Improved Exact Enumerative Algorithms for the Planted (l, d)-Motif Search Problem

Shunji Tanaka, *Member, IEEE*

***Abstract*—In this paper efficient exact algorithms are proposed for the planted (l, d)-motif search problem. This problem is to find all motifs of length l that are planted in each input string with at most d mismatches. The "quorum" version of this problem is also treated in this paper to find motifs planted not in all input strings but in at least q input strings. The proposed algorithms are based on the previous algorithms called qPMSPruneI and qPMS7 that traverse a search tree starting from a l-length substring of an input string. To improve these previous algorithms, several techniques are introduced, which contribute to reducing the computation time for the traversal. In computational experiments, it will be shown that the proposed algorithms outperform the previous algorithms.**

***Index Terms*—Planted (l, d)-motif search problem, closest substring problem, exact enumerative algorithm, tree search**

I. Introduction

This study will propose efficient exact enumerative algorithms for the planted (l, d)-motif search problem. This problem is to extract common substrings that appear in every input string with some mismatches allowed. Formally, it is described as follows. Let Σ be an alphabet (a set of letters) and s_i ($1 \leq i \leq N$) be input strings of length L over Σ, i.e., $s_i \in \Sigma^L$. The planted (l, d)-motif search problem is to find all the strings $t \in \Sigma^l$ such that for all i, there exists an l-length substring o_i of s_i satisfying $d_{\mathrm{H}}(t, o_i) \leq d$, where $d_{\mathrm{H}}(x, y)$ denotes the Hamming distance between the two strings x and y. This problem is known to be NP-hard [1]. In this study the "quorum" version of the problem, which is referred to as the planted (l, d, q)-motif search problem, is also treated: the problem to find all the strings t such that there exists an l-length substring o_i of s_i satisfying $d_{\mathrm{H}}(t, o_i) \leq d$ for at least q input strings s_i. Hereafter, t and o_i are called a motif and an occurrence, respectively.

The planted (l, d)-motif search problem is also referred to as the closest substring problem in the literature. Studies on the closest substring problem are primarily in the field of computer science, which focus on theoretical worst-case computational complexity such as PTAS (polynomial-time approximation scheme), parameterized algorithms, and so on [2]–[8]. On the other hand, the purpose of studies on the planted (l, d)-motif search problem is to construct practical algorithms applicable to motif finding in DNA and protein sequences. This paper follows the latter line of research. The primary purpose of this study is to construct practically efficient algorithms for the planted (l, d)-motif search problem and the planted (l, d, q)-motif search problem.

S. Tanaka is with the Institute for Liberal Arts and Sciences, and with the Department of Electrical Engineering, Kyoto University, Kyotodaigaku-Katsura, Nishikyo-ku, Kyoto 615-8510, Japan. e-mail: tanaka@kuee.kyoto-u.ac.jp.

Various methods have been proposed so far to solve the planted (l, d)-motif search problem heuristically or exactly. Among them, enumerative methods can be roughly categorized into four approaches. The first approach searches for the occurrences o_i, in place of the motif t itself, that satisfy $d_{\mathrm{H}}(o_i, o_j) \leq 2d$ for any $1 \leq i, j \leq N$, $i \neq j$. For this purpose, a graph is constructed by assigning a vertex to every substring of length l in the input strings. A pair of vertices is connected by an edge when they represent substrings in different input strings and when the Hamming distance between the substrings is less than or equal to $2d$. Then, cliques of size N are searched for in this graph. The existing algorithms in this category, which are sometimes referred to as the sample-driven approach, are: WINNOWER [9], cWINNOWER [10], the algorithm in [11], DPCFG [12], RecMotif [13], ListMotif [14], and TreeMotif [15].

The second approach searches for the motifs directly by extending the length of the motifs from zero to l. Namely, a trie of depth l is traversed where a node at depth k represents a k-length prefix of the motif. The existing algorithms in this category are SPELLER [16], WEEDER [17], MITRA [18], CENSUS [19], and RISOTTO [20].

The third approach first enumerates candidate motifs, and then searches them for feasible ones. The existing algorithms in this category are Voting Algorithm [21], PMS1 [22], PMS2 [22], PMS3 [22], the algorithm in [23], PMSi [24], PMSP [24], stemming [25], PMS4 [26], PMS5 [27], PMS6 [28], and PairMotif [29]. These algorithms differ greatly from each other on how to generate the candidate motifs. For example, Voting Algorithm considers all the strings of length l at first, while the other algorithms restrict the initial candidate set by the information of input strings.

The last approach searches for the motifs by choosing a candidate from an input string and then modifying its letters one by one. Since the Hamming distance between an occurrence and a motif should be at most d, a search tree of depth d is traversed where each node represents a candidate motif. The existing algorithms in this category are PMSPrune [30], Pampa [31], PMS3p [32], Provable [8], qPMSPruneI [33], and qPMS7 [33].

Among the above-mentioned algorithms, the most efficient exact algorithms would be qPMSPruneI and qPMS7 in [33] when d/l is large. In [33], it was shown by computational experiments for "challenging" DNA and protein instances that qPMSPruneI performs well for DNA sequences when $l \leq 15$ and qPMS7 for the other cases. To the best of the author's knowledge, qPMS7 is currently the only algorithm that can solve DNA instances with $N = 20$, $L = 600$ and $(l, d) = (23, 9)$ within 1 day or so. qPMSPruneI and qPMS7 are not the best from the viewpoint of computational complexities. However, it does not follow that an algorithm with a less computational complexity works well for randomly generated instances. Indeed, Provable [8], which was proposed to improve the computational complexity, was shown to be at most competitive with PMSPrune, an older and thus much slower version of qPMSPruneI.

The purpose of this study is to improve qPMSPruneI and qPMS7. The key observations are as follows:

1) The algorithms work more efficiently by reducing the size of search trees.
2) It is also important to reduce the computation time necessary for checking whether subtrees in a search tree can be pruned or not.
3) The root node of a search tree is an l-length substring of an input string, and search trees are traversed for every substring. If two root substrings are similar, the corresponding search trees will also be similar.

By noting these, several techniques will be introduced. In computational experiments, it will be exhibited that the improved algorithms outperform qPMSPruneI and qPMS7, and that challenging DNA instances with $N = 20$, $L = 600$ and $(l, d) = (25, 10)$ become solvable for the first time in 15 hours or so on a desktop computer (single-threaded).

The rest of this paper is organized as follows. In Section II, notations and definitions will be introduced. In Section III, qPMSPruneI and qPMS7 will be described briefly. Next, qPMSPruneI will be improved in Sections IV and V, and qPMS7 will be improved in Section VI. Then, in Section VII, the effectiveness of the proposed algorithms will be verified by computational experiments. Finally, Section VIII will summarize the results in this paper.

II. Notations and Definitions

First, the notations and definitions in this paper will be presented.

The input data of the problem are given in TABLE I. Throughout this paper, all strings are assumed to be over Σ. The notations and definitions are summarized in TABLE II. With regard to $R_i(a, b, c)$, $R_i(a, b, c) \cap R_k(a, b, c) = \emptyset$ holds for any $i \neq k$, and

$$|a| = |b| = |c| = \left| \bigcup_{i=1}^{5} R_i(a, b, c) \right|. \quad (1)$$

In addition, $d_{\mathrm{H}}(a, b)$, $d_{\mathrm{H}}(b, c)$, and $d_{\mathrm{H}}(c, a)$ are expressed by

$$d_{\mathrm{H}}(a, b) = |R_3(a, b, c)| + |R_4(a, b, c)| + |R_5(a, b, c)|, \quad (2)$$
$$d_{\mathrm{H}}(b, c) = |R_2(a, b, c)| + |R_3(a, b, c)| + |R_5(a, b, c)|, \quad (3)$$
$$d_{\mathrm{H}}(c, a) = |R_2(a, b, c)| + |R_4(a, b, c)| + |R_5(a, b, c)|. \quad (4)$$

III. Previous Algorithms

In this section, qPMSPruneI and qPMS7 proposed in [33] will be reviewed.

TABLE I
Input of the Problem

Σ :	The alphabet.
s_i :	The input string of length L over Σ $(1 \leq i \leq N)$.
l :	The motif length.
d :	The maximum number of mismatches allowed in each occurrences.
q :	The minimum number of input strings that should have at least one occurrence.

TABLE II
Notations and Definitions

$\|a\|$:	The length of a string a.
$a[j]$:	The jth letter of a string a.
$a \circ b$:	The string generated by concatenating two strings a and b.
$d_{\mathrm{H}}(a, b)$:	The Hamming distance between two strings a and b of the same length. It is given by the number of positions j such that $a[j] \neq b[j]$.
s_{ij}^{l} :	The l-length substring of an input string s_i that starts from the jth position.
$\mathcal{S}_i$:	The set of all the l-length substrings of an input string s_i. $\mathcal{S}_i = \{s_{i1}^{l}, \ldots, s_{i,L-l+1}^{l}\}$.
$\mathcal{S}_i^{\mathrm{d}}$:	The set of all the $(l+1)$-length substrings of an input string s_i defined by $\mathcal{S}_i^{\mathrm{d}} = \{s_{i0}^{l+1}, \ldots, s_{i,L-l+1}^{l+1}\}$. Here, $s_{i0}^{l+1}[1] = s_{i,L-l+1}^{l+1}[l+1] = \emptyset$, and $d_{\mathrm{H}}(\emptyset, \alpha) = \infty$ is assumed for any $\alpha \in \Sigma$.
$\mathcal{B}(a, R)$:	The set of strings in the sphere of radius R centered at a . $\mathcal{B}(a, R) = \{b \mid \|b\| = \|a\|,\ d_{\mathrm{H}}(a, b) \leq R\}$.
$n_{\mathrm{B}}(l, d)$:	$\|\mathcal{B}(a, d)\|$ for an l-length string a.
$x\|_P$:	The substring of x composed by sequencing the letters of x at the positions in a vector P. For example, $x\|_P =$ GCA for $x =$ ACCGAT and $P = (4, 2, 5)$.
$P(j)$:	The jth element of a vector P.
$P_+(j)$:	The j-dimensional vector composed of the first j elements of a vector P.
$P_-(j)$:	The $(\|P\| - j)$-dimensional vector composed of the last $\|P\| - j$ elements of a vector P.
I :	The vector of length l defined by $I = (1, \ldots, l)$.
$R_1(a, b, c)$:	The set of indices j satisfying $a[j] = b[j] = c[j]$.
$R_2(a, b, c)$:	The set of indices j satisfying $a[j] = b[j] \neq c[j]$.
$R_3(a, b, c)$:	The set of indices j satisfying $c[j] = a[j] \neq b[j]$.
$R_4(a, b, c)$:	The set of indices j satisfying $b[j] = c[j] \neq a[j]$.
$R_5(a, b, c)$:	The set of indices j satisfying $a[j] \neq b[j]$, $b[j] \neq c[j]$, and $c[j] \neq a[j]$.

A. qPMSPruneI

Although qPMSPruneI as well as qPMS7 is for the planted (l, d, q)-motif search problem, $q = N$ is assumed at first for ease of explanation. Suppose that $x_0 \in \mathcal{S}_1$ is an occurrence of a motif t. Then, $t \in \mathcal{B}(x_0, d)$ holds because $d_{\mathrm{H}}(t, x_0) \leq d$. In other words, $t \in \mathcal{B}(x_0, d)$ for some $x_0 \in \mathcal{S}_1$ if t is a motif. Therefore, qPMSPruneI searches for motifs by enumerating $\mathcal{B}(x_0, d)$ for every $x_0 \in \mathcal{S}_1$. Hereafter, x_0 is referred to as the root occurrence. Since letters in at most d positions are different between the root occurrence x_0 and any $y \in \mathcal{B}(x_0, d)$, y is uniquely expressed by a sequence of the pairs of a position and a letter as $(p_1, \alpha_1), \ldots, (p_{d'}, \alpha_{d'})$, where $d' = d_{\mathrm{H}}(x_0, y) \leq d$, $1 \leq p_1 < p_2 < \cdots < p_{d'} \leq l$, and $x_0[p_j] \neq \alpha_j$ for any $1 \leq j \leq d'$. In this case, y is given by

$$y[j] = \begin{cases} x_0[j] & \text{if } j \neq p_k,\ \forall k, \\ \alpha_k & \text{if } j = p_k. \end{cases} \quad (5)$$

By noting this, qPMSPruneI traverses a tree of depth d to enumerate all $y \in \mathcal{B}(x_0, d)$. In this tree, a node at depth k

corresponds to the kth pair (p_k, α_k) and thus can be regarded as representing the string x_k expressed by (p_1, α_1), …, (p_k, α_k). The string x_k differs from its parent string x_{k-1} only at position p_k. Similarly, a child string x_{k+1} differs from x_k only at position p_{k+1}.

To find motifs, whether x_k is a valid motif or not is checked at every node (at depth k). This is performed by checking whether there exists $y_i \in \mathcal{S}_i$ satisfying $d_{\mathrm{H}}(x_k, y_i) \le d$ for every i $(2 \le i \le N)$. The primary advantage in enumerating $\mathcal{B}(x_0, d)$ by the tree traversal is that it is not necessary to consider all the candidate occurrences in $\mathcal{S}_i$ $(2 \le i \le N)$. To see this, let us define $x_{01} = x_0|_{I_+(p_k)}$, $x_{02} = x_0|_{I_-(p_k)}$, $x_{k1} = x_k|_{I_+(p_k)}$, and $x_{k2} = x_k|_{I_-(p_k)} = x_{02}$. Let us also denote by $\mathcal{O}(x_k, d)$ the set of all the offspring of x_k (including x_k itself). Then,

$$\begin{aligned}\mathcal{O}(x_k, d) &= \{x'_k \in \mathcal{B}(x_0, d) \mid x'_k|_{I_+(p_k)} = x_{k1}\}\\ &= \{x_{k1} \circ x'_{k2} \mid x'_{k2} \in \mathcal{B}(x_{02}, d - d_{\mathrm{H}}(x_{k1}, x_{01}))\}.\end{aligned} \tag{6}$$

Hence, for some y to be an occurrence of an offspring of x_k,

$$\mathcal{O}(x_k, d) \cap \mathcal{B}(y, d) \neq \emptyset \tag{7}$$

should be satisfied. If we define $y_1 = y|_{I_+(p_k)}$ and $y_2 = y|_{I_-(p_k)}$, this condition can be rewritten as

$$\mathcal{B}(x_{02}, d - d_{\mathrm{H}}(x_{k1}, x_{01})) \cap \mathcal{B}(y_2, d - d_{\mathrm{H}}(x_{k1}, y_1)) \neq \emptyset. \tag{8}$$

Therefore, if (8) is not satisfied, y cannot be an occurrence of any offspring of x_k. From the triangle inequality,

$$2d - d_{\mathrm{H}}(x_{k1}, x_{01}) - d_{\mathrm{H}}(x_{k1}, y_1) \ge d_{\mathrm{H}}(x_{02}, y_2) \tag{9}$$

is necessary for (8) to be satisfied. This condition can be further transformed into

$$2d - k \ge d_{\mathrm{H}}(x_{k1}, y_1) + d_{\mathrm{H}}(x_{02}, y_2) = d_{\mathrm{H}}(x_k, y), \tag{10}$$

because $d_{\mathrm{H}}(x_{k1}, x_{01}) = d_{\mathrm{H}}(x_k, x_0) = k$. In summary, y cannot be an occurrence of any (candidate) motif $t \in \mathcal{O}(x_k, d)$ if (10) is not satisfied. Therefore, it is not necessary to check this y in the subtree rooted at x_k. To take advantage of this observation, (10) is checked for all $y \in \mathcal{S}_i$ $(2 \le i \le N)$ at each node representing x_k, and those breaking (10) are removed from the corresponding sets when the subtree rooted at x_k is traversed. When $\mathcal{S}_i$ becomes empty for some i $(2 \le i \le N)$, the subtree can be pruned.

We should compute $d_{\mathrm{H}}(x_k, y)$ to check whether x_k is a valid motif and whether (10) is satisfied. qPMSPruneI first computes the Hamming distances $d_{\mathrm{H}}(t_1, y)$ for all $t_1 \in \mathcal{S}_1$ and all $y \in \mathcal{S}_i$ $(2 \le i \le N)$, which takes $O(NL^2)$ time. By using them, the table of $d_{\mathrm{H}}(x_0, y)$ for $x_0 \in \mathcal{S}_1$ and all $y \in \mathcal{S}_i$ $(2 \le i \le N)$ is constructed and initialized at the root node of a tree. Then, at each node x_k, $d_{\mathrm{H}}(x_k, y)$ can be computed incrementally from $d_{\mathrm{H}}(x_{k-1}, y)$ of the parent x_{k-1} in $O(1)$ time because x_k differs from x_{k-1} only at one position.

When $q < N$, i.e. the planted (l, d, q)-motif search problem is considered, $\mathcal{S}_1$ does not necessarily have an occurrence of a motif. It follows that the above algorithm is not valid because it searches only $\mathcal{B}(x_0, d)$ $(x_0 \in \mathcal{S}_1)$. In this case, at least one set among $\mathcal{S}_1$, …, $\mathcal{S}_{N-q+1}$ should include an occurrence. Hence, the root occurrence x_0 of a tree should be taken not only from $\mathcal{S}_1$ but also from $\mathcal{S}_2$, …, $\mathcal{S}_{N-q+1}$. The pruning condition should also be modified: The subtree rooted at x_k can be pruned if the number of empty sets among $\mathcal{S}_1$, …, $\mathcal{S}_N$ is more than $N - q$, or, equivalently, if the number of nonempty sets (except that from which the root occurrence is taken) is less than $q - 1$.

The pseudocode of qPMSPruneI is shown in Fig. 1. FeasibleOccurrences2(k, x_k, $\mathcal{Q}$) in line 3 of qPMSPruneI_Tree removes infeasible candidate occurrences from $\mathcal{Q}$ that break (10). IsMotif(x_k, q', $\mathcal{T}$) in line 11 checks whether there exist at least $q'(= q - 1)$ sets in $\mathcal{T}$ that have an occurrence within a Hamming distance d from x_k. Namely, it returns "true" if

$$\left|\{\mathcal{Q} \in \mathcal{T} \mid \min_{y \in \mathcal{Q}} d_{\mathrm{H}}(x_k, y) \le d\}\right| \ge q'. \tag{11}$$

The time and space complexities of qPMSPruneI are given by $O((N - q + 1)NL^2 n_{\mathrm{B}}(l, d))$ and $O(NL^2)$, respectively.

B. qPMS7

qPMS7 also searches for motifs by traversing trees. Let us first assume $q = N$ as in the preceding subsection. The primary difference from qPMSPruneI is that it utilizes $r \in \mathcal{S}_2$ as well as $x_0 \in \mathcal{S}_1$ to traverse a tree. In the following, r is referred to as the reference occurrence. A node at depth k represents a pair of a position and a letter (p_k, α_k) as in qPMSPruneI, while the corresponding string x_k is constructed in a different way. More specifically, x_k is expressed by its parent x_{k-1} as follows:

1) $x_k|_{I_+(p_k-1)} = x_{k-1}|_{I_+(p_k-1)}$,
2) $x_k[p_k] = \alpha_k \neq x_{k-1}[p_k]$,
3) If $d_{\mathrm{H}}(x_k|_{I_+(p_k)}, x_0|_{I_+(p_k)}) > d_{\mathrm{H}}(x_k|_{I_+(p_k)}, r|_{I_+(p_k)})$, $x_k|_{I_-(p_k)} = x_0|_{I_-(p_k)}$. Otherwise, $x_k|_{I_-(p_k)} = r|_{I_-(p_k)}$.

It is worth noting that only 3) is different from x_k in qPMSPruneI, where $x_k|_{I_-(p_k)} = x_0|_{I_-(p_k)}$ always holds.

For y to be an occurrence at a node representing x_k,

$$\begin{aligned}\mathcal{B}(x_{k2}, d - d_{\mathrm{H}}(x_{k1}, x_{01})) \cap \mathcal{B}(r_2, d - d_{\mathrm{H}}(x_{k1}, r_1))&\\ \cap \mathcal{B}(y_2, d - d_{\mathrm{H}}(x_{k1}, y_1)) \neq \emptyset&\end{aligned} \tag{12}$$

should be satisfied, where $r_1 = r|_{I_+(p_k)}$ and $r_2 = r|_{I_-(p_k)}$. This condition is a natural extension of (8) that utilizes only x_k. In [27], it is shown that (12) can be checked by solving an ILP (integer linear programming) problem that depends on the eight variables $d - d_{\mathrm{H}}(x_{k1}, x_{01})$, $d - d_{\mathrm{H}}(x_{k1}, r_1)$, $d - d_{\mathrm{H}}(x_{k1}, y_1)$, and $R_j(x_{02}, r_2, y_2)$ $(1 \le j \le 5)$. To avoid solving the ILP problem every time when (12) is checked, it is solved in advance for every possible combination of the eight variables, and an eight-dimensional table is constructed. qPMS7 employs this table for checking (12), which takes only $O(1)$ time because all the eight values can be computed incrementally in $O(1)$ time.

When $q < N$, the root occurrence x_0 and the reference occurrence r should also be taken from those other than $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively, as in qPMSPruneI. Therefore, the pseudocode of qPMS7 is described as in Fig. 2. In qPMS7, it is sufficient to consider $x_0 \in \mathcal{S}_{i_1}$ and $r \in \mathcal{S}_{i_2}$ for $1 \le i_1 < i_2 \le N -$

```
qPMSPruneI
$\mathcal{M} \leftarrow \emptyset$
for $i = 1$ to $N - q + 1$ do
    for $j = 1$ to $L - l + 1$ do
        $\mathcal{T} \leftarrow \{\mathcal{S}_h \mid 1 \le h \le N,\ h \ne i\}$
        qPMSPruneI_Tree(0, $s^l_{ij}$, 0, $\mathcal{T}$)
    end for
end for
Output $\mathcal{M}$
```

```
qPMSPruneI_Tree($k$, $x_k$, $p_k$, $\mathcal{T}$)
$\mathcal{T}' \leftarrow \emptyset$
for all $\mathcal{Q} \in \mathcal{T}$ do
    $\mathcal{Q}' \leftarrow$ FeasibleOccurrences2($k$, $x_k$, $\mathcal{Q}$)
    if $\mathcal{Q}' \ne \emptyset$ then
        $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{\mathcal{Q}'\}$
    end if
end for
if $|\mathcal{T}'| < q - 1$ then
    return
end if
if IsMotif($x_k$, $q - 1$, $\mathcal{T}'$) = true then
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{x_k\}$
end if
if $k = d$ then
    return
end if
for $p_{k+1} = p_k + 1$ to $l$ do
    for all $\alpha \in \Sigma \setminus \{x_k[p_{k+1}]\}$ do
        $x_{k+1} \leftarrow x_k$
        $x_{k+1}[p_{k+1}] \leftarrow \alpha$
        qPMSPruneI_Tree($k + 1$, $x_{k+1}$, $p_{k+1}$, $\mathcal{T}'$)
    end for
end for
```

```
FeasibleOccurrences2($k$, $x_k$, $\mathcal{Q}$)
$\mathcal{Q}' \leftarrow \emptyset$
for all $y \in \mathcal{Q}$ do
    if $d_{\mathrm{H}}(x_k, y) \le 2d - k$ then
        $\mathcal{Q}' \leftarrow \mathcal{Q}' \cup \{y\}$
    end if
end for
return $\mathcal{Q}'$
```

```
IsMotif($x$, $q'$, $\mathcal{T}$)
matched $\leftarrow 0$
for all $\mathcal{Q} \in \mathcal{T}$ do
    found $\leftarrow$ false
    for all $y \in \mathcal{Q}$ do
        if $d_{\mathrm{H}}(x, y) \le d$ then
            found $\leftarrow$ true
            break the inner loop
        end if
    end for
    if found = true then
        matched $\leftarrow$ matched + 1
        if matched $\ge q'$ then
            return true
        end if
    end if
end for
return false
```

Fig. 1. Pseudocode of qPMSPruneI

```
qPMS7
$\mathcal{M} \leftarrow \emptyset$
for $i_1 = 1$ to $N - q + 1$ do
    for $j_1 = 1$ to $L - l + 1$ do
        for $i_2 = i_1 + 1$ to $N - q + 2$ do
            for $j_2 = 1$ to $L - l + 1$ do
                $\mathcal{T} \leftarrow \{\mathcal{S}_h \mid 1 \le h \le N,\ h \ne i_1,\ h \ne i_2\}$
                qPMS7_Tree(0, $s^l_{i_1 j_1}$, $s^l_{i_2 j_2}$, $s^l_{i_1 j_1}$, 0, $\mathcal{T}$)
            end for
        end for
    end for
end for
Output $\mathcal{M}$
```

```
qPMS7_Tree($k$, $x_0$, $r$, $x_k$, $p_k$, $\mathcal{T}$)
$\mathcal{T}' \leftarrow \emptyset$
for all $\mathcal{Q} \in \mathcal{T}$ do
    $\mathcal{Q}' \leftarrow$ FeasibleOccurrences3($x_0$, $r$, $x_k$, $p_k$, $\mathcal{Q}$)
    if $\mathcal{Q}' \ne \emptyset$ then
        $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{\mathcal{Q}'\}$
    end if
end for
if $|\mathcal{T}'| < q - 2$ then
    return
end if
if $d_{\mathrm{H}}(x_k, x_0) \le d$ and $d_{\mathrm{H}}(x_k, r) \le d$ and IsMotif($x_k$, $q - 2$, $\mathcal{T}'$) = true
then
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{x_k\}$
end if
if $k = d$ then
    return
end if
for $p_{k+1} = p_k + 1$ to $l$ do
    for all $\alpha \in \Sigma \setminus \{x_k[p_{k+1}]\}$ do
        $z|_{I_+(p_{k+1}-1)} \leftarrow x_k|_{I_+(p_{k+1}-1)}$
        $z[p_{k+1}] \leftarrow \alpha$
        $x_{k+1}|_{I_+(p_{k+1})} \leftarrow z$
        if $d_{\mathrm{H}}(z, x_0|_{I_+(p_{k+1})}) > d_{\mathrm{H}}(z, r|_{I_+(p_{k+1})})$ then
            $x_{k+1}|_{I_-(p_{k+1})} \leftarrow x_k|_{I_-(p_{k+1})}$
        else
            $x_{k+1}|_{I_-(p_{k+1})} \leftarrow r|_{I_-(p_{k+1})}$
        end if
        qPMS7_Tree($k + 1$, $x_0$, $r$, $x_{k+1}$, $p_{k+1}$, $\mathcal{T}'$)
    end for
end for
```

```
FeasibleOccurrences3($x_0$, $r$, $x_k$, $p_k$, $\mathcal{Q}$)
$\mathcal{Q}' \leftarrow \emptyset$
$d_{\mathrm{x}} \leftarrow d - d_{\mathrm{H}}(x_k|_{I_+(p_k)}, x_0|_{I_+(p_k)})$
$d_{\mathrm{r}} \leftarrow d - d_{\mathrm{H}}(x_k|_{I_+(p_k)}, r|_{I_+(p_k)})$
for all $y \in \mathcal{Q}$ do
    $d_{\mathrm{y}} \leftarrow d - d_{\mathrm{H}}(x_k|_{I_+(p_k)}, y|_{I_+(p_k)})$
    if $\mathcal{B}(x_0|_{I_-(p_k)}, d_{\mathrm{x}}) \cap \mathcal{B}(r|_{I_-(p_k)}, d_{\mathrm{r}}) \cap \mathcal{B}(y|_{I_-(p_k)}, d_{\mathrm{y}}) \ne \emptyset$ then
        $\mathcal{Q}' \leftarrow \mathcal{Q}' \cup \{y\}$
    end if
end for
return $\mathcal{Q}'$
```

Fig. 2. Pseudocode of qPMS7

$q + 2$. The time and space complexities of qPMS7 are given by $O((N - q + 1)^2 N L^3 n_{\mathrm{B}}(l, d))$ and $O(NL^2)$, respectively, It follows that qPMS7 is slower than qPMSPruneI by a factor of $O((N - q + 1)L)$ in the worst case. However, the former is superior to the latter in practice when d/l is large or $|\Sigma|$ is large.

IV. TraverStringSingle: An Improved Version of qPMSPruneI

In this section, three improvements for qPMSPruneI will be proposed one by one. The algorithms with these is named TraverStringSingle.

A. Strict Check of (8)

qPMSPruneI checks the feasibility of an occurrence y by employing (10). However, (10) is only a necessary condition for (8) and hence redundant candidate occurrences may be considered. Therefore, a necessary and sufficient condition is checked in the proposed algorithm, which is given by the following theorem.

Theorem 1: Two strings a and b of the same length satisfy

$$\mathcal{B}(a, d_{\mathrm{a}}) \cap \mathcal{B}(b, d_{\mathrm{b}}) \ne \emptyset, \tag{13}$$

if and only if

$$d_{\mathrm{a}} \ge 0,\ d_{\mathrm{b}} \ge 0, \tag{14}$$

$$d_{\mathrm{a}} + d_{\mathrm{b}} \ge d_{\mathrm{H}}(a, b). \tag{15}$$

The proof is direct from the triangle inequality and thus is omitted.

From this theorem, we can see that not only (10) but also

$$d \geq d_{\mathrm{H}}(x_{k1},\, x_{01}), \tag{16}$$

$$d \geq d_{\mathrm{H}}(x_{k1},\, y_1), \tag{17}$$

are necessary for (8). Since (16) is always satisfied because $d_{\mathrm{H}}(x_{k1},\, x_{01}) = k \leq d$, (17) is checked as well as (10). In practice, the following equivalent condition is checked instead:

$$d \geq d_{\mathrm{H}}(x_k,\, y) - d_{\mathrm{H}}(x_{02},\, y_2). \tag{18}$$

As already explained in the preceding section, $d_{\mathrm{H}}(x_k,\, y)$, which also appears in (10), can be computed incrementally. To compute $d_{\mathrm{H}}(x_{02},\, y_2)$, a table of $d_{\mathrm{H}}(x_0|_{I_-(j)},\, y|_{I_-(j)})$ $(0 \leq j \leq l-1)$ is constructed for all the candidate occurrences y in advance at the root node of every search tree. This table construction takes $O(NLl)$ time for one tree, and hence the total time complexity is given by $O((N-q+1)NL^2l)$. Therefore, it does not increase the time complexity of the overall algorithm because $l \leq n_{\mathrm{B}}(l,\, d)$ holds if $d > 0$.

B. Elimination of Unnecessary Combinations

When $q < N$, qPMSPruneI calls qPMSPruneI_Tree(0, s^l_{ij}, 0, $\mathcal{T}$) by choosing the set of candidate occurrences as $\mathcal{T} = \{\mathcal{S}_h \,|\, 1 \leq h \leq N,\, h \neq i\}$ for $1 \leq i \leq N-q+1$ (see the pseudocode of qPMSPruneI in Fig. 1). However, this choice of $\mathcal{T}$ is redundant. For example, suppose that $N = 3$ and $q = 2$. In this case, at least two input strings should have occurrences. It follows that all the possible combinations of the input strings that have an occurrence are $\{\mathcal{S}_1,\, \mathcal{S}_2,\, \mathcal{S}_3\}$, $\{\mathcal{S}_1,\, \mathcal{S}_2\}$, $\{\mathcal{S}_1,\, \mathcal{S}_3\}$, and $\{\mathcal{S}_2,\, \mathcal{S}_3\}$. Among these, $\{\mathcal{S}_1,\, \mathcal{S}_2,\, \mathcal{S}_3\}$, $\{\mathcal{S}_1,\, \mathcal{S}_2\}$, and $\{\mathcal{S}_1,\, \mathcal{S}_3\}$ that include $\mathcal{S}_1$ are considered by calling qPMSPruneI_Tree(0, s^l_{1j}, 0, $\{\mathcal{S}_2,\, \mathcal{S}_3\}$). To check the last combination $\{\mathcal{S}_2,\, \mathcal{S}_3\}$, we need not assume an occurrence in $\mathcal{S}_1$ and hence it is sufficient to choose a root occurrence from $\mathcal{S}_2$ and an occurrence from $\mathcal{S}_3$. In other words, it is sufficient to call qPMSPruneI_Tree(0, s^l_{2j}, 0, $\{\mathcal{S}_3\}$). Therefore, line 4 of qPMSPruneI in Fig. 1 can be modified into "$\mathcal{T} \leftarrow \{\mathcal{S}_h \,|\, i+1 \leq h \leq N\}$." It is expected that the algorithm becomes more efficient because the number of candidate occurrences that should be considered is reduced.

C. String Reordering

In qPMSPruneI, a subtree is pruned if the number of input strings including an occurrence becomes less than q. Because the input string from which the root occurrence is taken is always assumed to include an occurrence (the root occurrence itself), the pruning condition in qPMSPruneI_Tree is given by $|\mathcal{T}'| < q-1$ in line 8 of the pseudocode in Fig. 1. It is possible to check this condition before $\mathcal{T}'$ is completely constructed from $\mathcal{T}$. More specifically, $|\mathcal{T}'| < q-1$ is satisfied if the number of the sets in $\mathcal{T}$ that newly become empty exceeds $|\mathcal{T}| - q + 1$. This fact implies that the order of checking the feasibility of occurrences in Q (in line 3) is important to prune the subtree as early as possible. From (10), $\mathcal{Q} \in \mathcal{T}$ becomes empty by the feasibility check of the candidate occurrences when

$$\min_{y \in \mathcal{Q}} d_{\mathrm{H}}(x_k,\, y) > 2d - k \tag{19}$$

is satisfied. If we note $-1 \leq d_{\mathrm{H}}(x_{k-1},\, y) - d_{\mathrm{H}}(x_k,\, y) \leq 1$, we can say that $\mathcal{Q}$ is more likely to become empty as $\min_{y \in \mathcal{Q}} d_{\mathrm{H}}(x_{k-1},\, y)$ becomes larger. Therefore, the feasibility check is applied to $\mathcal{Q} \in \mathcal{T}$ in its nonincreasing order. To achieve this, the elements of $\mathcal{T}'$ are sorted in the nonincreasing order of

$$\min_{y \in \mathcal{Q}} d_{\mathrm{H}}(x_k,\, y) \tag{20}$$

after the feasibility check is finished. Ties are broken by $|\mathcal{Q}|$: Q with a smaller cardinality is checked earlier. The time complexity of this reordering is $O(N \log N)$ for each node, and hence the total time complexity is given by $O((N-q+1)NLn_{\mathrm{B}}(l,\, d)\log N)$.

D. TraverStringSingle

The algorithm with the three improvements in Sections IV-A-IV-C is named TraverStringSingle. Its pseudocode is shown in Fig. 3. IsMotifFast is an improved version of IsMotif in Fig. 1 that returns "false" as soon as the number of subsets $Q \in \mathcal{T}$ that do not have any occurrence exceeds $|\mathcal{T}| - q'$ (see also Section IV-C). Clearly, the time complexity of TraverStringSingle is given by $O((N-q+1)NL(L+\log N)n_{\mathrm{B}}(l,\, d))$. The space complexity of qPMSPruneI and qPMS7 is $O(NL^2)$ due to the computation of $d(x_0,\, y)$. On the other hand, in TraverStringSingle the table of $d_{\mathrm{H}}(x_0|_{I_-(j)},\, y|_{I_-(j)})$ $(0 \leq j \leq l-1)$ is constructed at the root node of each search tree and $d_{\mathrm{H}}(x_0,\, y) = d_{\mathrm{H}}(x_0|_{I_-(0)},\, y|_{I_-(0)})$ is computed at the same time. Thus the space complexity of TraverStringSingle reduces to $O(NLl)$, although the time complexity for $d(x_0,\, y)$ increases from $O((N-q+1)NL^2)$ of qPMSPruneI and qPMS7 to $O((N-q+1)NL^2l)$.

V. TraverStringDouble: Further Improvement on qPMSPruneI

TraverStringSingle proposed in the preceding section can further be improved by noting the similarity in the structure of adjacent search trees. This section will introduce an improved version of TraverStringSingle, which is named TraverStringDouble.

A. Tree Pairing

Here, $q = N$ is assumed for simplicity of explanation. In qPMSPruneI, a search tree is traversed from s^l_{1j} for every j. Since s^l_{ij} and $s^l_{i,j+1}$ have a common $(l-1)$-length substring, the two search trees rooted at s^l_{ij} and $s^l_{i,j+1}$ are assumed to have a similar structure. This motivates the new algorithm to traverse the adjacent two search trees at the same time. In the following, the case when $L-l+1$ is even will be explained. If $L-l+1$ is odd, the last search tree rooted at $s^l_{1,L-l+1}$ is traversed by TraverStringSingle_Tree in Fig. 3, and the other search trees are paired.

```
TraverStringSingle
 1: $\mathcal{M} \leftarrow \emptyset$
 2: for $i = 1$ to $N - q + 1$ do
 3:     for $j = 1$ to $L - l + 1$ do
 4:         $\mathcal{T} \leftarrow \{\mathcal{S}_h \mid i + 1 \le h \le N\}$
 5:         TraverStringSingle_Tree(0, $s^l_{ij}$, $s^l_{ij}$, 0, $\mathcal{T}$)
 6:     end for
 7: end for
 8: Output $\mathcal{M}$
```

```
TraverStringSingle_Tree($k$, $x_0$, $x_k$, $p_k$, $\mathcal{T}$)
 1: $\mathcal{T}' \leftarrow \emptyset$
 2: missed $\leftarrow 0$
 3: for all $\mathcal{Q} \in \mathcal{T}$ do
 4:     $\mathcal{Q}' \leftarrow$ FeasibleOccurrencesStrict2($k$, $x_0$, $x_k$, $p_k$, $\mathcal{Q}$)
 5:     if $\mathcal{Q}' \ne \emptyset$ then
 6:         $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{\mathcal{Q}'\}$
 7:     else
 8:         missed $\leftarrow$ missed $+ 1$
 9:         if missed $> |\mathcal{T}| - q + 1$ then
10:             return
11:         end if
12:     end if
13: end for
14: if IsMotifFast($x_k$, $q - 1$, $\mathcal{T}'$) = true then
15:     $\mathcal{M} \leftarrow \mathcal{M} \cup \{x_k\}$
16: end if
17: if $k = d$ then
18:     return
19: end if
20: Sort the elements of $\mathcal{T}'$ in the nonincreasing order of (20).
21: for $p_{k+1} = p_k + 1$ to $l$ do
22:     for all $\alpha \in \Sigma \setminus \{x_k[p_{k+1}]\}$ do
23:         $x_{k+1} \leftarrow x_k$
24:         $x_{k+1}[p_{k+1}] \leftarrow \alpha$
25:         TraverStringSingle_Tree($k + 1$, $x_0$, $x_{k+1}$, $p_{k+1}$, $\mathcal{T}'$)
26:     end for
27: end for
```

```
FeasibleOccurrencesStrict2($k$, $x_0$, $x_k$, $p_k$, $\mathcal{Q}$)
 1: $\mathcal{Q}' \leftarrow \emptyset$
 2: for all $y \in \mathcal{Q}$ do
 3:     $d_0 \leftarrow d + d_{\mathrm{H}}(x_0|_{I_-(p_k)}, y|_{I_-(p_k)})$
 4:     if $d_{\mathrm{H}}(x_k, y) \le \min(d_0, 2d - k)$ then
 5:         $\mathcal{Q}' \leftarrow \mathcal{Q}' \cup \{y\}$
 6:     end if
 7: end for
 8: return $\mathcal{Q}'$
```

```
IsMotifFast($x$, $q'$, $\mathcal{T}$)
 1: matched $\leftarrow 0$
 2: missed $\leftarrow 0$
 3: for all $\mathcal{Q} \in \mathcal{T}$ do
 4:     found $\leftarrow$ false
 5:     for all $y \in \mathcal{Q}$ do
 6:         if $d_{\mathrm{H}}(x, y) \le d$ then
 7:             found $\leftarrow$ true
 8:             break the inner loop
 9:         end if
10:     end for
11:     if found = true then
12:         matched $\leftarrow$ matched $+ 1$
13:         if matched $\ge q'$ then
14:             return true
15:         end if
16:     else
17:         missed $\leftarrow$ missed $+ 1$
18:         if missed $> |\mathcal{T}| - q'$ then
19:             return false
20:         end if
21:     end if
22: end for
```

Fig. 3. Pseudocode of TraverStringSingle

```
TraverStringDouble
 1: $\mathcal{M} \leftarrow \emptyset$
 2: for $i = 1$ to $N - q + 1$ do
 3:     $j \leftarrow 1$.
 4:     while $j \le L - l + 1$ do
 5:         if $j < N - q + 1$ then
 6:             $\mathcal{T} \leftarrow \{\mathcal{S}^{\mathrm{d}}_h \mid i + 1 \le h \le N\}$
 7:             TraverStringDouble_Tree(0, $s^{l+1}_{ij}$, $s^{l+1}_{ij}$, 1, $\mathcal{T}$)
 8:         else
 9:             $\mathcal{T} \leftarrow \{\mathcal{S}_h \mid i + 1 \le h \le N\}$
10:             TraverStringSingle_Tree(0, $s^l_{ij}$, $s^l_{ij}$, 0, $\mathcal{T}$)
11:         end if
12:         $j \leftarrow j + 2$.
13:     end while
14: end for
15: Output $\mathcal{M}$
```

```
TraverStringDouble_Tree($k$, $x^{\mathrm{d}}_0$, $x^{\mathrm{d}}_k$, $p_k$, $\mathcal{T}$)
 1: $\mathcal{T}' \leftarrow \emptyset$
 2: missed $\leftarrow 0$
 3: for all $\mathcal{Q} \in \mathcal{T}$ do
 4:     $\mathcal{Q}' \leftarrow$ FeasibleOccurrencesStrict2Double($k$, $x^{\mathrm{d}}_0$, $x^{\mathrm{d}}_k$, $p_k$, $\mathcal{Q}$)
 5:     if $\mathcal{Q}' \ne \emptyset$ then
 6:         $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{\mathcal{Q}'\}$
 7:     else
 8:         missed $\leftarrow$ missed $+ 1$
 9:         if missed $> |\mathcal{T}| - q + 1$ then
10:             return
11:         end if
12:     end if
13: end for
14: if IsMotifDoubleA($x^{\mathrm{d}}_k|_A$, $q - 1$, $\mathcal{T}'$) = true then
15:     $\mathcal{M} \leftarrow \mathcal{M} \cup \{x^{\mathrm{d}}_k|_A\}$
16: end if
17: if IsMotifDoubleB($x^{\mathrm{d}}_k|_B$, $q - 1$, $\mathcal{T}'$) = true then
18:     $\mathcal{M} \leftarrow \mathcal{M} \cup \{x^{\mathrm{d}}_k|_B\}$
19: end if
20: if $k = d$ then
21:     return
22: end if
23: Sort the elements of $\mathcal{T}'$ in the nonincreasing order of (30).
24: for $p_{k+1} = p_k + 1$ to $l$ do
25:     for all $\alpha \in \Sigma \setminus \{x_k[p_{k+1}]\}$ do
26:         $x^{\mathrm{d}}_{k+1} \leftarrow x^{\mathrm{d}}_k$
27:         $x^{\mathrm{d}}_{k+1}[p_{k+1}] \leftarrow \alpha$
28:         TraverStringDouble_Tree($k + 1$, $x^{\mathrm{d}}_0$, $x^{\mathrm{d}}_{k+1}$, $p_{k+1}$, $\mathcal{T}'$)
29:     end for
30: end for
31: for all $\alpha \in \Sigma \setminus \{x_k[1]\}$ do
32:     $x_{k+1} \leftarrow x^{\mathrm{d}}_k|_A$
33:     $x_{k+1}[1] \leftarrow \alpha$
34:     if IsMotifDoubleA($x_{k+1}$, $q - 1$, $\mathcal{T}'$) = true then
35:         $\mathcal{M} \leftarrow \mathcal{M} \cup \{x_{k+1}\}$
36:     end if
37: end for
38: for all $\alpha \in \Sigma \setminus \{x_k[l + 1]\}$ do
39:     $x_{k+1} \leftarrow x^{\mathrm{d}}_k|_B$
40:     $x_{k+1}[l] \leftarrow \alpha$
41:     if IsMotifDoubleB($x_{k+1}$, $q - 1$, $\mathcal{T}'$) = true then
42:         $\mathcal{M} \leftarrow \mathcal{M} \cup \{x_{k+1}\}$
43:     end if
44: end for
```

Fig. 4. Pseudocode of TraverStringDouble

The combined search tree is traversed as follows. Let us define vectors A, B, C, and D by

$$A = (1, \ldots, l), \quad B = (2, \ldots, l + 1),$$
$$C = (2, \ldots, l), \quad D = (1, \ldots, l + 1). \tag{21}$$

The root occurrence x^{d}_0 of length $l + 1$ is taken from $\{s^{l+1}_{11}, \ldots, s^{l+1}_{1,L-l}\} = \mathcal{S}^{\mathrm{d}}_1 \setminus \{s^{l+1}_{10}, s^{l+1}_{1,L-l+1}\}$ and assumed to denote two occurrences of length l, $x^{\mathrm{d}}_0|_A$ and $x^{\mathrm{d}}_0|_B$, at the same time. A node at depth k corresponds to a pair of a position and a letter (p_k, α_k) as in qPMSPruneI, where $1 \le p_k \le l+1$ and $\alpha_k \in \Sigma \setminus x^{\mathrm{d}}_0[p_k]$. The node becomes a leaf when $p_k = 1$ or $p_k = l + 1$, and otherwise, p_k ($p_k \ne 1$, $p_k \ne l + 1$) is chosen to satisfy $p_{k-1} < p_k$ ($p_{k-1} \ne 1$, $p_{k-1} \ne l + 1$). The string x^{d}_k represented by the node is constructed by:

1) If $p_k = 1$, x^{d}_k is an l-length string, and $x^{\mathrm{d}}_k[1] = \alpha_k$, $x^{\mathrm{d}}_k|_{(2,\ldots,l)} = x^{\mathrm{d}}_{k-1}|_C$.
2) If $p_k = l+1$, x^{d}_k is an l-length string, and $x^{\mathrm{d}}_k|_{(1,\ldots,l-1)} = x^{\mathrm{d}}_{k-1}|_C$, $x^{\mathrm{d}}_k[l] = \alpha_k$.

3) Otherwise, x_k^{d} is an $(l+1)$-length string, and

$$x_k^{\mathrm{d}}[j] = \begin{cases} x_{k-1}^{\mathrm{d}}[j] & \text{if } j \neq p_k, \\ \alpha_k & \text{if } j = p_k. \end{cases} \tag{22}$$

A pair of candidate occurrences is also denoted by an $(l+1)$-length string. More specifically, for $y^{\mathrm{d}} \in \mathcal{S}_j^{\mathrm{d}}$ $(2 \le j \le N)$, $y^{\mathrm{d}}|_A$ and $y^{\mathrm{d}}|_B$ are assumed to be candidate occurrences of $x_k^{\mathrm{d}}|_A$ and $x_k^{\mathrm{d}}|_B$, respectively, when $|x_k^{\mathrm{d}}| = l+1$. From (10) and (17), for at least $y^{\mathrm{d}}|_A$ or $y^{\mathrm{d}}|_B$ to be an occurrence of x_k^{d} with $|x_k^{\mathrm{d}}| = l+1$,

$$2d - k \ge d_{\mathrm{H}}(x_k^{\mathrm{d}}|_A,\, y^{\mathrm{d}}|_A), \tag{23}$$

$$d \ge d_{\mathrm{H}}(x_k^{\mathrm{d}}|_{C_+(p_k-1)},\, y^{\mathrm{d}}|_{C_+(p_k-1)}), \tag{24}$$

or

$$2d - k \ge d_{\mathrm{H}}(x_k^{\mathrm{d}}|_B,\, y^{\mathrm{d}}|_B), \tag{25}$$

$$d \ge d_{\mathrm{H}}(x_k^{\mathrm{d}}|_{C_+(p_k-1)},\, y^{\mathrm{d}}|_{C_+(p_k-1)}), \tag{26}$$

should be satisfied. Let us define $d_{\mathrm{t}}(x_0^{\mathrm{d}},\, y^{\mathrm{d}})$ by

$$d_{\mathrm{t}}(x_0^{\mathrm{d}},\, y^{\mathrm{d}}) = \min(d_{\mathrm{H}}(x_0^{\mathrm{d}}[1],\, y^{\mathrm{d}}[1]),\, d_{\mathrm{H}}(x_0^{\mathrm{d}}[l+1],\, y^{\mathrm{d}}[l+1])). \tag{27}$$

Then, at least one of (23) and (25) is satisfied when

$$2d - k \ge d_{\mathrm{H}}(x_k^{\mathrm{d}}|_C,\, y^{\mathrm{d}}|_C) + d_{\mathrm{t}}(x_0^{\mathrm{d}},\, y^{\mathrm{d}}). \tag{28}$$

On the other hand, (24) (or (26)) can be transformed into

$$d \ge d_{\mathrm{H}}(x_k^{\mathrm{d}}|_C,\, y^{\mathrm{d}}|_C) - d_{\mathrm{H}}(x_0^{\mathrm{d}}|_{C_-(p_k-1)},\, y^{\mathrm{d}}|_{C_-(p_k-1)}). \tag{29}$$

Therefore, (28) and (29) are checked for the feasibility of y^{d}. For this purpose, $d_{\mathrm{H}}(x_k^{\mathrm{d}}|_C,\, y^{\mathrm{d}}|_C)$ is computed incrementally, while $d_{\mathrm{H}}(x_0^{\mathrm{d}}|_{C_-(p_k-1)},\, y^{\mathrm{d}}|_{C_-(p_k-1)})$ is computed from a table constructed in advance at the root node as $d_{\mathrm{H}}(x_{02},\, y_2)$ in (18).

When two search trees are traversed separately, we should check (23) and (24) for $x_k^{\mathrm{d}}|_A$ in the first search tree and (25) and (26) for $x_k^{\mathrm{d}}|_B$ in the second search tree. On the other hand, we only need to check (28) and (29) for x^{d} in the paired search tree. It follows that the tree pairing enables us to reduce the computational efforts required for the feasibility check.

B. *TraverStringDouble*

The algorithm with all the four improvements in Sections IV and V-A is named TraverStringDouble. Its pseudocode is shown in Fig. 4. When $L - l + 1$ is odd, TraverStringSingle_Tree is called in line 10 of TraverStringDouble. FeasibleOccurrencesStrict2Double in line 4 of TraverStringDouble_Tree returns the candidate occurrences that satisfy (28) and (29) as FeasibleOccurrencesStrict2 in Fig. 3. In line 23 of TraverStringDouble_Tree, the sets in $\mathcal{T}'$ are sorted in the nonincreasing order of

$$\min_{y^{\mathrm{d}} \in \mathcal{Q}} (d_{\mathrm{H}}(x_k^{\mathrm{d}}|_C,\, y^{\mathrm{d}}|_C) + d_{\mathrm{t}}(x_0^{\mathrm{d}},\, y^{\mathrm{d}})), \tag{30}$$

by taking into account (28) instead of (10). The procedure IsMotifDoubleA$(x, q', \mathcal{T})$ in lines 14 and 34 of TraverStringDouble_Tree returns "true" when

$$\left| \{\mathcal{Q} \in \mathcal{T} \mid \min_{y^{\mathrm{d}} \in \mathcal{Q}} d_{\mathrm{H}}(x,\, y^{\mathrm{d}}|_A) \le d\} \right| \ge q', \tag{31}$$

as IsMotifFast for TraverStringSingle in Fig. 3. Similarly, IsMotifDoubleB$(x, q', \mathcal{T})$ in lines 17 and 41 of TraverStringDouble_Tree returns "true" when

$$\left| \{\mathcal{Q} \in \mathcal{T} \mid \min_{y^{\mathrm{d}} \in \mathcal{Q}} d_{\mathrm{H}}(x,\, y^{\mathrm{d}}|_B) \le d\} \right| \ge q'. \tag{32}$$

The pseudocodes of FeasibleOccurrencesStrict2Double, IsMotifDoubleA, and IsMotifDoubleB are omitted here. The time and space complexities of TraverStringDouble are same as those of TraverStringSingle: $O((N-q+1)NL(L+\log N)n_{\mathrm{B}}(l,\, d))$ and $O(NLl)$, respectively.

VI. TraverStringRef: An Improved Version of qPMS7

Next, qPMS7 will be improved so that a new algorithm named TraverStringRef is obtained. In this section, the proposed three improvements will be explained one by one.

A. *Feasibility Check without Precomputed Table*

In qPMS7, the feasibility check of an occurrence is performed by a table computed in advance. However, it takes 10 seconds to read the table because its size is $(l+1)^5(d+1)^3$ (its actual file size is approximately 300MB). To avoid reading such a large file of the precomputed table, the following theorem is exploited.

Theorem 2: Three strings a, b and c of the same length satisfy

$$\mathcal{B}(a,\, d_{\mathrm{a}}) \cap \mathcal{B}(b,\, d_{\mathrm{b}}) \cap \mathcal{B}(c,\, d_{\mathrm{c}}) \neq \emptyset, \tag{33}$$

if and only if

$$d_{\mathrm{a}} \ge 0,\ d_{\mathrm{b}} \ge 0,\ d_{\mathrm{c}} \ge 0, \tag{34}$$

$$d_{\mathrm{a}} + d_{\mathrm{b}} \ge d_{\mathrm{H}}(a,\, b), \tag{35}$$

$$d_{\mathrm{b}} + d_{\mathrm{c}} \ge d_{\mathrm{H}}(b,\, c), \tag{36}$$

$$d_{\mathrm{c}} + d_{\mathrm{a}} \ge d_{\mathrm{H}}(c,\, a), \tag{37}$$

$$d_{\mathrm{a}} + d_{\mathrm{b}} + d_{\mathrm{c}} \ge |R_2(a,\, b,\, c)| + |R_3(a,\, b,\, c)| + |R_4(a,\, b,\, c)| + 2|R_5(a,\, b,\, c)|. \tag{38}$$

This theorem is an extension of the necessary and sufficient condition given in [11] that covers only the case when $d_{\mathrm{a}} = d_{\mathrm{b}} = d_{\mathrm{c}}$. The proof is shown in Supplemental Material.

To apply this theorem to the feasibility check, the string represented by a node in the search tree is constructed not as in qPMS7, but as in qPMSPruneI. Namely, the string x_k for a node at depth k that corresponds to $(p_k,\, \alpha_k)$ is given by

$$x_k[j] = \begin{cases} x_{k-1}[j] & \text{if } j \neq p_k, \\ \alpha_k & \text{if } j = p_k. \end{cases} \tag{39}$$

Then, applying this theorem to (12) yields

$$\begin{aligned} d &\geq d_{\mathrm{H}}(x_{k1},\, x_{01}), && (40)\\ d &\geq d_{\mathrm{H}}(x_{k1},\, r_1), && (41)\\ d &\geq d_{\mathrm{H}}(x_{k1},\, y_1), && (42)\\ 2d &\geq d_{\mathrm{H}}(x_{k1},\, x_{01}) + d_{\mathrm{H}}(x_{k1},\, r_1) + d_{\mathrm{H}}(x_{k2},\, r_2), && (43)\\ 2d &\geq d_{\mathrm{H}}(x_{k1},\, r_1) + d_{\mathrm{H}}(x_{k1},\, y_1) + d_{\mathrm{H}}(r_2,\, y_2), && (44)\\ 2d &\geq d_{\mathrm{H}}(x_{k1},\, x_{01}) + d_{\mathrm{H}}(x_{k1},\, y_1) + d_{\mathrm{H}}(x_{k2},\, y_2), && (45)\\ 3d &\geq d_{\mathrm{H}}(x_{k1},\, x_{01}) + d_{\mathrm{H}}(x_{k1},\, r_1) + d_{\mathrm{H}}(x_{k1},\, y_1)\\ &\quad + |R_2(x_{k2},\, r_2,\, y_2)| + |R_3(x_{k2},\, r_2,\, y_2)|\\ &\quad + |R_4(x_{k2},\, r_2,\, y_2)| + 2|R_5(x_{k2},\, r_2,\, y_2)|. && (46) \end{aligned}$$

By substituting

$$\begin{aligned} x_{k2} &= x_{02}, && (47)\\ d_{\mathrm{H}}(x_{k1},\, x_{01}) &= k, && (48)\\ d_{\mathrm{H}}(x_{k1},\, r_1) &= d_{\mathrm{H}}(x_k,\, r) - d_{\mathrm{H}}(x_{02},\, r_2), && (49)\\ d_{\mathrm{H}}(x_{k1},\, y_1) &= d_{\mathrm{H}}(x_k,\, y) - d_{\mathrm{H}}(x_{02},\, y_2) && (50) \end{aligned}$$

into the above inequalities, we obtain

$$\begin{aligned} d &\geq k, && (51)\\ d &\geq d_{\mathrm{H}}(x_k,\, r) - d_{\mathrm{H}}(x_{02},\, r_2), && (52)\\ d &\geq d_{\mathrm{H}}(x_k,\, y) - d_{\mathrm{H}}(x_{02},\, y_2), && (53)\\ 2d - k &\geq d_{\mathrm{H}}(x_k,\, r), && (54)\\ 2d &\geq d_{\mathrm{H}}(x_k,\, r) + d_{\mathrm{H}}(x_k,\, y)\\ &\quad - d_{\mathrm{H}}(x_{02},\, r_2) - d_{\mathrm{H}}(x_{02},\, y_2) + d_{\mathrm{H}}(r_2,\, y_2), && (55)\\ 2d - k &\geq d_{\mathrm{H}}(x_k,\, y), && (56)\\ 3d - k &\geq d_{\mathrm{H}}(x_k,\, r) + d_{\mathrm{H}}(x_k,\, y)\\ &\quad - d_{\mathrm{H}}(x_{02},\, r_2) - d_{\mathrm{H}}(x_{02},\, y_2)\\ &\quad + |R_2(x_{02},\, r_2,\, y_2)| + |R_3(x_{02},\, r_2,\, y_2)|\\ &\quad + |R_4(x_{02},\, r_2,\, y_2)| + 2|R_5(x_{02},\, r_2,\, y_2)|\\ &= d_{\mathrm{H}}(x_k,\, r) + d_{\mathrm{H}}(x_k,\, y) - |R_4(x_{02},\, r_2,\, y_2)|. && (57) \end{aligned}$$

Here, (2) and (4) are employed to derive (57). Since (51) is trivial and thus can be removed, we should check (52)–(57). For this purpose, $d_{\mathrm{H}}(x_k,\, r)$ and $d_{\mathrm{H}}(x_k,\, y)$ are computed incrementally, while $d_{\mathrm{H}}(x_{02},\, r_2)$, $d_{\mathrm{H}}(x_{02},\, y_2)$, $d_{\mathrm{H}}(r_2,\, y_2)$, and $|R_4(x_{02},\, r_2,\, y_2)|$ are computed from a table constructed at the root node.

It is worth noting that (52) and (54), which correspond to (18) and (10), respectively, can be checked independently of y.

B. Elimination of Unnecessary Combinations

The same argument holds for qPMS7 as that in Section IV-B for qPMSPruneI, and unnecessary checks can be suppressed when $q < N$. In this case, line 6 of qPMS7 in Fig. 2 is replaced by $\mathcal{T} \leftarrow \{\mathcal{S}_h \,|\, i_2 + 1 \leq h \leq N\}$.

C. String Reordering

This improvement is also similar to that in Section IV-C for qPMSPruneI. It is expected that the subtree rooted at the current node is pruned as early as possible by checking the feasibility of the occurrences $y \in \mathcal{Q}$ ($\mathcal{Q} \in \mathcal{T}$) in the nonincreasing order of (20). The difference is that it is also applied when the input string from which the reference occurrences are taken is determined. Since the number of strings in $\mathcal{B}(x_0,\, d) \cap \mathcal{B}(r,\, d)$ is a nondecreasing function of $d_{\mathrm{H}}(x_0,\, r)$, the reference occurrences r are taken from the input string $\mathcal{S}_i$ that maximizes

$$\min_{r \in \mathcal{S}_i} d_{\mathrm{H}}(x_0,\, r), \tag{58}$$

to reduce the size of the search tree.

TABLE III
Number of Nodes at Each Depth ($|\Sigma| = 4$, $l = 6$, and $d = 3$)

depth	$(x_0,\, r)$	
	(AAAAAA, AAGGGG)	(AAAAAA, GGGGAA)
0	1	1
1	16	10
2	60	42
3	112	112
total	189	165

D. Position Reordering

To make the pruning of subtrees as efficient as possible, the structure of the search trees is investigated.

As explained in Section VI-A, (52) and (54) can be checked independently of the candidate occurrences y, and only those nodes satisfying (52) and (54) are traversed in a search tree. The observation here is that the structure of the search tree changes in accordance with x_0 and r even if $d_{\mathrm{H}}(x_0,\, r)$ does not change. For example, suppose $|\Sigma| = 4$, $l = 6$, and $d = 3$, and consider the two cases where $(x_0,\, r)$=(AAAAAA, AAGGGG) and $(x_0,\, r)$=(AAAAAA, GGGGAA). In the former case, the number of feasible nodes at depth 1 satisfying (52) and (54) is 16 (CAAAAA, GAAAAA, TAAAAA, ACAAAA, AGAAAA, ATAAAA, AACAAA, AAGAAA, AATAAA, AAACAA, AAAGAA, AAATAA, AAAACA, AAAAGA, AAAATA, AAAAAG), while it is 10 (CAAAAA, GAAAAA, TAAAAA, ACAAAA, AGAAAA, ATAAAA, AACAAA, AAGAAA, AATAAA, AAAGAA) in the latter case. Moreover, the total number of nodes in the two cases are 189 and 165, respectively, as summarized in TABLE III. It follows that the tree traversal is more efficient and the pruning of a subtree cuts more nodes in the latter case than in the former case. It should be noted that this property does not always hold. The above two cases do not make any difference when $d_{\mathrm{H}}(x_0,\, r) \geq 2d - 1$, and, moreover, the opposite property is true when $d_{\mathrm{H}}(x_0,\, r)$ is small (typically, $d_{\mathrm{H}}(x_0,\, r) \leq d$). However, we can almost always assume $d_{\mathrm{H}}(x_0,\, r) > d$ because r is chosen from the input string $\mathcal{S}_i$ that maximizes (58).

To take advantage of this observation, the positions of strings are reordered so that the positions where the letters of x_0 and r are different come earlier. To achieve this, an l-dimensional vector J is computed for each pair of x_0 and r at the root node of the search tree, where

$$\begin{aligned} x_0[J(i)] &\neq r[J(i)], \quad 1 \leq i \leq d_{\mathrm{H}}(x_0,\, r), && (59)\\ x_0[J(i)] &= r[J(i)], \quad d_{\mathrm{H}}(x_0,\, r) + 1 \leq i \leq l. && (60) \end{aligned}$$

```
TraverStringRef
 1: $\mathcal{M} \leftarrow \emptyset$
 2: for $i_1 = 1$ to $N - q + 1$ do
 3:     for $j_1 = 1$ to $L - l + 1$ do
 4:         $\mathcal{T} \leftarrow \{\mathcal{S}_h \mid i_1 + 1 \le h \le N\}$
 5:         Sort the elements of $\mathcal{T}$ in the nonincreasing order of (58)
 6:         while $|\mathcal{T}| \ge q - 2$ do
 7:             $\mathcal{R} \leftarrow$ first element of $\mathcal{T}$
 8:             $\mathcal{T} \leftarrow \mathcal{T} \setminus \mathcal{R}$
 9:             for all $r \in \mathcal{R}$ do
10:                 Initialize the vector $J$
11:                 if $d_{\mathrm{H}}(s^l_{i_1 j_1}, r) \le 2d$ then
12:                     TraverStringRef_Tree(0, $s^l_{i_1 j_1}$, $r$, $s^l_{i_1 j_1}$, 0, $\mathcal{T}$, $J$)
13:                 end if
14:             end for
15:         end while
16:     end for
17: end for
18: Output $\mathcal{M}$
```

```
TraverStringRef_Tree($k$, $x_0$, $r$, $x_k$, $p_k$, $\mathcal{T}$, $J$)
 1: $\mathcal{T}' \leftarrow \emptyset$
 2: missed $\leftarrow 0$
 3: for all $\mathcal{Q} \in \mathcal{T}$ do
 4:     $\mathcal{Q}' \leftarrow$ FeasibleOccurrencesReordered3($k$, $x_0$, $r$, $x_k$, $p_k$, $\mathcal{Q}$, $J$)
 5:     if $\mathcal{Q}' \ne \emptyset$ then
 6:         $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{\mathcal{Q}'\}$
 7:     else
 8:         missed $\leftarrow$ missed $+ 1$
 9:         if missed $> |\mathcal{T}| - q + 2$ then
10:             return
11:         end if
12:     end if
13: end for
14: if IsMotifFast($x_k$, $q - 2$, $\mathcal{T}'$) $=$ true then
15:     $\mathcal{M} \leftarrow \mathcal{M} \cup \{x_k\}$
16: end if
17: if $k = d$ then
18:     return
19: end if
20: Sort the elements of $\mathcal{T}'$ in the nonincreasing order of (20).
21: for $p_{k+1} = p_k + 1$ to $l$ do
22:     for all $\alpha \in \Sigma \setminus \{x_k[J(p_{k+1})]\}$ do
23:         $x_{k+1} \leftarrow x_k$
24:         $x_{k+1}[J(p_{k+1})] \leftarrow \alpha$
25:         $d_0 \leftarrow d + d_{\mathrm{H}}(x_0|_{J_-(p_{k+1})}, r|_{J_-(p_{k+1})})$
26:         if $d_{\mathrm{H}}(x_{k+1}, r) \le \min(d_0, 2d - k - 1)$ then
27:             TraverStringRef_Tree($k + 1$, $x_0$, $r$, $x_{k+1}$, $p_{k+1}$, $\mathcal{T}'$, $J$)
28:         end if
29:     end for
30: end for
```

Fig. 5. Pseudocode of TraverStringRef

Since its time complexity is $O(l)$, this reordering does not affect the overall time complexity of the algorithm. It is necessary to modify the feasibility check as well as the branching to take this reordering into account. However, this modification is direct and thus the detailed explanation is omitted.

E. *TraverStringRef*

The new algorithm TraverStringRef with the the four improvements from qPMS7 is summarized in Fig. 5. In line 11 of TraverStringRef and in line 26 of TraverStringRef_Tree, (52) and (54) are checked at the same time. FeasibleOccurrencesReordered3 in line 4 of TraverStringRef_Tree returns candidate occurrences that satisfy (53), (55), (56), and (57), where x_{02}, y_2, and r_2 are replaced by

$$x'_{02} = x_0|_{J_-(p_k)}, \quad y'_2 = y|_{J_-(p_k)}, \quad r'_2 = r|_{J_-(p_k)}, \qquad (61)$$

respectively, due to the position reordering in the preceding subsection. Its pseudocode is omitted. The time complexity of TraverStringRef is given by that of TraverStringSingle and TraverStringDouble multiplied by $O((N - q + 1)L)$ and thus $O((N - q + 1)^2 N L^2 (L + \log N) n_{\mathrm{B}}(l, d))$ (see also the time complexity of qPMS7 in Section III-B). The space complexity does not change from TraverStringSingle and TraverStringDouble, and is given by $O(NLl)$.

TABLE IV
PARAMETER SETTING FOR DATA SET

parameter	setting
$\|\Sigma\|$	4 (DNA), 20 (protein)
N	20
L	600
q	10, 20
l	13, 15, 17, ...

VII. Computational Experiments

The effectiveness of the proposed algorithms will be examined by computational experiments. As in previous studies including [33], the random data set was generated according to the FM (fixed number of mutations) model [9]. First, N base strings of length L were generated randomly so that each letter in Σ appears with the equal probability $1/|\Sigma|$. Then, a motif of length l was generated in the same manner. Next, q occurrences were generated from the motif by mutating letters at exactly d random positions[1] Finally, they were planted in q input strings chosen randomly, where the planting locations were also determined randomly. The parameter setting is summarized in Table IV. The remaining parameter, d, was determined to generate "challenging" instances. Its choice follows [33]. For each combination of parameters, 10 instances were generated.

The proposed algorithms were written in C, whose source code is downloadable from https://sites.google.com/site/shunjitanaka/motif. The computation was performed on a laptop computer with an Intel Core i7-3610QM CPU (2.3GHz) and 16GB memory. In the tables shown from now on, TraverStringSingle, TraverStringDouble, and TraverStringRef are abbreviated to TravStrS, TravStrD, and TravStrR, respectively.

A. *Comparison with qPMSPruneI and qPMS7*

First, TraverStringDouble and TraverStringRef will be compared with qPMSPruneI and qPMS7, respectively. The program of qPMS7 was downloaded from http://pms.engr.uconn.edu/downloads/qPMS7.zip. Since the program of qPMSPruneI was not available, the CPU times of qPMSPruneI were estimated from those given in [33] by using the results of qPMS7. More specifically, it was calculated as follows: (The average CPU time of qPMS7 obtained in this study)×(The ratio of the CPU time of qPMSPruneI to that of qPMS7 in [33]). It should be noted that qPMS7 did not work correctly for the protein instances with $|\Sigma| = 20$ because the program always assumes $|\Sigma| = 21$ for them.

The results are summarized in Tables V and VI, where the minimum (min), average (ave), and maximum (max) CPU

[1]In [33], the authors claims that the data set was generated by mutating letters at most d positions. However, it seems that letters at exactly d positions were mutated also in their data set.

TABLE V
COMPUTATIONAL RESULTS FOR DNA SEQUENCES ($|\Sigma| = 4$)

(a) $q = 20$ (100% match)

(l, d)		TravStrD	TravStrR	qPMSPruneI*	qPMS7
	min	4.0	10.9		39
(13, 4)	ave	4.0	11.3	14.3	39.4
	max	4.2	11.6		40
	min	27.6	45.5		129
(15, 5)	ave	28.3	46.5	132.6	132.6
	max	29.8	48.3		134
	min	199.3	175.2		499
(17, 6)	ave	202.4	179.5	1046.8	509.5
	max	209.2	182.3		529
	min	1527.4	728.4		2182
(19, 7)	ave	1547.4	744.6	8465.6	2240.9
	max	1581.8	775.5		2325
	min	10781.7	3039.2		10367
(21, 8)	ave	10943.9	3098.6	71749.6	10994.6
	max	11116.9	3192.1		11352
	min		12721.9		52741
(23, 9)	ave		13027.0		56097.2
	max		13385.5		62164
	min		51796.5		
(25, 10)	ave		53781.6		
	max		56628.8		

*Estimated from [33] and the results of qPMS7.

(b) $q = 10$ (50% match)

(l, d)		TravStrD	TravStrR	qPMSPruneI*	qPMS7
	min	2.0	5.5		31
(13, 3)	ave	2.0	5.5	12.8	31.0
	max	2.0	5.6		31
	min	13.3	31.5		151
(15, 4)	ave	13.5	32.0	126.7	152.0
	max	13.6	32.2		153
	min	120.7	162.9		839
(17, 5)	ave	123.0	166.1	1116.5	850.7
	max	124.5	168.8		857
	min	1137.7	763.1		4770
(19, 6)	ave	1148.6	779.9	10540.8	4865.0
	max	1174.2	794.2		4978
	min	10020.7	3659.2		28820
(21, 7)	ave	10067.7	3700.6		29258.0
	max	10149.8	3745.7		29856
	min		17676.5		
(23, 8)	ave		17922.2		
	max		18181.7		

*Estimated from [33] and the results of qPMS7.

TABLE VI
COMPUTATIONAL RESULTS FOR PROTEIN SEQUENCES ($|\Sigma| = 20$)

(a) $q = 20$ (100% match)

(l, d)		TravStrD	TravStrR	qPMSPruneI*	qPMS7
	min	98.5	3.2		59
(13, 6)	ave	105.6	3.3	916.5	61.1
	max	113.7	3.4		64
	min	644.2	3.9		80
(15, 7)	ave	666.5	4.1	6790.7	89.6
	max	720.9	4.5		114
	min	2262.7	4.9		104
(17, 8)	ave	2394.8	6.0	34680.0	231.2
	max	2746.1	9.6		838
	min	7554.7	9.4		157
(19, 9)	ave	11379.0	17.8		1891.2
	max	18031.7	37.8		8672
	min		44.2		
(21, 10)	ave		109.6		
	max		286.9		
	min		125.8		
(23, 11)	ave		1068.3		
	max		2070.3		
	min		1712.1		
(25, 12)	ave		8333.8		
	max		25639.2		

*Estimated from [33] and the results of qPMS7.

(b) $q = 10$ (50% match)

(l, d)		TravStrD	TravStrR	qPMSPruneI*	qPMS7
	min	6.8	3.2		18
(13, 5)	ave	7.0	3.2	97.1	18.2
	max	7.2	3.2		19
	min	27.9	5.2		29
(15, 6)	ave	30.3	5.3	403.3	32.7
	max	33.9	5.3		43
	min	215.9	9.1		51
(17, 7)	ave	236.7	9.2		74.0
	max	262.0	9.3		146
	min	671.7	15.2		87
(19, 8)	ave	1328.5	16.3		413.9
	max	4158.6	21.7		865
	min		25.2		359
(21, 9)	ave		28.0		2551.7
	max		32.6		10653
	min		45.5		
(23, 10)	ave		141.0		
	max		628.0		
	min		94.0		
(25, 11)	ave		406.0		
	max		1788.4		
	min		226.1		
(27, 12)	ave		8956.5		
	max		68598.1		

*Estimated from [33] and the results of qPMS7.

times are shown in seconds for DNA sequences ($|\Sigma| = 4$) and protein sequences ($|\Sigma| = 20$), respectively. From these tables, we can verify that TraverStringDouble and TraverStringRef outperform qPMSPruneI and qPMS7, respectively. In the case of the DNA sequences, TraverStringDouble is 5 or 6 times as fast as qPMSPruneI, and TraverStringRef is 3 or 4 times as fast as qPMS7 when $q = 20$ (TABLE V(a)). The advantage of the proposed algorithms is more apparent when $q = 10$ (TABLE V(b)), probably due to the improvement in Sections IV-A and VI-A. Indeed, TraverStringDouble is 10 times as fast as qPMSPruneI (except for $l = 13$), and TraverStringRef is 5 or 6 times as fast as qPMS7. With regard to the relation between the proposed two algorithms, TraverStringDouble is faster than or at least competitive with TraverStringRef when $l \leq 17$. It will be because the time complexity of TraverStringDouble is $O((N-q+1)NL(L+\log N)n_{\mathrm{B}}(l,\, d))$ in the worst case, while that of TraverStringRef is $O((N - q+1)^2 NL^2(L+\log N)n_{\mathrm{B}}(l,\, d))$ due to the consideration of reference occurrences. Nonetheless, its effect makes TraverStringRef outperform TraverStringDouble when $l \geq 19$.

In the case of the protein sequences (TABLE VI), qPMS7 (and qPMSPruneI, probably) would be slower than was expected because of an additional unnecessary letter ($|\Sigma| = 21$ instead of $|\Sigma| = 20$). Even if it is taken into account, the improvement from the previous algorithms is more than that for the DNA sequences: For the instances with $(l,\, d,\, q) = (17,\, 8,\, 20)$, TraverStringDouble and TraverStringRef are approximately 15 times and 40 times as fast as qPMSPruneI and qPMS7, respectively. A similar tendency is observed in

TABLE VII
EFFECT OF IMPROVEMENTS FOR DNA SEQUENCES
(TRAVERSTRINGDOUBLE)

(a) $(l,\ d) = (17,\ 6)$, $q = 20$ (100% match)

	TravStrS		TravStrS w/o IV-C		TravStrS w/o IV-C, IV-A	
	time	ratio	time	ratio	time	ratio
min	277.9	1.35	372.9	1.82	641.0	3.11
ave	280.7	1.39	379.3	1.87	648.5	3.20
max	283.4	1.39	384.0	1.91	654.1	3.26

(b) $(l,\ d) = (17,\ 5)$, $q = 10$ (50% match)

	TravStrS		TravStrS w/o IV-C		TravStrS w/o IV-C, IV-A	
	time	ratio	time	ratio	time	ratio
min	168.9	1.39	265.4	2.17	347.5	2.84
ave	172.1	1.40	269.2	2.19	353.2	2.87
max	173.8	1.40	272.9	2.20	362.7	2.95

TABLE VIII
EFFECT OF IMPROVEMENTS FOR PROTEIN SEQUENCES
(TRAVERSTRINGDOUBLE)

(a) $(l,\ d) = (17,\ 8)$, $q = 20$ (100% match)

	TravStrS		TravStrS w/o IV-C		TravStrS w/o IV-C, IV-A	
	time	ratio	time	ratio	time	ratio
min	3305.5	1.46	5417.9	2.27	18553.4	7.67
ave	3531.2	1.47	6116.8	2.55	21590.1	9.00
max	4050.4	1.49	7681.9	2.80	27381.6	10.02

(b) $(l,\ d) = (17,\ 7)$, $q = 10$ (50% match)

	TravStrS		TravStrS w/o IV-C		TravStrS w/o IV-C, IV-A	
	time	ratio	time	ratio	time	ratio
min	282.9	1.31	433.6	1.99	786.2	4.09
ave	310.3	1.45	490.2	2.58	923.5	6.05
max	344.6	1.60	532.7	3.18	1029.9	8.93

TABLE IX
EFFECT OF IMPROVEMENTS FOR DNA SEQUENCES
(TRAVERSTRINGREF)

(a) $(l,\ d) = (19,\ 7)$, $q = 20$ (100% match)

	w/o VI-D		w/o VI-C		w/o VI-D, w/o VI-C	
	time	ratio	time	ratio	time	ratio
min	1065.1	1.46	947.7	1.27	1490.1	2.00
ave	1097.0	1.47	979.1	1.32	1536.1	2.06
max	1154.8	1.49	1011.0	1.36	1598.5	2.16

(b) $(l,\ d) = (19,\ 6)$, $q = 10$ (50% match)

	w/o VI-D		w/o VI-C		w/o VI-D, w/o VI-C	
	time	ratio	time	ratio	time	ratio
min	920.4	1.20	977.3	1.26	1276.5	1.65
ave	942.8	1.21	995.3	1.38	1300.4	1.67
max	962.8	1.21	1029.8	1.30	1331.2	1.68

TABLE X
EFFECT OF IMPROVEMENTS FOR PROTEIN SEQUENCES
(TRAVERSTRINGREF)

(a) $(l,\ d) = (19,\ 9)$, $q = 20$ (100% match)

	w/o VI-D		w/o VI-C		w/o VI-D, w/o VI-C	
	time	ratio	time	ratio	time	ratio
min	12.1	1.10	13.6	1.20	16.9	1.54
ave	26.0	1.37	121.2	8.51	360.7	25.49
max	73.2	1.93	259.8	27.64	1085.1	78.97

(b) $(l,\ d) = (19,\ 8)$, $q = 10$ (50% match)

	w/o VI-D		w/o VI-C		w/o VI-D, w/o VI-C	
	time	ratio	time	ratio	time	ratio
min	15.3	1.00	17.1	1.12	17.3	1.14
ave	16.8	1.02	21.8	1.32	33.9	1.94
max	24.6	1.13	34.8	1.87	119.5	5.50

the case of $q = 10$. In addition, TraverStringRef always outperforms TraverStringDouble as qPMS7 outperforms qPMSPruneI. Another difference from the DNA sequences is that the variation of the CPU times is more significant. One reason for it would be that the number of motifs found varies greatly. Indeed, the minimum and maximum numbers of motifs found for the protein instances with $(l,\ d,\ q) = (27,\ 12,\ 10)$ are 566 and 13,953,017, respectively. This fact implies that the number of nodes that should be traversed varies greatly, which resulted in the variation of CPU times. The protein instances in this study seems too challenging: The motifs were so weak that it was difficult to distinguish them from the background noise, and the number of motifs found was affected severely by the randomness of the instances.

B. Effect of Proposed Improvements

Next, the effect of the proposed improvements will be examined. The algorithms without some of the improvements were applied to the instances with $l = 17$ (TraverStringDouble) or $l = 19$ (TraverStringRef). The results are summarized in Tables VII–X. In these tables, "time" denotes the CPU time of the algorithms, and "ratio" is the ratio of the CPU time over that of the original algorithm, i.e. TraverStringDouble in Tables VII and VIII and TraverStringRef in Tables IX and X. A larger "ratio" means that the corresponding improvements removed from the original algorithm are more effective.

The results indicate that all the proposed improvements contribute to the reduction of the computation time. From Tables VII and VIII, we can see that TraverStringDouble with the tree pairing in Section V-A is about 1.4 times as fast as TraverStringSingle without it. The string reordering in Section IV-C is at least as efficient as this improvement, while the strict check in Section IV-A is more efficient for the protein instances. It would be because the improvement by the strict check in pruning subtrees becomes more notable when the size of a search tree increases as $|\Sigma|$ increases. "TravStrS w/o IV-C, IV-A" in Tables VII(a) and VIII(a) is still faster than qPMSPruneI, although the former should be identical to the latter for the instances with $q = N$ because the improvement in Section IV-B is active only when $q < N$. The algorithm seems to have been implemented less efficiently in qPMSPruneI.

With regard to TraverStringRef in Tables IX and X, the position reordering in Section VI-D is competitive with the string reordering in Section VI-C for the DNA sequences, while the effect of the former is not so impressive and the latter is much more effective for the protein instances especially

TABLE XI
EFFECT OF MULTI-THREADING FOR DNA SEQUENCES (4 THREADS)

(a) $q = 20$ (100% match)

(l, d)		TravStrD		TravStrR	
		time	1/ratio	time	1/ratio
	min	1.2	3.11	3.0	3.57
(13, 4)	ave	1.2	3.25	3.1	3.64
	max	1.3	3.39	3.2	3.70
	min	7.4	3.62	12.3	3.55
(15, 5)	ave	7.7	3.67	12.8	3.65
	max	8.1	3.75	13.2	3.69
	min	53.2	3.70	48.3	3.53
(17, 6)	ave	54.3	3.73	50.0	3.59
	max	55.4	3.78	51.2	3.68
	min	409.3	3.69	201.6	3.48
(19, 7)	ave	416.0	3.72	206.4	3.61
	max	426.4	3.74	221.7	3.66
	min	2888.8	3.63	835.5	3.59
(21, 8)	ave	2969.8	3.69	853.5	3.63
	max	3051.0	3.73	875.8	3.67
	min			3443.8	3.62
(23, 9)	ave			3549.3	3.67
	max			3657.6	3.72
	min			14345.2	3.53
(25, 10)	ave			14839.9	3.62
	max			15322.3	3.70

(b) $q = 10$ (50% match)

(l, d)		TravStrD		TravStrR	
		time	1/ratio	time	1/ratio
	min	0.6	3.53	1.5	3.56
(13, 3)	ave	0.6	3.63	1.5	3.64
	max	0.6	3.71	1.5	3.70
	min	3.7	3.47	8.8	3.62
(15, 4)	ave	3.8	3.57	8.8	3.65
	max	3.9	3.64	8.9	3.72
	min	33.7	3.40	45.5	3.62
(17, 5)	ave	34.6	3.55	45.5	3.65
	max	36.2	3.68	46.4	3.69
	min	318.1	3.42	214.4	3.61
(19, 6)	ave	324.8	3.54	214.4	3.64
	max	343.4	3.58	219.1	3.67
	min	2771.6	3.28	1043.3	3.44
(21, 7)	ave	2940.3	3.43	1043.3	3.55
	max	3055.9	3.66	1087.7	3.63
	min			5064.2	3.43
(23, 8)	ave			5064.2	3.54
	max			5260.3	3.63

TABLE XII
EFFECT OF MULTI-THREADING FOR PROTEIN SEQUENCES (4 THREADS)

(a) $q = 20$ (100% match)

(l, d)		TravStrD		TravStrR	
		time	1/ratio	time	1/ratio
	min	26.7	3.60	0.9	3.26
(13, 6)	ave	28.9	3.66	1.0	3.41
	max	31.1	3.70	1.1	3.53
	min	174.8	3.42	1.1	3.35
(15, 7)	ave	190.9	3.49	1.2	3.48
	max	203.9	3.69	1.3	3.62
	min	656.8	3.27	1.4	3.24
(17, 8)	ave	700.1	3.42	1.8	3.43
	max	802.1	3.51	2.8	3.54
	min	2196.6	2.78	2.6	2.99
(19, 9)	ave	3493.3	3.30	5.5	3.32
	max	5475.5	3.54	12.7	3.57
	min			17.4	2.35
(21, 10)	ave			37.1	2.93
	max			88.7	3.39
	min			59.4	1.75
(23, 11)	ave			446.6	2.43
	max			870.2	3.01
	min			779.0	1.83
(25, 12)	ave			3347.3	2.54
	max			11547.6	3.19

(b) $q = 10$ (50% match)

(l, d)		TravStrD		TravStrR	
		time	1/ratio	time	1/ratio
	min	2.0	3.22	0.9	3.24
(13, 5)	ave	2.1	3.31	0.9	3.41
	max	2.2	3.44	1.0	3.65
	min	8.9	2.65	1.5	3.39
(15, 6)	ave	9.8	3.10	1.5	3.53
	max	12.1	3.34	1.6	3.61
	min	62.8	2.74	2.5	3.38
(17, 7)	ave	75.3	3.17	2.6	3.49
	max	95.6	3.67	2.7	3.64
	min	201.4	1.62	4.2	3.41
(19, 8)	ave	574.7	2.96	4.6	3.58
	max	2567.4	3.35	6.1	3.68
	min			7.1	3.23
(21, 9)	ave			8.0	3.51
	max			10.1	3.56
	min			12.8	1.98
(23, 10)	ave			57.5	3.16
	max			317.4	3.61
	min			26.0	2.66
(25, 11)	ave			133.9	3.22
	max			599.5	3.62
	min			80.6	1.67
(27, 12)	ave			3353.5	2.24
	max			22614.1	3.03

when $q = N$. These results imply that TraverStringRef works differently for the DNA sequences and for the protein instances. Indeed, the number of search trees not pruned at the root node in TraverStringRef was 500 on average for the protein instances with $(l, d, q) = (19, 9, 20)$, while that for the DNA instances with $(l, d, q) = (19, 7, 20)$ was 174,000 on average. This difference can be explained by the probability of a random string to fall into $\mathcal{B}(x_0, d) \cap \mathcal{B}(r, d)$. It decreases exponentially as $|\Sigma|$ increases, and a search tree is more likely to be pruned by the feasibility check when x_0 and r are not valid occurrences, because at least $q - 2$ out of the other $N - 2$ input strings should have an occurrence belonging to $\mathcal{B}(x_0, d) \cap \mathcal{B}(r, d)$, whereas it rarely happens by chance.

C. Effect of Multi-threading

The algorithms are easy to parallelize by traversing several search trees in parallel [34]. The results of the algorithms run in 4 threads are shown in Tables XI and XII, where "1/ratio" denotes the ratio between the CPU times in 4 threads and in a single thread (given in Tables V and VI). A larger "1/ratio" means that multi-threading works more effectively. These tables verify that multi-threading makes the algorithms 3 times faster on average, except for the protein instances with $N = 20$, $L = 600$, and $(l, d, q) = (23, 11, 20), (25, 12, 20), (27, 12, 10)$. As explained in Section VII-A, motifs in the protein instances with large l's are so weak that traversing some specific search trees requires a considerable amount of computation time because they include many valid motifs. Thus the CPU time of each thread was not

TABLE XIII
RESULTS FOR REAL DNA DATA SETS

data set	(l, d)	number of motifs found
preproinsulin	(15, 1)	1
	(15, 2)*	379
DHFR	(11, 1)	7
	(11, 2)*	9880
c-fos	(9, 1)	3
	(9, 2)*	7492
metallothionein	(15, 2)*	8
	(15, 3)	3487
Yeast ECB	(16, 3)*	117
	(16, 4)	8648

*Choice of (l, d) in [29].

balanced, which made the multi-threading less advantageous.

D. Results for Real Data Sets

The proposed algorithms were applied to the real DNA data sets, preproinsulin, DHFR, c-fos, metallothionein and Yeast ECB data sets as in [29]. The results are summarized in TABLE XIII. The CPU time was less than 1 s even in a single-thread and so is omitted. In the table, only the number of motifs found is shown to concentrate on the planted motif finding problem itself, although post-processing is necessary to filter the motifs by some scoring schemes in order to predict motifs. It was verified that all the motifs predicted in [29] are among the motifs found by the algorithm when the same (l, d) as in [29] is chosen.

VIII. CONCLUSION

This study proposed several improvements for the existing algorithms for the planted motif search problem. Computational experiments showed that TraverStringDouble and TraverStringRef outperform qPMSPruneI and qPMS7, respectively. Specifically, TraverStringRef is the first algorithm that solves the challenging DNA instances with $(l, d, q) = (25, 10, 20)$ in a reasonable computation time. Although its practical effectiveness is obvious, the theoretical time complexity is not so impressive because it is worse than that of TraverStringDouble. It would be necessary to reduce the time complexity of TraverStringRef in future research. It is possible to apply the tree pairing in Section V-A also to TraverStringRef. However, the result of preliminary experiments was not positive due to complicated pruning conditions. Hence, it is another direction of future research to improve TraverStringRef by the tree pairing in a more sophisticated way.

REFERENCES

[1] M. Frances and A. Litman, "On covering problems of codes," *Theory of Computing Systems*, vol. 30, no. 2, pp. 113–119, Apr. 1997.

[2] M. Li, B. Ma and L. Wang, "On the closest string and substring problems," *Journal of the ACM*, vol. 49, no. 2, pp. 157–171, Mar. 2002.

[3] J. Gramm, R. Niedermeier, and P. Rossmanith, "Fixed-parameter algorithms for closest string related problems," *Algorithmica*, vol. 37, no. 1, pp. 25–42, Sep. 2003.

[4] P. A. Evans and Andrew D. Smith, "Complexity of approximating closest substring problems," *Proc. Symposium on Fundamentals of Computation Theory (FCT 2003)*, LNCS 2751, Springer, pp. 210–221, 2003.

[5] D. Marx, "Closest substring problems with small distances," *SIAM Journal on Computing*, vol. 38, no. 4, pp. 1382–1410, 2008.

[6] J. Wang, J. Chen, and M. Huang, "An improved lower bound on approximation algorithms for the closest substring problem," *Information Processing Letters*, vol. 107, no. 1, pp. 24–28, June 2008.

[7] B. Ma and X. Sun, "More efficient algorithms for closest string and substring problems," *SIAM Journal on Computing*, vol. 39, no. 4, pp. 1432–1443, 2009.

[8] Z.-Z. Chen and L. Wang, "Fast exact algorithms for the closest string and substring problems with application to the planted (L, d)-motif model, " *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 8, no. 5, pp. 1400–1410, Sep.-Oct. 2011.

[9] P. A. Pevzner and S.-H. Sze, "Combinatorial approaches to finding subtle signals in DNA sequences," *Proc. 8th International Conference on Intelligent Systems for Molecular Biology (ISMB2000)*, pp. 269–278, Aug. 2000.

[10] S. Liang, "cWINNOWER algorithm for finding fuzzy DNA motifs," *Proc. Computational Systems Bioinformatics Conference (CSB2003)*, pp. 260–265, Aug. 2003.

[11] S.-H. Sze, S. L. Lu, and J. Chen, "Integrating sample-driven and pattern-driven approaches in motif finding," *Proc. 4th Workshop on Algorithms in Bioinformatics (WABI2004)*, LNCS 3240, Springer, pp. 438–449, Sep. 2004.

[12] X. Yang and J. C. Rajapakse, "Graphical approach to weak motif recognition," *Genome Informatics*, vol. 15, no. 2, pp. 52–62, Dec. 2004.

[13] H. Q. Sun, M. Y. H. Low, W. J. Hsu, and J. C. Rajapakse, "RecMotif: A novel fast algorithm for weak motif discovery," *BMC Bioinformatics*, vol. 11 (Suppl. 11), art. no. S8, 2010.

[14] H. Q. Sun, M. Y. H. Low, W. J. Hsu, and J. C. Rajapakse, "ListMotif: A time and memory efficient algorithm for weak motif discovery," *Proc. IEEE International Conference on Intelligent Systems and Knowledge Engineering (ISKE2010)*, pp. 254–260, Nov. 2010.

[15] H. Q. Sun, M. Y. H. Low, W. J. Hsu, C. W. Tan, and J. C. Rajapakse, "Tree-structured algorithm for long weak motif discovery," *Bioinformatics*, vol. 27, no. 19, pp. 2641–2647, Oct. 2011.

[16] M.-F. Sagot, "Spelling approximate repeated or common motifs using a suffix tree," *Proc. Third Latin American Theoretical Informatics Symposium (LATIN'98)*, LNCS 1380, Springer, pp. 374–390, Apr. 1998.

[17] G. Pavesi, G. Mauri, and G. Pesole, "An algorithm for finding signals of unknown length in DNA sequences," *Bioinformatics*, vol. 17 (Suppl. 1), pp. S207–S214, June 2001.

[18] E. Eskin and P. A. Pevzner, "Finding composite regulatory patterns in DNA sequences," *Bioinformatics*, vol. 18 (Suppl. 1), pp. S354-S363, July 2002.

[19] P. A. Evans and A. D. Smith, "Toward optimal motif enumeration," *8th International Workshop on Algorithms and Data Structures (WADS2003)*, LNCS 2748, Springer, pp. 47–58, July-Aug. 2003.

[20] N. Pisanti, A. M. Carvalho, L. Marsan, and M.-F. Sagot, "RISOTTO: Fast extraction of motifs with mismatches," *Proc. 7th Latin American Theoretical Informatics Symposium (LATIN2006)*, LNCS 3887, Springer, pp. 757–768, Mar. 2006.

[21] F. Y. L. Chin and H. C. M. Leung, "Voting algorithms for discovering long motifs," *Proc. Third Asia-Pacific Bioinformatics Conference (APBC2005)*, pp. 261–271, Jan. 2005.

[22] S. Rajasekaran, S. Balla, and C.-H. Huang, "Exact algorithms for planted motif problems," *Journal of Computational Biology*, vol. 12, no. 8, pp. 1117–1128, Oct. 2005.

[23] S.-H. Sze and X. Zhao, "Improved Pattern-Driven Algorithms for Motif Finding in DNA Sequences," LNCS 4023, Springer, pp. 198–211, 2006.

[24] J. Davila, S. Balla, and S. Rajasekaran, "Space and time efficient algorithms for planted motif search," *Proc. 2nd International Workshop on Bioinformatics Research and Applications (IWBRA'06)*, pp. 822–829, May 2006.

[25] P. P. Kuksa and V. Pavlovic, "Efficient motif finding algorithms for large-alphabet inputs," *BMC Bioinformatics*, vol 11 (Suppl. 8), art. no. S1, May 2010.

[26] S. Rajasekaran and H. Dinh, "A speedup technique for (l, d) motif finding algorithms," *BMC Research Notes*, vol. 4, art. no. 54, Mar. 2011.

[27] H. Dinh, S. Rajasekaran, V. K. Kundeti, "PMS5: An efficient exact algorithm for the (l, d)-motif finding problem," *BMC Bioinformatics*, vol. 12, art. no. 410, Oct. 2011.

[28] S. Bandyopadhyay, S. Sahni, and S Rajasekaran, "PMS6: A fast algorithm for motif discovery," *Proc. 2nd IEEE International Conference on Computational Advances in Bio and Medical Sciences (ICCABS2012)*, Feb. 2012. doi:10.1109/ICCABS.2012.6182627.

[29] Q. Yu, H. Huo, Y. Zhang, and H. Guo, "PairMotif: a new pattern-driven algorithm for planted (l, d) DNA motif search, " *PLoS One*, vol. 7, art. no. e48442, Oct. 2012.

[30] J. Davila, S. Balla, and S. Rajasekaran, "Fast and practical algorithms for planted (l, d) motif search," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 4, no. 4, pp. 544–552, Oct.-Dec. 2007.

[31] J. Davila, S. Balla, and S. Rajasekaran, "Pampa: An improved branch and bound algorithm for planted (l, d) motif search," BECAT Technical Report, School of Engineering, University of Connecticut, BECAT/CSE-TR-07-5, 2007. http://becat.engr.uconn.edu/becat_technical_reports/BECAT-CSE-TR-07-5.pdf.

[32] D. Sharma and S. Rajasekaran, "A simple algorithm for (l, d) motif search," *Proc. IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB'09)*, pp. 148–154, Mar.-Apr. 2009.

[33] H. Dinh, S. Rajasekaran, and J. Davila, "qPMS7: a fast algorithm for finding (l, d)-motifs in DNA and protein sequences," *PLoS One*, vol. 7, no. 7, art. no. e41425, July 2012.

[34] M. M. Abbas, M Abouelhoda, and H. M. Bahig, "A hybrid method for the exact planted (l, d) motif finding problem and its parallelization," *BMC Bioinformatics*, vol. 13 (Suppl. 17), art. no. S10, Dec. 2012.

Shunji Tanaka (M'05) received BE, ME and PhD (Eng) degrees in electrical engineering from Kyoto University in 1993, 1995 and 2000, respectively. He is an Associate Professor with the Institute for Liberal Arts and Sciences at Kyoto University since 2014. His current research interests include exact and heuristic algorithms for real-world problems such as production scheduling, elevator group control, bioinformatics and so on.