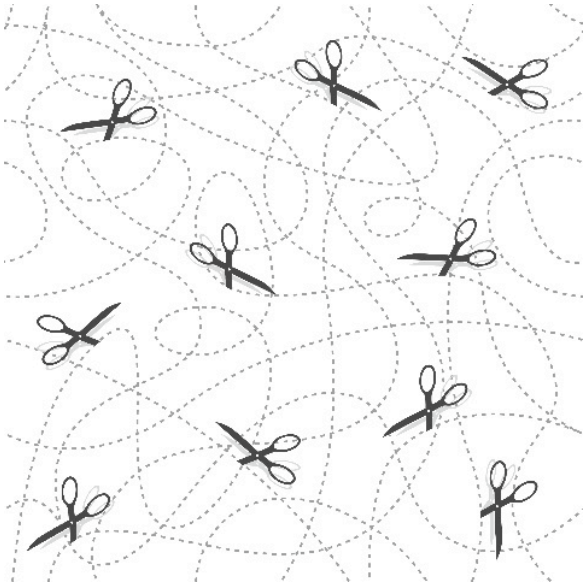
Chapter 6

Suffix Trees and its Applications



As mentioned earlier, Apostolico[Apo85] pointed out the "myriad of virtues" possessed by suffix trees. There are too many to go into detail on each. We will cover the most fundamental and useful ones. We refer the reader to [Gus97] and the references therein for an exhaustive list of the applications of suffix trees.

6.1 Exact String Matching

The most immediate application is to solve the substring problem stated at the beginning of the previous chapter. Recall that the substring problem consists of pre-processing the text such that a query of a given pattern is performed in time proportional only to the length of P. The following demonstration shows how to use suffix trees to solve this problem.

Assume we have at hand the suffix tree corresponding to text T. By performing breadth-first search (or other tree-search techniques), we find a path in the suffix tree that matches

pattern P. If no path is found, we may safely state that P is not in T. Otherwise, assume there is such a path. Note that, by the properties of suffix trees, that path is unique. Consider the edge at which that path ends, and for that edge consider its farthest node from the root (recall that a tree is a directed graph). That node is the root of a subtree with, say, k leaves. Given that every path from the root to the leaves of that subtree spells out a suffix of T having P as a substring, the number of occurrences of P in T is exactly k. Since the leaves of the suffix tree are numbered with the starting positions of the suffixes, the exact positions where P occurs can also be recovered from the suffix tree.

Example. Consider pattern $P = \{\texttt{aba}\}$ and text $P = \{\texttt{cabacabadabac}\}$. Suffix tree $\mathcal{T}(T)$ is shown in Figure 6.1 (the terminal symbol \$ has been added).

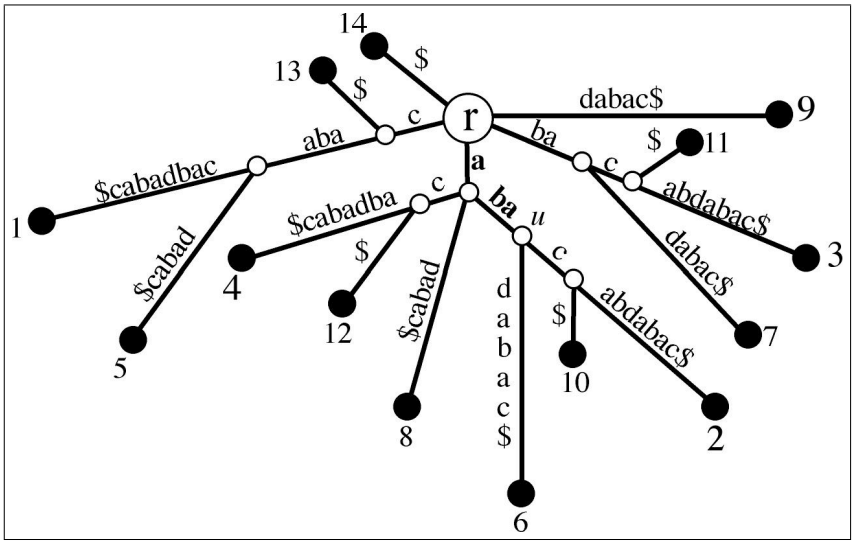


Figure 6.1: Solving the computation string matching problem with suffix trees.

The path that matches pattern $P = \{\texttt{aba}\}$ is shown in boldface in the tree; it ends at node u. The subtree rooted at node u has three leaves numbered $2, 6$ and 10. These three leaves correspond to the three occurrences of string $\{\texttt{aba}\}$ in T. Moreover, the labels of those leaves provide the positions of the pattern P in T.

Finally, we observe that the substring problem can be solved in time $O(m)$ with preprocessing time $O(n)$. In the query mode and for large databases, this represents a great saving of time.

6.2 Longest Common Substring of Two Strings

One of the first problems to arise in string pattern recognition, both because of its theoretical and practical interest, was the longest common substring problem of two strings. The problem is stated as follows.

The longest common substring of two strings (LCS problem): Given two strings S_1, S_2, find the longest common substring occurring both in S_1 and S_2.

The longest common substring of two strings S_1, S_2 will be denoted by $LCS(S_1, S_2)$. Let $S_1 = \{\texttt{contumaciously}\}$, $S_1 = \{\texttt{contumeliously}\}$ be two strings. Their $LCS(S_1, S_2)$ is the set formed by strings $\{\texttt{contum}\}$ and $\{\texttt{iously}\}$, both with 6 characters each. When there is no common substring, $LCS(S_1, S_2)$ is $\{\emptyset\}$.

A naive algorithm could easily take quadratic time or higher. Again, this problem can be solved in linear time by means of suffix trees. Let us first introduce the concept of **generalized suffix tree** of a set of strings. Let $\{S_1, \ldots, S_k\}$ a set of k strings and let $\{\$_1, \ldots, \$_{k-1}\}$ a set of terminal symbols such that $\$_i \notin S_j$, for any i, j. Consider the string $S = S_1\$_1S_2\$_2 \ldots S_{k-1}\$_{k-1}S_k$. The suffix tree of $\mathcal{T}(S)$ is called the generalized suffix tree of the set $S_1, \ldots, S_k$.

Example. Consider three strings $S_1 = \{\texttt{abac}\}, S_2 = \{\texttt{bada}\}$ and $S_3 = \{\texttt{acd}\}$. The set of terminal symbols will be $\{\$, \#\}$; symbol % is the terminal symbol needed for constructing the suffix tree. The resulting suffix tree $\mathcal{T}(S_1\$_1S_2\#S_3) = \mathcal{T}(\texttt{abac\$bada\#acd})$ is shown in Figure 6.2.

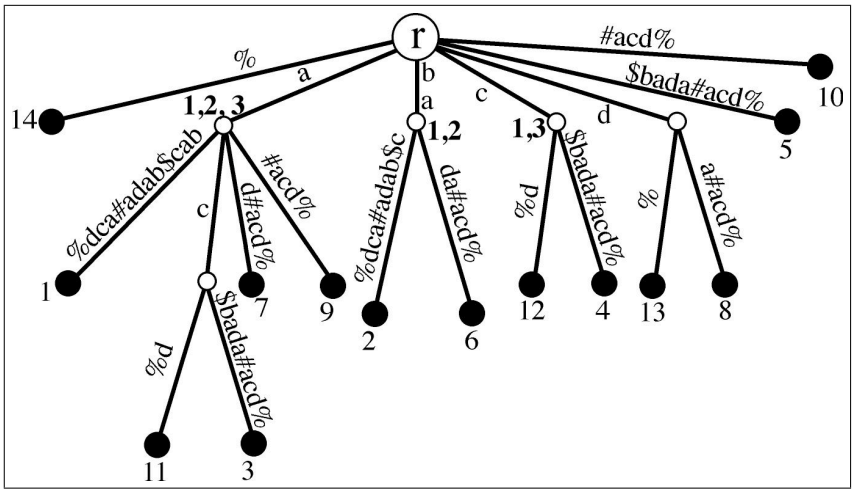


Figure 6.2: Generalized suffix tree of a set of strings.

In order to solve the LCS problem, we just build the generalized tree of strings S_1 and S_2, $\mathcal{T}(S_1\$S_2)$. When constructing it, mark each internal node with a label 1 or 2, depending on what string the current suffix is from. For example, in Figure 6.2 there is an internal nodes marked with three labels, $1, 2, 3$, showing a common substring to S_1, S_2 and S_3, namely, string $\{\texttt{a}\}$. The leaves of the subtree rooted at that internal node posses numbers from the

three strings S_1, S_2 and S_3. Other internal nodes are just marked with two numbers such as the corresponding to string {ba} or string {c}.

Therefore, a path-label composed of internal nodes marked with both numbers will spell out a common substring to S_1 and S_2. Finding $LCS(S_1, S_2)$ is achieved by just outputting the deepest string in the tree whose edges are both marked by 1 and 2. Marking the internal nodes and finding the deepest string can all be performed in time proportional to $O(|S_1| + |S_2|)$.

6.3 The Substring Problem for a Database of Patterns

This problem is a generalization of the substring problem. The text is now formed by a set of strings, normally a large one, called the **database of patterns**. The text is known and fixed. In the context of Bioinformatics, the database of patterns is a big biological database in which later on many DNA sequences are searched for. Several strings are presenting (query mode) and the algorithm has to determine what strings in the database contain the query string as a substring. Of course, pre-processing of the database is allowed and once this is done, the queries has to be answered in time proportional to their size. This problem is formally stated as follows.

The substring problem for a database of patterns (SPDP): Given a set of texts $T_1, \ldots, T_k$, the database of patterns, pre-process them so that the string matching computation problem is solved in time proportional to the length of pattern P.

Again, the solution comes to hand through generalized suffix trees. All the patterns of the text, $T_1, \ldots, T_k$, are concatenated with terminal symbols in between. A big text $T = T_1\$_1T_2 \ldots T_{k-1}\$_{k-1}T_k$ is built, where $\$_i \notin T_j$, for any i, j. Constructing $\mathcal{T}(T)$ takes linear time on the size of T. By following a similar argument to that of the substring problem, we can identify the occurrences of P in $\{T_1, \ldots, T_k\}$. Indeed, label the internal nodes with the corresponding text-labels and after that find those nodes marked by all the texts. They will return the solution to this problem; if no such node is found, then pattern P is not in all of the patterns of the database. The time complexity is again $O(|P|)$, once $\mathcal{T}(T)$ is built.

6.4 DNA Contamination

This problem has its origins in Bioinformatics as the title suggests. When DNA is sequenced, or more in general when DNA is processed, it is very undesirable that DNA from other sources become inserted in the sample. This situation can lead to invalid conclusions about the DNA sequences being studied. There have been very embarrassing cases. Some time ago, a few

scientists announced they have sequenced dinosaur's DNA from a bone, but other scientists harbour suspicions because of DNA contamination, as was the case finally. Normally, DNA sequences of many contaminants are known.

As a computation problem, the DNA problem is stated as follows.

> **The DNA contamination problem (DNAC problem)**: Given a string S, the newly sequenced string, and a set of source of possible contaminants, $C_1, \ldots, C_k$, find all substrings of S that occur in the set of contaminants having a length greater than certain number l.

Once more suffix trees will allow us to solve the problem in linear time. First, build a string C by concatenating all the possible contaminants. As before, insert terminal symbols between consecutive contaminants; the terminal symbols do not occur in any of the strings $C_1, \ldots, C_k$. Next, compute a generalized suffix tree for S and C. Then mark internal nodes with labels showing where the suffixes come from (either S or C). Finally, by traversing the tree, report all the marked nodes with both labels that are at a depth greater than l. If no node of that kind is found, we may state that sample S is not contaminated. The complexity of the whole process is proportional to the size of C and S.

Exercises

1 Given a string S, design an algorithm to find the substring with the longest prefix that is also a suffix of that substring. Prove the correctness of your algorithm and compute its complexity.

2 Given a string S, design an algorithm to find the longest repeated substring in S. Prove the correctness of your algorithm and compute its complexity.

3 **Palindromes**. A palindrome is a string such that when read forwards and backwards the same string is obtained. For example, string {`abcdcba`} is a palindrome. Design an algorithm to find the longest palindrome in a string. Prove the correctness of your algorithm and compute its complexity.

6.5 Chapter Notes

We have shown some of the basic applications of suffix trees. However, there are many more than we list here due to lack of space and also the scope of these notes. Other applications of suffix trees include: the common substrings of more than two strings, a problem that appears very often in Bioinformatics; computation of the longest substring common to at

least a given number of strings; computation of the matching statistics; computation of all-pairs suffix- prefix matching; computation of repetitive structures in DNA. The list could go on. The reader is referred to [Gus97] and the references therein for more and deeper information on those problems.