4

Suffix trees and suffix arrays

The Burrows-Wheeler Transform has a very close relationship with suffix trees and suffix arrays — the array of indexes to the sorted array of substrings generated during the transform is essentially a suffix array, which in turn is a representation of the information in a suffix tree. As pointed out by Burrows and Wheeler in their original work (Burrows and Wheeler, 1994), the problem of sorting the rotated matrices is the major bottleneck in performing the transformation, and this is essentially an exercise in suffix sorting. This relationship between the BWT and suffix arrays and suffix trees also has important implications in the applications of the BWT, and in its relationship with other compression schemes, such as PPM. Analyzing the performance of the BWT is greatly simplified by an understanding of the construction and complexity of suffix trees and suffix arrays.

In this chapter we study suffix trees and suffix arrays in more detail. While this is motivated by their relationship with the BWT, suffix trees and suffix arrays have become important data structures in their own right, especially for problems in pattern matching, full-text indexing, compression, and other applications.

4.1 Suffix Trees

The suffix tree is a data structure used to represent the set of all suffixes of a string. It has a strong resemblance to the trie data structure (Knuth, 1973; Gonnet et al., 1992). However, unlike the trie, the suffix tree provides a more compact representation of the suffixes. While the size of the trie could be quadratic with respect to the length of the input string, the suffix tree provides a linear space representation of the suffixes.

The suffix tree is efficient in both time and space, and it is used for a variety of applications, such as in pattern matching (Ukkonen, 1993), multiple sequence alignment (Delcher et al., 1999; Kurtz et al., 2004), the identification of repetitions in genome-scale biological sequences (Bieganski et al., 1994;

Volfovsky et al., 2001), and in lossy image compression (Atallah et al., 1999). Apostolico (1985), Giancarlo (1995) and Grossi and Vitter (2005) discussed various applications of the suffix tree. More recently, *Wired* magazine reported the use of the suffix tree data structure in studying an age-old Inca Mystery (Cook, 2007) about the existence of written communication using knots and threads in old Peruvian culture.

Various algorithms have been developed for linear time construction of suffix trees (Weiner, 1973; McCreight, 1976; Ukkonen, 1995). In this section, we will discuss basic algorithms for constructing suffix trees, with an emphasis on newer approaches that lend themselves to direct construction of suffix arrays (without suffix trees). The book by Dan Gusfield (Gusfield, 1997) provides a comprehensive treatment of suffix trees, their construction, and applications.

4.1.1 Basic notations and definitions

We will continue to use $T = T[1 \dots n]$ as our input text. T is a string of length n, over an alphabet Σ. In this book, we assume that for a given string the symbol alphabet is fixed. Thus, we will treat the alphabet size as being constant, unless otherwise stated. Let $T = \alpha\beta\gamma$, for some strings α, β, and γ (α and γ could be empty). The string β is called a *substring* of T, α is called a *prefix* of T, while γ is called a *suffix* of T. The prefix α is called a proper prefix of T if $\alpha \neq T$. Similarly, the suffix γ is called a proper suffix of T if $\gamma \neq T$. We will also use $t_i = T[i]$ to denote the i-th symbol in T — both notations are used interchangeably. We use $T_i = T[i \dots n] = t_i t_{i+1} \dots t_n$ to denote the i-th suffix of T. Similarly, we use $T^i = T[1 \dots i] = t_1 t_2 \dots t_i$ to denote the i-th prefix of T.

For simplicity in constructing suffix trees, we usually ensure that no suffix of the string is a proper prefix of another suffix. This can be done by placing a sentinel symbol at the end of T, such that the sentinel does not appear anywhere else in T. In practice this is often achieved by simply appending a special symbol, say \$ to T, such that $\$ \notin \Sigma$. This constraint implies that each suffix of T will have its own unique leaf node in the suffix tree of T, since any two suffixes of T will eventually follow separate branches in the tree. Unless otherwise stated, we assume that this special symbol has been appended to each string.

Given a string T of length n, its suffix tree $\mathcal{T}_T$ is a rooted tree with n leaves, where the i-th leaf node corresponds to the i-th suffix T_i of T. Except for the root node and the leaf nodes, every node must have at least two descendant child nodes. Each edge in the suffix tree $\mathcal{T}_T$ represents a substring of T, and no two edges out of a node start with the same character. For a given edge, the *edge label* is simply the substring in T corresponding to the edge. We use l_i to denote the i-th leaf node. Then, l_i corresponds to T_i, the i-th suffix of T. Figure 4.1 shows the list of suffixes, the suffix trie, and the suffix tree for an example string $T =$ `acracca`\$.

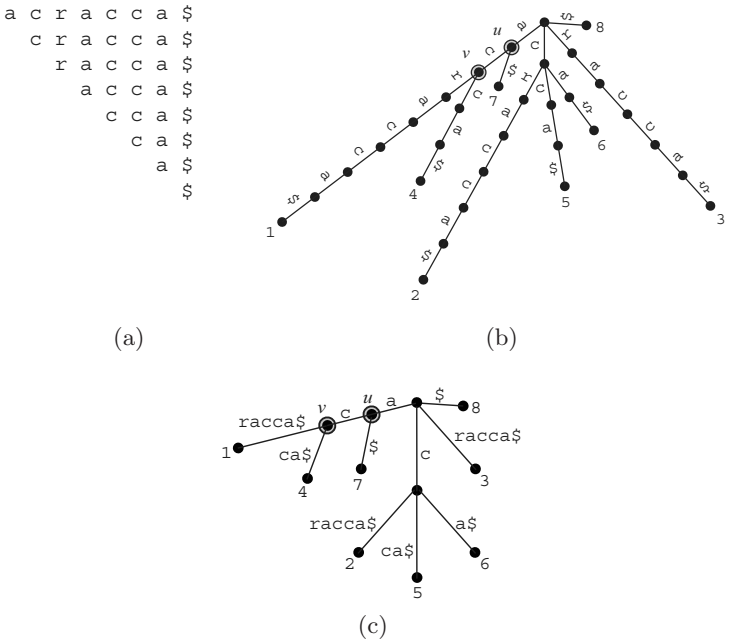


Fig. 4.1. Suffix structures for the string T = `acracca$`: (a) list of suffixes; (b) suffix trie, and (c) suffix tree. The number at each leaf node corresponds to the starting position of the corresponding suffix in the original string. See the text for explanation of the marked nodes u and v

For edge (u, v) between nodes u and v in $\mathcal{T}_T$, the edge label (denoted $label(u, v)$) is a non-empty substring of T. An edge is called a `t`-edge if its edge label starts with the symbol `t`. For a given node u in the suffix tree, its *path label*, $L(u)$ is defined as the label of the path from the root node to u. Since each edge represents a substring in T, $L(u)$ is essentially the string formed by the concatenation of the labels of the edges traversed in going from the root node to the given node, u. The *string depth* of node u, (also called its length) is simply $|L(u)|$, the number of characters in $L(u)$. Using the labels in the suffix tree in Figure 4.1, we will have $L(u) =$ `a`, $L(v) =$ `ac`, $label(u, v) =$ `c`, and the string depth of $v = 2$ (this also applies to the suffix trie). The number at each leaf node corresponds to the starting position of the corresponding suffix in the original string, T.

Properties of a suffix tree

Before discussing the construction of suffix trees, we summarize their basic properties. Given the string $T = T[1 \ldots n]\$$, of length n, but with the end of string symbol appended to give a sequence with a total length $n+1$, the suffix tree of the resulting string $T\$$ will have the following properties:

1. Exactly $n + 1$ leaf nodes;
2. At most n internal (or branching) nodes (the root node is considered an internal node);
3. Every *distinct* substring of T is encoded exactly once in the suffix tree. Each distinct substring is spelled out exactly once by traveling from the root node to some node u, such that $L(u)$ is the required substring. Note that the node u may be an implicit node (see Section 4.1.3).
4. No two edges out of a given node in the suffix tree start with the same symbol.
5. Every internal node has at least two outgoing edges.

Properties (1), (2), (4), and (5) imply that a suffix tree will have at most $2n + 1$ total nodes, and at most $2n$ edges;

The suffix tree is similar in spirit to the traditional trie data structure (Gonnet et al., 1992). The major difference is the notion of *path-label compression* and *edge-label compression* used in suffix trees. Thus, the suffix tree is generally viewed as a compacted suffix trie, as can be seen in the difference between the two trees in Figure 4.1. Path label compression and edge-label compression are critical for the linear time and linear space complexity of suffix tree construction. The notion of *path-label compression* is related to the requirement that every internal node, except the root node, must have at least two descendants, so we can remove all the internal nodes that have only one descendant in the suffix trie. The characters that make up the labels for the edges linking the removed nodes are concatenated in order, starting from the node nearest to the root. Thus, for the suffix tree, the edge labels can be substrings of the original sequence, rather than single symbols, which will reduce the number of nodes. For *edge-label compression*, rather than writing out the edge label explicitly, we use pointers into the original string to indicate the starting and ending positions of the substring corresponding to an edge label. This requires that the original string T must be available, in order to determine the edge labels explicitly. This reduces the potential $O(n^2)$ space required for suffix trees to $O(n)$, since the maximum number of edges will be $2n$. Figure 4.2 shows the suffix tree representation using edge-label compression for the example string used in Figure 4.1. For example, the edge label `c` is now replaced with the pair (2,2), while `ca$` is replaced with (6,8). Notice that under edge-label compression, the same edge (substring) could be represented with different pairs of pointers, for instance, the edge label `c` could have been represented with any of the pairs (2,2), (5,5), or (6,6). In practice, the pair representing the first occurrence of the substring in T, or the current occurrence, is generally used.

4.1.2 Construction of a suffix tree

Construction of the suffix tree for a string is not difficult. A simple algorithm that accomplishes this task for any given string is given in Algorithm 4.1. However, building the suffix tree *efficiently* is the key challenge.

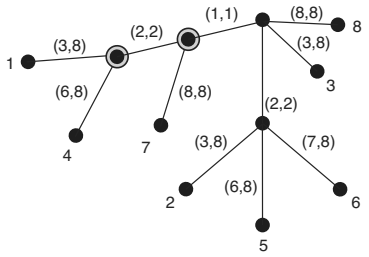


Fig. 4.2. Suffix tree for the string $T =$ `acracca$` using edge-label compression

---

SIMPLE-SUFFIX-TREE-ALGORITHM(T)
  Create the *root* node, with empty string
  for $i \leftarrow 1$ **to** n **do**
    Traverse current tree from the *root*
    Match symbols in the edge label one-by-one with symbols in
        the current suffix, T_i
    if a mismatch occurs **then**
      Split the edge at the position of mismatch to create a new
          node, if need be
      Insert suffix T_i into the suffix tree at the position of mismatch
    end if
  end for

Algorithm 4.1: Simple suffix tree construction algorithm

A step-by-step construction of a suffix tree using Algorithm 4.1 is shown in Figure 4.3 for the sample string $T =$ `acracca$`. First, the root node is created. Then the first suffix $T_1 = T =$ `acracca$` is inserted by attaching it to the root node. To insert the next suffix, $T_2 =$ `cracca$`, the algorithm traverses the edge, matching its edge label symbol-by-symbol. A mismatch occurs at the first position, hence suffix T_2 is attached to the root node. Suffix T_3 is inserted in the same way. To insert suffix $T_4 =$ `acca$`, we traverse from the root, on the `a`-edge until the mismatch with the third symbol `c`. Since the match occurred in the middle of an edge, we split the edge to create a new node at the mismatch position. Then the suffix is inserted by attaching a leaf node to the newly created node. The edge from the node to the leaf is then labeled with the remaining symbols in the suffix being inserted, starting with the mismatched character in the suffix. This process of matching, edge-splitting, and insertion continues until we reach the last suffix (the last symbol) of the string.

The above algorithm, although simple to implement, unfortunately requires construction time that is proportional to the square of n, the length of the string. This $O(n^2)$ complexity may not pose a problem for short sequences

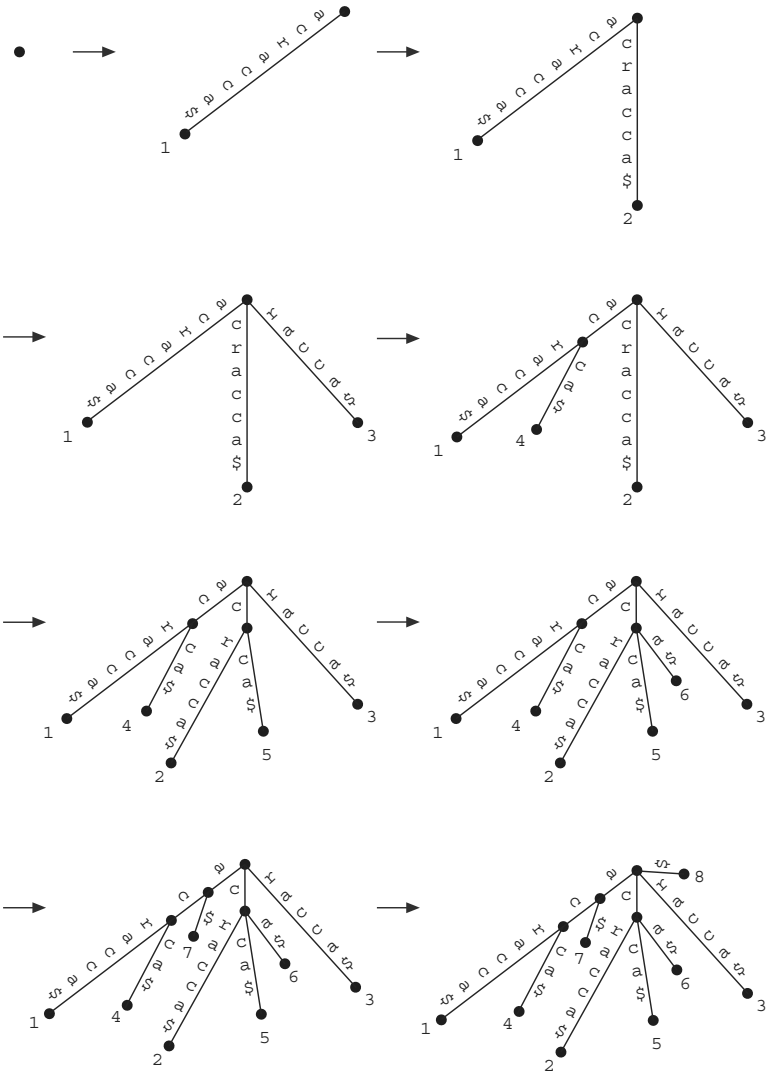


Fig. 4.3. Step-by-step construction of a suffix tree using the simple algorithm

with a few symbols. However, for most practical applications of suffix trees, such as in whole-genome sequence analysis with input strings that could have billions of symbols, more efficient approaches are required.

4.1.3 Ukkonen's suffix tree algorithm

Several methods have been proposed for constructing suffix trees in linear time and linear space. Ukkonen's algorithm (Ukkonen, 1995) is popular mainly because it is easier to understand and implement, and also because of its relatively small memory requirement. Our discussion of Ukkonen's algorithm mainly follows the description in Ukkonen (1995) and Gusfield (1997). Complete details can be found in the book or in the original paper by Ukkonen.

Ukkonen's algorithm is based on an observation on the relationship between the suffixes of substrings from the same string. Given a string $T = t_1 t_2 \dots t_n$, and the substring $T^i = t_1 t_2 \dots t_i$ (i.e. the i-th prefix of T), we observe that $T^i = T^{i-1} t_i$. Thus, the suffixes of T^i can be obtained from the suffixes of its longest proper prefix $T^{i-1} = t_1 t_2 \dots t_{i-1}$ by appending the new symbol t_i at the end of each suffix of T^{i-1}, and by adding the empty suffix. Therefore, the suffix tree of $T = t_1 t_2 \dots t_n = T^n$ can be constructed using a left to right scan, by first building the suffix tree for T^0, the empty string, expanding this to obtain the suffix tree for T^1, and continuing in this way until we build the suffix tree of $T = T^n$ from that of T^{n-1}. This incremental construction and the left-to-right scanning ability also imply that the method can be used to construct suffix trees online, that is, the algorithm can build the suffix tree piece-by-piece as a new symbol is received, without having the entire input string available at the beginning. To get the algorithm to work in linear time, Ukkonen used various clever ideas based on the properties of suffix trees.

Ukkonen's algorithm starts with an *implicit suffix tree*. Given a string T, and its suffix tree $\mathcal{T}_T$, its implicit suffix tree is obtained from $\mathcal{T}_T$, by removing the special symbol \$ from the edge labels, removing each node that has no label, and removing any node that has less than two children. The implicit suffix tree is constructed incrementally as described above. The last implicit suffix tree is then converted to a true suffix tree using a simple linear time traversal of the implicit suffix tree. The implicit suffix tree represents all the suffixes of a string, since each suffix is spelled out by some path from the root, whereby the path can end inside an edge. However, each suffix may not have a unique leaf node in the implicit suffix tree. If the last character in the string is unique (i.e. does not appear anywhere else in the string), then the implicit suffix tree and the true suffix tree will be the same. Figure 4.4 shows the implicit suffix tree for the example string $T =$ `acracca`; notice that nodes 7 and 8 are no longer shown explicitly.

Ukkonen's algorithm also made use of *suffix links*. The notion of suffix links is based on a well-known fact about suffix trees (Weiner, 1973; McCreight, 1976), namely, if there is an internal node u in $\mathcal{T}_T$ such that its path label $L(u) = a\alpha$ for some single character $a \in \Sigma$, and a (possibly empty) string $\alpha \in \Sigma^*$, then there is a node v in $\mathcal{T}_T$ such that $L(v) = \alpha$. A pointer from node u to node v is called a *suffix link*. If α is an empty string, then the pointer goes from u to the root node. In its simplest form, the suffix link from a given

leaf node points to the leaf node that corresponds to the longest proper suffix of the suffix represented by the leaf node.

We can now look at Ukkonen's algorithm in more detail. We will give a high level description of the algorithm in terms of its *phases* and the updates in each phase. We then describe the cases involved in performing the suffix updates at each step of the algorithm. Ukkonen's algorithm may be understood by first describing how we can use it to construct a suffix *trie*. We then describe the modifications to the basic algorithm to yield the suffix tree. To explain Ukkonen's algorithm, we use a somewhat longer example string, $T = \texttt{mississippi\$}$, as this captures all the update cases that will be encountered using the algorithm.

Suffix Trie Construction

Let $\mathcal{J}_T$ denote the suffix trie of the string T. The suffix trie for the string $T = t_1 t_2 \ldots t_n = T^n$ is constructed incrementally, from the suffix trie of its longest proper prefix $T^{n-1} = t_1 t_2 \ldots t_{n-1}$. In turn $\mathcal{J}_T^{n-1}$ is constructed from $\mathcal{J}_T^{n-2}$, and so on. Thus, Ukkonen's algorithm constructs the suffix tree in phases, whereby $\mathcal{J}_T^i$ is constructed in phase i of the algorithm. During phase 1 of the algorithm $\mathcal{J}_T^1$ is constructed from T^0, the empty suffix (empty string). Then $\mathcal{J}_T^2$ is constructed from $\mathcal{J}_T^1$ in phase 2, and so on, until finally, $\mathcal{J}_T^n$, the required suffix trie is constructed from $\mathcal{J}_T^{n-1}$ in phase n. Thus, the major question is how we construct $\mathcal{J}_T^i$, the suffix trie for the prefix $T^i = t_1 t_2 \ldots t_i$ from $\mathcal{J}_T^{i-1}$, the suffix trie of its longest proper prefix.

To construct $\mathcal{J}_T^i$ from $\mathcal{J}_T^{i-1}$, we need to append the next symbol t_i to the end of *each* suffix in $\mathcal{J}_T^{i-1}$, and add the empty suffix. Thus, we visit each suffix of $\mathcal{J}_T^{i-1}$, starting from its longest suffix, walking up to the shortest suffix (which corresponds to the empty string), updating the tree as we walk. Let node u in $\mathcal{J}_T^{i-1}$ be the current node during the walk. How the tree is updated depends on whether or not there is a t_i-edge that starts from node u.

The following update cases can occur:

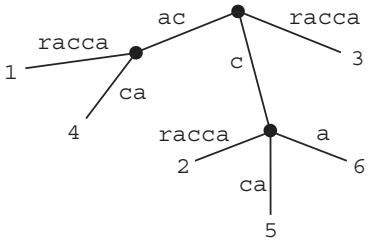


Fig. 4.4. Implicit suffix tree for the string $T = \texttt{acracca}$

Case A : Node u does *not* have a path that started with symbol t_i. This could happen in two ways:

1. Node u is a leaf node. Here, we update the tree by appending t_i to $L(u)$, the path label for node u. No new node is created. This can be seen in Figure 4.5 (going from $\mathcal{J}_T^1$ to $\mathcal{J}_T^2$), extending the leaf node with edge label `m` to `mi`; or in Figure 4.5 going from $\mathcal{J}_T^8$ to $\mathcal{J}_T^9$, extending the leftmost branch from `mississi` to `mississip`.
2. Node u is not a leaf node, but no edge from node u is a t_i-edge. This means that there is at least one edge that continues from node u, but no such edge starts with the symbol t_i. Here, we update the tree by creating a new leaf node starting from u, with edge label t_i. Examples can be seen in Figure 4.5 ($\mathcal{J}_T^1$), creating the leaf node from the root, with edge label `m`; or in Figure 4.5 ($\mathcal{J}_T^9$), creating the five leaf nodes, each with edge label `p`, resulting in parent nodes with ≥ 2 outgoing edges.

Case B : There is a t_i-edge emanating from u. This means that node u is not a leaf node, and the string $L(u)t_i$ has already been added to the suffix trie. Since the end of a suffix need not be explicit in an implicit suffix tree, no update is needed. This can be seen in Figure 4.5 ($\mathcal{J}_T^6$).

The root node, along with the set of nodes with old t_i-edges, the set of nodes with new t_i-edges created from $\mathcal{J}_T^{i-1}$, and the empty string represent $\mathcal{J}_T^i$, the suffix trie of T^i.

Suffix links and boundary paths

Determining which nodes in $\mathcal{J}_T^{i-1}$ should get new t_i edges is performed using suffix links. Given the definition of suffix links, each node in $\mathcal{J}_T^{i-1}$ that represents a suffix of T^{i-1} can be found by traversing a path of suffix links that start from the node with the largest depth (i.e. the node corresponding to the longest suffix $t_1 t_2 \ldots t_{i-1}$), and ends at the shortest suffix (the empty string). This path is called the *boundary path* of the suffix trie $\mathcal{J}_T^{i-1}$. During this traversal, for each node on the boundary path that does not have a t_i-edge, one is created. The new nodes are then connected with new suffix links that form a new path, starting from the suffix $t_1 t_2 \ldots t_{i-1}$. This new path thus corresponds to the boundary path for the new suffix trie, $\mathcal{J}_T^i$.

An important observation is that the traversal of the boundary path of $\mathcal{J}_T^{i-1}$ can be stopped whenever the first node, say u, is found such that node u already has a t_i-edge. This corresponds to Case B above. We can terminate the traversal at this point because if the string αt_i is a substring of T^{i-1}, for some string α, then each suffix of αt_i (which cannot be longer than $|\alpha t_i|$) must be a substring of T^{i-1}. So, these shorter substrings must have already been added to the suffix trie at some earlier phase. Thus, we can stop the update earlier, whenever Case B applies. This means that the algorithm creates a new node only when Case A2 applies.

Figure 4.5 shows the step-by-step construction of the suffix trie for the string $T =$ `mississippi` using Ukkonen's algorithm. The figure shows all 11 phases of the construction, and the suffix links created at each phase. In some cases, only the suffix links from the current and last phase are included, for ease of presentation. The suffix trie for $T = T^n$ is thus constructed by starting with the empty string, ($\mathcal{J}_T^0 = \epsilon$). This contains just the root node, and an *auxiliary* node (denoted as $\perp$ in the figures), and the links between them. The auxiliary node makes it possible to avoid an explicit distinction between the root and non-root nodes (i.e. between the empty suffix and non-empty suffixes) in the algorithm. Technically, the auxiliary node is defined as the inverse of any symbol in the alphabet. That is, for all $\sigma \in \Sigma$, $\sigma^{-1}\sigma = \epsilon$, the empty string. Therefore any σ-transition from $\perp$, the auxiliary node should lead to the root node, independent of σ. This is because the root node corresponds to ϵ, the empty suffix.

Figure 4.5 shows the algorithm repeating for each suffix T^i, for $i = 1, 2, \ldots, n$. The algorithm is optimal in the sense that it requires time that is linear with respect to its output size. However, this output size is proportional to the number of substrings in the original string, which could be quadratic with respect to the length of the string. Thus the running time will be in $O(n^2)$.

From suffix trie to suffix tree

The above suffix-trie construction algorithm can be turned into a linear-time algorithm for building suffix trees with some important modifications. The suffix tree of T provides a more compact representation of the suffixes of T by considering only a subset of the nodes in $\mathcal{J}_T$. This subset still includes all the suffixes of T.

Path compression and edge label compression. The first improvement is the use of path compression on the nodes of $\mathcal{J}_T$, which means that only internal nodes with at least two edges are allowed in the tree. The suffix tree $\mathcal{T}_T$ will thus contain only *explicit nodes* in the suffix trie $\mathcal{J}_T$, that is, the set of all branching nodes and all leaf nodes in $\mathcal{J}_T$. By definition, the root node is considered a branching node. The non-explicit nodes (called *implicit nodes*) are not stored in the suffix tree. The use of path compression, however, implies that for suffix tree construction, we may need to split an edge at an implicit node as the algorithm progresses (see below). A second improvement is the use of edge-label compression, whereby each edge label (which is a substring in T) is represented by a pair of pointers, say (p, q), where p and q point respectively to the start and end of the corresponding substring in T. That is, given an edge label, say α, p and q are chosen such that $\alpha = T[p \ldots q] = t_p t_{p+1} \ldots t_q$.

With this indexing, a copy of the original string T is needed as part of the representation of the suffix tree. This means that any substring can be accessed in constant time using its starting and ending pointers. An important advantage here is that we now need only a constant number of symbols (simply

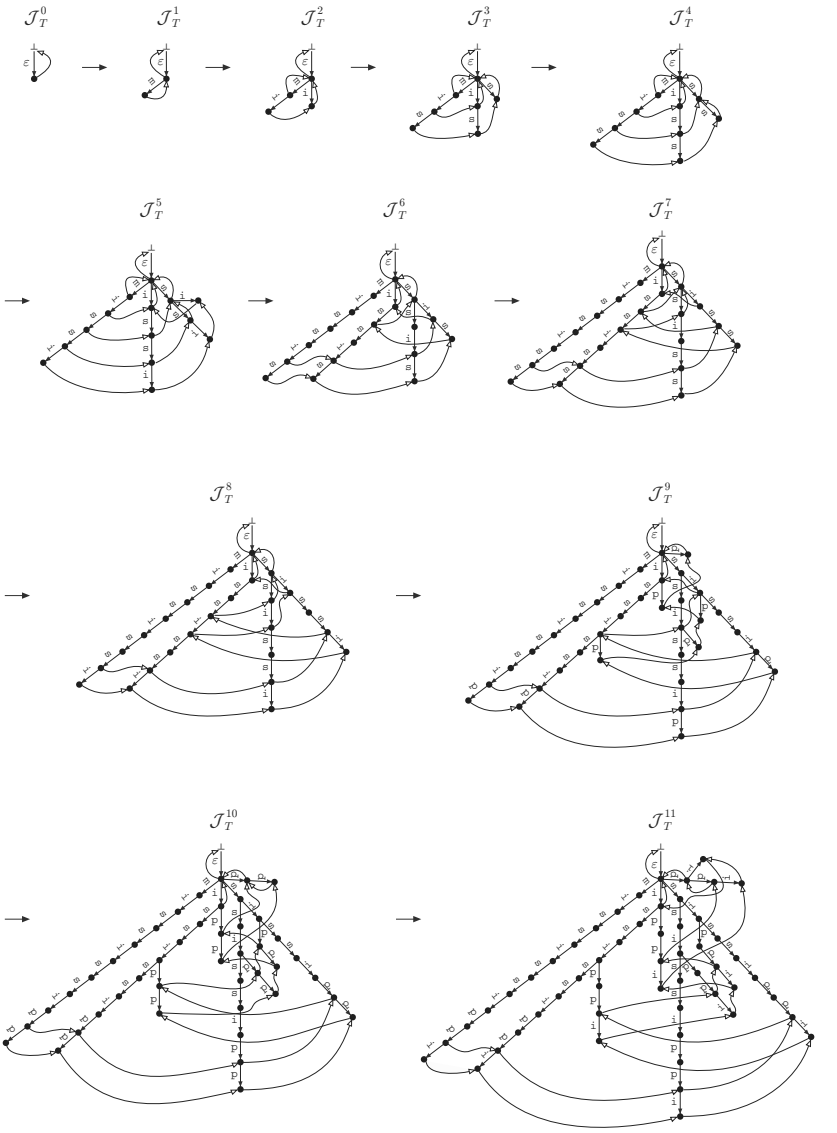


Fig. 4.5. Step-by-step construction of the suffix trie for the sequence $T =$ `mississippi` using Ukkonen's algorithm. The darker (filled) arrows represent transitions from one node to the other; the lighter (open) arrows are suffix links. For clarity, only the last two layers of suffix links are shown in some cases

two pointers) to represent each edge label, rather than requiring memory space that is proportional to the length of the label. Since we have one leaf for each non-empty suffix of T, we can have at most n leaves in the implicit suffix tree. Further, the implicit suffix tree of T can contain no more than $n-1$ branching nodes. The number of edges will be at most $2n-1$. This means that we can represent $\mathcal{T}_T$ using $O(n)$ space.

The suffix links are now defined only for branching nodes. However, at times imaginary suffix links, called *implicit suffix links*, are used. These links are defined between explicit nodes. This is mandated by the path compression, since some new explicit nodes may need to be created between two implicit nodes.

A leaf node stays a leaf node. Another important observation with the construction algorithm is that, once a leaf node is created and given a label, say l_i, then the node will remain a leaf node with the same label until the end of the algorithm. Thus, the node will never have a descendant node, rather, it can be extended only via character concatenation (update Case A1 as described previously). As it is sometimes put, 'once a leaf, always a leaf' (Gusfield, 1997). Recall that to extend $\mathcal{J}_T^{i-1}$ to $\mathcal{J}_T^i$, we need to extend the edges to each leaf node by appending the symbol t_i to the edge label. Ordinarily this concatenation will have to be performed for each leaf node at each phase of the algorithm. These extensions can, however, be avoided by a simple modification: whenever a new leaf node is created in phase i, we set its edge label (using the two pointers) to be (i, n), that is, to the substring $t_i, t_{i+1} \ldots t_n$ in T. Therefore explicit updates via character concatenations are no longer required at later phases of the algorithm. When an edge is split, the end point of the leaf node will still be fixed (at n, the end of the string), although the starting position in T may change.

Active point and end point. Ukkonen's algorithm also used the notions of an *active point* and *end point*. During the traversal of the boundary path at the current phase, the first non-leaf node encountered is called the *active point*, while the first node where Case B applies (i.e. the first node with a t_i transition) is called the *end point*. The active point and end point are key instruments used in achieving the linear time performance of the algorithm. Examples of active and end points are shown in Figure 4.6. We have repeated the suffix trie at some phases in Figure 4.5, but now with the active point and end point clearly marked at each phase. Since we no longer need to perform explicit updates via Case A1 (from the foregoing discussion), and we had already observed that we can stop traversal of the boundary path whenever Case B applies (see previous discussion under suffix links and boundary paths), it means that we need to perform updates only on the implicit and explicit nodes on the boundary path between the active point and the end point.

Another important observation here is that the end point in the current phase directly defines the active point in the next phase; the next node on the t_{i-1} edge from the endpoint in $\mathcal{J}_T^{i-1}$ becomes the active point in $\mathcal{J}_T^i$. This

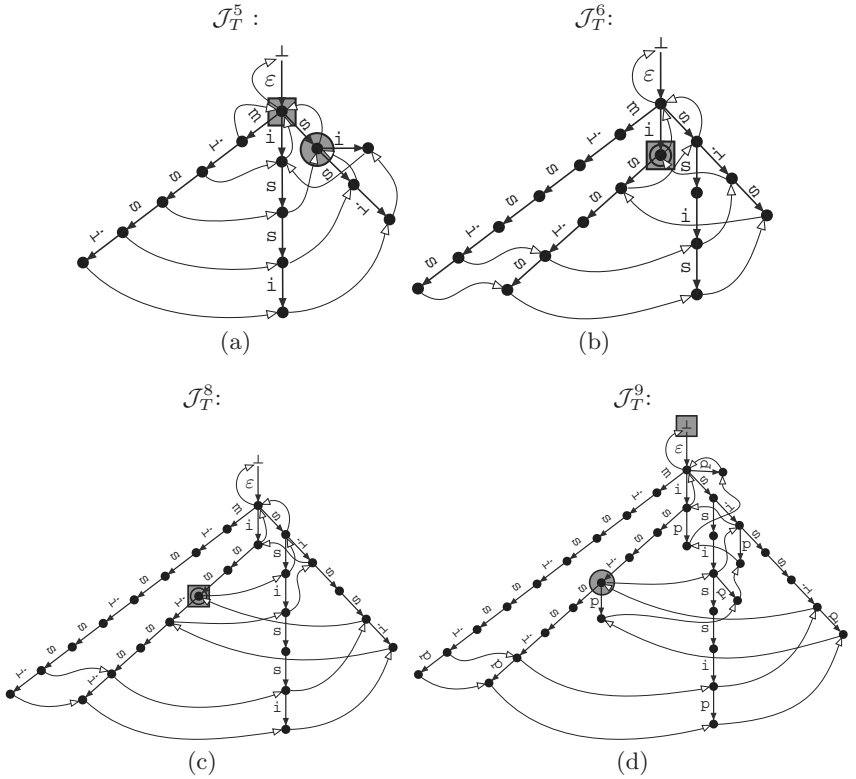


Fig. 4.6. Active points and end points for $\mathcal{J}_T^5$, $\mathcal{J}_T^6$ and $\mathcal{J}_T^8$, and $\mathcal{J}_T^9$ of the sequence $T = \texttt{mississippi}$. A circle on a node denotes an active point, a box denotes the end point, and a bulls-eye denotes a node that is both an active point and an end point

is very important, as it means that we can avoid a lot of computations that would otherwise be required to update the nodes in the suffix tree. By keeping record of the end point we already know the active point in the next phase. Since updates are performed only between active points and end points, it means that constructing $\mathcal{J}_T^i$ from $\mathcal{J}_T^{i-1}$ requires only one overlapping explicit update (one performed at the end point of $\mathcal{J}_T^{i-1}$, or at the active point of $\mathcal{J}_T^i$). This is shown in Figure 4.6.

Splitting a node. The final issue we need to discuss is how tree updates are performed when the active point is at an implicit node (i.e. between two explicit nodes). Updating the tree in this situation involves three steps: testing the node for possible splitting, splitting the node if necessary, and updating the new explicit node. First, we test to know if the edge needs to be split. If the continuing implicit edge (beyond the active point) is a t_i-edge, we do nothing (since we are working with implicit suffix trees). Otherwise we split the edge at the active point to obtain an explicit node at that point. Then

we attach a leaf node to the newly created node. Let edge (u, w) be the edge before splitting, with edge label (p, q). Let v be the newly created explicit node at the active point. Thus, u is the parent node of v, while w is now a child node of v. The label for the upper edge (that is, $label(u, v)$) is given as $(p, |L(v)|) = (p, p + |label(u, v)|)$, while the label for the edge that continues from v (i.e $label(v, w)$) is given as $(|L(v)| + 1, q)$. Then we apply the update cases at this newly created explicit node. This will thus create a new leaf node from the new explicit node (Case A2). The edge from the newly created explicit node to this leaf node will be a t_i-edge, with edge label $(l_i + |L(v)|, n)$.

Figure 4.7 shows an example of the process for node splitting in Ukkonen's algorithm. The figure shows the suffix tree in phase 5 of the algorithm before node splitting, just after node splitting but before update of the newly created node, and after updating the node. Notice the new suffix link created after node splitting.

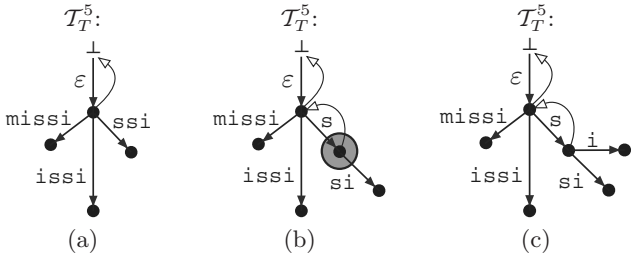


Fig. 4.7. Node splitting during suffix tree construction: (a) tree before node splitting; (b) tree just after node splitting, before node updating; (c) tree after node update. The marked node indicates the node where the splitting occurred

The complete construction phases for building the suffix tree for the running example $T =$ `mississippi` are shown in Figure 4.8. We have used the substrings as the edge labels for clarity of presentation. In practice, the pair of their starting and ending positions in T will be used.

4.1.4 From implicit suffix tree to true suffix tree

The above algorithm constructs the implicit suffix tree for a given string T. The final step is to convert this implicit suffix tree to a true suffix tree. This can be performed by appending a special end-of-string symbol, $\$, (\$ \notin \Sigma)$ to T, and letting the algorithm continue with this new symbol. Alternatively, we can traverse the boundary path from the leaf node of $\mathcal{T}_T^n$ up to the root, and make all nodes on the path explicit. Either way, each leaf node in the resulting tree will correspond to one unique suffix in the original string T. The resulting tree is the true suffix tree. The above requires only $O(n)$ time to traverse the

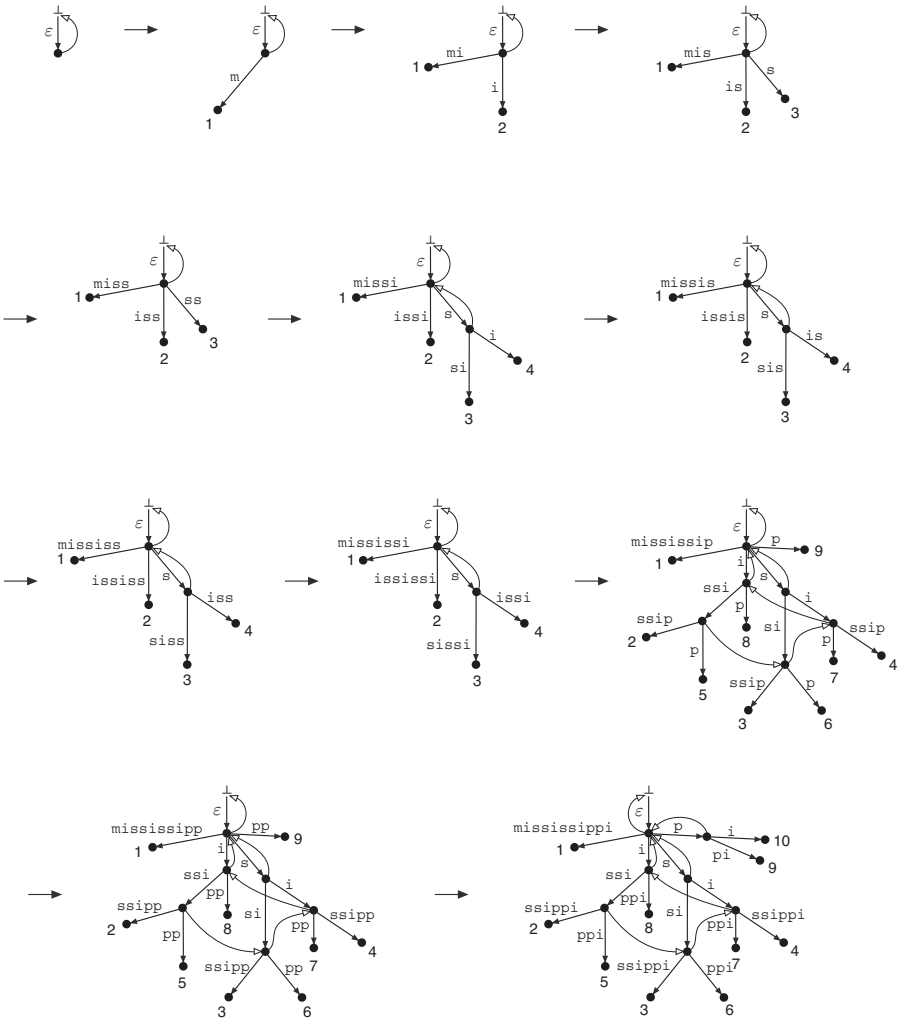


Fig. 4.8. Step-by-step construction of the suffix tree for the string $T =$ `mississippi` using Ukkonen's algorithm

leaf nodes in the tree. Therefore, overall, Ukkonen's algorithm requires $O(n)$ space and time for the construction of the suffix tree for a string of length n.

Figure 4.9 shows the final suffix tree obtained using Ukkonen's algorithm, for $T =$ `mississippi$`. For comparison, we have included the suffix tree with strings for the edge labels, and the final tree with edge-label compression. These can be compared with the suffix trie for the same string shown earlier.

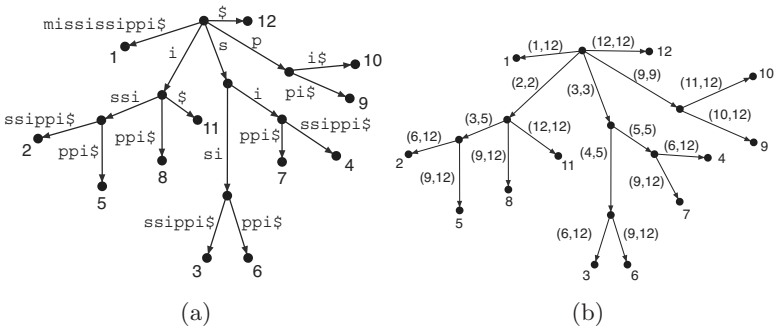


Fig. 4.9. (a) Final suffix tree for the string $T =$ `mississippi` using Ukkonen's algorithm; (b) the same suffix tree represented using edge-label compression

4.1.5 Farach's recursive construction

Farach and colleagues (Farach, 1997; Farach and Muthukrishnan, 1996; Farach-Colton et al., 2000) introduced a fundamentally different method for constructing suffix trees in linear time. Their approach makes use of a recursive decomposition of the original string, whereby the suffix trees of smaller subsets of the suffixes are constructed, and then combined to form the required suffix tree for the original sequence. This new approach has led to some new insights into the properties of suffix trees and their construction. More recently, it has provided a motivation for new methods to construct suffix arrays directly, without first constructing the suffix tree (Kärkkäinen et al., 2006; Kim et al., 2005). This so called *direct suffix sorting* paradigm holds significant promise, especially with respect to efficient computation of the Burrows-Wheeler Transform (both the forward and inverse transformation). We present Farach's suffix tree construction method in detail because of its relationship to some of the direct suffix sorting approaches, and to the BWT.

Basic Algorithm

Farach's algorithm makes use of the relationship between the *longest common prefix* of two strings, and the *lowest common ancestor* of two nodes in a tree. $\text{LCP}(\alpha, \beta)$ denotes the longest common prefix between two strings α and β, and $|\text{LCP}(\alpha, \beta)|$ denotes the length of $\text{LCP}(\alpha, \beta)$. Where the intended meaning is clear from the context, we shall use LCP interchangeably to stand for both the longest common prefix, and its length. $\text{LCA}(u, v)$ denotes the lowest common ancestor of two nodes, u and v, in a tree (that is, $\text{LCA}(u, v)$ is the node furthest from the root that has both u and v as a descendant[1]). An important relationship between the LCP and LCA is the following:

[1] The lowest common ancestor is sometimes referred to in the literature as the *least* or *most recent* common ancestor.

$$\text{LCP}(L(u), L(v)) = L(\text{LCA}(v, u)), \text{ for all nodes } u, v \in \mathcal{T}_T.$$

Thus,

$$|\text{LCP}(L(u), L(v))| = |L(\text{LCA}(v, u))|.$$

It has been shown (Harel and Tarjan, 1984; Schieber and Vishkin, 1988; Bender and Farach-Colton, 2000) that after linear-time preprocessing on a tree, the LCA between any two nodes on the tree can be computed in constant time. This means that essentially, after such a linear-time preprocessing, two arbitrary strings can be compared for equality, or to determine if one is a prefix of the other, in constant time.

Let T be the given string. The odd tree of T, denoted $\mathcal{T}_T^o$ is defined as the suffix tree of all suffixes that start at an odd position in T. Similarly, the even tree of T, denoted $\mathcal{T}_T^e$ is defined as the suffix tree of all suffixes that start at an even position in T. Each of $\mathcal{T}_T^o$ and $\mathcal{T}_T^e$ is half the size of $\mathcal{T}_T$, the true suffix tree for T, with each having $n/2$ leaves. Farach's algorithm then proceeds in three steps:

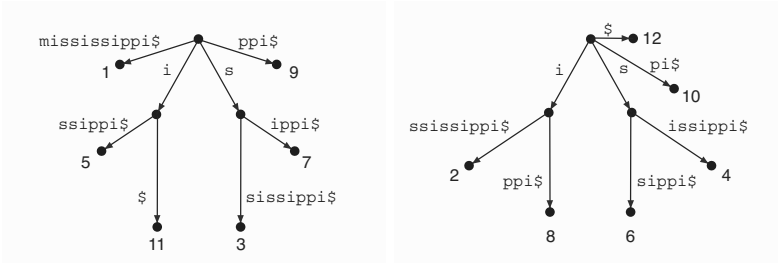
1. Construct $\mathcal{T}_T^o$, the odd tree of T.
2. Using $\mathcal{T}_T^o$, construct $\mathcal{T}_T^e$, the even tree of T.
3. Merge $\mathcal{T}_T^o$ and $\mathcal{T}_T^e$, to form $\mathcal{T}_T$, the final suffix tree for T.

Figure 4.10 shows examples of the odd and even trees, using the example string $T =$ `mississippi`. We describe each step of the algorithm in more detail below. Farach's algorithm relies heavily on integer sorting to achieve linear time complexity, and assumes an integer alphabet $\Sigma = \{1, 2, 3, \ldots\}$, where $|\Sigma| \leq n$. For a general alphabet, we need to map the symbols in the alphabet to an integer alphabet before applying the algorithm, and after construction we map the integer symbols back to their corresponding symbols in the general alphabet. Since the number of unique symbols in a string cannot be greater than the length of the string, this mapping ensures that the linear time complexity of the algorithm extends to general alphabets.

Constructing $\mathcal{T}_T^o$, the odd tree

Construction of the odd tree is performed in four substeps:

1. **Map pairs of symbols to single characters.** From the original string, $T = t_1 t_2 \ldots t_n$, form $n/2$ pairs of symbols from $t_{2i-1}t_{2i}$, $1 \leq i \leq n/2$ (we can pad the string with an extra '$' symbol if n is odd). Radix sort the pairs, and remove duplicates, to form a sorted list, SL. This requires only linear time. Then, convert the pairs into their corresponding integers, based on their rank in SL. The result is a string S' of length $n/2$, defined as follows: $S'[i] =$ rank of $\langle t_{2i-1}t_{2i} \rangle$ in SL. Using the example $T =$ `mississippi$`, and assuming the following mapping of the symbols

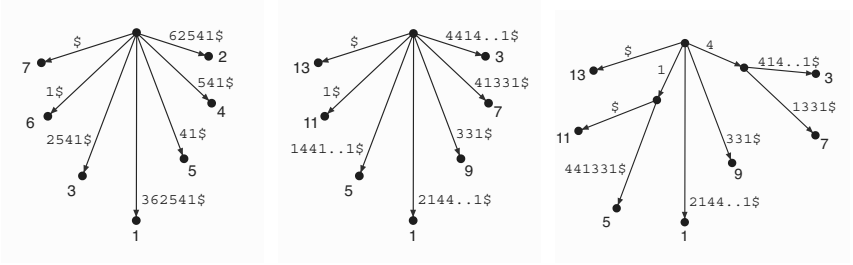


(a) (b)

Fig. 4.10. Odd and even trees for the string $T =$ `mississippi`: (a) odd tree; (b) even tree

to an integer alphabet : $i \rightarrow 1, m \rightarrow 2, p \rightarrow 3, s \rightarrow 4$, we get the following result for the mapping: $T =$ 21441441331\$; Symbol Pairs = { 21, 44, 14, 41, 33, 1\$ }; SL = [1\$, 14, 21, 33, 41, 44] ; $S' =$ 362541\$;

2. **Recursively construct** $\mathcal{T}_{S'}$, the suffix tree of the mapped sequence, S'. Figure 4.11a shows $\mathcal{T}_{S'}$, the suffix tree for the mapped sequence S'.



(a) (b) c)

Fig. 4.11. Constructing the odd tree for $T =$ `21441441331$`: (a) suffix tree for the mapped string $S' =$ `362541$`; (b) initial odd tree constructed from (a); (c) final odd tree after adjustments on (b)

3. **Construct** $\mathcal{T}_T^o$ **from** $\mathcal{T}_{S'}$. First, we observe the relationship between T and S'. Each odd suffix in T has a corresponding suffix in S'. In fact, the odd suffix $t_{2i-1}t_{2i}\ldots t_n\$$ in T corresponds to the suffix $S'[i]S'[i+1]\ldots S'[\frac{n}{2}]\$$ in S'. Thus, the leaf node l_i in $\mathcal{T}_{S'}$ corresponds to the leaf node l_{2i-1} in $\mathcal{T}_T^o$. Any given internal node in $\mathcal{T}_{S'}$ with string depth d becomes an internal node in $\mathcal{T}_T^o$, with depth $2d$. Thus, we can construct $\mathcal{T}_T^o$ from $\mathcal{T}_{S'}$ by replacing the indexes of the leaf nodes and the lengths of the edge labels in $\mathcal{T}_{S'}$ by the corresponding values in $\mathcal{T}_T^o$. However, given that two symbols in T are mapped to one symbol in S', the tree $\mathcal{T}_T^o$

constructed as above may not form a suffix tree, as some nodes may have two edges with edge labels that start with the same symbol (see Figure 4.11b, symbols 1 and 4.) This requires some adjustment to the tree.

4. **Adjust the final suffix tree as needed.** Here, for a given node u, all the edges that start with the same symbol are combined by introducing a new node (say v), between u and the child nodes. The edge (u, v) is then labeled with the symbol shared by the child nodes. If the edge labels for all descendant nodes from u start with the same symbol, the above procedure will imply that node u will have one child node after the adjustment. Thus, node u will be removed from the tree. Of course, this is done while retaining the information in the edge from node u. This adjustment clearly takes linear time with respect to the number of nodes, and the number of edges, each of which is in turn linear with respect to the size of the string, S'.

Figure 4.11 shows $\mathcal{T}_{S'}$, the suffix tree for S', the odd tree constructed from S' (before adjustment), and the final odd tree after adjustment. Let $\varphi(n)$ be the time required to construct the suffix tree for a length-n string, $T = t_1 t_2 \dots t_n$. Then, the time to construct the odd tree using the above procedure will be given by $\varphi(\frac{n}{2}) + O(n)$.

Constructing $\mathcal{T}_T^e$, the even tree

Constructing the even tree from the odd tree is performed based on the fact that, given an inorder traversal of the leaves of a tree, and the depth of the LCA of adjacent leaves in this ordering, we can re-construct the suffix tree in linear time. Consider the example in Figure 4.9 for $T =$ `mississippi`. Traversing the leaf nodes from left to right in a depth-first manner will produce the following list of leaf nodes: $L_1 = [1, 2, 5, 8, 11, 3, 6, 7, 4, 9, 10, 12]$. Now, the list of LCA depths for the adjacent nodes in L_1 will be: $L_2 = [0, 4, 1, 1, 0, 3, 1, 2, 0, 1, 0]$. With only this information, we can easily re-construct the suffix tree using a simple procedure as shown in Algorithm 4.2.

Figure 4.12 shows the result of the first five steps of the algorithm, using the suffix tree of Figure 4.9. The final result of the algorithm is the suffix tree for the given string. Farach's algorithm constructs the suffix tree such that the resulting leaf nodes are sorted in lexicographic order, based on their corresponding suffixes. That is, the edges emanating from each given node are sorted lexicographically by their edge labels. Notice that the algorithm for constructing the suffix tree using the LCA of adjacent leaf nodes also works for suffix trees with sorted suffixes. Therefore, to construct the even tree from the odd tree, we need to derive the sorted suffixes of the even tree from the odd tree, and also the length of the longest common prefix of adjacent suffixes in this ordering.

Obtaining the sorted even suffixes. The current tree $\mathcal{T}_T^o$ already encodes the odd suffixes in lexicographic order. Thus, a simple inorder traversal

SUFFIX-TREE-FROM-LCA-LIST(L_1, L_2)
/* L_1: List of adjacent leaf nodes in a tree: $l_1, l_2, \ldots l_{n+1}$ */
/* L_2: List of depth of LCA of adjacent nodes in the tree */

Create *root* node, with empty string
Set $l_0 \leftarrow$ *root* node; set $d_0 \leftarrow 0$
for $i \leftarrow 1$ **to** $n+1$ **do**
  Compute $d_i \leftarrow |\text{LCP}(l_{i-1}, l_i)|$ using L_2
  Starting from the *root*, jump to position d_i along the path $L(l_{i-1})$
  if d_i falls at an edge (i.e. between two nodes) **then**
    Let the edge be (u_{i-1}, v), with edge label (p, q)
    Split the edge (u_{i-1}, v) to create a new node u_i at this position
    Label edge (u_{i-1}, u_i): $label(u_{i-1}, u_i) \leftarrow (p, p + d_i - 1)$
    Label edge (u_i, v): $label(u_i, v) \leftarrow (p + d_i, q)$
  else /*d_i falls at an existing node */
    Call this node u_i
  end if
  Attach a new leaf node l_i at node u_i
  Label edge (u_i, l_i): $label(u_i, l_i) \leftarrow (l_i + d_i, n + 1)$
end for

Algorithm 4.2: Suffix tree from LCA of adjacent nodes

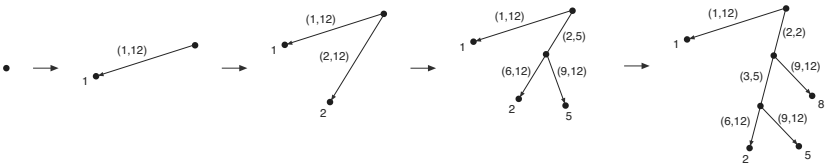


Fig. 4.12. First few steps in constructing the suffix tree from the list of depths for the LCA of adjacent nodes, using $L_1 = [1, 2, 5, 8, 11, 3, 6, 7, 4, 9, 10, 12]$ and $L_2 = [0, 4, 1, 1, 0, 3, 1, 2, 0, 1, 0]$. The final result should correspond to the suffix tree in Figure 4.9b

of the tree will produce a list of the odd suffixes in sorted order. Observe that an even suffix is just one symbol followed by an odd suffix. Thus, to obtain the sorted even suffixes, we form two-element tuples using the pairs $\langle t_{2i}, r_{2i+1} \rangle$, where r_j is the *rank* of suffix T_j in the ordered odd suffixes. The tuples are already sorted by the second element (the odd suffixes). Then, stable-sorting the tuples using radix sort on the first elements will produce the sorted list of even suffixes.

Using the previous example, the ordering of the odd suffixes (the leaves in the odd tree $\mathcal{T}_T^o$) will produce the list: $L_s^o = [13, 11, 5, 1, 9, 7, 3]$. These are then combined with the symbols at the corresponding even positions (left of each odd position) in T to give the tuples: $[\langle 1, 7 \rangle, \langle 4, 3 \rangle, \langle 4, 6 \rangle, \langle 1, 5 \rangle, \langle 3, 2 \rangle, \langle \$, 1 \rangle]$.

Notice that the rank for T_1 (i.e. 4) is not used, since there is no even suffix to the left of position 1 in T. Stable-sorting this list using radix sort with the first element as the key will produce the ordered list: $L_E = [\langle \$, 1\rangle, \langle 1, 5\rangle, \langle 1, 7\rangle, \langle 3, 2\rangle, \langle 4, 3\rangle, \langle 4, 6\rangle]$. We call L_E the sorted even tuples of T. This is the lexicographically ordered list of even suffixes. Using the lists L_E and L_s^o, the corresponding ordered even positions in T can be obtained easily:

$$L_s^e = L_s^o[L_E[i, 2]] - 1, \qquad i = 1, 2, \ldots, \frac{n}{2}.$$

For the running example, the ordered positions will be: $L_s^e = [12, 8, 2, 10, 4, 6]$.

Computing the LCP between the sorted even suffixes. Given the relationship between the LCA and LCP, the LCP between adjacent elements in the sorted even suffixes can be determined based on the LCA between the leaf nodes in the odd tree. After linear time LCA preprocessing (Harel and Tarjan, 1984; Schieber and Vishkin, 1988; Bender and Farach-Colton, 2000) of the odd tree, we can determine the LCP of any two suffixes, represented by two leaves (say, l_{2i}, l_{2j} in $\mathcal{T}_T^e$) using the relation:

$$\text{LCP}(l_{2i}, l_{2j}) = \begin{cases} \text{LCP}(l_{2i+1}, l_{2j+1}) + 1, & : \text{ if } t_{2i} = t_{2j} \\ 0 & : \text{ otherwise} \end{cases}$$

As described earlier, the value of $\text{LCP}(l_{2i+1}, l_{2j+1})$ can be obtained after linear time preprocessing of the odd tree as follows:

$$\text{LCP}(l_{2i+1}, l_{2j+1}) = \text{LCP}(T_{2i+1}, T_{2j+1}) = L(\text{LCA}(l_{2i+1}, l_{2j+1}))$$

where $L(u)$ is the path label of node u. Thus, given the ordered list of even suffixes and the computed LCP's between adjacent elements in this ordering, the even tree can be easily computed in linear time. The even tree of the running example with $T =$ `mississippi` is shown in Figure 4.13(b).

Merging the odd and even trees

The final suffix tree $\mathcal{T}_T$ is obtained by merging $\mathcal{T}_T^o$ and $\mathcal{T}_T^e$, the odd and even tree respectively. This merging process is performed in two steps:

1. **Initial merging.** The first step of the merging process is simple. Farach's merging algorithm uses a *coupled-depth-first* traversal (coupled DFS) of $\mathcal{T}_T^o$ and $\mathcal{T}_T^e$. The coupled DFS starts with the root nodes in the two trees. Next, it will take two edges from the respective trees with the same starting symbol, and recursively merge the two subtrees. Repeating this coupled-DFS procedure on each pair of edges with the same starting symbol from the roots of $\mathcal{T}_T^o$ and $\mathcal{T}_T^e$ produces an initial merged tree. Notice that at any given stage, merging is performed only if both the odd tree and even tree share an edge with the same starting symbol. Otherwise there is nothing to merge.

The only complication in this procedure comes from the fact that we are merging suffix trees (i.e. compacted tries), rather than simple suffix tries. The problem is that even though two edges may start with the same symbol, their edge labels may be different — one edge label could also be a prefix of another. Thus, we may need to stop the merging at the position of first mismatch. To check if the edge labels match symbol-by-symbol would lead to an $O(n^2)$ time for merging. To solve this problem, Farach used a clever approach: at any step during the coupled-DFS merge process, simply merge any two edges that share the same starting symbol. That is, if two edges start with the same symbol, then assume naïvely that the shorter edge is a prefix of the longer edge, break the longer edge, and merge its prefix with the shorter edge. Merging is therefore performed only on equal-length edges. This simple strategy guarantees that we cannot fail to merge any two edges that need to be merged, and requires only linear time to run. However, it could merge more than we need at times. Thus, we have to undo the extraneous merging that may be performed by the algorithm. We call this partial "unmerge" process *merge refinement*.

2. **Merge refinement.** Let $\mathcal{M}_T$ be the resulting merged tree using the simple naïve strategy above. For each node in $\mathcal{M}_T$, we need to determine whether the node requires merge refinement (i.e. merging went too far), and then perform the partial unmerge procedure if needed. First, we preprocess $\mathcal{M}_T$ in linear time, so that LCA queries can be answered in constant time. The path label for each node in $\mathcal{M}_T$ can be determined by using the corresponding node in either $\mathcal{T}_T^o$ or $\mathcal{T}_T^e$. Let u be a node in $\mathcal{M}_T$ such that it is the lowest common ancestor of two leaf nodes l_{2i} and l_{2i-1}. Let $L(u)$ be the path label of u in its corresponding node in $\mathcal{T}_T^o$ or $\mathcal{T}_T^e$. Let $\hat{L}(u)$ be the current path label of u in $\mathcal{M}_T$. That is, $\hat{L}(u) = \text{LCP}(l_{2i}, l_{2i-1})$, since $u = \text{LCA}(l_{2i}, l_{2i-1})$.
Node u is thus declared as properly merged if $|L(u)| = |\hat{L}(u)|$. Otherwise, $(|L(u)| > |\hat{L}(u)|)$ and merging went too far at the node, and hence we have to perform merge refinement by partially unmerging the merged edges. To do this, we introduce a new node, v in $\mathcal{M}_T$, such that the parent of v is set to the parent of u, and v is the parent of l_{2i} and l_{2i-1}. Node v is sometimes called the *refinement node*. Then, set the path label of node v as $L(v) = \hat{L}(u)$. Unmerge any merged edges between the odd tree and even tree under u. Attach the odd tree and even tree under node u to node v, maintaining the sorted order of the edges from node v. When the unmerge procedure is completed on all nodes that had extraneous merging in $\mathcal{M}_T$, the result will be $\mathcal{T}_T$, the final suffix tree for the input string. The overall unmerging procedure requires linear time processing, which means that Farach's recursive algorithm for suffix tree construction will run in $O(n)$ time.

Figure 4.13c shows the merged tree for the running example, before merge refinement to obtain the final suffix tree in Figure 4.13d. The edges and nodes

involved in merge refinement are indicated using an oval. We can notice that the initial merged tree before merge refinement has a structure that is generally similar to the final suffix tree; the difference is only in their edge labels.

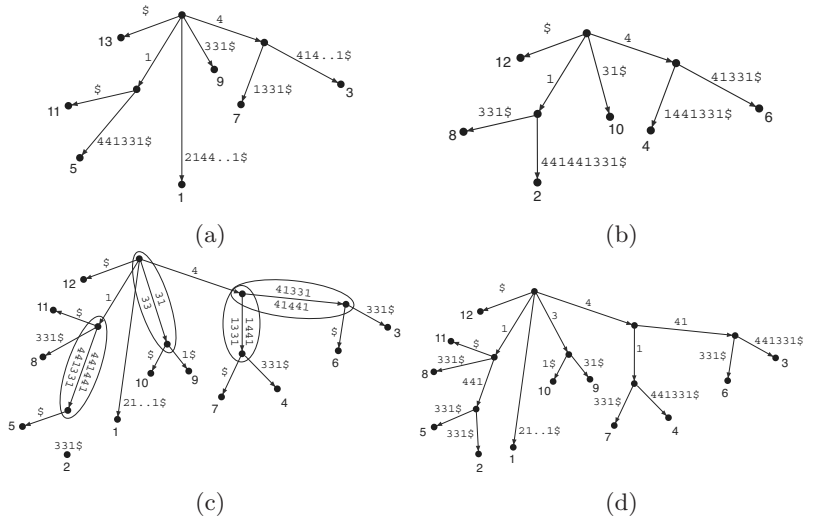


Fig. 4.13. Merging odd and even trees to construct the final suffix tree: (a) the odd tree $\mathcal{T}_T^o$; (b) the even tree $\mathcal{T}_T^e$; (c) the initial merged (overmerged) tree $\mathcal{M}_T$; (d) the final suffix tree $\mathcal{T}_T$ obtained after merge refinement via partial unmerging on $\mathcal{M}_T$. Marked edges with two labels in (c) indicate 'overmerged' edges which require merge refinement. Notice the difference between the LCP of the two labels and the edge length in each case

4.1.6 Generalized suffix trees

For applications involving multiple files, the suffix tree can be constructed for each file independently, and searches or other types of analysis can be performed on each suffix tree. However, for some specific applications, especially for problems such as searching for multiple patterns simultaneously over all the files in a database, a better approach would be to construct a single suffix tree for all the files, and then perform the search on this single tree. Such a single suffix tree constructed for multiple strings is called a *generalized suffix tree* (Gusfield, 1997).

For h multiple strings, $T_1, T_2, \ldots T_h$, building the generalized suffix tree is simple. First, concatenate all the strings into one string: $T = T_1\$_1 * T_2\$_2 * \ldots * T_h\$_h$, where the $\$_i$'s are end of string delimiters, and $*$ denotes concatenation. Then, construct the suffix tree of T, the concatenated string. The overall time

and space complexity is still linear with respect to $|T|$, the overall length of the concatenated sequence.

4.1.7 Implementation issues

The practical time and space requirements for suffix tree construction, and also the space needed to store the suffix tree after construction, depend on the specific implementation method used. After construction, the required space needed to store the suffix tree will include that for the original text, edge labels, node labels for both branching and leaf nodes, and the space to indicate the parent for each node. During construction (and for some applications), we have to add the space required for the suffix links. In the following discussion, by suffix tree construction, we will assume Ukkonen's algorithm; some methods such as Farach's recursive construction may require more space.

The major consideration is how the outgoing edges from a node in the suffix tree are represented. The three major representations used for outgoing edges are arrays, linked lists, and binary search trees. However, independent of the specific method adopted, we can use a simple analysis to provide an idea of the space requirements of a suffix tree. Assume that a pointer is represented as an integer, and that each integer requires 4 bytes to store. Also, we assume that we are using, say, an ASCII alphabet with 256 symbols; notice that in the worst case we may be looking at an alphabet size as large as n, the size of the sequence, or a system such as Unicode may require two bytes per character, but for the meantime we will assume one byte for each character of the text, which is sufficient to compare memory requirements. Based on the properties of a suffix tree, we can expect to use n bytes for the original text, $2n$ integers for suffix links, $2n$ integers for edge labels, $2n$ integers for node labels (including branching nodes and leaf nodes), and $2n$ integers for indicating node parents. Therefore, we need a total of $8n$ integers plus n bytes, or $33n$ bytes to store the suffix tree with suffix links (for instance during construction), and $25n$ bytes without the suffix links (for instance, for later use, or during search).

Now, consider the effect of the specific representation used for the branching edges at each node. The required cost can be broken down into four components: the cost of storing T, the original sequence ($n/4$ integers); cost of branching nodes, cost of leaf nodes, and cost of edge labels. Let n_d be the number of internal nodes in the suffix tree. The cost of edge labels will be $n + n_d - 1$ integers, independent of the specific node representation. Let c_L, c_P, c_B and c_{SL} be the respective cost (in integers) of representing the node label, parent label, edges at the branching node, and suffix links, at each node. (From the discussion above, $c_L = c_P = c_{SL} = 1$ integer, but we these notations for clarity.) Thus, for all the branching nodes, the space cost will be $n_d(c_L + c_P + c_B + c_{SL})$. For the non-branching (i.e. leaf) nodes, there are no outgoing edges, so the cost will be $n_d(c_L + c_P + c_{SL})$.

With the simple array (also called vector) representation, the first symbols on each branch from a node are represented as an array with $|\Sigma|$ elements.

This gives, $c_{SL} = |\Sigma|$ bytes at each node. Thus, using the array representation at each node, the overall cost of storing the suffix tree will be $(4n + n_d(3 + \frac{1}{4}|\Sigma|) + \frac{n}{4})$ integers. With the linked list representation, the major difference is that the first symbols on each edge at each branching node are now represented using a linked list, rather than an array. The advantage is that we need to provide space for only the symbols that actually appear at the node. Without keeping a record of the count of symbols at each node, we can use the fact that we have a maximum of $2n$ edges in the suffix tree to bound the cost. Thus, we need at most $2n$ pointers (i.e. $2n$ integers) for all the liked lists used for the suffix tree. This can be compared with the $\frac{1}{4}n_d|\Sigma|$ integers required by the vector representation, which can grow as large as $\frac{1}{4}n|\Sigma|$ integers in the worst case. Thus, with the linked-list approach, the total cost for storing the suffix tree will be $(6n + 3n_d + \frac{n}{4})$ integers. Another approach is to use a balanced binary search tree to implement the linked list structure used above. This comes at a slightly reduced performance in search time, but is generally more space efficient than the previous two methods. Use of the binary tree approach makes more sense if the alphabet size is very large.

We can observe the significance of n_d, the number of interior nodes in the tree. Clearly, this number varies with the input string, and could significantly affect the required space. Also, for very large alphabets, the vector representation, in its simplest form as given above, will lead to a huge storage requirement. An improvement could be to use a bit map at each node to indicate which symbols in Σ are the starting symbols for the edge labels for all the edges from the node. This avoids the need to reserve space for the symbols that are not represented at a given node. Results in Manber and Myers (1993) show that the linked-list implementation provided an overall best result with respect to space efficiency. Methods based on hash functions were described by McCreight (1976), while Kurtz (1999) and Andersson and Nilsson (1995) provide more discussions on space-efficient construction and representation of suffix trees.

4.2 Suffix arrays

An important data structure, closely related to the suffix tree, is the *suffix array*. The suffix array simply provides a lexicographically ordered list of all the suffixes of a string. If the element at position i in the suffix array is j, it means that T_j, the suffix starting at position j in T is the i-th smallest suffix of T. This is essentially what the array R is in the Burrows-Wheeler Transform (shown in Figure 2.2b for encoding, and in Figure 2.8 for decoding), except the end of the string is treated slightly differently in each case. Combining the suffix array with the (length of) LCP of adjacent suffixes in this array provides a powerful data structure for pattern matching. With this combination, decisions on the occurrence (or otherwise) of a pattern P of length m in the string T of length n can be made in $O(m + \log n)$ time.

The suffix array can be used in most (though, not all) situations where a suffix tree can be used. However, as was shown in the previous section, given the suffix array and the LCP information, the suffix tree can be constructed in linear time. The major motivation for the use of suffix arrays, rather than suffix trees is their smaller memory footprint. Although the theoretical space complexity is linear for both data structures, typically, the suffix tree requires about three to five times more space than the corresponding suffix array of a string. The construction time for both algorithms is also $O(n)$ on average. For suffix arrays, algorithms that run in $O(n \log n)$ worst case are relatively easy to develop, but $O(n)$ worst case algorithms are much harder to come by. Traditionally, the construction of the suffix arrays have often required more practical running time than suffix trees (see Manber and Myers (1990, 1993) for examples). However, this is now changing, as indicated in the recent survey by Puglisi et al. (2007).

To discuss suffix arrays we will need some more notation. We will continue to use $T = t_1 t_2 \dots t_n$ to denote the input sequence of length n, with symbol alphabet Σ. For any two strings, say α and β, we use $\alpha \prec \beta$ to indicate that the string α lexicographically precedes the string β. (This includes the case where α and β could be individual symbols, from the same alphabet, i.e. $|\alpha| = |\beta| = 1$.) We use \$ as the end of string symbol, where $\$ \notin \Sigma$ and $\$ < \sigma, \forall \sigma \in \Sigma$. We also use the notion of order-k sorting. We say that a set of strings are order-k sorted if the strings are sorted by their first k symbols. Thus, for any two strings, say α and β, we use the notation $\alpha \prec_k \beta$ to indicate that string α precedes string β in the order-k sorted listing. Further, we use $\mathcal{A}_T$ to denote the suffix array of a string T. Where the string in question is clear from the context, we may drop the T subscript for simplicity.

4.2.1 Traditional string sorting

Sorting a set of strings in a given order is an age-old problem in computer science. A simple approach is to imagine the strings as vectors, with each row corresponding to one of the strings in the set. Then, sorting the strings can be performed by using standard sorting algorithms such as quicksort to sort the vectors. Radix sort can also be easy to apply in this environment, particularly if the alphabet is small, such as in a DNA sequence. Another approach is to sort each column using character-by-character comparisons, starting from the leftmost column. After the k-th iteration, rows that have the same symbol up to the current (i.e. k-th) column are grouped together. At the next iteration, sorting using the next column is then restricted to the rows in the same group (that is, rows that are the same based on order-k sorting). Repeating this process for all the groups of rows with the same symbol at each iteration, and for all the columns, produces a sorted array of the original vector of strings. The column-wise sorts can be performed using a fast character sorting algorithm such as QSORT, an improved quicksort algorithm reported in Bentley

and McIlroy (1993). This simple algorithm will lead to an expected time in $O(n \log n)$, with a potential $O(n^2 \log n)$ worst case.

Bentley and Sedgewick (1997) proposed a better approach, based on multi-key quicksort. Using code similar to the QSORT algorithm, they sorted a given set of strings by applying the idea of symbol-by-symbol ternary recursive decomposition on the strings. Basically, given a string α, they group every other string β into three partitions, viz:

$\beta \prec_k \alpha$: strings that are less than α,
$\alpha =_k \beta$: strings that are equal to α, and
$\alpha \prec_k \beta$: strings that are greater than α.

Here ternary partitioning is based on a pivot defined by the first k-symbols in each string. Essentially, k is the number of active dimensions, that is, the number of symbols (starting from the leftmost symbol) that need to be considered for a match. With this partitioning, sorting at any stage involves mainly the strings that remain in the equal partition. Strings in this equal partition can be compared (up to the k-th symbol) without requiring symbol-by-symbol comparisons on the first k symbols. This approach results in a sorting algorithm that runs in $O(n_s \log n_s + k n_s)$ worst case time, where n_s is the number of strings, and k is the required number of active dimensions. When k is small, the time reduces to $O(n_s \log n_s)$. However, for the specific problem of sorting the suffixes of a string, we will have $n_s = n$, and $k = n$, leading to an $O(n^2)$ time for suffix sorting.

In general, the longest common prefix, LCP, (see Section 4.1.5) provides an important mechanism to estimate the level of difficulty in sorting the suffixes of a given string. Define the average LCP, and maximum LCP as follows:

$$meanLCP = \frac{1}{n-1} \sum_{i=1}^{n-1} \text{LCP}(T_{\mathcal{A}[i]}, T_{\mathcal{A}[i+1]}) \tag{4.1}$$

$$maxLCP = \max_{1 \le i < n} \{\text{LCP}(T_{\mathcal{A}[i]}), T_{\mathcal{A}[i+1]})\} \tag{4.2}$$

In Equations 4.1 and 4.2, we have used LCP as the length of the longest common prefix. Usually the average LCP (and also the maximum LCP) between any two adjacent suffixes in the sorted list provides a rough indication of the number of symbol comparisons that will be needed to sort the suffixes. Larger values of these statistics imply more difficulty in performing the suffix sorting. Let $\mu = meanLCP$. For methods that are based on standard string matching using symbol- by-symbol comparisons, the average case complexity will be in $O(\mu n \log n)$. When μ is small, or independent of n, this will result in $O(n \log n)$ time. However, since μ could be in $O(n)$, without careful consideration this could lead to a worst-case complexity of $O(n^2 \log n)$ for such schemes.

4.2.2 Suffix arrays via suffix trees

A theoretically faster approach to construct the suffix array is via the suffix tree. Given the suffix tree, the suffix array can be constructed in linear time by a simple inorder traversal of the suffix tree. For instance, using Farach's recursive construction, all the edges from any given node are implicitly sorted by their edge label. Thus, assuming the edges at each node are sorted from left to right in ascending lexicographic order, a simple depth-first traversal of $\mathcal{T}_T$, the suffix tree of T, will produce the suffix array, $\mathcal{A}_T$, the lexicographically sorted list of all the suffixes in T.

Given the space requirement of suffix trees, there has been interest in direct construction of suffix arrays, without the need to first construct the suffix tree (so called *direct suffix sorting* problem). In the following, we describe some of the proposed methods for direct suffix sorting, starting with the Manber-Myers algorithm (Manber and Myers, 1993).

4.2.3 Manber-Myers suffix sorting algorithm

The problem with using the algorithms that sort a set of strings for the problem of suffix sorting is that they ignore important properties of the suffixes of a string, which are not often shared by a random collection of strings. For instance, suffixes of a string share a lot of common substrings, which can be exploited for more efficient sorting. Moreover, conventional sorting algorithms based on symbol-by-symbol comparisons are often limited to $O(n \log n)$ expected time, with some requiring a quadratic time or more in the worst case[2]. The suffix tree on the other hand, requires much more space than may be needed, and for some applications, avoiding the complications required for efficient suffix tree construction could be advantageous.

Manber and Myers (Manber and Myers, 1990, 1993) were the first to propose an algorithm that directly computes the suffix array for a string, without the need for an initial construction of the suffix tree. Their algorithm performs suffix sorting in phases, using the idea of *successive doubling*, earlier used by Karp et al. (1972) for identification of repeats in a string. Using successive doubling, suffix sorting on a string of length n is performed using a maximum of $\lceil \log n \rceil$ phases. First, the suffixes are placed into buckets according to their first symbols. Essentially, the buckets can be viewed as the first pass on the suffix array $\mathcal{A}_T$, whereby the array is sorted only by the first symbol in each suffix (the $\prec_k$ ordering, with $k = 1$). Thus, consecutive entries in the same bucket will have the same first character.

The above can be accomplished in linear time using a bucket sort. This is phase 0 of the algorithm, and the sorted results correspond to the order-1

[2] We note that fast algorithms for suffix arrays can be used to sort a set of strings (in linear time with respect to the total length of the strings), by simply generating the generalized suffix array of the set, and using simple bookkeeping.

sorted suffixes. Then, at each subsequent phase, the buckets are partitioned by sorting according to *double* the number of symbols used in the previous phase. This means that the number of symbols affected in phase i will be 2^i. Therefore, after the i-th phase, $(i = 0, 1, 2, \ldots, \lceil \log n \rceil)$, the suffixes will be order-k sorted, where $k = 2^i$. Thus, after the i-th phase in the algorithm, each bucket will hold only suffixes with the same first k symbols, $k = 2^i$. The major challenge is how the elements in each order-k bucket (which are so far sorted according to the $\prec_k$ ordering) can now be sorted to obtain the order-$2k$ buckets (with elements in $\prec_{2k}$ ordering) all in linear time.

The key observation is that, after order-k bucketing, for a given suffix, say T_i, the next k symbols of T_i are just the first k symbols of suffix T_{i+k}. Thus, given two suffixes, say T_i and T_j in the same order-k bucket, (i.e. $T_i =_k T_j$), the relative order of T_{i+k} and T_{j+k} (with respect to the $\prec_k$ ordering) is immediately available. Thus, at phase-$2k$ $(k = 2, 4, 8, \ldots)$, in the algorithm (we already have order-1 buckets from the initial bucket sort), we sort the suffixes by starting with the first suffix in the first bucket (which necessarily contains the smallest order-k suffixes). Let this first suffix at $\mathcal{A}_T[1]$ be T_i. Given that T_i starts with the smallest order-k suffix, then, the suffix T_{i-k} must be the first suffix in its order-$2k$ bucket. Therefore, we need to move T_{i-k} to the start of its bucket. Then, take the next suffix according to the $\prec_k$ ordering (i.e. the suffix at $\mathcal{A}_T[2]$, say T_j), and move suffix T_{j-k} to the next available place in its own bucket. The phase continues with this movement of suffixes until all the suffixes in all buckets have been processed. The algorithm will stop when each bucket contains exactly one suffix, which will occur after $\lceil \log_2 n \rceil$ phases at the most. A basic description of Manber and Myers' suffix sorting algorithm is given in Algorithm 4.3.

```
MANBER-MYERS-SUFFIX-SORTING(T)
/* Returns the suffix array in the array A_T */
 Perform initial bucket sort on T to produce initial sorted array A_T
 for u ← 0 to ⌈log n⌉ do
    k ← 2^u
    for v ← 1 to n do
        i ← A_T[v]
        if (i − k > 0) or (i + k ≤ n + 1) then
            Move suffix T_{i−k} to the next available slot in its bucket (in A_T)
        end if
    end for
 end for
```

Algorithm 4.3: Basic Manber-Myers suffix sorting algorithm

Figure 4.14 shows a run of the algorithm on the example string, $T =$ `mississippi$`. For a given suffix T_i, T_i is not moved in its bucket when $i + k > n + 1$, or if $i - k \leq 0$. Thus we have:

- In phase $k = 1$, no movement at step v=6, since $i - k = 0$.
- In phase $k = 2$, no movement in step 4 ($i-k = 0$), and in step 6 ($i-k < 0$); at step 1, T_{11} was not moved in its current position, since $11 + k > n + 1$, with $k = 2, n = 11$.
- In phase $k = 4$, no movement in steps 4, 6, and 11 ($i - k < 0$), and in step 10 ($i - k = 0$); no more movements within the p-bucket, since for each suffix T_i in this partition, $i + k > n + 1$, with $k = 4, n = 11$.

The final sorted array is given by the values in $\mathcal{A}_T$ at the end of the last phase (the last column in the table): $\mathcal{A}_T = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]$.

Bucket	Phase $k=0$ (Bucket Sorting)		Phase $k=1$			Phase $k=2$			Phase $k=4$		
	Suffix	$\mathcal{A}$	Step	Suffix	$\mathcal{A}$	Step	Suffix	$\mathcal{A}$	Step	Suffix	$\mathcal{A}$
$	$	12		$	12		$	12		$	12
i	ississippi$	2	1	i$	11	◇ 1	i$	11	◇	i$	11
	issisppi$	5	7	ippi$	8	8	ippi$	8	8	ippi$	8
	ippi$	8	9	ississippi$	2	11	ississippi$	2	9	issippi$	5
	i$	11	11	issippi$	5	12	issippi$	5	10	ississippi$	2
m	mississippi$	1	2	mississippi$	1	◇4	mississippi$	1	◇4	mississippi$	1
p	ppi$	9	5	pi$	10	2	pi$	10	◇	pi$	10
	pi$	10	8	ppi$	9	7	ppi$	9	◇	ppi$	9
s	ssissippi$	3	3	sissippi$	4	3	sippi$	7	3	sippi$	7
	sissippi$	4	4	sippi$	7	5	sissippi$	4	5	sissippi$	4
	ssippi$	6	10	ssissippi$	3	9	ssissippi$	3	11	ssippi$	6
	sippi$	7	12	ssippi$	6	10	ssippi$	6	12	ssissippi$	3

Fig. 4.14. Sample run of Manber-Myers suffix sorting algorithm on the string $T =$ `mississippi$`. The final result (the suffix array) is the last column in the figure. The symbol $\diamond i$ indicates that in step i of the current phase, no more comparison is needed. The symbol $\diamond$ (without a number) indicates that the corresponding suffix is now in its final position in the suffix array, and thus requires no more movements

Notice that only pointers to the suffixes need to be moved, there is no need to physically copy the suffixes. Thus, each phase requires $O(n)$ time to complete in the worst case, leading to an overall worst case time of $O(n \log n)$ for the algorithm. On average, however, the algorithm will run in $O(n \log \log n)$ time, since the length of the maximum LCP will be in $O(\log_{|\Sigma|} n)$ on average, for a string with uniformly distributed symbols (Karlin et al., 1983). This can be further reduced to $O(n)$ expected time, using a linear-time mapping of appropriately sized small substrings to integers before sorting begins, for instance, using a hash function (see Section 7.1.4). Further details of these improvements are provided in the original paper (Manber and Myers, 1993).

An improvement and perhaps simplification of the algorithm could be to use radix sort at each phase, rather than the successive doubling and move-

ment of suffixes. That is, after the first phase of bucket sorting, subsequent phases are performed by forming a tuple using the current symbol, and the bucket number of its next symbol in T. Let $\mathcal{A}_T$ be the current suffix array at phase k. This will contain the indexes for the order-k sorted suffixes. As in the original algorithm, $\mathcal{A}_T$ will be progressively sorted as the algorithm progresses. Let $B[i]$ be the bucket number for the i-th suffix in $\mathcal{A}_T$. Suffixes in the same bucket are given the same bucket number. Let $\tilde{\mathcal{A}}_T$ be the inverse of $\mathcal{A}_T$, that is, if $\mathcal{A}_T[i] = j$, then $\tilde{\mathcal{A}}_T[j] = i$. Then, at each phase the tuple is formed as follows: $\langle B[i], B[\tilde{\mathcal{A}}[\mathcal{A}_T[i] + 1]]\rangle$. Then we radix-sort the set of tuples in linear time. We continue in this way until each bucket contains exactly one suffix. Overall, the complexity remains the same as in the original algorithm, but using radix sorting will simplify the required movements needed when using successive doubling. On average, this would also require less space in practice, since after the initial bucket sort, radix sorting can be confined to smaller buckets at subsequent phases.

The basic Manber and Myers algorithm has been implemented by McIlroy and McIlroy in the SSORT suffix sorting routine. Larsson and Sadakane (1999, 2007) proposed various improvements to the basic algorithm by combining the successive doubling technique with the ternary-partitioning quicksort proposed by Bentley and Sedgewick (1997).

4.2.4 Linear-time direct suffix sorting

More recently, there has been interest in constructing direct suffix sorting algorithms that do not use the suffix tree data structure, but still run in linear time in the worst case. Example algorithms that achieve this running time complexity can be found in Kärkkäinen et al. (2006), Ko and Aluru (2005) and Kim et al. (2005). Here we describe the KS Algorithm (Kärkkäinen and Sanders, 2003; Kärkkäinen et al., 2006) in more detail, given its simplicity.

The KS suffix sorting algorithm

Kärkkäinen and Sanders (Kärkkäinen et al., 2006) proposed a divide and conquer approach similar to Farach's suffix tree construction method of Section 4.1.5, but for direct construction of suffix arrays. Here, rather than dividing the sequence into two symmetric parts, the sequence was divided into two unequal parts by considering suffixes that begin at positions $((i-1) \bmod 3 \neq 0)$ in the sequence. These suffixes are recursively sorted, and then the remaining suffixes are sorted based on information in the first part which is already sorted. The two sorted lists of suffixes are then combined using a merging step to produce the final suffix array. Thus, a major difference is in the way they divided the sequences into two parts, and in the merging step. Also, the use of a 2/3 recursion (rather than the traditional half recursion) significantly simplified the later merging stage, since a relative order between any conflicting symbols can be found in at most two steps of comparison.

Following Farach's recursive suffix tree construction algorithm (Farach, 1997; Farach and Muthukrishnan, 1996; Farach-Colton et al., 2000) introduced in Section 4.1.5, we describe the basic KS-Algorithm as follows:

1. Classify the suffixes into `Type 1` and `Type 2` suffixes as follows[3]:
   - T_i is a `Type 1` suffix if : $(i-1) \bmod 3 = 0$
   - T_i is a `Type 2` suffix if : $(i-1) \bmod 3 \neq 0$
2. Sort `Type 2` suffixes to form $\mathcal{A}^2_T$.
3. Using $\mathcal{A}^2_T$, construct $\mathcal{A}^1_T$, the sorted order for `Type 1` suffixes.
4. Merge $\mathcal{A}^2_T$ and $\mathcal{A}^1_T$ to form $\mathcal{A}_T$, the final suffix array for T.

Sorting `Type 2` suffixes. The major problem in the KS Algorithm is the second step – sorting the `Type 2` suffixes to form $\mathcal{A}^2_T$. This is performed in a recursive manner. First, the algorithm sorts the `Type 2` suffixes based on their first three symbols. For suffix $T_i, (i-1) \bmod 3 \neq 0$, this will be the trigrams or triplets $[T_i[1], T_i[2], T_i[3]] = T[i \ldots i+2]$. If all the triplets are unique (and hence have unique ranks in the sorted order), the step is complete. This will mean that the `Type 2` suffixes have a maximum LCP of 3. In most general cases, however, the maximum LCP will be more than 3, and hence more computation is required to complete the sorting. To do this, a new string of integers is formed by writing out the triplets from the `Type 2` suffixes in their order of occurrence in T, and then replacing each triplet with its rank in the current sorted order. Call this new string S. Notice the similarity between the string S, and the string S' used in Farach's suffix tree construction (see Section 4.1.5). The algorithm is then applied recursively on S to construct its suffix array, $\mathcal{A}_S$. The array $\mathcal{A}_S$ is equivalent to (has a one-to-one mapping with) $\mathcal{A}^2_T$, the required sorted order of the original `Type 2` suffixes from T.

Sorting `Type 1` suffixes using sorted `Type 2` suffixes. After sorting the `Type 2` suffixes, $\mathcal{A}^1_T$ the sorted order of the `Type 1` suffixes can be deduced from $\mathcal{A}^2_T$ by forming the tuple: $\langle T_i[1], \text{rank of } T_{i+1} \text{ in } \mathcal{A}^2_T \rangle$ for each $(i-1) \bmod 3 = 0$. This is equivalent to the pair $\langle T_i[1], \tilde{\mathcal{A}}_S[i+1] \rangle$, where $\tilde{\mathcal{A}}_S$ is the inverse of $\mathcal{A}_S$. The pairs can then be sorted in linear time, using radix sort to give $\mathcal{A}^1_T$.

Merging sorted `Type 1` and `Type 2` suffixes. The final step is to merge $\mathcal{A}^1_T$ and $\mathcal{A}^2_T$ to form the required suffix array $\mathcal{A}_T$. The key is that conflicts that can arise during the merging can each be resolved by using $\mathcal{A}^2_T$. To compare a `Type 1` suffix T_i with a `Type 2` suffix T_j, at the merge stage, we need to consider two cases:

- **1-Compare Case:** If $(j-1) \bmod 3 = 1$, we compare $\langle T_i[1], \text{rank of } T_{i+1} \text{ in } \mathcal{A}^2_T \rangle$, versus $\langle T_j[1], \text{rank of } T_{j+1} \text{ in } \mathcal{A}^2_T \rangle$. For this simple case, the relative order of both T_{i+1} and T_{j+1} are available from $\mathcal{A}^2_T$.

[3] We use `Type 1` and `Type 2` for ease of description. These were not necessarily used by the authors in their original work.

- **2-Compare Case:** If $(j-1) \bmod 3 = 2$, we compare $\langle T_i[1], T_i[2],$ rank of T_{i+2} in $\mathcal{A}_T^2\rangle$, versus $\langle T_j[1], T_j[2],$ rank of T_{j+2} in $\mathcal{A}_T^2\rangle$. Again, for this more difficult case, the tie is broken using the triplet, since the relative order of both T_{i+2} and T_{j+2} are also available from $\mathcal{A}_T^2$.

Below, we further explain the working of the KS Algorithm using an example. Consider the string $T =$ `mississippi$`. First, we group the suffixes into their respective types.

`Type 1` suffixes for T: $T_i|(i-1) \bmod 3 = 0$:

^{1}`mississippi$`
^{4}`sissippi$`
^{7}`sippi$`
^{10}`pi$`

For `Type 1` suffixes, we have used superscripts to indicate their corresponding positions in the parent sequence. Sorting these suffixes will produce $\mathcal{A}_T^1$. `Type 2` suffixes use the rule: `Type 2` suffixes for T: $T_i|(i-1) \bmod 3 \neq 0$. Table 4.1 shows the `Type 2` suffixes, and their sorted order, based on the triplets ($\prec_3$ ordering). Complete sorting of the `Type 2` suffixes will produce $\mathcal{A}_T^2$.

Suffix Position	Suffix	Trippes	Sorted Triples	Sorted Positions	Label (Index)
2	`ississippi$`	`iss`	`$$$`	12	1
3	`ssissippi$`	`ssi`	`i$$`	11	2
5	`issippi$`	`iss`	`ipp`	8	3
6	`ssippi$`	`ssi`	`iss`	2	4
8	`ippi$`	`ipp`	`iss`	5	4
9	`ppi$`	`ppi`	`ppi`	9	5
11	`i$`	`i$$`	`ssi`	3	6
12	`$`	`$$$`	`ssi`	6	6

Table 4.1. `Type 2` suffixes after the first level of iteration in the KS Algorithm using the example $T =$ `mississippi`

Since the labels are not unique for all the triplets, we need to construct a new string, and apply the algorithm recursively. The new string will be: $S = 46463521$. We divide S into its `Type 1` and `Type 2` suffixes. The `Type 1` suffixes for S will be the set: { ^{1}46463521$, ^{4}63521$, ^{7}21$ }. Sorting these suffixes will produce the suffix array $\mathcal{A}_S^1$. Table 4.2 shows the `Type 2` suffixes for string S, and their sorted order, based on the triplets ($\prec_3$ ordering). A complete sorting of these `Type 2` suffixes will produce $\mathcal{A}_S^2$.

From Table 4.2 we have $\mathcal{A}_S^2 = [8, 5, 3, 6, 2]$ and $\tilde{\mathcal{A}}_S^2 = [5, 3, 2, 4, 1]$. We determine the order of the `Type 1` suffixes in S by forming the array of tu-

ples: $[\langle 4,5\rangle, \langle 6,2\rangle, \langle 2,1\rangle]$. Sorting these tuples will give $\mathcal{A}_S^1 = [7,1,4]$. (In this particular case, the sorted order is available, without forming the tuples.)

The next step is to merge $\mathcal{A}_S^1$ and $\mathcal{A}_S^2$ to form $\mathcal{A}_S = [8,7,5,3,1,6,4,2]$. These are based on the indexes in the shorter string S. We map these back to their original positions in T to obtain $\mathcal{A}_T^2 = [12,11,8,5,2,9,6,3]$. Now, we deduce the sorted order for the `Type 1` suffixes in T by sorting the tuples: $[\langle m,5\rangle, \langle s,4\rangle, \langle s,3\rangle, \langle p,2\rangle]$. The result will be $\mathcal{A}_T^1 = [1,10,7,4]$. Finally, we merge $\mathcal{A}_T^1$ and $\mathcal{A}_T^2$ to form $\mathcal{A}_T$, the required suffix array: $\mathcal{A}_T = [12,11,8,5,2,1,10,9,7,4,6,3]$.

Note that the number of `Type 1` suffixes is $n/3$, while that of `Type 2` suffixes is $2n/3$. This is important, since the recursive call applies only to `Type 2` suffixes. Thus, the running time for the algorithm is given by the solution to the recurrence : $\varphi(n) = \varphi(\lceil 2n/3 \rceil) + O(n)$. This gives $\varphi(n) = O(n)$.

The KA algorithm

Ko and Aluru (2005) also used recursive partitioning, but following a fundamentally different approach to construct the suffix array in linear time and space. They use a binary marking strategy whereby each suffix in T is classified as either an S-suffix or an L-suffix, depending on the relative order with its next neighbor. An S-suffix is a suffix that is lexicographically smaller than its right-neighbor in T, while an L-suffix is one that is lexicographically larger than its right-neighbor. That is, T_i is an S-suffix if $T_i \prec T_{i+1}$, otherwise T_i is an L-suffix. This classification is motivated by the observation that an S-suffix is always lexicographically greater than any L-suffix that starts with the same character. The two types of suffixes are then treated differently: the S-suffixes are sorted recursively by performing some special distance computations. The L- suffixes are then sorted using the sorted order of the S-suffixes. The classification scheme is very similar to an approach used earlier by Itoh and Tanaka (1999), (see Section 4.3), but the algorithm in Itoh and Tanaka (1999) runs in $O(n \log n)$ time on average, and $O(n^2 \log n)$ worst case.

Suffix Position	Suffix	Trippes	Sorted Triples	Sorted Positions	Label (Index)
2	6463521$	646	1$$	8	1
3	463521$	463	352	5	2
5	3521$	352	463	3	3
6	521$	521	521	6	4
8	1$	1$$	646	2	5

Table 4.2. `Type 2` suffixes after the second level of recursion in the KS Algorithm using $S = 46463521\$$

4.3 Space issues in suffix trees and suffix arrays

One major motivation for the use of suffix arrays over suffix trees is the small memory footprint of the former. The time and space requirements of the two data structures can be considered from two view points: at the construction stage, and after construction (for storage, or at the time of use, for instance, during search). While the suffix array generally requires a smaller space to store, and less time for searching, traditionally, their construction requires more time than suffix trees (Manber and Myers, 1993). The recent survey in Puglisi et al. (2007), however, shows that some recent suffix sorting algorithms can be faster than suffix arrays, even at construction time. Further, although the suffix array also requires less space than suffix trees during construction, it is still important to consider the actual space needed by a given suffix sorting algorithm. With the increasing input data size for these algorithms (for instance, in genomic applications with potentially billions of symbols in one genome), the space requirement during construction is becoming critical.

There are three major approaches to dealing with the problem of the space required for the construction and use of suffix trees and suffix arrays: (1) space-efficient suffix tree/suffix array construction, (2) compressed suffix trees/suffix arrays, and (3) the construction of suffix trees/suffix arrays in external storage. Below we discuss methods that have been proposed for space-aware suffix sorting. We discuss compressed suffix trees and compressed suffix arrays in Section 8.1, given their very close relationship with BWT-based compressed full-text indexing and other applications discussed in that chapter. The further reading section at the end of this chapter provides some pointers to key references on constructing suffix trees and suffix arrays in secondary storage.

Lightweight suffix array construction

There has been some effort to reduce the actual space requirement in suffix array construction. Algorithms that aim at this reduced space requirement are sometimes called *lightweight* suffix sorting algorithms. More specifically, Manzini and Ferragina (2004) use the term "lightweight" to refer to a suffix sorting algorithm that requires no more than $5n$ bytes plus a small extra space in constructing the suffix array of a string of length n. This is based on the assumption that the alphabet size is no more than 256 characters (which means 1 byte is enough to hold each symbol), and that integers are stored in 4 bytes (32 bits), as is done in most current machine models. Thus, lightweight algorithms require little or no memory beyond those needed to store the text itself (n bytes) and those needed to store the suffix array ($4n$ bytes).

Existing lightweight algorithms can be characterized by the following four steps:

Bucketing. Typically, they start with an initial bucketing to partition the suffixes, usually based on the first one or first two symbols.

Suffix classification. The suffixes are then classified or grouped into different suffix types, often based on their relationship with neighboring suffixes in the string.

Sorting within groups. Based on the suffix group, the lightweight algorithms often use standard string sorting algorithms (such as ternary quicksort (Bentley and McIlroy, 1993), or multikey quicksort (Bentley and Sedgewick, 1997)) to sort suffixes in the same group. Some also use other existing suffix sorting algorithms at this stage, for instance, Seward's COPY algorithm (Seward, 2000) or Larsson and Sadakane's QSUFSORT algorithm (Larsson and Sadakane, 2007).

Derived sorting. Finally, the lightweight algorithms typically exploit the fact that we are sorting suffixes of the same string, by deriving the sorted order of certain buckets or suffixes from previously sorted suffixes.

The difference between the various lightweight suffix sorting algorithms is primarily in the specific approach they used in one or more of the above steps. We describe the popular lightweight algorithms below, with an emphasis on Itoh and Tanaka's TWO-STAGE algorithm, one of the earliest lightweight algorithms.

Itoh-Tanaka Algorithm. Itoh and Tanaka (1999) proposed the TWO-STAGE algorithm for suffix sorting. After initial bucketing based on the first symbol in each suffix, the suffixes in each bucket are then classified into `Type A` and `Type B` suffixes as follows:

- T_i is `Type A` suffix if : $T[i+1] \prec T[i]$
- T_i is `Type B` suffix if : $T[i] \prec T[i+1]$

The `Type B` suffixes are then sorted using a standard string sorting algorithm. Specifically, they used a hybrid of sorting algorithms, depending on the size of the group. They proposed simple insertion sort for small buckets, multikey quicksort (Bentley and Sedgewick, 1997) for medium sized buckets, and MSD radix sort (McIlroy et al., 1993) for large groups.

Note that when a `Type A` and `Type B` suffix are in the same bucket, the `Type A` suffix will always precede the `Type B` suffix. Itoh and Tanaka then used the key observation that, after all `Type B` suffixes have been correctly sorted, the sorted order of `Type A` suffixes can be directly derived from the sorted `Type B` suffixes in one single pass. That is, we simply scan over the suffix array being constructed in ascending order; for a given suffix, say T_i, check if suffix T_{i-1} is `Type A`; if so, move the suffix T_{i-1} to the first empty position in its bucket. Figure 4.15 shows the result of the Itoh-Tanaka TWO-STAGE algorithm on our sample string, $T =$ `mississippi$`.

As can be observed, when the number of `Type A` suffixes is relatively large compared to the number of `Type B` suffixes, the algorithm will work faster in practice. To increase the number of `Type A` suffixes, Itoh and Tanaka suggested the use of buckets based on the first two symbols at the bucketing step (i.e. using $\prec_2$ ordering, rather than $\prec_1$).

Position	1	2	3	4	5	6	7	8	9	10	11	12
Symbol	m	i	s	s	i	s	s	i	p	p	i	$
Type	A	B	B	A	B	B	A	B	B	A	A	–

(a)

Bucket	Position	Suffix	Type	Type B Sorted	Type A (Induced sort)	Merged
$	12	$	–	12		12
i	2	ississippi$	B		11	11
	5	issippi$	B	8		8
	8	ippi$	B	5		5
	11	i$	A	2		2
m	1	mississippi$	A		1	1
p	9	ppi$	B		10	10
	10	pi$	A	9		9
s	3	ssissippi$	B		7	7
	4	ssissippi$	A		4	4
	6	ssippi$	B	6		6
	7	sippi$	A	3		3

(b)

Fig. 4.15. Itoh-Tanaka TWO-STAGE algorithm on the string $T =$ `mississippi$`: (a) classification of suffixes; (b) sorting process

Two other popular lightweight algorithms are Seward's COPY algorithm (Seward, 2000) and Manzini and Ferragina's DEEP-SHALLOW algorithm (Manzini and Ferragina, 2004). Algorithm COPY performs bucketing on the suffixes based on the first symbols, and for each bucket it further partitions the suffixes into smaller buckets based on their second symbols. It then sorts these smaller partitions using ternary quicksort. When a bucket (say with initial symbol σ) is completely sorted, COPY can deduce the sorted order of all other smaller partitions that have symbol σ as its second symbol directly, by a single pass over the bucket. These are therefore not directly sorted using the string sorting algorithm.

Algorithm DEEP-SHALLOW extends the COPY algorithm in several ways. Rather than use ternary quicksort (Bentley and McIlroy, 1993) for sorting the smaller partitions, they use the multikey quicksort proposed by Bentley and Sedgewick (1997). More importantly, they divide the suffixes within a smaller partition into two parts, based on a threshold on the length of their common prefix. For suffixes with the LCP less than the threshold, they use multikey quicksort as above. For those with LCP beyond the threshold, they abandoned the multikey quicksort, and used a different string sorting algorithm. They call the former approach (for small LCP) *shallow sorting*, and the latter (for smaller partitions with large LCP) *deep sorting*. In Manzini and Ferragina (2004), three algorithms for deep sorting were proposed, one generalizing the

idea of direct determination of the sorted order of the smaller partitions based on an already sorted bucket. Recent related work on light-weight suffix sorting has been reported by Maniscalco and Puglisi (2006).

Other algorithms that are closely related to the lightweight algorithms include Larsson and Sadakane's QSUFSORT (Larsson and Sadakane, 1999, 2007), and more recently Schürmann and Stoye's BKPR (bucket-pointer refinement) algorithm (Schürmann and Stoye, 2007). Both algorithms use the successive doubling technique of Manber and Myers as their basic working principle. Both also have a space requirement of $8n$ bytes, with a worst case complexity in $O(n \log n)$ for the former, and $O(n^2)$ for the latter. Schürmann and Stoye focused on strings with variable LCP's as may be needed for applications such as in bioinformatics, using a hybrid of both standard sorting and suffix sorting algorithms.

4.4 Further reading

The suffix tree was originally introduced by Weiner (1973), with space-efficient constructions considered by McCreight (1976). Related data structures such as suffix tries and Patricia trees (Morrison, 1968; Knuth, 1973; Gonnet et al., 1992) have also been studied. Ukkonen (1995) made the suffix tree easier to understand, and showed a simpler method to construct the tree in linear time. Giegerich and Kurtz (1997) showed the close relationship between the seemingly different approaches to suffix tree construction. Farach et al. (Farach, 1997; Farach-Colton et al., 2000) introduced a fundamentally different approach to constructing suffix trees, by simple recursive decomposition. Szpankowski (1993a,b) analyzed suffix trees and the generalized suffix tree. Suffix tree on words are studied in Andersson et al. (1999). Gusfield (1997) provides a detailed study on suffix trees and their various applications. See also Apostolico (1985) for other applications of suffix trees.

The suffix array was introduced by Manber and Myers in the early 1990s (Manber and Myers, 1990, 1993) as a more space efficient alternative to suffix trees. Their algorithm required $O(n \log n)$ time in the worst case, with an average time of $O(n)$. Around the same period, Gonnet et al. (1992) introduced the PAT-array, a very closely-related data structure. Grossi and Vitter (2005) presented various applications of the suffix array data structure. Spurred by the introduction of the BWT in 1994 (Burrows and Wheeler, 1994), which relied heavily on sorting the suffixes of a string, various methods and algorithmic improvements were proposed for the suffix sorting problem (Larsson, 1998; Larsson and Sadakane, 2007; Itoh and Tanaka, 1999; Seward, 2000). While some of the algorithms improved the space needed to construct the suffix array, most of these algorithms still required $O(n \log n)$ time or more in the worst case. It was not until 2003 that algorithms that can construct the suffix array in linear time and linear space in the worst case were introduced (Kärkkäinen and Sanders, 2003; Kärkkäinen et al., 2006; Kärkkäinen, 2007;

Ko and Aluru, 2003, 2005; Kim et al., 2003, 2005; Adjeroh and Nan, 2008). Puglisi et al. (2007) provide a recent survey on suffix arrays. The problem of constructing lightweight suffix sorting algorithms that run in linear time worst case is still open.

Given the problem of space, especially for certain applications that require huge data sizes, more recent efforts have focused mainly on methods to reduce the space requirements for suffix trees and suffix arrays. Space efficient construction of suffix trees were studied in Kurtz (1999) and Andersson and Nilsson (1995). Compressed suffix trees were studied in Munro et al. (2001), Grossi and Vitter (2005), and Kim and Park (2005), while compressed suffix arrays were considered in Grossi and Vitter (2000), Grossi and Vitter (2005), Hon et al. (2003a), and Na (2005). Algorithms that can perform the construction in external storage were proposed in Bieganski et al. (1994), Clark and Munro (1996), Farach-Colton et al. (2000), Hunt et al. (2001), and Cheung et al. (2005) for suffix trees, and in Kärkkäinen et al. (2006) and Crauser and Ferragina (2002) for suffix arrays. Ferragina (2005) looks at the general problem of string searching in external memory. Franceschini and Muthukrishnan (2007) reports on in-place suffix sorting.