𝕋ℂ𝕊-𝕋ℝ-𝔸-08-35

𝕋ℂ𝕊 Technical Report

Algorithms for Finding a Minimum Repetition Structure of a String or a Tree

by

Atsuyoshi Nakamura
Tomoya Saito
Mineichi Kudo

Division of Computer Science

Report Series A

February 8, 2008

Hokkaido University
Graduate School of
Information Science and Technology

Email: atsu@ist.hokudai.ac.jp

Phone: +81-011-706-6806
Fax: +81-011-706-7832

Algorithms for Finding a Minimum Repetition Structure of a String or a Tree

February 8, 2008

Abstract

A *repetition representation string*, RRS for short, is a representation for a set of disjoint or nested tandem arrays in a string. The length of an RRS depends on a set of tandem arrays it represents. We show an $O(n^3)$ dynamic programming algorithm for finding a shortest RRS representing a given string with length n. The algorithm can be extended to an algorithm of the same time complexity for finding a minimum *repetition representation tree*, RRT for short, representing a given labeled ordered tree with n nodes. Furthermore, corresponding problems for *width-k* subclasses of RRSs and RRTs are shown to be solved in time $O(k^2n)$ when functions of deciding the size of a RRS and a RRT satisfy a certain condition.

1 Introduction

Contiguous repeats appeared in a sequence have special meanings in many cases. For example, contiguously repeated melody forms an impressive part of a music piece, contiguous repeats in a DNA sequence have been shown to cause human disease [1], and a number of data records embedded in a Web page can be recognized by detecting contiguously repeated HTML tag structure [2].

A contiguous repeat of a substring embedded in a string is called a *tandem array*, and it is also called a *tandem repeat* when the number of repetition is two. The problem of finding tandem arrays and repeats have been studied for more than two decades in the fields of computer science, mathematics and biology [3]. Efficient algorithms for finding *all* or *primitive* tandem repeats have been proposed [3, 4].

A tandem array is representable by $(r)^h$, which means h times repetition of a string r. By repeatedly substituting this representation for a contiguous substring corresponding to a tandem array, a number of tandem arrays embedded in a string s can be represented in a special string r that is composed of not only alphabets appeared in s but also parentheses and superscript numbers. We call such a representation r a *repetition representation string*, *RRS* for short. An RRS can not represent all tandem arrays but can represent a set of disjoint or nested tandem arrays simultaneously. Note that there are many RRS representing a string.

In this paper, we show an $O(n^3)$ algorithm for finding a shortest RRS representing a given string with length n. Here, using an alphabet weight function w_Σ and a repetition weight function w_R, the length $l(r)$ of an RRS r is defined as the sum of its component lengths, which is $w_\Sigma(a)$ if the component is an alphabet a and is $l(r') + w_R(h)$ if the component is a tandem array $(r')^h$. Our algorithm is very simple one using a dynamic programming technique. When the *width* of tandem arrays $(r)^h$, the length of the string represented by r, represented in an RRS is restricted to at most k, and when $w_R(2), w_R(3), ...$ is an arithmetic progression sequence, an $O(k^2n)$ algorithm is shown.

Furthermore, our $O(n^3)$ and $O(k^2n)$ algorithms can be extended to algorithms of the same time complexity for finding a minimum *repetition representation tree*, RRT for short, representing a given labeled ordered tree with n nodes. Here, an RRT is a labeled ordered tree that may have special nodes called *repetition information nodes* labeled $h \geq 2$ which represent an h times repetition of a sequence of the subtrees rooted by its child nodes. Our algorithm uses divide-and-conquer strategy and recursively solves subproblems for 1-height smaller trees, subtrees rooted by child nodes of the root node. Using the solutions for the subproblems, the problem is reduced to the one for finding a shortest RRS.

2 Finding a Shortest Repetition Representation String

2.1 Problem Setting

Let Σ be a finite set of alphabets. A *string* is defined inductively as follows. Special symbol λ, called a *null string*, and all alphabets are strings. If s_1 and s_2 are strings, then s_1s_2, concatenation of s_1 and s_2, is also a string. Note that $\lambda s = s\lambda = s$ for all strings s. Here, we define more general notion called a *repetition representation string*, RRS for short, as follows. First, all strings are RRSs. If r_1 and r_2 are RRSs, then r_1r_2, concatenation of r_1 and r_2, is also an RRS. If r is an RRS, then $(r)^h$ ($h \geq 2$), *h times repetition of* r, is also an RRS that is another representation of concatenation $\underbrace{rr\cdots r}_{h\text{times}}$ of the same RRSs r. Note that parentheses '(', ')' and codes for numbers are special symbols and not contained in Σ. Though $(r)^h$ is generally called a *tandem array*, we use term "repetition" instead of tandem array in the rest of the paper. *Expansion* of $(r)^h$ is concatenation $\underbrace{rr\cdots r}_{h\text{times}}$ of the same RRSs r. Reversely, *reduction* of concatenation $\underbrace{rr\cdots r}_{h\text{times}}$ of the same RRSs r, is an RRS $(r)^h$. String s is said to be represented by RRS r if expansions of all repetitions in r change r to s.

Example 1 *String abaababaab is represented by RRS $(ab(a)^2b)^2$. Note that there are many RRSs representing abaababaab such as abaababaab itself and $(aba)^2b(a)^2b$.*

Next, we define the *length* of an RRS. The length of an RRS can be calculated using given two weight functions, *alphabet weight function* w_Σ and *repetition weight function* w_R. Function w_Σ is a real-valued function on Σ, and function w_R is a real-valued function on the set of natural numbers at least 2. Given w_Σ and w_R, the length $l(r)$ of an RRS r is recursively defined as follows:

$$\begin{aligned} l(\lambda) &= 0, \\ l(a) &= w_\Sigma(a) \text{ for } a \in \Sigma, \\ l(r_1 r_2) &= l(r_1) + l(r_2) \text{ for all RRSs } r_1 \text{ and } r_2, \text{ and} \\ l((r)^h) &= l(r) + w_R(h). \end{aligned}$$

Example 2 *Let $w_\Sigma(a) = 1$ for all alphabets $a \in \Sigma$, $w_R(h) = 1$ for all $h \geq 2$. Then, $l(abaababaab) = 10$, $l((ab(a)^2b)^2) = 6$ and $l((aba)^2b(a)^2b) = 8$.*

Problem we deal with in this section is the following one.

Problem 1 *Given a string s, an alphabet weight function w_Σ and a repetition weight function w_R, find a shortest RRS r that represents s.*

2.2 Algorithm

In this subsection, we describe an algorithm for finding a shortest RRS for an arbitrary string $s(= a_1 a_2 \cdots a_n)$ with length n, an arbitrary alphabet weight function w_Σ and an arbitrary repetition weight function w_R. Let $s[i,j]$ denote the substring $a_i a_{i+1} \cdots a_j$ of s, and let $r[i,j]$ denote a shortest RRS representing $s[i,j]$. Then, the following proposition holds.

Proposition 1 *For all $1 \leq i < j \leq n$, one of the following cases holds.*

Case 1 *$r[i,j] = (r[i, i+h-1])^d$ for some $h \geq 1$ and $d \geq 2$ with $dh = j-i+1$*

Case 2 *$r[i,j] = r[i, i+d]r[i+d+1, j]$ for some $0 \leq d < j-i$*

(Proof) If $r[i,j] = (r')^d$ for some RRS r', r' must be an RRS representing $s[i, i-1+h]$ where $h = (j-i+1)/d$. RRS r' must be a shortest RRS representing $s[i, i-1+h]$ because, if not, $l(r') > l(r[i, i-1+h])$ holds and it leads that $l(r[i,j]) = l(r') + w_R(d) > l(r[i, i-1+h]) + w_R(d) = l((r[i, i-1+h])^d)$, which contradicts the fact that $r[i,j]$ is the shortest RRS representing $s[i,j]$. Thus, Case 1 holds in this case.

If there do not exist r' and $d \geq 2$ such that $r[i,j] = (r')^d$, then $r[i,j] = r_1 r_2$ for some RRSs r_1 and r_2. RRSs r_1 and r_2 must be RRSs representing $s[i, i+d]$ and $s[i+d+1, j]$, respectively, for some $0 \leq d < j-i$. RRSs r_1 and r_2 must be shortest RRSs representing $s[i, i+d]$ and $s[i+d+1, j]$, respectively, because, if not,

$l(r_1) > l(r[i, i+d])$ or $l(r_2) > l(r[i+d+1, j])$ holds and it leads to a contradiction with the fact that $r[i, j]$ is a shortest RRS. Thus, Case 2 holds in this case. □

For given $1 \le i < j \le n$, assume that $r[i', j']$ is already known for all $1 \le i' < j' \le n$ with $j' - i' < j - i$. Using the above proposition, $r[i, j]$ is obtainable by searching a shortest RRS among $\lfloor (j - i + 1)/2 \rfloor$ possibilities of Case 1 and $j - i$ possibilities of Case 2. If $l(r[i', j'])$ for all $1 \le i' < j' \le n$ with $j' - i' < j - i$ is already calculated, the length for each possible RRS can be calculated in constant time. Thus, such search can be finished in $O(j - i + 1)$ time. This means that a shortest RRS $r[1, n]$ for a given string s is efficiently obtainable by dynamic programming using the fact that $r[i, i] = s[i, i]$ for all $1 \le i \le n$.

```
ShortestRRS % [Finding a Shortest RRS Representing String $a_1 a_2 \cdots a_n$]
Input:  $a_1 a_2 \cdots a_n$: string
        $w_\Sigma$: alphabet weight function
        $w_R$: repetition weight function
Output: $r$ : shortest RRS representing $a_1 a_2 \cdots a_n$
        $l_r$ : length of $r$
 1: $l[i,i] \leftarrow w_\Sigma(a_i)$ for $i = 1, 2, ..., n$
 2: $l[i,j] \leftarrow \infty$ for $i = 1, 2, ..., n$ and $j = 2, 3, ..., n$ with $i < j$
 3: for $h = 1$ to $n$ do
 4:   for $i = 1$ to $n - h + 1$ do
 5:     for $d = 0$ to $h - 2$ do
 6:       if $l[i, i+d] + l[i+d+1, i+h-1] < l[i, i+h-1]$ then
 7:         $l[i, i+h-1] \leftarrow l[i, i+d] + l[i+d+1, i+h-1]$
 8:         type$[i, i+h-1] \leftarrow (2, d)$
 9:       end if
10:     end for
11:     for $s = i + h$ to $n - h + 1$ by $h$ do
12:       if $a_s a_{s+1} \cdots a_{s+h-1} \ne a_i a_{i+1} \cdots a_{i+h-1}$ then break
13:       if $l[i, i+h-1] + w_R((s-i)/h + 1) < l[i, s+h-1]$ then
14:         $l[i, s+h-1] \leftarrow l[i, i+h-1] + w_R((s-i)/h + 1)$
15:         type$[i, s+h-1] \leftarrow (1, h)$
16:       end if
17:     end for
18:   end for
19: end for
20: $l_r \leftarrow l[1, n]$
21: $r \leftarrow$ ConstructShortestRRS($a_1 a_2 \cdots a_n$, type)
22: return $(r, l_r)$
```

Figure 1: Algorithm ShortestRRS

Algorithm ShortestRRS shown in Fig. 1 is the one that constructs a shortest RRS representing a given string using dynamic programming. In the algorithm, $l[i, j]$ $(1 \le i \le j \le n)$ is the length of a shortest RRS representing string $a_i a_{i+1} \cdots a_j$. At Line 1, all $l[i, i]$ for $i = 1, 2, ..., n$ are set to the correct value. All the other $l[i, j]$ are set to ∞ at first (Line 2), but those are updated in the for-loop from Line 3 to 19, and all $l[i,j]$ are set to the correct values by the end of the loop. In the for-loop, h increases one by one from 1 to n, and all $l[i,i+h_0-1]$ for $1 \leq i \leq n-h_0+1$ are set to the correct value when $h = h_0$. More precisely, in another for-loop from Line 4 to 18 inside the for-loop with $h = h_0$, each $l[i_0, i_0+h_0-1]$ is set to the correct value when $i = i_0$. In the for-loop from Line 5 to 10, the lengths of $h_0 - 1$ Case-2 possibilities are calculated for $l[i_0, i_0+h_0-1]$, and $l[i_0, i_0+h_0-1]$ is set to the correct value at the end of this for-loop. This is because the lengths of all the Case-1 possibilities for $l[i_0, i_0+h_0-1]$ are already calculated when $h < h_0$, that is, Case-1 possibility $(r[i,i+h_1-1])^d$ is calculated when $h = h_1 < h_0$. The lengths of all the Case-1 possibilities are calculated in the for-loop from Line 11 to 17. In the for-loop, the lengths of $(r[i,i+h-1])^d$ for $2 \leq d \leq \lfloor (n-i+1)/h \rfloor$ are calculated for current h and i. Finally, at the end of the for-loop with variable h, the length of a shortest RRS representing $a_1a_2\cdots a_n$, $l[1,n]$, is calculated.

A shortest RRS r can be constructed from the given string and 'type' because 'type' contains the information of which possibility is a shortest RRS: $r[i,j] = r[i+d]r[i+d+1,j]$ if type$[i,j] = (2,d)$ and $r[i,j] = (r[i,i+h-1])^{(j-i+1)/h}$ if type$[i,j] = (1,h)$, where the first component of 'type' indicates which case is selected. Starting from $r[1,n]$, recursively apply this refinement using 'type' information, an expression that only contains $r[i,i]$ for $i = 1,2,...,n$ except special symbols for repetition can be obtained. By replacing all $r[i,i]$ in the expression with a_i, a shortest RRS r is constructed. ConstructShortestRRS is the algorithm that executes this procedure. Each refinement and replacement above can be done in constant time and there are at most n refinements and at most n replacements, so ConstructShortestRRS runs in time $O(n)$.

Proposition 2 *Algorithm ShortestRRS runs in time* $O(n^3)$.

(Proof) Computational time needed outside the for-loop from Line 3 to 19 is at most $O(n^2)$. So, we only have to show that computational time needed for lines from Line 5 to 17, which are inside the two outer for-loops, is $O(n)$. The first for-loop from Line 5 to 10 is trivially $O(n)$. The number of executing the inside of the for-loop from Line 11 to 17 is at most n/h_0 when $h = h_0$, and $O(h_0)$ time is necessary for Line 12 in this case. Thus, $O(n)$ time is also needed for this for-loop. Therefore, computational time needed for lines from Line 5 to 17 is $O(n)$. □

2.3 Fast Algorithm for Width-k RRSs

Computational time $O(n^3)$ of algorithm ShortestRRS seems too slow for practical use. In this subsection, we consider a restricted class of *width-k RRSs*, for which a shortest one can be constructed more efficiently.

Let $(r)^h$ be an arbitrary repetition of RRS r. The *width* of repetition $(r)^h$ is defined as the length of string s represented by r. Here, the length of string $a_1a_2\cdots a_n$ for $a_i \in \Sigma$ is n, which is different from the length of an RRS.

RRS r is said to be *width-k* if the width of any repetition appeared in r is at most k.

Let $s(= a_1a_2\cdots a_n)$ denote an arbitrary string and let $r_k[i,j]$ denote a shortest width-k RRS representing $s[i,j]$. We also define $r_{k,j}[1,i]$ as a shortest width-k RRS with form $r_k[1,i-jh](r_k[i-jh+1,i-(j-1)h])^h$ for some $h \geq 2$ when such RRSs exist. Width-k RRS $r_{k,j}[1,i]$ is a shortest one among those representing $s[1,i]$ and ending with a width-j repetition. Note that $r_{k,j}[1,i]$ is NOT always defined. To simplify our notation, we let $r_{k,0}[1,i]$ denote a shortest width-k RRS among those representing $s[1,i]$ and containing no ending repetition. Then, the following fact trivially holds.

fact 1 *$r_k[1,i]$ is a shortest width-k RRS among $r_{k,j}[1,i]$ for $j = 0,1,...,k$.*

By the fact above, if $r_{k,j}[1,i]$ for $j = 0,1,...,k$ can be efficiently constructed from $r_{k,j}[1,i']$ for $j = 0,1,...,k$ and $i' < i$, then $r_k[1,n]$ can be efficiently calculated using dynamic programming. We show that this is true. For $j = 0$, the following proposition trivially holds, so we omit its proof.

Proposition 3 $r_{k,0}[1,i] = r_k[1,i-1]a_i$

For $j = 1,2,...,k$, we need the following property of repetition weight function w_R to efficiently construct $r_{k,j}[1,i]$.

$$w_R(h+1) = w_R(h) + \alpha \text{ for some constant } \alpha \text{ and for all } h \geq 2. \quad (1)$$

When w_R has this property, the following proposition holds.

Proposition 4 *Assume that w_R satisfies (1) and that $1 \leq j \leq \min\{i/2, k\}$ and $a_{i-j+1}a_{i-j+2}\cdots a_i = a_{i-2j+1}a_{i-2j+2}\cdots a_{i-j}$. If $r_{k,j}[1,i] = r_k[1,i-jh](r_k[i-jh+1,i-(j-1)h])^h$ for $h \geq 3$, then $r_{k,j}[1,i-j] = r_k[1,i-jh](r_k[i-jh+1,i-(j-1)h])^{h-1}$.*

(Proof) Assume that

$$r_{k,j}[1,i-j] = r_k[1,i-jh'](r_k[i-jh'+1,i-(j-1)h'])^{h'-1}$$

for some $h' \neq h$, and also assume that

$$\begin{aligned} &l(r_k[1,i-jh'](r_k[i-jh'+1,i-(j-1)h'])^{h'-1}) \\ &\quad < \; l(r_k[1,i-jh](r_k[i-jh+1,i-(j-1)h])^{h-1}). \end{aligned}$$

Then,

$$\begin{aligned} &l(r_k[1,i-jh'](r_k[i-jh'+1,i-(j-1)h'])^{h'}) \\ &\quad = \; l(r_k[1,i-jh'](r_k[i-jh'+1,i-(j-1)h'])^{h'-1}) + w_R(h') - w_R(h'-1) \\ &\quad = \; l(r_k[1,i-jh'](r_k[i-jh'+1,i-(j-1)h'])^{h'-1}) + w_R(h) - w_R(h-1) \\ &\quad < \; l(r_k[1,i-jh](r_k[i-jh+1,i-(j-1)h])^{h-1}) + w_R(h) - w_R(h-1) \\ &\quad = \; l(r_k[1,i-jh](r_k[i-jh+1,i-(j-1)h])^{h}) \end{aligned}$$

This contradicts the fact that $r_{k,j}[1,i] = r_k[1, i-jh](r_k[i-jh+1, i-(j-1)h])^h$. □

By this proposition, $r_{k,j}[1,i]$ for $j = 1, 2, ..., k$ is either modified $r_{k,j}[1, i-j]$ that is constructed from $r_{k,j}[1, i-j]$ by repeating the last repetition one more time, or $r_k[1, i-2j](r_k[i-2j+1, i-j])^2$, if it is defined.

```
ShortestW$k$RRS %[Finding a Shortest Width-$k$ RRS Representing String $a_1a_2\cdots a_n$]
Input:  $a_1a_2\cdots a_n$: string
        $w_\Sigma$: alphabet weight function
        $w_R$: repetition weight function
          satisfying $w_R(h+1) = w_R(h) + \alpha$ for some real number $\alpha$
Output: $r$ : shortest RRS representing $a_1a_2\cdots a_n$
        $l_r$ : length of $r$
 1: $(l, \text{type}) \leftarrow$ ShortestW$k$RRSLengthTable($a_1a_2\cdots a_n, w_\Sigma, w_R$)
 2: $l_{k,0}[1,0] = 0$, $j^*[0] = 0$
 3: for $i = 1$ to $n$ do
 4:   $l_{k,0}[1,i] = l_{k,j^*[i-1]}[1, i-1] + l[i,i]$, $j^*[i] = 0$
 5:   for $j = 1$ to $\min\{i, k\}$ do
 6:     if $j \le i/2$ and $a_{i-j+1}a_{i-j+2}\cdots a_i = a_{i-2j+1}a_{i-2j+2}\cdots a_{i-j}$ then
 7:       if $l_{k,j}[1, i-j] + \alpha \le l_{k,j^*[i-2j]}[1, i-2j] + l[i-2j+1, i-j] + w_R(2)$
          then
 8:         $l_{k,j}[1,i] = l_{k,j}[1, i-j] + \alpha$
 9:         $h[j,i] = h[j, i-j] + 1$
10:       else
11:         $l_{k,j}[1,i] = l_{k,j^*[i-2j]}[1, i-2j] + l[i-2j+1, i-j] + w_R(2)$
12:         $h[j,i] = 2$
13:       end if
14:     else
15:       $l_{k,j}[1,i] = \infty$
16:     end if
17:     if $l_{k,j}[1,i] < l_{k,j^*[i]}[1,i]$ then $j^*[i] = j$
18:   end for
19: end for
20: $l_r \leftarrow l_{k,j^*[n]}[1,n]$
21: $r \leftarrow$ ConstructShortestW$k$RRS($a_1a_2\cdots a_n$, type, $j^*$, $h$)
22: return $(r, l_r)$
```

Figure 2: Algorithm ShortestWkRRS

Now, we know that width-k RRSs $r_{k,j}[1,i]$ for $j = 0, 1, ..., k$ and their shortest one $r_k[1,i]$ can be constructed from $r_{k,j}[1,i']$ and $r_k[1,i']$ for $i' < i$, and $r_k[g,h]$ for $1 \le g \le h < i$ and $h - g + 1 \le k$. A dynamic programming using this fact is algorithm ShortestWkRRS shown in Fig. 2.

For all $1 \le i \le j \le n$ with $j - i + 1 \le k$, the algorithm first calculates length $l[i,j]$ of a shortest one $r_k[i,j]$ among the RRSs representing $s[i,j]$. This task is done by executing algorithm ShortestWkRRSLengthTable shown in Fig. 3. Note that ShortestWkRRSLengthTable is a modified ShortestRRS in Fig. 1 which is modified so as not to calculate $l[i,j]$ for $j-i+1>k$.

Then, the algorithm calculates length $l_{k,j}[1,i]$ of the width-k RRS $r_{k,j}[1,i]$ for $i=1,2,...,n$ and $j=0,1,...,k$ using dynamic programming. Note that $r_{k,j}[1,i]$ is set to ∞ when $r_{k,j}[1,i]$ is not defined at Line 15. In the algorithm, we calculates

$$j^*[i]=\arg\min_{0\le j\le k} r_{k,j}[1,i]$$

instead of calculating the length of $r_k[1,i]$, which is equal to $l_{k,j^*[i]}[1,i]$ using $j^*[i]$.

A shortest width-k RRS $r_k[1,n]$ representing inputted string $a_1a_2\cdots a_n$ can be constructed using 'type' information returned by ShortestWkRRSLengthTable, $j^*[i]$ and $h[j,i]$ for $i=1,2,...,n$ and $j=1,2,...,k$, where $h[j,i]$ is the number of times repeated in the last repetition of $r_{k,j}[1,i]$. The construction can be done as follows. First, using $j^*[n]$ and $h[j^*[n],n]$, you know that

$$\begin{aligned} & r_k[1,n] \\ & = \begin{cases} r_k[1,n-j^*[n]h[j^*[n],n]](r_k[n-j^*[n]+1,n])^{h[j^*[n],n]} & \text{if } j^*[n]>0 \\ r_k[1,n-1]a_n & \text{if } j^*[n]=0. \end{cases} \end{aligned}$$

RRS $r_k[n-j^*[n]+1,n]$ can be constructed by calling ConstructShortestRRS($a_{n-j^*[n]+1}a_{n-j^*[n]+2}\cdots a_n$,type), a function used in Algorithm ShortestRRS. (See Fig. 1.) Thus, construction problem of $r_k[1,n]$ is reduced to a smaller construction problem of $r_k[1,n-j^*[n]h[j^*[n],n]]$ when $j^*[n]>0$ and $r_k[1,n-1]$ when $j^*[n]=0$. Repeated application of this reduction finally leads to construction problem of $r_k[1,0]$, which is trivially λ. By this procedure, Algorithm ConstructShortestWkRRS at Line 21 can construct $r_k[1,n]$ in time $O(n)$.

Proposition 5 *Algorithm ShortestWkRRS runs in time $O(k^2n)$.*

(Proof) Computational time needed for ShortestWkRRSLengthTable is $O(k^2n)$ because the inside of the loop from Line 3 to 19 is executed at k times, the inside of the loop from Line 5 to 10 is also executed at most k times, and the loop from Line 11 to 17 is executed at most k/h times while time $O(h)$ is necessary at Line 12.

Computational time needed for lines from Line 3 to 19 in ShortestWkRRS is also $O(k^2n)$ by the following reason. One execution of the inside of the loop from Line 5 to 18 needs $O(k)$ time because execution of Line 6 needs $O(k)$ and other lines are executed in constant time. Since the insides of the nested for-loops are executed at most n times and k times, respectively, $O(k^2n)$ time is necessary in total.

Thus, together with the fact ConstructShortestWkRRS needs $O(n)$ time, we can conclude that ShortestWkRRS runs in time $O(k^2n)$. □

```
**ShortestW$k$RRSLengthTable** %[Calculate the Length $l[i,j]$ of the Shortest
                              RRS Representing String $a_i a_{i+1} \cdots a_j$ for All $0 < j - i < k$]
Input:  $a_1 a_2 \cdots a_n$: string
        $w_\Sigma$: alphabet weight function
        $w_R$: repetition weight function
Output: $l$ : lengths of the shortest RRS representing
                              string $a_i a_{i+1} \cdots a_j$ for all $0 < j - i < k$
        type : structure information of the shortest RRS representing
                              string $a_i a_{i+1} \cdots a_j$ for all $0 < j - i < k$
1: $l[i,i] \leftarrow w_\Sigma(a_i)$ for $i = 1, 2, ..., n$
2: $l[i,j] \leftarrow \infty$ for $i = 1, 2, ..., n, j = 1, 2, ..., n, 0 < j - i < k$
3: for $h = 1$ to $k$ do
4:   for $i = 1$ to $n - h + 1$ do
5:     for $d = 0$ to $h - 2$ do
6:       if $l[i, i+d] + l[i+d+1, i+h-1] < l[i, i+h-1]$ then
7:         $l[i, i+h-1] \leftarrow l[i, i+d] + l[i+d+1, i+h-1]$
8:         type$[i, i+h-1] \leftarrow (2, d)$
9:       end if
10:    end for
11:    for $s = i + h$ to $i + k - h$ by $h$ do
12:      if $a_s a_{s+1} \cdots a_{s+h-1} \neq a_i a_{i+1} \cdots a_{i+h-1}$ then break
13:      if $l[i, i+h-1] + w_R((s-i)/h + 1) < l[i, s+h-1]$ then
14:        $l[i, s+h-1] \leftarrow l[i, i+h-1] + w_R((s-i)/h + 1)$
15:        type$[i, s+h-1] \leftarrow (1, h)$
16:      end if
17:    end for
18:  end for
19: end for
20: return $(l, \text{type})$
```

Figure 3: Algorithm ShortestWkRRSLengthTable

3 Finding a Minimum Repetition Representation Tree

3.1 Problem Setting

An *ordered tree* is a rooted tree in which all nodes having the same parent node are totally ordered. In this paper, all child nodes of a tree in any figure are supposed to have left-to-right ordering. A *labeled ordered tree* is an ordered tree in which all nodes are labeled. We assume that labels are members in Σ, a finite set of alphabets.

In this section, we introduce the following notation. Let $\pi(T)$ denote the root node of an ordered tree T. A subtree rooted by node v in a tree T is denoted by T_v. Let $c(v)$ denote the number of child nodes of a node v, and let v^i denote the ith child node of v. For notational simplicity, we use T^i instead of $T_{\pi(T)^i}$, the subtree rooted by the ith child node of the root node of T. Let $\phi(v)$ denote the label of a node v. We let $V(T)$ denote the set of nodes in a tree T.

Here, we define a *repetition representation tree*, RRT for short, which is a generalized concept of a labeled ordered tree. Only one difference from a general labeled ordered tree is that an RRT may contain special nodes called *repetition information nodes* whose label is not an alphabet in Σ but a number at least 2. A repetition information node v labeled number h represents such a repetition structure that the subtree sequence rooted by v is repeated h times as a subtree sequence directly rooted by the v's parent node p. (See Fig.4.) Any repetition information node must be neither a root node nor a leaf node.

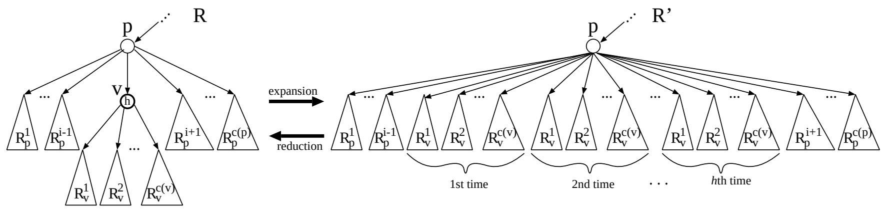

Figure 4: Repetition information node and its expansion

The operation of transforming an RRT with repetition information node v into an RRT representing the same tree without node v is called *expansion* of v, and the opposite operation is called *reduction* to v. When all the repetition information nodes in RRT R are expanded, a general labeled ordered tree T is obtained. Then, we say that R represents T.

For a repetition information node v in an RRT R, the *width* of v is defined as the number of v's child nodes in an RRT R' which is made by expanding all repetition information nodes below v. Note that a repetition information node v with width k in an RRT R represents the repetition of k subtree sequence in a labeled ordered tree T represented by R. RRT R is said to be *width-k* if the width of any repetition information node is at most k.

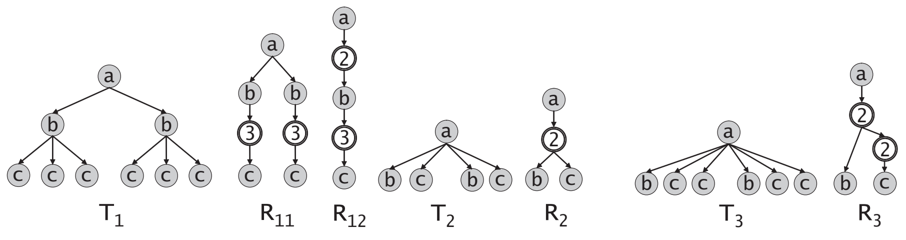

Figure 5: Examples of RRTs

Example 3 *T_1, T_2 and T_3 are general labeled ordered trees and also RRTs representing themselves. R_{11} and R_{12} are RRTs representing T_1, R_2 is an RRT representing T_2,* *and R_3 is an RRT representing T_3. Among R_{11}, R_{12}, R_2 and R_3, the first two RRTs are width-*1*, the first three RRTs are width-*2*, and the all RRTs are width-*3.

Similar to the definition of the length of an RRS in the previous section, we define the *size* $m(R)$ of an RRT R using given two functions *alphabet weight function* w_Σ and *repetition weight function* w_R as follows.

$$m(R) = \begin{cases} 0 & \text{if } R \text{ is an empty tree} \\ w_R(\phi(\pi(R))) + \sum_{i=1}^{c(\pi(R))} m(R^i) & \text{if } \pi(R) \text{ is a repetition information node} \\ w_\Sigma(\phi(\pi(R))) + \sum_{i=1}^{c(\pi(R))} m(R^i) & \text{otherwise} \end{cases}$$

RRT-version of our problem is the following.

Problem 2 *Given a labeled ordered tree T, an alphabet weight function w_Σ and a repetition weight function w_R, find a minimum RRT R that represents T.*

Remark 1 *When the height of T is one, the above problem can be reduced to an RRS-version problem. Let T be a height-1 tree with leaf node label sequence $a_1, a_2, ..., a_d$. Then, the reduced RRS-version problem is that of finding a shortest RRS r representing string $a_1 a_2 \cdots a_d$ for the same alphabet and repetition weight functions. There is a one-to-one correspondence between an RRT R representing T and an RRS $\phi(\pi(T))r$, which is a concatenation of the root label $\phi(\pi(T))$ and an RRS r representing $a_1 a_2 \cdots a_d$. Do tree traversal of R, and output '(' at the first visit of a repetition information node, output '$)^h$' at the last visit of a repetition information node labeled h, and output an alphabet a at the first visit of an node labeled a. Then, outputted string $\phi(\pi(T))r$ is an RRS corresponding to R. RRS $\phi(\pi(T))r$ is a unique RRS encoding of an RRT R representing a height-1 tree T, and the conversion of the opposite direction is also possible. Furthermore, $m(R) = l(r) + w_\Sigma(\phi(\pi(T)))$ when the same w_Σ and w_R are used. Thus, minimizing $m(R)$ is equivalent to minimizing $l(r)$. By this reduction, the above problem can be solved when the height of T is at most one.*

3.2 Algorithm

We show an algorithm for finding a minimum RRT representing a given arbitrary labeled ordered tree T, given an arbitrary alphabet weight function w_Σ and an arbitrary repetition weight function w_R.

The algorithm is named MinimumRRT and shown in Fig. 6. It uses divide-and-conquer strategy. First, it solves subproblems of finding a minimum RRT R_i representing an labeled ordered subtree T^i for each $i = 1, 2, ..., c(\pi(T))$. These subproblems are solved by recursively calling algorithm MinimumRRT. Then, the problem

```
MinimumRRT %[Construct the Minimum RRT Representing
                                          a Labeled Ordered Tree $T$]
Input: $T$: labeled ordered tree
       $w_\Sigma$: alphabet weight function
       $w_R$: repetition weight function
Output:$R$: minimum RRT $R$ representing $T$
       $m_R$: size of a minimum RRT $R$ representing
       $T$
 1: if $\pi(T)$ has no child node then
 2:    $R \leftarrow T$, $m_R \leftarrow w_\Sigma(\phi(\pi(T)))$
 3: else
 4:    $v \leftarrow \pi(T), d \leftarrow c(v)$
 5:    $(R_i, m_i) \leftarrow$ MinimumRRT$(T^i, w_\Sigma, w_R)$ for $i = 1, 2, ..., d$
 6:    $\psi(v) \leftarrow$ AssignLabel$(\phi(v)\psi(v^1)\psi(v^2)\cdots\psi(v^d))$
 7:    $w_{\Sigma'}(\psi(v^i)) \leftarrow m_i$ for $i = 1, 2, ..., d$
 8:    $(r, m_R) \leftarrow$ ShortestRRS$(\psi(v^1)\psi(v^2)\cdots\psi(v^d), w_{\Sigma'}, w_R)$
 9:    $R \leftarrow$ ConstructMinimumRRT$(r, R_1, R_2, ..., R_d, \phi(v))$
10: end if
11: return $(R, m_R)$
```

Figure 6: Algorithm MinimumRRT

of constructing a minimum RRT R representing T from $R_1, R_2, ..., R_{c(\pi(T))}$ is reduced to the problem of finding a shortest RRS r representing a string corresponding to the sequence $T^1, T^2, ..., T^{c(\pi(T))}$. The reduced problem is solved by calling algorithm ShortestRRS developed in the previous section.

Algorithm MinimumRRT uses two other algorithms, AssignLabel and Construct-MinimumRRT.

AssignLabel is an algorithm that assigns a label in $\Sigma' \supseteq \Sigma$ to an inputted string, and returns the label. The algorithm assigns an inputted string itself for a string composed of one alphabet in Σ. If the same string as the one inputted before is inputted, then the already assigned label is returned. Otherwise, the algorithm assigns a new label. Note that AssignLabel assigns the same label if and only if inputted strings are the same. In MinimumRRT, AssignLabel is used for labeling each node v with $\psi(v)$, which is determined by the subtree T_v. This means that

$$\psi(u) = \psi(v) \Leftrightarrow T_u = T_v. \tag{2}$$

Though AssignLabel uses labels $\phi(v)$ and $\psi(v^1), \psi(v^2), ..., \psi(v^{c(v)})$ alone for calculation of $\psi(v)$, (2) holds even for $\psi(v)$ calculated like this when (2) holds for $\psi(v^1), \psi(v^2), ..., \psi(v^{c(v)})$ by induction. Since (2) trivially holds for leaf nodes, (2) holds for all nodes in T.

ConstructMinimumRRT is an algorithm that constructs a minimum RRT R representing T from an RRS r representing $\psi(v^1)\psi(v^2)\cdots\psi(v^{c(v)})$, minimum RRTs $R_1, R_2, ..., R_{c(v)}$ representing $T^1, T^2, ..., T^{c(v)}$, respectively, and the root label $\phi(v)$ of T, where $v =$

$\pi(T)$. The algorithm converts an RRS r to the corresponding RRT R by the conversion described in Remark 1. Only one difference from the conversion is to add R_i instead of a leaf node labeled $\psi(v^i)$.

Theorem 1 *Given a labeled ordered tree T, an alphabet weight function w_Σ and a repetition weight function w_R, algorithm MinimumRRT outputs a minimum RRT R representing T and its size $m(R)$.*

(Proof) The proof is an induction on the height h of T. When $h = 0$, $\pi(T)$ has no child node, so the minimum RRT is T itself and its size is $w_\Sigma(\phi(\pi(T)))$, which are outputted by the algorithm.

Let $h = h_0$, and suppose that the theorem holds when $h \leq h_0 - 1$. Then, (R_i, m_i) for $i = 1, 2, ..., c(\pi(T))$ obtained at Line 5 is a pair of the minimum RRT R_i representing T^i and its size m_i by the assumption because the height of T^i is at most $h_0 - 1$. In any RRT R' representing T, the subtree R'_i corresponding to T^i, that is, the subtree that becomes T^i by expanding all repetition information nodes included in it, can be replaced with any R''_i representing T^i. This means that a minimum R^* representing T must have a minimum R^*_i representing T^i as a subtree corresponding to T^i for each i, and any R^*_i representing T^i is possible if R^*_i has minimum size among RRTs representing T^i. Thus, by applying reduction operations to T' that is an RRT made from T by replacing each T^i with R_i, a minimum RRT R^* can be obtained. During such a reduction, R_i does not change, so it can be replaced by a node labeled an alphabet $\psi(v^i) \in \Sigma'$ with $m_{\Sigma'}(\psi(v^i)) = m(R_i) = m_i$, where v^i is the root node of T^i. The same result can be obtained by substituting R_i for a node labeled $\psi(v^i)$ after the reduction. Therefore the problem is reduced to that of finding a minimum RRT representing a height-1 tree with a leaf node label sequence $\psi(v^1), \psi(v^2), ..., \psi(v^{c(v)})$ and the root labeled $\phi(v)$, where v is the root node of T. By Remark 1, this can be solved by finding a shortest RRS r representing the string $\psi(v^1)\psi(v^2)\cdots\psi(v^{c(v)})$. Since MinimuRRT conducts just the same process as we described above, the theorem holds when $h = h_0$. □

For the class of width-k RRTs, the following corollary holds.

Corollary 1 *Given a labeled ordered tree T, a alphabet weight function w_Σ and a repetition weight function w_R satisfying (1), a modified MinimumRRT made by replacing ShortestRRT with ShortestWkRRT outputs a minimum width-k RRT R representing T and its size $m(R)$.*

Theorem 2 *Algorithm MinimumRRT runs in time $O(n^3)$ for an arbitrary labeled ordered tree with n nodes.*

(Proof)

For a given tree T, algorithm MinimumRRT is called just once for each node, precisely speaking, just once for a subtree rooted by each node. Thus, let us consider process time needed for each node and its summation. For leaf nodes, constant time is necessary for the algorithm. Thus, time $O(n)$ is enough for the algorithm to process all the leaf nodes.

For each internal node v, we claim that $O(c(v)^3)$ is necessary for the algorithm. Since we do not have to take account of the process time of recursively called MinimumRRT here, time $O(c(v))$ is needed for Line 5. AssignLabel runs in time $O(c(v))$ under the assumption of no limitation on the amount of memory. ShortestRRS runs in time $O(c(v)^3)$ by Proposition 2. All the other parts passed when the else-clause is executed including ConstructMinimumRRT are processed in time $O(c(v))$. Therefore, time $O(c(v)^3)$ is necessary for processing each internal node.

Let I denote the set of internal nodes in T. Then, $\sum_{v\in I} c(v) = n-1$ holds. Therefore, $\sum_{v\in I} c(v)^3 \leq n^3$ holds. Thus, time$O(n^3)$ is enough for the algorithm to process all the internal nodes.

Totally, MinimumRRT runs in time $O(n^3)$. □

The following corollary for width-k RRTs can be proved similarly.

Corollary 2 *A modified MinimumRRT made by replacing ShortestRRT with ShortestWkRRT runs in time $O(k^2n)$ for an arbitrary labeled ordered tree with n nodes.*

4 Discussion

For a string s with length n, one repetition (tandem array) can be specified by a triple (i, d, h), where i is its starting position, d is its width and h is the number of repetitions. Since $h \geq 2$, the number of triples can be

$$\sum_{d=1}^{\lfloor n/2\rfloor} \sum_{i=d}^{n} (\lfloor i/d\rfloor - 1) = \Theta(n^2 \log n).$$

The problem of finding a shortest RRS representing s is a kind of finding an optimal combination of a (possibly large) number of repetitions under a certain criterion. The RRS-form constraint is equivalent to the condition that any two members (i_1, d_1, h_1) and (i_2, d_2, h_2) in a combination must be

$$\textit{disjoint}\ (i_1 + d_1h_1 \leq i_2 \text{ or } i_2 + d_2h_2 \leq i_1)$$

or

$$\textit{nested}\ (i_1 \leq i_2, i_2 + d_2h_2 \leq i_1 + d_1 \text{ or } i_2 \leq i_1, i_1 + d_1h_1 \leq i_2 + d_2).$$

Recently, repetition structure analysis of web pages for information extraction have been studied by several researchers [2, 5]. In such a analysis, contents such as texts and images are not considered generally, and only HTML or XML tag sequences are analysed. Such a sequence is a string as it is, and is also represented by a labeled ordered tree called a "tag tree". The problems dealt in the analysis can be seen as the one of finding a good combination of repetitions under the RRS-form or RRT-form constraint. In this section, we show difference between repetition-analysis results by the conventional methods [2, 5] and those of our method using concrete examples.

First, let us consider how to select nested repetitions. Approach taken by Nanno, Saito and Okumura [5] is a *bottom-up selection.* Their method first finds a non-overlapped (best) combination of repetitions among those including no repetitions inside. Then, it replaces every found repetition with one appropriate alphabet and repeats the same procedure until no repetition is found. For a string $abbcaabbca$, there are four repetitions $(2,1,2),(5,1,2),(7,1,2)$ and $(1,5,2)$, and only the last repetition includes other repetitions inside. Thus, the first three repetitions are selected at the first round[1], and the string is modified to $aBcABca$ by substitutions of selected repetitions ($aa \to A, bb \to B$). Since the modified string has no repetition, the RRS made by bottom-up selection approach is $a(b)^2c(a)^2(b)^2ca$. For $w_\Sigma(a) = w_\Sigma(b) = w_\Sigma(c) = 1$ and $w_R(h) = 1$ for all $h \geq 2$, our algorithm outputs $(a(b)^2ca)^2$ for the input string $abbcaabbca$ because $l(a(b)^2c(a)^2(b)^2ca) = 10$ and $l((a(b)^2ca)^2) = 6$. By virtue of global optimization, the combination of repetitions our method selected looks more non-trivial than the one selected by the bottom-up approach.

Second, consider a greedy approach, which repeatedly selects one best repetition among the rest repetitions. What is the best repetition depends on what criterion is used. How long substring is covered by a repetition seems one of proper criterions, and actually, Liu, Grossman and Zhai [2] used this criterion[2] to select a repetition. For a string $abaababaabbaab$, the repetition covering the longest substring is $(1,5,2)$, which covers the first 10-length string. Thus, using the greedy approach with the above criterion, RRS $(ab(a)^2b)^2b(a)^2b$ is made finally. However, for the above w_Σ and w_R, the length of $(ab(a)^2b)^2b(a)^2b$ is 10, which is larger than the length 9 of another RRS $(aba)^2(b(a)^2b)^2$ representing the same string. In this case again, global optimization makes our method success to find a more non-trivial RRS than the methods using the greedy approach does. .

5 Concluding Remarks

What set of repetitions in a sequence or a tree is a structurally essential one? Finding such a set of repetitions is an important task in structural analysis of sequences and trees. Our $O(n^3)$ algorithms seems too slow for practical use, but the $O(k^2n)$ algorithms appears fast enough when the width of repetitions are always smaller than a certain bound k. For example, width bound 20 seems large enough for extraction

[1]Since their method finds a combination that maximizes the sum of the number of repetitions, all the three repetitions are selected.

[2]Their method also prefers smaller starting position.

of data records from HTML tag trees. Actually, we are now using our algorithm in our online application of extracting keyword related items from an arbitrary Web pages [6]. Unfortunately, repetitions our algorithms deal with are only *identical* ones. Extension of our algorithms so as to deal with *non-identical* repetitions would enable wide range of applications.

Acknowledgements

We would like to thank Prof. Hiroki Arimura for his valuable comments.

References

[1] Benson, G.: Tandem repeats finder: a program to analyze DNA sequences, *Nucleic Acids Research*, 27(2), pp.573-580, 1999.

[2] Liu, B., Grossman, R. and Zhai, Y.: Mining Data Records in Web Pages, *In Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp.601-606, 2003.

[3] Stoye, J. and Gusfield, D.: Simple and Flexible Detection of Contiguous Repeats Using a Suffix Tree, *Theoretical Computer Science*, 270, pp.843-856, 2002.

[4] Main, M. and Lorentz, R.: An $O(n \log n)$ algorithm for finding all repetitions in a string, *Journal of Algorithms*, 5, pp.422-432, 1984.

[5] Nanno, T., Saito, S. and Okumura, M.: Structuring Web Pages Based on Repetition Of Elements, *IPSJ Journal*, 45(9), pp.2157-2167, 2004. (In Japanese.)

[6] Nakamura, A., Hasegawa, H., Saito, T. and Kudo, M.: Flexible Wrappers for Keyword-Related Information, *Hokkaido University Division of Computer Science TCS Technical Report Series A*, TCS-TR-A-07-24, 2007.