Suffix Trees and their Applications in String Algorithms*

Roberto Grossi†
Dipartimento di Sistemi e Informatica
Università di Firenze
50134 Firenze, Italy

Giuseppe F. Italiano‡
Dipartimento di Matematica Applicata ed Informatica
Università "Ca' Foscari" di Venezia
Venezia, Italy

Keywords: Pattern matching, String algorithms, Suffix tree.

Abstract: The suffix tree is a compacted trie that stores all suffixes of a given text string. This data structure has been intensively employed in pattern matching on strings and trees, with a wide range of applications, such as molecular biology, data processing, text editing, term rewriting, interpreter design, information retrieval, abstract data types and many others.

In this paper, we survey some applications of suffix trees and some algorithmic techniques for their construction. Special emphasis is given to the most recent developments in this area, such as parallel algorithms for suffix tree construction and generalizations of suffix trees to higher dimensions, which are important in multidimensional pattern matching.

*Work partially supported by the ESPRIT BRA ALCOM II under contract no. 7141 and by the Italian MURST Project "Algoritmi, Modelli di Calcolo e Strutture Informative".

†Part of this work was done while the author was visiting AT&T Bell Laboratories. Email: grossi@di.unipi.it

‡Work supported in part by the Commission of the European Communities under ESPRIT LTR Project no. 20244 (ALCOM–IT), by the Italian MURST Project "Efficienza di Algoritmi e Progetto di Strutture Informative", and by a Research Grant from University of Venice "Ca' Foscari". Part of this work was done while at University of Salerno. Email: `italiano@dsi.unive.it`. URL: `http://www.dsi.unive.it/~italiano`.

Contents

1 Introduction

The *suffix tree* is a powerful and versatile data structure that has applications in many string algorithms [77, 101]. It is basically a compacted trie storing the suffixes of a given string, so that all the possible substrings of the string are represented by some (unique) path descending from the root. The power of a suffix tree lies mainly in its ability to encode all the suffixes of the given string in linear space. This succinct encoding enables one to retrieve a large amount of information from the index: for instance, it can be used as a diagram of state transitions for an automaton that recognizes all the substrings of the given string.

The importance of the suffix tree is underlined by the fact that it has been rediscovered many times in the scientific literature, disguised under different names, and that it has been studied under numerous variations. Just to mention a few appearances of the suffix tree, we cite the compacted bi-tree [101], the prefix tree [24], the PAT tree [50], the position tree [3, 65, 75], the repetition finder [82], and the subword tree [8, 24]. The ability of the suffix tree to represent all the substrings in linear space has inspired several variations. The suffix array [76], cactus suffix array [63], dynamic suffix array [37], PAT array [50] and SB-tree [38] are examples of arrays or trees containing the suffixes of the given string in the lexicographic order obtained by visiting the leaves of the corresponding suffix tree. The directed acyclic word graph (DAWG) and minimal suffix and factor automata [16, 28, 30] are either labeled graphs or automata recognizing the substrings of the given string (or only its suffixes), whose nodes can be related to those of the suffix tree built on the reversed string. The complete inverted file [17] is a compacted DAWG that is augmented with extra information on the nodes, equivalently obtained from the suffix tree of the given string by merging its edge-isomorphic subtrees and deleting part of the resulting structure.

In the known literature, an implicit definition of the suffix tree can be already found in Morrison's Patricia trees [78]. However, Weiner was the first to introduce explicitly the suffix tree in [101] (the original name was compacted bi-tree). Following the pioneering work of Weiner, several linear time and space constructions have been given later by McCreight [77], Pratt [82], Slisenko [92], and Chen and Seiferas [24] (some of those algorithms have been reviewed in [24, 47, 74]). The constructions in [24, 101] have also the advantage of being on-line, under the assumption that the input string is read one character at a time from right to left. A left-to-right on-line (although not linear-time) construction has been described by Majster and Reiser [75] and Kempf, Bayer and Güntzer [65]. An on-line linear-time algorithm has been given by Ukkonen [99], and a real-time construction has been given by Slisenko [92] and Kosaraju [67]. Most of the above constructions work also for strings drawn from a large alphabet, at the price of a logarithmic slow-down in time complexity (in the size of the alphabet). The first parallel algorithm for building the suffix tree has been presented by Landau and Vishkin [70]. Apostolico et al. [9] have given the first efficient parallel construction that has optimal work for a large alphabet. Hariharan [57], Sahinalp and Vishkin [86], and Farach and Muthukrishnan [35] have devised parallel constructions whose work is optimal also for a small alphabet.

The statistical behavior of suffix trees has been studied under general and mild probabilistic frameworks by Apostolico and Szpankowski [12], Blumer *et al.* [18], Devroye et al. [32], Grassberger [51], Jacquet and Szpankowski [60], Shields [89], and

Szpankowski [94, 95]. One of the main properties of the suffix tree is that its asymptotic expected depth is logarithmic in the length of the given string, even though it may be linear in the worst case. O'Connor and Snider [81] have related a complexity measure for random strings in cryptology, called maximum order complexity, to the statistical properties of the suffix trees.

The notion of suffix tree has been extended to square matrices by Gonnet [48, 49], Giancarlo [43], and Giancarlo and Grossi [46]. This data structure can be efficiently deployed in pattern matching algorithms in higher dimensions, an area which is gaining growing interest due to its applications to low-level image processing [85], image compression [93], and visual databases in multimedia systems [62]. The problem of building a tree data structure representing all submatrices of a given matrix has been shown to be computationally harder than the problem of building a tree data structure representing only the square submatrices [44], and it has been considered in [45]. A somewhat relaxed definition of suffix tree for labeled trees, storing the node-to-root paths of the given tree as strings in a compacted trie, has been introduced by Kosaraju [66] and used also by Dubiner *et al.* [33], for tree pattern matching purposes. Another interesting generalization of the suffix tree to parameterized strings (or, p-strings) has been introduced by Baker [13] to find program fragments in a software system that are identical except for a systematic change of parameters.

Suffix trees find a wide variety of applications in many different areas related to string processing, such as: string matching [6, 29, 101]; approximate string matching [23, 42, 71, 72, 79, 98]; finding longest repeated substrings [101]; finding squares [10, 68] and repetitions in a string [10]; computing statistics for the non-overlapping occurrences [11]; finding the longest match between all ordered suffix-prefix pairs of a given set of strings [55]; finding the longest substring that appears in h out of k strings, for any $h \geq 2$ [58]; computing characteristic strings [59]; matching a string as an arbitrary path of an unrooted labeled tree [4]; performing efficient dictionary matching [6, 5, 7, 21, 43]; data compression schemes [39, 40, 73, 83, 84, 102, 103]; searching for the longest run of a given motif in molecular sequences [53, 54, 100]; metric distance on strings [34]; complexity measure on random strings for cryptology [81]; inverted indices [22]; analyzing genetic sequences [25, 23]; finding duplication in programming code [13]; generating names for programs in assembly tasks [14]; testing unique decipherability for a set of words [83]; detecting similarities of a polygon in pattern recognition [96]; and so forth.

This paper surveys some applications of suffix trees and some the algorithmic techniques used for the construction of this ubiquitous data structure. Special emphasis is given to the most recent developments in this area, such as parallel algorithms for suffix tree construction and generalizations of suffix trees to higher dimensions, which are important in multidimensional pattern matching. The remainder of this paper is organized as follows. In Section 2, we define the suffix tree data structure, describing some of its applications and generalizations in Section 3. Several algorithms for its sequential and parallel construction are described in Sections 4 and 5, respectively. Finally, the generalization of the suffix tree to a square matrix along with its sequential and parallel construction are given in Sections 7 and 8. Section 9 contains some concluding remarks.

2 The Suffix Tree

Let x be a string of n characters, drawn from an ordered alphabet Σ. We denote x as $x[1{:}n]$. Let \$ be a special character, matching no character in Σ. The suffix tree T of $x\$$ is a trie (digital search tree) containing all the suffixes of $x\$$. The character \$ is a right endmarker, and its goal is to separate (in T) suffix $x[i{:}n]$ from suffix $x[j{:}n]$, for $i > j$, whenever the former is a prefix of the latter. This results in the existence of a leaf in T for each suffix of $x\$$, since any two suffixes of $x\$$ will eventually go their separate ways in T. Consequently, each leaf of T can be labeled with a distinct integer j such that the path from the root to the leaf (labeled) j corresponds to the suffix $x[j{:}n]$. Furthermore, the path from the root of T to an internal node u corresponds to a substring of x.

The number of different substrings of x that are encoded in T can be quite large. Indeed, even strings using only two distinct characters can have as many as $\Omega(n^2)$ different substrings. One such example is given by $x = a^{n/2}b^{n/2}$ for $a, b \in \Sigma$, which has $(n/2+1)^2$ distinct substrings (including the empty substring). However, there are compact (and equivalent) representations of the suffix tree that have at most $2n$ nodes, such as the ones defined by Weiner [101], McCreight [77], Pratt [82] or Slisenko [92]. An obvious way to compact a suffix tree is to make it a *compacted* trie by omitting internal nodes of degree one (also called *unary* nodes). The size of the obtained representation is at most $2n+1$, since the leaves are at most $n+1$ (one for each suffix of $x\$$), and in a tree with no internal unary nodes, the number of internal nodes is bounded by the number of leaves. Note that the advantage of having $O(n)$ rather than $O(n^2)$ nodes is crucial in many applications: for instance if the string is a piece of a DNA sequence, it can contain $n \approx 10^5$ characters; without the use of a suffix tree, we would need as many as $n^2 \approx 10^{10}$ memory cells to represent all possible substrings!

More formally, the following constraints placed on T are able to limit its size to $O(n)$.

(T1) An arc of T may store any nonempty substring of $x\$$, which is represented as a pair of integers to indicate its starting position and its length inside $x\$$.

(T2) Each internal node of T must have at least two outgoing arcs.

(T3) Substrings represented by sibling arcs of T must begin with different characters.

For each suffix tree node, let *pathstring* be string obtained by concatenating the sequence of labels encountered along the path from the root to that node. Note that the above constraints guarantee that a suffix tree node can be named unambiguously by its pathstring. We say therefore that such a node is the *locus* of a string y whenever its pathstring is equal to y. An extension of a string y is any string of which y is a prefix. The *extended locus* of y is the locus of the smallest extension of y (inclusive) having locus defined. If y itself has locus defined, then its locus and extended locus coincide.

We now illustrate the definition of suffix tree with an example. The suffix tree T of the string $\omega = x\$ = bbabab\$$ is represented in Fig. 1. String ω has seven suffixes, namely *bbabab*\$, *babab*\$, *abab*\$, *bab*\$, *ab*\$, *b*\$, and \$, which are numbered from 1 to 7. Using the suffix tree, we can check whether a given string is a substring of ω. For instance, *aba* is a substring of ω since it has extended locus in T. Indeed, there exists a

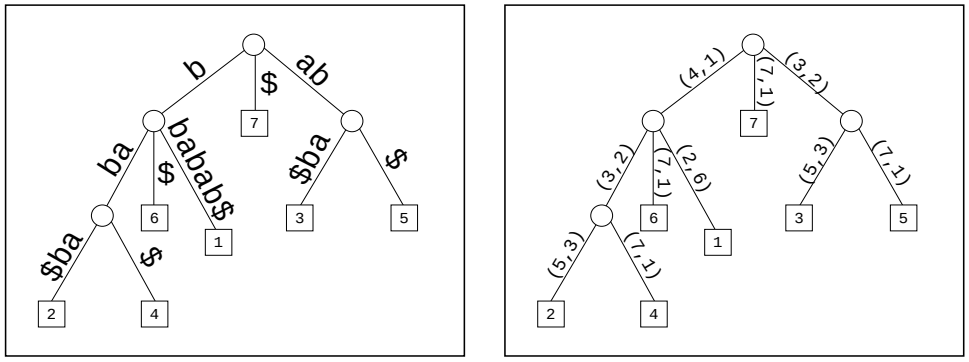

Figure 1. The suffix tree for the string $\omega = bbabab\$$, and its compact representation.

node in the suffix tree such that its pathstring starts with *aba*, namely leaf 3. Instead, the string *abaa* is not a substring of ω, since the last character a does not match along the pathstring indicated by *aba* (i.e., *abaa* has not extended locus in T). In Fig. 1 it is shown also a more compact representation of the suffix tree, where each string is replaced by a pair of integers: the number of starting position and the length of the substring [77].

It can be easily verified from Fig. 1 that the suffix tree obeys constraints (T1), (T2) and (T3) above. Furthermore, it is clear that each leaf is locus of exactly one suffix, which can be obtained as the pathstring associated with that leaf. A consequence of constraints (T1), (T2) and (T3) on the suffix tree T for $x\$$ is given by the three following properties:

Node Existence: There is a leaf for each suffix of $x\$$, along with an internal node for each (possibly empty) substring y of $x\$$ such that both ya and yb are substrings of $x\$$, where $a, b \in \Sigma \cup \{\$\}$ and $a \neq b$.

Completeness: A string $y \in \Sigma^*$ is substring of x if and only if y has extended locus in T, since a substring of x is a prefix of some suffix of x.

Common Prefix: If two suffixes of x share a prefix, say y, then they must share the path in T leading to the extended locus of y.

The above three properties capture the essence of a suffix tree and its algorithmic implications, as it will be discussed throughout the paper.

3 Some Applications and Generalizations of Suffix Trees

We discuss next few applications of the suffix tree. The suffix tree has been intensively employed in pattern matching problems on strings, matrices and trees. A typical pattern matching problem consists of locating all the occurrences of a given string, matrix or tree, called the *pattern*, as a substructure of another string, matrix or tree, called the *text*. Pattern matching problems have a wide range of applications, such as molecular biology, data processing, text editing, term rewriting, interpreter design, information retrieval, abstract data types and many others.

One classical example is *string matching*, which consists of finding all the occurrences of a pattern y as substring of a text x [20, 69] (see also the survey of Aho [1]). Crochemore *et al.* [29] have used the suffix tree built on the pattern y to speed up linear-time algorithms for string matching both in practice and on the average. The suffix tree defined on a dynamic set of strings, instead of a single string, has been used by Amir *et al.* [6] to obtain a dynamic version of the static Aho-Corasick dictionary automaton [2]. The Aho-Corasick automaton finds the multiple occurrences of a given set $\{y_1, \ldots, y_k\}$ of patterns simultaneously into a text x.

In many applications, the text (e.g., the Oxford English Dictionary or a DNA sequence) is fixed and static, with the above string matching query being repeated on-line for different patterns many times. Thus it is better to build the suffix tree T on $x\$$ as shown next [101]. Assume that y occurs at least once in x: the Completeness and Common Prefix properties guarantee that there is a one-to-one correspondence between all occurrences of y in x, and the leaves of T that are descending from the extended locus of y. In the example of Fig. 1, the occurrences of $y = ba$ in the text $x = bbabab$ are represented by the leaves numbered 2 and 4. Otherwise, if the pattern y does not occur in the text x there is no pathstring spelled by y in T: in Fig. 1, the pattern $y = abaa$ does not occur in the text x, since the last a does not match the pathstring spelled by aba. Therefore, in both cases, computing the suffix tree of $x\$$ enables one to find easily and efficiently all the occurrences of a pattern in the preprocessed text x. With a similar reasoning, it is possible to find the longest prefix of y occurring in x, in time proportional only to the length of such a prefix. Furthermore, associating the number of descending leaves with each node of T, the frequency or number of occurrences of y in x can be known without accessing all the leaves explicitly. These and other queries are typical of complete inverted files (e.g., see [17]). In several common situations, such as text editors and text retrieval systems [97, Sect.5.3], the preprocessed text x may undergo some changes. A recent line of research studies how to handle this dynamic case without building the suffix tree T from scratch each time [36, 37, 52].

In many cases, the pattern occurrences in the text may be approximate, that is, we allow a class of errors or transformations in the pattern (e.g., a word misspelling or a DNA mutation). In *approximate string matching*, character mismatches, insertions and deletions are considered in finding the pattern occurrences. Several researchers independently discovered the dynamic programming table solving the approximate string matching problem (see Sankoff and Kruskal's book [87] and Galil and Giancarlo's survey [41] for a list of references to the literature). To the best of our knowledge, it is an open problem to build a suffix-tree-like data structure that allows approximate queries to be performed on-line on a preprocessed text, requiring provably good worst case bounds [101]: so far, some elegant solutions with expected sublinear time queries have been proposed by Chang and Lawler [23] and Meyers [80], and a very nice approach that avoids, in many cases, the recomputation of equal portions of the dynamic programming table via suffix trees has been presented by Ukkonen [98]. Still, suffix trees turn out to be very useful to speed up the dynamic programming computation for solving approximate string matching (e.g., see [23, 42, 71, 72, 79, 98]). Among others, one basic technique, which is now common for many string algorithms, has been employed for the first time by Landau and Vishkin [71, 72]. It uses the suffix tree T to compute in constant time the *longest common prefix* of any two given suffixes of

x. Indeed, by the Common Prefix Property, the longest common prefix of two suffixes has locus in the least common ancestor (LCA) of the two corresponding leaves, which can be computed in constant time after a linear time preprocessing to answer LCA queries [56, 88]. Another technique, presented by Chang and Lawler [23], uses the suffix tree on the pattern y to compute *matching statistics* for a text x in linear time: for each position j of x, find the longest prefix of $x[j{:}n]$ occurring as a substring of y, and its corresponding extended locus in the suffix tree for y. An alternative solution for matching statistics is applying the external matching problem for file transmission [101] by building one suffix tree at time on the strings $y@w$, where @ is a separator and w is taken over $O(|x|/|y|)$ overlapping substrings of x of size $2|y|$; however, the technique in [23] is *on-line* because it can work also on an already built suffix tree for y. In [23], it is shown how to employ such a matching statistics to obtain expected sublinear time approximate queries.

Alternative forms of pattern matching, which should be more properly called pattern matching combinatorics or statistics require to detect substructures of the text that satisfy certain properties (such as repetitions, palindromes, etc... etc...). The problem of finding the palindromes of maximal length in a string x can be easily solved in linear time with suffix trees, for a constant sized alphabet. First build a suffix tree T on the string $\omega = x@x^R\$$, where @ is a distinct separator not occurring elsewhere and x^R is the reversed string of x. Preprocess T to answer LCA queries [56, 88] so to apply the aforementioned technique of [71, 72] for finding the longest common prefix of any two suffixes of ω. For each position j of x, we want to find the maximal palindrome having center in j, that is, the maximum k such that either $x[j{:}j+k-1] = (x[j-k{:}j-1])^R$ or $x[j+1{:}j+k] = (x[j-k{:}j-1])^R$. The former condition is for palindromes of even length; the latter for those of odd length. It suffices therefore to find k in constant time as the length of the longest common prefix between the two suffixes of ω starting either in positions j and $2|x|+3-j$ or in positions $j+1$ and $2|x|+3-j$, respectively.

Another example is finding the *longest repeated substrings* in x. A substring is repeated in x if it appears at least twice in x. With a naive approach, it would require quadratic time to find the longest repeated substrings. However, by the Node Existence property on the suffix tree T, a substring of x is repeated if and only if it has *extended* locus in an *internal* node of T. In particular, the strings having *locus* in the internal nodes of T are the candidates for being the longest repeated substrings, and the total number of those strings is upper bounded by the number of internal nodes of T, i.e. $|x|$. It suffices to take the longest ones with a simple visit of T in linear time. In the same fashion, in coding theory [89], suffix trees can be used to find the shortest prefix of each suffix of $x\$$ that does not occur elsewhere in x as follows. For each suffix $x[j,n]\$$, take the pathstring y having locus in the parent of leaf j, and append to y the $(j+|y|)$–th character of $x\$$, say a. Thus y is a prefix of $x[j,n]\$$ that occurs at least twice in x, but ya is a prefix of $x[j,n]\$$ that occurs only once. Problems of this kind are found also in data compression schemes [39, 73, 83, 84, 102, 103], in compressing assembly code [40], and in searching for the longest run of a given motif in molecular sequences [53, 54, 100].

Suffix trees help also to design elegant algorithms for finding squares [10, 68] and repetitions in a string [10]; computing statistics for the non-overlapping occurrences [11]; finding the longest match between all ordered suffix-prefix pairs of a given set of strings [55]; finding the longest substring that appears in h out of k strings, for any

$h \geq 2$ [58]; computing characteristic strings [59]; matching a string as an arbitrary path of an unrooted labeled tree [4]; performing efficient dictionary matching [6, 5, 7, 21, 43]. Other interesting applications are described in the excellent survey of Apostolico [8].

Beside pattern matching, suffix trees have been applied to many other problems, such as metric distance on strings [34], complexity measure on random strings for cryptology [81], inverted indices [22], analyzing genetic sequences [25, 23], finding duplication in programming code [13], generating names for programs in assembly tasks [14], and testing unique decipherability for a set of words [83].

Suffix trees can be used for detecting similarities of a polygon in pattern recognition (in [96], a polygon structure graph is used for this purpose). Given a polygon Q with m edges, let $x_1y_1x_2y_2\cdots x_my_m$ be the sequence of internal angles x_i and edge lengths y_i of Q, read in clockwise order, and let $string(Q) = x_1y_1\cdots x_my_mx_1y_1\cdots x_{m-1}y_{m-1}$. Then, two pieces of Q's contour are similar if they are equal as substrings in $string(Q)$. This observation leads to efficient algorithms for finding the rotational symmetries in Q, performing contour matching (called coastline matching in [96]) and similarity scaling (by deleting the lengths y_i's from $string(Q)$). With an analogous trick, we can partition a set of polygons into equivalence classes of similar polygons, or detect similarity between two polygons (see also [19, 90]).

We now mention the generalization of the suffix tree for parameterized pattern matching of Baker [13], which gives an important application in software maintenance. The problem consists of finding duplications of code in large software systems, by tracking down matches between different sections of code. We are looking not only for exact matches, but also for parameterized matches (in short p-matches). P-matches occur when sections of code may match except for the renaming of some parameters (e.g., identifiers and constants). It is important to detect this kind of duplications, as they are undesirable because of their possible association with bugs.

Exact matches can be found by using suffix trees in a plain fashion, as described before. To find parameterized matches requires to augment the notion of suffix tree in order to take into account the parameters, as follows. *Parameterized strings*, or *p-strings* in short, are strings that contain both ordinary characters drawn from the alphabet Σ, and parameter characters drawn from another finite alphabet Π. We assume that Σ and Π are disjoint, there is an ordering defined in both alphabets, and any two characters can be compared in constant time. Two p-strings yield a p-match if one can be transformed into the other by applying a one-to-one function that renames the parameter characters. For example, if $\Sigma = \{a, b\}$ and $\Pi = \{x, y, z\}$, then there is a p-match between $abxyabxyx$ and $abyzabyzy$ by renaming simultaneously x with y and y with z in the first p-string. To handle p-matches, Baker defines a parameterized suffix tree (or *p-suffix tree*), which generalizes the notion of suffix tree.

The crucial idea behind the p-suffix tree is to chain together occurrences of the same parameter so that matching parameters correspond to matching chains. These chains will then be encoded in the p-suffix tree. In particular, we chain together occurrences of the same parameter by associating non-negative integers to parameters, as follows. For each parameter, the leftmost occurrence is represented by a 0, and each successive occurrence is represented by the difference in position with the previous occurrence. An integer representing the difference in position is called a *parameter pointer*. For instance, we represent the chain of parameters of the p-string $abxyabxyx$ as $ab00ab442$.

We refer to this as $prev(abxyabxyx)$. Note that $prev(abxyabxyx) = prev(abyzabyzy)$. Indeed, it can be easily shown that any two p-string s and q yield a p-match if and only if $prev(s) = prev(q)$.

Let s be a p-string of length n, and let $s[i]$ be the i-th character of s: $s = s[1{:}n]$. The i-th *p-suffix* of s is $psuffix\ (s, i) = prev\ (s[i{:}n])$. Based upon this definition, a character of $prev(s)$ corresponds to a different value in $psuffix(s, i)$ if and only if it is a parameter pointer pointing to a position before i. Note that the following property holds:

P-matching: Let p and s be any two p-strings: p matches at position i of s if and only if $prev(p)$ is a prefix of $psuffix(s, i)$.

The value of the j-th character of $psuffix(s, i)$ can be easily computed by looking at j and the corresponding character $b = s[j + i - 1]$ of $prev(s)$. We define this in a function f: if b is a parameter pointer larger than $(j - 1)$, then $f(b, j) = 0$; otherwise $f(b, j) = b$. We are now ready for the definition of p-suffix tree.

The p-suffix tree of a p-string s is a compacted trie that stores all the p-suffixes of $s\$$. Analogously to the case of a suffix tree, each arc represents a non-empty substring of a p-suffix, each internal node has at least two outgoing arcs, and substrings represented by sibling arcs must begin with different characters. A consequence of the previous definitions is that the pathstring of each leaf gives a distinct p-suffix of s. Once again, if a p-string s has length n, its p-suffix tree has $O(n)$ size. By Property P-matching, the search of all the occurrences of a p-string p into a p-string s can be accomplished analogously to the case of suffix trees of strings. Indeed, we can follow the path spelled by successive characters of $prev(p)$ going downward from the root in the p-suffix tree of s. This search requires $O(|p|(\log(|\Sigma| + |\Pi|)))$ time, by using balanced search trees to organize the adjacency lists in the p-suffix tree. As for suffix trees, the p-matches can be calculated from the descending leaves. To find duplication in code, we can apply the substring statistics, discussed earlier for ordinary suffix trees, to the p-suffix tree.

In summary, testing whether a p-string p has a p-match with a substring of another p-string s can be done in time $O(|p| \log(|\Sigma| + |\Pi|))$ and space $O(|s|)$ using p-suffix trees. All the positions of s at which p has a p-match can be reported in time $O(|p| \log(|\Sigma| + |\Pi|) + k)$, where k is the total number of p-matches of p in s.

4 Sequential Construction of a Suffix Tree

In this section we describe efficient algorithms for the construction of the suffix tree of a string. In the following, we assume that $x[1{:}n]$ is a string over the alphabet Σ, and \$ is the endmarker. We will first describe the simplified version of Weiner's construction given by Chen and Seiferas [24], which scans the input string from right to left. Next, we present the algorithm by McCreight [77], which is based upon a left-to-right scanning. We also sketch the method of Amir *et al.* [6] for extending McCreight's construction to handle a dynamic set of strings.

4.1 The algorithm of Chen and Seiferas

Chen and Seiferas is basically a simplified version of Weiner's algorithm, and maintains the following auxiliary structures. For the sake of presentation, the nodes will be

identified with the substrings they represent. For each node z and for each character $a \in \Sigma$, we define three types of links. The first is a link to the node (if any) that is the locus of the shortest extension of substring za. This is called the $a-$*extension* link of z. The second is a link to the node (if any) that is the locus of the shortest extension of substring az. This is called the $a-$*shortcut* link of z. The third is a link to the prefix parent of z (if any). This is called the *prefix* link of z.

As mentioned before, we associate with each node a pair of positions that locates one occurrence of the corresponding substring into $x\$$. The main goal of the suffix tree is to construct prefix and extension links, while we build shortcut links only for efficiency issues during the construction. Note that prefix links are actually just the reverses of extension links, so we need to specify only how to build one of them.

The algorithm works from right to left. The suffix tree of $x\$ = \$$ trivially consists of two nodes and can be obviously built in constant time. To build the suffix tree of $az\$$, we assume inductively that we built already the suffix tree of $z\$$, and that we have pointers to the root and to the leaf $z\$$ of this tree. Now we show how to build the suffix tree of $az\$$ starting from the suffix tree of $z\$$. Note that the new substrings to be represented are prefixes of $az\$$. Let y be the longest prefix of $az\$$ that is already a substring of $z\$$: to build the new tree we have to install $az\$$ as a new extension starting from y.

Hence, the first problem is to locate y in the suffix tree of $z\$$. Note that y might not necessarily be a node in the suffix tree of $z\$$, in which case we will have to install it. We find y starting from the root as follows. If the root does not have an a-extension, then y is the root itself since in this case $y = \epsilon$. If the root has an a-extension, we could find y by tracing $az\$$ along extension links down from the root until $az\$$ departs from the substrings in the tree. For strings like $a^n\$$, however, this would accumulate overall $O(n^2)$ time to install all suffixes. Instead, we do the following.

Let v be the second suffix of y (i.e., $y = av$): we claim that v must be a node in the suffix tree of $z\$$. To prove the claim, we use some properties that follow from the definition of y. First, y does not contain the endmarker $\$$ since y is at the same time a prefix of $az\$$ and a substring of $z\$$. Let b be the character following y in $az\$$: note that b is always defined. Since $yb = avb$ is a prefix of $az\$$, it follows that vb is a substring of $z\$$. Second, yb cannot be a substring of $z\$$ (otherwise y would not be the longest prefix of $az\$$ that is a substring of $z\$$). Since y is a substring of $z\$$ and y does not contain $\$$, there must be a character c (possibly $c = \$$), such that $c \neq b$ and yc is a substring of $z\$$. Thus, there exists $c \neq b$ such that vc is a substring of $z\$$. In summary, there exist b and c with $b \neq c$, such that vb and vc are both substrings of $z\$$: hence, v must be a node in the suffix tree of $z\$$ (Node Existence property).

We trace up along prefix links from $z\$$, looking for node v: note that it can be easily recognized, since it will be the first node with an a-shortcut. We follow this shortcut, which leads us either to $y = av$ if this is already a node, or to its shortest extension avw that is a node. In the latter case, we have to install y as a new node between avw and its prefix parent, and set the shortcut links departing from y equal to the ones from avw. The a-shortcut links arriving to the new node y will be directed from the nodes on the prefix path from v up to the last node v' not already having an a-shortcut link to the prefix parent of node av.

When y has been found or installed, we install the new node $az\$$ as an extension of y. Shortcut links to $az\$$ will be directed from the nodes on the prefix path from $z\$$ up

to the last node not already having an a-shortcut link (note that the prefix parent of such a node is node v). Notice that both these and the shortcut links directed to y (in case y was installed) require a traversal of the path from $z\$$ up to v'. This shows that the time required to build the suffix tree of $az\$$ from the suffix tree of $z\$$ is proportional to the number of nodes along the path from $z\$$ to v'.

We use this observation to compute the total time required to build the suffix tree of the string $x\$$. To bound this time, we use an amortization argument and show that the length of the path from $z\$$ to v' can be amortized against the reduction in depth from $z\$$ to $az\$$. That is, the more we go up on the prefix path from $z\$$, the less we have to go up from $az\$$ in the next step. Let $\pi_{z\$}$ be the path from the root to $z\$$, and let π_y be the path from the root to $y = av$. Note that if av'' is a node in π_y, there must be a node v'' in $\pi_{z\$}$ above v' (recall that v' is along the path to v). Indeed, the fact that av'' is a node means that $av''b$ and $av''c$ occur in $az\$$ for two distinct characters b and c. This implies that also $v''b$ and $v''c$ occur in $z\$$. That is, there is a node for v''. Therefore the path π_y from the root to y is no larger than the path $\pi_{z\$}$ from the root to $z\$$. Since $az\$$ is installed as an extension below y, the reduction in depth from $z\$$ to $az\$$ is enough to compensate for the time spent in the path between $z\$$ and v', except for some small additive constant. Since the greatest possible increase in depth of the tree is constant for each iteration, the total depth reduction, and therefore the total running time of the algorithm, must be linear.

4.2 The algorithm of McCreight

We now give a high-level description of the suffix tree construction by McCreight [77]. For the sake of presentation, the nodes will be identified with the substrings they represent (the root represents the empty string). There are still three types of links. Two of them, the extension and prefix links, are defined as before. The third link is called the *suffix* link, and is defined as follows: for each node az other than the root, with $a \in \Sigma$ and $z \in \Sigma^*$, the suffix link connects node az to node z. Note that such a link is basically the reverse of the a-shortcut link from z to az in Chen and Seiferas' algorithm. As for the root, we can safely assume that its suffix link points to the root itself. As before, the main goal of the suffix tree is build prefix and extension links, while suffix links are for efficiency issues.

The algorithm works from left to right, and it has $n + 1$ steps. In step i, for $1 \leq i \leq n + 1$, the i-th suffix $x[i{:}n]\$$ is installed in T_{i-1}, assuming that T_{i-1} is the compacted trie built on the first $i - 1$ suffixes of $x\$$. Initially, for $i = 1$, tree T_1 for $x[1{:}n]\$$ is trivially composed of two nodes and it can be computed in constant time. To produce T_i for $i > 1$ we must locate in T_{i-1} the extended locus of the largest prefix of $x[i{:}n]\$$. Such a prefix is called $head_i$: it is a node in T_i, but not necessarily in T_{i-1}. Leaf $x[i{:}n]\$$ is installed as a child of $head_i$ in T_i.

Once again, we could find $head_i$ starting from the root, but it would accumulate to $O(n^2)$ overall time to install all suffixes. Instead, the algorithm of McCreight cleverly computes the location of $head_i$ in T_{i-1} with the help of $head_{i-1}$ in T_{i-1} and the suffix links. Indeed, the relation between $head_{i-1}$ and $head_i$ is as follows:

Lemma 4.1 (McCreight [77]). *The second suffix of $head_{i-1}$ is a prefix of $head_i$.*

Let w be the second suffix of $head_{i-1}$ (if $head_{i-1}$ is the empty string then w also is the empty string). Notice that by definition of $head_{i-1}$ there exists a path in T_{i-1} corresponding to w (i.e., w has extended locus in T_{i-1}). We could therefore think of using the suffix link from $head_{i-1}$ to reach the node w, and from there to follow the path to $head_i$ (note that $w = head_i$ if and only if $|head_i| = |head_{i-1}| - 1$). Unfortunately, the locus for string w is not guaranteed to exist in T_{i-1} (while the extended locus for w does exist). So the suffix link for $head_{i-1}$ in T_{i-1} is not always defined. However, as we show next, the suffix links for all other nodes are always defined. Indeed, the suffix link for the root is always defined and it points to the root itself. If az is a node of T_{i-1}, other than the root and $head_{i-1}$, then there exist two substrings azb and azc such that they are prefixes of some of the first $i-2$ suffixes of $x\$$, for $a, b, c \in \Sigma$ with $b \neq c$, and $z \in \Sigma^*$. Thus zb and zc are also two distinct prefixes of the first $i-1$ suffixes of $x\$$, implying that z is a node in T_{i-1}. By using an inductive argument, it follows that the suffix link from az to z is always defined.

We now show how to locate the extended locus of w and $head_i$ in T_{i-1}. Then it is an easy task to create a node for $head_i$ (if it is not already in T_{i-1}) and its leaf $x[i{:}n]\$$, and to make the suffix link in $head_{i-1}$ point to its correct location. This way, the tree T_i is correctly produced. To locate $head_i$ three substeps are carried out:

M1 : If the suffix link for $head_{i-1}$ is defined, set w as the node that can be reached from $head_{i-1}$ through its suffix link, and go to Substep **M3** skipping Rescanning.

M2 : (*Rescanning*) This phase locates the extended locus of string w in T_{i-1}, installing a node w if it is not there. Let f be the parent of node $head_{i-1}$ in tree T_{i-1}. Let f' the node that can be reached from f through its suffix link. String w can be found descending from f' (including itself). It is reached by branching recursively in T_{i-1} in the following way. Let u be the current node, initially set to f', and a be the character such that ua is a prefix of $x[i{:}n]\$$. If there is a branch out of u with initial character equal to a, then follow it to reach its child u'. If the length of substring u' is less than $|head_{i-1}| - 1$ (by Lemma 4.1), then set $u := u'$ and apply recursively the branching. If the length of substring u' is equal to $|head_{i-1}| - 1$, set $w = u'$. Otherwise, the length of substring u' is greater than $|head_{i-1}| - 1$: install a node w between u and u'. In this case, the character following w in $x[i{:}n]\$$ is different from the one following w along its unique child u'. So after Substep **M3** node w will not be unary. In all cases, the suffix link of $head_{i-1}$ can be correctly set to w, which is surely a node now.

M3 : (*Scanning*) In this phase, we locate (and possibly install) $head_i$ starting from the node w that is reached in Substep **M2**. The major difference between Scanning and Rescanning is that in Rescanning the length of w is known beforehand because of Lemma 4.1, while in Scanning the length of $head_i$ is not known in advance. Let w' be the substring such that $x[i{:}n]\$ = w\,w'$. Starting from w, and examining one character of w' at the time from left to right, we spell w' going deeper and deeper in the tree. When we stop, we install $head_i$ if it is not already a node, and install the new suffix as a child of $head_i$.

It has been shown by McCreight that the overall number of suffix links traversed and nodes rescanned in Substep **M2**, and of characters scanned in Substep **M3** accumulates to $O(n)$. This time analysis holds if the alphabet Σ is of constant size, so that we can follow in $O(1)$ a link labeled with a certain character from a given node. If this is not the case, and the size $|\Sigma|$ of the alphabet is not a constant, the adjacency lists for nodes in T must be organized by means of balanced search trees. This adds an $O(\log \min(|\Sigma|, n))$ factor to the linear bound.

Theorem 4.1 (McCreight [77], Pratt [82]). *Let Σ be an ordered alphabet, and x a string of n characters over Σ. The suffix tree of $x\$$ can be computed in time $O(n \log \min(|\Sigma|, n))$.*

Since building a suffix tree of x implicitly gives an ordering of the characters in x, no suffix tree construction can be faster than sorting the characters in x.

4.3 Extension of McCreight's construction to a set of strings

An interesting modification of McCreight's construction to a dictionary of strings $x_1, \ldots, x_k$, which can be updated by adding or removing strings, is given as part of the algorithms for solving the dynamic dictionary matching [6] and the all pairs suffix-prefix problem [55]. Briefly, the construction of the suffix tree for the string $x_1\$_1 \cdots x_k\$_k$, where all \$'s are distinct, is simulated without introducing the $O(\log k)$ overhead due to the distinct \$'s. Let n be the sum of the lengths of the strings. The main ideas of Amir *et al.* [6] can be summarized as follows:

- The suffix tree for $x_1\$_1 \cdots x_k\$_k$ is isomorphic to the compacted trie for all suffixes of $x_1\$_1$, all suffixes of $x_2\$_2$, etc. Moreover, $\$_1, \ldots, \$_k$ are simulated with a single character \$ that does not match itself. That requires storing the number of suffixes having locus in the same leaf.

- To add a new string $x_{k+1}\$_{k+1}$ in $O(|x_{k+1}| \log \min(|\Sigma|, n))$ time, simply start from the root and go to Step M3 of McCreight's algorithm. Indeed, the second suffix w of the head of the last inserted suffix '$\$_k$' (Lemma 4.1) is the empty string.

- To remove $x_i\$_i$ in $O(|x_i| \log \min(|\Sigma|, n))$ time, notice that the two permuted strings $x_1\$_1 \cdots x_{i-1}\$_{i-1}x_i\$_i x_{i+1}\$_{i+1} \cdots x_k\$_k$ and $x_1\$_1 \cdots x_{i-1}\$_{i-1}x_{i+1}\$_{i+1} \cdots x_k\$_k x_i\$_i$ yield two isomorphic suffix trees. We remove the leaves corresponding to the suffixes of $x_i\$_i$ taken in decreasing length. We are guaranteed that, when removing a unary node v that is parent of one of those leaves, no suffix link is pointing to v.

A formal description and the proof of correctness can be found in [6].

5 Parallel Construction of a Suffix Tree

In this section we describe algorithms for the parallel construction of a suffix tree. After describing the general naming technique of Karp et al. [64], we start with the algorithm

of Apostolico et al. [9], and describe next the latest more efficient parallel constructions of Sahinalp and Vishkin [86] and Hariharan [57]. We remark that a fast Las Vegas randomized algorithm, having optimal work and logarithmic time, has been devised by Farach and Muthukrishnan [35]. We present the results in [9, 64, 57, 86] using the Arbitrary Concurrent Read Concurrent Write (CRCW) PRAM model (e.g., see [61]). In this model, any number of processors is allowed to read from and to write to the same memory location simultaneously: in case of a writing conflict, only one processor arbitrarily succeeds.

5.1 The Karp-Miller-Rosenberg naming technique

We present the doubling technique introduced by Karp *et al.* [64] for sequential algorithms, which has been generalized to parallel algorithms in [9, 31]. The goal is to label all the substrings, whose length is a power of two, belonging to a given string $x[1{:}n]$. Labels are integers between 1 and $n+1$, called *names*, so that equal substrings get the same name.

Let Σ be the alphabet, and assume that $\Sigma \subseteq \{1, 2, \ldots, n\}$. Indeed if this is not the case, the characters of x can be sorted, and consistently numbered from 1 to n, in $O(\log n)$ time with n processors [26]. In this way, each character of Σ can be stored in one memory cell, and any two characters can be compared in $O(1)$ time. Without loss of generality, we suppose also that $\log n$ is an integer.

Throughout the computation, the algorithm uses n processors $P_1, \ldots, P_n$. Initially, processor P_i is allocated to the ith character $x[i]$, for $1 \leq i \leq n$. We also use an $n \times (n+1)$ matrix BB, called *Bulletin Board*, which is common to all processors and is not initialized. The access to BB is ruled as follows. When different processors attempt to write in the same location of BB, only one arbitrarily succeeds: we call this the *winner*.

We append n characters \$ to x as endmarkers. For the sake of presentation, the new string obtained will be still denoted by $x[1{:}2n]$. Note that we do not have to add explicitly the last $n-1$ endmarkers, as we can assume implicitly that $x[i] = \$$ whenever $i > n$. The algorithm builds n arrays $NAME_i[g]$, for $1 \leq i \leq n$ and $0 \leq g \leq \log n$, whose entries contain integers between 1 and $n+1$ (i.e., the names). Entry $NAME_i[g]$ records the name of $x[i{:}i+2^g-1]$, for $0 \leq g \leq \log n$. The goal is to set $NAME_i[g] = NAME_j[g]$ if and only if $x[i{:}i+2^g-1] = x[j{:}j+2^g-1]$. Note that, if $i > n$, the algorithm can safely assume that $NAME_i[g] = n+1$ for any g (the name for any substring of \$'s is $n+1$). Thus, once the arrays $NAME$ have been built, it is possible to check in constant time the equality of two substrings whose lengths are powers of two. To compute the names, we use the following doubling technique [64]:

> For $g > 0$, substring $x[i{:}i+2^g-1]$ can be seen as the concatenation of a prefix $x[i{:}i+2^{g-1}-1]$ and a suffix $x[i+2^{g-1}{:}i+2^g-1]$ of length 2^{g-1} each.

The above observation implies that the value of $NAME_i[g]$ depends on the pair $(NAME_i[g-1], NAME_{i+2^{g-1}}[g-1])$ whenever $g > 0$. This suggests to compute $NAME_i[g]$ by recursively computing $NAME_i[g-1]$ and $NAME_{i+2^{g-1}}[g-1]$. To assign equal names to equal pairs (corresponding to equal substrings), we use the Bulletin Board BB. The algorithm consists of $\log n + 1$ steps. Step g, for $g = 0, 1, \ldots, \log n$, can be outlined as follows:

$g = 0$: For each i such that $1 \leq i \leq n$, processor P_i sets $NAME_i[0] = x[i]$ in parallel. (Recall that P_i is allocated to $x[i]$ which is an integer).

$g > 0$: Assume that $NAME_i[g-1]$ is correctly computed for $1 \leq i \leq n$, and let $\eta_1 = NAME_i[g-1]$ and $\eta_2 = NAME_{i+2^{g-1}}[g-1]$ be the names assigned to the substrings $x[i{:}i+2^{g-1}-1]$ and $x[i+2^{g-1}{:}i+2^g-1]$ respectively. Then, for $1 \leq i \leq n$ in parallel, processor P_i attempts to write its index i into cell $BB[\eta_1, \eta_2]$. The index of the winner becomes the name of the substring $x[i{:}i+2^g-1]$. Thus, P_i sets $NAME_i[g] = BB[\eta_1, \eta_2]$.

It is not difficult to see that the naming can be correctly performed by repeating the above steps for $g = 0, 1, \ldots, \log n$. This requires $O(\log n)$ time with n processors on a CRCW PRAM.

In the above computation, we point out the crucial use of BB to assign equal integers in $[1, n]$ to equal pairs in $O(1)$ time. This use can be extended to assign equal integers in $[1, n]$ to equal tuples of k integers in $[1, n]$, in $O(k)$ time with a linear number of processors. This is useful to generalize the Karp *et al.* technique to square matrices, of size $n \times n$, in the following inductive fashion. Consider a submatrix M of size $2^g \times 2^g$. If $g = 0$, then M contains a single character, which becomes its name. If $g > 0$, then assume that all names have been correctly given to the submatrices of size $2^{g-1} \times 2^{g-1}$. Partition M into four submatrices of size $2^{g-1} \times 2^{g-1}$, whose names, say $\eta_1, \ldots, \eta_4$, exist by inductive hypothesis. Assign equal integers to equal quadruples in $O(1)$ time and linear work, storing them in the entries of BB. This way, we obtain the name for M by accessing the entry of BB containing the integer for the quadruple $(\eta_1, \ldots, \eta_4)$.

As a final remark, it is observed in [9] that the $O(n^2)$ space of the Bulletin Board BB can be reduced to $O(n^{1+\epsilon})$, for any fixed $\epsilon > 0$, with an $O(1/\epsilon)$ slow-down factor in the time and work.

5.2 The algorithm of Apostolico *et al.*

The parallel algorithm of Apostolico *et al.* [9] for building the suffix tree of a string $x = x[1{:}n]$ works in two phases. We follow the description given by JáJá [61]. In the first phase, called *naming*, the algorithm applies the technique of Karp et al. [64] to x, as described in Section 5.1. In the second phase, called *refining*, the algorithm produces a sequence of $\log n + 1$ refinement trees $D^{(\log n)}, D^{(\log n - 1)}, \ldots, D^{(0)}$ that approximate more and more precisely the suffix tree of x.

The initial tree $D^{(\log n)}$ consists of a root and n children, one for each suffix of x. The root represents the empty string, and the n leaves are labeled with the n suffixes of x. That is, the leaves are numbered from 1 to n: leaf i is labeled with the suffix $x[i{:}2n]$ (there is also an implicit leaf that does not participate to the refinement phase because it corresponds to all suffixes $x[i{:}2n]$ for $i > n$).

The refinement phase consists of an initial step in which $D^{(\log n)}$ is built and then of $\log n$ refinement steps: at refinement step r, tree $D^{(r)}$ is transformed into tree $D^{(r-1)}$. Each tree $D^{(r)}$, for $0 \leq r \leq \log n$, is such that its arcs are directed from children to the parent, and its nodes are labeled with substrings of x (as opposed to the suffix tree, where the arcs are labeled). Recall that a substring $x' = x[i{:}j]$ labeling a node can be compactly represented as the pair $(i, j - i + 1)$; the first component gives the starting position of x' inside x, and the second component gives the length of x'.

The final tree $D^{(0)}$ will be the suffix tree for x, except that the direction of the arcs must be reversed, and the labels must be moved from the nodes to the arcs.

Given two substrings x_1 and x_2 of x, *a refiner* is an integer $\ell > 0$ such that the first ℓ characters of the two substrings are equal, that is, they have a prefix of length ℓ in common. Tree $D^{(r)}$, for $\log n \geq r \geq 0$, meets the following condition, which we call *condition*(r):

(i) $D^{(r)}$ has n leaves and no unary nodes (that is, nodes with only one child).

(ii) Each internal node is labeled with a substring of x.

(iii) No pair of substrings labeling sibling nodes admits refiner 2^r.

(iv) Each leaf i is labeled with a distinct suffix $x[i{:}2n]$, which is stored as the pathstring to leaf i, for $1 \leq i \leq n$.

We observe that *condition*$(\log n)$ is trivially met by the initial tree $D^{(\log n)}$. At the end of the refinement phase, *condition*(0) guarantees that $D^{(0)}$ fulfills the definition of suffix tree, provided that the arcs are reversed, and the labels moved from the nodes to the arcs.

The key operation is to transform $D^{(r)}$ into $D^{(r-1)}$: we call this a *refinement step*. Before outlining this step, we need some terminology. In $D^{(r)}$, the children of a given node are referred to as a *nest*. We say that two nodes of $D^{(r)}$ in the same nest are *equivalent* if and only if the strings labeling them have refiner 2^{r-1}, that is, they share a common prefix of length 2^{r-1}. The detection and elimination of refiners 2^{r-1} inside the nests is the heart of the refinement step from $D^{(r)}$ to $D^{(r-1)}$.

The refiners 2^{r-1} are found using the *split labels* of the nodes of $D^{(r)}$. A split label for a node v, labeled with a substring $x' = x[i{:}j]$, for $j \geq i + 2^{r-1} - 1$, is a pair (η_1, η_2), where now η_1 is an arbitrarily chosen processor index that is assigned to the nest containing v, and η_2 is the name of the prefix $x[i{:}i + 2^{r-1} - 1]$ of length 2^{r-1} in x'. Thus, two nodes are considered equivalent if and only if they have the same split label, that is, the two nodes belong to the same nest and their labels share a refiner 2^{r-1}. Equivalent nodes correspond to substrings containing a common prefix of length 2^{r-1}: the locus of the common prefix will be a new node introduced in $D^{(r-1)}$.

A refinement steps consists of the following three substeps:

A1 : Partition the nodes of each nest into equivalence classes with the same split label. We use again the Bulletin Board BB to assign the same name to equal pairs.

A2 : For each equivalence class $\mathcal{C}$ in the nest with $|\mathcal{C}| > 1$, a new node w is created. The label of w is the prefix of length 2^{r-1} that is common to the substrings represented by the labels of nodes in $\mathcal{C}$. The parent u of the nodes in the nest becomes the parent of the new node w, which in turn becomes the parent of the nodes in $\mathcal{C}$. The labels of the nodes in $\mathcal{C}$ are consistently updated.

A3 : This substep performs a final check to remove unary nodes. Let $\tilde{D}^{(r)}$ be the tree resulting after the execution of Substep A2. Since each node of $D^{(r)}$ has at least two children, the only unary nodes in $\tilde{D}^{(r)}$ are the ones that had their

nest partitioned in one equivalence class. For each $v \in D^{(r)}$ such that its nest resulted in one equivalence class at the end of Substep A1, remove v from $\tilde{D}^{(r)}$ and make its only (newly created) child v' in that tree child of the parent of v in $\tilde{D}^{(r)}$ (it is shown in [9] that such a removal does not apply simultaneously to v and its parent). Labels are updated accordingly.

It is possible to show that, if $D^{(r)}$ satisfies *condition*(r), the tree $D^{(r-1)}$ obtained by the synchronous application of Substeps A1, A2 and A3 satisfies *condition*$(r-1)$. We omit this proof, the details of the implementation and the processor allocation, referring the interested reader to reference [9].

Theorem 5.1 (Apostolico *et al.* [9]). *A suffix tree for a string x of length n can be built in $O(\log n)$ time with n processors (i.e., $O(n \log n)$ work) on an Arbitrary CRCW PRAM. The required space is $O(n^{1+\epsilon})$ for any fixed $\epsilon > 0$.*

5.3 The algorithm of Sahinalp and Vishkin

We give a brief description of the optimal parallel construction of the suffix tree for a string $x = x[1{:}n]$, when $|\Sigma| = O(1)$, devised by Sahinalp and Vishkin [86]. It takes doubly logarithmic time and linear work on a CRCW PRAM; the required space is superlinear, as the one for the algorithm of Apostolico *et al.* [9].

The main bottleneck in the algorithm of Apostolico *et al.* is the use of the Karp *et al.* naming technique, described in Section 5.1. Recall that such a technique requires to assign names to all the $n - 2^g + 1$ substrings of x having length 2^g, for each $g = 0, \ldots, \log n$. Sahinalp and Vishkin observe in [86] that the naming assignment accumulates to overall $O(n \log n)$ work, because it is difficult to make a consistent selection among the overlapping substrings of x. Indeed, all those substrings "look-alike," and thus it is difficult to choose the ones that will be actually needed in the suffix tree construction. Moreover, trying to apply the Karp *et al.* technique only to a subset of $O(n)$ substrings of x is difficult, in the sense that it produces a rigid recursive partition of substrings into smaller substrings that does not take into account of the similarities (i.e., equal substrings could be partitioned differently). To "break symmetries" in all the substrings of x, Sahinalp and Vishkin apply successfully the deterministic coin tossing technique [27] (see also [61]), producing a recursive partition of the substrings that is not rigid, but driven by the characters in x. This way, they obtain a basic algorithm that builds the suffix tree for x in $O(\log^2 n)$ time and $O(n \log \log n)$ work, when $|\Sigma| = poly(n)$. The work bound of the basic algorithm is then reduced to optimal $O(n)$ when $|\Sigma| = O(1)$.

We present below the basic algorithm, assuming that $\Sigma \subseteq \{1, 2, \ldots, n\}$. There are three main stages.

The first stage requires $O(\log^2 n)$ time and $O(n \log^* n)$ work. The symmetries are broken by attaching labels only to $O(n)$ substrings of x as follows. A sequence of $O(\log n)$ strings $S(0), S(1), \ldots$ is produced, where $S(0) = x\$$. The symbols in $S(i)$ are called *i-characters*. Each $S(i)$ has length at most $n/2^i$, and it is obtained by "shrinking" $S(i-1)$ (if $i > 0$) in $O(\log n')$ time and $O(n' \log^* n')$ work, where $n' = |S(i-1)| \leq n/2^{i-1}$. More precisely, $S(i-1)$ is partitioned into *at most* $n/2^i$ sets of contiguous $(i-1)$-characters, each set becoming an *i-character* in $S(i)$. The i-characters are

then identified by integers in $[1, n]$, called *i-labels*, such that equal i-characters receive the same i-label. An i-character naturally induces a substring of $S(0) = x\$$ called *i-substring*.

In [86], a list of rules is given to partition $S(i-1)$. Roughly speaking, the rules for determining the sets in the partition of $S(i-1)$ are applied in the following order. An $(i-1)$-character forms a singleton in the partition whenever the corresponding $(i-1)$-substring is longer than 2^{i+1}: we call it a *long* $(i-1)$-character. Next, consider the other $(i-1)$-characters. Each largest run of consecutive and equal $(i-1)$-characters (at least two) becomes a set in the partition, called *repeated* substring. Consider again the remaining $(i-1)$-characters. Note that each of them is preceded and followed by an $(i-1)$-character that is not equal to it. Thus each largest run of consecutive $(i-1)$-characters forms a set in the partition, called *changing* substring, after the removal of its first and last $(i-1)$-character (there are at least three $(i-1)$-characters). Notice that a changing substring contains no two consecutive equal $(i-1)$-characters. We need to partition the changing substrings of length at least two into sets of two or three $(i-1)$-characters. We use deterministic coin tossing [27]. First, the jth $(i-1)$-character in a changing substring is labeled with an integer tag_j, which is the position of the least significative bit that differs in the binary representations of the jth and $(j+1)$th $(i-1)$-characters in that changing substring, respectively. To break symmetry, we apply the deterministic coin tossing technique on the sequence $tag_1, tag_2, \ldots$ for each changing substring in parallel, by dealing properly with the local maxima and minima (see procedure CONTENT-BASED in [86]). The changing substrings are partitioned into smaller sets of size at least two: those having length greater than three are still changing substring if we replace the jth $(i-1)$-character with tag_j. Thus we apply recursively the procedure and terminate in $O(\log^* n)$ steps.

A final adjustment must be done on some of the singleton sets in the resulting partition of $S(i-1)$, in order to guarantee $|S(i)| \leq n/2^i$. If the singleton is the initial set in $S(i-1)$, then it is merged with its right neighbor set in the partition; otherwise, it is merged with its left neighbor set whenever such a neighbor is a repeated substring of exactly two equal $(i-1)$-characters.

After that, the size of the above sets in $S(i-1)$ is computed in $O(\log n)$ time and linear work. Then, those sets are transformed into the i-characters of $S(i)$, thus shrinking $S(i-1)$, by assigning them the i-labels with the Bulletin Board BB (see Section 5.1 and [86, Sect.2.2.3]). That can be done in $O(1)$ time, observing that the sets are composed of either $O(1)$ distinct elements, or l equal elements, for some l. In both cases, we can represent them with tuples of constant size. The partition and the labeling satisfy the following two important properties.

Partition Consistency: Let X, Y be any two substrings of $S(i-1)$ such that $X = Y$. Then X and Y are partitioned in the same way, except possibly for at most $\log^* n + 1$ $(i-1)$-characters at the beginning and at the end of both X, Y.

Label Consistency: If two sets in the partition of $S(i-1)$ (i.e., two i-characters in $S(i)$) are equal, then they receive the same i-label.

The second stage of the basic algorithm requires $O(\log n \log^* n)$ time and $O(n \log^* n)$ work. Another sequence of $O(\log n)$ strings $C(0), C(1), \ldots$ is produced. We have

$C(0) = S(0) = x\$$ and $C(i)$ is obtained recursively from $S(i)$, where $|C(i)| = |S(i)|$. The characters of $C(i)$ are called *i-cores* and they are introduced for the following motivation. By the Partition-Consistency property, if $X = Y$ then the margins of both X and Y can be partitioned differently, and thus at a certain point of the suffix tree construction we could not detect similarities because of the different partition on the margins of X and Y. The idea behind i-cores is to add properly a *left* and *right extension* to each i-character so that $C(i)$ is obtained as a "relaxed" partition of $C(i-1)$, in which the sets may be *overlapping*. In this way, the margins of X and Y, of at most $\log^* n + 1$ $(i-1)$-characters each, will be always covered by some i-core during the suffix tree construction.

Formally, each $C(i)$ is defined recursively in terms of $C(j)$, $0 \leq j \leq i-1$, and $S(j)$, $0 \leq j \leq i$ (recall that $C(0) = S(0)$). For the jth i-character s of $S(i)$, the the jth i-core c of $C(i)$ that s *spans* is the substring of x obtained by taking the i-substring corresponding to s and extending it to the left and to the right with the rules below. We need before some definitions. Let $s_a, s_{a+1}, \ldots, s_b$ be the $(i-1)$-characters that form the set of $S(i-1)$ originating the i-character s of $S(i)$, in the first stage. The *left vicinity* of s is defined by the $\log^* n + 3$ $(i-1)$-characters $s_{a-\log^* n-3}, \ldots, s_{a-1}$ and, symmetrically, the *right vicinity* is given by $s_{b+1}, \ldots, s_{b+\log^* n+3}$. By induction, we know the $(i-1)$-cores, called c_k, spanned by the $(i-1)$-characters s_k, for each k.

The following rules are applied in order to define c in $C(i)$. If s is originated from a singleton containing only s_a, and s is a long i-character, then $c = c_a$. Otherwise, c is the concatenation (with overlaps) of a string x_{left} with the $(i-1)$-cores $c_a, c_{a+1}, \ldots, c_b$, and the resulting string is concatenated (with overlaps) with another string x_{right}, where x_{left} and x_{right} have $O(2^i \log^* n)$ length [86] and are determined as follows. If no $(i-1)$-character in the left vicinity $s_{a-\log^* n-3}, \ldots, s_{a-1}$ of s is long, then x_{left} is the concatenation (with overlaps) of the $(i-1)$-cores $c_{a-\log^* n-3}, \ldots, c_{a-1}$. Else, let s_{le} be the rightmost long $(i-1)$-character in the left vicinity of s. Note that s_{le} must have been originated, in the first stage, by at least 2^{i-l+1} l-characters of $S(l)$ for some $l < i$ (i.e., it was always a singleton for $l+1, \ldots, i-1$). We take x_{left} as the concatenation (with overlaps) of the l-cores corresponding to the rightmost 2^{k-l+1} of those l-characters, and of the $(i-1)$-substrings induced by $s_{le+1}, \ldots, s_{a-1}$. A symmetrical definition holds for x_{right} and the right vicinity of s.

As observed in [86], an i-core can be visualized as a "double staircase" given by the recursion on its left and right vicinity. Each step of the staircase contains consecutive k-substrings, for some $k \leq i$. Cores satisfy the following important properties, which allow us to assign efficiently *names* to cores with the Bulletin Board BB.

Core Consistency: Let X, Y be any two substrings of x such that $X = Y$. If X is an i-core for some i, then also Y is an i-core. Moreover, they receive equal names.

Middle Core Identity: Suppose that the i-character s is originated from the $(i-1)$-characters $s_a, \ldots, s_b$ such that $b - a \geq 16(\log^* n + 3) + 2$. Then the $(i-1)$-cores $c_a, \ldots, c_b$ have *equal* names except (at most) the first and the last $8(\log^* n + 3) + 1$ ones.

Note that Core Consistency property is stronger than Partition Consistency. The above two properties and other results in [86] give a strong characterization of each

core in terms of $O(\log^* n)$ different names, which can be recursively computed. By induction, assigning names to the cores in $C(i)$ requires $O(\log^* n)$ time and $O(\frac{n}{2^i} \log^* n)$ work with the Bulletin Board BB.

The third stage of the basic algorithm requires $O(\log^2 n)$ time and $O(n \log\log n)$ work. The suffix tree for x is built by reversing the iterations done in the second stage. Namely, let $T(i)$ be the suffix tree build on the string $C(i)$. We compute $T(i)$ assuming to have correctly computed $T(i+1)$ (the initial tree is easy to build because it has constant size). When we reach $T(0)$, we obtain the suffix tree for $x\$ = C(0)$.

The transformation of $T(i+1)$ into $T(i)$ is performed by improving the resolution of $T(i+1)$. We replace each $(i+1)$-core c of $C(i+1)$ with the more dense set of i-cores of $C(i)$ that are contained in c as substrings. We say in that case that c *covers* those i-cores. In [86], it is shown that the i-cores covered by any $(i+1)$-core can be represented as a string of $O(\log^* n)$ pairs, where each pair contains the name of one i-core and an integer denoting the number of its repetitions. More precisely, tree $T(i)$ is obtained from $T(i+1)$ as two intermediate versions, $T(i)_0$ and $T(i)_1$.

Tree $T(i)_0$ is obtained first by replacing only the $(i+1)$-cores that label the arcs departing from $T(i+1)$ with the strings of i-cores that they cover. Next, the remaining cores are replaced similarly, avoiding the duplication caused by the overlaps of the i-cores produced by any two $(i+1)$-cores labeling two consecutive arcs in a path of $T(i+1)$ (recall that the cores give a partition with overlaps).

Note that, since a suffix of $C(i)$ is implicitly mapped into a suffix of x, the suffix of x corresponding to a suffix of $C(i+1)$ may be different from the suffix of x corresponding to the suffix of $C(i)$ obtained by the above replacement. That is due to the fact that the left extension of an $(i+1)$-character may contain long i-characters (cf. the second stage). Thus the suffixes of x stored implicitly in $T(i)_0$ may be different from those in $T(i+1)$.

Now, two sibling arcs in $T(i)_0$ may share a common prefix of i-cores, but such a prefix cannot be longer than the former $(i+1)$-core, by the Core Consistency property and the fact that sibling arcs in $T(i+i)$ start with different $(i+1)$-characters. We refine $T(i)_0$ by using the fact that any $(i+1)$-core can be represented as a string of $O(\log^* n)$ pairs (it is done in [86] with the Bulletin Board BB and the parallel integer sorting [15]). Then we preprocess $T(i)_0$ to answer LCA queries, and obtain a preorder visit of its leaves with the Euler tour [61].

To produce $T(i)_1$, we define the *tail* of an i-core c in $C(i)$ as the sequence of names of the i-cores that are found starting from c (inclusive) to the rightmost i-core that is covered by the first full $(i+1)$-core to the right of c. Informally, tails represent a sort of "synchronization" between the i-cores in $C(i)$ and the suffixes stored into $T(i)_0$. Again, in [86], it is shown that a tail can be represented as a string of $O(\log^* n)$ pairs. We divide the i-cores into equivalence classes of equal tails. For each equivalence class, the root of $T(i)_1$ is linked to a distinct child, labeling such an arc with the tail representing that class. To build the compacted tries descending from those children, we observe that they can be obtained as a contraction of $T(i)_0$, by using LCA queries and the order of the leaves of $T(i)_0$ in its Euler tour. Finally, we need to refine the tails labeling the arcs departing from the root of $T(i)_1$, applying a method similar to the one described for $T(i)_0$. That completes the description of the basic algorithm that builds the suffix tree for x in $O(\log^2 n)$ time and $O(n \log\log n)$ work, when $|\Sigma| = poly(n)$. Note that

the $O(n \log\log n)$ work is dominated by the parallel integer sorting [15].

The work of the basic algorithm is then reduced to optimal $O(n)$ when $|\Sigma| = O(1)$, by applying the accelerating cascades method (e.g., see [61]) and other ad hoc techniques. We refer the interested reader to reference [86].

Theorem 5.2 (Sahinalp and Vishkin [86]). *The suffix tree for x can be built in $O(\log^2 n)$ time with $O(n \log\log n)$ work on a CRCW PRAM when $|\Sigma| = O(poly(n))$. The work can be reduced to $O(n)$ if $|\Sigma| = O(1)$. The required space is $O(n^{1+\epsilon})$ for any fixed $\epsilon > 0$.*

5.4 The algorithm of Hariharan

We describe how to build the suffix tree for a *binary* string $x = x[1{:}n] \in \{0,1\}^*$ in poly-logarithmic time and linear work (and space) on a CRCW PRAM, using the algorithm by Hariharan [57]. His construction works also on the weaker CREW PRAM within the same bounds, and it can be extended to the case in which $x \in \Sigma^*$, for an arbitrary alphabet Σ, after that the characters in x have been sorted by alphabet, coding each character in Σ as a binary sequence of $O(\log|\Sigma|)$ length. Compared to the algorithm of Apostolico et al. [9], the algorithm of Hariharan exploits useful combinatorial properties of strings (such as periodicity [69]) along with the repetitive nature of the suffix tree. The main ideas are outlined here, referring the interested reader to [57] for a deeper insight.

In the following, let $p = x\$$ and assume that $n \geq 8$. Let the *r-suffix tree* r-ST of x, with $r \geq 1$, be the compacted trie built on all substrings of p of length r and all suffixes of p of length less than r, excluding the suffix '\$'. When $r > n$, the r-ST is the suffix tree of x. Our goal is the construction of the sequence of trees r-ST, $2r$-ST, 2^2r-ST, ..., $2^s r$-ST, where $r = 2\lceil \log^3 n \rceil$ and $s = O(\log n)$ is the smallest integer such that $2^s r \geq n+1$. We will use T_i to denote $2^i r$-ST, for $0 \leq i \leq s$. The algorithm consists of s *stages*, the i-th one building T_i.

In the initial stage, $i = 0$, tree T_0 is constructed by using a concurrent version of McCreight's algorithm in $O(r \log n) = O(\log^4 n)$ time and $O(n)$ work. The stage has two steps. In the first step, an approximate version, T_0', of T_0 is built in $O(r \log n)$ time and $O(n)$ work. Tree T_0' is as T_0 except that its leaves store *all* the substrings of length smaller than or equal to $2r$. In the second step, T_0' is further processed to obtain T_0 in $O(\log n)$ time and $O(n)$ work. All leaves f are removed such that either $|f| > r+1$, or $|f| < r+1$ and f is not a suffix of p. The possibly created unary nodes are also removed.

The challenging task is to perform the first step in $O(r \log n)$ time and $O(n)$ work. String x is partitioned into $k = O(n/r)$ substrings $x_1, \ldots, x_k$ of length $2r$ (possibly, $|x_k| \leq 2r$). Any two adjacent strings x_h and x_{h+1} overlap in exactly r characters, for $1 \leq h < k$. Tree T_0' is built as the compacted trie of all the suffixes of each string $p_h = x_h\$_h$, where $1 \leq h \leq k$ and $\$_h \neq \$_{h'}$. To build T_0', each processor is allocated to a string p_h, having the task of inserting the suffixes of p_h, except '$\$_h$', from left to right, with the sequential algorithm of McCreight. Unfortunately, there are two main problems arising from the concurrent execution of McCreight's algorithm on a common data structure.

1. Several processors might simultaneously try to break an arc for installing an arbitrarily long path of nodes (cf. Steps M2 and M3 in Section 4.2).

2. After a node has been installed, the suffix link of its parent could not have been set. This link is required to be in place by the analysis of McCreight's algorithm.

The above two problems are solved, assuming first that the algorithm is divided into *time-steps*. A time-step corresponds to one step of McCreight's algorithm (i.e., either a scanning step M3 or a rescanning step M2). Globally, there are $O(r)$ time steps, each one requiring $O(1)$ time and $O(k) = O(n/r)$ work.

Now, the second problem above is circumvented by simply *waiting* at the node in which the suffix link must be followed, whenever it is not still set. Waiting time is measured in terms of *wasted* time-steps until some other processor sets such a link. Although this might cause a large delay in the overall number of time-steps necessary to insert the suffixes of some p_h, a clever amortized analysis in [57] shows that the processor assigned to string p_h, for $1 \leq h \leq k$, takes at most $14r$ time-steps.

Next, the first problem above is solved by organizing the processors into lists as follows. We define the *current string length* of the processor assigned to a string p_h as the integer $i - j$, where i is the position of the rightmost scanned character in p_h and j is the position of the last installed suffix of p_h. For each node u, there are two lists of processors, maintained ordered according to the processors' current string length. The first list, l_1, contains the processors at node u such that each of them may perform either a rescanning or a scanning step. The second list, l_2, contains the processors waiting at node u for its suffix link to be set. If such a link is already set then l_2 is empty. Note that the order on l_1 implies that, if the processors in l_1 want simultaneously to break an arc and transform it into a path of nodes, they are already sorted by the point at which they want to break this arc. After each time-step, the lists for all nodes are reorganized in $O(\log n)$ time and $O(n/r)$ work.

The stages $i = 1, 2, \ldots, s$ following the initial stage, where $s = O(\log n)$, are in charge of producing T_i. They could be easily performed in $O(\log^2 n)$ time and $O(n \log^2 n)$ work as follows.

Let $leaf_i(j) = f$ be the leaf $f \in T_i$ that stores the string $p[j{:}j + 2^i r - 1]$ (such a string will be denoted by f itself). For any leaf $f \in T_i$, we indicate with $positions_i(f) = \{j : f = leaf_i(j)\}$ the set of all positions j of p in which f occurs as a substring. Suppose to have correctly produced T_{i-1}, and preprocessed it in $O(\log n)$ time and $O(n)$ work to answer LCA queries in constant time [88]. This way, we can **(1)** compute the longest common prefix between any two substrings of p, of length $2^i r$ (see Section 3), and **(2)** compare lexicographically those two substrings, both operations taking constant work. It suffices to show how to obtain T_i from T_{i-1} in $O(\log n)$ time and $O(n \log n)$ work.

Using comparisons as specified in (2), we sort the substrings of p having length $2^i r$ in $O(\log n)$ time and $O(n \log n)$ work. A bucket $B(f)$ containing the substrings of p having length $2^i r$ and whose starting positions are given by $positions_{i-1}(f)$ is associated with each leaf $f \in T_{i-1}$ not being a suffix of p (note that the substrings in $B(f)$ share the same prefix f of length $2^{i-1} r$). After sorting, the equal substrings in each bucket form an an equivalence class, and a representative string for each class is chosen. Fix $L_i(f)$ to be the ordered list of these representative strings in $B(f)$. Recall

that the longest common prefix between any two consecutive strings in $L_i(f)$ can be computed in constant work, as mentioned in (1). We can use now the result of Farach and Muthukrishnan [35], allowing a compacted trie to be built for a lexicographically ordered list of strings in logarithmic time and linear work, once we know the longest common prefix of any two contiguous strings in the list. Therefore we can construct a compacted trie on $L_i(f)$ and then merge the root of such a compacted trie with f (because of this, f might no longer be a node), thus obtaining T_i.

There are two major points in reducing the overall work from $O(n \log^2 n)$ to $O(n)$ in the above computation of $T_1, \ldots, T_s$, while retaining $O(n)$ space and increasing the time bound from $O(\log^2 n)$ to $O(\log^4 n)$. Fix stage i. The first point is to obtain $L_i(f)$ for each leaf $f \in T_{i-1}$, without sorting explicitly all the representative strings in the buckets. The second point is to answer to LCA queries without doing the preprocessing in [88] of T_{i-1} from scratch, in each stage i. Using those two points and the algorithm in [35], the compacted tries that should be installed in T_{i-1} to obtain T_i can be built in $O(|T_i|-|T_{i-1}|)$ work and logarithmic time. Now, how to simulate the LCA preprocessing in the second point is shown in [57], by maintaining some suitable data structure. We discuss the first point, i.e., how to produce $L_i(f)$ for each $f \in T_{i-1}$, in the rest of this section. Actually, a list $L'_i(f)$ is produced, where $L'_i(f)$ is defined as $L_i(f)$, but the substrings starting in the positions given by $positions_{i-1}(f)$ are taken of length between $2 \cdot 2^{i-1}r$ and $3 \cdot 2^{i-1}r$, rather than $2 \cdot 2^{i-1}r$ only. Such a length, say $\ell(f)$, depends on the leaf f. If $\ell(f) = 2 \cdot 2^{i-1}r$ then $L'_i(f) = L_i(f)$; else, $L_i(f)$ can be easily obtained from $L'_i(f)$ in linear work, using the longest common prefix of consecutive strings in $L'_i(f)$ along with list ranking [61]. Thus, we focus only on how to build the ordered lists $L'_i(f)$ for each $f \in T_{i-1}$.

We describe first an initial computation taking $O(\log n)$ time and $O(n)$ work. It finds the *origin leaves* in T_0. Then we describe how such a computation is deployed for computing the lists $L'_i(f)$ in $O(\log^3 n)$ time and $w(i) = O(n/\log n + n/2^{i-1} + |T_i| - |T_{i-1}|)$ work, for each stage $i = 1, 2, \ldots, s$, where $\sum_{i=1}^{s} w(i) = O(n)$.

We need some preliminary definitions. H_i indicates the set of leaves of T_i (or, equivalenty, the distinct substrings of p of length $2^i r$ and the suffixes of p of length less than $2^i r$, except for '\$'). We define *0-links, 1-links*, and *\$-links*, also called *next-links*. Given a leaf $l \in H_i$, the *0-link* points to $l' \in H_i$ such that l and l' occur in two consecutive positions in p (i.e., the second suffix of l is a prefix of l'), and the last character of l' is 0. A similar definition applies to *1-links* and *\$-links*. Next-links play a crucial role and naturally induce, along with H_i, a *sparse digraph* denoted by J_i.

We describe the initial computation for the *origin leaves*: they form a subset of the vertices H_0 of the graph J_0. Let $G = J_0 = (H_0, E)$, where E is the set of next-links on H_0. Our goal is to produce a sequence of subgraphs G_1, G_2, G_3 of G, such that G_3 is a forest of rooted trees of $\lceil \log^2 n \rceil$ height and whose roots are defined to be the *the origin leaves* of T_0.

First, G_1 is obtained from G by removing the next-links from $leaf_0(j-1)$ to $leaf_0(j)$, for all positions j such that j is a *β-index*. More precisely, let $u = p[j{:}j + r - 1]$, $v = p[j-1]u$, and v' be v with $p[j-1]$ complemented. Then j is called a β-index if **(1)** v and v' occur at least once in p, and **(2)** either v occurs fewer times than v' in p, or v occurs as many times as v' and $p[j-1] = 0$. For technical reasons, $j = 1$ is always a β-index. All β-indices and their corresponding nodes in G can be determined in $O(1)$

time and $O(n)$ work, looking at vertices in G having indegree 2 (see [57, Lemma 5.1]). Second, G_2 is obtained from G_1 by removing one edge per cycle in G_1. The connected components in G_2 are rooted trees. Third, G_3 is obtained by splitting the trees in G_2: the unique edge incident onto each vertex of G_2 at distance a multiple of $\lceil \log^2 n \rceil$ from the root is removed. Now, the connected components in G_3 are rooted trees of $\lceil \log^2 n \rceil$ height. The roots of the trees in G_3 are chosen as the *origin leaves* in T_0.

We describe now stage i, which is composed of three main parts.

In the first part of the stage, a subset of special leaves, called *sources*, is selected from T_{i-1} using the origin leaves of T_0. For this purpose, consider the ordered set S of positions j in p such that $leaf_0(j)$ is an origin leaf. It is shown in [57] how to partition S into maximal contiguous subsequences, with the property that each such subsequence, say $j_1, j_2, \ldots, j_h$, verifies $leaf_0(j_1) = leaf_0(j_2) = \cdots = leaf_0(j_h)$ and no β-index k exists with $j_1 < k \leq j_h$. Without getting into much detail, some special positions, called *α-, γ-, and δ-indices*, are selected from those subsequences (i.e., the α-indices correspond to all singleton subsequences, the δ-indices are the first ones in the non-singleton subsequences, with the γ-indices being taken every $\Theta(\log^2 n)$ positions that differ by a multiple of some suitable period, in the non-singleton subsequences). Combinatorial properties in [57], based upon an upper bound called *bifurcation number* and the choice of $r = 2\lceil \log^3 n \rceil$ and $n \geq 8$, crucially show that the number of α-, β-, γ-, and δ-indices is $O(n/\log^2 n)$. Moreover, each subsequence $j_1, j_2, \ldots, j_h$ in S induces a periodic substring $p[j_1{:}end(j_1) - 1]$ (with $p[j_1{:}end(j_1)]$ not periodic), of period length $j_2 - j_1 = j_3 - j_2 = \cdots = j_h - j_{h-1} \leq \lceil \log^2 n \rceil$.

We can finally define a leaf $f \in T_{i-1}$ as a *source* if $f = leaf_{i-1}(j)$ for some j satisfying one of the following: **(1)** either no next-link is incident on f in J_{i-1}, or $j \neq 1$ is a β-index and there one next-link incident on f in J_{i-1}; **(2)** j is either an α-index or a γ-index; **(3)** j is a δ-index and $j + 2^{i-1}r - 1 < end(j)$. What is important to notice here is that there are $O(n/\log^2 n)$ sources f and positions j such that $leaf_{i-1}(j) = f$, since there are so many α-, β-, γ-, and δ-indices. The list of source leaves can be built in $O(\log n)$ time and $O(n/\log n)$ work (i.e., we implicit remove edges from J_{i-1}, similarly to the edge removal from $G = J_0$ to obtain G_3: the sources of T_{i-1} are the roots of the resulting trees, of $O(\log^2 n)$ height). From now on, if f is a source leaf then let $rep\text{-}strings(f)$ be the substrings of length $3 \cdot 2^{i-1}r$ that start in the positions of p given by $positions_{i-1}(f)$ (recall that the length of f as string is $2^{i-1}r$).

In the second part of stage i, $L'_i(f)$ is obtained only for the source leaves $f \in T_{i-1}$. We give here only a brief sketch. There are two cases.

If f is a source leaf satisfying conditions (1) and (2), then it means that either $positions_{i-1}(f) = \{1\}$ or $positions_{i-1}(f)$ contains only α-indices or only β-indices or only γ-indices (with $rep\text{-}strings(f)$ having period length larger than $\lceil \log^2 n \rceil$). Their overall number is $O(n/\log^2 n)$. Thus, to obtain the lists $L'_i(f)$, we can explicitly sort $rep\text{-}strings(f)$, for all those f, using the LCA queries in $O(n/\log n)$ work.

If f is a source leaf satisfying condition (3) then f must be periodic with period length at most $\lceil \log^2 n \rceil$. Then take any δ-index $j \in positions_{i-1}(f)$ and consider the other positions in its subsequence (called *family*). It is shown that only a subset of the strings $rep\text{-}strings(f)$ are distinct for all such j: those strings have starting positions in a consecutive portion of the family of each δ-index j. Moreover, they are already sorted for each such j, and their total number accumulates to $O(|T_i| - T_{i-1}|)$. By exploiting

the relationship between the periodicities (e.g., the position $end(j)$ associated to each δ-index j) and the lexicographic order of the strings in $rep\text{-}strings(f)$, it is shown in [57] that the lists associated with each δ-index above are almost globally sorted. Thus the lists $L'_i(f)$ are computed in $O(n/\log n + |T_i| - T_{i-1}|)$ work as the result of cleverly merging those ordered lists, which are available for each δ-index j.

Finally, in the third part of stage i, $L'_i(f)$ is produced for *all* leaves $f \in T_{i-1}$, using the computation in the first two parts and the repetitive nature of the suffix tree. By repetitive, we mean informally the following. Consider a generic node u of the suffix tree. If there is only one suffix link incident on u, say from node v, then the subtree rooted at u is isomorphic to the one rooted in v. Thus, assuming that u and v are leaves of T_{i-1}, the order in list $L'_i(u)$ is induced from the one in $L'_i(v)$. If there are two incident suffix links (recall that $|\Sigma| = 2$), say from v and w, then the subtree rooted at u can be obtained by "merging" the subtrees rooted at v and w. Thus, assuming that u, v and w are leaves of T_{i-1}, the order in list $L'_i(u)$ is induced from the one obtained by merging $L'_i(v)$ and $L'_i(w)$. Using our notation, the following is done.

A next-link in the graph J_{i-1} is said to be *frozen* if either it is incident upon a source leaf or it leads from $leaf_{i-1}(j-1)$ to $leaf_{i-1}(j)$ for some β-index $j \neq 1$. Consider now the graph J'_{i-1} obtained from J_{i-1} by removing the frozen next-links. Its connected components are rooted trees of height at most $3\lceil \log^2 n \rceil$, having a source as root [57, Lemma 5.13], for which we have already computed the list L'_i.

Given a non-source leaf f, we define the ordered list $L'_i(f)$ as composed of the representative substrings of length $3 \cdot 2^{i-1} r - k$, where $k \leq 3\lceil \log^2 n \rceil \leq r$ is the distance of f from the root of its connected component. This implies that $3 \cdot 2^{i-1} r - k \geq 2 \cdot 2^{i-1} r$. We visit the rooted trees in breadth-first, starting from the roots. The overall work of the computation below is $O(n/2^{i-1} + n/\log n + |T_i| - |T_{i-1}|)$ in stage i; time is $O(\log^3 n)$.

We refer the interested reader to [57] for more details and the correctness of the algorithm.

Theorem 5.3 (Hariharan [57]). *Let Σ be an already sorted alphabet, and x a string of length n. The suffix tree for x can be built in $O(\log^4 n)$ time, $O(n \log |\Sigma|)$ work and $O(n \log |\Sigma|)$ space on a CREW PRAM.*

Notice that it is still a main open problem to find a deterministic parallel algorithm that builds the suffix tree in optimal $O(n \log \min(n, |\Sigma|))$ work and $O(\log n)$ time (i.e., as fast as the one of Apostolico *et al.* [9] but having the same work and space as the sequential bound in Theorem 4.1). To the best of our knowledge, such bounds can be achieved only with a Las Vegas randomized algorithm, devised by Farach and Muthukrishnan [35].

6 The LSuffix Tree

In this section we generalize the notion of suffix tree to square matrices. Given an $n \times n$ square matrix A, whose entries are drawn from an ordered alphabet Σ, we would like to build a compacted trie similar to the suffix tree of a string. Namely, for each square submatrix of A, there must be a path in the trie which corresponds to that submatrix. The idea of a compact index for a square matrix was first introduced in the PAT trees

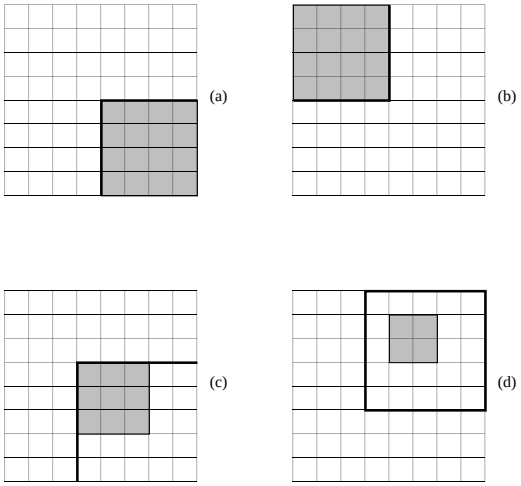

Figure 2. (a) The shaded region is the 5th suffix of the matrix A; (b) The 4th prefix of A; (c) The shaded submatrix is the 3rd prefix of the 4th suffix of A (the one with bold boundaries); (d) A_3 is illustrated as a bold square: the shaded submatrix of A is the 2nd prefix of the 2nd suffix of A_3.

of Gonnet [48, 49], where submatrices were represented as semi-infinite spirals called *sispirals*. The construction in [48, 49] requires $O(n^4)$ worst case time and $O(n^2 \log n)$ average time (for $|\Sigma| = O(1)$). A more efficient worst-case solution, based upon a different linear representation called *Lstring*, was later developed by Giancarlo [43], with his work on Lsuffix trees requiring $O(n^2 \log n)$ worst-case time. Later, the notion of suffix tree has been generalized to square matrices by Giancarlo and Grossi [46], for a wide class of linear representations of square matrices, giving rise to the definition of an exponential number of families of indices, including PAT trees and Lsuffix trees. Their construction works uniformly for any such index, requiring $O(n^2 \log n)$ worst case time and $O(n^2)$ average time (for $|\Sigma| = O(1)$). For the sake of brevity, we will only describe here the Lsuffix trees.

We first extend to square matrices some concepts already defined for strings. For $1 \leq j \leq n$, the square submatrix $A[j{:}n, j{:}n]$ is the *j-th suffix* of A, and the square submatrix $A[1{:}j, 1{:}j]$ is the *j-th prefix* of A (see Figg. 2a,2b). 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. 2c). We number each diagonal (not necessarily the main one) of A by d if its elements are $A[i, j]$ with $j - i = d$ and $|d| < n$. Let A_d be the square submatrix of A whose main diagonal is the d-th diagonal of A. Each square submatrix of A can be described as the prefix of a suffix of some diagonal submatrix A_d, for $|d| < n$ (see Fig. 2d).

We adopt a linear representation of a square matrix $A[1{:}n, 1{:}n]$ called *Lstring* [5, 43]. We divide A into n "L-shaped characters," the i-th being composed of row $A[i, 1{:}i-1]$ and column $A[1{:}i, i]$ (see Fig. 3a). The L-shaped characters are called *Lcharacters*. We write down the Lcharacters in one dimension, in the order given by their top-down appearance in A (see Fig. 3b). This way, we get a representation of A in terms of a string of Lcharacters that is called *Lstring*. We also need for Lstrings the notion of a *chunk*, which is the analog of the notion of substring for strings. Informally, we obtain

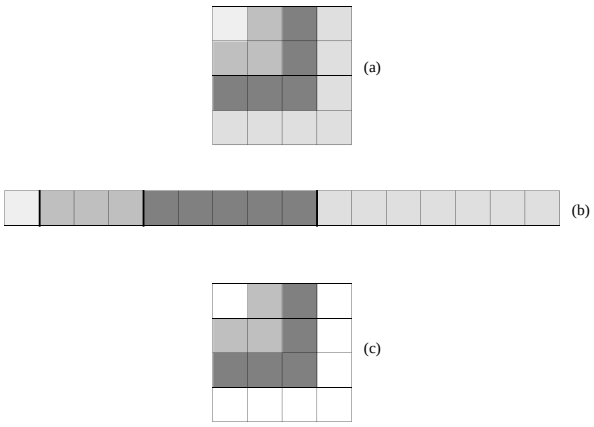

Figure 3. (a) a matrix A divided into Lcharacters (= same shading); (b) a linear representations as Lstrings; (c) a chunk composed of the 2nd and 3rd Lcharacter.

a chunk if we write down the Lcharacters of A in one dimension, in the order given by their top-down appearance in A and starting at row i and ending at row j (see Fig. 3c): Lstrings represent matrices while chunks represent "L-shaped" pieces of matrices.

We now describe how to generalize the notion of suffix trees to Lstrings. Before, we need more terminology. For any square matrix $A = A[1{:}n, 1{:}n]$, the corresponding Lstring $\mathrm{L}a = \mathrm{L}a[1{:}n]$ can be obtained as the concatenation of strings $\mathrm{L}a[i] = A[i, 1{:}i-1]A[1{:}i, i]$, each of length $2i-1$, for $1 \leq i \leq n$ (see Figg. 4a,4b). Each string can be seen as an Lcharacter. We denote the chunk from the i-th to the j-th Lcharacter as $\mathrm{L}a[i{:}j]$. The length of $\mathrm{L}a$ is n, that is, the number of Lcharacters in the Lstring. The j-th *Lprefix* of $\mathrm{L}a$, denoted $\mathrm{L}a[1{:}j]$, is the Lstring corresponding to $A[1{:}j, 1{:}j]$, the j-th prefix of A. Given A, it is easy to obtain $\mathrm{L}a[1{:}j]$ since we can take $A[1{:}j, 1{:}j]$ and write it down as an Lstring. Notice that $\mathrm{L}b$ is an Lprefix of $\mathrm{L}a$ if and only if the square matrix B corresponding to $\mathrm{L}b$ is equal to a prefix of A (see Fig. 4c). The j-th Lsuffix of $\mathrm{L}a$, denoted $\mathrm{L}suf_j(\mathrm{L}a)$, is the Lstring corresponding to $A[j{:}n, j{:}n]$, the j-th suffix of A. Notice, however, that $\mathrm{L}suf_j(\mathrm{L}a) \neq \mathrm{L}a[j{:}n]$ for $j > 1$. Once again, $\mathrm{L}b$ is Lsuffix of $\mathrm{L}a$ if and only if the square matrix B corresponding to $\mathrm{L}b$ is equal to a suffix of A (see Fig. 4d).

Let $\mathrm{L}a_d$ be the Lstring corresponding to a diagonal submatrix A_d, for $|d| < n$. By definition of prefix and suffix of a square matrix, every square submatrix of A is prefix of a suffix of some diagonal submatrix A_d. Thus, the correspondence between $\mathrm{L}a_d$ and A_d implies that every square submatrix of A corresponds to an Lstring, which is Lprefix of an Lsuffix of some Lstring $\mathrm{L}a_d$. Therefore, our Lsuffix tree must represent all Lsuffixes of $\mathrm{L}a_d$, for all diagonal $|d| < n$. That is, for each Lsuffix of $\mathrm{L}a_d$ there is a path from the root to a leaf "spelling out" that Lsuffix. This will guarantee the Completeness Property. The Common Prefix Property will be guaranteed by the fact that matrices with common prefixes yield Lstrings with common Lprefixes.

The Lsuffix tree for one Lstring (and corresponding square matrix) is like the suffix tree for a string. Just like the last character of a string is required to be unique for ordinary suffix trees, we require that the last Lcharacter of an Lstring be unique for Lsuffix trees. If necessary, we add an extra Lcharacter containing only \$'s. We denote

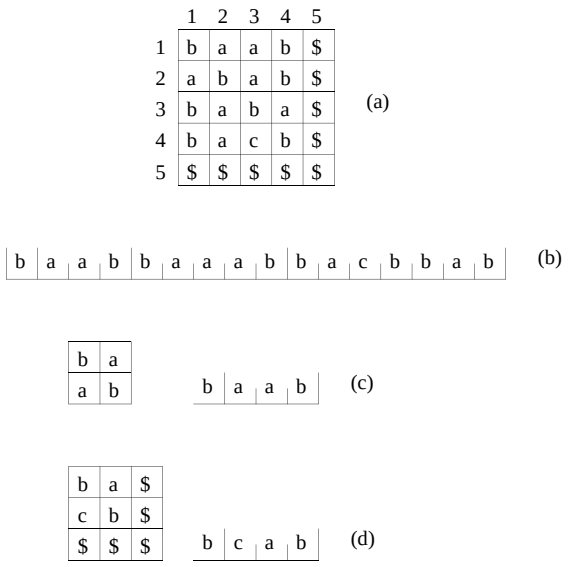

Figure 4. (a) matrix $A\hat{\$}$; (b) Lstring L$a$ corresponding to A ($ not shown); (c) the second prefix of A and its Lstring; (d) the third suffix of A and its Lstring.

such a special Lcharacter as $\hat{\$}$. With this addition, no Lsuffix of a given Lstring is Lprefix of any other Lsuffix of any other Lstring. When $\hat{\$}$ is appended to an $n \times n$ matrix, that matrix is augmented with a bottom row and rightmost column of $'s (e.g., see Fig. 4a). For our purposes, we need to define the Lsuffix tree for a set of Lstrings. Let C be the set of Lstrings corresponding to the diagonal submatrices $A_d\hat{\$}$, for $|d| < n$. That is, $C = \{\mathrm{L}a_d\hat{\$},\ \text{for } |d| < n\}$.

The *Lsuffix tree* LT_A for a matrix A is a compacted trie built on the set C and defined over the alphabet LΣ, which satisfies the following constraints:

(LT1) An arc of LT_A may store any nonempty chunk.

(LT2) Each internal node of LT_A must have at least two outgoing arcs.

(LT3) Chunks represented by sibling arcs start with different Lcharacters, which are of the same length as strings in Σ^*.

The concatenation of the (chunks represented by the) labels on the path from the root to a leaf gives exactly one Lsuffix of some Lstring in the set C (see Fig. 5).

Notice the analogy between Properties (LT1)–(LT3) and those for ordinary suffix trees described in Section 2. The size of the Lsuffix tree LT_A is $O(n^2)$ because there are n^2 leaves (one for each Lsuffix of the Lstrings in C) and no unary nodes. It is possible to represent the chunks labeling the arcs of LT_A in a compact way (see Fig. 5). Namely, a chunk L$suf_j(\mathrm{L}a_i\hat{\$})[p{:}q]$ is represented as a quadruple (p, q, j, i); an Lsuffix L$suf_j(\mathrm{L}a_i\hat{\$})$ corresponding to a leaf is represented as a pair (j, i), which will be considered as a quadruple $(1, \infty, j, i)$. We will refer to both types of quadruples as *descriptors*. Given a square matrix A, it is possible to locate in constant time a chunk from the corresponding descriptor and vice versa [43]. This implies that Lstrings do

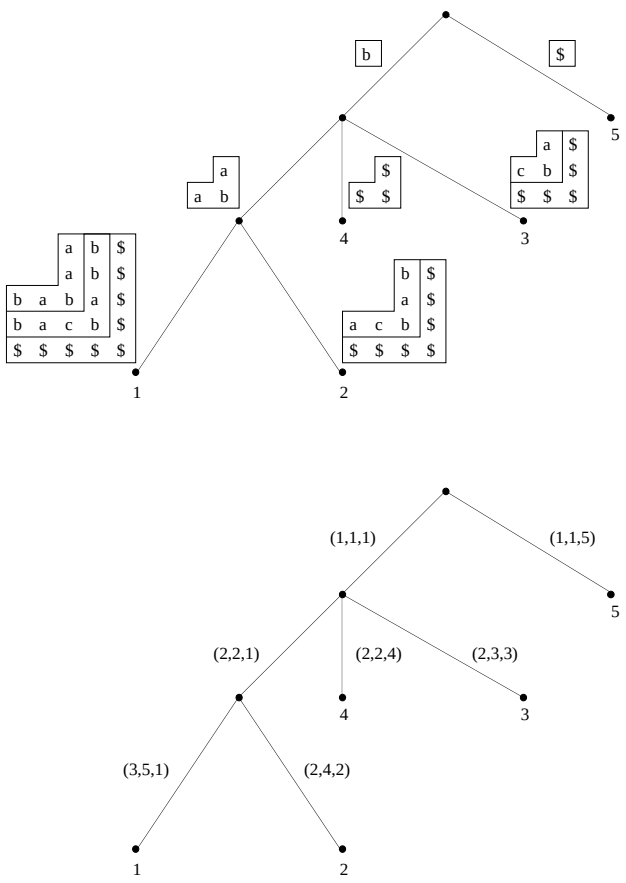

Figure 5. An Lsuffix tree for the Lstring corresponding to the main diagonal of matrix A in Fig. 4a, and its representation with descriptors. Here triples are used instead of quadruples since there is only one Lstring.

not have to be built explicitly from submatrices A_d, for $|d| < n$. However, we will use Lstrings for the sake of definition.

We point out that the $O(n^2)$ node outdegree of the Lsuffix tree $\mathrm{L}T_A$ can be reduced to $O(|\Sigma|)$ by transforming $\mathrm{L}T_A$ into a compacted trie over Σ [43]. This allows the $O(m^2 \log n)$ searching time for an $m \times m$ pattern matrix in $\mathrm{L}T_A$ to be reduced to $O(m^2 \log |\Sigma|)$ time.

7 Sequential Construction of an LSuffix Tree

In this section we describe the algorithm by Giancarlo [43], which computes the Lsuffix tree for a square matrix $A\hat{\$} = A[1{:}n+1, 1{:}n+1]$. This algorithm is a nontrivial generalization to Lstrings of the algorithm by McCreight for a set of strings described in Sections 4 and 4.3. The algorithm inserts the Lsuffixes of all Lstrings $\mathrm{L}a_d\hat{\$}$ with $|d| < n$. Without loss of generality, we focus on inserting one such Lstring, say $\mathrm{L}a\hat{\$}$.

Analogously to the algorithm of McCreight, the suffixes of $\mathrm{L}a\hat{\$}$ are inserted from longest to shortest, into a tree of initially two nodes. Denote by $\mathrm{L}T_i$ the tree at the end

of the i-th iteration. In other words, LT_i is a compacted trie over the alphabet LΣ of the Lcharacters from which we built the set of Lsuffixes L$suf_j(\text{L}a\hat{\$})$, for $1 \leq j \leq i$.

We denote by L$head_i$ the Lstring corresponding to the longest prefix of $A[i{:}n+1, i{:}n+1]$ which is also a prefix of $A[j{:}n+1, j{:}n+1]$ for some $1 \leq j < i$. Namely, L$head_i$ is the longest Lprefix of L$suf_i(\text{L}a\hat{\$})$ that is also an Lprefix of L$suf_j(\text{L}a\hat{\$})$ for some $j < i$. Note that L$head_i$ has a locus in LT_i by definition, but not necessarily has a locus in LT_{i-1}.

In analogy with Section 4, tree LT_1 for $A[1{:}n+1, 1{:}n+1]$ is composed of two nodes, and we obtain LT_i from LT_{i-1} as follows. We first find the *contracted* locus of L$head_i$ in LT_{i-1}: it is the locus of the longest Lprefix of L$head_i$ that is defined. Next, if it is needed we create a locus for L$head_i$ and install a new leaf representing L$suf_j(\text{L}a\hat{\$})$ as a child of the new locus. As in Section 4, this is accomplished through a rescanning and a scanning phase.

However, the main obstacle to the generalization of McCreight's algorithm is that the suffix links crucially rely on the following property [43]:

Distinct Right Context: For a string $x[1{:}n]$, if the longest common prefix of two suffixes $x[i{:}n]$ and $x[j{:}n]$ is of length $\ell > 0$, for $1 \leq i, j < n$, then the longest common prefix of suffixes $x[i+1{:}n]$ and $x[j+1{:}n]$ is of length $\ell - 1$.

The above property does not extend to Lstrings and matrices. Indeed, given two Lstrings LzLb and LzLc, with Lb and Lc being Lcharacters, L$b \neq$ Lc does not imply L$suf_2(\text{L}z\text{L}b) \neq$ L$suf_2(\text{L}z\text{L}c)$, as it happens for strings. Thus the rescanning phase has a major difference with the algorithm of McCreight for strings: if we try to define a suffix link for any node in LT_{i-1} in analogy with the definition given in Section 4, we cannot be guaranteed the existence in LT_{i-1} of the node where the suffix link should point to. This implies that even though an Lstring Lv has a locus in LT_{i-1}, it is not guaranteed that its second suffix L$suf_2(\text{L}v)$ has a locus (but there is a path corresponding to it). To circumvent this problem, we can use a relaxed notion of suffix links, which can be defined as a function SL_{i-1} as follows. If p is the root of LT_{i-1}, then $SL_{i-1}(p) = p$. Otherwise, let p be locus of L$v \neq \hat{\$}$: it is $SL_{i-1}(p) = z$, where z is the contracted locus of L$suf_2(\text{L}v)$. Once again, as in case of strings, $SL_{i-1}(\text{L}head_{i-1})$ is not always defined. However, SL_{i-1} is defined for all other nodes. Since contracted loci of Lstrings may change due to the insertion of new nodes, the suffix links we just defined change *dynamically* from LT_{i-1} to LT_i. Due to their dynamic nature, we store SL_{i-1} with the help of the dynamic trees of Sleator and Tarjan [91]. Since SL_{i-1} can be used to move efficiently from the locus of L$head_{i-1}$ to the contracted locus of L$suf_2(\text{L}head_{i-1})$, the rest of L$head_i$ (i.e., the scanning phase) can be found similarly to the scanning phase of McCreight's algorithm. To do this, we need primitives that manage Lcharacters in constant time, which can be done by building ordinary suffix trees on the rows and the columns of $A\hat{\$}$, and computing their longest common prefix similarly to [71, 72].

The following is an outline on how to insert Lsuffixes into the Lsuffix tree in the i-th iteration:

G1 : If the function SL_{i-1} is defined for L$head_{i-1}$, then set y as the node pointed by $SL_{i-1}(\text{L}head_{i-1})$ and skip Rescanning.

G2 : (*Rescanning*) Given LT_{i-1}, the locus of L$head_{i-1}$ in LT_{i-1}, and SL_{i-1}, find the contracted locus y of Lsuf_2(L$head_{i-1}$). Node y is located by taking a leaf q descending from the locus of L$head_{i-1}$ such that q is not the locus of Lsuf_{i-1}(L$a\hat{\$}$). Indeed, y must be an ancestor of the node pointed by $SL_{i-1}(q)$, which is defined by induction.

G3 : (*Scanning*) Start from node y and skip the Lcharacters that remains in Lsuf_2(L$head_{i-1}$) because surely they match. Then the contracted locus of L$head_i$ is found by examining one by one the Lcharacters that follow Lsuf_2(L$head_{i-1}$) in Lsuf_i(L$a\hat{\$}$) to go deeper and deeper in the tree. If needed, create a locus for L$head_i$ and install a new leaf representing L$suf_i(a\hat{\$})$ as a child of the new locus. Transform SL_{i-1} into SL_i.

We omit the remaining details of this construction, and its extension to a set C of Lstrings, giving only the following theorem.

Theorem 7.1 (Giancarlo [43]). *Given an $n \times n$ matrix A, the Lsuffix tree for A can be built in $O(n^2(\log n + \log|\Sigma|))$ time.*

8 Parallel Construction of an LSuffix Tree

We now describe the algorithm of Giancarlo and Grossi [45] for the parallel construction of the Lsuffix tree, which takes $O(\log n)$ time with $O(n^2 \log n)$ work on an Arbitrary CRCW PRAM. This algorithm has a high-level organization that is similar to the algorithm of Apostolico et al. [9] described in Section 5 (hereafter referred to as $AILSV$). However, some of the notions and tools used by $AISLV$ do not generalize to Lstrings and chunks, and we need new notions to make the algorithm work. In particular, in order to perform a refinement step, $AILSV$ implicitly uses the following properties of a string $x[1{:}n]$:

Positional Independence: For $i \neq j$, substrings $x[i{:}i+k-1]$ and $x[j{:}j+k'-1]$ admit a refiner 2^g, for $g \geq 0$ and $2^g \leq \min(k, k')$, if and only if $NAME_i[g] = NAME_j[g]$, that is, the names assigned to the two prefixes of length 2^g can be compared independently of their position inside x.

Name Splitting: Given a name for a substring $x[i, i+2^r-1]$, it can be always divided into a pair of names for its prefix $x[i, i+2^{r-1}-1]$ and its suffix $x[i+2^{r-1}, i+2^r-1]$, respectively. That is, $NAME_i[r]$ can always be split into two existing names, $NAME_i[r-1]$ and $NAME_{i+2^{r-1}}[r-1]$ to test the refiner 2^{r-1}.

Substring Inheritance: Let S_j and S_i be the set of substrings of length a power of two respectively taken from suffixes $x[j{:}n]$ and $x[i{:}n]$, for $j > i$. Then, $S_j \subseteq S_i$, that is, all the substrings in S_j are inherited from those in S_i. Thus it is sufficient to compute only the names of the substrings in S_1.

We now turn to the algorithm for the parallel construction of the Lsuffix tree of a square matrix A of size n^2. Assume w.l.o.g. that n is a power of two, and A is padded with n Lcharacters $\hat{\$}$. As $AILSV$, also this algorithm has two main phases: naming and refining.

The first phase gives names to the square submatrices of size $2^g \times 2^g$, for $0 \leq g \leq \log n$ as described in Section 5.1.

In the second phase, we also produce a sequence of refinement trees $\mathrm{L}D^{(r)}$, for $r = \log n, \ldots, 0$. Once again, the refinement step hinges on the notion of *refiner*. A *refiner* for two labels (p, q, j, i) and (p, q', j', i') in a nest is an integer $\ell > 0$ such that $(p, p+\ell-1, j, i) = (p, p+\ell-1, j', i')$. That is, the first ℓ Lcharacters of chunks $\mathrm{L}suf_j(\mathrm{L}a_i)[p{:}q]$ and $\mathrm{L}suf_{j'}(\mathrm{L}a_{i'})[p{:}q']$ are equal. Again, the arcs of tree $\mathrm{L}D^{(r)}$ are directed upwards, and the labels are on the nodes rather than on the arcs. The initial tree $\mathrm{L}D^{(\log n)}$ is composed of a root and n^2 leaves, one for each Lsuffix of $\mathrm{L}a_d\hat{\$}$, for $|d| < n$. The n^2 leaves are labeled with the descriptors of the Lsuffixes of $\mathrm{L}a_d\hat{\$}$, for $|d| < n$. The final tree $\mathrm{L}D^{(0)}$ is the Lsuffix tree, except that the direction of the arcs should be reversed. Tree $\mathrm{L}D^{(r)}$, $\log n \geq r \geq 0$, satisfies the same conditions satisfied by the Lsuffix tree (given in Section 6), with (LT3) replaced by the following:

(LT3') No two labels of nodes in the same nest admit a refiner 2^r. Moreover, the first Lcharacters of these labels have the same number of matrix elements and the same shape.

In analogy with $AILSV$, we also transform $\mathrm{L}D^{(r)}$ into $\mathrm{L}D^{(r-1)}$ by partitioning the nodes of each nest into equivalence classes. However, we need a new notion of equivalence because the Positional Independence Property for strings does not hold for an Lstring $La[1{:}n]$. Indeed, for $i \neq j$ and $g \geq 0$, chunks $\mathrm{L}a[i{+}1{:}i{+}2^g]$ and $\mathrm{L}a[j{+}1{:}j{+}2^g]$ of the same length are not even comparable: the former chunk is composed of a number of $(i{+}2^g)^2 - i^2$ characters in Σ that is different from the number of $(j{+}2^g)^2 - j^2$ characters of the latter chunk. Hence, the definition of equivalence classes for nests must account also for the constraint that chunks in the same class must be of comparable shape too, rather than of the same length only (as a straightforward generalization from $AILSV$ would suggest). Without getting into the details, we mention that it is possible to manage the new constraint on the equivalence classes through the use of Condition (LT3').

As in $AILSV$, we need to assign names to chunks and to compare and split such names efficiently. However, we cannot use the same ideas as in $AILSV$ because such ideas are based on the Name Splitting and Substring Inheritance Properties for strings, which do not generalize to Lstrings (and matrices, resp.). Indeed, an Lstring cannot be split into an Lprefix and an Lsuffix, but an Lprefix and a chunk (i.e., an L-shaped part). Such a chunk can be successively split only into chunks. Since there can be matrices (with $\Theta(n^2)$ distinct elements) that give rise to $\Omega(n^3 \log n)$ distinct chunks of length a power of two, versus $O(n^2 \log n)$ submatrices (hence, names) of side a power of two, the names for all such chunks cannot be computed beforehand with only $O(n^2 \log n)$ work. Instead, we proceed *on demand*, that is, when the names of the chunks are actually needed by the algorithm.

Given a chunk α, we compute an integer $name(\alpha)$ between 1 and n^2 on demand, so that equal chunks have equal names and such names can be efficiently split. If (p, q, j, i)

is the descriptor for α, for $p \geq 2$, then α corresponds to the portion of submatrix $A_i[j{:}j+q-1, j{:}j+q-1]$ obtained by removing the submatrix $A_i[j{:}j+p-2, j{:}j+p-2]$. The former submatrix is the smallest submatrix of A containing α. We call such a submatrix *capsular*. Capsular matrices are crucial for our task of assigning names to chunks on demand: Let α and β be two chunks of the same length, which are composed of the initial Lcharacters labeling two nodes in the same nest, and let M_α, M_β be their corresponding capsular submatrices. The chunks are equal if and only if their capsular submatrices are equal. That is, $\alpha = \beta$ if and only if $M_\alpha = M_\beta$ (see [45]).

The strong property above shows that the label $name(\alpha)$ for α can be computed by assigning a unique integer to M_α, through the use of Bulletin Board BB together with the names of the four overlapping submatrices, of side a power of two, which cover M_α. (If M_α has side a power of two, then the four submatrices are equal to M_α.) Recall that the names of all such submatrices have been computed during the first phase of our algorithm. In this way, splitting and comparing names of chunks on demand is reduced to a constant-time computation on their capsular matrices. Note that without using $name(\alpha)$, a "thin and slim" chunk α should have been decomposed into $\Omega(n)$ submatrices (hence, names) of side a power of two. Now, it is possible to define a split label (η_1, η_2) for a node u in tree $\mathrm{L}D^{(r)}$, where η_1 is the index of any chosen processor in the nest containing u, and $\eta_2 = name(\alpha)$ for a chunk α composed of the first 2^{r-1} Lcharacters labeling u. Using these split labels it is possible to partition the nests into equivalence classes similarly to $AILSV$.

We omit the details, as well the processor allocation and the analysis of this construction.

Theorem 8.1 (Giancarlo and Grossi [45]). *The Lsuffix tree for a square matrix A of size n^2 can be built in $O(\log n)$ time with n^2 processors on an Arbitrary CRCW PRAM. The required space is $O(n^{2+\epsilon})$ for any fixed $\epsilon > 0$.*

We finally remark that another interesting problem is to find an $O(n^2 \log |\Sigma|)$ work parallel construction of the Lsuffix tree. It would improve both Theorems 7.1 and 8.1.

9 Conclusions

In this paper we have surveyed the suffix tree, an ubiquitous data structure that appears in different fields related to string processing. We have presented some of its applications in different areas, and have described the main algorithmic techniques used for its sequential and parallel construction. We have given a particular emphasis to the newest developments related to suffix trees, such as parallel algorithms for suffix tree construction and generalizations of suffix trees to higher dimensions, which are important in multidimensional pattern matching.

Acknowledgments

We are indebted to Dany Breslauer for many useful comments, to R. Giegerich for sending us reference [47], and to Gaston Gonnet for pointing out reference [48].

References

[1] Aho, A. V., Algorithms for finding patterns in strings, in *Handbook of Theoretical Computer Science, vol. A*, J. van Leeuwen ed., MIT Press, Cambridge, MA, 255–300, (1990).

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

[3] Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *The design and analysis of computer algorithms*, Addison-Wesley, Reading, MA, (1974).

[4] Akutsu, T., A linear time pattern matching algorithm between a string and a tree, *Combinatorial Pattern Matching*, 1–10, (1993).

[5] Amir, A., and Farach, M., Two-dimensional dictionary matching, *Information Processing Letters*, 44, 233-239, (1992).

[6] Amir, A., Farach, M., Galil, Z., Giancarlo, R., and Park, K., Dynamic dictionary matching, *Journal of Computer and System Science*, 49, 208–222, (1994).

[7] Amir, A., Farach, M., and Matias, Y., Efficient randomized dictionary matching algorithms, *Combinatorial Pattern Matching*, 262–275, (1992).

[8] Apostolico, A., The myriad virtues of subword trees, in *Combinatorial algorithms on words*, A. Apostolico and Z. Galil eds., Springer-Verlag, Berlin, 85–95, (1985).

[9] Apostolico, A., Iliopoulos, C., Landau, G. M., Schieber, B., and Vishkin, U., Parallel construction of a suffix tree with applications, *Algorithmica*, 3, 347–365, (1988).

[10] Apostolico, A., and Preparata, F. P., Optimal off-line detection of repetitions in a string, *Theoret. Comp. Sci.* 22, 297–315, (1983).

[11] Apostolico, A., and Preparata, F. P., Structural properties of the string statistics problem, *Journal of Comput. and Syst. Sci.*, 31, 394–411, (1985).

[12] Apostolico, A., and Szpankowski, W., Self-alignment in words and their applications, *J. Algorithms*, 13, 446–467, (1992).

[13] Baker, B.S., A theory of parameterized pattern matching: algorithms and applications, *Proc. 25th Symp. on Theory of Computing*, 71–80, (1993).

[14] Bagget, P., Ehrenfeucth, A., and Perry, M., A technique for designing computer access and selecting good terminology, *Proc. Rocky Mountains Conference on Artificial Intelligence*, Breit International Inc., Boulder, Colorado, (1986).

[15] Bhatt, P.C., Diks, K., Hagerup, T., Prasad, V.C., Radzik, T., and Saxena, S., Improved deterministic parallel integer sorting, *Information and Computation*, 94, 29–47, (1991).

[16] Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M. T., and Seiferas, J., The smallest automaton recognizing the subwords of a text, *Theoret. Comput. Sci.*, 40, 31–55, (1985).

[17] Blumer, A., Blumer, J., Haussler, D., McConnell, R., and Ehrenfeucht, A., Complete inverted files for efficient text retrieval and analysis, *J. ACM* 34, 578–595, (1987).

[18] Blumer, A., Ehrenfeucht, A., and Haussler, D., Average size of suffix trees and DAWGs, *Discrete Appl. Math.* 24, 37–45, (1989).

[19] Booth, K.S., Lexicographically least circular substrings, *Information Processing Letters*, 10, 240-242, (1980).

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

[21] Breslauer, D., Dictionary-matching on unbounded alphabets: uniform-length dictionaries, *Combinatorial Pattern Matching*, 184–197, (1994).

[22] Cardenas, A. F., Analysis and performance of inverted data base structures, *Comm. ACM*, 5, 253–263, (1975).

[23] Chang, W. I., and Lawler, E. L., Sublinear approximate string matching and biological applications, *Algorithmica*, 12, 327–344, (1994).

[24] Chen, M. T., and Seiferas, J., Efficient and elegant subword tree construction, in *Combinatorial algorithms on words*, A. Apostolico and Z. Galil eds., Springer-Verlag, Berlin, 97–107, (1985).

[25] Clift, B., Haussler, D., McConnell, R., Schneider, T. D., and Stormo, G. D., Sequences landscapes, *Nucleic Acids Research*, 4, 141–158, (1986).

[26] Cole, R., Parallel merge sort, *SIAM J. Comput.*, 17, 770–785, (1988).

[27] Cole, R., and Vishkin, U., Deterministic coin tossing with application to parallel list ranking, *Information and Control*, 70, 32–53, (1986).

[28] Crochemore, M., Transducers and repetitions, *Theoretical Computer Science* 45, 63–86, (1986).

[29] Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., and Rytter, W., Speeding up two string-matching algorithms, *Algorithmica*, 12, 247–267, (1994).

[30] Crochemore, M., and Rytter, W., Parallel construction of minimal suffix and factor automata, *Inf. Proc. Let.* 35, 121–128, (1990).

[31] Crochemore, M., and Rytter, W., Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays, *Theoretical Computer Science*, 88, 59–82, (1991).

[32] Devroye, L., Szpankowski, W., and Rais, B., A note on the height of suffix trees, *SIAM J. Comput.*, 21, 48–53, (1993).

[33] Dubiner, M., Galil, Z., and Magen, E., Faster tree pattern matching, *J. ACM*, 14, 205-213 (1994).

[34] Ehrenfeucht, A., and Haussler, D., A new distance metric on strings computable in linear time, *Disc. Applied Math.*, 20, 191-203, (1988).

[35] Farach, M., and Muthukrishnan, S., Private Communication, (1994).

[36] Ferragina, P., Incremental text editing: a new data structure, *Proc. European Symposium on Algorithms*, 495–507, (1994).

[37] Ferragina, P., and Grossi, R., Fast incremental text editing, *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 531–540, (1995).

[38] Ferragina, P., and Grossi, R., A fully-dynamic data structure for external substring search, *Proc. ACM Symposium on Theory of Computing* (1995).

[39] Fiala, E. R., and Greene, D. H., Data compression with finite windows, *Comm. ACM*, 32, 490–505, (1989).

[40] Fraser, C., Wendt, A., and Myers, E. W., Analyzing and compressing assembly code, *Proc. SIGPLAN Symp. on Compiler Construction*, 117–121, (1984).

[41] Galil, Z., and Giancarlo, R., Data structures and algorithms for approximate string matching, *J. Complexity*, 4, 33–72, (1988).

[42] Galil, Z., and Park, K., An improved algorithm for approximate string matching, *SIAM J. Comput.*, 19, 989–999, (1990).

[43] Giancarlo, R., The suffix tree of a square matrix, with applications, *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, 402–411, (1993). To appear in *SIAM J. Comput.*

[44] Giancarlo, R., An index data structure for matrices, with applications to fast two-dimensional pattern matching, *Proc. of Workshop on Algorithms and Data Structures*, (1993).

[45] Giancarlo, R., and Grossi, R., Parallel construction and query of suffix trees for two-dimensional matrices, *Proc. ACM Symp. on Parallel Algorithms and Architectures*, (1993).

[46] Giancarlo, R., and Grossi, R., On the construction of classes of suffix trees for square matrices: algorithms and applications, *Proc. International Colloquium on Automata, Languages, and Programming*, (1995).

[47] Giegerich, R., and Kurtz, S., From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction, Technical report 94-03, Universität Bielefeld, Technische Fakultät, Germany, (1994).

[48] Gonnet, G. H., Efficient searching of text and pictures. Technical report OED-88-02, University of Waterloo, (1988).

[49] Gonnet, G. H., and Baeza-Yates, R., *Handbook of Algorithms and Data Structures*. Addison-Wesley, (1991).

[50] Gonnet, G. H., Baeza-Yates, R. A., and Snider, T., New indices for text: PAT trees and PAT arrays. *Information Retrieval: Data Structures and Algorithms*, W.B. Frakes and R.A. Baeza-Yates, Eds., Prentice-Hall, 66–82, (1992).

[51] Grassberger, P., Estimating the information content of symbol sequences and efficient codes, *IEEE Trans. Information Theory* 35, 669–675, (1991).

[52] Gu, M., Farach, M., and R. Beigel, An efficient algorithm for dynamic text indexing, *Proc. ACM-SIAM Symposium on Discrete Algorithms*, (1994).

[53] Guibas, L., and Odlyzko, A., Periods in strings, *J. Combinatorial Theory Ser.A*, 30, 19-43, (1981).

[54] Guibas, L., and Odlyzko, A., String overlaps, pattern matching, and nontransitive games, *J. Combinatorial Theory Ser.A*, 30, 183-208, (1981).

[55] Gusfield, D., Landau, G. M., and Schieber., B., An efficient algorithm for all pairs suffix-prefix problem, *Information Processing Letters*, 41, 181–185, 1992.

[56] Harel, H. T., and Tarjan, R. E, Fast algorithms for finding nearest common ancestors, *SIAM Journal on Computing*, 13, 338–355, (1984).

[57] Hariharan, R., Optimal parallel suffix tree construction, *Proc. 26th Symp. on Theory of Computing*, (1994).

[58] Hui, L.C.K, Color set size problem with applications to string matching, *Combinatorial Pattern Matching*, 230–243, (1992).

[59] Ito, M., Shimizu, K., Nakanishi, M., and Hashimoto, A., Polynomial-time algorithms for computing characteristic strings, *Combinatorial Pattern Matching*, 274–288, (1994).

[60] Jacquet, P., and Szpankowski, W., Autocorrelation on words and its application: Analysis of suffix trees by string-ruler approach, *J. Combinatorial Theory Ser.A*, 66, 237-269, (1994).

[61] JáJá, J., *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA, (1992).

[62] Jain, R., Workshop report on visual information systems, Tech. Rep., National Science Foundations, (1992).

[63] Karkkainen, J., Suffix cactus: a cross between suffix tree and suffix array, *Proc. Combinatorial Pattern Matching*, (1995).

[64] Karp, R. M., Miller, R. E., and Rosenberg, A. L., Rapid identification of repeated patterns in strings, trees and arrays, *Proc. 4th Annual ACM Symp. on Theory of Comput.*, 125–136, (1972).

[65] Kempf, M., Bayer, R., and Güntzer, U., Time optimal left to right construction of position trees, *Acta Informatica*, 24, 461–474, (1987).

[66] Kosaraju, S.R., Fast pattern matching in trees, *Proc. 30th IEEE Symp. on Found. of Computer Science*, 178-183, (1989).

[67] Kosaraju, S.R., Real-time pattern matching and quasi-real-time construction of suffix trees, *Proc. 26th Symp. on Theory of Computing*, (1994).

[68] Kosaraju, S.R., Computation of squares in a string, *Combinatorial Pattern Matching*, 146-150 (1994).

[69] D. E. Knuth, J. H. Morris and V. R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.*, 6 (1977), 63–78.

[70] Landau, G. M., and Vishkin, U., Introducing efficient parallelism into approximate string matching, *Proc. 18th Symp. on Theory of Computing*, 220–230, (1986).

[71] Landau, G. M., and Vishkin, U., Fast string matching with k differences, *Journal of Computer and System Science*, 37, 63–78, (1988).

[72] Landau, G. M., and Vishkin, U., Fast parallel and serial approximate string matching, *J. Algorithms*, 10, 157–169, (1989).

[73] Lempel, A., and Ziv, A., On the complexity of finite sequences, *IEEE Trans. Information Theory*, 22, 75–81, (1976).

[74] López-Ortiz, A., Linear pattern matching of repeated substrings, *SIGACT News*, 25, 114-121, (1994).

[75] Majster, M. E., and Reiser, A., Efficient on-line construction and correction of position trees, *SIAM J. Comput.*, 9, 785–807, (1980).

[76] Manber, U., and Myers, G., Suffix arrays: a new method for on-line string searches, *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms*, 319–327, (1990).

[77] McCreight, E. M., A space-economical suffix tree construction algorithm, *J. ACM*, 23, 262–272, (1976).

[78] Morrison, D. R., PATRICIA – Practical algorithm to retrieve information coded in alphanumeric, *J. ACM*, 15, 514–534, (1968).

[79] Myers, E., An $O(ND)$ difference algorithm and its variations, *Algorithmica*, 1, 251-266, (1986).

[80] Myers, E., A sublinear algorithm for approximate keyword searching, *Algorithmica*, 12, (1994).

[81] O'Connor, and Snider, Suffix trees and string complexity, *Advances in Cryptology: Proc. of EUROCRYPT*, LNCS 658, (1992).

[82] Pratt, V., Improvements and applications for the Weiner repetition finder, Unpublished manuscript, (1975).

[83] Rodeh, M., A fast test for unique decipherability based on suffix trees, *IEEE Trans. Information Theory*, 28, 648-651, (1982).

[84] Rodeh, M., Pratt, V., and Even, S., Linear algorithm for data compression via string matching, *J. ACM*, 28, 16–24, (1991).

[85] Rosenfeld, A., and Kak, A. C., *Digital Picture Processing*, Academic Press, (1982).

[86] Sahinalp, S.c., and Vishkin, U., Symmetry breaking for suffix tree construction *Proc. 26th Symp. on Theory of Computing*, (1994).

[87] Sankoff, D., and Kruskal, J. B., eds., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, MA, (1983).

[88] Schieber, B., and Vishkin, U., On finding lowest common ancestor: simplification and parallelization, *SIAM J. Comp.*, 17, 1253-1262, (1988).

[89] Shields, P., Entropy and prefixes, *Annals of Probability*, 20, 403-409, (1992).

[90] Shiloach, Y., Fast canonization of circular strings, *J. of Algorithms* 2 (1981) 107-121.

[91] Sleator, D. D., and Tarjan, R. E., A data structure for dynamic trees, *Journal of Computer and System Science*, 24, 362–381, (1983).

[92] Slisenko, O., Detection of periodicities and string-matching in real time, *Journal of Soviet Mathematics*, 22, 1316–1387, (1983).

[93] Storer, J.A., Private communication, (1995).

[94] Szpankowski, W., A generalized suffix tree and its (un)expected asymptotic behaviors, *SIAM J. Comp.*, 22, 1176-1198, (1993).

[95] Szpankowski, W., Asymptotic properties of data compression and suffix trees, *IEEE Trans. Information Theory*, 33, (1993).

[96] Tanimoto, S. L., A method for detecting structure in polygons, *Pattern Recognition*, 13, 389–394 (1981).

[97] Teskey, F.N., *Principles of text processing*, J. Wiley & Sons, 1983.

[98] Ukkonen, E., Approximate string-matching over suffix trees, *Combinatorial Pattern Matching*, 228-242, (1993).

[99] Ukkonen, E., On-line construction of suffix trees, Tech. Report A-1993-1, Department of Computer Science, University of Helsinki, Finland, (1993).

[100] Waterman, M., ed., *Mathematical methods for DNA sequences*, CRC Press Inc., Boca Raton, FL, (1991).

[101] Weiner, P., Linear pattern matching algorithm, *Proc. 14th IEEE Symp. on Switching and Automata Theory*, 1–11, (1973).

[102] Wyner, A., and Ziv, J., Some asymptotic properties of the entropy of a stationary ergodic data source with applications to data compression, *IEEE Trans. Information Theory* 35, 1250–1258, (1989).

[103] Ziv, J., and Lempel, A., A universal algorithm for sequential data compression, *IEEE Trans. Information Theory* 23, 337–343, (1977).