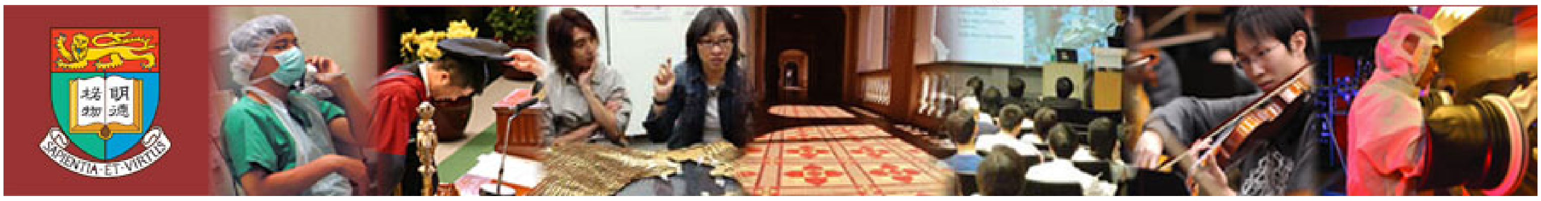

Dynamic Dictionary Matching and Compressed Suffix Trees

Ho-Leung Chan* Wing-Kai Hon* Tak-Wah Lam* Kunihiko Sadakane†

Abstract

Recent breakthrough in compressed indexing data structures has reduced the space for indexing a text (or a collection of texts) of length n from $O(n \log n)$ bits to $O(n)$ bits, while allowing very efficient pattern matching [10, 13]. Yet the compressed nature of such indices also makes them difficult to update dynamically. This paper presents the first $O(n)$-bit representation of a suffix tree for a dynamic collection of texts whose total length is n, which supports insertion and deletion of a text T in $O(|T| \log^2 n)$ time, as well as all suffix tree traversal operations, including forward and backward suffix links. This work can be regarded as a generalization of the compressed representation of static texts. Our new suffix tree representation serves as a core part in a compact solution for the dynamic dictionary matching problem, i.e., providing an $O(d)$-bit data structure for a dynamic collection of patterns of total length d that can support the dictionary matching query efficiently. When compared with the $O(d \log d)$-bit suffix tree based solution of Amir *et al.*, the compact solution increases the query time by roughly a factor of $\log d$ only. In the study of the above results, we also derive the first $O(n)$-bit representation for maintaining n pairs of balanced parentheses in $O(\log n / \log \log n)$ time per operation, matching the time complexity of the previous $O(n \log n)$-bit solution.

1 Introduction

This paper studies the compact solution of the following dynamic data structure problems: generalized suffix trees, dynamic dictionary matching, and parentheses maintenance.

Suffix Trees and Dynamic Dictionary Matching. Given a text T of length n, a suffix tree [18, 24] for T is a compact trie containing all suffixes of T, with each leaf storing the position of the corresponding suffix and each internal node storing a special pointer called a *suffix link*. We assume that characters are chosen from a constant size alphabet. A suffix tree occupies $O(n \log n)$ bits of space and supports finding all occurrences of a given pattern P in T in $O(|P| + occ)$ time, where *occ* denotes the number of occurrences. The notion of suffix tree can be generalized for a collection of texts, storing all suffixes of the texts in the collection. Such a suffix tree allows pattern searching to be performed over all texts in $O(|P|+occ)$ time. Furthermore, McCreight [18] showed that this generalized suffix tree can be updated in $O(t)$ time when a text of length t is inserted into or deleted from the collection.

Suffix trees find application in other complicated string matching problems (e.g., [12, 15]), for which efficient solutions rely on not only the efficient pattern matching of suffix trees, but also the tree structure and the provision of suffix links. Among such problems, the dynamic dictionary matching problem is one of the most well studied [1, 2, 3, 4, 5, 23], which is required to index a collection of patterns $\{P_1, P_2, \ldots, P_k\}$ with total length d, so as to answer efficiently the occurrences of all P_i in any given text T, and allow efficient insertion and deletion of patterns. Most of the previous solutions for dynamic dictionary matching are based on suffix trees. In particular, Amir *et al.* [4] showed that updating a pattern P can be done in $O(|P| \log d / \log \log d)$ time and a dictionary matching query for a text T takes $O((|T| + occ) \log d / \log \log d)$ time.[1]

Compressed Indexing Data Structures. The need of indexing very long genome sequences (e.g., a human genome contains 2.8G base pairs) has triggered the research on compressed indexing data structures that

*Department of Computer Science, The University of Hong Kong, Hong Kong, {hlchan,wkhon,twlam}@cs.hku.hk

†Department of Computer Science and Communication Engineering, Kyushu University, Japan, sada@csce.kyushu-u.ac.jp

[1]Sahinalp and Vishkin [23] devised a new data structure called fat-tree, and improved the update time to $O(|P|)$, and query time to $O(|T| + occ)$.

use $O(n)$ bits instead of $O(n \log n)$ bits. The past few years have witnessed two breakthrough results. The first one is the Compressed Suffix Arrays (CSA) by Grossi and Vitter [13], and the second one is the FM-index by Ferragina and Manzini [10]. These indexes are compressed versions of suffix arrays [17], occupying only $O(n)$ bits, yet supporting efficient pattern searching. Chan *et al.* [7] further showed that CSA and FM-index can be combined together to index a dynamic collection of texts $\{X_1, X_2, \cdots, X_\ell\}$, allowing searching for any given pattern P in all X_i's in $O(|P| \log n + occ \log^2 n)$ time, and more importantly, they showed that these $O(n)$-bit data structures can be updated in $O(|X| \log n)$ time when a text X is inserted or deleted. However, CSA or FM-index does not represent a suffix tree in the sense that the corresponding tree structure and suffix links are not captured, and thus they are not sufficient for solving the dynamic dictionary matching problem.

It is natural to ask whether we can have a compressed version of a suffix tree for a dynamic collection of texts. That is, we want to support queries about the suffix tree structure (namely, parent, child, sibling, edge label, and leaf label) and suffix links, while allowing efficient update due to insertion and deletion of texts. Sadakane [22] has made a step towards this goal; his work gives an $O(n)$-bit representation for a suffix tree which can avoid storing pointers, but his work assumes a static text (or a static collection of texts) so that the underlying data structures are rigidly packed together and thus cannot be updated efficiently. The challenge for 'dynamizing' a compressed suffix tree lies in two aspects: structural and algorithmic. Structurally, the compressed suffix tree should not only be compact, but also be flexible enough to allow efficient updates. Algorithmically, we have to find efficient updating methods that are tailored for the underlying data structures. This often requires supporting operations other than the basic navigational operations for traversing the suffix tree.

Compressed Suffix Trees. In this paper, we give the first $O(n)$-bit representation of a suffix tree that allows efficient update. Our solution is comprised of several dynamic data structures for representing CSA and FM-index, as well as the tree structure. The latter inspires us to study a compact representation for maintaining a sequence of balanced parentheses (see the discussion below). Retrieving an edge label and leaf label requires $O(\log^2 n)$ time, while other navigation queries, including suffix links, can be performed in $O(\log n)$ time. More importantly, we allow the retrieval of backward suffix links [24], which turns out to be crucial for supporting efficient update of this representation. Apparently, representing backward suffix links is more demanding than that for the (forward) suffix links, because each internal node of a suffix tree may have more than one backward suffix link, while some internal nodes may have none. Nevertheless, we are able to show that FM-index already allows us to recover the backward suffix links efficiently.

As mentioned before, given a suffix tree representing a collection of texts, one can use McCreight's method to insert or delete a text X efficiently. Note that McCreight's insertion method updates the suffix tree by adding suffixes of X one by one from the longest to the shortest one. This creates a fundamental technical problem as both CSA and FM-index should be constructed and updated in an ascending order of the suffixes; as these indices are only well-defined for representing a collection of texts and all their suffixes. This motivates us to take an asymmetric approach with the provision of the two types of suffix links. Precisely, insertion is based on the framework of Weiner's suffix tree construction method, where we start from adding the shortest suffix to the longest one, exploiting backward suffix links. For deletion, it is based on McCreight's method with forward suffix links. Both can be done in $O(|X| \log^2 n)$ time. Another interesting idea is that edge labels are only implicitly stored by the compact data structures, which can be computed efficiently when needed. Furthermore, when the data structures are updated, the correctness of the edge labels are automatically maintained.

Based on our compact representation of a suffix tree, we can adapt the work of Amir *et al.* [4] to give the first $O(d)$-bit solution for the dynamic dictionary matching problem. Our solution supports updating of a pattern P in $O(|P| \log^2 d)$ time, and a dictionary query for a text T in $O((|T| + occ) \log^2 d)$ time.

Parentheses Maintenance. To represent a generalized suffix tree, we need a compact representation of the tree structure. This can be done using a sequence of balanced parentheses [16, 19]. For a sequence of n pairs of balanced parentheses, the basic queries include *find-match* and *enclose*, which find the position of the matching parenthesis and the nearest pair of enclosing parentheses, respectively. For the static case, the best known solution is by Munro and Raman [19], which supports these operations in $O(1)$ time and occupies only $2n + o(n)$ bits. When we need to maintain the parentheses under insertion and deletion, the best result is by Amir *et al.* [4], which requires $O(n \log n)$ bits, while supporting each operation, including an update, in $O(\log n / \log \log n)$ time. In this paper, we propose the first $O(n)$-bit representation for maintaining the balanced parentheses, with $O(\log n / \log \log n)$ time per operation, thus matching the best result with space complexity of $O(n \log n)$ bits.

As for theoretical interest, we observe that the classical problem for maintaining a sequence of bits under updates, with *rank* and *select* queries supported, can be reduced to the parentheses maintenance problem. Then based on the lower bound result from Fredman and Saks [11], we can conclude that for any data structure for the parentheses maintenance, there exists a sequence of operations requiring $\Omega(\log n / \log \log n)$ amortized time per operation.

Finally, we also consider a more complicated operation called *double-enclose*, which finds the nearest parenthesis pair that encloses two input parenthesis pairs. We show that with an $O(n)$-bit data structure, this operation can be achieved in $O(\log n)$ time.

Organization. The remaining of the paper is organized as follows. Section 2 gives a brief review on the suffix trees, suffix arrays, CSA and FM-index. Sections 3, 4 and 5 are devoted to our solutions for the dynamic compressed suffix tree, parentheses maintenance and dynamic dictionary matching, respectively.

2 Preliminaries

In this section, we give a brief review on suffix trees [18, 24], suffix arrays [17], Compressed Suffix Arrays [13], and FM-index [10]. Let $T[1..n] = T[1]T[2]\cdots T[n]$ be a string of length n over a finite alphabet Σ. For any $i = 1, \ldots, n$, $T[i..n]$ is a suffix of T.

Suffix Tree. The suffix tree is a compact trie that contains all suffixes of T. Each edge is labeled by a pair of integers specifying a substring of T, and each leaf is labeled by the starting position of the corresponding suffix of T. We also store a suffix link for each internal node, which is defined as follows. We define the *path label* of a node u as the string formed by concatenating the edge labels on the path from the root to u. Then, the suffix link of u is a pointer from u to another node v such that the path label of v is the same as the path label of u with the first character removed. Note that suffix link for every internal node exists. A suffix tree can be stored in $O(n \log n)$ bits.

A generalized suffix tree is a suffix tree containing the suffixes of all texts in a collection. Each edge is labeled by three integers, specifying which substring of which text. A generalized suffix tree can be updated efficiently to allow insertion or deletion of a text in the collection. Precisely, insertion or deletion of a text of length t can be done in $O(t)$ time. Searching where a pattern P appears in the collection is also efficient, using $O(|P| + occ)$ time, where occ denotes the total number of occurrences.

Suffix Arrays, CSA and FM-index. By enumerating the leaves of a suffix tree from left to right, we obtain the suffix array $\mathtt{SA}[1..n]$ of T, which is an array of integers such that $T[\mathtt{SA}[i]..n]$ is the lexicographically i-th smallest suffix of T [17]. The main component of CSA is the function $\Psi[1..n]$ where $\Psi[i] = \mathtt{SA}^{-1}[\mathtt{SA}[i]+1]$. In other words, let i be the lexicographical order of the suffix $T[\mathtt{SA}[i]..n]$. Then, $\Psi[i]$ gives the lexicographical order of the suffix $T[\mathtt{SA}[i] + 1..n]$. The Ψ array admits an $O(n)$-bit representation. We can count the number of occurrences of a pattern P in T using $O(|P| \log n)$ queries to Ψ [13].

The main component of FM-index is the function *count*, which is defined based on the $\mathtt{BWT}$ array [6]. For $i = 1, \ldots, n$, $\mathtt{BWT}[i]$ is the character $T[\mathtt{SA}[i] - 1]$. For each character $c \in \Sigma$ and $i = 1, \ldots, n$, the function $count(c, i)$ is the number of times character c appears in $\mathtt{BWT}[1..i]$. Similar to the Ψ of CSA, $count(c, i)$ admits an $O(n)$-bit representation. We can count the number of occurrences of a pattern P in T using $O(|P|)$ queries to *count* [10]. See the figure below for an example of the Ψ, $\mathtt{BWT}$ and *count* functions.

$T =$ abbaaaba\$

i	suffixes in sorted order	SA[i]	$\Psi[i]$	BWT[i]	*count*("a", i)	*count*("b", i)
1	\$	9	6	a	1	0
2	a \$	8	1	b	1	1
3	a a a b a \$	4	4	b	1	2
4	a a b a \$	5	5	a	2	2
5	a b a \$	6	7	a	3	2
6	a b b a a a b a \$	1	9	\$	3	2
7	b a \$	7	2	a	4	2
8	b a a a b a \$	3	3	b	4	3
9	b b a a a b a \$	2	8	a	5	3

In fact, CSA and FM-index can be generalized to index a collection of texts $\{T_1, T_2, \ldots, T_k\}$ instead of a single text. The definition is slightly changed as the suffix array now corresponds to all suffixes of all texts in the collection. We say $\mathtt{SA}[i] = (j, \ell)$ if the suffix $T_j[\ell..|T_j|]$ is the lexicographically i-th suffix, and $\mathtt{SA}[i]+1$ now refers to the tuple $(j, \ell+1)$, which represents the suffix $\mathtt{SA}[i]$ with the first character removed. Under this minor modification, CSA and FM-index are well-defined. In particular, Chan *et al.* [7] showed that CSA and FM-index can be combined to index a dynamic collection of texts. The updating process can be summarized by the following lemma.

LEMMA 2.1. ([7]) *Let $\mathcal{C} = \{T_1, T_2, \ldots, T_k\}$ be a set of k distinct strings. Let n be the total length of all strings in $\mathcal{C}$. We can maintain CSA and FM-index for $\mathcal{C}$ in $O(n)$-bit space such that inserting or deleting a text $T[1..t]$ takes $O(t \log n)$ time. Precisely, the updating is done by t steps, each taking $O(\log n)$ time. For insertion, the i-th step produces the index for $\mathcal{C} \cup \{T[t-i+1..t]\}$; for deletion, the i-th step produces the index for $(\mathcal{C} - \{T\}) \cup \{T[i+1..t]\}$.*

In addition, the above index supports retrieving any Ψ entry in $O(\log n)$ time. For an SA entry, it can be computed in $O(\log^2 n)$ time using FM-index, and we denote this time as t_{SA}. Also, we can perform pattern searching based on the *backward search* algorithm [10], which is described as follows.

LEMMA 2.2. ([7]) *Given the FM-index for a dynamic collection of texts $\mathcal{C}$. Let i be the lexicographical order of some pattern P among all suffixes of texts in $\mathcal{C}$. Then, for any character c, the FM-index supports a function $FM(i, c)$ that computes the lexicographical order of cP among all suffixes of texts in $\mathcal{C}$. The time required is $O(\log n)$.*

3 Compressed Suffix Tree

In this section, we describe an $O(n)$-bit representation of a suffix tree for a dynamic collection of texts. We call such a representation a *compressed suffix tree.* Our main result is stated in the following theorem.

THEOREM 3.1. *Let $\mathcal{C} = \{T_1, T_2, \ldots, T_k\}$ be a collection of texts with total length n. We can maintain a compressed suffix tree for $\mathcal{C}$, which uses $O(n)$-bit space and supports the following queries about the suffix tree for $\mathcal{C}$: finding the root in $O(1)$ time, and finding the parent, left child, left sibling, right sibling, and suffix link of a node in $O(\log n)$ time. The edge label and leaf label can be computed in $O(\log^2 n)$ time. Inserting or deleting of a text T in $\mathcal{C}$ can be done in $O(|T| \log^2 n)$ time.*

Roughly speaking, information about a suffix tree is stored using the following $O(n)$-bit data structures.

1. The tree structure is represented by a list of balanced parentheses.
2. Information about suffix links and leaf labels can be deduced from CSA and FM-index.
3. Information about the edge labels is deduced from leaf labels together with an auxiliary data structure called LCP which maintains the length of the longest common prefix between any two adjacent leaves.

When a text is inserted into or deleted from $\mathcal{C}$, one naive way to update the compressed suffix tree is to decompress it back to the original suffix tree, perform update on the uncompressed suffix tree, and then compress it back to the above data structures. Yet, such approach is very time consuming and requires $O(n \log n)$-bit working space. We show that we can update the compressed suffix tree efficiently by working on the data structures directly in the compressed format. Intuitively, our compressed suffix tree supports the navigation operations of the normal suffix trees. Thus, we can simulate an updating algorithm for normal suffix tree, in order to determine how an update changes the original suffix tree.

Then, we show how to convert the changes into actual modifications on the data structures. Finally, we show how to implement the data structures to support the required updates efficiently.

3.1 Tree Structure and Navigation Operations.

The tree structure of a suffix tree is represented by a list of parentheses as follows: Traverse the suffix tree in a depth-first-search order; at the first time a node is visited, append a "(" to the list, and at the last time a node is visited, append a ")" to the list. Note that the list of parentheses is balanced and each node in the suffix tree is represented by a pair of matching parentheses. Therefore, we can specify a node u in the suffix tree using the position of the open parenthesis that represents u. To support efficient navigation operations on the suffix tree, we require efficient operations on the balanced parentheses, as shown in the next lemma, where the proof of which is deferred to Section 4.

LEMMA 3.1. *We can maintain a list $\mathcal{B}$ of n pairs of balanced parentheses in $O(n)$-bit space and support each of the following operations in $O(\log n)$ time.*

- *find-match(u): Find the matching parenthesis of u.*
- *enclose(u): Find the nearest pair of matching parentheses that encloses u.*
- *double-enclose(u, v): Find the nearest pair of matching parentheses that encloses both u and v.*
- *rank-leaf(u), select-leaf(i): A pair of consecutive matching parentheses is called a leaf in $\mathcal{B}$. The operation rank-leaf(u) counts the number of leaves from the beginning of $\mathcal{B}$ up to location of u. The operation select-leaf(i) finds the i-th leaf in $\mathcal{B}$.*
- *insert(ℓ, r), delete(ℓ, r): Insert or delete the matching parentheses pair located at (ℓ, r).*

For a node u, its parent is given by *enclose*(u), the left child is $u + 1$, the left sibling is *find-match*$(u - 1)$, and the right sibling is *find-match*$(u) + 1$.

Lowest common ancestor, leaf rank and selection, leftmost and rightmost leaf. The list of balanced parentheses supports other queries about the suffix tree. In particular, the lowest common ancestor of two nodes u and v is *double-enclose*(u, v). The *rank* of a leaf u, which is the lexicographical order of the suffix corresponding to it, is *rank-leaf*(u). The i-th leaf, which is the one corresponding to the lexicographically i-th suffix, is given by *select-leaf*(i). The leftmost leaf and the rightmost leaf of the subtree rooted at u can be found by *rank-leaf*$(u-1)+1$ and *rank-leaf*(*find-match*(u)), respectively. Each of the above operations takes $O(\log n)$ time.

Leaf labels and suffix links are deduced from the tree structure, CSA, and FM-index as follows.

Leaf labels. For any leaf v, let i be its rank. The suffix corresponding to v has lexicographical order i among all suffixes in the suffix tree. Thus, the leaf label of v is $\texttt{SA}[i]$, which can be found using the FM-index. Finding i and $\texttt{SA}[i]$ takes totally $O(\log n + t_{SA})$ time.

Suffix links. Consider an internal node u. Let u_ℓ and u_r be the leftmost leaf and rightmost leaf in the subtree rooted at u, respectively. Let x and y be the leaf rank of u_ℓ and u_r. $\Psi[x]$ gives the rank of a leaf whose leaf label is that of u_ℓ with the first character removed. Similarly, $\Psi[y]$ gives the rank of a leaf whose leaf label is that of u_r with the first character removed. Let v be the lowest common ancestor of *select-leaf*$(\Psi[x])$ and *select-leaf*$(\Psi[y])$. We notice that the path label of v is that of u with the first character removed. Thus, v is the node pointed by the suffix link of u. The above steps takes $O(\log n)$ time.

Finally, we describe an auxiliary data structure called LCP for computing the edge labels.

Edge labels. Recall that for any node u, the edge label of u is the string on the edge from u's parent to u. More precisely, the edge label is represented by a tuple (j, s, ℓ) such that $T_j[s..s + \ell - 1]$ is the string on the edge. To compute the edge labels, we dynamize Sadakane's LCP data structure [21], which uses $O(n)$ bits to store the length of the longest common prefix between any two adjacent leaves in the suffix tree. Then, the value *LCP*(i), which is the length of the longest common prefix between the i-th leaf and the $(i + 1)$-th leaf, can be retrieved in $O(\log n)$ time. In addition, when we insert a new suffix to become the i-th leaf of the suffix tree, if we can find the lengths of the longest common prefix of this suffix with the original $(i - 1)$-th and i-th smallest suffix, we can update the LCP in $O(\log n)$ time to reflect the insertion of this suffix. On the other hand, when we delete the i-th smallest suffix, if we can find the length of the longest common prefix between the original $(i-1)$-th and $(i+1)$-th smallest suffix, we can perform the update in $O(\log n)$ time.

Based on LCP, we can find the path label and then the edge label of a node u in $O(\log n + t_{SA})$ time as follows. If u is a leaf, then the path label of u is the leaf label. Otherwise, we find the rightmost leaf x rooted at u's leftmost child, and compute its rank i. We notice that the path label of u is the longest common prefix between x and the leaf with rank $i+1$, and its length is given by $LCP(i)$. Thus, with the leaf label of x and $LCP(i)$, we can deduce the path label of u. To find the edge label of u, we find the path label of u and the path label of u's parent. The edge label of u can be calculated accordingly. The process takes $O(\log n + t_{SA})$ time.

3.2 Inserting and Deleting a Text. Assume that we have the list of balanced parentheses, CSA, FM-index and LCP representing the suffix tree for a collection of texts $\mathcal{C}$. To insert a new text T into $\mathcal{C}$, we update the data structures to reflect the change that all suffixes of T are inserted into the suffix tree. We perform the update in $|T|$ rounds such that in the i-th round, the i-th shortest suffix $T[|T|-i+1..|T|]$ is inserted as a new leaf into the suffix tree. Each round involves updating the list of balanced parentheses, CSA, FM-index and LCP. Thus, we maintain an invariance that at the end of the i-th round, the data structures represent the compressed suffix tree for the collection $\mathcal{C} \cup \{T[|T|-i+1..|T|]\}$.

In each round, updating CSA and FM-index can be done according to Lemma 2.1. The key concern is updating the list of balanced parentheses and LCP, which is done by the following two steps: calculating the new suffix tree information, and updating the data structures according to the new suffix tree.

For the first step, we observe that our compressed suffix tree supports the navigation operations on normal suffix tree, so we can make use of Weiner's algorithm to calculate the location of the new leaf. However, Weiner's algorithm involves the following notion of backward suffix links.

DEFINITION 3.1. *Consider a suffix tree for a collection of texts. For any internal node u and any character c, the backward suffix link of u with respect to c is a pointer to the internal node v such that the path label of v is the character c concatenated with the path label of u. The backward suffix link is null if no such v exists.*

Note that if the backward suffix link of u with respect to a character c points to a node v, then the suffix link of v points to u. Unlike the original Weiner's algorithm, we cannot store the backward suffix links for each internal node explicitly, because it would take $O(n \log n)$ bits. Instead, we will show how to calculate it using our $O(n)$-bit data structures in $O(\log n)$ time.

Yet, for our suffix tree representation, we also need to know the longest common prefix between the newly added leaf and its two adjacent leaves in order to update the LCP. We show that these lengths can be calculated efficiently from the old LCP. After the information about the new suffix tree is obtained, we can proceed to the second step to update the data structures accordingly.

Assume that we are in the $(i+1)$-th round of an update. That is, the suffix $S = T[|T|-i+1..|T|]$ is just inserted into the suffix tree in the last round. Let $c = T[|T|-i]$ be a character and we want to insert the suffix cS into the suffix tree. The two steps go as follows.

3.2.1 Calculating the New Suffix Tree Information. To calculate information about the new suffix tree, we need the use of backward suffix links. We first show how to calculate the backward suffix link of a node efficiently.

LEMMA 3.2. *Consider a compressed suffix tree for a collection of texts $\mathcal{C} = \{T_1, T_2, \cdots, T_k\}$ with total length n. For any internal node u and character c, the backward suffix link of u with respect to c can be found in $O(\log n)$ time.*

Proof. For any internal node u, let S be the path label of u. We first assume that the backward suffix link of u with respect to c exists. That is, there is an internal node v with path label cS. Let a and b be the leftmost and rightmost leaf of u, respectively. Let x and y be the leftmost and rightmost leaf of v. For any internal node p and any leaf q in the subtree rooted at p, we let $E(p,q)$ be the concatenation of edge labels from p to q.

By the definition of a suffix tree, there is a leaf m in the subtree rooted at u such that $E(u,m)$ equals $E(v,x)$. As a is the leftmost leaf in the subtree rooted at u, $E(u,a)$ is either lexicographically smaller than or equal to $E(v,x)$. In both cases, *FM*(*rank-leaf*$(a),c)$ gives the leaf rank of x. Similarly, $E(u,b)$ is lexicographically equal to or greater than $E(v,y)$. If $E(u,b)$ is lexicographically equal to $E(v,y)$, *FM*(*rank-leaf*$(b),c)$ is the leaf rank of y; otherwise, *FM*(*rank-leaf*$(b),c)$ is one greater than the leaf rank of y. We will test both cases. We find the *FM*(*rank-leaf*$(a),c)$-th and the *FM*(*rank-leaf*$(b),c)$-th leaf, and find their lowest common ancestor v'. If the suffix link of v' points to u, then the backward suffix link of u with respect to c is v'. We repeat the test using the (*FM*(*rank-leaf*$(b),c)-1)$-th leaf. If both cases fail, we conclude that the backward suffix link of u with respect to c is null. The above steps take $O(\log n)$ time. □

Location of the leaf corresponding to cS. We follow Weiner's algorithm to determine where the leaf should be added. Let w be the leaf for the suffix S, whose location is known by the end of last round. We start at w, traverse up the tree and look for the first node u with a non-null backward suffix link with respect to c.

If such a node u is found, we follow the backward suffix link to a node v. Let c' be the first character on the path from u to w. If there is no edge out of v with first character being c', then the leaf for cS is attached as a child of v. Otherwise, we let (v,v') be an edge going out of v with first character being c'. The leaf for the suffix cS should be attached to a new internal node on this edge.

If no such node u is found when we traverse from w up to the root, the leaf for the suffix cS is attached to the root or to a new internal node on an edge out of the root.

The above steps calculate location of the new leaf in $O(e_i \log n + t_{SA})$ time, where $e_i \geq 1$ is the number of edges traversed when we go up from the leaf w searching for the node u. The term t_{SA} is needed because when we arrive at the node v or arrive at the root, we need to find the first character of each outgoing edge, which requires finding the edge labels.

The longest common prefix information. Recall that the suffix $S = T[|T| - i + 1..|T|]$ is inserted to the suffix tree in the last round, and now we want to insert the suffix cS into the suffix tree, where $c = T[|T| - i]$. We show how to calculate the longest common prefix between the leaf corresponding to cS and its two adjacent leaves efficiently.

Let x be the lexicographical order of S among all suffixes in the suffix tree, which is known by the end of last round. Let $j = FM(x,c)$. By Lemma 2.2, j is the lexicographical order of cS among all suffixes in the suffix tree, and the leaf representing cS will be inserted as the j-th leaf in the suffix tree. The length of the longest common prefix between cS and the suffix corresponding to the $(j-1)$-th leaf can be calculated as follows.

Lemma 3.3. *The length of the longest common prefix between cS and the suffix corresponding to the $(j-1)$-th leaf can be found in $O(\log n + t_{SA})$ time.*

Proof. Let $c'S'$ be the suffix corresponding to the $(j-1)$-th leaf, where c' is a character and S' is a string. If $c \neq c'$, the longest common prefix of cS and $c'S'$ has length 0. Otherwise, we notice that the $\Psi(j-1)$-th leaf is the leaf corresponding to the suffix S'. Thus, the length of the longest common prefix between cS and $c'S'$ is 1 + the longest common prefix between S and S', where S and S' are the suffixes corresponding to the x-th and $\Psi(j-1)$-th leaf, respectively. We find the lowest common ancestor of the x-th and the $\Psi(j-1)$-th leaf. The length of the path label for the lowest common ancestor gives the length of the longest common prefix. The above steps take $O(\log n + t_{SA})$ time, which is dominated by the time to find the path label. □

Calculating the length of the longest common prefix between cS and the suffix corresponding to the j-th leaf is identical.

3.2.2 Updating the Data Structures.

After the information about new suffix tree is known, we update the data structures to actually reflect the change that the suffix cS is inserted into the suffix tree. CSA and FM-index can be updated in $O(\log n)$ time by Lemma 2.1. It remains to update the list of balanced parentheses and LCP.

Recall that the list of balanced parentheses represents the tree structure of the suffix tree. The previous calculation finds where the leaf corresponding to the suffix cS is attached to the suffix tree, so the list of parentheses can be updated accordingly. There are two cases where the new leaf is inserted. If the leaf is attached as the x-th child of an existing node u, we insert a pair of consecutive matching parentheses, such that it is enclosed by the parentheses representing u, and its location represents the x-th child of u. Otherwise, the leaf is attached to a newly created internal node m on some existing edge. Let (u, v) be the edge where u is the parent of v. We insert a pair of parentheses representing m, which is inside u and immediately enclosing v. We also insert a pair of consecutive matching parentheses within m. The above steps takes $O(\log n)$ time.

Finally, we update LCP according to the calculated values of the longest common prefix. Recall that $LCP(j)$ is the length of longest common prefix between the j-th leaf and the $(j+1)$-th leaf. Assume that cS is inserted as j-leaf of the suffix tree, we need to change the value of $LCP(j-1)$ to the length of the longest common prefix between cS and the originally $(j-1)$-th leaf. Also, we need to insert a new value as $LCP(j)$, which is the length of the longest common prefix between cS and the originally j-th leaf. It takes $O(\log n)$ time to update the LCP.

3.2.3 Overall Time Complexity.

Consider the i-th round where we are inserting the i-th shortest suffix of T into the suffix tree. We calculate the new suffix tree information in $O(e_i \log n + t_{SA})$ time, where $e_i \geq 1$ is the number of edges traversed when we calculate the locations to insert the new leaf. Then we perform the changes on the data structures in $O(\log n)$ time. Note that it takes more time to calculate how the data structures are changed, than actually perform the change. The total time to insert a text T is $O(\sum_{i=1..|T|} e_i \log n + |T| \cdot t_{SA})$. Similar to the analysis of the Weiner's algorithm, we can show $\sum_{i=1..|T|} e_i \leq 3|T|$, so the time to insert T is $O(|T|(\log n + t_{SA})) = O(|T| \log^2 n)$. Note that once the list of balanced parentheses, CSA, FM-index and LCP are updated, the data structures represent the updated suffix tree. In particular, the edge labels are updated automatically.

When we delete a text T from $\mathcal{C}$, we delete all suffixes of T from the suffix tree starting from the longest one. We first locate the leaf for the suffix T and then reverse the steps of insertion. It takes $O(|T| \log^2 n)$ time to delete all suffixes of T.

4 Parentheses Maintenance

In this section, we consider compressed data structures for maintaining a list of n pairs of balanced parentheses. We first show an $O(n)$-bit data structure that supports finding the matching parenthesis, the nearest enclosing parentheses, and updating in $O(\log n / \log\log n)$ time. Then, we give another $O(n)$-bit data structure that supports finding the nearest enclosing parentheses for two given parentheses and updating in $O(\log n)$ time. Together, they prove Lemma 3.1.

Finding the matching and nearest enclosing parentheses. We divide the list of n pairs of parentheses into segments of length $\log^2 n / \log\log n$ to $2\log^2 n / \log\log n$. The segments are stored in leaves of a B-tree, and each internal node of the B-tree has $\log^{\frac{1}{4}} n$ to $2\log^{\frac{1}{4}} n$ children. For each internal node, as the number of children is small, we can build a searchable partial sum data structure[20] on information of the children, which allows a number of queries and updates in constant time. As a result, finding the matching and nearest enclosing parentheses takes time proportional to the height of the tree, which is $O(\log n / \log\log n)$. Details are as follows.

We will consider the enclose operation only. For an internal node u, let *close*[i] be the number of unmatched closing parentheses in the subtree rooted at the i-th child of u. We further divide these unmatched closing parentheses into two types: those with matching parentheses located in a subtree rooted at some other child of u (calling them near-unmatched closing parentheses); and those with matching parentheses located outside the tree rooted at u (calling them far-unmatched closing parentheses). We store these two numbers for the i-th child as *near-close*[i] and *far-close*[i] respectively. The values *open*[i], *near-open*[i] and *far-open*[i] are defined similarly.

We build a searchable partial sum data structure[20] on the *close* array, which maintains an array of at most $2\log^{\frac{1}{4}} n$ integers and supports the operations *sum*(j) =

$\sum_{i=1}^{j} close[i]$ and $search(x) = \min\{j | sum(j) \geq x\}$ in $O(1)$ time. We also build the searchable partial sum data structure on the each of the remaining five arrays.

Given a parenthesis i,[2] to find the open parenthesis enclosing i, we traverse down the tree to locate the leaf containing i. We scan the leaf to search for an open parenthesis enclosing i. If no such parenthesis is found, we traverse up the tree. We maintain an invariance that whenever we leave a node u, we know the number of unmatched closing parentheses inside the subtree rooted at u that are to the left of i. This information can be maintained in $O(1)$ time, based on the searchable partial sum data structures. Furthermore, at any internal node v, we can determine in constant time whether the enclosing parenthesis is in the tree rooted at v. If yes, we traverse down the tree looking for that parenthesis. The whole process takes time proportional to the height of the tree, which is $O(\log n / \log \log n)$.

Finding the double-enclose parentheses. We divide the list of parentheses into segments of length $\log n$ to $2 \log n$. The segments are stored as leaves of a red-black tree. For each internal node, we store information about its two children, so finding the double-enclose parentheses takes time proportional to the height of the red-black tree, which is $O(\log n)$. Details are as follows.

Let $excess(\ell, i)$ be the number of open parentheses minus the number of closing parentheses in the range $[\ell, i]$. For a range $[\ell, r]$, we say $min\text{-}excess(\ell, r) = i_0$, if for $\ell \leq i \leq r$, $excess(\ell, i)$ is minimized when $i = i_0$. The nearest enclosing parentheses for both ℓ and r is the nearest enclosing parentheses for $min\text{-}excess(\ell, r)$. Thus, finding $double\text{-}enclose(\ell, r)$ is reduced to finding $min\text{-}excess(\ell, r)$.

Furthermore, we observe that for any $b \in [\ell, r]$, $min\text{-}excess(\ell, r)$ is either $min\text{-}excess(\ell, b)$ or $min\text{-}excess(b + 1, r)$. Precisely, let i_0' and i_0'' denote the former and latter term. Then, $min\text{-}excess(\ell, r)$ is i_0' if $excess(\ell, i_0') < excess(\ell, b) + excess(b + 1, i_0'')$, and it is i_0'' otherwise.

Based on this observation, we store extra information in red-black tree to allow efficient calculation of the function *min-excess*. Precisely, for each internal node u, let x and y be the leftmost and rightmost parentheses in the subtree rooted at u; we store two values i and $excess(x, i)$, where i is $min\text{-}excess(x, y)$. Then, $min\text{-}excess(\ell, r)$ for any ℓ and r can be computed when we traverse from the leaf containing ℓ, and from the leaf containing r, to their lowest common ancestor in the red-black tree. This gives the following lemma and concludes the section.

LEMMA 4.1. *Given two parentheses ℓ and r, we can find $min\text{-}excess(\ell, r)$ in $O(\log n)$ time.*

5 Dynamic Dictionary Matching

Given a dynamic collection of patterns $\mathcal{D} = \{P_1, P_2, \ldots, P_k\}$ of total length d, we want to maintain an index on $\mathcal{D}$ such that when an arbitrary text T is given, we can efficiently answer the dictionary matching query which locates the occurrences of all patterns in T.

We follow the idea of Amir *et al.* [4], and maintain a compressed suffix tree for the collection of patterns. Dictionary matching query is basically done by a traversal on the suffix tree based on T. As required by [4], we also maintain a data structure which for any internal node u of the suffix tree, reports all patterns in $\mathcal{D}$ that are prefix to the path label of u. This is useful for reporting occurrences of patterns when we deduce that the path label of u is matching some part of T. To do so, we intuitively mark all the internal nodes of the suffix tree whose path label matches a pattern in $\mathcal{D}$. Then, to report patterns that are prefix to the path label of u, we report all the marked nodes on the path from u to the root. This marked tree structure can be represented compactly by a list of the balanced parentheses, and maintained based on Lemma 3.1. To report occurrences of all patterns in T, it takes $O(|T| \log^2 d)$ time to traverse the compressed suffix tree and takes $O(occ \log^2 d)$ time to report the *occ* occurrences. Since both the compressed suffix tree and the list of parentheses allow efficient updates, we obtain a compact solution for the dynamic dictionary matching problem as follows.

THEOREM 5.1. *Let $\mathcal{D} = \{P_1, P_2, \ldots, P_k\}$ be a dynamic collection of patterns with total length d. We can maintain an $O(d)$-bit index for $\mathcal{D}$, such that a dictionary matching query for a text T takes $O((|T| + occ) \log^2 d)$ time. Inserting or deleting a pattern P in $\mathcal{D}$ takes $O(|P| \log^2 d)$ time.*

[2]We refer to a parenthesis in the list by its index. Parenthesis $i < j$ if i is on the left of j.

References

[1] A. Aho and M. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] A. Amir and M. Farach. Adaptive Dictionary Matching. In *Proceedings of Symposium on Foundations of Computer Science*, pages 760–766, 1991.

[3] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic Dictionary Matching. *Journal of Computer and System Sciences*, 49(2):208–222, 1994.

[4] A. Amir, M. Farach, R. Idury, A. La Poutre, and A. Schaffer. Improved Dynamic Dictionary Matching. *Information and Computation*, 119(2):258–282, 1995.

[5] A. Amir, M. Farach, and Y. Matias. Efficient Randomized Dictionary Matching Algorithms (Extended Abstract). In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 262–275, 1992.

[6] M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, California, 1994.

[7] H. L. Chan, W. K. Hon, and T. W. Lam. Compressed Index for a Dynamic Collection of Texts. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 445–456, 2004.

[8] W. I. Chang and E. L. Lawler. Sublinear Approximate String Matching and Biological Applications. *Algorithmica*, 12(4/5):327–344, 1994.

[9] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[10] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Proceedings of Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[11] M. L. Fredman and M. E. Saks. The Cell Probe Complexity of Dynamic Data Structures. In *Proceedings of Symposium on Theory of Computing*, pages 345–354, 1989.

[12] R. Grossi and G. F. Italiano. Suffix Trees and their Applications in String Algorithms. In *Proceedings of South American Workshop on String Processing*, pages 57-76, 1993.

[13] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *Proceedings of Symposium on Theory of Computing*, pages 397–406, 2000.

[14] R. Grossi, A. Gupta and J. S. Vitter. When Indexing Equals Compression: Experiments with Compressing Suffix Arrays and Applications. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 636–645, 2004.

[15] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, New York, 1997.

[16] G. Jacobson. Space-efficient Static Trees and Graphs. In *Proceedings of Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[17] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[18] E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[19] J. I. Munro and V. Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[20] R. Raman, V. Raman, and S. S. Rao. Succinct Dynamic Data Structures. In *Proceedings of Workshop on Algorithms and Data Structures*, pages 426–437, 2001.

[21] K. Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 225–232, 2002.

[22] K. Sadakane. Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems*, accepted.

[23] S. C. Sahinalp and U. Vishkin. Efficient Approximate and Dynamic Matching of Patterns Using a Labeling Paradigm. In *Proceedings of Symposium on Foundations of Computer Science*, pages 320–328, 1996.

[24] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of Symposium on Switching and Automata Theory*, pages 1–11, 1973.