6.851: Advanced Data Structures Spring 2007

Lecture 9 — March 12, 2007

Oren Weimann *Scribe: Kuat Yessenov and Kevin Wang*

1 Overview

In the last lecture we began the topic of string matching and document retrieval. In the string matching problem, we are given a an alphabet Σ, text T, and a pattern P. We may ask questions such as whether P is a substring of T, how many occurrences of P in T are there, and where do they occur? To solve this problem, we introduced suffix arrays and suffix trees. Suffix arrays are arrays of size $O(|T|)$ that store all suffixes of T in lexicographic order. We can query for the existence of P in T in $O(|P| + \lg |T|)$ time with the aid of a longest common prefix array. Suffix trees, which are compressed tries of all suffixes of T, can answer search queries in $O(|P|)$ time, but require $O(|T| \cdot |\Sigma|)$ space and $O(|T| + sort(|\Sigma|))$ preprocessing time. The space can be reduced to $O(|T|)$ by storing a BST instead of an array in every suffix tree node. However, this comes at the cost of $O(|P| \lg |\Sigma|)$ query time.

In this lecture we introduce a new data structure called a *Suffix Tray* that combines the best aspects of suffix arrays and suffix trees. We will achieve $O(|P| + lg|\Sigma|)$ time queries with only $O(|T|)$ space. We then discuss approximate string matching–that is matching strings within an error tolerance under different distance metrics such as the Hamming distance or edit distance. We examine several solutions to this problem with increasingly better run times.

2 Suffix Trays

Suffix trays were invented by Cole, Kpelowitz, and Lewenstein in 2006 [1] and achieve $O(|P|+lg|\Sigma|)$ queries. A suffix tray is a combination of a suffix tree ST and a suffix array SA over the same alphabet Σ and text T. We note that the internal nodes in a subtree of ST correspond to an interval in SA. Given an interval I in SA, we can search for a pattern P in I in $O(|P| + \lg |I|)$ time with the aid of a longest common prefix array. The idea behind suffix trays is to begin the search inside ST, but then branch off into SA and search the appropriate (and short) interval where P may lie. The key part is figuring out when to jump from ST to SA.

2.1 Suffix Tree Decomposition

To solve this problem, we decompose the nodes of ST into one of three types:

1. Σ-node – This has at least $|\Sigma|$ leaves in its subtree, but each of its children do not.
2. Branching-Σ-node – This has at least two children that have $\geq |\Sigma|$ leaves in their subtree.
3. All the other – The rest of the nodes that don't fall into either of the above categories.

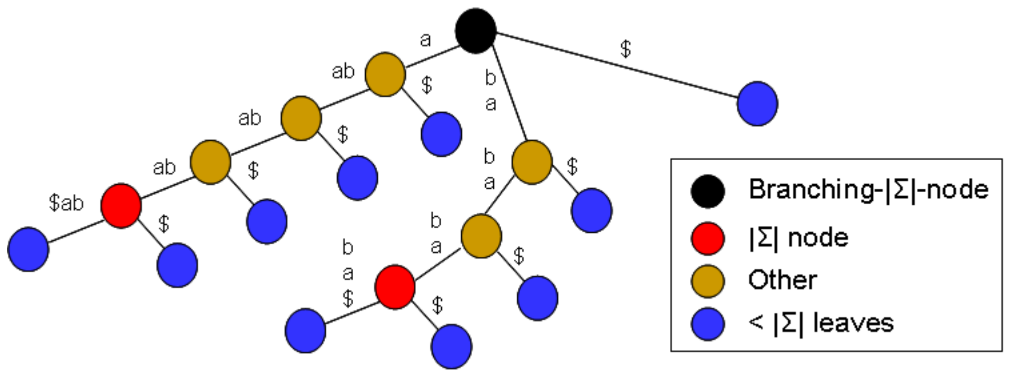

Figure 1: Suffix Tree for the text T = abababababa$.

Figure 1 shows a Σ decomposition for the text T = "ababababa". The \$ symbol denotes the end of string.

2.1.1 Σ-nodes

We note that the number of leaves in a subtree of a Σ-node is $\leq |\Sigma|^2$. This is because none of the children of the Σ-node has more than $|\Sigma|$ leaves in their subtrees, so in the worst case we get $|\Sigma|$ children $\times$ $|\Sigma|$ leaves per child's subtree $= |\Sigma|^2$ leaves. Therefore, the time to search through the interval spanned by a Σ-node is $O(|P| + \lg|\Sigma|)$. This means if we can get to a Σ-node in time asymptotically equivalent to that, then we've achieved our desired runtime.

2.1.2 Branching-Σ-nodes

The key point to realize here is that the number of branching-Σ-nodes is $O(\frac{|T|}{|\Sigma|})$. To see this, note that there are $O(\frac{|T|}{|\Sigma|})$ Σ-nodes (as they are all disjoint) and that there is one-to-one correspondence between Branching-Σ-nodes and Σ-nodes

This useful fact allows us to store a $|\Sigma|$ sized array for $O(1)$ access to children in every branching-Σ-node. This only costs $O(\frac{|T|}{|\Sigma|}) \cdot |\Sigma| = O(|T|)$ space.

2.1.3 Other nodes

The remaining nodes have at most one Σ-node in its subtree. This means of its $k \leq |\Sigma|$ immediate children, one of them will have Σ-node in its subtree. We label this edge with a σ during the construction. The other $k-1$ will have $\leq |\Sigma|$ leaves. These $k-1$ children therefore form two intervals of size $\leq |\Sigma|^2$, so searching those interval will take $O(|P| + \lg|\Sigma|)$ time. Refer to Figure 2 for a diagram.

2.2 Navigation (Search) Algorithm

Searching through a suffix tray is straightforward. At each of the three nodes, we do the following:

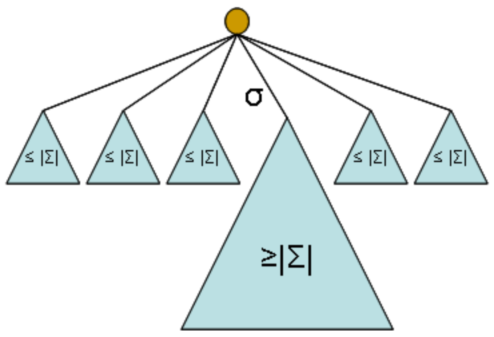

Figure 2: The subtree structure of an "other" node. Only one subtree can have $\geq |\Sigma|$ leaves, making the remaining subtrees quickly searchable.

1. Σ-node – Search through the suffix array interval corresponding to its subtree: $O(|P| + \lg |\Sigma|)$ time.
2. Branching-Σ-node – Look at Σ array and go on to child: $O(1)$ time.
3. Other – Look at character along σ branch. If match, we go down that branch, else search the other children via the suffix array: $O(1)$ time in the first case, $O(|P| + \lg |\Sigma|)$ in the second.

It costs $O(|P|)$ to walk branching-Σ-nodes and compare single characters (when we hit other nodes), and $O(|P| + \lg |\Sigma|)$ time to search a $|\Sigma|^2$ size interval in SA. Therefore, the total runtime $O(|P| + \lg |\Sigma|)$ as desired, with only $O(|T|)$ space.

3 Approximate String Matching

The approximate string matching problem is defined as follows: given an error tolerance k and a text T, we need find occurrences of a pattern P in T within *error* k. There are commonly used error metrics.ways to measure error, such as:

- Hamming distance – the number of character mismatches.
- Edit distance – the number of edits (insertions, deletions, substitutions) needed to produce an exact match.

In this lecture, we work primarily with the Hamming distance.

In the "online" variation of the problem, we are given both P and T together. We describe an algorithm which can perform the query in $O(|T| \cdot k)$ time. Given P and T, we construct a suffix tree on the concatenated string $P\$T$ supporting lowest common ancestor queries. For every location i in T, we perform k LCA queries to check if P appears in this location as described in Figure 3. At each step, we use the least common ancestor to identify the longest common prefix of two suffixes and move to the next character after the first mismatch.

In the "offline" variation, we would like to construct a data structure which can preprocess a text T and answer queries which are given online. The best currently-known bounds, given by [5], are:

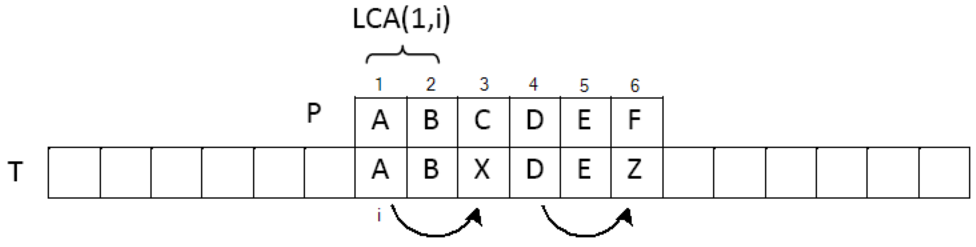

Figure 3: Approximate string matching "online" algorithm.

- space and preprocessing: $O(|T|\frac{(c\lg|T|)^k}{k!})$
- query: $O(|P| + \frac{(c\lg|T|)^k}{k!}\lg\lg|T|) + 3^k \cdot (\#\text{ occurrences}))$ (the last term appears only for the edit distance)

4 Searching with Wildcards

We will focus on a subproblem of the above. Again we are given T and k for preprocessing. But now, the query consists of a pattern P that contains at most k "don't care" characters (the ? wildcards). We are to find "exact" matches of P, where wildcards match any character. The best known solution [5] solves the problem in $O(|T|\lg^k|T|)$ space and $O(2^k \lg\lg|T| + |P| + \#\text{ occurrences})$ query time.

All the solutions we will discuss involve the use of suffix trees. An obvious simple solution is to walk down the suffix tree while matching P and simply branch $|\Sigma|$ ways every time a ? is encountered in P. Thus, queries take at most $O(|\Sigma|^k \cdot |P|)$. Compared to the best solution we mentioned above, the simple solution is lacking in that there is a dependence on alphabet size (which may be very large) and that the dependence on the pattern length is multiplicative instead of additive.

We now describe how to improve the Σ^k factor to 2^k. To do so, we perform a decomposition of the suffix tree similar to heavy-light decomposition we saw before. We call an edge to a child *heavy* if the subtree rooted in that child contains the most nodes among all other children. The intuition is that we now only differentiate between light edges and a single heavy edge whenever we encounter a ?. Because light subtrees are small, we group them together in one big chunk. Specifically, for each node in the *primary* suffix tree, we store a *secondary* suffix tree on the union of light subtrees of that node, except the first characters of each subtree.

If $k > 1$, we recurse k times so that there are $k+1$ "levels" of secondary trees. Since the light depth is $O(\lg|T|)$ in a heavy-light decomposition, each leaf appears in $O(\lg^k|T|)$ trees. Thus, the solution takes $O(|T|\lg^k|T|)$ space and preprocessing, and $O(2^k \cdot |P|)$ query time.

As mentioned above, there is a way to make the $|P|$ factor additive in the query time. The idea is to find a way to quickly (in $\lg\lg|T|$ time) determine whether we should take the light/heavy branch. We will not delve into the specifics of this solution, but mention that using the suffix tree from above, least common ancestor queries, and *level ancestor* queries, we can detect whether one of the 2^k branches is "good".

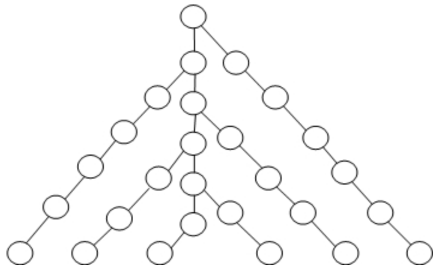

Figure 4: The maximum path depth in the long path decomposition can be as high as $O(\sqrt{n})$.

5 Level Ancestor Problem

For the rest of this lecture, we shift our attention to the level ancestor problem. We are given a static rooted tree, which can be preprocessed. Then, for the level-ancestor query we would like to find for any node v and a number l the l^{th} ancestor of v. This is equivalent to finding the depth-d ancestor of v, where $d + l = \text{depth}(V)$.

Various solutions to this problem have been proposed [4, 6, 2, 3]. We will discuss the solution in [3], by Bender and Farach-Colton. We present gradual steps leading to a solution that encompasses the different improvements, and ends up taking linear space and preprocessing time, with constant query time.

5.1 Lookup table

An immediate solution is to store a *lookup table* for each node on all possible queries. This gives total space $O(n^2)$ and constant query time.

5.2 Jump pointers

For another solution we want to keep *jump pointers* in the same way as we do for skip lists. Each node stores pointers to 1st, 2nd, 4th, …, 2^i-th ancestors. This takes $O(n \lg n)$ total space. To perform queries, we recursively go up $\lfloor\lfloor l \rfloor\rfloor = 2^{\lfloor \lg l \rfloor}$ times. We know that $l/2 < \lfloor\lfloor l \rfloor\rfloor \leq l$, so queries take $O(\lg n)$.

5.3 Long path decomposition

We preprocess the tree as follows:

1. Take a longest root-to-leaf path and recurse on the remaining connected components;
2. Store each path as an array ordered by depth (so that nodes in the path may be randomly accessed), and store a pointer to its parent path;
3. For each node, store the path to which it belongs and its index in the array for the path.

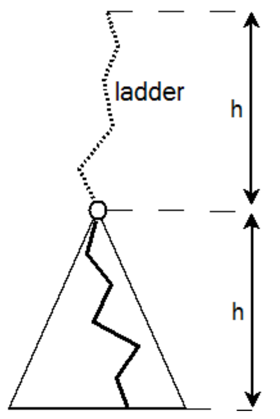

Figure 5: The ladder decomposition.

Let HEIGHT(v) be the height of node v in its path, i.e. the number of nodes beneath it.

The space usage is clearly $O(n)$. To answer queries, get the path for the node and check if it is long enough for the queried ancestor height; if not, recurse. The query time is therefore linear in the number of paths traversed. Unfortunately, Figure 4 shows that the number of paths can be as high as $O(\sqrt{n})$.

5.4 Ladder decomposition

This extends the long path decomposition in a simple but effective fashion. We extend the length of each path upwards by a factor of 2 (i.e. extend a path of length l up by l levels), unless, of course, we hit the root while going up. The extension is called the *ladder*. See Figure 5.

Instead of storing an array with a path, we store an array with the path plus the ladder. The space is still linear, because the ladder can be amortized against the path. However, queries can now be answered in $O(\lg n)$ time if we perform them as in long path decomposition but also use ladders. Indeed, notice that any node v of height h lies on a path in longest path decomposition of length at least h. Then the height of the top node in v's ladder is at least $2h$. Therefore, each step either doubles the height, or finishes.

5.5 Combine ladder decomposition & jump pointers

The idea is that jump pointers start with large jumps (that become exponentially smaller), and the ladder decomposition starts with small jumps (that become exponentially larger). Then, we can combine these and have one big jump with jump pointers and another big jump with ladders. A query proceeds as follows:

- take one jump pointer to go up $\lfloor\lfloor l\rfloor\rfloor > l/2$ nodes. Call the intermediate node that is reached v'. We have HEIGHT(v') $> l/2$, because we know there is a path of length $\lfloor\lfloor l\rfloor\rfloor$ below v'.
- take one ladder step. Because HEIGHT(v') $> l/2$, we know that the ladder of v' extends at least $l/2$ above v' (or it includes the root), so we can get to the correct ancestor right away.

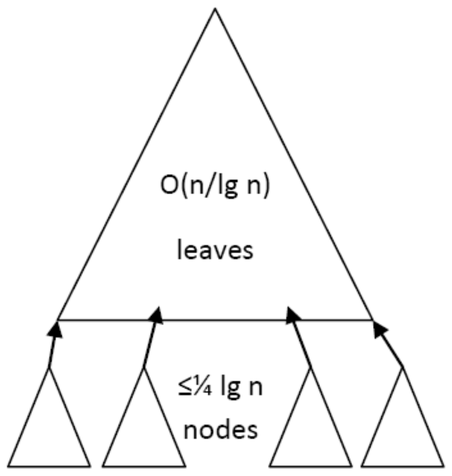

Figure 6: We separate bottom trees of size at most $\frac{1}{4}\lg n$ from the top tree.

This strategy gives constant-time queries, but $O(n \lg n)$ space for the jump pointers.

5.6 Jump pointer tuning

We want to store jump pointers only on leaves to reduce space. To accommodate that, we can store an arbitrary *leaf descendant* of every non-leaf node. The depth-d ancestor of V is the same as the depth-d ancestor of its leaf descendant, so we can start queries at leaves. The space usage is $O(n + L \lg n)$ where L is the number of leaves.

5.7 ART decomposition

We want to limit the number of leaves in a tree to $O(n/\lg n)$ so we can use the previous solution to obtain a linear space bound. To do that, we separate the *bottom* trees of size at most $\frac{1}{4}\lg n$ from the *top* tree as shown on the Figure 6. Since the total number of distinct trees on $\frac{1}{4}\lg n$ nodes is a Catalan number $C_{\frac{1}{4}\lg n} \leq 4^{\frac{1}{4}\lg n} = \sqrt{n}$, we can store a lookup table for every possible bottom tree in $O(\sqrt{n}\lg^2 n)$ space. To perform a query, we start from a leaf and look in a bottom tree. If we do not find the requested ancestor there, we get the parent of the root of the bottom tree, and perform the query on the top tree as in the previous solution. The query time is still constant but the space is now $O(n)$.

5.8 Weighted level ancestor

A more general version of the problem involves weights on edges. For the data structure mentioned in the wild card searches section, we need ancestor queries in a compressed trie to reduce $|P|$ bound to $\lg\lg|T|$. In a trie an edge may represent more than one letter, so we put weights on the edges representing the length of the string. Since the total weight along any path is still bounded by n, our level ancestor solutions above can be modified to the weighted case.

References

[1] Richard Cole, Tsvi Kopelowitz, Moshe Lewenstein. *Suffix Trays and Suffix Trists: Structures for Faster Text Indexing*. ICALP 2006: 358-369

[2] Stephen Alstrup, Jacob Holm. *Improved Algorithms for Finding Level Ancestors in Dynamic Trees*. ICALP 2000: 73-84

[3] Michael A. Bender, Martin Farach-Colton. *The Level Ancestor Problem simplified*. Theor. Comput. Sci. 321(1): 5-12 (2004)

[4] Omer Berkman, Uzi Vishkin. *Finding Level-Ancestors in Trees*. J. Comput. Syst. Sci. 48(2): 214-230 (1994)

[5] Richard Cole, Lee-Ad Gottlieb, Moshe Lewenstein. *Dictionary matching and indexing with errors and don't cares*. STOC 2004: 91-100

[6] Paul F. Dietz. *Finding Level-Ancestors in Dynamic Trees*. WADS 1991: 32-40