CMSC858P: Algorithms for Biosequence Analysis (Spring 2008)

Thursday February 28

Lecture: Dr. Mihai Pop

Scribe: Mohammadreza Ghodsi

Suffix trees properties recap

- Since there is a "$" after each suffix, each leaf in the suffix tree corresponds to one suffix.
- Each internal node of the tree corresponds to a common prefix of two suffixes in the string (that is a repeat)
- Suffix links can be used in different algorithms that traverse suffix trees. For example given the suffix tree for S_1 if we are to find the longest common substring of S_1 and S_2 we can do the following: start matching S_2 to suffix tree of S_1. When a mismatch occurs follow the suffix link (if mismatch occurs in the root then just shift to the next character in S_2). The deepest node that we get to during this process corresponds to a longest common substring of S_1 and S_2.

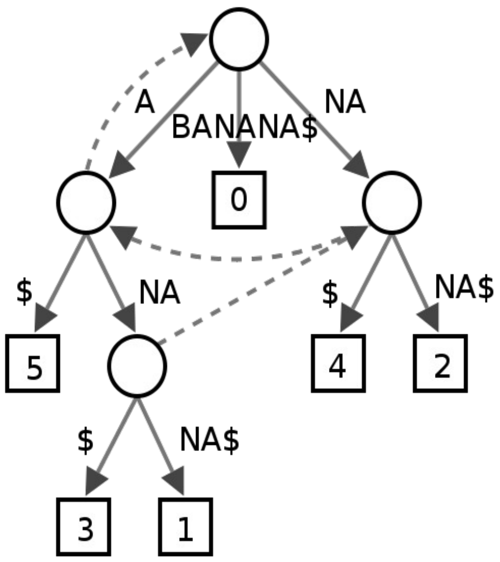

Figure: Suffix tree for the string `BANANA` padded with `$`. The six paths from the root to a leaf (shown as boxes) correspond to the six suffixes `A$`, `NA$`, `ANA$`, `NANA$`, `ANANA$` and `BANANA$`. The numbers in the boxes give the start position of the corresponding suffix. Suffix links drawn dashed.

Longest prefix-suffix match

Given *k* strings $S_1, S_2, \ldots, S_k$ we wish to find for every pair S_i and S_j, the longest suffix of S_i that matches a prefix of S_j.

In order to do this efficiently we first build a suffix tree for all of the strings. Our goal is then to find internal nodes on the path from the leaf corresponding to complete S_j to the root that are connected by a single "$" to a leaf (which corresponds to a suffix of some string). Note that if more than one suffix of

S_i match prefixes of S_j then we should report the longest one.

First, for each internal node of the tree we need to compute *L(v)*, a linked list of all children of *v* that are attached to it by a "\$". We can do this in one traversal of the tree. The following pseudo code does a depth first search traversal of the tree.

```
for each c in children of v do
     if c is a $-leaf  add c to L(v)
     recurse on non-leaf children of v
```

This computation is linear in the size of the tree (and therefore in the sum of sizes of input strings)

Next, we do another DFS of the tree, maintaining *k* stacks, one for each string. During the execution of the algorithm each stack will contain the places that we have seen each string in *L(v)* for all *v* on the path from current node to the root. On top of the stack *i* would be the last (deepest) node that had a \$-child to a suffix of *i*.

Now to during traversal of tree whenever we come across a leaf that corresponds to complete string S_j all we need to do is to look at the top items on all the stacks to find the longest suffix of any string S_i that matches a prefix of S_j.

Suffix Arrays

A suffix array is an array giving the suffixes of a string in lexicographical order. If we sort the suffixes of the word "mississippi" we would get

T_{11} = i
T_8 = ippi
T_5 = issippi
T_2 = ississippi
T_1 = mississippi
T_{10} = pi
T_9 = ppi
T_7 = sippi
T_4 = sissippi
T_6 = ssippi
T_3 = ssissippi

The space required for this is quadratic but we can just store the suffix numbers (here 11,8,5,...). That would make the space requirement linear in the size of the input.

If we build a suffix array of text, the naïve algorithm for searching a pattern would be a simple binary search. this would O(n log m) time (where n and m are sizes of pattern and text respectively). This is worse than O(n) we would get using a suffix tree. We can improve search time by taking advantage of the fact that the strings are lexicographically ordered.

First attempt to improve running time is when during the binary search we are comparing pattern to the middle suffix between two (left and right) suffixes we can avoid comparing the first characters of the pattern that have already been matched to both left and right suffixes. More precisely if the first l characters of pattern match to L and the first r characters of pattern match to R. we can start matching pattern to M right after position min(l,r) .

Title:figure.fig
Creator:fig2dev Version 3.2 Patchlevel 5
CreationDate:Sun Mar 2 23:59:28 2008

This would result in practice in O(n + log m) running time. But in order to be able to prove a bound for the worst case we need a more clever trick.

A second provably better in the worst case implementation is to store some additional information. namely for each i, j we store $LCP(i,j)$ which is the length of the longest common prefix of suffix i and suffix j.

Title:figure2.fig
Creator:fig2dev Version 3.2 Patchlevel 5
CreationDate:Mon Mar 3 00:12:35 2008

let l be the number of characters from the pattern that match suffix L. and similarly for r and R. If we are about to compare pattern P to M then there are a few cases (note that we already know $L<P<R$):

- $l<LCP(L,M)$: then we just need to recurse on [M,R]. no character comparison is needed
- $l>LCP(L,M)$: again without any character comparison we know P is to the left of M so we would recurse on [L,M]
- $l=LCP(L,M)$: in this case we actually have to continue comparing characters of the pattern to string M starting from position LCP(L,M) onward.

We can do this on the side which matched pattern the longest. So without loss of generality we can assume $l>r$. The total number of characters that match in this algorithm is O(n). The number of mismatches is equal to the number of recursions and therefore is O(log m). So total running time is O(n + log m).

In the next session we will see:

How to compute suffix arrays.

How to compute LCP values.