UNIVERSITY OF LIVERPOOL

Department of Computer Science
Sebastian Wild

Date: 2021-04-14
Version: 2021-04-14 12:55

Tutorial 6 for COMP 526 – Applied Algorithmics, Spring 2021

Problem 1 (Suffix trees and friends)

Consider the text $T =$ `abbabbaa$`.

What is n here? (exactly follow the convention from the lecture!)

Construct/Draw the

1. standard (not compacted) trie of all suffixes of T,
2. suffix tree of T (human version) with string labels on edges and leaves,
3. suffix tree of T (computer version) as it is stored, i.e., offsets in nodes, starting index in leaves, first characters on edges,
4. suffix array $L[0..n]$ of T,
5. the inverse suffix array $R[0..n]$, and
6. the LCP array.
7. Annotate the internal nodes in the suffix tree with their string depth. Explain the connection between string depths and the LCP array.
8. Use the above structures to find the longest repeated substring in T.

Problem 2 (Suffix links)

In this exercise, we extend suffix trees by so-called *suffix links*. These allow further efficient algorithms on (generalized) suffix trees and are based on the following theorem.

Theorem SL: Let $\mathcal{T}$ be the suffix tree for a string $T[0..n)$. Let further v be an internal node of $\mathcal{T}$, i.e., a node with at least 2 children, which is reached by traversing along the string $w = w[0..m)$ in $\mathcal{T}$. If $m \geq 1$, then traversing along $w[1..m)$ in $\mathcal{T}$ also leads to an internal node v in $\mathcal{T}$.

Prove Theorem SL.

Why care for Theorem SL? The usefulness of this result lies in the ability to store at each internal node w a pointer to the corresponding internal node v; these pointers are called suffix links.

These can be used as another efficient string matching algorithm, using the suffix tree for the *pattern* $P[0..m)$. Conceptually, we traverse this pattern suffix tree with each *text* suffix $T[i..n)$ and check if we reach the leaf for the entire pattern $P[0..m)$.

Implemented naively like this, it would correspond to just the brute-force string matching, but we can instead use suffix links as a shortcuts: We traverse with the text suffix $T[i..n)$ in the pattern suffix tree, following links as long as they match the next text character. Upon a mismatch, we use one suffix link to take use directly to the internal node where we would have ended when traversing with $T[i+1..n)$ right away. If the mismatch happens "inside" an edge, we use the suffix link of the parent and re-traverse from there.

An amortized analysis shows that we spend constant time per text suffix on average, even though we only have suffix links for internal nodes and occasionally might have to re-traverse a long path after following a suffix link: Whenever we traverse down, we increase the string depth of our current position (the internal node we are at). We only ever decrease that depth when following a suffix link, and by exactly 1 only; we do that n times. Since the depth is always between 0 and m, the total number of re-traversals can only exceed the total number of suffix links followed by m.