Faster Suffix Tree Construction with Missing Suffix Links*

Richard Cole
Courant Institute, NYU
cole@cs.nyu.edu

Ramesh Hariharan
Indian Institute of Science
ramesh@csa.iisc.ernet.in

ABSTRACT

We consider suffix tree construction for situations with missing suffix links. Two examples of such situations are suffix trees for parameterized strings and suffix trees for 2D arrays. These trees also have the property that the node degrees may be large. We add a new back-propagation component to McCreight's algorithm and also give a high probability perfect hashing scheme to cope with large degrees. We show that these two features enable construction of suffix trees for general situations with missing suffix links in $O(n)$ time, with high probability. This gives the first randomized linear time algorithm for constructing suffix trees for parameterized strings.

1. INTRODUCTION

The *Suffix Tree* of a given string of length n is the compacted trie of all its suffixes. This tree has size $O(n)$ and can be constructed in $O(n)$ time [9; 13; 12]. Suffix trees have several applications (see [6]). One of the main applications of suffix trees is to preprocess a text in linear time so as to answer pattern occurrence queries in time proportional to the length of the query and independent of the length of the preprocessed text. The preprocessing involves building the suffix tree for the text. Next, given a query pattern, the unique path down the suffix tree traced by this pattern is determined; each leaf of the tree which lies further down from this path corresponds to an occurrence of the pattern.

Parameterized Suffix Trees. Baker [1] generalized the definition of suffix trees to *parameterized* strings, i.e., strings having variable characters or *parameters* in addition to the usual fixed *symbols*. The set of parameters and the set of symbols are disjoint. Two parameterized strings are said to *match* each other if the parameters in one can be *consistently* replaced with the parameters in the other to make the two string identical. Here, consistency demands that all occurrences of a particular parameter are replaced by the same parameter and distinct parameters are replaced by distinct parameters. Baker [1] gave a definition of suffix trees for parameterized text strings t so as to facilitate answering pattern occurrence queries in time independent of the text length $|t|$.

2D Suffix Trees. Giancarlo [5] generalized suffix trees to two-dimensional texts t in order to answer pattern occurrence queries (i.e., find all occurrences of a given square array p in the square text t) in time independent of $|t|$.

Suffix Tree Construction. There are several algorithms for constructing the suffix tree of a string drawn from a constant-sized alphabet set in $O(n)$ time. These include the algorithms by McCreight [9], Weiner [13] and Ukkonen [12]. All these algorithms exploit an important property of suffix trees, namely, each node has an outgoing suffix link.

Farach [3] showed how to construct suffix trees in $O(n)$ time even when the alphabet size was not constant but some polynomial in n. This algorithm differs from the others above in that it is not sweep-based and seems to be less critically dependent on the existence of outgoing suffix links. However, it requires renaming pairs of adjacent characters to get a string of size half that of the original string. The suffix tree for this smaller string is built recursively; Farach shows how the suffix tree of the original string can be obtained from the suffix tree of this smaller string in $O(n)$ time.

In contrast to suffix trees for strings, suffix trees for both parameterized strings and 2-D arrays lack the suffix link property, i.e., there could be nodes in the tree without an outgoing suffix link defined. In addition, the node degrees in these suffix trees need not be bounded by a constant. Due to these two problems, the best constructions known until recently for suffix trees for parameterized strings [1] and 2-D arrays [5] took $O(n \log n)$ time, where n is the size of the input string/array. In each case, the problem of missing suffix links was handled by using a dynamic tree data structure [11]; this data structure is used to find the insertion site of the next suffix in $O(\log n)$ time. Further, the problem of large node degrees was handled by the standard approach of maintaining a binary search tree, which also gave a $\Theta(\log n)$ overhead.

Recently, Kim and Park [8] used the paradigm of Farach [3] to give an $O(n)$ time algorithm for 2-D suffix tree construction (for polynomially bounded alphabet size). However, it is not clear how to apply this paradigm to the case of parameterized strings. In particular, it is not clear how the renaming of pairs of adjacent characters mentioned above

*This work was supported in part by NSF grants CCR-9503309 and CCR-9800085.

can be accomplished in such a way that the suffix tree of the given string can be obtained from the suffix tree of the renamed string in $O(n)$ time.

Our Contribution. We present two new tools in this paper.

(i) The first tool is aimed at tackling the problem of missing suffix links. We augment McCreight's algorithm with a new feature which copies nodes backwards (imagine suffix links as going forwards), thus adding additional nodes and suffix links to the suffix tree. Using a non-trivial accounting procedure, we show that this back-propagation adds only $O(n)$ extra nodes and accomplishes the construction of the suffix tree in $O(n)$ time even with missing suffix links. The back-propagation is similar to fractional cascading, as used in many pointer based data structures of bounded degree (when viewed as graphs); the difficulty here is that the degrees are potentially unbounded, which appears to necessitate a quite different analysis.

(ii) The time analysis in (i) assumes that given a node x and a character a, the unique edge from x to a child of x starting with the character a is computable in $O(1)$ time. To enable this for high degree nodes x, we give an extension of the dynamic version of the Fredman-Komlos-Szemeredi perfect hashing scheme [4] which supports insertions of items from a polynomial sized range in amortized constant time and linear space, with high probability (as compared to the previous expected time result of Dietzfelbinger et al. [2]). Searching for an item requires worst case constant time. In fact, the items being in a polynomial sized range is not necessary for our hashing scheme; it suffices if they can be hashed into a polynomial sized range in linear time.

The above two tools provide a unified framework from which randomized $O(n)$ time algorithms for constructing suffix trees for regular strings, parameterized strings and 2-D arrays are easily derived. These algorithms work with high probability. This is the first $O(n)$ time randomized algorithm for parameterized suffix tree construction; the previous best algorithm[1] took $O(n \log n)$ deterministic time. The suffix trees we construct also have the property that the unique path in the tree corresponding to a given pattern string p can be found in $O(p)$ time, regardless of the degrees of the nodes.

2. THE GENERAL SETTING

Before describing our algorithm, we describe the general setting for which our algorithm works. We need the following definitions.

Compacted Trie. A compacted trie is a tree data structure defined on a collection of strings. This tree has one leaf per string in this collection, and each internal node has at least 2 children. Therefore, the number of nodes is linear in the number of strings in the given collection. Each edge of the tree is associated with (or labelled with) some substring of one of the strings in the given collection. The key property is that for every pair of leaves, the string formed by concatenating the edge labels on the path from the root to the least common ancestor of these two leaves is the longest common prefix of the strings associated with the two leaves.

In this paper, we are interested in compacted tries of certain kinds of string collections.

Quasi-Suffix Collections. An ordered collection of strings $s_1, s_2, \ldots, s_n$ is called a quasi-suffix collection if and only if the following conditions hold. Let $|s|$ denote the length of string s.

1. $|s_1| = n$ and $|s_i| = |s_{i-1}| - 1$. Therefore, $|s_n| = 1$.
2. No s_i is a prefix of another s_j.
3. Suppose strings s_i and s_j have a common prefix of length $l > 0$. Then s_{i+1} and s_{j+1} have a common prefix of length at least $l - 1$.

We will assume that all the strings are drawn from an alphabet of size polynomial in n.

Character Oracles. Note that the total length of the strings in a quasi-suffix collection of n strings is $O(n^2)$, while our aim is to achieve $O(n)$ time construction for the compacted trie. Therefore, we cannot afford to read the collections explicitly. Instead, we will assume an oracle which supplies the ith character of the jth string of the collection on demand, in $O(1)$ time.

Multiple Quasi-Suffix Collections. Consider m distinct quasi-suffix collections, each collection having at most n strings. These m quasi-suffix collections constitute a multiple quasi-suffix collection if conditions 2 and 3 above hold for any pair of strings s_i, s_j over all the m collections (in other words, these conditions hold for pairs within each collection and for pairs drawn from distinct collections as well).

Our main result will be the following.

THEOREM 1. *Let ϵ be any positive constant. The compacted trie of a quasi-suffix collection of n strings can be constructed in $O(n)$ time and space with failure probability at most $\frac{O(\log n)}{2^{\Theta(n^{1-\epsilon}/\log n)}}$, given the above character oracle. Further, the compacted trie of a multiple quasi-suffix collection comprising m quasi-suffix collections of at most n strings each can be constructed in $O(nm)$ time and space with failure probability at most $\frac{O(\log mn)}{2^{\Theta((mn)^{1-\epsilon}/\log mn)}}$.*

2.1 Examples of Quasi-Suffix Collections

The significance of the above theorem comes from the following examples of quasi-suffix collections. The simplest example is the collection of all suffixes of a string s with a special end-of-string symbol. This is a quasi-suffix collection but with a stronger property, namely, condition 3 in the definition of quasi-suffix collections is satisfied with equality. The compacted trie of these suffixes is the well-known suffix tree of the string s. Next, we give two more significant examples, for which equality need not hold in condition 3.

2.1.1 Suffix Trees for Parameterized Strings

Recall from the introduction that a parameterized string s has *parameters* and *symbols*. The alphabet from which parameters are derived is disjoint from the alphabet from which symbols are derived. Further, both alphabet sizes are polynomial in n, the length of s. As is standard, assume that s ends in a symbol \$ which does not occur elsewhere is s. From s, Baker [1] defined the following collection of strings.

Each suffix s' of s is mapped to a string $num(s')$ with parameters replaced by numbers and symbols retained as such (assume that symbols are not numbers). The replacement of parameters is done as follows. The first occurrence of each parameter in s' gets value 0 in $num(s')$. Subsequent occurrences of a parameter get values equal to the distance from the previous occurrence of the same parameter. Consider the collection of strings $\{num(s')|s' \text{ suffix of } s\}$ in decreasing length order. Baker [1] defined the suffix tree of parameterized string s to be the compacted trie of this collection. That this collection of strings is indeed a quasi-suffix collection can be seen as follows.

Condition 1 clearly holds and condition 2 follows from the occurrence of the special symbol \$ at the end of s. Condition 3 is shown to hold next. Note that if s'_i and s'_{i+1} are two consecutive suffixes of s, then $num(s'_{i+1})$ can be obtained from $num(s'_i)$ as follows: for each well-defined index $k > 0$, set $num(s'_{i+1})[k]$ to $num(s'_i)[k+1]$ if $num(s'_i)[k+1] \neq k$, and set $num(s'_{i+1})[k]$ to 0, otherwise. Next, consider two suffixes s'_i and s'_j of s. From the above observation, it follows that if $num(s'_i)$ and $num(s'_j)$ have a common prefix of length $k+1$, then $num(s'_{i+1})$ and $num(s'_{j+1})$ have a common prefix of length k. Further, if $num(s'_i)$ and $num(s'_j)$ differ at location $k+1$, then $num(s'_{i+1})$ and $num(s'_{j+1})$ could be identical at location k if one of $num(s'_i)[k+1], num(s'_j)[k+1]$ equals k and the other equals 0. Condition 3 is now easily seen to hold.

The character oracle for the above quasi-suffix collection is easily implemented in $O(1)$ time, after the following precomputation: for each occurrence of a parameter in s, determine the previous occurrence, if any, of this parameter in s. This precomputation is easily done in $O(n)$ time.

2.1.2 Suffix Trees for 2-D Arrays

Consider a 2-D array s having size $m \times n$, $m \geq n$, and characters drawn from some polynomial range. For each square subarray s' of s which is *maximal* (i.e., touches either the right boundary or the bottom boundary or both boundaries of s), Giancarlo [5] defined a string $num(s')$ as follows.

Defining $num(s')$. Partition s' into L's as in [5] (an L is formed by taking a prefix of a row and a prefix of a column, with the common point being at the bottom-right; both prefixes have equal lengths; the resulting shape is actually the mirror image of the character L). $num(s')$ will be a sequence of numbers, one number for each such L; these numbers are arranged in increasing order of L sizes. The number for a particular L is obtained by reading this L as a string and then mapping strings to integers in such a way that distinct strings map to distinct integers (by using, for example, the Karp-Rabin fingerprinting scheme [7], which ensures this property with inverse polynomial failure probability). Finally, append a special end-of-string symbol \$ to $num(s')$, as was done for parametrized strings.

The Quasi-Suffix Collections. Consider a particular top-left to bottom-right diagonal and consider all maximal square subarrays of s with top-left point on this diagonal. The $num()$ strings corresponding to these subarrays are easily seen to form a quasi-suffix collection. Thus each top-left to bottom-right diagonal gives a quasi-suffix collection of strings. Since there are $m + n - 1$ diagonals, we have $m + n - 1 = O(m)$ distinct quasi-suffix collections in all. It is easy to check that these $m + n - 1$ quasi-suffix collections together constitute a multiple quasi-suffix collection (we will use distinct end-of-string symbols for each diagonal to satisfy condition 2 for pairs of strings drawn from distinct collections). Note that the number of strings in each collection is at most n. Giancarlo [5] defined the common compacted trie of these $m+n-1$ collections to be the suffix tree of s.

The Character Oracle. A character oracle which works with inverse polynomial failure probability in $O(1)$ time after $O(mn)$ preprocessing is easy to implement using the Karp-Rabin fingerprinting scheme. The preprocessing involves computing prefix sums for each row and column.

2.2 Proving Theorem 1

The rest of the paper is devoted to proving Theorem 1. First, we will describe how to construct the compacted trie of a *single* quasi-suffix collection of n strings in $O(n)$ time with high probability. This algorithm can be extended to multiple quasi-suffix collections (like those resulting from 2D arrays) easily with an $O(m)$ multiplicative time and space overhead (assuming m collections of at most n strings each). This extension is sketched briefly in Section 6.

Our algorithm for a single quasi-suffix collection will have two components. The first component is a modification of McCreight's algorithm and is described in Section 4 and Section 5. In these sections, we will assume that the unique child of any given node with edge label beginning with a given character can be determined in $O(1)$ time. The second component, i.e., a dynamic perfect hashing scheme described below, will handle this problem.

Note that in all the above examples of quasi-suffix collections, the alphabet size is a polynomial in n. Thus to access the unique edge with a particular starting character from a node, we need to perfectly hash $O(n)$ pairs, where the first entry in the pair is a node number and the second entry is a character from the alphabet. Each such pair can be treated as a number from a range polynomial in n. We give a dynamic hashing scheme which will perfectly hash an item from a polynomial in n range in Section 7.

Before giving our algorithms, we need an outline of McCreight's algorithm for constructing the suffix tree of a string.

3. McCREIGHT'S ALGORITHM

The use of *suffix links* is crucial to this algorithm. Suffix links are defined as follows.

Definition. For a node x, let $str(x)$ denote the substring associated with the path from the root of the tree to x. A suffix link points from a node x to a node y such that $str(y)$ is just $str(x)$ with the first character removed. Let $link(x)$ denote this node y. Let $par(x)$ denote the parent of x. For a string u, define $node(u)$ to be that node x, if any, for which $str(x) = u$.

Since condition 3 in the definition of quasi-suffix collections is satisfied with equality for the collection of suffixes of a string, suffix links are defined for each node x in the suffix tree, i.e., for each node x, a node $y = link(x)$ with the above description exists.

McCreight's construction inserts suffixes into the suffix tree one by one, in order of decreasing length. For each suffix, one new leaf and possibly one new internal node are inserted. This sequence of insertions can be divided into phases. There are two kinds of phases, *scanning* and *rescanning*.

A scanning phase begins with a hitherto unread portion of the input string being scanned and a portion of some path

down the suffix tree being traversed, culminating in the insertion of a new leaf as a child of a new or existing internal node x_1. Portions of the suffix tree traversed in this scanning step are called *scanned* portions. Subsequent to this scanning, a rescanning phase begins in which a series of internal nodes $x_2, \ldots, x_k$ is inserted[1], without the text being scanned any further. x_i's suffix link points to x_{i+1}, $1 \le i \le k-1$; further, x_k's suffix link points to an already present node x_{k+1}. The next scanning phase begins by scanning the tree downwards from x_{k+1}.
The process of inserting x_{i+1}, following the insertion of x_i, $1 \le i \le k-1$, is as follows. The tree is *rescanned* downwards from $link(par(x_i))$ until the right position for x_{i+1} is found. Rescanning requires determining that path down the tree from $link(par(x_i))$ whose edge labels form the same substring as the label on the edge between $par(x_i)$ and x_i. Such a path is guaranteed to exist by condition 3 in the definition of quasi-suffix collections; further equality in this condition implies that the termination point of this path is the right location for x_{i+1}. Since the above path is guaranteed to exist, it suffices to examine just the first character on each edge to determine this path, as opposed to examining all the characters comprising the edge label; thus the term rescanning as opposed to scanning.

4. OUR ALGORITHM

As in McCreight's algorithm, we will insert the strings in the given collection $s_1, \ldots, s_n$ in the compacted trie in decreasing order of length. Much of the algorithm remains the same; however, we make two key modifications. The first involves traversing the path up the tree from a newly inserted node to find an ancestor with a suffix link. The second involves copying nodes backwards while rescanning down the tree from the destination of the above suffix link. These changes affect only the rescanning algorithm; the scanning part remains unchanged. We describe these changes in detail next.

Defining Suffix Links. For a node x, $link(x)$ is now defined to be that node y such that if $str(x)$ is a prefix of some s_i then $str(y)$ is a prefix of s_{i+1}; further $|str(y)| = |str(x)| - 1$. Note that since condition 3 in the definition of quasi-suffix collections need not be satisfied with equality, $link(x)$ need not be defined for every node x. Also note that if $link(x)$ exists then it is unique; this follows because if $str(x)$ is a prefix of s_i and of s_j then s_{i+1} and s_{j+1} agree in the first $|str(x)| - 1$ characters.

Backing Up. Recall the process of inserting x_{i+1}, following the insertion of x_i, $1 \le i \le k-1$, from our description of McCreight's algorithm above. Now, since $link(par(x_i))$ need not be defined, we must traverse up the tree from x_i until a node with a suffix link defined is found. We call this node $nanc(x_i)$ ($nanc$ stands for nearest ancestor). Next, the tree is rescanned downwards from $link(nanc(x_i))$, as before. See Figure 1.

Real and Imaginary Nodes. Recall from our description of McCreight's algorithm above that a rescanning phase terminates when it is found that the suffix link of x_k points to

[1]With each of these internal nodes, a child leaf corresponding to a new suffix is also inserted.

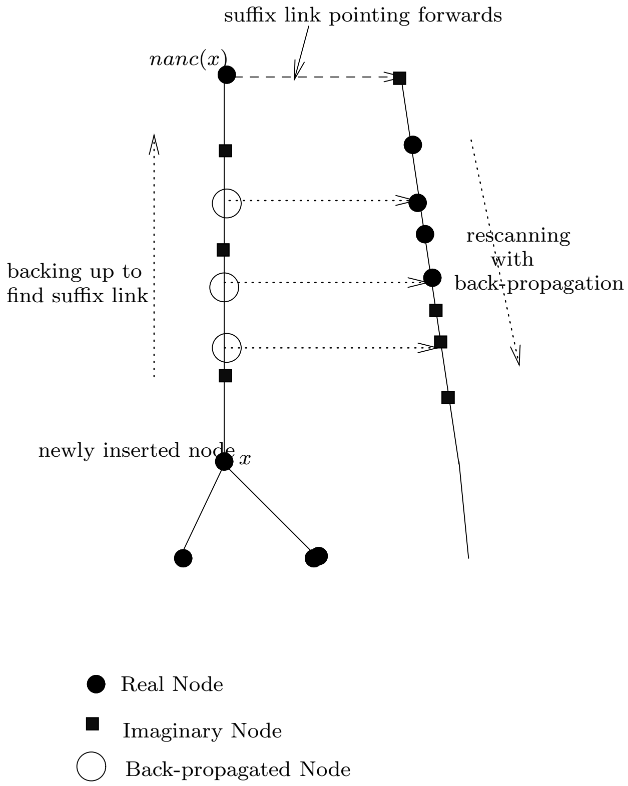

Figure 1: Backing up and Back-propagation

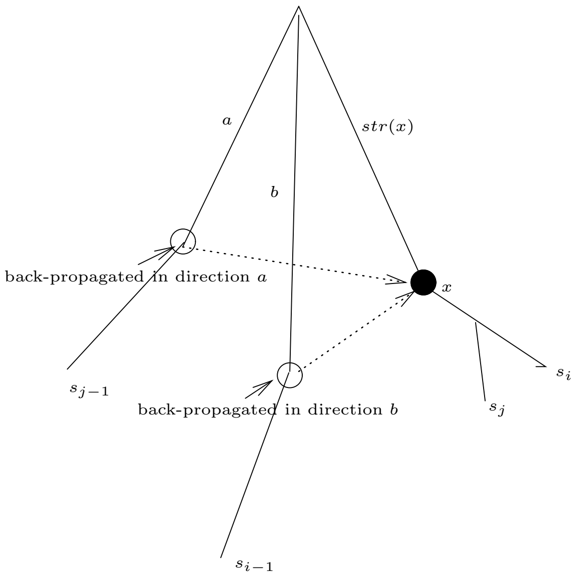

Figure 2: Direction of Back-propagation

an already existing node x_{k+1}. In our situation, the termination condition of a rescanning phase is different. A rescanning phase ends when it is found that either:

1. x_k's suffix link points to an already existing node x_{k+1}, or,

2. x_k's suffix link is not defined, i.e., it points to the middle of an edge.

We introduce an *imaginary node* x_{k+1} in the latter case; note that this imaginary node has only one child and does not have an outgoing suffix link. Internal nodes which are not imaginary will be called *real*; so $x_1 \dots x_k$ are real and have at least 2 children each. The current rescanning phase ends at x_{k+1} and a new scanning phase begins by scanning downwards from x_{k+1}.

Note that there are just $O(n)$ real nodes and $O(n)$ imaginary nodes (at most one real internal node, one leaf, and one imaginary node are inserted per suffix). Since real nodes have at least two children each, imaginary nodes have just one child each, and the number of leaves is n, the total number of children over all real and imaginary nodes is $O(n)$. Also note that the total length of the scanned portions of the tree in McCreight's algorithm in $O(n)$ and this remains the same for our algorithm. We state these facts below for future reference.

FACT 1. *(i) The number of real and imaginary nodes together is $O(n)$.*

(ii) The total number of children of real and imaginary nodes together is $O(n)$.

(iii) The total length of the scanned portions of the tree is $O(n)$ (the length of a single scanned portion is the number of characters, not nodes, encountered in the path scanned).

Back-Propagated Nodes. Other than real and imaginary nodes, our construction will involve internal nodes of a third kind, called *back-propagated nodes.* Back-propagated nodes will always have suffix links and only one child each. They are defined as follows. In the following, think of suffix links as pointing forwards (see Figure 1).
Suppose our algorithm has just inserted a node x. When the appropriate path starting an $link(nanc(x))$ is rescanned in order to determine the node $link(x)$, several nodes could be encountered in the process. If more than 2 nodes are encountered, then alternate nodes are propagated back to the path $(nanc(x), x)$ (i.e., new nodes with suffix links pointing to the traversed nodes are set up on this path), taking care that the first and the last nodes traversed are not propagated back. The new nodes are called back-propagated nodes.

Direction of Back-Propagation. Note that a node could be back-propagated in several different directions, i.e., several back-propagated nodes could have their suffix links pointing to this node. Further, a back-propagated node could be propagated backwards further, forming a chain of back-propagated nodes.

Definitions. For a node x, let $prev(x)$ be a set of strings defined as follows. For each s_i in the given quasi-suffix collection having prefix $str(x)$, $prev(x)$ contains the prefix of s_{i-1} of length $|str(x)| + 1$. Note that $prev(x)$ is a set and not a multiset; therefore all strings in it are distinct. *Direction a* is said to be *valid* for node x if string a appears in $prev(x)$. Node x is said to be *back-propagated in direction a* if there exists a string a in $prev(x)$ such that $node(a)$ exists and is a back-propagated node (see Figure 2). Note that the suffix link of $node(a)$ points to x under these conditions, i.e., $link(node(a)) = x$.

The following invariant is maintained by our algorithm by virtue of the fact that only alternate nodes encountered are back-propagated and the first and last nodes encountered are not back-propagated.

INVARIANT 1. *If a node x is back-propagated in direction a then its parent is not back-propagated in direction a', where a' is a prefix of a.*

5. TIME COMPLEXITY

There are two aspects to the time taken. The first involves backing-up from x to $nanc(x)$, subsequent to the insertion of x. The second involves rescanning the appropriate path down from $link(nanc(x))$ until $link(x)$ is located. We account for these two aspects of the time separately.
We make a few remarks on the second aspect here. Each step taken here involves one of the following:

1. Creating a new back-propagated node.

2. Adding a suffix link to an already existing node. This happens when one seeks to back-propagate a node but the site of this back-propagation is already occupied by some other node. For this to happen, the latter node must not have a suffix link, i.e., it must be an imaginary node. A suffix link is now added to this imaginary node.

3. Creating a new real or imaginary node. This is the node $link(x)$.

Since only one real or imaginary node is added when rescanning from $link(nanc(x))$ to $link(x)$, the time taken in this rescanning is proportional to $O(1)$ plus the number of nodes back-propagated in this process plus the number of imaginary nodes for which suffix links are set up in this process. Since each imaginary node can get only one suffix link during the course of the entire algorithm, bounding the above time boils down to bounding the number of back-propagated nodes by $O(n)$.

5.1 Bounding Back-Propagated Nodes

This will use a charging argument, where each back-propagated node will be charged to either some real/imaginary node, or to some character in the string s_1. Each real/imaginary node and each character in s_1 will be charged $O(1)$ in the process. The $O(n)$ bound will follow from Fact 1. It may be that a node created by back propagation subsequently becomes real or imaginary. These nodes are not counted; only nodes that are not real or imaginary when the full tree is built are counted.
Note that a back-propagation chain always starts at a real or an imaginary node. We will define a tree for each real or imaginary node x as follows.

Defining $BP{-}tree(x)$. All nodes in this tree other than the root x are back-propagated nodes. Those back-propagated nodes which are back-propagated from x (i.e., have suffix links pointing to x) are children of x in this tree. Trees rooted at these children are defined recursively, i.e., children of a node are those which are back-propagated from that node. The leaves of this tree are those nodes which are not back-propagated at all.

Consider the forest of $BP-Trees(*)$ rooted at the various real/imaginary nodes that are back-propagated. Each back-propagated node appears in exactly one tree in this forest.

Decomposing $BP-tree(x)$ into paths. We partition the nodes of this tree into paths. The first path is the minimal path starting from the root x and ending on a node y with the following property: either there exists a valid direction a such that y has not been back-propagated in this direction or there is no valid direction for y. Clearly, such a node y must exist. But for the termination restriction, the path starting at the root is chosen arbitrarily. Once nodes in this path are removed, the subtrees hanging off this path are decomposed recursively.

Clearly, each back-propagated node will belong to exactly one of the various paths formed above. Think of each path as going backwards from its start node.

Accounting for long paths. We show that the sum of the lengths of all the paths obtained above is proportional to the number of such paths plus $O(n)$.
Consider any path obtained above. Let x be any node on this path, other than its start node. $link(x)$ is the node from which x was back-propagated, in direction, say a. Note that $link(x)$ will precede x in the path being considered.
By Invariant 1, the parent $par(link(x))$ of $link(x)$ in the compacted trie has not been back-propagated in the direction a', where a' is the prefix of a such that $|a'|$ equals $|str(par(link(x)))| + 1$; a', of course, is a valid direction for $par(link(x))$ (because a is valid for $link(x)$ itself). It follows that either $par(link(x))$ is a real/imaginary node or $par(link(x))$ is a back-propagated node and the last node in its path. In either case, we charge $par(link(x))$ for x.
Clearly, each real/imaginary node and each back-propagated node which is the last node in its respective path will be charged an amount equal to the number of its children, in this process. From Fact 1(ii), this charge sums to $O(n)$ for real/imaginary nodes. Note that back-propagated nodes have only one child each. Thus, it now suffices to bound the total number of paths.

Bounding the total number of paths. We will extend the above paths backwards to form a collection of *extended paths*, as below.
Consider any one path, and let x be the last node on this path. The extension to this path is performed as follows. Start at x and follow that direction backwards along which x was not back-propagated (there is at least one such direction, unless there are no valid directions for x). Next, repeatedly follow any arbitrarily chosen valid direction backwards. This extension need not always encounter a node (in fact we will stop when we hit a node); it is allowed to cut through edges [2]. So if a particular step of this extension leads to the middle of an edge e, take an arbitrary valid direction back from that character on e. Continue this extension until either a node is reached or there is no valid direction left to go further.
Thus an extended path consists of an initial prefix of nodes (i.e., the path itself), followed by a walk which cuts through edges, and possibly terminates on a node. Again, note that we think of a path as going backwards. We have the following claims.

LEMMA 1. *Two distinct extended paths cannot intersect (i.e., they cannot cut through the same point on some edge or have a node in common), except that the last node of one can be the first node of the other.*

Proof. Since forward directions are always unique, two extended paths can intersect otherwise only if the start node of one path is contained in the other path and is not the last node on that path. This is a contradiction since all the unextended paths begin at nodes, the unextended paths are node disjoint, and the extension of a path terminates as soon as a node is reached. □

LEMMA 2. *If an extended path terminates by reaching a node y (and not by running out of valid directions), then y cannot be a back-propagated node.*

Proof. Let x be the last node of the path whose extension is under consideration. Suppose y is a back-propagated node. Then clearly, x must have been back-propagated in the direction implied by y (note that there is a unique valid direction for x along which it must be back-propagated, a unique valid direction along which the resulting node must be back-propagated, and so on, until y is reached). But we started the extension of this path by choosing a direction along which x was not back-propagated, a contradiction. □

LEMMA 3. *The total number of paths is $O(n)$, and hence the total number of back-propagated nodes is $O(n)$.*

Proof. Consider a particular extended path. If it ends at a node without running out of valid directions, this node must be real/imaginary, by Lemma 2; the current path is then charged to this node. By Lemma 1, each real/imaginary node is just charged once.
On the other hand, if this extended path ends because all further valid directions backwards are exhausted, then the substring associated with the termination point is a prefix of s_1. Further, by Lemma 1, different extended paths which end in this way are associated with distinct prefixes of s_1. Thus the number of paths is $O(n)$.
The lemma follows from the argument given earlier that the number of back-propagated nodes is proportional to the number of paths plus $O(n)$. □

5.2 Backing-up Time

It remains to account for the time taken to determine $nanc(x)$ after the insertion of x. Note that all such nodes x for which $nanc(x)$ will be determined are distinct real nodes (because each imaginary node starts a new scanning phase).
This computation requires traversing upwards from x until the nearest node with a suffix link is found. All nodes encountered on the way must be imaginary (real and back-propagated nodes have suffix links) and we need to account for the time taken to traverse these nodes.

[2] We have defined valid directions only for nodes in the compacted trie. However, this definition can be extended for points in the middle of an edge in the obvious way, i.e., by imagining as if a node is present at that point.

The key claim is the following. Note that an imaginary node y signals the beginning of a new scanning phase in McCreight's algorithm, in which the tree is scanned downwards starting at y, until a new leaf is inserted as a child of a new or existing internal node z.

LEMMA 4. *The total number of times imaginary node y can be encountered while determining $nanc(*)$ over the entire algorithm is at most $|str(z)| - |str(y)|$.*

Proof. Note that z is a real node after the above scanning phase starting at y finishes. y could be encountered once while setting up $link(z)$. Subsequently, since $link(z)$ is in place, y will be encountered only when finding $nanc(z')$, where z' is real and on the path from y to z. There can be at most $|str(z)| - |str(y)|$ such distinct real nodes z'. □.

COROLLARY 1. *The total time taken in traversing imaginary nodes while determining $nanc(*)$ is $O(n)$.*

Proof. Follows from Fact 1 and Lemma 4. □

Theorem 1 now follows for quasi-suffix collections, assuming that the correct child of a particular node can be found in $O(1)$ time. The extension to quasi-suffix collections is sketched next.

6. ALGORITHM FOR MULTIPLE QUASI-SUFFIX COLLECTIONS

We sketch how to extend the above algorithm to a multiple quasi-suffix collection comprising m quasi-suffix collections of $O(n)$ strings each. The time taken will be $O(mn)$.
Suffix links and back-propagation directions need to be redefined appropriately as follows. Let s_i^k denote the ith string in the kth quasi-suffix collection under consideration (assume an arbitrary ordering on the m quasi-suffix collections).

Suffix Links. For a node x, $link(x)$ is now defined to be that node y such that if $str(x)$ is a prefix of some s_i^k then $str(y)$ is a prefix of s_{i+1}^k; further $|str(y)| = |str(x)| - 1$. Note that since condition 3 in the definition of quasi-suffix collections need not be satisfied with equality, $link(x)$ need not be defined for every node x. Also note that if $link(x)$ exists then it is unique; this follows because if $str(x)$ is a prefix of s_i^k and of $s_j^{k'}$ then s_{i+1}^k and $s_{j+1}^{k'}$ agree in the first $|str(x)| - 1$ characters.

Back-Propagation Directions. For a node x, let $prev(x)$ be a set of strings defined as follows. For each s_i^k having prefix $str(x)$, $prev(x)$ contains the prefix of s_{i-1}^k of length $|str(x)| + 1$. Note that $prev(x)$ is a set and not a multiset; therefore all strings in it are distinct. *Direction a* is said to be *valid* for node x if string a appears in $prev(x)$.

The Algorithm. The algorithm inserts each collection in turn into the current compacted trie. The first string of each quasi-suffix collection starts a new rescanning phase beginning at the root of the compacted trie. The subsequent strings in the collection are inserted as in the previous algorithm. Note that the size of the compacted trie will now be $\Theta(mn)$. Fact 1 continues to hold with $O(n)$ replaced by $O(mn)$. The analysis is as before with the following two changes. All $O(n)$ terms are replaced by $O(mn)$. Further, in Lemma 3, if an extended path ends because all further valid directions backwards are exhausted then the substring associated with the termination point is a prefix of the first string in one of the m quasi-suffix collections being considered.

7. THE HASHING SCHEME

Recall from Section 2.2 that we need to perfectly hash $O(n)$ pairs, where the first entry in each pair is a node number and the second entry is a character from the alphabet. Each such pair can be treated as a number from a range polynomial in n. We give a dynamic hashing scheme which will perfectly hash an item from a polynomial in n range in $O(n)$ time, with high probability. The time taken to access a particular item will be $O(1)$ and the total space is $O(n)$.
Fredman-Komlos-Szemeredi [4] showed how n items from the range $[0 \dots poly(n)]$ can be hashed into the range $[0 \dots s]$ without any collisions, where $s = \theta(n)$. Their algorithm takes $O(n)$ time and space and works by choosing randomly from a family of almost-universal hash functions (assuming constant time arithmetic on $O(\log n)$ bits). It ensures no collisions with probability at least $1/2$.
This was generalized by Dietzfelbinger et al. [2] to the dynamic setting. The expected amortized insertion/deletion time for their algorithm is $O(1)$; searching takes $O(1)$ worst case time. Our aim is to strengthen the above algorithm so it works with high probability, not just with constant probability. To this end, we modify the FKS perfect hashing scheme to make it work with high probability, first in the static setting, and then in the dynamic setting.
First, we present the static algorithm. The key idea is to create several perfect hashing subproblems and to apply the FKS scheme on each independently to obtain high success probability.

7.1 The Static Hashing Scheme

The following steps are performed. Let ϵ be any positive constant. The time and space taken by our data structure will be linear, but with an $\frac{1}{\epsilon}$ constant factor. The failure probability will decrease as ϵ gets closer to 0.

Step 1. Start with an imaginary array A of size n^c, where the n items to be hashed come from the range $1 \dots n^c$. Each item indexes into an unique element in this array. Next, repeatedly partition this array as in Step 2.

Step 2. Construct a *partition tree* as described below. Each node in this tree will have a subarray of A associated with it. The depth of this tree will be a constant and the number of nodes will be $O(n)$. The root of this tree is A itself. It has n^ϵ children, each associated with a distinct subarray of A of size $n^{c-\epsilon}$ obtained by partitioning A into n^ϵ disjoint pieces. Each subarray with more than n^ϵ items is recursively partitioned; the remaining subarrays become leaves. Each leaf has at most n^ϵ items. Clearly, the number of levels in this tree is $O(\frac{c}{\epsilon}) = O(1)$ and the total size in $O(n)$. The total time taken to set up the tree is easily seen to be $O(n)$.

Step 3. Next, we consider each leaf of the above tree in turn, and the items in the subarray associated with this leaf. We perfect-hash these items using the FKS perfect hashing scheme. Since this scheme succeeds only with probability $1/2$, several trials may be required before these items are perfectly hashed. We show that with high probability, the total time taken in this process over all leaves is $O(n)$.

7.2 Time Complexity

We need to bound the time taken to perform several FKS perfect-hashings, where the total sizes of all subproblems is n, each subproblem has size at most n^ϵ, and a subproblem is performed successfully in linear time with probability 1/2.

Size Categories. Divide the leaves into $O(\log n)$ quadrupling categories by size (i.e., number of associated items). Consider just leaves in any one size category, namely, the category in which leaf sizes are in the range $\frac{n^\epsilon}{4^{i+1}} \cdots \frac{n^\epsilon}{4^i}$, $i \geq 0$. We will show that the time taken for this category is proportional to the sum of the sizes of leaves in this category plus $O(\frac{n}{2^i})$, with failure probability $\frac{O(\log n)}{2^{\Theta(\frac{2^i n^{1-\epsilon}}{\log n})}}$. It follows that the total time taken over all categories is $O(n)$, with failure probability $\frac{O(\log n)}{2^{\Theta(\frac{n^{1-\epsilon}}{\log n})}}$.

A leaf is said to *succeed* when the items in it are perfectly hashed. A *round* refers to one trial for each of the relevant leaves. The trials for the various leaves can be imagined to have proceeded in rounds, with leaves succeeding in one round dropping out of the subsequent rounds. We organize the rounds into groups.

Grouping Rounds. The 0th group comprises rounds performed before the number of unsuccessful leaves in this size category drops below $\frac{n^{1-\epsilon}2^i}{\log n}$. For $j \geq 1$, the jth group comprises rounds performed after the number of unsuccessful leaves in this size category drops below $\frac{n^{1-\epsilon}2^i}{2^{j-1}\log n}$ but before this number drops below $\frac{n^{1-\epsilon}2^i}{2^j \log n}$.

We show that group 0 has $O(i + \log\log n)$ rounds and that each group $j \geq 1$ has $O(2^j)$ rounds, with failure probability $\frac{O(\log n)}{2^{\Theta(\frac{n^{1-\epsilon}2^i}{\log n})}}$ (over all groups). Further, we show that with the same failure probability, every two consecutive rounds in group 0 reduce the number of unsuccessful leaves by half. The total time taken for rounds in group 0 is then proportional to the sum of leaf sizes in this category. The time taken for rounds in the other groups is

$$O(\sum_{j=1}^{\Theta(\log n)} [2^j \frac{n^{1-\epsilon}2^i}{2^{j-1}\log n}\frac{n^\epsilon}{4^i}]) = O(\frac{n}{2^i}),$$

as required.

The Key Property. To show the above claims on the number of rounds in each group, we will need the following property, obtained using the Chernoff bound. If there are $\#u$ unsuccessful leaves at some instant of time, then half these leaves succeed in the next $2k$ rounds, with failure probability $\frac{1}{2^{\Theta(\#uk)}}$.

Group 0. First, consider group 0. If the number of unsuccessful leaves at some instant is at least $\frac{n^{1-\epsilon}2^i}{\log n}$, then two rounds will halve the number of unsuccessful leaves, with failure probability at most $\frac{1}{2^{\Theta(\frac{n^{1-\epsilon}2^i}{\log n})}}$ (apply the above property with $k = 1$ and $\#u \geq \frac{n^{1-\epsilon}2^i}{\log n}$). Note that the number of leaves in the size category being considered is at most $\frac{n}{n^\epsilon/4^{i+1}} = n^{1-\epsilon}4^{i+1}$, to begin with. It follows that group 0 has $2(i + 2 + \log\log n)$ rounds, and halving occurs in each pair of consecutive rounds, with failure probability at most $\frac{(i+2+\log\log n)}{2^{\Theta(\frac{n^{1-\epsilon}2^i}{\log n})}} = \frac{O(\log n)}{2^{\Theta(\frac{n^{1-\epsilon}2^i}{\log n})}}$.

Other Groups. Next, consider group j, $j \geq 1$. Applying the above property with $k = 2^j$ and $\#u \geq \frac{n^{1-\epsilon}2^i}{2^j \log n}$, we get that group j has 2.2^j rounds with failure probability $\frac{1}{2^{\Theta(\frac{n^{1-\epsilon}2^i 2^j}{2^j \log n})}} = \frac{1}{2^{\Theta(\frac{n^{1-\epsilon}2^i}{\log n})}}$. Finally, adding up the failure probability over all $O(\log n)$ groups gives $\frac{O(\log n)}{2^{\Theta(\frac{n^{1-\epsilon}2^i}{\log n})}}$, as required.

The total time and space taken above is thus $O(n)$, with high probability. Searching for an element requires following the unique path down the partition tree to reach the relevant perfect-hash table. These operations are easily seen to take $O(1)$ worst-case time.

7.3 The Dynamic Hashing Scheme

The dynamic version of the above static scheme maintains the partition tree described in Step 2 above at each instant (with the same parameters, i.e., A has size n^c and the branching factor in n^ϵ; here n is the total number of items which will ever be inserted).
Initially the partition tree will have just an empty root node. This tree will build up as insertions are made. The size of the partition tree at any instant will be proportional to the number of items in it. Further, at each instant, the perfect-hash structure at any leaf will have an associated *capacity.* This capacity will be at least the number of items at that leaf but no more than twice this quantity. It follows that the total space required at any instant will be proportional to the number of items present.
The algorithm for an insertion is described next. Note that our compacted tree application involves only insertions and no deletions.

Insertions. On an insertion x, the path down this partition tree to the appropriate leaf v is traced in $O(1)$ time. Subsequently, there are two cases depending upon how many items are already present in this leaf v.
First, suppose v has more than n^ϵ items, including x. Then the subarray associated with v is subdivided as in Step 2 of the static algorithm and the subtree rooted at v is developed. Each leaf in this tree will have at most n^ϵ elements in it. The elements in each of these leaves are then perfect-hashed.
Next, suppose v has at most n^ϵ items, including x, Then the items already in v would have been perfect-hashed; further, this perfect-hash structure will have a certain capacity. If this capacity is equaled by the insertion of x, then all the items in v (including x) are rehashed into a perfect-hash structure of twice the capacity. Otherwise, if this capacity is not equaled, then v is perfect-hashed. If there is no collision, then v's insertion is complete. Otherwise, if there is a collision, then all the items in v along with x are perfect-hashed again.

Time Analysis. We will show that the total time taken to perform n insertions is $O(n)$ with failure probability at most $\frac{O(\log n)}{2^{\Theta(n^{1-\epsilon}/\log n)}}$. To show the above, the following facts need to be noted.

1. The height of the partition tree is $O(1)$, therefore, the time spent in developing leaves into subtrees on insertion is just $O(n)$ over all n insertions.

2. The perfect-hash structure at any leaf in the partition tree begins with capacity which is twice the number

of items currently in the structure. Future insertions increase this number until it equals the capacity. Until this happens, this perfect-hash structure stays in place, though it may have to be rebuilt as many times as collisions are caused by insertions. Once the number of items matches the capacity, this perfect hash-structure is abandoned and a new perfect-hash structure with twice the capacity is put in place.

3. The total capacities of all perfect-hash structures which were ever in existence at any time during the n insertions is $O(n)$ (note that when a perfect-hash structure at a leaf is replaced by a new structure with twice the capacity, each structure is counted separately in the above sum). This follows from the doubling of capacities at a leaf and from the constant depth of the partition tree.

4. When the capacity of a perfect-hash structure at a leaf is doubled, the probability that this structure needs rebuilding before the number of items in it equals the new capacity is at most 1/2. Further, the time taken for rebuilding a particular perfect-hash structure is proportional to its capacity.

   Note the difference from the static case, where a perfect-hash trial succeeds on the items currently present with probability 1/2. Now, this is replaced by the fact that a perfect-hash trial succeeds with probability 1/2 even on future insertions as long as the capacity is not equaled.

Thus, to establish the total time bound above, it suffices to bound the total time taken for rebuilding the perfect-hash structures at the various leaves. This in turn boils down to the following question: what is the total time taken to perform several FKS perfect-hashings, where the total sizes of all subproblems is $\Theta(n)$, each subproblem has size at most n^{ϵ}, and a subproblem is performed successfully in linear time with probability 1/2? The analysis is now identical to the static case.

We conclude with two remarks on generalizing the above scheme when the number of items is unknown and deletions need to be performed as well. Neither of these is relevant to our application of constructing suffix trees.

Unknown Number of Items. Suppose the number of items to be hashed is an unknown quantity m, with each item coming from the range $1 \dots n^c$. Then we start with an initial estimate of 1, and double the estimate each time it is equaled by insertions. Suppose the current estimate is $2e$, and the number of items inserted is e. We first hash these items into an imaginary array A of size $(2e)^c$. No collisions occur, with inverse polynomial (in e) failure probability (using families of almost-universal hash functions). Subsequently, we build the partition tree with degree $(2e)^{\epsilon}$. When the number of insertions equals $2e$, we double our estimate to $4e$ and rebuild the entire structure. If the total number of insertions is m, then the total time and space required is $O(m)$, with failure probability inverse polynomial in m. This failure probability can be reduced to $\frac{1}{m^{\Theta(\log m)}}$ by using a family of hash functions defined by Siegel [10], instead of a family of almost-universal hash functions.

Deletions. Deletions can be easily handled as follows. A deleted item is just marked as deleted, without causing any other change to the data structure. Whenever the number of items marked as deleted becomes a constant fraction of the number of items currently in the data structure the entire structure is rebuilt on the undeleted items. The running time remains $O(m)$ for m insertions and deletions, with the same failure probability as above. The space at any instant is proportional to the number of undeleted items.

8. REFERENCES

[1] B. Baker. *A theory of parameterized pattern matching: Algorithms and applications.* Proceedings of the 25th ACM Symposium on Theory of Computing, 1993, 71–80.

[2] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F.M. auf der Heide, H. Rohnert, and R.E. Tarjan. *Dynamic perfect hashing: Upper and lower bounds.* SIAM Journal on Computing, 23, 1994, 738–761.

[3] M. Farach. *Optimal suffix tree construction with large alphabets*, Proceedings of the 38th IEEE Annual Symposium on Foundations of Computer Science, 1997, 137–143.

[4] M.L. Fredman, J. Komlos and E. Szemeredi. *Storing a sparse table with O(1) worst case access time*, Journal of the ACM, 31, 1984, 538–544.

[5] R. Giancarlo. *A generalization of suffix trees to square matrices with applications*, SIAM Journal of Computing, 1995, 520–562.

[6] D. Gusfield. *Algorithms on strings, trees and sequences.* Cambridge University Press, 1997.

[7] R. Karp and M. Rabin. *Efficient Randomized Pattern Matching*, IBM Journal of Research and Development, 1987.

[8] D.K. Kim and K. Park. *Linear time construction of 2-D suffix trees.* Proceedings of the 26th International Colloquium on Automata, Languages, and Programming, 1999.

[9] E.M. McCreight. *A space economical suffix tree construction algorithm*, Journal of the Association of Computing Machinery, 2, 1976, 262–272.

[10] A. Siegel. *On universal classes of fast high performance hash functions, their time space trade-off, and their applications*, Proceedings of the 30th IEEE Annual Symposium on Foundations of Computer Science, 1989, 20–25.

[11] D. Sleator, R. Tarjan. *A Data Structure for Dynamic Trees*, Journal of Computer and System Sciences, 26, 1983, 362–391.

[12] E. Ukkonen. *Constructing suffix trees on-line in linear time*, Proceedings of the IFIP 12th World Computer Congress, 1992, 484–492.

[13] P. Weiner. *Linear pattern matching algorithms*, Proceedings of the 4th IEEE Annual Symposium on Switching and Automata Theory, 1973, 1–11.