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.

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, and describe some of its applications. Several algorithms for its sequential construction are described in Section 3. Section 4 contains some applications and extensions of suffix trees. Finally, Section 5 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 there are at most $n + 1$ leaves (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 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 will 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 node in the suffix tree such that its pathstring starts with aba, namely leaf 3. Instead,

4
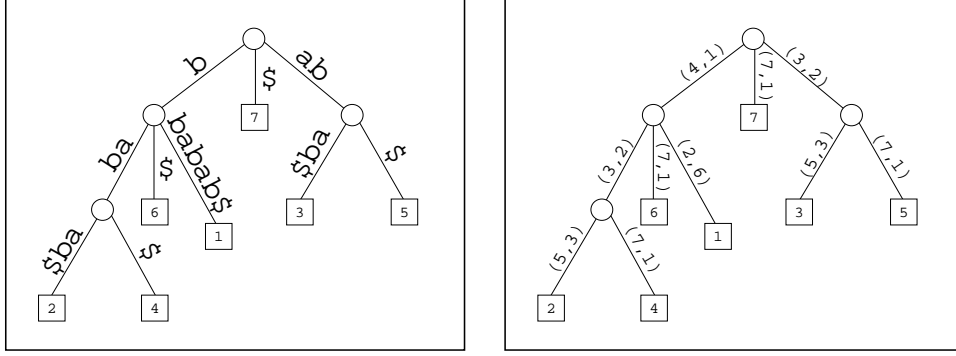
Figure 1. The suffix tree for the string $\omega = bbabab\$$, and its compact representation.

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.

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.

We now show the suffix tree "at work" by briefly discussing few of its typical uses on strings. 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 suf-

fix 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]. We will see some other extensions and generalizations of suffix trees in Section 4.

3   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.

3.1   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.

The first problem is how 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. Fortunately, the following lemma shows how to avoid this pathological behavior.

Lemma 3.1. *Let y be the longest prefix of $az\$$ that is already a substring of $z\$$, and let v be the second suffix of y (i.e., $y = av$). Then v must be a node in the suffix tree of $z\$$.*

Proof: To prove the lemma, we use some properties that follow directly 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 $z\$$: by the Node Existence Property, v must be a node in the suffix tree of $z\$$. □

Using Lemma 3.1, we can locate y as follows. 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'. This observation will be crucial for our time analysis.

Theorem 3.1 (Chen and Seiferas [24]). *Let Σ be an ordered alphabet, with constant $|\Sigma|$ and x be a string of n characters over Σ. The suffix tree of $x\$$ can be computed in total time $O(n)$.*

Proof: We use the previous observation to compute the total time required to build the suffix tree of the string $x\$$. Namely, we show by an amortization argument that at each step 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

9

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. □

3.2   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 3.2 (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 3.2), 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 3.2, 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 3.2 (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.

3.3   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 3.2) 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].

4   Extensions and Generalizations of Suffix Trees

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 $string(Q) = x_1y_1 \cdots x_my_m x_1y_1 \cdots x_{m-1}y_{m-1}$. 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.

5   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

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).