Brute Force Construction

Let us now look at algorithms for constructing the suffix tree. We start with a brute force algorithm with time complexity $\Theta(DP(T_{[0..n]}))$. This is later modified to obtain a linear time complexity.

The idea is to add suffixes to the trie one at a time starting from the longest suffix. The insertion procedure is essentially the same as in Algorithm 3.26 (AC trie construction) except it has been modified to work on a compact trie instead of a trie.

The suffix tree representation uses four functions:

child(u, c) is the child v of node u such that the label of the edge (u, v) *starts* with the symbol c, and $\perp$ if u has no such child.

parent(u) is the parent of u.

depth(u) is the length of the string S_u represented by u.

start(u) is the starting position of some occurrence of S_u in T.

Then

- $S_u = T[\mathit{start}(u) \ldots \mathit{start}(u) + \mathit{depth}(u))$.
- $T[\mathit{start}(u) + \mathit{depth}(\mathit{parent}(u)) \ldots \mathit{start}(u) + \mathit{depth}(u))$ is the label of the edge $(\mathit{parent}(u), u)$.
- A *pair* (u, d) with $\mathit{depth}(\mathit{parent}(u)) < d \leq \mathit{depth}(u)$ represents a position on the edge $(\mathit{parent}(u), u)$ corresponding to the string $S_{(u,d)} = T[\mathit{start}(u) \ldots \mathit{start}(u) + d)$.

Note that the positions (u, d) correspond to nodes in the uncompact trie.

Algorithm 4.2: Brute force suffix tree construction
Input: text $T[0..n]$ ($T[n] = \$$)
Output: suffix tree of T: $root$, *child*, *parent*, *depth*, *start*

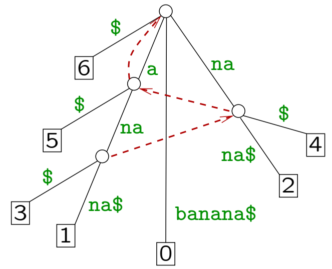
create new node $root$; $\mathit{depth}(root) \leftarrow 0$
$u \leftarrow root$; $d \leftarrow 0$
for $i \leftarrow 0$ *to* n *do* // insert suffix T_i
    while $d = \mathit{depth}(u)$ *and* $\mathit{child}(u, T[i+d]) \neq \perp$ *do*
        $u \leftarrow \mathit{child}(u, T[i+d])$; $d \leftarrow d + 1$
        while $d < \mathit{depth}(u)$ *and* $T[\mathit{start}(u) + d] = T[i+d]$ *do* $d \leftarrow d + 1$
    if $d < \mathit{depth}(u)$ *then* // we are in the middle of an edge
        create new node v
        $\mathit{start}(v) \leftarrow i$; $\mathit{depth}(v) \leftarrow d$
        $p \leftarrow \mathit{parent}(u)$
        $\mathit{child}(v, T[\mathit{start}(u) + d]) \leftarrow u$; $\mathit{parent}(u) \leftarrow v$
        $\mathit{child}(p, T[i + \mathit{depth}(p)]) \leftarrow v$; $\mathit{parent}(v) \leftarrow p$
        $u \leftarrow v$
    create new leaf w // w represents suffix T_i
    $\mathit{start}(w) \leftarrow i$; $\mathit{depth}(w) \leftarrow n - i + 1$
    $\mathit{child}(u, T[i+d]) \leftarrow w$; $\mathit{parent}(w) \leftarrow u$
    $u \leftarrow root$; $d \leftarrow 0$

Suffix Links

The key to efficient suffix tree construction are suffix links:

slink(u) is the node v such that S_v is the longest proper suffix of S_u, i.e., if $S_u = T[i..j)$ then $S_v = T[i+1..j)$.

Example 4.3: The suffix tree of $T =$ `banana$` with internal node suffix links.



Suffix links are well defined for all nodes except the root.

Lemma 4.4: If the suffix tree of T has a node u representing $T[i..j)$ for any $0 \leq i < j \leq n$, then it has a node v representing $T[i+1..j)$

Proof. If u is the leaf representing the suffix T_i, then v is the leaf representing the suffix T_{i+1}.

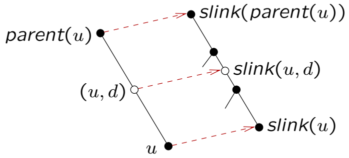
If u is an internal node, then it has two child edges with labels starting with different symbols, say a and b, which means that $T[i..j)a$ and $T[i..j)b$ occur somewhere in T. Then, $T[i+1..j)a$ and $T[i+1..j)b$ occur in T too, and thus there must be a branching node v representing $T[i+1..j)$. $\square$

Usually, suffix links are needed only for internal nodes. For root, we define *slink*$(root) = root$.

Suffix links are the same as Aho–Corasick *failure links* but Lemma 4.4 ensures that *depth*(*slink*(u)) = *depth*$(u) - 1$. This is not the case for an arbitrary trie or a compact trie.

Suffix links are stored for compact trie nodes only, but we can define and compute them for any position represented by a pair (u, d):

slink(u, d)
- $v \leftarrow$ *slink*(*parent*(u))
- *while depth*$(v) < d - 1$ *do*
- $\quad v \leftarrow$ *child*$(v, T[$*start*$(u) +$ *depth*$(v) + 1])$
- *return* $(v, d - 1)$



The same idea can be used for computing the suffix links during or after the brute force construction.

ComputeSlink(u)
(1) $v \leftarrow$ *slink*(*parent*(u))
(2) *while depth*$(v) <$ *depth*$(u) - 1$ *do*
(3) $\quad v \leftarrow$ *child*$(v, T[$*start*$(u) +$ *depth*$(v) + 1])$
(4) *if depth*$(v) >$ *depth*$(u) - 1$ *then*
(5) $\quad$ create new node w
(6) $\quad$ *start*$(w) \leftarrow$ *start*$(u) + 1$; *depth*$(w) \leftarrow$ *depth*$(u) - 1$; *slink*$(w) \leftarrow \perp$
(7) $\quad p \leftarrow$ *parent*(v)
(8) $\quad$ *child*$(w, T[$*start*$(v) +$ *depth*$(w)]) \leftarrow v$; *parent*$(v) \leftarrow w$
(9) $\quad$ *child*$(p, T[$*start*$(w) +$ *depth*$(p)]) \leftarrow w$; *parent*$(w) \leftarrow p$
(10) $\quad v \leftarrow w$
(11) *slink*$(u) \leftarrow v$

The algorithm uses the suffix link of the parent, which must have been computed before. Otherwise the order of computation does not matter.

The creation of a new node on lines (4)–(10) is not necessary in a fully constructed suffix tree, but during the brute force algorithm the necessary node may not exist yet:

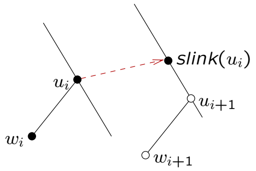
- If a new internal node u_i was created during the insertion of the suffix T_i, there exists an earlier suffix T_j, $j < i$ that branches at u_i into a different direction than T_i.
- Then *slink*(u_i) represents a prefix of T_{j+1} and thus exists at least as a position on the path labelled T_{j+1}. However, it may be that it does not become a branching node until the insertion of T_{i+1}.
- In such a case, ComputeSlink(u_i) creates *slink*(u_i) a moment before it would otherwise be created by the brute force construction.

McCreight's Algorithm

McCreight's suffix tree construction is a simple modification of the brute force algorithm that computes the suffix links during the construction and uses them as *short cuts*:

- Consider the situation, where we have just added a leaf w_i representing the suffix T_i as a child to a node u_i. The next step is to add w_{i+1} as a child to a node u_{i+1}.

- The brute force algorithm finds u_{i+1} by traversing from the root. McCreight's algorithm takes a short cut to *slink*(u_i).



- This is safe because *slink*(u_i) represents a prefix of T_{i+1}.

Algorithm 4.5: McCreight
Input: text $T[0..n]$ ($T[n] = \$$)
Output: suffix tree of T: $root$, *child*, *parent*, *depth*, *start*, *slink*

(1) create new node $root$; *depth*($root$) $\leftarrow 0$; *slink*($root$) $\leftarrow root$
(2) $u \leftarrow root$; $d \leftarrow 0$
(3) *for* $i \leftarrow 0$ *to* n *do* // insert suffix T_i
(4)   *while* $d =$ *depth*(u) *and* *child*($u, T[i+d]$) $\neq \perp$ *do*
(5) $u \leftarrow$ *child*($u, T[i+d]$); $d \leftarrow d + 1$
(6)   *while* $d <$ *depth*(u) *and* $T[$*start*$(u) + d] = T[i + d]$ *do* $d \leftarrow d + 1$
(7)   *if* $d <$ *depth*(u) *then* // we are in the middle of an edge
(8)   create new node v
(9)   *start*(v) $\leftarrow i$; *depth*(v) $\leftarrow d$; *slink*(v) $\leftarrow \perp$
(10) $p \leftarrow$ *parent*(u)
(11)   *child*($v, T[$*start*$(u) + d]$) $\leftarrow u$; *parent*(u) $\leftarrow v$
(12)   *child*($p, T[i +$ *depth*$(p)]$) $\leftarrow v$; *parent*(v) $\leftarrow p$
(13) $u \leftarrow v$
(14)   create new leaf w // w represents suffix T_i
(15)   *start*(w) $\leftarrow i$; *depth*(w) $\leftarrow n - i + 1$
(16)   *child*($u, T[i+d]$) $\leftarrow w$; *parent*(w) $\leftarrow u$
(17)   *if* *slink*(u) $= \perp$ *then* ComputeSlink(u)
(18) $u \leftarrow$ *slink*(u); $d \leftarrow d - 1$

Theorem 4.6: Let T be a string of length n over an alphabet of constant size. McCreight's algorithm computes the suffix tree of T in $\mathcal{O}(n)$ time.

Proof. Insertion of a suffix T_i takes constant time except in two points:

- The while loops on lines (4)–(6) traverse from the node *slink*(u_i) to u_{i+1}. Every round in these loops increments d. The only place where d decreases is on line (18) and even then by one. Since d can never exceed n, the total time on lines (4)–(6) is $\mathcal{O}(n)$.

- The while loop on lines (2)–(3) during a call to ComputeSlink(u_i) traverses from the node *slink*(*parent*(u_i)) to *slink*(u_i). Let d'_i be the depth of *parent*(u_i). Clearly, $d'_{i+1} \geq d'_i - 1$, and every round in the while loop during ComputeSlink(u_i) increases d'_{i+1}. Since d'_i can never be larger than n, the total time in the loop on lines (2)–(3) in ComputeSlink is $\mathcal{O}(n)$.

$\square$

There are other linear time algorithms for suffix tree construction:

- Weiner's algorithm was the first. It inserts the suffixes into the tree in the opposite order: $T_n, T_{n-1}, \ldots, T_0$.

- Ukkonen's algorithm constructs suffix tree first for $T[0..1)$ then for $T[0..2)$, etc.. The algorithm is structured differently, but performs essentially the same tree traversal as McCreight's algorithm.

- All of the above are linear time only for constant alphabet size. Farach's algorithm achieves linear time for an integer alphabet of polynomial size. The algorithm is complicated and unpractical.

Applications of Suffix Tree

Let us have a glimpse of the numerous applications of suffix trees.

Exact String Matching

As already mentioned earlier, given the suffix tree of the text, all occ occurrences of a pattern P can be found in time $\mathcal{O}(|P| + occ)$.

Even if we take into account the time for constructing the suffix tree, this is asymptotically as fast as Knuth–Morris–Pratt for a single pattern and Aho–Corasick for multiple patterns.

However, the primary use of suffix trees is in *indexed string matching*, where we can afford to spend a lot of time in preprocessing the text, but must then answer queries very quickly.

Approximate String Matching

Several approximate string matching algorithms achieving $\mathcal{O}(kn)$ worst case time complexity are based on suffix trees.

Filtering algorithms that reduce approximate string matching to exact string matching such as partitioning the pattern into $k + 1$ factors, can use suffix trees in the filtering phase.

Another approach is to generate all strings in the k-neighborhood of the pattern, i.e., all strings within edit distance k from the pattern and search for them in the suffix tree.

The best practical algorithms for indexed approximate string matching are hybrids of the last two approaches.

Text Statistics

Suffix tree is useful for computing all kinds of statistics on the text. For example:

- The *number of distinct factors* in the text is exactly the number of nodes in the (uncompact) trie. Using the suffix tree, this number can be computed as the total length of the edges plus one (root/empty string). The time complexity is $\mathcal{O}(n)$ even though the resulting value is typically $\Theta(n^2)$.

- The *longest repeating factor* of the text is the longest string that occurs at least twice in the text. It is represented by the deepest internal node in the suffix tree.

Generalized Suffix Tree

A generalized suffix tree of two strings S and T is the suffix three of the string $S£T\$$, where £ and \$ are symbols that do not occur elsewhere in S and T.

Each leaf is marked as an S-leaf or a T-leaf according to the starting position of the suffix it represents. Using a depth first traversal, we determine for each internal node if its subtree contains only S-leafs, only T-leafs, or both. The deepest node that contains both represents the *longest common factor* of S and T. It can be computed in linear time.

The generalized suffix tree can also be defined for more than two strings.