1

Suffix Trees and Suffix Arrays

Srinivas Aluru
Iowa State University

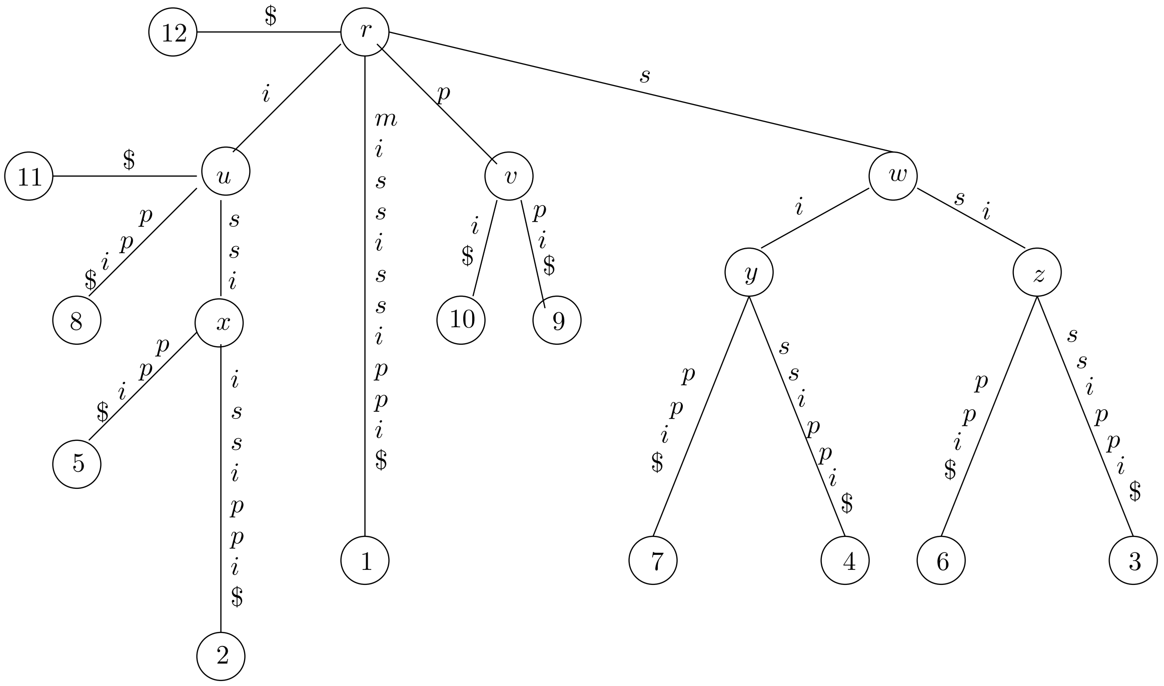
1.1 Basic Definitions and Properties

Suffix trees and suffix arrays are versatile data structures fundamental to string processing applications. Let s' denote a string over the alphabet Σ. Let $\$ \notin \Sigma$ be a unique termination character, and $s = s'\$$ be the string resulting from appending \$ to s'. We use the following notation: $|s|$ denotes the size of s, $s[i]$ denotes the i^{th} character of s, and $s[i..j]$ denotes the substring $s[i]s[i+1]\ldots s[j]$. Let $suff_i = s[i]s[i+1]\ldots s[|s|]$ be the suffix of s starting at i^{th} position.

The suffix tree of s, denoted $ST(s)$ or simply ST, is a compacted trie of all suffixes of string s. Let $|s| = n$. It has the following properties:

1. The tree has n leaves, labelled $1 \ldots n$, one corresponding to each suffix of s.
2. Each internal node has at least 2 children.
3. Each edge in the tree is labelled with a substring of s.
4. The concatenation of edge labels from the root to the leaf labelled i is $suff_i$.
5. The labels of the edges connecting a node with its children start with different characters.

The paths from root to the suffixes labelled i and j coincide up to their longest common prefix, at which point they bifurcate. If a suffix of the string is a prefix of another longer suffix, the shorter suffix must end in an internal node instead of a leaf, as desired. It is to avoid this possibility that the unique termination character is added to the end of the string. Keeping this in mind, we use the notation $ST(s')$ to denote the suffix tree of the string obtained by appending \$ to s'.



SA	12	11	8	5	2	1	10	9	7	4	6	3
Lcp	0	1	1	4	0	0	1	0	2	1	3	▨

FIGURE 1.1: Suffix tree, suffix array and *Lcp* array of the string *mississippi*. The suffix links in the tree are given by $x \rightarrow z \rightarrow y \rightarrow u \rightarrow r$, $v \rightarrow r$, and $w \rightarrow r$.

As each internal node has at least 2 children, an n-leaf suffix tree has at most $n - 1$ internal nodes. Because of property (5), the maximum number of children per node is bounded by $|\Sigma| + 1$. Except for the edge labels, the size of the tree is $O(n)$. In order to allow a linear space representation of the tree, each edge label is represented by a pair of integers denoting the starting and ending positions, respectively, of the substring describing the edge label. If the edge label corresponds to a repeat substring, the indices corresponding to any occurrence of the substring may be used. The suffix tree of the string *mississippi* is shown in Figure 1.1. For convenience of understanding, we show the actual edge labels.

The suffix array of $s = s'\$$, denoted $SA(s)$ or simply SA, is a lexicographically sorted array of all suffixes of s. Each suffix is represented by its starting position in s. $SA[i] = j$ iff $Suff_j$ is the i^{th} lexicographically smallest suffix of s. The suffix array is often used in conjunction with an array termed *Lcp* array, containing the lengths of the longest common prefixes between every consecutive pair of suffixes in SA. We use $lcp(\alpha, \beta)$ to denote the longest common prefix between strings α and β. We also use the term *lcp* as an abbreviation for the term *longest common prefix*. $Lcp[i]$ contains the length of the *lcp* between $suff_{SA[i]}$ and $suff_{SA[i+1]}$, i.e., $Lcp[i] = lcp(suff_{SA[i]}, suff_{SA[i+1]})$. As with suffix trees, we use the notation $SA(s')$ to denote the suffix array of the string obtained by appending \$ to s'. The suffix and *Lcp* arrays of the string *mississippi* are shown in Figure 1.1.

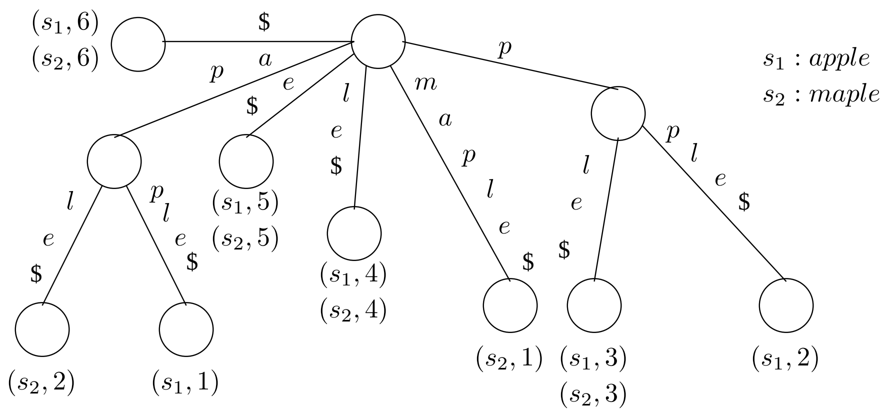
Let v be a node in the suffix tree. Let *path-label*(v) denote the concatenation of edge labels along the path from root to node v. Let *string-depth*(v) denote the length of *path-label*(v). To differentiate this with the usual notion of depth, we use the term *tree-depth* of a node to denote the number of edges on the path from root to the node. Note that the length of the longest common prefix between two suffixes is the string depth of the lowest common ancestor of the leaf nodes corresponding to the suffixes. A repeat substring of string S is *right-maximal* if there are two occurrences of the substring that are succeeded by different characters in the string. The path label of each internal node in the suffix tree corresponds to a right-maximal repeat substring and vice versa.

Let v be an internal node in the suffix tree with path-label $c\alpha$ where c is a character and α is a (possibly empty) string. Therefore, $c\alpha$ is a right-maximal repeat, which also implies that α is also a right maximal repeat. Let u be the internal node with path label α. A pointer from node v to node u is called a *suffix link*; we denote this by $SL(v) = u$. Each suffix $suff_i$ in the subtree of v shares the common prefix $c\alpha$. The corresponding suffix $suff_{i+1}$ with prefix α will be present in the subtree of u. The concatenation of edge labels along the path from v to leaf labelled i, and along the path from u to leaf labelled $i+1$ will be the same. Similarly, each internal node in the subtree of v will have a corresponding internal node in the subtree of u. In this sense, the entire subtree under v is contained in the subtree under u.

Every internal node in the suffix tree other than the root has a suffix link from it. Let v be an internal node with $SL(v) = u$. Let v' be an ancestor of v other than the root and let $u' = SL(v')$. As *path-label*(v') is a prefix of *path-label*(v), *path-label*(u') is also a prefix of *path-label*(u). Thus, u' is an ancestor of u. Each proper ancestor of v except the root will have a suffix link to a distinct proper ancestor of u. It follows that *tree-depth*$(u) \geq$ *tree-depth*$(v) - 1$.

Suffix trees and suffix arrays can be generalized to multiple strings. The generalized suffix tree of a set of strings $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$, denoted $GST(\mathcal{S})$ or simply GST, is a compacted trie of all suffixes of each string in $\mathcal{S}$. We assume that the unique termination character \$ is appended to the end of each string. A leaf label now consists of a pair of integers (i, j), where i denotes the suffix is from string s_i and j denotes the starting position of the suffix in s_i. Similarly, an edge label in a GST is a substring of one of the strings. An edge label is represented by a triplet of integers (i, j, l), where i denotes the string number, and j and l denote the starting and ending positions of the substring in s_i. For convenience of understanding, we will continue to show the actual edge labels. Note that two strings may have identical suffixes. This is compensated by allowing leaves in the tree to have multiple labels. If a leaf is multiply labelled, each suffix should come from a different string. If N is the total number of characters (including the \$ in each string) of all strings in $\mathcal{S}$, the GST has at most N leaf nodes and takes up $O(N)$ space. The generalized suffix array of $\mathcal{S}$, denoted $GSA(\mathcal{S})$ or simply GSA, is a lexicographically sorted array of all suffixes of each string in $\mathcal{S}$. Each suffix is represented by an integer pair (i, j) denoting suffix starting from position j in s_i. If suffixes from different strings are identical, they occupy consecutive positions in the GSA. For convenience, we make an exception for the suffix \$ by listing it only once, though it occurs in each string. The GST and GSA of strings *apple* and *maple* are shown in Figure 1.2.

Suffix trees and suffix arrays can be constructed in time linear to the size of the input. Suffix trees are very useful in solving a plethora of string problems in optimal run-time bounds. Moreover, in many cases, the algorithms are very simple to design and understand. For example, consider the classic pattern matching problem of determining if a pattern P occurs in text T over a constant sized alphabet. Note that P occurs starting from position i in T iff P is a prefix of $suff_i$ in T. Thus, whether P occurs in T or not can be determined



GSA	$(s_1, 6)$	$(s_2, 2)$	$(s_1, 1)$	$(s_1, 5)$	$(s_2, 5)$	$(s_1, 4)$	$(s_2, 4)$	$(s_2, 1)$	$(s_1, 3)$	$(s_2, 3)$	$(s_1, 2)$

FIGURE 1.2: Generalized suffix tree and generalized suffix array of strings *apple* and *maple*.

by checking if P matches an initial part of a path from root to a leaf in $ST(T)$. Traversing from the root matching characters in P, this can be determined in $O(|P|)$ time, independent of the size of T. As another application, consider the problem of finding a longest common substring of a pair of strings. Once the GST of the two strings is constructed, all that is needed is to identify an internal node with the largest string depth that contains at least one leaf from each string. These and many other applications are explored in great detail in subsequent sections. Suffix arrays are of interest because they require much less space than suffix trees, and can be used to solve many of the same problems. We first concentrate on linear time construction algorithms for suffix trees and suffix arrays. The reader interested in applications can safely skip to Section 1.3.

1.2 Linear Time Construction Algorithms

In this section, we explore linear time construction algorithms for suffix trees and suffix arrays. We also show how suffix trees and suffix arrays can be derived from each other in linear time. In suffix tree and suffix array construction algorithms, three different types of alphabets are considered – a constant or fixed size alphabet ($|\Sigma| = O(1)$), integer alphabet ($\Sigma = \{1, 2, \ldots, n\}$), and arbitrary alphabet. Suffix trees and suffix arrays can be constructed in linear time for both constant size and integer alphabets. The constant alphabet size case covers many interesting application areas, such as English text, or DNA or protein sequences in molecular biology. The integer alphabet case is interesting because a string of length n can have at most n distinct characters. Furthermore, some algorithms use a recursive technique that would generate and require operating on strings over integer alphabet, even when applied to strings over a fixed alphabet.

1.2.1 Suffix Trees vs. Suffix Arrays

We first show that the suffix array and *Lcp* array of a string can be obtained from its suffix tree in linear time. Define lexicographic ordering of the children of a node to be the order based on the first character of the edge labels connecting the node to its children. Define lexicographic depth first search to be a depth first search of the tree where the children of each node are visited in lexicographic order. The order in which the leaves of a suffix tree are visited in a lexicographic depth first search gives the suffix array of the corresponding string. In order to obtain *lcp* information, the string-depth of the current node during the search is remembered. This can be easily updated in $O(1)$ time per edge as the search progresses. The length of the *lcp* between two consecutive suffixes is given by the smallest string-depth of a node visited between the two suffixes.

Given the suffix array and the *Lcp* array of a string s ($|s\$| = n$), its suffix tree can be constructed in $O(n)$ time. This is done by starting with a partial suffix tree for the lexicographically smallest suffix, and repeatedly inserting subsequent suffixes in the suffix array into the tree until the suffix tree is complete. Let T_i denote the compacted trie of the first i suffixes in lexicographic order. The first tree T_1 consists of a single leaf labelled $SA[1]$ connected to the root with an edge labelled $suff_{SA[1]} = \$$.

To insert $SA[i+1]$ into T_i, start with the most recently inserted leaf $SA[i]$ and walk up $(|suff_{SA[i]}| - |lcp(suff_{SA[i]}, suff_{SA[i+1]})|) = ((n - SA[i] + 1) - Lcp[i])$ characters along the path to the root. This walk can be done in $O(1)$ time per edge by calculating the lengths of the respective edge labels. If the walk does not end at an internal node, create an internal node. Create a new leaf labelled $SA[i+1]$ and connect it to this internal node with an edge. Set the label on this edge to $s[SA[i+1] + Lcp[i]..n]$. This creates the tree T_{i+1}. The procedure works because $suff_{SA[i+1]}$ shares a longer prefix with $suff_{SA[i]}$ than any other suffix inserted so far. To see that the entire algorithm runs in $O(n)$ time, note that inserting a new suffix into T_i requires walking up the rightmost path in T_i. Each edge that is traversed ceases to be on the rightmost path in T_{i+1}, and thus is never traversed again. An edge in an intermediate tree T_i corresponds to a path in the suffix tree ST. When a new internal node is created along an edge in an intermediate tree, the edge is split into two edges, and the edge below the newly created internal node corresponds to an edge in the suffix tree. Once again, this edge ceases to be on the rightmost path and is never traversed again. The cost of creating an edge in an intermediate tree can be charged to the lowest edge on the corresponding path in the suffix tree. As each edge is charged once for creating and once for traversing, the total run-time of this procedure is $O(n)$.

Finally, the *Lcp* array itself can be constructed from the suffix array and the string in linear time [14]. Let R be an array of size n such that $R[i]$ contains the position in SA of $suff_i$. R can be constructed by a linear scan of SA in $O(n)$ time. The *Lcp* array is computed in n iterations. In iteration i of the algorithm, the longest common prefix between $suff_i$ and its respective right neighbor in the suffix array is computed. The array R facilitates locating an arbitrary suffix $suff_i$ and finding its right neighbor in the suffix array in constant time. Initially, the length of the longest common prefix between $suff_1$ and its suffix array neighbor is computed directly and recorded. Let $suff_j$ be the right neighbor of $suff_i$ in SA. Let l be the length of the longest common prefix between them. Suppose $l \geq 1$. As $suff_j$ is lexicographically greater than $suff_i$ and $s[i] = s[j]$, $suff_{j+1}$ is lexicographically greater than $suff_{i+1}$. The length of the longest common prefix between them is $l - 1$. It follows that the length of the longest common prefix between $suff_{i+1}$ and its right neighbor in the suffix array is $\geq l - 1$. To determine its correct length, the comparisons need only start from the l^{th} characters of the suffixes.

To prove that the run time of the above algorithm is linear, charge a comparison between the r^{th} character in suffix $suff_i$ and the corresponding character in its right neighbor suffix in SA to the position in the string of the r^{th} character of $suff_i$, i.e., $i + r - 1$. A comparison made in an iteration is termed successful if the characters compared are identical, contributing to the longest common prefix being computed. Because there is one failed comparison in each iteration, the total number of failed comparisons is $O(n)$. As for successful comparisons, each position in the string is charged only once for a successful comparison. Thus, the total number of comparisons over all iterations is linear in n.

In light of the above discussion, a suffix tree and a suffix array can be constructed from each other in linear time. Thus, a linear time construction algorithm for one can be used to construct the other in linear time. In the following subsections, we explore such algorithms. Each algorithm is interesting in its own right, and exploits interesting properties that could be useful in designing algorithms using suffix trees and suffix arrays.

1.2.2 Linear Time Construction of Suffix Trees

Let s be a string of length n including the termination character \$. Suffix tree construction algorithms start with an empty tree and iteratively insert suffixes while maintaining the property that each intermediate tree represents a compacted trie of the suffixes inserted so far. When all the suffixes are inserted, the resulting tree will be the suffix tree. Suffix links are typically used to speedup the insertion of suffixes. While the algorithms are identified by the names of their respective inventors, the exposition presented does not necessarily follow the original algorithms and we take the liberty to comprehensively present the material in a way we feel contributes to ease of understanding.

McCreight's Algorithm

McCreight's algorithm inserts suffixes in the order $suff_1, suff_2, \ldots, suff_n$. Let T_i denote the compacted trie after $suff_i$ is inserted. T_1 is the tree consisting of a single leaf labelled 1 that is connected to the root by an edge with label $s[1..n]$. In iteration i of the algorithm, $suff_i$ is inserted into tree T_{i-1} to form tree T_i. An easy way to do this is by starting from the root and following the unique path matching characters in $suff_i$ one by one until no more matches are possible. If the traversal does not end at an internal node, create an internal node there. Then, attach a leaf labelled i to this internal node and use the unmatched portion of $suff_i$ for the edge label. The run-time for inserting $suff_i$ is proportional to $|suff_i| = n - i + 1$. The total run-time of the algorithm is $\Sigma_{i=1}^{n}(n - i + 1) = O(n^2)$.

In order to achieve an $O(n)$ run-time, suffix links are used to significantly speedup the insertion of a new suffix. Suffix links are useful in the following way − Suppose we are inserting $suff_i$ in T_{i-1} and let v be an internal node in T_{i-1} on the path from root to leaf labelled $(i-1)$. Then, $\textit{path-label}(v) = c\alpha$ is a prefix of $suff_{i-1}$. Since v is an internal node, there must be another suffix $suff_j$ $(j < i - 1)$ that also has $c\alpha$ as prefix. Because $suff_{j+1}$ is previously inserted, there is already a path from the root in T_{i-1} labelled α. To insert $suff_i$ faster, if the end of path labelled α is quickly found, comparison of characters in $suff_i$ can start beyond the prefix α. This is where suffix links will be useful. The algorithm must also construct suffix links prior to using them.

LEMMA 1.1 Let v be an internal node in $ST(s)$ that is created in iteration $i - 1$. Let $\textit{path-label}(v) = c\alpha$, where c is a character and α is a (possibly empty) string. Then, either there exists an internal node u with $\textit{path-label}(u) = \alpha$ or it will be created in iteration i.

Proof As v is created when inserting $suff_{i-1}$ in T_{i-2}, there exists another suffix $suff_j$ $(j < i - 1)$ such that $lcp(suff_{i-1}, suff_j) = c\alpha$. It follows that $lcp(suff_i, suff_{j+1}) = \alpha$. The tree T_i already contains $suff_{j+1}$. When $suff_i$ is inserted during iteration i, internal node u with path-label α is created if it does not already exist.

The above lemma establishes that the suffix link of a newly created internal node can be established in the next iteration.

The following procedure is used when inserting $suff_i$ in T_{i-1}. Let v be the internal node to which $suff_{i-1}$ is attached as a leaf. If v is the root, insert $suff_i$ using character comparisons starting with the first character of $suff_i$. Otherwise, let $\textit{path-label}(v) = c\alpha$. If v has a suffix link from it, follow it to internal node u with path-label α. This allows skipping the comparison of the first $|\alpha|$ characters of $suff_i$. If v is newly created in iteration $i-1$, it would not have a suffix link yet. In that case, walk up to parent v' of v. Let β denote the label of the edge connecting v' and v. Let $u' = SL(v')$ unless v' is the root, in which case let u' be the root itself. It follows that $\textit{path-label}(u')$ is a prefix of $suff_i$. Furthermore, it is guaranteed that there is a path below u' that matches the next $|\beta|$ characters of $suff_i$. Traverse $|\beta|$ characters along this path and either find an internal node u or insert an internal node u if one does not already exist. In either case, set $SL(v) = u$. Continue by starting character comparisons skipping the first $|\alpha|$ characters of $suff_i$.

The above procedure requires two different types of traversals $-$ one in which it is known that there exists a path below that matches the next $|\beta|$ characters of $suff_i$ (type I), and the other in which it is unknown how many subsequent characters of $suff_i$ match a path below (type II). In the latter case, the comparison must proceed character by character until a mismatch occurs. In the former case, however, the traversal can be done by spending only $O(1)$ time per edge irrespective of the length of the edge label. At an internal node during such a traversal, the decision of which edge to follow next is made by comparing the next character of $suff_i$ with the first characters of the edge labels connecting the node to its children. However, once the edge is selected, the entire label or the remaining length of β must match, whichever is shorter. Thus, the traversal can be done in constant time per edge, and if the traversal stops within an edge label, the stopping position can also be determined in constant time.

The insertion procedure during iteration i can now be described as follows: Start with the internal node v to which $suff_{i-1}$ is attached as a leaf. If v has a suffix link, follow it and perform a type II traversal to insert $suff_i$. Otherwise, walk up to v's parent, take the suffix link from it unless it is the root, and perform a type I traversal to either find or create the node u which will be linked from v by a suffix link. Continue with a type II traversal below u to insert $suff_i$.

LEMMA 1.2 The total time spent in type I traversals over all iterations is $O(n)$.

Proof A type I traversal is performed by walking down along a path from root to a leaf in $O(1)$ time per edge. Each iteration consists of walking up at most one edge, following a suffix link, and then performing downward traversals (either type II or both type I and type II). Recall that if $SL(v) = u$, then $\textit{tree-depth}(u) \geq \textit{tree-depth}(v) - 1$. Thus, following a suffix link may reduce the depth in the tree by at most one. It follows that the operations that may cause moving to a higher level in the tree cause a decrease in depth of at most 2 per iteration. As both type I and type II traversals increase the depth in the tree and there are at most n levels in ST, the total number of edges traversed by type I traversals over all the iterations is bounded by $3n$.

LEMMA 1.3 The total time spent in type II traversals over all iterations is $O(n)$.

Proof In a type II traversal, a suffix of the string $suff_i$ is matched along a path in T_{i-1}

until there is a mismatch. When a mismatch occurs, an internal node is created if there does not exist one already. Then, the remaining part of $suff_i$ becomes the edge label connecting leaf labelled i to the internal node. Charge each successful comparison of a character in $suff_i$ to the corresponding character in the original string s. Note that a character that is charged with a successful comparison is never charged again as part of a type II traversal. Thus, the total time spent in type II traversals is $O(n)$.

The above lemmas prove that the total run-time of McCreight's algorithm is $O(n)$. McCreight's algorithm is suitable for constant sized alphabets. The dependence of the run-time and space for storing suffix trees on the size of the alphabet $|\Sigma|$ is as follows: A simple way to allocate space for internal nodes in a suffix tree is to allocate $|\Sigma|+1$ pointers for children, one for each distinct character with which an edge label may begin. With this approach, the edge label beginning with a given character, or whether an edge label exists with a given character, can be determined in $O(\log|\Sigma|)$ time. However, as all $|\Sigma|+1$ pointers are kept irrespective of how many children actually exist, the total space is $O(|\Sigma|n)$. If the tree is stored such that each internal node points only to its leftmost child and each node also points to its next sibling, if any, the space can be reduced to $O(n)$, irrespective of $|\Sigma|$. With this, searching for a child connected by an edge label with the appropriate character takes $O(|\Sigma|)$ time. Thus, McCreight's algorithm can be run in $O(n\log|\Sigma|)$ time using $O(n|\Sigma|)$ space, or in $O(n|\Sigma|)$ time using $O(n)$ space.

Generalized Suffix Trees

McCreight's algorithm can be easily adapted to build the generalized suffix tree for a set $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$ of strings of total length N in $O(N)$ time. A simple way to do this is to construct the string $S = s_1\$_1 s_2\$_2 \ldots s_k\$_k$, where each $\$_i$ is a unique string termination character that does not occur in any string in $\mathcal{S}$. Using McCreight's algorithm, $ST(S)$ can be computed in $O(N)$ time. This differs from $GST(\mathcal{S})$ in the following way: Consider a suffix $suff_j$ of string s_i in $GST(\mathcal{S})$. The corresponding suffix in $ST(S)$ is $s_i[j..|s_i|]\$_i s_{i+1}\$_{i+1} \ldots s_k\$_k$. Let v be the last internal node on the path from root to leaf representing this suffix in $ST(S)$. As each $\$_i$ is unique and *path-label*(v) must be a common prefix of at least two suffixes in S, *path-label*(v) must be a prefix of $s_i[j..|s_i|]$. Thus, by simply shortening the edge label below v to terminate at the end of the string s_i and attaching a common termination character \$ to it, the corresponding suffix in $GST(\mathcal{S})$ can be generated in $O(1)$ time. Additionally, all suffixes in $ST(S)$ that start with some $\$_i$ should be removed and replaced by a single suffix \$ in $GST(\mathcal{S})$. Note that the suffixes to be removed are all directly connected to the root in $ST(S)$, allowing easy $O(1)$ time removal per suffix. Thus, $GST(\mathcal{S})$ can be derived from $ST(S)$ in $O(N)$ time.

Instead of first constructing $ST(S)$ and shortening edge labels of edges connecting to leaves to construct $GST(\mathcal{S})$, the process can be integrated into the tree construction itself to directly compute $GST(\mathcal{S})$. When inserting the suffix of a string, directly set the edge label connecting to the newly created leaf to terminate at the end of the string, appended by \$. As each suffix that begins with $\$_i$ in $ST(S)$ is directly attached to the root, execution of McCreight's algorithm on S will always result in a downward traversal starting from the root when a suffix starting from the first character of a string is being inserted. Thus, we can simply start with an empty tree, insert all the suffixes of one string using McCreight's algorithm, insert all the suffixes of the next string, and continue this procedure until all strings are inserted. To insert the first suffix of a string, start by matching the unique path in the current tree that matches with a prefix of the string until no more matches are possible, and insert the suffix by branching at this point. To insert the remaining suffixes, continue as described in constructing the tree for one string.

This procedure immediately gives an algorithm to maintain the generalized suffix tree of a set of strings in the presence of insertions and deletions of strings. Insertion of a string is the same as executing McCreight's algorithm on the current tree, and takes time proportional to the length of the string being inserted. To delete a string, we must locate the leaves corresponding to all the suffixes of the string. By mimicking the process of inserting the string in GST using McCreight's algorithm, all the corresponding leaf nodes can be reached in time linear in the size of the string to be deleted. To delete a suffix, examine the corresponding leaf. If it is multiply labelled, it is enough to remove the label corresponding to the suffix. It it has only one label, the leaf and edge leading to it must be deleted. If the parent of the leaf is left with only one child after deletion, the parent and its two incident edges are deleted by connecting the surviving child directly to its grandparent with an edge labelled with the concatenation of the labels of the two edges deleted. As the adjustment at each leaf takes $O(1)$ time, the string can be deleted in time proportional to its length.

Suffix trees were invented by Weiner [23], who also presented the first linear time algorithm to construct them for a constant sized alphabet. McCreight's algorithm is a more space-economical linear time construction algorithm [19]. A linear time on-line construction algorithm for suffix trees is invented by Ukkonen [22]. In fact, our presentation of McCreight's algorithm also draws from ideas developed by Ukkonen. A unified view of these three suffix tree construction algorithms is studied by Giegerich and Kurtz [10]. Farach [7] presented the first linear time algorithm for strings over integer alphabets. The algorithm recursively constructs suffix trees for all odd and all even suffixes, respectively, and uses a clever strategy for merging them. The complexity of suffix tree construction algorithms for various types of alphabets is explored in [8].

1.2.3 Linear Time Construction of Suffix Arrays

Suffix arrays were proposed by Manber and Myers [18] as a space-efficient alternative to suffix trees. While suffix arrays can be deduced from suffix trees, which immediately implies any of the linear time suffix tree construction algorithms can be used for suffix arrays, it would not achieve the purpose of economy of space. Until recently, the fastest known direct construction algorithms for suffix arrays all required $O(n \log n)$ time, leaving a frustrating gap between asymptotically faster construction algorithms for suffix trees, and asymptotically slower construction algorithms for suffix arrays, despite the fact that suffix trees contain all the information in suffix arrays. This gap is successfully closed by a number of researchers in 2003, including Käräkkanen and Sanders [13], Kim *et al.* [15], and Ko and Aluru [16]. All three algorithms work for the case of integer alphabet. Given the simplicity and/or space efficiency of some of these algorithms, it is now preferable to construct suffix trees via the construction of suffix arrays.

Käräkkanen and Sanders' Algorithm

Käräkkanen and Sanders' algorithm is the simplest and most elegant algorithm to date to construct suffix arrays, and by implication suffix trees, in linear time. The algorithm also works for the case of an integer alphabet. Let s be a string of length n over the alphabet $\Sigma = \{1, 2, \ldots, n\}$. For convenience, assume n is a multiple of three and $s[n+1] = s[n+2] = 0$. The algorithm has the following steps:

1. Recursively sort the $\frac{2}{3}n$ suffixes $suff_i$ with $i \ mod \ 3 \neq 0$.
2. Sort the $\frac{1}{3}n$ suffixes $suff_i$ with $i \ mod \ 3 = 0$ using the result of step (1).
3. Merge the two sorted arrays.

To execute step (1), first perform a radix sort of the $\frac{2}{3}n$ triples $(s[i], s[i+1], s[i+2])$ for each $i\ mod\ 3 \neq 0$ and associate with each distinct triple its rank $\in \{1, 2, \ldots, \frac{2}{3}n\}$ in sorted order. If all triples are distinct, the suffixes are already sorted. Otherwise, let $suff'_i$ denote the string obtained by taking $suff_i$ and replacing each consecutive triplet with its corresponding rank. Create a new string s' by concatenating $suff'_1$ with $suff'_2$. Note that all $suff'_i$ with $i\ mod\ 3 = 1$ ($i\ mod\ 3 = 2$, respectively) are suffixes of $suff'_1$ ($suff'_2$, respectively). A lexicographic comparison of two suffixes in s' never crosses the boundary between $suff'_1$ and $suff'_2$ because the corresponding suffixes in the original string can be lexicographically distinguished. Thus, sorting s' recursively gives the sorted order of $suff_i$ with $i\ mod\ 3 \neq 0$.

Step (2) can be accomplished by performing a radix sort on tuples $(s[i], rank(suff_{i+1}))$ for all $i\ mod\ 3 = 0$, where $rank(suff_{i+1})$ denotes the rank of $suff_{i+1}$ in sorted order obtained in step (1).

Merging of the sorted arrays created in steps (1) and (2) is done in linear time, aided by the fact that the lexicographic order of a pair of suffixes, one from each array, can be determined in constant time. To compare $suff_i$ ($i\ mod\ 3 = 1$) with $suff_j$ ($i\ mod\ 3 = 0$), compare $s[i]$ with $s[j]$. If they are unequal, the answer is clear. If they are identical, the ranks of $suff_{i+1}$ and $suff_{j+1}$ in the sorted order obtained in step (1) determines the answer. To compare $suff_i$ ($i\ mod\ 3 = 2$) with $suff_j$ ($i\ mod\ 3 = 0$), compare the first two characters of the two suffixes. If they are both identical, the ranks of $suff_{i+2}$ and $suff_{j+2}$ in the sorted order obtained in step (1) determines the answer.

The run-time of this algorithm is given by the recurrence $T(n) = T\left(\lceil \frac{2n}{3} \rceil\right) + O(n)$, which results in $O(n)$ run-time. Note that the $\frac{2}{3} : \frac{1}{3}$ split is designed to make the merging step easy. A $\frac{1}{2} : \frac{1}{2}$ split does not allow easy merging because when comparing two suffixes for merging, no matter how many characters are compared, the remaining suffixes will not fall in the same sorted array, where ranking determines the result without need for further comparisons. Kim *et al.*'s linear time suffix array construction algorithm is based on a $\frac{1}{2} : \frac{1}{2}$ split, and the merging phase is handled in a clever way so as to run in linear time. This is much like Farach's algorithm for constructing suffix trees [7] by constructing suffix trees for even and odd positions separately and merging them. Both the above linear time suffix array construction algorithms partition the suffixes based on their starting positions in the string. A completely different way of partitioning suffixes based on the lexicographic ordering of a suffix with its right neighboring suffix in the string is used by Ko and Aluru to derive a linear time algorithm [16]. This reduces solving a problem of size n to that of solving a problem of size no more than $\lceil \frac{n}{2} \rceil$, while eliminating the complex merging step. The algorithm can be made to run in only $2n$ words plus $1.25n$ bits for strings over constant alphabet. Algorithmically, Käräkkanen and Sanders' algorithm is akin to mergesort and Ko and Aluru's algorithm is akin to quicksort. Algorithms for constructing suffix arrays in external memory are investigated by Crauser and Ferragina [5].

It may be more space efficient to construct a suffix tree by first constructing the corresponding suffix array, deriving the *Lcp* array from it, and using both to construct the suffix tree. For example, while all direct linear time suffix tree construction algorithms depend on constructing and using suffix links, these are completely avoided in the indirect approach. Furthermore, the resulting algorithms have an alphabet independent run-time of $O(n)$ while using only the $O(n)$ space representation of suffix trees. This should be contrasted with the $O(|\Sigma|n)$ run-time of either McCreight's or Ukkonen's algorithms.

1.2.4 Space Issues

Suffix trees and suffix arrays are space efficient in an asymptotic sense because the memory required grows linearly with input size. However, the actual space usage is of significant concern, especially for very large strings. For example, the human genome can be represented as a large string over the alphabet $\Sigma = \{A, C, G, T\}$ of length over 3×10^9. Because of linear dependence of space on the length of the string, the exact space requirement is easily characterized by specifying it in terms of the number of bytes per character. Depending on the number of bytes per character required, a data structure for the human genome may fit in main memory, may need a moderate sized disk, or might need a large amount of secondary storage. This has significant influence on the run-time of an application as access to secondary storage is considerably slower. It may also become impossible to run an application for large data sizes unless careful attention is paid to space efficiency.

Consider a naive implementation of suffix trees. For a string of length n, the tree has n leaves, at most $n - 1$ internal nodes, and at most $2n - 2$ edges. For simplicity, count the space required for each integer or a pointer to be one word, equal to 4 bytes on most current computers. For each leaf node, we may store a pointer to its parent, and store the starting index of the suffix represented by the leaf, for $2n$ words of storage. Storage for each internal node may consist of 4 pointers, one each for parent, leftmost child, right sibling and suffix link, respectively. This will require approximately $4n$ words of storage. Each edge label consists of a pair of integers, for a total of at most $4n$ words of storage. Putting this all together, a naive implementation of suffix trees takes $10n$ words or $40n$ bytes of storage.

Several techniques can be used to considerably reduce the naive space requirement of 40 bytes per character. Many applications of interest do not need to use suffix links. Similarly, a pointer to the parent may not be required for applications that use traversals down from the root. Even otherwise, note that a depth first search traversal of the suffix tree starting from the root can be conducted even in the absence of parent links, and this can be utilized in applications where a bottom-up traversal is needed. Another technique is to store the internal nodes of the tree in an array in the order of their first occurrence in a depth first search traversal. With this, the leftmost child of an internal node is found right next to it in the array, which removes the need to store a child pointer. Instead of storing the starting and ending positions of a substring corresponding to an edge label, an edge label can be stored with the starting position and length of the substring. The advantage of doing so is that the length of the edge label is likely to be small. Hence, one byte can be used to store edge labels with lengths < 255 and the number 255 can be used to denote edge labels with length at least 255. The actual values of such labels can be stored in an exceptions list, which is expected to be fairly small. Using several such techniques, the space required per character can be roughly cut in half to about 20 bytes [17].

A suffix array can be stored in just one word per character, or 4 bytes. Most applications using suffix arrays also need the *Lcp* array. Similar to the technique employed in storing edge labels on suffix trees, the entries in *Lcp* array can also be stored using one byte, with exceptions handled using an ordered exceptions list. Provided most of the *lcp* values fit in a byte, we only need 5 bytes per character, significantly smaller than what is required for suffix trees. Further space reduction can be achieved by the use of compressed suffix trees and suffix arrays and other data structures [9, 11]. However, this often comes at the expense of increased run-time complexity.

1.3 Applications

In this section, we present algorithms for several string problems using suffix trees and suffix arrays. While the same run-time bounds can be achieved for many interesting applications with either a suffix tree or a suffix array, there are others which involve a space vs. time trade off. Even in cases where the same run-time bound can be achieved, it is often easier to design the algorithm first for a suffix tree, and then think if the implementation can be done using a suffix array. For this reason, we largely concentrate on suffix trees. The reader interested in reading more on applications of suffix arrays is referred to [1, 2].

1.3.1 Pattern Matching

Given a pattern P and a text T, the pattern matching problem is to find all occurrences of P in T. Let $|P| = m$ and $|T| = n$. Typically, $n >> m$. Moreover, T remains fixed in many applications and the query is repeated for many different patterns. For example, T could be a text document and P could represent a word search. Or, T could be an entire database of DNA sequences and P denote a substring of a query sequence for homology (similarity) search. Thus, it is beneficial to preprocess the text T so that queries can be answered as efficiently as possible.

Pattern Matching using Suffix Trees

The pattern matching problem can be solved in optimal $O(m+k)$ time using $ST(T)$, where k is the number of occurrences of P in T. Suppose P occurs in T starting from position i. Then, P is a prefix of $suff_i$ in T. It follows that P matches the path from root to leaf labelled i in ST. This property results in the following simple algorithm: Start from the root of ST and follow the path matching characters in P, until P is completely matched or a mismatch occurs. If P is not fully matched, it does not occur in T. Otherwise, each leaf in the subtree below the matching position gives an occurrence of P. The positions can be enumerated by traversing the subtree in time proportional to the size of the subtree. As the number of leaves in the subtree is k, this takes $O(k)$ time. If only one occurrence is of interest, the suffix tree can be preprocessed in $O(n)$ time such that each internal node contains the label of one of the leaves in its subtree. Thus, the problem of whether P occurs in T or the problem of finding one occurrence can be answered in $O(m)$ time.

Pattern Matching using Suffix Arrays

Consider the problem of pattern matching when the suffix array of the text, $SA(T)$, is available. As before, we need to find all the suffixes that have P as a prefix. As SA is a lexicographically sorted order of the suffixes of T, all such suffixes will appear in consecutive positions in it. The sorted order in SA allows easy identification of these suffixes using binary search. Using a binary search, find the smallest index i in SA such that $suff_{SA[i]}$ contains P as a prefix, or determine that no such suffix is present. If no suffix is found, P does not occur in T. Otherwise, find the largest index $j(\geq i)$ such that $suff_{SA[j]}$ contains P as a prefix. All the elements in the range $SA[i..j]$ give the starting positions of the occurrences of P in T.

A binary search in SA takes $O(\log n)$ comparisons. In each comparison, P is compared with a suffix to determine their lexicographic order. This requires comparing at most $|P| = m$ characters. Thus, the run-time of this algorithm is $O(m \log n)$. Note that while this run-time is inferior to the run-time using suffix trees, the space required by this algorithm is only n words for SA apart from the space required to store the string. Note that the *Lcp* array is not required. Assuming 4 bytes per suffix array entry and one byte per character in the string, the total space required is only $5n$ bytes.

The run-time can be improved to $O(m + \log n)$, by using slightly more space and keeping track of appropriate *lcp* information. Consider an iteration of the binary search. Let $SA[L..R]$ denote the range in the suffix array where the binary search is focussed. To begin with, $L = 1$ and $R = n$. At the beginning of an iteration, the pattern P is known to be lexicographically greater than or equal to $suff_{SA[L]}$ and lexicographically smaller than or equal to $suff_{SA[R]}$. Let $M = \lceil \frac{L+R}{2} \rceil$. During the iteration, a lexicographic comparison between P and $suff_{SA[M]}$ is made. Depending on the result, the search range is narrowed to either $SA[L..M]$ or $SA[M..R]$. Assume that $l = |lcp(P, suff_{SA[L]})|$ and $r = |lcp(P, suff_{SA[R]})|$ are known at the beginning of the iteration. Also, assume that $|lcp(suff_{SA[L]}, suff_{SA[M]})|$ and $|lcp(suff_{SA[M]}, suff_{SA[R]})|$ are known. From these values, we wish to determine $|lcp(P, suff_{SA[M]})|$ for use in next iteration, and consequently determine the relative lexicographic order between P and $suff_{SA[M]}$. As SA is a lexicographically sorted array, P and $suff_{SA[M]}$ must agree on at least $min(l, r)$ characters. If l and r are equal, then comparison between P and $suff_{SA[M]}$ is done by starting from the $(l+1)^{th}$ character. If l and r are unequal, consider the case when $l > r$.

> Case I: $l < |lcp(suff_{SA[L]}, suff_{SA[M]})|$. In this case, P is lexicographically greater than $suff_{SA[M]}$ and $|lcp(P, suff_{SA[M]})| = |lcp(P, suff_{SA[L]})|$. Change the search range to $SA[M..R]$. No character comparisons are needed.
>
> Case II: $l > |lcp(suff_{SA[L]}, suff_{SA[M]})|$. In this case, P is lexicographically smaller than $suff_{SA[M]}$ and $|lcp(P, suff_{SA[M]})| = |lcp(Suff_{SA[L]}, suff_{SA[M]})|$. Change the search range to $SA[L..M]$. Again, no character comparisons are needed.
>
> Case III: $l = |lcp(suff_{SA[L]}, suff_{SA[M]})|$. In this case, P agrees with the first l characters of $suff_{SA[M]}$. Compare P and $suff_{SA[M]}$ starting from $(l+1)^{th}$ character to determine $|lcp(P, suff_{SA[M]})|$ and the relative lexicographic order of P and $suff_{SA[M]}$.

Similarly, the case when $r > l$ can be handled such that comparisons between P and $suff_{SA[M]}$, if at all needed, start from $(r+1)^{th}$ character. To start the execution of the algorithm, $lcp(P, suff_{SA[1]})$ and $lcp(P, suff_{SA[n]})$ are computed directly using at most $2|P|$ character comparisons. This ensures $|lcp(P, suff_{SA[L]})|$ and $|lcp(P, suff_{SA[R]})|$ are known at the beginning of the first iteration. This property is maintained for each iteration as L or R is shifted to M but $|lcp(P, suff_{SA[M]})|$ is computed. For now, assume that the required $|lcp(suff_{SA[L]}, suff_{SA[M]})|$ and $|lcp(suff_{SA[R]}, suff_{SA[M]})|$ values are available.

LEMMA 1.4 The total number of character comparisons made by the algorithm is $O(m + \log n)$.

Proof The algorithm makes at most $2m$ comparisons in determining the longest common prefixes between P and $suff_{SA[1]}$ and between P and $suff_{SA[n]}$. Classify the comparisons made in each iteration to determine the longest common prefix between P and $suff_{SA[M]}$ into *successful* and *failed* comparisons. A comparison is considered successful if it contributes the longest common prefix. There is at most one failed comparison per iteration, for a total of at most $\log n$ such comparisons over all iterations. As for successful comparisons, note that the comparisons start with $(max(l, r) + 1)^{th}$ character of P, and each successful comparison increases the value of $max(l, r)$ for next iteration. Thus, each character of P is involved only once in a successful comparison. The total number of character comparisons is at most $3m + \log n = O(m + \log n)$.

It remains to be described how the $|lcp(suff_{SA[L]}, suff_{SA[M]})|$ and $|lcp(suff_{SA[R]}, suff_{SA[M]})|$ values required in each iteration are computed. Suppose the Lcp array of T is known. For any $1 \leq i < j \leq n$,

$$|lcp(suff_{SA[i]}, suff_{SA[j]})| = min_{k=i}^{j-1} Lcp[k]$$

The *lcp* of two suffixes can be computed in time proportional to the distance between them in the suffix array. In order to find the *lcp* values required by the algorithm in constant time, consider the binary tree corresponding to all possible search intervals used by any execution of the binary search algorithm. The root of the tree denotes the interval $[1..n]$. If $[i..j]$ $(j - i \geq 2)$ is the interval at an internal node of the tree, its left child is given by $[i..\lceil \frac{i+j}{2} \rceil]$ and its right child is given by $[\lceil \frac{i+j}{2} \rceil..j]$. The execution of the binary search tree algorithm can be visualized as traversing a path in the binary tree from root to a leaf. If *lcp* value for each interval in the tree is precomputed and recorded, any required *lcp* value during the execution of the algorithm can be retrieved in constant time. The leaf level in the binary tree consists of intervals of the type $[i..i+1]$. The *lcp* values for these $n-1$ intervals is already given by the *Lcp* array. The *lcp* value corresponding to an interval at an internal node is given by the smaller of the *lcp* values at the children. Using a bottom-up traversal, the *lcp* values can be computed in $O(n)$ time. In addition to the *Lcp* array, $n-2$ additional *lcp* values are required to be stored. Assuming approximately 1 byte per *lcp* value, the algorithm requires approximately $2n$ bytes of additional space. As usual, *lcp* values larger than or equal to 255, if any, are stored in an exceptions list and the size of such list should be very small in practical applications.

Thus, pattern matching can be solved in $O(m \log n)$ time using $5n$ bytes of space, or in $O(m + \log n)$ time using $7n$ bytes of space. Abouelhoda *et al.* [2] reduce this time further to $O(m)$ time by mimicking the suffix tree algorithm on a suffix array with some auxiliary information. Using clever implementation techniques, the space is reduced to approximately $6n$ bytes. An interesting feature of their algorithm is that it can be used in other applications based on a top-down traversal of suffix tree.

1.3.2 Longest Common Substrings

Consider the problem of finding a longest substring common to two given strings s_1 of size m and s_2 of size n. To solve this problem, first construct the GST of strings s_1 and s_2. A longest substring common to s_1 and s_2 will be the path-label of an internal node with the greatest string depth in the suffix tree which has leaves labelled with suffixes from both the strings. Using a traversal of the GST, record the string-depth of each node, and mark each node if it has suffixes from both the strings. Find the largest string-depth of any marked node. Each marked internal node at that depth gives a longest common substring. The total run-time of this algorithm is $O(m + n)$.

The problem can also be solved by using the suffix tree of one of the strings and suffix links. Without loss of generality, suppose the suffix tree of s_2 is given. For each position i in s_1, we find the largest substring of s_1 starting at that position that is also a substring of s_2. For position 1, this is directly computed by matching $suff_1$ of s_1 starting from the root of the suffix tree until no more matches are possible. To determine the longest substring match from position 2, simply walk up to the first internal node, follow the suffix link, and walk down as done in McCreight's algorithm. A similar proof shows that this algorithm runs in $O(m + n)$ time.

Now consider solving the longest common substring problem using the GSA and *Lcp* array for strings s_1 and s_2. First, consider a one string variant of this problem $-$ that of computing the longest repeat in a string. This is given by the string depth of the deepest internal node in the corresponding suffix tree. All children of such a node must be leaves. Any consecutive pair of such leaves have the longest repeat as their longest common prefix.

Thus, each largest value in the *Lcp* array reveals a longest repeat in the string. The number of occurrences of a repeat is one more than the number of consecutive occurrences of the corresponding largest value in the *Lcp* array. Thus, all distinct longest repeats, and the number and positions of their occurrences can be determined by a linear scan of the *Lcp* array.

To solve the longest common substring problem, let v denote an internal node with the greatest string depth that contains a suffix from each of the strings. Because such a pair of suffixes need not be consecutive in the suffix array, it might appear that one has to look at nonconsecutive entries in the *Lcp* array. However, the subtree of any internal node that is a child of v can only consist of suffixes from one of the strings. Thus, there will be two consecutive suffixes in the subtree under v, one from each string. Therefore, it is enough to look at consecutive entries in the GSA. In a linear scan of the GSA and *Lcp* arrays, find the largest *lcp* value that corresponds to two consecutive suffixes, one from each string. This gives the length of a longest common substring. The starting positions of the suffixes reveals the positions in the strings where the longest common substring occurs. The algorithm runs in $O(m+n)$ time.

1.3.3 Text Compression

Compression of text data is useful for data transmission and for compact storage. A simple, not necessarily optimal, data compression method is the Ziv-Lempel compression [24, 25]. In this method, the text to be compressed is considered a large string, and a compact representation is obtained by identifying repeats in the string. A simple algorithm following this strategy is as follows: Let T denote the text to be compressed and let $|T| = n$. At some stage during the execution of the compression algorithm, suppose that the string $T[1..i-1]$ is already compressed. The compression is extended by finding the length l_i of a largest prefix of $suff_i$ that is a substring of $T[1..i-1]$. Two cases arise:

1. $l_i = 0$. In this case, a compressed representation of $T[1..i]$ is obtained by appending $T[i]$ to the compressed representation of $T[1..i-1]$.
2. $l_i > 0$. In this case, a compressed representation of $T[1..i+l_i-1]$ is obtained by appending (i, l_i) to the compressed representation of $T[1..i-1]$.

The algorithm is initiated by setting $T[1]$ to be the compressed representation of $T[1..1]$, and continuing the iterations until the entire string is compressed. For example, executing the above algorithm on the string *mississippi* yields the compressed string $mis(3,1)(2,3)(2,1)p$ $(9,1)(2,1)$. The decompression method for such a compressed string is immediate.

Suffix trees can be used to carry out the compression in $O(n)$ time [20]. They can be used in obtaining l_i, the length of the longest prefix of $suff_i$ that is a substring of the portion of the string already seen, $T[1..i-1]$. If j is the starting position of such a substring, then $T[j..j+l_i-1] = T[i..i+l_i-1]$ and $i \geq j + l_i$. It follows that $|lcp(suff_j, suff_i)| \geq l_i$. Let $v = lca(i,j)$, where i and j are leaves corresponding to $suff_i$ and $suff_j$, respectively. It follows that $T[i..i+l_i-1]$ is a prefix of $path\text{-}label(v)$. Consider the unique path from the root of $ST(T)$ that matches $T[i..i+l_i-1]$. Node v is an internal node in the subtree below, and hence j is a leaf in the subtree below. Thus, l_i is the largest number of characters along the path $T[i..n]$ such that $\exists$ leaf j in the subtree below with $j + l_i \leq i$. Note that any j in the subtree below that satisfies the property $j + l_i \leq i$ is acceptable. If such a j exists, the smallest leaf number in the subtree below certainly satisfies this property, and hence can be chosen as the starting position j.

This strategy results in the following algorithm for finding l_i: First, build the suffix tree of T. Using an appropriate linear time tree traversal method, record the string depth of each node and mark each internal node with the smallest leaf label in its subtree. Let $min(v)$ denote the smallest leaf label under internal node v. To find l_i, walk along the path $T[i..n]$ to identify two consecutive internal nodes u and v such that $min(u) + string\text{-}depth(u) < i$ and $min(v) + string\text{-}depth(v) \geq i$. If $min(v) + string\text{-}depth(u) > i$, then set $l_i = string\text{-}depth(u)$ and set the starting position to be $min(u)$. Otherwise, set $l_i = i - min(v)$ and set the starting position to be $min(v)$.

To obtain $O(n)$ run-time, it is enough to find l_i in $O(l_i)$ time as the next l_i characters of the string are compressed into an $O(1)$ space representation of an already seen substring. Therefore, it is enough to traverse the path matching $T[i..n]$ using individual character comparisons. However, as the path is guaranteed to exist, it can be traversed in $O(1)$ time per edge, irrespective of the length of the edge label.

1.3.4 String Containment

Given a set of strings $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$ of total length N, the string containment problem is to identify each string that is a substring of some other string. An example application could be that the strings represent DNA sequence fragments, and we wish to remove redundancy. This problem can be easily solved using suffix trees in $O(N)$ time. First, construct the $GST(\mathcal{S})$ in $O(N)$ time. To find if a string s_i is contained in another, locate the leaf labelled $(s_i, 1)$. If the label of the edge connecting the leaf to its parent is labelled with the string \$, s_i is contained in another string. Otherwise, it is not. This can be determined in $O(1)$ time per string.

1.3.5 Suffix-Prefix Overlaps

Suppose we are given a set of strings $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$ of total length N. The suffix-prefix overlap problem is to identify, for each pair of strings (s_i, s_j), the longest suffix of s_i that is a prefix of s_j. Suffix-prefix overlaps are useful in algorithms for finding the shortest common superstring of a given set of strings. They are also useful in applications such as genome assembly where significant suffix-prefix overlaps between pairs of fragments are used to assemble fragments into much larger sequences.

The suffix-prefix overlap problem can be solved using $GST(\mathcal{S})$ in optimal $O(N+k^2)$ time. Consider the longest suffix α of s_i that is a prefix of s_j. In $GST(\mathcal{S})$, α is an initial part of the path from the root to leaf labelled $(j, 1)$ that culminates in an internal node. A leaf that corresponds to a suffix from s_i should be a child of the internal node, with the edge label \$. Moreover, it must be the deepest internal node on the path from root to leaf $(j, 1)$ that has a suffix from s_i attached in this way. The length of the corresponding suffix-prefix overlap is given by the string depth of the internal node.

Let M be a $k \times k$ output matrix such that $M[i, j]$ should contain the length of the longest suffix of s_i that overlaps a prefix of s_j. The matrix is computed using a depth first search (DFS) traversal of $GST(\mathcal{S})$. The GST is preprocessed to record the string depth of every node. During the DFS traversal, k stacks $A_1, A_2, \ldots, A_k$ are maintained, one for each string. The top of the stack A_i contains the string depth of the deepest node along the current DFS path that is connected with edge label \$ to a leaf corresponding to a suffix from s_i. If no such node exists, the top of the stack contains zero. Each stack A_i is initialized by pushing zero onto an empty stack, and is maintained during the DFS as follows: When the DFS traversal visits a node v from its parent, check to see if v is attached to a leaf with edge label \$. If so, for each i such that string s_i contributes a suffix labelling the leaf, *push* $string\text{-}depth(v)$ on to stack A_i. The string depth of the current node can be easily maintained during the DFS traversal. When the DFS traversal leaves the node v to return back to its parent, again identify each i that has the above property and *pop* the top element from the corresponding stack A_i.

The output matrix M is built one column at a time. When the DFS traversal reaches a leaf labelled $(j, 1)$, the top of stack A_i contains the longest suffix of s_i that matches a prefix of s_j. Thus, column j of matrix M is obtained by setting $M[i, j]$ to the top element of stack S_i. To analyze the run-time of the algorithm, note that each *push* (similarly, *pop*) operation on a stack corresponds to a distinct suffix of one of the input strings. Thus, the total number of *push* and *pop* operations is bounded by $O(N)$. The matrix M is filled in $O(1)$ time per element, taking $O(k^2)$ time. Hence, all suffix-prefix overlaps can be identified in optimal $O(N + k^2)$ time.

1.4 Lowest Common Ancestors

Consider a string s and two of its suffixes $suff_i$ and $suff_j$. The longest common prefix of the two suffixes is given by the path label of their lowest common ancestor. If the string-depth of each node is recorded in it, the length of the longest common prefix can be retrieved from the lowest common ancestor. Thus, an algorithm to find the lowest common ancestors quickly can be used to determine longest common prefixes without a single character comparison. In this section, we describe how to preprocess the suffix tree in linear time and be able to answer lowest common ancestor queries in constant time [3].

Bender and Farach's lca ***algorithm***

Let T be a tree of n nodes. Without loss of generality, assume the nodes are numbered $1 \ldots n$. Let $lca(i, j)$ denote the lowest common ancestor of nodes i and j. Bender and Farach's algorithm performs a linear time preprocessing of the tree and can answer *lca* queries in constant time.

Let E be an Euler tour of the tree obtained by listing the nodes visited in a depth first search of T starting from the root. Let L be an array of level numbers such that $L[i]$ contains the tree-depth of the node $E[i]$. Both E and L contain $2n - 1$ elements and can be constructed by a depth first search of T in linear time. Let R be an array of size n such that $R[i]$ contains the index of the first occurrence of node i in E. Let $RMQ_A(i, j)$ denote the position of an occurrence of the smallest element in array A between indices i and j (inclusive). For nodes i and j, their lowest common ancestor is the node at the smallest tree-depth that is visited between an occurrence of i and an occurrence of j in the Euler tour. It follows that

$$lca(i, j) = E[RMQ_L(R[i], R[j])]$$

Thus, the problem of answering *lca* queries transforms into answering range minimum queries in arrays. Without loss of generality, we henceforth restrict our attention to answering range minimum queries in an array A of size n.

To answer range minimum queries in A, do the following preprocessing: Create $\lfloor \log n \rfloor + 1$ arrays $B_0, B_1, \ldots, B_{\lfloor \log n \rfloor}$ such that $B_j[i]$ contains $RMQ_A(i, i + 2^j)$, provided $i + 2^j \leq n$. B_0 can be computed directly from A in linear time. To compute $B_l[i]$, use $B_{l-1}[i]$ and $B_{l-1}[i + 2^{l-1}]$ to find $RMQ_A(i, i + 2^{l-1})$ and $RMQ_A(i + 2^{l-1}, i + 2^l)$, respectively. By comparing the elements in A at these locations, the smallest element in the range $A[i..i+2^l]$ can be determined in constant time. Using this method, all the $\lfloor \log n \rfloor + 1$ arrays are computed in $O(n \log n)$ time.

Given an arbitrary range minimum query $RMQ_A(i, j)$, let k be the largest integer such that $2^k \leq (j - i)$. Split the range $[i..j]$ into two overlapping ranges $[i..i + 2^k]$ and $[j - 2^k..j]$. Using $B_k[i]$ and $B_k[j - 2^k]$, a smallest element in each of these overlapping ranges can be located in constant time. This will allow determination of $RMQ_A(i,j)$ in constant time. To avoid a direct computation of k, the largest power of 2 that is smaller than or equal to each integer in the range $[1..n]$ can be precomputed and stored in $O(n)$ time. Putting all of this together, range minimum queries can be answered with $O(n \log n)$ preprocessing time and $O(1)$ query time.

The preprocessing time is reduced to $O(n)$ as follows: Divide the array A into $\frac{2n}{\log n}$ blocks of size $\frac{1}{2}\log n$ each. Preprocess each block such that for every pair (i,j) that falls within a block, $RMQ_A(i,j)$ can be answered directly. Form an array B of size $\frac{2n}{\log n}$ that contains the minimum element from each of the blocks in A, in the order of the blocks in A, and record the locations of the minimum in each block in another array C. An arbitrary query $RMQ_A(i,j)$ where i and j do not fall in the same block is answered as follows: Directly find the location of the minimum in the range from i to the end of the block containing it, and also in the range from the beginning of the block containing j to index j. All that remains is to find the location of the minimum in the range of blocks completely contained between i and j. This is done by the corresponding range minimum query in B and using C to find the location in A of the resulting smallest element. To answer range queries in B, B is preprocessed as outlined before. Because the size of B is only $O\left(\frac{n}{\log n}\right)$, preprocessing B takes $O\left(\frac{n}{\log n} \times \log \frac{n}{\log n}\right) = O(n)$ time and space.

It remains to be described how each of the blocks in A is preprocessed to answer range minimum queries that fall within a block. For each pair (i,j) of indices that fall in a block, the corresponding range minimum query is precomputed and stored. This requires computing $O(\log^2 n)$ values per block and can be done in $O(\log^2 n)$ time per block. The total run-time over all blocks is $\frac{2n}{\log n} \times O(\log^2 n) = O(n \log n)$, which is unacceptable. The run-time can be reduced for the special case where the array A contains level numbers of nodes visited in an Euler Tour, by exploiting its special properties. Note that the level numbers of consecutive entries differ by $+1$ or -1. Consider the $\frac{2n}{\log n}$ blocks of size $\frac{1}{2}\log n$. Normalize each block by subtracting the first element of the block from each element of the block. This does not affect the range minimum query. As the first element of each block is 0 and any other element differs from the previous one by $+1$ or -1, the number of distinct blocks is $2^{\frac{1}{2}\log n - 1} = \frac{1}{2}\sqrt{n}$. Direct preprocessing of the distinct blocks takes $\frac{1}{2}\sqrt{n} \times O(\log^2 n) = O(n)$ time. The mapping of each block to its corresponding distinct normalized block can be done in time proportional to the length of the block, taking $O(n)$ time over all blocks.

Putting it all together, a tree T of n nodes can be preprocessed in $O(n)$ time such that lca queries for any two nodes can be answered in constant time. We are interested in an application of this general algorithm to suffix trees. Consider a suffix tree for a string of length n. After linear time preprocessing, lca queries on the tree can be answered in constant time. For a given pair of suffixes in the string, the string-depth of their lowest common ancestor gives the length of their longest common prefix. Thus, the longest common prefix can be determined in constant time, without resorting to a single character comparison! This feature is exploited in many suffix tree algorithms.

1.5 Advanced Applications

1.5.1 Suffix Links from Lowest Common Ancestors

Suppose we are given a suffix tree and it is required to establish suffix links for each internal node. This may become necessary if the suffix tree creation algorithm does not construct suffix links but they are needed for an application of interest. For example, the suffix tree may be constructed via suffix arrays, completely avoiding the construction and use of suffix links for building the tree. The links can be easily established if the tree is preprocessed for lca queries.

Mark each internal node v of the suffix tree with a pair of leaves (i, j) such that leaves labelled i and j are in the subtrees of different children of v. The marking can be done in linear time by a bottom-up traversal of the tree. To find the suffix link from an internal node v (other than the root) marked with (i, j), note that $v = lca(i, j)$ and $lcp(suff_i, suff_j) = path\text{-}label(v)$. Let $path\text{-}label(v) = c\alpha$, where c is the first character and α is a string. To establish a suffix link from v, node u with path label α is needed. As $lcp(suff_{i+1}, suff_{j+1}) = \alpha$, node u is given by $lca(i + 1, j + 1)$, which can be determined in constant time. Thus, all suffix links can be determined in $O(n)$ time. This method trivially extends to the case of a generalized suffix tree.

1.5.2 Approximate Pattern Matching

The simpler version of approximate pattern matching problem is as follows: Given a pattern P ($|P| = m$) and a text T ($|T| = n$), find all substrings of length $|P|$ in T that match P with at most k mismatches. To solve this problem, first construct the GST of P and T. Preprocess the GST to record the string-depth of each node, and to answer lca queries in constant time. For each position i in T, we will determine if $T[i..i+m-1]$ matches P with at most k mismatches. First, use an lca query $lca((P, 1), (T, i))$ to find the largest substring from position i of T that matches a substring from position 1 and P. Suppose the length of this longest exact match is l. Thus, $P[1..l] = T[i..i + l - 1]$, and $P[l + 1] \neq T[i + l]$. Count this as a mismatch and continue by finding $lca((P, l + 2), (T, i + l + 1))$. This procedure is continued until either the end of P is reached or the number of mismatches crosses k. As each lca query takes constant time, the entire procedures takes $O(k)$ time. This is repeated for each position i in T for a total run-time of $O(kn)$.

Now, consider the more general problem of finding the substrings of T that can be derived from P by using at most k character insertions, deletions or substitutions. To solve this problem, we proceed as before by determining the possibility of such a match for every starting position i in T. Let $l = string\text{-}depth(lca((P, 1), (T, i)))$. At this stage, we consider three possibilities:

1. Substitution $-$ $P[l + 1]$ and $T[i + l]$ are considered a mismatch. Continue by finding $lca((P, l + 2), (T, i + l + 1))$.
2. Insertion $-$ $T[i+l]$ is considered an insertion in P after $P[l]$. Continue by finding $lca((P, l + 1), (T, i + l + 1))$.
3. Deletion $-$ $P[l + 1]$ is considered a deletion. Continue by finding $lca((P, l + 2), (T, i + l))$.

After each lca computation, we have three possibilities corresponding to substitution, insertion and deletion, respectively. All possibilities are enumerated to find if there is a sequence of k or less operations that will transform P into a substring starting from position i in T. This takes $O(3^k)$ time. Repeating this algorithm for each position i in T takes $O(3^k n)$ time.

The above algorithm always uses the longest exact match possible from a given pair of positions in P and T before considering the possibility of an insertion or deletion. To prove the correctness of this algorithm, we show that if there is an approximate match of P starting from position i in T that does not use such a longest exact match, then there exists another approximate match that uses only longest exact matches. Consider an approximate match that does not use longest exact matches. Consider the leftmost position j in P and the corresponding position $i+k$ in T where the longest exact match is violated. i.e., $P[j] = T[i+k]$ but this is not used as part of an exact match. Instead, an insertion or deletion is used. Suppose that an exact match of length r is used after the insertion or deletion. We can come up with a corresponding approximate match where the longest match is used and the insertion/deletion is taken after that. This will either keep the number of insertions/deletions the same or reduce the count. If the value of k is small, the above algorithms provide a quick and easy way to solve the approximate pattern matching problem. For sophisticated algorithms with better run-times, see [4, 21].

1.5.3 Maximal Palindromes

A string is called a palindrome if it reads the same forwards or backwards. A substring $s[i..j]$ of a string s is called a maximal palindrome of s, if $s[i..j]$ is a palindrome and $s[i-1] \neq s[j+1]$ (unless $i = 1$ or $j = n$). The maximal palindrome problem is to find all maximal palindromes of a string s.

For a palindrome of odd length, say $2k+1$, define the center of the palindrome to be the $(k+1)^{th}$ character. For a palindrome of even length, say $2k$, define the center to be the position between characters k and $k+1$ of the palindrome. In either case, the palindrome is said to be of radius k. Starting from the center, a palindrome is a string that reads the same in both directions. Observe that each maximal palindrome in a string must have a distinct center. As the number of possible centers for a string of length n is $2n-1$, the total number of maximal palindromes of a string is $2n-1$. All such palindromes can be identified in linear time using the following algorithm.

Let s^r denote the reverse of string s. Construct a GST of the strings s and s^r and preprocess the GST to record string depths of internal nodes and for answering lca queries. Now, consider a character $s[i]$ in the string. The maximal odd length palindrome centered at $s[i]$ is given by the length of the longest common prefix between $suff_{i+1}$ of s and $suff_{n-i+2}$ of s^r. This is easily computed as the string-depth of $lca((s, i+1), (s^r, n-i+2))$ in constant time. Similarly, the maximal even length palindrome centered between $s[i]$ and $s[i+1]$ is given by the length of the longest common prefix between $suff_{i+1}$ of s and $suff_{n-i+1}$ of s^r. This is computed as the string-depth of $lca((s, i+1), (s^r, n-i+1))$ in constant time.

These and many other applications involving strings can be solved efficiently using suffix trees and suffix arrays. A comprehensive treatise of suffix trees, suffix arrays and string algorithms can be found in the textbooks by Gusfield [12], and Crochemore and Rytter [6].

References

References

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *2nd Workshop on Algorithms in Bioinformatics*, pages 449–63, 2002.

[2] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *International Symposium on String Processing and Information Retrieval*, pages 31–43. IEEE, 2002.

[3] M.A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Theoretical Informatics Symposium*, pages 88–94, 2000.

[4] R. Cole and R. Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM Journal on Computing*, 31:1761–1782, 2002.

[5] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[6] M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific Publishing Company, Singapore, 2002.

[7] M. Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997.

[8] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47, 2000.

[9] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *41th Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

[10] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19:331–353, 1997.

[11] R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Symposium on the Theory of Computing*, pages 397–406. ACM, 2000.

[12] D. Gusfield. *Algorithms on Strings Trees and Sequences.* Cambridge University Press, New York, New York, 1997.

[13] J. Kärkkänen and P. Sanders. Simpler linear work suffix array construction. In *International Colloquium on Automata, Languages and Programming*, page to appear, 2003.

[14] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *12th Annual Symposium, Combinatorial Pattern Matching*, pages 181–92, 2001.

[15] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *14th Annual Symposium, Combinatorial Pattern Matching*, 2003.

[16] P. Ko and S. Aluru. Space-efficient linear-time construction of suffix arrays. In *14th Annual Symposium, Combinatorial Pattern Matching*, 2003.

[17] S. Kurtz. Reducing the space requirement of suffix trees. *Software - Practice and Experience*, 29(13):1149–1171, 1999.

[18] U. Manber and G. Myers. Suffix arrays: a new method for on-line search. *SIAM Journal on Computing*, 22:935–48, 1993.

[19] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–72, 1976.

[20] M. Rodeh, V.R. Pratt, and S. Even. A linear algorithm for data compression via string matching. *Journal of the ACM*, 28:16–24, 1981.

[21] E. Ukkonen. Approximate string-matching over suffix trees. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Combinatorial Pattern Matching, 4th Annual Symposium*, volume 684 of *Lecture Notes in Computer Science*, pages 228–242, Padova, Italy, 1993. Springer.

[22] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14:249–60, 1995.

[23] P. Weiner. Linear pattern matching algorithms. In *14th Symposium on Switching and Automata Theory*, pages 1–11. IEEE, 1973.

[24] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

[25] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.

Index