# Lecture 5: Suffix Trees

## Longest Common Substring Problem

Given a text T = "GGAGCTTAGAACT" and a string P = "ATTCGCTTAGCCTA", how do we find the longest common substring between them?

Here the longest common substring would be "GCTTAG". However none of the linear time string search algorithms for exact matching that we have learnt so far allows us to do this in linear time. We could try chopping off the characters from the beginning, and matching the remaining pattern repeatedly. That would be an inefficient method. Suffix trees allow us to do such searches efficiently. However just a naively constructed suffix tree would not give us an efficient algorithm.

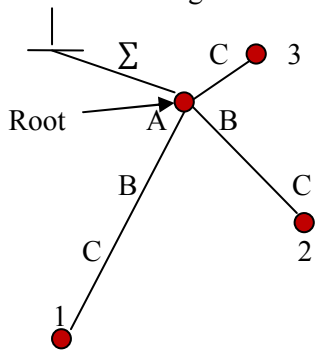
## What is a suffix tree?

Suffix tree, as the name suggests is a tree in which every suffix of a string S is represented. More formally defined, suffix tree is an automaton which accepts every suffix of a string. Definition of suffix trees, as given in section 5.2 of Gusfield:

"A suffix tree $T$ for an $m$-character string $S$ is a rooted directed tree with exactly $m$ leaves numbered $1\ to\ m$. Each internal node, other than the root, has at least two children and each edge is labeled with a non-empty substring of $S$. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf $i$, the concatenation of the edge-labels on the path from the root to leaf $i$ exactly spells out the suffix of $S$ that starts at position $i$. That is, it spells out $S[i..m]$. "

It is evident from the above definition that the strings are labeled on the edges, and we get a new node in the tree only when there is a split, that is, the label on the edge leading to the node is the prefix of more than one suffix of the string. Thus a suffix tree differs from a suffix trie which takes $O(|N|^2)$ space.

Example of suffix tree for the string "ABC"




1, 2 and 3 represent the ends of suffixes starting at positions 1, 2 and 3 respectively. These are the leaf nodes.

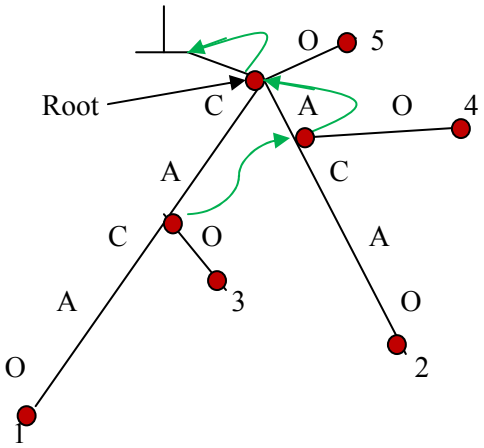
**Suffix links**

One important feature of suffix trees are suffix links. Suffix links are well defined for suffix trees. If there is a node $v$ in the tree with a label $c\alpha$, where $c$ is a character and $\alpha$ is a string (non-empty), then the suffix link of $v$ points to $s(v)$, which is a node with label $\alpha$. If $\alpha$ is empty, then the suffix link of $v$, i.e, $s(v)$ is the root. Suffix links exist for every internal (non-leaf) node of a suffix tree. This can be easily proved. (Refer Lemma 6.1.1, Corollary 6.1.1 and Corollary 6.1.2 of Gusfield).

Suffix links are similar to failure functions of Aho-Corasick algorithm. By following these links, one can jump from one suffix to another, each suffix starting exactly one character after the first character of its preceding suffix. Thus using suffix links it is possible to get a linear time algorithm for the previously stated problem: matching the longest common substring. It is because suffix links keep track of what or how much of the string has been matched so far. When we see a mismatch, we can jump along the suffix link to see if there is any match later in the text and the string. Not only that, this jumping trick also helps us to save computation time while constructing suffix trees. Thus suffix links are very important constructs of suffix trees.

Example of a suffix tree with suffix links:

Suffix tree for the string "CACAO". The green arrows represent the suffix links.




**Construction of Suffix trees: Ukkonen's Algorithm**

We start by creating the suffix tree for all prefixes of the string $S$. We first build the implicit suffix trees for all the prefixes $S[1..i]$, increasing the length of the prefix by one character per stage, till our prefix is equal to the string. At this stage, we convert the final stage implicit suffix tree to a true suffix tree. A true suffix tree would have exactly $m$ leaves, where $m$ is the length of the string, each leaf representing one of the $m$ suffixes. An implicit suffix tree on the other hand, may not have a leaf for each suffix; however it encodes all the suffixes of $S[1..i]$. Every suffix of $S[1..i]$ would be found on some path from the root of such an implicit suffix tree $T_i$.

At a high level, the method of construction of suffix trees is as follows:

If we have the implicit suffix tree $T_i$ for $S[1..i]$ in the $i^{th}$ stage, then in the $(i+1)^{th}$ stage we simply add the character $S(i+1)$ to every suffix in $T_i$ to get $T_{i+1}$. Continuing in this manner to $m$ stages, we get the

implicit suffix tree for the entire string, which we then convert to the actual suffix tree. Construction of suffix trees by Ukkonen's Algorithm is $O(m)$. Construction of the first stage ($i = 1$) is very easy. It is just a single edge from the root labeled by the first character of the string, and this of course ends in a leaf. However, what we have described so far for the rest of the construction, is not linear time, rather quite an expensive process. The naive running time of the described algorithm would be $O(m^3)$. There would be $m$ stages of construction of implicit suffix trees, and in each stage $i$, there would be $i$ suffixes. For each of the $i$ suffixes, we would have to travel the entire length of the suffix down from the root to the leaf to reach their end, where we would have to append $S(i+1)$. Also, we need to add a new edge for $S(i+1)$ from the root. The length of each such path, for the suffix $S[j..i]$ would be $O(i+1-j)$. Hence the time required would be $O(m^3)$.

When we follow each suffix branch to add the new character in the $(i+1)^{th}$ stage, the following three cases are possible:

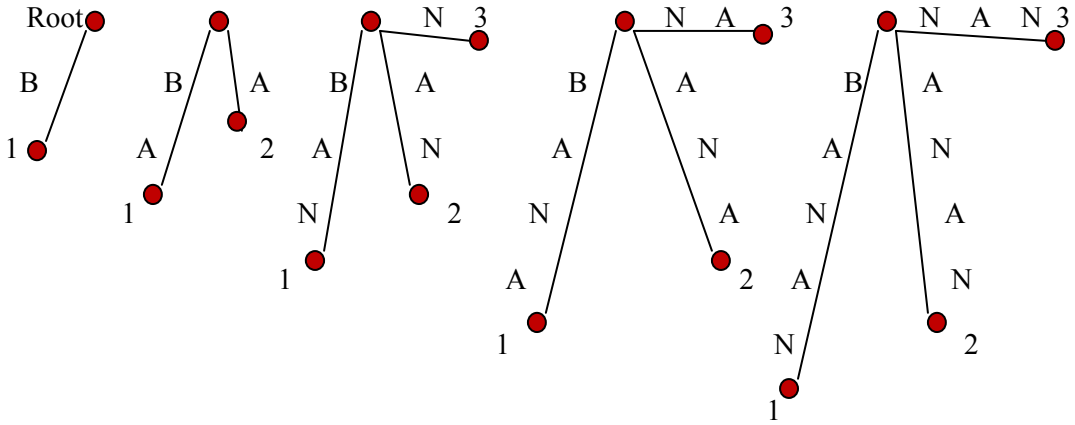
i) The current search path ends in a leaf. In this case, we simply add the extra character to the edge leading into the leaf node.

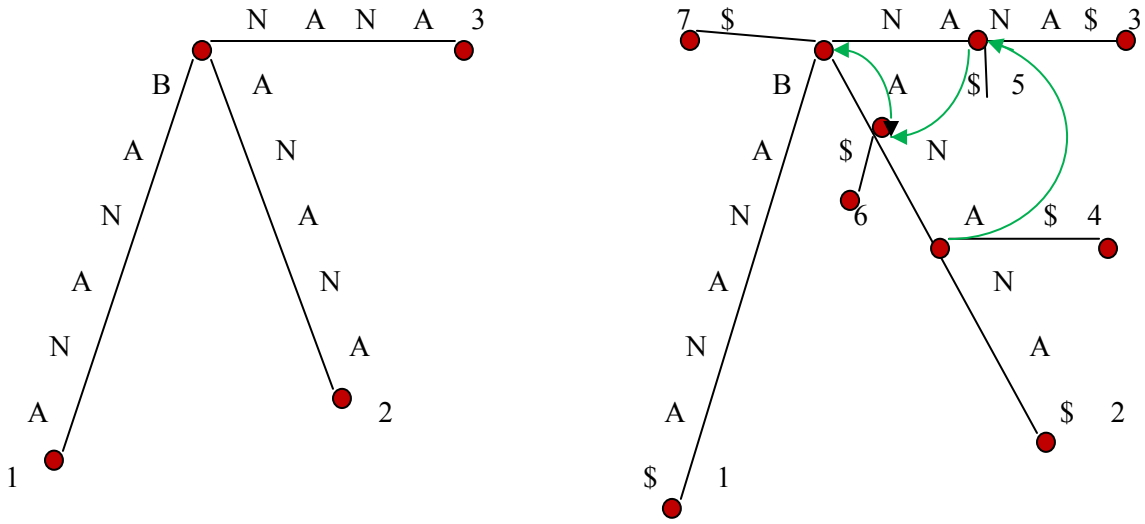
ii) The current search path ends in the middle of an edge and the following character is not equal to $S(i+1)$. In this case, create a new internal node at that point and add an edge leading out from the internal node into a new leaf node, labeled by $S(i+1)$.

iii) The current search path ends in the middle of an edge, and the following character is equal to $S(i+1)$. In this case, do nothing, since the suffix is already present in the implicit tree.

We had previously stated that suffix links exist for every internal node of a tree. So, what happens when case (ii) occurs and we create a new node $v$ for the suffix starting at the $j^{th}$ location of the string, at stage $i$? The suffix link node $s(v)$ for this node $v$ either already exists, or is created in the next step when we look at the suffix starting at position $(j+1)$ and the link $(v, s(v))$ is added in this step $(j+1)$. (Ref Corollary 6.1.1 in Gusfield) . Also, as stated by Dr. Pop in the suffix trees summary he provided, "the suffix links are simply updated by keeping track of the last node created by the algorithm and creating a link from it to the next node being created".

Example of construction of suffix trees for string "BANANA$"






There are a number of tricks that can be used to reduce the running time down to linear.

**Trick 1: Suffix Links**

Jump to successive suffixes via suffix links, as already described before. This is one of the prime tricks which helps us to achieve the linear time bound which Knuth had predicted to be impossible.

The basic idea is to reduce the amount of work done in traversing each path from the root to locate the end of the next suffix. For the first suffix ($j = 1$) in every stage $i$, we can keep a pointer to its leaf, so that we can quickly jump to the leaf and add the new character. For every other suffix ($j > 1$), we walk up at most one node up to a node $v$ with a suffix link $s(v)$ or to the root, and follow the suffix link to continue the walk again from $s(v)$. This saves the work that would otherwise be spent in moving from the root down to $s(v)$. This method works because the label on the edge leading to $v$ (from which we jumped) is $x\alpha$, where $\alpha$ is a string (possibly empty), and the suffix link leads to a node $s(v)$ with label $\alpha$ on the edge leading to it. Hence any suffix starting with $\alpha$ must be ending in the subtree containing $s(v)$.

**Trick 2: Skip / Count**

This is known as the **skip/count** trick. This helps us to find the place for adding the new character $S(i+1)$ in the $(i+1)^{th}$ stage very quickly.

Just reducing the work down from the root to $s(v)$ is not enough to reduce the runtime. By using the skip/count trick, we reduce the work done in walking down from $s(v)$ to the leaf to be equal to the number of nodes on the path, instead of the length of the path.

Let us assume a few simple implementation details: i) we know the number of characters on each edge, ii) we can find the character from any given position of $S$ in constant time.

Now, let the length of the path down from $s(v)$ be $n$. Since there cannot be more than one edge out of $s(v)$ that starts with the same character, the first character in the suffix will appear on only one edge out

of $s(v)$. If the number of characters on this edge is $p$, and $p < n$, then let $n = n - p$, and the next position on the string to be matched is $q = p + 1$. Now we again select the edge out of the current node which starts with character $S(q)$. If the label on this path $p < n$, then again $n = n - p$ and $q = q + p$. Continuing in this manner, if on some edge, $p > n$, then the suffix ends exactly $n$ characters down that edge. Thus we have reduced the work to be equal to the number of nodes in the path (instead of the length of the suffix on that path). This saves work because now we can reach the end of each edge in constant time as we know the number of characters on each edge as per our implementation assumption.

**Why does this help us to reduce the work?** Let the node depth of a node $v$ be the number of nodes on the path from root to $v$ (Ref: Gusfield). The node depth of any node $v$ is at most 1 higher than node-depth of node pointed by suffix link from $v$. (Ref: Lemma 6.1.2, Gusfield)

When $v$ has a suffix link to $s(v)$, and if the label on the path from the root to $v$ is $x\alpha$, where $\alpha$ is a non-empty string, then the label on the path from the root to $s(v)$ is α (by def. of suffix links). Therefore the ancestors of $v$ have suffix links to ancestors of $s(v)$. Since node depths of any two ancestors must be different, each ancestor of $v$ has a distinct link to each ancestor of $s(v)$. The only extra ancestor that $v$ can possible have is the ancestor labeled by a single character $x$ . Hence the node depth of $v$ can be at most one more than depth of $s(v)$.

The rest of the argument is similar to the potential function argument of KMP and Aho-Corasick algorithm. We know the maximum depth of a node can be at most $m$ (the length of the string). Now, in order to find a suffix link, we might have to walk up at most one node. Also, we might lose one more in node depth when we follow the suffix link to the new node. Since there are at most $m$ suffixes in each stage, we can decrement the depth at most $2m$ times. Since the maximum depth is $m$, the total number of increments to the node depth is bounded by $3m$, which is O(m).

However, this only ensures that each of the $i$ phases run in $O(m)$, and the entire algorithm is still $O(m^2)$. The following two tricks help us to reduce it to $O(m)$.

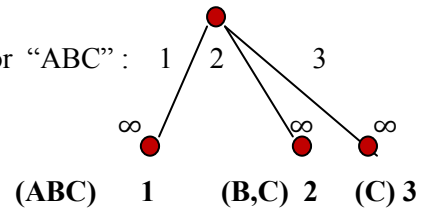
Before we can describe the next trick, one point needs to be mentioned. The edge labels are represented as pairs of integers $(p, e)$, for the suffix $S[p..e]$. Since we can retrieve the characters from the string in each position in constant time, we continue to describe the algorithm as if the edges were labeled with the entire strings. This representation also helps to save storage space from $O(m^2)$ to $O(m)$ (under the standard RAM model where each integer can be represented in $log\ m$ bits).

**Trick 3: Once a leaf, always a leaf**

If at some point in the algorithm, a leaf is created and labeled $j$ (for the suffix starting at position $j$), then throughout the algorithm, this will remain a leaf with label $j$ in all the successive tree creations in the algorithm. We do not ever extend a leaf node to an edge in the algorithm. In phase $(i + 1)$, when we extend the suffixes in stage $i$, we simply add the character $S(i + 1)$ to the leaves. So once a leaf is created, we extend it only by the case (i) described previously, by simply appending the character to its end. Also, it is obvious that for every leaf node with the edge label $(p, q)$ leading to it, the value of $q = i$ in the $i^{th}$ stage, since a leaf can only represent the end of a suffix $S[p..i]$.

This suggests that we can reduce the work further. The integer label of each leaf would be anyway updated in the final stage of construction in the actual suffix tree to $m$ (length of the string). Hence there is no need of explicitly changing the integer label of a leaf node in every stage. This means once a leaf is created, in every subsequent stage we can avoid having to go and update the leaf in every stage. We can just label every leaf as $\infty$, i.e,, every edge leading to a leaf can be labeled as $(p, \infty)$ in the intermediate stages.

Example implicit suffix tree for "ABC" :




One more trick is necessary to make Trick 3 more effective and to actually reduce the run time to linear.

**Trick 4**

If at any stage $i$, the character $S[i]$ is found in the tree $T_{i-1}$ (case (iii) described earlier) while searching for some suffix starting at position $j$, then it will be found in all the subsequent suffixes of that stage. It is because if $S[j..i]$ is a suffix of $S$, then $S[(j+1)..i]$, $S[(j+2)..i]$ etc. will all be suffixes of S, and if $S[j..i]$ has been found in the tree, the others must also be present. Hence we can just stop the current stage of the algorithm at that point (since it is an implicit suffix tree we do not explicitly label the number of each suffix). We simply remember the starting position of the suffix where we stopped in some stage $i$ of the algorithm. Let this suffix number or position be $j*$.

By trick 3, we know that only the character extensions due to case (ii) need to be done explicitly. Also, any extension by case (ii) would create a new leaf. So if the extensions due to case (i) and (ii) end at $j*$ in stage $i$, i.e., $j*$ is the suffix for which we first see case (iii) (following character is already present in tree), then in the next stage we can start again from $j*$ and ignore the values of $j < j*$. The suffixes $(j < j*)$ would be extended implicitly.

(Note: the first time we see an extension by case (iii) in stage $i$, we cannot see any more extensions by case(i) or (ii) after that in that stage.)

Also, when we start from $j*$ in the next stage, we already know where $j*$ ends (because we traversed it in the previous stage). Hence we avoid any work due to up-walking, following suffix link etc for $j*$.

**Putting all the tricks together**

The implicit extensions in every stage can be thought to take constant time, hence the time taken by them over all the $m$ stages is $O(m)$.

For the explicit extensions, if the algorithm stops at $j = j*$ in the current stage, it again starts from $j*$ in the next stage. Hence this index cannot decrease between successive stages, rather remains the same. The index can increase to at most $m$. Also there are $m$ stages. Therefore the algorithm can make at most $2m$

explicit extensions. We also know that each explicit extension takes time proportional to the number of node skips in each extension by the skip/count trick.

Now, let us consider how many node skips can take place during the course of the algorithm. As we go from one stage to the next, by trick 4, our node depth does not change. Every time we explicitly extend some stage, our node depth can decrease by at most 2 by up walking and following suffix link. When we down-walk by the skip/count trick, our node depth increases by one at each node skip. The total node depth can be $m$ (length of the string). Also, we have shown that there are at most $2m$ explicit extensions. Hence the total amount of work in all down-walking is bounded by $O(m)$. (Ref Theorem 6.1.2 in Gusfield).

When we finish constructing the suffix trees for all the stages $1..m$, we construct the true suffix tree from it. First, we append \$ to our string $S$, and build the suffix tree for $S\$$ from our implicit tree for $S$. This will ensure that every suffix will now be explicitly represented by leaves in the tree. Then we change the labels on the all the $m$ leaves to $m$ from $\infty$. Thus the entire suffix tree is constructed in $O(m)$ time.