Experiments with Suffix Trees for Extracting Repeating Patterns from Trees

6.854/18.415 - Advanced Algorithms
Final Project

Andrew Hogue*

December 12, 2003

Abstract

We explore methods for finding repeating patterns in labeled tree structures using the *suffix tree* data structure. Because suffix trees are typically used for string manipulation, we develop algorithms for inserting trees as strings, and for recovering their hierarchical structure once patterns have been found. We also explore an extended data structure, the *suffix multitree*, which allows us to better take advantage of the hierarchical structure of trees when looking for patterns.

*ahogue@lcs.mit.edu

1 Introduction and Related Work

In this paper we explore various methods for using suffix trees to extract repeating patterns from tree structures. We present experimental results showing different ways to utilize the standard suffix tree structure. We also give an extension which allows us to insert hierarchical data directly into a *suffix multitree*, as well as retrieve repetitive structures.

The suffix tree data structure was developed for use on strings, both for pattern matching and for discovering repeating substrings. The initial work was done by Weiner [8], who demonstrated a linear-time construction. McCreight [5] and Ukkonen [7] later developed more space-efficient algorithms, also with linear-time construction.

Techniques for pattern recognition and matching on trees have several applications. Much of the seminal pattern matching work was done in the field of compiler optimization [2]. In molecular and computational biology, pattern recognition is vital to mapping DNA and RNA chains effectively, as well as exploring protein and gene regulation networks [1, 4]. Our research, in performing wrapper induction on semantic data on the World Wide Web, requires efficient pattern inference and matching based on positive examples provided by a user [3].

We proceed as follows. We first define common terms and algorithms, as well as our experimental test structures, in Section 2. In Section 3 we discuss our implementation and experiments using existing suffix tree data structures. In Section 4 we explore an extension to suffix trees which allows us to take more advantage of the hierarchical nature of the tree structures we are inserting. We conclude with a discussion of future work in Section 5.

2 Definitions

Throughout this paper, we consider finding maximal repeating structures in a labeled tree T with n nodes, maximum depth d, and maximum branching factor b. The subtree rooted at node v_i of T is denoted $T[i]$. The parent of a node v_i is $parent(v_i)$, and the set of children of v_i are $children(v_i)$.

2.1 Maximal Repeats in Strings

For a string S, a maximal pair is defined as a pair of identical substrings α and β, such that the character to the immediate left (and right) of α is different from the character to the immediate left (and right) of β. A maximal repeat is any substring which appears in at least one maximal pair.

All maximal repeats of a string may be extracted from its suffix tree using the following method: During construction of the suffix tree, for each leaf in the suffix tree, maintain the *left-character* of the suffix which that leaf represents (that is, the character $S[i-1]$ of the suffix $S[i..n]$). Define a node v of the suffix tree to be *left-diverse* if at least two leaves in the subtree rooted at v have different left-characters. Gusfield [1] proves that the substring represented by an internal node v is a maximal repeat if and only if v is left-diverse.

Because of the fact that left-diversity propagates upwards (i.e., if a node is left-diverse, its parent is also left diverse), the left-diversity of all nodes may be computed in $O(n)$ time by a depth-first traversal of the tree. This gives a total $O(n)$ time to retrieve all maximal repeats from a string.

2.2 Maximal Repeats in Trees

In trees, we define a maximal repeat similarly:

- Two subtrees $T[i]$ and $T[j]$ are said to have an *alignment* if the roots v_i and v_j have the same label, and there exists a one-to-one mapping between some subset of $children(v_i)$ and $children(v_j)$ such that each pair of mapped children also has an alignment[1]. Note that the set of aligned children may be empty.

- A *maximal pair* is any pair of aligned subtrees $T[i]$ and $T[j]$, such that $parent(v_i)$ is different from $parent(v_j)$, and for each pair of matching leaves v_k and v_l of $T[i]$ and $T[j]$, $children(v_k) \cap children(v_l) = \emptyset$ (that is, the alignment could not be extended "downwards" in the tree any further by adding any children of the leaf nodes).

- A *maximal repeat* is any subtree which appears in at least one maximal pair.

2.3 Repeating Structures in Trees

There are two main structural forms we will focus on for finding maximal repeats in this paper[2]. The first is repetitive *subtree structure*. An example,

[1]If we are considering *ordered* trees, the additional requirement may be made that the ordering of the children must be maintained in the mapping.

[2]These structures were chosen based on a survey of common semantic structures in HTML [3].

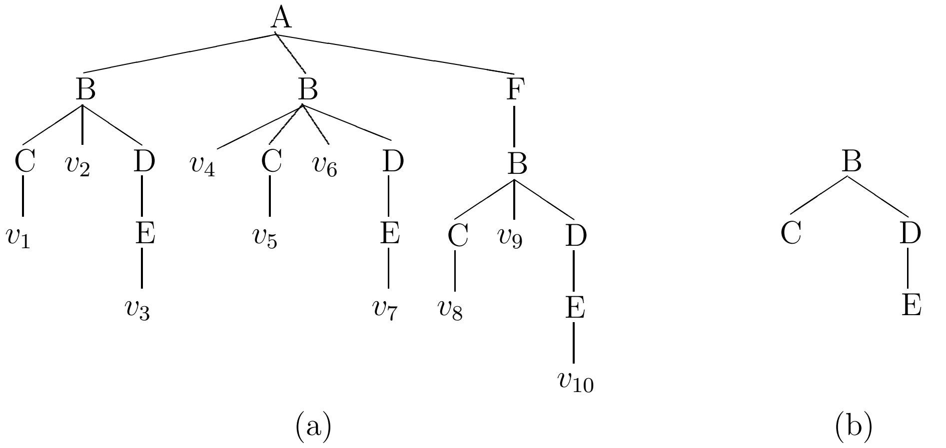

Figure 1: (a) Example tree T_1, with patterns grouped by subtree. (b) The largest desired maximal repeat of T_1.

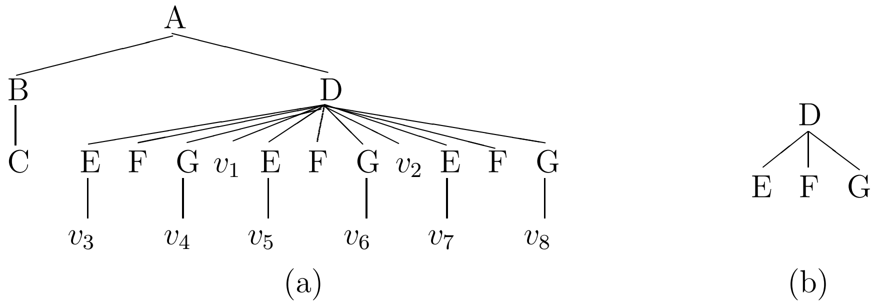

Figure 2: (a) Example tree T_2, with patterns grouped by sibling. (b) The largest desired maximal repeat of T_2.

which we will label T_1, is shown in Figure 1(a). This example shows a repetitive structure, shown in 1(b), rooted at nodes labeled B, occurring throughout the tree.

The second type of structure is repetitive *sibling structure*. An example tree, which we will call T_2, is shown in Figure 2(a). This tree demonstrates the repeating structure $\{E, F, G\}$, shown in 2(b), occurring amongst sibling children of the same node.

In both example trees, nodes labeled with capital letters represent actual labeled nodes, whereas nodes labeled with variable names such as v_i represent placeholders for one or more nodes or subtrees which may vary.

3 Experimental Results

To begin our testing we implemented the standard suffix tree data structure, as well as the algorithm for finding maximal repeats described in Section 2.1, using the Java programming language. We then proceeded to experiment with various methods of inserting T_1 and T_2 into the suffix tree.

In each experiment below, each individual node of the tree is treated as a separate character by the suffix tree. Two nodes are considered equal if their labels are equal.

3.1 Linear-Time Insertions

In this section we explore various ways to insert the nodes of a tree T in $O(n)$ time, and discuss experimental results based on the example trees. For each of these methods, the linear insertion time comes from the fact that each node in the tree appears exactly once in the insertion order into the suffix tree, which has construction time $O(n)$.

3.1.1 Depth First

We performed three separate depth-first traversals of each tree, in preorder, inorder, and postorder. The nodes returned by each traversal were then fed as an array into separate suffix trees, and the maximal repeats were computed.

For each of the traversal orders, in both T_1 and T_2, we found very poor extraction of the repeating structures. In T_1, the postorder traversal located the $\{E, D, B\}$ branch. However, the length of this branch was only a result of the order of traversal – if either node D or E had had a right sibling, the branch would not have been found as a repeat. The longest repeat found in T_2 was of length 2, and also depended heavily on the traversal order.

We postulate that the traversal orders of depth-first search are effectively random with respect to the different forms that repeating structures may take in trees. Because depth-first search descends all the way down to the leaf nodes of a particular branch of the tree, repeating structures that are *internally* located in the tree tend to be "broken up" by the traversal order. Below, we consider traversals which attempt to more effectively localize nodes which appear near each other in the tree.

3.1.2 Breadth First

We next performed a breadth-first traversal of both trees. The results of this traversal were fed into a suffix tree, and the maximal repeats were computed.

Breadth-first traversal was less effective than depth-first at pulling out repeating structures in T_1. This seems to be because of the number of variable nodes among the children of the B nodes in T_1, which break up the traversal order and stop the suffix tree from picking up repeating patterns.

On the other hand, breadth-first traversal was very successful at pulling out the repeating pattern $\{E, F, G\}$ in T_2. This is not surprising, as the repeating pattern is very "horizontal" in the tree, so the breadth-first traversal does well at grouping the similar structures together in the suffix tree.

3.2 Path Insertions

To try to capture some of the "vertical" repeating structure in the trees, we next decomposed the trees into paths from root to leaf. Each path was inserted into the suffix tree as a separate array, each with a unique terminal character. Maximal repeats were then computed from this suffix tree.

Construction time for this structure may be measured in terms of the number of paths inserted. There is one path for each leaf node, which, in a tree of depth d and branching factor b, results in $O(b^d)$ paths. Each path is of length $O(d)$, requiring $O(d)$ time to insert. Thus, the total insertion time is $O(b^d d)$. In a binary tree, with $d = \lg n$ and $b = 2$, this reduces to $O(n \log n)$.

Results for inserting paths were good for T_1. The sequences $\{B, C, E\}$, $\{B, C\}$, and $\{B\}$ were each picked out as maximal repeats. These sets may be unioned to form the maximal pattern shown in Figure 1 (b). However, performing this union accurately requires extra information which the suffix tree does not contain, an issue addressed in Section 4.

Inserting the paths of T_2, on the other hand, resulted in no repeats found. This is because all repeated nodes ($\{E, F, G\}$) have the same parent node, D, which disqualifies them from being considered as maximal repeats (because they are not left-diverse).

One modification which we found to slightly improve these results is to label the edges of each tree before inserting the paths into the suffix tree. This method only works for ordered trees, as edges were labeled with the index in the set of children which they represent. Edge labels also effectively double the size of each path being inserted.

After adding edge labels, the paths were reinserted. We found similar

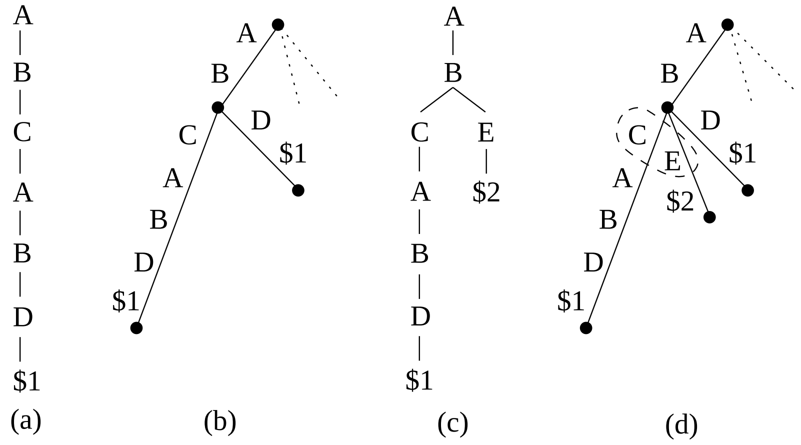

Figure 3: Modification of a suffix tree to support inserting tree structures. (a) A string, represented vertically. (b) The suffix tree for the string in (a), simplified to show only the A-labeled branch out of the root. (c) A modified version of the string, making it a tree. (d) The new suffix multitree for the tree in (c). The dashed line around nodes C and E represents the branch set of node B to which they both belong.

results for T_1. In adition, we found individual repeating nodes $\{E\}$, $\{F\}$, and $\{G\}$ in tree T_2, although they were not linked to their parent node D because each had a different child number.

4 An Extension: Suffix Multitrees

We note the following points about the experimental results from Section 3:

- Breadth-first traversal effectively locates sibling, or "horizontal" patterns in the tree.
- Path traversal effectively locates parent-child, or "vertical" patterns in the tree.

Based on these observations, we propose a modified suffix tree structure, the *suffix multitree*, which may be used to locate full, tree-structured repeats in a tree without having to reconstruct them from separate, linear elements.

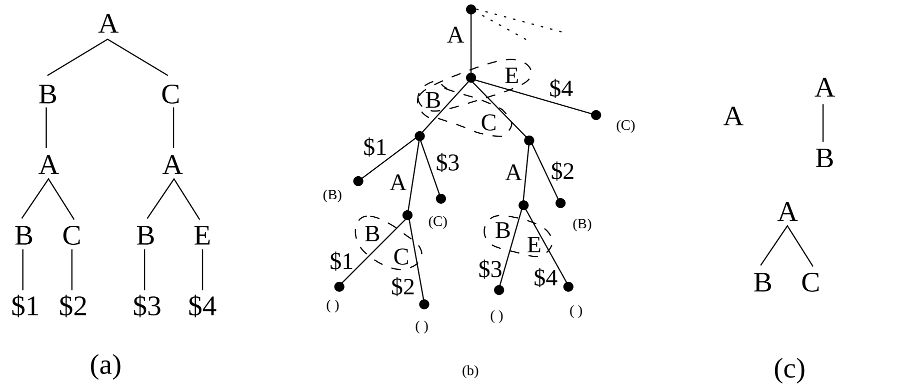

Figure 4: (a) Another example multitree. Note that node A has different sets of children depending on where it occurs in the tree. (b) The suffix multitree of the tree. Left-characters for each leaf are shown in parentheses. (c) The maximal repeating structures of the tree.

4.1 Turning a String into a Tree

We begin by considering how to update the suffix tree data structure to maintain information about trees. We note that we may think of a string as a "tree without branches," that is, a tree where every node has only one child. For example, in Figure 3 (a) and (b), we show the string $ABCABD$ and part of its suffix tree.

We now modify the string and add a branch, as shown in Figure 3 (c). If we now insert all paths from root to leaf as in Section 3.2, we get the suffix tree shown in Figure 3 (d). To counteract the difficulties in reconstructing the tree substructures noted in Section 3.2, we augment the suffix tree by maintaining *branch sets* at each node. These sets represent all nodes branching from an internal node in the suffix tree which appeared together in the original tree. For example, in Figure 3 (d), the nodes labeled C and E are in the same branch set of the node below branch B.

Another example tree and its multitree are shown in Figure 4 (a) and (b). In this tree, the nodes labeled A have different children at different points in the tree. This fact is represented by the overlapping branch sets in its suffix multitree.

We now proceed to describe modifications to the suffix tree construction and maximal repeats algorithms to handle the new multitree structures.

4.2 Modifying the Suffix Tree Construction Algorithm

Rather than inserting distinct paths into the suffix tree, as in Section 3.2, we instead insert full subtrees to construct a suffix multitree:

- For the root node v of the tree T, we insert a branch b out of the root of the suffix multitree, as in normal suffix trees.
- For each child of v, we insert a *separate* branch out of b.
- We add a branch set to b indicating that these branches are tied together.
- We then proceed down each branch in the tree, continuing to insert new branches and branch sets into the multitree.
- This process is repeated, inserting each child of the root of T as a branch out of the root of the suffix multitree, and so on down the tree, similar to the construction of a normal suffix tree.

This construction method is a direct analog of the construction of a standard suffix tree. Similar to the original construction algorithm, it may be improved through the use of suffix links, which allow us to jump to other parts of the multitree without traversing all the way to the root of the tree. Suffix links are also augmented with the branch sets which they encompass, so we only follow suffix links which represent the correct subtrees. With these modifications, a suffix multitree may be constructed in linear time with respect to the number of edges in the tree, which is equivalent to $O(n)$.

4.3 Modifying the Maximal Repeats Algorithm

To efficiently recover maximal repeating structures from a suffix multitree, we must also make small modifications to the maximal repeats algorithm described in Section 2.1. The concept of left-diversity still applies, and still propagates up the tree (that is, if a node is left-diverse, so is its parent).

However, with branch sets, the structures that are propagated get "broken up" as they move up the tree. For instance, in Figure 4, both children of the uppermost branch set $\{B, C\}$ are left-diverse. Thus, both B and C are propagated up the tree as children of A to create the bottom repeating structure shown in Figure 4 (c).

On the other hand, both children of the uppermost branch set $\{B, E\}$ are *not* left-diverse - node B is, but node E is not. Thus, in this case, only B

is propagated upwards, and we get the $A - B$ repeating structure in Figure 4 (c).

Similar to the original maximal repeats algorithm, the multitree maximal repeats algorithm requires only a single traversal of the tree, propagating left-diversity and current repeats up the tree. This may be done in linear time in the size of the tree, or $O(n)$.

4.4 Experimental Results on Multitrees

We extended the Java suffix tree implementation of Section 3 to insert full subtrees instead of paths using the branch set structure.

On example tree T_1 (Figure 1 (a)), the suffix multitree performed very well, extracting the desired pattern exactly.

On example tree T_2 (Figure 2 (a)), we obtained many more results than desired. Each result was a permutation of the desired $\{E, F, G\}$ subset. We believe these results were obtained because branch sets treat the children of a node as an unordered set, rather than an ordered list. Possible modifications to correct this issue are discussed below.

5 Future Work

As mentioned above, the current branch set implementation does not deal well with all *ordered* substructures. An enhancement to correct this would update the branch sets to be branch lists. However, this change requires us to redefine the semantics of a "repeat" in the tree - do we allow gaps between successive nodes in the repeats? Do we allow different *sized* gaps between different repeats? Both of these issues may be categorized as defining the semantics of *wildcards* in the trees, and are beyond the scope of this paper.

The idea of pattern matching in trees is something of a special case of the suffix multitree. We can perform offline preprocessing to construct the suffix multitree of the "text" tree T to be matched against. We then attempt to insert the pattern tree P into the multitree. If we can insert P without creating any new branches, and without creating any new branch sets, we have found a match for P in T.

Another interesting extension is the idea of *approximate* repeats. Ukkonen [6] addresses the issue of approximate matching of strings with an edit distance less than some k. These algorithms should be extensible to suffix multitrees without major modifications.

References

[1] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, 1997.

[2] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *J. Association of Computing Machinery*, 29(1):68–95, January 1982.

[3] A. Hogue. Tree pattern inference and matching for wrapper induction on the world wide web. http://secondthought.org/notes/thesisProposal.ps, 2003.

[4] S. Kurtz. Foundations of sequence analysis (lecture notes). http://bibiserv.techfak.uni-bielefeld.de/media/seqanalysis/prscript.pdf, 2001.

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

[6] E. Ukkonen. Approximate string–matching over suffix trees. In *Proc. 4th Annual Symposium on Combinatorial Pattern Matching*, pages 229–242, 1993.

[7] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14:249–260, 1995.

[8] P. Weiner. Linear pattern matching algorithms. In *Proc. of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.