Building Suffix Links

April 4, 2016

1 Introduction

We show how to build the suffix links in a suffix tree in linear time, that is, we give a direct proof of [1, Theorem 8.18] avoiding the use of [1, Lemma 8.17] whose proof has a flaw although the lemma itself is correct[1].

We assume that the tree has been built on a text of length n over the integer alphabet $[1..\sigma]$ and denote the maximal string depth (path length) of the internal nodes in the tree by d (d can typically be much smaller than n). We recall that the path of a node v is the concatenation of all edge labels from the root to v (this is denoted $\ell(v)$ in [1]). The algorithm we present will only use arrays of size σ and d. The numbers manipulated by the algorithm are in the ranges $[1..\sigma]$ and $[1..d]$. Everything else will be lists and pointers. The main property of the suffix tree we will use is the following:

Lemma 1 *An internal node v in the suffix tree will have a Weiner link labelled by character c if and only if at least one of its children has a Weiner link labelled by c and it will have an explicit Weiner link labelled by c if and only if it has at least two children with Weiner links labelled by c.*

Proof Recall that an internal node v is labelled by right maximal string P and has an implicit Weiner link with character c if cP occurs in the text. This is only possible if cPa appears in the text for some character a, which means that a child of v labelled by a has a Weiner link labelled by c. This proves the first case of the claim. For the second case, an explicit Weiner link with character c will exist if and only if cP appears in the text and is right maximal, implying the existence of a node v' labelled by string cP. This implies that the text contains the substrings cPa and cPb for two distinct characters a and b which can only happen if two edges from v have labels starting with a and b respectively and their destination nodes have Weiner link labelled by c.

2 The Algorithm

We first give an overview of the algorithm before entering into details. We recall that an explicit Weiner link is the reverse of a suffix link. Our algorithm will produce explicit Weiner links and then reverse them to induce suffix links. The algorithm derives from two observations. The first observation is that the nodes at string depth $i+1$ will have suffix links to nodes at string depth i. Suppose that we order the nodes with the same string depth in left-to-right order (equivalently in

[1] `http://www.genome-scale.info/errata.html`

lexicographic order of their paths). Our second observation is that suffix links from nodes labelled by paths prefixed with the same character will preserve the order.

This suggests the following algorithm: first build for each string depth the ordered list of nodes with that string depth, and build for every node the list of characters that label its explicit Weiner links. The latter is easily done by making use of lemma proved above. Second, build the suffix links that start from nodes of string depth $i+1$, by traversing the lists of ordered nodes of string depth $i+1$ and i. This is done by first clustering the first list according to first characters of the paths (building a sublist c for each character c) and then traversing the second list, and for each node, determining the target of its explicit Weiner links by looking at the list of their labels: the target of the explicit Weiner link labelled by c is determined as the first element in sublist c and is then removed from the sublist. We now describe the details of the algorithm.

2.1 First Phase

In this phase the suffix tree is traversed in top-down left-to-right order. It is assumed that the string depth (path lengths) of traversed nodes are known during the traversal. If they are not stored in the suffix tree structure, we can compute them on the fly by keeping the string depth in the stack (of depth at most d) used to traverse the suffix tree. That is the string depth of a node v can be induced (during the traversal) from the string depth of its parent v' by adding the length of the label of the edge that connects v to v' with the string depth of v'.

The output of this phase consists in two elements. First element is an array $D[1..d]$ of d lists, where d is the maximal string depth. An item in each list is a pair $(\texttt{node}, \texttt{char})$. The second element of the output is an association of a list of characters labelling the explicit Weiner links starting from each node of the tree (the details of this association is given below). We now describe the construction of these outputs. While traversing the tree, when a node v has a path of length i and (the path) is prefixed by character c (the node is in the subtree of the root's child whose label start by character c), simply add the pair (v, c) at end of list $D[i]$. The list of characters that label explicit Weiner links from each node is built easily by merging the lists of characters that label all Weiner links from its children. The construction algorithm maintains two lists of Weiner link characters for every node (one list for explicit Weiner links and the other for all Weiner links). The merging is done for each node when the node is traversed the second time. It also uses two temporary structures, a stack that stores a set of distinct characters and an array $C[1..\sigma]$ of counters in $[1..\sigma]$ that associates a count to each character. Initially the stack is empty and all counters in C are set to zero. The lists for leaves are easily computed: a leaf representing a suffix preceded by character c will have its two lists containing the single element c. For an internal node, the two lists of (resp. explicit and all) Weiner link characters are built from the lists of (all) Weiner link characters of its children as follows: for each child traverse the list of its Weiner link characters and for each character c increment $C[c]$ and check if $C[c] = 1$ ($C[c]$ was equal to zero before incrementation). If that is the case, add c to the stack. At the end the list of (all) Weiner link characters is the list of characters contained in the stack and the list of explicit Weiner link characters is the list of characters in the stack for which $C[c] > 1$. At the end the stack is restored to the empty state and counters in C are reset to zero by traversing the list of Weiner link characters and resetting $C[c] = 0$ for each traversed character c. The correctness of the computation follows from Lemma 1.

2.2 Second Phase

The second phase traverses the array D level by level. At step i, we take the list $L = D[i+1]$ (which is a list of nodes at depth $i+1$ sorted in lexicographic order of their paths) and split its elements into α arrays, where α is the number of distinct characters that appear in the list L (as second component of the pairs). That is, produce a sparse array of sublists $L'[1..\sigma]$ from $D[i+1]$, where only α positions point to sublists of nodes at depth $i+1$ sorted in lexicographic order (α is the number of distinct characters that appear in first positions of these nodes path's) The sublists are produced as follows: initially all sublists are empty (all positions in L' are set to null pointer). Then the list L is traversed, and for each pair (v, c), the node v is added to the sublist $L'[c]$. Now traverse the list $L = D[i]$ (which is also sorted in lexicographic order), and for each node (first component of a pair) in the list determine the explicit Weiner link as follows: if the explicit Weiner link has label c, simply take its target as the node v, the head of sublist $L'[c]$ and then remove v from the sublist. Finally the suffix links can be induced by just reversing the directions of the produced explicit Weiner links. The correctness of the algorithm follows from the fact that the target nodes of the suffix links will have the same order as the source nodes as long as the path labels of the latter start with the same character.

2.3 Analysis

The linear running time of the algorithm follows from the fact that the number of Weiner links is linear [1, Observation 8.14].

References

[1] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, 2015.