CS 194-1 Algorithms for Computational Biology **Spring 2012**

Lecture 3: September 4

Lecturer: Nir Yosef *Scribe: Rohin Shah*

3.1 Motivation

Last time, we saw that given a text T and a pattern P, we could solve the exact string matching problem and find all occurrences of P in T in linear time, or $O(|T|+|P|)$, using the Z-algorithm on the string $S = P\$T$, where \$ represents a termination character outside the alphabet of either input string. Furthermore, we found that this termination character simplified our space complexity to $O(|P|) \ll O(|T|)$ (for long T, as in bioinformatics applications).

Unfortunately, when we consider problems such as matching d patterns, $P_1, \ldots, P_d$, to a text T, the Z-algorithm is no longer enough. In the case of short reads against a human genome, wherein both $|T|$ and d are in general $O(10^9)$ and $O(10^8)$, respectively, the Z-algorithm would run as $O(d|T| + \sum_{i=1}^{d} |P_i|)$, which is grossly impractical.

If we could remove the factor of d, the algorithm would run in time linear with the size of the inputs. We also note that the matching problem compounds in the case of approximate matching, which is relevant because sequencing is error prone ($\sim 1/100$ error rate). Maybe if we employed a better data structure, we could solve these problems. In any case, it would be useful to have a preprocessed structure of the text T that could exist independent of the pattern strings P_i, which might not be known ahead of time.

Thus, we move to *suffix trees.*

3.2 Suffix Trees

Before we can define a suffix tree, we must define a suffix, which requires a definition for a substring and subsequence of a string. The intuitive difference between a subsequence and a substring is shown in Figure 3.1. The definition of a suffix follows.

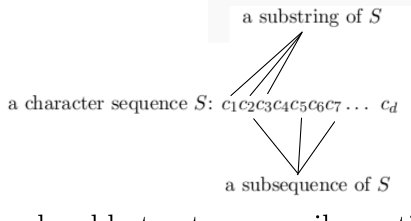
a substring of S
a character sequence S: $c_1c_2c_3c_4c_5c_6c_7 \ldots\ c_d$
a subsequence of S

Figure 3.1: A subsequence is an ordered but not necessarily contiguous set of characters from S. A substring is a contiguous subsequence of S. Here $c_1c_2c_3$ is a substring and $c_2c_5c_7$ is a subsequence.

Definition 3.1 (suffix). *A suffix $S[i..m]$ of a string S of length m is a substring of S that begins at position i of S and ends at the last position of S.*

Next we describe the construction of a suffix tree for the string $S = banana\$$, where \$ is a termination character. A diagram of the process is shown in Figure 3.2.

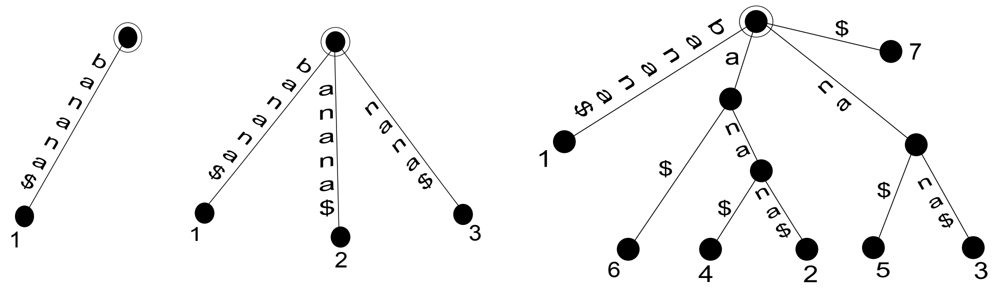

Figure 3.2: A naive approach for constructing a suffix tree of **banana$**

The string $S = banana\$$ has the following suffixes:

$S[1..7] = banana\$$
$S[2..7] = anana\$$
$S[3..7] = nana\$$
$S[4..7] = ana\$$
$S[5..7] = na\$$
$S[6..7] = a\$$
$S[7..7] = \$$

We define a root, as circled above, and define an edge starting from the root with edge-label $S[1..7] = banana\$$. This edge terminates with a leaf node numbered as the starting position of its edge's suffix, in this case the leaf labeled 1.

Then we take the first character of the next suffix, $S[2..7] = anana\$$, and compare it with the first character on the only edge out of the root. Because b is not equal to a, $S[2..7]$ defines the edge-label of a new edge stemming off the root, terminating in a leaf labeled 2.

We continue this way until we reach a suffix, in this case $S[4..7] = ana\$$, whose first character matches the first character of one of the edge-labels from the root. In this case, that edge is labeled $S[2..7] = anana\$$, but we notice that walking down this edge, $ana\$$ and $anana\$$ diverge at the \$ position on the smaller suffix. Thus at this point of divergence we branch off with the offending character \$, which labels an edge to a new leaf, 4.

In this way we naively build our suffix tree. Note several facts. The number of leaves generated in this fashion is equal to the string length. Also, each interior vertex (including the root) must necessarily have more than one child by our construction. Finally, each edge has been labeled by a nonempty string.

These latter two conditions are important but do not hold for all strings. Consider, for example, $S = aaba$; if one tries to build a suffix tree out of S, then on the path labeled by $S[4..4] = a$, the edge adjacent to the leaf will always be labeled with the empty string. More generally, any string that has a suffix, which appears to be a prefix of some longer substring $S[i \dots m]$, will not have a suffix tree. However, this problem can be fixed easily by appending a special character (e.g., \$), which we shall call the terminator, not in the alphabet of S, at end of the string S.

We define a suffix tree formally below.

Definition 3.2 (suffix trees)**.** *Given a string S with $|S| = m$, a suffix tree $\mathcal{T}(S)$ of the string S satisfies the following conditions:*

1. *it is a rooted directed tree with exactly m leaves labeled from 1 to m;*
2. *except for the root, every interior vertex has at least two children;*

3. *each edge is labeled by a nonempty substring of S (written in the direction of the edge);*
4. *no two edges out of a vertex can have edge labels that begin with the same character;*
5. *for each $i = 1, 2, \ldots, m$, the concatenation of the edge labels on the path from the root to the leaf i spells out $S[i \ldots m]$.*

As noted above, any string $S\$$ ending with a *termination character* is then guaranteed to have a suffix tree. Furthermore, this termination character guarantees that the root also has two children.

3.2.1 Algorithms for Constructing a Suffix Tree

Convince yourself that linear-time suffix tree construction is not trivial. In fact, the trivial solution was sketched above, and is described formally below.

Algorithm 1
Input: A string S of length m.
Output: A suffix tree $\mathcal{T}(S\$)$, where \$ is a special character not in the alphabet of S.

1. Add to the root an edge labeled by the string $S\$$ and its vertex labeled by **1**.
2. For $i = 2, 3, \ldots, m$, successively enter the suffix $S[i \ldots m]\$$ into the current tree as follows. Compare the first character of the suffix $S[i..m]$ in question against the first label character of each edge outgoing from the root.
   (a) If no match is found, create a new leaf labeled i, with edge-label given by $S[i..m]$. Move on to the next suffix.
   (b) If the characters match, walk down that edge, comparing edge and suffix characters until either a node is discovered or a mismatch is noted.
      i. If a mismatch is noted, branch off the edge from the position immediately preceding, creating a leaf node with edge-label given by the remaining characters in the suffix. Label the newly created leaf by i. This step may require adding a vertex on an existing edge which will break that edge into two.
      ii. If a node is discovered, compare the current character with the first character of each of the edge-labels outgoing, and continue the traversal accordingly, as in 2 above.

Remark. Given that the size of the alphabet is finite, this naive algorithm takes $O(m^2)$ time to build a suffix tree. For clarity, consider the worst case input: a string of m of the same character.

Luckily, there are well-known linear time algorithms for constructing a suffix tree that make it possible for us to solve the *exact matching problem* in linear time (in the length of the text).

Now, we present a list of authors who developeded $O(m)$ time algorithms for creating a suffix tree of a string S of length m. We will not go into details of any algorithm but references are provided for those interested.

1. Peter Weiner: *Linear Pattern Matching Algorithms.* Proc. 14th FOCS pp. 1-11; 1973. This is the first time when the notion of suffix trees (or position trees) is introduced. Although its time complexity is linear in m, its space complexity is exponential in m.
2. Edward M. McCreight: *A Space-Economical Suffix Tree Construction Algorithm.* JACM 23(2): 262-272 (1976). This algorithm is linear in time but quadratic in space complexity.
3. Esko Ukkonen: *On-Line Construction of Suffix Trees.* Algorithmica 14(3): 249-260 (1995). This algorithm runs in linear time and uses linear space and also it is an online algorithm,

which means, it can process new input without having to use the already-processed input from the start.

4. Martin Farach-Colton: *Optimal Suffix Tree Construction with Large Alphabets.* In Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, pp. 137-143; 1997.
   Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. *On the Sorting-complexity of Suffix Tree Construction.* JACM, 47(6):987-1011, 2000.
   This algorithm employs the divide-and-conquer technique, runs in linear time and uses linear space.

3.3 Applications of Suffix Trees

3.3.1 Exact Pattern Matching

In the previous section, we introduced the suffix tree and claimed that it could be used to design a linear time algorithm that solves the exact pattern matching problem. Here we will flesh out this solution, assuming the length of the alphabet is constant, and thus that the outdegree of any interior node in a suffix tree is at most a constant.

Lemma 3.3. *For a given pattern P and a text T, the following are equivalent:*

1. *P occurs in T.*
2. *P is a prefix of some suffix α of T.*
3. *P labels a* unique *path starting from the root in the suffix tree of T.*

Lemma 3.3 is shown in Figure 3.3, for $T = banana\$$ and $P = ana$.

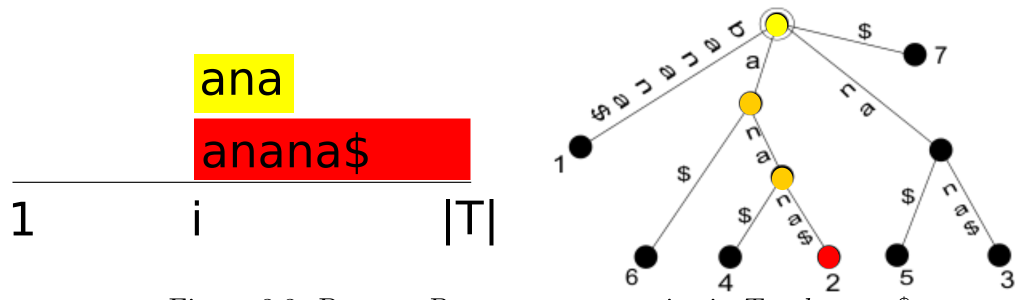

Figure 3.3: Pattern $P = ana$ occurs twice in $T = banana\$$.

Note that in Lemma 3.3, the path is unique because of Definition 3.2.4. For contradition, suppose we have a suffix tree as defined in Definition 3.2. Suppose two suffixes both containing P as a prefix branch down the suffix tree at some point before their path edges trace out P. But then at that branch character, both have the same value, therefore the tree generated is not a suffix tree. Hence we have a contradiction, and Lemma 3.3 holds.

From this lemma, it follows that the number of occurrences of P in T is exactly the number of leaves in the subtree consisting of all the descendants of v, where v is the unique vertex such that the label of the path from the root to v is P. In the case above, v would be the lowest yellow vertex. Because there are two leaves in the subtree rooted at v, (leaves 2 and 4), we know the substring $P = ana$ appears twice in $T = banana\$$.

Note that we can also recover where P occurs in T by recovering the labels of those leaves that are descendants of v.

Thus, we might consider the following algorithm as a candidate for linear time exact string matching.
Algorithm 2
Input: A pattern P and a text T.
Output: All occurrences of P in T.

1. Construct a suffix tree of the string $T\$$.
2. Find a unique branch (adjacent to the root) that has a prefix matched with P. If there is no such branch, then P does not occur in T. If there is one, proceed to the next step.
3. Given that P exists as a prefix of some edge-label in the suffix tree of T, in tracing down the suffix tree using the character values of P, at some point P will be exhausted. Trace the edge at which this occurs to the next downstream node. The subtree defined by this node is v. Use depth-first search to enumerate all the leaves of the subtree v. These represent all occurrences of P in T.

As for the time analysis, we assume that the set of the alphabet for T constant. For the first step, we learn from the previous section that building a suffix tree will take only $O(|T|)$ time. The second step takes $O(|P|)$ in total, as there can be at most a constant number of branches adjacent to the root because the size of the alphabet is constant. Then the last step, which uses depth-first search through a subtree, runs in $O(e)$ time where e is the number of edges of the subtree. But since each non-leaf vertex has at least two outdegree by Definition 3.2.2, e is linear in K – the number of leaves in the subtree by Lemma 3.5 in Appendix. Note that the number of leaves is also equal to the number of occurrences of P in T. Therefore, the running time of Algorithm 2 is $O(|T|+|P|+K)$.

Remark. This algorithm can be extended easily when there are multiple of patterns, say $P_1 \dots, P_d$. Once we have constructed a suffix tree of T; we can use it to find occurrences of P_i for every i. Therefore, the running time will be $O(|T|+\sum_{i=1}^{d}(|P_i|+K_i))$, where K_i is the number of occurrences of P_i in T. In truth, if we did not care about space complexity the depth-first search could be swapped in place of a constant time lookup data structure.

3.3.2 Longest Common Substring

In this section, we discuss another use of suffix trees in solving the longest common substring problem. This LCS problem in fact occurs in some biological contexts, for example, finding conserved regions in the genome and identifying DNA contamination. The details of DNA contamination can be found in Gusfield's book. We will present a linear time algorithm for solving the LCS problem. However, we need an extended version of a suffix tree, called *generalized suffix trees (GST)*, which is a suffix tree for two or more strings with distinct terminating characters. An example of a GST of $S_1 = \mathbf{aba\$}, S_2 = \mathbf{ab\#}$ is displayed in Figure 3.4.

It should be noted that we label each leaf with (t, p), where t is the string type and p is the start position of the corresponding substring.

Also note that there is an algorithm that builds a GST of $S_1, \dots, S_k$ in linear time $O(\sum_{i=1}^{k}|S_i|)$. If we are not concerned with space complexity, a naive approach might be to concatenate $S_1, \dots, S_k$ with different interspaced terminating characters, building a suffix tree of the resulting string in

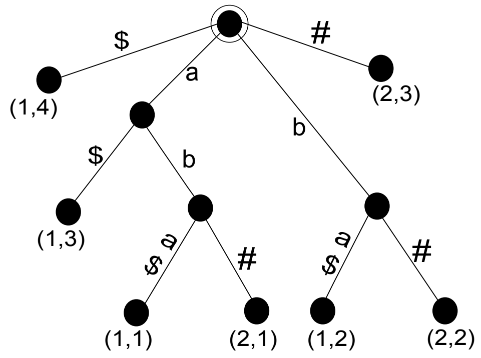

Figure 3.4: A generalized suffix tree of $S_1 =$ **aba$** and $S_2 =$ **ab#**.

time $O(\sum_{i=1}^{k} |S_i|)$. Then a simple depth first search through the tree would suffice to allow us to cut off all characters past the terminating character in any leaf, generating the desired generalized suffix tree. This method works only because the terminating characters are unique, so it is not possible to branch past one in an edge label.

Next we present an algorithm that solves the LCS problem for two strings.

Algorithm 3
Input: Two strings S_1 and S_2.
Output: A longest commom substring of S_1 and S_2.

1. Build a GST of $S_1\$$ and $S_2\#$.
2. Use 3 colors (white, black, gray) to color all the vertices of the tree, from the bottom up (i.e. leaves $\rightarrow$ root), by the following criterion.
    - For each leaf labeled with (t, p), if $t = 1$, then color the leaf with color white; or if $t = 2$, color the leaf with black.
    - For the next level of vertices, we color each vertex based on the colors of its children. If all of its children are colored white, then color the parent vertex white; or if all of its children are colored black, then color it with black. If there is some whitened child and some blackened one, then color the vertex with gray. If there is a child with color gray, color it with gray as well.
    - Repeat the above step until every vertex is colored.
3. Find the longest path (in terms of the length of its label) from the root to a gray vertex. The label of the path is a longest common substring.

Remark. It is easy to see that this algorithm works out because, for any vertex that is colored gray, there must be a path from the root to a leaf $(1, p_1)$ via this vertex and another path from the root to a leaf $(2, p_2)$ via this same vertex. The former path's label indeed corresponds to a substring of S_1 and the latter corresponds to a substring of S_2. Therefore, the label of the path from the root to that common vertex is a common substring of S_1 and S_2. The deepest (in terms of path label) such common vertex is the longest common substring of the two input strings.

For the time analysis, we take it for granted that we can build a GST of S_1, S_2 in linear time $O(|S_1| + |S_2|)$. The second step involves coloring all the vertices of the tree (by depth-first search), so it simply takes linear time in #vertices + #edges, hence # leaves by Lemma 3.5 in the Appendix.

But the number of leaves equals $|S_1|+|S_2|+2$. This second step thus takes linear time $O(|S_1|+|S_2|)$ as well.

In the last step, we can perform breadth-first search which takes linear time in #edges + #vertices, hence $O(|S_1| + |S_2|)$ for the same reason as above. Note that it is also possible to traverse the tree bottom-up. For the interested reader, a linear time method that has constant lookup is also possible. See reference [3] below.

In sum, this algorithm runs in linear time in the total sum of the lengths of the two strings.

3.3.3 Common Substrings of K strings

Now, we want to find long substrings that are common to many strings. Specifically, given an input set of K strings $S = \{S_1, S_2, \cdots S_k\}$, we want to find the longest substring that is present in at least k of the strings in S, for all k such that $2 \leq k \leq K$.

We define $l(k)$ to be the length of the longest substring that is common to k of the strings in S. For example, if the input set of strings is $S = \{bread, sabres, macabre, breakfast, barefoot\}$, the output of the algorithm would be:

k	$l(k)$	`substring`
2	4	`brea`
3	3	`bre`
4	3	`bre`
5	2	`re`

Note for $k = 2$, `abre` would also be a valid substring.

Using a suffix tree allows for an $O(Kn)$ solution, where n is the total length (that is, $n = \sum_i |S_i|$). Amazingly, there is also an $O(n)$ solution, but that uses the constant-time lowest common ancestor trick, and so it will not be explained here.

The $O(Kn)$ algorithm takes inspiration from the solution to the longest common substring for 2 strings. As before, we create a generalized suffix tree for the K strings, which again we assume can be done in $O(n)$ time (recall that $n = \sum_i |S_i|$).

In the $K = 2$ case, we then proceeded to color each of the vertices black, white or gray. The purpose of this was to identify nodes which contained leaves from both string 1 and string 2. Similarly, now we want to know for each node how many *distinct* strings it contains leaves for. We call these C-values:

Definition 3.4 (C-values)**.** *For every internal node v of a generalized suffix tree T of strings $S_1, S-2, \cdots S_K$, define $C(v)$ to be the number of distinct strings whose suffixes appear at the leaves in the subtree of v.*

The C-values can easily be computed through a depth-first search that propagates information from the bottom up. However, each node will need to know exactly which of the strings from $S_1 \cdots S_k$ it contains in its leaves, which takes $O(K)$ space and $O(K)$ time to update. Since there are $O(n)$ nodes in total, this takes $O(Kn)$ time.

Once the C-values are known, we can simply traverse the tree to compute the $l(k)$ values. (At a node v, compare the string-depth of the label to $l(C(v))$ and if it is larger set $l(C(v))$ to be the label until v.) However, this is not exactly correct - at this point $l(k)$ will be the length of the longest substring that occurs *exactly* k times, whereas what we want is the length of the longest substring that occurs *at least* k times. This can be easily fixed by making one more pass through the l values from $l(K)$ to $l(2)$. For each i, if $l(i + 1) > l(i)$, we set $l(i) = l(i + 1)$. This then yields the correct values.

The running time for building the generalized suffix tree is $O(n)$. As argued above, computing the C-values takes $O(Kn)$ time. The traversal through the tree computing initial $l(k)$ values takes $O(n)$ time, since it is just a traversal of the tree which does a constant amount of work at each node. The final pass through the $l(k)$ values to fix them takes $O(K)$ time. So, in total, the algorithm takes $O(n + Kn + n + k) = O(Kn)$ time.

3.4 Appendix and References

Lemma 3.5. *Let T be a connected tree with the property that every interior vertex (except for the root) has at least two outdegree. Then the number of edges is linear in the number of leaves.*

Proof. Let L be the number of leaves, NL be the number of non-leaf vertices, and E is the number of edges. So $E = L + NL - 1$. Because every non-leaf vertex (except for root) has at least two outdegree and E is equal to the total sum of the outdegrees, we have $E \geq 2(NL - 1) + 1$. From this inequality, we get $L + NL - 1 \geq 2NL - 1 \Longrightarrow L \geq NL$. Therefore, $E \leq 2L - 1$. $\square$

References

1. J. Ian Munro: *Succint Data Structures (and Text Search) CS 840 Lecture Note - Unit 3*, University of Waterloo, 2010, `http://www.cs.uwaterloo.ca/~imunro/cs840/Unit_3.pdf`.
2. Dan Gusfield: *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
3. Daniel P.: *Range Minimum Query and Lowest Common Ancestor*, TopCoder, `http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor`