Sorting leaf-lists in a tree

Christian N. S. Pedersen* Jens Stoye†

1 Introduction

Let T be a rooted tree of size n with leaves labelled with distinct elements from an ordered set. We describe a simple method to obtain the sorted leaf-lists of all nodes one at a time in a depth-first fashion in $O(n \log n)$ time and $O(n)$ space. Applying the method to a suffix tree can be used to find various kinds of repeats in a string.

2 Definitions

The following definition divides the nodes in T into small and big nodes.

Definition 1 *Consider a node with one or more children. The child that roots the biggest subtree is said to be a big node (or a big child), the other children are said to be small nodes (or small children). Ties are broken arbitrarily. The root is defined to be a small node.*

The fact that each node has exactly one big child motivates the following definition of a big path.

Definition 2 *A big path starts at a small node and ends at a leaf such that all nodes along the path, except the small node starting the path, are big nodes.*

Note that each small node starts exactly one big path, and that each node belongs to exactly one big path.

3 Sorting method

Let $v_1 \to v_2 \to \cdots \to v_k$ be a big path. If we know the sorted leaf-list at the small node v_1 then we can get the sorted leaf-list at the big nodes $v_2, v_3, \ldots, v_k$ one by one by the following procedure.

1. *Marking:* Traverse all subtrees rooted by small children of the nodes along the big path. If a leaf with label i is in the subtree rooted by small child u then mark element i in the sorted leaf-list at v_1 as going to node u.

2. *Copying:* Scan the sorted and marked leaf-list at v_1 in sorted order. Append each marked element to the leaf-list at the node it is marked with. This step creates the sorted leaf-lists at all small nodes hanging off the big path.

3. *Pruning:* For each $i = 2, 3, \ldots, k$ the sorted leaf-list at v_i is obtained by removing all elements that belong to a leaf-list at a small child of v_{i-1}. This step creates the sorted leaf-lists for the nodes along the big path one at a time.

Since each node belongs to exactly one big path, we can get the sorted leaf-lists at each node one at a time by running the above procedure for all big paths. To do this we must start with the big path starting at the root. The sorted leaf-list at the root is either known or can be obtained in time $O(n \log n)$. After having processed this big path we can continue with any big path starting at a small node at which the sorted leaf-list is known. Since Step 2 creates the sorted leaf-lists at all small nodes hanging off the big path being processed, and since each small node is hanging off some big path, we will at some point know the sorted leaf-list at any small node. Hence, we are able to run the above procedure for all big paths.

The running time is easy to analyze. Each of the three steps takes time proportional to the size of the subtree rooted at the small node starting the big path. Hence, running the procedure along all big paths takes time proportional to the sum of the sizes of all subtrees rooted by small nodes. It is well known that this sum is $O(n \log n)$. To analyze the required space, we observe that any element at any time is in at most two sorted leaf-lists. This observation follows

*BRICS, Department of Computer Science, University of Aarhus, Denmark. E-mail: cstorm@brics.dk. Work done while visiting University of California at Davis.

†Department of Computer Science, University of California at Davis. E-mail: stoye@cs.ucdavis.edu. Supported by the German Academic Exchange Service (DAAD).

because the elements we copy in Step 2 are removed as the leaf-list is pruned along the big path in Step 3. Hence, running the procedure along all big paths requires $O(n)$ space.

4 Applications

Crochemore [1] describes how to report all primitive tandem repeats in a string in time $O(n \log n)$ using a partitioning of the string positions. The partitioning does not use the suffix tree of the string but if explained in terms of the suffix tree the partitioning essentially constructs the sorted leaf-lists one at a time in a breadth-first fashion. Our sorting method constructs the sorted leaf-lists one at a time in a depth-first fashion. All that is needed in order to report all primitive tandem repeats is simple bookkeeping on the sorted leaf-lists. The sorting method (along with the suffix tree) can thus be used to report all primitive tandem repeats in a string in time $O(n \log n)$.

We will now shortly describe how the sorting method (along with a suffix tree) also can be used to detect various other kinds of repeats in a string.

Definition 3 *We say that (p, q, α) is a pair in a string S if $\alpha = S[p..p + |\alpha| - 1] = S[q..q + |\alpha| - 1]$ and $p < q$. The pair is left-maximal (right-maximal) if the characters to the immediate left (right) of two occurrences of α are different. It is maximal if it is right- and left-maximal. The gap of a pair (p, q, α) is the number of characters $q - p - |\alpha|$ between the two occurrences of the substring α. If the gap is non-negative then the pair is non-overlapping.*

Gusfield [2, Sect. 7.12.3] describes how to report all maximal pairs in a string using the suffix tree of the string. We will describe how to report all non-overlapping maximal pairs as well as all maximal pairs with a gap at most some constant c in time $O(n \log n + z)$, where z is the number of reported pairs. We will only describe how to find right-maximal pairs. Maximal pairs can be found by an extension similarly to that in [2, Sect. 7.12.3].

Observe that (p, q, α) is a right-maximal pair if and only if p and q are leaves in subtrees rooted by different children of the node with path-label α. To report all non-overlapping right-maximal pairs (p, q, α) where α is the path-label of node v_{i-1} we only have to extend Step 3 in the sorting procedure.

Let $L_1, L_2, \ldots, L_r$ be the sorted leaf-lists at the small children of v_{i-1}. In Step 3 we construct the sorted leaf-list at v_i by removing all elements in these lists from the sorted leaf-list at v_{i-1}. Let $\{e_1, e_2, \ldots, e_l\}$ be what remains of the sorted leaf-list at v_{i-1} after having removed all elements in $L_1, L_2, \ldots, L_j$ for some $j \leq r$. Using this partially pruned leaf-list we can report all non-overlapping pairs where one of the occurrences is in L_j.

For each p in L_j we report the pairs (e_1, p, α), $(e_2, p, \alpha), \ldots, (e_s, p, \alpha)$, where e_s is the maximum element such that (e_s, p, α) is non-overlapping, and we report the pairs $(p, e_l, \alpha), (p, e_{l-1}, \alpha), \ldots, (p, e_{l-t}, \alpha)$, where e_{l-t} is the minimum element such that (p, e_{l-t}, α) is non-overlapping. It is easy to see that running this procedure for each L_j will report all non-overlapping pairs of α (the path-label of v_{i-1}) and that the running time only increases with a term proportional to the number of reported pairs.

To report all pairs with gap at most c we can use almost the same extension of Step 3 as above. For each element p in L_j we first find the maximum element e_m in $\{e_1, e_2, \ldots, e_l\}$ which is less than p. We then report the pairs $(e_m, p, \alpha), (e_{m-1}, p, \alpha), \ldots, (e_{m-s}, p, \alpha)$, where e_{m-s} is the minimum element such that the gap of (e_{m-s}, p, α) is less than c, and we report the pairs $(p, e_{m+1}, \alpha), (p, e_{m+2}, \alpha), \ldots, (p, e_{m+t}, \alpha)$, where e_{m+t} is the maximum element such that the gap of (p, e_{m+t}, α) is less than c. The problem of finding e_m in constant time can be solved by some simple bookkeeping while pruning the sorted leaf-list along the big path.

References

[1] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.

[2] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambrigde University Press, 1997.