1 Suffix tree

Given a string $S$, a suffix tree is leaf-labeled by suffix number in $S$ at leaves. Edges are labeled with substrings of $S$. The key property is: from root to leaf, the concatenation of substring labels along the path spells out the suffix on that leaf. Also, no two edges out a node has same starting char. A very useful property is, every substring can be found on the tree, starting from the root. And the search path is unique. I would suggest you to think about how suffix tree is different from keyword tree in Aho-Corasick algorithm. Example: let string $S = \text{tartar}$. Here, $\$ is assumed to be the smallest in the alphabet. Now try to build suffix tree yourself.

Why is suffix tree (ST) useful? Suppose we want to find whether pattern $P = \text{tar}$ is in $T = \text{tartar}$. We first build a suffix tree for $T$. Then search for $P$ in the suffix tree. For this, we just match $P$ with the unique path that agree with $P$. The first thing to note the path is indeed unique: that is, at each node, we will have at most one branch to follow. This is because in suffix tree, each branch starts with different char (otherwise, the two branches must share longest prefixes and thus violate the ST property). If we can not continue matching (while still something left in $P$), we conclude $P$ is not in $T$. Otherwise, all leaves under the current ending position has an exact match of $P$. Why? The crucial point is that pattern $P$ starting in $T[j]$, iff suffix $T[j..m]$ starts with $P$. Thus all positions of $T$ matching $P$ must have the same $T$. When alphabet size is constant, the search for path to go can be done in constant time. Thus, pattern matching can be done in linear time.

Now how can we construct ST? A trivial procedure is as follows: add each suffix (from suffix 1 to $n$) and each time refine (by splitting existing branches) the current ST to accommodate the current suffix. The running time is clearly $O(n^2)$. The first issue is: the number of characters in the tree is $O(n^2)$ because there is a lot of repetition. But this is easy to address: each label of edge in the tree is a substring of $T$. Thus can use two number: $[s,e]$ to represent a substring instead of explicitly listing all characters in the substring. This allows $O(n)$ space for ST. In 1973, Werner gives the first $O(n)$ algorithm. During the next 10-15 years, a few other $O(n)$ algorithms appeared. All have some similarity on the high-level. Here: we mainly cover Ukkonen’s algorithm as in Gusfield’s book. This algorithm is based on the so-called implicit ST (iST). An iST is obtained by first removing $\$ and removing edges from ST with empty labels afterwards. Now you should try the above tartar example to see what iST looks like. Note that conversion between ST and iST is simple: say you want to construct a ST from iST. Then we just add one more character $\$ to the tree (extend) and that is it. Why?

Observation one: if last character is unique (nowhere else in $S$), then iST is almost the same as ST: add one more last symbol at the root; add one more char for each leaf. This clearly can be done in $O(n)$ time.

Why? You can not have a suffix being prefix of some other suffix. So can not shrink a leaf into the middle of edge. However, if the last character is not unique, then iST can have suffix that correspond not to leaf; not even internal nodes because a node in the original ST may have been removed. Again, look at the tartar$\$$ example to see the situation. How to add $\$ in iST? This is exactly what Ukkonen’s algorithm will do: each time it builds an iST by having one more symbol. That is, this algorithm builds iST for prefix $T[1..i]$ for increasing $i$. Note: this algorithm can be thought of as on-line algorithm: you can run the algorithm without knowing full $T$. Here is the
high-level procedure. iST for $T[1..1]$ is trivial (one edge). Then for each $i = 2, \ldots, n$, we consider each $j = 1, \ldots, i$. Refine suffix $T[j,i-1]$ in the previous iST (for $T[1, i-1]$) so that $T[j,i]$ is in the iST. So what do we mean by refine?

Suppose we have an iST for $T[1..i-1]$. Now we have one more symbol and the text is $T[1..i]$, and need to have an iST that has each of the new (longer) suffix. Often, this needs to just add one more character in the leaf edge label (if the suffix corresponds to a leaf). But since in iST, a suffix may be everywhere in the middle of an edge (and without any mark at all), then we have two more cases: if the new (longer) suffix is already contained in the old iST (i.e. trace this new suffix and we end inside some edge and have all matched), in this case, nothing needs to be done; Why? We know a leaf is from somewhere in the middle till the end of this suffix. Thus when extending, no need to adjust the ending of leaves: they all end at the current suffix boundary. Otherwise, if we have a mismatch, then we need to create a new branch out of an edge of iST. As an example, let $S = axabxb$. Suppose we have $S[1..5]$’s iST, and then we need to add b. For the first four suffixes, we just extend leaf. For the fifth suffix, we need to create a new branch. For the sixth suffix, we do nothing because b is already a prefix of suffix $bxb$. Before continuing, we analyze the time.

May need $O(n)$ rounds for each addition of character in $T$; $O(n)$ steps for extending each suffix of $T[1..i]$; each extension needs to trace to the position where is the place to extend (again $O(n)$). So the overall time is $O(n^3)$!

Ukkonen has several tricks to speed up this to $O(n)$. A major trick is suffix link (SL). Finding the suffix to extend takes time. Suffix links provides a faster way to finding the position of iST to refine. For a node $v$ with path label $x\alpha$ (where $x$ is single character), if there is another node $u$ in iST with path label $\alpha$, we create a SL from $v$ to $u = s(v)$. If $\alpha$ is empty, then $s(v)$ is the root. Can we have internal nodes of iST without SL? The following lemma says NO.

**Lemma 1.1.** (6.1.1 in Gusfield’s book) When a new internal node with path label $x\alpha$ is added to iST, either there is already a node with path label $\alpha$ in iST or a new node will be created with $\alpha$ path label (and thus we can create a SL from $v$ to $s(v)$).

Why? New internal node is created only when adding the new character $T[i]$, we reach a position of iST (with path label $x\alpha$) that continue with a different char (say $c$) other than the one in the suffix. Thus we have a suffix in the prior round w/ path label $\alpha$ followed by $c$ as well. Then if $\alpha$ continues with another symbol (other than $c$), then $\alpha$ already labels an internal node. If $\alpha$ continues only with $c$, then at the end of the iteration round, will need to extend with the new symbol (the one in the $x\alpha$ case). Here, you should pause here to see the properties yourself.

SL allows shortcuts when searching for positions to refine. First, $T[1..i]$ is always a leaf in iST (it is the longest). So in each round, we keep a pointer to the leaf correspond to this suffix. And when extend, just add the new symbol. But how to find other suffixes? We consider the suffix $T[2..i]$ (others are similar). Note $T[1..i]$ has one more char than $T[2..i]$. Let edge $(v, 1)$ be the leaf edge entering to suffix 1. If $v$ is the root, then we just start from the root and walk down by comparing to find where to extend. But if $v$ is an internal node (and thus has a SL): then we can follow SL to $s(v)$ and this way avoid comparing the path label of $s(v)$! Once reaching $s(v)$ we then continue walking down. For other extension, the same approach applies: each time we walk up to only one node in iST; follow the SL (or just start from root); walk down by comparing; once find the place, then extend the suffix with rules before; also need to maintain SL for newly created internal nodes.

SL clearly saves time, but itself does not fully provide an improvement of time in worst case: walking down by direct comparison seems to still take $O(n)$ time in worst case (and thus still have $O(n^3)$ time). Here is trick 2: Skip/Count. When walking down from a node for matching some substring of $T$ in iST, recall there is only a single possible edge to follow. On that edge, there is
no need to compare with any char other than the starting symbol; that is, we just follow it w/o explicit comparison! Wait: can we be wrong (i.e. although the substring we try to match matches the first char of the edge but does not match for the following?) This will not happen: since there is a single edge and every path label appears in $T$; if there is chance for wrong match, then must have internal nodes breaking this edge. This tricks means we only need to consider the number of tree nodes (rather than the length of $\alpha$).

Now we need to consider the property of the node-depth (i.e. the number of nodes on the path from root to node $u$).

**Lemma 1.2.** (6.1.2 in Gusfield’s book) Depth of $v$ is at most one larger than depth of $s(v)$.

Why? Consider each internal node $x$ on the path from root to $v$. $s(x)$ must also appear along the path from root to $s(v)$ (i.e. prefix). Also, there can be no sharing of these $s(x)$ since the depth is different. So, the number of ancestral nodes from root to $v$ has at most one more than that for $s(v)$. Now an important observation.

**Theorem 1.3.** (Theorem 6.1.1 in Gusfield’s book). Each extension phase (for adding a new char) takes $O(n)$ time (instead of $O(n^2)$).

Proof. Each extension takes up to $O(n)$ extensions; each extension first go up by one node in iST; then walk down the iST. Note: following SL will reduce node-height by 1 at most. So the total node height decrease is at most $2n$: $n$ due to walk up and $n$ due to SL. Note: each down-walk takes $O(1)$ time and increase node height by 1. But since the node height is at most $n$. Thus, the total increase of node height is no more than $3n$. So, the total number of downward traversal is bounded by $3n$. The total time is thus $O(n)$.

Well this is nice but the time is only reduced to $O(n^2)$. We now give two more tricks and we get $O(n)$ time. First observation: if we ever find a suffix extension leading to no-op (when the new suffix is already in iST), then every follow-up suffix is also in. Why? in this case, whenever $\alpha$ appears in $T$, it always followed by the same char (as in the new suffix). Thus, the shorter suffix does the same. And in this case, no new internal node is created and no SL needs to add. This means we can stop whenever no-op extension is found. Second observation: once a leaf, always a leaf. Note: when a leaf is created, its label can be extended but it nevers becomes an internal node (i.e. branching out of it). This is ensured by how extension is performed: no rule will split from a leaf (only adding a new char). Then, what should we do with the existing leaves? We just extend the range by adding one char. Now since we are extending and note each following phase we just keep add one by one. Instead of updating each time one by one, we only need to mark these leaves as ending at the $e$ (the global ending point of range which increase from 1 to $n$). Then when we construct all iST, just replace these $e$ with $n$ (that is it). One last thing: the refinement of each phases will create new leaves (once created, only need to extend since they are always leaves). Note the number of leaves involved during the initial sequence of type-1/2 will not decrease: leaf remains a leaf. Thus, we can just skip them and only perform implicit refinement as before. Between phases, we only share one: the previous ones are done for existing leaves.

With all these tricks, I claim the time runs is $O(n)$. Why? Time for all implicit extension is constant each phase and thus $O(n)$ overall. For explicit extensions, since each phase can share just one, the overall number is at most $2n$ and thus $O(n)$ as well. One last issue: the time for searching for extension by walking done may still be an issue. But a similar aggregate analysis shows that the number of down-walks can not be too large: each down walk of nodes increase node-height by 1; each explicit extension increase the node-height by 2 (node up-walk and SL); thus, the total num of down walk is bounded by $O(n)$ since there are up to $2n$ explicit relaxation.