Suffix Tree of Alignment: An Efficient Index for Similar Data

Joong Chae Na[1], Heejin Park[2], Maxime Crochemore[3], Jan Holub[4], Costas S. Iliopoulos[3], Laurent Mouchard[5], and Kunsoo Park[6,⋆]

[1] Sejong University, Korea
[2] Hanyang University, Korea
[3] King's College London, UK
[4] Czech Technical University in Prague, Czech Republic
[5] University of Rouen, France
[6] Seoul National University, Korea
kpark@snu.ac.kr

Abstract. We consider an index data structure for similar strings. The generalized suffix tree can be a solution for this. The generalized suffix tree of two strings A and B is a compacted trie representing all suffixes in A and B. It has $|A|+|B|$ leaves and can be constructed in $O(|A|+|B|)$ time. However, if the two strings are similar, the generalized suffix tree is not efficient because it does not exploit the similarity which is usually represented as an alignment of A and B.

In this paper we propose a space/time-efficient *suffix tree of alignment* which wisely exploits the similarity in an alignment. Our suffix tree for an alignment of A and B has $|A|+l_d+l_1$ leaves where l_d is the sum of the lengths of *all* parts of B different from A and l_1 is the sum of the lengths of *some* common parts of A and B. We did not compromise the pattern search to reduce the space. Our suffix tree can be searched for a pattern P in $O(|P| + occ)$ time where occ is the number of occurrences of P in A and B. We also present an efficient algorithm to construct the suffix tree of alignment. When the suffix tree is constructed from scratch, the algorithm requires $O(|A| + l_d + l_1 + l_2)$ time where l_2 is the sum of the lengths of other common substrings of A and B. When the suffix tree of A is already given, it requires $O(l_d + l_1 + l_2)$ time.

Keywords: Indexes for similar data, suffix trees, alignments.

1 Introduction

The *suffix tree* of a string S is a compacted trie representing all suffixes of S [18,22]. Over the years, the suffix tree has not only been a fundamental data structure in the area of string algorithms but also it has been used for many applications in engineering and computational biology. The suffix tree can be

⋆ Corresponding author.

© Springer-Verlag Berlin Heidelberg 2013

constructed in $O(|S|)$ time for a constant alphabet [18,21] and an integer alphabet [8], where $|S|$ denotes the length of S. The suffix tree has $|S|$ leaves and requires $O(|S|)$ space.

We consider storing and indexing multiple data which are very similar. Nowadays, tons of new data are created every day. Some data are totally original and substantially different from existing data. Others are, however, created by modifying some existing data and thus they are similar to the existing data. For example, a new version of a source code is a modification of its previous version. Today's backup is almost the same as yesterday's backup. An individual Genome is more than 99% identical to the Human reference Genome (the 1000 genome project [1]). Thus, storing and indexing similar data in an efficient way is becoming more and more important.

Similar data are usually stored efficiently: When new data are created, they are aligned with the existing ones. Then, the resulting alignment shows the common and different parts of the new data. By only storing the different parts of the new data, the similar data can be stored efficiently.

When it comes to indexing, however, neither the suffix tree nor any variant of the suffix tree uses this similarity or alignment to index similar data efficiently. Consider the *generalized suffix tree* [2,10] for two similar strings $A =$ `aaatcaaa` and $B =$ `aaatgaaa`. Three common suffixes `aaa`, `aa`, `a` are stored twice in the generalized suffix tree. Moreover, two similar suffixes `aaatcaaa` and `aaatgaaa` are stored in distinct leaves even though they are very similar. Thus, the generalized suffix tree has $|A| + |B|$ leaves, most of which are redundant.

Recently, there have been some studies concerning efficient indexes for similar strings. Mäkinen et al. [16,17] first proposed an index for similar (repetitive) strings using run-length encoding, a suffix array, and BWT [5]. Huang et al. [11] proposed an index of size $O(n + N \log N)$ bits where n is the total length of common parts in one string, N is the total length of different parts in all strings. Their basic approach is building separately data structures for common parts and ones for different parts between strings. A self-index based on LZ77 compression [23] has been also developed due to Kreft and Navarro [13]. Another index based on Lemple-Ziv compression scheme is due to Do et al. [7]. They compressed sequences using a variant of the relative Lempel-Ziv (RLZ) compression scheme [14] and used a number of auxiliary data structures to support fast pattern search. Navarro [19] gave a short survey on some of these indexes.

Although these studies assume slightly different models on similar strings, most of them adopt classical compressed indexes to utilize the similarity among strings, that is, they focus on how to efficiently represent or encode common (repetitive) parts in strings. However, none of them support linear-time pattern search. Moreover, their pattern search time do not depend on only the pattern length but also the text length, and some indexes require (somewhat complicated) auxiliary data structures to improve pattern search time. In short, those data structures achieve smaller indexes by sacrificing pattern search time.

In this paper, we propose an efficient index for similar strings without sacrificing the pattern search time. It is a novel data structure for similar strings, named *suffix tree of alignment*. We assume that strings (texts) are aligned with each others, e.g., two strings A and B can be represented as $\alpha_1\beta_1\ldots\alpha_k\beta_k\alpha_{k+1}$ and $\alpha_1\delta_1\ldots\alpha_k\delta_k\alpha_{k+1}$, respectively, where α_i's are common chunks and β_i's and δ_i's are chunks different from the other string. (We note that the given alignment is not required to be optimal.) Then, our suffix tree for A and B has the following properties. (It should be noted that our index and algorithms can be generalized to three or more strings, although we only describe our contribution for two strings for simplicity.)

- **Space Reduction**: Our suffix tree has $|A| + l_d + l_1$ leaves where l_d is the sum of the lengths of *all* chunks of B different from A (i.e., $\Sigma_{i=0}^{k}|\delta_i|$) and l_1 is the sum of the lengths of *some* common chunks of A and B. More precisely, l_1 is $\Sigma_{i=0}^{k}|\alpha_i^*|$ where α_i^* is the longest suffix of α_i appearing at least twice in A or in B. The value of α_i^* is $O(\log\max(|A|,|B|))$ on average for random strings [12]. Furthermore, the values of l_d and l_1 are very small in practice. For instance, consider two human genome sequences from two different individuals. Since they are more than 99% identical, l_d is very small compared to $|B|$. We have computed α_i^* for human genome sequences and found out α_i^* is very close to $\log\max(|A|,|B|)$, even though human genome sequences are not random. Hence, our suffix tree is space-efficient for similar strings. Note that the space of our index can be further reduced in the form of compressed indexes such as the compressed suffix tree [9,20]. Our index is an important building block (rather than a final product) towards the goal of efficient indexing for highly similar data.
- **Pattern Search**: Our index is achieved without compromising the linear-time pattern search. That is, using our suffix tree, one can search a pattern P in $O(|P| + occ)$ time, where occ is the number of occurrences of P in A and B. In addition to the linear-time pattern search, we believe that our index supports the most of suffix tree functionalities, e.g., regular expression matchings, matching statistics, approximate matchings, substring range reporting, and so on [3,4,10], because our index is a kind of suffix trees.

We also present an efficient algorithm to construct the suffix tree of alignment. One naïve method to construct our suffix tree is constructing the generalized suffix tree and deleting unnecessary leaves. However, it is not time/space-efficient.

- When our suffix tree for the strings A and B is constructed from scratch, our construction algorithm requires $O(|A|+l_d+l_1+l_2)$ time where l_2 is the sum of the lengths of other parts of common chunks of A and B. More precisely, l_2 is $\Sigma_{i=1}^{k+1}|\hat{\alpha}_i|$ where $\hat{\alpha}_i$ is the longest prefix of α_i such that $d_i\alpha_i$ appears at least twice in A and B (d_i is the character preceding α_i in B. Likewise with l_1, the value of l_2 is also very small compared to $|A|$ or $|B|$ in practice.
- Our algorithm is incremental, i.e., we construct the suffix tree of A and then transform it to the suffix tree of the alignment. Thus, when the suffix tree of A is already given, it requires $O(l_d + l_1 + l_2)$ time. $O(l_d + l_1 + l_2)$ is the minimum time required to make our index a kind of suffix tree so that linear-time pattern search is possible on both A and B. Furthermore, our

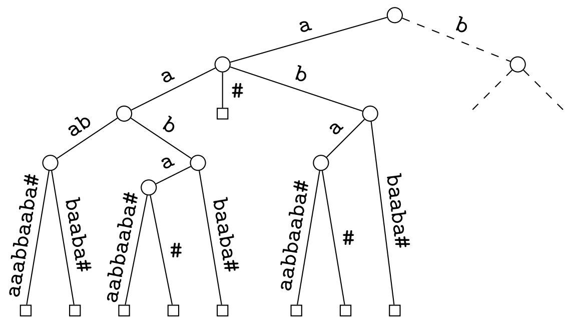

Fig. 1. The suffix tree of string aaabaaabbaaba#

algorithm can be applied to the case when some strings are newly inserted or deleted.

– Our algorithm uses constant-size extra working space except for our suffix tree itself. Thus, it is space-efficient compared to the naïve method.

The space/time-efficiency of our construction algorithm becomes large when handling many strings. The efficiency is feasible when the alignment has been computed in advance, which is the case in some applications. For instance, in the Next-Generation Sequencing, the reference genome sequence is given and the genome sequence of a new individual is obtained by aligning against the reference sequence. So, when a string (a new genome sequence) is obtained, the alignment is readily available. Moreover, since our index does not require that the given alignment is optimal, we can use a near-optimal alignment instead of the optimal alignment if the time to compute an alignment is an important issue. Since the given strings are assumed to be highly similar, a near-optimal alignment can be computed fast from exact string matching instead of dynamic programming requiring much time.

2 Preliminaries

2.1 Suffix Trees

Let S be a string over a fixed alphabet Σ. A substring of S beginning at the first position of S is called a *prefix* of S and a substring ending at the last position of S is called a *suffix* of S. We denote by $|S|$ the length of S. We assume that the last character of S is a special symbol $\# \in \Sigma$, which occurs nowhere else in S.

The *suffix tree* of a string S is a compacted trie with $|S|$ leaves, each of which represents each suffix of S. Figure 1 shows the suffix tree of a string aaabaaabbaaba#. For formal descriptions, the readers are referred to [6,10]. McCreight [18] proposed a linear-time construction algorithm using auxiliary links called *suffix links* and also an algorithm for an *incremental editing*, which transform the suffix tree of $S = \alpha\beta\gamma$ to that of $S' = \alpha\delta\gamma$ for some (possibly empty) string α, β, δ, γ.

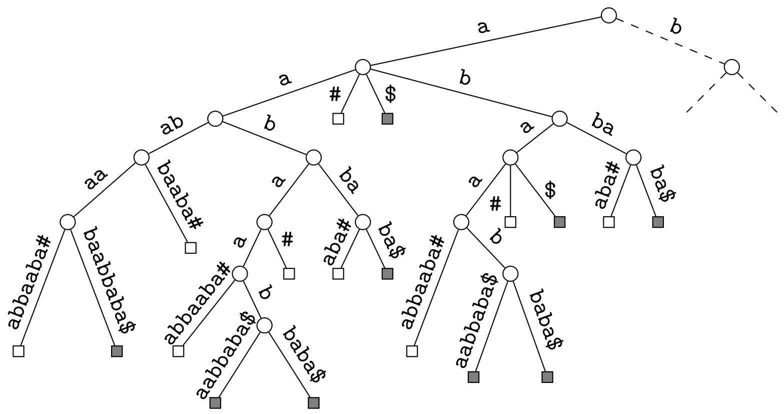

Fig. 2. The generalized suffix tree of two strings A = aaabaaabbaaba# and B = aaabaabaabbaba$. Leaves denoted by white squares and gray squares represent suffixes of A and B, respectively.

The *generalized suffix tree* of two strings A and B is a suffix tree representing all suffixes of the two strings. It can be obtained by constructing the suffix tree of the concatenated string AB where it is assumed that the last characters of A and B are distinct [2,10]. Thus, the generalized suffix tree has $|A| + |B|$ leaves and can be constructed in $O(|A| + |B|)$ time. Figure 2 shows the generalized suffix tree of two strings aaabaaabbaaba# and aaabaabaabbaba$.

2.2 Alignments

Given two strings A and B, an alignment of A and B is a mapping between the two strings that represents how A can be transform to B by replacing substrings of A into those of B. For example, let $A = \alpha\beta\gamma$ and $B = \alpha\delta\gamma$ for some strings α, β, γ, and δ ($\neq \beta$). Then, we can get B from A by replacing β with δ. We denote this replacement by alignment $\alpha(\beta/\,\delta)\gamma$.

More generally, an alignment of two strings $A = \alpha_1\beta_1 \ldots \alpha_k\beta_k\alpha_{k+1}$ and $B = \alpha_1\delta_1 \ldots \alpha_k\delta_k\alpha_{k+1}$, for some $k \geq 1$, can be denoted by $\alpha_1(\beta_1/\,\delta_1)\ldots\alpha_k(\beta_k/\,\delta_k)$ α_{k+1}. For simplicity, we assume that both A and B end with the special symbol $\# \in \Sigma$, which is contained in α_{k+1}. Without loss of generality, we assume the following conditions are satisfied for every $i = 1, \ldots, k$.

- α_{i+1} is not empty (α_1 can be empty).
- Either β_i or δ_i can be empty.
- The first characters of $\beta_i\alpha_{i+1}$ and $\delta_i\alpha_{i+1}$ are distinct.

Note that these conditions are satisfied for the optimal alignments by most of popular distance measures such as the edit distance [15]. Moreover, alignments unsatisfying the conditions can be easily converted to satisfy the conditions. If α_{i+1} ($i = 1, \ldots, k-1$) is empty, β_i and β_{i+1} (δ_i and δ_{i+1}) can be merged. (Note that α_{k+1} cannot be empty since $\#$ is contained in α_{k+1}.) If both β_i and δ_i are empty, α_i and α_{i+1} can be merged. Finally, if the first characters of $\beta_i\alpha_{i+1}$ and $\delta_i\alpha_{i+1}$ are identical (say c), we include c in α_i instead of $\beta_i\alpha_{i+1}$ and $\delta_i\alpha_{i+1}$.

3 Suffix Tree of Simple Alignments

In this section, we define the suffix tree of a simple alignment ($k = 1$) and present how to construct the suffix tree.

3.1 Definitions

For some strings α, β, γ, and δ, let $\alpha(\beta/\,\delta)\gamma$ be an alignment of two strings $A = \alpha\beta\gamma$ and $B = \alpha\delta\gamma$. We define suffixes of the alignment, called *alignment-suffixes* (for short *a-suffixes*). Let α^a and α^b be the longest suffixes of α which occur at least twice in A and B, respectively, and let α^* be the longer of α^a and α^b. That is, α^* is the longest suffix of α which occurs at least twice in A or in B. Then, there are 4 types of a-suffixes as follows.

1. a suffix of γ,
2. a suffix of $\alpha^*\beta\gamma$ longer than γ,
3. a suffix of $\alpha^*\delta\gamma$ longer than γ,
4. $\alpha'(\beta/\,\delta)\gamma$ where α' is a suffix of α longer than α^*. (Note that an a-suffix of this type represents two normal suffixes derived from A and B.)

For example, consider an alignment `aaabaa(abba/baabb)aba#`. Then, α^a and α^b are `baa` and `aabaa`, respectively, and α^* is `aabaa`. Since α^* is `aabaa`, `ba#`, `abaa`$\underline{\texttt{abba}}$`aba#`, $\underline{\texttt{aabb}}$`aba#`, and `aaabaa(`$\underline{\texttt{abba/baabb}}$`)aba#` are a-suffixes of type 1, 2, 3, and 4, respectively (underlined strings denote symbols in β and δ). The reason why we divide a-suffixes longer than $(\beta/\,\delta)\gamma$ into ones longer than $\alpha^*(\beta/\,\delta)\gamma$ (type 4) and the others (types 2 and 3), or why α^* becomes the division point, has to do with properties of suffix trees and we explain the reason later.

The *suffix tree of alignment* $\alpha(\beta/\,\delta)\gamma$ is a compacted trie representing all a-suffixes of the alignment. Formally, the suffix tree T for the alignment is a rooted tree satisfying the following conditions.

1. Each nonterminal arc is labeled with a nonempty substring of A or B.
2. Each terminal arc is labeled with a nonempty suffix of $\beta\gamma$ or $\delta\gamma$, or with $\alpha'(\beta/\,\delta)\gamma$, where α' is a nonempty suffix of α.
3. Each internal node v has at least two children and the labels of arcs from v to its children begin with distinct symbols.

Figure 3 shows the suffix tree of the alignment `aaabaa(abba/baabb)aba#`.

The differences from classic suffix trees of strings (including generalized suffix trees) are as follows. To reduce space, we represent common suffixes of A and B with one leaf. For example, there exists one leaf representing `aba#` in Figure 3 because `aba#` is common suffixes of A and B (type 1). However, suffixes of A and B longer than γ derived from $(\beta/\delta)\gamma$ are not common and thus we deal with these suffixes separately (types 2 and 3). For suffixes longer than $(\beta/\delta)\gamma$,

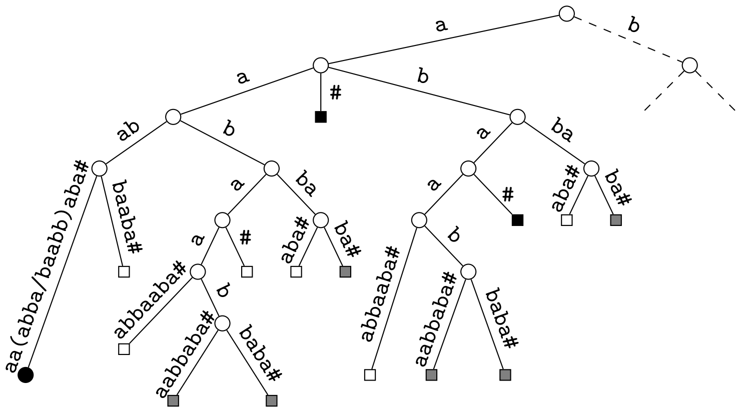

Fig. 3. The suffix tree of an alignment `aaabaa(abba/baabb)aba#`. Leaves denoted by black squares, white squares, gray squares, and black circles represent a-suffixes of types 1, 2, 3, and 4, respectively.

we have two cases. First, consider an a-suffix $\alpha'(\beta/\,\delta)\gamma$ (type 4) such that α' is a suffix of α longer than α^*, e.g., `aaabaa(abba/baabb)aba#`. Due to the definition of α^*, α' appears only once in each of A and B (at the same position) and we can represent $\alpha'(\beta/\,\delta)\gamma$ with one leaf by considering the terminal arc connected to the leaf is labeled with an alignment not a string, e.g., the leftmost (black circle) leaf in Figure 3. However, it cannot be applicable to an a-suffix $\alpha''(\beta/\,\delta)\gamma$ such that α'' is a suffix of α^*, e.g., `abaa(abba/baabb)aba#`. Since α'' appears at least twice in A or in B, $(\beta/\,\delta)$ may not be contained in the label of one arc. Thus, we represent the a-suffix by two leaves, one of which represents $\alpha''\beta\gamma$ (type 2) and the other $\alpha''\delta\gamma$ (type 3), e.g., leaf x representing `abaaabbaaba#` and leaf y representing `abaabaabbaba#` in Figure 3.

Pattern search can be solved using the suffix tree of alignment in the same way as using suffix trees of strings except for handling terminal arcs labeled with alignments. When we meet a terminal arc labeled with an alignment $\alpha'(\beta/\,\delta)\gamma$ during search, we first compare α' with the pattern and then decide which of β and δ we compare with the pattern by checking the first symbols of $\beta\gamma$ and $\delta\gamma$. This comparison is in fact similar to branching at nodes.

3.2 Construction

We describe how to construct the suffix tree T for an alignment. We assume the suffix tree T^A of string A is given. (T^A can be constructed in $O(|A|)$ time [18,21].) To transform T^A into the suffix tree T of the alignment, we should insert the suffixes of B into T^A. We divide the suffixes of B into three groups: suffixes of γ, suffixes of $\alpha^*\delta\gamma$ longer than γ, and suffixes of $\alpha\delta\gamma$ longer than $\alpha^*\delta\gamma$, which correspond to a-suffixes of types 1, 3, and 4, respectively. First, we do not have to do anything for a-suffixes of type 1. The suffixes of type 1 (suffixes of γ) already exist in T^A because these are common suffixes in A and B.

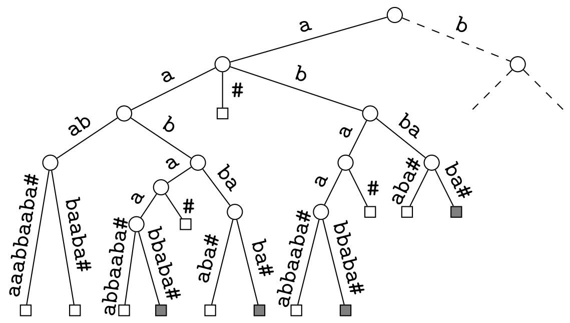

Fig. 4. The tree when Step A is applied to the suffix tree of A in Figure 1

Inserting the suffixes of B longer than γ consists of three steps. We *explicitly* insert the suffixes shorter than or equal to $\alpha^*\delta\gamma$ (type 3) in Steps A and B, and *implicitly* insert the suffixes longer than $\alpha^*\delta\gamma$ (type 4) in Step C as follows.

A. Find α^a and insert the suffixes of $\alpha^a\delta\gamma$ longer than γ.
B. Find α^* and insert the suffixes of $\alpha^*\delta\gamma$ longer than $\alpha^a\delta\gamma$.
C. Insert *implicitly* the suffixes of $\alpha\delta\gamma$ longer than $\alpha^*\delta\gamma$.

In Step A, we first find α^a in T^A using the doubling technique in incremental editing of [18] as follows. For a string χ, we call a leaf a χ-*leaf* if the suffix represented by the leaf contains χ as a prefix. Then, if and only if a string χ occurs at least twice in A, there are at least two χ-leaves in T^A. To find α^a, we check for some suffixes α' of α whether or not there are at least two α'-leaves in T^A. Let $\alpha_{(j)}$ be the suffix of α of length j. We first check whether or not there are at least two $\alpha_{(j)}$-leaves in increasing order of $j = 1, 2, 4, 8, \ldots, |\alpha|$. Suppose $\alpha_{(h)}$ is the shortest suffix among these $\alpha_{(j)}$'s such that there is only one $\alpha_{(j)}$-leaf. (Note that $h/2 \leq |\alpha^a| < h$.) Then, α^a can be found by checking whether or not there are at least two $\alpha_{(j)}$-leaves in decreasing order of $j = h-1, h-2, \ldots$, which can be done efficiently using suffix links [18].

After finding α^a, we insert the suffixes from the longest $\alpha^a\delta\gamma$ to the shortest $d\gamma$ where d is the last character of $\alpha\delta$: Inserting the longest suffix is done by traversing down the suffix tree from the root and inserting the other suffixes can be done efficiently using suffix links [18,21]. Figure 4 shows the tree when Step A is applied to the suffix tree of A in Figure 1.

Let T' be the tree when Step A is finished. In Step B, we first find α^* using T' and insert into T' the suffixes longer than $\alpha^a\delta\gamma$ from the longest $\alpha^*\delta\gamma$ to the shortest in the same way as we did in Step A. Unlike Step A, however, we have the following difficulties for finding α^* because T' is an incomplete suffix tree: i) suffixes of B longer than $\alpha^a\delta\gamma$ are not represented in T', ii) both suffixes of A and suffixes of B are represented in one tree T', and iii) some suffixes of B (a-suffixes of type 1) share leaves with suffixes of A but some suffixes (a-suffixes of type 3) of B do not.

But we show that T' has sufficient information to find α^*. (Recall α^* is the longest suffix of α occurring at least twice in A or in B.) Notice that our goal in

Step B is finding α^* but not α^b. If $|\alpha^b| \leq |\alpha^a|$, for no suffix α' of α longer than α^a, there are at least two α'-leaves in T', in which case $\alpha^* = \alpha^a$. Thus, we do not need to consider suffixes of α shorter than or equal to α^a.

Lemma 1. *For a suffix α' of α longer than α^a, if and only if α' occurs at least twice in B, there are at least two α'-leaves in T'.*

Proof. (If) We first show that there is an α'-leaf in T' due to the occurrence occ_1 of α' as a suffix of α. Since α is common in A and B, occ_1 appears in both A and B as prefixes of $\alpha'\beta\gamma$ and $\alpha'\delta\gamma$, respectively. Note that $\alpha'\delta\gamma$ is not represented in T' but $\alpha'\beta\gamma$ is. Hence, there is an α'-leaf f_1 in T' due to occ_1.

Next, we show that there is another α'-leaf in T' due to an occurrence occ_2 of α' other than occ_1 in B. Let p_1 and p_2 be the start positions of occ_1 and occ_2 in B, respectively, and let p_a be the start position of the suffix $\alpha^a\delta\gamma$ in B. We first prove by contradiction that occ_2 is contained in $\alpha^a\delta\gamma$. Suppose otherwise, that is, p_2 precedes p_a. We have two cases according to which of p_1 and p_2 precedes. First consider the case that p_2 precedes p_1, In this case, occ_2 is properly contained in α, which means that α' appears at least twice in α and also in A. This contradicts with the definition of α^a since α' is longer than α^a. Consider the case that p_1 precedes p_2. Let α'' be the suffix of α starting at p_2. Then, $|\alpha'| > |\alpha''| > |\alpha^a|$ and α'' is a prefix of occ_2. Furthermore, α'' is also a prefix of $\alpha'\delta\gamma$. It means that α'' occurs twice in α as a proper prefix of α' and a proper suffix of α'. This contradicts with the definition of α^a since α'' is longer than α^a. Therefore, p_2 does not precede p_a, which means occ_2 is contained in $\alpha^a\delta\gamma$.

Now we show that there is an α'-leaf f_2 in T' due to occ_2 and f_2 is distinct from f_1. Let η be the suffix of B starting at position p_2. Then, η is a proper suffix of $\alpha^a\delta\gamma$ since p_2 follows p_a. Because T' represents all suffixes of $\alpha^a\delta\gamma$, there exists an α'-leaf f_2 representing η in T'. Moreover, suffixes of A and B share leaves in T' only if they are suffixes of γ. Since the suffix $\alpha'\beta\gamma$ represented by f_1 is longer than γ, f_1 and f_2 are distinct.

(Only if) We prove by contradiction the converse, i.e., if α' occurs only once in B, there is only one α'-leaf in T'. Suppose there are two α'-leaves in T'. Since α' occurs only once in B, no suffix of B except for $\alpha'\delta\gamma$ contains α' as a prefix. Moreover, there is no leaf representing $\alpha'\delta\gamma$ in T' because $|\alpha'\delta\gamma| > |\alpha^a\delta\gamma|$ and no suffix of B longer than $\alpha^a\delta\gamma$ is represented in T'. Thus, no α'-leaf in T' represents a suffix of B and the two α'-leaves in T' represent two suffixes of A. It means α' occurs twice in A, which contradicts with the definition of α^a that α^a is the longest suffix of α occurring at least twice in A since $|\alpha'| > |\alpha^a|$. Therefore, there is only one α'-leaf in T' if α' occurs only once in B. □

Corollary 1. *For a suffix α' of α longer than α^a, if and only if α' occurs at least twice in A or in B, there are at least two α'-leaves in T'.*

By Corollary 1, we can find α^* by checking for some suffixes α' of α longer than α^a whether or not there are at least two α'-leaves in T'. It can be done in $O(|\alpha^*|)$ using the way similar to Step A. When Step B is applied to the tree in Figure 4, the resulting tree is the same as the tree in Figure 3 except that the terminal arc connected to the leftmost leaf (black circle) is labeled with suffix `aaabbaaba#` of string A but not with a-suffix `aa(abba/baabb)aba#` of the alignment.

In Step C, for every suffix α' of α longer than α^*, we *implicitly* insert the suffix $\alpha'\delta\gamma$ of B. Since the suffix $\alpha'\delta\gamma$ of B and the suffix $\alpha'\beta\gamma$ of A (a-suffixes of type 4) should be represented by one leaf, we do not insert a new leaf but convert the leaf representing $\alpha'\beta\gamma$ to represent the a-suffix $\alpha'(\beta/\,\delta)\gamma$. It can be done by replacing *implicitly* every β properly contained in labels of terminal arcs with $(\beta/\,\delta)$. Consequently, we explicitly do nothing in Step C, and these implicit changes are already reflected in the given alignment. For example, the suffix tree in Figure 3 is obtained by replacing implicitly the label `aa`$\underline{\texttt{abba}}$`aba#` of the terminal arc connected to the leftmost leaf (black circle) with a-suffix `aa(abba/baabb)aba#` of the alignment.

We consider the time complexity of our algorithm. In step A, finding α^a takes $O(|\alpha^a|)$ time and inserting suffixes takes $O(|\alpha^a\delta\hat{\gamma}|)$ time, where $\hat{\gamma}$ is the longest prefix of γ such that $d\hat{\gamma}$ occurs at least twice in A and B (where d is the character preceding γ). For detailed analysis, the readers are referred to [18]. In step B, similarly, finding α^* takes $O(|\alpha^*|)$ time and inserting suffixes takes $O(|\alpha^*\delta\hat{\gamma}|)$ time. Step C takes no time since it is implicitly done. Thus, we get the following theorem.

Theorem 1. *Given an alignment $\alpha(\beta/\,\delta)\gamma$ and the suffix tree of string $\alpha\beta\gamma$, the suffix tree of $\alpha(\beta/\,\delta)\gamma$ can be constructed in $O(|\alpha^*| + |\delta| + |\hat{\gamma}|)$ time.*

4 Suffix Tree of General Alignments

We extend the definitions and the construction algorithm into more general alignments. Let $\alpha_1(\beta_1/\,\delta_1)\ldots\alpha_k(\beta_k/\,\delta_k)\alpha_{k+1}$ be an alignment of two strings $A = \alpha_1\beta_1\ldots\alpha_k\beta_k\alpha_{k+1}$ and $B = \alpha_1\delta_1\ldots\alpha_k\delta_k\alpha_{k+1}$. For $1 \le i \le k+1$, let α_i^a and α_i^b be the longest suffixes of α_i occurring at least twice in A and B, respectively, and let α_i^* be the longer of α_i^a and α_i^b. That is, α_i^* is the longest suffix of α_i which occurs at least twice in A or in B. Moreover, let $\hat{\alpha}_i$ be the longest prefix of α_i such that $d_i\hat{\alpha}_i$ occurs at least twice in A and B where d_i is the character preceding α_i in B.

The suffix tree of the alignment is a compacted trie that represents the following a-suffixes of the alignment.

1. a suffix of α_{k+1},
2. a suffix of $\alpha_i^*\beta_i\alpha_{i+1}\ldots\alpha_{k+1}$ longer than $\alpha_{i+1}\ldots\alpha_{k+1}$,
3. a suffix of $\alpha_i^*\delta_i\alpha_{i+1}\ldots\alpha_{k+1}$ which is longer than $\alpha_{i+1}\ldots\alpha_{k+1}$,
4. $\alpha_i'(\beta_i/\,\delta_i)\ldots\alpha_{k+1}$, where α_i' is a suffix of α_i longer than α_i^*.

Given the suffix tree of A, the suffix tree of the alignment can be constructed as follows (the details are omitted).

A1. Find α_i^a using the suffix tree of A for each i $(1 \le i \le k)$.
A2. Insert the suffixes of $\alpha_i^a\delta_i\alpha_{i+1}\ldots\alpha_{k+1}$ longer than $\alpha_{i+1}\ldots\alpha_{k+1}$ for each i.
B1. Find α_i^* for each i.

B2. Insert the suffixes of $\alpha_i^*\delta_i \dots \alpha_{k+1}$ longer than $\alpha_i^a\delta_i \dots \alpha_{k+1}$ for each i.
C. Insert *implicitly* the suffixes of $\alpha_i\delta_i \dots \alpha_{k+1}$ longer than $\alpha_i^*\delta_i \dots \alpha_{k+1}$ for each i.

Theorem 2. *Given an alignment $\alpha_1(\beta_1/\,\delta_1)\dots\alpha_k(\beta_k/\,\delta_k)\alpha_{k+1}$ and the suffix tree of string $\alpha_1\beta_1\dots\alpha_k\beta_k\alpha_{k+1}$, the suffix tree of the alignment can be constructed in time at most linear to the sum of the lengths of α_i^*, δ_i, $\hat{\alpha}_{i+1}$ for $1 \le i \le k$.*

Our definitions and algorithms can be also extended into alignments of more than two strings. For example, consider an alignment $\alpha(\beta/\,\delta/\,\vartheta)\gamma$ of three strings $A = \alpha\beta\gamma$, $B = \alpha\delta\gamma$, and $C = \alpha\vartheta\gamma$ such that the first characters of $\beta\gamma$, $\delta\gamma$, and $\vartheta\gamma$ are distinct. We define α^a, α^b, and α^c as the longest suffix of α which occurs at least twice in A, B, and C, respectively, and α^* as the longest of α^a, α^b, and α^c. Then, there are 5 types of a-suffixes. (Suffixes of $\alpha^*\vartheta\gamma$ longer than γ are added as a new type of a-suffixes.) The suffix tree of the alignment can be defined similarly and constructed as follows: From the suffix tree of $\alpha\beta\gamma$, we construct the suffix tree of $\alpha(\beta/\,\delta)\gamma$ by inserting some suffixes of B, and then convert into the suffix tree of $(\beta/\,\delta/\,\vartheta)\gamma$ by inserting suffixes of C (and some suffixes of B occasionally). We omit the details.

Acknowledgements. Joong Chae Na was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(2012-0003214), and by the IT R&D program of MKE/KEIT [10038768, The Development of Supercomputing System for the Genome Analysis]. Heejin Park was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(2012-0006999), by Seoul Creative Human Development Program (HM120006), by the Proteogenomics Research Program through the National Research Foundation of Korea funded by the Korean Ministry of Education, Science and Technology, and by the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning(2012-054452). Jan Holub has been partially supported by the Czech Science Foundation (GAČR) as project No. 13-03253S. Kunsoo Park was supported by National Research Foundation of Korea-Grant funded by the Korean Government(MSIP) (2012K1A3A4A07030483).

References

1. The 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing 467(7319), 1061–1073 (2010)
2. Amir, A., Farach, M., Galil, Z., Giancarlo, R., Park, K.: Dynamic dictionary matching. J. Comput. Syst. Sci. 49, 208–222 (1994)
3. Baeza-Yates, R.A., Gonnet, G.H.: Fast text searching for regular expressions or automaton searching on tries. J. ACM 43(6), 915–936 (1996)
4. Bille, P., Gørtz, I.L.: Substring range reporting. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 299–308. Springer, Heidelberg (2011)

5. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, California (1994)
6. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific Publishing, Singapore (2002)
7. Do, H.H., Jansson, J., Sadakane, K., Sung, W.-K.: Fast relative lempel-ziv self-index for similar sequences. In: Snoeyink, J., Lu, P., Su, K., Wang, L. (eds.) FAW-AAIM 2012. LNCS, vol. 7285, pp. 291–302. Springer, Heidelberg (2012)
8. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. ACM 47(6), 987–1011 (2000)
9. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. 35(2), 378–407 (2005)
10. Gusfield, D.: Algorithms on Strings, Tree, and Sequences. Cambridge University Press, Cambridge (1997)
11. Huang, S., Lam, T.W., Sung, W.K., Tam, S.L., Yiu, S.M.: Indexing similar dna sequences. In: Chen, B. (ed.) AAIM 2010. LNCS, vol. 6124, pp. 180–190. Springer, Heidelberg (2010)
12. Karlin, S., Ghandour, G., Ost, F., Tavare, S., Korn, L.J.: New approaches for computer analysis of nucleic acid sequences. Proc. Natl. Acad. Sci. 80(18), 5660–5664 (1983)
13. Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. Theor. Comput. Sci. (to appear)
14. Kuruppu, S., Puglisi, S.J., Zobel, J.: Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 201–206. Springer, Heidelberg (2010)
15. Levenshtein, V.: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady 10(8), 707–710 (1966)
16. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of individual genomes. In: Batzoglou, S. (ed.) RECOMB 2009. LNCS, vol. 5541, pp. 121–137. Springer, Heidelberg (2009)
17. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. J. Comput. Bio. 17(3), 281–308 (2010)
18. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM 23(2), 262–272 (1976)
19. Navarro, G.: Indexing highly repetitive collections. In: Arumugam, S., Smyth, B. (eds.) IWOCA 2012. LNCS, vol. 7643, pp. 274–279. Springer, Heidelberg (2012)
20. Sadakane, K.: Compressed suffix trees with full functionality. Theor. Comput. Sci. 41(4), 589–607 (2007)
21. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
22. Weiner, P.: Linear pattern matching algorithms. In: Proc. of the 14th IEEE Symp. on Switching and Automata Theory, pp. 1–11 (1973)
23. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. on Information Theory 23(3), 337–343 (1977)