Chapter 5

Suffix Trees and its Construction

5.1 Introduction to Suffix Trees

Sometimes fundamental techniques do not make it into the mainstream of computer science education in spite of its importance, as one would expect. Suffix trees are the perfect case in point. As Apostolico[Apo85] expressed it, suffix trees possess "myriad of virtues." Nevertheless, even elementary texts on algorithms, such as [AHU83], [CLRS01], [Aho90] or [Sed88], present brute-force algorithms to build suffix trees, notable exceptions are [Gus97] and [Ste94]. That is a strange fact as much more complicated algorithms than linear constructions of suffix trees are found in many computer science curricula. The two original papers that put forward the first linear-time algorithms have a reputation of being obscure and that they might have contributed to this situation. In the following, we will try to give a clear and thorough explanation of their construction.

Initially, building suffix trees appeared in order to solve the so-called **substring problem**. This problem can be stated as follows.

The substring problem: Pre-process text T so that the computation string matching problem is solved in time proportional to m, the length of pattern P.

This problem has many more applications than the reader can imagine. Perhaps, the most immediate one is performing intensive queries on a big database, which is represented by T. Once the suffix tree for T is built each query is proportional to $O(m)$, not like the algorithms seen so far, which would take $O(n + m)$ time. However, the most important feature of suffix trees is the way it exposes the internal structure of a string and how it eases the access to it.

Among the applications of suffix trees we can mention solving the exact string matching problem (both SME and SMC problems), the substring problem, the longest common substring of two strings problem and the DNA contamination problem. All these problems will be dealt with in the next few sections. For many other applications of suffix trees, the reader is referred to [Gus97] and [Apo85].

In 1973 Weiner [Wei73] presented the first linear-time algorithm for constructing suffix trees. This 11-page paper had an obscure terminology that made it hard to read and be understood. Three years later McCreight [McC76] gave a different algorithm, much better in terms of space efficiency, which was also linear. McCreight's paper was much more streamlined than Weiner's. For twenty years no new algorithm was come up with until 1995. In that year Ukkonen [Ukk95] put forward a clean, easy-to-understand algorithm to build suffix trees. Following Gusfield's presentation [Gus97], we will start by showing a naive algorithm and we will refine it, from top to bottom, until obtaining the desired linear-time algorithm.

5.2 Definitions and the Naive Algorithm

Again, as it happened with the Z algorithm, not only suffix trees are used in string matching, but also in other contexts. Thus, we will describe suffix trees on a generic string S of length s (not to confuse with the shift s used in previous chapters).

A **suffix tree** of a string S is a tree with the following properties:

SF1: It has exactly s leaves numbered from 1 to s.

SF2: Except for the root, every internal node has at least two children.

SF3: Each edge is labelled with a non-empty substring of S.

SF4: No two edges starting out of a node can have string-labels beginning with the same character.

SF5: The string obtained by concatenating all the string-labels found on the path from the root to leaf i spells out suffix $S[i..m]$, for $i = 1, \ldots, s$.

Example. Consider string $S = \{\texttt{abcabx}\}$. Figure 5.1 shows the suffix tree corresponding to S.

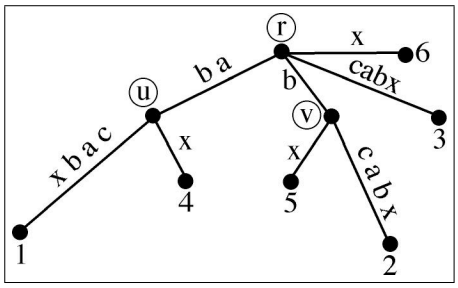

Figure 5.1: The suffix tree of a string.

We can easily check that properties SF1–SF4 are satisfied. As for the fifth property, note that for leaf 4, says, the path from r to 4 is $\{\texttt{abx}\}$, which gives the suffix $S[4..6]$. From now on, the path between two nodes u, v will be denoted by $u \longrightarrow v$. Path $r \longrightarrow 5$ is $S[5..6] = \{\texttt{bx}\}$ and path $r \longrightarrow 2$ spells out $S[2..6] = \{\texttt{bcabx}\}$. (Note the reader that some edge-labels are read from left to right and others right to left; do not be confused by this fact.)

However, there is a catch hidden in the above definition. The definition may suggest that any string must have an associated suffix tree, but that is not true in all instances. The problem arises when one suffix of S matches a prefix of another suffix. In that case it is not possible to build a suffix tree satisfying the definition given above. To illustrate this situation consider string $S' = S - \{\texttt{x}\} = \{\texttt{abcab}\}$. Now, suffix $S'[5..6] = \{\texttt{ab}\}$ matches a prefix of suffix $S'[1..6] = \{\texttt{abcab}\}$. If suffix $S'[5..6] = \{\texttt{ab}\}$ ends at a leaf, necessarily labels $\{\texttt{ab}\}$ and $\{\texttt{abcab}\}$ will start from the root, which would violate property SF5; see Figure 5.2 (a). On the other hand, if suffix $S'[1..6] = \{\texttt{abcab}\}$ is found as the concatenation of $S'[1..2] = \{\texttt{ab}\}$ and $S'[3..5] = \{\texttt{cab}\}$, then there will not be a path from the root to a leaf spelling out $S'[1..2] = \{\texttt{ab}\}$; see Figure 5.2 (b).

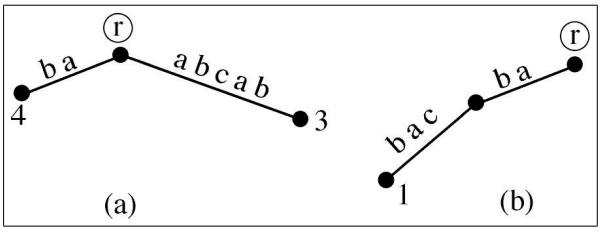

Figure 5.2: The bad cases of the definition of suffix tree.

In order to avoid this problem, from now on we will add a special character to the end of string S. This character, customarily denoted by \$, does not appear in string S. Character

\$ is called the **termination character**. This way, no suffix of S matches a prefix of another suffix of S. In the following, the suffix tree of $\mathcal{T}(S)$ will be defined as the suffix tree of $\mathcal{T}(S\$)$.

We call the **label of a path** starting at the root and ending at a node the concatenation of the strings in its order found along that path. The **path-label of a node** is the label of the path from the root to the node. Given a node v, the **string-depth** of v is the number of characters in the edge ending at v.

Example. In the suffix tree of $S = \{\texttt{abcabx}\}$ depicted in Figure 5.1 the label of path $r \longrightarrow 2$ is $\{\texttt{bcabx}\}$. The path-label of node u is $\{ab\}$. The string-depth of u is 2, the number of characters of string $\{\texttt{ab}\}$.

Next, we describe a brute-force algorithm to build a suffix tree of a string. We call $\mathcal{T}(S)$ the suffix tree of string S. The algorithm works in an incremental way, by processing the m suffixes $S[i..m]$ of S, from $i = 1$ to m, one by one. Let $\mathcal{T}_i$ be the tree obtained at step i. We will show how to construct $\mathcal{T}_{i+1}$ from $\mathcal{T}_i$. Tree $\mathcal{T}_m$ will be the final tree $\mathcal{T}(S)$.

At the first step, $S[1..m]\$$ is inserted in an empty tree and suffix tree $\mathcal{T}_1$ is composed of a unique node. At step $i+1$, suffix $S[i+1..m]\$$ is inserted in $\mathcal{T}_i$ as follows. Traverse the tree starting at the root r and find the longest prefix that matches a path of $\mathcal{T}_i$. If such prefix is not found, this is because none of the previous suffixes $S[j..m]\$$, for $j = 1$ to $j = i$, starts by character $S[i+1]$. In this case, a new leaf numbered $i+1$ with edge-label $S[i+1..m]\$$ is created and joined to the root.

Thus, assume there is a path such that $S[i+1..m]\$$ is a prefix of maximal length of that path. Because of the presence of the termination character \$, the prefix cannot be a substring of a suffix entered into the tree early on. Therefore, there is a character at position k such that $S[i+1..k]\$$ is the prefix; let $S[k..m]\$$ be the non-matching substring. Let (u, v) be the edge of the path where character $S[k]$ is found. The algorithm creates a new node w and a leaf numbered $i+1$. Node w splits edge (u, v) into edges (u, w) and (w, v), and edge joining w and leaf $i+1$ receives edge-label $S[i+1..k]\$$. The algorithm finishes when the termination character is inserted.

For illustration purposes, let us follow the construction of the suffix tree for $S\$ = \{\texttt{abcab\$}\}$. The first tree $\mathcal{T}_1$ is just $S[1..5]\$ = \{\texttt{abcab\$}\}$, as shown in Figure 5.3 (a). Next, $S[2..5]\$ = \{\texttt{bcab\$}\}$ is inserted into $\mathcal{T}_1$. Since $b \neq a$, there is no common prefix of $S[1..5]\$$ and $S[2..5]\$$. Thus, a new leaf numbered 2 is created. The same situation happens with the third suffix $S[3..6]\$ = \{\texttt{cab\$}\}$; see Figure 5.3 (b) and (c).

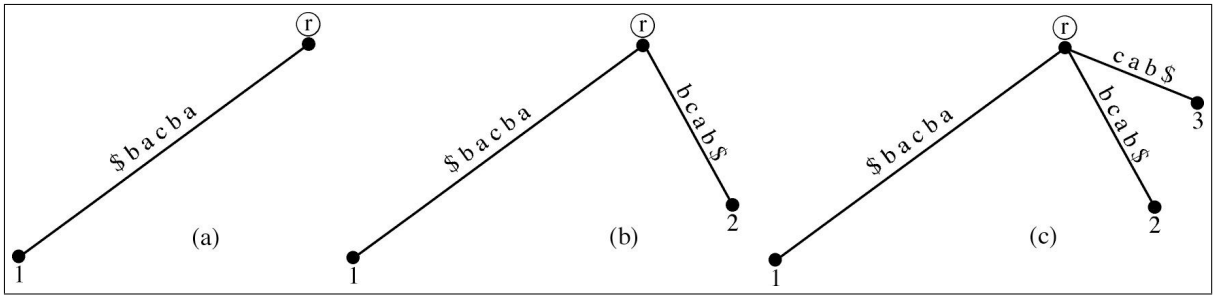

Figure 5.3: Processing the first three suffixes.

The fourth suffix $S[4..5]\$ = \{\texttt{ab\$}\}$ shares a common prefix with $S[1..5]\$ = \{\texttt{abcab\$}\}$. The longest common prefix is $\{\texttt{ab}\}$. Therefore, a new node u is inserted, which splits $\{\texttt{abcab\$}\}$ into substrings $\{\texttt{ab}\}$ and $\{\texttt{cab\$}\}$, as shown in Figure 5.4 (a). Moreover, leaf 4 is created. Next, suffix $S[5..5]\$ = \{\texttt{b\$}\}$ is processed. Suffix $S[5..5]$ is found as a prefix of $S[2..5]\$ = \{\texttt{bcab\$}\}$. Consequently, edge $(r, 2)$ is split into edges (r, v) and $(v, 2)$ and edge $(v, 5)$ is also added to the suffix tree; see Figure 5.4 (b). Finally, the termination character $\{\texttt{\$}\}$ is processed, which merely produces an isolated edge, depicted in Figure 5.4 (c).

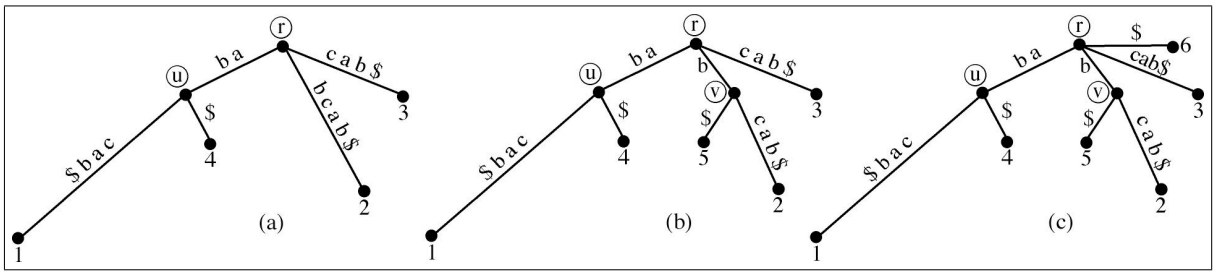

Figure 5.4: Processing the last three suffixes.

Theorem 5.2.1 *The naive algorithm described above correctly constructs the suffix tree of a string.*

Proof: The proof of this theorem is left as an exercise to the reader. ∎

Theorem 5.2.2 *The naive algorithm runs in $\Theta(s^2)$, where s is the length of string S.*

Proof: At step i the algorithm finds the longest common prefix between $S[i..s]$ and the previous i suffixes $S[1..s]\$, \ldots, S[i-1..s]\$$. Let $S[i..k]\$$ be such prefix. Then $k+1$ comparisons have been performed to identify such prefix. After that, the remaining of the suffix, $S[k+1..s]\$$ has to be read and assigned to the new edge. Therefore, the total amount of work at step i is $\Theta(i)$. Hence, the overall complexity is $\sum_{i=1}^{s} \Theta(i) = \Theta(s^2)$. ∎

Exercises

1 Suppose you want to use the Karp-Rabin algorithm to solve the substring problem. What is the best you can do?

2 Build and draw the suffix trees for string $S_1 = \{\texttt{abcdefg}\}, S_2 = \{\texttt{aaaaaaax}\}, S_3 = \{\texttt{abcabcx}\}$. Assume that the last character is the termination character.

3 Any suffix tree will always have at least an edge joining the root and a leaf, namely, the one corresponding to the termination character (it will be edge $(r, s+1)$). If we do not count that edge, is there a string whose suffix tree is a complete binary tree?

4 Prove that the naive algorithm to build suffix trees given in the text is correct.

5 Find out what the best case for the naive algorithm for building suffix trees is.

6 With regard to string S, what is the degree of the root of $\mathcal{T}(S)$? What is the number of internal nodes of $\mathcal{T}(S)$?

7 Given a string S, its **reverse string** is denoted by S^r. The reverse string is defined as the string S output in reverse order. Is there any relation between $\mathcal{T}(S)$ and $\mathcal{T}(S^r)$?

5.3 Ukkonen's Linear-Time Algorithm

Ukkonen's original algorithm is described in terms of finite automata. Although that choice may result in a more compact, concentrated exposition of the algorithm, we prefer to give an explanation without that terminology, as Gusfield [Gus97] does. Ukkonen's idea for computing suffix trees in linear time is a combination of several nice insights into the structure of suffix trees and several clever implementation details. The way we will explain the algorithm is to start off by presenting a brute-force algorithm, and after that introduce Ukknonen's speed-ups so that the linear-time algorithm is achieved. Ukkonen's algorithm computes suffix trees from another data structures, the so-called implicit suffix trees. Computing implicit suffix trees in a straightforward way does not give linear complexity. Special pointers called suffix links are then implemented to speed up tree traversal. In addition, edge-label compression is also applied so that memory space is reduced. Finally, a couple of properties, the halt condition and the fast leaf update rule, will take part to skip redundant insertions in the tree and speed up others.

5.3.1 Implicit Suffix Trees

An **implicit suffix tree** for a string S is a tree obtained from $\mathcal{T}(S\$)$ by performing the following operations:

1. Remove all the terminal symbols \$.

2. From the resulting tree, remove all edges without label.

3. Finally, from the resulting tree, remove all nodes that do not have at least two children.

The implicit tree associated with S will be denoted by $\mathcal{I}(S)$. The implicit tree of $S[1..i]$ is defined as the implicit tree of $S[1..i]\$$; it will be denoted by $\mathcal{I}_i(S)$. When S is fixed, we will just write $\mathcal{I}$ or $\mathcal{I}_i$. Figure 5.5 illustrates the implicit suffix tree construction for string $S\$ = \{\texttt{abcab\$}\}$.

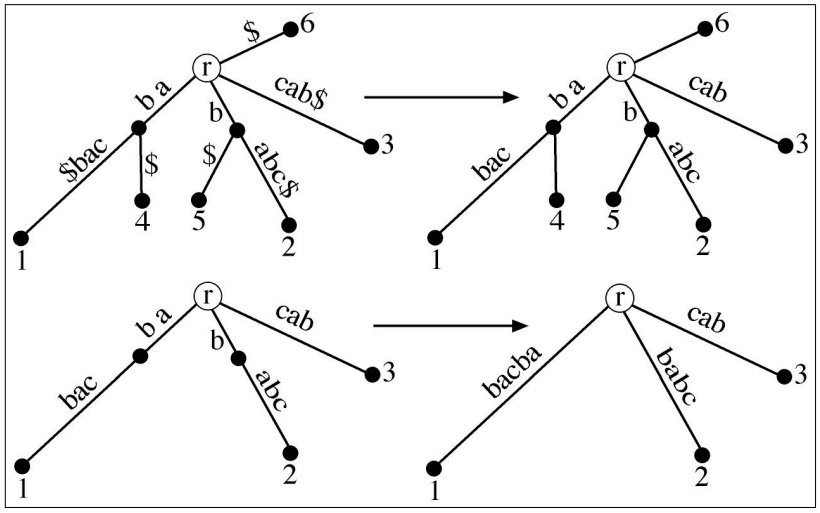

Figure 5.5: Implicit suffix trees.

Implicit trees are introduced because (hopefully) they will require less space, one of the reasons explaining the high complexity of the naive construction of suffix trees. The following result characterizes when an implicit suffix tree possesses fewer leaves than the original suffix tree.

Theorem 5.3.1 *Let S be a string of length s. Its implicit suffix tree will have less than s leaves if and only if at least one of its suffixes is a prefix of another suffix of S.*

Proof: The proof has two parts.

$\Longrightarrow$) Assume that $\mathcal{I}(S)$ has fewer than s leaves. That can only happen because there was an edge (u, i) ending at leaf i with label \$. By property SF2, node u has at least two children and, therefore, path $r \longrightarrow u$ spells out a common prefix for two suffixes of S.

$\Longleftarrow$) Assume that there exists at least one suffix of S (but not of $S\$$) that is a prefix of another suffix of S. Among all nested suffixes, select the one with minimum length. Let $S[i..s]$ and $S[j..s]$ be those suffixes such that $1 \leq i < j \leq s$ and $S[j..s] \sqsubset S[i..s]$. When suffix $S[j..s]$ is inserted in the tree, a new leaf with label \$ is created. This edge will be later removed when the suffix tree is pruned. ∎

In particular, if S ends by a character not found anywhere else in S, this result guarantees that $\mathcal{T}(S\$)$ and $\mathcal{I}(S\$)$ have $s+1$ and s leaves, respectively. Figure 5.7 shows the implicit suffix tree for string $S = \{\texttt{abcabd}\}$, which has 6 leaves, just one less than $\mathcal{T}(S\$)$.

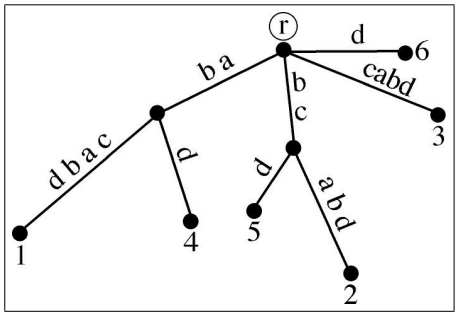

Figure 5.6: An implicit tree with one leaves less than $\mathcal{T}(S\$)$.

Exercises

1 Obtain the suffix trees and their associated implicit suffix trees for strings $S_1 = \{\texttt{abcdefg}\}, S_2 = \{\texttt{aaaaaaa}\}$ and $S_3 = \{\texttt{abcabc}\}$.

2 With regard to string S, what is the degree of the root of $\mathcal{I}(S)$? What is the number of internal nodes of $\mathcal{I}(S)$?

3 Given the following implicit suffix tree, find the suffix tree it comes from.

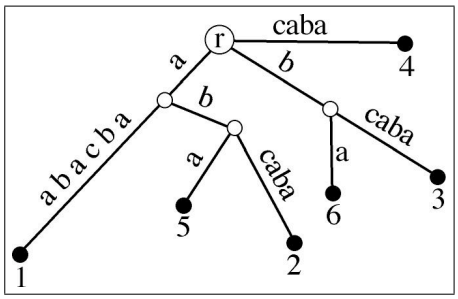

Figure 5.7: Computing suffix trees from their implicit counterparts.

5.3.2 Suffix Extension Rules

Ukkonen's algorithm builds the implicit suffix tree $\mathcal{I}_{i+1}(S)$ from $\mathcal{I}_i(S)$ in an incremental way. In his terminology that step from $\mathcal{I}_i(S)$ to $\mathcal{I}_{i+1}(S)$ is called a **transition**, but it is also known as a **phase** ([Gus97]). Unlike the naive algorithm, Ukkonen's algorithm processes string S by introducing the consecutive prefixes of S $S[1..i]$, for $i = 1$ to $i = s$, one by one. As we will

see later, there is a strong reason to proceed so. Furthermore, each phase is decomposed into suffix extensions. If we are in phase i, Ukkonen's algorithm will take prefix $S[1..i]$ and insert all its suffixes $S[j..i]$, for $j = 1$ to $j = i$, in its order in the tree. Each of these insertions is called a **suffix extension**, which is produced by the following procedure. Assume we are in suffix extension j of phase $i + 1$. The procedure has the following steps:

1. Find the unique path that spells out suffix $S[j..i]$.
2. Find the end of such path.
3. Extend suffix $S[j..i]$ by adding character $S[i + 1]$.

The extension actually needs a more precise description as three cases may arise. The following rules, the **extension rules**, specify the three cases for suffix extension; see Figure 5.9.

Rule 1: Path containing $S[j..i]$ ends at a leaf. Character $S[i + 1]$ is simply added to $S[j..i]$ to produce label $S[j..i]S[i + 1] = S[j..i + 1]$.

Rule 2: Path containing $S[j..i]$ does not end at a leaf. Next character to label $S[j..i]$ is not character $S[i + 1]$. Split the edge and create a new node u and a new leaf numbered j; assign character $S[i + 1]$ to the label of $(u, i + 1)$. The new leaf represents the suffix starting at position j.

Rule 3: Path containing $S[j..i]$ does not end at a leaf. Next character to label $S[j..i]$ is character $S[i + 1]$. Then, $S[i + 1]$ is already in the tree. No action is taken.

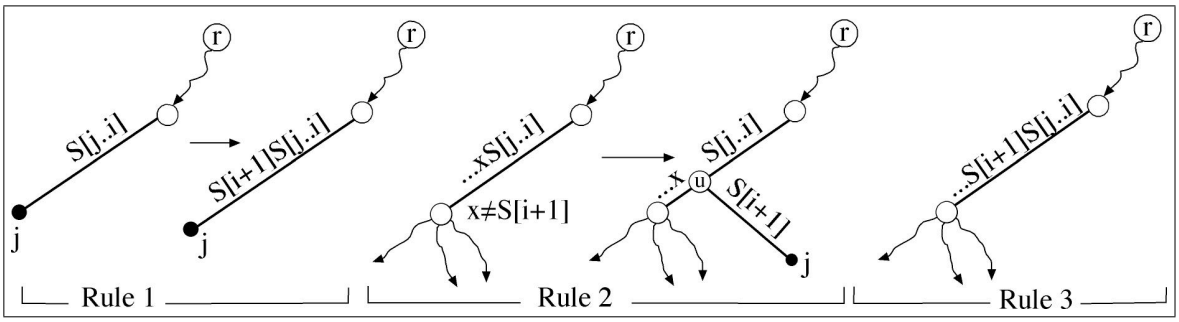

Figure 5.8: Suffix extension rules.

Example. The implicit suffix tree of string $S = \{\texttt{abba}\}$ is displayed in Figure 5.9.

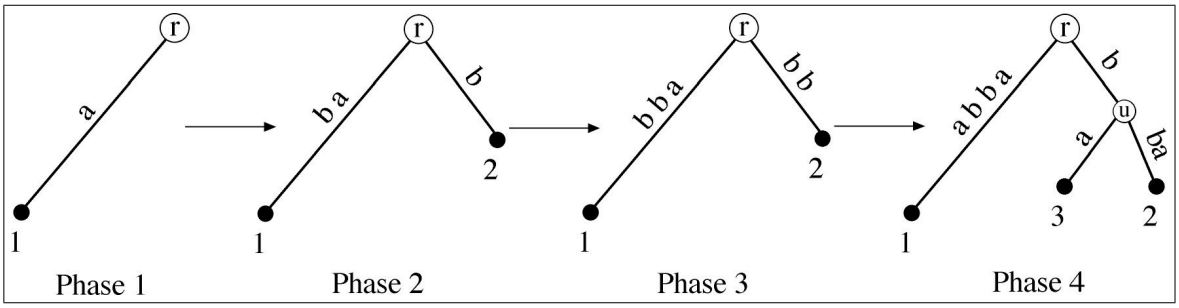

Figure 5.9: Suffix extension rules.

Suffix rule extensions have been traced for this string and the results are in Table 5.1.

Char.	Phase	Extension	Rule	Node	Label
{a}	1	1	2	$(r, 1)$	{a}
{ab}	2	1	1	$(r, 1)$	{ab}
{b}	2	2	2	$(r, 2)$	{b}
{abb}	3	1	1	$(r, 1)$	{abb}
{bb}	3	2	1	$(r, 2)$	{bb}
{b}	3	3	3	$(r, 2)$	{b}
{abba}	4	1	1	$(r, 1)$	{abba}
{bba}	4	2	1	$(r, 2)$	{bba}
{ba}	4	3	2	$(u, 3)$	{a}
{a}	4	4	3	$(u, 1)$	{a}

Table 5.1: Applying suffix extension rules to build implicit suffix trees.

So far, the description of this algorithm may seem ineffective in terms of complexity as compared to the naive algorithm presented earlier. Processing all phases and extensions

already takes $\sum_{i=1}^{s} \Theta(i) = \Theta(s^2)$ time. Moreover, each suffix extension, if performed in a direct way, may take time proportional to the size of the current tree. In total, the complexity may reach $O(s^3)$ and certainly be $\Omega(s^2)$. What is then the point to use this algorithm? We beg the reader to lay to our account and be patient. After all speed-ups are introduced we will see how the complexity is shrunk down to the desirable linear complexity. Waiting will be worth it.

Exercises

1 Build the implicit suffix tree for string $S = \{\texttt{aaba}\}$ and trace its construction as done in the text.

2 Give a string S of arbitrary length such that its implicit suffix tree only has two leaves.

5.3.3 Suffix Links

Suffix links are simply pointers between internal nodes of an (implicit) suffix trees. They will allow us to lower the time complexity of phase i to $\Theta(i)$. This, in turn, will lower the time complexity of the whole algorithm to $\Theta(s^2)$. Additional speed-ups will eventually lead to the desired linear-time algorithm. Suffix links save time on traversing the tree from the root to the suffixes of S.

Let A be an arbitrary substring of S, including the possibility $A = \varepsilon$. Let z be a single character of S. Suppose there are two internal nodes v, w, the former having path-label $\{zA\}$ and the latter having path-label $\{A\}$. A pointer from v to w is called a **suffix link**. The case $A = \varepsilon$ will also be taken into account and its suffix link then points to the root. The root itself is not considered as an internal node and, therefore, no suffix link comes out from it. Typically, zA will be a suffix $S[j..i]$ and A the next suffix $S[j+1..i]$.

Example. Let $S = \{\texttt{aabbabaa}\}$ be a string. Figure 5.10 displays $\mathcal{I}(S)$ along with its suffix links (dashed lines). Tree $\mathcal{I}(S)$ has four suffix links coming from its four internal nodes v_1, v_2, v_3 and v_4. In general, a suffix link between two nodes v, w will be denoted by $v \longrightarrow w$. Do not confuse this notation with paths from the root to nodes, denoted similarly; context will make the difference. Path-label of v_4 is $\{\texttt{ba}\}$ and path-label of v_1 is $\{\texttt{a}\}$; therefore, there is a suffix link $v_4 \longrightarrow v_1$. Analogously, v_1 and v_3 have suffix links to the root because their respective path-labels are one-single character. Finally, there is another suffix link from v_2 to v_3.

The following results will prove that in any implicit suffix tree *every* internal node has one suffix link. This fact is not obvious from the definition itself. The first theorem studies the mechanism for creation of new internal nodes.

Theorem 5.3.2 *Assume the algorithm is executing extension j of phase $i+1$. If a new internal node v with path-label $S[j..i+1]$ is created, then one of the two following situations is true:*

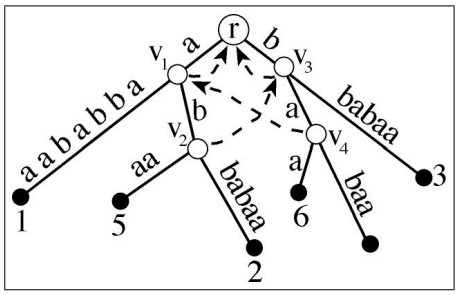

Figure 5.10: Suffix links in the implicit suffix tree of string {aabbabaa}.

1. *Path-label $S[j+1..i+1]$ already exists in the current tree.*

2. *An internal node at the end of string $S[j..i+1]$ will be created in extension $j+1$ of phase $i+1$.*

Proof: New internal nodes are only created when extension Rule 2 is applied. Since the algorithm is in extension j of phase $i+1$, it has to find the end of path $S[j..i]$ and insert v after such path. Let us assume that the end of path $S[j..i]$ is on edge (u_1, u_2). Let us call B the continuation of the path containing $S[j..i]$ on edge (u_1, u_2). Such continuation must exist; otherwise, Rule 2 could not be applied; see Figure 5.11.

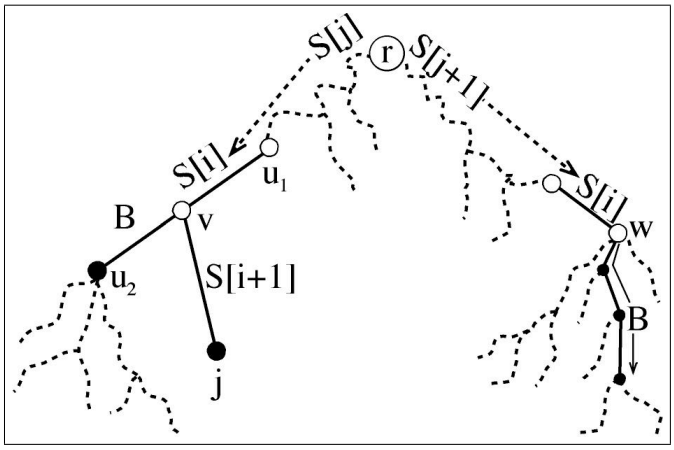

Figure 5.11: Creating a new internal node at extension j of phase i.

By Rule 2 edge (u_1, u_2) is split, internal node v and leaf j are inserted and label $S[i+1]$ is assigned to node (v, j).

Let us now consider extension $j+1$ of phase $i+1$, in which suffix $S[j+1..i+1]$ of prefix $S[1..i+1]$ is added to the current tree. Since $S[j+1..i]$ was already added in extension $j+1$ of phase i, path $S[j+1..i]$ is already found in the tree. Moreover, B continues path $S[j+1..i]$. If B is the only string continuing the path, then a new internal node w will be

created and path $S[j+1..i]$ then ends at an internal node. If there is another continuation, it must be through another branch and, therefore, there must be an internal node already. In this case, path $S[j+1..i]$ also ends at an internal node. ∎

Theorem 5.3.3 *Assume the algorithm is executing extension j of phase $i+1$. Then, any newly created node will have a suffix link by the end of extension $j+1$.*

Proof: The proof will be by induction on the number of phases.

Base case: $\mathcal{I}_1$ is just a one-edge tree and has no suffix links.

Inductive case: Assume that at the end of phase i every internal node has a suffix link. If an internal node is created by extension j of phase $i+1$, by Theorem 5.3.2, it will have a suffix link by the end of extension $j+1$. Finally, note that no new internal node is created in extension $i+1$. This extension just consists of inserting character $S[i+1]$ in the tree. This insertion is always managed by Rule 1 or Rule 3. ∎

By combining Theorem 5.3.2 and 5.3.3 we have proved the existence of suffix links for each internal node. The following theorem states this fact formally.

Theorem 5.3.4 *Let $\mathcal{I}_i$ an implicit suffix tree. Given an internal node v with path-label $S[j..i+1]$, there always exists one node w of $\mathcal{I}_i$ with path-label $S[j+1..i+1]$.*

Given that any internal node v has a suffix link, we will call $s(v)$ the node to which the suffix link of v points to.

Exercises

1 Identify the suffix links in the implicit suffix tree of string $S = \{\texttt{aabcaba}\}$.

5.3.4 Extension Algorithm

Now, we will show how to use suffix links to reduce the time complexity of performing extensions in Ukkonen's algorithm. Assume the algorithm is just starting phase $i+1$. We describe the whole process, which will be called the **extension algorithm**.

- **Extension 1, phase** $i+1$. The first suffix to be inserted in the tree is $S[1..i+1]$. First, suffix $S[1..i]$ must be located and, subsequently, by the suffix extension rules, $S[i+1]$ is inserted. Since $S[1..i]$ is the longest string represented so far in the tree, it must end at a leaf of $\mathcal{I}_i$. By keeping a pointer f to the leaf containing the current full string, this extension can be handled in constant time. Note that Rule 1 always manages the suffix extension from $S[1..i]$ to $S[1..i+1]$.
- **Extension 2, phase** $i+1$. Here Theorem 5.3.4 comes into play. Consider the edge ending at leaf 1. Call v the internal node (or possibly the root) forming that edge and

call α the edge-label of $(v, 1)$. In order to find $S[2..i]$, we walk up from leaf 1 to node v; from there we follow the suffix link of v, which points to node $s(v)$. Once in $s(v)$ we walk down the path labelled α until reaching the end of path $S[2..i]$. Then, we apply the pertinent suffix extension rule and insert $S[i+1]$ accordingly. See Figure 5.12 for an illustration of this process.

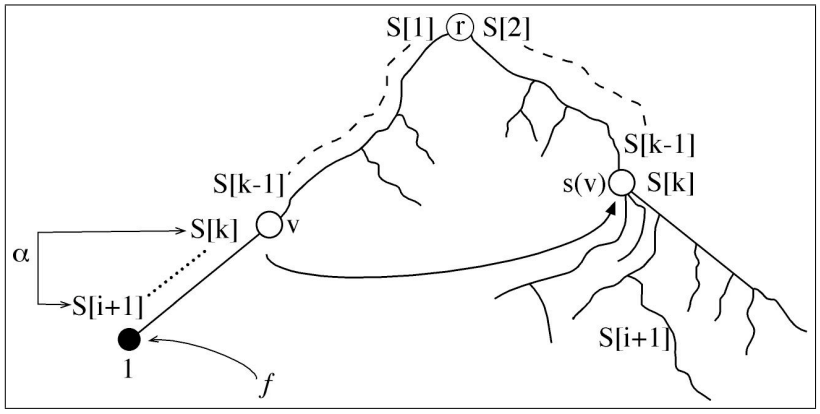

Figure 5.12: Extension algorithm.

Let k be the position such that edge-label α is $S[k..i+1]$. After traversing suffix link $(v, s(v))$, we are at the beginning of another copy of α in *another* subtree of $\mathcal{I}_i$. Because of previous insertions of suffixes, that subtree may not be simply an edge, but we know that there is one path $S[k..i]$ from $s(v)$ at the end of which character $S[i+1]$ will be inserted.

- **Extensions $j > 2$, phase $i + 1$.** The procedure is the same as in extension 2. The only difference is that in extensions $j > 2$, the end of $S[j..i]$ may be at a node having already a suffix link (that is, it is an internal node with degree greater than 2). In this case, the algorithm should follow that suffix link.

Example. Let us consider the implicit suffix tree of string $S = \{\texttt{aabbaacb}\}$. In extension 5, phase 7 suffix $S[5..7] = \{\texttt{aac}\}$ of prefix $S[1..7] = \{\texttt{aabbaac}\}$ is inserted in the tree. In order to do so, suffix $S[5..6] = \{\texttt{aa}\}$ is searched for from the root and after that suffix character $S[7] = \{\texttt{c}\}$ is then finally inserted. The tree at this point, including its suffix links, is displayed in Figure 5.13.

Character $S[7] = \{\texttt{c}\}$ is inserted by virtue of Rule 2, which gives place to node 3. By using the extension algorithm described above we do not need to start the subsequent searches from the root for the remaining extensions. Indeed, for extension 6, where $S[6..6] = \{\texttt{a}\}$ has to be located, we just walk up one node and find that path. Because character $S[7] = \{\texttt{c}\}$ is not found on the only outgoing edge from node 1, Rule 2 must be applied and a new leaf is added. For extension 7, we just follow the suffix link at node 1 and arrive at the root. Character $S[7] = \{\texttt{c}\}$ is not found among the outgoing edges and thus has to be inserted.

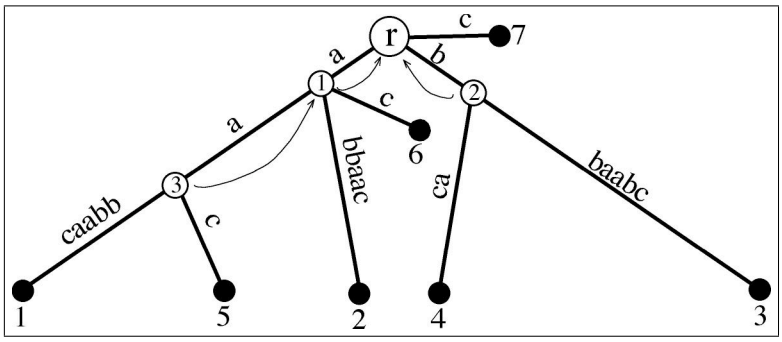

Figure 5.13: Extension 5, phase 7 of Ukkonen's algorithm.

To finish this section, we will introduce one more speed-up for computing extensions, the so-called **skip/count speed-up**. In terms of time bounds, we have not achieved a linear-time bound yet. Even if a proper implementation of suffix links were used, the worst-case time complexity would still be cubic. In each extension a linear number of comparisons in the length of the string is required. The next speed-up will show how to reduce the complexity of walking on the tree. The resulting complexity will be proportional to the number of nodes rather than to the number of characters on the tree edges. The number of nodes is linear (Euler's formula), whereas the number of characters may be quadratic.

Recall that, in general, α will be the edge-label of node (v, j) in extension j of phase $i + 1$; see Figure 5.14. Let c be the length of string α. As we know, there is a path starting at the subtree rooted at $s(v)$ spelling out path α itself. Let us call $c_1, \ldots, c_k$ the lengths of the edges of that path. First, we find the correct outgoing edge from $s(v)$. Let c_1 be its length. We compare c and c_1; if $c > c_1$, then we have to carry on. At this point we set a variable h to $c_1 + 1$. This variable will point to the following character to be searched at each node. At the following node, we check again the condition $c > c_1 + c_2$. If satisfied, the end of path α has not been found. We set h to $c_1 + c_2 + 1$ and search for character h of α among all the current outgoing edges. At a generic node we check the condition $c > \sum_{i=1}^{l} c_i$, where l is the number of nodes already visited. If true, we set h to $\sum_{i=1}^{l} c_i + 1$ and jump to the next node. Eventually, c will be less or equal than $\sum_{i=1}^{k} c_i$. When that happens, we can safely state that the end of path α is character $\sum_{i=1}^{l} c_i - c$ of the current edge. Figure 5.14 illustrates this description of the algorithm.

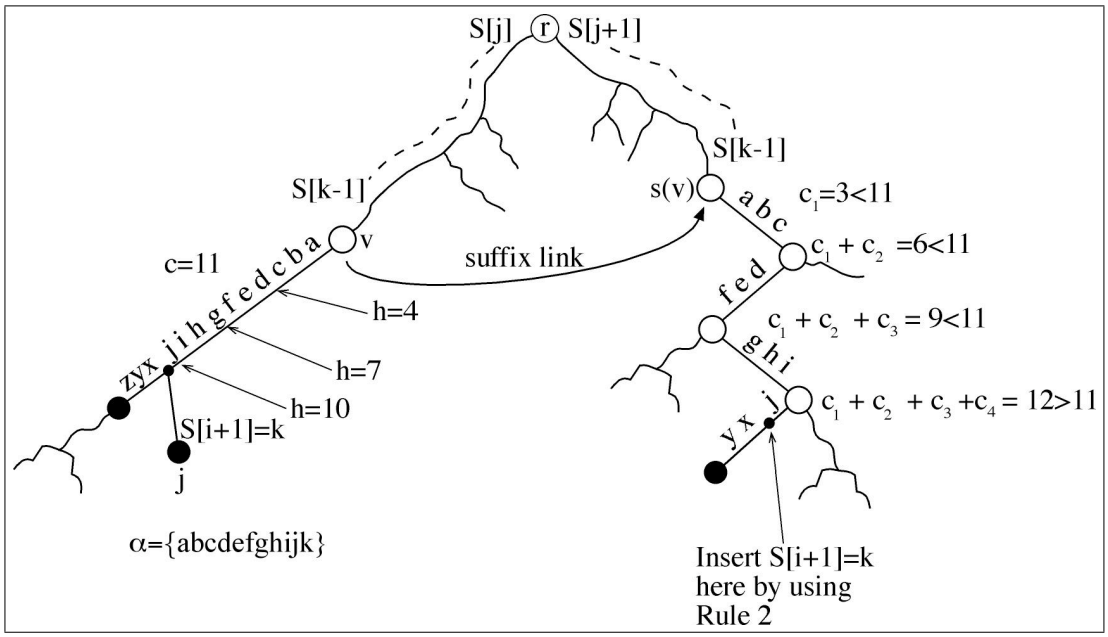

Figure 5.14: The skip/count speed-up.

For this speed-up to work properly two operations have to be implemented so that they take constant time: (1) retrieval of the number of characters on each edge; (2) extraction of any character from S at any given position.

We call the **node-depth** of a node v the number of nodes on the path from the root to v. This definition will be used in the proof of the following theorem.

Theorem 5.3.5 *Let $(v, s(v))$ be a suffix link traversed in some extension j, phase $i+1$. Then node-depth of v is greater than the node-depth of $s(v)$ at most by one.*

Proof: Consider the paths from the root $S[j..i+1]$ and $S[j+1..i+1]$ and an internal node v in path $S[j..i+1]$. The key of this proof is to establish that every ancestor of v is linked through suffix links to a unique ancestor of $s(v)$ that is an internal node. By Theorem 5.3.4, this is true for all internal nodes except for the first character of path $S[j..i+1]$. This first character is linked to the root. By the definition of suffix link, no two internal nodes in path $S[j+1..i+1]$ can receive the same suffix link. From this correspondence between internal nodes on both paths, it follows that the difference between the node-depth of v and the node-depth of $s(v)$ cannot be greater than one. ∎

Now we will introduce the definition of **current node-depth**. This definition will be used to prove that the skip/count speed-up actually computes any extension in $O(s)$ time. The current node-depth is the depth of the node currently being processed by the algorithm.

Theorem 5.3.6 *If the skip/count speed-up is implemented, any phase of Ukkonen's algorithm takes $O(s)$ time.*

Proof: In a particular extension the algorithm does the following: (1) starts at an edge; (2) it walks up to a node; (3) it follows its suffix link to another node; (4) and from there it walks down until reaching the point where inserting a character by applying the proper suffix extension rules. Steps (1), (2) and (3) take constant time as seen in the discussion above. As for step (4), we note that the number of nodes in the down-walk is bounded by s and at each node, because of the skip/count speed-up, we do a constant amount of work.

It remains to prove that over the whole phase the complexity is still linear in s. We will prove that claim by analysing how the current node-depth varies as the algorithm is executed. Clearly, the complexity is proportional to number of times the current node-depth is changed. When an up-walk is performed the current node-depth just decreases by one; after following the suffix link the current node-depth is decreased at most by one. In the down-walk step the current node-depth is increased by some amount. Over the whole phases that amount is not greater than the height of the implicit suffix tree, which is $O(s)$. Therefore, the total amount of work of computing the phases in Ukkonen's algorithm is $O(s)$. ∎

As a consequence of the previous theorems we can state the following result, which is actually the first decrease of the cubic complexity.

Theorem 5.3.7 *Ukkonen's algorithm can be implemented with suffix links to run in $O(s^2)$ time.*

Exercises

1 Compute the implicit suffix tree for string $S = \{\texttt{abababababaa}\}$ and check the order in which the internal nodes are created. Can it, then, be guaranteed that internal nodes are created in increasing order of depth in each phase?

2 Show a string S whose suffix tree construction takes quadratic time, if the skip/count speed-up is not used.

Finally, for arbitrary-length strings, choose an even number s and just set S as

$$\{(\texttt{ab})^{\frac{(s-2)}{2}}\texttt{a}(\texttt{ab})^{\frac{(s-2)}{2}}\texttt{b}\}.$$

5.3.5 Edge-Label Compression

There is an intrinsic difficulty that does not allow us to lower the complexity of computing suffix trees to linear and that is the amount of space required to store edge-labels in the tree. In the worst case, an $\Omega(s^2)$ amount of memory may be required to store the information during the computation of intermediate implicit suffix trees. For example, string $S = \{\texttt{abcdefghijklmnopqrstuvwxyz}\}$ has a tree with a root with 26 outgoing edges. The total sum of the edge-labels is proportional to s^2. Even for finite-sized alphabet strings suffix trees using quadratic memory are relatively easy to find; see the exercises.

An alternate labelling scheme, called **edge-label compression**, is needed to actually shrink down the complexity of the algorithm. The algorithm has a copy of string S at any moment during its execution. We will replace the label (substring) of each edge by a pair of indices pointing to the initial and final positions in string S. By doing so, each edge only keeps a constant amount of memory. Since the total number of edges is $\Theta(s)$, the storage of the edge-labels has been reduced to $O(s)$.

Figure 5.15 shows an implicit suffix tree of string $S\{\texttt{aabbaabb}\}$ with the ordinary labels and the new labels.

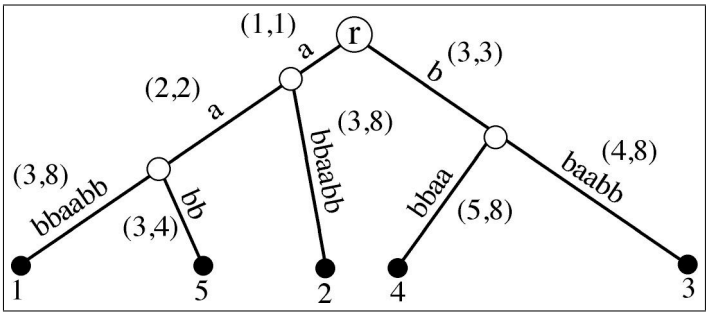

Figure 5.15: Edge-label compression.

It remains to address the question of how the suffix extension rules are applied with those indices.

1. Rule 1. If an edge has label (j, i), adding character $S[i+1]$ is just updated to $(j, i+1)$.
2. Rule 2. Let (u_1, u_2) be the edge where the new internal node will be inserted. Let (j, i) be its label and k be the position after which the new node v is created. Rule 2 then creates three new edges: (u_1, v) with label (j, k), (v, u_2) with label $(k+1, i)$, and node (v, j) with label $(i+1, i+1)$.
3. Rule 3. No change in the edge-label is made.

Therefore, the suffix rule extensions works with the new label scheme.

Exercises

1 Give an infinite family of strings such that the total length of the edge-labels is $\Omega(s^2)$, where s is the length of the string. Suppose that the alphabet is finite.

5.3.6 Phase Algorithm

Finally, we will introduce two more speed-ups, namely, the **halt condition** and the **fast leaf update rule**. These two speed-ups are heuristics that deal with the interaction of successive

phases. It could seem that in an given phase its extensions are built by applying the suffix extensions rules in an unpredictable manner. However, this is not the case. Application of suffix extension rules actually happen in order, Rule 1 being the first to be applied, then Rule 2 and finally Rule 3. This particular fact, if implemented correctly, will allow the algorithm to run in linear time. This part of the algorithm is called the **phase algorithm**.

The halt condition[1] reads as follows: If Rule 3 is applied in extension j of phase $i+1$, then it will also be applied in the next extensions until the end of the current phase. This observation is easy to prove. Suffix $S[j..i+1]$ is inserted by locating the end of path $S[j..i]$ and then inserting character $S[i+1]$ according to the suffix extension rules. If $S[j..i+1]$ is already in the tree, that means that there are at least two identical substrings in S occurring at different positions. If substring $S[j..i+1]$ is already in the tree, so will strings $S[j+1..i+1], \ldots, S[i+1..i+1]$.

Example. Let $S = \{\texttt{aabacaabac}\}$ be a string. Assume that the algorithm is in phase 9 and extension 6. The prefix $S[1..9] = \{\texttt{aabacaaba}\}$ is being processed; suffixes $S[1..9] = \{\texttt{aabacaaba}\}$ to $S[5..9] = \{\texttt{aaba}\}$ have already inserted in the tree. Now, the algorithm is going to insert suffix $\{\texttt{aaba}\}$. This suffix already appeared at position 1; hence, Rule 3 correctly detects the situation when inserting character $S[9] = \{\texttt{a}\}$ (in boldface) after path $S[6..8] = \{\texttt{aba}\}$; see Figure 5.16. By the halt condition, all the remaining suffixes are already in the tree and the current phase is terminated.

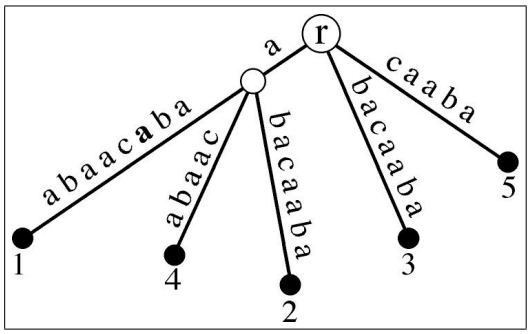

Figure 5.16: The halt condition.

Therefore, once Rule 3 is applied, in order to save time, the algorithm should jump to the next phase. In this case, those phases are said to be built **implicitly**. We call **explicit extensions** are those actually built by the algorithm (like the ones managed by Rule 2).

The second speed-up, the fast leaf update rule[2], takes advantage of the fact that, once leaf j is created, it will remain a leaf until the end of the execution of the algorithm. Note that the suffix extension rules do not schedule any mechanism to modify leaf labels. Hence, once a leaf is numbered j in phase $i+1$, Rule 1 will always apply to extension j of the subsequent phases (from phase $i+1$ to phase s).

[1] In [Gus97] this condition is called with a more striking name, a show stopper.

[2] Gusfield [Gus97] this rule receives the rather long name of "once in a leaf, always in a leaf."

A clever application of this speed-up leads to an essential reduction of the running time. Let us trace Ukkonen's algorithm and find out how to obtain such reduction. The algorithm starts with phase 1. This phase just consists of inserting a character $S[1]$ in an empty tree. We just apply Rule 2 and create the root and a leaf numbered 1 with edge-label $\{S[1]\}$. In the next few phases, more suffixes are added to the tree. Note that the first few insertions are always performed by applying Rule 1. Moreover, after a few insertions through Rule 1, Rule 2 is necessarily applied and, finally, either the phase is done or Rule 3 is applied. Why is Rule 1 first applied when a new phase starts? Assume we are about to start phase $i+1$; further assume that in phase i the implicit suffix tree $\mathcal{I}(S)$ has already k leaves. Consider all the paths from the root to each of those k leaves. These suffixes can only be updated by Rule 1.

Suppose that after extension l of phase $i+1$, we have to apply Rule 2. By the extension algorithm, Rule 2 is applied until either the root is reached or Rule 3 is applied. Rule 1 cannot be applied in between. Otherwise, character $S[l+1]$ would be a leaf edge and this would have been updated in an extension prior to extension l. Therefore, when running the algorithm, there are l implicit suffix extensions through Rule 1, followed by some explicit extensions through Rule 2, say until extension r, and finally Rule 3 is applied from extension $r+1$ to extension $i+1$. Since the number of leaves between two consecutive phases can only be constant or increase, value l is non-decreasing. As stated before, l is equal to the number of current leaves in the tree. Value r is equal to l plus the number of current internal nodes.

In order to take advantage of this situation the algorithm must keep a variable r that stores the last time Rule 2 was applied in phase i. Phase $i+1$ can start in extension r since the $r-1$ previous extensions are all implicit.

Example. Consider again string $S = \{\texttt{aabacaabacx}\}$. In Table 5.2 the first few values r for the phase algorithm are represented.

Phase	Suffix	Value r	Rule 3
Phase 1	{a}	$r = 1$	No
Phase 2	{aa}	$r = 1$	Yes
Phase 3	{aab}	$r = 3$	No
Phase 4	{aaba}	$r = 3$	Yes
Phase 5	{aabac}	$r = 5$	No
Phase 6	{aabaca}	$r = 5$	Yes

Table 5.2: Tracing the values of variable r in the phase algorithm.

Note that from this point on the number of implicit extensions will be at least 5. Figure 5.17 shows the implicit suffix tree after phase 5 is completed.

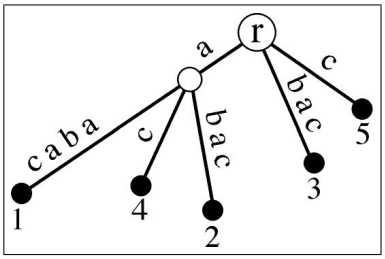

Figure 5.17: The fast leaf update rule.

It only remains to detail how the implicit extensions of Rule 1 will be carried out. If the algorithm actually access to each leaf edge to do so, it will take linear time per phase. Instead of it, we will replace each leaf edge $(p, i+1)$ by (p, e), where e is a variable containing the current phase. By updating that variable at the beginning of each phase all implicit extensions are performed in constant time.

Example. In the previous example, phase 6 starts by implicitly building extensions 1 to 5. If the variable e is kept throughout the algorithm, those extensions are built by setting $e = 6$. That would imply adding character $S = \{\texttt{c}\}$ at the end of the all leaf edge-labels. This updating takes constant time.

Exercises

1 Show a string S whose suffix tree construction takes quadratic time, if the fast leaf update rule speed-up is not used.

2 Assume the algorithm is starting to process phase $i + 1$.

(a) Describe a string such that no extension will apply Rule 2 after the extensions managed by Rule 1.

(b) Describe a string such that all extensions will apply Rule 2 after the extensions managed by Rule 1.

5.3.7 Generating $\mathcal{T}(S)$ from $\mathcal{I}(S)$

To fully finish the description of Ukkonen's algorithms we need to detail how to obtain the *original* suffix tree $\mathcal{T}(S)$ from the *implicit* suffix tree. Add the termination character \$ to S and run Ukkonen's algorithm for the last time. Since the termination character is not in S, no suffix is now a prefix of any other suffix. Therefore, each suffix will end in a leaf. In other words, the degree-1 nodes that were deleted when creating the implicit suffix trees are put back in the tree. It is also necessary to replace variable e by its true value s.

5.4 Chapter Notes

We have seen how to construct suffix trees by using Ukkonen's algorithm. Suffix trees belong to a general class of data structures specialized in storing a set of strings so that string operations are performed quickly. Other data structures related to suffix trees are tries, keywords trees, radix trees or Patricias [Sed90], [GBY91], [Wik09e]. Morrison [Mor68] was the first to introduce Patricia tries as an index for searching in marked-up text.

The goodness of suffix trees resides in the way it exposes the internal structure of a string. Such representation of the string structure allows solving many problems in string pattern recognition. In the next chapter, we will see how to easily solve the exact string matching problem (SMC), the substring problem for a database of patterns, the longest common substring of two strings, computation of palindromes and the DNA contamination problem. There are many other problems that can be solved via suffix trees; see [Gus97] and [Apo85] and the references therein for more information.

Compared to other algorithms, suffix trees are suitable when the text is fixed and known, and the pattern varies (query mode for the pattern). This is a very common and important situation appearing mainly in Bioinformatics. Think of the text as a large biological database in which many different DNA sequences are searched in. To mention another field, in Music Technology, the text is usually a big database of musical excerpts grouped by composer or gender. A piece of music is searched in the text under several similarity criteria with the aim of attributing authorship or classifying the piece.

When the text is not known or both the pattern and the text are presented at the same time algorithms like the Karp-Morris-Pratt or Boyer-Moore algorithms are better than suffix trees.

As discussed in the outset, suffix trees are not widely taught, in spite of its relevance. There is a big difference between Ukkonen's original paper [Ukk95], who described his algorithm in very abstract terms, and Gusfield's book, who wrote an accessible account of Ukkonen's algorithm. Nevertheless, Gusfield's account has not made Ukkonen's algorithm to enter in the main algorithms and data structures texts. The following list of references tries to facilitate the access to resources where nice descriptions of suffix trees and its computation can be found. Allison [All09] built a web page where he explains how to construct suffix using Ukkonen's terminology, but in a more approachable way. His web page has an application to demonstrate how suffix trees are built. The web page [Wik09f] contains a very

complete list of the applications of suffix trees. Lewis [Lew09] described several applications of suffix trees to Computational Biology; his paper is quite self-contained and points to many other interesting sites. Giegerich and Kurtz reviewed in [GK97] the linear time suffix tree constructions by Weiner, McCreight, and Ukkonen. This is a very recommendable paper to gain insight in the suffix tree construction from different approaches. Tata et al. [THP05] address the problem of constructing suffix trees when the text is a very large database. Here, due to the scope of these notes, we have not tackled with the many subtleties involved in the actual programming of suffix trees. The probabilistic analysis of the construction of suffix trees is dealt with in the papers [Szp93b] and [Szp93a] by Wojciech Szpankowski. Grossi and Italiano [GI93] wrote a survey on suffix trees and its applications.