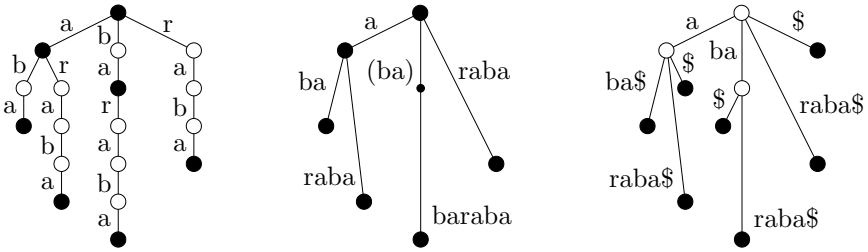
10b. Ukkonen's Suffix Tree Algorithm



Suffixes of the word „baraba": trie, suffix tree, ST with a dollar

Observation: Consider a suffix tree for a word σ. Nodes of the tree (including hidden nodes) correspond to prefixes of suffixes of σ, that is the subwords of σ. Leaves of the tree are suffixes, which occur nowhere else in σ — these are called the *non-nested* suffixes. Internal nodes correspond to *branching subwords* — subwords $\alpha \subseteq \sigma$ such that $\alpha a \subset \sigma$ and $\alpha b \subseteq \sigma$ for two different symbols a and b.

Ukkonen's algorithm

Ukkonen described an algorithm [1] for constructing the suffix tree without the terminating dollar. It is an incremental algorithm. It starts with a tree for an empty word and it gradually appends new characters to the word, while updating the suffix tree. Each character is added in amortized constant time. Therefore, it constructs the ST for a word σ in time $\mathcal{O}(|\sigma|)$.

We will assume that edges to children of a single node can be indexed by their initial symbols — this certainly holds if the alphabet is fixed and small. In case it is not, we can use hash tables instead of arrays.

Observation: When we extend a word σ to σa, ST changes as follows:

1. All existing nodes of the tree (including hidden ones) correspond to subwords of σ. These are also subwords of σa, so they occur as nodes of the new tree, too.
2. If β was a branching subword, it stays branching — so internal nodes stay such.
3. Each new suffix βa is obtained by extending some original suffix β. Here:
    - If β was non-nested (so a leaf), βa will be also non-nested. So leaves stay leaves, but the labels of their edges must be extended by a. To make this efficient, we introduce *open edges,* whose labels index σ from a given position to the end. So leaves will take care of themselves.
    - If β was a nested suffix (i.e., an internal or hidden node), then:
        - Either βa is present in σ. Then it is a nested suffix of the new word and the tree needs no change;

- or βa does not occur in σ. Then we have to create a new leaf with an open edge and possibly a new internal node, under which the new leave will be connected.

This describes how the tree changes when a new symbol is appended to σ. It remains to show how to do it efficiently.

Nested suffixes: First of all, we need to recognize which suffixes are nested and which are not. It helps that nested suffixes form an interval:

> **Lemma:** If α is a nested suffix of a word σ and β is a suffix of α, then β is also a nested suffix of σ.
>
> *Proof:* The word σ contains both αx and αy for some distinct symbols x and y. Since α ends with β, there must be both βx and βy somewhere in σ, so β is nested. ♡

It therefore suffices to maintain the *longest nested suffix* of the word σ. We will all it the *active suffix* and denote it by $\alpha(\sigma)$. Each suffix $\beta \subseteq \sigma$ is nested if and only if $|\beta| \leq |\alpha(\sigma)|$.

The active suffix demarcates a boundary between non-nested and nested suffixes. How does this boundary move when we extend σ to σa? The answer is simple:

> **Lemma:** For every σ and a: $\alpha(\sigma a)$ is a suffix of $\alpha(\sigma)a$.
>
> *Proof:* Both $\alpha(\sigma a)$ and $\alpha(\sigma)a$ are suffixes of σa, so it suffices to compare their lengths. The word $\beta :=$ „$\alpha(\sigma a)$ without the final a“ is a nested suffix of σ, so $|\beta| \leq |\alpha(\sigma)|$, which implies $|\alpha(\sigma a)| = |\beta a| \leq |\alpha(\sigma)a|$. ♡

In other words, the boundary can only move to the right or stay as it was. This leads to the following idea.

Sketch of algorithm: We maintain $\alpha = \alpha(\sigma)$ and when appending a new character a, we check if αa stays nested. If it does, nothing changes. If it doesn't, we add a new leaf and possibly also a new internal node; then we remove the first character of α and continue checking.

Observation: Appending a character to σ causes an amortized constant number of tree modifications. This holds because every modification shortens α, but α grows only by a single character per append to σ. It remains to show how to perform each modification in (amortized) constant time. In order to accomplish that, we need a suitable representation of the word α, which supports efficient appends, cuts of the first character, and tests of existence of the corresponding node in the tree.

Reference pairs

Definition: A *reference pair* for a word $\alpha \subseteq \sigma$ is a pair (π, τ), where π is a tree node, τ an arbitrary word and $\pi\tau = \alpha$. Furthermore, we know that $\tau \subseteq \sigma$, so τ can be represented by a pair of indices in σ.

A reference pair is *canonical,* if there is no edge from the node π with a label, which is a prefix of τ. (Please note that such an edge can be found by just comparing

the first character of its label with $\tau[0]$ and checking if the length of the label is at most $|\tau|$. It is not necessary to check if the rest of the label matches $\tau[1:]$.)

Observation: Each word $\alpha \subseteq \sigma$ has exactly one canonical reference pair representing it. (Among all reference pairs representing α, it is the one with the deepest possible node π.)

Definition: The *back edge* $back(\pi)$ leads from an internal node π to an internal node, which represents $\pi[1:]$. (Let us observe that such a node must exist: every suffix of a branching subword is also branching.)

Operations: We are going to represent the active suffix α by a reference pair (π, τ). We need to perform the following operations on the pair:

- *Append a symbol a:* We append a to τ. We obtain a reference pair for αa, but it is not necessarily canonical. Before appending, we can easily check if α is present in the tree.
- *Removing the first character:* If π is not the tree root, we replace π by $back(\pi)$ and keep τ. Otherwise π is the empty string, so we remove the first character from τ (we just increment the corresponding start index).
- *Canonicalize:* The previous two operations can produce a non-canonical pair. So we check if the pair is canonical: for $\tau \neq \varepsilon$, we test if there is an edge from π indexed by $\tau[0]$ is short enough to be a prefix of τ. If it is, we follow this edge, which makes π longer and τ shorter; then we repeat the test. Let us see that this takes amortized constant time, because every step of canonicalization shortens τ, but τ grows only by a single character per append operation.

Now, all building blocks are ready and we can formulate the complete algorithm.

The full algorithm

1. *Input:* $\alpha = \alpha(\sigma)$ represented by a canonical reference pair (π, τ), the suffix tree T for σ, back edges *back*, and a new symbol a.
2. We check if αa is present in the tree, and create it if needed:
3. If $\tau = \varepsilon$: ($\alpha = \pi$ is an internal node)
4. If there is an edge from the node π whose label starts with a, then αa is present.
5. If there is no such edge, it isn't present, so we create a new open edge going from π to a new leaf.
6. Otherwise $\tau \neq \varepsilon$: (α is a hidden node)
7. We find an edge from π whose label starts with τ (by checking $\tau[0]$).
8. If τ in the label is followed by a, then αa is present.
9. Otherwise, it isn't present, so we subdivide the edge by creating a new node such that the edge from π to the new node is labeled by τ. The new node will have an open edge to a new child.

10. If αa was not present, we remove the first character of α and restart from step 2.
11. Now, we know that αa is present, so we update the reference pair to represent αa.
12. We recalculate back edges (see below).
13. *Output:* $\alpha = \alpha(\sigma a)$ as a canonical reference pair (π, τ), the suffix tree T for σa, back edges *back*.

Back edges: It remains to show how to make back edges of new internal nodes. We can observe that if we create a node, this node corresponds to the current α and its back edge goes to $\alpha[1:]$ — this is the node that we create (or discover that it already exists) in the next iteration of the main loop. During the next iteration, this back edge will not be needed yet, because $|\pi|$ is always decreasing. Therefore we can delay creation of back edges by one iteration. This way, we create back edges in constant time per node.

Literatura

[1] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.