p, q, l, r]$ and $[p, q, \gamma, \alpha]$. The computation of these ranges is described in detail in the next section 6.2.

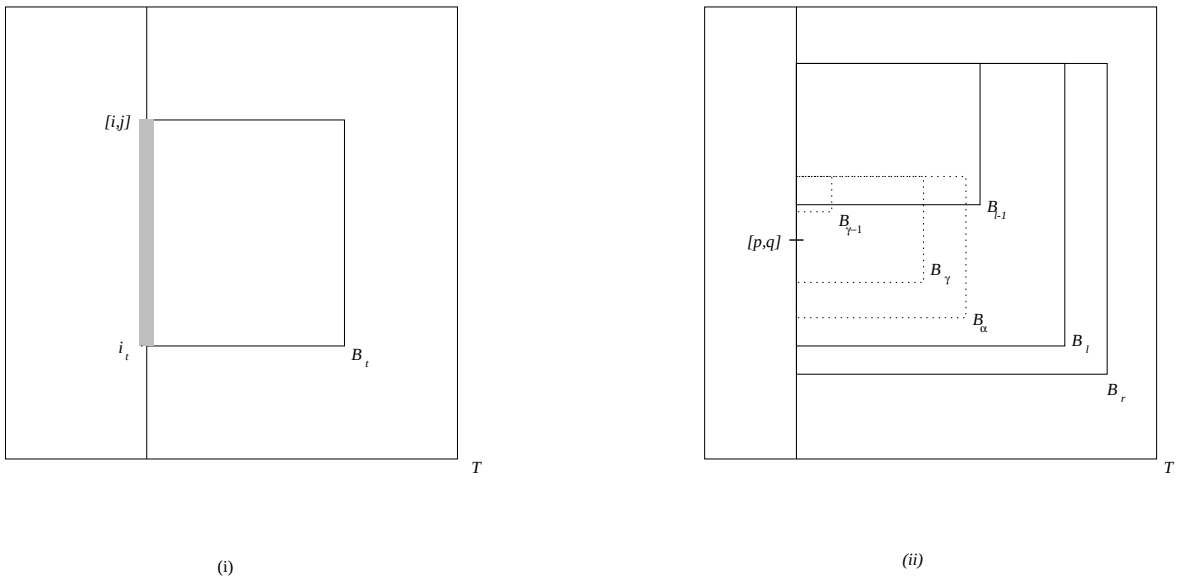


Figure 5

STEP 4. Next we define the *Left* and *Right* functions that will allow constant time deletion and insertion in updating the list D_t, which will be kept as a doubly linked list. We formally define

$$Left([p, q, l, r]) := [p, q', l', r']$$

if and only if $[p, q']$ is the first entry to the left of $[p, q]$ in the p-th row of T for which $l' \leq l \leq r'$, where $[p, q, l, r]$ and $[p, q', l', r']$ are the ranges associated with positions $[p, q]$ and $[p, q']$ respectively; otherwise $Left([p, q, l, r]) := [p, 1, -, -]$ (Note that $[p, 1, -, -]$ is not a "true" range, it is merely used here as an endpoint for the *Left* function).

We formally define

$$Right([p,q,l,r]) := [p,q'',l'',r'']$$

if and only if $[p,q'']$ is the first entry to the right of $[p,q]$ in the p-th row of T for which $l'' \leq l \leq r''$, where $[p,q,l,r]$ and $[p,q'',l'',r'']$ are the ranges associated with positions $[p,q]$ and $[p,q'']$ respectively; otherwise $Right([p,q,l,r]) := [p,n,-,-]$ (Note that $[p,n,-,-]$ is not a "true" range, it is merely used here as an endpoint for the *Right* function).

STEP 5. First we need the following definitions. For each position $[p,q]$ in D_t , we define $gap_t([p,q]) = q'' - q$, where $[p,q'',l'',r''] = Right[p,q,l,r]$. Although, the function gap_t provides us with information of the gaps within D_t, we need to classify these gaps according to their length. Thus, for $1 \leq s \leq k$, we define $GAP(s)$ to be the list of all positions $[p,q]$ of D_t whose $gap_t([p,q])$ satisfy

$$b_{s-1} < gap_t([p,q]) \leq b_s \quad \text{or equivalently} \quad s := \mathcal{C}(gap_t([p,q])).$$

We also define

$$\overline{GAP}(t) := GAP(k) \cup GAP(k-1) \cup \cdots \cup GAP(t)$$

The list $\overline{GAP}(t)$ contains the positions of B_t occurrences, whose distance (gap) to the next B_t occurrence is larger than b_t.

In this step we will consider the borders from largest to smallest. Let B_t be the current border. Assume that we have computed the list D_i and its associated $GAP(i)$ for all $i \geq t$ and we proceed to construct D_{t-1} and its associated $GAP(i), i = k, k-1, ..., t, t-1$ as follows:

We have to consider ranges that fall in one of the following three categories:

(i) The ranges of positions that are not members of D_t but which are members of D_{t-1}. These are ranges of positions $[p,q]$ of T which are not within the first column of an occurrence of B_t, but which are within the first column of an occurrence of B_{t-1}. These are ranges of the form $[p,q,l,t-1]$ (terminating with $t-1$). In order to create D_{t-1}, these ranges have to be added to the list D_t ; their insertion will alter the list GAP, hence we also need to modify the list $GAP(t)$ in order to obtain the list $GAP(t-1)$.

(ii) The ranges of positions that are members of D_t but which are not members of D_{t-1}. These are ranges of positions $[p,q]$ of T which are within the first column of an occurrence of B_t but not within the first column of an occurrence of B_{t-1}. These are ranges of the form $[p,q,t,r]$ (with t in the third position). In order to

create D_{t-1}, these ranges have to be deleted from the list D_t; their deletion will alter the list GAP, hence we also need to modify $GAP(t)$ in order to obtain the list $GAP(t-1)$.

(iii) The ranges of positions that are members of both D_t and D_{t-1}. There is no need to consider these ranges, since they do not cause any change in gapsize.

First we consider all positions $[p,q]$ with range $[p,q,l,t-1]$. Note that $[p,q]$ is in D_{t-1}, but $[p,q]$ is not in D_t (see (i) above). We will obtain D_{t-1} by inserting each of these positions $[p,q]$ in D_t and modify the associated GAP lists accordingly, by means of the following operations:

5.1 Let $[p,q',l',r'] = Left[p,q,l,t-1]$. The insertion of $[p,q]$ in D_t will narrow the gap at position $[p,q']$, (in other words $gap_t([p,q']) \neq gap_{t-1}([p,q'])$), hence we delete $[p,q']$ from the list $GAP(\mathcal{C}(gap_t([p,q']))$.

5.2 The gap size at position $[p,q']$ is $gap_{t-1}([p,q']) = q-q'$, thus we add $[p,q']$ into $GAP(\mathcal{C}(q-q'))$. Note that we round up the gap sizes to the nearest border size – since only borders are candidates for covers.

5.3 Now we consider the gap size at $[p,q]$. Let $[p,q'',l'',r''] = Right[p,q,l,t-1]$. The position $[p,q'']$ is the nearest position to the right of $[p,q]$ in D_{t-1}. Therefore $gap_{t-1}([p,q]) = q''-q$. Thus we add $[p,q]$ into $GAP(\mathcal{C}(q''-q))$. Again we round up the gap size to the nearest border size.

Now we consider all positions $[p,q]$ with range $[p,q,t,r]$. Note that $[p,q]$ belongs to D_t but it is not in D_{t-1} (see (ii) above). We update the lists D and GAP as follows:

5.4 Delete $[p,q]$ from the $GAP(\mathcal{C}(gap_t([p,q]))$. This is done, because the position $[p,q]$ is not a member of D_{t-1}.

5.5 Let $[p,q',l',r'] = Left[p,q,t,r]$ and $[p,q'',l'',r''] = Right[p,q,t,r]$ The gap at $[p,q']$ (in D_{t-1}) has became larger (than the gap in D_t) with the deletion of the position $[p,q]$. The new gap is between positions $[p,q']$ and $[p,q'']$, therefore we place (p,q') into $GAP(\mathcal{C}(q''-q'))$.

After all ranges have been processed (as needed in cases (i) and (ii) above) the border B_{t-1} is a cover if and only if $\overline{GAP}(t-1)$ is empty. Checking whether or not $\overline{GAP}$ is empty can be done in constant time, by keeping $GAP(k), GAP(k-1), ..., GAP(t-1)$ as a doubly linked list.

Theorem 6.1 Algorithm 6.1 checks whether B_i, for all $1 \leq t \leq k$, covers the matrix T in $O(n^2+R)$, where R is the number of ranges.

Proof. Step 1 and Step 2 require $O(n^2)$ operations by Theorems 4.1 and 5.1. The computation of the ranges in Step 3 is shown next (Theorem 6.2) and it requires $O(R)$ time. Step 4 has also a one to one relationship with the number of ranges, also requiring $O(R)$ time. One can easily deduce that steps 5.1-5.5 require $O(1)$ operations each, for adding and deleting items in doubly linked lists; the total number of operations of Step 5 is also bounded by $O(R)$. □

6.2 Computing The Ranges

Recall, that each position $[p, q]$ of T is associated with a range $[p, q, l, r]$ if and only if the position $[p, q]$ is within the first column of an occurrence of all of the following borders $B_l, B_{l+1}, ..., B_r$, but the position $[p, q]$ is not within the first column of an occurrence of either B_{l-1} or B_{r+1}. Let $\mathcal{B}_j$ denote the first column of the border B_j, for some $j = 1, ..., k$. Also, we say that a substring s of x, *terminates at* position j of x if and only if $x[j - m + 1..j] = s[1..m]$, where m is the length of s. The computation of the ranges is based upon the following three simple facts:

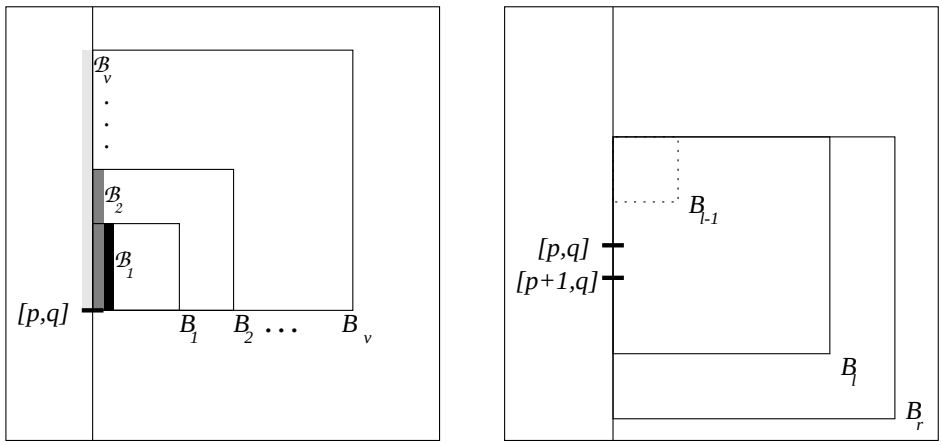
Fact 3. Let $\mathcal{B}_v$ be the first column of the largest border B_v that terminates at position $[p, q]$ of T. Then $\mathcal{B}_1, \mathcal{B}_2, ..., \mathcal{B}_{v-1}$ are all terminating at $[p, q]$.

This follows from the fact that $\mathcal{B}_j$ is a border of $\mathcal{B}_v$ for $j = 1, 2, ..., v - 1$ (see Figure 6-(i)). Let $[p_j, q]$, for $j = 1, 2, ..., v$, be the position that $\mathcal{B}_j$ occurs as a suffix of $\mathcal{B}_v$.

Fact 4. Suppose that there is no (non-empty) border whose first column terminates at position $[p, q]$ of T and there is no (non-empty) border that occurs at $[p + 1, q]$. If $[p, q, l, r]$ is a range for $[p, q]$, then $[p + 1, q, l, r]$ is a range for $[p + 1, q]$.

This follows from the fact that $\mathcal{B}_j$, for all $j = l, l + 1, ..., r$, covers the position $[p, q]$ of T, and since $\mathcal{B}_j$ does not terminate at position $[p, q]$, for all $j = l, l + 1, ..., r$, it also covers $[p + 1, q]$ (See Figure 6-(ii)). Also,no new border occurs at $[p + 1, q]$. Thus, in this case every range for $[p, q]$ is also a range for $[p + 1, q]$.

Fact 5. Let $\mathcal{B}_v$ be the first column of the largest border B_v that terminates at position $[p, q]$ of T. The position $[p+1, q]$ of T is covered by B_j, for some $j = 1, ..., v$, if and only if B_j occurs in one of the following positions $\{[p_j + 1, q], [p_j + 2, q], ..., [p + 1, q]\}$.



(i) **(ii)**

Figure 6

The main steps of the computation of the ranges are as follows: Assume that we have computed all ranges for positions $[i, q]$ for $i = 1, 2, ..., p$, for the q-th column of T, and we proceed to compute the ranges for position $[p+1, q]$ of T. Let B_t be the largest border that occurs at $[p+1, q]$. We consider two cases:

1. There is no border that terminates at $[p, q]$. In this case the borders that cover $[p, q]$, also cover $[p+1, q]$ (Fact 4). Therefore, for each range $[p, q, l, r]$, we create one of the following new ranges: either $[p+1, q, 1, r]$ (when $l \leq t \leq r$) or $[p+1, q, 1, t]$ (when $t > r$) or $[p+1, q, l, r]$ and $[p+1, q, 1, t]$ (when $t < l$). This follows from Fact 4 combined with the fact that B_t occurs at position $[p+1, q]$ of T.

2. Let B_v be the largest border whose first column terminates at $[p, q]$. Let $[p_j, q]$, for $j = 1, 2, ..., v$, be the position that $\mathcal{B}_j$, occurs as a suffix of $\mathcal{B}_v$. For $j = 1, 2, ..., v$, we check whether there is another occurrence of B_j in positions below $[p_j, q]$ and above of $[p+2, q]$. If such an occurrence does not exist, then we insert j into a set named LIST. From Fact 5, the members of the set LIST are the indices of borders that do not cover $[p+1, q]$, hence for each range $[p, q, l, r]$, we create ranges $[p+1, q, l, s_1 - 1]$,$[p+1, q, s_j - 1, s_{j+1} - 1]$, $j = 2, 3, ..., m-1$ and $[p+1, q, s_m - 1, r]$, for all $s_i \in$ LIST : $l \leq s_1 < s_2 < ... < s_m < r$. These ranges now need to be modified to take into the account the occurrence of B_t at $[p+1, q]$, in an identical manner to step 1 above.

The pseudo-code below provides a detailed account of the computation of the ranges.

Algorithm 6.2

begin
  for $j = 1$ **to** n **do**
    if B_t occurs at position $[1, j]$ **then** create the range $[1, j, 1, t]$;
  od
  for $v = 1$ **to** k **do**
    TERMINATE$[p, q] \leftarrow v$, for all positions $[p, q]$ in T that B_v occurs;
  od
  for $q = 1$ **to** n **do**
    for $p = 1$ **to** n **do**
      Let B_t be that largest border that occurs at position $[p + 1, q]$;
      if TERMINATE$[p, q] = \emptyset$ **then**
        for all ranges $[p, q, l, r]$ of position $[p, q]$ **do**
          if $l \leq t \leq r$ **then** create the range $[p + 1, q, 1, r]$;
          if $t > r$ **then** create the range $[p + 1, q, 1, t]$;
          if $t < l$ **then** create the ranges $[p + 1, q, 1, t]$ and $[p + 1, q, l, r]$;
        od
      if TERMINATE$[p, q] = v$ **then**
        for $j = 1$ **to** v **do**
          if B_j does not occur between positions $[p - b_j + 2, q]$ and $[p + 1, q]$ **then**
            add j to the LIST
        od
        for all ranges $[p, q, l, r]$ associated with $[p, q]$ **do**
          Compute $\{s_i \in \text{LIST} \ : \ l \leq s_1 < s_2 < ... < s_m < r\}$;
          Create ranges $[p + 1, q, l, s_1 - 1]$,$[p + 1, q, s_m - 1, r]$ and
                $[p + 1, q, s_j - 1, s_{j+1} - 1]$, $j = 2, 3, ..., m - 1$;
        od
        for all ranges $[p + 1, q, l, r]$ of position $[p + 1, q]$ **do**
          if $l \leq t \leq r$ **then** create the range $[p + 1, q, 1, r]$;
          if $t > r$ **then** create the range $[p + 1, q, 1, t]$;
          if $t < l$ **then** create the ranges $[p + 1, q, 1, t]$ and $[p + 1, q, l, r]$;
        od
    od
  od
end. □

Theorem 6.2 Algorithm 6.2 computes all ranges in $O(n^2 + R)$ time.

Proof. The computation of the list TERMINATE takes at most $O(n^2)$ units of time.

The computation of the set LIST can also be done in $O(n^2)$ units of time (in total) by preprocessing: for each occurrence of B_j we pre-compute the nearest position in the same column that B_j re-occurs; this is similar to step 4 of Algorithm 6.1 and preprocessing requires $O(n^2)$ units of time.

All internal **for** loops go through all of the ranges in a column and the outer loop goes through all columns; since each of the **if** statements requires constant time, the loops require $O(R)$ time. □

6.3 Counting the Number of Ranges

We have seen that there are cases that two or more occurrences of a border can cover a position of T but they lead to just one range. For example, in Figure 6-(i), $B_1, B_2, ...B_v$ all cover $[p, q]$, but they define only one range $[p, q, 1, v]$. In order to be able to count the ranges, we need to identify one of these occurrences, and we choose the one that is the closest to (and covers) the position whose range is in question; we say that these occurrences originate (establish) a range for that position. This will enable us to distinguish between originating and non-originating border occurrences and consequently count the number of ranges accordingly. Formally:

Let $[p, q, l, r]$ be a range of the position $[p, q]$ of a square matrix T. There exists a position $[p', q]$ of T such that

(i) The border B_t of T occurs at $[p', q]$, with t such that $l \leq t \leq r$.

(ii) The position $[p, q]$ is within the occurrence of the border B_t at $[p', q]$ (i.e., B_t covers $[p, q]$).

(iii) There is no position $[p'', q]$ of T, with $p'' > p$ with the above two properties.

We say that the border occurrence at position $[p', q]$ *originates* the range $[p, q, l, r]$ at $[p, q]$. In Figure 6-(i), assuming that there no other borders occurring, the occurrence of B_1 (that terminates at $[p, q]$) originates the range $[p, q, l, v]$.

The lemma below establishes the criteria under which, a border occurrence at a position originates a range.

Lemma 6.3 Suppose that the borders B_t and B_s of T occur at the positions $[c, q], [e, q]$ of T and that the occurrence of B_s at $[e, q]$ originates a range $[p, q, l, r]$ for position $[p, q]$. The occurrence of B_t at $[c, p]$ originates a range $[p, q, l', r']$ for position $[p, q]$ if and only if

$$b_t > p - c \quad \text{and} \quad s < l' \tag{6.1}$$

Proof. The border B_t at $[c, p]$ originates a range at $[p, q]$ if and only if it covers $[p, q]$, that is $[p, q]$ is a position within the occurrence of B_t at $[c, q]$ (see Figure 7-(i)), hence $b_t > p - c$.

The occurrence of B_t at $[c,q]$ implies that the borders $B_{l'}, B_{l'+1}, ..., B_t$ cover $[p,q]$, where $l' = \mathcal{C}(p-c)$. Similarly the occurrence B_s at $[e,p]$ implies that the borders $B_l, B_{l+1}, ..., B_s$ cover $[p,q]$, with $l = \mathcal{C}(p-v)$. Thus B_t originates a new range at $[p,q]$ if and only if $s < l' = \mathcal{C}(p-c)$. □

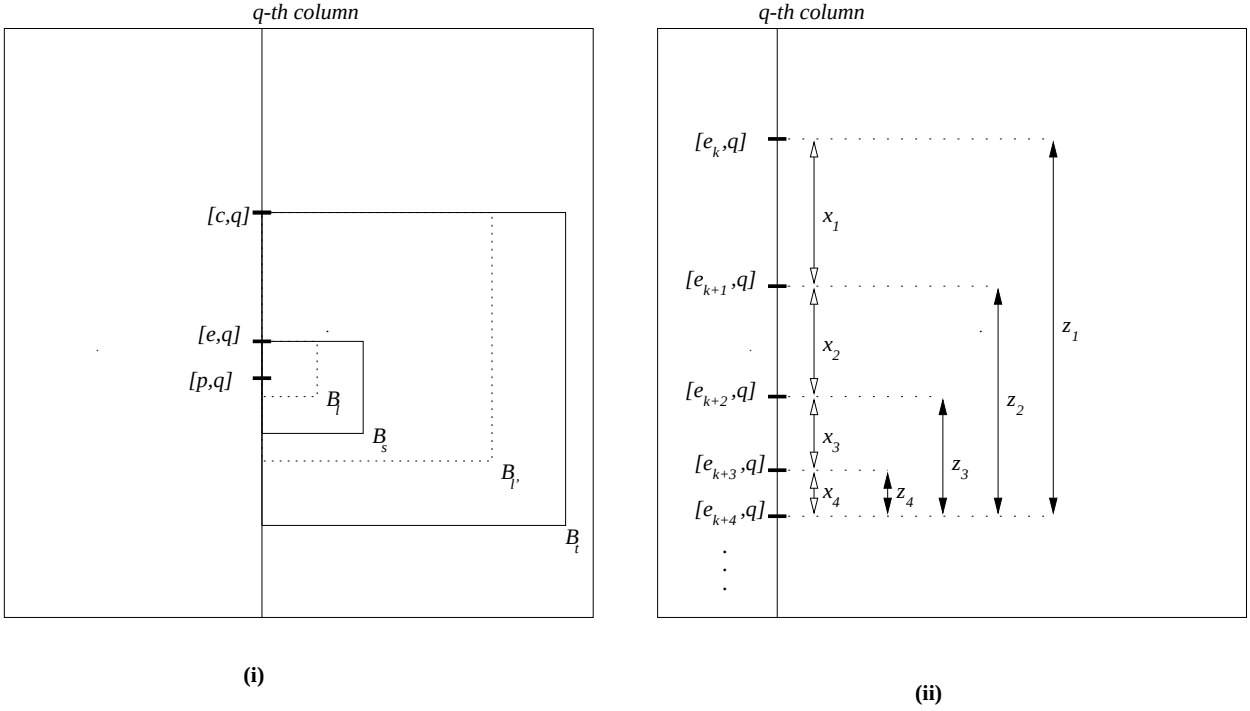


Figure 7

The following lemma establishes an upper bound on the number of ranges originated by a chain of border occurrences.

Lemma 6.4 Let $[e_1,q], [e_2,q], ..., [e_m,q]$ be the positions of the q-th column of T that a non-empty border of T occurs. Let $d_j = e_{j+1} - e_j$, $j = 1, ..., m-1$. If the occurrence at $[e_j,q]$ originates a range for $[e_{j+1},q]$ for every $i = 1, ..., m-1$, then

$$d_j \;\geq\; \frac{3}{2}\, d_{j+1} \tag{6.2}$$

and the total number of ranges originated by these occurrences is $O(n)$.

Proof. By induction on the number of occurrences. One can show that it holds for $j = 5$. Assume that positions $[e_{k+1},q], [e_{k+2},q], ..., [e_m,q]$ satisfy (6.2); we will next show $[e_k,q], [e_{k+1},q], ..., [e_m,q]$ also satisfy (6.2)

Let x_j, $j = 1,2,3,4$, be the string that starts at position $[e_{k-j+1},q]$ and has length d_{k-j+1} (see Figure 7-(ii)). From Lemma 6.3 and (6.1) follows that $|x_j| > |x_{j+1}|$ and therefore x_{j+1} is a prefix of x_j, for $j = k, ..., m-1$. Using the facts that x_2 is a

prefix of x_1, x_3 is a prefix of x_2 and x_4 is a prefix of x_3 (see Figure 7-(ii)), one can find strings c, f, g, h such that

$$x_1 = cfgh$$
$$x_2 = cfg$$
$$x_3 = cf$$
$$x_4 = c$$

Also let

$$z_1 = cfghcfgcfc$$
$$z_2 = cfgcfc$$
$$z_3 = cfc$$
$$z_4 = c$$

be the substrings of the q-th column of T starting at positions $[e_k, q], [e_{k+1}, q], [e_{k+2}, q]$ and $[e_{k+3}, q]$ respectively and terminating at $[e_{k+4}, q]$

Case of $g < c$: From Lemma 6.3 and (6.1) follows that z_3 is a prefix of z_2, hence we have that $c = gs$ for some string s. From z_1 and z_2 above one can see that $x_2 s$ is a prefix of hx_2x_3. If $|h| \leq |x_2|/2$ then we have that

$$x_2 = h^{\lambda}h', \text{ for some prefix } h' \text{ of } h \text{ and integer } \lambda \geq 2$$

which implies $c = h^{\mu}h''$, for some integer $\mu < \lambda$ and some prefix h'' of h and which in turn implies at least another border occurrence aoriginating a range at the position $[e_{k+2} - |c| + 1, q]$ between $[e_{k+1}, q]$ and $[e_{k+2}, q]$, a contradiction. Thus we have

$$|h| > |x_2|/2 \quad \rightarrow \quad d_k = |x_1| = |x_2| + |h| > \frac{3}{2}\, d_{k+1}$$

Case of $g \geq c$. This is similar to the case above.

We have shown that (6.2) holds and furthermore one can observe that border occurrences at positions $[e_1, q], [e_2, q] ..., [e_j, q]$ originate j ranges to positions between $[e_j, q]$ and $[e_{j+1}, q]$, or jd_j in total for that region. The total number of ranges is

$$\Sigma_{j=1}^{m} j d_j \leq \Sigma_{j=1}^{m} j (\frac{2}{3})^{j-1} d_1 = O(n) \qquad \square$$

Theorem 6.5 The cardinality of the list of ranges created in each column is $O(n)$. Thus the total number of ranges $R = O(n^2)$.

Proof. Let $[e_1, q], ..., [e_m, q]$ be the positions of the q-th column of T that a non-empty border of T occurs. The worst case arises when the occurrence at $[e_i, q]$, $i = 1, ..., m-1$ originates ranges to all the positions below it, i.e., $[e_j, q]$, $j = i+1, ..., m-1$. Thus from Lemma 6.4, the number of ranges is at most $O(n)$ for each column of T or $R = n^2$. □

Theorem 6.6 Algorithm 6.1 computes all square covers of a square matrix T in linear time.

Proof It follows from Theorem 6.5 and Theorem 6.1. □

7. Conclusion and Open Problems

The Aho-Corasick Automaton depends on the alphabet; it is an open question whether the all-covers of a square matrix can be computed in linear time independent of the alphabet. A natural extension of the problems presented here is the design of algorithms for computing rectangular covers for rectangular matrices. Another extension of the above problem is that of computing *approximate* covers that allow the presence of errors.

Also of interest is the PRAM complexity of both prefix string matching and all covers problems (on square and rectangular matrices). An optimal PRAM algorithm for computing the smallest cover was given in [IK96] but the optimal computation of all covers is still an open problem (see [IK96b]). In particular the PRAM relationship between the prefix string matching problem and the computation of the diagonal failure function; the PRAM computation of the failure function (see [GP94]) was done using the prefix string matching algorithm, the reverse way of the methods used here.

8. References

[ABF92] A. Amir, G. Benson and M. Farach, Alphabet independent two dimensional matching, *Proc. 24th ACM Symposium on Theory of Computing,* 59-68, 1992

[ABF94] A. Amir, G. Benson and M. Farach, Alphabet independent two dimensional matching, SIAM Journal of Computing 23 (2): 313-323 (1994)

[AC75] A.V. Aho and M.J. Corasick, Efficient string matching, *Comm. ACM,* Vol 18, No 6, 333-340, 1975

[AE93] A. Apostolico and A. Ehrenfeucht, Efficient detection of quasiperiodicities in strings, *Theoret. Comput. Sci,* 119, 247-265, 1993

[AFI91] A. Apostolico, M. Farach and C.S. Iliopoulos, Optimal superprimitivity testing for strings, *Inform. Process. Lett.* 39, 17-20, 1991

[BBIP94] A. M. Ben-Amram, O. Berkman, C.S. Iliopoulos and K. Park, The subtree max gap problem with application to parallel string covering, *Proc. 5th ACM-SIAM Symp. Discrete Algorithms,* 501-510, 1994

[Br92] D. Breslauer, An on-line string superprimitivity test, *Inform. Process. Lett.* 44, 345-347, 1992

[Br95] D. Breslauer, Testing string superprimitivity in parallel, *to appear in Inform. Process. Lett.*

[BV94] O. Berkman and U. Vishkin, Finding level-ancestors in trees, *Journal of computer and System Sciences,* Vol 48, 214-230, 1994

[DS] A.M. Duval and W.F. Smyth, Covering a circular string with substrings of fixed length, to appear in the *Int. Journal of Foundations of Computer Science.*

[HT84] D. Harel & R. E. Tarjan, Fast Algorithms for finding nearest common ancestors, in *SIAM J. Computing* 13 (2) (1984) 338-355

[GP94] L. Gasieniec and K. Park, Work-time optimal parallel prefix matching, *Proc 2nd European Symposium on Algorithms*, Utrecht, (1994)

[G93] R. Giancarlo, The suffix tree of a square matrix, with applications, *ACM-SIAM Proc. 4th Symposium on Discrete Algorithms,* 402-411, 1993

[G95] R. Giancarlo, A generalization of the Suffix Tree to Square Matrices with Applications, *SIAM J. Computing* Vol 24, No. 3, 520-562, (1995)

[IK96] C. S. Iliopoulos and M. Korda, Optimal parallel superprimitivity testing for square arrays, to appear in *Par. Proc. Letters*

[IK96-b] C. S. Iliopoulos and M. Korda, Parallel two dimensional covering, in Proc. 7-th Australasian Workshop on Combinatorial Algorithms, Magnetic Island, University of Sydney, pp62-74 (1996)

[IMP93] C.S. Iliopoulos, D.W.G. Moore and K. Park, Covering a string, *Proc 4th Symp. Combinatorial Pattern Matching, Lecture Notes in Computer Science,* Vol 684, 54-62, 1993

[IP94] C.S. Iliopoulos and K. Park, An optimal $O(\log \log n)$ time algorithm for parallel superprimitivity testing, *Journal of the Korea Information Science Society,* Vol 21, No 8, 1400-1404, 1994

[KMP77] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings , *SIAM Journal of Computing,* Vol 6, 322-350, 1977

[ML84] G.M. Main and R.J. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string, *Journal of Algorithms,* Vol 5, 422-432, 1984.

[MS94] D.W.G. Moore and W.F. Smyth, Computing the covers of a string in linear time, *Proc. 5th ACM-SIAM Symp. Discrete Algorithms,* 511-515, 1994

[SV88] B. Scheiber and U. Vishkin, On finding lowest common ancestors: simplification and parallelisation, *SIAM Journal of Computing,* Vol 17, No 6, 1253-1262, 1988

[W73] P. Weiner, Linear Pattern Matching Automaton, *Proc. of the 14 IEEE Symposium on Switching and Automata Theory Symposium*, p 1-11, (1983)