1 Suffix Trees

Consider a string T of length t (long). Our goal is to preprocess T and construct a data structure, to allow various kinds of queries on T to be done efficiently. The most basic example of which is simply this: given a pattern P of length p, find all occurrences of P in the text T. What is the performance we aiming for?

- The time to find all occurrences of pattern P in T should be $O(p+k)$ where k is the number of occurrences of P in T.
- Moreover, ideally we would require $O(t)$ time to do the preprocessing, and $O(t)$ space to store the data structure.

Suffix trees are a solution to this problem, with all these ideal properties.[1] They can be used to solve many other problems as well. In this lecture, we'll consider the alphabet size to be $|\Sigma| = O(1)$.

1.1 Tries

The first piece of the puzzle is a *trie*, a data structure for storing a set of strings. This is a tree, where each edge of the tree is labeled with a character of the alphabet. Each node then implicitly represents a certain string of characters. Specifically a node v represents the string of letters on the edges we follow to get from the root to v. (The root represents the empty string.) Each node has a bit in it that indicates whether the path from the root to this node is a member of the set—if the bit is set, we say the node is marked.

Since our alphabet is small, we can use an array of pointers at each node to point at the subtrees of it. So to determine if a pattern P occurs in our set we simply traverse down from the root of the tree one character at a time until we either (1) walk off the bottom of the tree, in which case P does not occur, or (2) we stop at some node v. We now know that P is a prefix of some string in our set. And if v is marked, then P is in our set, otherwise it is not.

This search process takes $O(p)$ time because each step simply looks up the next character of P in an array of child pointers from the current node. (We used that $|\Sigma| = O(1)$ here.)

1.2 Returning to Suffix Trees

Our first attempt to build a data structure that solves this problem is to build a trie which stores all the strings which are suffixes of the given text T. It's going to be useful to avoid having one suffix match the beginning of another suffix. So in order to avoid this we will affix a special character denoted "$" to the end of the text T, which occurs nowhere else in T. (This character is lexicographically less than any other character.)

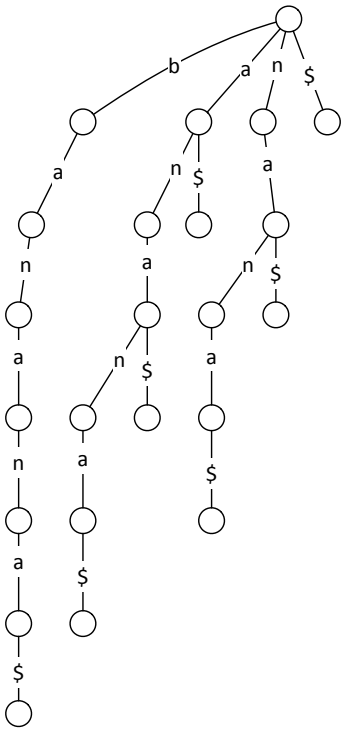
For example if the text were $T =$ `banana$`, the suffixes of T are then

```
banana$
```

[1] Suffix trees were invented by Peter Wiener.

```
anana$
nana$
ana$
na$
a$
$
```

(Since we appended the $ sign to the end, we're not including the empty prefix here.) And the trie of these suffixes as described above would be the one below:



Suppose we have constructed the trie containing all these suffixes of T and we store at each node the count of the number of leaves in the subtree rooted at that node. Now given a pattern P, we can count the number of occurrences of P in T in $O(|P|)$ time: we just walk down the trie and when we run out of P we look at the count of the node v we're sitting on. It's our answer.

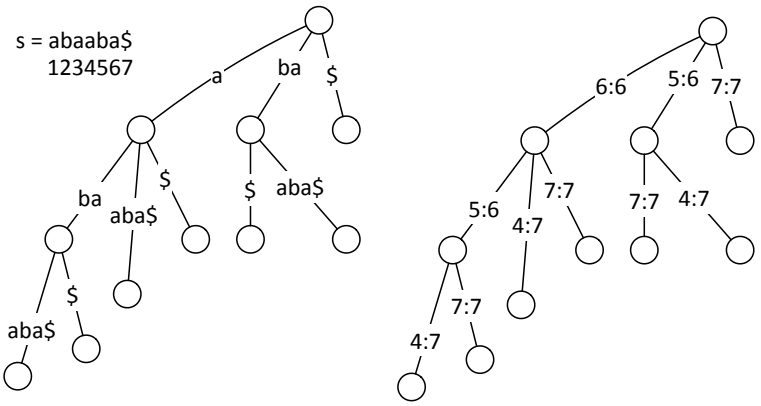
But there are a number of problems with this solution. First of all, the space to store this tree (as described) could be as large as $\Theta(t^2)$. Also, it's unsatisfactory in that it does not tell us where in s these patterns occur. Finally, it will also take too long to build it.

Shrinking the tree to $O(t)$ space. The first two issues are easy to handle: we can create a "compressed" tree of size only $O(t)$. Since no string occurs as a prefix of any other, we can divide the nodes of our trie into *internal* and *leaf* nodes.

The leaf nodes have no children, and represent a suffix of T. So we can have each leaf node point to the place in T where the given suffix begins.

Moreover, if there is a long path in the trie with branching factor 1 at each node on that path. We can compress that path into a single edge that represents the entire path. Moreover, the string of

characters corresponding to that path must occur in T, so we can represent it implicitly by a pair of pointers into the string T. So an edge is now labeled with a pair of indices into T instead of just a single character (or a string). Here's an example (with the substrings labeling the edges on the left, and the start-end pairs labeling them on the right):



This representation uses $O(t)$ space. (We count pointers as $O(1)$ space.) Why? Each internal node now has degree at least 2, hence the total number of nodes in the tree is at most twice the number of leaves. (Exercise: prove this!) But each leaf corresponds to some suffix of T, and there are t suffixes.

Building the tree — preview. What about the time to build the data structure? Let's first look at the naïve construction, by adding suffixes into it one at a time. To add a new suffix, we walk down the current tree until we come to a place where the path leads off of the current tree. (This must occur because the suffix is not already in the tree.) This could happen in the middle of an edge, or at an already existing node. In the former case, we split the edge in two and add a new node with a branching factor of 2 in the middle of it. In the latter case we simply add a new edge from an already existing node. In either case the process terminates with a tree containing $O(t)$ nodes, and the running time of this naive construction algorithm is $O(t^2)$. We will see how to do a better than this in the next lecture. In fact, it is possible to build a suffix tree on a string of length t in time $O(t)$.

2 Applications of Suffix Trees

We've seen how suffix trees can do exact search in time proportional to query string, once the tree is built. There are many other applications of suffix trees to practical problems on strings. Gusfield discusses many of these in his book. We'll just mention just a few here.

2.1 Simple Queries

Suffix trees make it easy to answer common (and less common) kinds of queries about strings. For example: it's easy to:

- Check whether P is a suffix of T: follow the path for q starting from the root and check whether you end at a leaf.
- Count the number of occurrences of P in T: follow the path for q; the number of leaves under the node you end up at is the number of occurrences of P. If you are going to answer this

kind of query a lot, you can store the number of leaves under each node in the nodes.

- Find the lexicographically (alphabetically) first suffix: start at the root, repeatedly follow the edge labeled with the lexicographically (alphabetically) smallest letter.
- Find the longest repeat in T. That is, find the longest string r such r occurs at least twice in T: Find the deepest node that has ≥ 2 leaves under it.

2.2 Longest Common Substring of Two Strings

Given two strings S and T, what is the longest substring that occurs in both of them? For example if $S =$ `boogie` and $T =$ `ogre` then the answer is `og`. How can one compute this efficiently? The answer is to use suffix trees. Here's how.

Construct a new string $U = S\%T$. That is, concatenate S and T together with an intervening special character that occurs nowhere else (indicated here by "%"). Let n be the sum of the lengths of the two strings. Now construct the suffix tree for U. Every leaf of the suffix tree represents a suffix that begins in S or in T. Mark every internal node with two bits: one that indicates if this subtree contains a substring of S, and another for T. These bits can be computed by depth first search in linear time. Now take the deepest node in the suffix tree (in the sense of the longest string in the suffix tree) that has both marks. This tells you the the longest common substring.

This is a very elegant solution to a natural problem. Before suffix trees, an algorithm of Karp, Miller, and Rosenberg gave an $O(n \log n)$ time solution, and Knuth had even conjectured a lower bound of $\Omega(n \log n)$.

2.3 Searching for Matching Strings in a Database

The above idea can be extended to more than 2 strings. This is called a *generalized suffix tree*. To represent a set of strings $D = \{S_1, \ldots, S_m\}$, you concatenate the strings together with a unique character and label the leaves with the index of the string in which that suffix begins. To find which strings in D contain a query string q you follow the path for q and report the indices that occur in the subtree under the node at which you stopped.

2.4 Longest Common Extension

Here's another problem were we can process T and then must answer queries about T on the fly.

Problem 1 (Longest common extension) *We are given strings S and T. In the future, many pairs of indices (i, j) will be provided as queries, where i is a position in S and j is a position in T. We want to quickly find: the longest substring of S starting at i that matches a substring of T starting at j.*

In pictures:

To do this, we need to introduce the concept of a *lowest common ancestor* (LCA) of two nodes in a tree.

Definition 2 *The* lowest common ancestor *(LCA) of two tree nodes u and v is the deepest node that is on the path from the root to u and the path from the root to v*

It's easy to pre-process the tree so that $LCA(u, v)$ can be found in $O(1)$ time if we are allowed to use $O((\text{size of the tree})^2)$ space: just create a 2D array A so that entry $A[u, v]$ gives the LCA of u and v. I turns out that you can do this in *linear* space, which is a nice result by Gabow and Tarjan (1983). We won't have time to go into how to do that, however. But now we can see how to use LCA to answer LCE queries:

Preprocessing:

1. Build a suffix tree R for $S\%T$. (Takes time $O(|S| + |T|)$)
2. Pre-process R so that lowest common ancestors (LCA) can be found in constant time (see above)
3. Build an array mapping indices i and j to the leaf nodes representing those suffixes.

When we see query (i, j):

1. Find the leaf nodes for i and j. (Takes $O(1)$ time using the array.)
2. Return the string represented by the $LCA(i, j)$ (Takes $O(1)$ time using the LCA pre-processing.)

2.5 Finding Palindromes

Suppose you want to find all the maximal even palindromes in T. An *even palindrome* is a string $\alpha\alpha^R$, where α^R means reversing the string. A maximal even palindrome is a palindrome that can't be extended longer in T. In pictures:

The trick here is to notice that when we reverse T, the character just to the right of the center of a palindrome at position i moves to position $n - i - 1$ (starting indexing at 0) and the palindrome also flips around.

To find all maximal even palindromes in T:

1. Process T and T^R so that LCE queries can be solved in constant time (see previous section).
2. For every position i in T: Compute $LCE(i, n - i)$.

This takes $O(|S|)$ time to pre-process and execute the LCEs, giving a *linear* time algorithm to find all the maximal palindromes.

2.6 Finding All k-Mismatch Occurrences of a Pattern

Say two strings u and v of the same length have k *mismatches* if they are equal in all but k positions.

Problem 3 *Given a long string T and a shorter string P and an integer k, find all the positions in T that are the start of a string that has k mismatches with P.*

This can be done in $O(k|T|)$ time using LCE. Let's see how we check whether index i in T is the start of a k mismatch of P:

```
function checkMismatch(i, P, k):
    j = 0       // current position in P
    c = 0       // number of mismatches found so far
    repeat until c > k:
        j = j + LCE(i,j) + 1 // jump past next mismatch in P and T
        i = i + LCE(i,j) + 1
        if j >= |P|+1: return true  // if we ran out of pattern
        c = c + 1
    end
    return false // if we ran out of mismatches
```

The above check takes time at most $O(k)$ since each LCE query takes $O(1)$ time.

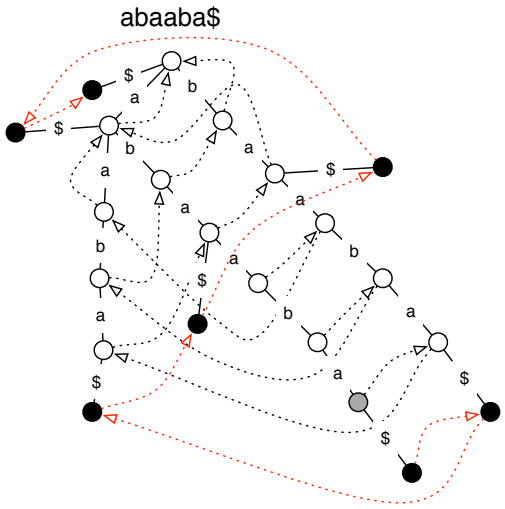
2.7 Longest Common Substring of T and Online Queries q

This problem is different than the one in section 2.2 because we aren't allowed to pre-process q. T is some large database of text, which we can pre-process, but then queries come later on, and we need to answer them efficiently by finding the longest common substring between T and each query q.

To solve this problem, we need to introduce the concept of a *suffix link*:

Definition 4 *A* suffix link *is an extra pointer leaving each node in a suffix tree. If a node u represents string $x\alpha$ (where x is a character and α is a string) then u's suffix link connects from u to the node representing the string α.*

Every node has a suffix link. Why? Every node represents the prefix of some suffix. Suppose node u represents the prefix of suffix i of length m. Then u's suffix link should point to the node on the path representing the prefix of length $m-1$ of suffix $i+1$. A trie with all the suffix links shown as dashed arrows is given below:



If node representing string $x\alpha$ exists in our tree (even after compression), then a node representing α must exist. The reason for this is since the node $x\alpha$ exists, it has two children y and z. That means both the strings αy and αz must exist in the string. So the node representing by α must have 2 children too.

Using suffix links we can get an $O(|q|)$-time algorithm to find the longest common substring between T and q:

1. Walk down the tree, following q
2. If you hit a dead-end at node u,
   (a) save the depth of node u if it is deeper than the deepest dead end you found so far.
   (b) follow the suffix link out of u.
   (c) Go to step 1 (starting from your current place in q)
3. When you exhaust q, return the string represented by the deepest node you visited.

The first idea here is the the longest common substring starts at some suffix — we just don't know which suffix, so we try them all starting with suffix 0. The second idea here is that following a suffix link chops off the first character of q taking you to the place in the tree that you would have been at if the query had started with the 2nd character of q.