Generalization of a Suffix Tree for RNA Structural Pattern Matching

Tetsuo SHIBUYA
IBM Tokyo Research Laboratory,
1623-14, Shimotsuruma, Yamato-shi, Kanagawa 242-8502, Japan.
Phone: +81-46-215-5915 Fax: +81-46-273-7428
e-mail: tshibuya@jp.ibm.com

Abstract

In molecular biology, it is said that two biological sequences tend to have similar properties if they have similar 3-D structures. Hence, it is very important to find not only similar sequences in the string sense, but also structurally similar sequences from databases. In this paper, we propose a new data structure that is a generalization of a parameterized suffix tree (p-suffix tree for short) introduced by Baker. We call it the structural suffix tree or s-suffix tree for short. The s-suffix tree can be used for finding structurally related patterns of RNA or single-stranded DNA. Furthermore, we propose an $O(n(\log|\Sigma| + \log|\Pi|))$ on-line algorithm for constructing it, where n is the sequence length, $|\Sigma|$ is the size of the normal alphabet, and $|\Pi|$ is that of the alphabet called "parameter", which is related to the structure of the sequence. Our algorithm achieves linear time when it is used to analyze RNA and DNA sequences. Furthermore, as an algorithm for constructing the p-suffix tree, it is the first on-line algorithm, though the computing bound of our algorithm is same as that of Kosaraju's best-known algorithm. The results of computational experiments using actual RNA and DNA sequences are also given to demonstrate our algorithm's practicality.

keywords: pattern matching, suffix tree generalization, parameterized suffix tree, computational biology, RNA structure matching

1 Introduction

The 3-D structure of a biological sequence plays a major role in determining its functions and properties, and sequences that have similar structures often have similar functions, even if the sequences themselves are not similar. Thus it is very important to predict, compare, or find structures of sequences in molecular biology, but they are very difficult and challenging tasks. On the other hand, stringological comparison of biological sequences without considering their structures is much easier. There are many efficient algorithms for it. For example, we can find frequently appearing substrings in a sequence very efficiently using a very useful data structure called suffix trees [9, 10, 15, 18, 21]. In this paper, we strive for generalizing the suffix trees to do structural analysis of RNA sequences as efficiently as we do ordinary string analysis.

RNA sequences consist of four kinds of bases: A (adenine), U (uracil), C (cytosine), and G (guanine). Note that in DNA, T (thymine) is present instead of U. A and U (T for DNA) are said to be complements of each other, and C and G are also complementary bases. RNA and single-stranded DNA sequences often form some structures by combining two complementary base pairs. It is known that double-stranded DNA sequences sometimes form such structures by becoming single-stranded locally. Many computational studies have been done to predict RNA secondary structure, comparing a new sequence with a known RNA structure, searching a known RNA or DNA structures from large databases, and so on [2, 10, 12, 13, 16, 17, 19, 20]. But many of them are much slower than ordinary string comparison analyses which are often done in linear time. Our work will find a way out of it.

Let us consider the two RNA sequences in Figure 1 (1). The two sequences are not at all similar to each other: there are no identical bases in identical positions. In sequence 1, A's are located at the 1st, 3rd, 8th, and 15th positions. In sequence 2, C's are located at the same position as A's in sequence 1. Similarly, A's, U's, and G's in sequence 2 are located at the same positions as G's, C's, and U's in sequence 1, respectively. Recall that A and U can combine with each other, and that C and G can also combine with each other. We then notice the following fact: If two bases in one of these sequences can combine with each other, then in the other sequence, two bases at the same two positions are also able to combine with each other. This implies that a structure that can be formed by one of the sequences can also be formed by the other sequence. Thus there is a strong possibility that these two sequences have the same structure, and consequently may have similar properties. For example, Figure 1 (2) shows one of the structures that can be formed by both sequences.

To deal with these structurally related sequences, we propose a new matching paradigm called the structural matching (s-matching for short), which is a generalization of the parameterized matching (p-matching for short) introduced by Baker [3, 4, 5, 6, 7]. Note that we will describe the p-matching paradigm in section 2.1 as a preliminary. In the s-matching paradigm, a notion of matching is different from an ordinary matching, as in the following definition.

Definition 1 *Let Σ and Π be disjoint finite alphabets. We call the characters in Σ the "fixed symbols" and those in Π the "parameters." Some of the characters in Π have one-to-one correspondences to other characters in Π, and two characters that correspond to each other are called complementary characters or complements of the other. No two characters can be complements of one same character, and let complement(x) be a function that outputs the complement of a parameter x. A string in $(\Sigma \cup \Pi)^*$ is called a structural string, or s-string for short. Two s-strings S and S' are said to s-match if they satisfy the following two conditions: (1) there exists a one-to-one mapping from Π to Π such that S becomes S' as a result of applying it, and (2) if x is mapped to y in the mapping, then the complement of x is also mapped to the complement of y in the mapping.*

For example, if $\Sigma = \{\mathtt{A},\mathtt{B}\}$, $\Pi = \{x, y, z, w\}$, and x and y are complements of z and w, respectively, then ABxByAzwz and ABwBxAyzy s-match, but ABxByAzwz and ABwBxAzyz do not. Note that if there are no complementary pairs in Π, an s-string is same as the p-string proposed by Baker, which will be introduced in section 2.1.

Then a pair of structurally related RNA sequences stated above can be described as a pair of s-matching sequences in the following situation if they have at least one pair of complementary bases in them: $\Sigma = \phi$, $\Pi = \{$`A, U, C, G`$\}$, and `A` and `C` are complementary characters of `U` and `G`, respectively. If two RNA sequences s-match with each other, it can be said that there is a high possibility that the two sequences have the same structure and that they may have similar properties as a result. For example, the two sequences in Figure 1 (1) s-match.

We will generalize the suffix tree data structure to deal with the s-matching strings, and call it the structural suffix tree (or s-suffix tree for short). Using the s-suffix tree, we can efficiently find a set of substrings in some given sequence(s) that are structurally related (maybe identical), query substrings that might be structurally related to another given string, and so on. We will at first propose an $O(n(\log|\Sigma| + |\Pi|^2))$ on-line algorithm for constructing the s-suffix tree, where n is the sequence length, $|\Sigma|$ is the size of the normal alphabet, and $|\Pi|$ is that of the parameter alphabet. After that, we will improve it to an $O(n(\log|\Sigma| + \log|\Pi|))$ time algorithm. These algorithms achieve linear time when they are used to analyze RNA and DNA sequences. Furthermore, as an algorithm for constructing the p-suffix tree, our algorithm is the first on-line algorithm, though the computing bound of our algorithm is same as that of Kosaraju's best-known algorithm [14].

In the following, we first introduce suffix trees and p-suffix trees, and briefly describe Ukkonen's suffix tree construction algorithm and Baker's p-suffix tree construction algorithm, as preliminaries in section 2. In section 3, we first define the s-suffix trees, and we then propose an efficient on-line algorithms for constructing the s-suffix trees based on the Ukkonen's algorithm. In section 4, we give the results of computational experiments using HIV RNA complete sequences and DNA sequences of E. coli (Escherichia coli). In section 5, we will give concluding remarks.

2 Preliminaries

2.1 Suffix Trees and p-Suffix Trees

The suffix tree [9, 10, 15, 18, 21] of a string $S \in \Sigma^n$ is the compacted trie of all the suffixes of $S^+ = S\$$ where \$ is a character such that $\$ \notin \Sigma$. This data structure is very useful for various problems in sequence pattern matching. Using it, we can query a substring of length m in $O(m \log|\Sigma|)$ time [10]. Moreover, we can find frequently appearing substrings in a given sequence or a common substring of many sequences very easily.

The tree has $n+1$ leaves, and each internal node has more than one child. Each edge is labeled with a non-empty substring of S^+, and no two edges out of a node can have labels that start with the same character. Each node is labeled with the concatenated string of edge labels on the path from the root to the node, and each leaf has a label that is a different suffix of S^+. Because each edge label is represented by the first and the last indices of the corresponding substring in S^+, the data structure can be stored in $O(n)$ space.

This data structure was first proposed by Weiner [21], who gave an $O(n|\Sigma|)$ algorithm for constructing it, where n is the string length and $|\Sigma|$ is the size of the alphabet. McCreight [15] improved it by giving an $O(n \log|\Sigma|)$ algorithm. After that, Ukkonen [18] proposed an on-line $O(n \log|\Sigma|)$ algorithm, which processes a string character by character from left to right. Then Farach [9] proposed an $O(n)$ algorithm for an integer alphabet $\{1, \ldots, n\}$.

A parameterized string, or a p-string for short, is a string over the union of two alphabets

Σ and Π, where Σ is an ordinary alphabet and Π is a set of parameters [3, 4, 5, 6, 7]. Two p-strings are said to match if they are same except for a one-to-one correspondence between the characters in Π occurring in them. For example, two p-strings $\mathtt{AC}x\mathtt{BC}yzy\mathtt{A}zx\mathtt{C}$ and $\mathtt{AC}y\mathtt{BC}zxz\mathtt{A}xy\mathtt{C}$ match ($\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $\Pi = \{x, y, z\}$).

Let $S[i]$ denote the ith character of S, and $S[i..j]$ denote a substring of S that starts at position i and ends at position j. As in [5], we define prev(S) for any p-string S as follows:

Definition 2 *Let N be the set of nonnegative integers. Consider a string $S[1..n] \in (\Sigma \cup \Pi)^n$. If $S[i] \in \Pi$, let c_i be the index of the nearest same parameter in Π to the left, i.e., $c_i < i$, $S[c_i] = S[i]$ and $S[k] \neq S[i]$ for any k such that $c_i < k < i$. If such c_i does not exist, let $c_i = i$. Now, replace $S[i]$ with $i - c_i \in N$ if $S[i] \in \Pi$, for all i: We let the obtained string in $(\Sigma \cup N)^n$ be prev(S).*

For example, prev($\mathtt{AC}x\mathtt{BC}yzy\mathtt{A}zx\mathtt{C}$) = $\mathtt{AC0BC002A38C}$. We can compute the prev encoding for string S of size n in $O(n \cdot \log \min(n, |\Pi|))$ time and $O(n)$ space by means of a balanced tree structure, which can be computed on-line. If Π is known and can be used as an index to a table of $|\Pi|$, it can be computed in $O(n + |\Pi|)$ time and space. (The algorithm for it will be given in section 3.1.) The p-suffix tree of a p-string S is the compacted trie of all the prev-encoded suffixes, *i.e.*, prev($S^+[i..n+1]$) for all positions i, where $S^+ = S\$$ and \$ is a character in neither Σ nor Π. We here consider \$ as an ordinary alphabet character, not as a parameter. Notice that a suffix of a prev encoding of string S is sometimes different from the prev encoding of the suffix of S with the same length, which cause a difficulty in constructing the p-suffix trees. Baker [3, 5, 6] proposed this data structure and showed that it can be constructed in $O(n(|\Pi| + \log |\Sigma|))$ time. Kosaraju [14] improved the time by giving an $O(n(\log |\Pi| + \log |\Sigma|))$ algorithm. Both of the algorithms are based on McCreight's suffix tree construction algorithm [15] and neither supports on-line computation. This paper will give an on-line algorithm for the same task, based on Ukkonen's algorithm [18].

In the following sections, we use the following definitions. In a suffix tree, let $parent(u)$ be the parent node of node u, let σ_u be the string label of node u, and let $node(\alpha)$ be node u in the tree such that $\sigma_u = \alpha$ if it exists. Let the length of an edge (v, w) ($v = parent(w)$) be the length of the corresponding substring, *i.e.*, $|\sigma_w| - |\sigma_v|$. A 'locus' is a position on an edge (including both ends, *i.e.*, nodes) of a suffix tree. Let z be a locus on an edge (v, w) ($v = parent(w)$) whose distance from v is d ($0 < d \leq |\sigma_w| - |\sigma_v|$), and let α be a prefix of σ_w whose length is $|\sigma_v| + d$. We call z the locus of α. Conversely, we call α the label of the locus z. The locus of α can be considered to be the same as $node(\alpha)$ if it exists, therefore we also call $node(\alpha)$ the locus of α in this case. The suffix link of u is a link to a node with label α if u has a label of $c\alpha$, where c is any single character. It is known that a suffix link always exists for any u except for the root in a suffix tree [10, 15, 18]. If u is the root we let its suffix link be u itself. Let $sl(u)$ be the suffix link of u.

2.2 Ukkonen's Suffix Tree Construction Algorithm

In this section, we briefly describe Ukkonen's suffix tree construction algorithm. The implicit suffix tree of S is the compacted trie of all the suffixes of S, and a label for an edge that ends at a leaf is represented by only the first index of the label. Let T_i ($1 \leq i \leq n+1$) denote the implicit suffix tree of $S^+[1..i]$, where $n = |S|$. Ukkonen's algorithm consists of $n+1$ phases, and in the ith phase, we construct the implicit suffix tree T_i from T_{i-1}.

In the ith phase, we construct a new node $u = node(S^+[j..i])$ for all $1 \leq j \leq i$ in increasing order, if there is no locus for $S^+[j..i]$ in the tree. When we must construct such new node u, if there is no node with a label of $S^+[j..i-1]$, we must also construct a new internal node at the locus of $S^+[j..i-1]$, and let it be the parent of u. We call this procedure for single j the jth extension of the ith phase. Notice that we do not have to construct node $u = node(S^+[j..i])$ if $v = node(S^+[j..i-1])$ was a leaf in the previous phase, because of the definition of the implicit suffix tree: σ_v is $S^+[j..i]$ in this phase. Thus, if there is a leaf for each of $node(S^+[j..i-1])$ for all $j < k$ in phase $i-1$, we can begin by constructing $node(S^+[j+1..i])$ in this phase. Furthermore, if there is a locus for $S^+[j..i]$ for some j, there already exist loci for $S^+[k..i]$ $(k > j)$ too, and that there is no need to construct nodes for them in this phase.

Ukkonen's algorithm maintains at each node u of the suffix tree a suffix link $sl(u)$. In any phase, we construct new leaves $u_j = node(S^+[j..i])$ for several consecutive j's and a new internal node u'_j =node($S^+[j..i-1]$) if necessary, in the manner described above. Notice that $u_{j+1} = sl(u_j)$ and $u'_{j+1} = sl(u'_j)$ if they exist. For the last u'_j to be constructed in this phase, we will encounter an existing node at the locus $sl(u'_j) = S^+[j+1..i-1]$ in the next extension of the algorithm. For the last u_j to be constructed in this phase, which is called an 'active point' by Ukkonen, the suffix link will be found as u_{j+1} constructed in a later phase. In this way, we can maintain the suffix links. Note that the suffix links of leaves are not necessary to be maintained in the Ukkonen's algorithm, because we do not use them. Using the suffix links, we can construct node $u_j = node(S^+[j..i])$ faster: As $sl(parent(u_{j-1}))$ must be an ancestor of u_j, we can find the locus of $S^+[j..i-1]$ by tracing edges from $sl(parent(u_{j-1}))$. We call tracing from the suffix link to the target locus "scanning." (Note that Ukkonen called it "canonizing.") Let w be a node scanned in the jth extension of the ith phase. Then the label of w is $S^+[j..f_{j,w}]$ where $f_{j,w} = j + |\sigma_w| - 1$. As the value $f_{j,w}$ differs from any other scanned nodes in any phase of the algorithm, the number of scanned nodes is at most $n+1$. Moreover, the number of constructed nodes is also $O(n)$ and the number of phases is $n+1$. Therefore the total time complexity of this algorithm is $O(n \log |\Sigma|)$, as it takes $O(\log |\Sigma|)$ time to find the outgoing edge with the desired label from a node.

2.3 Baker's p-Suffix Tree Construction Algorithm

In this section, we briefly introduce McCreight's suffix tree construction algorithm [15] and Baker's p-suffix tree construction algorithm [3, 5, 6] which is based on the McCreight's algorithm.

The McCreight's algorithm computes the suffix tree by inserting suffixes, $S^+[1..n+1], S^+[2..n+1], \ldots, S^+[n+1..n+1]$, one after the other in this order, into an initially empty tree. We call the the insertion procedure of nodes related to the suffix $S^+[i..n+1]$ the ith phase. In the ith phase, the algorithm first finds the locus of the longest prefix of the suffix $S^+[i..n+1]$ that exists in the trie constructed in the previous $i-1$ phases. Then the algorithm adds a new node to the locus and add a new leaf whose label is $S^+[i..n+1]$ as a child of the new node. Like the Ukkonen's algorithm, it also uses suffix links to find where to insert the new node and the new leaf to the trie. After a new node v and a new leaf are inserted, it scans the trie from $w = sl(parent(v))$ to find the new locus to insert another node for the next suffix, as w is an ancestor of the next locus to find. The algorithm also finds the suffix link of v during the scanning, which is guaranteed to exist. The time complexity is $O(n \log |\Sigma|)$, which is same as Ukkonen's, but it does not support on-line computation.

Baker's p-suffix tree construction algorithm is based on the McCreight's algorithm. It also inserts p-suffixes one by one in the same order as in the McCreight's algorithm. The difference between the two algorithms lies in the suffix links, except for which the two algorithms are the same. In p-suffix trees, unlike ordinary suffix trees, the existence of suffix links are not guaranteed for all the nodes in the tree. For example, consider the case that two substrings $x\mathtt{BC}yzy\mathtt{A}zx$ and $y\mathtt{BC}xwx\mathtt{A}wz$ exist in the target string. The prev-encoding of these two substrings are $0\mathtt{BC}002\mathtt{A}38$ and $0\mathtt{BC}002\mathtt{A}30$, and there exists a node v at the locus of $0\mathtt{BC}002\mathtt{A}3$. The suffix link of v must points to a node at the locus of $\mathtt{BC}002\mathtt{A}3$, but the prev-encoding of two substrings $\mathtt{BC}yzy\mathtt{A}zx$ and $\mathtt{BC}xwx\mathtt{A}wz$ are both $\mathtt{BC}002\mathtt{A}30$, and there is no guarantee of the existence of the node. Baker's algorithm uses implicit suffix link that points to the node above the locus of the corresponding suffix. While computing the p-suffix tree, the implicit suffix links must be changed often, because the node to point must be changed when new nodes are inserted between the pointed node and the desired locus. Baker showed that the maintenance of the implicit suffix links during the computation of a p-suffix tree can be done in $O(n|\Pi|)$ time in total, and the rest of the computation can be done in $O(n \log |\Sigma|)$ time. Thus the total computation time of the Baker's algorithm is $O(n(|\Pi| + \log |\Sigma|))$. Kosaraju [14] further improved the time complexity to $O(n(\log |\Pi| + \log |\Sigma|))$ by maintaining the implicit suffix links with a data structure called a concatenable queues [1] (c-queue for short), but it is a very complicated algorithm.

Our algorithm presented hereinafter will be based on the Ukkonen's algorithm, not on the McCreight's algorithm, but it will also use the implicit suffix links (though the definition is a little different). The maintenance of the implicit suffix links is also a problem in our algorithm, and we will solve it in the next section. We will also use the c-queues to improve the time complexity of our algorithm as Kosaraju did to the Baker's algorithm, but in a different way.

3 Structural Suffix Trees

3.1 s-Encodings and s-Suffix Trees

In this section we define the s-encodings and the s-suffix trees, which are useful to study s-strings. The following two encodings are useful for determining s-matching of two sequences described in Definition 1. One is prev(S) that is already defined in Definition 2. The other is compl(S) defined as follows:

Definition 3 *Let N be the set of nonnegative integers ($N \cap (\Sigma \cup \Pi) = \emptyset$). Consider a string $S[1..n] \in (\Sigma \cup \Pi)^n$. If $S[i] \in \Pi$, let c_i be the index of the nearest complementary parameter in Π to the left, i.e., $c_i < i$, $S[c_i] = x_i$ and $S[k] \neq x_i$ for any k such that $c_i < k < i$, where x_i is the complement of $S[i]$. If such c_i does not exist, let $c_i = i$. Now, replace $S[i]$ with $i - c_i \in N$ if $S[i] \in \Pi$, for all i: We let the obtained string in $(\Sigma \cup N)^n$ be compl(S).*

For example, compl($\mathtt{AB}x\mathtt{B}y\mathtt{A}zwz$) $=$ $\mathtt{AB}0\mathtt{B}0\mathtt{A}436$ if $\Sigma = \{\mathtt{A},\mathtt{B}\}$, $\Pi = \{x, y, z, w\}$, and x and y are complements of z and w, respectively. This definition is very similar to that of prev encoding.

The complement of a given character can be accessed in $O(\log |\Pi|)$ time if the information is stored in a sorted array or a balanced tree data structure, either of which can be constructed in

$O(|\Pi| \log |\Pi|)$ time. If Π can be used as an index to a table, the complement can be obtained in $O(1)$ time. The computation of these two encodings can be computed on-line as follows. Let *last_position*(p) be an indexing data structure for parameters $p \in |\Pi|$ whose contents are integers. If Π is known and can be used as an index to a table of $|\Pi|$, an update of the content for a parameter can be done in $O(1)$ time. Otherwise the update takes $O(\log m)$ time, where m is the number of used parameters, using a balanced search tree.

Algorithm 1 (prev & compl Computation) *Let last_position$(p) = 0$ for all the parameters p at first. For $1 \leq i \leq n$, let prev$(S)[i]$ = compl$(S)[i] = S[i]$ if $S[i]$ is an ordinary alphabet character, and otherwise do the following.*

1. *Set 0 to prev$(S)[i]$ if last_position(prev$(S)[i]) = 0$. Otherwise set $i -$ last_position(prev$(S)[i]$) to prev$(S)[i]$.*
2. *Similarly, set 0 to compl$(S)[i]$ if last_position(complement(prev$(S)[i]$)) $= 0$. Otherwise set $i -$ last_position(complement(prev$(S)[i]$)) to compl$(S)[i]$.*
3. *Set i to last_position$(S[i])$.*

In this way, we can compute on-line both the prev and the compl encodings for a string S of size n in $O(n \cdot \log \min(n, |\Pi|))$ time and $O(n)$ space. If Π is known and can be used as an index to a table of $|\Pi|$, it can be computed in $O(n + |\Pi|)$ time and space. The relationship between the s-matching and these two encodings is given in the following lemma.

Lemma 1 *s-strings S and S' are an s-match if and only if prev(S) = prev(S') and compl(S) = compl(S').*

Proof: Two s-matching s-strings satisfy the two conditions in Definition 1. Trivially, applying to S a one-to-one mapping from Π to Π does not change $\text{prev}(S)[i]$ for any i, because the position of the previous same character of any character does not change. Furthermore, because of the condition (2), the mapping does not change $\text{compl}(S)[i]$ for any i.

Next, assume that two s-strings S and S' satisfy $\text{prev}(S)[i] = \text{prev}(S')[i]$ for any i. If $S[i] = S[j]$ $(i < j)$, there is a sequence of indices $i_1, i_2, \ldots, i_k$ such that $i = i_1$, $j = i_k$, and $i_l = i_{l+1} - \text{prev}(S)[i_{l+1}]$ for all l $(1 \leq l < k)$. As the prev encoding of S' is equal to S, $S'[i_1] = S'[i_2] = \cdots = S'[i_k]$. Thus we conclude that $S'[i] = S'[j]$ if $S[i] = S[j]$. By doing the same discussion we can prove that $S[i] = S[j]$ if $S'[i] = S'[j]$. Thus the two s-strings satisfy the condition (1).

Finally, assume that the two s-strings S and S' satisfy $\text{compl}(S)[i] = \text{compl}(S')[i]$ and $\text{prev}(S)[i] = \text{prev}(S')[i]$ simultaneously for any i. Consider two indices i and j such that $i < j$ and $S[i] = \textit{complement}(S[j])$. Let $k = j - \text{compl}(S)[j]$. Then $S[i] = S[k]$. As already proved, it means that $S'[i] = S'[k]$. As the compl encodings of the two s-strings are the same, $S'[i] = S'[k] = \textit{complement}(S'[j])$. By doing the same discussion, we can prove that $S[i] = \textit{complement}(S[j])$ if $S'[i] = \textit{complement}(S'[j])$. Consequently they satisfy the condition (2). Hence we conclude that S and S' are an s-match if and only if $\text{prev}(S) = \text{prev}(S')$ and $\text{compl}(S) = \text{compl}(S')$. □

Furthermore, we obtain the following lemma.

Lemma 2 *Assume that $S[1..i]$ and $S'[1..i]$ are an s-match. Then, if prev(S)[i+1] = prev(S')[i+1] $\neq 0$, compl(S)[i + 1] = compl(S')[i + 1]. Similarly, if compl(S)[i + 1] = compl(S')[i + 1] $\neq 0$, prev(S)[i + 1] = prev(S')[i + 1].*

Proof: At first, we consider the former case where $\text{prev}(S)[i+1] = \text{prev}(S')[i+1] \neq 0$. Let j be $i+1-\text{prev}(S)[i+1]$ $(= i+1-\text{prev}(S')[i+1])$. According to the definition of the prev encoding, $S[j] = S[i+1]$ and $S'[j] = S'[i+1]$. Let k be the largest integer such that $S[k] = \mathit{complement}(S[j])$ and $1 \leq k \leq i$. If it does not exist, let $k = i+1$. As $S[1..i]$ and $S'[1..i]$ are an s-match, the k value is also the largest integer such that $S'[k] = \mathit{complement}(S'[j])$ if $k \leq i$. It means that $\text{compl}(S)[i+1] = \text{compl}(S')[i+1] = i+1-k$.

The latter case can be proved as follows. Let j be $i+1-\text{compl}(S)[i+1]$ $(= i+1-\text{compl}(S')[i+1])$. According to the definition of the compl encoding, $S[j] = \mathit{complement}(S[i+1])$ and $S'[j] = \mathit{complement}(S'[i+1])$. Let k be the largest integer such that $S[k] = \mathit{complement}(S[j])$ and $1 \leq k \leq i$. If it does not exist, let $k = i+1$. As $S[1..i]$ and $S'[1..i]$ are an s-match, the k value is also the largest integer such that $S'[k] = \mathit{complement}(S'[j])$ if $k \leq i$. It means that $\text{prev}(S)[i+1] = \text{prev}(S')[i+1] = i+1-k$. □

It means that, when we check s-matches of strings, we do not have to see the other encoding if one of the encodings encodes a character to a non-zero number. To check s-matching easier based on this lemma, we define a new encoding called s-encoding as follows.

Definition 4 *For a given string S, compute prev(S) and compl(S). If prev(S)[i] = 0, replace it with −compl(S)[i], which is a nonpositive value. We call this new encoded string in $(\Sigma \cup I)^*$ (I: integer) as a structural encoding of S, or an s-encoding for short. Let sencode(S) denote the s-encoding of S.*

Notice that, if we have the $\text{prev}(S)$ and $\text{compl}(S)$ encoded strings, we can obtain the value of $\text{sencode}(S[i..n])[j]$ for any i and j in constant time as follows. If $0 < \text{prev}(S)[i+j-1] < j$, $\text{sencode}(S[i..n])[j] = \text{prev}(S)[i+j-1]$. Otherwise, if $0 < \text{compl}(S)[i+j-1] < j$, $\text{sencode}(S[i..n])[j] = \text{compl}(S)[i+j-1]$. Otherwise, $\text{sencode}(S[i..n])[j] = 0$. We call $\text{sencode}(S[i..n)$ $(1 \leq i \leq n = |S|)$ an s-suffix of $\text{sencode}(S)$ for any s-string S. But $\text{sencode}(S[i..n])$ is not always a suffix of $\text{sencode}(S)$. It is because $\text{sencode}(S[i..n])[j-i+1] \neq \text{sencode}(S[1..n])[j]$. if $\text{sencode}(S[1..n])[j] > j-i$.

Now we can check s-matching much easier based on the following lemma.

Lemma 3 *The s-strings S and S' are an s-match if and only if sencode(S) = sencode(S').*

Proof: According to Lemma 1, s-strings S and S' are an s-match if and only if $\text{prev}(S) = \text{prev}(S')$ and $\text{compl}(S) = \text{compl}(S')$. Therefor it is trivial that $\text{sencode}(S) = \text{sencode}(S')$ if the s-strings S and S' are an s-match. Moreover, if $\text{sencode}(S) = \text{sencode}(S')$, the two s-strings satisfy $\text{prev}(S) = \text{prev}(S')$ and $\text{compl}(S) = \text{compl}(S')$ according to Lemma 2. Hence we conclude that the s-strings S and S' are an s-match if and only if $\text{sencode}(S) = \text{sencode}(S')$. □

The structural suffix tree of string S, or the s-suffix tree of S for short, is the compacted trie of the s-encoded strings of all the suffixes of $S^+ = S\$$, where \$ is a character that is in neither Σ nor Π. We here consider \$ as an ordinary alphabet character, not as a parameter. As in the case of an ordinary suffix trees, the tree has $n+1$ leaves, and each internal node has more than one child. Each edge is labeled with a non-empty substring of s-encoded suffixes in $(\Sigma \cup I)^*$. We call these labels s-labels to distinguish them with the labels of the ordinary suffix trees. Each node has an s-label of the concatenated string of edge s-labels on the path from the root to the node, and each leaf has an s-label of an encoding of different suffix of S^+. Each edge s-label is represented by the first and the last indices of the corresponding substring of S^+. Figure 2 shows an example of the s-suffix tree, in which the s-labels are described with the prev and compl encodings.

In the s-suffix tree, we call the locus of sencode(α) just the locus of $\alpha \in (\Sigma \cup \Pi)^*$. We refer to α as the original substring of the s-label of the locus. There can be several original substrings for one locus. As in the case of the ordinary suffix trees, let σ_u denote the s-label of u, and $node(\alpha')$ denote the node at the locus of s-label α' if it exists. Let $node(\alpha)$ ($\alpha \in (\Sigma \cup \Pi)^*$) also denote the node at the locus of sencode(α). As in the case of p-suffix trees, the nodes in the s-suffix trees does not always have explicit suffix links to other nodes. For s-suffix trees, we define suffix links as follows, which is similar to the definition of the suffix links for the p-suffix trees by Baker.

Definition 5 *Let u be a node in an s-suffix tree. If u is the root, the suffix link of u is the root itself. Otherwise, let α' be the s-label of u. The suffix link is a pointer to a node or an edge at the locus of an s-suffix of α' whose length is $|\alpha'|-1$. Let $sl(u)$ denote the suffix link of u as in the case of suffix trees. $sl(u)$ is said to be explicit if it is a node, and otherwise it is said to be implicit.*

We say an s-string P 's-appears' in an s-string T if the s-encoding of P is equivalent to the s-encoding of some substring in T. Using the s-suffix tree of T, we can check whether or not a given structural pattern of size m s-appears in T in $O(m \log(|\Sigma| + |\Pi|))$ time, as the number of children of a node is bounded by $|\Sigma| + |\Pi|$. Moreover, we can use this data structure for finding a set of substrings that s-match with each other. We call such substrings s-repeats. Consider an s-repeat α of length l that s-appears r times ($r > 1$). If any s-string that is constructed by extending α to the right, such as αc ($c \in (\Sigma \cup \Pi)$), s-appears less than r times, we call α a "maximal structural pattern." A maximal structural pattern corresponds to the s-label of an internal node in the s-suffix tree, and r equals to the number of leaves under the corresponding node. Thus we can list s-repeats whose lengths are larger than l and that s-appear more than r times, just by traversing the corresponding nodes and outputting their s-labels, which can be done in $O(n + T_{output})$ time, where T_{output} is the output size.

3.2 Basic Algorithm for Constructing s-Suffix Trees

In this section, we describe the basic algorithm for constructing the s-suffix tree based on Ukkonen's algorithm. Note that any algorithm for constructing the s-suffix trees for s-strings can be used also for constructing the p-suffix trees for p-strings, as the s-string is a generalization of the p-string.

The implicit s-suffix tree of S is the compacted trie of all the s-encoded suffixes of S, and an s-label for an edge that ends at a leaf is represented by only the first index of the s-label in it. Let T_i denote the implicit s-suffix tree of $S^+[1..i]$ for an integer i ($0 < i \leq |S|+1$). Like Ukkonen's algorithm, our basic algorithm consists of $n+1$ phases, and in the ith phase, we construct an implicit s-suffix tree T_i from T_{i-1}. As in Ukkonen's algorithm, we construct a new node $u = node(S[j..i])$ for all $1 \leq j \leq i$ in increasing order, if there is no locus for $S[j..i]$ in the tree in the ith phase.

We call this procedure for single j the jth extension as in the description of Ukkonen's algorithm. The ith phase is described in Algorithm 2 below. Furthermore, the overall algorithm is given in Algorithm 3. You will find that the algorithm is very similar to the Ukkonen's algorithm. The difference lies in the existence of the implicit suffix links. In Algorithm 2, we begin the procedure from the j_ith extension, and we let *start_locus* be some ancestor locus (a node or an edge) of the locus of $S[j_i..i]$.

Algorithm 2 (Algorithm for the ith Phase: *sstree_phase*$(i, j_i, start\_locus)$**)** *Let* $j = j_i$, *and* l *be start_locus, and execute the following extension procedure.*

1. *Find the locus (l') of $S^+[j..i-1]$, which is guaranteed to exist, by tracing the tree from locus l, which we call scanning. During the scanning, we do not have to consult any s-labels except for the first characters of the traced edges. We call the traced nodes and edges 'scanned nodes' and 'scanned edges' respectively.*
2. *If we have created a node v_{j-1} in Step 4 of the previous iteration, set l' to $sl(v_{j-1})$ as its suffix link. Note that l' is not always a node.*
3. *If there is already the locus of $S^+[j..i]$ under the locus of $S^+[j..i-1]$, set j to next_j and the locus of $S^+[j..i]$ to next_locus, and this algorithm (the ith phase) is finished.*
4. *If there exists no node at the locus of $S^+[j..i-1]$, create it. Let this new internal node be v_j. If there are implicit suffix links that point to the edge split by this node insertion, correct them.*
5. *Under the node at the locus of $S^+[j..i-1]$, create a new leaf for the locus of $S^+[j..i]$. The s-label of this newly added edge is represented by only the first index, that is i.*
6. *Let w be the parent node of the locus of $S^+[j..i-1]$. Set $j+1$ to j and $sl(w)$ to l. Goto Step 1.*

Algorithm 3 (Basic s-Suffix Tree Construction Algorithm) *Let T be a tree with only one node at the root. Let $i = j_1 = 1$, and start_locus be the root. For all i from 1 to $n+1$ in increasing order, do the following.*

1. *Execute sstree_phase$(i, j_i, start_locus)$.*
2. *Let j_{i+1} be next_j and start_locus be next_locus, both of which are set in Step 3 of Algorithm 2.*

In the following, we analyze the time complexity of this algorithm. In the algorithm, there are two problems caused by the implicit suffix links. One is the analysis of the computation time for keeping the implicit suffix links correct in Step 4 of Algorithm 2. The other is the analysis of the number of scanned nodes in Step 1 of Algorithm 2. First, we deal with the former problem, and after that we discuss the latter problem.

Related to the implicit suffix links, we give the following lemma.

Lemma 4 *Let u be a node with an implicit suffix link and $d = |\sigma_u|$. Then the first character of the s-label of any of the outgoing edges from u must be one of d, 0, and $-d$. Furthermore, if it is d, its corresponding compl value must be 0.*

Proof: Let e_1 and e_2 be two of the outgoing edges from u. Let z_1 and z_2 be the loci on e_1 and e_2 whose distances from u are 1, and corresponding s-labels be αc_1 and αc_2, respectively. (c_i is a s-encoded character and α is a s-encoded substring.) Let $\alpha' c'_1$ and $\alpha' c'_2$ be the s-suffixes of αc_1 and αc_2 whose lengths are just 1 smaller, and let z'_1 and z'_2 be the loci of $\alpha' c'_1$ and $\alpha' c'_2$. If u has an implicit suffix link, the two loci z'_1 and z'_2 must be the same locus, which means that $c'_1 = c'_2 = 0$, though $c_1 \neq c_2$ as z_1 and z_2 are different loci. It means that c_i must be one of d, 0, and $-d$. If $c_i = d$ and corresponding compl value is not 0, c'_i becomes non-0 value. Thus if $c_i = d$, its corresponding compl value is 0. □

We use the term 'zero-node' for a node with more than one outgoing edge that has an s-label starting with either of d, 0, or $-d$, (which we call a 'zero-edge') where d is the s-label length of the node, regardless of whether its suffix link is implicit or not. We call a zero-edge a "positive zero-edge", a "normal zero-edge" or a "negative zero-edge", if the the first s-encoded character of its s-labels is d, 0 or $-d$, respectively.

We next give the following two lemmas related to zero-nodes and zero-edges.

Lemma 5 *A positive zero-edge cannot be an ancestor of another positive zero-edge. Similarly, a negative zero-edge cannot be an ancestor of another negative zero-edge. There are at most $|\Pi|$ normal zero-edges on a path from the root to a leaf.*

Proof: The first character (parameter) of the original substring of the s-label of a positive zero-edge e is same as the first character of the original substring of the s-label of the end node v of e, according to the definition. Even if a descendent edge of e has an s-label such that the first character of the original substring of the s-label of e is same as the first character of the original substring of the s-label of v, it is also same as the first character of the original substring of the s-label of e and it cannot be a positive zero-edge.

Similarly, the first character of the original substring of the s-label of a negative zero-edge e' is the complement of the first character of the original substring of the s-label of the end node v' of e', according to the definition. Even if the original substring of the s-label of a descendent edge of e is the complement of the first character of the original substring of the s-label of v, it is the character same as the first character of the original substring of the s-label of e and it cannot be even a negative zero-edge, because its prev-encoded character is not 0.

The third proposition is more trivial. There are at most $|\Pi|$ 0s in any s-encoded string according to its definition. Consequently there are at most $|\Pi|$ normal zero-edges on a path from the root to a leaf. □

Lemma 6 *On a path from the root to a leaf, there are at most $|\Pi| + 1$ zero-nodes.*

Proof: There are at most $|\Pi|$ 0s in any prev-encoded string. Thus there are at most $|\Pi|$ normal or negative zero-edges on a path from the root to a leaf. There are only 1 positive zero-edge at most on the path according to Lemma 5. Therefore, we conclude that there are at most $|\Pi| + 1$ zero-nodes on any path from the root to any leaf. □

We call a set of nodes a zero-chain if the nodes form a subpath on a path from the root to a leaf in the tree and all the edges between them are zero-edges of the same kind (*i.e.*, if one edge is a normal zero-edge, then the others are also normal zero-edges, for example). Then we obtain the following theorem related to the implicit suffix links:

Theorem 1 *For any edge e, the set of nodes (including nodes that are not inserted to the suffix tree yet) having implicit suffix links to e forms at most $2|\Pi| + 1$ zero-chains. Furthermore, the length of each zero-chain is at most $|\Pi|$.*

Proof: Let v and v' be two nodes with implicit suffix links to the same edge. If v is an ancestor of v' and there is a node u between v and v', it is obvious that $sl(u)$ is also between $sl(v)$ and $sl(v')$.

If neither of these two nodes is an ancestor of the other, let w be the lowest common ancestor of v and v' in the s-suffix tree. Note that it does not occur to the nodes in any p-suffix trees. w is not the root, because both of the first characters of the s-labels of v and v' must be 0. Let $\text{suffix}_i(S)$ denote $S[i..|S|]$. Since one of the s-encoded strings of $\text{suffix}_2(\sigma_v)$ and $\text{suffix}_2(\sigma_{v'})$ must be a prefix of the other, both of the outgoing edges of w to ancestors of v and v' must be zero-edges. Thus w must be a zero-node.

Lemma 5 implies that, under the negative or positive zero-edge out of w, there is only one zero-chain formed by the set of nodes having implicit suffix links to e. Furthermore, there are at most $|\Pi|$ normal zero-edges on a path to a leaf from the root, according to Lemma 5. One zero-node can have three outgoing zero-edges at most. Two of the edges can have only one such zero-chain under each edge, and the other node have at most $|\Pi| - 1$ zero-node under the edge. This means that there are at most $2|\Pi| + 1$ such zero-chains. Also according to Lemma 5, it is obvious that the lengths of the zero-chains are at most $|\Pi|$. □

Figure 3 shows a picture of these zero-chains. According to this theorem, there are $O(|\Pi|^2)$ implicit suffix links to one edge, and it takes $O(|\Pi|^2)$ time to update all the corresponding implicit suffix links one by one when we split an edge in Step 4 of Algorithm 2. In the case of the p-suffix tree (*i.e.*, in case there are no complementary character pairs), such nodes form only one zero-chain, and the number of implicit suffix links to one edge is at most $|\Pi|$. Hence the update time is $O(|\Pi|)$ for p-suffix trees. We will improve these time complexities in the next section.

We next discuss the number of nodes scanned in the algorithm.

Theorem 2 *The number of scanned edges that are not normal zero-edges is at most n.*

Proof: In constructing the s-suffix tree of a string S, consider that an edge (u, v) $(u = parent(v))$ that is not a normal zero-edge is scanned when we search for the locus of $S^+[j..i]$ in the jth extension of the $(i+1)$th phase. Let $u = node(S^+[j..k])$.

Suppose that there is a node w at the locus of $S^+[j'..k]$ in the j'th extension of the i'th phase such that $k < i' \le i$ and $j' < j$. Note that we do not perform the j''th $(j'' \ge j)$ extension in the i'th phase $(i' < i)$ of the algorithm. As (u, v) is not a normal zero-edge, $\text{sencode}(S^+[j'+1..k+1])[k - j' + 1]$ cannot be 0 and therefore w has an explicit suffix link to some node. But it means that u is pointed by the suffix link of a scanned node in the previous extension, and u is not a scanned node. Hence there cannot be any node like w if (u, v) is a scanned edge, and the value k is different for any scanned node u. Accordingly, we conclude that the total number of such edges is at most n. $\square$

According to Lemma 6, the number of normal zero-edges that are scanned in a single phase is at most $|\Pi|$. Thus the total number of nodes that have implicit suffix links and are scanned in the algorithm is at most $n|\Pi|$. We assume that the outgoing normal zero-edge from a node can be accessed in $O(1)$ time. Thus the total scanning time will be $O(n(|\Pi| + \log|\Sigma|))$.

Thus we conclude that the total computing time of our basic algorithm is $O(n(|\Pi|^2 + \log|\Sigma|))$. For p-suffix trees, this algorithm achieves $O(n(|\Pi| + \log|\Sigma|))$ time. If $|\Pi|$ and $|\Sigma|$ are constant, both of them are $O(n)$. In fact, in the problem of RNA/DNA structural matching ($|\Sigma| = 0$ and $|\Pi| = 4$), this basic algorithm is efficient enough as we will show in the experiment section. In the following sections, we consider how to improve the computation time to $O(n(\log|\Pi| + \log|\Sigma|))$.

3.3 Faster Updating Algorithm for Implicit Suffix Links

In this section, we consider how to reduce the updating time of the implicit suffix links to $O(\log|\Pi|)$, which was $O(|\Pi|^2)$ in the basic algorithm. By doing so, we can achieve $O(n(|\Pi| + \log|\Sigma|))$ time.

For maintaining the implicit suffix links, we use a balanced tree structure called a concatenable queue [1], or a c-queue for short, which represents a list of items sorted by the values that are given to the items. It takes $O(\log m)$ time to insert a new item into a c-queue of size m. For any value x, it takes $O(\log m)$ time to split a c-queue Q of size m into two c-queues Q_1 and Q_2 so that the items with values with larger than x are all in Q_1 and the others are all in Q_2.

For each edge that is pointed by any implicit suffix links, we construct a c-queue which represents a list of nodes that have an implicit suffix link to the edge sorted by the s-label lengths of the nodes. Moreover, we use a pointer to the c-queue instead of the implicit suffix link to the edge to maintain the implicit suffix links. We call it a q-pointer. The c-queue has a pointer to the edge instead to enable tracing the implicit suffix links. When an edge is split, a corresponding c-queue Q is also split to two c-queues. Let the larger one of the two be Q_1 and the other be Q_2. We let Q_1 be the successor of Q, *i.e.*, we let the memory address of Q_1 be the same as Q. By doing so, we do not have to change the q-pointers of the nodes listed in Q_1. For the nodes listed in Q_2 whose size is smaller than the half of the size of Q, we have to change the q-pointers to point at Q_2.

As the number of insertions and splits are $O(n)$ and the sizes of the c-queues are $O(|\Pi|^2)$, the computation time for maintaining the c-queues is $O(n \log|\Pi|)$. Add to it, we need to analyze the computation time for updating the q-pointers, for which we give the following lemma.

Lemma 7 *Assume that total m nodes are inserted to a c-queue Q or its descendant split c-queues. The total number of updates of the q-pointers of the m nodes is at most $m \log m$, in which the base of* $\log$ *is* 2.

Proof: Assume that all the insertion operations to the c-queues are done before any of the split operations. Notice that the number of the updates in this case is larger than or equal to the actual number of updates. If $m = 1$, the number of updates is 0 and the proposition is true. Suppose that the number of the updates is at most $x \log x$ for any c-queue of size x such that $x < m$, and that a c-queue Q of size m is split into two c-queues of sizes x and $x+y$ ($x > 0$, $y \geq 0$, $m = 2x+y$). Then the number of the updates for Q is at most $x \log x + (x+y)\log(x+y) + x = x\log(2x) + (x+y)\log(x+y) < (2x+y)\log(2x+y) = m \log m$. It holds for any split of Q. Thus the number of updates for Q is at most $m \log m$. $\square$

Therefore, the total number of the updates of q-pointers is also at most $O(n \log |\Pi|)$ as the largest size of m is $O(|\Pi|^2)$ according to Theorem 1. Hence the total time for maintaining the implicit suffix links with the c-queues is $O(n \log |\Pi|)$ time, and consequently we can construct the s-suffix tree in $O(n(|\Pi| + \log |\Sigma|))$ time.

3.4 Faster Scanning Algorithm

In this section, we finally improve the algorithm to $O(n(\log |\Pi| + \log |\Sigma|))$, which is same as the Kosaraju's best-known algorithm [14] if it is used for p-suffix trees. A zero-chain C is called normal if the edges in the subpath formed by C are normal zero-edges. According to Theorem 2, we only have to care about normal zero-chains to improve the time complexity. A zero-chain C is called maximal if C is a zero-chain and $C \cup \{x\}$ is not a zero-chain for any node x. We maintain all the normal maximal zero-chains in the tree using c-queues. Let $ll(e)$ be the s-label length of the node at the end of the edge e. For each normal zero-chain, we maintain a c-queue using $ll(e)$ as the key of edge e. This structure can be split in $O(\log |\Pi|)$ time when a new node is inserted among the chain, and a new zero-edge can be inserted into a chain also in $O(\log |\Pi|)$ time, as the size of zero-chains is $O(|\Pi|)$ according to Theorem 1. Furthermore, there are $O(n)$ normal zero-chains at most. Hence total time for maintaining them is $O(n \log |\Pi|)$.

Consider the situation that we have just constructed a new node by splitting an edge (u, v) ($u = parent(v)$) by inserting a node t. In the next extension, we will scan a path from $sl(u)$ to $sl(v)$ to find the locus of $sl(t)$. In Ukkonen's algorithm, we do not have to maintain suffix links of leaves, but we maintain these suffix links of leaves to know $sl(v)$ even if v is a leaf for this algorithm. The suffix links of leaves can be very easily maintained as the leaves are constructed one by one from those with longer s-labels. Now let e be a zero-edge encountered in scanning from $sl(u)$ to $sl(v)$. Let C be the set of zero-edges in the zero-chain which includes e. We can find the zero-edge e' in C nearest to a leaf such that $ll(e')$ is not larger than the s-label length of $sl(t)$ in $O(\log |\Pi|)$ time, using the above c-queue. According to [8], we can compute the lowest common ancestor (LCA) of two nodes of a suffix tree in a constant time even while we are constructing the tree. Thus we can find the LCA w of the edge e' and the node $sl(v)$ in a constant time. Then what we have to do is to start scanning from w, because w must be $sl(t)$ itself or an ancestor of $sl(t)$. Figure 4 shows the picture of this technique. For the zero-chains, we give the following theorem.

Theorem 3 *The number of encountered maximal normal zero-chains in scanning is at most* $n+1$.

Proof: There must be at least one edge that is not a normal zero-edge between two maximal normal zero-chains encountered in scanning, due to the definition of maximal normal zero-chains. Moreover, the number of scanned edges that are not normal zero-edges is at most n, according to Theorem 2. Thus we can conclude that the number of encountered normal zero-chains in scanning is at most $n+1$. $\square$

Thus the total time for scanning zero-edges with this technique is $O(n \log |\Pi|)$, and we finally achieved an $O(n(\log |\Pi| + \log |\Sigma|))$ time algorithm.

4 Computational Experiments

In this section, we describe experiments on the s-suffix trees of RNA and DNA sequences, where $\Sigma = \phi$, $\Pi = \{\mathtt{A},\mathtt{U},\mathtt{G},\mathtt{C}\}$, `A` is the complement of `U` and `G` is the complement of `C`. (In DNA sequences, `T` is present instead of `U`.)

We conducted experiments on three HIV (human immunodeficiency virus) RNA complete sequences: (A) a sequence of length 9719 (accession number: K03455), (B) a sequence of length 9748 (accession number: X01762) and (C) a sequence of length 8981 (accession number: AF067156). We also use four DNA sequences of E. coli, each of which has the same length, 1 Mbp $= 1,000,000$ bp. The length of the full genome sequence of E. coli is about 4.64 Mbp, and these four sequences are the following regions of the sequence: (D) 1 bp–$1,000,000$ bp, (E) $1,000,001$ bp–$2,000,000$ bp (F) $2,000,001$ bp–$3,000,000$ bp, and (G) $3,000,001$ bp–$4,000,000$ bp.

First, we compare the construction time and the size of the s-suffix tree with those of the normal suffix tree of the same sequences. Table 1 shows the construction time (in seconds) using a PentiumIII CPU running at 2.2 GHz, and the numbers of nodes in the suffix trees and the s-suffix trees of the seven sequences. According to the table, the construction time of the s-suffix trees is only about 1.5 times larger than the time for the ordinary suffix trees, though we use the naive $O(n(|\Pi|^2 + \log |\Sigma|))$ algorithm.

The s-suffix tree uses more memory space than the ordinary suffix trees for the following two kinds of data. One is the list of implicit suffix links stored in each edge. Note that it is needed only during the construction of the tree. For a string of size n, it requires memory of $3n$ words at most, as there are at most n implicit suffix links and $2n$ edges in the tree. Note that it should be much smaller in ordinary. It becomes several times larger if we maintain the implicit suffix links with c-queues, but we do not use c-queues in the experiments. The other is the prev and compl encoded string of the original string, which requires $2n$ words memory. Note that the prev and compl values are very small integers and can be stored in 1 byte in most cases in the case of RNA strings. As the suffix tree implementation requires $5n$ to $9n$ words (depending on implementations), our algorithm requires less than twice memory as the suffix tree construction algorithms, if the number of the nodes in the two trees are similar. In fact, both the number of nodes in the suffix tree and that of the s-suffix tree are very similar and they are about 1.6 to 1.7 times the length of the sequence for any sequences, according to the experiments in Table 1. More precisely, the numbers of nodes in the s-suffix trees are slightly smaller than those of the normal suffix trees in all cases.

We also give the experimental results of an experiment to find maximal structural patterns which are longer than l and repeated more than r times for some given l and r. Table 2 shows two examples of maximal structural patterns found in E. coli sequence (D): (1) is a set of patterns of length 15 that s-appears four times, and (2) is a set of patterns of length 19 that s-appears 2 times in the sequence. Every sequence is different from the others, but these sequences s-match with each other. Table 3 shows the number of maximal patterns whose lengths (l's) are larger than some given length. In the table, a “normal pattern” means an ordinary string pattern that can be found with an ordinary suffix tree. Notice that the structural patterns include the normal patterns. According to the table, the proportion of normal patterns increases with the lengths of the patterns, which means that there are much more ordinary long patterns than the structural long patterns. But we succeeded in finding many structural patterns, and we think they may include some interesting patterns.

5 Concluding Remarks

We have proposed a new data structure called the structural suffix tree, or s-suffix tree for short. We also proposed an on-line $O(n(\log|\Sigma|+\log|\Pi|))$ algorithm for constructing it, where Σ is an alphabet of fixed symbols and Π is an alphabet of parameters. This data structure enables an efficient search for frequent patterns of structures of RNA sequences or single-stranded DNA sequences. It also enables a common structure pattern to be efficiently found in more than one sequence. We also showed the practicality of our data structure and our algorithm by reporting computational experiments for finding structural patterns from RNA sequences of HIV and DNA sequences of E. coli.

Several tasks remain for the future. Two sequences can have the same structure even if they do not have the same s-encoded string patterns. Furthermore, it is difficult to apply our algorithm to the problem of proteins, where the combinations are far more complicated. Thus we should strive to create more general data structures and algorithms for structural pattern matching of biological sequences. Ordinary suffix trees for strings of an integer alphabet $\{1,\ldots,n\}$ can be constructed in linear time regardless of the alphabet size [9]. It is an open problem whether or not such a linear time algorithm exists for constructing s-suffix trees or p-suffix trees.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Design and Analysis of Algorithms,* Addison Wesley Publishing Co., Reading, Mass., 1974.

[2] V. R. Akmaev, S. T. Kelley, and G. D. Stormo, A Phylogenetic Approach to RNA Structure Prediction, *Proc. 7th International Conference on Intelligent Systems for Molecular Biology*, 1999, pp. 10-17.

[3] B. S. Baker, A Program for Identifying Duplicated Code, *Computing Science and Statistics*, Interface Foundation of North America, 1992, pp. 49-57.

[4] B. S. Baker, Parameterized Pattern Matching by Boyer-Moore-type Algorithms, *Proc. 6th Annual ACM-SIAM Symp. Discrete Algorithms,* 1995, pp. 541-550.

[5] B. S. Baker, Parameterized Pattern Matching: Algorithms and Applications, *J. Comp. Syst. Sci.,* **52(1)**(1996), 28-42.

[6] B. S. Baker, Parameterized Duplication in Strings: Algorithms and Application to Software Maintenance, *SIAM J. Comput.,* **26(5)**(1997), 1343-1362.

[7] B. S. Baker, Parameterized Diff, *Proc. 10th ACM-SIAM Symp. Discrete Algorithms,* 1999, pp. 854-855.

[8] R. Cole and R. Hariharan, Dynamic LCA Queries on Trees, *Proc. 10th ACM-SIAM Symp. Discrete Algorithms,* 1999, pp. 235-244.

[9] M. Farach, Optimal Suffix Tree Construction with Large Alphabets, *Proc. 38th IEEE Symp. Foundations of Computer Science,* 1997, pp. 137-143.

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

[11] D. Harel and R. R. Tarjan, Fast Algorithms for Finding Nearest Common Ancestors, *SIAM J. Computing,* **13**(1984), 338-355.

[12] H. P. Lenhof, K. Reinert, and M. Vingron, A Polyhedral Approach to RNA Sequence Structure Alignment, *Proc. 2nd Annual International Conference on Computational Molecular Biology (RECOMB '98)*, 1998, pp. 153-162.

[13] R. B. Lyngso, M. Zuker, and C. N. S. Pedersen, Internal Loops in RNA Secondary Structure Prediction, *Proc. 3nd Annual International Conference on Computational Molecular Biology (RECOMB '99)*, 1999, pp. 260-267.

[14] S. R. Kosaraju, Faster Algorithms for the Construction of Parameterized Suffix Trees, *Proc. 36th IEEE Symp. Foundations of Computer Science,* 1995, pp. 631-637.

[15] E. M. McCreight, A Space-Economical Suffix Tree Construction Algorithm, *J. ACM,* **23**(1976), pp. 262-272.

[16] J. Setubal and J. Meidanis, *Introduction to Computational Molecular Biology*, PWS Pub. Co., Boston, 1997.

[17] D. H. Turner, N. Sugimoto, and S. M. Freier, RNA Structure Prediction, *Ann. Rev. Biophys. Chem.*, **17**(1988), pp. 167-192.

[18] E. Ukkonen, On-Line Construction of Suffix-Trees, *Algorithmica,* **14**(1995), pp. 249-60.

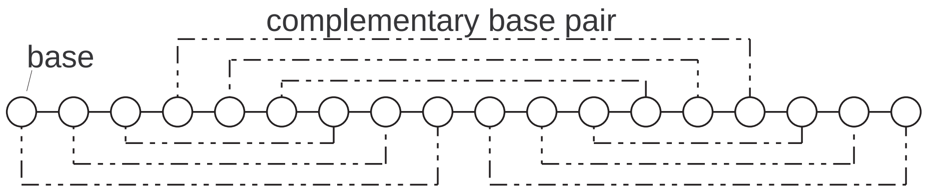
[19] Z. Wang and K. Zhang, Finding Common RNA Secondary Structures from RNA Sequences, *Proc. 4th Symposium on Combinatorial Pattern Matching,* Springer-Verlag LNCS 1645, 1999, pp. 258-269.

[20] M. S. Waterman, *Introduction to Computational Biology*, Capman & Hall, London, 1995.

[21] P. Weiner, Linear Pattern Matching Algorithms, *Proc. 14th Symposium on Switching and Automata Theory,* 1973, pp. 1-11.

Sequence 1: `AUAUCGUAUGGCCGAGCC`
Sequence 2: `CGCGUAGCGAAUUACAUU`

(1) Example sequences



(2) Candidate structure

Figure 1: Examples of sequences that have high possibility to have a same structure.

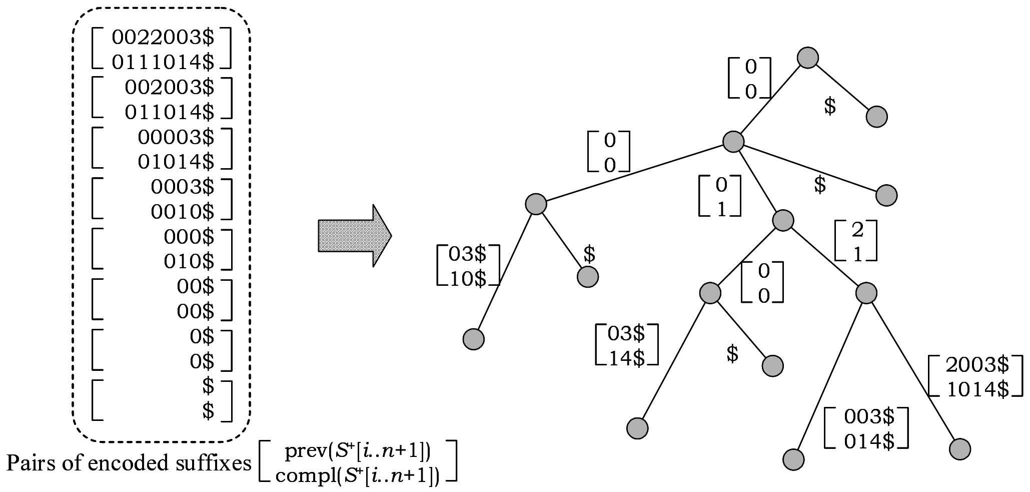


Figure 2: The structrual suffix tree of an RNA string $S =$ "`AUAUCGU`".

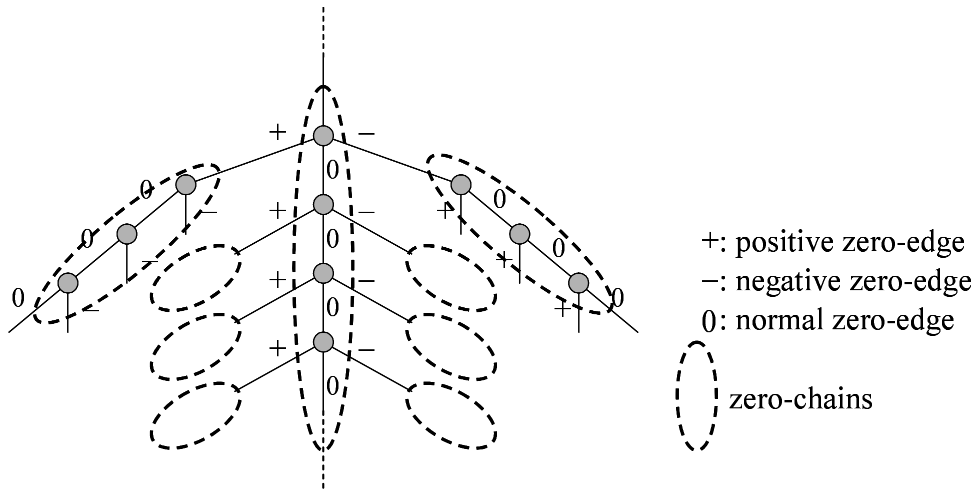


Figure 3: An example of zero-chains of nodes having implicit suffix links to a same edge.

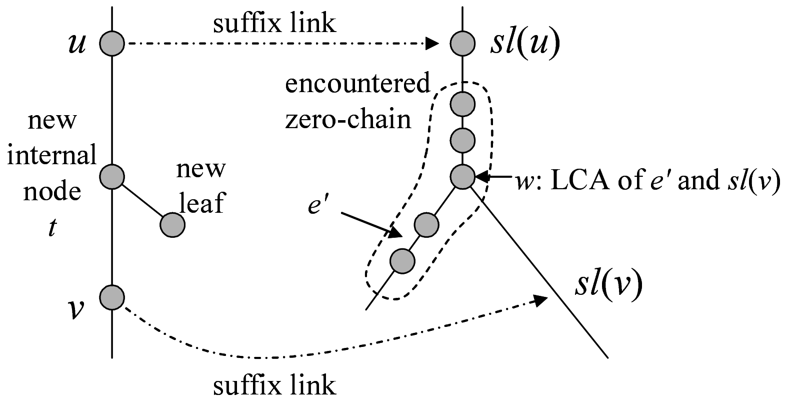


Figure 4: Faster scanning by using the dynamic LCA technique.

Table 1: Construction time and the number of nodes in suffix trees and s-suffix trees.

Sequence		(A)	(B)	(C)	(D)	(E)	(F)	(G)
Length		9719	9748	8981	1000000	1000000	1000000	1000000
Suffix tree	Time (sec)	0.02	0.02	0.02	2.64	2.72	2.75	2.67
	#nodes	16135	16217	14710	1640492	1635995	1638043	1638008
s-Suffix tree	Time (sec)	0.02	0.02	0.02	3.89	3.91	3.86	3.94
	#nodes	16033	16132	14666	1631525	1628821	1630104	1628923

Table 2: Examples of maximal structural patterns.

(1)

Position	Sequence
646095	`CCCGCTTCGGCTTCA`
703617	`GGGCGTTGCCGTTGA`
779110	`TTTATGGTAATGGTC`
888469	`TTTATCCTAATCCTG`

(2)

Position	Sequence
371484	`ACTGCGCCATGAAGATGAC`
884639	`GACTATAAGCTGGTGCTGA`

Table 3: Number of structural/normal patterns.

(1) HIV RNA sequences

l	Pattern	(A)	(B)	(C)
≥ 5	Structural	5329	5061	4887
	Normal	1381	1147	1000
≥ 10	Structural	670	451	282
	Normal	479	363	126
≥ 15	Structural	336	123	4
	Normal	336	123	3

(2) E. coli sequences

l	Pattern	(D)	(E)	(F)	(G)
≥ 10	Structural	495371	499205	498728	497701
	Normal	90968	85899	88681	90298
≥ 15	Structural	4723	4140	4466	4529
	Normal	2402	1728	2095	2147
≥ 20	Structural	330	106	192	192
	Normal	330	103	192	190