Optimal Suffix Tree Construction with Large Alphabets

Martin Farach*
Rutgers University

April 21, 1997

Abstract

The suffix tree of a string is the fundamental data structure of combinatorial pattern matching. Weiner [Wei73], who introduced the data structure, gave an $O(n)$ time algorithm algorithm for building the suffix tree of an n character string drawn from a constant size alphabet. In the comparison model, there is a trivial $\Omega(n \log n)$ time lower bound based on sorting, and Weiner's algorithm matches this bound trivially. Since Weiner's paper, the main open question has been how to deal with integer alphabets. There is no super-linear lower bound, and the fastest known algorithm was the $O(n \log n)$ time comparison based algorithm. We settle this open problem by closing the gap: we build suffix trees in linear time for integer alphabet.

*Department of Computer Science, Rutgers University, Piscataway, NJ 08855, USA. (*farach@cs.rutgers.edu*, *http://www.cs.rutgers.edu/~farach*). Supported by NSF Career Development Award CCR-9501942, NATO Grant CRG 960215, NSF/NIH Grant BIR 94-12594-03-CONF and an Alfred P. Sloan Research Fellowship.

1 Introduction

Given a string $S \in \Sigma^n$, the *suffix tree* T_S of S is the compacted trie of all the suffixes of $S\yen$, $\yen \notin \Sigma$. For many reasons, this is the fundamental data structure in combinatorial pattern matching. It has a compact $O(n)$ space representation which has many elegant uses. Furthermore, Weiner [Wei73], who introduced this powerful data structure, showed that T_S can be constructed in $O(n)$ time for constant size alphabet. This construction and its analysis are nontrivial. Considerable effort has been put into producing simplified linear time suffix tree constructions [CS85, McC76], though all such algorithms have been variants of the original approach of Weiner.

The construction of suffix trees remains an active area of research [DK95, Kos94, FM96]. Several open problem remain. The most important of these has to do with the size of the alphabet and its effect on the time needed to build suffix trees. In addition to the case of constant sized alphabet, there are two other significant cases: the case of unbounded alphabet, in which string characters can only be manipulated by comparison in some total order on $\Sigma \cup \{\yen\}$; and the case of integer alphabet.

Before we review the complexity of building a suffix tree, we consider the desired format of our output. Suffix trees are often used as indices. Therefore, the operation they should support is that of tracing from the root with some query pattern. Thus, one typically assumes a format for the suffix tree in which the edges leaving a node are lexicographically sorted, that is, they are sorted by the first character of the string labeling them. In this representation, sorting is a lower-bound for suffix tree construction, since we can use the suffix tree to retrieve the sorted order of the input in linear time.

Therefore, in the general alphabet case, we have a lower bound of $\Omega(n \log n)$ for suffix tree construction[1]. Any of the linear time algorithms for suffix tree construction on constant size alphabet generalizes to give an $O(n \log n)$ time algorithm in this case. More generally, if σ distinct characters occur in a string, the lower and upper bounds in the character comparison model match, at $\Theta(n \log \sigma)$.

The interesting case occurs when the alphabet consists of integers. Since sorting is still a lower bound for suffix tree construction in this case, we can assume that input characters have been sorted. If we replace each character by its rank in the sorted list, we get an equivalent integer alphabet in the range $[1, n]$. In this case, there is no non-trivial lower bound. However, the best upper bound known is the straightforward $O(n \log n)$. Note that in [DK95], Delcher and Kosaraju claimed a linear time algorithm for this problem, but this algorithm turned out to have a bug in it, as they noted in [DK96], where they also posed closing the log gap in the complexity of suffix tree construction for the linear alphabet case as an important and difficult open problem.

In this paper, we solve this open problem and close the gap by giving a linear time suffix tree algorithm for integer alphabets in the range $[1, n]$. We do so by introducing an algorithm which completely breaks with the Weiner approach. Before outlining the algorithmic approach, we give some preliminary definitions.

[1] In this case, it is possible to give a "stronger" lower bound. The $\Omega(n \log n)$ bound can be gotten from element distinctness, even without assuming the sorted output form.

1.1 Preliminaries

Let $S \in \{1, \ldots, n\}^n$. The suffix tree T_S of S is the compacted trie[2] of all the suffixes of $S¥$, where $¥ \notin \{1, \ldots, n\}$. Throughout the paper, we will assume that suffix trees are represented as follows. Leaf l_i represents suffix $S[i, n]$. Given i, we can access l_i in constant time. Each internal node v has a length $L(v)$ which is the sum of the edge lengths on the path from the root to v. Then the string at v, denoted $\sigma(v)$, is $S[i, i + L(v) - 1]$ where l_i is any leaf below v. The children of node v are stored in a list sorted by the first character on the edge from v. See Figure 1.1 for an example.

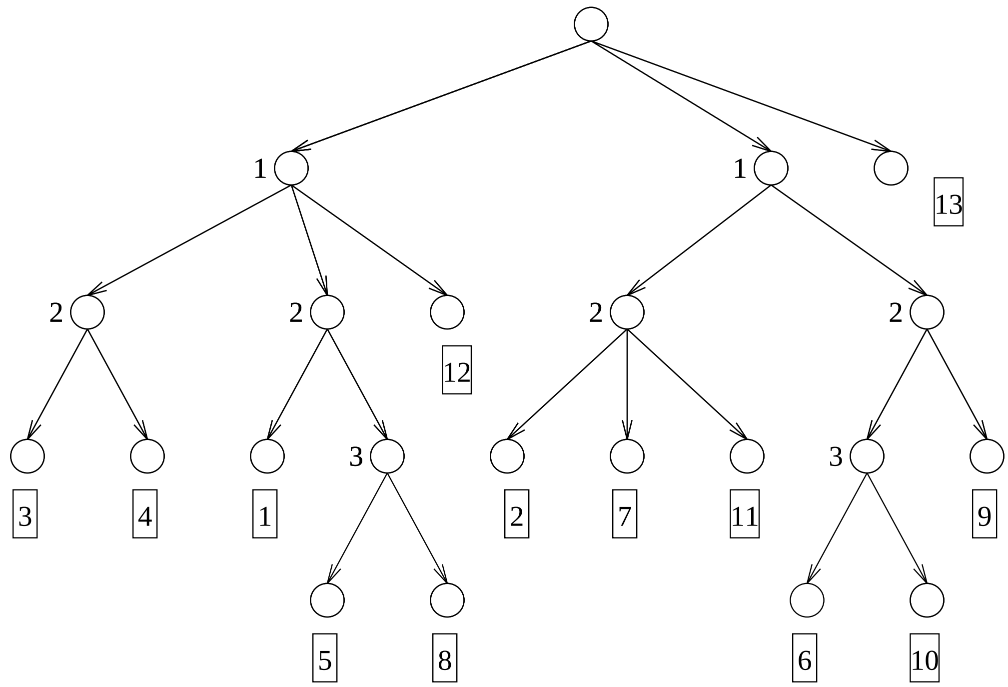

Figure 1.1: Suffix tree of string $S = 121112212221¥$. Numbers in boxes represent which suffix ends at that leaf. Numbers next to internal nodes represent the string length ($L(\cdot)$) of that node.

The following well-known lemma gives suffix trees a nice structure.

Lemma 1.1 ([Wei73]) *Let $a \in \{1, \ldots, n\}$ and $\alpha \in \{1, \ldots, n\}^*$. If there is a node v in T_S such that $\sigma(v) = a\alpha$, then there is a node w in T_S such that $\sigma(w) = \alpha$.*

Given this lemma, we can define, for every node v in a suffix tree, the *suffix link* $sl(v) = w$, where v and w are defined as in Lemma 1.1. Notice that $sl(\cdot)$ links form a tree rooted at the root of T_S. The depth of any node v in this $sl(\cdot)$ tree is then just $L(v)$.

Let $\texttt{lcp}(\alpha, \beta)$ be the *length of the longest common prefix* of two strings. Let $|\alpha|$, the length of string α, be m, for $\alpha \in \Sigma^m$. Let $\texttt{lca}(v, w)$ be the *least common ancestor* of two nodes in a tree. The

[2] A compacted trie differs from a trie in that paths of nodes, each having degree 2 (i.e. one child and its parent), are replaced by edges labeled by the appropriate substring.

property of suffix trees most often exploited algorithmically is the following relationship between lcp in S and lca in T_S.

$$\forall v, w \in T_S, \qquad \mathtt{lcp}(\sigma(v), \sigma(w)) = |\sigma(\mathtt{lca}(v, w))|.$$

The nice thing about this equality is that least common ancestors can be computed in constant time after linear preprocessing [HT84]. Thus arbitrary substring equality can be tested in constant time after a linear preprocessing of a string, as long as the suffix tree can be built in linear time.

1.2 Algorithm Outline

As noted above, a suffix tree is the compacted trie of the suffixes of a string. Our approach for finding the suffix tree is to compute the compacted trie of different subsets of the suffixes of a string and then merge the trees. In particular, we will show how to recursively compute the compacted trie of all suffixes beginning at odd positions. We call this tree T_o and note that it has $n/2$ leaves, and is therefore half the size of T_S. From T_o we will show how to compute T_e, the compacted trie of suffixes beginning in even positions. Character sorting will play a prominent rôle in both these steps, and so this is the point where we use the assumption that the alphabet consists of small integers.

Finally, we must merge T_o and T_e into the final tree. This step is the crux of the matter. Notice that while both T_o and T_e have representations which are of size $O(n)$, they implicitly share $O(n^2)$ characters in common. We must therefore exploit structural properties of suffix trees to perform this step in $O(n)$ time. This step does not rely on sorting, and so it works on odd and even trees over an arbitrary alphabet. While a similar approach was taken in [FM96], the algorithm in that paper relied heavily on randomization and only works for binary alphabets. The main result in that paper was to achieve a logarithmic time and linear work suffix tree construction on a PRAM for binary strings.

In Sections 2, 3, and 4, we present the three main steps of the algorithm.

2 Building the Odd Tree

In this section, we show how to compute T_o, the compacted trie of all suffixes of S beginning in odd positions. Figure 2.2 gives the odd tree for our example string. We first map pairs of characters into single characters as follows. For $i = 1$ to $n/2$, form pairs $\langle S[2i-1], S[2i]\rangle$. Radix sort them in linear time, remove duplicates, and compute the function $S'[i] = \text{rank of } \langle S[2i-1], S[2i]\rangle$ in the sorted list. S' can be computed in linear time and is a string of length $n/2$ over integer alphabet $[1, n/2]$. For our example, $S' = 212343\yen$.

Now recursively compute $T_{S'}$, the suffix tree of S'. See Figure 2.3 for $T_{S'}$ for our example string. Notice that any odd suffix $S[2i-1]\ldots S[n]\yen$ is a suffix $S'[i]\ldots S'[n/2]\yen$. So each leaf l_i of $T_{S'}$ becomes l_{2i-1} of T_o. Any internal node of $T_{S'}$ with length i becomes an internal node of length T_o with length $2i$. Call this tree T'. So T' only differs from $T_{S'}$ in its labels.

T' and T_o are quite similar. T' contains all odd suffixes of S but is not the compacted trie of these suffixes. To see why, notice that all internal nodes of T' have even length, while two odd suffixes of S may have a longest common prefix of odd length. Put another way, if node u in $T_{S'}$

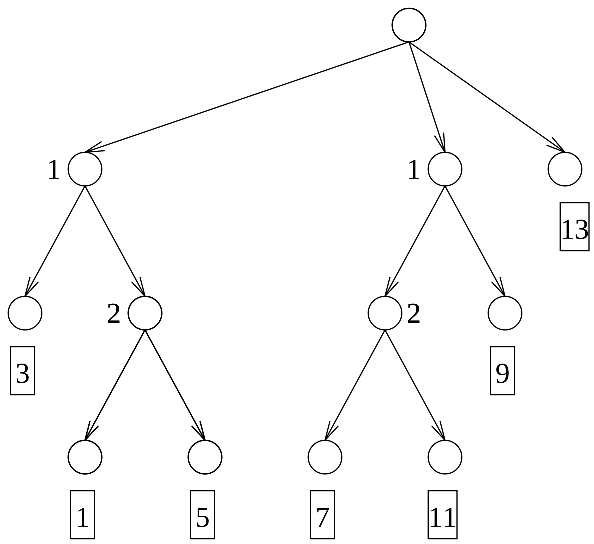

Figure 2.2: Odd tree of string $S = 121112212221$¥

has two children v and w so that the first character on edge (u, v) is $\langle a, b\rangle$ and the first character on edge (u, w) is $\langle a, c\rangle$, for $a, b, c \in [1, n]$, then the corresponding node u in T' would have two children whose edges start with the same character, and so T' is not a trie at all.

So we must do a final patching of T' in which we take, for all internal nodes u, all of u's children whose edges start with each character and introduce a new node between u and these children. This can be done very quickly since the edges coming from any node are lexicographically sorted by their first two characters (since we chose the alphabet for S' by the rank of character pairs in such a lexicographic sorting). So for each edge, we need only check if its first character agrees with the first character of the proceeding edge, and if so, make the minor adjustment. Finally, it may be that the edges to all of u's children starts with the same character, in which case u would end up with only one child. If so, we delete u. This clearly takes only constant time per edge and constant time per node, and so linear time overall.

We conclude that

Lemma 2.1 *If $T(n)$ is the time it takes our algorithm to build the suffix tree of a string $S \in \{1, \ldots n\}^n$, then T_o can be built in $T(n/2) + O(n)$.*

We will show that we can get T_e from T_o in linear time, and that we can merge T_o and T_e in linear time. This will establish the main claim, that we can build a suffix tree in linear time.

3 Building the Even Tree

Suppose we are given an inorder traversal of the leaves of a tree, and the depth of the `lca` of adjacent leaves in this ordering. Then it is a straightforward exercise to reconstruct the tree in

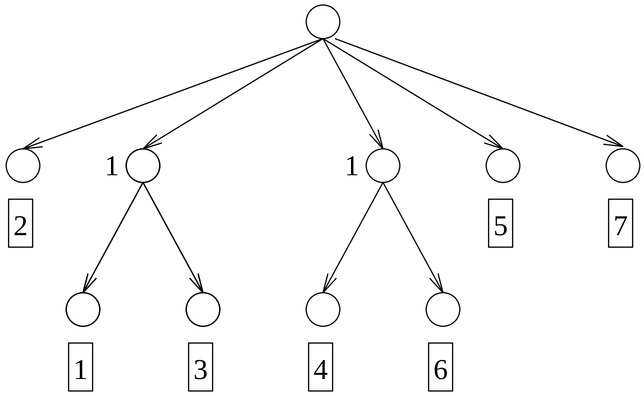

Figure 2.3: Suffix tree of string $S' = 212343\yen$

linear time. Similarly, suppose we are given the lexicographic traversal of the leaves of a compacted trie, which we call the *lex-ordering*, and the length of the longest common prefix of adjacent leaves. Then we can reconstruct the trie in linear time [FM96]. We derive these two pieces of information for the even tree from the odd tree.

First, consider the lex-ordering. We must derive the lexicographically sorted order of the even suffixes of S. Notice that we have such a lexicographic ordering of the odd suffixes of S, given trivially by the lex-ordering of the leaves of T_o. Also, notice that an even suffix is a single character followed by an odd suffix. Thus, we have a type of radix sort problem. We need only stably sort the lex-ordered leaves of T_o, using the key $S[2i]$ for any odd suffix $S[2i+1]$. By the correctness of radix sort [AHU74], the even suffixes are correctly sorted. Therefore, we can get the lex-ordering of the leaves of T_e in linear time. The details of the whole process are evident from the following example.

In our example string, we get the ordering $[3, 1, 5, 7, 11, 9, 13]$ for the leaves of T_o (See Figure 2.2). We must therefore stably sort the follow tuples: $[\langle 2,3\rangle, \langle 1,5\rangle, \langle 2,7\rangle, \langle 2,11\rangle, \langle 1,9\rangle, \langle 1,13\rangle]$. To repeat, the 1's and 2's placed in the tuples are the 2nd, 4th, 6th, 10th, 8th and 12th characters of the string, which are the characters preceding the 3rd, 5th, 7th, 11th, 9th, and 13th suffixes. Notice that we do not include the 1 suffix since it is not preceded by any character. Stably sorting by the first number yields the ordering $[\langle 1,5\rangle, \langle 1,9\rangle, \langle 1,13\rangle, \langle 2,3\rangle, \langle 2,7\rangle, \langle 2,11\rangle]$. Subtracting one from the second index completes the computation and yields $[4, 8, 12, 2, 6, 10]$ as the sorted order of the even suffixes.

Now, given two leaves l_{2i}, l_{2j} of T_e, we wish to determine $\texttt{lcp}(l_{2i}, l_{2j})$. As noted above, we can preprocess T_o in linear time so that we can get $\texttt{lcp}$ information about odd suffixes in constant time. Having done this, we note that

$$\texttt{lcp}(l_{2i}, l_{2j}) = \begin{cases} \texttt{lcp}(l_{2i+1}, l_{2j+1}) + 1 & \text{if } S[2i] = S[2j]; \\ 0 & \text{otherwise.} \end{cases}$$

These operations yield the tree in Figure 3.4.

We conclude that

Lemma 3.1 *Given $S \in \{1, \ldots, n\}^n$ and its odd tree T_o, T_e can be constructed in $O(n)$ time.*

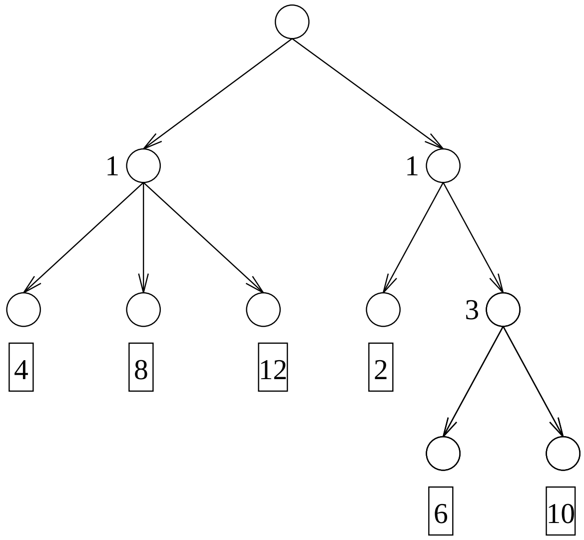

Figure 3.4: Even tree of string $S = 121112212221$¥

4 Merging the trees

In this section, we show how to merge T_e and T_o into T_S. This routine will run in linear time, thus completing the suffix tree construction algorithm.

First, consider how one would merge T_e and T_o if they were uncompacted, so that each edge would be labeled by a single character. The obvious procedure is a coupled depth first search tour of the two trees. That is, start by identifying the roots of both trees. Now simultaneously take the edge whose label is 1 in both trees, and recursively merge the two subtrees. When the DFS merge of these two subtrees is done, then we do the same for the edge whose label is 2, and so on for all the edges incident on the root. At any stage, if only one of the trees has an edge labeled i, then we need not recurse on that subtree since there is nothing to merge.

If we were merging uncompacted tries, then this procedure would give an optimal and simple method for doing so. Now consider just the first step of this procedure on a compacted trie. We take the children of the root whose first character is a 1 (or whatever the first shared character is). Each of these edges represents a string of up to $\Omega(n)$ characters, and so we cannot spend the time to see if the edges really agree on all the characters. This would give an $O(n^2)$ time algorithm for merging. Also, we must take care of details such as the fact that the edges we are merging need not be of the same length, and so, when we merge two edges, we must see if the shorter is a prefix of the longer and break the longer edge at an appropriate length.

Suppose we had an oracle for telling if two substrings of S are equal. Then we could use it in our coupled depth first search, in place of simply testing character equality. Most of the difficulty in merging the trees would be overcome. Note than even still, such an oracle would be insufficient to solve the problem, at least by such a naïve coupled-DFS algorithm, since two edges might agree on some proper prefix, and thus require partial merging. Furthermore, no such oracle is available, so instead, we use a weak oracle which will never tell us that equal edges are different, but may sometimes tell us that unequal strings are the same. Then we will overestimate the depth to which the trees need to be merged, but we will never merge too little. We will use such a weak oracle, and then show how to unmerge the parts of the trees which should not have been merged. This second part will take advantage of the structural information of suffix trees. In fact, this checking part will be so good that we can use *any* weak oracle. We will show that it suffices to report that two edges are equal if they share the same first character. To repeat, using this simple oracle will allow us to merge properly the parts of T_o and T_e which need to be merged, but may merge two subtrees which should not be merged. So we must show how to unmerge the incorrectly merged parts. See Figure 4.5 for the result of such a merger on our example string.

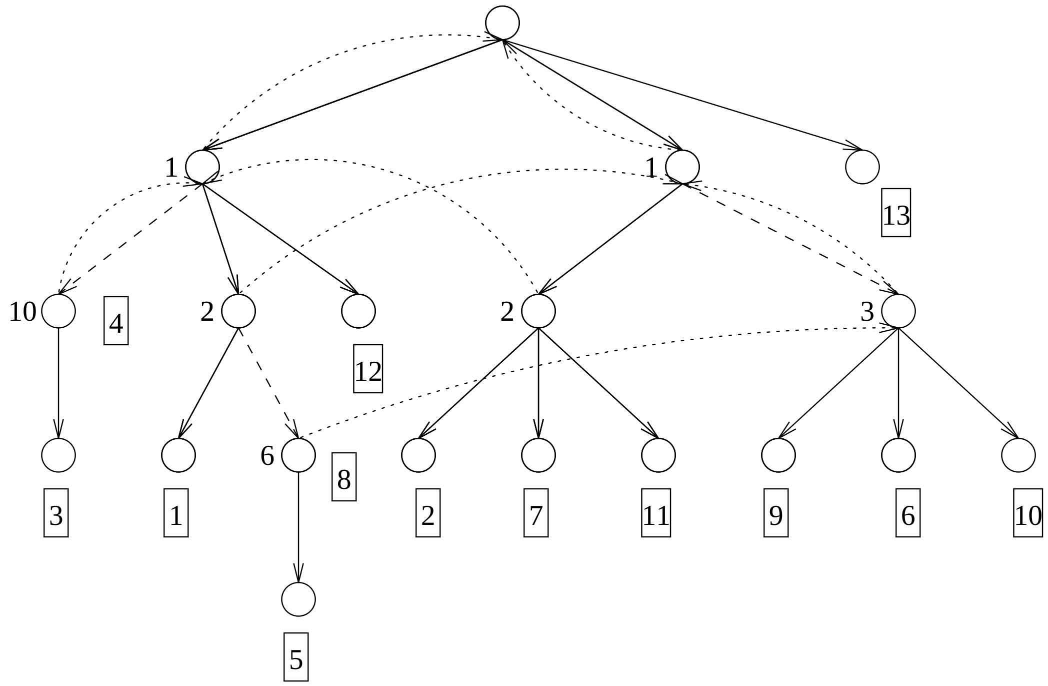

Figure 4.5: "Merging" even and odd trees of string S = 121112212221¥ yields this tree. The dashed edges are edges which have been merged too far. The dotted lines are the $d(\cdot)$ tree described below.

Let M be the tree derived by a merge-DFS of T_o and T_e using our weak oracle, that is, M is obtained by merging T_o and T_e via a coupled DFS. If we decide that two edges are equal, by testing if they start with the same character, then we break the longer of the two edges and merge its prefix with the shorter of the two edges. In this way, we only merge edges of equal length. This procedure proceeds down to the leaves in $O(n)$ time.

Consider some node u. We wish to check if the merge-DFS went too far in merging until u, or if the tree structure is good as far down as u. We call a node of M *odd* if it occurred in T_o and *even* if it occurred in T_e. Notice that a node, for example the root, can be both odd and even. Recall that $L(u) = |\sigma(u)|$. $L(u)$ can be looked up for each node in M by consulting T_o or T_e. Let l_{2i} and

l_{2j-1} be descendants such that u is their least common ancestor in M. Such an odd/even pair must always exist for nodes in the merged region, even if the merged node is a leaf in the original tree. Let $\hat{L}(u)$ be defined to be $\texttt{lcp}(l_{2i}, l_{2j-1})$. Now we check the status of u as follows.

Lemma 4.1 *Let u, l_{2i} and l_{2j-1} be as defined above. u is properly merged in T_S if $L(u) = \hat{L}(u)$.*

The alternative is that $L(u) > \hat{L}(u)$, in which case we have merged too far. So we must show how to compute $\hat{L}(u)$ for all merged nodes u, and show what to do when Lemma 4.1 is violated.

4.1 Computing $\hat{L}$

Recall that $\hat{L}(u) = \texttt{lcp}(l_{2i}, l_{2j-1})$, for some l_{2i}, l_{2j} descendants of u. But

$$\texttt{lcp}(l_{2i}, l_{2j-1}) = \begin{cases} 1 + \texttt{lcp}(l_{2i+1}, l_{2j}) & \text{if } S[2i] = S[2j-1]; \\ 0 & \text{otherwise.} \end{cases}$$

Let $v = \texttt{lca}(l_{2i+1}, l_{2j})$. Then we will show that $\hat{L}(u) = 1 + \hat{L}(v)$, as long as u is not the root of M. Define $d(u) = v$ where u and v are as above. Compute d for every merged node in M, other than the root. See the dotted edges in Figure 4.5.

Lemma 4.2 *The function d defines a tree on the nodes of M. Further, $\hat{L}(u)$ = the depth of u in the d tree.*

Proof: In our definition of the d function, there is some non-determinism in terms of the choice of descendants l_{2i} and l_{2j-1}. However, it doesn't matter which odd and even descendants we pick, since they all have the same longest common prefix. This is because they all have the same least common ancestor in T_S. Thus, $\hat{L}(u)$ has the same value, no matter which odd and even descendant leaves we pick.

Now we show that if $d(u) = v$, then $\hat{L}(u) = 1 + \hat{L}(v)$. This establishes both that the d pointers form a tree, and, since $\hat{L}(root) = 0$, that $\hat{L}(u)$ is the depth of u in this tree. Since $d(u) = v$, there is some pair l_{2i}, l_{2j-1} with lca u, such that the lca of l_{2i+1}, l_{2j} is v. But we know that $\texttt{lcp}(l_{2i}, l_{2j-1}) = 1 + \texttt{lcp}(l_{2i+1}, l_{2j})$. Finally, by definition $\hat{L}(u) = \texttt{lcp}(l_{2i}, l_{2j-1})$ and, from the observation above that all odd/even descendant pairs give the same $\hat{L}$, $\hat{L}(v) = \texttt{lcp}(l_{2i+1}, l_{2j})$, thus establishing the lemma. ■

The depth of every node in the d tree is determined in linear time by DFS. The only other step is the computation of least common ancestors, which, as noted above, requires linear processing and constant time queries [HT84]. Notice, for example, that in Figure 4.5, $\hat{L}(\texttt{lca}(l_5, l_8)) = 3$, $\hat{L}(\texttt{lca}(l_6, l_9)) = 2$ and $\hat{L}(\texttt{lca}(l_9, l_{10})) = 2$.

4.2 Using $\hat{L}$ to Unmerge the Tree

We now show how to use $\hat{L}$ to partially unmerge T_o and T_e within M. We can either *completely unmerge* a node u or *partially unmerge* it. Let $p(u)$ be the parent of u in the original tree (T_e or T_o). Let $p'(u)$ be the parent of u in M. Completely unmerging a node u means setting $p'(u) = p(u)$, that is, reseting the parent of u to be its original parent in the unmerged tree. If Lemma 4.1 is violated, so that $L(u) > \hat{L}(u)$, then a complete unmerge is not always the correct thing to do.

Consider, for example, Figure 4.5. It would be incorrect to completely unmerge node $\mathtt{lca}(l_5, l_8)$. Instead we need to introduce a node which is the parent of l_5 and l_8 and set its string length to 3, its depth in the d tree. This is a partial unmerged. Both types of unmerges are handled by the following procedure.

Define a node u to be a *border* node if $\hat{L}(u) < L(u)$, but $\hat{L}(p'(u)) = L(p'(u))$. Notice that the set of border nodes form an anti-chain in M. We can easily find all the border nodes in linear time. Now, for each border node u, we perform the following operations. Let T_o^u be the odd tree below u and let T_e^u be the even tree below u in M. No node in T_o^u or T_e^u should be merged, so unmerge them all by reseting, for each such node v, $p'(v) = p(v)$. The remaining question is what node to hang T_o^u and T_e^u off of. But the set of border nodes are exactly those nodes at which we need a partial unmerge. These are handled, for each border node u, as follows

1. Create a new node u' and make it a child of $p'(u)$.
2. Set $L(u') = \hat{L}(u)$.
3. Hang T_o^u and T_e^u off of u'. Order the children of u' lexicographically.

This implements the partial unmerge case described above.

When this procedure is completed for all border nodes, the tree will be correct, since Lemma 4.1 will be satisfied for every node. The total running time is linear. We conclude that:

Theorem 4.3 *Given a string $S \in \{1, \ldots, n\}^n$, the suffix tree T_S of S can be deterministically constructed in $O(n)$ time and space.*

Proof: By Lemmas 2.1 and 3.1, and the discussion of this section. Note finally that the children of all nodes are sorted by their first character. Thus the tree produced is properly sorted, as required by the recursion. ∎

Acknowledgements

Many thanks to Paolo Ferragina, Raffaele Giancarlo, and S. Muthukrishnan for helpful comments.

References

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[CS85] M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, chapter 12, pages 97–107. NATO ASI Series F: Computer and System Sciences, 1985.

[DK95] A. Delcher and S. Kosaraju. Large-scale assembly of dna strings and space-efficient construction of suffix trees. *Proc. of the 27th Ann. ACM Symp. on Theory of Computing*, 1995.

[DK96] A. Delcher and S. Kosaraju. Large-scale assembly of dna strings and space-efficient construction of suffix trees (corrections). *Proc. of the 28th Ann. ACM Symp. on Theory of Computing*, 1996.

[FM96] M. Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. *Proc. of 23rd International Colloquium on Automata Languages and Programming*, 1996.

[HT84] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13:338–355, 1984.

[Kos94] S. Kosaraju. Real-time suffix tree construction. *Proc. of the 26th Ann. ACM Symp. on Theory of Computing*, 1994.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[Wei73] P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.