Suffix Trees...

Suffix Trees

Sufix trees form the backbone of many string-processing algorithms, including *string matching* and *palindrome discovery*.

Definition. Given a string S of length m in alphabet Σ, a ***suffix tree*** $T(S)$ is a rooted directed tree with the following properties:

- $T(S)$ has m leaves labeled $1, \ldots, m$.
- Each internal node other than the root node *has at least two children.*
- Each edge $e = (v, u)$ has a label $label(e)$, which a *non-empty substring* of S.
- No two edges (v, u) and (v, w) can have labels that start with the same character.
- Let $r, v_1, \ldots, v_k, i$ be the path in $T(S)$ from the root node r to the leaf node i. Then

$$label(r, v_1)label(v_1, v_2)\ldots label(v_k, i) = S[i, \ldots m].$$

Example. Figure 1 shows the suffix tree constructed for the string $S =$ `ATTAC`.

Note. Suffix trees cannot be constructed if one suffix of S matches a prefix of another suffix in S (i.e., if, essentially, the last character in S occurs elsewhere in S). To rectify this, we use a special character $\$ \notin \Sigma$ as the terminating character of all strings S we consider.

Terminology. The **label of a path** from root r of $T(S)$ to a node v (a.k.a., the **path-label of** v), denoted $label(v)$ is defined as

$$label(v) = label(r, v_1)label(v_1, v_2)\ldots label(v_k, v),$$

where $r, v_1, \ldots, v_k, v$ is the path from r to v in $T(S)$.

The **string-depth** of a node v in $T(S)$ is $|label(v)|$: the length of the path-label of v.

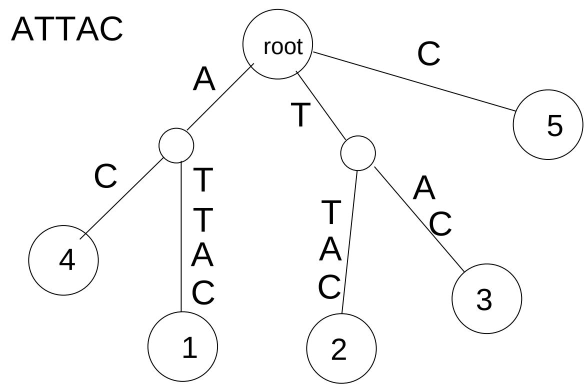

Figure 1: Suffix tree for string `ATTAC`.

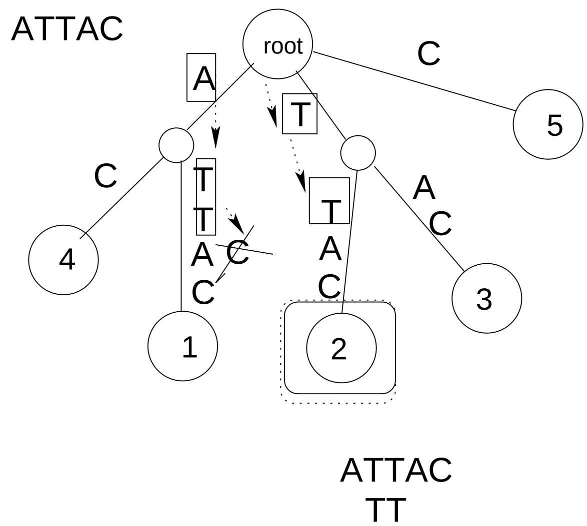

Figure 2: Searching for occurrences of `TT` (successful) and `ATTC` (unsuccessful) in string `ATTAC`.

String Matching Using Suffix Trees.

The following algorithm finds all matches of string P in a query string S.

1. Build $T(S)$.
2. Match P along a path in S.
   (a) If P is exhausted, then **all leaves** below the current position in the tree will contain the positions of occurrences of P in S.
   (b) If P cannot be matched, then P does not occur in S.

Example. Figure 2 shows a successful attempt to find a substring `TT` in string $S =$ `ATTAC`, and an unsuccessful attempt to find a substring `ATTC`.

Analysis. We note the following:

- The matching path of P in $T(S)$ is unique (i.e., no two different paths in $T(S)$ starting from the root node share a prefix).
- If Σ is fixed, then each node can be traversed in constant time $O(|\Sigma|)$.
- Matching P in $T(S)$ takes $O(|P|)$ time.

Naïve Algorithm for Suffix Tree Construction

Suffix tree construction problem. Given a string $S = s_1 \ldots s_n \in \Sigma^*$, build a tree $T(S\$)$, where "$" $\notin \Sigma$ is a special terminating character.

Algorithm. The algorithm constructs a sequence of trees $N_1, \ldots, N_{n+1}$, where $N_{n+1} = T(S\$)$. The construction proceeds as described in the following inductive procedure:

1. N_1 has two nodes, r (root) and 1, and an edge $(r, 1)$ with $label(r, 1) = S\$$.
2. Let N_i be constructed. We construct N_{i+1} as follows. Starting at r (root node of N_i), traverse N_i matching characters $s_{i+1}s_{i+2} \ldots s_n\$$ to the path label.
    - If some prefix $s_{i+1} \ldots s_{i+j}$ matches a path label in the middle of an edge label for some edge (u, v), then
        - Create a new node v'.
        - Delete edge (u, v). Insert edge (u, v') labelled with the prefix of $label(u, v)$ which matched the suffix of $s_{i+1} \ldots s_{i+j}$.
        - Insert edge (v', v) labeled with the remainder of $label(u, v)$.
        - Create a new node $i + 1$. Insert edge $(v', i + 1)$. Set $label(v', i + 1) = s_{i+j+1} \ldots s_n\$$.
    - If some prefix $s_{i+1} \ldots s_{i+j}$ matches a path-label of some node $v \in N_i$, then
        - Create new node $i + 1$.
        - Insert an edge $(v, i + 1)$. Set $label(i + 1) = s_{i+j+1} \ldots s_n\$$.

Example. Figure 3 shows the steps of construction of $T($`ATTAC$`$)$ using the naïve algorithm. On steps 3 and 4 of the construction, the longest match (`"T"` and `"A"` respectively) splits an edge label and leads to creation of internal nodes.

Analysis. Each step of the algorithm requires $O(n)$ operations. There are $n + 1$ steps, hence the naïve algorithm for suffix tree construction works in $O(n^2)$ time.

Linear-time Construction of Suffix Trees

We describe a linear-time algorithm proposed by Ukkonen[1].

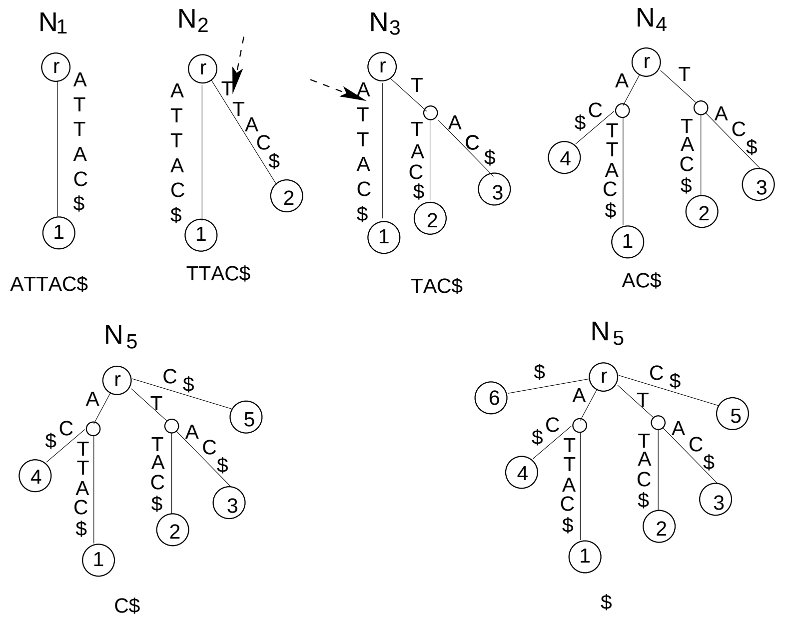

Figure 3: Construction of the suffix tree for string `ATTAC$` using the naïve algorithm.

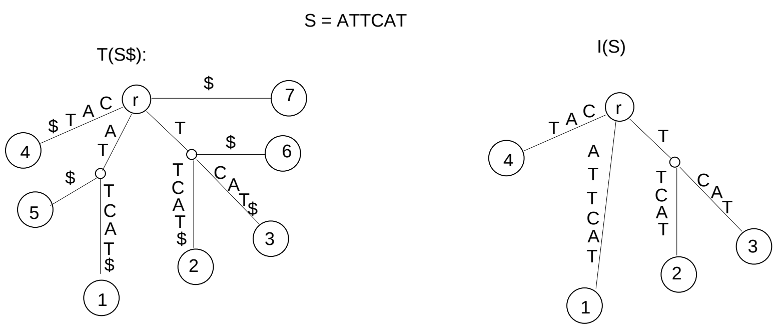

Figure 4: The suffix tree and the implicit suffix tree for string `ATTCAT`.

Implicit suffix trees. Given a string $S = s_1 \ldots s_n$, its *implicit suffix tree* is a tree constructed from the suffix tree $T(S\$)$ as follows:

1. Remove \$ from every edge label.

2. Remove any edge without a label.

3. Remove any node that has only one child.

Given a string S, $I(S)$ denotes its implicit suffix tree. $I_i(S)$ denotes the implicit suffix tree of the substring $s_1 \ldots s_i$ of S.

Example. Figure 4 shows the suffix tree and the implicit suffix tree for a string `ATTCAT`.

Note. The implicit suffix tree for S has fewer leaves than $T(S\$)$ iff at least one suffix of S is a prefix of another suffix of S.

Ukkonen's algorithm for building implicit suffix trees. Given a string $S = s_1 \ldots s_n$, The algorithm builds a sequence of implicit suffix trees $I_1(S), \ldots I_n(S)$ as follows.

1. $I_1(S)$ contains two nodes, r (root) and 1 and and edge $(r, 1)$ labeled $label(r, 1) = s_1$.
2. Let $I_i(S)$ be constructed. $I_{i+1}(S)$ is constructed in a sequence of $i + 1$ extensions:
   - On extension $1 \leq j \leq i + 1$, find the path $s_j \ldots s_i$ in $I_i(S)$.
   - If needed (see below), extend the path by adding s_{i+1} to the last edge label.

Suffix extensions. Suffix extensions are performed using the following rules. Let $\beta = s_j \ldots s_i$ for extension step j of ith iteration of the algorithm. Suffix extention rules ensure that the suffix βs_{i+1} is found in the implicit suffix tree $I_{i+1}(S)$. Three cases are possible:

- **Rule 1.** In $I_i(S)$, β ends at a leaf node. Let (v, k) be the last edge of the path. Then we extend the label of (v, k):
$$label_{i+1}(v, k) = label_i(v, k)s_{i+1}.$$
- **Rule 2.** In $I_i(S)$, there is at least one path extending β and no path extending β starts with s_{i+1}. Then,
  - Create a new leaf node $i + 1$.
  - If β stops at an internal node v, add an edge $(v, i + 1)$, label it with $label_{i+1}(v, i + 1) = s_{i+1}$.
  - If β stops in the middle of an edge label for some edge (v, u), then
    * create a new internal node w;
    * remove (v, u);
    * add edge (v, w) labeled with the part of β that was the prefix of $label(v, u)$;
    * add edge (w, u) labeled with the part of $label(v, u)$ that follows β.
    * add edge $(w, i + 1)$ labeled $label_{i+1}(w, i + 1) = s_{i+1}$.
- **Rule 3.** There is a path following β in $I_i(S)$ that starts with s_{i+1}. In this case, do nothing.

Example. The three extension rules are illustrated in Figure 5. Figure 5 shows the progression of implicit suffix trees I_1, I_2, I_3 and I_4 for string $S =$`ATAC`. Extension steps are marked as numeric labels near the paths to which they correspond.

Rule 1 is used to extend $I_1(S)$ to $I_2(S)$ on extension steps 1 and 2. **Rule 2** is used exactly once in the construction of $I_4(S)$ on extension step 3: at this point, we are trying to place the suffix `AC` into the tree, and this leads to a split on a path from root to node 1. **Rule 3** is used on extension step 3 when building $I_3(S)$ - suffix `A` is found in the tree, and no alterations are made.

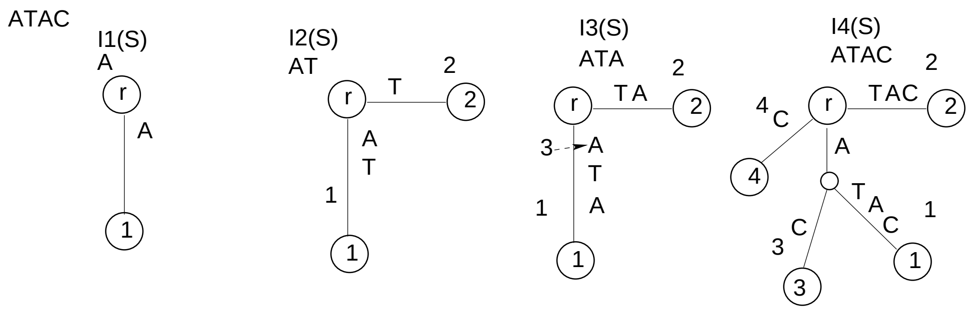

Figure 5: Suffix extension rules for Ukkonen's implicit suffix tree construction algorithm.

Analysis. The key step of Ukkonen's algorithm is building extensions on each extension step. There are $O(n^2)$ extension steps. Each step can *naïvely* take $O(n)$ to complete, hence a naïve implementation of Ukkonen's algorithm will have running time $O(n^3)$, which is worse than the direct naïve construction of a suffix tree.

Efficient implementation of Ukkonen's Algorithm.

Problem. Given $\beta = s_j \dots s_i$ and $N_i(S)$, locate the ends of β in $N_i(S)$.

Naïve discovery of β in $N_i(S)$ is what gives us the $O(n^3)$ algorithm.

Challenge. Speed up suffix discovery.

Speedup 1: Suffix links. Consider a string $x\alpha$ where $x \in \Sigma$ and $\alpha \in \Sigma^*$ (possibly empty). Let v be an internal node of some implicit suffix tree $I_i(S)$ and let $label(r, v) = x\alpha$. If there is another node $s(v) \neq v$ such that $label(r, s(v)) = \alpha$, then a *suffix link* is a pointer from s to $s(v)$.

Lemma. Consider the process of building $I_{i+1}(S)$ from $I_i(S)$. Let new internal node v be added to $I_{i+1}(S)$ on extensions step j, and let $label(r, v) = x\alpha$. Then, either there exists an internal node u, such that $label(r, u) = \alpha$, or, on extension $j+1$ of the current phase, an internal node u, s.t., $label(r, u) = \alpha$ will be created.

Corollary. In Ukkonen's algorithm, every newly created internal node will have a suffix link from it by the end of the next extension.

Corollary. In any $I_i(S)$, if some internal node v has path-label $x\alpha$, there exists a node $s(v)$ in $I_i(S)$ with path-label α.

First extension in an implicit suffix tree. Given $I_i(S)$, the first extension step of $I_{i+1}(S)$ extends $s_1 \dots s_i$, which is a path to a leaf node in $I_i(S)$. We can store a pointer to this node in the root r (note: it will be the same node on every step phase $i = 1, \dots n$. Therefore, we can perform extension step $j = 1$ in constant time on each phase.

Using suffix links. We use the suffix links to speed up tree traversal during extension steps as follows:

1. Let $i+1$ be the phase, and $j \geq 2$ be the extension step. At this point, we assume that we know where the string $s_{j-1} \ldots s_i$ ends in $I_i(S)$.
2. Find the first node v above the end of $s_{j-1} \ldots s_i$ that either is the root r, or has a suffix link from it. Let γ is the string between v and s_i.
3. If v is NOT root, traverse suffix link from v to $s(v)$. Walk down from $s(v)$ following γ.
4. If $v = r$, follow $s_j \ldots s_i$.
5. Use extension rules to get $s_j \ldots s_i s_{i+1}$ is in the tree.
6. If used extension **rule 2** and created a new internal node w, then $s(w)$ is the end node for the suffix link from w. Create suffix link $(w, s(w))$.

Skip/count trick. On extension step $j+1$ we start at $s(v)$ and traverse the string γ down the tree. We want to make this traversal efficient.

Naïve traversal is $O(|\gamma|)$. Let $|\gamma| = g$ and let $\gamma = b_1 \ldots b_g$.

First character of γ must appear at exactly one edge $(s(v), w)$ out of $s(v)$. Let $g' = |label(s(v), w)|$ be the number of characters on that edge. If $g' \leq g$, then we can skip directly to w, w/o traversing γ along the edge. From w we need to find $b_{g'+1} \ldots b_k$. We repeat the same trick.

If $g' > g$, then skip to character g on $label(s(v), w)$.

Edge-label compression. All edge labels are substrings of S. Instead of explicitly writing them out – it might take $O(n^2)$ space (and thus require quadratic time to access — we replace each edge label with a pair of numbers i, j, such that the edge label is $s_i \ldots s_j$ and i, j is the smallest pair of indexes for which it is true.

Rule 3 trick. Any phase $i+1$ can be ended immediately after **rule 3** is applied.

Final trick. Once a leaf node is created in some $I_i(S)$, it will remain a leaf node in all followup $I_k(S)$, $k > i$.

On phase $i+1$, when a leaf edge is created to be labeled with $s_p \ldots s_{i+1}$ ($[p, i+1]$), replace $i+1$ with some index e signifying *"the current end"*. On each phase, set e to $i+1$ once.

Single phase algorithm. The tricks above lead to the following single phase algorithm.

1. On phase $i+1$:
2. Increment e to $i+1$. (this implements all extensions $1 \ldots j_i$).
3. Explicitly compute extensions starting at $j_i + 1$ until reaching extension j^* where **rule 3** applies, or until all extensions are done.
4. Set j_{i+1} to $j^* - 1$ for the next phase.

Creating the true suffix tree. We can construct $T(S\$)$ from $I_n(S)$ by adding \$ to the end of S and extending $I_n(S)$ to $I_{n+1}(S\$)$.

References

[1] Esko Ukkonen (1995). On-line Construction of Suffix Trees, *Algorithmica*, Vol. 14, pp. 249-260.