6.4. Generalized suffix tree for a set of strings

We have so far seen methods to build a suffix tree for a single string in linear time. Those methods are easily extended to represent the suffixes of a set $\{S_1, S_2, \ldots, S_z\}$ of strings. Those suffixes are represented in a tree called a *generalized* suffix tree, which will be used in many applications.

A conceptually easy way to build a generalized suffix tree is to append a different end of string marker to each string in the set, then concatenate all the strings together, and build a suffix tree for the concatenated string. The end of string markers must be symbols that are not used in any of the strings. The resulting suffix tree will have one leaf for each suffix of the concatenated string and is built in time proportional to the sum of all the string lengths. The leaf numbers can easily be converted to two numbers, one identifying a string S_i and the other a starting position in S_i.

One defect with this way of constructing a generalized suffix tree is that the tree represents substrings (of the concatenated string) that span more than one of the original strings. These "synthetic" suffixes are not generally of interest. However, because each end of string marker is distinct and is not in any of the original strings, the label on any path from the root to an internal node must be a substring of one of the original strings. Hence by reducing the second index of the label on leaf edges, without changing any other parts of the tree, all the unwanted synthetic suffixes are removed.

Under closer examination, the above method can be simulated without first concatenating the strings. We describe the simulation using Ukkonen's algorithm and two strings S_1 and S_2, assumed to be distinct. First build a suffix tree for S_1 (assuming an added terminal character). Then starting at the root of this tree, match S_2 (again assuming the same terminal character has been added) against a path in the tree until a mismatch occurs. Suppose that the first i characters of S_2 match. The tree at this point encodes all the suffixes of S_1, and it implicitly encodes every suffix of the string $S_2[1..i]$. Essentially, the first i phases of Ukkonen's algorithm for S_2 have been executed on top of the tree for S_1. So, with that current tree, resume Ukkonen's algorithm on S_2 in phase $i + 1$. That is, walk up at most one node from the end of $S_2[1..i]$, etc. When S_2 is fully processed the tree will encode all the suffixes of S_1 and all the suffixes of S_2 but will have no synthetic suffixes. Repeating this approach for each of the strings in the set creates the generalized suffix tree in time proportional to the sum of the lengths of all the strings in the set.

There are two minor subtleties with the second approach. One is that the compressed labels on different edges may refer to different strings. Hence the number of symbols per edge increases from two to three, but otherwise causes no problem. The second subtlety is that suffixes from two strings may be identical, although it will still be true that no suffix is a prefix of any other. In this case, a leaf must indicate all of the strings and starting positions of the associated suffix.

As an example, if we add the string *babxba* to the tree for *xabxa* (shown in Figure 6.1), the result is the generalized suffix tree shown in Figure 6.11.

6.5. Practical implementation issues

The implementation details already discussed in this chapter turn naive, quadratic (or even cubic) time algorithms into algorithms that run in $O(m)$ worst-case time, assuming a fixed alphabet Σ. But to make suffix trees truly practical, more attention to implementation is needed, particularly as the size of the alphabet grows. There are problems nicely solved

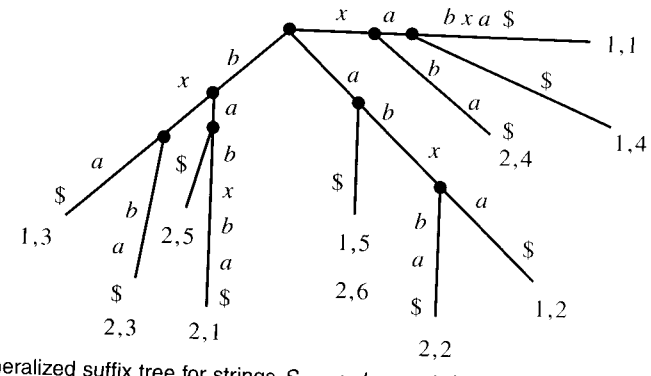

Figure 6.11: Generalized suffix tree for strings $S_1 = xabxa$ and $S_2 = babxba$. The first number at a leaf indicates the string; the second number indicates the starting position of the suffix in that string.