Contracted Suffix Trees: A Simple and Dynamic Text Indexing Data Structure

Andrzej Ehrenfeucht[1], Ross M. McConnell[2], and Sung-Whan Woo[2]

[1] Dept. of Computer Science, 430 UCB, University of Colorado at Boulder, Boulder, CO 80309-0430, USA.
[2] Dept. of Computer Science, Colorado State University, Fort Collins, CO 80523-1873 USA

Abstract. We address the problem of finding the locations of all instances of a string P in a text T, where of T is allowed to facilitate the queries. Previous data structures for this problem include the suffix tree, the suffix array, and the compact DAWG. We modify a data structure called a *sequence tree*, which was proposed by Coffman and Eve for hashing, and adapt it to the new problem. We can then produce a list of k occurrences of any string P in T in $O(||P|| + k)$ time. Because of properties shared by suffixes of a text that are not shared by arbitrary hash keys, we can build the structure in $O(||T||)$ time, which is much faster than Coffman and Eve's algorithm. These bounds are as good as those for the suffix tree, suffix array, and the compact DAWG. The advantages are the elementary nature of some of the algorithms for constructing and using the data structure and the asymptotic bounds we can give for updating the data structure when the text is edited.

1   Introduction

In this paper, we consider the problem of finding occurrences of a pattern string P in a text T, where preprocessing of T is allowed in order to create a data structure that speeds up the search.

Let m denote the length $||P||$ of P, let n denote the length $||T||$ of T, and let k be the number of positions in T where P occurs as a substring. For simplicity, we assume for the moment that the size of the alphabet Σ is fixed.

Previous data structures for this problem include the *suffix tree* [1], the *compact directed acyclic word graph (compact DAWG)* [2], and the *suffix array* [3]. The first two approaches take $O(n)$ time to build the data structure, and $O(m + k)$ time to find the k positions where the pattern string occurs.

The suffix array can be constructed in $O(n)$ time, and takes $O(m + \log n)$ time to produce a pointer to a list of occurrences of of P in T. A slightly slower approach takes $O(m \log n)$ time, and this approach is of practical interest because of its simplicity.

In this paper, we describe an alternative to these data structures, which we call a **contracted suffix tree.** It can also be constructed in $O(n)$ time, and takes $O(m + k)$ time to find the k occurrences of the pattern string. Unlike the

suffix array, it does not give a pointer to a list of occurrences, but it can be made the basis of a data type that takes $O(m)$ time to input P and $O(1)$ time to produce an occurrence each time one is requested, even if the user does not ask for all occurrences. This is just as good as producing a list, since it provides what amounts to a list iterator. (The suffix tree can also be augmented with extra pointers to provide such an interface.)

A special case of the contracted suffix tree, called the *position heap* can be made the basis of a list interface that takes $O(m)$ time to input P and produces a list iterator that gives the positions in left-to-right order of the occurrences in the text, at a cost of $O(\log k)$ time per element. The previous approaches give the occurrences in an order that has no relation to their left-to-right order in the text.

Like the suffix tree and the compact DAWG, and unlike the suffix array, our bounds must be increased by a $\log |\Sigma|$ factor when the size of the alphabet, Σ, is introduced as a variable. This factor comes from the time required to find the child of a node on the child edge labeled b. This can be improved to $O(1)$ *expected* time with a hash table that returns the child given a hash key consisting of the parent and a letter. This is nevertheless a disadvantage when compared to suffix arrays.

The proposed approach has the advantage that it has a good time bound for modifying the data structure after arbitrary text edits on T. The *generalized suffix tree* allows search for a pattern string in a collection of texts. In [4], it is shown that it is possible to implement it to allow insertion and removal of any text X in the collection in $O(||X||)$ time. However, X must be inserted or removed in its entirety and smaller edits on X are not supported. Very recently, Salson *et. al.* have given an approach that takes $O(n)$ worst-case time to modify a variant of the suffix array after an arbitrary edit operation on T [5]. This is as bad as the cost as discarding the suffix array and rebuilding it from the beginning. However, they argue that their approach is much more efficient in practice, and support this with empirical studies on benchmarks.

Except when T has very low entropy, our proposed data structure can be updated efficiently when the text T is modified. Let $h(T)$ be the length of the largest substring X of T that is repeated more than $||X||$ times in T. A few moment's consideration reveals that $h(T)$ can be expected to be quite small for most practical applications. The expected value of $h(T)$ is $O(\log n)$ when T is a randomly-generated string. Since few applications deal with random strings, a more important observation is that long repeated substrings in T have little impact on the value of $h(T)$ unless they are repeated an inordinate number of times.

Updating the proposed structure after deletion or insertion of a consecutive block of b characters takes $O((h(T)^2 + bh(T)))$ time. The tradeoff of implementing it in this way is that searches take $O(m \log k + k)$ time, rather than $O(m + k)$ time, to produce the k occurrences of the pattern string. In the worst case, as when $T = a^n$, $h(T) = \Theta(n)$, and one can resort to the $O(n)$ bound obtained by discarding and rebuilding the structure. However, the bound is a stronger

one than $O(n)$ because it characterizes analytically the relationship between the running time and an easily-understood property of the text.

We can give a somewhat stronger bound as follows. Let $h(T, i)$ be the length of the longest string X that has more than $||X||$ occurrences in T and an occurrence containing a pointer to position i in T. We can update the structure after deletion of a block of b characters in $O((h'^2 + bh') \log n)$ time, where h' is the maximum of $h(T, i)$ and $h(T', i)$ over the positions i that were affected by the insertion or deletion and T' is the final string. Thus, the update is efficient unless it occurs inside a very large section of the text that is repeated very many times. Updates only take significant time when they occur in regions of the text that have very low entropy. Indeed, the proposed structure has a relationship to a structure used in the Lempel-Ziv data compression algorithm [6].

An even stronger result is that moving a consecutive block of b characters from one place to another in the text takes $O((h(T)^2)$ time, independently of the number b of characters in the block. This is a common editing operation and the correction to a common type of error in databases of genetic sequences.

2   Preliminaries

Let λ be the null string. If $X = x_1 x_2 ... x_k$ is a string, we let $||X||$ denote the length k of x. The **reverse** of X is the string $X^R = x_k x_{k-1} ... x_1$. If Y is a prefix $x_1 x_2 ... x_i$ of X, let $X - Y$ be the result $x_{i+1} ... x_k$ of removing Y from the front of X.

For reasons that will become clear shortly, we adopt the convention of numbering the positions of the text T from right to left, so $T = t_n t_{n-1} ... t_1$. Let T_i denote the suffix $t_i t_{i-1} ... t_1$ beginning at position i. Let us distinguish a **substring** $P = p_1 p_2 ... p_m$ of a T from an **instance** (P, i) of P in T, where $P = t_i t_{i-1} ... t_{i-m+1}$. The null substring, λ, is considered to occur at every position.

Definition 1. *A* **trie** *on alphabet Σ denotes a rooted tree T with the following properties:*

1. *Each edge is labeled with a character;*
2. *For each node u and letter $b \in \Sigma$, there is at most one edge with label b from u to a child of u.*

Given a trie, let us say that the **label of a path** from the root to a node u is the string given by the sequence X of characters that occur on edges of the path. This is the **path label** of u. Because of the second property, the path label uniquely identifies u. We therefore adopt the convention of treating the node and its path label as interchangeable objects. For example, we may consider whether a *string X* is a *node* of the trie, or whether one *node* is a *substring* of another. Note that one node is a prefix of another if and only if it is an ancestor in the trie.

A basic operation on a trie takes an input string $P = p_1 p_2 ... p_m$ and finds the largest prefix P' of P that is a node of the trie. Since $|\Sigma|$ is fixed, this is easily accomplished in $O(||P'||)$ time by starting at the root and iteratively taking edges labeled with the sequence of letters from P, until P is exhausted or a node is encountered that doesn't have a child on the next letter of P. Let us call this operation **indexing** into the trie.

3   Sequence Hash Trees

A data structure of Coffman and Eve [7], called a **sequence hash tree**, was designed for the problem of implementing hash tables (dictionaries) whose keys are strings. It consists of a trie for indexing into the table. The structure of the tree depends on the order in which the strings are inserted. We describe a small variant that is easier to adapt to our substring matching problem, below.

Let $\mathcal{S} = (S_1, S_2, \ldots S_n)$ be a given ordering of the strings. Without loss of generality for our purposes, we may assume that no string is $\mathcal{S}$ is a prefix of any other. The trie that they construct is defined by induction, as follows. If $i = 1$, the trie H_1 is just a root node with a pointer to S_1. If $i > 1$, then H_i is obtained from H_{i-1} by finding the shortest prefix Xb of S_i that is not already a node of the trie. A new node Xb is added as the child of node X on edge labeled b, and a pointer is installed from it to S_i.

Figure 1 gives an example. To find an occurrence of a string S in the hash table, they index into the trie $H_n = H$ on the longest prefix X of S that is a node of H. For each node on the path from the root to X, they check whether the hash-table entry the node points to matches S.

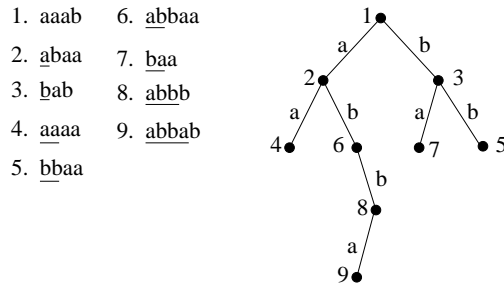|	
1. aaab	6. a̲b̲baa
2. a̲baa	7. b̲aa
3. b̲ab	8. a̲b̲b̲b
4. a̲a̲aa	9. a̲b̲b̲ab
5. b̲baa	

Fig. 1. The sequence hash tree of a set of strings. Each string is installed at the shortest prefix that isn't already a node of the sequence hash tree. The shape of the tree depends on the order in which the strings are inserted.

Their algorithm for deleting string from the tree consists of finding a leaf descendant Y of the node X that points to the string, copying Y's pointer to X, and deleting Y. The insertion algorithm indexes into the new string S until it reaches a null pointer, creates a new leaf child X on that pointer, and makes

it point to S. Both of these operations take time proportional to the depth of the tree.

4 Contracted Suffix Trees

We consider how to apply Coffman and Eve's sequence hash trees to our problem of finding all occurrences of P in T. We build the sequence hash tree for the set of *suffixes* of T, and record, in the node generated by each suffix, a pointer to where the suffix begins in T. P occurs at location i in T if and only if it is a prefix of suffix T_i. We append a special character \$ to the end of T to ensure that no suffix of T is a prefix of another. Instead of looking for an entry that P matches, we look for all entries that P is a prefix of; the locations where these suffixes begin in T gives the locations where P occurs.

   Coffman and Eve's paper has received little attention since it was published in 1970, due, in no doubt, to the existence of superior ways of implementing a hash table. In the present paper, we show that this data structure is a much richer when considered in the context of the new problem. The structure of the set of suffixes of a text T allows us to derive interesting and algorithmically useful properties that do not apply in the general case addressed by Coffman and Eve. In particular, we show that it has height at most $h(T)$, and show that if the suffixes are inserted in ascending order of length, it is now possible to build the data structure in time that is linear in $n = ||T||$, that is, in $O(1)$ time, amortized, per hash key. We show how the tree can be augmented with *maximal reach pointers* so that finding all k entries that have P as a prefix takes $O(m+k)$ worst-case time, independently of the height of the tree.

   The $h(T)$ height bound gives an $O(h(T))$ bound for Coffman and Eve's operations for inserting and deleting a suffix. This gives the $O((b + h(T))h(T))$ bound for updating the data structure when a consecutive block of b characters is deleted and the $O((h(T))^2)$ bound for moving a block of b characters from one place in T to another.

   To distinguish this special case of their structure, we will call it the **contracted suffix tree**; in contrast to the suffix tree, which has a path for each suffix, the contracted suffix tree only has paths for prefixes of the suffixes.

4.1 A naive query algorithm for contracted suffix trees

In this section, we give the **naive query algorithm**, which determines the k occurrences of P in T in $O(m^2 + k)$ time, where m is the length of P. Below, we show how to improve this to $O(m + k)$. As explained below, the $O(m^2)$ component is overly pessimistic in practice, however, and the naive approach takes $O(m + k)$ expected time on random strings. It may be competitive with the $O(m + k)$ worst-case approach for many applications, due to its simplicity and lower space requirements.

Lemma 1. *If P is not a node of a contracted suffix tree of T, it has fewer than $||P||$ occurrences in T.*

Proof. Every suffix of T that has P as a prefix results in a new node of the tree that is either a proper prefix of P or that has P as a prefix. Since P does not occur in the tree, it is not a prefix of any node in the tree. Therefore, the number of suffixes of T that have P as a prefix, hence the number of occurrences of P, is bounded by the number of proper prefixes of P.

Lemma 2. *The height of a contracted suffix tree of T is at most $h(T)$.*

Proof. Let $X = x_k x_{k-1} \ldots x_1$ be a deepest leaf of the tree. Let X_i denote the prefix $x_k x_{i-1} \ldots x_i$ of X. For each i from 1 through k, X_i occurs at least i times in T because it has at least i descendants, $\{X_i, X_{i-1}, \ldots X_1\}$, and each of these points to an occurrence of a substring of which X_i is a prefix. Therefore, $X_{\lceil k/2 \rceil}$ has length $\lfloor k/2 \rfloor$ and occurs at least $\lceil k/2 \rceil$ times in T. It must be that $k/2$ is a lower bound on $h(T)$, so the height k is $O(h(T))$.

A node X contains a pointer to a position i where X occurs. It follows that if P is a node of the contracted suffix tree, the positions in T pointed to by ancestors of P may or may not be occurrences of P, while the positions pointed to by descendants of n are all occurrences of P. This gives a simple algorithm for a special case of the query:

– **Case 1** *P is a node in H.*
  The applicability of the case is detected by indexing into H on P. For each ancestor (prefix) X' of P, determine whether P occurs at the position i pointed to by X'. In addition to these, report positions contained in all descendants of P. (See Figure 2.)

For each ancestor X' of P, it takes $O(m)$ time in the worst case to check whether P occurs at the position pointed to by X'. There are at most m ancestors of P, so the total time spent on these checks is $O(m^2)$. Finding the k' descendants of x takes $O(k')$ time, since no checking is required. The total is $O(m^2 + k)$.

– **Case 2:** *P is not a node in H.*
  Let X be the longest prefix of P that is a node in H. For each ancestor (prefix) X' of X, determine whether P occurs at the position i pointed to by X', and report the ones that do. (See Figure 2.)

Since Case 2 applies, X is a proper prefix of P. Let b be the character that follows this instance of X in P. Since X has no child on edge labeled b, any descendant of X contains a pointer to a position that is an occurrence of Xc for $c \neq b$. Therefore, no descendant of X is an occurrence of P, and we may drop the k from the $O(m^2 + k)$ bound we got in Case 1 to get an $O(m^2)$ bound.

4.2   A naive construction algorithm

Coffman and Eve' algorithm for inserting an entry to the sequence hash tree takes $O(h)$ time, where h is the height of the tree. By Corollary 2, we may use this algorithm to insert each suffix of T, yielding a contracted suffix tree in $O(nh(T))$ time.
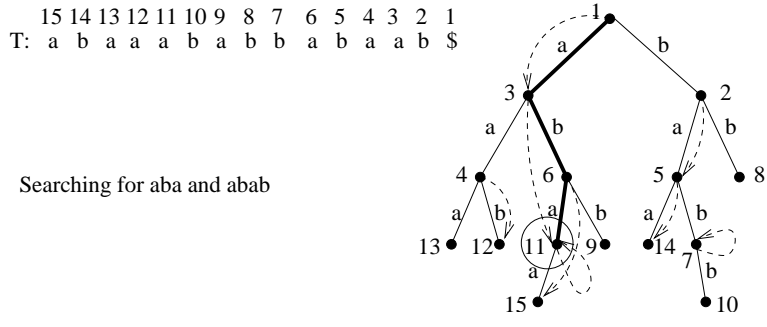
Fig. 2. The string *aba* is the node labeled with a pointer to position 11, so it falls into Case 1. The position labels of ancestors of this node, $\{1, 3, 6\}$ may or may not be occurrences of *aba*, and this must be checked. A naive approach checks whether the occurrence of *ab* at position 6 is followed by an *a* and the occurrence of *a* at position 3 is followed by *ba*, and the occurrence of the null string at position 1 is followed by *aba*. This approach takes $O(m)$ time per ancestor. This can be improved to $O(1)$ time per ancestor by checking whether the *maximal-reach pointer* of the ancestor (dashed arrows) points to a descendant of node *aba* (Section 5). The position labels of descendants of *aba* $\{11, 15\}$ must be occurrences of *aba*, and do not have to be checked. The string *abab* falls into Case 2. Node *aba* is the largest prefix that is a node of the tree. Only the ancestors $\{1, 3, 6\}$ of this node can be occurrences of *abab*. This can be checked by verifying that the maximal reach pointer points to *aba*, indicating an occurrence of *aba* not followed by *a*, and, if so, checking whether the occurrence is followed by *b*.

5   An $O(m + k)$ bound for searches

At a node X pointing to position i of T, let Y be the longest prefix of T_i that is a node of a contracted suffix tree H of T. Clearly, X is a prefix of Y, though it is possible that it is not a proper prefix. We install a **maximal reach pointer** from node X to node Y. (See figure 2.)

Lemma 3. *The position i in an ancestor X of P is an occurrence of P if and only if X's maximal reach pointer points to a (not necessarily proper) descendant of P.*

Proof. The nodes of the tree that have P as a prefix are the descendants of P. If X's maximal reach pointer points to a descendant of P, then T_i has P as a prefix, and P occurs at position i. If X's maximal reach pointer does not point to a descendant of P, then since P is a node of the tree, P is not a prefix of T_i, which means that P does not occur at position i.

The **naive algorithm** for installing the maximal-reach pointers is to revisit each position i of T after the suffix heap tree H has been built, indexing into H on the suffix beginning at i, passing through the node containing i, and stopping when a node encountered that has no child on the next letter of T. This is the node that must be pointed to by the node containing position i. Since the height of H is $O(h(T))$ this gives an $O(nh(T))$ algorithm for installing the maximal-reach pointers, which adds nothing to the asymptotic time bound for building H with the naive construction algorithm.

After we construct the tree, we perform a depth-first search of the tree to label each node with a discovery and finishing time, as described in [8]. These are essentially preorder and postorder numbers. Their purpose is to allow us to determine, given nodes X and Y, whether X is an ancestor of Y; this is the case if and only if X has an earlier discovery time and a later finishing time than Y does. We also keep a pointer from each position in T to the node of H that points to it.

Case 1 queries. As before, to find the k' occurrences of P listed in descendants of P, we visit P's subtree in $O(k')$ time. The difference now is that we can determine at each ancestor X' of P whether the position i it contains is an occurrence of P by checking whether the maximal reach pointer of X' points to a (not necessarily proper) descendant of P. This test takes $O(1)$ time using the preorder and postorder numbers, giving the $O(m)$ bound for finding the remaining occurrences of P that its ancestors point to.

Case 2 queries. For Case 2, there are $O(m)$ occurrences of P by Lemma 1. We partition P into substrings by finding the maximal prefix P_1 of P that is a node of H, then the maximal prefix of $P - P_1$ that is a node of H, etc., yielding $(P_1, P_2, \ldots, P_k)$.

Since we are in Case 2, there are $k \geq 2$ strings in this sequence. Since $||P_1||$ is the longest prefix of P that is a node of the tree, any proper descendant X of P_1 fails to be a prefix of P, hence the position at X is not an occurrence of P. Only (not necessarily proper) ancestors of P_1 can contain pointers to occurrences of P. Since P_1 is a prefix of P, only nodes that point to an occurrence of P_1 are candidates to be occurrences of P. An ancestor is a candidate if and only if its maximal-reach pointer points to P_1; if its maximal reach pointer points to a proper descendant of P_1, it points to a node that is not a prefix of P.

By induction on j, we now find which candidate locations are occurrences of $P_1 P_2 \ldots P_j$, and show that there are $O(||P_j||)$ of them, as follows. If $j = 1$, the candidate positions are the $O(||P_1||)$ candidates described above. If $1 < j < k-1$, the candidate positions are the positions of $P_1 P_2 \ldots P_j$, and we assume by induction that there are $O(||P_j||)$ of them. We test for each candidate position i whether i is an occurrence of P_{j+1} by checking whether $i' = i - ||P_1 P_2 \ldots P_j||$ is an occurrence of P_{j+1} that's followed by $P_{j+2} P_{j+3} \ldots P_k$. We do this by determining whether i' occurs at a (not-necessarily proper) ancestor Y of P_{j+1} and Y's maximal reach pointer points to a (not-necessarily proper) descendant of P_{j+1}. This takes $O(||P_j||)$ time using the preorder and postorder numbers, and since there are $O(||P_{j+1}||)$ ancestors of P_{j+1}, it yields $O(||P_{j+1}||)$ positions. If $j = k-1$, $P_{j+2} P_{j+3} \ldots P_k$ is empty, so we also test whether i' occurs in a descendant of $P_{j+1} = P_k$.

The time bound for finding all occurrences of $P = P_1 P_2 \ldots P_k$ is thus $O(||P_1|| + ||P_2|| + \ldots + ||P_k||) = O(m)$.

6 Building a contracted suffix tree in $O(n)$ time

We begin by installing the text in an array of characters in $O(n)$ time, so that we can access the letter in any position number i in $O(1)$ time.

Let $H(T)$ denote the unique result of inserting the suffixes of T into the contracted suffix tree in ascending order of their length. This special case of the contracted suffix tree is called the **position heap** [9].

One advantage of the position heap is that it can be used to produce in $O(||P||)$ time an iterator on a list of positions in order in which they occur in the text, at a cost of spending $O(\log k)$, rather than $O(1)$, for returning the next element of the list. The positions are in heap order, that is, the position at each node is smaller than the positions at its children. In Case 1, the positions in ancestors of P therefore occur in sorted order, and for the descendants, a priority queue can be used to manage the topmost descendants that haven't already been returned. In Case 2, the candidate positions are found at ancestors of P_1, which occur in sorted order, and elements are subsequently deleted from this list to give the list of occurrences of P. This gives them in right-to-left order; if left-to-right order is desired, the position heap for the reverse of the text can be used.

Let the **dual** $D(T)$ of the position heap $H(T)$ be the trie where for each node X of $H(T)$, the **reverse** X^R of X is a node of $D(T)$ (see Figure 3).

It is tempting to think that the dual is just the position heap of the reverse of the text, but it is easily verified that this is not the case.
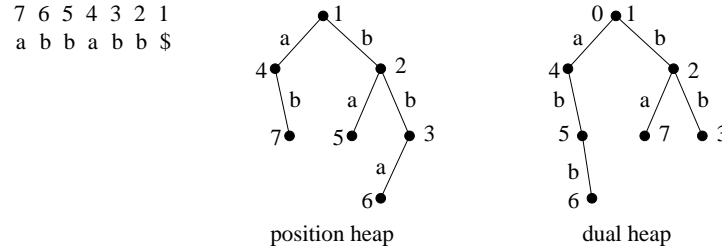

Fig. 3. The position heap and its dual for the text *abbabbb*. The labels of the path leading to a node in the dual is the reverse of the labels of the path leading to it in the position heap.

Let us say that a set of S of strings is **hereditary** if, whenever $X \in S$, every substring of X is also in S.

Lemma 4. *The nodes of the position heap are a hereditary family of strings.*

Proof. Let us show this by induction on the length of $T_i = t_i t_{i+1}...t_1$. The lemma is trivially true for $H(T_1)$, which has only one node, the empty string. Otherwise, we adopt as the induction hypothesis that the nodes of $H(T_{i-1})$ have the hereditary property. Since $H(T_i)$ differs from $H(T_{i-1})$ only by the addition of a node X, $H(T_i)$ can only fail to have the hereditary property if some proper substring of X fails to be a node of T_i.

This can't be the case if $||X|| < 2$, since λ is a node of $H(T_i)$. Suppose $||X|| \geq 2$. We can then write X as $aX'b$. The parent of $aX'b$ is aX', hence it is a node of $H(T_{i-1})$. Since aX' is longer than X', X' is a node in T_{i-2}. Also, $X'b$ is a prefix of T_{i-1}, and since X' is a node of T_{i-2}, $X'b$ is either added at step $i-1$ or is already a node of T_{i-2}. In either case, it is a node of $H(T_{i-1})$. We conclude that aX' and $X'b$ are nodes of T_{i-1}. By the induction hypothesis, every substring of aX' and $X'b$ is a node of T_{i-1}, hence of T_i, and these are every proper substring of the new node $X = aX'b$.

The lemma is not true for sequence hash trees: the substring *bba* of the node *abba* labeled 9 in Figure 1 is not a node of the tree.

Corollary 1. *The set of nodes of $D(T)$ is the same as the set of nodes of $H(T)$.*

Proof. By definition, every node of $H(T)$ is a node of $D(T)$. It remains to show that every node of $D(T)$ is a node of $H(T)$. Let X be an arbitrary node of $H(T)$. By Lemma 4, not only is every prefix of a node X of $H(T)$ a node of $H(T)$, but so is every suffix. This implies that every ancestor of X in $D(T)$ is a node of $H(T)$. There are no nodes on any path of $D(T)$ that fail to be a node of $H(T)$.

We implement $H(T)$ and $D(T)$ on the same set of nodes, so that each node has a parent in $H(T)$ and a parent in $D(T)$. We continue to refer to each node by its path label X in $H(T)$, even when considering it as a node of $D(T)$. Equivalently, each node of $D(T)$ is denoted by the sequence X of labels on edges from the node to the root of $D(T)$.

We get an $O(n)$ time bound for constructing the position heap by simultaneously constructing the position heap and its dual. During construction, we need the edges to go from child to parent in the position heap and from parent to child in the dual. After the tree is constructed, the dual heap can be discarded and the edges of the position heap can be reversed in $O(n)$ time, to go from parent to child, by bucket sorting edges according to destination vertex.

When going from $H(T_{i-1})$ to $H(T_i)$, a new node must be added to hold a pointer to position i. Let a be the first letter of T_i and let X be the node added at step $i-1$. If position i is the first time a was encountered, we add a new child of the root on edge labeled a in both the position heap and the dual. Otherwise, let aX' be the longest prefix of T_i that is a node of $H(T_{i-1})$. Because X was added at step $i-1$, it follows from the hereditary property that aX was not a node of $H(T_{i-1})$. Therefore, X' is a proper prefix of X. We find X' by ascending through proper ancestors (prefixes) of X in the position heap. X' is the first one that has a child on edge labeled a in the dual.

Let $b = t_{i-||aX'||}$. Since the text is in an array of characters, we may look up b in $O(1)$ time. According the constructive definition of H, the new node in H is $aX'b$. In H, this is the child of aX' on edge labeled b. According to the definition of the dual, $aX'b$ is also the new node of the dual. It must be the child of $X'b$ on edge labeled a. $X'b$ is already known: it was the child of X' encountered on the path from X to X'.

Let us now show that iterating this procedure from 1 through n gives an $O(n)$ time bound for finding $H(T)$ and $D(T)$. We use an amortized analysis to bound this cost over all iterations. The only difficulty is bounding the amount of time traversing the path from X up to X'. Let the *current depth* be the depth of the last node added. The key to the analysis is that each node traversed on this path decreases the current depth by one, and adding the new node $aX'b$ then increases the depth by two. Since the initial depth is 0, the total number of times the depth decreases is bounded by the number of times it increases, which is $O(n)$.

The procedure for adding the maximal-reach pointers to $H(T)$ in $O(n)$ time is similar. We install maximal reach pointers in nodes in the same order in which those nodes were added to $H(T)$. This time, we let the current depth be the depth of the node X pointed to by the next node's maximal reach pointer. The next node pointed to is obtained by searching upward from X to find the lowest ancestor X' such that aX' is a node; this is the node pointed to by the next maximal-reach pointer. The amortized analysis of the running time is the same as it is for the construction.

7 Dynamic Texts

When T changes dynamically, we can no longer assume that the characters of T are in an array.

The nodes of the contracted suffix tree were previously labeled with discovery and finishing times in a depth-first search on the tree. We replace these with pointers into a data structure for dynamic ordered lists that is suitable for looking up in $O(\log n)$ time which of two elements is earlier in the list. The order of elements in this list are the discovery and finishing times of nodes in the current tree. A balanced binary tree where the elements appear in inorder suffices, for example. Each node points to the two elements corresponding to its discovery and finishing times. When a new node is created, its discovery-time element is inserted immediately after the finishing-time element of its left neighbor, or of its parent if it has no left neighbor. Its finishing element is handled symmetrically. When it is deleted, its discovery- and finishing-time elements are simply removed from the list. This list allows one to determine whether one node is an ancestor of another in $O(\log n)$ time, rather than $O(1)$ time. This raises the worst-case time bound for finding which ancestors of P point to occurrences of P from $O(1)$ to $O(\log n)$. This increases the worst-case bound for finding all k occurrences of P from $O(m + k)$ to $O(m \log n + k)$.

Similarly, the text must be implemented with a data structure that allows insertion and deletion of blocks of text in $O(\log n)$, and indexing the character that is currently in position i. This can be carried out with similar schemes; one simple scheme which takes $O(\log n)$, amortized, is based on splay trees, and is described in [10]. If this structure is used, finding all k occurrences of P takes $O(m \log n + k)$, amortized.

Figure 4 shows how to modify the tree when a single character is deleted, in the case the b at position p_6. Each underline under the text represents the occurrence of a node X pointed to X. We can remove p_6 with Coffman and Eve's deletion operation, which replaces it with a leaf descendant, in this case p_{12}. However, p_7 is at node aba, which indicates that there is an occurrence of aba, represented by an underline, which extends from position p_7 to p_5. This is no longer the case, because the middle letter of aba is the b that was deleted at position p_6. We must therefore remove p_7, because it is at a node that is no longer a prefix of the suffix that begins at p_7. Similarly, we must remove p_8, which resides at aab, whose last letter is the deleted one. Let us call positions such as p_7 and p_8 that are not deleted but must be moved to a new location the *affected positions*.

The middle tree of Figure 4 shows the tree after the deleted and affected positions have been removed. We then reinsert the new suffixes beginning at the affected positions with Coffman and Eve's insertion operation.

Since each node is a string of length $O(h(T))$, there are $O(h(T))$ affected positions, and each one takes $O(h(T) + \log n)$ amortized time to remove. The $\log n$ time comes from the need to remove and insert its discovery and finishing times in the dynamic lists (and the $O(\log n)$ amortized bound comes if you use the splay-tree data structure in [10] for representing the text). $O(h(T) + \log n)$
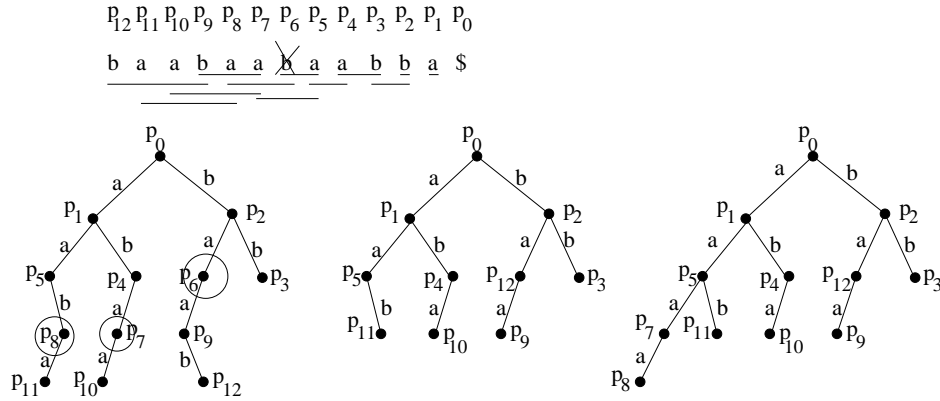
$p_{12}\ p_{11}\ p_{10}\ p_9\ p_8\ p_7\ p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0$

b a   a   b   a   a   b   a   a   b   b   a $

Fig. 4. Modifying a contracted suffix tree when a character is deleted from T.

amortized is $O(h(T))$, amortized, since $\log n = O(h(T))$. Similarly, each affected position takes $O(h(T'))$ amortized time to insert, where T' is the revised text. An easy way to see that $h(T')$ is $O(h(T))$ is to observe that the insertion of the $h(T)$ positions can add at $O(h(T))$ to the height of the middle tree. This gives a total of $O([h(T)]^2)$ time to delete a character. Insertion of a character works similarly and has the same time bound.

When a consecutive block of b characters is deleted, there are still $O(h(T))$ affected characters that precede the b characters of the block. Each of the b deleted characters and the $O(h(T))$ affected characters takes $O(h(T))$ amortized time to process, so the time to delete b consecutive characters from T is $O((b + h(T))h(T))$, amortized. Similarly, the insertion of b consecutive characters takes $O((b + h(T'))h(T'))$ amortized time, where T' is the new text that results from the insertion.

To get the $O([h(T)]^2)$ amortized bound for moving a consecutive block of b characters from one position to another in the text, we observe that we only need to move affected positions. These are $O(h(T))$ positions at the end of the block, $O(h(T))$ positions preceding the initial location of the block, and $O(h(T))$ positions preceding the new location of the block. The remaining positions in the block are at nodes that continue to be prefixes of the suffixes that they point to, so they don't need to be moved.

References

1. Weiner, P.: Linear pattern-matching algorithms. In: Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory. Institute of Electrical Electronics Engineers, London (1973) 1–11
2. Blumer, A., Blumer, J., Ehrenfeucht, D., Haussler, D., McConnell, R.: Complete inverted files for efficient text retrieval and analysis. Journal of the ACM **34** (1987) 578–595

3. Manber, U., Myers, E.: Suffix arrays: a new method for on-line search. SIAM J. Comput. **22** (1993) 935–948
4. Ferragina, P., Grossi, R., Montangero, M.: On updating suffix tree labels. Theor. Comput. Sci. **201**(1-2) (1998) 249–262
5. Salson, M., Lecroq, T., Lonard, M., Mouchard, L.: Dynamic burrows-wheeler transform. Theoretical Computer Science (2009) accepted.
6. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory **24** (1978) 530–536
7. Coffman, E., Eve, J.: File structures using hashing functions. Communications of the ACM **13** (1970) 427–432
8. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. Mc-Graw Hill, Boston (2001)
9. Ehrenfeucht, A., McConnell, R.M.: String searching. In Mehta, D., Sahni, S., eds.: Handbook of Data Structures and Applications. CRC Press (2005)
10. Tarjan, R.E.: Data structures and network algorithms. Society for Industrial and Applied Math., Philadelphia (1983)