# Algorithmic Design / Algorithmic Data Mining

## Data Mining - Lecture 7

*Tommaso Padoan* <tommaso.padoan@units.it>

## Suffix tree

A suffix tree is a data structure in the form of a tree that exposes the internal structure of a string. It can be used to perform pattern matching in linear time (hence achieving the same efficiency of KMP) but also to solve more complex problems related to strings. In a sense, it works oppositely to KMP, by performing the preprocessing of the text, instead of the pattern, and then allowing the matching of any pattern, as many as needed, with no more preprocessing. Because of this, it is especially useful for applications where many, possibly different, pattern searches are requested on the same text.

A suffix tree $\mathcal{T}$ for a string $S$ is a directed tree with exactly $|S|$ leaves, numbered from $0$ to $|S|-1$, where:

- every (non-root) inner node has at least two children
- every edge is labelled with a nonempty substring of $S$
- no two edges outgoing from a node have labels beginning with the same character
- the concatenation of the labels of the edges from the root to each leaf $i$ is exactly the suffix of $S$ starting from position $i$, that is, $S[i, \dots, |S|-1]$.

The properties above imply that inner nodes cannot have more children than the size $|\Sigma|$ of the alphabet, which is assumed to be finite. Since a string $S$ has exactly $|S|$ different nonempty suffixes, the tree must, as required, have $|S|$ leaves, one for each suffix. Note that not all strings can have a suffix tree. In particular, this happens for strings that have a suffix which is also the prefix of another suffix. Indeed, the path for the former suffix should also be a subpath of the latter, thus not leading to a leaf, and so having strictly fewer leaves than suffixes. To solve this problem, before building the suffix tree, a string is extended at the end with a *termination* symbol $\$ \notin \Sigma$ not belonging to the alphabet. Treating $\$$ as any other character, now we are sure that no suffix can be the prefix of another, and so a suffix tree is guaranteed to exist.

A suffix tree is more efficient, in both time to operate with and space, than listing all the suffixes of a string. This is because the tree structure is exploited to have the common prefix of different suffixes represented by a single shared path labelled with that prefix. As we will see, this allows to efficiently find all occurrences of any requested pattern in a given text whose suffix tree has been built.

Observe that a suffix tree for a string $S$ cannot have more than $2|S|$ nodes. In fact, since we know that there are exactly $|S|$ leaves and $1$ root, and every other inner node must have at least two children, the number of the latter cannot be more than $|S| - 1$. Since edges are labelled with substrings of $S$, and there is an edge (the ingoing one) for each node in the tree, other than the root, the total space occupied by labels can be up to $O(|S|^2)$ quadratic in the size of the string. A more efficient alternative consists in labelling edges with the starting and ending positions of the corresponding substring, instead of the substring itself. In this way, every edge is labelled with just two indices, taking $\Theta(1)$ space, reducing the total space occupied by labels to just linear $\Theta(|S|)$, since the number of edges is also linear.

## Construction

We first present a basic way to build the suffix tree for a string $S$ of size $|S| = n$, assumed to end with the special termination symbol $\$ \notin \Sigma$ (otherwise the string could simply be extended with that symbol at the end). The construction can be divided in $n$ phases, one for each different suffix of the string. Indeed, phase $i$, for $0 \leq i < n$, will add to the tree the suffix of $S$ starting from position $i$, while adjusting the tree structure as needed. Initially the tree contains only the root. During the first phase $0$, a leaf is created, numbered $0$, and linked as the only child to the root through an edge labelled with the whole string $S$ itself, which is the suffix starting from position $0$. Then, in a generic phase $i$, the suffix starting from position $i$, that is $S[i, \ldots, n{-}1]$, is matched, for as many characters as possible, against the labels of edges, starting from those outgoing from the root. Since no two edges outgoing from a node have labels beginning with the same character, from every inner node there can be at most one outgoing edge matching the current character of the suffix, possibly none. Furthermore, at some point a character will not match any of the existing edge labels, because the suffix cannot be the prefix of another suffix thanks to the termination symbol. Note that such a mismatch can happen either in the middle of an edge label between two nodes, or at the beginning of every edge outgoing from an inner node. In the latter case, a new edge is added, labelled with the remaining unmatched characters of the suffix, outgoing from that node and leading to a new leaf numbered $i$. In the other case, the current half-matched edge is split into two separate edges, adding a new inner node in between them, one labelled with the matched characters, while the other starting from the mismatch up to the end of the original label. Then, as in the previous case, a new edge is added, labelled with the remaining unmatched characters of the suffix, outgoing from the inner node that was just created and leading to a new leaf numbered $i$. After all the $n$ phases have been performed, the tree will contain a leaf for each different suffix of $S$, and it will satisfy all the required properties. Note that the same steps can be adjusted to produce the suffix tree labelled with intervals of indices instead of substrings, with no additional work needed.

The time complexity of the procedure above is $O(n^2)$ quadratic in the size of the string, since each phase $1 \leq i < n$, that is, except the first one, may perform up to $n - i$ comparisons, thus requiring at most $\sum_{i=1}^{n-1} n - i = \frac{n^2-n}{2} = \Theta(n^2)$ total comparisons in the worst case. Actually, the procedure can be quite improved, up to obtain linear time complexity. Even though we will not see any such algorithm, here we show the key idea that stands at their basis.

Recall that, in the procedure above, at the end of phase $i$ a new leaf $v_i$ was created as a child of a new or already existing inner node $u_i$, whose path from the root is labelled with a prefix of the suffix $S[i, \ldots, n{-}1]$. The next phase $i{+}1$ aims at creating a new leaf $v_{i+1}$ for the suffix $S[i{+}1, \ldots, n{-}1]$, as a child of a node $u_{i+1}$ whose path from the root will be labelled with a prefix of the latter suffix. To do this, the procedure would search the node $u_{i+1}$ starting from the root, and descending edges as long as the concatenation of their labels matches a prefix of the suffix. Notice that, the two suffixes of phases $i$ and $i{+}1$, respectively, only differ for one character, that is $S[i]$, indeed $S[i, \ldots, n{-}1] = S[i]S[i{+}1, \ldots, n{-}1]$. Because of this, the prefix corresponding to the searched node $u_{i+1}$, or that of a close ancestor, will actually be a proper suffix of the prefix corresponding to the node $u_i$ of the previous phase. Therefore, the procedure could be much faster if we were able to reach the node $u_{i+1}$, or a close ancestor, directly from $u_i$, instead of searching for it starting from the root of the tree. This kind of link (from $u_i$ to $u_{i+1}$ or one of its close ancestors) is referred to as *suffix link*, and can actually be efficiently computed while constructing the suffix tree. Formally, a suffix link from a node $u$ leads to a node $\mathit{slink}(u) = v$ whose path from the root is labelled with the longest proper suffix of the string labelling the path from the root to $u$. Using suffix links, each phase, including the additional work to compute such links, will take *amortized* constant time. Thus, the total time complexity for the $n$ phases will be just $\Theta(n)$ linear in the size $n$ of the string.

## Pattern matching

The pattern matching procedure to find all occurrences of a pattern $P$ in a text $T$ whose suffix tree has already been computed is quite simple. Starting from the root of the suffix tree, the pattern is matched, character by character, against the labels of descending edges, until one of the following two cases is found.

1. A character of the pattern cannot be matched by any of the currently reached edge labels.
2. All characters of the pattern have been matched, reaching a node $u$ or halfway to such a node (on an edge with leftover characters).

In case (1), the pattern never occurs in the text, and so the procedure terminates, returning the failure. In case (2), instead, we know that the pattern occurs at least once in the text. Then, to find and return all occurrences, we just need to collect all the numbers assigned to the leaves of the subtree rooted in the node $u$. Those numbers will be exactly the positions in which the pattern occurs in the text. The subtree rooted in $u$ can be explored in any way ensuring linear time of its size (e.g. depth-first).

Then, in case (1) the procedure takes at most $\Theta(|P|)$ time to match as many characters of the pattern as possible, until failing and terminating. In case (2), instead, after $\Theta(|P|)$ time to match the entire pattern, it takes at most $\Theta(|T|)$ additional time to find all the leaves in the subtree rooted in $u$ (recall that the suffix tree has at most $2|T|$ nodes). Since $|P| \leq |T|$, this leads to a $\Theta(|T|)$ total time complexity. Furthermore, such asymptotic value does not change even if including the computation of the suffix tree, which also takes $\Theta(|T|)$.

## Exercise

Draw the two alternative suffix trees (using explicit substrings or intervals of indices, respectively) for the text $abaaba$. Then, on one of the trees, highlight the path(s) traversed to find all occurrences of the pattern $a$, specifying also the order in which nodes are visited.