CS 124	**Lecture 22**	**Spring 2011**

Suffix trees are an old data structure that have become new again, thanks to a recent new linear time algorithm for constructing suffix trees due to Ukkonen that proves more useful for many applications. Here, we will describe a suffix tree and discuss their classical use, pattern matching.

22.1 Definition

A suffix tree T is built for a string $S[1 \dots m]$. The tree is rooted and directed with m leaves, which are numbered from 1 to m. Each edge is labeled with a nonempty substring of S. The internal nodes of the tree (other than the root) all have at least two outgoing edges, and the labels of all outgoing edges are labeled with different characters. By following the path from the root to leaf i and concatenating the edge labels, one obtains the suffix $S[i \dots m]$.

An example of a suffix tree for the string *xyzxzxy*\$ is given in Figure 22.1. The figure helps understand some important points about the suffix tree. First, each internal node has two or more children with different starting characters along the edges, since otherwise the node could be removed or moved in order to make this the case. Also, it is important that the last character of the string be a “unique” character, as this guarantees that the suffix tree as defined actually exists. For example, suppose our string was just *xyzxzxy*. The suffix tree would remain largely the same. In particular, in the not-quite-suffix tree in Figure 22.1 the path for the suffix *xy* does not end at a leaf, violating the definition. The problem is that the suffix *xy* is also the prefix of the string. This problem can be avoided by terminating the string with a special character that does not appear elsewhere, since then no suffix can also be a prefix (except for the entire string itself). Hence, from now on, we will assume all strings end with a special character \$.

It is also worth noting that a more convenient represenation of the suffix tree does not actually label the edges with characters. Instead, these labels can be represented by a pair of indices; labeling an edge $[i, j]$ represents that the edge label corresponds to characters $S[i \dots j]$. Besides saving space and ensuring that each edge is conveniently represented by two numbers, this scheme is important for the linear time algorithm for suffix tree construction.

22.2 Construction algorithm

To see that constructing suffix trees is possible, let us consider a simple $O(m^2)$ algorithm. Before beginning, we emphasize that in this case, the O notation is being used to hide a potentially substantial constant, that depends on the size of the alphabet. That is, if our alphabet is Σ, the O notation is hiding some factor dependent on $|\Sigma|$.

The goal is to build up the tree, one suffix at a time. We think of the intermediate results we get at each stage as partial trees, $T_1, T_2, \ldots, T_m$. Initially the tree T_1 consists of one edge, with label $S[1\ldots m]$; the end node is labeled with 1. For tree T_i, we modify the tree so that the suffix $S[i\ldots m]$ is handled properly. To do this, we start from the root and follow the path down the tree matching characters from $S[i\ldots m]$ as far as possible. This just requires character comparisons, and the path followed is necessarily unique since no two edges leading out of a node are labeled with string that begin with the same character.

(Note, however, that whenever we reach an intermediate node in the tree, we have to look at all the branches and decide which one, if any, to follow. Since there is at most one branch for each character, there are at most $|\Sigma|$ branches; since $|\Sigma|$ is a constant, this takes only constant time! In practice, one might want to set up a hash table based on a number assigned to each node and the first character on an edge in order to make finding the right edge branch out from a node more efficient.)

At some point, no further matches are possible. Note that this cannot happen at a leaf node, because we end our string with the special character \$. Therefore it either happens when our character matching is either in the middle of an edge or at a node. In the first case, we break the edge into two edges by inserting a new node. The edge to the new node contains the characters that have matched so far along the old edge, and the edge from the new node contains the remaining characters from the old edge. With this addition, we can now add the remainder of the suffix $S[i\ldots m]$ by adding another edge from the new node. If instead when no further matches are possible we are at a node, we can simply add a new edge from that node with the remainder of the suffix $S[i\ldots m]$. In both cases, when we add the new edge, we label the new leaf with the value i. The time to add each suffix is proportional to the length of the suffix, leading to an $O(m^2)$ algorithm.

Although this algorithm is very simple, the quadratic construction time is extremely limiting. Suffix trees are used, for example, for large pattern matching problems, where the input strings might be DNA strands of thousands or even millions of characters. Quadratic time will not suffice for these applications.

Fortunately, there are slightly more complex construction algorithms that require only $O(m)$ time. We will not discuss the algorithm at this point; the details and the subsequent analysis would require a non-trivial amount of

time. A reasonable introduction to the algorithm, however, has been written by Mark Nelson and has appeared in Dr. Dobb's Journal. You can currently find it at
http://www.dogma.net/markn/articles/suffixt/suffixt.htm.

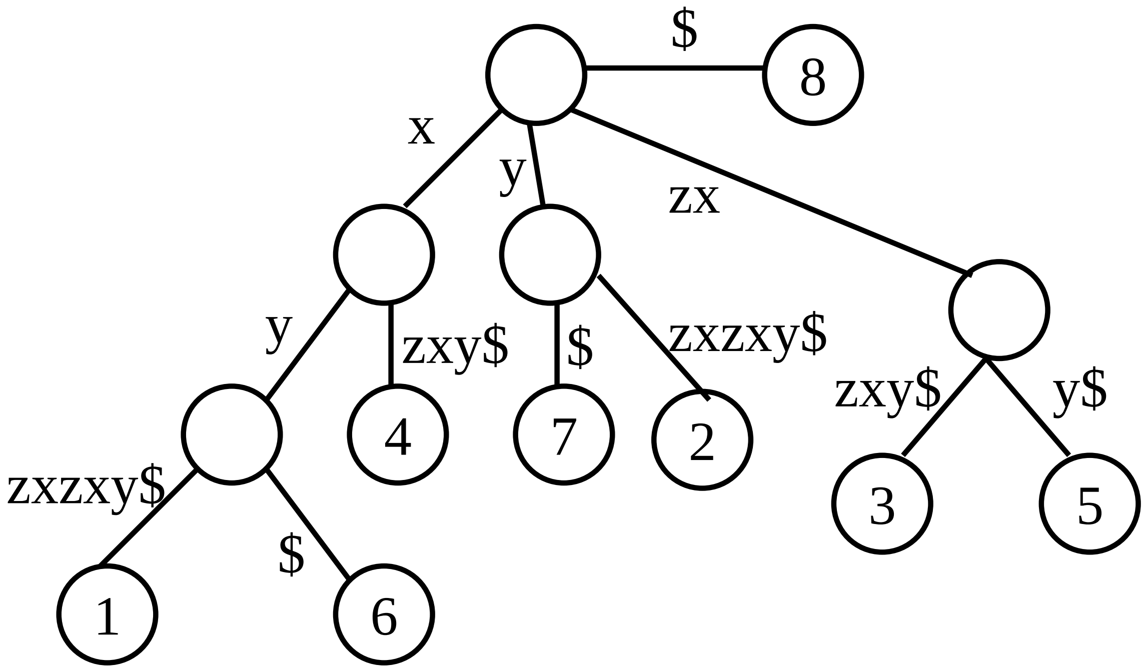
22.3 Using suffix trees for pattern matching

Once we have constructed our suffix tree, we can use it to efficiently solve pattern matching problems. There are of course other methods for pattern matching, but using suffix trees has an interesting advantage. Once the suffix tree has been constructed, finding all the occurences of any pattern $P[1\dots n]$ in the string S takes time $O(n+k)$, where k is the number of times that the string S appears in the text. So by incurring a one-time preprocessing charge to establish the suffix trees, we can handle any pattern matching problem after that in time essentially proportional to the length of the pattern, independent of the length of the original string! This is quite powerful, particularly for things like DNA databases, where the underlying database is large and fixed but must be able to deal with lots of queries.

Suppose that P lies in the string S; for example, suppose P corresponds to $S[i\dots i+n-1]$. Then P is the prefix of the suffix $S[i\dots m]$. Hence, if we starting matching characters in P against the labels in the suffix tree for S, we will follow part of the path from the root to the leaf vertex labeled i. Hence, to find all occurences of P in S, start at the root, and match down the tree as far as possible. This takes time $O(n)$. If P does not match some path in the tree, then P does not lie in S. If P does match some path in the tree, in matches down to some point z. All the leaves in the subtree below z correspond to suffixes for which P is a prefix, so the labels on these leaves correspond to locations that begin an occurence of P. To find these positions, we just traverse the subtree below z, using for example depth first search. If there are k leaves, the depth first search takes only $O(k)$ time.

22.4 Representation

An important point about suffix trees: to make sure everything takes linear time, it is important to use the correct representation. For example, we do not explicitly label each edge with a group of characters– this could take as much as $\Omega(n^2)$ time to just write down! Instead, each edge is labeled with a pair of values, representing characters. For example, an edge labeled $[a,b]$ should be thought of as being labeled by the character $S[a]\dots S[b]$. Hence each edge is just labeled by two numbers, and only linear space is required.



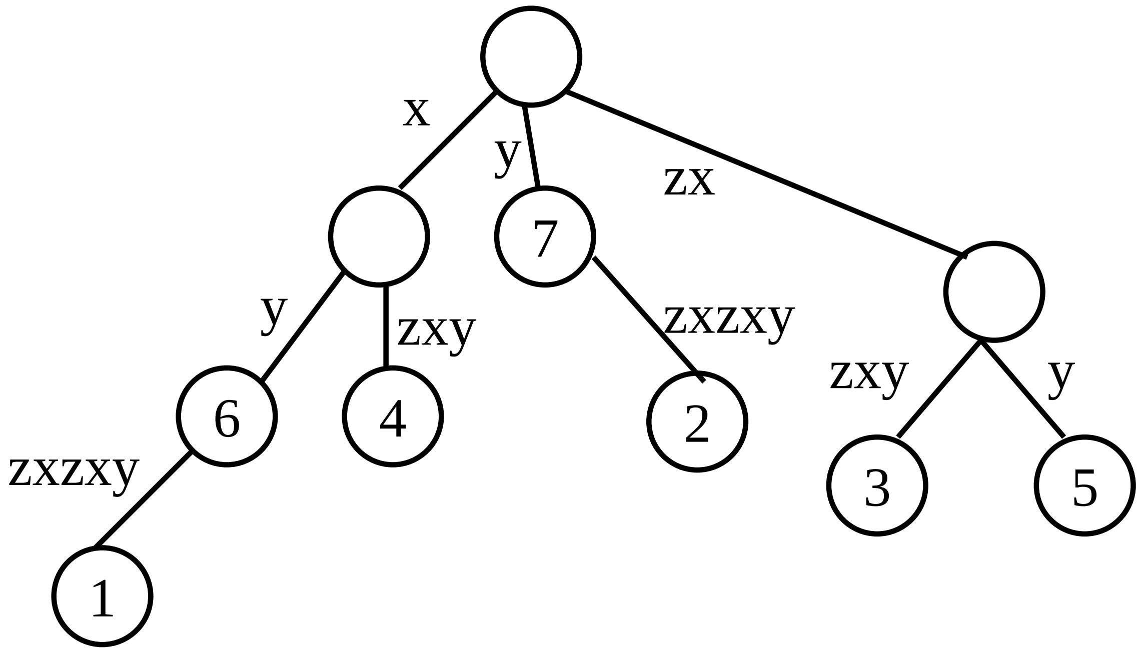


Figure 22.1: A true suffix tree (top); why we need the $ character (bottom).

22.5 Generalized suffix trees

You may want to put a set $\{S_1, S_2, \ldots, S_k\}$ of strings in a suffix tree data structure. (Note– we assume each string ends with the special character \$.) The structure in this case is called a generalized suffix tree. There are two primary differences. First, now each leaf node may contain multiple pairs of numbers. Each pair of numbers identifies a string S_i and a location where the suffix from the root to that leaf starts in S_i. Note that multiple strings can have a suffix that share a leaf node! Second, each edge label must be represented by three numbers: a number i and a pair $[a,b]$ represent that the characters on the edge label are $S_i[a] \ldots S_i[b]$.

Construcing a generalized suffix tree can easily be done by extending our quadratic time algorithm. However, the linear time algorithm for suffix trees can also be used to build a generalized suffix tree. Hence if $m = \sum_{i=1}^{k} |S_i|$, constructing the generalized suffix tree can be done in $O(m)$ time.

22.6 Longest common extension

Using generalized suffix trees and the LCA algorithm, we can solve a very general problem called the *largest common extension problem*. Given strings S_1 and S_2, we wish to pre-process the string so that we can answer questions of the following form: given a pair (i, j), find the longest substring of S_1 that begins at position i that matches a substring of S_2 that begins at position j.

We will use linear time pre-processing and linear space, after which we can answer queries in constant time.

The solution is to build a generalized search tree for S_1 and S_2. When we build this tree, we should also compute the *string depth* of each node. The string depth of a node is simply the number of characters along the edges from the root to that node. Notice the string depth is *not* the same as the tree depth. Also, after building the tree, we precompute the information necessary to do LCA queries on the tree.

Given a pair (i, j) we compute the least common ancestor u of the leaf nodes corresponding to the suffix beginning at i in S_1 and the suffix beginning at j in S_2. The path from the root to u is longest common extension, and hence the string depth of this node is all we need.

22.7 Maximal palindromes

A palindrome is a string that reads the same forwards as backwards, such as *axbccbxa*.

A substring U of a string S is a maximal palindrome if and only if it is a palindrome and extending it one character in both directions yields a string that is not a palindrome. Generally we separate even-length maximal palindromes, or even palindromes for short, and odd-length maximal palindromes (odd palindromes) for convenience. For example, in $S = axbccbbbaa$, the maximal even palindromes are $bccb, bb$, and aa. The string bbb is a maximal odd palindrome, and we will skip writing the maximal odd palindromes of length 1. Note that every palindrome is contained in a maximal palindrome.

Here is a simple way to find *all* even-length maximal palindromes in linear time. (Finding odd-length maximal palindromes is similar.)

Consider S and S^r, the reversal of S. There is a palindrome of length $2k$ with the middle just after position q if the string of length k starting from position $q+1$ of S matches the string of length k starting from position $n-q+1$ of S^r. In particular, this palindrome will be maximal if this is the length of the longest match from these positions.

Thus, solving the even-length maximal palindrome problem corresponds to computing the longest common extension of $(q+1, n-q+1)$ for all possible q. The data stucture can be processed in linear time, and each of the linear number of queries can be answered in constant time, so the total time is linear.