CS5238 Combinatorial methods in bioinformatics 2002/2003 Semester 1

Lecture 4: Suffix Tree and Suffix Link- August 30, 2002

Lecturer: Wing-Kin Sung *Scribe: Huang Xiao Ning , Pan Yu, Song Ya Xiao, Wang Yun Yan, Yin Jun*

4.1 Suffix Tree

Suffix tree is a fundamental data structure for combinatorial pattern matching[1]. In this section, we will give a brief introduction to suffix tree. First, we define the suffix tree. Then we show some simple applications of it. And at last we describe Ukkonen's algorithm to construct the suffix tree.

4.1.1 What is a Suffix Trie

Suffix trie is a tree containing all possible suffices of a string S where each edge is labelled by a character, e.g. $S = acacag\$$, where \$ means the end of S. For $S = acacag\$$, all possible suffices are in Table 4.1.

	Suffix
1	$acacag\$$
2	$cacag\$$
3	$acag\$$
4	$cag\$$
5	$ag\$$
6	$g\$$
7	\$

Table 4.1: Suffices of String S

The suffix trie built according to the former table is shown in Figure 4.1. Every possible suffix is a path from the root to the leaf.

4.1.2 Suffix Tree

1. Definition:
   A suffix tree is the compacted trie of all the suffices of a string S, contracting nodes with only one child. The suffix tree made from the former suffix trie is shown in Figure 4.2. In this figure, the path-label of node V is "aca"; and "ca" is an edge label; "g\$" is a leaf edge for leaf node 4.

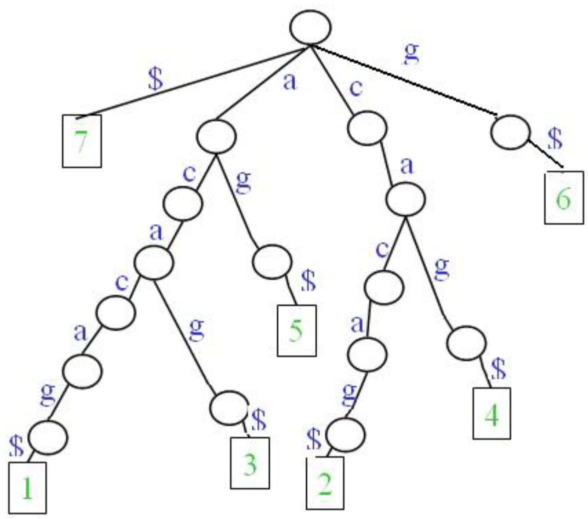

Figure 4.1: A suffix trie

2. A suffix tree can be stored in O(nlogn)-bit space :
   why ? suffix tree has exactly n leaves and at most 2n edges. The label of each edge can be represented using 2 indices. Thus,suffix tree can be stored in $O(nlogn)$ bits. For example, in Figure 4.4, the edge "a" is represented by (1,1); the edge "ca" is represented by (2,3); the edge "cag" is represented by (4,7). We should notice that the end of every leaf edge should be 7, the last index of S. Thus for leaf edges we only need to store the start index. E.g. the edge "cag" is represented by (4) .

3. Generalized suffix tree:
   A generalized suffix tree is a suffix tree for more than one string. e.g. $S_1 = acgat\sharp$, $S_2 = cgt\$$. The generalized suffix tree of them is shown in Figure 4.5. the leaves with underlined labels are the leaves of S_2, and the rest of the leaves are the leaves of S_1.

4.2 Some Simple applications of suffix tree

After introducing how to construct a suffix tree, we show that suffix tree is a very powerful tool to solve many problems. Here we list several simple applications of suffix tree to show how powerful a suffix tree is.

1. Find all occurrences of a substring Q in S.
   Such query is often issued in the area of bioinformatics, data-mining, etc. An occurrence of a substring Q in a string S is a list of all the positions where Q starts. For example, in string $S = acacag\$$,we want to find the occurrence of substring $Q = aca\$$. It is obvious that the start place of suffix

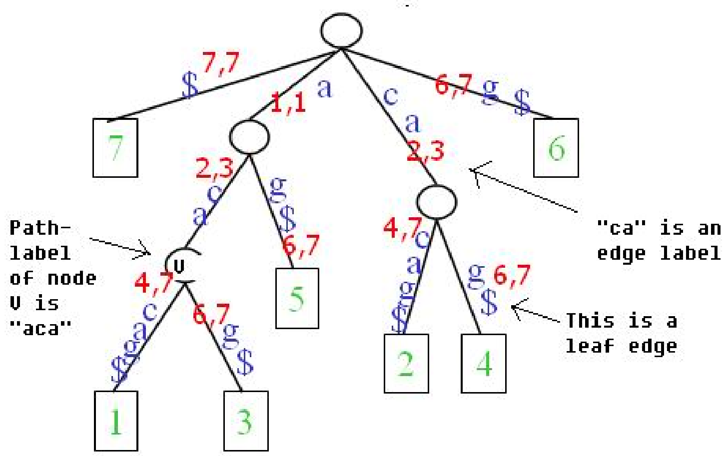

Figure 4.2: A suffix tree

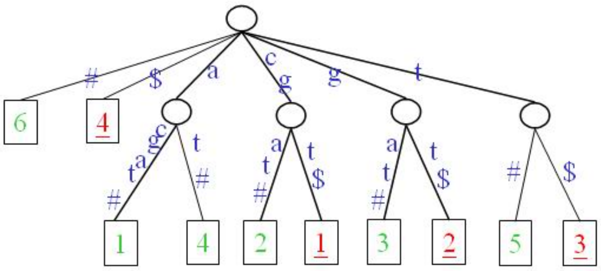

Figure 4.3: The generalized suffix tree

<u>*aca*</u>*cag*\$ is the start place of Q; the start place of <u>*acag*</u>\$ is the start place of Q too. So if we find out the occurrence of *acacag*\$ and *acag*\$, we can find out the occurrence of Q. The occurrences can be found easily using a suffix tree. For time execution, Since a node can be found by traversing at most $\mid Q \mid$ edges and occ(total number of occurrences) leaves are visited, so the time complexity is O($\mid Q \mid$ + occ).

2. Find the longest repeated substring in S.
   In order to find the longest repeated substring in S, we just need to find the deepest internal node in the suffix tree of S. For example, the longest repeated substring of $S = acacag$ can be found by locating the deepest internal node in the suffix tree of S as show in Figure 4.4. From the figure, the deepest internal node in the tree is the father of leaf 3 and leaf 1, so the longest repeated substring is from the root to this node: *aca*.

3. Find the longest common substring of two or more sequences.
   For example, consider two strings S_1 and S_2, we can find the longest common substring of S_1 and S_2 by utilizing the generalized suffix tree for S_1 and S_2 as follows: First, build generalized suffix tree for $S_1\sharp$ and $S_2\$$; Then, mark each internal node with leaves representing suffices of both S_1 and S_2. Finaly, report the deepest marked node.

4.3 Construction of Suffix Tree

In the previous section, we have introduced how powerful the suffix tree is. In this section, the algorithm for constructing the suffix tree is presented. In fact, a suffix tree can be constructed in O(n) time. Weiner's algorithm (1973)[1] requires linear time for constant size alphabet but much space is needed. McCreight's algorithm (1976)[2] requires linear time for constant size alphabet and quadratic space. Later in 1995, Ukkonen presented an online algorithm which requires linear time for constant size alphabet. Contribution of this algorithm is that less space required. Ukkonen's algorithm is discussed in detail here.

This section presents a detailed algorithm to construct the suffix tree. Consider $S = s_1s_2..s_n$ where $s_n = \$$. First, we present a straightforward algorithm, whose time complexity is $O(n^2)$. We initialize the tree with only a root and include suffices into the tree one by one for $i = n$ to 1. Though such algorithm is straightforward, the time is $O(n^2)$, which is costly. In this section we introduce another algorithm: Ukkonen's Algorithm[3]. Before detailed this algorithm, we should define the concept of Implicit suffix tree.

An implicit suffix tree is obtained by removing \$ from edge labels first. Then,

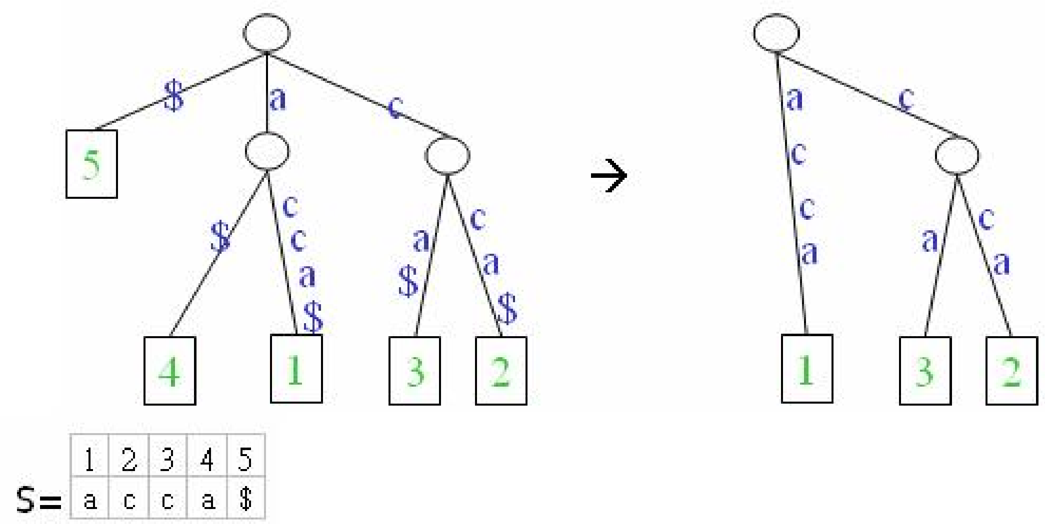

Figure 4.4: Obtain Implicit Suffix Trie

edges with no label are removed. At last removing nodes with only one child. An example is given in Figure 4.4. Ukkonen's Algorithm[3] is described as follow.

Denote I_i be the implicit suffix tree for S[1..i]. At first we should construct I_1. It is easy and explicit to do so. Then the idea is to construct I_{i+1} from I_i for $i = 1$ to $m - 1$. The process is illustrate in Figure 4.5. For phase i(constructing I_{i+1} from I_i), there are i+1 extensions. Each extension is done by locating the end of the path from the root labelled $S[j..i]$ and extends the path with character S[i+1]. Such extension should obey one of the following 3 rules:

1. Rule 1: If $S[j..i]$ ends at a leaf, S[i+1] is appended to the end of the label on the leaf edge.
2. Rule 2: If no path from the end of $S[j..i]$ starts with character S[i+1], create a new leaf edge labelled with character S[i+1].This is the only extension that will increase the number of leaves in the tree.
3. Rule 3: If some path from the end of $S[j..i]$ starts with character S[i+1], then we do nothing on the tree. The reason is that the ending S[i+1] already appears.

The process of constructing I_4 from I_3 (phase 3) for $S = acca\$$ is illustrated in Figure 4.5. If we evaluate Phase i, the process of constructing I_{i+1} from I_i, we can conclude several observations. Suppose rule 3 is applied to extension j, the path labelled S[j..i] in the current tree I_i must continue with character S[i+1].

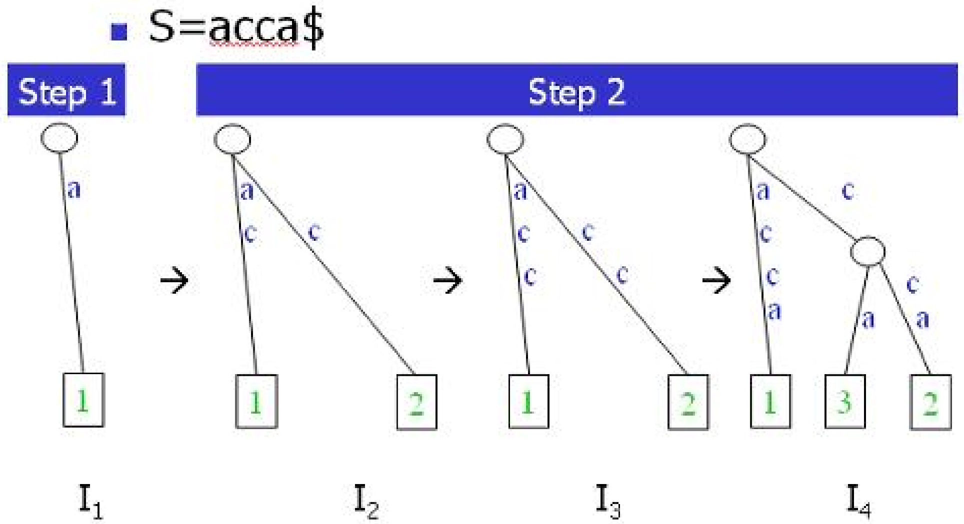

Figure 4.5: Illustration of the Whole Algorithm

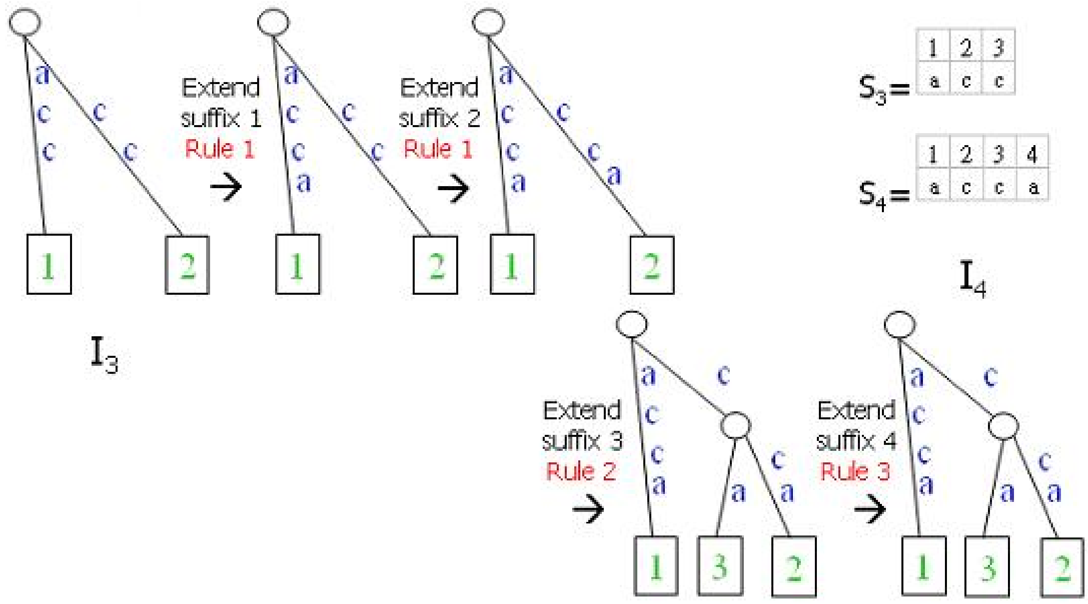

Figure 4.6: Illustration the construction of I_4 from I_3

It implies that, for $j' \geq j$, the path labelled $S[j'..i]$ also continue with character $S[i+1]$. Thus rule 3 will be applied for extension j'. Therefore, if we apply rule 3 for extension j, then rule 3 will be applied for extension j' for $j' = j, .., i+1$. Thus, we need to do nothing for $j' = j+1, .., i+1$. Conclusively, once we apply rule 3 for some extension j in Phase i, we can stop. Such conclusion reduce the cost of the work.

Another observation is that once we form a leaf for a suffix in I_i, the leaf appears in $I_{i'}$ for i'>i.The reason is quite simply. If we trace the process of the construction, we find that there is no rule for removing leaves. So such observation is correct.

Based on previous observations,we can reach a rather useful conclusion: In Phase i, let j_i be the last extension using rule 1 or rule 2. In other word, extension j' is based on rule 3 for j'$>j_i$. Then, in Phase i+1, extension j' for j'$\leq j_i$ is based on rule 1. Furthermore, Extension j' for $j' > j_{i-1}$, cannot be based on rule 1.

Then we can improve the algorithm. For phase i ,when doing extension j for $j \in [1, j_{i-1}]$, these extensions are based on rule 1. So we can do nothing. When $j \in [j_{i-1}, i+1]$, we first find the end of the path labelled $\beta = S[j..i]$ and extend it with character S[i+1] by applying rule 2 or 3. If the extension is by rule 3,it means that it is the end of this phase.Thus, we only need to set $j_i = j-1$ and end phase i.
Construction using this algorithm improves the whole process. Detailed description is showed below:

Phase 1: we compute extension $1..j_1+1$.
Phase 2: we compute extension $j_1+1..j_2+1$.
Phase 3: we compute extension $j_2+1..j_3+1$.
...
Phase i: we compute extension $j_{i-1}+1..j_i+1$.
...
So, we do at most 2n extensions in all phases. Because for each extension j, it takes O(n) time to find the path labelled S[j..i]. Thus, the total time is O(n^2). In the next section, the process can be further accelerated using suffix link.

4.4 Suffix Link

4.4.1 Idea of Suffix Link

Though we can greatly reduce work by the above two observations, the extension steps based on rule 2 and rule 3 in each phase are still time-consuming. In Phase

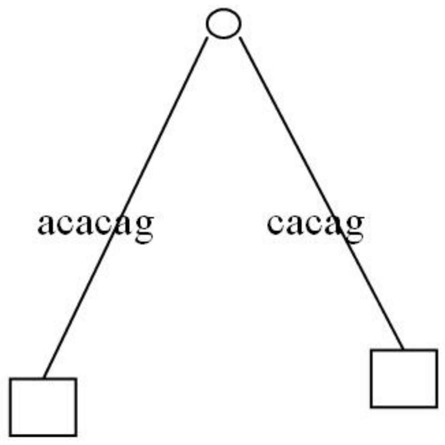

Figure 4.7: Suffix Tree After Phase 4 Extension 2

i for Extension $j > j_{i-1}$, we need to locate $s[j..i]$ by walking down the tree from the root, and then determine whether the extension should be based on rule 2 or rule 3. Walking down the tree is very time consuming. For example, suppose we are doing Extension 3 in Phase 5 for the string "acacag". Figure 4.7 shows the suffix tree after Phase 4. At that time, $j_4 = 2$.

In Phase $i = 5$, Extension $(j_{i-1}+1) = 3$, we have to locate $s[j_{i-1}+1..5]$="aca" by walking down the tree from the root, and check whether the suffix "acag" should be added to the tree or already exists in the tree. Since "acag" does not exist in the tree, we add the extension based on rule 2. Then, we have to check the suffix "cag" to see if it is still based on rule 2. Similarly, we walk down the tree from the root, check the suffices "cag", "ag" and "g", and then apply rule 2 to them.

To speed up the process, we make the third important observation on the suffices needed to be checked in a phase which rule is suitable to it. In the above example, these suffices are "acag", "cag", "ag" and "g", in which the neighbor suffices share the same suffix except that the later suffix misses the first letter compared to the former suffix. In other words, there is some relationship between these suffices. Below we introduce a new data structure suffix link, which takes advantage of the relationship to speed up the extensions.

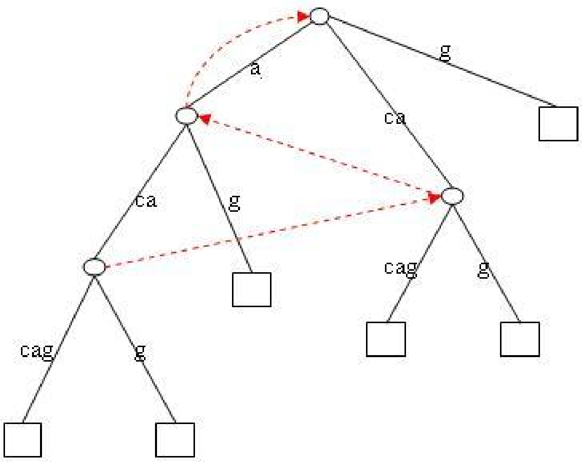

Figure 4.8: Suffix Tree and Suffix Links of acacag

4.4.2 Definition of Suffix Link

Suffix link is defined as follows: For an internal node v with path-label $x\alpha$, if there is another node s(v) with path-label α, then we create a suffix link from v to s(v).

Lemma 4.1 *For an implicit suffix tree T_i for $s[1..i]$, each internal node must have a suffix link, assuming that the root has a suffix link to itself.*

Proof: Consider any internal node v of T_i with the path-label xα, which is the common prefix of s[j..i] and s[j'..i]. So α is the common prefix of s[j+1..i] and s[j'+1..i]. Thus, there exists an internal node v' of T_i with the path-label α. We have a suffix link from v to v'. The lemma follows. ■

For every internal node v, if we represent a suffix link using a dashed line, from v to s(v), Figure 4.8 shows the suffix links for internal nodes in the implicit suffix tree of “acacag”.

4.4.3 Creation, Maintenance and Taking Advantage of Suffix Link

Suffix links must be created first before they can speed up the extensions. Suppose an extension of a phase is based on rule 2, a new internal node will be created. However, the suffix link of this newly created node doesn't exist at that time. Therefore, we need a method to create the suffix links.

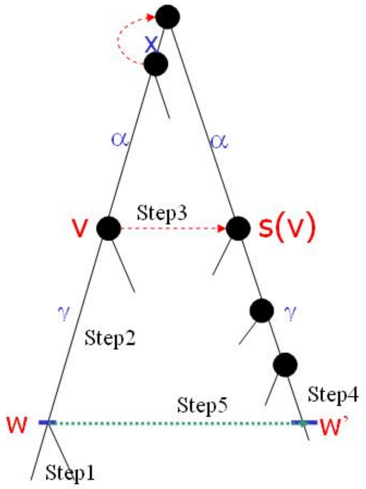

Figure 4.9: Step (1) - (5) of Creating Suffix Links

Assume we have maintained suffix links for all internal nodes before Phase i. At the beginning of Phase i, we are at position w, which is the end of $s[j_{i-1}+1..i]$, where rule 2 or rule 3 is used last time. If rule 2 is used last time, w is an internal node. If rule 3 is used last time, w may be at the middle of an edge. Then, for each subsequent Extension j, we do the following steps:

Step (1): Check whether rule 2 or rule 3 should be applied in Extension j, and apply the corresponding rules to Extension j.
Step (2): Find the first internal node v at or above w, which has suffix link. (We assume that the root has a suffix link to itself)
Step (3): Go through the suffix link of v to s(v)
Step (4): Go down zero or more nodes until we find the end of $s[j+1..i]$, says, w'.
Step (5): If $w \neq v$, it means w does not have a suffix link. Create a suffix link from w to w'
Step (6): If Extension j is the last extension based on rule 2 or is based on rule 3, set $j_i = j-1$ and go to Phase i+1. Otherwise, set w=w', and go to the next extension, Extension j+1.

Figure 4.9 shows how to perform the step (1) - (5).

Even if we have to spend time creating suffix links for newly created internal nodes, the existing suffix links still help to speed up the extensions. Without them, after we finish an extension, we have to return to the root and then locate a new position from the root. The existing suffix links prevent us from unnecessary going up and down.

Let's look at an example: Phase 5 of creating suffix links for the implicit suffix tree of "acacag". At the beginning of Phase $i = 5$, $j_{i-1} = 2$, we have already got the tree shown in Figure 4.10.

In Extension 3, we find the suffix "acag" does not exist in the tree, so we apply rule 2: split the edge "acacag" at point w, and create an internal node w and a new leaf, the edge between them is "g". Then we try to find the first internal node at or above w who has a suffix link, in this case, the root. We assume the root has a suffix link to itself. Going though the suffix link of the root, we arrive at the root again. Then we go down several nodes and find point w', which is the end of $s[j+1..i] = s[4..5]$. Since w does not have a suffix link, we create a suffix link from w to w'. (See figure 4.10) Extension 3 is not the last extension of Phase 5, we set w=w', and go on with Extension 4.

Extension 4 is quite similar to Extension 3. We apply rule 2, split the edge at w, create an internal node and a new leaf, and mark the new edge "g". Then we go up and find the root as the first internal node who has a suffix link. Going through the suffix link and going down, we find the w' as the end of s[5..5]. Then we create a suffix link from w to w', set w=w', and go on with the next Extension. Figure 4.10 shows the result after Extension 4.

Extension 5 is also similar. Figure 4.10 shows the result after Extension 5.

Extension 6 is based on rule 2. Since w is the root in this extension, who has already a suffix link, no suffix link is created in this extension. See figure 4.10

Situation is better if the internal node w is located at an existing internal node who has already got a suffix link. Let us see another example: Suppose we are going to do the last phase of building the suffix tree of the string "acacagacat", when $j_{i-1} = 6$ at that time. Figure 4.11 shows the suffix tree before Extension 6:

In this case, because w has already the suffix link, the first internal node at or above w is w itself. We go through the suffix link. We need not go down a number of nodes according to step (4) of creating suffix links. You can see from figure 4.12 that step (4) is really skipped.

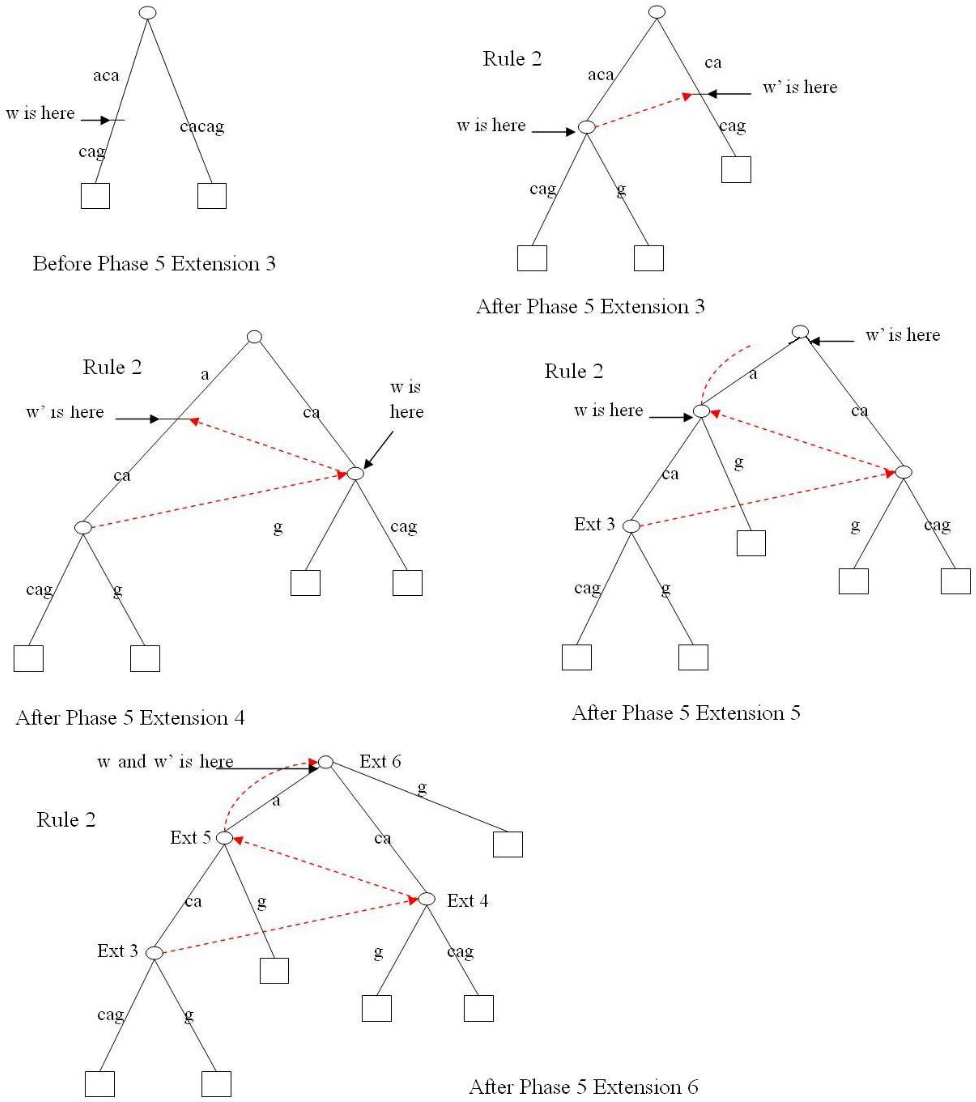

Figure 4.10: After Phase 5 Extension 6 for Suffix Tree of acacag

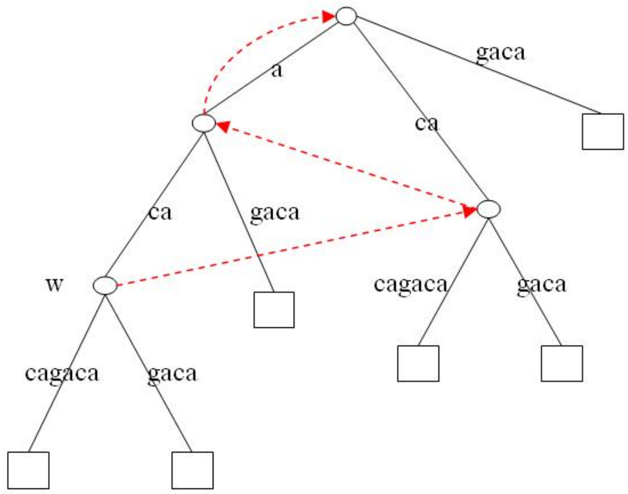

Figure 4.11: Before Extension 6 for Suffix Tree of acacagacat

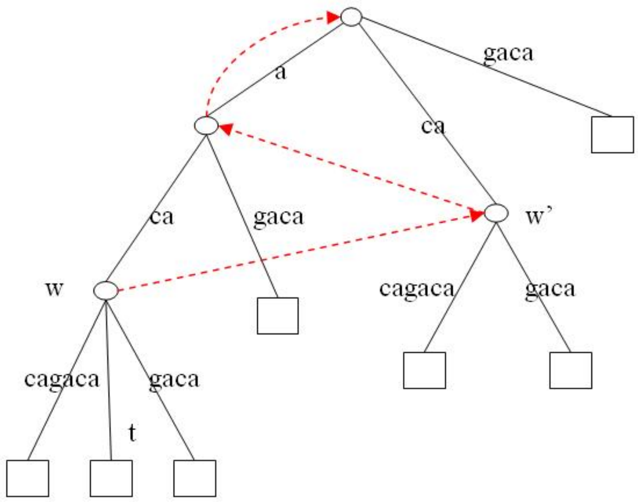

Figure 4.12: After Extension 6 for Suffix Tree of acacagacat

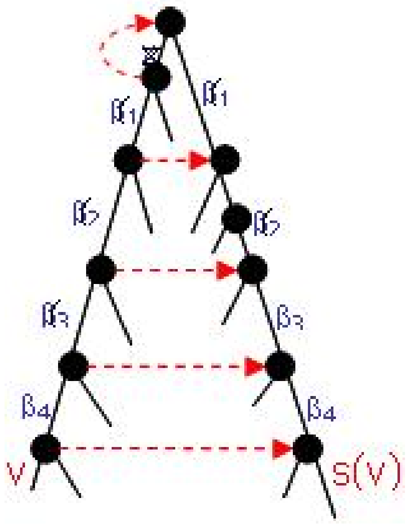

Figure 4.13: Time Complexity

4.4.4 Time Complexity

With the help of suffix links, the extension based on rule 2 or rule 3 are simplified. In the suffix link algorithm, step (1), (2), (3), (5) and (6) take O(1) time; as for step (4), it takes amortized O(1) time. So, each extension can be solved in O(1) time. As there are 2n extensions, the total time is O(n).

Why does step (4) takes amortized O(1) time? Notate node depth to be the number of edges from the root. Then, in each extension, step (2) reduces the node-depth by at most 1. For every internal node v, node-depth of v is equal or less than 1 + node-depth of s(v). (see figure 4.13) Therefore, step (3) reduces the node-depth by at most 1. Since there are 2n extensions, all steps (2) and (3) can reduce the node-depth by at most 4n. Since the maximum node-depth is n, all steps (4) can at most increase the node-depth by 5n. By amortization, each step (4) go down amortized O(1) nodes, which takes amortized O(1) time.

In conclusion, with the help of the suffix link, we can build the implicit suffix tree in O(n) time instead of O(n^2) time.

4.4.5 From Implicit Suffix Tree to Explicit Suffix Tree

Suppose we have already got an implicit suffix tree for s[1..n], we can convert it to an explicit suffix tree. First we add the character '\$' at the end of the string s[1..n]. In other words, let s[n+1]='\$'. Then we perform Phase n, to add '\$' into the suffix tree.

Since '$' is the last character of string s[1..n+1], it is the common suffix for all the suffices in the string. Therefore it is also the last character on every edge connecting to a leaf in the explicit suffix tree. Since '$' is a new character for the string, for any suffix in the implicit suffix tree, adding this character will not use rule 3. It will use rule 1 to append '$' to each edge connecting to a leaf; and use rule 2 to split some edges and for each edge to be splitted create a new internal node and a new branch from the internal node labelled '$'. These actions are reverse of creating an implicit suffix tree from an explicit suffix tree.

4.5 Application

4.5.1 Remains of US military personnel

Task:
To identify the US military identity when killed. From each live US military, a small interval of his DNA is extracted (which are likely to be unique for different people) and forms a database. When someone is killed, we can extract his DNA from his remain. However, due to the condition of the remain, we may not be able to get the complete information of the desired DNA interval, and may only get a substring S. We find all strings in the database containing S as a substring to determine who it may be.

Formalization:
The above task can be formalized as the substring problem for a database of patterns. Given a set of strings, database, of total length n, for any query string S, find all strings in the database containing S as a substring.

Solution:
We can follow the following instructions:

- Build a generalized suffix tree for all strings in the database. This takes O(n) time.
- For every query S, find all occurrences of S in the suffix tree. All the leaves in the subtree rooted at x are the occurrences; This takes $O(|Q|+occ)$ time where occ is the total number of occurrences.

How simple! How fast! :)

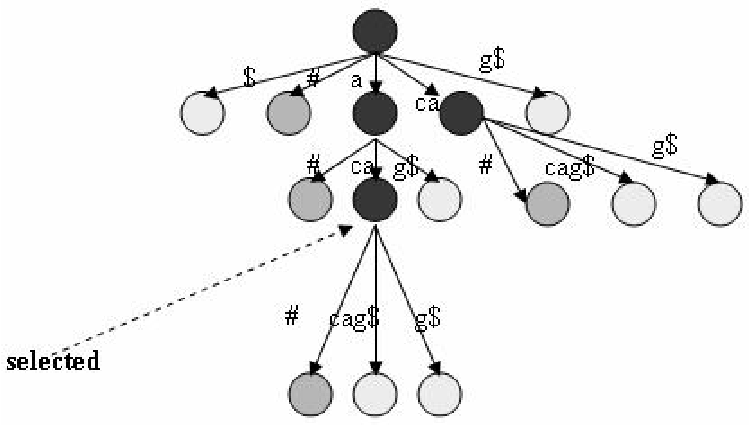

Figure 4.14: example of DNA contamination problem

4.5.2 DNA contamination problem

Task:
Contamination is a problem in bio engineering. Let S be a sequence reconstructed from a genome, says, C.elegans, S is suspected to be contaminated by cloning vectors, PCR primers, or genome of the host organism (e.g. yeast), how can we determine it?

Formalization:
Given a string S_1 (the sequenced string) and a string S_2 (the combined sources of possible contamination), find all substrings of S_2 that occur in S_1 and are longer than some given length x. If some substrings of S_2 occur in S_1, S_1 is suspected to be contaminated.

Solution:
(It is a problem similar to "find the longest common substring" problem described in Section 4.2.3)
We can solve this problem through the following instructions:

- Build a generalized suffix tree for S_1 and S_2.
- Mark each internal node with leaves representing suffixes of both S_1 and S_2.
- Report all marked nodes with string depth of x or greater.

Let's look at a simple example of this solution. Suppose $S_1 = acacag\$$, $S_2 = aca\sharp$, and $x = 2$.

First, build a generalized suffix tree for S_1 and S_2 (shown in Figure 4.14).

Second, mark every internal node (black color in Figure 4.14) whose subtree contains both S_1 and S_2 leaves (white stands for S_1 leaves, gray stands for S_2 leaves).

Last, select out the marked internal nodes whose path labels are longer than x(=2). In this case, only one internal node is selected (shown in Figure 4.14). Report this one, and in most cases, S_1 will be determined contaminated.

4.5.3 Longest common prefix problem

Formalized task:
Given a string S for any i, j, how to find the length of the longest common prefix of suffixes i and j of S.

Solution:
The basic idea is that the lowest common ancestor (shortened by lca) of leaves i and j identifies the longest common prefix. The solution for lca is first obtained by Harel and Tarjan [5], and later simplified by Schieber and Vishkin [6].

The conclusion is that given a suffix tree of size n, after an O(n) time's preprocessing, the lca for any two nodes can be returned in O(1) time. Based on the solution for lca, we can see that, after an O(n) time preprocessing, the length of the longest common prefix of suffix i and suffix j can be obtained in O(1) time for any i and j.

4.5.4 Application: Finding Palindrome

Definition:

Palindrome: Given a string S, palindrome is a substring u of S, such that $u = u^r$, where u^r denotes the reverse string of u. E.g. $ACAGACA$ is a palindrome as $ACAGACA = (ACAGACA)^r$.

Properties:

1. If $S[i..i+k-1] = S^r[n-i+1..n-i+k]$, then u=S[i-k+1..i+k-1] is an odd length palindrome;(see Figure 4.15)
2. If $S[i..i+k-1] = S^r[n-i+2..n-i+k+1]$, than u=S[i-k..i+k-1] is an even length palindrome.(see Figure 4.16)

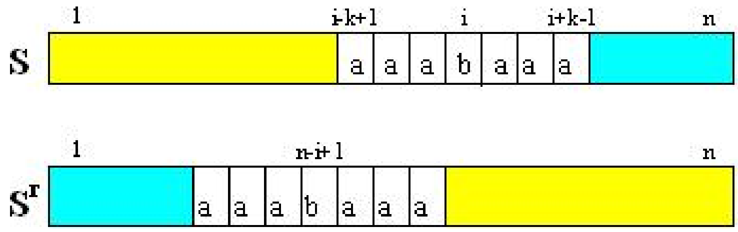

Figure 4.15: odd length palindrome

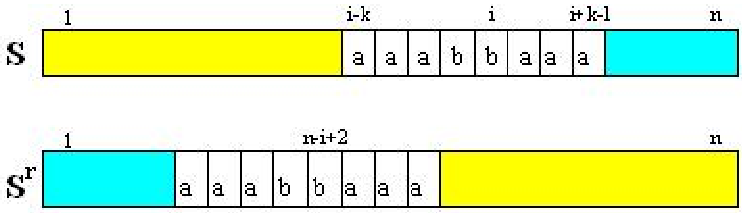

Figure 4.16: even length palindrome

Complemented palindrome: Given a string S, palindrome is a substring u of S, such that $u = \bar{u}^r$.where $\bar{u}$ is complement of u.
E.g. ACAUGU is a complemented palindrome.

Maximal palindrome: Given a string S, a palindrome $u = S[i..i + |U| - 1]$ is called a maximal palindrome of S, if S[i'..j'] is not a palindrome for any $[i'..j'] \supset [i..i + |U| - 1]$.

With this definition of maximal palindrome, every palindrome is contained in a maximal palindrome. In other word, maximal palindromes are a compact way to represent all palindromes.
We can define maximal complemented palindrome similarly.

Formalized task:

1. Given a sting S of length n, which represents the genome, the problem is to locate all maximal palindromes in S.

2. Given a sting S of length n, which represents the genome, the problem is to locate all maximal complemented palindromes in S.

In this section, we only give solution for task (1), which is to find maximal palindromes. Note that maximal complemented palindromes can be found similarly.

Solution:
Preprocess S and S^r so that any longest common prefix query can be answered in constant time (See Section 4.5.3);
For i=1 to n,

- Find the longest common prefix for (S_i, S^r_{n-i+1}). If the longest prefix is k, we find an odd length maximal palindrome $S[i-k+1..i+k-1]$.
- Find the longest common prefix for (S_i, S^r_{n-i+2}). If the longest prefix is k, we find an even length maximal palindrome S[i-k..i+k-1].

Analysis:
The preprocessing takes O(n) time, and each longest common prefix query can be found in O(1) time. So we can find the maximal palindromes in O(n) time.

4.6 Suffix Array

Above are just simple applications which will show up that with the help of suffix tree, some time consuming queries will be accomplished in a much shorter time. But suffix tree takes so much space. Can we reduce the space complexity?

Advantage of suffix array: It is a data structure, which has a similar functionality as suffix tree, but it requires <u>fewer</u> bits, namely O(n log n) bits compared to $O(n|\sum|logn)$ bits needed by suffix tree [7].
E.g. Consider $S = acacag\$$, where \$ denotes the ending of the string. The list of all possible suffices is as follows:

$1->acacag\$$
$2->cacag\$$
$3->acag\$$
$4->cag\$$
$5->ag\$$
$6->g\$$
$7->\$$
We sort these suffices by order of the alphabets, then we get:
SA[i] $->$ Suffix:
$7->\$$
$1->acacag\$$
$3->acag\$$
$5->ag\$$
$2->cacag\$$
$4->cag\$$

$6- > g\$$
The array SA[ ] is called the suffix array.
Note that each index can be stored using O(log n) bits. In total, the data structure can be stored in O(n log n) bits.

4.6.1 Construct suffix array from suffix tree

Because the leaves of a suffix tree are in suffix array order, thus we can construct the suffix array by "lexical" depth-first traversal of suffix tree T.
Analysis:
It takes O(n) linear time to construct the suffix tree T, and O(n) time to do the "lexical" depth-first traversal of T. This method is only theoretical, but not practical. Because it need to construct the suffix tree first, and the working space used to store the suffix tree during construction is large, namely $O(n|\sum|logn)$ bits.
Usually, we do sorting to get a suffix array instead of constructing a suffix array from a suffix tree.

4.6.2 Application

Task:
To find all occurrence of query Q in a string S using suffix array.

Solution:

1. Input suffix array of S and the query Q of length m;
2. Use binary search.

Algorithm:

1. Let L and R be the left and right boundaries of SA such that $suffix_{SA[L]} <= Q <= suffix_{SA[R]}$ (initially, L=1 and R=n+1)
   Denote l and r be the longest common prefix of Q and $suffix_{SA[L]}$ and $suffix_{SA[R]}$.
   Let mlr = min(l, r)
2. Let M = (L+R)/2
   We know that the first mlr characters of $suffix_{SA[M]}$ and Q are the same. Starting from the position mlr, find the last match position m. Find the lex-order of $suffix_M$ and Q by comparing $suffix_{SA[M]}[m+1]$ and Q[m+1]
   If $suffix_{SA[M]}>$ Q, set R=M and r=m else set L=M and l=m
3. Goto 1

Analysis:
Each comparison of binary search takes at most O(m) time (because it compares two strings of length m, rather than two bits), and there are log n comparisons performed in binary search. So in the worst case, it takes O($m$log$n$) time.But in practice, this method takes O(m+log$m$+log$n$) time.
Therefore, when space is limited, suffix array is a better choice than suffix tree.

4.7 Summary

In this chapter, we learned:

- **What is a suffix tree**
- **Algorithms to construct a suffix tree**
- **Simple applications of suffix tree**
- **Using suffix array instead of suffix tree to save space**

This chapter describes the use of suffix tree and suffix array with the assumption that the patterns match exactly with no error. However, these methods can be generalized to handle inexact pattern match.

References

[1] P. WEINER , "Linear pattern matching algorithms" ,*Switching and Automata Theory* 1-11, 1973.

[2] E. M. MCCREIGHT , "A space-economical suffix tree construction algorithm", *Journal of ACM* , 23:262-272, 1976.

[3] E. UKKONE, "On-line construction of suffix-trees", *Algorithmica*, 14:249-260, 1995.

[4] MARTIN FARACH , "Optimal Suffix Tree Construction with Large Alphabets", FOCS , 1997.

[5] D. HAREL and R. E. TARJAN, "Fast algorithms for finding nearest common ancestors", *SIAM Journal of Computing*, 13:338-355, 1984.

[6] B. SCHIEBER and U. VISHKIN "On finding lowest common ancestors: simplifications and parallelization",*SIAM Journal of Computing*, 17:1253-1262, 1988.

[7] U. Manber and G. Myers "Suffix arrays: a new method for on-line string searches",*SIAM Journal of Computing*, 22(5):935-948, 1993.