Department of Mathematics and Computer Science
University of Southern Denmark

January 24, 2017
RF

Suffix Trees and Suffix Arrays.

One specific set S of strings to store is all the suffixes of a base string X. That is, $S = \{X[i..(n-1)] \mid 0 \leq i \leq n\}$, where $n = |X|$. If we run the methods from the note on searching in a sorted set, we can in $O(|x| + \log n + |R|)$ find the set R of all suffixes in X which has the pattern x as a prefix. Since

$$\begin{array}{c} x \text{ occurs as a pattern at position } i \text{ in } X \\ \Updownarrow \\ x \text{ is a prefix of the suffix } X[i..(n-1)] \end{array}$$

we have an efficient way of doing exact pattern matching in X. Note that here, the string X is preprocessed, and can answer queries for many patterns. In the KMP and BM algorithms, the pattern x is preprocessed, and could conceptually be used with many strings X without repeating the preprocessing (although this matters less there, as the prepocessing takes $O(|x|)$ time, which is smaller than the subsequent search time).

The array $S[]$ of the sorted suffixes is called the *suffix array* of X. It uses space $O(n)$, as does the array with $\text{lcp}(k)$ values. This is important here, since the set S of strings is efficiently represented by just the string X (and $S[]$ simply contains the starting indices in X of the suffixes, referenced in their lexicographically sorted order). This is just n characters, not $\Theta(n^2)$, which would be the combined length of the suffixes if they were stored explicitly and separately.

How do we generate the sorted array for S? Even with a size $O(1)$ alphabet, using MSD Radixsort could take $\Theta(n^2)$ time, as the total length L of the strings sorted is $\Theta(n^2)$. However, the strings have lots of overlap, which conceivably could be exploited. Indeed, a landmark result by Weiner from 1973 showed that suffix arrays can be built in $O(n)$ time (for $O(1)$ alphabets). The original algorithm is complex. Subsequent work by McCreight (1976), Ukkonen (1995) and Farach (1997) provided alternative algorithms with additional features, but none of these are simple, either. Recent work

has given many further alternatives, of which we later in this course cover the algorithm by Kärkkäinen and Sanders (2003), as this is particularly clean and simple. Its running time is $O(n)$ plus the time to sort the characters in the string (which is $O(n)$ for alphabets polynomial in n, by LSD Radixsort, and $O(n \log n)$ for comparison based alphabets, by e.g. Mergesort). Also the array LCP of $\mathrm{lcp}(k)$ values (needed for the search time stated above) can be found in $O(n)$ time in the suffix array setting. We cover later an algorithm for this by Kasai, Lee, Arimura, Arikawa, and Park (2001).

Actually, the 1970-90 work in this area did not deal with the suffix array as such, but rather with the trie built on S. In its plain form, this structure can have $\Theta(n^2)$ nodes. However, if we collapse all paths of unary nodes to single tree edges, all internal nodes of the resulting tree will be at least binary. The (possibly many) characters of such a path constitute a consecutive segment of X, so these characters can be represented by the starting and ending index of the segment (assuming X is stored with the trie), which is $O(1)$ space per edge. As there are at most n leaves, this means that there are at most $n-1$ internal nodes (the value if all internal nodes are binary). A tree of $2n-1$ nodes has $2n-2$ edges, so the entire structure (including X) now uses $O(n)$ space. Such a trie is called a *compressed trie*. Often, we end the string X in a character `$` smaller than any other character. Then no suffix is a prefix of another suffix, meaning that all the suffixes in S end in a leaf in the trie. Hence the trie has exactly n leaves, which are in 1-1 correspondence with the suffixes of X—a normalization which is often convenient. Such a compressed trie built on the suffixes of $X\$$ is called the *suffix tree* for X.

The suffix tree is a very versatile structure, which allows a surprising number of structural questions and search questions on the string X to be answered efficiently. However, being a tree with explicit nodes, the exact space usage in terms of bytes is often kn for $k \approx 40$, whereas the suffix array has $k \approx 4$. Since the string itself has $k \approx 1$, the size of the data structure may be the limiting factor for large strings (of, say, DNA). Surprisingly, the idea of using the simple suffix array was not introduced until around 1990 by Manber and Meyers (and, independently, Gonnet).

In terms of construction (and disregarding the space usage during this), the suffix array and the suffix tree methods are equivalent, though:

Lemma 1 *From a suffix tree for X we can in $O(n)$ time construct the suffix array and the LCP array for X. Conversely, from the suffix array and the LCP array for X we can in $O(n)$ time construct suffix tree for X.*

Proof: The first conversion can be performed during a DFS-traversal of the suffix tree: By the annotation of the edges, we can maintain the depth (distance from the root) in terms of characters during the traversal, and from this and the annotation of the edge above a leaf, we can find the starting index of the suffix of represented by the leaf. The leaves will be met in lexicographical order (if we recurse on the edges below an internal node in the order given by their first character). Hence, the suffix array can be constructed during the DFS traversal. When we meet leaf number k (except for the first), we will know $\mathrm{lcs}(k-1)$ if we always store the depth in terms of characters of the most recently visited internal node where the DFS-traversal changed direction from "coming up" (finishing the recursion on one child of the node) to "going down" (starting the recursion on the next child of the node). Hence, the LCP array can be constructed during the DFS traversal, which takes (n) time.

The second conversion can be done by adding the leaves of the compressed trie in increasing order (available via the suffix array). All insertions work on the right-most path of the current trie. An insertion if necessary climbs upward on this path until the depth (in characters) of the next LCP value is met. At this point a new edge with the new right-most leaf should be inserted. This depth may be at an existing internal node, in which case the new edge is put below that node. Else this depth is in the middle of an existing edge in the compressed trie. In the latter case, this edge is "broken" and one new internal node is inserted at the breakpoint, a node which has two edges below it (the lower part of the existing edge, and the new edge). There are n insertions, each of which introduces at most one new internal node on the right-most path. We may in insertion k move upwards past t_k internal nodes on the right-most path. However, these internal nodes then leave the right-most path, so the total climbing work $\sum t_k$ cannot be more than the total number of internal nodes created, which is n. Besides the climbing, each insertion takes $O(1)$ time. Hence, the total time is $O(n)$. $\square$