JOURNAL OF COMPLEXITY **15**, 72–127 (1999)
Article ID jcom.1998.0495, available online at http://www.idealibrary.com on IDEAL®

On-line Construction of Two-Dimensional Suffix Trees*

Raffaele Giancarlo† and Daniela Guaiana‡

Dipartimento di Matematica ed Applicazioni, Università di Palermo, Via Archirafi, 34, 90123 Palermo, Italy

Received June 26, 1997

DEDICATED TO ZVI GALIL IN HONOR OF HIS 50TH BIRTHDAY

We say that a data structure is built *on-line* if, at any instant, we have the data structure corresponding to the input we have seen up to that instant. For instance, consider the suffix tree of a string $x[1, n]$. An algorithm building it *on-line* is such that, when we have read the first i symbols of $x[1, n]$, we have the suffix tree for $x[1, i]$. We present a new technique, which we refer to as *implicit updates*, based on which we obtain: (a) an algorithm for the *on-line* construction of the Lsuffix tree of an $n \times n$ matrix A—this data structure is the two-dimensional analog of the suffix tree of a string; (b) simple algorithms implementing primitive operations for **LZ1-type** *on-line lossless* image compression methods. Those methods, recently introduced by Storer, are generalizations of **LZ1-type** compression methods for strings. For the problem in (a), we get nearly an order of magnitude improvement over algorithms that can be derived from known techniques. For the problem in (b), we do not get an asymptotic speed-up with respect to what can be done with known techniques; rather we show that our algorithms are a natural support for the primitive operations. This may lead to faster implementations of those primitive operations. To the best of our knowledge, our technique is the first one that effectively addresses problems related to the *on-line* construction of two-dimensional suffix trees.

1. INTRODUCTION

The suffix tree T_x of a string x [28, 39] is a very useful data structure with applications ranging from string matching [4] to computational

* Work supported in part by Grants from the Italian Ministry of Scientific Research and from the Italian National Research Council. A preliminary version of this paper has been presented at the 5th European Symposium on Algorithms, 1997.

† Part of this work was done while visiting Bell Labs of Lucent Technologies, U.S.A. E-mail: raffaele@altair.math.unipa.it.

‡ E-mail: daniela@altair.math.unipa.it.

0885-064X/99 $30.00

biology [6, 24]. For the *RAM* model of computation [1], we know how to build it in $O(n)$ time and space [7, 9, 28, 39], where n is the length of the string x, and the existence of algorithms for its construction in quasi-real-time has been investigated [24, 35]. For the *PRAM* model of computation [13], we also know how to build it optimally [5, 10, 19, 32].

Ukkonen [38] has recently added to the above collection a simple and elegant linear time algorithm for the *on-line* construction of the suffix tree. The algorithm is *on-line* because, after i steps, it knows only the first i symbols of x and it has built the suffix tree for $x[1, i]$. The algorithm by Ukkonen [38] is based on a clever observation about the encoding of information on the edges of the suffix tree. Due to its simplicity, the algorithm is very useful in application areas where the string x is provided *on-line* and, after each new symbol is read, we need to know the suffix tree for the string "we have seen so far." One area is given by the fast implementation of *on-line lossless* **LZ1-type** text compression methods [12, 25, 30]. A text compression method is classified as **LZ1** (see [36]), if it uses the basic ideas presented by Ziv and Lempel in their seminal work on text compression [26, 40].

The state of the art about the construction of two-dimensional data structures that are analogous to the suffix tree of a string is not so rich. Those data structures find natural applications in low level image processing [31], image data compression [37], and visual data bases [22]. Let A be an $n \times m$ matrix with entries defined over a finite alphabet Σ and assume that $n \geqslant m$. Informally, a two-dimensional analog of the suffix tree for A is a tree data structure storing all submatrices of A. Let's call this data structure *an index* for A. It must support a wide variety of queries (see [14] for a list), the most important of which is the following: given a pattern matrix $PAT[1 : s_1, 1 : s_2]$, find all submatrices of A that are equal to PAT. Those submatrices are called *occurrences* of PAT in A. An index can support this kind of query in time that depends *only* on the size of PAT and the number of occurrences of PAT in A.

Early results investigating the construction of index data structures for matrices are reported in [18]. Recently, it has been shown that any index requires $\Omega(nm^2)$ time to be built [14] and efficient algorithms have been proposed both for the *RAM* and *PRAM* model of computation [8, 14, 16]. All those algorithms are a polylogarithmic factor away from optimal time and work. Because of the lower bound stating that it is not possible to build index data structures for an $n \times m$ matrix in time "close" to nm, those data structures may not have wide applicability. Fortunately, for the important special case in which A is an $n \times n$ matrix, much better results are available. Indeed, the Lsuffix tree of a square matrix has been recently proposed [15, 17]. It is an index that compactly stores all *square submatrices* of A. An example of this data structure is provided in Fig. 1 and

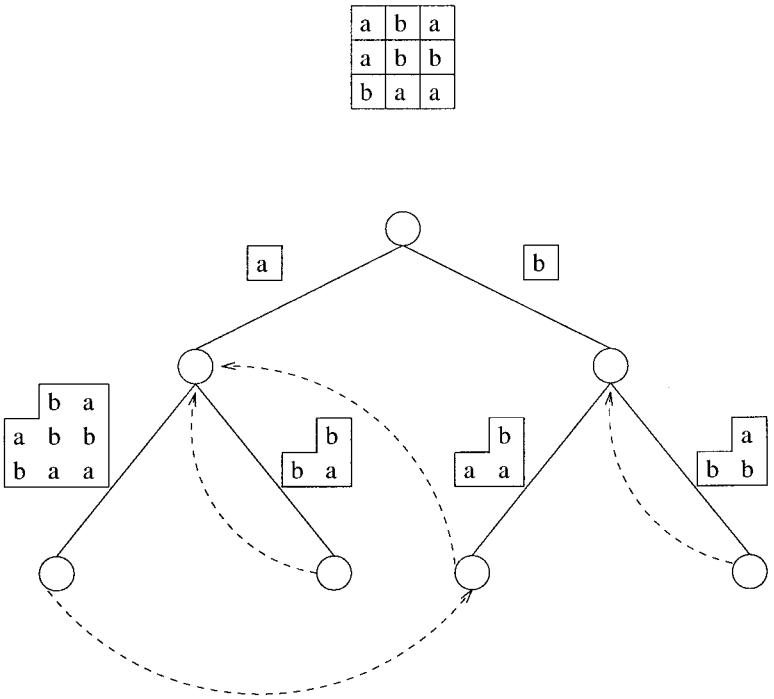

FIG. 1. The Lsuffix tree of the matrix shown at the top of the figure. Dotted lines are suffix links departing from each leaf.

a formal definition is given Subsection 2.1. It can be built in $O(n^2 \log n)$ time and takes $O(n^2)$ space. Moreover, it can support many queries in optimal time (see [15] for a list).

Both for the case in which the index represents all submatrices of the given matrix and for the special case of the Lsuffix tree, all the algorithms we have mentioned so far are *off-line*, i.e., they must know the matrix ahead of time. We present a new technique, which we refer to as *implicit updates*, that allows us to build the suffix tree of a square matrix *on-line* and to implement primitive operations for **LZ1-type** *on-line lossless* image compression methods [37] (see also [27, 33]). We now give a detailed description of our contributions. Our model of computation is the *RAM* [1].

1.1. *On-line Suffix Trees for Square Matrices*

Let $W_p = A[1 : p, 1 : p]$. An $n \times n$ matrix A is read *on-line* as a sequence of matrices $W_1, W_2, ..., W_n$. At time p, we know W_p and nothing else about A.

At time $p+1$, we get W_{p+1} by getting in input subrow $A[p+1, 1:p+1]$ and subcolumn $A[1:p+1, p+1]$. We remark that the particular input ordering we have chosen for A makes the description of the algorithms easier. However, W_p could be analogously defined when A is given in input in row or column major order and our algorithms would work also in those cases. Let LT_p be the Lsuffix tree for matrix W_p. Again, it is a data structure that compactly stores all square submatrices of W_p (see Fig. 1). Since A is given *on-line* as a sequence of matrices W_p, $1 \leqslant p \leqslant n$, the Lsuffix tree for A can be built as a sequence of trees $LT_1, LT_2, ..., LT_n$. We have new algorithms such that:

(a1) The time taken to build the entire sequence of trees is $O(n^2 \log^2 n)$.

(a2) Assume that, when we have seen only a part of A, say W_p, we are given a pattern $PAT[1:m, 1:m]$ and we want to check whether PAT occurs in W_p. We can answer that query in $O(m^2 \log |\Sigma|)$ time. We can further extend that query to report all *occ* occurrences of PAT in W_p in $O(m^2 \log |\Sigma| + occ \log p)$ time.

We point out that we can build the Lsuffix tree for A *on-line* by a simple modification of the *off-line* algorithm in [15]. However, we would get an $O(n^3 \log n)$ time algorithm. Therefore, the new *on-line* algorithm in **(a1)** is nearly an order of magnitude improvement with respect to what is already available [15] and a $\log^2$ factor away from optimal time.

As for the analogy between the *on-line* algorithm by Ukkonen [38] for the construction of the suffix tree of a string and our technique, we point out that we make use of Ukkonen's clever observation mentioned earlier, but that is only a very small part of our technique. Ukkonen's algorithm does not seem to be extendible to the on-line construction of a suffix tree of a matrix. Indeed, when LT_p becomes LT_{p+1}, some leaves of LT_p can be transformed into internal nodes. This will not happen in the case of strings and the proof of linearity of Ukkonen's algorithm is heavily based on that fact. Getting around this difficulty turns out to be a challenging technical task.

1.2. **LZ1-Type** *Compression Methods for Images*

Storer [37] has recently proposed a new family of methods for *on-line lossless* compression of images. They are generalizations to images of **LZ1-type** compression methods for text (for discussions of those latter methods, the reader is referred to [26, 36, 40]). Related work on extending **LZ1-type** methods from text to images is presented in [27, 33]. We briefly mention results quantifying how well those new methods are expected to compress images. Sheinwald, Lempel, and Ziv [27, 33] show that, asymptotically,

methods in this new family provide an optimal compression ratio with respect to what can be done by "finite state encoders." Experiments by Storer [37] show that the compression ratio obtained by those methods is 70% of the best available compression ratio, obtained for instance via JBIG. Those results are considered encouraging [37]. Indeed, no optimization of the many parameters involved in the implementation of this new type of method has been done yet.

Here we confine our attention to the primitive operations and data structures needed to implement the compression methods in the new family and provide efficient algorithms for those operations. We describe the simplest method in this new family describing the primitive operations needed to implement it. We remark that the same primitive operations can be used to support all other methods in the family.

Assume that the image is an $n \times n$ matrix A, with entries drawn from a finite alphabet Σ. Assume that we want to compress A on-line, i.e., while the matrix is given as input. In analogy with **LZ1-type** methods for strings [12, 30, 36, 40], we use a window that slides over the matrix sweeping all of it. Consider the "L-shaped" window of height h covering part of the matrix (see Fig. 2). In analogy with the sweeping order described in [33], the window slides along the main diagonal of A and, initially, it is an $h \times h$ square covering the upper left corner of A. Let $h(p) = p + h - 1$ and let $F_{h(p)}$ be the "L-shaped" window that starts in row and column p and ends

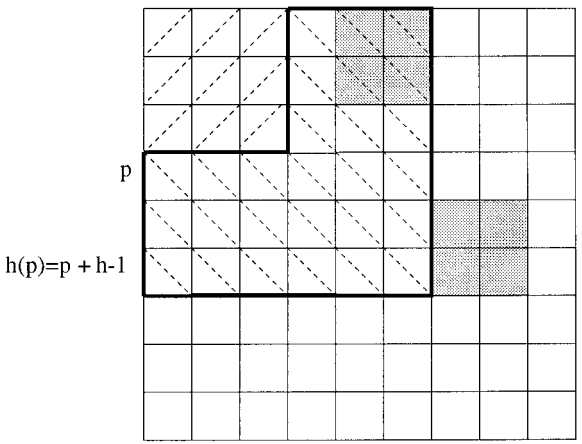

FIG. 2. The L-shaped region within bold lines is the window F_6 with $h = 3$. The 3×3 matrix in the upper left corner of A is known, but it is not accessible any longer. Notice that this latter matrix plus the window give W_6. The Lsuffix tree for F_6 (not shown) represents all square submatrices of A that are also in the window. The light shaded part outside the window is a maximal match for the neighbor position (7,5) of F_6. It matches the light shaded part in the window.

in row and column $p+h-1$ of A (see Fig. 2), for $1 \leqslant p \leqslant n-h+1$. We refer to the entries on row and column $p+h$ of A, just outside the border of $F_{h(p)}$, as the *neighbors* of $F_{h(p)}$.

A is given *on-line*. Assume that when the window has reached row $h(p)$ of A, we have seen A up to row $h(p+h)$, i.e., $W_{h(p+h)}$ is available. Moreover, assume that we have compressed $W_{h(p)}$. The next "compression step" works as follows. For each neighbor position of $F_{h(p)}$, find the largest square matrix originating in that position and that appears within the window (one such match is reported in Fig. 2). Let $Match(F_{h(p)})$ be the query that reports all those maximal matrices. The area S of the matrix A covered by the results of the query (see Fig. 3) is encoded in compressed form: we replace the matrices in S by pointers to their occurrences in the window.

There are several methods and strategies currently under investigation to generate the encoding from the results of $Match(F_{h(p)})$ and, probably, the best method can be identified only experimentally (see [37]). Once we have compressed the area outside the window, we slide it by some amount and repeat the process.

The compression method we have outlined has the following problem as one of its main computational bottlenecks:

- Maintain a data structure that represents all square submatrices of A that are within the window $F_{h(p)}$. Since the window slides over A, the data structure must be dynamically changed. Moreover, it must support the query *Match*.

From now on, as far as data compression is concerned, we concentrate on the above subproblem. The data structure that seems to naturally fit the description is the Lsuffix tree of a square matrix [15]. It can be defined

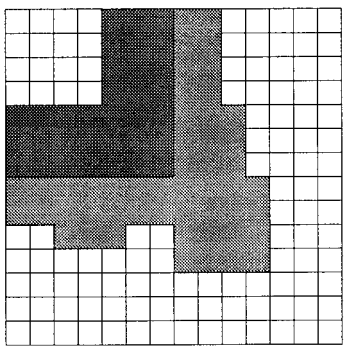

FIG. 3. The dark shaded part is the window F_7, with $h=3$. The light shaded part is the area S covered by the (possibly overlapping) matrices reported by $Match(F_7)$.

for the part of A covered by the window. We have new algorithms such that:

(b1) While the window slides over A, they can maintain the Lsuffix tree of the window in a total of $O(n^2 \log^2 n)$ time. For each $F_{h(p)}$, $1 \leqslant p \leqslant n-h+1$, the total space used is $O(|F_{h(p)}|) = O(h^2 + ph)$. $Match(F_{h(p)})$ can be answered in $O(S \log^2 n)$ time, where S is the total area covered by the matrices in $Match(F_{h(p)})$ (see Fig. 3).

We point out that we can use other types of windows getting results analogous to the ones reported in **(b1)** for the L-shaped window of Fig. 2. In particular, we can use rectangular windows of finite size that sweep matrix A in row or column major order or according to the Peano–Hilbert space filling curve [20, 29].

Our contribution in this area is best explained by drawing an analogy with text data compression. Rodeh, Even, and Pratt [30] obtained the first linear time algorithm implementing the "sliding window" text compression algorithm by Lempel and Ziv [26]. The method consists of a finite window that slides over the text and, through pattern matching queries involving the text covered by the window, a compressed version of the text is obtained. Two suffix trees are needed by the algorithms in [30]. This approach has the drawback of wasting space (two suffix trees) and of a complicated pattern matching query. Fiala and Green [12] devised an algorithm that uses only one copy of the suffix tree and that supports a simple implementation of the pattern matching query. However, this approach implies some nontrivial bookeeping for the dynamic maintenance of the suffix tree and it is linear-time only for windows of constant size. Finally, Larsson [25] has recently presented a very natural algorithm for the implementation of the "sliding window" text compression method retaining all the advantages of Fiala and Green's, being somewhat simpler to describe and linear-time for all window sizes. Fundamental for the algorithm by Larsson is the algorithm by Ukkonen for the *on-line* construction of the suffix tree.

As for image data compression, we could use the ideas in [30] for text compression together with the algorithm for the *off-line* construction of the Lsuffix tree [15] to implement the "sliding window" and the query $Match(F_{h(p)})$ with the same time bounds as in **(b1)**. However, this approach would have the same type of problems as the linear time algorithm for text compression by Rodeh *et al.* [30]. Here, instead, we propose an approach that is similar to the one by Larsson for text data compression. Once we have the algorithm for the *on-line* construction of the Lsuffix tree, it lends itself very naturally to be extended to implement the "sliding window" and the query *Match*.

1.3. *Techniques*

We now briefly describe the new technique, which we refer to as *implicit updates*, that consists of the following. Let LT_p be the Lsuffix tree for W_p. When W_p becomes W_{p+1} by getting in input a subrow and subcolumn of A, we need to transform LT_p in LT_{p+1}. As discussed in Section 3, that transformation involves the following: (i) some leaves and internal nodes of LT_p generate new leaves and internal nodes which are part of LT_{p+1}, (ii) the remaining leaves and internal nodes of LT_p will need some adjustments, and after that, they can be part of LT_{p+1}. *Implicit updates* is based on a suitable encoding of the needed adjustments. Moreover, the technique allows us to process the leaves and internal nodes in (i) *without touching* the ones that can be adjusted implicitly. This is a nontrivial task since the leaves and internal nodes in (i) and (ii) do not partition LT_p in an orderly fashion.

We point out that, for *two-dimensional dynamic dictionary matching* (see for instance [3, 15, 21] for problem definitions and algorithms), there are techniques for the dynamic maintenance of two-dimensional data structures. *Implicit updates* solves a set of technical problems distinct from the ones solved by the techniques in two-dimensional dynamic dictionary matching. Indeed, it is an interesting open problem to extend those latter techniques to the *on-line* problems we address in this paper.

1.4. *Organization*

The remainder of this paper is organized as follows. In Section 2 we briefly recall from [15] the definition of Lsuffix tree of a matrix and we state some facts that are used throughout the paper. In Section 3 we present the structural changes that must be handled when LT_p is transformed in LT_{p+1}. The algorithm taking care of those changes consists of two phases: *Frontier Expansion* and *Internal Structure Expansion*. Frontier Expansion identifies all leaves of LT_p, that are transformed into internal nodes. It turns out that those leaves can be quickly identified while skipping the processing of leaves that cause no changes in LT_p. However, implementation of this simple observation turns out to be nontrivial. An outline of Frontier Expansion is given in Section 4 and the technical details are presented in Section 6. Internal Structure Expansion identifies the submatrices of W_{p+1} that cause changes in LT_p affecting edges and internal nodes. Here the idea is quite simple. Candidate submatrices are kept in a priority queue. The submatrix of highest priority is extracted from the queue and it is established whether one of its suitably chosen submatrices causes the mentioned changes. If it does, the changes are carried out and the suitably chosen submatrix is inserted in the queue. Else, it is discarded. Section 5 gives an outline of *Internal Structure Expansion* while the details

are presented in Section 7. Section 8 outlines how to perform efficiently "comparisons" of pieces of matrices. Those primitives are used by the algorithms in the previous sections, but their detailed description can be deferred. Section 9 summarizes the algorithm for the on-line construction of the Lsuffix tree for a matrix and the corresponding time analysis. Sections 10 and 11 present the pattern matching applications. The next two sections give the algorithms for the image data compression method introduced in Subsection 1.2 and the last section contains some concluding remarks and open problems.

2. BASIC NOTIONS

In this section, we define the Lsuffix tree of an $n \times n$ matrix A. We also give some notions that are needed in later sections.

2.1. *The LSuffix Tree of a Matrix*

Intuitively, the Lsuffix tree of a matrix is a data structure that compactly stores all square submatrices of A. The presentation of this data structure is organized as follows. We first define Lstrings, which are a suitable linear representation of square matrices. Then, we define the Lsuffix tree for one Lstring. The Lsuffix tree for a matrix A will be defined as the Lsuffix tree for a collection of Lstrings. The material in this subsection is presented in a more detailed form in [15].

• *Lstrings.* We need a suitable linear representation of matrices, i.e., a representation of matrices as "strings." Rather than being formal, we define it in intuitive terms. (This representation, denoted Lstring, has been introduced in [15] and, with a different formalism, by Amir and Farach [2].) Given $A[1:n, 1:n]$, consider the string obtained by concatenating row $A[i, 1:i-1]$ with column $A[1:i, i]$, the first taken from left to right and the second taken from top to bottom. This string can be thought of as an atomic item, i.e., a character, and we refer to it as the ith *Lcharacter* of A. For the matrix in Fig. 4a, an example is given in Fig. 4b. We denote those characters as Lcharacters because of the shape they have when they are seen in terms of subrows and subcolumns of A, as it is shown in Fig. 4c. Notice that each Lcharacter is composed of "subatomic parts" that are the characters of Σ. Now, consider the string obtained by concatenating the first, second, ..., nth Lcharacter of A. We get a representation of A in terms of a string of L-shaped characters, which we call *Lstring*. Again, an example is given in Fig. 4d. We also need for Lstrings the notion of a chunk, which is the analog of the notion of substring for strings. A *chunk* is obtained by writing down the Lcharacters of A in one dimension, from

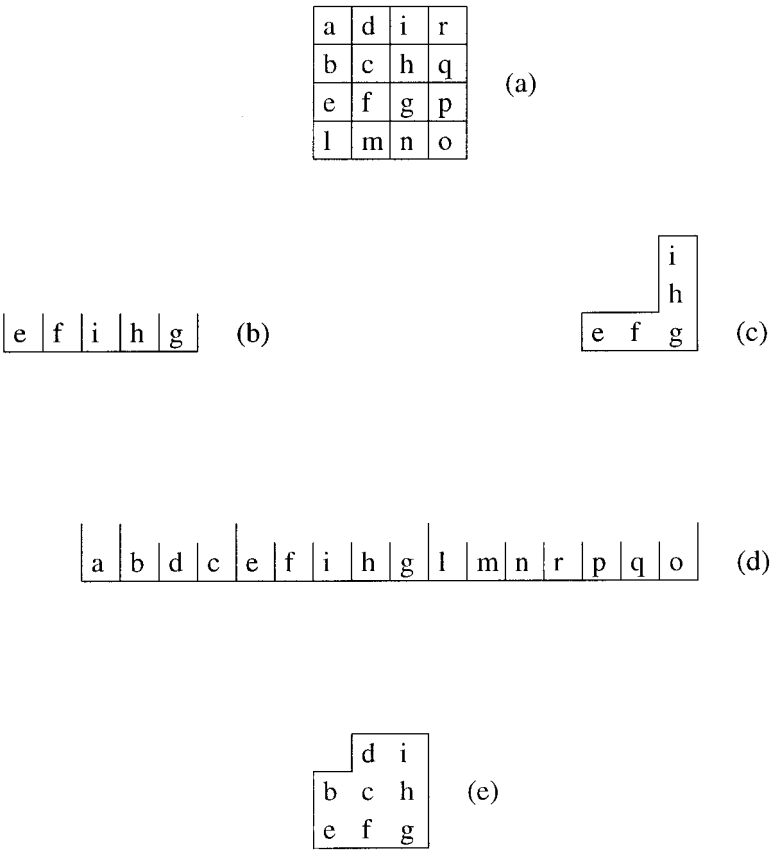

FIG. 4. (a) A matrix; (b) the third Lcharacter of the matrix in (a); (c) its representation in terms of subrows and subcolumns of the matrix in (a); (d) the Lstring corresponding to the matrix in (a), where each Lcharacter is delimited by a tall bar; (e) a chunk of the matrix in (a).

the kth to the jth, $k \leqslant j$. An example is given in Fig. 4e. As it is evident from Fig. 4e, chunks are intended to represent L-shaped pieces of matrices centered around the main diagonal. This is in contrast with Lstrings that are intended to represent matrices.

The notion of Lstring just outlined can be stated formally by suitably defining that object as a string over an alphabet of Lcharacters $L\Sigma = \bigcup_{i=1}^{\infty} \Sigma^{2i-1}$ (for details see [15]). Two Lcharacters are *equal* if and only if they are equal as strings over Σ. Given two Lcharacters La and Lb, we say that La is *lexicographically smaller than or equal to* Lb if and only if the string corresponding to La is lexicographically smaller than or equal to that corresponding to Lb (the two strings are of the same length). In what

follows, given a matrix B, we use B to denote also the Lstring corresponding to it.

• *Suffix and Prefix of a Matrix.* For $1 \leqslant j \leqslant n$, submatrix $A[j:n, j:n]$ is the jth *suffix* of A and submatrix $A[1:j, 1:j]$ is the jth *prefix* of A (see Figs. 5a–5b). Note that any square submatrix of A whose upper left corner lies on the main diagonal of A can be described as a prefix of a suffix of A (see Fig. 5c).

• *Tries for Matrices.* Once we have a way of representing matrices in one dimension, i.e., as strings, it is easy to define tries that represent them (see [23] for the definition of a trie). Again, rather than being formal, we resort to an example. Consider three matrices X, Y, and Z (see Fig. 6a). We can represent those matrices with a tree by letting matrices that have prefixes in common share the same path in the tree (see Fig. 6b). We can label the arcs of this tree with Lcharacters, as shown in Fig. 6b. In general, a trie representing a set of matrices can be defined as a trie over the alphabet $L\Sigma$, representing the set of Lstrings corresponding to the matrices. Tries for matrices (and therefore Lstrings) can have nodes of outdegree one (in analogy with tries for strings—see Fig. 6b). We can compact them by compressing chains of nodes of outdegree one. Labels on the edges of the compressed structure are now chunks (see Fig. 6c for an example).

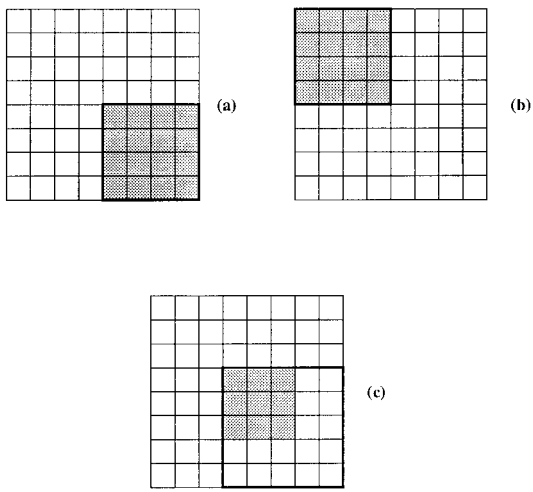

FIG. 5. (a) the fifth suffix of a matrix: (b) the fourth prefix of a matrix; (c) the third prefix of the fifth suffix of a matrix.

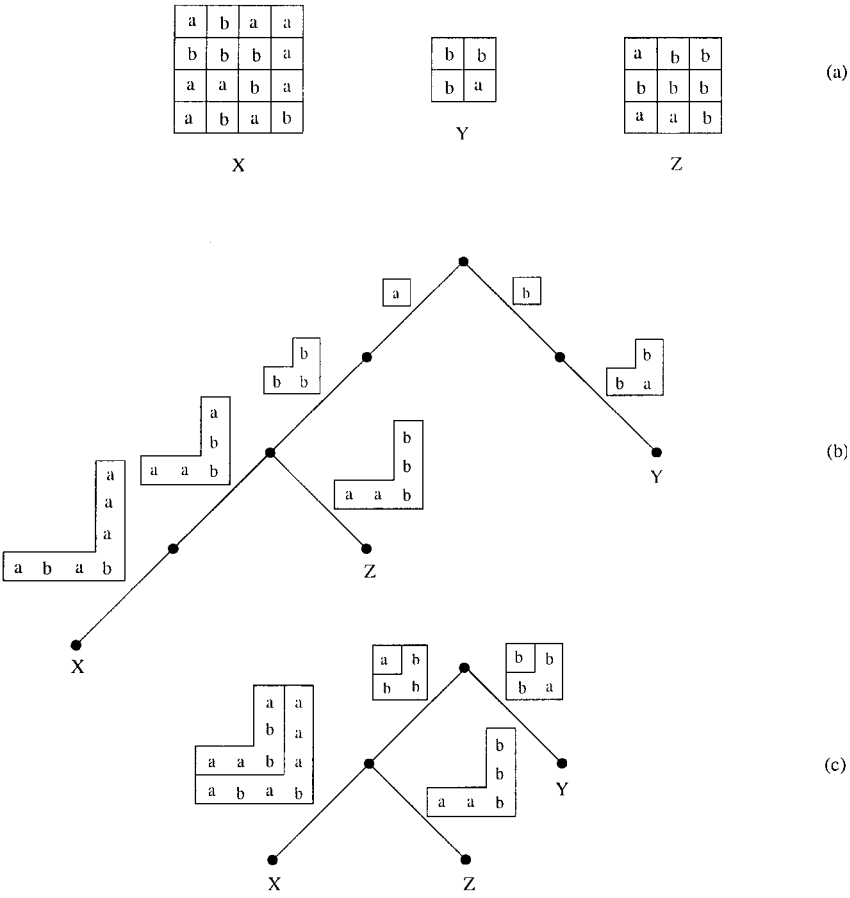

FIG. 6. (a) Three matrices X, Y, and Z, and (b) their trie; (c) the compact version of the trie.

• *The Lsuffix Tree.* We first define the Lsuffix tree for an Lstring, then extend this notion to become the Lsuffix tree of a matrix. We anticipate that the definitions given here are somewhat different than the ones originally given in [15]. We also discuss those differences.

The *Lsuffix tree* for the *Lstring corresponding to* matrix $A[1:n, 1:n]$ is a compacted trie over the alphabet $L\Sigma$ representing all suffixes of $A[1:n]$.

Let us number each diagonal of A by d, if its elements are $A[i, j]$, with $i-j=d$, $0 \leqslant |d| < n$. Let A_d be the square submatrix of A whose main diagonal is d (see Fig. 7).

The *Lsuffix tree for matrix A* is the Lsuffix tree of the Lstring corresponding to A_d, for $0 \leqslant |d| \leqslant n-1$ (see again Fig. 1 for an example).

Notice that each square submatrix of A is prefix of some suffix of some A_d, $0 \leqslant |d| \leqslant n-1$. An example is given in Fig. 7. Based on this observation, one can easily show that the Lsuffix tree for A represents all square submatrices of A [15]. Let $W_p = A[1{:}\ p, 1{:}\ p]$, for $1 \leqslant p \leqslant n$. For the diagonals of W_p, we follow the same numbering used for the diagonals of A. Let D_p be the set of those diagonal indices. For $d \in D_p$, let $W_{p,d}$ be the square submatrix of W_p of longest side length whose main diagonal is d. The Lsuffix tree $\mathrm{L}T_p$ for W_p is the Lsuffix tree for the Lstrings corresponding to $W_{p,d}$, for $d \in D_p$.

• *Relation with Previous Definitions of Lsuffix Tree.* Notice that there is no one-to-one correspondence between the leaves of $\mathrm{L}T_p$ and the suffixes of $W_{p,d}$, for $d \in D_p$. This is in contrast with [15]. Indeed, the definition given in [15] generalizes to matrices that of suffix tree for a string due to McCreight [28] while here we have generalized the definition of suffix tree given by Ukkonen [38]. We illustrate the difference between those two definitions in terms of strings.

Let x be a string over the alphabet Σ and let \$ be a symbol not in Σ. In the usual definition of suffix tree T_x for string x [28], the string x is given *off-line* and T_x is a trie storing all suffixes of x\$. Since \$ is not in Σ, all of the suffixes of x\$ are distinct. Because of this distinctness and by definition of trie, each of those suffixes must correspond to a unique path from the root of T_x to one of its leaves. That immediately implies a one-to-one correspondence between the leaves of T_x and the suffixes of x\$ which, in turn, implies a one-to-one correspondence between the leaves of T_x and the suffixes of x. In Ukkonen's case [38], the definition of suffix tree is the same except that it is given with respect to x rather than x\$. Indeed, the string x is obtained *on-line*. Therefore, it does not seem to be convenient to append an endmarker symbol at the end of a string that may be extended to the right. However, without the endmarker symbol, suffixes can be prefixes of other suffixes and the one-to-one correspondence mentioned earlier is lost.

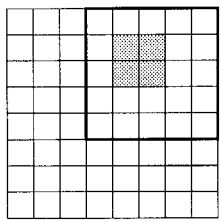

FIG. 7. The submatrix delimited by dark lines is A_{-3}. The submatrix highlighted by the dashed region is the second prefix of the second suffix of A_{-3}.

2.2. *Some Useful Facts*

We start by stating some assumptions and terminology. To simplify the boundary cases of our algorithms, we introduce a dummy matrix $W_{p,d}\$$, for each $d \in D_p$. Each $W_{p,d}\$$ has $W_{p,d}$ as a second suffix and it is different from all other $W_{p,d'}\$$'s, with $d' \neq d$. We remark that one can think of $W_{p,d}\$$ as a matrix in which the first row and column have the special symbol \$ and the remaining part is $W_{p,d}$. Although it would be intuitively correct to denote the dummy matrix by $\$W_{p,d}$, we prefer the notation $W_{p,d}\$$ to be uniform with the standard notation $x\$$ used for strings. We also introduce dummy leaves in $\mathrm{L}T_p$, each of which is associated to a distinct $W_{p,d}\$$. Each dummy leaf is connected to the root of $\mathrm{L}T_p$ by a dummy edge.

• *Strings and Nodes in Tries.* We need to recall some definitions for tries for strings given in [28] and extend them to tries for matrices. Given a compacted trie T over the alphabet Σ, a node u is the *locus* of a string α if and only if the concatenation of the labels on the path from the root of T to u is equal to α. Note that a string may not have a locus in T. The *extended locus* of α is the locus of the shortest string (if any) with locus in T and that has α as prefix. The *contracted locus* of α is the locus of the longest string having locus defined in T and that is prefix of α. It can be easily shown that the locus, contracted locus, and extended locus of a string are unique nodes in T. When a string has a locus defined in T, then its extended and contracted loci are the same node.

Analogous definitions hold for matrices. Given a compacted trie $\mathrm{L}T$ over the alphabet $\mathrm{L}\Sigma$, a node u is the *locus* of a matrix B if and only if the concatenation of the labels on the path from the root of $\mathrm{L}T$ to u is equal to B. Note that a matrix may not have a locus in $\mathrm{L}T$. The *extended locus* of B is the locus of the shortest matrix (if any) with locus in $\mathrm{L}T$ and that has B as prefix. The *contracted locus* of B is the locus of the longest matrix having locus defined in $\mathrm{L}T$ and that is prefix of B. It can be easily shown that the locus, contracted locus, and extended locus of a matrix are unique nodes in $\mathrm{L}T$. When a matrix has a locus defined in $\mathrm{L}T$, then its extended and contracted loci are the same node.

• *Distance of a Node from the Root of* $\mathrm{L}T_p$. Given a node u in $\mathrm{L}T_p$, let $l(u)$ be the side length of the matrix having locus in u. We refer to $l(u)$ as the *distance* of u from the root of $\mathrm{L}T_p$. For each node u of $\mathrm{L}T_p$, we store $l(u)$ at that node.

• *Equivalence Classes of Matrices.* Consider the matrix obtained by concatenating the labels on the path from the root of $\mathrm{L}T_p$ to a leaf f. That matrix can be suffix of more than one $W_{p,d}\$$, $d \in D_p$. Therefore, all suffixes of the $W_{p,d}\$$'s that "end" at f are an equivalence class, which we denote by

CLASS[f]. In what follows, $M(f)$ denotes an arbitrarily chosen matrix in *CLASS*[f].

• *Extension Matrices.* We now define extension matrices (an example is given in Fig. 8). Assume that $W_{p,d_1}, ..., W_{p,d_g}$ have their suffixes of side length q equal. Denote this suffix matrix by M. Assume also that no other matrix, among the $W_{p,d}$\$'s, has a suffix equal to M. Now, for each W_{p+1,d_e}, $1 \leqslant e \leqslant g$, take its suffix of side length $q+1$. Among those suffixes, keep only one copy of equal matrices. The result is the *set of extensions* of M. Each matrix in the set is referred to as *an extension matrix* of M. Notice that M has at least one extension. Let $\tilde{M}_1, ..., \tilde{M}_r$, be the extensions of M. We point out that they all differ on their last Lcharacter because they all have M as prefix.

• *Blossom Forest of* LT_p. We now define a useful tool for the fast transformation of LT_p into LT_{p+1}. Indeed, it will play a key role for the efficient implementation of all our algorithms. Intuitively, it is a forest of trees where each node is a leaf of LT_p. In this forest, the links go from child to parent and correspond to suffix links in LT_p. We now define it formally. Let f be a leaf of LT_p and assume that $M(f)$ is such that its second suffix has extended locus in a node u. We create a link, via a pointer, from f to u and refer to this pointer as the *suffix link* from f to u. Suffix links are defined for each leaf f of LT_p. For an example, see Fig. 1 again. Notice that the definition of suffix link does not follow the one in [28, 38] because here we are not granted that given a node u of LT_p locus of a matrix B, then there exists a node u' in LT_p that is locus of the second suffix of B (see

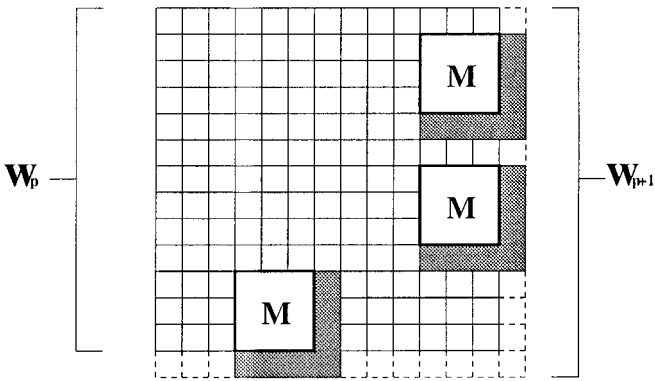

FIG. 8. Assume that matrix M (of side 3) appears only as suffix of $W_{p,7}$, $W_{p,-4}$, $W_{p,-9}$. Consider the three suffixes (of side 4) M_1, M_2, M_3 of $W_{p+1,7}$, $W_{p+1,-4}$, $W_{p+1,-9}$, respectively. Each of them is obtained by appending the dark Lcharacter to each of the occurrences of M. Assume that $\tilde{M}_1 = M_1 = M_2 \neq M_3 = \tilde{M}_2$. The set of extensions of M is given by $\{\tilde{M}_1, \tilde{M}_2\}$.

[15] for a discussion illustrating this fact). We divide suffix links in external and internal. A suffix link from f to u is *external* if and only if u is a leaf and it is the locus of the second suffix of $M(f)$. Otherwise, it is *internal*. Notice that the suffix link leaving a dummy leaf f (corresponding to some $W_{p,d}\$$) points to the leaf f' such that $W_{p,d}\$ \in CLASS[f']$. Therefore, all suffix links departing from dummy leaves are external. Consider all external suffix links in LT_p as edges and the leaves at their endpoints as nodes. It can be easily shown that the graph so obtained is a forest of trees. Each node of a given tree has the direction of the edge from parent to child reversed. We refer to this forest as the *blossom forest* of LT_p. Recalling that all $W_{p,d}\$$'s are distinct, one can easily show that there is a one-to-one correspondence between those matrices and the leaves in the forest.

• LT_p *Can Be Stored in* $O(|W_p|) = O(p^2)$ *Space.* Indeed, one can easily show that LT_p has $O(p^2)$ nodes. Moreover, the label, i.e., a chunk, on each of its edges can be represented by means of a quadruple of integers, as we now explain (the ideas are as in [15, 28]). Consider an edge (u, v) of LT_p and assume that it is labeled with the chunk corresponding to the rows and columns from the qth to the gth in the lth suffix of $W_{p,d'}\$$. We represent this chunk by the quadruple $(c, c', l(u), d')$, where c and c' are the projections of row q and row g of the lth suffix of $W_{p,d'}\$$ on the rows of W_p. Given the quadruple, we can easily access in constant time the subrows and subcolumns of W_p that correspond to the given chunk. For details, see [15].

• *Encoding of the Labels on the Edges of* LT_p *Entering Leaves.* For this kind of label, we use a special form of the encoding described earlier. As shown in Sections 3, 4, and 6, this encoding will allow us to save a substantial amount of work in transforming LT_p into LT_{p+1}. Let (u, v) be the same edge as above, labeled with the same chunk, but assume that v is a leaf. In that case, we are sure that the matrix M obtained by concatenating the Lcharacters on the path from the root of LT_p to v is equal to a suffix of $W_{p,d'}\$$. Since that suffix "touches" one of the boundaries of W_p, i.e., either row or column p, we can think of M as touching the same boundary too. But then, since the chunk labeling (u, v) is equal to an L-shaped part of M ending in its lower right corner, we can think of that chunk as touching the same boundary of W_p as M. We use the encoding $(c, \infty, l(u), d')$ rather than $(c, c', l(u), d')$. ∞ stands for the fact that the given chunk touches one of the boundaries of W_p. Notice that we can easily compute the row c' of W_p where that chunk ends. Indeed, when $d' \geqslant 0$, it is p; else it is $p + d'$. This encoding is analogous to the one devised by Ukkonen [38] for his *on-line* algorithm for the construction of the suffix tree of a string. We point out that, through the encoding of the label entering leaf

f, we can access in constant time an occurrence of $M(f)$ in W_p, i.e., the topmost and leftmost entry of a submatrix of W_p equal to $M(f)$.

• *Comparing Lcharacters Efficiently.* Given two Lcharacters of W_p of the same length as strings, we can establish whether they are equal or which one is lexicographically smaller than the other in $O(\log |W_p|)$ time, for $1 \leqslant p \leqslant n$. The proof of this statement is deferred until Section 8. We assume that for each Lcharacter that we need to compare, we know a quadruple identifying it as a chunk.

• *Dynamic Trees.* We maintains $\mathrm{L}T_p$ as a dynamic tree [34]. Dynamic trees are a data structure that supports efficient implementation of several operations involving nodes of a forest of trees. We now list the main operations we are interested in. *Link*(u, v) creates an edge from u to v, where v is the root of tree T in the forest and u is a node in a different tree T'. *Cut*(u, v) breaks the edge (u, v) creating a new tree with v as *root*. *Expose*(u) returns the path from u to the root of its tree in a balanced search tree. The leaves of the balanced search tree are the nodes on the mentioned path, sorted by their distance, i.e., number of tree edges, from the root. All of the mentioned operations can be implemented in logarithmic time in the size of the trees they operate on.

• *Auxiliary Data Structures.* Consider a node u of $\mathrm{L}T_p$. We keep its edges in a balanced search tree, lexicographically sorted according to the first Lcharacter of the chunk labeling each of them. Moreover, u also has a *descendant* pointer to an arbitrarily chosen leaf in its subtree. Those "pointers" are implemented as follows. Assume that leaf f has at least one node with a descendant pointer to f. We store all such nodes in a mergeable heap [1]. The root of such a heap contains f. Now, each node with descendant pointer to f points to itself in the heap of f.

3. FROM $\mathrm{L}T_p$ TO $\mathrm{L}T_{p+1}$—CHANGES IN THE STRUCTURE OF $\mathrm{L}T_p$

When W_p is extended to W_{p+1}, each matrix $W_{p,d}\$$ is extended by one Lcharacter to become a matrix $W_{p+1,d}\$$, for $d \in D_p$. Moreover, two new matrices are created: $W_{p+1,p}\$$ and $W_{p+1,-p}\$$. We want to transform $\mathrm{L}T_p$ into $\mathrm{L}T_{p+1}$. We first show how to insert $W_{p+1,p}\$$ in $\mathrm{L}T_p$ (the same ideas apply to the insertion of $W_{p+1,-p}\$$) and then concentrate on how the extension of $W_{p,d}\$$ into $W_{p+1,d}\$$ affects the structure of $\mathrm{L}T_p$, for $d \in D_p$.

• *Insertion of* $W_{p+1,p}\$$ *in* $\mathrm{L}T_p$. The second suffix of $W_{p+1,p}\$$ is $W_{p+1}[p+1, 1]$. We create a dummy leaf v corresponding to $W_{p+1,p}\$$ and we find the extended locus u of $W_{p+1}[p+1, 1]$ in $\mathrm{L}T_p$. If it exists, there is a matrix in $\mathrm{L}T_p$ that already has $W_{p+1}[p+1, 1]$ as prefix. All we need

to do is to set a suffix link from v to u. If there is no extended locus, then $W_{p+1}[p+1, 1]$ is not in LT_p. We create a new leaf u as child of the root of LT_p and we label it with $W_{p+1}[p+1, 1]$. Moreover, we create a suffix link from v to u and another one from u to the root of LT_p. Indeed, the root of LT_p is the locus of the empty matrix, which is the second suffix of $W_{p+1}[p+1, 1]$. This procedure takes constant time. In the reminder of this paper we will no longer mention the changes in LT_p associated to $W_{p+1, p}\$$ and $W_{p+1, -p}\$$.

• *Extension Matrices and the Structure of* LT_p. We now discuss how the extension of $W_{p, d}\$$ into $W_{p+1, d}\$$ affects the structure of LT_p, for $d \in D_p$. Notice that, since $W_{p+1, d}\$$ has $W_{p, d}\$$ as longest proper prefix, each suffix of $W_{p+1, d}\$$ can be obtained by appending an Lcharacter at the end of the "corresponding" suffix in $W_{p, d}\$$. But, by definition, LT_p "stores" all suffixes of $W_{p, d}\$$ and LT_p "stores" all suffixes of $W_{p+1, d}\$$, for all $d \in D_p$. Therefore, the transformation of LT_p into LT_{p+1} consists of extending the suffixes in LT_p by one Lcharacter. More specifically, let M be a suffix of some $W_{p, d}\$$ and let $\tilde{M}$ be one of its extensions. If $\tilde{M}$ has already an extended locus in LT_p, then this matrix will cause no change. Otherwise, we must insert it in LT_p and that may cause the following three types of changes: (i) leaves that become internal nodes by generating new leaves; (ii) internal nodes that generate new leaves; (iii) edges that are broken by the insertion of new internal nodes and leaves. The following facts state which changes need to be performed and under which conditions. In particular, the first one states under which conditions changes involving a matrix M having locus at a leaf of LT_p can be done implicitly, i.e., they can be ignored by using the special encoding of chunks on the edges of LT_p entering leaves.

FACT 3.1. *Let $u \in LT_p$ be a leaf. Assume that $M(u)$ has only one extension $\tilde{M}$. The concatenation of the labels on the path from the root of LT_p to u gives exactly $\tilde{M}$, when those labels are "decoded" with respect to the matrices $W_{p+1, d}\$$'s. Therefore, u is the locus of $\tilde{M}$ in LT_{p+1} and we need not modify u when LT_p is transformed into LT_{p+1}. That is, u can be updated implicitly.*

Proof. In order to represent $\tilde{M}$ in LT_{p+1}, we need to append one Lcharacter at the end of the path from the root of LT_p to u. Indeed, since $M(u)$ is the longest proper prefix of $\tilde{M}$, we can represent $\tilde{M}$ in LT_p by creating a new leaf u', u' becomes child of u, and the edge (u, u') is labeled with the last Lcharacter of $\tilde{M}$. But, among the suffixes of $W_{p, d}\$$'s, $M(u)$ is the only matrix that has $\tilde{M}$ as extension and that matrix is also its only extension. Therefore, we have that (in LT_{p+1}) the path $((f, u); (u, u'))$ is unary and must be compressed, where $f = parent(u)$ in LT_p. We can obtain

the same result by appending the required Lcharacter at the end of the path from the root of LT_p to u. We can perform such a change *implicitly* i.e., we do nothing, by using the special encoding that we have for the labels of edges entering leaves. We now explain this idea. Assume that the chunk on the edge (f, u) is encoded in terms of the occurrence of $M(u)$ as a suffix of $W_{p,d}$, for some $d \in D_p$. So, it is of the form $(c, \infty, l(u), d)$, where ∞ stands for p. Now, let us "decode" that quadruple with respect to W_{p+1}. In that case, ∞ stands for $p+1$. Since $M(u)$ has only one extension matrix and it must be a suffix of $W_{p+1,d}$, we have that the quadruple, when referred to W_{p+1}, corresponds to the chunk of M labeling the edge (f, u) in LT_p plus the new Lcharacter that we need to append to get $\tilde{M}$ in LT_{p+1}.

FACT 3.2. *Let $u \in LT_p$ be a leaf. Assume that $M(u)$ has $r \geqslant 2$ extensions $\tilde{M}_1, ..., \tilde{M}_r$. We create r new leaves (in LT_{p+1}, each leaf is the locus of one extension of $M(u)$). Those leaves become children of u in LT_{p+1} and u becomes an internal node.*

FACT 3.3. *Let $\tilde{M}$ be a matrix that has no extended locus in LT_p. Assume that $\tilde{M}$ is an extension of some suffix M of some $W_{p,d}\$$ and v is the extended locus of M in LT_p. v can be a leaf only when it is not the locus of M. We create a new leaf $\hat{f}$, which is the locus of $\tilde{M}$ in LT_{p+1}. If v is the locus of M in LT_p, $\hat{f}$ becomes child of v in LT_{p+1}. Else, we break the edge $(parent(v), v)$ in LT_p to create a new node v' that will be the locus of M in LT_{p+1}. In this case, the new leaf $\hat{f}$ becomes child of v' in LT_{p+1}.*

We use our new technique to identify and carry out the changes outlined above. It works in two phases: *Frontier Expansion* and *Internal Structure Expansion*. The first phase takes care of the changes described in Fact 3.2 avoiding consideration of the leaves that can be updated implicitly (the ones satisfying Fact 3.1). We outline this part in Section 4 and we give additional details in Section 6. The second phase takes care of the changes described in Fact 3.3. We outline this part in Section 5 and give additional details in Section 7.

• *Relation with On-line Suffix Tree Algorithm* [38]. Consider a string $x[1, p]$ and let ST_p be the suffix tree built by the on-line algorithm by Ukkonen. When $x[1, p]$ is extended to become $x[1, p+1]$, ST_p must also be modified to become ST_{p+1}. The only changes that can take place are analogous to (ii)–(iii) already described for LT_p. That is, no leaf of ST_p becomes an internal node. The reason why there is this difference between LT_p and ST_p is the following. Each leaf f of ST_p is the locus of a unique suffix of $x[1: p]$. When that suffix is extended by one character, the new suffix will still have locus in f and it will still be the unique such suffix. This

is not true for LT_p. Indeed, each leaf f' of LT_p can be the locus of a set of matrices, the ones in $CLASS[f']$. Although those matrices are equal, their extensions are not guaranteed to be equal. In this case, f' must become an internal node and new leaves must be created.

4. FRONTIER EXPANSION-OUTLINE

Here we present the main ideas supporting the procedure that identifies and performs the transformations involving leaves of LT_p. Indeed, at a high level, we describe the main tasks of *Frontier Expansion* and how they are carried out. Additional details on the implementation of those tasks are presented in Section 6, where we also provide a correctness proof and time analysis of the overall procedure.

As implied by Facts 3.1–3.2, the identification of which leaf u of LT_p generates new leaves in LT_{p+1} reduces to the computation of how many extension matrices $M(u)$ has. *Frontier Expansion* performs this task by restricting attention only to the leaves of LT_p that actually generate new leaves. The main tool used to achieve this goal is the blossom forest of LT_p. Moreover, this phase also performs the ancillary task of building part of the blossom forest of LT_{p+1} (the remaining part is built by the next phase). Indeed, when we need to transform LT_{p+1} into LT_{p+2}, we must know the blossom forest of LT_{p+1}.

The main ideas that allow *Frontier Expansion* to perform its two tasks are quite simple, when stripped off of the many details involved. However, proving that it correctly works in the amount of time that we need does not seem to be obvious. *Frontier Expansion* is implemented through a visit of each tree of the blossom forest of LT_p. In the remainder of this section (and unless otherwise specified), we concentrate on the nodes and leaves of the blossom forest since their correspondence with the leaves of LT_p is clear. We describe the visit of a single tree $\mathcal{T}$ in the blossom forest. It is the same for all others and it can be carried out independently (and therefore we can process the trees in the forest in any order). Recall that the edges in $\mathcal{T}$ are directed from children to parent. So, the indegree of each node in $\mathcal{T}$ is equal to the number of its children.

We need the following definitions. An *intersection node* in $\mathcal{T}$ is a node with at least two children. A *chain* of nodes in $\mathcal{T}$ is a path that either starts *at a leaf* or *at an intersection node* and it is the longest path such that all of its nodes, except the start nodes, have indegree equal to one. Notice that a chain either ends at the root of $\mathcal{T}$ or at a child of an intersection node. We say that a chain is *initial* if and only if the first node on the chain is a leaf of $\mathcal{T}$.

PROCEDURE `Visis`($\mathscr{T}$).

(V1) Mark all initial chains of $\mathscr{T}$ as *processed* and the remaining ones as *free*.

(V2) While there are free chains do: pick a free chain $\mathscr{C}$ such that all incoming chains have been processed; `Process`($\mathscr{C}$); mark $\mathscr{C}$ as processed.

Processing of a chain of nodes of $\mathscr{T}$, done by `Process`, is again conceptually simple. Starting with the first node on the chain, we compute extension matrices for the nodes until we either reach the end of the chain or we reach a node u such that $M(u)$ has only one extension matrix. In that case, we skip at the end of the chain and we are done. In particular, for a node u that is not skipped, we need to perform the following: (a) computation of the extension matrices of $M(u)$; (b) the possible transformation of u into an internal node of $\mathrm{L}T_p$ by the creation of new leaves; (c) the set up of suffix links going into the new leaves (this last step is required since we need the blossom forest of $\mathrm{L}T_{p+1}$); (d) modification of some auxiliary data structures.

5. INTERNAL STRUCTURE EXPANSION–OUTLINE

Here we consider the cases in which internal nodes of $\mathrm{L}T_p$ generate new leaves, or edges of $\mathrm{L}T_p$ are broken by the insertion of new internal nodes and leaves. Again, those changes are ruled by extension matrices. At a high level, we present the organization of this phase, leaving out implementation details that are presented in Section 7. We also prove a few facts that are used in Section 7 to establish correctness and perform the time analysis of this phase. We start with the following preliminary observation.

FACT 5.1. *Let $\tilde{M}$ be a matrix that has an extended locus v in $\mathrm{L}T_p$. Then, all the suffixes of $\tilde{M}$ have extended locus in $\mathrm{L}T_p$.*

Fact 5.1 can be used as follows. Once we have identified an extension matrix that has already extended the locus in $\mathrm{L}T_p$, we can ignore that matrix and all of its suffixes (which are also extension matrices) since all of those matrices are already in $\mathrm{L}T_p$ (and therefore in $\mathrm{L}T_{p+1}$).

The matrices considered by *Internal Structure Expansion* are suffixes of $W_{p,d}$\$'s that do not have a locus at a leaf of $\mathrm{L}T_p$. Starting from an initial set identified at the end of *Frontier Expansion*, *Internal Structure Expansion* consists of dynamically maintaining a set of extension matrices for which we are sure Fact 3.3 holds. A matrix is discarded, together with its suffixes, as soon as Fact 5.1 holds. Indeed, we keep in a priority queue a set of

extension matrices that are either part of the initial configuration of the queue (which will be specified shortly) or have already generated new leaves in LT_p. The matrix with highest priority is the one with longest side length and ties are broken arbitrarily.

We now outline the configuration of the queue and the processing of the matrix $\tilde{M}$ of highest priority in the queue at an arbitrary time instant. Informally, the main task is to check whether the second suffix of $\tilde{M}$ has an extended locus in LT_p. If it does not, we have found another extension matrix that generates a new leaf in LT_p.

- *Configuration of the Queue.* The set of matrices that are initially in the queue Q is obtained as follows. Consider a leaf u in LT_p that is also root of a tree in the blossom forest of LT_p. Independently of how many extensions $M(u)$ has, all those matrices are initially inserted in Q. Except for this initial set of matrices, all other matrices in the queue at any time instant have generated a new leaf in LT_p, which will be a leaf of LT_{p+1}. For each matrix in the queue, we keep the leaf f, locus of that matrix in LT_{p+1}.

PROCEDURE `Process-Matrix`($\tilde{M}$).

(PM1) Check whether $\tilde{M}'$ is in Q, where $\tilde{M}'$ is the second suffix of $\tilde{M}$.

(PM2) Assume that there is a matrix $\tilde{Z}' = \tilde{M}'$ already in the queue and let f' be the new leaf of LT_{p+1} associated with $\tilde{Z}'$. Set a suffix link (valid in LT_{p+1}) from f to f', where f is the leaf of LT_{p+1} associated with $\tilde{M}$. Discard $\tilde{M}'$. Exit.

(PM3) Assume that $\tilde{M}'$ is not in the queue. Check whether it has an extended locus z in LT_p and, if it does, set up a suffix link from f to z and discard $\tilde{M}'$. Exit.

(PM4) Assume that $\tilde{M}'$ is not in the queue and that it has no extended locus in LT_p. Let $\tilde{M}'$ be the longest proper prefix of $\tilde{M}'$. Let v' be the extended locus of M' in LT_p. We **assume** that, when we enter this step, we know **whether or not** v' is the locus of M'. We apply the transformations in Fact 3.3 as follows. Assume that v' is the locus of M'. We create a new leaf f', which will be a new child of v' in LT_{p+1}. Assume that v' is the extended locus of M' in LT_p. In LT_{p+1}, f' is a child of a new node g and g "splits" the edge (w, v'), where w is the parent of v' in LT_p. In both cases we set a new suffix link (valid in LT_{p+1}) from f (the new leaf generated by $\tilde{M}$) pointing to f'.

LEMMA 5.1. *Assume that steps* (PM1) *and* (PM3) *in Procedure* `Process-Matrix` *are implemented correctly. Moreover, assume also that the stated information is available when we enter step* (PM4). *Then, Internal Structure Expansion correctly identifies the extension matrices satisfying Fact* 3.3 *and correctly carries out the appropriate changes in the structure of* LT_p.

Moreover, for each new leaf that is created, the appropriate suffix link departing from it is created.

Proof. Let $\tilde{M}_i$ be one of the extension matrices that is initially placed in the queue. Let $M_i(u)$ be the matrix of which $\tilde{M}_i$ is extension, where u is a leaf in $\mathrm{L}T_p$ and also a root of a tree $\mathscr{T}$ in the blossom forest of $\mathrm{L}T_p$. Now, all extension matrices that have $\tilde{M}_i$ as suffix are those associated to the matrices that have loci in the nodes of $\mathscr{T}$ and none of them plays any role in the modification of the internal structure of $\mathrm{L}T_p$. However, the suffixes of $\tilde{M}_i$ may affect the internal structure of $\mathrm{L}T_p$. Therefore, we can conclude that the suffixes of matrices initially placed in the queue are a maximal set of matrices that can cause changes in the internal structure of $\mathrm{L}T_p$. In order to prove the lemma, all we need to do is to arbitrarily pick a matrix from the initial configuration of the queue and show that all suffixes of the chosen matrix inserted in the queue (and therefore processed by Procedure `Process-Matrix`) cause changes to the internal structure of $\mathrm{L}T_p$, while the remaining ones can be ignored. In addition, we also show that suffix links are properly set. We pick $\tilde{M}_i$.

Consider the suffixes of $\tilde{M}_i$ that are processed by Procedure `Process-Matrix` and let $\tilde{M}_{i,e}$ be the suffix of shortest side length that is processed by the procedure. Let S be the set of all suffixes of $\tilde{M}_i$ of side length longer than that of $\tilde{M}_{i,e}$. It is a simple exercise to show that when Procedure `Process-Matrix` is called on a matrix $\tilde{M}'$ in S, step (`PM4`) is executed for the second suffix $\tilde{M}'$ of $\tilde{M}$ and that the appropriate changes to the structure of $\mathrm{L}T_p$ are performed. Moreover, the appropriate suffix link from f to f' is set up, where f and f' are the leaves of $\mathrm{L}T_{p+1}$ generated by $\tilde{M}$ and $\tilde{M}'$, respectively.

As for $\tilde{M}_{i,e}$, we have to consider two separate cases. When its second suffix $\tilde{M}'_{i,e}$ is the empty matrix, then its extended locus is the root of $\mathrm{L}T_p$. Therefore, we discard $\tilde{M}'_{i,e}$ in step (`PM3`), completing the processing of all suffixes of $\tilde{M}_i$. Moreover, in step (`PM3`), we correctly set up a suffix link from the leaf $\tilde{f}$ generated by $\tilde{M}_{i,e}$ to the root of $\mathrm{L}T_p$.

Assume now that $\tilde{M}'_{i,e}$ is not the empty matrix. Since $\tilde{M}'_{i,e}$ is the last suffix of $\tilde{M}_i$ that is processed, we must have that $\tilde{M}'_{i,e}$ is discarded either in step (`PM2`) or in step (`PM3`). In the first subcase, insertion of $\tilde{M}'_{i,e}$ in the queue is redundant. In the second subcase, we discard $\tilde{M}'_{i,e}$ because of Fact 5.1. However, in both subcases, we have completed the processing of all suffixes of $\tilde{M}_i$ relevant for the changes in the structure of $\mathrm{L}T_p$. Moreover, we have correctly set up a suffix link from $\tilde{f}$ to the appropriate node. Details are left to the reader.

LEMMA 5.2. *During the execution of Internal Structure Expansion, the maximum number of matrices in Q is $O(|\mathrm{L}T_{p+1}| - |\mathrm{L}T_p| + p)$ which, in turn, is $O(|W_p|)$.*

Proof. The number of matrices in the initial configuration of the queue is $O(|\mathrm{L}T_{p+1}| - |\mathrm{L}T_p| + p)$. Indeed, we have $O(p)$ leaves in $\mathrm{L}T_p$ and the total number of new leaves they can generate is $O(|\mathrm{L}T_{p+1}| - |\mathrm{L}T_p|)$. Now, a matrix in the queue is removed and it can be charged for the insertion of a new matrix (its second suffix) only when that matrix generates a new leaf in $\mathrm{L}T_{p+1}$. Therefore, the total number of matrices in the queue is still $O(|\mathrm{L}T_{p+1}| - |\mathrm{L}T_p| + p)$. But $|\mathrm{L}T_{p+1}| = O(|W_p|)$ and the lemma follows.

LEMMA 5.3. *Step* (PM2) *of Procedure* `Process-Matrix` *is the only one that creates intersection nodes in the blossom forest of* $\mathrm{L}T_{p+1}$ *during Internal Structure Expansion.*

Proof. Indeed, step (PM3) creates an internal suffix link, while the suffix link created in step (PM4) goes into a leaf that has been just created and, at least at the time of its creation, it is not known yet whether it will be an intersection node in the blossom forest of $\mathrm{L}T_{p+1}$. Now consider a matrix $\tilde{M}$ given in input to Procedure `Process-Matrix` and assume that its second suffix $\tilde{M}'$ is already in the queue. In this case, there has been another extension matrix $\tilde{Z} \neq \tilde{M}'$ of the same side length as $\tilde{M}$ that also has its second suffix $\tilde{Z}' = \tilde{M}'$ and that was processed before $\tilde{M}$. Let $\tilde{Z}$ be the first such a matrix given in input to `Process-Matrix`. During the execution of `Process-Matrix`$(\tilde{Z})$, $\tilde{Z}'$ has been inserted in the queue and a leaf f', locus of $\tilde{Z}'$ in $\mathrm{L}T_{p+1}$, has been created. Moreover, in step (PM4), a suffix link from $\tilde{f}$ to f' has been set up, where $\tilde{f}$ is the locus of $\tilde{Z}$ in $\mathrm{L}T_{p+1}$. Therefore, when step (PM2) is executed for $\tilde{M}$, there is already a suffix link (valid in $\mathrm{L}T_{p+1}$) pointing to f'. ∎

6. FRONTIER EXPANSION—DETAILS

Here we give the details for the implementation of the visit of a tree $\mathscr{T}$ of the blossom forest of $\mathrm{L}T_p$. Recall from Subsection 2.2 that $CLASS[u]$ is the equivalence class of matrices having locus in u. The presentation is organized as follows. Let u be a leaf of $\mathrm{L}T_p$. In Subsection 6.1 we show how to compute the extension matrices of $M(u) \in CLASS[u]$. This gives us information to establish how many (if any) new leaves u generates in $\mathrm{L}T_{p+1}$. In Subsection 6.2 we show how to connect, through suffix links, the leaves of $\mathrm{L}T_{p+1}$ "created" during the processing of u. Then, in Subsection 6.3, we show how to process the chains of $\mathscr{T}$ and in the remaining subsection we prove correctness and perform the time analysis of *Frontier Expansion.*

For each leaf $u \in \mathrm{L}T_p$, let $\mathscr{R}(u)$ be the set of leaves for which u is *responsible* in $\mathrm{L}T_{p+1}$. That is, if u generates no new leaf in $\mathrm{L}T_{p+1}$ (because

Fact 3.1 applies to u), then $\mathscr{R}(u) = \{u\}$. Else, when Fact 3.2 applies to u, it is the set of r new leaves. Moreover, given $M(u) \in CLASS[u]$, let $\mathscr{E}(u)$ be the set of extensions of $M(u)$. We say that *we know* the matrices in $\mathscr{E}(u)$ when, for each matrix in the set, we know the start point of one of its occurrences in W_{p+1}.

6.1. *Computing Extension Matrices of $M(u)$ and the Set $\mathscr{R}(u)$*

Again, let u be a non-dummy leaf of LT_p and let $u'_1, u'_2, ..., u'_c$ be all the leaves of LT_p having a suffix link pointing to u. Let $\mathscr{PE}(u) = \bigcup_{i=1}^{c} \mathscr{E}(u'_i)$, where $\mathscr{E}(u'_i)$ is the set of extension matrices of $M(u'_i)$, for $1 \leqslant i \leqslant c$. Assuming that we know the matrices in $\mathscr{PE}(u)$, we show how to compute $\mathscr{E}(u)$.

Notice that $\mathscr{PE}(u)$ is a set of matrices because each $M(u'_i)$ is distinct, all of its extensions are distinct, and those extensions all have $M(u'_i)$ as prefix. It can be easily shown that two distinct matrices in $\mathscr{PE}(u)$ may have equal second suffixes. Let $\mathscr{G}$ be the multiset of the second suffixes of matrices in $\mathscr{PE}(u)$, i.e., if two matrices in $\mathscr{PE}(u)$ have equal second suffixes we take both of those suffixes. The following lemma characterizes the extension matrices of $M(u)$ in terms of the multiset $\mathscr{G}$.

LEMMA 6.1. *Assume that we know $\mathscr{PE}(u)$ and let $\mathscr{G}$ be the multiset defined above. Divide $\mathscr{G}$ into equivalence classes of equal matrices. The extensions of $M(u)$ are obtained by picking a representative matrix for each equivalence class. When $|\mathscr{PE}(u)| = |\mathscr{G}| \geqslant 2$, the needed partition can be computed by lexicographically sorting the last Lcharacters of matrices in $\mathscr{G}$ in $O(|\mathscr{PE}(u)| \log^2 |W_p|)$ time.*

Proof. We first show that each extension of $M(u)$ is equal to at least one matrix in $\mathscr{G}$. Observe that since u is not a dummy leaf (it has a suffix link pointing to it), $M(u)$ cannot be equal to any $W_{p,d}\$$, $d \in D_p$. Letting q be the side length of $M(u)$, notice that the side length of $M(u'_i)$ is $q+1$, for $1 \leqslant i \leqslant c$. Now, pick an extension $\tilde{M}$ of $M(u)$. Without loss of generality, assume that $\tilde{M}$ is suffix of $W_{p+1,d'}\$$, for some $d' \in D_p$. Among the $M(u'_i)$'s, pick the M' that is equal to the suffix of $W_{p,d'}\$$ of side length $q+1$ (M' must exist because $M(u)$ is a proper suffix of $W_{p,d'}\$$ and the set of matrices $M(u'_i)$, $1 \leqslant i \leqslant c$, are all and only the matrices that are both equal to suffixes of $W_{p,d}\$$'s and that have $M(u)$ as second suffix). Consider the extension of M' obtained through $W_{p+1,d'}\$$. That extension has $\tilde{M}$ as second suffix and that suffix is in $\mathscr{G}$. Therefore, given an extension of $M(u)$, there is at least one matrix in $\mathscr{G}$ that is equal to it. Using arguments similar to the ones just outlined, we can show that each matrix in $\mathscr{G}$ is equal to an extension of M. We omit the details. The first part of the lemma follows.

As for the second part, observe that each matrix in $\mathscr{G}$ has $M(u)$ as its longest proper prefix. So, two of those matrices may differ only on their

last Lcharacter. This fact implies that we can partition $\mathscr{G}$ into equivalence classes by lexicographically sorting the last Lcharacters of matrices in $\mathscr{G}$. Since we know, for each matrix in $\mathscr{PE}(u)$, one of its occurrences in W_{p+1}, we also know, for each matrix in $\mathscr{G}$, where it occurs in W_{p+1}. Therefore, we can access in constant time the last Lcharacter of each of those matrices in terms of subrows and subcolumns of W_{p+1}. As pointed out in Subsection 2.2, the comparison of two Lcharacters, taken from subrows and subcolumns of W_{p+1}, can be done in $O(\log |W_{p+1}|)$ time. Therefore, the lexicographic sort of $|\mathscr{G}|$ Lcharacter costs $O(|\mathscr{G}| \log |\mathscr{G}| \log |W_{p+1}|)$ time. But, $|\mathscr{G}| = |\mathscr{PE}(u)|$, $|W_{p+1}| = O(|W_p|)$ and $|\mathscr{PE}(u)| \leqslant |W_{p+1}|$. Therefore, we can lexicographically sort the last Lcharacters of matrices in $\mathscr{G}$ in $O(|\mathscr{PE}(u)| \log^2 |W_p|)$ time. ▮

For future reference, it is convenient to single-out the following consequences of Lemma 6.1:

FACT 6.1. *Under the same assumptions as in Lemma* 6.1, *once we know the equivalence classes in* $\mathscr{G}$, *we also know the matrices in* $\mathscr{E}(u)$ *and its cardinality. Moreover, we can compute the set* $\mathscr{R}(u)$ *in* $O(|\mathscr{R}(u)|)$ *time and establish a one-to-one correspondence between the matrices in* $\mathscr{E}(u)$ *and the nodes in* $\mathscr{R}(u)$.

Proof. For each matrix in $\mathscr{E}(u)$, we know one of its occurrences in W_{p+1}. Indeed, by assumption, we know the matrices in $\mathscr{PE}(u)$ and the matrices in $\mathscr{E}(u)$ are a subset of the second suffixes of matrices in $\mathscr{PE}(u)$. As for the second part, if $|\mathscr{E}(u)| = 1$, i.e., $M(u)$ has only one extension matrix, then $\mathscr{R}(u) = \{u\}$. When $|\mathscr{E}(u)| > 1$, we generate a distinct node for each matrix in $\mathscr{E}(u)$ and the set of those new nodes is $\mathscr{R}(u)$.

FACT 6.2. *Assume that* u'_1 *is the only leaf of* LT_p *with a suffix link pointing to u. Moreover, assume that* $M(u'_1)$ *has only one extension matrix. Then,* $M(u)$ *has only one extension matrix. The extension matrix of* $M(u)$ *is the second suffix of the extension matrix of* $M(u'_1)$.

FACT 6.3. *Let* $\mathscr{PR}(u) = \bigcup_{i=1}^{c} \mathscr{R}(u'_1)$, *i.e., the union of the sets of nodes for which each* u'_i, $1 \leqslant i \leqslant c$, *is responsible in* LT_{p+1}. *The partition of* $\mathscr{G}$ *in equivalence classes induces a partition of* $\mathscr{PR}(u)$ *in equivalence classes. Indeed, two leaves* v_1 *and* v_2 *in* $\mathscr{PR}(u)$ *are equivalent if and only if the extension matrices (from* $\mathscr{PE}(u)$*) of which they are loci in* LT_{p+1} *have their second suffix in the same equivalence class of* $\mathscr{G}$. *Once that the partition of* $\mathscr{G}$ *is known, the partition of* $\mathscr{PR}(u)$ *can be computed in* $O(|\mathscr{PR}(u)|)$ *time.*

6.2. *Setting Up Suffix Links*

We now outline how to connect, through suffix links, the leaves for which u is responsible in LT_{p+1} to the ones for which each u'_i is responsible,

for $1 \leqslant i \leqslant c$. That is needed in order to generate part of the blossom forest of $\mathrm{L}T_{p+1}$. It is convenient to consider two separate cases, one of which allows us to keep part of a chain (of suffix links) in the blossom forest of $\mathrm{L}T_p$ as a chain (of suffix links) in the blossom forest of $\mathrm{L}T_{p+1}$. We need a preliminary observation. Assume that $|\mathscr{P}\mathscr{R}(u)| = 1$. Then, there is only one leaf u'_1 with a suffix link pointing to u in $\mathrm{L}T_p$. Therefore, u cannot be the start of a chain in the blossom forest of $\mathrm{L}T_p$ and u'_1 and u are on the same chain $\mathscr{C}$.

(S1) Assume that $|\mathscr{P}\mathscr{R}(u)| = 1$ and assume that u'_1 is the leftmost node in $\mathscr{C}$ such that $|\mathscr{E}(u'_1)| = 1$, i.e., $M(u'_1)$ has only one extension matrix. We can keep the suffix links from u'_1 to the end of $\mathscr{C}$ as part of the blossom forest of $\mathrm{L}T_{p+1}$.

(S2) Assume that $|\mathscr{P}\mathscr{R}(u)| > 1$. Consider the partition of $\mathscr{P}\mathscr{R}(u)$ in equivalence classes, as outlined in Fact 6.3. Each node in the same equivalence class must be connected, via a suffix link, valid in $\mathrm{L}T_{p+1}$, to the corresponding node in $\mathscr{R}(u)$.

LEMMA 6.2. *Assume that the conditions in* (S1) *are satisfied. The leaves of* $\mathrm{L}T_p$ *from* u'_1 *to the end of* $\mathscr{C}$ *can be updated implicitly, i.e. they can be skipped. Moreover, we can correctly keep the suffix links from* u'_1 *to the end of* $\mathscr{C}$ *as part of the blossom forest of* $\mathrm{L}T_{p+1}$.

Proof. By assumption, u'_1 is the leftmost node on the chain $\mathscr{C}$ such that $|\mathscr{E}(u'_1)| = 1$. Starting with the edge (u'_1, u) and ending with the last edge on the chain $\mathscr{C}$, one can repeatedly apply Fact 6.2 to each edge $(\tilde{u}_1, \tilde{u}_2)$ to show that $M(\tilde{u}_2)$ has only one extension matrix $\tilde{M}$ and that the only extension matrix of $M(\tilde{u}_1)$ has $\tilde{M}$ as second suffix. This latter observation together with Fact 3.1 immediately implies the lemma. ∎

LEMMA 6.3. *Step* (S2) *correctly connects, via suffix links valid in* $\mathrm{L}T_{p+1}$, *the nodes in* $\mathscr{P}\mathscr{R}(u)$ *with the nodes in* $\mathscr{R}(u)$. *This step can be implemented to take* $O(|\mathscr{P}\mathscr{R}(u)| + |\mathscr{R}(u)|)$ *time.*

Proof. The proof is a direct consequence of Facts 6.1 and 6.3. Details are omitted. ∎

6.3. *Processing of the Chains of* $\mathscr{T}$

Here we give a detailed description of Procedure `Process`. It assumes that the following invariant holds for each chain $\mathscr{C}'$ of $\mathscr{T}$ that has been marked as processed by Procedure `Visit`. Recall from Section 4 that a chain is marked as processed either when it is an initial chain of $\mathscr{T}$ or when it has already been processed by Procedure `Process`. We show that the same invariant holds once a chain $\mathscr{C}$ has been processed.

Invariant 6.1. Let $\mathscr{C}'$ be a chain that has been processed. $\mathscr{C}'$ is stored at the leaves of a balanced search tree [1], where the "left to right" order of nodes in the chain is preserved at the leaves of the balanced search tree. Let u' be the last node in $\mathscr{C}'$. We know the matrices in $\mathscr{E}(u')$, i.e., the extensions of $M(u')$, and the set $\mathscr{R}(u')$ of the leaves of $\mathrm{L}T_{p+1}$ for which u' is responsible. Moreover, for each $v \in \mathscr{R}(u')$, we have computed the subtree of the blossom forest of $\mathrm{L}T_{p+1}$ rooted at v and that subtree is partitioned into chains, which are represented in balanced search trees.

PROCEDURE `Process`($\mathscr{C}$).

(PN0) $u \leftarrow u_0$, where u_0 is the first node on the chain $\mathscr{C}$.

(PN1) Following the same notation as in Lemma 6.1, lexicographically sort the last Lcharacter of matrices in $\mathscr{G}$ to compute $\mathscr{E}(u)$, the set of extension matrices of $M(u)$. Compute the set $\mathscr{R}(u)$.

(PN2) Use case (S2) to connect, via suffix links, the nodes in $\mathscr{P}\mathscr{R}(u)$ with the corresponding ones in $\mathscr{R}(u)$. We now have to update the chains in the (growing) blossom forest of $\mathrm{L}T_{p+1}$ and establish whether we are finished processing this chain. We have two cases.

(PN3) Assume that $|\mathscr{R}(u)| \geqslant 2$. Let f be a node in $\mathscr{R}(u)$. If there is exactly one node v in $\mathscr{P}\mathscr{R}(u)$ connected to f through suffix links (valid in $\mathrm{L}T_{p+1}$), we add f at the end to the chain of which v is part by inserting it as the rightmost node in the corresponding balanced search tree. If more than one node in $\mathscr{P}\mathscr{R}(u)$ is connected through suffix links to f, then f is an intersection node in the blossom forest of $\mathrm{L}T_{p+1}$ and we start a new chain and the corresponding balanced search tree consists of f only. The same reasoning applies to all other nodes in $\mathscr{R}(u)$. Set u to be the next node on the chain $\mathscr{C}$, if any, and go to step (PN1).

(PN4) Assume that $|\mathscr{R}(u)| = 1$. Skip at the end of the chain $\mathscr{C}$. Keep the chain from u to the last node in $\mathscr{C}$ as part of the blossom forest of $\mathrm{L}T_{p+1}$. Since u is the start of a new chain in $\mathrm{L}T_{p+1}$, this latter step involves splitting the balanced search representing $\mathscr{C}$ to obtain a new balanced search tree representing the chain from u to the last node in $\mathscr{C}$. Exit the procedure.

LEMMA 6.4. *Fix a chain $\mathscr{C}$ of $\mathscr{T}$ that is free. Assuming that Invariant* 6.1 *holds for all chains $\mathscr{C}'$ that have been processed before $\mathscr{C}$, it holds once $\mathscr{C}$ has been processed.*

Proof. Consider u_0, the first node on the chain. We first show that, once u_0 has been processed, we have that: (a) we know the matrices in $\mathscr{E}(u_0)$, i.e., the extensions of $M(u_0)$, and the set $\mathscr{R}(u_0)$ of the leaves of

LT_{p+1} for which u_0 is responsible and (b) for each $v \in \mathscr{R}(u_0)$, we have computed the subtrees of the blossom forest of LT_{p+1} rooted at v and that subtree is partitioned into chains, which are represented as balanced search trees.

• *Proof of* (a). Since Invariant 6.1 holds when u_0 is processed, we know $\mathscr{PR}(u_0)$. Moreover, since u_0 is an intersection node in $\mathscr{T}$, we must have $|\mathscr{PR}(u_0)| > 1$. Therefore, by Lemma 6.1, step (PN1) correctly computes the set $\mathscr{E}(u_0)$ of extension matrices of $M(u_0)$. Moreover, we know those matrices, i.e. for each of them, we know the start point of one of its occurrences in W_{p+1}. Once that we know $\mathscr{E}(u_0)$, we can correctly compute $\mathscr{R}(u_0)$ by Fact 6.1.

• *Proof of* (b). Since $|\mathscr{PR}(u_0)| > 1$ also $|\mathscr{PR}(u_0)| > 1$ and therefore, by Lemma 6.3, step (PN2) correctly sets up the suffix links between the nodes in $\mathscr{PR}(u_0)$ and the ones in $\mathscr{R}(u_0)$. Let $u'_1, ..., u'_s$ be the children of u_0 in $\mathscr{T}$. Those nodes are all at the the end of their chains and therefore, by Invariant 6.1, we have computed the subtree of the blossom forest of LT_{p+1} rooted at each of those nodes and that subtree is correctly partitioned into chains. Now, if $|\mathscr{R}(u_0)| > 1$, step (PN3) correctly extends those chains, keeping the appropriate partition into chains and the corresponding balanced search trees. Therefore (b) holds. If $|\mathscr{R}(u_0)| = 1$, notice that u_0 is an intersection node in the blossom forest of LT_{p+1} because $|\mathscr{PR}(u_0)| > 1$. Moreover, by Lemma 6.2, we can keep the chain of nodes from u_0 to the end of $\mathscr{C}$ as part of the blossom forest of LT_{p+1}. Step (PN4) correctly performs that task, keeping the balanced search tree associated with $\mathscr{C}$. In addition, Invariant 6.1 states that we have computed the subtrees of the blossom forest of LT_{p+1} rooted at each of the nodes in $\mathscr{PR}(u_0)$ and that those subtrees are correctly partitioned into chains. Therefore (b) holds.

In order to complete the proof of the lemma, repeatedly prove statements analogous to (a) and (b) for each node of $\mathscr{C}$, proceeding from left to right, until one either reaches the end of $\mathscr{C}$ or a node such that $|\mathscr{R}(u)| = 1$. The proofs are analogous to the ones given for u_0, except that knowledge of the sets $\mathscr{PE}(u)$ and $\mathscr{PR}(u)$ is now granted by the fact that we have computed them rather than by Invariant 6.1. ▮

There are some additional, and somewhat standard, implementation details that we outline for convenience of the reader. Some of them are related to the efficient implementation of some operations in Procedure Process while others are related to the maintenance of the auxiliary data structures introduced in Section 2.2.

(I1) Once we know $\mathscr{R}(u)$, it is an easy matter to transform u into an internal node of LT_{p+1}, while preserving the dynamic tree representation

of LT_p and LT_{p+1}. Using standard dynamic tree operation, the time is $O(|\mathscr{R}(u)| \log |W_p|)$. We omit the details on how to actually compute the labels on the new edges that are created. In this case, the time is $O(|\mathscr{R}(u)|)$.

(I2) Assume that u generates new leaves in LT_{p+1}. We must move the "descendant pointers" pointing to u to one of its leaves in LT_{p+1}. We proceed as follows. We arbitrarily choose a leaf $\hat{u} \in LT_{p+1}$, descendant of u in that tree. The root of the heap of u is labeled $\hat{u}$ and u is inserted in the heap. That has the effect of moving the heap of u to $\hat{u}$ and of setting a descendant pointer to a leaf in its subtree. Time is $O(\log |W_p|)$.

(I3) Assume that u generates c' new leaves in LT_{p+1}. We need to store the children of u in a balanced search tree, sorted according to the first Lcharacter of the label on their incoming edges. Let $\alpha_1, ..., \alpha_{c'}$ be the strings corresponding to those Lcharacters. Standard algorithms for insertion into balanced search trees [1] can be used to build the needed data structure, except that, as stated in Subsection 2.2, the lexicographic comparison of any two strings here costs $O(\log |W_p|)$ time. Therefore, setting up this balanced search tree for u costs $O(c' \log^2 |W_p)$ time.

LEMMA 6.5. *Procedure* `Process` *can be implemented to take* $O((|\mathscr{P}\mathscr{R}(u_0)| + |NEW(\mathscr{C})|) \log^2 |W_p|)$ *time, where* u_0 *is the first node on the chain* $\mathscr{C}$ *and* $NEW(\mathscr{C})$ *is the set of new leaves in* LT_{p+1} *that is generated by the nodes of* $\mathscr{C}$.

Proof. Consider u_0. By Lemma 6.1, the lexicographic sort in step (PN1) takes $O(|\mathscr{P}\mathscr{R}(|u_0)| \log^2 |W_p|)$ time. Once that we know the lexicographic sort, computation of the set $\mathscr{R}(u_0)$ can be done in linear time by Fact 6.1. Moreover, by Lemma 6.3, step (PN2) can be implemented with an additional cost of $O(|\mathscr{P}\mathscr{R}(u_0)| + |\mathscr{R}(u_0)|)$ time.

Assume now that $|\mathscr{R}(u_0)| > 1$. All operations involving those nodes in keeping the chains of the blossom forest of LT_{p+1} represented as balanced search trees take a total of $O(|\mathscr{R}(u_0)| \log |W_p|)$ time. Moreover, the additional bookeeping outlined in (I1)–(I2) takes the same amount of time while that in (I3) takes $O(|\mathscr{R}(u_0)| \log^2 |W_p|)$ time (u_0 generates $|\mathscr{R}(u_0)|$ new leaves, which are its children). Therefore, when $|\mathscr{R}(u_0)| > 1$, the total time to process u_0 is $O((|\mathscr{P}\mathscr{R}(u_0)| + |\mathscr{R}(u_0)|) \log^2 |W_p|)$.

Assume that $|\mathscr{R}(u_0)| = 1$ and consider step (PN4). Since the chain $\mathscr{C}$ is represented as a balanced search tree, skipping at the end of $\mathscr{C}$ takes $O(\log |W_p|)$ time, which is also the time to complete this step.

Analogous bounds hold for all other nodes that are processed by the procedure. Since the $\mathscr{R}$ set of a node of $\mathscr{C}$ is the $\mathscr{P}\mathscr{R}$ set of the next node on the chain and nodes are processed as long as they generate new leaves in LT_{p+1}, we have that the claimed bound follows. ∎

6.4. *Frontier Expansion—Correctness and Timer Analysis*

Here we show the following:

LEMMA 6.6. *Frontier Expansion correctly identifies all leaves of* $\mathrm{L}T_p$ *that generate new leaves in* $\mathrm{L}T_{p+1}$. *Moreover, for each leaf* $\bar{u} \in \mathscr{R}(u)$, *u a leaf of* $\mathrm{L}T_p$, *it correctly computes the subtree of the blossom forest of* $\mathrm{L}T_{p+1}$ *rooted at* $\bar{u}$. *That subtree is correctly partitioned into chains. The entire procedure takes* $O((|\mathrm{L}T_{p+1}| - |\mathrm{L}T_p| + p) \log^2 |W_p|)$ *time.*

Proof. We address correctness first. Recall from Section 4 that Frontier Expansion consists of the visit of each tree of the blossom forest of $\mathrm{L}T_p$, which can be carried out independently. Therefore, we can establish correctness for the visit of a single tree $\mathscr{T}$. Recall also from Section 4 that a visit of a tree $\mathscr{T}$ consists of skipping the initial chains of $\mathscr{T}$. The remaining ones are then processed according to Procedure `Process`. Now, assume that all initial chains of $\mathscr{T}$ are stored in a balanced search tree. Apply Lemma 6.2 to each of them. This immediately establishes that after step (V1) of Procedure `Visit`, Invariant 6.1 holds for the initial chains of $\mathscr{T}$. Now, the proof of correctness can be completed by induction on the number of processed chains, using the fact that we process a chain only when the chains coming into it have been processed and therefore, by Lemma 6.4, those chains satisfy Invariant 6.1.

Let $p_{\mathscr{T}}$ be the number of leaves in $\mathscr{T}$. As for the time analysis of Procedure `Visit`, notice that processing of each initial chain costs $O(\log |W_p|)$ time for a total of $O(p_{\mathscr{T}} \log |W_p|)$ time. We also have that $\Sigma_{\tilde{c}} |\mathscr{P}\mathscr{R}(\tilde{c})| < p_{\mathscr{T}} + |NEW(\mathscr{T})|$, where the sum is taken over all intersection nodes of $\mathscr{T}$ and $NEW(\mathscr{T})$ is the set of new leaves generated in $\mathrm{L}T_{p+1}$ by nodes in $\mathscr{T}$. Using this inequality and the bound in Lemma 6.5, we have that the total time to process the noninitial chains of $\mathscr{T}$ is bounded by $O((p_{\mathscr{T}} + |NEW(\mathscr{T}|) \log^2 |W_p|)$. Accounting also for the time to process all the initial chains of $\mathscr{T}$, we have that this latter bound gives us the time complexity of a visit to a tree $\mathscr{T}$ of the blossom forest of $\mathrm{L}T_p$. Since the number of leaves in the blossom forest of $\mathrm{L}T_p$ is equal to the number of dummy matrices $W_{p,d}\$$ which, in turn, is $O(p)$, we have that the bound of the lemma follows. ∎

7. INTERNAL STRUCTURE EXPANSION—DETAILS

Recall from Section 5 that *Internal Structure Expansion* keeps extension matrices in a priority queue and processes them. An extension matrix can be in the queue either because it is part of the initial configuration of the queue or it has generated new leaves in $\mathrm{L}T_p$. Here we present implementation

details of Procedure `Process-Matrix` and then discuss overall correctness and time analysis of *Internal Structure Expansion.*

7.1. *Preliminaries*

• *Representation of Matrices in the Queue.* Consider a matrix $\tilde{M}$ in the queue and let M be its longest proper prefix. Assume that M is of side length c and let v be its extended locus in $\mathrm{L}T_p$. $\tilde{M}$ is represented in the queue by the following triple: c, v, and the string γ corresponding to the last Lcharacter of $\tilde{M}$ (this latter is encoded through pointers to an occurrence of that string as subrow and subcolumn of W_{p+1}). Define a linear order relation among the nodes of $\mathrm{L}T_p$, say, according to the memory address where the node is stored. The matrices in the priority queue are represented by means of a balanced search tree, lexicographically sorted according to the triples representing them. Therefore, we can efficiently extract the element of maximum priority but we can also search for elements in the queue.

We point out that, at the end of *Frontier Expansion*, we have all the needed information to represent the initial set of matrices in the queue. Indeed, recall from Section 5 that this set is given by the extension matrices corresponding to the leaves for which each root of tree in the blossom forest of $\mathrm{L}T_p$ is responsible in $\mathrm{L}T_{p+1}$. That is, consider a leaf u in $\mathrm{L}T_p$ that is also root of a tree in the blossom forest of $\mathrm{L}T_p$. Independently of how many extensions $M(u)$ has, all those matrices are initially inserted in the queue. Now, fix an extension matrix $\tilde{M} \in \mathscr{E}(u)$ and assume that $\mathscr{R}(u) = \{u\}$, i.e., u is a leaf in $\mathrm{L}T_p$ and in $\mathrm{L}T_{p+1}$. u is the locus of $M(u)$ in $\mathrm{L}T_p$ and $l(u)$ is the side length of $M(u)$. Moreover, the last Lcharacter of $\tilde{M}$ can easily be located as a subrow and subcolumn of W_{p+1} (we omit the details). Assume now that $|\mathscr{R}(u)| > 1$, i.e., u generates new leaves in $\mathrm{L}T_p$. Notice that $\mathscr{R}(u)$ and $\mathscr{E}(u)$ have been computed by *Frontier Expansion*, as stated by Fact 6.1. In particular, for each $\tilde{M} \in \mathscr{E}(u)$, we know one of its occurrences in W_{p+1} as well as the leaf f corresponding to it in $\mathscr{R}(u)$.

We now present a useful subroutine (see also [15]). Let M be a matrix having an extended locus v in $\mathrm{L}T_p$ and let M' be its second suffix. Assume that M is of side length c. We find the extended locus v' of M' in $\mathrm{L}T_p$ as follows.

PROCEDURE `Exlocus`(M').

(`Ex1`) Through its descendant pointer, pick the leaf q pointed to by v. Follow the suffix link from q to, say, q'. We perform expose(q') to get the path from q' to the root of $\mathrm{L}T_p$ in a search tree. Then, using this search tree we find v' by binary search: it is the closest node to the root such that $l(v') \geqslant c-1$, where $c-1$ is the side length of M'.

LEMMA 7.1. *Procedure* `Exlocus` *correctly finds the extended locus of* M' *in* $\mathrm{L}T_p$ *in* $O(\log |W_p|)$ *time, where* M' *is the second suffix of* M.

Proof. Let Z be the matrix of which q is locus in $\mathrm{L}T_p$. Notice that M is prefix of Z. By definition of suffix link, the matrix Z' having locus in q' has the second suffix of Z as prefix. Therefore, the second suffix of M is on the path **p** from q' to the root of $\mathrm{L}T_p$ and we can correctly find it by binary search as we now explain. The nodes of the path **p** are at the leaves of the binary search tree returned by *expose*(q') sorted, from left to right, according to their distance (number of tree edges) from the root. Moreover, for each node u in **p**, we have the value of $l(u)$ stored in it (see Section 2). But the list of l values of nodes in **p**, taken from left to right, is a decreasing sequence of integers. Therefore we can find the node v' in **p** closest to the root such that $l(v') \geqslant c-1$ by binary search, where $c-1$ is the side length of M'. The time bound follows since access to q via descendant pointers, *expose*, and the binary search all take $O(\log |W_p|)$ time.

7.2. *Implementation*

With reference to Section 5, we now consider the implementation of two steps in Procedure `Process-Matrix`, namely, (`PM1`) and (`PM3`).

- *Step* (`PM1`). Check whether the second suffix $\tilde{M}'$ of $\tilde{M}$ is in the queue. Let M be the longest proper prefix of $\tilde{M}$ and assume that it is of side c. We have three cases: (a) $c=0$, i.e., $\tilde{M}$ is of side length one and $\tilde{M}'$ is the empty matrix. Then, $\tilde{M}'$ is not in the queue. (b) $c=1$, i.e., $\tilde{M}$ is a 2×2 matrix. Check directly whether the triple $(0, root, \gamma')$ is in the queue, where *root* is the root of $\mathrm{L}T_p$ and $\gamma' = \tilde{M}[2,2]$. (c) $c>1$. $\tilde{M}$ is represented in the queue by the triple (c, v, γ). Therefore, we know the extended locus v of M in $\mathrm{L}T_p$ and the last Lcharacter γ of $\tilde{M}$. Let M'' be the second suffix of M and let v'' be extended locus of M'' in $\mathrm{L}T_p$. We need to check whether in Q there is a triple $(c-1, v'', \gamma')$, where γ' is the last Lcharacter of $\tilde{M}'$. The only nontrivial task is to find v'' and it can be performed by Procedure `Exlocus`.

FACT 7.1. *The stated implementation of step* (`PM1`) *of Procedure* `Process-Matrix` *correctly checks whether* $\tilde{M}'$ *is in the queue in* $O(\log^2 |W_p|)$ *time.*

Proof. The only nontrivial case is (c), which we now discuss. Correctness follows from Lemma 7.1, since we are using Procedure `Exlocus`. As for the time analysis, the call to `Exlocus` takes $O(\log |W_p|)$ time by Lemma 7.1. Moreover, checking whether the triple $(c-1, v'', \gamma')$ is in the queue takes $O(\log^2 |W_p|)$ time. Indeed, by Lemma 5.2, the maximum number of triples in Q is $O(|W_p|)$. Since Q is represented in a balanced search

tree, we need to perform $O(\log |W_p|)$ compared isons to find the appropriate triple. However, some of those comparisons involve the lexicographic comparison of Lcharacters. As stated in Subsection 2.2, each of those comparisons costs $O(\log |W_p|)$ time. The bound follows. ▮

• *Step* (PM3). Check whether $\tilde{M}'$ has an extended locus z in $\mathrm{L}T_{p+1}$ and, if it does, set up a suffix link from f to z and discard $\tilde{M}'$. We address the first part only, since the second part is standard. When $\tilde{M}'$ is the empty matrix, its locus is the root of $\mathrm{L}T_p$. Assume otherwise. Using Procedure Exlocus, we first find the extended locus v' of M' in $\mathrm{L}T_p$, where M' is the longest proper prefix of $\tilde{M}'$. Indeed, as in the implementation of step (PM1), we use knowledge of M, the longest proper prefix of $\tilde{M}$ and of its extended locus in $\mathrm{L}T_p$. We have to consider two subcases: **(1)** v' is the locus of M' and **(2)** its complement. Consider **(1)**. We check whether the label of any edge leaving v' starts with γ', where γ' is the last Lcharacter of M'. If such an edge (v', z) exists, we have found the extended locus of $\tilde{M}'$ in $\mathrm{L}T_p$, otherwise no such a node exists. Consider (**2**). On the edge (w, v'), consider the Lcharacter β corresponding to γ' (we can identify this Lcharacter by using $l(w)$, i.e., the side length of the matrix having locus in w, and the side length of M'). If $\beta = \gamma'$ then $z = v'$, i.e., v' is the extended locus of $\tilde{M}'$ in $\mathrm{L}T_p$, otherwise no such a node exists.

FACT 7.2. *The stated implementation of step* (PM3) *of Procedure* Process-Matrix *correctly checks whether* $\tilde{M}'$ *has an extended locus in* $\mathrm{L}T_p$ *in* $O(\log^2 |W_p|)$ *time. Moreover, it provides the needed information to step* (PM4) *of Procedure* Process-Matrix, *that is, whether or not* v' *is the locus of* M' *in* $\mathrm{L}T_p$, *where* M' *is the longest proper prefix of* $\tilde{M}'$.

Proof. The only nontrivial case is when $\tilde{M}'$ is not the empty matrix. We limit our discussion to this case only. By Lemma 7.1, we correctly find the extended locus of M' in $\mathrm{L}T_p$. At that point it is a simple exercise to check whether v' is the locus of M' in $\mathrm{L}T_p$. This provides the needed information when one enters step (PM4). Since M' is the longest proper prefix of $\tilde{M}'$, it is an easy matter to show that cases **(1)** and **(2)** correctly find the extended locus of $\tilde{M}'$, if it exists in $\mathrm{L}T_p$.

As for the time analysis, the call to Procedure Exlocus takes $O(\log |W_p|)$ time, by Lemma 7.1. We now bound the time taken by cases **(1)** and **(2)**. Consider case **(1)**. Recall from Subsection 2.2 that we store the edges leaving v' in a balanced search tree (sorted according to the lexicographic order on the first Lcharacter labeling each of them). Therefore, we can find (if it exists) the edge (v', z) by searching γ' in that balanced search tree. Since each comparison of Lcharacters costs $O(\log |W_p|)$ time and the number of edges leaving a node is $O(|W_p|)$, we have that this case can be implemented to take $O(\log^2 |W_p|)$ time. Consider now case **(2)**.

The most costly operation is the comparison of two Lcharacters, which can be done in $O(\log |W_p|)$ time. Therefore, the time bound stated in the fact follows. ▮

Remark 7.1. Procedure `Process-Matrix` must also perform additional operations aimed at maintaining the representation of $\mathrm{L}T_{p+1}$ as a dynamic tree as well as the auxiliary data structures described in Section 2. Those additional tasks are essentially the same as (I1)–(I3) outlined for Procedure `Process` in Section 6.3 and they can be handled in essentially the same way and with analogous time bounds. Details are omitted and left to the reader.

Remark 7.2. Procedure `Process-Matrix` must also handle the partition of the blossom forest of $\mathrm{L}T_{p+1}$ into chains and must keep those chains stored in balanced search trees. By Lemma 5.3, during step (PM2) we need to split a chain, while in all other steps where the Procedure creates suffix links, a chain grows by the addition of a new node to its right. All those operations can be performed as already explained in Subsection 6.3 and within the same time bounds.

7.3. *Correctness and Time Analysis*

LEMMA 7.2. *Internal Structure Expansion correctly identifies the extension matrices satisfying Fact* 3.3 *and correctly carries out the corresponding transformation in* $\mathrm{L}T_p$. *Moreover, for the new leaves, the appropriate suffix links, pointing to and departing from them, are set.* ▮

Proof. Facts 7.1 and 7.2 guarantee that the assumptions of Lemma 5.1 are satisfied, which immediately implies the lemma.

LEMMA 7.3. *Let f be a leaf of* $\mathrm{L}T_{p+1}$ *associated to a matrix* $\tilde{M}$ *inserted in the queue. When M is removed from the queue, the subtree of the blossom forest of* $\mathrm{L}T_{p+1}$ *rooted at f has been computed. That subtree is correctly partitioned into chains, which are represented as balanced search trees.*

Proof. Consider the set of matrices inserted in the queue during *Internal Structure Expansion* and sort them by side length. Consider a matrix $\tilde{M}$ of maximum side length. This matrix must be part of the initial configuration of the queue. Since the initial configuration of the queue is composed of all the extension matrices of the roots of trees in the blossom forest of $\mathrm{L}T_p$, we have that there is a leaf u of $\mathrm{L}T_p$ such that $\tilde{M}$ is extension matrix of $M(u)$. But then, $f \in \mathscr{R}(u)$ and, by Lemma 6.6, we have computed the subtree of the blossom forest of $\mathrm{L}T_p$, which is partitioned into chains represented in balanced search trees. This implies that the lemma is true for all matrices $\tilde{M}$ of maximum side length.

Pick a matrix $\tilde{M}$ of side length d. Assume that the lemma is true for all matrices inserted in the queue and of side length $d' > d$. We show that the lemma holds for $\tilde{M}$ of side length d. We have to consider two cases.

Case (1). Assume that $\tilde{M}$ is part of the initial configuration of the queue. Then, the proof is analogous to the one already given for matrices of maximum side length.

Case (2). Assume that $\tilde{M}$ is not a part of the initial configuration of the queue. When $\tilde{M}$ is extracted from the queue, all matrices of side length $d+1$ have already been processed. Moreover, by Lemma 5.1, all the leaves associated to those matrices of side length $d+1$ have a valid suffix link (in $\mathrm{L}T_{p+1}$) departing from each of them. Therefore, at the time $\tilde{M}$ is extracted from the queue, the leaf f associated with it has already all the needed suffix links pointing to it. By Remark 7.2, we correctly split the chain of which f is part, when f is an intersection node. Since the statement of the lemma holds for all leaves of $\mathrm{L}T_{p+1}$ having a suffix link pointing to f, we have that it holds also for f. ∎

LEMMA 7.4. *At the end of Internal Structure Expansion we have the blossom forest of* $\mathrm{L}T_{p+1}$, *which is correctly partitioned into chains.*

Proof. By Lemma 6.6, at the end of *Frontier Expansion* we have computed part of the blossom forest of $\mathrm{L}T_{p+1}$. Now, by Lemma 7.2, *Internal Structure Expansion* correctly extends this part of the blossom forest of $\mathrm{L}T_{p+1}$ to become the whole forest. The fact that each tree of this blossom forest is correctly partitioned into chains, represented in balanced search trees, is granted by Lemma 6.6 and 7.3.

LEMMA 7.5. *Internal Structure Expansion takes* $O((|\mathrm{L}T_{p+1}| - |\mathrm{L}T_p| + p) \log^2 |W_p|)$ *time.*

Proof. The work done by *Internal Structure Expansion* is bounded by the number of matrices inserted in the queue. By Lemma 5.2, that number is $O(|\mathrm{L}T_{p+1}| - |\mathrm{L}T_p| + p)$. Now, by Facts 7.1–7.2 and the Remarks 7.1–7.2 about bookeeping and maintenance of the blossom forest of $\mathrm{L}T_{p+1}$, the time to process each of those matrices is $O(\log^2 |W_p|)$.

8. COMPARING LCHARACTER EFFICIENTLY

In this section, we show how to compare two Lcharacters of W_{p+1} efficiently, i.e., in $O(\log |W_{p+1}|)$ time. We start by introducing some data structures.

Consider the matrix W_p and let $row_{1,p}, ..., row_{p,p}$ be the strings corresponding to its rows, taken from left to right. Let $T_{rows,p}$ be the suffix tree

representing the suffixes of those strings. Here we use the definition of suffix tree introduced by Ukkonen [38], so there is no one-to-one correspondence between the leaves of $T_{rows,\,p}$ and the suffixes of $row_{1,\,p}$, ..., $row_{p,\,p}$. For $1 \leqslant i \leqslant p$, let $row_{i,\,p}\$$ be a dummy string that has $row_{i,\,p}$ as second suffix. In analogy with LT_p, we define suffix links on the leaves of $T_{rows,\,p}$, including the dummy leaves representing the p dummy strings. Also in this case, the external suffix links form a forest of trees, which we denote as the blossom forest of $T_{rows,\,p}$. Moreover, we assume that auxiliary data structures, analogous to the ones defined for LT_p (see Subsection 2.2), are also available for $T_{rows,\,p}$. In particular, both $T_{rows,\,p}$ and each tree in its blossom forest are represented as dynamic trees [34]. Let $col_{1,\,p}$, $col_{2,\,p}$, ..., $col_{p,\,p}$ be the strings corresponding to the columns of W_p, taken from top to bottom. Define $T_{cols,\,p}$ in analogy with $T_{rows,\,p}$ including dummy leaves associated to the dummy strings $col_{1,\,p}\$$, $col_{2,\,p}\$$, ..., $col_{p,\,p}\$$. We assume that $T_{cols,\,p}$ has analogous auxiliary data structures as $T_{rows,\,p}$.

The procedure that compares Lcharacters efficiently follows closely the one given in [15], with one important technical difference due to the use of different versions of suffix trees. Indeed, in [15], the procedure uses suffix trees analogous to $T_{rows,\,p}$ and $T_{cols,\,p}$ to represent the rows and columns of the input matrix as strings. Since in [15] the input is given *off-line*, those trees are defined as in McCreight [28] and therefore they preserve the one-to-one correspondence between their leaves and the suffixes of the strings represented in them. Such a one-to-one correspondence is very useful in efficiently finding some information needed by the procedure. Here we do not have such a one-to-one correspondence (the suffix trees are defined as in Ukkonen [38]) and we have to find alternative ways to gather efficiently the same information. For that reason, we introduce Clusters in Subsection 8.3.

The remainder of this section is organized as follows. We first give an outline of the algorithm assuming that some of its steps can be implemented efficiently. Then, we develop the needed machinery to achieve this goal. That involves the transformation of $T_{rows,\,p}$ and $T_{cols,\,p}$ in $T_{rows,\,p+1}$ and $T_{cols,\,p+1}$, respectively, and the construction of clusters for both of those trees. We present only the procedures for $T_{rows,\,p}$, since those for $T_{cols,\,p}$ are analogous.

8.1. *Outline of the Algorithm to Compare Lcharacters*

Consider two Lcharacters, from submatrices of W_{p+1}, that we need to compare. We assume that each of those Lcharacters is given in terms of a quadruple encoding it as a chunk. Therefore, we can compute in constant time which subrow and subcolumn of W_{p+1} gives each Lcharacter as a string of Σ^*. Let α_1 and α_2 be those two strings and assume that they are

both of length $2g-1$. Assume that the prefix of length $g-1$ of α_1 (α_2, resp.) is a prefix of $row_{k_1}[j_1, p+1]$ ($row_{k_2}[j_2, p+1]$, resp.) and assume that the suffix of length g of α_1 (α_2, resp.) is prefix of $col_{k'_1}[j'_1, p+1]$ ($col_{k'_2}[j'_2, p+1]$, resp.).

PROCEDURE `Compare`.

(C1) Identify a leaf f_1 (f_2, resp.) of $T_{rows,\, p+1}$ that is descendant of the extended locus of $rows_{k_1}[j_1, p+1]$ ($row_{k_2}[j_2, p+1]$, resp.). Using $T_{cols,\, p+1}$, perform the same step for $col_{k'_1}[j'_1, p+1]$ and $col_{k'_2}[j'_2, p+1]$.

(C2) Find the lowest common ancestor v_1 of f_1 and f_2 in $T_{rows,\, p+1}$. Using $T_{cols,\, p+1}$, we perform the same operations for $col_{k'_1}[j'_1, p+1]$ and $col_{k'_2}[j'_2, p+1]$ to identify a node v_2 analogous to v_1.

(C3) Using v_1 and v_2, establish whether α_1 is lexicographically smaller than or equal to α_2 and compute the length of the longest prefix that those two strings have in common.

LEMMA 8.1. *Assume that $T_{rows,\, p+1}$ and $T_{cols,\, p+1}$ are available and that step* (C1) *of Procedure* `Compare` *can be correctly implemented to take $O(\log p)$ time. Then, Procedure* `Compare` *compares two Lcharacters from submatrices in W_{p+1} and establishes which one is lexicographically smaller than or equal to the other in $O(\log p)$ time. In the same time bound, we can also compute the length of the longest prefix that those two Lcharacters have in common.*

Proof. By assumption, step (C1) can be correctly executed in $O(\log\ p)$ time. Now, step (C2) can be correctly implemented to take $O(\log\ p)$ time. Indeed, $T_{rows,\, p+1}$ is a dynamic tree. Therefore, finding the lowest common ancestor of f_1 and f_2 takes $O(\log\ p)$ time [34]. Analogous considerations hold for $T_{cols,\, p+1}$. As for (C3), it can be implemented to take constant time. Details are as in [15].

8.2. *Transformation of $T_{rows,\, p}$ in $T_{rows,\, p+1}$*

When W_p becomes W_{p+1}, a new character is appended at the end of each string $row_{i,\, p}\$$, $1 \leqslant i \leqslant p$. Moreover, we get a new string: $row_{p+1,\, p+1}$. We must modify $T_{rows,\, p}$ accordingly. As it can be easily verified, appending a new character at the end of $row_{i,\, p}\$$, $1 \leqslant i \leqslant p$, causes changes in $T_{rows,\, p}$ analogous to the ones we have described for $\mathrm{L}T_p$, when W_p becomes W_{p+1} (see Section 3). Using this observation, the algorithm is the following:

PROCEDURE `Transform`.

(T1) Apply to $T_{rows,\, p}$ the same algorithms (specialized to strings) described in Sections 3–7. Let $T'_{rows,\, p}$ be the resulting tree.

(T2) Insert all suffixes of $row_{p+1,\,p+1}\$$ in $T'_{rows,\,p}$. The result is $T_{rows,\,p+1}$.

LEMMA 8.2. *Assume that $T_{rows,\,p}$ is available. When W_p becomes W_{p+1}, $T_{rows,\,p}$ can be transformed in $T_{rows,\,p+1}$ in $O((|T_{rows,\,p+1}| - |T_{rows,\,p}| + p) \log p)$ time.*

Proof. Step (T1) is a specialization to strings of the procedures presented in Sections 3–7. Since each comparison of characters now costs $O(1)$ time, the time bounds in Lemmas 6.6 and 7.2 reduce to $O((|T'_{rows,\,p}| - |T_{rows,\,p}| + p) \log p)$. As for the implementation of step (T2), we claim that it can be done in $O((|T_{rows,\,p+1}| - |T'_{rows,\,p}| + p) \log p)$ time. Therefore, the time bound of the lemma follows. In order to sustain our claim, we limit ourselves to notice that step (T2) is a nearly standard exercise and we provide only a sketch of the solution.

- *Initial Step.* We start by inserting a dummy leaf representing $row_{p+1,\,p+1}\$$ and a dummy edge from the root to the new leaf. Then, we search for the extended locus v of the longest prefix y of $row_{p+1,\,p+1}$ in $T'_{rows,\,p}$. If $y = row_{p+1,\,p+1}$ we are done. Otherwise, we create a new leaf that is locus of $row_{p+1,\,p+1}$ and possibly a new internal node, locus of y.

- *The Second Suffix of $row_{p+1,\,p+1}$.* Assume that $y \neq row_{p+1,\,p+1}$, i.e., $row_{p+1,\,p+1}$ is not already in $T'_{rows,\,p}$. We need to check whether the second suffix of $row_{p+1,\,p+1}$ has extended locus in $T'_{rows,\,p}$. (If it does, we are done, since all of its remaining suffixes will have extended locus in $T'_{rows,\,p}$. Otherwise, $T'_{rows,\,p}$ must be modified to insert a new leaf—and possibly a new internal node—representing the second suffix of $row_{p+1,\,p+1}$.) Using Procedure Exlocus (see Section 7) specialized to strings and applied to $T'_{rows,\,p}$, we identify the extended locus v' of y', the second suffix of y. Using v' we can identify exactly where y' ends in $T'_{rows,\,p}$. Starting from there, we search for $row_{p+1,\,p+1}[|y'|+1,\, p+1]$ until we either find the extended locus of $row_{p+1,\,p+1}[2,\, p+1]$ (and we are done) or we "fall off" the tree, i.e., a mismatch is found. In the second case, we have identified where to "insert" the leaf representing $row_{p+1,\,p+1}[2,\, p+1]$.

- *The kth Suffix of $row_{p+1,\,p+1}$.* In order to insert the kth suffix of $row_{p+1,\,p+1}$ we do the same as for the second, where the role of $row_{p+1,\,p+1}$ is played by its $(k-1)$st suffix. ∎

8.3. *Clusters of $T_{rows,\,p+1}$*

We need some terminology. A *partial cluster* is a tree composed of suitably connected trees of the blossom forest of $T_{rows,\,p+1}$. A partial cluster is made final by adding an edge from the root of the partial cluster to one

of its (suitably chosen) descendants, i.e., we close a cycle in a partial cluster. We impose the constraint that there is exactly one cycle in a final cluster. Both partial and final clusters are maintained as dynamic trees.

The algorithm for the computation of final clusters proceeds in stages. In the initial stage, each tree of the blossom forest of $T_{rows,\,p+1}$ is in a partial cluster by itself. A stage consists of *linking* a partial cluster either to a partial cluster (to yield a new partial cluster) or to a final cluster (to yield a new final cluster). We stop when all clusters are final. During all stages, the root of a (partial or final) cluster corresponds to the root of a tree in the blossom forest of $T_{rows,\,p+1}$. Details are as follows.

PROCEDURE `Clusters`.

(`C11`) Mark each tree of the blossom forest of $T_{rows,\,p+1}$ as a partial cluster.

(`C12`) While there are partial clusters do: Pick a partial cluster C and perform steps (`C13`)–(`C17`). (Let u be the root of C and let x be the string that has locus in $u \in T_{rows,\,p+1}$. Moreover, let x' be the second suffix of x.)

(`C13`) Using the suffix link of u, find the extended locus u' of x' in $T_{rows,\,p+1}$.

(`C14`) Using the descendant pointer of u', pick the leaf $f \in T_{rows,\,p+1}$ pointed to by u'. (It is a leaf in the subtree of $T_{rows,\,p+1}$ rooted at u').

(`C15`) Check whether u and f are in the same partial cluster.

(`C16`) If they are not, add an edge from u to f. (This has the effect of linking the two clusters to create a new one.) If the cluster of which f is part is final, then the new cluster is also final.

(`C17`) If u and f are in the same cluster, we record f and we mark the cluster as final (we have just closed a cycle). Moreover, we also record the length of the cycle.

LEMMA 8.3. *Final Clusters of the blossom forest of $T_{rows,\,p+1}$ can be computed in $O(p \log p)$ time.*

Proof. We can have at most $p+1$ distinct trees in the blossom forest of $T_{rows,\,p+1}$. Therefore, we can have at most $p+1$ stages for the construction of clusters. All we need to show is that steps (`C13`)–(`C17`) can be implemented to take $O(\log p)$ time each. Now, for steps (`C13`)–(`C14`), this is obvious. As for step (`C15`), we do *expose*(f) to get the path $\hat{p}$ from f to the root of its cluster. If u is that root, then u and f are in the same cluster. We point out that, once we have done *expose*(f), we also know how many nodes are on $\hat{p}$ (that information is stored at the root of the balanced search tree storing $\hat{p}$—see [34]). Step (`C16`) can be implemented via the

operation *link* defined for dynamic trees. Finally, the number of nodes in $\hat{p}$ gives also the length of the cycle that is required in step (C17). ▮

Remark 8.1. Recall from Section 6 that chains, i.e., paths of unary nodes, in each tree of the blossom forest of LT_p are kept into balanced search trees. It is a simple exercise to show that, while clusters are linked together, chains in the resulting cluster can still be represented by balanced search trees, at no additional expense with respect to the time bound stated in Lemma 8.3. Therefore, we assume that chains in final clusters are represented as balanced search trees.

We now state some useful facts related to clusters. Consider a final cluster $\mathscr{C}$. Let s be the length of the cycle in it. Pick a node v in C and let x be the string that has locus in $v \in T_{rows,\, p+1}$. Assume that x is of length e. Let t be the number of nodes (including v) on the path pp from v to the root of C.

FACT 8.1. *Consider suffix* $x[i:e]$ *and assume that* $i \leqslant t$. *Let* $\hat{v}$ *be the ith node on the path pp. The ith suffix of x is prefix of the string that has locus in* $\hat{v} \in T_{rows,\, p+1}$, *independently of whether* $e \leqslant t$ *or* $e > t$.

Proof. Correctness of our claims can be easily shown by repeatedly applying the following observation to the nodes of *pp*: when v is child of v' in a cluster, the second suffix of the string having locus in v is prefix of the string having locus in v'. Indeed, either v and v' are connected by a suffix link going from v to v' or they are linked together by Procedure Cluster. ▮

FACT 8.2. *Consider the ith suffix of x and assume that* $e \geqslant i \geqslant t$. *Let* $i' = (i - t) \bmod s$ *and let* $\hat{v}$ *be the* i'*th node on the cycle in the cluster* (*we start counting nodes in a cycle from the root of the cluster, which is node zero of the cycle*). *The ith suffix of x is prefix of the string that has locus in* $\hat{v}$.

Proof. It is similar to the one outlined for Fact 8.1.

FACT 8.3. *Given v, the length of the cycle in C and i, we can compute* $\hat{v}$ *in* $O(\log p)$ *time.*

Proof. Indeed, each final cluster C is represented as a dynamic tree. Therefore, we can perform *expose*(v) to get the path pp from v to the root of C in a binary search tree. At that point, we also know t, the length of pp, and s, the length of the cycle in C. Therefore, we can establish whether Fact 8.1 or 8.2 applies. Finally, using the binary search tree representing the path pp we can pick the appropriate node in it by binary search (details are omitted). ▮

8.4. *Comparing Lcharacters–Final Details and Time Analysis*

Here we show that step (C1) of Procedure Compare can be implemented to take $O(\log p)$ time and then summarize the results of this section for later use.

LEMMA 8.4. *Assume that $T_{rows,\,p+1}$ and $T_{cols,\,p+1}$ are available and that their final clusters have been computed. Step* (C1) *of Procedure* Compare *can be implemented to take $O(\log p)$ time.*

Proof. We discuss only the case of the identification of a leaf f_1 of $T_{rows,\,p+1}$ that is descendant of the extended locus of $row_{k_1}[j_1, p+1]$ in that tree. The other cases can be handled similarly. Let v be the dummy leaf associated to $row_{k_1,\,p+1}\$$. Notice that $row_{k_1}[j_1, p+1]$ is the j_1th suffix of $row_{k_1,\,p+1}$. We can identify the root of the final cluster C of which v is part using the operation $expose(v)$. That also provides knowledge of the length of the cycle in C. Now, we have all the needed information to apply Fact 8.3 to identify a node $\hat{v}$ such that $row_{k_1}[j_1, p+1]$ is prefix of the string having locus in $\hat{v}$. Then, from $\hat{v}$, we can pick a leaf f_1 in the subtree rooted at $\hat{v}$. The lemma follows. ▮

LEMMA 8.5. *Assume that $T_{rows,\,p}$ and $T_{cols,\,p}$ are available. When W_p becomes W_{p+1}, we can* **(a)** *transform $T_{rows,\,p}$ in $T_{rows,\,p+1}$ in $O((|T_{rows,\,p+1}| - |T_{rows,\,p}| + p)\log p)$ time*; **(b)** *compute the final clusters of $T_{rows,\,p+1}$ in $O(p \log p)$ time. Analogous results hold for $T_{cols,\,p+1}$ Moreover, we can* **(c)** *compare two Lcharacters from submatrices in W_{p+1} and establish which one is lexicographically smaller than or equal to the other in $O(\log p)$ time*; **(d)** *in the same time bound, we can also compute the length of the longest prefix that those two Lcharacters have in common.*

Proof. Parts **(a)** and **(b)** follow from Lemmas 8.2 and 8.3, respectively. Parts **(c)** and **(d)** follow by combining Lemma 8.4 with Fact 8.1.

9. ON-LINE CONSTRUCTION OF LSUFFIX TREES— CORRECTNESS AND TIME ANALYSIS

In this section we address correctness and summarize the time taken to transform $\mathrm{L}T_p$ in $\mathrm{L}T_{p+1}$ and to maintain the blossom forest and all the additional auxiliary data structures. Moreover, we also analyze the time taken to build on-line the Lsuffix tree of an $n \times n$ matrix A. Let $|T_p| = |\mathrm{L}T_p| + |T_{rows,\,p}| + |T_{cols,\,p}|$.

LEMMA 9.1. Frontier Expansion *and* Internal Structure Expansion *correctly transform $\mathrm{L}T_p$ into $\mathrm{L}T_{p+1}$ and correctly maintain all the auxiliary data structures. The total time is $O((|T_{p+1}| - |T_p| + p)\log^2 |W_p|)$.*

Proof. The fact that LT_p is correctly transformed into LT_{p+1} follows from Lemmas 6.6 and 7.2. As for the auxiliary data structures, the blossom forest of LT_{p+1} is available, as stated in Lemma 7.4. $T_{rows,\,p+1}$ and $T_{cols,\,p+1}$ are also available, as stated in Lemma 8.5(a). As for the remaining data structures, we have outlined in Subsections 6.3 and 7.2 how they can be maintained while LT_p is transformed in LT_{p+1}. The time analysis follows by combining the bounds in Lemmas 6.6, 7.5, and 8.5. ∎

THEOREM 9.1. *The total time needed to build on-line the Lsuffix tree for an $n \times n$ matrix A, together with all of its auxiliary data structures, is $O(n^2 \log^2 n)$.*

Proof. The Lsuffix tree for A is obtained as a sequence of trees $LT_1, LT_2, ..., LT_n$. Since $|LT_1| = O(1)$ and it can be built in constant time we have, using Lemma 9.1, that the total time needed to obtain the entire sequence of trees is $O(\sum_{p=1}^{n-1}(|T_{p+1}| - |T_p| + p)\log^2 |W_p|)$. Since $|W_p| \leqslant n^2$, for each p, and $|T_n| - |T_1| = O(n^2)$, that sum is $O(n^2 \log^2 n)$. ∎

10. PATTERN MATCHING–CHECKING FOR AN OCCURRENCE

Let $PAT[1:m, 1:m]$ be a pattern matrix. Here we address the problem of checking whether PAT occurs in W_p.

Once LT_p is available, we can solve the stated problem by identifying the extended locus u of (the Lstring corresponding to) PAT in LT_p. The identification of u is a standard search of a string in a trie. However, care must be taken in the implementation of this search in LT_p. Indeed, the degree of each node LT_p may be as large as $O(|T_p|)$ and the time of a naive search procedure depends, among other parameters, on the maximum degree of each node in LT_p. A similar problem has been discussed and solved in [15]. In that case, some auxiliary data structures are introduced and the naive search procedure is modified so that, intuitively, the degree of each node in LT_p looks like as if it were bounded by $|\Sigma|$. Here we use exactly the same ideas as in [15], except that the auxiliary data structures must be dynamically changed. This is due to the fact that LT_p is transformed into LT_{p+1} and we need all the auxiliary data structures also for this latter tree. Therefore, at a high level, the pattern matching procedure and the auxiliary data structures are as in [15], but the low level implementation details for the maintenance of the auxiliary data structures are different.

The remainder of this section is organized as follows. We first introduce the auxiliary data structures, then discuss how to check for an occurrence of PAT in W_p, and finally outline how to dynamically change those data structures.

10.1. *Auxiliary Data Structures*

The auxiliary data structures are ordinary compacted tries $KEY(v)$, for each internal node $v \in LT_p$, storing strings in Σ^h, for some integer h. We now define $KEY(v)$.

For a given internal node $v \in LT_p$, let $w_1, w_2, ..., w_{c(v)}$ be the list of its children in LT_p. Let $(i_g, j_g, l(v), d_g)$ be the chunk labeling the arc from v to w_g, for $1 \leqslant g \leqslant c(v)$. Notice that all those chunks start with distinct Lcharacters that are of the same length $h = 2l(v) + 1$. Moreover, from the quadruple $(i_g, j_g, l(v), d_g)$, we can infer (in constant time) which subrow and subcolumn of W_p gives the first Lcharacter of that chunk. Let $string(w_g)$ be the string corresponding to that Lcharacter.

$KEY(v)$ is the compacted trie storing the strings $string(w_g)$, for $1 \leqslant g \leqslant c(v)$. Since no prefix of $string(w_g)$ is prefix of any other $string(w_{g'})$ (all arcs leaving v start with a different Lcharacter), there is a one-to-one correspondence between the leaves of $KEY(v)$ and the child of v. So, $KEY(v)$ has $O(c(v))$ leaves and internal nodes. Moreover, the substrings labeling the arcs of $KEY(v)$ can be represented in constant space by means of triples of integers, i.e., $string(w_g)[x: y]$ is represented by (x, y, g). So, the size of $KEY(v)$ is $O(c(v))$ and the total size of those trees is $O(|W_p|)$. We also point out that we store edges outgoing each vertex of $KEY(v)$ in a balanced search tree, sorted according to the first character on their labels. During a traversal of $KEY(v)$, that allows us to select which edge "to jump to" in $O(\log |\Sigma|)$ time. Moreover, $KEY(v)$ is maintained as a dynamic tree [34].

10.2. *Search for the Extended Locus of PAT in* LT_p

As already stated, the search for the extended locus of PAT in LT_p is a nearly standard search for the extended locus of an Lstring in a compacted trie defined over Σ. Indeed, given that we have reached node $v \in LT_p$, we select the child of v to jump to by using $KEY(v)$ and then we traverse the edge from v to that child comparing the Lcharacters on that edge with the corresponding ones in PAT. The entire procedure as well as its time analysis are exactly as in [15]. For convenience of the reader here we simply outline how to jump from v to one of its children in LT_p and how to traverse an edge of LT_p.

- *Jumping in* LT_p. Assume that, during the search procedure, we have reached an internal node $v \in LT_p$. Assume that $l(v) \leqslant m$ and that v is locus of $PAT[1 : l(v), 1 : l(v)]$. The child of v to jump to is selected as follows. Let $w_1, ..., w_{c(v)}$ be the children of v. Moreover, let *string* be the string obtained by concatenating subrow $PAT[l(v) + 1, 1 : l(v)]$ with subcolumn $PAT[1 : l(v) + 1, l(v) + 1]$. Such a string is of length $2l(v) + 1$,

which is also the length of $string(w_g)$, for $1 \leqslant g \leqslant c(v)$. The child w_g of v that we need to select must be the one such that *string* matches $string(w_g)$. We identify that w_g by traversing $KEY(v)$ by means of *string*. That traversal takes $O((l(v)) \log |\Sigma|)$ time because, given the triples labeling the arcs of $KEY(v)$, we can access each character of the string they encode in constant time. Moreover, given a character of Σ and a node $c \in KEY(v)$, we can select the arc outgoing c whose label starts with that character in $O(\log |\Sigma|)$ time (recall the definition of $KEY(v)$). Notice that if no such w_g exists, there is no occurrence of the pattern in W_p.

- *Traversing an Edge of* $\mathrm{L}T_p$. Assume that we have found w_g. We need now to check that the chunk on the arc from v to w_g matches the corresponding chunk of $PAT[1 : \min(m, l(w_g)),\ 1 : \min(m, l(w_g))]$. That can be done in $O(\min(m, l(w_g)^2 - l(v)^2))$ time through a standard pairwise character comparison.

We have (see [15]):

THEOREM 10.1. *Checking whether an $m \times m$ matrix PAT occurs in W_p takes $O(m^2 \log |\Sigma|)$ time.*

10.3. *Updating the KEY Data Structures*

As discussed in Section 3, when $\mathrm{L}T_p$ is transformed in $\mathrm{L}T_{p+1}$, we generate new internal nodes and new edges. Some of those new edges have a node of $\mathrm{L}T_p$ as one of its endpoints while others have a newly generated node. Those changes affect also the auxiliary data structures *KEY*. Indeed, for a newly generated node $v \in \mathrm{L}T_{p+1}$, we have to create $KEY(v)$. On the other hand, if v is an "old" node, i.e., one that already existed in $\mathrm{L}T_p$, then we have to add to $KEY(v)$ the new strings corresponding to the first Lcharacter of the label on each of the new edges leaving v.

We first outline the basic step of adding a new string α to an arbitrary trie *KEY* and then discuss how it is used to update a particular *KEY* data structure. Procedure `IS` inserts a string α in a generic trie *KEY* under the assumption that: (a) we know the string β in *KEY* that has the longest prefix in common with α and the length l of this prefix; (b) we know the leaf $f \in KEY$ that is locus of β.

PROCEDURE `IS` ($KEY(v)$, α).

(IS1) Perform $expose(f)$ to get the path from f to the root of $KEY(v)$ in a balanced search tree. Then find the closest node $w' \in KEY$ to the root such that $l \leqslant l(w')$.

(IS2) If $l = l(w')$, create a new leaf in *KEY* that becomes child of w' (and which is the locus of α in *KEY*). The new edge is labeled with $\alpha[l+1, q]$, where q is the length of α.

(IS3) If $l < l(w')$, break the edge $(parent(w'), w')$ to create a node $\hat{w}$, locus of $\alpha[1, l]$. Then, proceed as in step (IS2) but with $\hat{w}$ playing the role of w'.

FACT 10.1. *Procedure* IS *takes* $O(\log |KEY(v)|)$ *time, where* $|KEY(v)|$ *is the number of nodes and edges in* $KEY(v)$, *assuming that*: (a) *we know the string* β *in* KEY *that has the longest prefix in common with* α *and the length* l *of this prefix*; (b) *we know the leaf* $f \in KEY$ *that is locus of* β. ▌

Proof. Step (IS1) can be performed in $O(\log |KEY(v)|)$ time, since $KEY(v)$ is represented as a dynamic tree and the appropriate node w' can be found by binary search using the balanced search tree representing the path from f to the root of LT_p. The other two steps can be implemented to take constant time each. ▌

We can use Procedure IS as follows. Assume that a new edge (v, g) has just been created during the transformation of LT_p in LT_{p+1}. Recall from Subsections 6.3 and 7.2 (see for instance, (I3) in Subsection 6.3 and Remark 7.1 in Subsection 7.2) that when a new edge is created we update the balanced search tree giving the edges leaving v sorted according to the first Lcharacter of the chunks labeling them. As soon as the new edge is inserted into the balanced search tree, we use Procedure IS to insert the first Lcharacter labeling that edge in $KEY(v)$. We remark, leaving out the details, that, as a by-product of this insertion, we also have the information required in input by IS.

LEMMA 10.1. *The update of the* KEY *data structures can be performed without increasing the time taken to transform* LT_p *into* LT_{p+1}.

Proof. Observe that the total number of insertions into the KEY data structures is bounded by the number of new edges that we generate in LT_p to transform it into LT_{p+1}. By Fact 10.1, the time for each insertion is $O(\log |KEY(v)|) \leqslant O(\log |W_p|)$. Therefore, the total time to update the KEY data structures for LT_{p+1} is still within the bounds given in Lemma 9.1.

11. PATTERN MATCHING–FINDING ALL OCCURRENCES

Here we show how to find all occurrences of a matrix PAT in W_p. We assume that the trees of the blossom forest of LT_p have been assembled

into final clusters. Moreover, for each of those final clusters, we also assume that chains are represented into balanced search trees (recall Remark 8.1). To this end, we can use the algorithm outlined in Section 8 for the computation of the clusters of $T_{rows,\,p}$. This additional preprocessing step still takes time within the bounds of Lemma 9.1.

The main steps of the procedure that identifies all occurrences of PAT in W_p are as follows:

PROCEDURE `All-Occurrences`.

(A01) Find the extended locus u of PAT (see Section 10.2). Then, visit the subtree of $\mathrm{L}T_p$ rooted at u and mark all leaves in that subtree.

(A02) For each final cluster C that has a marked node (recall that the leaves of $\mathrm{L}T_p$ are nodes of clusters), use Procedure `Process-Clusters` to identify occurrences of PAT in W_p that can be charged to the marked nodes in C.

In order to present Procedure `Process-Cluster`, we need some observations about marked nodes and the leaves in a final cluster C. Consider a leaf $q \in C$, which must also be a dummy leaf in $\mathrm{L}T_p$. Therefore, it is associated with a dummy matrix, say $W_{p,\,d}\$$. Consider the sth suffix of $W_{p,\,d}\$$. It uniquely identifies position $(d+s, s-1)$ of W_p, when $d \geqslant 0$), and position $(s-1, -d+s)$, when $d<0$. We can identify all occurrences of PAT in positions of W_p "centered around diagonal d" by identifying all suffixes of $W_{p,\,d}\$$ that have PAT as prefix. To this end, the following two facts are useful. We omit their proof and limit ourselves to mention that they are a direct consequence of how final clusters are assembled.

FACT 11.1. *Let v be a marked node on the path from q to the root of C and let s be the distance of q from v, i.e., the number of nodes from q to v on that path (including q and v). The sth suffix of $W_{p,\,d}\$$ has PAT as prefix if and only if it is of side length at least m. Moreover, given p, d, s, and m, we can check for this latter condition in constant time.*

FACT 11.2. *Let c be the length of the cycle in C. Assume that v is the node number c' on the cycle (the root of C is the zeroth node on the cycle) and assume that it is marked. Let s' be the distance of q from the root of C. The sth suffix of $W_{p,\,d}\$$, $s = s' + tc + c'$, and $t \geqslant 0$ has PAT as prefix if and only if it has side length at least m.*

Notice that if a marked node v "flags" an occurrence of PAT for $s_0 = s' + c'$, then we have a sequence of occurrences at suffixes $s_t = s' + tc + c'$ until we find a suffix that has side length less than m. We now give an outline of Procedure `Process-Cluster`.

PROCEDURE `Process-Cluster`(C).

(PC1) Identify all leaves of C that descend from marked nodes and compute the cycle length of C. For each leaf q so identified perform the following two steps.

(PC2) Process all marked nodes on the path from q to the root of C in order of increasing distance from q. For each such a node, use Fact 11.1 to find the appropriate occurrences of PAT in W_p.

(PC3) For each marked node on the cycle, compute its number in the cycle and then use Fact 11.2 to identify the appropriate occurrences of PAT in W_p.

LEMMA 11.1. *The total time taken by all calls to Procedure* `Process-Cluster` *is* $O(occ \log p)$, *where occ is the number of occurrences of PAT in* W_p.

Proof. Fix a cluster C that has marked nodes in it. Fix a leaf q in C and assume that the leaf q is locus of the dummy matrix $W_{p,d}\$$. Notice that, for each marked node satisfying Facts 11.1–11.2, a distinct suffix of $W_{p,d}\$$ is reported as an occurrence of PAT in W_p. Therefore, the number of leaves q identified in step (PC1) is bounded by the number of occurrences of PAT in W_p that can be charged to C. Let occ_C be that number.

Now, since we know the marked nodes in C and the chains in C are represented by balanced search trees, we can perform step (PC1) in $O(occ_C \log p)$ time. Indeed, all we need to do is visit the subtrees of C that have roots at the "lowest" marked nodes in C. We do not need to traverse chains node-by-node because we can jump from end-to-end in a chain (they are stored into balanced search trees).

Let r be the number of marked nodes on the path from q to the root of C. It is a standard exercise to identify those nodes in $O(r \log r)$ time. Since each of those nodes gives a distinct occurrence of PAT in W_p, the time that can be charged to step (PC2) during the execution of Procedure `Process-Cluster`(C) is again bounded by $O(occ_C \log p)$. Similar arguments hold for step (PC3). Taking the sum of those times over all clusters involved, we obtain the claimed bound.

THEOREM 11.1. *Finding all occ occurrences of PAT in* W_p *takes* $O(m^2 \log |\Sigma| + occ \log p)$ *time.*

Proof. By Theorem 10.1, we can check whether PAT occurs in W_p in $O(m^2 \log |\Sigma|)$ time. That also gives us the extended locus u of PAT in LT_p. The number of leaves in the subtree of LT_p rooted at u is bounded by occ, the number of occurrences of PAT in W_p. Therefore, step (AO1) of Procedure `All-Occurrences` takes $O(m^2 \log |\Sigma| + occ)$ time. Using the time bound in Lemma 11.1 for step (AO2), the theorem follows. ∎

12. DATA COMPRESSION–SLIDING THE WINDOW

In this section we describe the data structures and the algorithms needed to dynamically maintain the information represented by the window described in Subsection 1.2 while it sweeps matrix A. Recall from that section that those data structures must support the query *Match* also described there and that we will discuss in Section 13.

Recall from Subsection 1.2 that $F_{h(p)}$ is the "L-shaped" window that starts in row and column p and ends in row and column $p+h-1$ of the $n \times n$ matrix A. For $d \in D_{h(p)}$, let $F_{h(p),d}$ be the suffix of $W_{h(p),d}$ of longest side length that is fully within $F_{h(p)}$. Notice that $F_{h(p),d}$ is for $F_{h(p)}$ analogous to $W_{h(p),d}$ for $W_{h(p)}$. Let $\mathrm{L}F_{h(p)}$ be the Lsuffix tree for window $F_{h(p)}$. It represents all suffixes of matrices $F_{h(p),d}\$$, $|d| < h(p)$. Moreover, let $T_{rows,h(p)}$ ($T_{cols,h(p)}$, resp.) denote the suffix tree for the rows (columns, resp.) of $F_{h(p)}$. The main data structures that we use to represent the information in the window $F_{h(p)}$ are $\mathrm{L}F_{h(p)}$, $T_{rows,h(p)}$, and $T_{cols,h(p)}$. The first one represents all square submatrices in the window while the other two are needed for a fast implementation of the query *Match*. We refer to those data structures as the *trees* for the window $F_{h(p)}$. At a high level, the algorithm sliding the window over the matrix A is as follows:

PROCEDURE `Slide-Window`.

(SW1) `Initialize`

(SW2) For $p = h$ to $n - h + 1$ do: `Advance`

Procedure `Initialize` builds the trees for $F_{h(1)}$, while Procedure `Advance` must transform the trees for $F_{h(p)}$ into the ones for $F_{h(p+1)}$. We anticipate that this involves insertion and deletion of nodes and edges from the trees for $F_{h(p)}$.

LEMMA 2.1. *Procedure* `Initialize` *can be implemented to take* $O(h^2 \log^2 h)$ *time.*

Proof. Since $F_{h(1)}$ is an $h \times h$ matrix, the trees for this window can be built in $O(h^2 \log^2 h)$ time, as stated by Theorem 9.1. ∎

Remark 12.1. Before we go on to describe Procedure `Advance`, one technical detail is worth mentioning. Fix a window $F_{h(p)}$. For the chunks labeling the edges of $\mathrm{L}F_{h(p)}$, we use an encoding analogous to the one described in Section 3 for $\mathrm{L}T_p$, except that now the encoding is expressed in terms of rows and columns within the window $F_{h(p)}$. Since the window slides over A and we delete nodes and edges from $\mathrm{L}F_{h(p)}$, some of those labels may become outdated, i.e., they are no longer valid. This is the same type of problem addressed by Ferragina, Grossi, and Montangero for

dynamic maintenance of suffix trees for strings [11]. They give a real-time algorithm for maintaining consistent the arc labels in a suffix tree under string insertion and deletion. In order to keep the labels of $LF_{h(p)}$ consistent, for $h \leqslant h(p) \leqslant n$, we use their technique. The same type of reasoning applies to the other two trees of $F_{h(p)}$. We will implicitly assume that the appropriate labels are maintained and we will not mention the costs of those operations in our time analysis.

12.1. *Procedure* Advance—*Outline*

Fix a window $F_{h(p)}$ and assume that the trees for that window are available. Procedure Advance must transform those trees into the ones for $F_{h(p+1)}$. This task is best described by dividing it into two subtasks. For that reason, we introduce an "intermediate" window and "intermediate" trees for that window.

Let $F'_{h(p)}$ be $F_{h(p)}$, with the addition of the subrow and subcolumn just outside of it. Let $F'_{h(p),d}$ be defined for $F'_{h(p)}$ as $F_{h(p),d}$ is defined for $F_{h(p)}$, except that $d \in D_{h(p+1)}$. Finally, $LF'_{h(p)}$ is the Lsuffix tree representing all suffixes of $F'_{h(p),d}\$$, $d \in D_{h(p+1)}$. $T'_{rows,h(p)}$ and $T'_{cols,h(p)}$ are defined as $T_{rows,h(p)}$ and $T_{cols,h(p)}$ but for $F'_{h(p)}$, respectively. We refer to those trees as the *trees* for $F'_{h(p)}$.

PROCEDURE Advance.

(Ad1) Transform the trees for $F_{h(p)}$ into the ones for $F'_{h(p)}$.

(Ad2) Transform the trees for $F'_{h(p)}$ into the ones for $F_{h(p+1)}$.

Let $|T_{h(p)}|$ be the sum of the sizes, i.e., nodes and edges, of the trees for window $F_{h(p)}$. Let $|T'_{h(p)}|$ be analogously defined for the intermediate window.

LEMMA 12.2. *Step* (Ad1) *can be performed in* $O((|T'_{h(p)}| - |T_{h(p)}| + p) \log^2 |F_{h(p)}|)$ *time.*

Proof. The transformation of the trees for $F_{h(p)}$ in the corresponding trees for $F'_{h(p)}$ can be carried out using the techniques described in Sections 3–8 for the transformation of LT_p in LT_{p+1} and the maintenance of the associated data structures. However, the time bound in Lemma 9.1 now becomes the one stated in the lemma. ∎

The next subsection gives details for the implementation of step (Ad2).

12.2. *Procedure* Advance—*Step* (Ad2)

Step (Ad2) consists of transforming the trees for $F'_{h(p)}$ into the ones for $F_{h(p+1)}$. We discuss only the transformation of $LF'_{h(p)}$ into $LF_{h(p+1)}$ since

the transformation of the other two trees can be obtained using essentially the same ideas.

• *Changes in the Structure of* $\mathrm{LF}'_{h(p)}$. Notice that $F_{h(p+1)}$ is obtained from $F'_{h(p)}$ by deleting all matrices $F'_{h(p),d}$ for $|d| \leqslant n-1$. Correspondingly, all those matrices must be deleted from $\mathrm{LF}'_{h(p)}$, possibly causing changes in $\mathrm{LF}'_{h(p)}$. Indeed, consider a matrix $F'_{h(p),d'}$ that must be removed. Since it is of side length $h+1$ and, with the exception of dummy matrices, no other matrix represented in $\mathrm{LF}'_{h(p)}$ is of longer side length, we must have that $F'_{h(p),d'}$ has locus at a leaf $f \in \mathrm{LF}'_{h(p)}$. By the same arguments, all other matrices in $CLASS[f]$ need to be removed. That fact may imply the removal of f from $\mathrm{LF}'_{h(p)}$. However, we need to establish when we can actually remove f (indeed, there may be a matrix Z, "ending" on the edge $(parent(f), f)$ that is also deleted, while we need to keep that matrix). The following two lemmas allow us to establish when we can actually remove f.

LEMMA 12.3. *Consider the edge* $(v, f) \in \mathrm{LF}'_{h(p)}$ *and assume there is no internal suffix link "ending" on that edge. Then, by removing the leaf* f, *we do not remove any suffix* M' *of any matrix* $F_{h(p+1),d}$, $d \in D_{h(p+1)}$.

Proof. The proof is by contradiction. Let M' be the matrix of longest side length, suffix of some $F_{h(p+1),d'}$, $d' \in D_{h(p+1)}$, such that this matrix is represented in $\mathrm{LF}_{h(p)}$ and, when f is removed, it is removed from that tree. Notice that M' has side length at most h. Therefore, f is its extended locus but it cannot be its locus. Moreover, since $F_{h(p+1),d'}$ is suffix of $F'_{h(p),d'}$, we have that M' is suffix of $F'_{h(p),d'}$. We have two cases.

Case $M' = F'_{h(p),d'}$. But then the dummy leaf associated to $F'_{h(p),d'}\$$ has a suffix link "pointing to M'" and there would be an internal suffix link "ending" on the edge (v, f) of $\mathrm{LF}'_{h(p)}$. A contradiction.

Case M' *Is a Proper Suffix of* $\mathrm{LF}'_{h(p),d'}$. Then, there exists a matrix M that is suffix of $\mathrm{LF}'_{h(p),d'}$ and that has M' as second suffix. M cannot have locus at a leaf of $\mathrm{LF}'_{h(p)}$ or there would be an internal suffix link "ending" on the edge (v, f) of $\mathrm{LF}'_{h(p)}$. So, M has extended locus in a node w. Let $\hat{M}$ be any matrix associated with a leaf in the subtree of $\mathrm{LF}'_{h(p)}$ rooted at w. $\hat{M}$ will be suffix of some $F'_{h(p),\hat{d}}$. Notice that M is proper prefix of $\hat{M}$. This fact implies that the second suffix Z of $\hat{M}$ has M' as proper prefix. Therefore, its extended locus must also be f. Notice that f cannot be its locus because the side length of Z is at most h. Since $F_{h(p+1),\hat{d}}$ is suffix of $\mathrm{LF}'_{h(p),\hat{d}}$, Z is a suffix of longer side length than M' satisfying the same assumptions, a contradiction. ∎

LEMMA 12.4. *Consider the edge* $(v, f) \in \mathrm{LF}'_{h(p)}$ *and assume there is an internal suffix link "ending" on that edge. Let* g *be the leaf of* $\mathrm{LF}'_{h(p)}$ *where the "deepest" such suffix link originates and let* Z *be the second suffix of*

$M(g)$. Z is suffix of some $F_{h(p+1),d'}$, $d' \in D_{h(p+1)}$. Moreover, there is no suffix of any matrix $F_{h(p+1),d}$, $d \in D_{h(p+1)}$, having side length longer than that of Z and that has f as extended locus.

Proof. When $g = f$, the lemma is obvious because Z is of side length h, which is also the maximum side length of any matrix $F_{p+1,d}$. For the case $f \neq g$, the proof is analogous to the one of Lemma = 12.3. ▌

• *Implementation of Step* (Ad2). The main point in the implementation of Step (Ad2) is how to process a leaf f such that $CLASS[f]$ contains only matrices that need to be removed. Using Lemmas 12.3 and 12.4, the following procedure performs that task.

PROCEDURE Delete(f).

(1) Let v be the parent of f in $\mathrm{L}F'_{h(p)}$. Assume that the conditions of Lemma 12.3 are satisfied. We delete f. Moreover, all dummy leaves with suffix links pointing to f have their suffix links reset to the node pointed to by the suffix link of f. If v has only one child f' left, v is eliminated and the edges $(parent(v), v), (v, f')$ are combined into $(parent(v), f')$.

(2) Assume that the conditions of Lemma 12.3 are not satisfied. Then, Lemma 12.4 implies that we can keep f as the locus of Z in $\mathrm{L}F_{h(p+1)}$. Let f' be the leaf it points to in $\mathrm{L}F'_{h(p)}$. The dummy leaves that have suffix links pointing to f in $\mathrm{L}F_{h(p)}$ must have a new suffix link pointing to f' in $\mathrm{L}F_{h(p+1)}$. The extended locus of the second suffix of Z is on the path from the root of $\mathrm{L}F'_{h(p)}$ to f'. (It can be found using Procedure Exlocus presented in Section 7.1). The new suffix link of f will point to that node.

LEMMA 12.5. *Step* (Ad2) *can be implemented to take* $O((|T_{h(p+1)}| - |T'_{h(p)}| + p) \log^2 |F_{h(p)}|)$ *time.*

Proof. All leaves that are candidates to be removed from $\mathrm{L}F'_{h(p)}$ can be identified in $O(p)$ time by means of the dummy leaves associated to the $F'_{h(p),d}$'s, $|d| \leqslant p-1$. Moreover, for each of those leaves we can check in constant time whether either the conditions of Lemma 12.3 or 12.4 are satisfied. Indeed, for each internal suffix link ending on an edge, we need to mark that edge. Details on how to maintain those marks are left to the reader.

Next we observe that, apart from logarithmic factors, each call to Delete(f) takes time proportional to the number of nodes deleted from $\mathrm{L}F'_{h(p)}$ during that call plus the number of matrices in $CLASS[f]$ (that must all be removed). One can easily show that analogous bounds hold for the transformation of $T'_{rows,h(p)}$ and $T'_{cols,h(p)}$ into $T_{rows,h(p+1)}$ and $T_{cols,h(p+1)}$. Therefore, the transformation of the trees for $F'_{h(p)}$ into the ones for $F_{h(p+1)}$ takes $O((|T'_{h(p)}| - |T_{h(p+1)}|) \log^2 |F'_{h(p)}|)$ time. Finally, we

obtain the bound stated in the lemma by using the fact that $|T'_{h(p)}| \leqslant |T_{h(p+1)}| + cp$, for some constant c. ▮

12.3. *Sliding the Window—Time Analysis*

THEOREM 12.1. *Procedure* `Slide-Window` *takes* $O(n^2 \log n^2)$ *time.*

Proof. By Lemma 12.1, Procedure `Initialize` takes $O(h^2 \log^2 h)$ time. For a window $F_{h(p)}$, we add the bound in Lemma 12.2 with the one in Lemma 12.5 to obtain that Procedure `Advance` takes $O((|T_{h(p+1)}| - |T_{h(p)}| + p) \log^2 |F_{h(p)}|)$ time. Summing those bounds over all window movements, we obtain the claimed time bound. ▮

13. DATA COMPRESSION—*Match*($F_{h(p)}$)

Assume that when the window is in row $h(p)$ of A, we have "seen" the matrix A up to row $h(p+h)$, i.e., $W_{h(p+h)}$ is available. Moreover, assume that $T_{rows,\, h(p+h)}$ and $T_{cols,\, h(p+h)}$ are available. We point out that, while the window slides over A, we will maintain those two trees rather than $T_{rows,\, h(p)}$ and $T_{cols,\, h(p)}$ without altering the time bound in Theorem 12.1. In order to simplify the description of $Match(F_{h(p)})$ we assume that $h(p+h) \leqslant n$. Now, fix a $d \in D_{h(p+1)}$ and consider the matrix B, suffix of height h of $W_{h(p+h),\, d}$. The longest prefix that B has in common with submatrices of $F_{h(p)}$ can be found using a standard visit of $LF_{h(p)}$. However, in order to compare Lcharacters from prefixes of B with Lcharacters from submatrices of $F_{h(p)}$, we use $T_{rows,\, h(p+h)}$, $T_{cols,\, h(p+h)}$ and the techniques of Section 8. The time bound to find the desired prefix (say, of height h') is $O(h' \log^2 n)$. Those observations justify the following:

THEOREM 13.1. *The query* $Match(F_{h(p)})$ *can be implemented to take* $O(S \log^2 n)$ *time, where* S *is the total area covered by the matrices in* $Match(F_{h(p)})$.

14. CONCLUDING REMARKS AND OPEN PROBLEMS

We have shown how to build the Lsuffix tree of a matrix on-line and we have applied our techniques to two-dimensional pattern matching and to the support of primitive operations for **LZ1-type** image compression methods. An interesting open problem is to reduce the time complexity of our on-line algorithm.

ACKNOWLEDGMENTS

The authors are deeply indebted to Jim Storer for suggesting some problems from data compression that led to this research and to Bruno Carpentieri, Shimon Even, and Roberto Grossi for helpful discussions. Moreover, our gratitude goes to the referees for their competent reviews and most valuable comments that greatly helped the presentation of our ideas.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, MA, 1974.
2. A. Amir and M. Farach, Two-dimensional dictionary matching, *Inform. Process. Lett.* **44** (1992), 233–239.
3. A. Amir, M. Farach, R. Idury, J. La Poutre, and A. Schaffer, Improved dynamic dictionary matching, *Inform. and Comput.* **119** (1995), 258–282.
4. A. Apostolico The myriad virtues of subword trees, *in* "Combinatorial Algorithms on words" (A. Apostolico, and Z. Galil, Eds.), NATO ASI Series F, Vol. 12, pp. 85–96, Springer-Verlag, Berlin, 1984.
5. A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber, and U. Vishkin, (1988), Parallel construction of a suffix tree with applications, *Algorithmica* **3** (1988), 347–365.
6. W. I. Chang and E. L. Lawler, Approximate string matching in sublinear expected time, *in* "Proc. 31st Symposium on Foundations of Computer Science," pp. 116–124, IEEE, New York 1990.
7. M. T. Chen and J. Seiferas, Efficient and elegant subword-tree construction, *in* "Combinatorial Algorithms on Words" (A. Apostolico and Z. Galil, Eds.), NATO ASI Series F, Vol. 12, pp. 97–107, Springer-Verlag, Berlin, 1984.
8. Y. Choi and T. W. Lam, Two-dimensional pattern matching on a dynamic library of texts, *in* "COCOON, 1995" (D. Z. Du and M. Li, Eds.), pp. 530–538, Lecture Notes in Comput. Sci., Springer-Verlag, New York/Berlin, 1995.
9. M. Farach, Optimal suffix tree construction with large alphabets, *in* "Proc. 38th Symposium on Foundations of Computer Science," pp. 134–146, IEEE, New York, 1997.
10. M. Farach and S. Mathukrishnan, Optimal logarithmic time randomized suffix tree construction, *in* "International Colloquium on Automata, Languages and Programming (ICALP '96)," Lecture Notes in Computer Science, pp. 550–561, Springer-Verlag, New York/Berlin, 1996.
11. P. Ferragina, R. Grossi, and M. Montangero, A note on updating suffix-tree labels, *in* "Italian Conference on Algorithms and Complexity (CIAC '97)," Lecture Notes in Computer Science, Vol. 1203, pp. 181–192, Springer-Verlag, New York/Berlin, 1997.
12. E. R. Fiala and D. H. Green, Data compression with finite windows, *Comm. Assoc. Comput. Mach.* **32** (1988), 490–505.
13. S. Fortune and J. Wyllie, Parallelism in random access machines, *in* "Proc. 10th Symposium on Theory of Computing," pp. 114–118, Assoc. Comput. Mach., New York, 1978.
14. R. Giancarlo (1993), An index data structure for matrices, with applications to fast two-dimensional pattern matching, *in* "Proc. of Workshop on Algorithms and Data Structures," pp. 337–348, Lecture Notes in Comput. Sci., Springer-Verlag, New York/Berlin, 1993.

15. R. Giancarlo, A generalization of the suffix tree to square matrices, with applications, *SIAM J. Comput.* **24** (1995), 520–562.
16. R. Giancarlo and R. Grossi, Parallel construction and query of suffix trees for two-dimensional matrices, *in* "Proc. of the 5th ACM Symposium on Parallel Algorithms and Architectures," pp. 86–97, 1993.
17. R. Giancarlo and R. Grossi, On the construction of classes of suffix trees for square matrices: Algorithms and applications, *Inform. and Comput.* **130** (1996), 151–182.
18. G. H. Gonnet, "Efficient Searching of Text and Pictures—Extended Abstract," Technical Report, University of Waterloo, OED-88-02, 1988.
19. R. Hariharan, Optimal parallel suffix tree construction, *in* "Proc. the 26th Symposium on Theory of Computing," pp. 290–299, Assoc. Comput. Mach., New York, 1994.
20. D. Hilbert, Über stetige Abbildung einer Linie auf ein Flachenstuck, *Math. Ann.* **38** (1891), 459–460.
21. I. Idury and A. Schaffer, Multiple matching of rectangular patterns, *in* "Proc. the 25th Symposium on Theory of Computing," pp. 81–90, Assoc. Comput. Mach., New York, 1993.
22. R. Jain, "Workshop Report on Visual Information Systems," Technical Report, National Science Foundations, 1992.
23. D. E. Knuth, "The Art of Computer Programming. Vol. 3. Sorting and Searching," Addison–Wesley, Reading, MA, 1973.
24. R. S. Kosaraju, Real-time pattern matching and quasi-real-time construction of suffix trees, *in* "Proc. the 26th Symposium on Theory of Computing," pp. 310–316, Assoc. Comput. Mach., New York, 1994.
25. N. Jesper Larsson, Extended application of suffix tree to data compression, *in* "Proc. Data Compression Conference," pp. 190–199, IEEE, New York, 1996.
26. A. Lempel and J. Ziv, On the complexity of finite sequences, *IEEE Trans. Inform. Theory* **22** (1976), 75–81.
27. A. Lempel and J. Ziv, Compression of two-dimensional data, *IEEE Trans. Inform. Theory* **32**, (1986), 2–8.
28. E. M. McCreight, A space economical suffix tree construction algorithm, *Assoc. Comput. Mach.* **23** (1976), 262–272.
29. G. Peano, Sur une courbe qui remplit toute une aire plaine, *Math. Ann.* **36** (1890), 157–160.
30. M. Rodeh, V. R. Pratt, and S. Even, Linear time algorithm for data compression via string matching, *J. Assoc. Comput. Match.* **28** (1981), Vol. 24.
31. A. Rosenfield and A. C. Kak, "Digital Picture Processing," Academic Press, San Diego, 1982.
32. S. C. Sahinalp and U. Vishkin, Symmetry breaking for suffix tree construction, *in* "Proc. the 26th Symposium on Theory of Computing," Assoc. Comput. Mach., pp. 300–309, New York, 1982.
33. D. Sheinwald, A. Lempel, and J. Ziv, Two-dimensional encodings by finite state encoders, *IEEE Trans. Comm.* **38** (1992), 341–347.
34. D. D. Sleator and R. E. Tarjan, A data structure for dynamic trees, *J. Comput. System Sci.* **26** (1983), 362–391.
35. A. O. Slisenko, Detection of periodicities and string matching in real time, *J. Soviet Math.* **22** (1983), 1316–1386.
36. J. Storer, "Data Compression: Methods and Theory," Comput. Sci. Press, Rockville, MD, 1988.

37. J. A. Storer, Lossles image compression using generalized LZ1-type methods, *in* "Proc. Data Compression Conference," pp. 290–299, IEEE, New York, 1996.
38. E. Ukkonen, On-line construction of suffix trees, *Algorithmica* **14** (1995), 249–260.
39. P. Wiener, Linear pattern matching algorithms, *in* "Proc. 14th Symposium on Switching and Automata Theory," pp. 1–11, IEEE, New York 1973.
40. J. Ziv and A. Lempel, A universal algorithm for sequential data compression, *IEEE Trans. Inform. Theory* **23** (1977), 337–343.