18.417 Introduction to Computational Molecular Biology

Lecture 11: October 14, 2004 — Scribe: Athicha Muthitacharoen

Lecturer: Ross Lippert — Editor: Tony Scelfo

Suffix Trees

This is one of the most complicated data structures we will cover in class. We will start with the history of suffix trees. In 1970, Knuth proposed that the problem "Given strings S1 and S2, find the largest common substring" is harder than just reading in the two strings ($> O(|S1| + |S2|)$). Back then it was hard to define a lower bound.

In 1973, Peter Wiener established that the lower bound was $O(|S1| + |S2|)$.

In 1976, McCreight gave a practical data structure to solve the problem.

Review: Keyword tree

Given the patterns $P_1 ... P_k$.

Can perform lookup P's in a text T in $O(|A||T|)$ time. To represent all substrings of S, we can build a keyword tree on all suffixes of S, since every suffix is some prefix of a suffix.

Example of an application in biology: Looking for a nucleotide sequence in the human genome is analogous to finding how many times a substring appears in a $3 \cdot 10^9$ character sequence.

What is a suffix tree?

Figure 11.1 illustrates an example of a suffix tree with suffix links.

If we build a keyword tree of all suffixes, it will take $O(|A| \cdot \frac{|S^2|}{2})$ time and $O(\frac{|S^2|}{2})$ space. Since failure links are parallel, We can exploit the fact that each string is some other string's suffix and compress the tree.

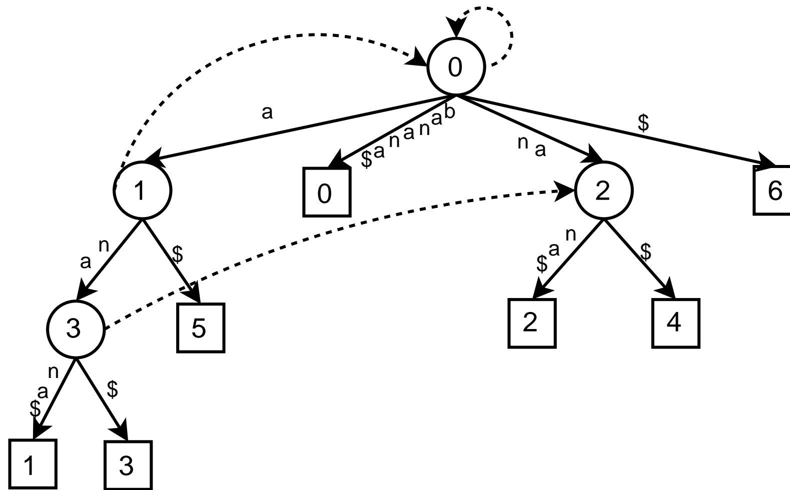


Figure 11.1: A suffix tree for the string `banana$`

Reading the tree: a path from the root to a leaf is always a suffix.

We create a new branch only if it is topologically interesting. This results in a reduction of the nodes, and we end up with $|S|$ leaves and less than $|S|$ non-leaves, according to some theorem.

Each node can be a substring, not just a character. To conserve space, we can represent each node by a pair (index, len) into the string, instead of the entire substring. $O(|S|)$ space. $O(1)$ per element.

The only failure links retained is between internal nodes and not to the leaves. In keyword trees, failure link $F(n) = m$, where m is the node corresponding to the largest prefix in the FSM, which is a suffix of string n. In suffix trees, m is the node corresponding to the first proper suffix of n.

Why do failure links point to branching internal nodes? Because w has to be a branching suffix.

Failures in suffix trees are a bit more complicated.

$$\text{Failures} = \begin{cases} \text{1. follow parent's link and rematch to position in the middle of the} \\ \text{substring or at a node.} \\ \text{2. follow your node's link.} \end{cases}$$

STFSM has scan time $O(|A||T|)$, once the tree is built. T is the whole text.

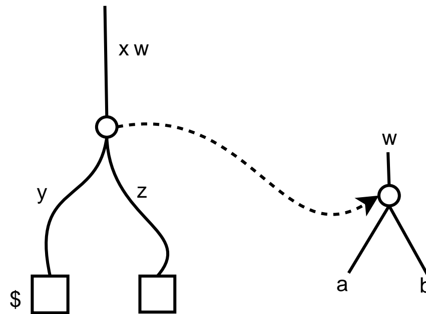
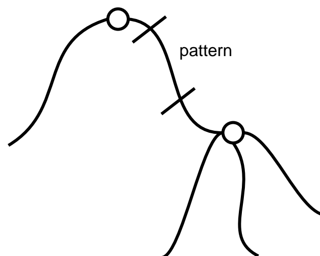


Figure 11.2: Diagram of branching nodes and a failure link from node xw to w

Pattern P lookup takes $O(|A||P|)$ time. A is the whole alphabet.

Applications of Suffix trees

Ex. Looking for a pattern.



Once found a pattern, the number of leaves are the number of times the pattern occurs.

Ex. Find a pattern A[10-20 chars]B[10-200 chars].

Ex. MUM (Maximal Unique Match)

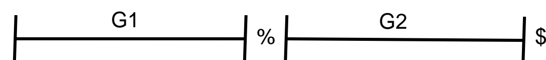
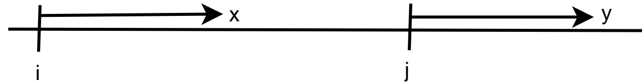


Figure 11.3: Concatenation of two genomes to in the MUM example

Concatenate G_1 and G_2, with special symbols at the end of each genome, and then build a tree of the concatenated string (which is easy), see Figure 11.3.

MUM is the match that the rightmost character does not match the other string's rightmost character AND that the leftmost character does not match the other string's leftmost character.

Ex. Longest Common Prefix



$$lcp(i,j) = \begin{cases} \text{depth their parent node, if i and j are siblings.} \\ \text{depth of the least common ancestor (LCA), otherwise.} \end{cases}$$

It takes $O(1)$ to find the LCA.

Other hacks: order children lexicographically. Makes it easier to compare to other trees.

Can use a more complicated suffix tree to explain MUMs.

Building Suffix Trees

Takes $O(n)$. Actually $O(|A||T|)$, but the size of the alphabet is constant. There are several proposals, such as Uckonnen and McCreight, but Uckonnen is just a reordering of McCreight's loops.

McCreight's method

1. Add $S[0...n]$ to tree, resulting in a big leaf.

2. Then, add $S[1...n]$, $S[2...n]$, ... Follow some existing branching until mismatch. If so, then branch, as in Figure 11.4

$$\text{At every step} \begin{cases} \text{add up to one internatl node} \\ \text{add exactly one leaf, } S[i...n] \end{cases}$$

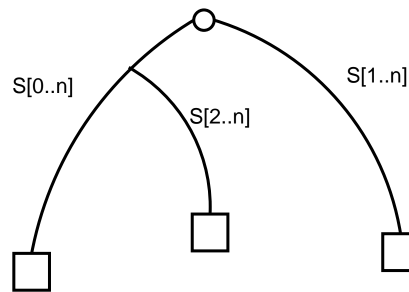


Figure 11.4: Snapshot of a suffix tree during the McCreight's method.

Code explanation (refer to lecture slide "Building a suffix tree")

Procedures

`fork`: adds a new internal node.

`get-child`: given node N, and character ch, returns the node that corresponds to ch.

`get-branch`: given suffix s, find s. If cannot find, create a new node and branch. *How to speed up* `get-branch` *to get* $O(1)$ *time?* Every time lookup up $S[i+1, ...]$, can do fast-match up to $d-1$ while not making a mistake. d is the maximum depth that the previous substring matched for which we created a new node. If we did not create a new node last time, we can move up to the parent node and fast-match down. Then create a new node if needed.