CSCI 104L Lecture 25 : Suffix Trees

Suffix Trees

Tries can be thought of as prefix trees: given a prefix of a string, you can quickly identify the possible ways to finish it. Thus it is often used for auto-complete.

What if we want something more powerful: the possibility to match a substring, rather than just a prefix?

A suffix tree of a word W is a compressed trie of all possible suffixes of W. When W is an n-character string, a suffix tree has the following properties:

- n leaves, labeled s[i:n-1], for all i.
- each non-leaf node has at least two children
- each edge is labeled with a substring of s.
- if e,e' are edges out of the same node, then the labels must start with different characters.
- for any root-leaf path ending in a node labeled i, the concatenation of the labels on the path is equal to s[i:n-1]

Question 1. Is 'issip' a substring? 'sipi'?

There will always be n leaf nodes. Since no node has a single child, there will be < n internal nodes. So there are O(n) nodes.

There is an algorithm (Ukkonen’s Algorithm) which can construct a suffix tree in O(n) time as well.

Suffix trees can solve the following problems (among others):

- Does a pattern t exist in the string s?
- How many times does t appear in s?
- What is the longest common substring of s and t?

You can find the longest common substring of s and t! You need suffix links to do this:
Add a suffix link from node xS to S. Add a string depth to each node. When you reach a dead-end, follow the suffix link. Record the greatest depth you reach.

Hash Tables

**Question 2.** A company has assigned a unique 3-digit ID to each of its 1000 employees. We want to design a data structure so that you can input an employee ID and quickly bring up their employee record. How should we implement this?

**Question 3.** Does something similar work if we want to do this from USC student ID to student records? Why or why not?

**Question 4.** What if we want to store the English dictionary as a set of strings, so we can quickly look up if something is a word or not. Can we do something like this for non-integer keys?

A hash function takes a valid input (in the case of the last question, a word in the English language) and outputs the entry in the array to store it. To be a good hash function it must:

- be efficient to calculate
- distribute the inputs well
- be consistent

**Question 5.** Is \( h(k) = 0 \) a good hash function? Why or why not?

**Question 6.** Is \( h(k) = k \mod m \), where \( m \) is the size of the table, a good hash function? Why or why not?

**Question 7.** Is \( h(k) = \) a random integer between 0 and \( m - 1 \), where \( m \) is the size of the table, a good hash function? Why or why not?

The goal is to design a hash function where the probability of collision is \( \leq \frac{1}{m} \). Any “good” hash function that satisfies this is called a “Universal Hash Function.”