II

String Data Structures

5

Lookup Tables, Suffix Trees and Suffix Arrays

Srinivas Aluru
Iowa State University

Pang Ko
Iowa State University

5.1 Introduction

Fundamental string data structures, and their myriad applications in computational molecular biology are the focus of this part of the handbook. Sequence alignments and string data structures form the twin foundations for many applications in computational genomics. The utility of string data structures stems from the fact that at a basic level, various types of DNA and RNA sequences, and protein sequences can be modeled as strings — DNA as strings over the alphabet {A,C,G,T}, RNA as strings over the alphabet {A,C,G,U}, and proteins as strings over an alphabet of size 20 corresponding to the 20 amino acid residues. While simplistic, modeling of biological sequences as mere strings serves as a sufficient level of abstraction for a plethora of applications.

Given the large volume of sequence data that many computational biology applications must deal with, proper organization of the data to facilitate fast access is important to achieve desirable run-times. From this perspective, string data structures serve the same purpose for biological sequence data as binary search trees serve for ordered numeric data, and quadtrees serve for spatial data.

String data structures are ideal for uncovering exact matching patterns in sequences. Due to evolutionary mechanisms which alter biomolecular sequences, errors introduced by experimental processes, and many other factors that permit variations — such as the degeneracy of genetic code, protein sequences with some sequence similarity showing significant structural similarity — one is rarely interested in exact matches as an end in itself. Despite this, exact matches play a role because they are typically fast — requiring linear time as opposed to the quadratic time of alignment algorithms. As an example, consider the task of finding good local alignments between a query sequence and a database consisting of tens to hundreds of millions of sequences. It is computationally expensive to do as many pairwise local alignments. If we are interested in a pairwise alignment only if it exhibits significant homology, such an alignment should also contain regions of exact matches. For instance, if an aligning region of $100bp$ length contains at most 4 positions of difference, there should be an exact match of length at least 20 in this region. Exact matches can be used as a filter to eliminate large number of pairs that would not yield a good local alignment by performing alignments only on pairs that have an exact matching region larger than a determined threshold. It is in this spirit that many problems related to exact matches find applications in computational biology. String data structures are also useful when performing approximate matches where only a small number of differences are permitted.

In this chapter, we provide a detailed introduction to the three most frequently used string data structures in computational molecular biology — lookup tables, suffix trees and suffix arrays. The focus of this chapter will be on algorithms for constructing these data structures, which tend to be somewhat complex in the case of suffix trees and suffix arrays. We will also explore the relationships between these data structures. Chapter 6 provides several illustrations of biological applications where suffix trees play a central role. A number of new research results on solving biological applications using the more space efficient suffix array data structure, and its augmented variants, are presented in Chapter 7.

5.2 Lookup Tables

Lookup table is a simple data structure that records the positions of occurrences of substrings of a prespecified length in one or more strings. Lookup tables are used in a number of important bioinformatic tools including such popular programs as BLAST [1, 2] for database searches, and CAP3 [15] for genome assembly.

We use the following notation throughout the chapter: Let s be a string over the alphabet Σ. $|s|$ 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 w denote a prespecified length, sometimes referred to as window-size. The lookup table is an array LT of size $|\Sigma|^w$, corresponding to the $|\Sigma|^w$ possible substrings of length w. Let $f : \Sigma \rightarrow \{0, 1, \ldots |\Sigma| - 1\}$ be the one-to-one function such that $f(c) = j - 1$ if c is the j^{th} lexicographically smallest character. For the purpose of the lookup table, any arbitrary ordering of the characters can be taken as lexicographic ordering. Using f, a substring of length w can be treated as a w digit number in a base $|\Sigma|$ system, and converted to its decimal equivalent. We use the notation $F(\alpha)$ to denote the decimal number corresponding to a w-long substring α.

Each entry in the lookup table LT points to a linked list of specific locations within the input set of strings where the substring corresponding to the index for the entry occurs. The lookup table for the DNA sequence CATTATTAGGA with $w = 2$ is shown in Figure 5.1. It is constructed by using the mapping A$\rightarrow$ 0, C$\rightarrow$ 1, G$\rightarrow$ 2 and T$\rightarrow$ 3. The substring TA corresponds to the index $(30)_4 = 12$. The entry at index 12 indicates that the substring TA occurs in the string starting at positions 4 and 7.

Let s be a string of length n. It is easy to construct the lookup table for s in $O(|\Sigma|^w + n)$ time. First, create and initialize each entry to a null list in $O(|\Sigma|^w)$ time. Then, insert

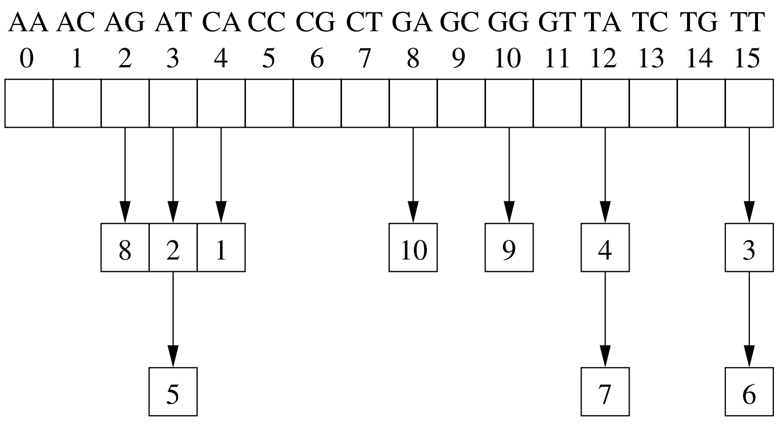


FIGURE 5.1: Lookup table for the DNA sequence CATTATTAGGA with $w = 2$. The table contains 16 entries, corresponding to the 16 different nucleotide sequences of length 2.

substrings one at a time. First compute $index = F(s[1..w])$ in $O(w)$ time. Insert position 1 in the linked list corresponding to $LT[index]$. Using the identity

$$F(s[k+1..k+w+1]) = \left(F(s[k..k+w] - f(s[k])|\Sigma|^{w-1}\right) \times |\Sigma| + f(s[k+w+1])$$

$F(s[k+1..k+w+1])$ can be computed from $F(s[k..k+w])$ in $O(1)$ time. As each starting position $1 \ldots n-w+1$ occurs in a linked list, the total size of all linked lists is $O(n)$ (typically $n >> w$). Thus, the size of the lookup table data structure is $O(|\Sigma|^w + n)$. The lookup table can be easily generalized to a set of strings. Let $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$ be a set of k strings of total length N. To create the corresponding lookup table, substrings from each of the strings are inserted in turn. A location in a linked list now consists of a pair giving the string number and the position of the substring within the string. The space and run-time required for constructing the lookup table is $O(|\Sigma|^w + N)$.

The size of the lookup table depends exponentially on the window-size w. To achieve space usage that is linear in the input data size, the value of w should be no greater than $\log_{|\Sigma|} N$. A window-size of 10 for DNA sequences assuming a 4-letter alphabet results in a lookup table with $2^{20} > 1$ million entries.

Lookup table is conceptually a very simple data structure to understand and implement. Once the lookup table for a database of strings is available, given a query string of length w, all occurrences of it in the database can be retrieved in $O(w + k)$ time, where k is the number of occurrences. The main problem with this data structure is its dependence on an arbitrary predefined substring length w. If the query string is of length $l > w$, the lookup table does not provide an efficient way of retrieving all occurrences of the query string in the database. Nevertheless, lookup tables are widely used in many bioinformatic programs due to their simplicity and ease of use.

5.3 Suffix Trees and Suffix Arrays

5.3.1 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'. Let $\mathit{suff}_i = s[i]s[i+1]\ldots s[|s|]$ 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, labeled $1 \ldots n$, one corresponding to each suffix of s.
2. Each internal node has at least 2 children.
3. Each edge in the tree is labeled with a substring of s.
4. The concatenation of edge labels from the root to the leaf labeled i is suff_i.
5. The labels of the edges connecting a node with its children start with different characters.

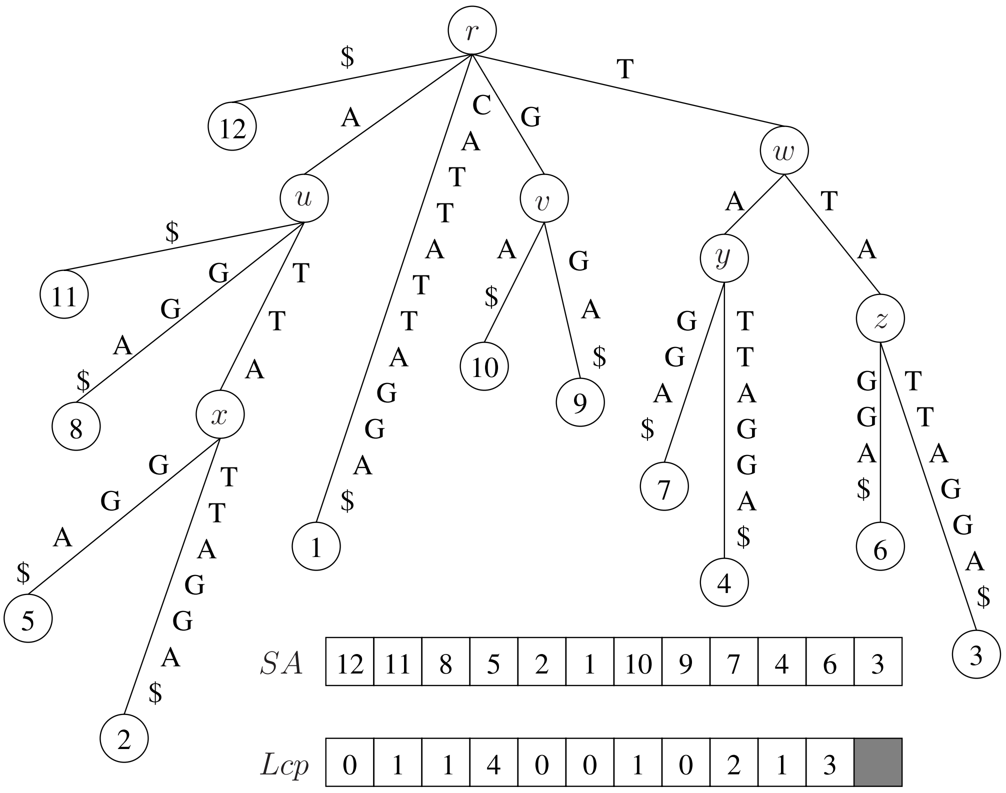
The paths from root to the leaves corresponding to the suffixes suff_i and suff_j coincide up to their longest common prefix, at which point they bifurcate. If a suffix of the string is a prefix of another longer suffix, the shorter suffix must end in an internal node instead of a leaf, as desired. It 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'. Throughout this chapter, '\$' is taken to be the lexicographically smallest character.

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 CATTATTAGGA is shown in Figure 5.2. 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 $\mathit{lcp}(\alpha, \beta)$ 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*. $\mathit{Lcp}[i]$ contains the length of the *lcp* between $\mathit{suff}_{SA[i]}$ and $\mathit{suff}_{SA[i+1]}$, i.e., $\mathit{Lcp}[i] = |\mathit{lcp}(\mathit{suff}_{SA[i]}, \mathit{suff}_{SA[i+1]})|$. 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 CATTATTAGGA are shown in Figure 5.2.

Let v be a node in the suffix tree. Let $\mathit{path\text{-}label}(v)$ denote the concatenation of edge labels along the path from root to node v. Let $\mathit{string\text{-}depth}(v)$ denote the length of $\mathit{path\text{-}label}(v)$. 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 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



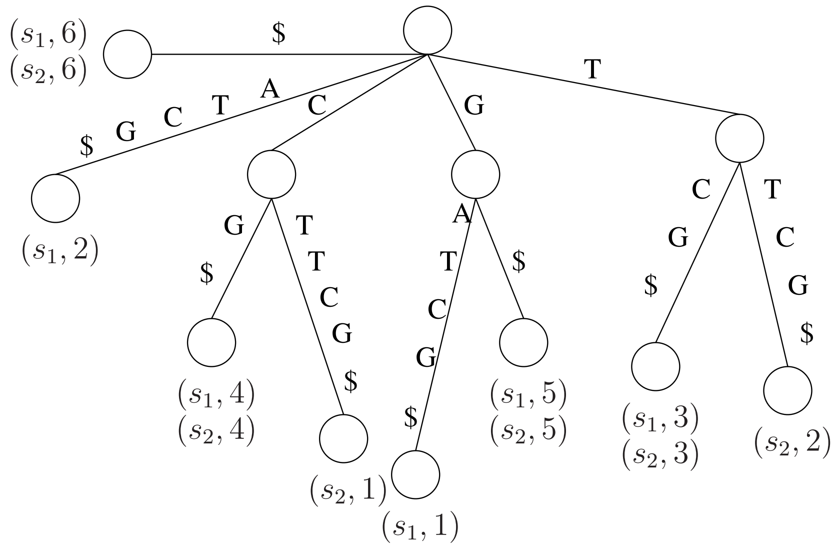
SA	12	11	8	5	2	1	10	9	7	4	6	3
Lcp	0	1	1	4	0	0	1	0	2	1	3	▨

FIGURE 5.2: Suffix tree, suffix array and *Lcp* array of the string CATTATTAGGA. The suffix links in the tree are given by $x \to z \to y \to u \to r$, $v \to r$, and $w \to r$.

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 labeled i, and along the path from u to leaf labeled $i+1$ 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



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

FIGURE 5.3: Generalized suffix tree and generalized suffix array of strings GATCG and CTTCG.

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 labeled, 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 GATCG and CTTCG are shown in Figure 5.3.

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 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. Applications of suffix trees in computational molecular biology are explored in great detail in the next chapter. 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. Such methods are explored in Chapter 7. In this chapter, we concentrate on linear time construction algorithms for suffix trees and suffix arrays.

5.3.2 Suffix Trees vs. Suffix Arrays

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. 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 leaves corresponding to 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 from 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 labeled $SA[1] = n$ connected to the root with an edge labeled $suff_{SA[1]} = \$$.

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 labeled $SA[i+1]$ 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 is illustrated in Figure 5.4. It works because no other suffix inserted so far shares a longer prefix with $suff_{SA[i+1]}$ than $suff_{SA[i]}$ does. 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 [20]. 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

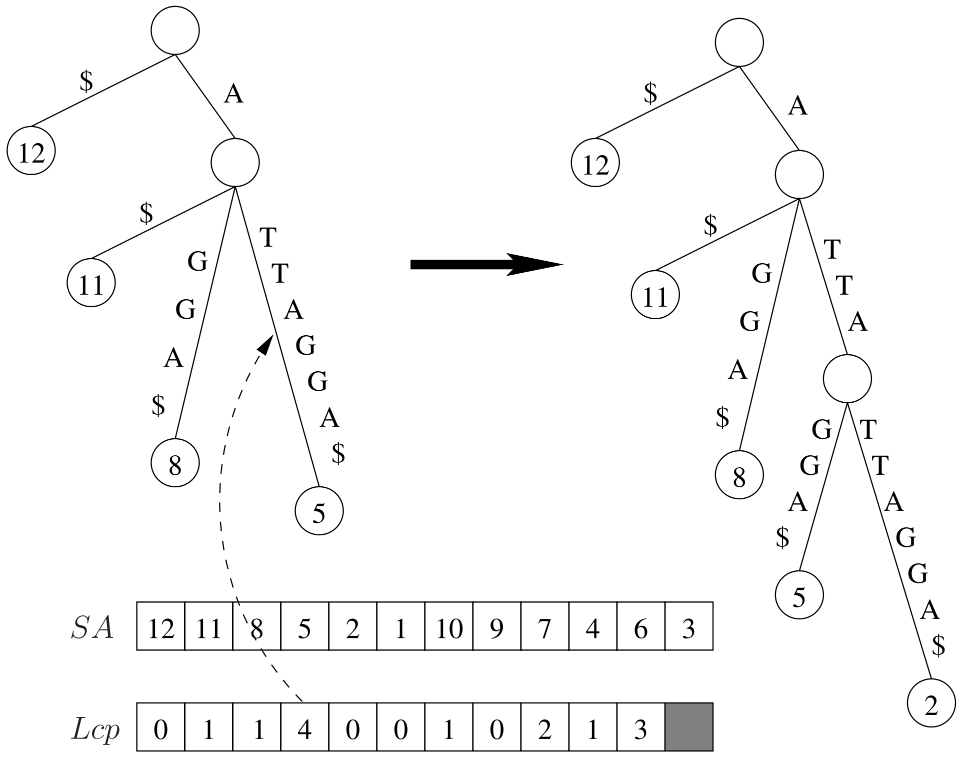


FIGURE 5.4: The construction of a suffix tree from the corresponding suffix and Lcp arrays. The example illustrates the insertion of $suff_2$ in the partial tree resulting from previously inserting the first four suffixes in the suffix array. The length of the lcp between the last inserted suffix and the new suffix gives the number of characters under the root and along the rightmost path at which the new leaf is inserted.

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 sections, 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.

In suffix tree and suffix array construction algorithms, three different types of alphabets are considered — a constant or fixed size alphabet ($|\Sigma|(1)$), 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 DNA and 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.

5.4 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.

5.4.1 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 labeled 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 labeled 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 labeled $(i-1)$. Then, $path\text{-}label(v) = c\alpha$ 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} labeled α. To insert $suff_i$ faster, if the end of path labeled 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 5.1 Let v be an internal node in $ST(s)$ that is created in iteration $i-1$. Let $path\text{-}label(v) = c\alpha$, where c is a character and α is a (possibly empty) string. Then, either there exists an internal node u with $path\text{-}label(u) = \alpha$ 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.

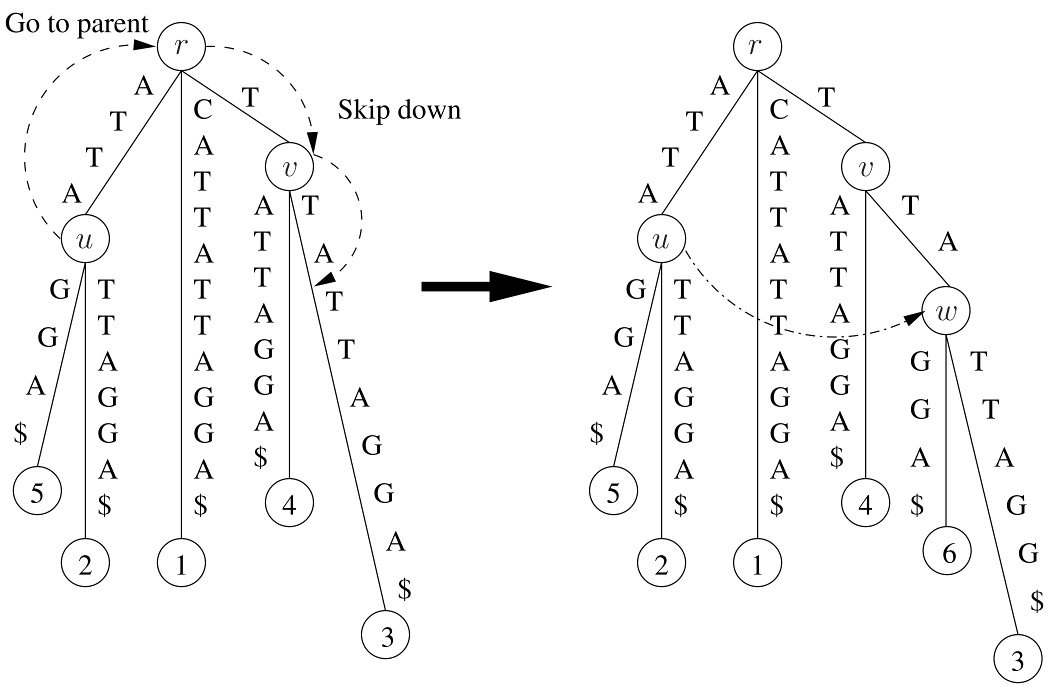


FIGURE 5.5: Illustration of suffix tree construction using McCreight's algorithm on the string CATTATTAGGA. The tree to the left is the compacted trie of suffixes 1 through 5. The process of inserting the next suffix $suff_6$ is shown in the figure.

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 $path\text{-}label(v) = c\alpha$. 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 $path\text{-}label(u')$ 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 $suff_i$.

This procedure is illustrated in Figure 5.5 for the string CATTATTAGGA. The tree to the left is the compacted trie after $suff_1$, $suff_2$, $suff_3$, $suff_4$ and $suff_5$ are inserted. To insert $suff_6$, consider the internal node u under which it is inserted as a leaf. Since u did not exist previously but was created during the insertion of $suff_5$, it does not have a suffix link yet. Therefore, walk up the 4 character edge to the parent of u to take a suffix link. However, the parent is the root r itself, and no suffix link is taken. To insert $suff_6$, walk down $4-1=3$ characters by only comparing one character per edge label and skipping edges at the rate of constant time per edge. At this position, create a new internal node w and set $SL(u) = w$. Continue to insert $suff_6$ below w.

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 5.2 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 5.3 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 labeled 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 $O(n)$.

5.4.2 Ukkonen's Algorithm

Ukkonen's suffix tree construction algorithm is also a linear time algorithm but with an important on-line property: The algorithm reads the input string one character at a time and maintains a suffix tree of the prefix of the string seen so far. As before, let s be a string of length n including the terminal '\$' character. The algorithm constructs a series of trees $T_1, T_2, \ldots, T_n$, where T_i is the suffix tree of $s[1..i]$. After constructing T_i, the algorithm reads $s[i+1]$ and updates T_i to create T_{i+1}. The total run-time spent by the time the algorithm constructs T_i is $O(i)$, even though the time spent in transitioning from one tree to the next is not necessarily constant.

When considering the string $s[1..i]$, a suffix of it may be repeated elsewhere in it because the unique terminal symbol is only at $s[n]$. Hence, a compacted trie of all suffixes of $s[1..i]$ may not have each suffix represented by a path that ends in a leaf. Therefore, we relax the definition of suffix trees by requiring that a downward path from the root corresponding to each suffix exist but not necessarily end in a leaf node. Such a tree is called *implicit suffix tree.* This would not pose any problem as the implicit suffix tree for $s[1..n]$ is the same as $ST(s)$ due to the terminal symbol $s[n] = \$$.

Ukkonen's algorithm employs a few additional ideas in conjunction with those already illustrated under McCreight's algorithm. Consider the prefix $s[1..i]$. We now use the notation $suff_k$ to denote the suffix starting from position k in the current string, i.e., $s[k..i]$. Let j be the position such that $suff_j = s[j..i]$ is the longest suffix of $s[1..i]$ that occurs elsewhere in it. Observe that the compacted trie of just suffixes $suff_1, suff_2, \ldots, suff_{j-1}$ is the same as the compacted trie of all suffixes $suff_1, suff_2, \ldots, suff_i$ of $s[1..i]$. In the implicit suffix tree of $s[1..i]$, the paths corresponding to first $j-1$ suffixes end in leaves and the paths corresponding to the remaining suffixes end otherwise.

Consider building T_{i+1} from T_i. Viewed naively, this requires extending all suffixes in T_i with the newly added character $s[i+1]$, and finally inserting a new suffix corresponding to the last character. Let j' be the position such that $s[j'..i+1]$ is the longest suffix of $s[1..i+1]$ that occurs elsewhere in it. Clearly, $j' \geq j$, where $s[j..i]$ is the longest suffix of $s[1..i]$ that occurs elsewhere in it. In creating T_{i+1}, the suffixes of $s[1..i+1]$ can be considered in three categories:

1. Suffixes $suff_i \ldots suff_{j-1}$: The corresponding suffixes in T_i already end in a leaf and they all need to be extended by the newly read character $s[i+1]$. Instead of explicitly doing this, this is implicitly achieved by assuming that all leaf labels (the labels of edges incident to leaves) end at the current end of the string.
2. Suffixes $suff_j \ldots suff_{j'-1}$: These suffixes are inserted in turn using ideas presented in McCreight's algorithm. This will be dealt in more detail later.
3. Suffixes $suff_{j'} \ldots suff_{i+1}$: We need not bother about these suffixes as the compacted trie of the suffixes in the two categories above automatically accounts for these suffixes also.

Observe that work is required only for inserting suffixes in category 2 above. The suffixes that are processed under category 2 in creating T_{i+1} from T_i, will become category 1 suffixes in subsequent tree constructions and are never worked on again. As the trees $T_1 \ldots T_n$ are constructed in Ukkonen's algorithm, each suffix is inserted as a category 2 suffix exactly once. Taken together, these suffix insertions can be thought of as similar to McCreight's suffix insertions. Essentially the same techniques will give linear run-time for these suffix insertions.

Consider T_i and let $s[j..i]$ be the longest suffix of $s[1..i]$ that is repeated elsewhere in it. This is actually realized while attempting to insert $s[j..i]$ while transitioning to T_i. The entire suffix $s[j..i]$ would be found within an already existing root to leaf path. This is a signal that T_i is already constructed. As we transition to T_{i+1}, the first suffix to insert is $s[j..i+1]$. Note that we are already at the end of the downward path from the root corresponding to $s[j..i]$. If the path can continue with $s[i+1]$, there are no category 2 insertions that need to be made. Otherwise, an internal node v is created at the end of $s[j..i]$ unless such a node v already exists, and a leaf attached to it using the edge label that start at $s[i+1]$ and ends at the current end of the string. Then, the next suffix is inserted as in McCreight's algorithm by taking suffix link from v if v was present beforehand, or by walking up to v's parent and taking suffix link from it if v was newly created and hence missing a suffix link from it. This process of inserting consecutive suffixes is carried out until one finds a suffix $s[j'..i+1]$ that is already represented, or the last suffix of the current string $s[i+1]$ is inserted.

The mechanism for moving from one suffix to the next is identical to the process described in McCreight's algorithm. Simply walk up from the current insertion point until the first node with a suffix link is reached, take the suffix link, walk down using type I traversal for the guaranteed portion of the match, and continue with type II traversal from that point on. Suffix links are also created during the execution of the algorithm as in McCreight's. Another way to view this algorithm is in terms of two shifting pointers j and i. The first pointer points to a suffix being inserted and the second pointer points to the current end of the string. If the suffix needs to be inserted, j is incremented by 1 to insert the next suffix. If the suffix is already found, i is incremented by 1 to switch to the next larger prefix of the string. As we advanced one of the pointers by 1 and the total length of the string is n, the number of steps before the two pointers sweep all the indices is at most $2n$. All of the suffix insertions together take $O(n)$ time, giving the algorithm a linear run-time.

McCreight's algorithm and Ukkonen's algorithm are 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 $O(1)$ 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 $O(n)$ time using $O(n|\Sigma|)$ space, or in $O(n|\Sigma|)$ time using $O(n)$ space. It is possible to obtain $O(n \log |\Sigma|)$ time with $O(n)$ space using an ordered list of pointers at each internal node. However, this is unlikely to be faster in practice, especially for small alphabet sizes such as for DNA and proteins.

5.4.3 Generalized Suffix Trees

The above linear time algorithms 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 a linear time 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 parent of the leaf representing this suffix in $ST(S)$. As each $\$_i$ is unique and $path\text{-}label(v)$ must be a common prefix of at least two suffixes in S, $path\text{-}label(v)$ 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 $GST(\mathcal{S})$. We will explain this in the context of using McCreight's algorithm. 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 labeled, 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 labeled 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 [29], 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 [26]. A linear time on-line construction algorithm for suffix trees was invented by Ukkonen [28]. 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 [6] 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 [7].

5.5 Linear Time Construction of Suffix Arrays

Suffix arrays were proposed by Manber and Myers [25] 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 Kärkkäinen and Sanders [19], Kim *et al.* [21], and Ko and Aluru [22, 23]. 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.

5.5.1 Kärkkäinen and Sanders' Algorithm

Kärkkäinen 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 $i \mathit{ mod } 3 \neq 0$.
2. Sort the $\frac{1}{3}n$ suffixes suff_i with $i \mathit{ mod } 3 = 0$ 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 $i \mathit{ mod } 3 \neq 0$ 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 $i \mathit{ mod } 3 = 1$ ($i \mathit{ mod } 3 = 2$, 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 $i \mathit{ mod } 3 \neq 0$.

Step (2) can be accomplished by performing a radix sort on tuples $(s[i], \mathit{rank}(\mathit{suff}_{i+1}))$ for all $i \mathit{ mod } 3 = 0$, where $\mathit{rank}(\mathit{suff}_{i+1})$ 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 ($i \mathit{ mod } 3 = 1$) with suff_j ($i \mathit{ mod } 3 = 0$), 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 ($i \mathit{ mod } 3 = 2$) with suff_j ($i \mathit{ mod } 3 = 0$), 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 [6] 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 more detailed account of Kärkkäinen and Sanders' Algorithm including pseudocode and an example suffix array construction, along with application of this algorithm to construct suffix arrays on disks can be found in Chapter 35.

s	M	I	S	S	I	S	S	I	P	P	I	$
Type	L	S	L	L	S	L	L	S	L	L	L	L/S
Pos	1	2	3	4	5	6	7	8	9	10	11	12

FIGURE 5.6: The string CATTATTAGGA$ and the types of its suffixes.

5.5.2 Ko and Aluru's Algorithm

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 [22, 23]. Consider a string s of size n over the alphabet $\Sigma = \{1 \dots n\}$. As before, we use '\$' to denote the last character of s, considered unique and lexicographically the smallest. For strings α and β, we use $\alpha \prec \beta$ to denote that α is lexicographically smaller than β.

A high level overview of the algorithm is as follows: The suffixes are classified into two types, S and L. Suffix suff_i is of type S if $\mathit{suff}_i \prec \mathit{suff}_{i+1}$, and is of type L if $\mathit{suff}_{i+1} \prec \mathit{suff}_i$. The last suffix suff_n is classified as both type S and type L. The positions of the type S suffixes partition the string into a set of substrings. We substitute each of these substrings by its rank among all the substrings and produce a new string s'. The suffixes of the new string are then recursively sorted. The suffix array of s' gives the lexicographic order of all type S suffixes. The lexicographic order of all suffixes can be deduced from this order.

The first step of the algorithm is to classify suffixes into types S and L. Consider suff_i $(i < n)$.

- If $s[i] < s[i+1]$, suff_i is of type S.
- If $s[i] > s[i+1]$, suff_i is of type L.
- If $s[i] = s[i+1]$, find the smallest $j > i$ such that $s[j] \neq s[i]$. If $s[j] > s[i]$, then $\mathit{suff}_i, \mathit{suff}_{i+1}, \dots, \mathit{suff}_{j-1}$ are of type S. Otherwise, they are all of type L.

Thus, all suffixes can be classified using a left to right scan of s in $O(n)$ time. The type of each suffix of the string CATTATTAGGA$ is shown in Figure 5.6.

LEMMA 5.4 A type S suffix is lexicographically greater than a type L suffix that begins with the same first character.

Proof Let suff_i be type S and suff_j be type L such that $s[i] = s[j] = c$. We can write $\mathit{suff}_i = c^k c_1 \alpha$ and $\mathit{suff}_j = c^l c_2 \beta$, where c^k and c^l denote the character c repeated for $k, l > 0$ times, respectively, $c_1 > c$, $c_2 < c$, and α and β are (possibly empty) strings.

Case 1: If $k < l$, c_1 is compared to a character c in c^l. Then $c_1 > c \Rightarrow \mathit{suff}_j \prec \mathit{suff}_i$.

Case 2: If $k > l$, c_2 is compared to a character c in c^k. Then $c > c_2 \Rightarrow \mathit{suff}_j \prec \mathit{suff}_i$.

Case 3: If $k = l$ then c_1 is compared to c_2. Since $c_1 > c$ and $c > c_2$, then $c_1 > c_2 \Rightarrow \mathit{suff}_j \prec \mathit{suff}_i$.

It follows that in the suffix array of s, among all suffixes that start with the same character, the type S suffixes appear after the type L suffixes.

Let A be an array containing all suffixes of s, not necessarily in sorted order. Let B be an array of all suffixes of type S, sorted in lexicographic order. Using B, the lexicographic

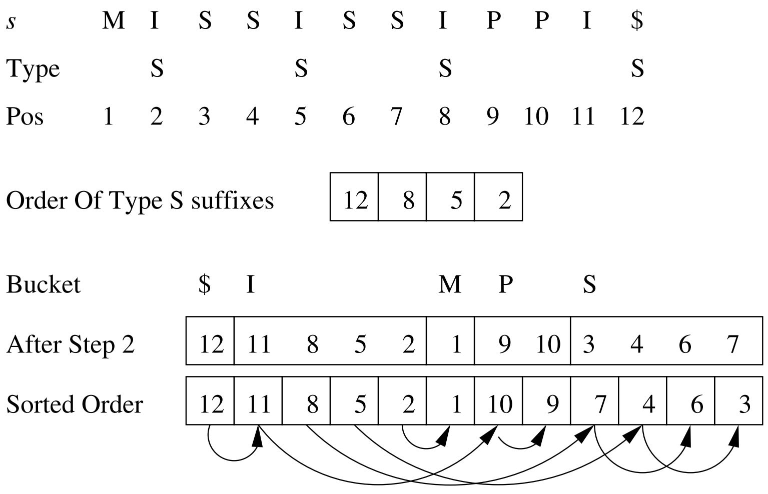

s M I S S I S S I P P I $
Type S S S S
Pos 1 2 3 4 5 6 7 8 9 10 11 12
Order Of Type S suffixes 12 8 5 2
Bucket $ I M P S
After Step 2 12 11 8 5 2 1 9 10 3 4 6 7
Sorted Order 12 11 8 5 2 1 10 9 7 4 6 3

FIGURE 5.7: Illustration of how to obtain the sorted order of all suffixes, from the sorted order of type S suffixes of the string CATTATTAGGA$.

sorted order of all suffixes of s can be computed as follows:

1. Bucket all suffixes of s according to their first character in array A in $O(n)$ time.
2. Scan B from right to left. For each suffix encountered in the scan, move the suffix to the current end of its bucket in A, and advance the current end by one position to the left. More specifically, the move of a suffix in array A to a new position should be taken as swapping the suffix with the suffix currently occupying the new position. After completion of the scan, all type S suffixes are in their correct positions in A. The time taken is $O(|B|)$, which is bounded by $O(n)$.
3. Scan A from left to right. For each entry $A[i]$, if $suff_{A[i]-1}$ is a type L suffix, move it to the current front of its bucket in A, and advance the front of the bucket by one. This takes $O(n)$ time. At the end of this step, A contains all suffixes of s in sorted order.

In Figure 5.7, the suffix pointed by the arrow is moved to the current front of its bucket when the scan reaches the suffix at the origin of the arrow. The following lemma proves the correctness of the procedure in Step 3.

LEMMA 5.5 In step 3, when the scan reaches $A[i]$, suffix $suff_{A[i]}$ is already in its sorted position in A.

Proof By induction on i. To begin with, the smallest suffix in s must be of type S and hence in its correct position $A[1]$. By inductive hypothesis, assume that $A[1], A[2], \ldots, A[i]$ are the first i suffixes in sorted order. We now show that when the scan reaches $A[i+1]$, then the suffix in it, i.e., $suff_{A[i+1]}$ is already in its sorted position. Suppose not. Then there exists a suffix referenced by $A[k]$ $(k > i+1)$ that should be in $A[i+1]$ in sorted order, i.e., $suff_{A[k]} \prec suff_{A[i+1]}$. As all type S suffixes are already in correct positions, both $suff_{A[k]}$ and $suff_{A[i+1]}$ must be of type L. Because A is bucketed by the first character of the suffixes prior to step 3, and a suffix is never moved out of its bucket, $\mathit{suff}_{A[k]}$ and $\mathit{suff}_{A[i+1]}$ must begin with the same character, say c. Let $\mathit{suff}_{A[i+1]} = c\alpha$ and $\mathit{suff}_{A[k]} = c\beta$. Since $\mathit{suff}_{A[k]}$ is type L, $\beta \prec \mathit{suff}_{A[k]}$. From $\mathit{suff}_{A[k]} \prec \mathit{suff}_{A[i+1]}$, $\beta \prec \alpha$. Since $\beta \prec \mathit{suff}_{A[k]}$, and the correct sorted position of $\mathit{suff}_{A[k]}$ is $A[i+1]$, β must occur in $A[1] \ldots A[i]$. Because $\beta \prec \alpha$, $\mathit{suff}_{A[k]}$ should have been moved to the current front of its bucket before $\mathit{suff}_{A[i+1]}$. Thus, $\mathit{suff}_{A[k]}$ can not occur to the right of $\mathit{suff}_{A[i+1]}$, a contradiction.

So far, we showed that if all type S suffixes are sorted, then the sorted position of all suffixes of s can be deduced in $O(n)$ time. A similar result can also be obtained by sorting all suffixes of type L: First bucket all suffixes of s based on their first characters into an array A. Scan the sorted order of type L suffixes from left to right and determine their correct positions in A by moving them to the current front of their respective buckets. Subsequently, scan A from right to left and when $A[i]$ is encountered, if $\mathit{suff}_{A[i]-1}$ is of type S, move it to the current end of its bucket. Since the suffix array of s can be deduced either from sorting all type S suffixes, or from sorting all type L suffixes, it is advantageous to choose the type which has fewer suffixes. Without loss of generality, assume there are fewer type S suffixes. We now show how to recursively sort these suffixes.

Define position i of s to be a type S position if suff_i is of type S, and similarly to be a type L position if suff_i is of type L. The substring $s[i..j]$ is called a type S substring if both i and j are type S positions, and every position in between is a type L position.

Our goal is to sort all type S suffixes in s. To do this we first sort all the type S substrings. The sorting generates buckets where all the substrings in a bucket are identical. The buckets are numbered using consecutive integers starting from 1. We then generate a new string s' as follows: Scan s from left to right and for each type S position in s, write the bucket number of the type S substring starting from that position. This string of bucket numbers forms s'. Observe that each type S suffix in s naturally corresponds to a suffix in the new string s'. In Lemma 5.6, we prove that sorting all type S suffixes of s is equivalent to sorting all suffixes of s'. We sort s' recursively.

We first show how to sort all the type S substrings in $O(n)$ time. Consider the array A, consisting of all suffixes of s bucketed according to their first characters. For each suffix suff_i, define its *S-distance* to be the distance from its starting position i to the nearest type S position to its left (excluding position i). If no type S position exists to the left, the *S-distance* is defined to be 0. Thus, for each suffix starting on or before the first type S position in s, its *S-distance* is 0. The type S substrings are sorted as follows (illustrated in Figure 5.8):

1. For each suffix in A, determine its *S-distance*. This is done by scanning s from left to right, keeping track of the distance from the current position to the nearest type S position to the left. While at position i, the *S-distance* of suff_i is known and this distance is recorded in array Dist. The *S-distance* of suff_i is stored in $\mathit{Dist}[i]$. Hence, the *S-distances* for all suffixes can be recorded in linear time.
2. Let m be the largest *S-distance*. Create m lists such that list j $(1 \le j \le m)$ contains all the suffixes with an *S-distance* of j, listed in the order in which they appear in array A. This can be done by scanning A from left to right in linear time, referring to $\mathit{Dist}[A[i]]$ to put $\mathit{suff}_{A[i]}$ in the correct list.
3. We now sort the type S substrings using the lists created above. The sorting is done by repeated bucketing using one character at a time. To begin with, the bucketing based on first character is determined by the order in which type S suffixes appear in array A. Suppose the type S substrings are bucketed according to their first $j-1$ characters. To extend this to j characters, we scan list j. For

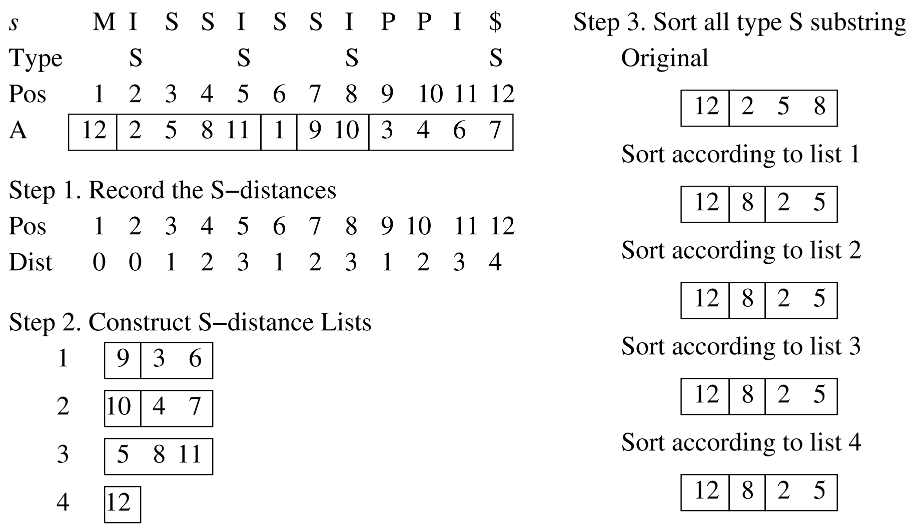

s M I S S I S S I P P I $
Type S S S S
Pos 1 2 3 4 5 6 7 8 9 10 11 12
A 12 2 5 8 11 1 9 10 3 4 6 7
Step 1. Record the S−distances
Pos 1 2 3 4 5 6 7 8 9 10 11 12
Dist 0 0 1 2 3 1 2 3 1 2 3 4
Step 2. Construct S−distance Lists
1 9 3 6
2 10 4 7
3 5 8 11
4 12
Step 3. Sort all type S substring
Original
12 2 5 8
Sort according to list 1
12 8 2 5
Sort according to list 2
12 8 2 5
Sort according to list 3
12 8 2 5
Sort according to list 4
12 8 2 5

FIGURE 5.8: Illustration of the sorting of type S substrings of the string CATTATTAGGA\$.

each suffix $suff_i$ encountered in the scan of a bucket of list j, move the type S substring starting at $s[i-j]$ to the current front of its bucket, then move the current front to the right by one. After a bucket of list j is scanned, new bucket boundaries need to be drawn between all the type S substrings that have been moved, and the type S substrings that have not been moved. Because the total size of all the lists is $O(n)$, the sorting of type S substrings only takes $O(n)$ time.

The sorting of type S substrings using the above algorithm respects lexicographic ordering of type S substrings, with the following important exception: If a type S substring is the prefix of another type S substring, the bucket number assigned to the shorter substring will be larger than the bucket number assigned to the larger substring. This anomaly is designed on purpose, and is exploited later in Lemma 5.6.

As mentioned before, we now construct a new string s' corresponding to all type S substrings in s. Each type S substring is replaced by its bucket number and s' is the sequence of bucket numbers in the order in which the type S substrings appear in s. Because every type S suffix in s starts with a type S substring, there is a natural one-to-one correspondence between type S suffixes of s and all suffixes of s'. Let $suff_i$ be a suffix of s and $suff'_{i'}$ be its corresponding suffix in s'. Note that $suff'_{i'}$ can be obtained from $suff_i$ by replacing every type S substring in $suff_i$ with its corresponding bucket number. Similarly, $suff_i$ can be obtained from $suff'_{i'}$ by replacing each bucket number with the corresponding substring and removing the duplicate instance of the common character shared by two consecutive type S substrings. This is because the last character of a type S substring is also the first character of the next type S substring along s.

LEMMA 5.6 Let $suff_i$ and $suff_j$ be two suffixes of s and let $suff'_{i'}$ and $suff'_{j'}$ be the corresponding suffixes of s'. Then, $suff_i \prec suff_j \Leftrightarrow suff'_{i'} \prec suff'_{j'}$.

Proof We first show that $suff'_{i'} \prec suff'_{j'} \Rightarrow suff_i \prec suff_j$. The prefixes of $suff_i$ and $suff_j$

corresponding to the longest common prefix of $suff'_{i'}$ and $suff'_{j'}$ must be identical. This is because if two bucket numbers are the same, then the corresponding substrings must be the same. Consider the leftmost position in which $suff'_{i'}$ and $suff'_{j'}$ differ. Such a position exists and the characters (bucket numbers) of $suff'_{i'}$ and $suff'_{j'}$ in that position determine which of $suff'_{i'}$ and $suff'_{j'}$ is lexicographically smaller. Let k be the bucket number in $suff'_{i'}$ and l be the bucket number in $suff'_{j'}$ at that position. Since $suff'_{i'} \prec suff'_{i'}$, it is clear that $k < l$. Let α be the substring corresponding to k and β be the substring corresponding to l. Note that α and β can be of different lengths, but α cannot be a proper prefix of β. This is because the bucket number corresponding to the prefix must be larger, but we know that $k < l$.

Case 1: β is not a prefix of α. In this case, $k < l \Rightarrow \alpha \prec \beta$, which implies $suff_i \prec suff_j$.

Case 2: β is a proper prefix of α. Let the last character of β be c. The corresponding position in s is a type S position. The position of the corresponding c in α must be a type L position. Since the two suffixes that begin at these positions start with the same character, the type L suffix must be lexicographically smaller then the type S suffix. Thus, $suff_i \prec suff_j$.

From the one-to-one correspondence between the suffixes of s' and the type S suffixes of s, it also follows that $suff_i \prec suff_j \Rightarrow suff'_{i'} \prec suff'_{j'}$.

From the above lemma, the sorted order of the suffixes of s' determines the sorted order of the type S suffixes of s. Hence, the problem of sorting the type S suffixes of s reduces to the problem of sorting all suffixes of s'. Note that the characters of s' are consecutive integers starting from 1. Hence the suffix sorting algorithm can be recursively applied to s'.

If s has fewer type L suffixes than type S suffixes, the type L suffixes are sorted using a similar procedure − Call $s[i..j]$ a type L substring if both i and j are type L positions, and every position in between is a type S position. Now sort all the type L substrings and construct the corresponding string s' obtained by replacing each type L substring with its bucket number. Sorting s' gives the sorted order of type L suffixes.

Thus, the problem of sorting the suffixes of a string s of length n can be reduced to the problem of sorting the suffixes of a string s' of size at most $\lceil \frac{n}{2} \rceil$, and $O(n)$ additional work. This leads to the recurrence $T(n) = T\left(\lceil \frac{n}{2} \rceil\right) + O(n)$, resulting in $O(n)$ run time. The algorithm can be made to run in only $2n$ words plus $1.25n$ bits for strings over constant alphabet [23]. Algorithmically, Kärkkäinen and Sanders' algorithm is akin to mergesort and Ko and Aluru's algorithm is akin to quicksort.

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.

5.6 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 [24].

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 [8, 11]. However, this often comes at the expense of increased run-time complexity.

5.7 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 [4].

5.7.1 Bender and Farach's *lca* algorithm

Let T be a tree of n nodes. Without loss of generality, assume the nodes are numbered $1 \dots n$. 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 $B_0, B_1, \dots, B_{\lfloor \log n \rfloor}$ 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.

5.7.2 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 labeled 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.

5.8 Conclusions

In this chapter, we focused on linear time construction algorithms for the three most important data structures used in computational biology — lookup tables, suffix trees, and suffix arrays. Some references for further study on this topic are provided in the References section. Compressed suffix arrays, which are briefly mentioned in Section 5.6 can be stored in $O(n)$ bits; Hon *et al.* provided the first linear time construction algorithm [14] for this data structure. In recent years, the size of biological databases has grown rapidly. This generated considerable interest in constructing and maintaining suffix trees and suffix arrays in secondary storage [3, 17, 27]. For a more detailed study of string data structures on secondary storage, the reader is referred to Chapter 35 of the handbook. Some biological applications are data and compute intensive, e.g. genome assembly of complex eukaryotic organisms and clustering large scale expressed sequence tag data. Parallelism is increasingly being used to solve such problems effectively (for example, see [16, 18]). Farach *et al.* [7], Futamura *et al.* [9] and Hariharan [13] have all studied the construction of suffix arrays or suffix trees in parallel environments.

The next two chapters explore in detail how suffix trees and suffix arrays are being used to support applications in computational biology. A comprehensive treatise of suffix trees, suffix arrays and string algorithms can be found in the textbooks by Gusfield [12], and Crochemore and Rytter [5].

Acknowledgements

This work was supported in part by the U.S. National Science Foundation under IIS-0430853.

References

[1] S.F. Altschul, W. Gish, W. Miller, and E.W. Myers *et al.* Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[2] S.F. Altschul, T.L. Madden, A.A. Schäffer, and J. Zhang *et al.* Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[3] S.J. Bedathur and J.R. Haritsa. Engineering a fast online persistent suffix tree construction. In *Proc. 20th International Conference on Data Engineering*, pages 720–731, 2004.

[4] M.A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 4th Latin American Theoretical Informatics Symposium*, pages 88–94, 2000.

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

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

[7] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.

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

[9] N. Futamura, S. Aluru, and S. Kurtz. Parallel suffix sorting. In *Proc. 9th International Conference on Advanced Computing and Communications*, pages 76–81, 2001.

[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 *Proc. 32nd annual ACM symposium on*

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] R. Hariharan. Optimal parallel suffix tree construction. *Journal of Computer and System Sciences*, 55(1):44–69, 1997.

[14] W.K. Hon, K. Sadakane, and W.K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 251–260, 2003.

[15] X. Huang and A. Madan. CAP3: A DNA sequence assembly program. *Genome Research*, 9(9):868–877, 1999.

[16] X. Huang, J Wang, S Aluru, and S.P. Yang *et al.* Pcap: a whole-genome assembly program. *Genome Research*, 13(9):2164–2170, 2003.

[17] E. Hunt, M.P. Atkinson, and R.W. Irving. Database indexing for large DNA and protein sequence collections. *The VLDB Journal*, 11(3):256–271, 2002.

[18] A. Kalyanaraman, S. Aluru, V. Brendel, and S. Kothari. Space and time efficient parallel algorithms and software for EST clustering. *IEEE Transactions on Parallel and Distributed Systems*, 14(12):1209–1221, 2003.

[19] J. Kärkkäinen and P. Sanders. Simpler linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming*, pages 943–955, 2003.

[20] T. Kasai, G. Lee, H. Arimura, and S. Arikawa *et al.* Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium, Combinatorial Pattern Matching*, pages 181–192, 2001.

[21] D.K. Kim, J.S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium, Combinatorial Pattern Matching*, pages 186–199, 2003.

[22] P. Ko and S. Aluru. Space-efficient linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium, Combinatorial Pattern Matching*, pages 200–210, 2003.

[23] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3:143–156, 2005.

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

[25] U. Manber and G. Myers. Suffix arrays: a new method for on-line search. *SIAM Journal on Computing*, 22:935–948, 1993.

[26] E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[27] S. Tata, R.A. Hankins, and J.M. Patel. Practical suffix tree construction. In *Proc. 13th International Conference on Very Large Data Bases*, pages 36–47, 2004.

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

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