IPSJ Transactions on Bioinformatics Vol. 1 13–22 (Nov. 2008)

Original Paper

A Linear Time Algorithm that Infers Hidden Strings from Their Concatenations

Tomohiro Yasuda[†1]

Let $\mathcal{T}$ be a set of hidden strings and $\mathcal{S}$ be a set of their concatenations. We address the problem of inferring $\mathcal{T}$ from $\mathcal{S}$. Any formalization of the problem as an optimization problem would be computationally hard, because it is NP-complete even to determine whether there exists $\mathcal{T}$ smaller than $\mathcal{S}$, and because it is also NP-complete to partition only two strings into the smallest common collection of substrings. In this paper, we devise a new algorithm that infers $\mathcal{T}$ by finding common substrings in $\mathcal{S}$ and splitting them. This algorithm is scalable and can be completed in $O(L)$-time regardless of the cardinality of $\mathcal{S}$, where L is the sum of the lengths of all strings in $\mathcal{S}$. In computational experiments, 40,000 random concatenations of randomly generated strings were successfully decomposed, as well as the effectiveness of our method for this problem was compared with that of multiple sequence alignment programs. We also present the result of a preliminary experiment against the transcriptome of *Homo sapiens* and describe problems in applications where real large-scale cDNA sequences are analyzed.

1. Introduction

Let $\mathcal{T}$ be a set of hidden strings and $\mathcal{S}$ be a set of their concatenations. We consider the problem of inferring $\mathcal{T}$ from $\mathcal{S}$ when only $\mathcal{S}$ is given. The problem is motivated by the analysis of cDNA sequences. A gene might have more than one cDNA sequence by inserting or deleting alternative segments. This phenomenon affects 35–74% of human genes [8),10)] and is prevalent in many organisms [14)]. Usually, alternatively spliced sequences are detected by aligning them with genomic sequences [7),10)]. Nonetheless, methods that require as input only cDNA sequences would be useful because not all organisms have had their genomic sequences determined. To clarify the problem, we show a small example.

Example 1 Suppose that we are given $\mathcal{S} = \{S_0, S_1, S_2\}$, where

$S_0 = \texttt{ACGGTCTAGAATAGCAGGCTCGTCCTATGGCATTTT}$,
$S_1 = \texttt{CATCTGGTAGCAGGCTCGTCCTATCCAAGTAAAGGAC}$,
$S_2 = \texttt{CATCTGGTAAGTGGGCCGTCCTAT}$.

These are concatenations of strings in a set $\mathcal{T} = \{T_i | 0 \leq i < 8\}$, where

$T_0 = \texttt{ACGGTCTAGAAT}$, $T_1 = \texttt{AGCAGGCTC}$,
$T_2 = \texttt{GTCCTAT}$, $T_3 = \texttt{GGCATTTT}$,
$T_4 = \texttt{CATCTGGT}$, $T_5 = \texttt{CCAAGT}$,
$T_6 = \texttt{AAAGGAC}$, $T_7 = \texttt{AAGTGGGCC}$.

In fact, S_0, S_1 and S_2 can be rewritten as:

$S_0 = T_0T_1T_2T_3$,
$S_1 = T_4T_1T_2T_5T_6$,
$S_2 = T_4T_7T_2$.

We aim at inferring $\mathcal{T}$ from $\mathcal{S}$. □

If we formalize the problem as an optimization problem, it is difficult to give an efficient solution. Let $|\mathcal{S}|$ and $|\mathcal{T}|$ respectively be the cardinalities of $\mathcal{S}$ and $\mathcal{T}$. Néraud [17)] considered the problem of determining, for a given set $\mathcal{S}$ of strings and an integer k, whether there exists a set $\mathcal{T}$ of strings such that $\mathcal{S} \subseteq \mathcal{T}^*$ and $|\mathcal{T}| \leq k$. Néraud proved that this problem is NP-complete even when $k = |\mathcal{S}| - 1$; that is, it is NP-complete to determine the existence of $\mathcal{T}$ that is smaller than $\mathcal{S}$. Lopresti and Tomkins [13)] studied the problem of comparing two strings by extracting collections of substrings and placing them into correspondence. They proved that, when substrings in the collection do not overlap each other in given strings whereas they cover the whole of the given strings, finding the smallest such collection is NP-complete. This problem is called the minimum common substring problem (MCSP), and has been attracting attention [3)–5)]. It was also proven that a restricted variant of MCSP called k-MCSP, where each symbol in the given strings occurs at most k times in each given string, is NP-hard for $k \geq 2$ [5)]. Thus, several approximation algorithms have been proposed for MCSP [3)–5)]. In addition, Lopresti and Tomkins [13)] proposed polynomial time algorithms for cases where the constraints on collections of substrings were relaxed. However, these algorithms compare only two strings. Any extension for more than two strings would be computationally hard, taking into account that obtaining the

†1 Central Research Laboratory, Hitachi, Ltd.

optimal multiple sequence alignment (MSA) is NP-hard [1),21)] whereas the optimal alignment of two strings can be obtained in polynomial time [16),20)]. Despite these difficulties, a method scalable to $|\mathcal{S}|$ is preferable for applications and even for accuracy, since the more strings we compare, the more chance we have to find strings in $\mathcal{T}$ by exploiting differences among strings in $\mathcal{S}$.

For practical uses, a lot of MSA programs have been available [2)]. However, their purpose is not to decompose given strings into substrings of which given strings are concatenations, but to obtain alignments by finding similar regions.

In this paper, we propose a fast and scalable algorithm for inferring $\mathcal{T}$ from $\mathcal{S}$. Our approach is based on finding common substrings and splitting them. We make two main contributions. First, we formally define a class of strings called disjoint common substrings (DCS's) that can be determined only from $\mathcal{S}$ and a positive integer parameter. Each of DCS's corresponds to a string in $\mathcal{T}$ or a concatenation of strings in $\mathcal{T}$ that always occur adjacent in the same order in $\mathcal{S}$. Second, we devise an algorithm that identifies all DCS's by all-to-all comparison of strings in $\mathcal{S}$. This algorithm can be completed in $O(L)$-time regardless of $|\mathcal{S}|$, where L is the sum of the lengths of all strings in $\mathcal{S}$. These contributions enable us to efficiently decompose a large number of strings into non-trivial, non-overlapping substrings.

The rest of this paper is organized as follows. In Section 2, we define terms and notations used in this paper. In Section 3, we give a formal definition of DCS's. In Section 4, an algorithm that identifies DCS's in $O(L)$-time is proposed. In Section 5, we show the results of computational experiments to present effectiveness of our method and to describe problems when we apply our method to real large-scale biological sequences. Finally in Section 6, we present our conclusions.

2. Preliminaries

Let $\mathcal{T}$ be a set of hidden strings and $\mathcal{S}$ be a set of their concatenations. Also, let N be the cardinality of $\mathcal{S}$, and L be the sum of the lengths of all strings in $\mathcal{S}$. We denote by Σ a finite alphabet of which strings in $\mathcal{S}$ consist, by Σ^* the set of possibly empty strings, and by Σ^+ the set of non-empty strings. We use the terms *string* and *sequence* interchangeably. For a string $s \in \Sigma^*$, the length of s is denoted by $|s|$. When $s = s_1 s_2 s_3$ for some $s_1, s_2, s_3 \in \Sigma^*$, they are respectively called a *prefix*, a *substring*, and a *suffix* of s. Each of them is *proper* if it is not identical to s. The position of each symbol in $S_i \in \mathcal{S}$ is indexed by an integer j, where $0 \leq j < |S_i|$. When s is a substring of $S_i \in \mathcal{S}$ beginning at the j-th position of S_i, we say that s *occurs at* (i, j) or that (i, j) is an *occurrence* of s. Let $Occ(s)$ be the set of all occurrences of s where $|s| > 0$. For any integer k, the set $\{(i, j+k) | (i, j) \in Occ(s)\}$ is denoted by $Occ(s) + k$. An empty set is denoted by $\emptyset$.

Let STree($\mathcal{S}$) be a generalized suffix tree [6)] that contains all strings in $\mathcal{S}$. Each string in $\mathcal{S}$ is given a distinct termination symbol at its right end so that identical suffixes of distinct strings in $\mathcal{S}$ end at distinct leaves in $\mathcal{S}$ [6)]. A *path-label* of a node v in STree($\mathcal{S}$) is the concatenation of edge labels from the root to v. We denote by $p(v)$ the string obtained by removing any termination symbol from the path-label of v. Let $\mathcal{L}(i, j)$ be the leaf of STree($\mathcal{S}$) that represents the j-th suffix of $S_i \in \mathcal{S}$.

We capture common substrings in $\mathcal{S}$ with maximal common substrings defined as follows.

Definition 1 (MCS) A string $m \in \Sigma^+$ is *a maximal common substring (MCS) for* $\mathcal{S}$ *and* l, if m is a substring of some $S_i \in \mathcal{S}$ and has the following properties:
(M1)$|m| \geq l$.
(M2)$Occ(m) \neq Occ(ms)$ for any $s \in \Sigma^+$.
(M3)$Occ(m) \neq Occ(sm) + |s|$ for any $s \in \Sigma^+$.
The set of all MCS's for $\mathcal{S}$ and l is denoted by MCS($\mathcal{S}, l$). □

We show MCS($\mathcal{S}, l$) for Example 1 in **Fig. 1**. MCS's are a natural extension of maximal repeats [6)] to identical substrings in an arbitrary number of given strings. MCS's can be identified in $O(L)$-time in the same way to identify maximal repeats. MCS's were also referred to as *core blocks* by Leung et al. [12)]. However, to emphasize that MCS's are common substrings in $\mathcal{S}$ and to simplify the exposition, we use this name and definition.

We also use the following class of substrings.

Definition 2 RightMCS($\mathcal{S}, l$) is a set of non-empty strings, each of which is a substring of some $S_i \in \mathcal{S}$ and satisfies (M1) and (M2). □

RightMCS($\mathcal{S}, l$) is a natural extension of strings considered in the DNA contam-

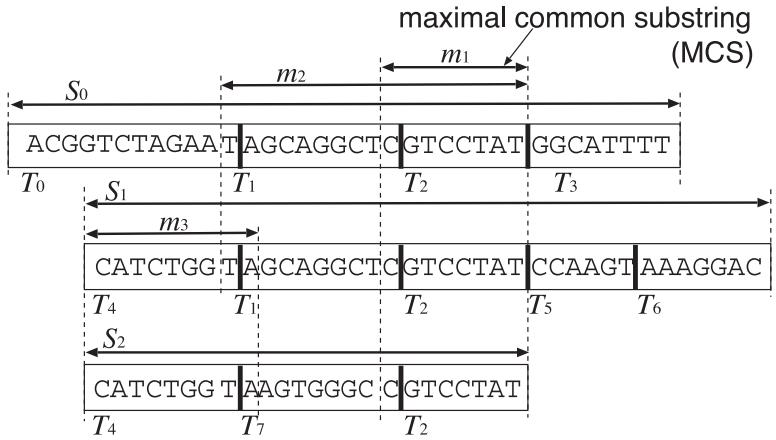

Fig. 1 MCS($\mathcal{S}, l$) for Example 1. Here, $l = 6$. Strings m_1, m_2, m_3, S_0, S_1, and S_2 are MCS's. They are all maximal, that is, they lose some of their occurrences if they are extended to either the left or right.

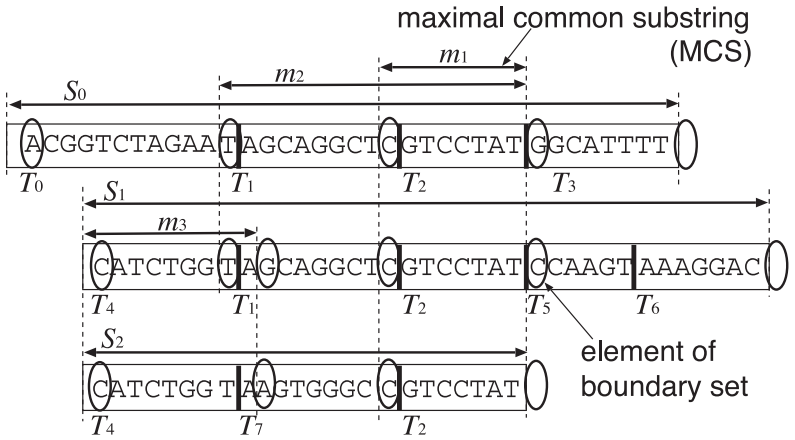

Fig. 2 The boundary set $\mathcal{B}(\mathcal{S}, l)$ for Example 1. Each circle at the j-th position of S_i indicates that $(i, j) \in \mathcal{B}(\mathcal{S}, l)$.

ination problem discussed in 6) where $|\mathcal{S}| = 2$. All strings in RightMCS($\mathcal{S}, l$) can be found in $O(L)$-time as $p(v)$ of nodes v in STree($\mathcal{S}$) such that $|p(v)| \geq l$ [6]. Note that any $r \in$ RightMCS($\mathcal{S}, l$) is a suffix of some MCS. In fact, there exists some $m \in$ MCS($\mathcal{S}, l$) such that $Occ(m)+|m| = Occ(r)+|r|$ for any $r \in$ RightMCS($\mathcal{S}, l$).

3. Definition of DCS's

In Fig. 1, m_1(=`CGTCCTAT`) is an MCS shared by all of S_0, S_1, and S_2, while m_2(=`TAGCAGGCT CGTCCTAT`) is shared by only S_0 and S_1. This suggests that there is a string in $\mathcal{T}$ shared by all of S_0, S_1, and S_2, and on its left, there is another string in $\mathcal{T}$ shared by only S_0 and S_1. To infer both of them, we should split m_2 at a boundary of m_1. We generalize this way of inference.

Definition 3 (Boundary set) The set called *boundary set for $\mathcal{S}$ and l*, denoted by $\mathcal{B}(\mathcal{S}, l)$, is defined by the following equation:

$\mathcal{B}(\mathcal{S}, l) = \mathcal{B}_L(\mathcal{S}, l) \cup \mathcal{B}_R(\mathcal{S}, l)$,

where

$$\mathcal{B}_L(\mathcal{S}, l) = \bigcup_{m \in \mathrm{MCS}(\mathcal{S}, l)} Occ(m),$$

$$\mathcal{B}_R(\mathcal{S}, l) = \bigcup_{m \in \mathrm{MCS}(\mathcal{S}, l)} (Occ(m) + |m|).$$

□

In **Fig. 2**, we show $\mathcal{B}(\mathcal{S}, l)$ for Example 1. By using boundary sets, we infer strings in $\mathcal{T}$ as substrings of given strings that do not cross over any boundaries of MCS's. It is these substrings that are identified by our method to infer strings in $\mathcal{T}$.

Definition 4 (DCS) A string $e \in \Sigma^+$ is *a disjoint common substring (DCS) for $\mathcal{S}$ and l*, if e is a substring of some $S_i \in \mathcal{S}$ and has the following properties:

(D1)$|e| \geq l$.

(D2)For any $(i, j) \in Occ(e)$ and any integer k such that $1 \leq k < |e|$, $(i, j + k) \notin \mathcal{B}(\mathcal{S}, l)$.

(D3)$\mathcal{B}(\mathcal{S}, l) \cap (Occ(e) + |e|) \neq \emptyset$.

(D4)$\mathcal{B}(\mathcal{S}, l) \cap Occ(e) \neq \emptyset$.

We denote by DCS($\mathcal{S}, l$) the set of all DCS's for $\mathcal{S}$ and l. □

Intuitively, (D2) means that e does not contain any boundaries of MCS's in its middle. In addition, (D3) and (D4) mean that e is maximal among those that

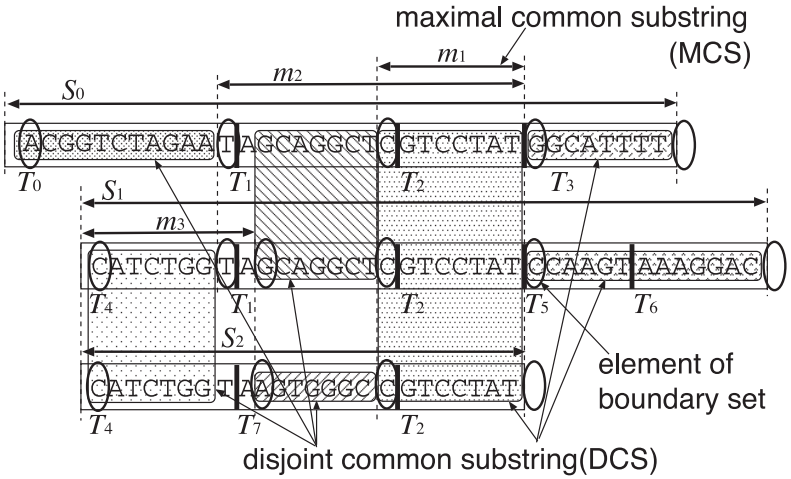

Fig. 3 DCS$(\mathcal{S}, l)$ for Example 1. DCS's, indicated by hatched areas, do not cross over any boundaries of m_1, m_2, and m_3. All of $T_0, \ldots, T_7$ except T_5 and T_6 are captured almost as a whole. There are unavoidable ambiguities at their boundaries.

satisfy (D2). In **Fig. 3**, we show DCS$(\mathcal{S}, l)$ for Example 1.

The choice of the parameter l is important. When a random string of length L in which each symbol is independently chosen from Σ with the same probability is given, the expected number of identical pairs of substrings of length l is no more than $E = L^2/(2|\Sigma|^l)$. To avoid occasional matches, it is recommended to set l so that E is sufficiently small. The value of l can also be easily determined by try and error, because our algorithm that identifies DCS$(\mathcal{S}, l)$ runs fast as shown later and because l is the only parameter.

Note that there are limitations as long as we are given only $\mathcal{S}$. Some ambiguities are unavoidable near boundaries of strings in $\mathcal{T}$. Besides, strings in $\mathcal{T}$ that always occur adjacent in the same order in $\mathcal{S}$ would be fused.

4. Algorithm that Identifies DCS's

Since a DCS is not always a path-label or $p(v)$ of a node v in STree$(\mathcal{S})$, STree$(\mathcal{S})$ cannot be directly used to identify DCS$(\mathcal{S}, l)$. We introduce a class of strings that bridge them.

Definition 5 Let $H(\mathcal{S}, l)$ be a set of strings such that any $h \in H(\mathcal{S}, l)$ has the following properties:
(H1)$h \in$ RightMCS$(\mathcal{S}, l)$.
(H2)For any proper prefix s of h, $s \notin$ RightMCS$(\mathcal{S}, l)$.
(H3)$\mathcal{B}(\mathcal{S}, l) \cap Occ(h) \neq \emptyset$. □

Lemma 1 Any $e \in$ DCS$(\mathcal{S}, l)$ is a prefix of some $h \in H(\mathcal{S}, l)$ such that $Occ(e) = Occ(h)$.

Proof 1 Let $s \in \Sigma^*$ be the longest such that $Occ(e) = Occ(es)$. We prove that $es \in H(\mathcal{S}, l)$, claiming that es satisfies (H1)–(H3).
(H1)Since $|es| \geq |e| \geq l$, es satisfies (M1). Because s is the longest such that $Occ(e) = Occ(es)$, es satisfies (M2). Therefore $es \in$ RightMCS$(\mathcal{S}, l)$, and thus es satisfies (H1).
(H2)The proof is by contradiction. Suppose that some $r \in$ RightMCS$(\mathcal{S}, l)$ is a proper prefix of es. If $|e| \leq |r|$, e is a prefix of r. Then, $Occ(e) \supseteq Occ(r) \supseteq Occ(es)$. Since r satisfies (M2), $Occ(r) \neq Occ(es)$. This implies $Occ(e) \neq Occ(es)$, which contradicts the definition of s. Therefore $|r| < |e|$, implying that r is a proper prefix of e. Since $Occ(r) \supseteq Occ(e)$, for some $(i, j) \in Occ(e)$, $(i, j + |r|) \in Occ(r) + |r|$. Because $Occ(r) + |r| = Occ(m) + |m|$ for some $m \in$ MCS$(\mathcal{S}, l)$, $(i, j + |r|) \in \mathcal{B}(\mathcal{S}, l)$. Therefore e does not satisfy (D2), which contradicts the definition of e. Consequently, such r cannot exist. Hence es satisfies (H2).
(H3)Since e satisfies (D4), $\mathcal{B}(\mathcal{S}, l) \cap Occ(es) = \mathcal{B}(\mathcal{S}, l) \cap Occ(e) \neq \emptyset$. Thus es satisfies (H3).
The string es is h claimed in the lemma. □

By Lemma 1, the definition of DCS$(\mathcal{S}, l)$ can be transformed as follows.

Lemma 2 For any non-empty substring e of some $S_i \in \mathcal{S}$, $e \in$ DCS$(\mathcal{S}, l)$ if and only if e satisfies (D1)–(D3) and the following condition:
(D5) For some $h \in H(\mathcal{S}, l)$, e is a prefix of h.
In other words, (D4) can be replaced with (D5).

Proof 2 By Lemma 1, any $e \in$ DCS$(\mathcal{S}, l)$ satisfies (D5). For the converse, let e be a non-empty substring of some $S_i \in \mathcal{S}$ that satisfies (D1)–(D3) and (D5). By the condition (D5), e is a prefix of some $h \in H(\mathcal{S}, l)$. Then, $Occ(e) \supseteq Occ(h)$. Since h satisfies (H3), $Occ(e) \cap \mathcal{B}(\mathcal{S}, l) \supseteq Occ(h) \cap \mathcal{B}(\mathcal{S}, l) \neq \emptyset$. Accordingly, e

satisfies (D4) and therefore $e \in \mathrm{DCS}(\mathcal{S}, l)$. □

Lemma 2 enables us to identify $\mathrm{DCS}(\mathcal{S}, l)$ by the following algorithm GET-DCS$(\mathcal{S}, l)$.

Algorithm: GET-DCS$(\mathcal{S}, l)$	
Step 1:	Construct STree$(\mathcal{S})$.
Step 2:	Identify RightMCS$(\mathcal{S}, l)$.
Step 3:	Identify MCS$(\mathcal{S}, l)$.
Step 4:	Identify $\mathcal{B}(\mathcal{S}, l)$.
Step 5:	Identify $H(\mathcal{S}, l)$.
Step 6:	Identify DCS$(\mathcal{S}, l)$.

As mentioned in Section 2, Steps 1–3 can be completed in $O(L)$-time [6]. Below we explain the other steps.

Step 4: Identify $\mathcal{B}(\mathcal{S}, l)$

Clearly, $\mathcal{B}_L(\mathcal{S}, l)$ can be identified by a depth-first traversal on STree$(\mathcal{S})$ in $O(L)$-time. Let us focus on $\mathcal{B}_R(\mathcal{S}, l)$. After initializing a set B to $\emptyset$, a depth-first traversal on STree$(\mathcal{S})$ is conducted. For any $\mathcal{L}(i, j)$ encountered, we add $(i, j+l)$ to B if there exists a node v such that $p(v) \in \mathrm{RightMCS}(\mathcal{S}, l)$ and $|p(v)| = l$ on the path from the root to $\mathcal{L}(i, j)$. When the depth-first traversal is completed, $B = \mathcal{B}_R(\mathcal{S}, l)$.

We show that this method correctly identifies $\mathcal{B}_R(\mathcal{S}, l)$. If (i, j) is added to B, $(i, j) \in Occ(r) + |r|$ for some $r \in \mathrm{RightMCS}(\mathcal{S}, l)$. Since some $m \in \mathrm{MCS}(\mathcal{S}, l)$ exists such that $Occ(r) + |r| = Occ(m) + |m|$, $(i, j) \in \mathcal{B}_R(\mathcal{S}, l)$. For the converse, suppose that $(i, j) \in \mathcal{B}_R(\mathcal{S}, l)$. Then, by the following lemma, $(i, j - l) \in Occ(r)$ for some $r \in \mathrm{RightMCS}(\mathcal{S}, l)$ such that $|r| = l$.

Lemma 3 Let r be a suffix of some $m \in \mathrm{MCS}(\mathcal{S}, l)$ such that $|r| = l$. Then, $r \in \mathrm{RightMCS}(\mathcal{S}, l)$.

Proof 3 Since $|r| = l$, r satisfies (M1). We prove that r satisfies (M2). By contradiction, suppose that $Occ(r) = Occ(rs)$ for a string $s \in \Sigma^+$. Since r is a suffix of m wherever m occurs, $Occ(m) = Occ(ms)$. Therefore m does not satisfy (M2), which contradicts the assumption that $m \in \mathrm{MCS}(\mathcal{S}, l)$. Accordingly s cannot exist, hence r satisfies (M2). Consequently, $r \in \mathrm{RightMCS}(\mathcal{S}, l)$. □

Algorithm: PREFIX-DCS$(\mathcal{S}, l)$

```
for i := 0 to N − 1 begin
  x := 1, j := |S_i| − 1
  repeat
    if P[i, j] → λ(h) then λ(h) := min{x, λ(h)}
    x := x + 1
    if (i, j) ∈ B(S, l) then x := 1
    j := j − 1
  until j < 0
end
```

Fig. 4 The algorithm PREFIX-DCS$(\mathcal{S}, l)$ executed in Step 6 of GET-DCS$(\mathcal{S}, l)$. This algorithm scans each $S_i \in \mathcal{S}$ from right to left by decreasing j one by one. When $\min\{x, \lambda(h)\}$ is evaluated in the inner-most loop, x is the length of the longest string s that occurs at (i, j) and satisfies $(i, j + k) \notin \mathcal{B}(\mathcal{S}, l)$ for $1 \le k < |s|$.

Step 5: Identify $H(\mathcal{S}, l)$

We identify $H(\mathcal{S}, l)$ by discarding any $p(v) \in \mathrm{RightMCS}(\mathcal{S}, l)$ from RightMCS$(\mathcal{S}, l)$ if $p(v)$ does not satisfy any one of (H2) or (H3), where v is a node in STree$(\mathcal{S})$. This can be done in $O(L)$-time by a depth-first traversal on STree$(\mathcal{S})$.

Step 6: Identify DCS$(\mathcal{S}, l)$

By Lemma 2, $\mathrm{DCS}(\mathcal{S}, l)$ can be obtained by searching for substrings that satisfy (D1)–(D3) and (D5). To avoid exhaustive search, we use variables $\lambda(h)$ provided for each $h \in H(\mathcal{S}, l)$, and a pointer table defined below.

Definition 6 (Pointer table) $P[i, j]$ $(0 \le i < N, 0 \le j < |S_i|)$ is a pointer that satisfies the following properties:

- $P[i, j] \to \lambda(h)$ if $(i, j) \in Occ(h)$ for some $h \in H(\mathcal{S}, l)$, where $P[i, j] \to \lambda(h)$ means $P[i, j]$ points to $\lambda(h)$,
- $P[i, j]$ is a null pointer otherwise. □

After each $P[i, j]$ is initialized to a null pointer, a depth-first traversal on STree$(\mathcal{S})$ is conducted. For each $\mathcal{L}(i, j)$, $P[i, j]$ is set so that $P[i, j] \to \lambda(p(v))$ if there is a node v such that $p(v) \in H(\mathcal{S}, l)$ on the path from the root to $\mathcal{L}(i, j)$.

After the pointer table is built, variables $\lambda(h)$ are initialized to $|h|$ for each $h \in H(\mathcal{S}, l)$. Then, the algorithm PREFIX-DCS$(\mathcal{S}, l)$ in **Fig. 4** is applied. For any $h \in H(\mathcal{S}, l)$, PREFIX-DCS$(\mathcal{S}, l)$ sets variables $\lambda(h)$ to the lengths of prefixes of h that satisfy (D2) and (D3). We show the behavior of PREFIX-DCS$(\mathcal{S}, l)$ for strings of Example 1 in **Fig. 5**. When PREFIX-DCS$(\mathcal{S}, l)$ is completed, for each

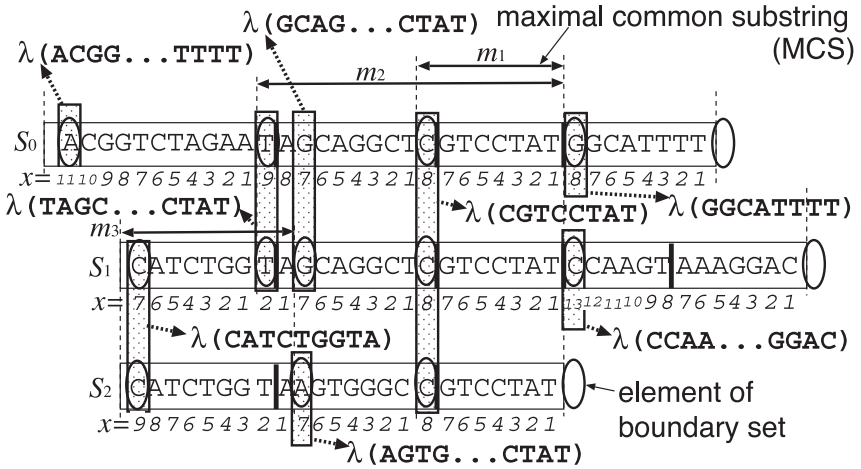

Fig. 5 Behavior of the algorithm PREFIX-DCS($\mathcal{S}, l$) for strings of Example 1. Digits below S_0, S_1 and S_2 indicate the values of x when $\min\{x, \lambda(h)\}$ is evaluated. Hatched areas indicate positions where $P[i, j]$ is not a null pointer.

$h \in H(\mathcal{S}, l)$, the prefix of h whose length is $\lambda(h)$ is a DCS if $\lambda(h) \geq l$. All Steps 1–6 can be completed in $O(L)$-time. Therefore,

Theorem 1 There is an algorithm that identifies DCS($\mathcal{S}, l$) in $O(L)$-time, where L is the sum of the lengths of all strings in $\mathcal{S}$.

5. Computational Experiments

We evaluated GET-DCS($\mathcal{S}, l$) by computational experiments. Here are definitions of terms used in evaluation. We say $e \in$ DCS($\mathcal{S}, l$) is *consistent* with a string $t \in \mathcal{T}$ if and only if $|Occ(e)| = |Occ(t)|$ and the overlap of e and t occupies at least 90% of both e and t wherever e or t occurs. Let n_{OK} be the number of strings in DCS($\mathcal{S}, l$) consistent with some $t \in \mathcal{T}$, and n_{NG} be $|\text{DCS}(\mathcal{S}, l)| - n_{OK}$. Below *recall* means $n_{OK}/|\mathcal{T}|$, while *precision* means $n_{OK}/|\text{DCS}(\mathcal{S}, l)|$. We used a Linux server with Opteron(tm) 252 processors and 14 GB of RAM.

5.1 Randomly generated strings

First, we tested GET-DCS($\mathcal{S}, l$) against randomly generated strings. We constructed a set $\mathcal{T}_0$ of strings by generating 100,000 random strings that consisted of `A`, `T`, `G`, and `C`. Their lengths were 50–240 bases and 145 bases on average. Then we constructed a set $\mathcal{S}$ of 40,000 strings, each of which was a concatenation of nine strings randomly chosen from $\mathcal{T}_0$. These parameters were determined to simulate the scale of coding sequences of *Homo sapiens* estimated with draft genomic sequences [10]. However, we used small $\mathcal{T}_0$ to see the ability of GET-DCS($\mathcal{S}, l$) to detect strings in $\mathcal{T}_0$ when they have enough chances to occur with different strings in $\mathcal{T}_0$. The lengths of strings in $\mathcal{S}$ were 1,305 bases on average, and 5.218×10^7 bases in total. We set l to 30. DCS's were evaluated against $\mathcal{T}_1 \subseteq \mathcal{T}_0$ that consisted of strings in $\mathcal{T}_0$ chosen at least one time to construct $\mathcal{S}$.

Table 1 Summary statistics of DCS's and a set $\mathcal{T}_1$ of randomly generated strings.

	DCS's	$\mathcal{T}_1$
number of strings	96,955	97,217
average length (bases)	142.3	144.9
variance of lengths	3,102	3,038
average occurrences	3.71	3.70
maximum occurrences	15	15

Table 2 Consistency of DCS's against $\mathcal{T}_1$.

n_{OK}	n_{NG}	recall	precision
96,198	757	0.9895	0.9922

As shown in **Table 1**, the summary statistics of DCS($\mathcal{S}, l$) and $\mathcal{T}_1$ were very close. In fact, most of the DCS's were consistent with strings in $\mathcal{T}_1$ (**Table 2**). We examined DCS's that were not consistent with any strings in $\mathcal{T}_1$. There were 258 groups of strings in $\mathcal{T}_1$ that always occurred adjacent in $\mathcal{S}$ in the same order. In addition, there were DCS's shortened to less than 90% of corresponding strings in $\mathcal{T}_1$ because of unavoidable ambiguity mentioned in Section 3. We confirmed that all DCS's except two short ones became consistent with strings in $\mathcal{T}_1$ if we merged each group of strings in $\mathcal{T}_1$ that always occurred adjacent in the same order and relaxed the threshold of consistency to 80%.

To demonstrate the scalability of our implementation of GET-DCS($\mathcal{S}, l$), we measured the increase in computation time while the number of given strings was increased. For each of $N \in \{1000i | 1 \leq i \leq 40\}$, we executed GET-DCS($\mathcal{S}, l$) against the first N strings of $\mathcal{S}$. As shown in **Fig. 6**, the computation time

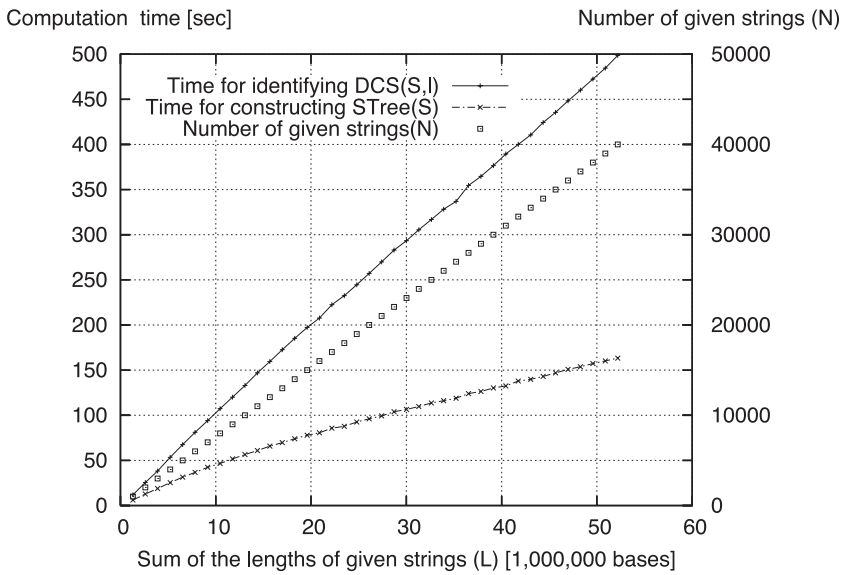

Fig. 6 Increase in computation time to identify DCS($\mathcal{S}, l$) from randomly generated strings while the number of given strings was increased. The computation time increased only linearly.

increased only linearly, which indicates that the implementation was scalable to the number of given strings.

5.2 Transcriptome of *Homo sapiens*

Next, we present a preliminary experiment to identify all alternative splicing of an organism by applying GET-DCS($\mathcal{S}, l$) to its transcriptome. We tested our method against all cDNA sequences of *Homo sapiens* in the RefSeq database [18] of release 28. Although sequence differences such as sequencing errors were reconciled to finished genomic sequences in this database [19], they still contain plenty of complex features of real cDNA sequences. We removed consecutive `A`'s at the end of each sequence to exclude poly(A) tails. Then, the set of obtained sequences were used as $\mathcal{S}$ in this experiment. There were 25,199 sequences, whose lengths were 3,050 bases on average and 7.686×10^7 bases in total. It took 826 seconds for GET-DCS($\mathcal{S}, l$) to identify DCS($\mathcal{S}, l$) from $\mathcal{S}$, where l was set to 30.

For 23,777 sequences in $\mathcal{S}$, positions of substrings corresponding to exons and alternative ends of exons were available in the data of NCBI Map Viewer ⋆1 based on Build36.3 of genomic sequences. After merging substrings corresponding to exons which always occurred adjacent together in $\mathcal{S}$, a set of the substrings, denoted by $\mathcal{T}_2$, was used for evaluation of DCS's in the 23,777 sequences. In this experiment, alternative ends of exons were treated in the same way as independent exons.

Table 3 Summary statistics of DCS's and a set $\mathcal{T}_2$ of substrings obtained by splitting all RefSeq cDNA sequences of *Homo sapiens* at positions of alternative splicing observed in the cDNA sequences.

	DCS's	$\mathcal{T}_2$
number of strings	51,811	29,833
average length (bases)	1,004	1,813
variance of lengths	2.38×10^6	4.27×10^6
average occurrences	1.70	1.57
maximum occurrences	100	24

Table 4 Consistency of DCS's with $\mathcal{T}_2$. (A) DCS's were evaluated as they were. (B) DCS's that always occurred adjacent in the same order were merged. (C) DCS's that occurred more than one time in a single string in $\mathcal{S}$ were removed. Then, DCS's that always occurred adjacent in the same order were merged.

	n_{OK}	n_{NG}	recall	precision
(A)	21,803	30,008	0.7308	0.4208
(B)	23,567	15,579	0.7900	0.6020
(C)	23,798	14,110	0.7977	0.6278

As shown in **Table 3**, DCS's were much shorter than strings in $\mathcal{T}_2$ on average, while the number of DCS's was much larger than $|\mathcal{T}_2|$. As shown in row (A) of **Table 4**, precision was poor. There were at least following three problems. First, some strings in $\mathcal{T}_2$ were divided into multiple DCS's because of sequence variations such as SNPs. Second, repeated elements were wrongly identified as DCS's. For example, sequence NM_002457.2 of the mucin 2 (MUC2) gene had a tandem repeat consisting of 100 copies of a 69-base sequence, which became a DCS that had the largest number of occurrences. Third, family genes sometimes shared long identical regions irrelevant to alternative splicing, which became DCS's.

We tried two extensions for GET-DCS($\mathcal{S}, l$) to obtain better results. First,

⋆1 `http://www.ncbi.nlm.nih.gov/mapview/`

to partly circumvent the first problem, we merged DCS's that always occurred adjacent in the same order. As shown in row (B) of Table 4, precision was substantially improved. Recall also became higher. Next, we removed DCS's that occurred at least twice in a sequence to reduce the effect of the second problem. Combined with the first extension, recall and precision were further improved as shown in row (C) of Table 4. One direction to overcome the third problem is to combine information of more than one DCS. When two sequences in $\mathcal{S}$ share a DCS, investigating whether they share other DCS's is a way to discriminate DCS's irrelevant to alternative splicing. Another way to screen out erroneous DCS's is to examine whether the order of DCS's is preserved in more than one sequence in $\mathcal{S}$.

5.3 Comparison with MSA Programs

We compared our method with MSA programs. From a large number of MSA programs available, we chose POA [11], DIALIGN2 [15] and L-INS-i of the MAFFT package [9], which use local alignment strategies. Since their output is not a set of substrings of given sequences but an alignment, we converted the alignment into a set of substrings as follows.

(A1) Find intervals in the alignment constantly shared by the same subset of given sequences.
(A2) Discard intervals that contain mismatches or are shorter than 5 bases.
(A3) Extract substrings indicated by the intervals.

The data sets so far are not appropriate here, because they are too large for MSA programs and are a mixture of sequences that should not be aligned. Therefore we extracted sequences of a single gene from $\mathcal{S}$ of the previous experiment. We chose the cAMP-responsive element modulator (CREM) gene, because it had the largest number of sequences in the data set. There were 21 sequences of the gene. Their lengths were 1,978 bases on average and 41,535 bases in total. The positions of exons were obtained again from the data of NCBI Map Viewer. In these sequences, all exons occurred as identical substrings at all their occurrences. The exons 8, 11 and 14 had alternative ends, which were treated as independent exons. For example, exon 8 was divided into $8'$ and $8''$.

With default parameters, none of POA, DIALIGN2, and L-INS-i produced satisfactory results. Although it seems quite easy to reveal that given sequences

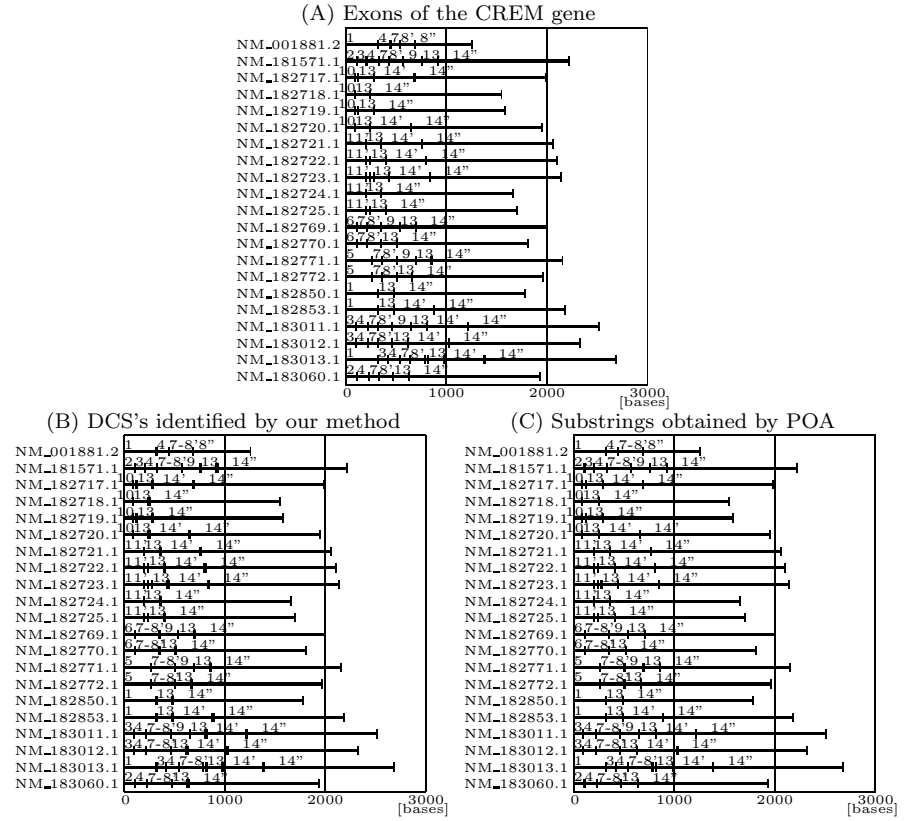

Fig. 7 (A) Exons of the CREM gene, (B) DCS's identified by GET-DCS($\mathcal{S}, l$), and (C) substrings obtained by POA. Numbers over substrings of sequences indicate corresponding exons. Since there are no spaces, numbers $11''$ and 12 are not shown.

are merely concatenations of exon sequences, the MSA programs suffered from weak similarities between different exons since they are tuned so that they can cope with distantly related sequences. We tried to tune their parameters. We successfully get POA to produce substrings consistent with almost all exons by setting the mismatch parameter to a huge negative value (-10^6), setting the gap open penalty to 45, and using guide trees. However, we could not obtain substrings consistent with more than four exons by DIALIGN2 and any better

Table 5 Lengths of exons of the CREM gene, and lengths of their overlaps with DCS's and substrings obtained by POA. Lengths of extra bases not in exons are in parentheses.

exon	length of exons	overlap with DCS's	overlap with POA substrings
1	321	320(0)	321(0)
2	108	108(0)	108(0)
3	98	97(0)	98(0)
4	124	122(0)	122(0)
5	265	263(0)	263(0)
6	110	109(0)	109(0)
7–8′	98+143	241(1)	241(1)
8″	571	570(0)	570(0)
9	189	187(0)	188(0)
10	88	86(0)	87(0)
11′	198	196(0)	198(0)
11″	43	40(0)	42(0)
12	36	33(0)	35(0)
13	157	154(0)	157(10)
14′	402	389(0)	392(10)
14″	1302	1292(0)	1292(0)

result by L-INS-i. Therefore, below we show only the result of POA. We set l to 20 for GET-DCS$(\mathcal{S}, l)$.

We show a schematic illustration of exons of the CREM gene and the results of our method and POA in **Fig. 7**. As shown in **Table 5**, substrings obtained by our method and POA were almost identical to all exons except that exons 7 and 8′ were fused since they always occurred together. Both methods wrongly dropped 10 bases at 5'-end of exons 14′ and 14″. This was caused by their identical 10-base prefixes. POA added the 10 bases to the 3'-ends of exons 13 and 14′, while our method excluded these 10 bases from any DCS since there was ambiguity.

It took only 0.035 seconds for our method to obtain the result, while it took 97.797 seconds for POA. Our method were about 2800 times faster. Besides, to find parameters that enabled POA to produce satisfactory results, POA had to be repeatedly executed.

6. Conclusions

We considered the problem of inferring a set $\mathcal{T}$ of hidden strings from a set $\mathcal{S}$ of their concatenations. We formalized this problem by defining a set DCS$(\mathcal{S}, l)$ of strings each of which corresponds to a string in $\mathcal{T}$ or a concatenation of strings in $\mathcal{T}$ that always occur adjacent in the same order, where l is a positive integer parameter. We proved that DCS$(\mathcal{S}, l)$ can be identified within $O(L)$-time by comparing strings in $\mathcal{S}$, where L is the sum of the lengths of all strings in $\mathcal{S}$. With our method, a set of 40,000 randomly generated strings were successfully decomposed into substrings of which they are concatenations. The RefSeq sequences of the human CREM gene were also decomposed into exons only with minor errors by our method about 2800 times faster than by a multiple sequence alignment (MSA) program POA, while other MSA programs suffered from weak similarities between different exons. If there is possibility that given sequences are concatenations of an unknown set of strings, it is worth trying our method to identify such set of strings. For real large data set of cDNA sequences, we have to enhance our method to cope with various features of cDNA sequences, including sequence variations such as SNPs, repeated elements, and identical regions of sequences of different genes.

Acknowledgments The author thanks Prof. Satoru Miyano of the Institute of Medical Science, the University of Tokyo, for his very helpful advice during the composition of this paper. The author also thanks Assoc. Prof. Seiya Imoto of the Institute of Medical Science, the University of Tokyo, for his kind advice and discussion. This work was partly supported by the New Energy and Industrial Technology Development Organization (NEDO), Japan.

References

1) Akutsu, T., Arimura, H. and Shimozono, S.: Hardness results on local multiple alignment of biological sequences, *IPSJ Trans. Bioinformatics*, Vol.48, No.SIG 5(TBIO 2), pp.30–38 (2007).
2) Blackshields, G., Wallace, I., Larkin, M. and Higgins, D.: Analysis and comparison of benchmarks for multiple sequence alignment, *In Silico Biology*, Vol.6, No.4, pp.321–339 (2006).
3) Chen, X., Zheng, J., Fu, Z., Nan, P., Zhong, Y., Lonardi, S. and Jiang, T.: Computing the assignment of orthologous genes via genome rearrangement, *Proc. 3rd Asia-Pacific Bioinformatics Conference* (*APBC*), pp.363–378 (2005).
4) Chrobak, M., Kolman, P. and Sgall, J.: The greedy algorithm for the minimum common string partition problem, *ACM Trans. Algorithms* (*TALG*), Vol.1, No.2, pp.350–366 (2005).

5) Goldstein, A., Kolman, P. and Zheng, J.: Minimum common string partition problem: hardness and approximations, *Proc. 15th International Symp. Algorithms and Computation* (*ISAAC*), pp.473–484 (2004).
6) Gusfield, D.: *Algorithms on strings, trees, and sequences*, Cambridge University Press, New York (1997).
7) Imanishi, T. et al.: Integrative annotation of 21,037 human genes validated by full-length cDNA clones, *PLoS Biology*, Vol.2, No.6, pp.856–875 (2004).
8) Johnson, J., Castle, J., Garrett-Engele, P., Kan, Z., Loerch, P., Armour, C., Santos, R., Schadt, E., Stoughton, R. and Shoemaker, D.: Genome-wide survey of human alternative pre-mRNA splicing with exon junction microarrays, *Science*, Vol.302, No.5653, pp.2141–2144 (2003).
9) Katoh, K., Kuma, K., Toh, H. and Miyata, T.: MAFFT version 5: improvement in accuracy of multiple sequence alignment, *Nucleic Acids Research*, Vol.33, No.2, pp.511–518 (2005).
10) Lander, E. et al.: Initial sequencing and analysis of the human genome, *Nature*, Vol.409, pp.860–921 (2001).
11) Lee, C., Grasso, C. and Sharlow, M.: Multiple sequence alignment using partial order graphs, *Bioinformatics*, Vol.18, No.3, pp.452–464 (2002).
12) Leung, M., Blaisdell, B., Burge, C. and Karlin, S.: An efficient algorithm for identifying matches with errors in multiple long molecular sequences, *J. Molecular Biology*, Vol.221, No.4, pp.1367–1378 (1991).
13) Lopresti, D. and Tomkins, A.: Block edit models for approximate string matching, *Theoretical Computer Science*, Vol.181, No.1, pp.159–179 (1997).
14) Maniatis, T. and Tasic, B.: Alternative pre-mRNA splicing and proteome expansion in metazoans, *Nature*, Vol.418, pp.236–243 (2002).
15) Morgenstern, B.: DIALIGN 2: improvement of the segment-to-segment approach to multiple sequence alignment, *Bioinformatics*, Vol.15, No.3, pp.211–218 (1999).
16) Needleman, S. and Wunsch, C.: A general method applicable to the search for similarities in the amino acid sequence of two proteins, *J. Molecular Biology*, Vol.48, No.3, pp.443–453 (1970).
17) Néraud, J.: Elementariness of a finite set of words is co-NP-complete, *Theoretical Informatics and Applications*, Vol.24, No.5, pp.459–470 (1990).
18) Pruitt, K., Tatusova, T. and Maglott, D.: NCBI reference sequences (RefSeq): a curated non-redundant sequence database of genomes, transcripts and proteins, *Nucleic Acids Research*, Vol.35, pp.D61–D65 (2007).
19) Pruitt, K.D., Tatusova, T. and Maglott D.R.: The Reference Sequence (RefSeq) Project, *The NCBI Handbook*, NCBI, chapter18 (2002).
20) Smith, T. and Waterman, M.: Identification of common molecular subsequences, *J. Molecular Biology*, Vol.147, pp.195–197 (1981).
21) Wang, L. and Jiang, T.: On the complexity of multiple sequence alignment, *J. Compututational Biology*, Vol.1, No.4, pp.337–348 (1994).

(Received April 1, 2008)
(Accepted May 28, 2008)
(Released November 28, 2008)

(Communicated by *Tetsuo Shibuya*)

Tomohiro Yasuda received his M.Sc. degree in Information Science from the University of Tokyo, Japan, in 2000. After joining Hitachi, he worked on developing systems that support sequencing of cDNAs. He is currently in charge of developing a system that searches huge text databases. He is a member of IPSJ and JSBi.