6.851: Advanced Data Structures Spring 2010

Lecture 7 – February 26, 2010

Prof. Andre Schulz *Scribe: Mark Chen*

1 Overview

In this lecture, we consider the string matching problem - finding all places in a text where some query string occurs. From the perspective of a one-shot approach, we can solve string matching in $O(|T|)$ time, where $|T|$ is the size of our text. This purely algorithmic approach has been studied extensively in the papers by Knuth-Morris-Pratt [6], Boyer-Moore [1], and Rabin-Karp [4].

However, we entertain the possibility that multiple queries will be made to the same text. This motivates the development of data structures that preprocess the text to allow for more efficient queries. We will show how to construct, use, and analyze these string data structures.

2 Storing Strings and String Matching

First, we introduce some notation. Throughout these notes, Σ will denote a finite alphabet. An example of a finite alphabet is the standard set of English letters $\Sigma = \{a, b, c, ..., z\}$. A fixed string of characters $T \in \Sigma^*$ will comprise what we call a *text*. Another string of characters $P \in \Sigma^*$ will be called a *search pattern*.

For integers i and j, define $T[i : j]$ as the substring of T starting from the i^{th} character and ending with the j^{th} character inclusive. We will often omit j and write $T[i :]$ to denote the suffix of T starting at the i^{th} character. Finally, we let the symbol $\circ$ denote concatenation. As a simple illustration of our notation, $(abcde[0 : 2]) \circ (cde[1 :]) = abcde$.

Now we can formally state the string matching problem: Given an input text $T \in \Sigma^*$ and a pattern $P \in \Sigma^*$, we want to find all occurrences of P in T. Closely related variants of the string matching problem ask for the first, first k, or some occurrences, rather than for all occurrences.

2.1 Tries and Compressed Tries

A commonly used string data structure is called a **trie**, a tree where each edge stores a letter, each node stores a string, and the root stores the empty string. The recursive relationship between the values stored on the edges and the values stored in the nodes is as follows: Given a path of increasing depth $p = r, v_1, v_2, ..., v$ from the root r to a node v, the string stored at node v_i is the concatenation of the string stored in v_{i-1} with the letter stored on $v_{i-1}v_i$. We will denote the strings stored in the leaves of the trie as words, and the strings stored in all other nodes as prefixes.

If there is a natural lexicographical ordering on the elements in Σ, we order the edges of every node's fan-out alphabetically, from left to right. With respect to this ordering, in order traversal of the leaves gives us every word stored in the trie in alphabetical order. In particular, it is easy to see that the fan-out of any node must be bounded above by the size of the alphabet $|\Sigma|$.

It is common practice to terminate strings with a special character $\$ \notin \Sigma$, so that we can distinguish a prefix from a word. The example trie in Figure 1 stores the four words ana\$, ann\$, anna\$, and anne\$.

If we assign only one letter per edge, we are not taking full advantage of the trie's tree structure. It is more useful to consider **compact** or **compressed** tries, tries where we remove the one letter per edge constraint, and contract non-branching paths by concatenating the letters on these paths. In this way, every node branches out, and every node traversed represents a choice between two different words. The compressed trie that corresponds to our example trie is also shown in Figure 1.

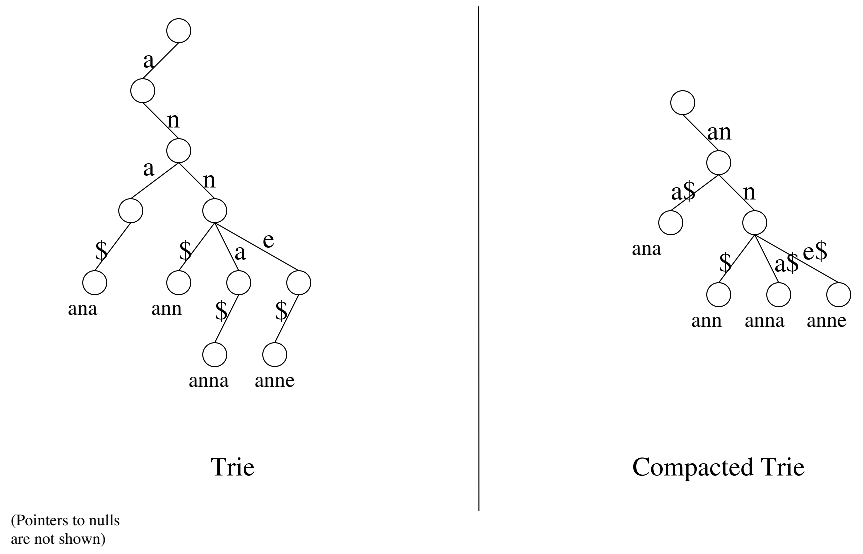

Figure 1: Trie and Conmpacted Trie Examples

2.2 Suffix Trees

A **suffix tree** is a compact trie built on the $|T| + 1$ suffixes of T. For example, if our text is the string $T = banana\$$, then our suffix tree will be built on $\{banana\$, anana\$, nana\$, ana\$, na\$, a\$\}$. For a non-leaf node of the suffix tree, define the letter depth of the node as the length of the prefix stored in the node.

Storing strings on the edges and in the nodes is potentially very costly in terms of space. For example, if all of the characters in our text are different, storage space is quadratic in the size of the text. To decrease storage space of the suffix tree to $O(|T|)$, we can replace the strings on each edge by the indices of its first and last character, and omit the strings stored in each node. We lose no information, as we are just removing some redundancies.

Suffix trees are versatile data structures that have myriad applications to string matching and related problems:

- *String Matching* To solve the string matching problem, note that a substring of T is simply a prefix of a suffix of T. Therefore, to find a pattern P, we walk down the tree, following the edge that corresponds to the next set of characters in P. Eventually, if the pattern matches, we will reach a node v that stores P. Finally, report all the leaves beneath v, as each leaf represents a different occurrence. There is a unique way to walk down the tree, since every edge in the fan out of some node must have a distinct first letter. Therefore, the total runtime of string matching is $O(|P| + k)$, where k denotes the size of the output.
- *Counting Occurrences* In this variant of the string matching problem, we must find the number of times the pattern P appears in the text. However, we can simply store in every node the size of the subtree at that node.
- *Longest Repeating Substring* To find the longest repeated substring, we look for the branching node with maximum letter depth.
- *Multiple Documents* When there are multiple texts in question, we can concatenate them with $\$_1, \$_2, ..., \$_n$. Then to find the longest common substring, we look for the node with the maximum letter depth with greater than one distinct $ below.

Compared to binary search trees, suffix trees allow for faster queries and utilize less storage, especially for texts with a large number of suffixes. Compared to hash tables, suffix trees allow for faster worst case queries, and avoid problems caused by collisions and computing hash functions.

3 Suffix Arrays

Suffix trees are powerful data structures with applications in fields such as computational biology, data compression, and text editing. However, a **suffix array**, which contains most of the information in a suffix tree, is a simpler and more compact data structure for many applications. The only drawback of a suffix array is that is that it is less intuitive and less natural as a representation. In this section, we define suffix arrays, show that suffix arrays are in some sense equivalent to suffix trees, and provide a fast algorithm for building suffix arrays.

3.1 Creating Suffix Arrays

Let us store the suffixes of a text T in lexicographical order in an intermediate array. Then the suffix array is the array that stores the index corresponding to each suffix. For example, if our text is $banana\$$, the intermediate array is $[\$, a\$, ana\$, anana\$, banana\$, na\$, nana\$]$ and the suffix array is $[6, 5, 3, 1, 0, 4, 2]$. Since suffixes are ordered lexicographically, we can use binary search to search for a pattern P in $O(|P| \log |T|)$ time.

We can compute the length of the longest common prefix between neighboring entries of the intermediate array. If we store these lengths in an array, we get what is called the **LCP array**. These LCP arrays can be constructed in $O(|T|)$ time, a result due to Kasai et al [5]. In the example above, the LCP array constructed from the intermediate array is $[0, 1, 3, 0, 0, 2]$. Using LCP arrays, we can improve pattern searching in suffix arrays to $O(|P| + \log |T|)$ (see homework).

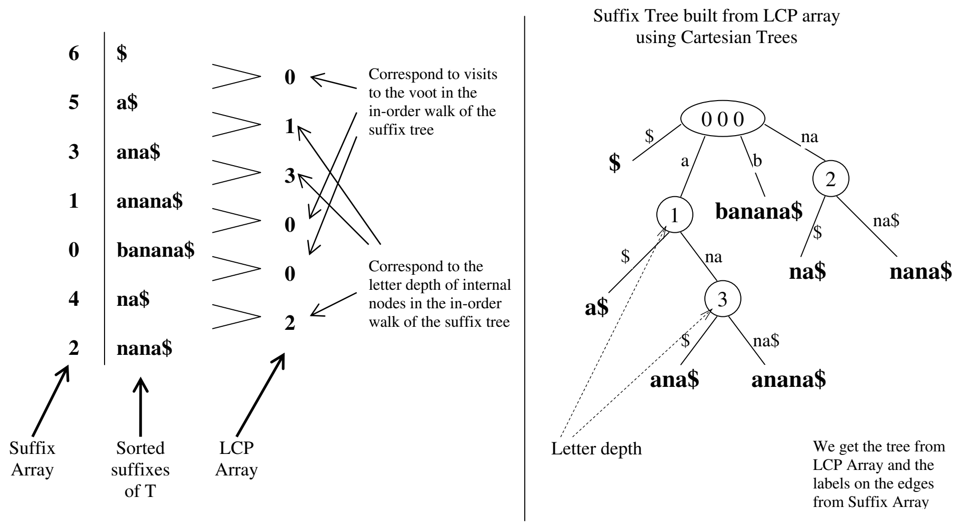
6 $
5 a$
3 ana$
1 anana$
0 banana$
4 na$
2 nana$
0
1
3
0
0
2
Correspond to visits to the voot in the in-order walk of the suffix tree
Correspond to the letter depth of internal nodes in the in-order walk of the suffix tree
Suffix Array
Sorted suffixes of T
LCP Array
Suffix Tree built from LCP array using Cartesian Trees
0 0 0
$
a
b
na
$
1
banana$
2
$
na
$
na$
a$
3
na$
nana$
$
na$
ana$
anana$
Letter depth
We get the tree from LCP Array and the labels on the edges from Suffix Array

Figure 2: Suffix Array and LCP Array examples and their relation to Suffix Tree

3.2 Suffix Arrays and Suffix Trees

To motivate the construction of a suffix array and a LCP array, note that a suffix array stores the leaves of the corresponding suffix tree, and the LCP array provides information about the height of internal nodes, as seen in Figure 2. Our aim now is to show that suffix trees can be transformed into suffix arrays in linear time and vice versa.

To formalize this intuition, we define the **Cartesian Tree** of an LCP array. To build a Cartesian tree, store the minimum over all LCP array entries at the root of the Cartesian tree, and recurse on the remaining array pieces. For a concrete example, see Figure 3 below.

3.2.1 From Suffix Trees to Suffix Arrays

To transform a suffix tree into a suffix array, we simply run an in-order traversal of the suffix tree. As noted earlier, this process returns all suffixes in alphabetical order.

3.2.2 From Suffix Arrays to Suffix Trees

To transform a suffix array back into a suffix tree, create a Cartesian tree from the LCP array associated to the suffix array. The numbers stored in the nodes of the Cartesian tree are the letter depths of the internal nodes! Hence, if we insert the entries of the suffix array in order into the Cartesian tree as leaves, we recover the suffix tree. In fact, we are rewarded with an augmented suffix tree with informationa bout the letter depths.

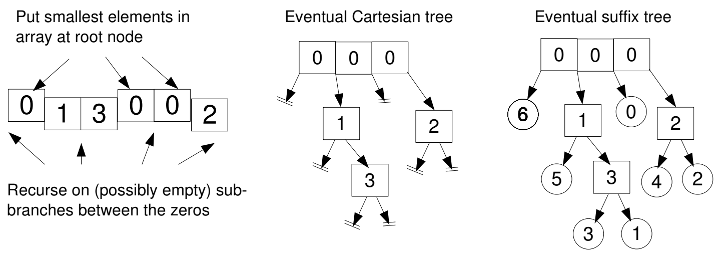
Put smallest elements in array at root node
0 1 3 0 0 2
Recurse on (possibly empty) sub-branches between the zeros
Eventual Cartesian tree
0 0 0
1
2
3
Eventual suffix tree
0 0 0
6
1
0
2
5
3
4
2
3
1

Figure 3: Constructing the Cartesian tree and suffix tree from LCP array

3.3 DC3 Algorithm for Building Suffix Arrays

Here, we give a description of the DC3 (Difference Cover 3) divide and conquer algorithm for building a suffix array in $O(|T| + sort(\Sigma))$ time. We closely follow the exposition of the paper by Karkkainen-Sanders-Burkhardt [3] that originally proposed the DC3 algorithm. Because we can create suffix trees from suffix arrays in linear time, a consequence of DC3 is that we can create suffix trees in linear time, a result shown independently of DC3 by Farach [2], McCreight [7], Ukkonen [9], and Weiner [10]

1. Sort the alphabet Σ. We can use any sorting algorithm, leading to the $O(\text{sort}(\Sigma))$ term.

2. Replace each letter in the text with its rank among the letters in the text. Note that the rank of the letter depends on the text. For example, if the text contains only one letter, no matter what letter it is, it will be replaced by 1. This operation is safe, because it does not change any relations we are interested in. We also guarantee that the size of the alphabet being used is no larger than the size of the text (in cases where the alphabet is excessively large), by ignoring unused alphabets.

3. Divide the text T into 3 parts and package triples of letters into *megaletters*. More formally, form T_0, T_1, and T_2 as follows:

$$\begin{aligned} T_0 = &\ < (T[3i], T[3i+1], T[3i+2]) \quad \text{for} \quad i = 0, 1, 2, \ldots > \\ T_1 = &\ < (T[3i+1], T[3i+2], T[3i+3]) \quad \text{for} \quad i = 0, 1, 2, \ldots > \\ T_2 = &\ < (T[3i+2], T[3i+3], T[3i+4]) \quad \text{for} \quad i = 0, 1, 2, \ldots > \end{aligned}$$

Note that T_i's are just texts with $n/3$ letters of a new alphabet Σ^3. Our text size has become a third of the original, while the alphabet size has cubed.

4. Recurse on $< T_0, T_1 >$, the concatenation of T_0 and T_1. Since our new alphabet is of cubic size, and our original alphabet is pre-sorted, radix-sorting the new alphabet only takes linear time. When this recursive call returns, we have all the suffixes of T_0 and T_1 sorted in a suffix array. Then all we need is to sort the suffixes of T_2, and to merge them with the old suffixes to get suffixes of T, because

$$\texttt{Suffixes}(T) \cong \texttt{Suffixes}(T_0) \cup \texttt{Suffixes}(T_1) \cup \texttt{Suffixes}(T_2)$$

If we can do this sorting and merging in linear time, we get a recursion formula $T(n) = T(2/3n) + O(n)$, which gives linear time.

5. Sort suffixes of T_2 using radix sort. This is straight forward to do once we note that

$$T_2[i:] \cong T[3i+2:] \cong (T[3i+2], T[3i+3:]) \cong (T[3i+2], T_0[i+1:]).$$

The logic here is that once we rewrite $T_2[i:]$ in terms of T, we can pull off the first letter of the suffix and pair it with the remainder. We end up with something where the index $3i+3$ corresponds with the start of a triplet in T_0, specifically, $T_0[i+1]$, which we already have in sorted order from our recursive call.

Thus, we can radix sort on two coordinates, the triplet $T_0[i+1]$ and then the single alphabet $T[3i+2]$, both of which we know the sorted orders of. This way, we get $T_2[i:]$ in sorted order. Specifically, the radix sort is just on two coordinates, where the second coordinate is already sorted.

6. Merge the sorted suffixes of T_0, T_1, and T_2 using standard linear merging. The only problem is finding a way to compare suffixes in constant time. Remember that suffixes of T_0 and T_1 are already sorted together, so comparing a suffix from T_0 and a suffix from T_1 takes constant time. To compare against a suffix from T_2, we will once again decompose it to get a suffix from either T_0 or T_1. There are two cases:

- Comparing T_0 against T_2:

$$\begin{array}{rrcl} & T_0[i:] & \texttt{vs} & T_2[j:] \\ \cong & T[3i:] & \texttt{vs} & T[3j+2:] \\ \cong & (T[3i], T[3i+1:]) & \texttt{vs} & (T[3j+2], T[3j+3:]) \\ \cong & (T[3i], T_1[i:]) & \texttt{vs} & (T[3j+2], T_0[j+1:]) \end{array}$$

  So we just compare the first letter and then, if needed, compare already sorted suffixes of T_0 and T_1.

- Comparing T_1 against T_2:

$$\begin{array}{rrcl} & T_1[i:] & \texttt{vs} & T_2[j:] \\ \cong & T[3i+1:] & \texttt{vs} & T[3j+2:] \\ \cong & (T[3i+1], T[3i+2], T[3i+3:]) & \texttt{vs} & (T[3j+2], T[3j+3], T[3j+4:]) \\ \cong & (T[3i+1], T[3i+2], T_0[i+1:]) & \texttt{vs} & (T[3j+2], T[3j+3], T_1[j+1:]) \end{array}$$

  So we can do likewise by first comparing the two letters in front, and then comparing already sorted suffixes of T_0 and T_1 if necessary.

4 Document Retrieval

The statement of the document retrieval problem is as follows: Given a collection of texts $\{T_1, T_2, ..., T_n\}$, we want to find all distinct documents that contain an instance of a query pattern P. Naively, we can store the suffixes of the text $T = T_1 \circ \$_1 \circ T_2 \circ \$_2 \circ ... \circ T_n \circ \$_n$ in a suffix tree and query for the pattern. However, the result of this query will include multiple matches of P within the same text, whereas we only want to find which texts contain P. Hence, a simple query doesn't give us the answer we want, and is costlier than we would like. The fix we present will follow the paper by Muthukrishan [8].

Consider the suffix array A of the augmented text T. The node corresponding to a pattern P in the suffix tree corresponds to some interval $[i, j]$ in the suffix array. Now the texts containing the pattern are precisely the indices of all the dollar signs appearing in the interval $[i, j]$ in A. Hence, solving document retrieval is equivalent to finding the position of the first dollar sign of each type within the interval $[i, j]$, as the suffixes at those positions cannot contain repeats of P.

To this end, augment the suffix array by connecting each dollar sign of a given type to the previous dollar sign of the same type. Then the queries can be answered with a **Range Minimum Data Structure**. In particular, this gives us a $O(|P| + d)$ runtime, where d is the number of texts to report.

References

R. Boyer and J. Moore. *A fast string searching algorithm.* Communications of the ACM, 20(10):762772, 1977.

M. Farach. *Optimal suffix tree construction with large alphabets.* In Proc. 38th Annual Symposium on Foundations of Computer Science, pages 137143. IEEE, 1997.

Juha Karkkainen, Peter Sanders, *Simple linear work suffix array construction*, In Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP03). LNCS 2719, Springer, 2003, pp. 943-955

R. M. Karp and M. O. Rabin. *Efficient randomized pattern-matching algorithms.* IBM Journal of Research and Development, 31:249260, March 1987.

T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. *Linear-time longest-common-prefix computation in suffix arrays and its applications.* In Proc.12th Symposium on Combinatorial Pattern Matching (CPM 01), pages 181192. Springer-Verlag LNCS n. 2089, 2001.

D. E. Knuth, J. H. Morris, and V. R. Pratt. *Fast pattern matching in strings.* SIAM Journal of Computing, 6(2):323350, 1977.

E. M. McCreight. *A space-economic suffix tree construction algorithm.* J. ACM,23(2):262272, 1976.

S. Muthukrishnan, *Efficient algorithms for document retrieval problems.* SODA 2002: 657-666

E. Ukkonen. *On-line construction of suffix trees.* Algorithmica, 14(3):249260, 1995.

P. Weiner. *Linear pattern matching algorithm.* In Proc. 14th Symposium on Switching and Automata Theory, pages 111. IEEE, 1973.