6.897: Advanced Data Structures Spring 2003

Lecture 11 — April 2, 2003

Prof. Erik Demaine *Scribes: Christos Kapoutsis, Loizos Michael*

1 Overview

Recall from last lecture that we are looking at the *document-retrieval* problem. The problem can be stated as follows:

> Given a set of texts $T_1, T_2, \ldots, T_k$ and a pattern P, determine the distinct texts in which the patterns occurs.

In particular, we are allowed to preprocess the texts in order to be able to answer the query faster. Our preprocessing choice was the use of a single suffix tree, in which all the suffixes of all the texts appear, each suffix ending with a distinct symbol that determines the text in which the suffix appears. In order to answer the query we reduced the problem to *range-min queries*, which in turn was reduced to the *least common ancestor* (LCA) problem on the *cartesian tree* of an array of numbers. The cartesian tree is constructed recursively by setting its root to be the minimum element of the array and recursively constructing its two subtrees using the left and right partitions of the array. The range-min query of an interval $[i, j]$ is then equivalent to finding the LCA of the two nodes of the cartesian tree that correspond to i and j.

In this lecture we continue to see how we can solve the LCA problem on any static tree. This will involve a reduction of the LCA problem back to the range-min query problem (!) and then a reduction of the latter problem to two special instances of the problem.

Recall that we need to solve the LCA problem as part of a number of reductions from the document-retrieval problem. However, we have an additional pendency from last lecture. We still need to show how the suffix tree that is used for the document-retrieval problem can be built, during the preprocessing phase, in time linear to the size of the involved texts. We will address this problem in the second part of this lecture.

2 Least Common Ancestor

The least common ancestor problem has been addressed (in some form or another) in many papers over the last twenty years (e.g. [5, 2, 1]). In this lecture we will present the solution suggested in the most recent work of Bender and Fanach-Colton [1]. The formal problem we are trying to address is:

> Preprocess a binary tree so that, given two nodes i and j of the tree, determine the node closest to the leaves that appears in both the paths of i and j towards the root.

We proceed to show how the LCA problem can be solved by reducing to the range-min query problem.

2.1 Reducing to Range-Min Queries

Having the tree at hand, the first thing to do, during preprocessing, is to label each node according to its depth in the tree, labelling the root with the number zero. We subsequently traverse all the nodes of the tree in an Euler tour to construct an array of the labels of the nodes, in the order visited by the tour. An Euler tour is simply a depth-first traversal of the tree with repeats. More precisely, nodes are visited when moving from parents to children and are revisited when moving from children to parents. Figure 1 illustrates the Euler tour on an example tree and the resulting array of numbers. Notice how the query on the nodes of the tree is reduced to a query in the resulting array of numbers.

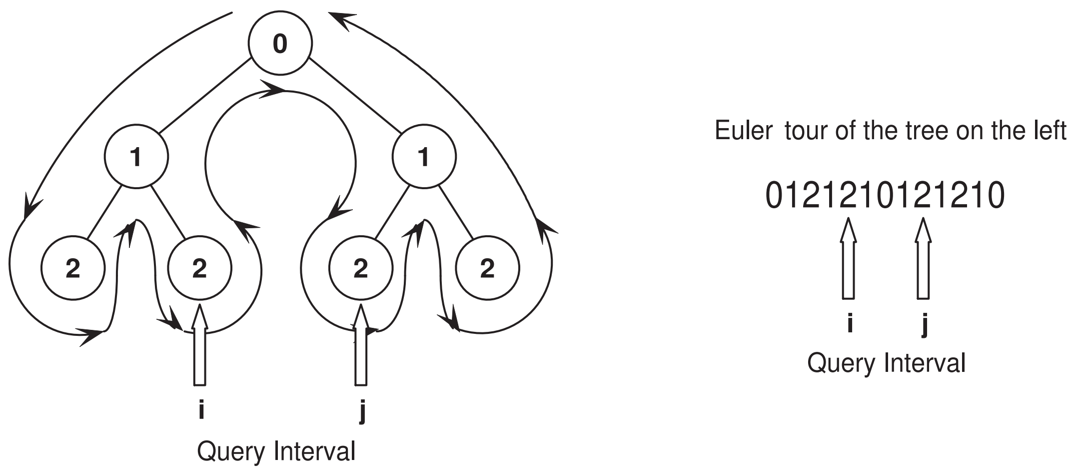


Figure 1: The Euler tour on an example tree and the resulting array of numbers.

Having constructed the array of numbers, we can make two observations. First notice that the LCA of two nodes is simply the node with the least depth (i.e. closest to the root), that lies between the two nodes in the Euler tour. Therefore, finding the specific node in the tree is equivalent to finding the minimum element in the proper interval in the array of numbers. The latter problem can be solved using range-min queries. But wouldn't that create a loop in our reduction, since we started by reducing the range-min query problem to the LCA problem? The answer is no, because of the second observation: Notice that the constructed array of numbers has a special property, known as the ± 1 *property*; each number differs by exactly one from its preceding number. Therefore, our reduction is to a special case of the range-min query problem, which can be solved without any further reductions.

Our goal, thus, is to solve the following problem:

> Preprocess an array of n numbers that satisfies the ± 1 property such that, given two indices i and j in the array, determine the index of the minimum element within the given range $[i, j]$, in $O(1)$ time and using $O(n)$ space.

2.2 Range-Min Queries with Superlinear Space

In this subsection we will see how we can address the problem in the special case where we have superlinear space available.

2.2.1 A Naïve Attempt

We start by examining a simple solution that will help us gain some insight into the problem. We will allow ourselves to use $O(m^2)$ space and constant time to answer the query for an array of size m.

The evident solution to this problem is to simply create a table with $O(m^2)$ entries, mapping all the intervals of the array to the corresponding minimum element of the interval. The query can then be answered with a single lookup into the table that takes constant time.

2.2.2 A Better Solution

Our first solution obtains the required constant time for answering queries, but suffers from the use of too much space. Fortunately, we can refine that result without much effort. We will still use a lookup table, but we will only store the answers for specific intervals of the array. In particular, we will store only intervals of length equal to some power of two. Starting from each position in the array of n numbers, there are at most $\lg m$ intervals of length equal to some power of two and thus in total, the table will contain only $O(m \lg m)$ entries. This indeed improves the space used, although seemingly not as much as we would like, as the space is still superlinear.

Now, in order to answer a range-min query for a given interval $[i, j]$ of length $l = j - i + 1$ we can simply query the table for the minima of the two intervals $[i, i + \llbracket l \rrbracket)$ and $(j - \llbracket l \rrbracket, j]$ and take the minimum of the two returned values.[1] Therefore, we have improved the space, while keeping a constant query time.

We keep this result and we proceed to examine another special case.

2.3 Range-Min Queries within All Small Intervals

In this subsection we will see how we can address the problem when our queries are restricted to a small interval of the array, in such a way that we can use our preprocessing to simultaneously solve all possible small intervals.

We consider an interval of the array of size k and assume that we are interested in answering queries that lie completely within this interval. Observe that the result of a range-min query within this interval is invariant to *translation*, i.e. offsetting all the numbers by some constant. Indeed, the index of the minimum element in the array will not change under translation. The first step, hence, is to translate the interval so that its first number becomes zero.

By the ± 1 property, there exist exactly 2^k possible instantiations of an interval of size k such that the first number is zero. To see this, observe that each number can be either larger or smaller of

[1] Recall that the notation $\llbracket l \rrbracket$ represents the maximum power of two that is at most l, i.e., $2^{\lfloor \lg l \rfloor}$.

the previous number by exactly one and thus we have exactly two choices for each number in the interval, beyond the first number.

Because we are interested in answering queries within this interval, then for each possible instantiation of the interval we can construct a lookup table as in Section 2.2.1, where each possible query is mapped to its minimum element.[2] Because there are 2^k possible instantiations, each of which has k^2 possible queries, for which we need to store an index that requires space $\lg k$, we have in total a requirement of $O(2^k \cdot k^2 \cdot \lg k)$ bits of space. This space is reasonable provided k is small.

2.4 The Final Construction

We are now ready to use our earlier results and address our original problem of answering a range-min query in $O(n)$ space and constant time.

Consider the array A of n numbers that is given as input to this problem. We divide the array into $m = 2n/\lg n$ buckets, each of size $k = (\lg n)/2$. During preprocessing we compute the minimum element of each bucket and store the results in an array B of size m. Figure 2 illustrates the two arrays and the buckets. The construction described so far can be clearly done in space $O(n)$.

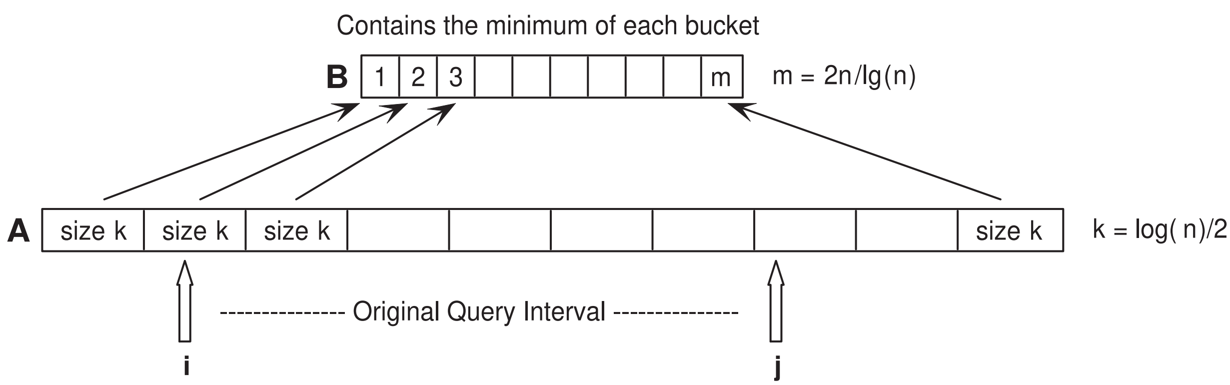


Figure 2: The partitioning of array A into buckets and the storage of the minimums of the buckets into array B.

We now construct a lookup table T_A for answering queries within a single bucket, using the construction described in Section 2.3. Notice that, because the size of a bucket is $k = (\lg n)/2$, this lookup table will require only $O(2^k \cdot k^2 \cdot \lg k) = O(2^{(\lg n)/2} \cdot ((\lg n)/2)^2 \cdot \lg((\lg n)/2)) = O(\sqrt{n} \cdot \lg^2 n \cdot \lg \lg n)$ bits of space, which is $o(n)$. Also notice that we can answer queries within any bucket using the same lookup table T_A, because it contains all the possible instantiations that can be found within the buckets. We simply need to store one word per bucket to indicate the input to the lookup table T_A.

Finally, we construct a lookup table T_B for answering queries within array B, using the construction described in Section 2.2.2. Notice that since the size of the array is $m = 2n/\lg n$, this lookup table will only require $O(m \lg m) = O((2n/\lg n)\lg(2n/\lg n)) = O((n/\lg n)(\lg n - \lg \lg n)) = O(n - (n \lg \lg n)/\lg n) = O(n)$ words of space.

[2]This naïve construction can be improved by using the construction of Section 2.2.2 to reduce the size of the table. However, these improvements will only offer a polylog space improvement that will not affect our final result.

In total, our construction involves two arrays A and B and two lookup tables T_A and T_B and requires $O(n)$ space. It remains to show how we can answer a query in constant time. A given interval $[i, j]$ will, in the general case, span many buckets, as illustrated in figure 3. Let L and R be the leftmost and rightmost buckets, respectively, that are **partially** included in the query interval and let X and Y be the leftmost and rightmost buckets, respectively, that are **completely** included in the query interval.

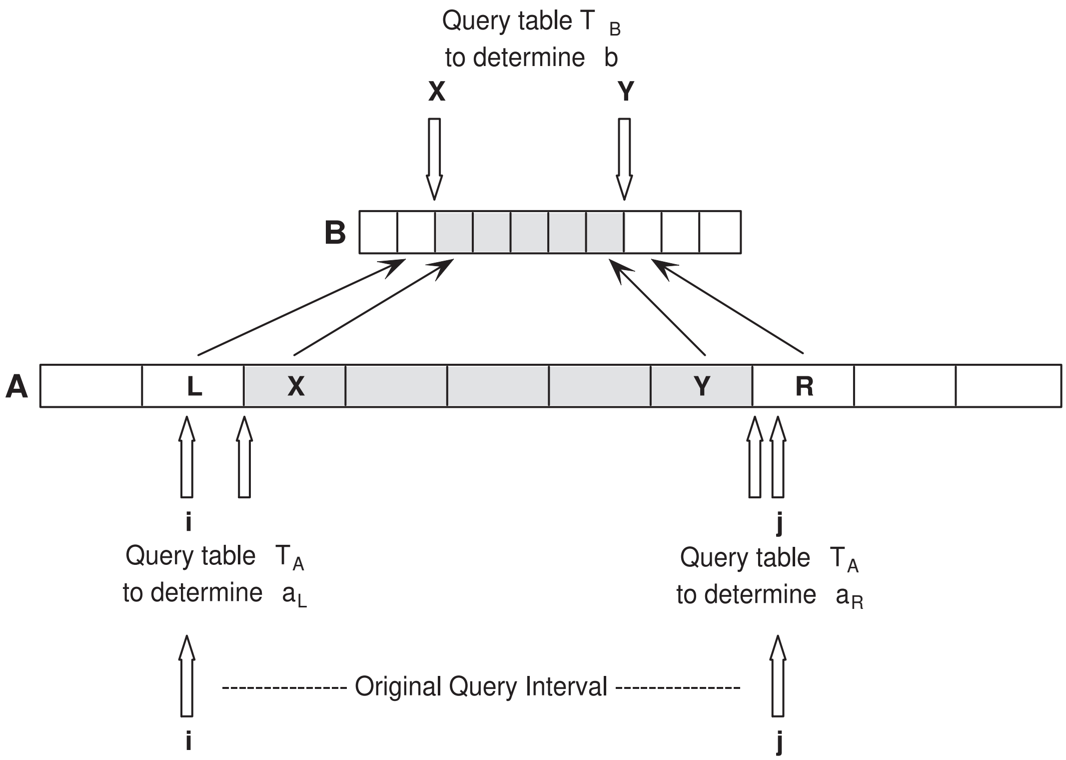
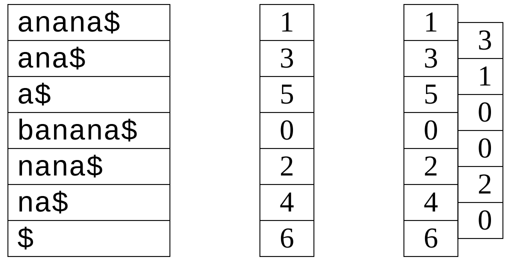


Figure 3: The reduction of the original query to three lookup table queries.

Let a_L be the minimum element of the L bucket that lies between i and the end of the bucket and let a_R be the minimum element of the R bucket that lies between the beginning of the bucket and j. These two values can be computed by two queries to the T_A table. Now let b be the minimum element of all the internal buckets that lie within $[X, Y]$. This value can be computed by computing the minimum amongst the minimums of each internal bucket, by querying the T_B lookup table for the minimum element of array B that lies within the interval $[X, Y]$. All three queries to the lookup tables take constant time, and we can compute $\min\{a_L, b, a_R\}$ in constant time, which is precisely the minimum element of array A that lies within $[i, j]$.

2.5 Conclusions

We have seen how to solve the range-min query problem in an array of n numbers that satisfies the ± 1 property, in $O(1)$ time and using $O(n)$ space. This implies a solution to the LCA problem in $O(1)$ time and $O(n)$ space, which subsequently implies a solution to the problems that we have seen in the last lecture, including general range-min queries and the document-retrieval problem.



(a) (b) (c)

Figure 4: (a) The suffix array of `banana$`. (b) The same array with suffixes represented implicitly, via their indices. (c) The augmented suffix array: each number on the right array is the LCP of the two prefixes that it touches on the left.

3 Building Suffix Trees in $O(n)$ Time

We now show how to quickly construct a single suffix tree containing the suffixes of all texts $T_1, T_2, \ldots, T_k$ — every suffix of T_i participates in this tree with a different end marker, $\$_i$. This section is based on [3, 4]. First we introduce a couple of extra notions related to suffix trees.

3.1 Suffix Array

Given a text T, the suffix array associated with it is just the lexicographically sorted array of all its suffixes. For our running example,

$$T = \overset{\texttt{0123456}}{\texttt{banana\$}},$$

the suffix array is that of Figure 4(a). To keep the space linear, we can represent each suffix implicitly, by its index in T, as in Figure 4(b). Notice that with a properly modified binary search on this array we can search T for a pattern P in time $O(|P| + \lg |T|)$, which is a little worse than the $O(|P|)$ time needed if the search is done using a suffix tree.

The suffix array can be easily derived from the suffix tree via a depth-first search, in linear time (Figure 5).

To make the translation possible the other way around, we need to augment the suffix array with extra information: for every two adjacent suffixes we store *the length of their longest common prefix* (LCP); see Figure 4(c). Now, to derive the suffix tree we basically build the cartesian tree of the array of LCP's, using the recursive procedure described earlier. Special care is taken so that a minimum that occurs more than once gives rise to only one node (with outdegree one greater than the number of occurrences). During the recursion, whenever a minimum is found on the left (right) end of the current subarray, we make its left (right) child be the first (second) suffix corresponding to this minimum LCP; see Figure 6.

Note that the LCP info in the augmented suffix array can also be derived from the suffix tree (during the same depth-first search that derives the suffix array itself), using the letter-depth information of the internal nodes. So, in total, *we can convert a suffix tree into the corresponding augmented*

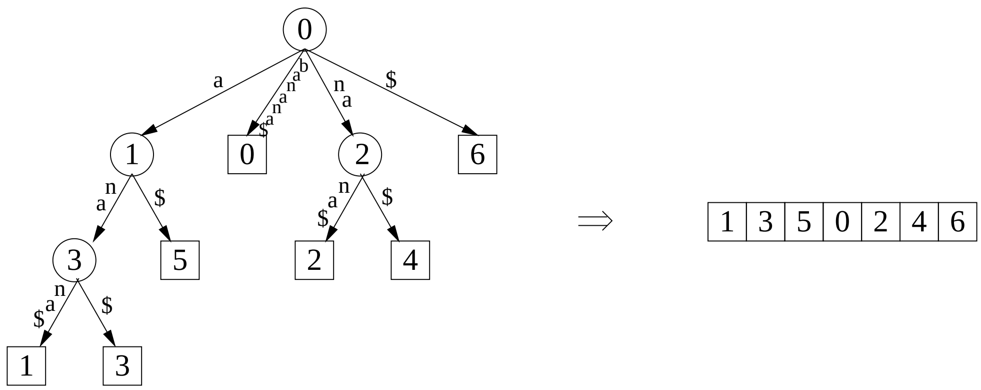


Figure 5: From the suffix tree to the suffix array.

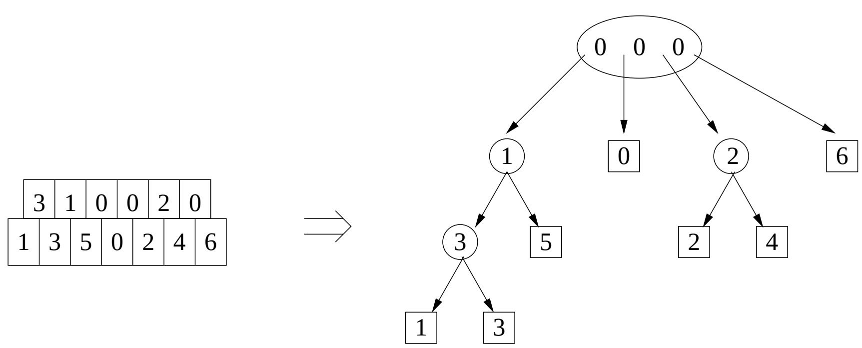


Figure 6: From the augmented suffix array to the suffix tree.

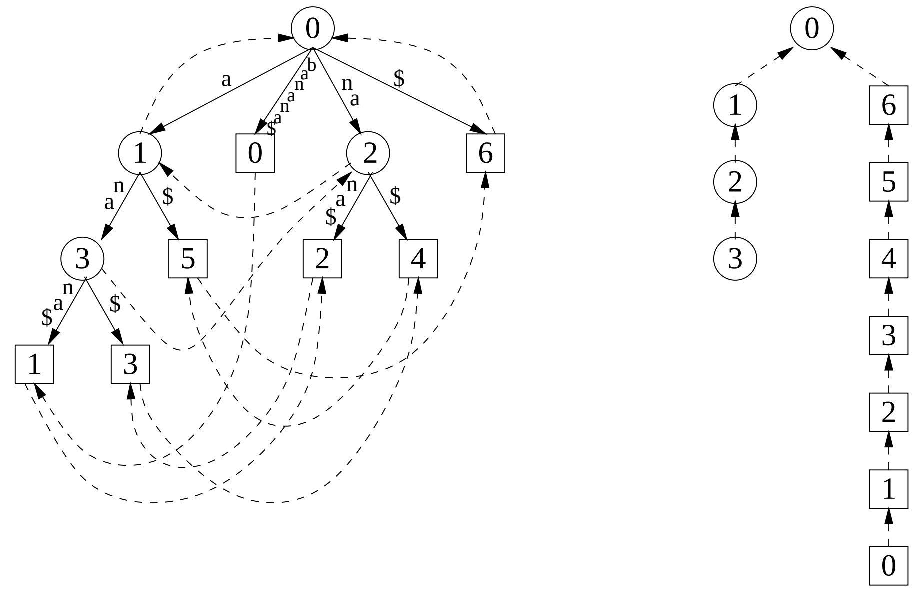


Figure 7: On the left, the suffix tree of `banana$`, with all suffix links as dashed arrows. On the right, the tree implied by the suffix links.

suffix array and vice-versa in linear time.

3.2 Suffix Links

Consider the suffix tree of a text T, fix any node u in it other than the root, and let ax be the nonempty string that appears on the edges of the path from the root down to u (i.e., the label of u). If i, j are two leaves that have u as their lowest common ancestor, then the corresponding suffixes $T[i:]$, $T[j:]$ of T have ax as their longest common prefix. Dropping the first character a from these two suffixes, we get the suffixes $T[i+1:]$, $T[j+1:]$, whose longest common prefix is just x. Therefore, the label of the lowest common ancestor v of the leaves $i+1$, $j+1$ is x. In total, starting from any node u with label ax, we can always find a unique node v with label x. We call it the *suffix link* of u (Figure 7).

Notice that, if we start at any node in the suffix tree and we repeatedly follow the suffix links, we will never repeat a node (because, each time we follow a link, the length of the label of the current node decreases by one) and we are bound to end up at the root (that's the only node with an empty label). Hence, suffix links form a tree on the nodes of the suffix tree; the depth of a node in this tree equals its letter depth in the suffix tree.

Also notice that, the suffix link of a node can be computed as described above (i.e., as the least common ancestor of $T[i+1:]$, $T[j+1:]$) in constant time. Thus, linear time suffices to compute the suffix links of all nodes in the tree. In particular, *if the letter-depth information for the internal nodes of a suffix tree is missing, linear time is enough to retrieve this information*: we just compute all suffix links, then depth-first search in the resulting tree, labeling every node with its depth.

3.3 Building the Suffix Tree of One Text

Let us now turn to the main subject of this section: how to build the suffix tree of a text quickly. In the comparison model, the running time will be $O(n \lg |\Sigma|)$, where Σ is the alphabet. This bound is optimal in the comparison model, even if the nodes of the suffix tree do not have to store their child links in sorted order: building a suffix tree in particular determines how many different letters of the alphabet occur (this is the number of children of the root), which requires $\Omega(n \lg |\Sigma|)$ time.[3] On the other hand, if the alphabet is $\{0, 1, \ldots, |\Sigma| - 1\}$ and hence can be sorted in linear time, the running time is $O(n)$. If we are allowed to hash the alphabet, the running time is $O(n + |\Sigma| \lg |\Sigma|)$ if we want each node's children to be in sorted order, and $O(n)$ time otherwise.

We will actually show how to construct the augmented suffix array of the input text in this time bound. The suffix tree itself can then be produced as explained in Section 3.1.

So, suppose T is our n-long input text over some alphabet Σ. We can think of T as a string over $\{1, 2, \ldots, n\}$; if this is not the case, we can just replace every symbol in T with its rank in Σ. With hashing, this costs linear time after Σ is sorted; without hashing, this costs $O(n \lg |\Sigma|)$ time.

The outline of the algorithm is as follows:

(i) From the n-long input T over $\{1, 2, \ldots, n\}$, we construct a $\frac{n}{2}$-long text T' over $\{1, 2, \ldots, \frac{n}{2}\}$.

(ii) We make a recursive call, to get the augmented suffix array A' of T'.

(iii) From A' and the relation between T and T', we construct the augmented suffix array A_e for the *even* suffixes of T (that is, the suffixes of T starting at positions $0, 2, 4, \ldots$).

(iv) From A_e we also derive the augmented suffix array A_o for the *odd* suffixes of T.

(v) We merge the two arrays, A_e and A_o, into the augmented suffix array A for *all* suffixes of T.

If $C(n)$ is the cost of our algorithm, the recursive call will cost time $C(\frac{n}{2})$ and all other steps will cost linear time. So, $C(n) = C(\frac{n}{2}) + O(n)$ for a linear total cost. Details follow.

i. Constructing T'. We first coalesce every symbol that lies at an even position in T with the symbol immediately to the right of it, considering the pair as a new, composite symbol. This way, T can be viewed as a $\frac{n}{2}$-long string over the alphabet $\{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$.

We then radix sort the list of $\frac{n}{2}$ composite symbols occuring in T. (To compare two composite symbols we just look at their lexicographic order as pairs of symbols of Σ.) In the sorted list, we remove all repetitions.

We finally go through T, replacing each composite symbol with its rank in the sorted list. The resulting string, T', is $\frac{n}{2}$-long with all its symbols in $\{1, 2, \ldots, \frac{n}{2}\}$.

ii. Getting the augmented suffix array of T'. We make a recursive call to the algorithm on T', to get the augmented suffix array A' of T'.

[3] Personal communication between Erik Demaine and Martin Farach-Colton, April 2003.

iii. Building A_e. We now obtain the augmented suffix array A_e for the *even* suffixes of T, by processing A'. In particular, we extract

- the sorted list of the *even* suffixes of T, and
- the LCP of every pair of adjacent (in this sorted list) *even* suffixes.

The first part is almost immediately given by the sorted list in A'. Just note that every suffix of T' corresponds to an even suffix of T; and this correspondence preserves the lexicographic ordering. However, we do need to double each entry, to have the indices point into the n-long T.

The second part can be derived almost as easily. The LCP of two adjacent even suffixes of T is either $2\times$(the LCP of the corresponding suffixes of T'), or $2\times$(the LCP of the corresponding suffixes of T')$+1$; depending on whether the two suffixes differ in the first (even-position) symbol after their longest even-length common prefix. Obviously, this check can be done in constant time.

Notice that a similar recursion could also give us the augmented suffix array A_o for the *odd* suffixes of T. However, we can't afford the time for this extra recursion. Instead, we compute A_o from A_e.

iv. Building A_o. Now we use A_e to construct the augmented suffix array A_o for the *odd* suffixes of T. Again, we extract

- the sorted list of the *odd* suffixes of T, and
- the LCP of every pair of adjacent (in this sorted list) *odd* suffixes.

For the first part, go through the sorted list of even suffixes that is contained in A_e and subtract 1 from each entry (ignore the entry 0, for the first even suffix). Now the list enumerates all odd suffixes sorted by their corresponding even suffixes, much like what we would have after a radix sort of the list that wouldn't have completed its last scan. So, to get the correct order we can just apply this last scan, a counting sort that on an entry i (for some odd i) uses $T[i]$ as the value that determines the new position of the entry.

For the second part, notice that if i, j are two odd suffixes adjacent in the sorted list, their LCP is 0, if $T[i] \neq T[j]$; otherwise, it's the minimum of all LCPs between the corresponding even suffixes $i+1$, $j+1$ in the list of A_e. In both cases we can compute the correct value in constant time, the second case requiring an appropriate range-min query on the list of LCPs of A_e.

v. Merging A_e and A_o. For the merging, we first translate A_e, A_o into the corresponding suffix trees, then merge the suffix trees into a single suffix tree, then translate back into an augmented suffix array, A. We know how to do the translations fast (Section 3.1), so we need only describe the merging of the two suffix trees.

Start by overlaying the roots of the two trees and try to merge them into a single root. The easy case occurs when all edges have labels starting with different symbols (Figure 8(a)). Then, we directly copy all subtrees under the new root. A slightly more interesting case occurs when some pair of edges have labels starting with the same symbol, but the labels remain identical all the way through. Again, all other subtrees are copied directly under the new root; the conflicting edge is copied, too, but its subtree is the one we get by recursively merging the corresponding subtrees (Figure 8(b)). The case when the labels are of different length but one of them is a prefix of the other can also be handled, similarly.

The hard case is when two labels start with the same symbol but differ later on. Then, we have to find out for how long they agree and act accordingly. Unfortunately, we can't afford the time to compare the two labels symbol by symbol ...

Let's **suppose we have an oracle** that can do this for us: tell, in constant time, whether two labels that start with the same symbol differ later on, or remain identical until the shorter one is exhausted. We show how to construct such an oracle later.

If the answer of the oracle is negative, we are in one of the last two cases above and we handle it accordingly. If the answer is positive, we know the labels don't match all the way through, but we don't know the extent to which they match. It turns out that this is not necessary at this point: we already know that an edge must be created under the new root, pointing to a new node v with two subtrees, the corresponding subtrees from the two original suffix trees (Figure 8(c)). The only thing we are missing is the letter depths of (some of) the newly added nodes. But this is okay, because we can retrieve this information after the tree is constructed, by computing the tree of the suffix links and calculating the depths in it (as explained in Section 3.2).

To implement the oracle that we have promised, we do the following. First, in a preprocessing phase, we calculate and store the fingerprint (in the Rabin-Karp sense) of every suffix $T[i:]$ of T. Then, the fingerprint of every substring $T[i:j]$ of T can be computed in constant time from the fingerprints of $T[i:]$, $T[j:]$. Hence, to check whether two labels that start with the same symbol remain identical, just trim the longest one to the length of the shortest one and compare the corresponding fingerprints.

Of course, this makes our algorithm randomized. However, it can be shown that the use of randomness can be avoided; in particular, there is a way to complete this last step (Step (v)) of the algorithm deterministically [3, 4].

3.4 Building the Suffix Tree of k Texts

Remember that our problem originally was the construction of the suffix tree of k texts $T_1, T_2, \ldots, T_k$ in time/space linear in the total size of the texts (modulo $\lg|\Sigma|$ terms). The previous section only shows how to do this for one text. If we can apply the algorithm to the concatenation of $T_1, T_2, \ldots, T_k$, then we obtain the suffixes of each of the T_i, but they continue on with $T_{i+1}, T_{i+2}, \ldots, T_k$. To fix this, we can trim the tree beyond any nodes labeled $\$_i$ for any i. The remaining tree will just have the suffixes of the individual T_i's.

References

[1] Michael A. Bender and Martin Farach-Colton, "The LCA Problem Revisited", in *Proceedings of LATIN 2000*, pages 88–94.

[2] Omer Berkman, Dany Breslauer, Zvi Galil, Baruch Schieber, and Uzi Vishkin, "Highly Parallelizable Problems (Extended Abstract)", in *Proceedings of STOC 1989*, pages 309–319.

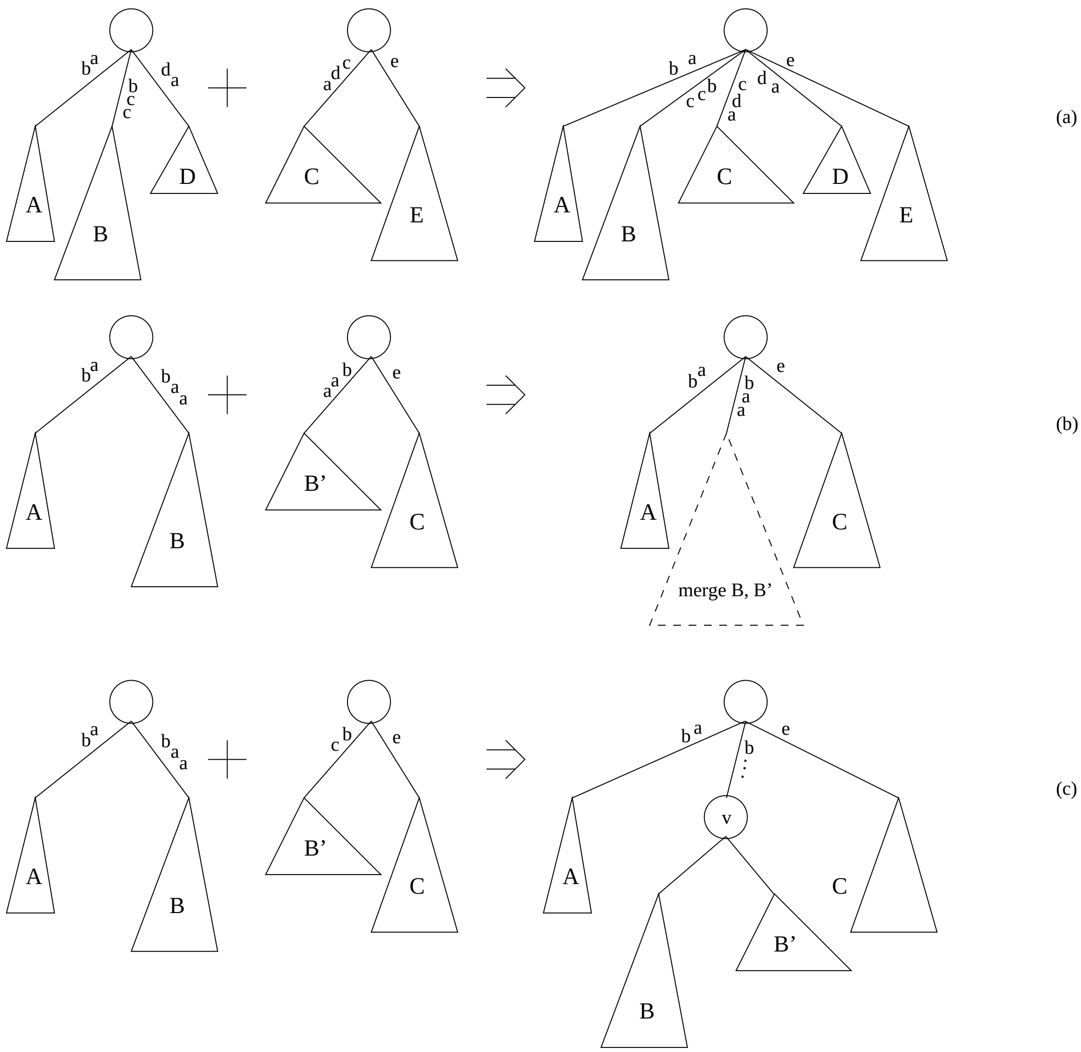


Figure 8: Merging the two suffix trees. (a) When all edge labels start with distinct symbols. (b) When every two edge labels that start with the same symbol continue identically as well. (c) When two edge labels start with the same symbol but differ later on.

[3] Martin Farach, "Optimal Suffix Tree Construction with Large Alphabets", in *Proceedings of FOCS 1997*, pages 137–143.

[4] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan, "On the sorting-complexity of suffix tree construction", In *Journal of the ACM* 47(6):987–1011, 2000.

[5] Dov Harel and Robert E. Tarjan, "Fast algorithms for finding nearest common ancestors" in *SIAM Journal on Computing*, 13:338–355, 1984.