6.851: Advanced Data Structures Spring 2007

Lecture 8 — March 7, 2007

Prof. Erik Demaine *Scribe: Aston Motes and Kah Keng Tay*

1 Overview

In this lecture we are starting a sequence of lectures about string data structures. Today's lecture will be on the string matching problem. In particular, we will consider suffix trees and their various applications.

In the string matching problem we are given an alphabet Σ, text T, and a pattern P, and ask various questions such as:

- Is there a substring of T matching P?
- How many substrings of T match P?
- Where are first/any k occurrences of P in T?
- Where are all occurrences of P in T?

There are two different approaches to solving such problems. The algorithmic approach solves a new instance of problem every time the algorithm is run. There are a number of well known algorithms that achieve linear time in the size of T such as Rabin-Karp [2], Boyer-Moore [1] and Knuth-Morris-Pratt [4]. In the data structural approach we can preprocess T, and then answer a query involving a pattern P much faster.

We will see today how to answer queries in $O(|P|)$ time using $O(|T||\Sigma|)$ space and $O(|T| + \text{sort}(\Sigma))$ preprocessing time. It is interesting to note that the space required is linear in the number of words, not the number of bits, as one might hope for. This is a $\lg n$ factor larger than the number of bits. We will discuss how to achieve linear space in the number of bits during the lecture on succinct data structures.

2 Relevant Concepts

Tries. A trie is a tree with children branches labeled with distinct letters from the alphabet Σ. The branches are ordered alphabetically. A trie can be built for any set of strings for any alphabet. The branching factor of every internal node is $|\Sigma|$. We will append a dollar sign character, \$, to the end of all strings. This is necessary in order to distinguish whether a particular path in the trie represents a string, or is just a prefix of one.

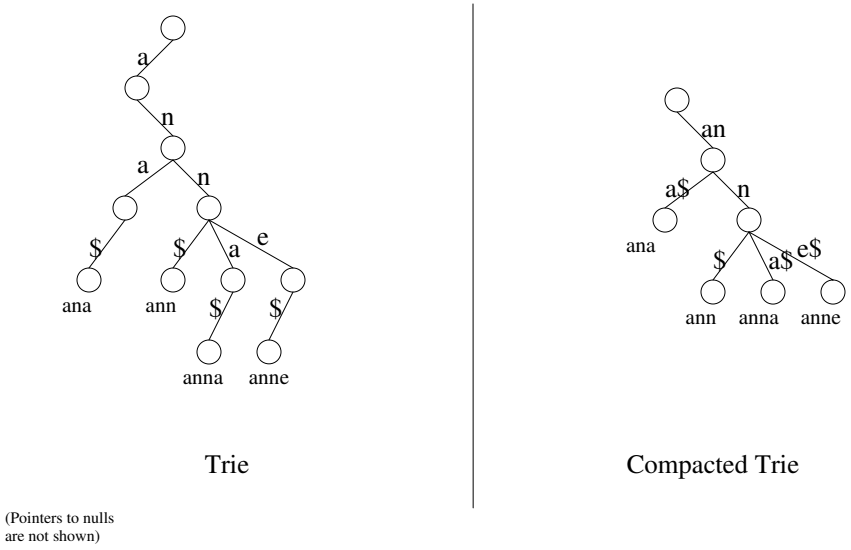

Figure 1: Trie and Conmpacted Trie Examples

Compressed Tries. We can coalesce non-branching paths in a given trie to reduce the number of edges to be at most twice the number of leaves (which is the number of strings we build the trie on). This yields a compressed or compacted trie. Figure 1 gives an example trie for the set of strings {ana, ann, anna, anne}, as well as its compact version.

To see why the number of edges is linear to the number of leaves, we observe that in compressing the trie, we remove all internal nodes (non-leaf nodes) that are non-branching. The only nodes that remain are branching nodes and leaves. Now if we remove the deepest branching node and just one of its children leaf nodes, we can repeat this process until there are no more branching nodes left. There will, however, be at least one leaf node remaining. Thus the number of branching nodes is strictly less than the number of leaf nodes. A tree has one fewer edge than number of nodes, and so the number of edges is at most twice the number of leaves.

Suffix Trees. The suffix tree of text T is a compacted trie on all the suffixes of T. For example, if our text is "banana$", the suffix tree is a compacted trie built for the set { banana$, anana$, nana$, ana$, na$, na$, a$, $ }. It is common to refer to a suffix by the index (starting at 0) of its first character.

To get linear space, all we have to do is not store the labels on edges explicitly, but, instead, store two indices: the position in T of the first and last characters of the label. Note that this is possible because each label is a substring of a suffix of T, thus is a substring of T. This way we have constant space for each edge, and the total complexity of the suffix tree is linear in the size of T (in words).

Suffix trees are important because they are very easy to query. Given a pattern P, all occurrences

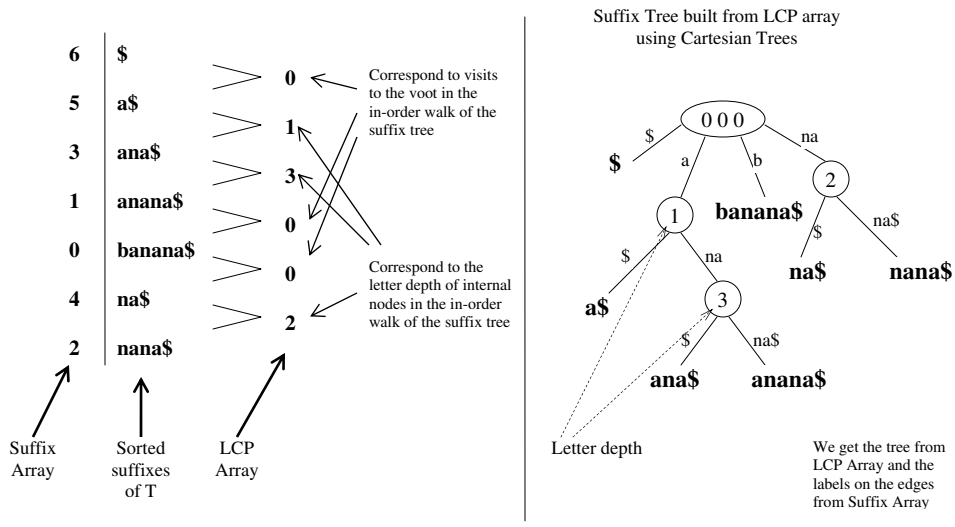

Figure 2: Suffix Array and LCP Array examples and their relation to Suffix Tree

of P in T can be found and reported in time $O(|P| + \text{output})$. This is done by simply walking down the suffix tree, always taking the edge that corresponds to the next character in P. Note that edges below any node differ in their first character, so it is easy to select which edge we want. Then, we can compare the successive characters in the pattern with the label. When we have explored the path R starting from the root such that labels on its edges give us P when concatenated, all the occurrences of P in T are in the subtree whose root is lowest node in R.

Suffix Arrays. A suffix array A of T is just a sorted array of suffixes of T. To avoid quadratic space we store only the indices of the suffixes, instead of full suffixes of T. For example, if T is equal to "banana$", the suffix array is [6, 5, 3, 1, 0, 4, 2] — corresponding to the alphabetical ordering of the suffixes { $, a$, ana$, anana$, banana$, na$, nana$ }. The suffix array is equivalent to an in-order traversal of the leaves of the suffix tree (see Figure 2). We can search for occurrences of P directly on the suffix array using binary search in $O(|P| \lg |T|)$ time. This can be improved to $O(|P| + \lg |T|)$.

Longest Common Prefix Array. We can consider an LCP array of size $|T| - 1$, in which the i^{th} element is the length of the longest common prefix of $A[i]$ and $A[i + 1]$, where A is the suffix array of T. For the "banana" example above this array will be [0, 1, 3, 0, 0, 2]. This process is visualized in the Figure 2.

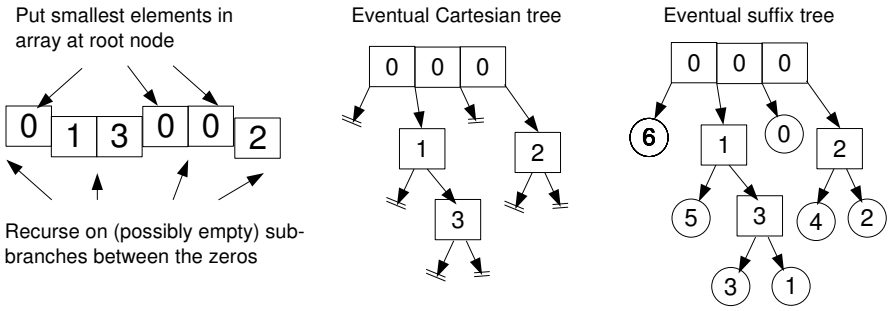

Figure 3: Constructing the Cartesian tree and suffix tree from LCP array

3 Constructing Suffix Trees

We now show how to construct suffix trees in linear time. There are many known algorithms for this, starting with classic solutions by Weiner [7] and McCreight [5] in the mid 70s. We will look at a recent construction algorithm found by Kärkkäinen and Sanders [3], which is surprisingly short and clean. This algorithm actually builds suffix arrays, but given a suffix array and its corresponding LCP array, we can also build a suffix tree.

3.1 Construction of the Suffix Tree from the LCP Array and the Suffix Array

Let's start by building up intuition on how the suffix tree S, the LCP array L, and the suffix array A are related to each other. First, notice that to get A from S, we can simply do an in-order walk of S. Furthermore, notice that zeros in L correspond to visits of the root in this in-order walk. But what do the other numbers in L mean? They are the "letter depth" of the internal nodes of the suffix tree. (By "letter depth of a node" we mean its depth in a non-compacted suffix tree, i.e. the number of letters that the path to it form the root contains). Finally, once we have the internal nodes of the suffix tree, we can distribute the edge labels in one pass of in-order walk since we know the order of the suffixes.

With the intuition in place, we note that we can use the Cartesian tree of L to get our suffix tree. Given the array L, we take the smallest elements and put them at the root. We split the array up based on the positions of these smallest elements, and then we recurse on sub-branches of this tree with the partitioned array. See Figure 3.

Once we have a Cartesian tree, we note that it represents the internal nodes of our suffix tree. We replace the null pointers with leaf nodes, which can be filled up with elements from our suffix array as we do an in-order traversal of the tree. Edges can be labeled similarly.

3.2 Construction of the LCP and Suffix Arrays

Let us introduce some notations. $T[i:]$ denotes the suffix of T starting at index i; <> denotes an array, (a, b) denotes an ordered pair, and (a, b, c) denotes an ordered triple. We will use $\cong$ sign to

denote different representations of the same data. How to transform one representation to another will be made obvious.

The algorithm below will construct the LCP and suffix arrays in $O(|T| + \text{sort}(\Sigma))$ time. We explain each step of the algorithm.

1. Sort Σ. In the first iteration use any sorting algorithm, leading to the $O(\text{sort}(\Sigma))$ term. In the following iterations we can use radix sort to sort in linear time (see below).

2. Replace each letter in the text with its rank among the letters in the text. This is called alphabet reduction. 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. Note that this is safe to do, 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 consider triples of letters to be one *megaletter*, i.e. change the alphabet. More formally, form T_0, T_1, and T_2 as follows:

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

Note that T_i's are just texts with $n/3$ letters of a new Σ^3 alphabet. 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 alphabet is just of cubic size, and we have our original alphabet pre-sorted, radix-sorting the new alphabet will take linear time in subsequent recursions. 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 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 do this sorting and merging in linear time, we get a recursion formula $T(n) = T(2/3n) + O(n)$, which gives linear time as desired.

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 we can first rewrite $T_2[i:]$ in terms of T, then 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. the radix sort is just on two coordinates, and the second one is already sorted. After sorting, we can create an LCP array for T_2. To do this, simply check if each of $T[3i+2]$ is equal to the first letter

of the preceding suffix. If they are distinct, the LCP between them is 0. Otherwise, we just lookup the LCP of corresponding $T_0[i+1:]$'s and add 1 to it.

6. Merge the sorted suffixes of T_0, T_1, and T_2. We use 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 again decompose it to get a suffix from either T_0 or T_1. There are two cases:

- Comparing T_0 against T_2:

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

  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:

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

  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.

Finally, we have to take care of the LCPs. This can be done exactly as above. Compare the first two characters of neighboring suffixes. If either the first pair or the second pair are different, the LCP is 0 or 1. If both are equal, then get the already computed LCP of the suffixes starting from third letter and add 2 to it.

To summarize at a high level, given a text, we compute its suffix and LCP arrays using the algorithm from above, then we construct the suffix tree using a Cartesian tree on the LCP array. Finally, we answer queries by walking down the suffix tree and returning all the suffixes located in the subtree rooted where the search ended.

4 Applications

There are quite a number of applications and variations of suffix trees:

- if we are interested in counting the number of occurrences of P, we can augment suffix trees by storing subtree-sizes at every node

- if we want to know the longest repeated substring in the text, we just need to find the deepest (in the "letter depth" sense) internal node in the suffix tree.
- multiple documents can be easily combined by introducing indexed dollar signs between texts: $T_1\$_1T_2\$_2\ldots$
- if we want the longest common substring between two documents, we combine them as above, and find the deepest node with both $\$_1$ and $\$_2$ in its subtree. The running time is $O(|T|)$

Document Retrieval. In the document retrieval problem (see [6]), we have a collection of documents and a pattern P and we want to find all distinct documents that contain P. We want a time bound of $O(|P| + d)$, where d is the number of documents to report. This cannot be achieved through a simple search, because we may be reporting many matches inside the same document. Nonetheless, we can still solve the problem using suffix trees, in combination with range-minimum queries (RMQ) which we will see in Lecture 16.

As above, we concatenate documents separating them with indexed dollar signs, and build a common suffix tree S. Consider a suffix array A of the concatenated texts (we can get it by an in-order walk of the suffix tree). When we search for P in S, we get a subtree, which corresponds to an interval $[i, j]$ in A. Notice that the documents that contain P are exactly the documents such that the interval $[i, j]$ in A contains at least one of the documents' dollar signs. There could be multiple dollar signs for each document, corresponding to each occurrence of P in the document. Let each dollar sign store a pointer (an index into the suffix array) to the previous dollar sign of its type (i.e. belonging to the same document). Our goal now is to find the dollar signs in $[i, j]$ with a pointer that points before i; these are the first occurrences of each type, corresponding to the first occurrence of P in each document. Finding these allow us to ignore repeat occurrences of P. To accomplish this, we find the position k of the minimum element in $[i, j]$ using a range-minimum query (RMQ) data structure, where the minimum element is one whose pointer is the minimum. This runs in $O(1)$ time. If the element points to something $\geq i$, we are done. Otherwise, we output that document, and recurse on $[i, k-1]$ and $[k+1, j]$ to find another minimum element excluding k, which could indicate another document having P. This would run in $O(d)$ time for d documents.

Longest Palindrome. Other interesting applications can be found if we combine suffix trees with lowest common ancestor (LCA) queries. Finding the LCA of 2 nodes in a suffix tree is identical to the range-minimum query problem and can be done in $O(1)$ time.

Notice that the LCA of two leaves is the longest prefix match of the two suffixes. Using this, we can find the longest palindrome in the string in $O(|T|)$ time. The longest common prefix of $T[i:]$ and $\text{reverse}(T)[-i:]$ gives the longest palindrome centered at i. This can be computed in constant time using an LCA query, so we can find the longest palindrome overall in linear time.

References

[1] R. S. Boyer, J. S. Moore, *A fast string searching algorithm.* Communications of the ACM, 1977. 20:762-772.

[2] Richard M. Karp, Michael O. Rabin, *Efficient randomized pattern-matching algorithms*. Technical Report TR-31-81, Aiken Computation Laboratory, Harvard University, 1981.

[3] Juha Kärkkäinen, Peter Sanders, *Simple linear work suffix array construction*. In Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP'03). LNCS 2719, Springer, 2003, pp. 943-955.

[4] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt, *Fast pattern matching in strings*. SIAM Journal on Computing, 6(2): 323350, 1977.

[5] Edward M. McCreight, *A Space-Economical Suffix Tree Construction Algorithm*, J. ACM, Vol 23 No 2, ACM Press 1976, pp 262-272.

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

[7] P. Weiner, *Linear pattern matching algorithms*, Conf. Record, IEEE 14th Annual Symposium on Switching and Automata Theory, pp. 1-11.