Advanced Data Structures	Jan-Apr 2012
Lecture 20 April 19, 2012	
Lecturer: Venkatesh Raman	*Scribe: Esha Ghosh*

1 Overview

In the previous lectures, we have discussed about Suffix Trees (ST), Suffix Arrays (SA) and LCP (Longest Common Prefix) Arrays. In this lecture we will discuss the linear time constructions of these data structures. Recall that, in the Suffix Array, $SA[i] = j$ means that suffix starting at j is the i^{th} in the lexicographic order of all suffixes, and in the LCP array, $LCP[i]$ contains the length of the longest prefix between $SA[i]$ and $SA[i+1]$.

2 Constructing Suffix Arrays (SA) and LCP Arrays from Suffix Trees (ST)

2.1 Constructing ST [1]

Let us consider the suffixes are inserted in the tree in the order $suf_1, suf_2, \ldots suf_n$. In the i^{th} step, suf_i is inserted into tree T_{i-1} to form tree T_i by starting from the root and following the unique path matching characters in suf_i one by one until no more matches are possible. If the traversal does not end at an internal node, create an internal node there and append the unmatched portion of suf_i as edge labels. Constructing Suffix Tree in the naive way will take $O(n^2)$ time.
For an $O(n)$ running time, suffix links are used to construct Compact Suffix Trees. Suppose we are inserting suf_i in T_{i-1} and let v be an internal node in T_{i-1} on the path from root to leaf labelled $(i-1)$.

$suf_{i-1} = c\alpha \ldots$ where $c \in \Sigma$ and $\alpha \in \Sigma^*$. Since v is an internal node, there must be another suffix $suf_j = c\alpha \ldots$,where $j < (i-1)$. Because suf_{j+1} is previously inserted, there is a already a path labelled α in T_{i-1}, i.e., $suf_{j+1} = \alpha \ldots$ Therefore $suf_i = \alpha \ldots$. To insert suf_i faster, comparison of characters in suf_i can start beyond the prefix α. This is how suffix links are exploited to save on time.

2.2 Contructing SA and LCP Arrays from ST

Now, SA can be constructed from ST simply by reading all the leaves of the ST in the left-right order. This takes time linear in the number of suffixes, $O(n)$.
In the ST, the string depth of the least common ancestor of two suffixes is the LCP length between those two suffixes.

3 Constructing LCP Arrays from SA [2]

Let R be an array of size n such that $R[i] = j \Rightarrow SA[j] = i$. R can be constructed by a linear scan of SA in $O(n)$ time. Now we do the following:

For $i = 1$ to $(n-1)$: Compute LCP(suf_i, SA$[R[i]+1]$).

The array R facilitates locating an arbitrary suffix suf_i and its right neighbor in SA in constant time. Initially, the length of the longest common prefix between suf_i and right neighbor in SA is computed manually and recorded. Now we need to argue that using the information LCP(suf_i, SA$[R[i]+1]$), it would be possible to compute LCP(suf_{i+1}, SA$[R[i+1]+1]$) in amortized constant time.

Computing LCP(suf_{i+1}, SA$[R[i+1]+1]$) from LCP(suf_i, SA$[R[i]+1]$):

Let suf_j be the right neighbor of suf_i in the SA and let LCP(suf_i,suf_j)= l, $l \geq 1$. As suf_j is lexicographically greater than suf_i and $T[i] = T[j]$, this implies suf_{j+1} is lexicographically greater than suf_{i+1}. The length of the longest common prefix between them is $l-1$. Then it follows that, the length of the longest common prefix between suf_{i+1} and its right neighbor in the suffix array is $\geq l-1$. So, we can skip the first $(l-1)$ characters and the comparison starts from the l^{th} character.

Analysis:

We charge a comparison between r^{th} the character of suf_i and the corresponding character in its right neighbor suffix in SA to $i+r-1$. There is only one failed comparison in each iteration of the algorithm. So the total number of failed comparisons is $O(n)$. As for successful comparisons, each position in the string is charged only once for a successful comparison. Thus, the total number of comparisons over all iterations is $O(n)$.

4 Constructing SA in Linear time [3]

Let T be a string of length n over the alphabet set Σ. For convenience, assume $n \equiv 0 (\text{mod } 3)$. Now use the following algorithms:

1. Sort all suf_i where i is not a multiple of 3. Let $R(i)$ denote the rank of suf_i.
2. Use the result of Step 1 to sort the $\frac{1}{3}n$ suffixes suf_i with $i \equiv 0 (\text{mod } 3)$.
3. Merge the two sorted lists from Step 1 and Step 2.

We assume $|\Sigma|$ is constant, as we well be using Radix sort to sort the suffixes.

4.1 Analysis

Step 1:

- Perform a radix sort of the $\frac{2}{3}n$ triples $(T[i], T[i+1], T[i+2])$ for $i \equiv 1$ or $2 (\text{mod } 3)$and associate with each distinct triple its rank $R(i) \in \{1, 2, \ldots \frac{2}{3}n\}$ in sorted order.
- If all triples are distinct, the suffixes are already sorted.
- Otherwise, create a new string $T' = {suf_1}' \boxdot {suf_2}'$. [1] Sorting T' recursively gives the sorted order of suf_i where $i (\text{mod } 3) \neq 0$.

Step 2:

Perform a radix sort on the tuples $(T[i], R(i+1))$ for all $i (\text{mod } 3) = 0$. Note that $R(i+1)$ is obtained from Step 1.

Step 3:

Here we essentially need to argue that we can compare two suffixes in constant time to merge the two lists obtained from Step 1 and Step 2 of the algorithm. To compare suf_i, $(i \text{ mod } 3 = 0)$ and suf_j, $(j \text{ mod } 3 = 1)$ compare $T[i]$ and $T[j]$. If they are unequal, the answer is clear. Otherwise the ranks of suf_{i+1} and suf_{j+1} in the sorted order obtained in Step 1 determines the answer. To compare suf_i, $(i \text{ mod } 3 = 2)$ with suf_j, $(j \text{ mod } 3 = 0)$, compare the first two characters of the two suffixes. If they are both identical, the ranks of suf_{i+2} and suf_{j+2} in the sorted order obtained in Step 1 determines the answer.

Running time:
The run-time of this algorithm is given by the recurrence $T(n) = T(\frac{2n}{3}) + O(n)$, which results in $O(n)$ running time.

References

[1] R. Giegerich and S. Kurtz, From Ukkonen to McCreight and Weiner. *A unifying view of linear-time suffix tree construction.* Algorithmica, 19:331-353, 1997.

[2] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. *Linear-time longest-common- prefix computation in suffix arrays and its applications.*, 12th Annual Symposium, Combinatorial Pattern Matching, pages 181-92, 2001.

[3] J. Kärkkänen and P. Sanders. *Simpler linear work suffix array construction.* In International Colloquium on Automata, Languages and Programming, 2003.

[4] Srinivas Aluru, *Suffix Trees and Suffix Arrays*, In Handbook of Data Structures and Applications, Edited by Dinesh P. Mehta and Sartaj Sahni, Chapman & Hall/CRC Computer and Information Science Series, Chapter 29 (21 pages), 2004.

[1] $\boxdot$ denotes concatenation.