An Efficient Exact Algorithm for the Motif Stem Search Problem over Large Alphabets

Qiang Yu, Hongwei Huo, Jeffrey Scott Vitter, Jun Huan, and Yakov Nekrich

Abstract—In recent years, there has been an increasing interest in planted (*l*, *d*) motif search (PMS) with applications to discovering significant segments in biological sequences. However, there has been little discussion about PMS over large alphabets. This paper focuses on motif stem search (MSS), which is recently introduced to search motifs on large-alphabet inputs. A motif stem is an *l*-length string with some wildcards. The goal of the MSS problem is to find a set of stems that represents a superset of all (*l*, *d*) motifs present in the input sequences, and the superset is expected to be as small as possible. The three main contributions of this paper are as follows: (1) We build motif stem representation more precisely by using regular expressions. (2) We give a method for generating all possible motif stems without redundant wildcards. (3) We propose an efficient exact algorithm, called StemFinder, for solving the MSS problem. Compared with the previous algorithms, StemFinder runs much faster and first solves the (17, 8), (19, 9) and (21, 10) challenging instances on protein sequences; moreover, StemFinder reports fewer stems which represent a smaller superset of all (*l*, *d*) motifs. StemFinder is freely available at http://sites.google.com/site/feqond/stemfinder.

Index Terms—Exact algorithms, motif stem search, planted (*l*, *d*) motif search

1 Introduction

Motif search is to find short similar sequence segments in a given set of sequences over an alphabet $\sum$, which plays an important role in discovering significant segments in biological sequences, such as transcription factor binding sites in DNA sequences [1]. The planted (*l*, *d*) motif search (PMS) [2] is a widely accepted formulation of the problem. A (*l*, *d*) motif is an *l*-mer (i.e., an *l*-length string over $\sum$) that spans all input sequences with up to *d* mismatches. The goal of the PMS problem is to find all (*l*, *d*) motifs present in the given sequences, and the PMS problem has been proven to be NP-complete [3].

The key to motif search lies in two points: a) how to represent the sequence motif using an appropriate model; b) how to design an efficient motif search algorithm. The most commonly used motif models are position weight matrices (PWM) [4] and consensus sequences [5]. Based on these two motif models, numerous motif search algorithms have been proposed.

The algorithms that model motifs using PWM usually employ statistical techniques [6], [7], [8]. These algorithms can report results in a short time, but cannot guarantee a global optimum. The exact algorithms, which use consensus sequences to represent motifs, are guaranteed to report all (*l*, *d*) motifs by traversing the whole search space. Most exact algorithms are pattern-driven. They take all string patterns of length *l* over $\sum$ as candidate motifs, and output the patterns that can span all input sequences. Typical pattern-driven algorithms aim to reduce candidate motifs through various means [9], [10], [11], [12], [13], [14], [15], [16]. Some other pattern-driven algorithms represent the input sequences as a suffix tree to accelerate the verification of candidate motifs [17], [18], [19]. The initial search space of pattern-driven algorithms is $O(|\sum|^l)$, which grows dramatically with the increase of $|\sum|$. Therefore, most existing exact algorithms are designed just for searching motifs in DNA sequences where $|\sum| = 4$, and they cannot search low-conserved motifs within an acceptable time in the data sets over large alphabets, such as the protein data sets where $|\sum| = 20$.

To improve the efficiency of the exact algorithms over large alphabets, Kuksa and Pavlovic [20] introduced the concept of motif stem in the field of motif search. A motif stem is an *l*-length string that may contain some wildcards, and it represents a set of candidate motifs. For example, assume that A*GT is a motif stem over $\sum$ = {A, G, C, T} where * denotes a wildcard. Then, A*GT represents four candidate motifs AAGT, AGGT, ACGT and ATGT. The goal of motif stem search (MSS) is to find a set of stems that represents a superset of all (*l*, *d*) motifs, and the superset is expected to be as small as possible. The time complexity of the MSS algorithms does not grow with the increase of the size of the alphabet, since in generating candidate motifs, the operation of expanding some positions to multiple characters over $\sum$ is replaced by placing wildcards in these positions.

MSS algorithms are the main subject of this paper. Stemming [20] is the first MSS algorithm, and it works as follows: first, select the *l*-mers that may be motif instances (i.e., motif occurrences) to form a set *I*; second, for each pair of *l*-mers *x* and *x*' in *I*, generate motif stems from *x* and *x*' by placing wildcards; third, verify motif stems and output the ones that occur in each input sequence. In a

- *Q. Yu is with the School of Computer Science and Technology, Xidian University, Xi'an, 710071, China. E-mail: qyu@mail.xidian.edu.cn.*
- *H. Huo is with the School of Computer Science and Technology, Xidian University, Xi'an, 710071, China, and the Information and Telecommunication of Technology Center, The University of Kansas, Lawrence, 66047, USA. E-mail: hwhuo@mail.xidian.edu.cn.*
- *J.S. Vitter, J. Huan and Y. Nekrich are with the Information and Telecommunication of Technology Center, The University of Kansas, Lawrence, 66047, USA. E-mail: {jsv, jhuan, yakov}@ittc.ku.edu.*

recent work [21], more efficient MSS algorithms MSS1 and MSS2 are proposed. MSS1 constructs a smaller set I and generates fewer stems than Stemming; also, MSS1 employs a different method for placing wildcards. MSS2 is an improvement of MSS1 obtained by accelerating the calculation of Hamming distances from the l-mers in an input sequence to that in another input sequence.

Despite the efforts for motif stem search, current MSS algorithms have several notable limitations. First, motif stems cannot be represented precisely with typical wildcards, since the wildcard * matches any character over $\sum$. For example, when we hope a stem only matches AAGT or AGGT, the stem A*GT fails to do so. The second limitation comes from the methods used to generate motif stems in current MSS algorithms. The current generation methods either miss some possible motif stems or place redundant wildcards, which is analyzed in detail in Section 6.1. Third, there is great potential for designing more efficient stem search algorithms. For example, as reported in [21], the fastest stem search algorithm MSS2 is only able to solve the challenging instance (11, 5) over $|\sum| = 20$ within four hours, even if it does not perform a post-processing (verifying candidate stems). Also, the reported stems can be further reduced to represent a smaller superset of all (l, d) motifs.

In this paper, we propose a new motif stem search algorithm named StemFinder that overcomes these limitations. To represent stems more precisely, we write stems as regular expressions by replacing typical wildcards * with the negative character sets [^]. A negative character set [^] matches any character not enclosed; for example, [^CT] represents any single character over $\sum$ except for C and T. StemFinder runs much faster than the previous stem search algorithms, and reports fewer stems corresponding to a smaller superset of all (l, d) motifs.

The rest of the paper is organized as follows. Section 2 gives the notations and problem definition, and reviews the previous stem generation methods. Section 3 describes how to represent motif stems using regular expressions. Section 4 introduces the method for generating motif stems. In Section 5, several techniques used in StemFinder as well as the StemFinder algorithm are described. Then, Section 6 presents the results and discussion. Finally, we conclude the paper in Section 7.

2 PRELIMINARIES

2.1 Notations and Problem Definition

In this paper, an l-mer is an l-length string over an alphabet $\sum$ without wildcards; a motif stem is an l-length string over the same alphabet that may contain wildcards. We say an l-mer x is covered by a motif stem s, if x is in the set of l-mers represented by s. Hereafter, a motif stem is called simply as a stem.

The notations used in this paper are summarized in Table 1. The probability p_k' and p_k are calculated by (1) and (2), respectively. The notations $R(i)$, $N_s(i)$ and $N_{rs}(i)$ imply the dependence of their values on the Hamming distance i between two l-mers, which will be discussed in detail in Section 4.

TABLE 1
Notations Used in This Paper

Notation	Explanation
$\|x\|$	The size of a set x or the length of a string x.
$P_m(x, x')$	The positions in the matching region of two l-mers x and x'. $P_m(x, x') = \{i: 1 \le i \le l, x[i] = x'[i]\}$.
$P_n(x, x')$	The positions in the non-matching region of two l-mers x and x'. $P_n(x, x') = \{i: 1 \le i \le l, x[i] \ne x'[i]\}$.
$P_{mn}(x, x', y)$	The positions where x matches x', and y matches neither x nor x', for the given three l-mers x, x' and y. $P_{mn}(x, x', y) = \{i: 1 \le i \le l, x[i] = x'[i], y[i] \ne x[i]$ and $y[i] \ne x'[i]\}$.
$P_{nn}(x, x', y)$	The positions where x, x' and y are mismatched with each other, for the given three l-mers x, x' and y. $P_{nn}(x, x', y) = \{i: 1 \le i \le l, x[i] \ne x'[i], y[i] \ne x[i]$ and $y[i] \ne x'[i]\}$.
$d_H(x, x')$	The Hamming distance between two l-mers x and x'. $d_H(x, x') = \|P_n(x, x')\| = l - \|P_m(x, x')\|$.
$M_d(x, x')$	The common d-neighbors of two l-mers x and x'. $M_d(x, x') = \{y: \|y\| = \|x\| = \|x'\|, d_H(y, x) \le d$ and $d_H(y, x') \le d\}$.
$C(x, S_i)$	The l-mers in the sequence S_i that are $2d$-neighbors of the l-mer x. $C(x, S_i) = \{y: \|y\| = \|x\|, y \in_l S_i$ and $d_H(y, x) \le 2d\}$.
$C(x, x', S_i)$	The l-mers in the sequence S_i that are common $2d$-neighbors of the l-mers x and x'. $C(x, x', S_i) = \{y: \|y\| = \|x\| = \|x'\|, y \in_l S_i, d_H(y, x) \le 2d$ and $d_H(y, x') \le 2d\}$.
p_k'	The probability that the Hamming distance between a fixed l-mer and a random l-mer is equal to k.
p_k	The probability that the Hamming distance between a fixed l-mer and a random l-mer is less than or equal to k.
$R(i)$	Given two l-mers x and x' with $d_H(x, x') = i$ and an arbitrary l-mer $y \in M_d(x, x')$, $R(i)$ denotes the set of all possible combinations of $\|P_{mn}(x, x', y)\|$ and $\|P_{nn}(x, x', y)\|$.
$N_s(i)$	The number of stems generated from two l-mers x and x' with $d_H(x, x') = i$.
$N_{rs}(i)$	The number of rough stems generated from two l-mers x and x' with $d_H(x, x') = i$. The concept of rough stem is described in Section 4.

$$p_k' = \binom{l}{k} \times \frac{(|\Sigma| - 1)^k}{|\Sigma|^l} \quad (1)$$

$$p_k = \sum_{i=0}^{k} p_k' \quad (2)$$

Problem Definition: Motif Stem Search (MSS) [21]. Given a set of n-length sequences $\{S_1, S_2, \ldots, S_t\}$ over an alphabet $\sum$ and nonnegative integers l and d, satisfying $0 \le d < l < n$, a (l, d) motif is an l-mer m such that each sequence S_i contains an l-mer m_i differing from m in at most d positions. The MSS problem is to find a set of stems so that the set of l-mers represented by these stems is a superset of all (l, d) motifs present in the t sequences.

There are two key indicators used to assess the MSS algorithms. One is the running time. The other is the number of l-mers covered by the reported stems. Although the MSS algorithms should be guaranteed to report the stems representing a superset of all (l, d) motifs, the size of the superset is not fixed due to different methods used to generate stems. Therefore, an efficient MSS algorithm indicates that it not only runs faster but also reports the stems covering fewer l-mers.

2.2 Previous Stem Generation Methods

This section briefly reviews the stem generation methods used in existing MSS algorithms, Stemming [20] and

MSS1/MSS2 [21]. The stem generation method is the core module of an MSS algorithm and it affects the number of l-mers covered by the reported stems.

Existing MSS algorithms as well as StemFinder first select multiple pairs of l-mers from input sequences. Then, for each selected pair of l-mers x and x', they generate the candidate stems by placing wildcards in x, differing only in the specific meanings of the used wildcards and the ways that wildcards are placed.

Stemming allows the wildcard to match any character over $\sum$ and generate stems by changing x as follows: if $d_H(x, x') \le d$, set i ($0 \le i \le d_H(x, x')$) positions in $P_n(x, x')$ as in x', place a ($0 \le a \le d_H(x, x') - i$) wildcards in the remaining $d_H(x, x') - i$ positions in $P_n(x, x')$, and place β ($0 \le \beta \le d - max(d_H(x, x') - i, a + i)$) wildcards in $P_m(x, x')$; otherwise, set i ($d_H(x, x') - d \le i \le d$) positions in $P_n(x, x')$ as in x', and place a ($0 \le a \le d - i$) wildcards in the remaining $d_H(x, x') - i$ positions in $P_n(x, x')$. We find that Stemming in this way cannot generate all possible candidate stems in some cases, and we give an example in Section 6.1.

In MSS1/MSS2, a wildcard matches any character over $\sum$ except for the character in the corresponding position of x. MSS1/MSS2 generates stems s by placing a wildcards in $P_n(x, x')$ of x and β wildcards in $P_m(x, x')$ of x. The range of a is considered as follows. If $d_H(x, x') \le d$, it is clear that a can vary from 0 to $d_H(x, x')$, namely $0 \le a \le d_H(x, x')$; otherwise, at least $d_H(x, x') - d$ wildcards have to be placed in $P_n(x, x')$, namely $d_H(x, x') - d \le a \le d$, to make $d_H(s, x) \le d$ and $d_H(s, x') \le d$ satisfied. Simultaneously, the range of β is determined by satisfying the same condition that $d_H(s, x) \le d$ and $d_H(s, x') \le d$, namely $a + \beta \le d$ and $(d_H(x, x') - a) + \beta \le d$, so the maximum value of β is $min\{d - a, d - (d_H(x, x') - a)\}$. Although MSS1/MSS2 can generate all possible candidate stems, it may place redundant wildcards, and thus the reported stems cover more unnecessary l-mers. The associated example and more detailed analysis are given in Section 6.1.

3 Stem Representation

In the previous MSS algorithms, for a stem s of length l, $s[i]$ ($0 \le i \le l$) is either an exact character over $\sum$ or a typical wildcard *. To represent stems more precisely, we introduce two new regular expression operators, namely the negative character set [^] and the choice operator |. Both of the two operators are used in a rough stem, which will be discussed in Section 4. Only the former operator is involved in the representation of a final stem, which is discussed in this section.

Specifically, we describe how to represent stems using regular expressions, by analyzing the relationships among three characters in a column of the alignment of three l-mers. Given three l-mers x, x' and y, assume that y is an arbitrary candidate motif shared by x and x', namely $y \in M_d(x, x')$. For the ith ($1 \le i \le l$) column of the three l-mers, there are five possible cases for the relationships among $x[i]$, $x'[i]$ and $y[i]$, as shown in Table 2. Under different cases, the number of characters matched by $y[i]$ is also different. For the Cases 1, 3 and 4, $y[i]$ corresponds to a unique character. For the Cases 2 and 5, $y[i]$ matches $|\sum|$ – 1 and $|\sum|$ – 2 characters, respectively. When $y[i]$ corresponds to multiple characters, we represent $y[i]$ using the negative character set [^]. Specifically, for Case 2, $y[i]$ is represented as $[^x[i]]$, which matches any character in $\sum$ excluding $x[i]$; for Case 5, $y[i]$ is represented as $[^x[i]x'[i]]$, which matches any character in $\sum$ excluding $x[i]$ and $x'[i]$.

TABLE 2
Represent $y[i]$ Using $x[i]$ and $x'[i]$

Relationships among $x[i]$, $x'[i]$ and $y[i]$	#[a]	$y[i]$
Case 1: $x[i] = x'[i] = y[i]$	1	$x[i]$
Case 2: $x[i] = x'[i]$, $y[i] \ne x[i]$ and $y[i] \ne x'[i]$	$\|\sum\| - 1$	$[^x[i]]$
Case 3: $x[i] \ne x'[i]$, $y[i] = x[i]$ and $y[i] \ne x'[i]$	1	$x[i]$
Case 4: $x[i] \ne x'[i]$, $y[i] \ne x[i]$ and $y[i] = x'[i]$	1	$x'[i]$
Case 5: $x[i] \ne x'[i]$, $y[i] \ne x[i]$ and $y[i] \ne x'[i]$	$\|\sum\| - 2$	$[^x[i]x'[i]]$

[a]*The number of characters matched by* $y[i]$.

According to the analysis above, assume that the regular expression s is a stem obtained from two l-mers x and x'. Then the ith position of s must fall into one of the three patterns: a specific character ($x[i]$ or $x'[i]$), a negative character set $[^x[i]]$ or a negative character set $[^x[i]x'[i]]$. In all the l positions of s, if there is a position i that corresponds to $[^x[i]]$ or $[^x[i]x'[i]]$, then s represents multiple candidate motifs; otherwise, s represents a single candidate motif. Let (l, d) = (7, 3), x = AAAAGGG and x' = AAAACCC; four possible stems are AAAAGGC, AAAAG[^GC]C, AA[^A]AGCC and A[^A]AAGC[^GC]. The method for generating stems from x and x' is described in the next section.

There are two benefits for the use of regular expressions to represent stems. On the one hand, stems are represented more precisely by using negative character sets than using typical wildcards, since the former match the interest characters and the latter match any character. On the other hand, stems can be converted into finite automatas [22] so that they can be verified efficiently in stem search.

4 Stem Generation

This section gives the method for generating all possible stems s from two given l-mers x and x'. Assume that an l-mer y is an arbitrary candidate motif covered by s, and y satisfies $d_H(y, x) \le d$ and $d_H(y, x') \le d$. The key of the generation method is to determine the positions corresponding to Case 2 and the positions corresponding to Case 5, namely $P_{mn}(x, x', y)$ and $P_{nn}(x, x', y)$, since these positions of s will be represented as negative character sets. More precisely, we obtain all possible combinations of $|P_{mn}(x, x', y)|$ and $|P_{nn}(x, x', y)|$, namely $R(d_H(x, x'))$. Hereafter, $P_m(x, x')$, $P_n(x, x')$, $P_{mn}(x, x', y)$ and $P_{nn}(x, x', y)$ are denoted simply as P_m, P_n, P_{mn} and P_{nn}, respectively.

The possible combinations of $|P_{mn}|$ and $|P_{nn}|$ are calculated as follows. First, since P_{mn} is a subset of P_m, we have $0 \le |P_{mn}| \le |P_m|$, namely $0 \le |P_{mn}| \le l - d_H(x, x')$; similarly, $0 \le |P_{nn}| \le d_H(x, x')$. Second, since all positions i in P_{mn} satisfy $y[i] \ne x[i]$ and $y[i] \ne x'[i]$, we have $|P_{mn}| \le d$, which is necessary for $d_H(y, x) \le d$ and $d_H(y, x') \le d$ to be satisfied; similarly, $|P_{nn}| \le d$. Third, $d_H(y, x) + d_H(y, x') \le 2d$ where $d_H(y, x) + d_H(y, x')$ can be represented as $2|P_{mn}|$

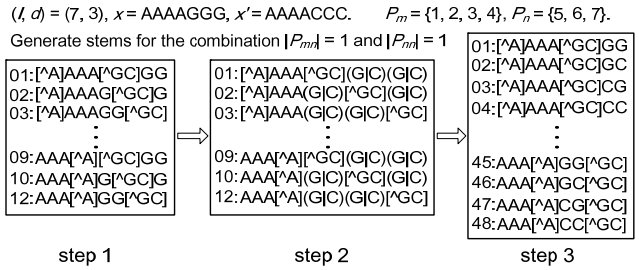

Fig. 1. An example for generating stems from two *l*-mers.

$+ 2|P_{nn}| + (d_H(x, x') - |P_{nn}|) = 2|P_{mn}| + |P_{nn}| + d_H(x, x')$, so we have $2|P_{mn}| + |P_{nn}| + d_H(x, x') \le 2d$, namely $2|P_{mn}| + |P_{nn}| \le 2d - d_H(x, x')$. Taking these considerations into account, we obtain the following inequalities:

$$\begin{cases} 0 \le |P_{mn}| \le \min\{l - d_H(x,x'), d\}, \\ 0 \le |P_{nn}| \le \min\{d_H(x,x'), d\}, \\ 2|P_{mn}| + |P_{nn}| \le 2d - d_H(x,x'). \end{cases} \quad (3)$$

Obviously, the values of $|P_{mn}|$ and $|P_{nn}|$, which depend on $d_H(x, x')$, can be calculated by solving (3). That is, $R(d_H(x, x')) = \{<|P_{mn}|, |P_{nn}|>: |P_{mn}|$ and $|P_{nn}|$ satisfy (3)$\}$. For example, when $(l, d) = (7, 3)$ and $d_H(x, x') = 3$, all possible combinations of $|P_{mn}|$ and $|P_{nn}|$ form $R(3) = \{<0, 0>, <0, 1>, <0, 2>, <0, 3>, <1, 0>, <1, 1>\}$.

For each possible combination of $|P_{mn}|$ and $|P_{nn}|$, we generate the stems from x and x' by rewriting a string s that is initialized as x, through three steps shown in Fig. 1. (1) Select $|P_{mn}|$ positions from P_m, and change the character of s in each selected position i to $[^x[i]]$; at the same time, select $|P_{nn}|$ positions from P_n, and change the character of s in each selected position i to $[^x[i]x'[i]]$. (2) For each position i that is in P_n but not selected in the previous step, change the corresponding character of s to $x[i]|x'[i]$. (3) For each position i of s that corresponds to $x[i]|x'[i]$, expand it to $x[i]$ and $x'[i]$.

The stems obtained in the second step are called *rough stems*, in regard to the stems obtained in the last step. Note that, for each position i in P_n except for the positions selected in the first step, the character of s (denoted by c) can be either $x[i]$ (corresponding to Case 3 in Table 2) or $x'[i]$ (corresponding to Case 4 in Table 2). In a rough stem, such character c is represented using the choice operator of regular expressions, namely $x[i]|x'[i]$. In a stem, such character c is represented exactly as $x[i]$ or $x'[i]$. Since the number of such characters c is $w = d_H(x, x') - |P_{nn}|$, each rough stem can be decomposed into 2^w stems. For example, in Fig. 1, each rough stem obtained through step 2 corresponds to $w = 2$, and it is decomposed or expanded to four stems through step 3.

Given two l-mers x and x' with $d_H(x, x') = i$, the number of generated rough stems, $N_{rs}(i)$, is calculated by (4), which sums the number of rough stems over all possible combinations of $|P_{mn}|$ and $|P_{nn}|$; similarly, the number of generated stems, $N_s(i)$, is calculated by (5).

$$N_{rs}(i) = \sum_{<\alpha,\beta>\in R(i)} \binom{l-i}{\alpha} \times \binom{i}{\beta} \quad (4)$$

$$N_s(i) = \sum_{<\alpha,\beta>\in R(i)} \binom{l-i}{\alpha} \times \binom{i}{\beta} \times 2^{i-\beta} \quad (5)$$

5 Stem Search Algorithm

The framework of StemFinder is: extract some pairs of l-mers from input sequences to form a set I so that at least one pair of motif instances is included; then, for each pair of l-mers in I, generate and verify stems; finally, report all valid stems. Although this framework is somewhat similar to that of the previous algorithms (Stemming [20] and MSS1/MSS2 [21]), StemFinder performs more efficiently by introducing several techniques described in Section 5.1 to 5.3. The whole algorithm of StemFinder, as well as its complexity analysis, is presented in Section 5.4.

5.1 Constructing Set I

The set I is composed of pairs of l-mers coming from different input sequences, and contains at least one element that is a pair of motif instances. According to the problem definition, there is a motif instance in each input sequence, and the Hamming distance between any two motif instances is less than or equal to $2d$. Thus, a typical method for constructing the set I is: for each l-mer x in S_1, select a reference sequence S_r from $\{S_2, \ldots, S_t\}$; for each l-mer x' in S_r, if $d_H(x, x') \le 2d$, namely $x' \in C(x, S_r)$, then add the pair of l-mers x and x' to the set I.

Furthermore, a good set I is composed of pairs of l-mers that correspond to as few stems as possible, which depends on how to select the reference sequence S_r for each l-mer x in S_1. Unlike [21], in which the selected S_r is the sequence in $\{S_2, \ldots, S_t\}$ that contains the minimum number of l-mers x' satisfying $d_H(x, x') \le 2d$, we select S_r based on the following observation.

Observation 1. *For two l-mers x and x', both the number of generated stems and the number of generated rough stems are different for distinct* $d_H(x, x')$.

We mainly consider the number of rough stems, since in the StemFinder algorithm we verify rough stems firstly and avoid the verification of most stems using pruning,

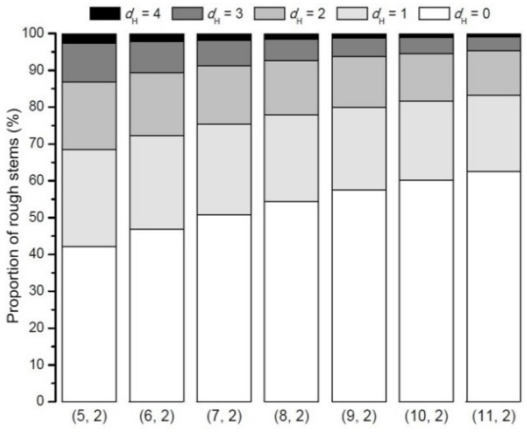

Fig. 2. Proportion of rough stems under different Hamming distances.

which is discussed in detail in Section 5.3. Fig. 2 shows the proportion of rough stems generated from two l-mers under different Hamming distances. Each stacked column in the figure corresponds to a (l, 2) problem instance with Hamming distances ranging from 0 to 4 ($2d$). The proportion of rough stems under the Hamming distance i ($0 \le i \le 2d$) is defined to be $N_{rs}(i)/N_{total}$, where $N_{total} = \sum N_{rs}(j)$ for $0 \le j \le 2d$. These (l, 2) instances represent the general cases, since they cover the instances from a low degenerate case (11, 2) to a highly degenerate case (5, 2). We can see that the number of rough stems differs greatly for distinct Hamming distances. Particularly, the number of rough stems for $d_H(x, x') = 0$ is ten times greater than the number of rough stems for $d_H(x, x') = 4$. Thus, the reference sequence S_r with a small value of $|C(x, S_r)|$ may not correspond to a small number of rough stems. For example, for two reference sequences S_{r1} and S_{r2} of the l-mer x for the problem instance (7, 2), assume that $C(x, S_{r1}) = \{x_1, x_2, x_3\}$, $C(x, S_{r2}) = \{x_4\}$, $d_H(x, x_1) = 4$, $d_H(x, x_2) = 4$, $d_H(x, x_3) = 3$ and $d_H(x, x_4) = 0$. Although $|C(x, S_{r1})| = 3$ is larger than $|C(x, S_{r2})| = 1$, the number of rough stems corresponding to S_{r1} is much smaller than that corresponding to S_{r2}.

In the light of the above, we select the reference sequence S_r for the l-mer x by minimizing the right side of (6). The selected S_r is the sequence in $\{S_2, \ldots, S_t\}$ that corresponds to the minimum number of rough stems.

$$\sum_{x' \in C(x, S_r)} N_{rs}(d_H(x, x')) = \min_{2 \le i \le t} \sum_{x' \in C(x, S_i)} N_{rs}(d_H(x, x')) \qquad (6)$$

5.2 Verifying Stems

We convert stems into deterministic finite automatas (DFA) and verify stems by scanning input sequences to check whether there is an occurrence of the verified stem in each sequence.

At first, let us determine the objects scanned by the DFA. Assume that s is a stem generated from l-mers x and x'. Then, only the l-mers z with $d_H(z, x) \le 2d$ and $d_H(z, x') \le 2d$ in input sequences could be the occurrences of s. Since the value of p_{2d} is small and it is approximately equal to 10^{-2} or 10^{-3} for common problem instances, the number of l-mers in input sequences that could be the occurrences of s is also small. Thus, we only need to check the l-mers that could be the occurrences of s, rather than the whole input sequences. Specifically, for an input sequence S_i, the scanned objects are the l-mers in $C(x, x', S_i)$.

Next, we introduce how to construct a DFA from a stem s and how to perform scan. Scanning an l-mer z is to check whether there are at most d positions where s mismatches z. As shown in Fig. 3(a), for the DFA directly constructed from a stem, once a mismatch occurs in some position, the DFA will immediately end the matching process. In order to allow at most d mismatches, we add a counter initialized as 0 to the DFA, as shown in Fig. 3(b). For any state u except for the end state, the next state is always $u + 1$ via any character c. If the character c is matched, the counter remains unchanged; otherwise, it is incremented by one. When arriving at the end state, if the counter is less than or equal to d, then the scanned l-mer is an occurrence of the stem s.

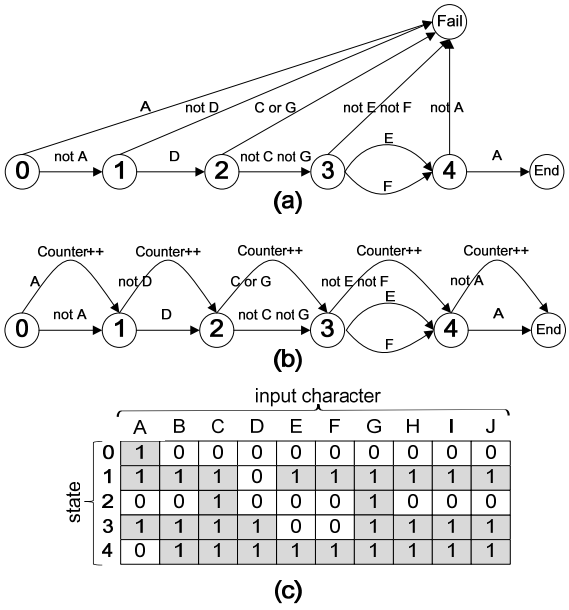

Fig. 3. DFA of the stem [^A]D[^CG](E|F)A (a) Initial DFA (b) DFA with counter (c) Bitmap of DFA with counter.

TABLE 3
Rules of Generating the Bitmap for a Stem s

$s[i]$	Generation Rule
[^$x[i]$]	Perform bitwise NOT on $T[i-1]$, and then set $T[i-1][x[i]]$ as 1
[^$x[i]x'[i]$]	Perform bitwise NOT on $T[i-1]$, and then set both $T[i-1][x[i]]$ and $T[i-1][x'[i]]$ as 1
$x[i]\|x'[i]$	Set both $T[i-1][x[i]]$ and $T[i-1][x'[i]]$ as 0
$x[i]$	Set $T[i-1][x[i]]$ as 0
$x'[i]$	Set $T[i-1][x'[i]]$ as 0

In order to scan l-mers more efficiently, we use a bitmap to equivalently represent the DFA with counter. As shown in Fig. 3(c), any element e in the bitmap corresponds to a state u and a character c; the value of e, either 0 or 1, records the increment of the counter when activating u via c. The bitmap for a stem s is a two-dimensional table T, which is constructed as follows: first, initialize all elements in T to 1; then, for each position i ($1 \le i \le l$) of s, change T according to the rules given in Table 3. Through querying the table T, we can scan an l-mer z with $O(l)$ time, and the number of mismatches N_{mis} is calculated by (7), where $T[i-1][z[i]]$ represents the element in T corresponding to the state $i - 1$ and the character $z[i]$. The storage space of a DFA with counter is $O(l|\sum|)$.

$$N_{mis} = \sum_{1 \le i \le l} T[i-1][z[i]] \qquad (7)$$

5.3 Accelerating Verification via Pruning

This section introduces a pruning technique to reduce the number of stems to be verified. As described in Section 4, each rough stem s can be decomposed into 2^w stems, where w is the number of such position i of s that corresponds to $x[i]|x'[i]$. For simplicity of explanation, let A denote the array of these w positions. Thus, the search space of s is a complete binary tree called search tree: the root is s; each leaf is one of the 2^w stems; in the ith ($0 < i < w$) level of the tree, there are 2^i internal nodes, obtained

by expanding s in the positions $A[1]$, …, $A[i]$. For example, assume that the root s = [^A]A[^GC](G|C)(G|C). Then the nodes in the first and the second level are {[^A]A[^GC]G(G|C), [^A]A[^GC]C(G|C)} and {[^A]A[^GC]GG, [^A]A[^GC]GC, [^A]A[^GC]CG, [^A]A[^GC]CC}, respectively.

Observation 2. *In the search tree of a rough stem, let q be an internal node, let y be a child node of q, and let z be a random l-mer. We have $d_H(y, z) = d_H(q, z)$ or $d_H(y, z) = d_H(q, z) + 1$, namely $d_H(y, z) \geq d_H(q, z)$. Here, $d_H()$ denotes the number of positions where a stem mismatches an l-mer, and it is calculated by* (7).

In terms of Observation 2，if an internal node q fails to span all input sequences, then the child nodes of q will also fail to do so. Therefore, when we search valid stems in the search tree, the subtrees of invalid nodes can be pruned. This pruning technique facilitates avoiding the verification of some invalid stems, especially for large alphabets.

Theorem 1. *In the search tree of a rough stem, let q be an internal node at level i, let y be a leaf in the subtree of q, and let z be a random l-mer. We have*

$$\Pr\left(d_H(q,z) = d_H(y,z)\right) = \left(\frac{|\Sigma|-1}{|\Sigma|}\right)^{w-i}. \quad (8)$$

Proof. The nodes q and y differ in the positions $A[i+1]$, …, $A[w]$; for any $j \in \{A[i+1], \ldots, A[w]\}$, $q[j]$ is represented as $x[j]|x'[j]$, while $y[j]$ is an exact character, either $x[j]$ or $x'[j]$. Let us first consider $d_H(q, z) \neq d_H(y, z)$. It holds if and only if there exists at least one position $j \in \{A[i+1], \ldots, A[w]\}$ corresponding to the case that $z[j] \neq y[j]$ but $z[j]$ can be matched by $q[j]$. For any position $j \in \{A[i+1], \ldots, A[w]\}$, the probability that this case occurs is equal to $1/|\Sigma|$. In other words, the probability that this case does not occur for the position j is equal to $1 - 1/|\Sigma| = (|\Sigma| - 1)/|\Sigma|$. When this case does not occur for all the positions $A[i+1], \ldots, A[w]$, $d_H(q, z) = d_H(y, z)$ holds. Therefore, the probability of $d_H(q, z) = d_H(y, z)$ is $((|\Sigma| - 1)/|\Sigma|)^{w-i}$. □

By Theorem 1, when the l-mer z is not an occurrence of the leaf y (a stem), the probability that z is not an occurrence of the internal node q (a rough stem) is at least $((|\Sigma| - 1)/|\Sigma|)^{w-i}$. This probability increases with the increase of $|\Sigma|$. Assume that $w - i = 3$. When $|\Sigma| = 4$, the probability is 0.42; whereas, when $|\Sigma| = 40$, the probability is 0.93. Therefore, the pruning technique is more effective for searching stems over large alphabets.

5.4 StemFinder

This section describes the whole algorithm of StemFinder by using the pseudocode shown in Algorithm 1, Algorithm 2 and Algorithm 3. Algorithm 1 corresponds to the main framework. Algorithm 2 and Algorithm 3 called by Algorithm1, correspond to the construction of the set I and the verification of the stems for a given rough stem, respectively. In Algorithm 3, the pruning technique is applied to searching the search tree in a depth-first manner.

Algorithm 1 StemFinder
Input: l, d, $\{S_1, S_2, \ldots, S_t\}$
Output: the set of stems M that covers all (l, d) motifs
1: $M \leftarrow \Phi$
2: sort input sequences into ascending order by length
3: **for** $i \leftarrow 0$ to $2d$ **do**
4: calculate $R(i)$
5: calculate $N_{rs}(i)$
6: $I \leftarrow$ GenerateSetI
7: **for** each $(x, x') \in I$ **do**
8: **for** each $<\alpha, \beta> \in R(d_H(x, x'))$ **do**
9: $s \leftarrow x$
10: replace α characters of s in P_m with [^$x[i]$]], forming the set of strings M_1
11: **for** each string s in M_1 **do**
12: replace β characters of s in P_n with [^$x[i]x'[i]$]], forming the set of strings M_2
13: **for** each string s in M_2 **do**
14: replace each character of s in P_n that is $x[i]$ with $x[i]|x'[i]$, forming the set of rough stems M_3
15: **for** each rough stem s in M_3 **do**
16: ordered set $A \leftarrow \{i: 1 \leq i \leq l \;\&\&\; s[i] \text{ is } x[i]|x'[i]\}$
17: VerifyRoughStem(s, A, 0)
18: **return** M

Algorithm 2 GenerateSetI
1: $I \leftarrow \Phi$
2: **for** each l-mer x in S_1 **do**
3: $N_{\min} \leftarrow \infty$
4: **for** $j \leftarrow 2$ to t **do**
5: $C(x, S_j) \leftarrow \Phi$
6: $N_j \leftarrow 0$
7: **for** each l-mer x' in S_j **do**
8: **if** $d_H(x, x') \leq 2d$ **then**
9: $C(x, S_j) \leftarrow C(x, S_j) \cup \{x'\}$
10: $N_j = N_j + N_{rs}(d_H(x, x'))$
11: **if** $N_j < N_{\min}$ **then**
12: $N_{\min} \leftarrow N_j$
13: $S_{\min} \leftarrow S_j$
14: **for** each l-mer $x' \in C(x, S_{\min})$ **do**
15: $I \leftarrow I \cup \{(x, x')\}$
16: **return** I

Algorithm 3 VerifyRoughStem(s, A, i)
1: **if** s cannot span $\{S_1, S_2, \ldots, S_t\}$ **then**
2: **return** // perform pruning
3: **else**
4: **if** $i < |A|$ **then**
5: $j \leftarrow A[i+1]$ // the index of A begins with 1
6: $s_1 \leftarrow s$ with $s[j]$ replaced by $x[j]$
7: $s_2 \leftarrow s$ with $s[j]$ replaced by $x'[j]$
8: VerifyRoughStem(s_1, A, i+1)
9: VerifyRoughStem(s_2, A, i+1)
10: **else**
11: $M \leftarrow M \cup \{s\}$

For Algorithm 1, line 2 sorts the input sequences into ascending order by length in $O(t \log(t))$ time, which facilitates forming a smaller set I for handling variable length sequences. Lines 3-5 calculate and cache the values of $R(i)$ and $N_{rs}(i)$ for all possible Hamming distances i. $R(i)$ is obtained by listing all $|P_{mn}|$ and $|P_{nn}|$ satisfying (3), and both $|P_{mn}|$ and $|P_{nn}|$ are less than or equal to d, so the time complexity of calculating all $R(i)$ for $0 \leq i \leq 2d$ is $O(d^3)$. $N_{rs}(i)$ is obtained along with the calculation of $R(i)$. Line 6 constructs the set of pairs of l-mers I by calling Algorithm 2. In terms of the description of Algorithm 2, the time complexity of constructing the set I is $O(tn^2l)$, where l corresponds to the time of calculating the Hamming distance

between two l-mers. Lines 7-17 generate and verify stems; the time complexity is the number of stems $|stems|$ multiplied by the time of verifying each stem $O(tnl)$, where l corresponds to the time of querying the table T when scanning an l-mer.

According to the analysis above, the StemFinder algorithm runs in $O(tl\lg t + d^3 + tn^2l + |stems|tnl)$ time. The expected number of $|stems|$ can be estimated as follows, which decreases with the increase of the size of the alphabet.

Theorem 2. *The expected number of stems generated by StemFinder is*

$$E(|stems|) = (n-l+1)^2 \sum_{i=0}^{2d} p_i' N_s(i) . \quad (9)$$

Proof. The result above is drawn from the assumption that each input sequence is composed of independent, uniformly distributed random characters coming from an alphabet $\sum$.

At first, let us briefly review the method for constructing the set I. For each of $n - l + 1$ l-mers x in S_1, selects a reference sequence S_r. For each of $n - l + 1$ l-mers x' in S_r, if $d_H(x, x') \le 2d$, then add the pair of l-mers x and x' to the set I.

Next, we consider a fixed Hamming distance i ($0 \le i \le 2d$). Since p_i', calculated by (1), denotes the probability that the Hamming distance between two l-mers is i, the expected number of pairs of l-mers in the set I with Hamming distance i is $(n - l + 1)^2 p_i'$. Moreover, the number of stems generated from a pair of l-mers with Hamming distance i is $N_s(i)$, which is calculated by (5). Thus, the number of stems generated from all pairs of l-mers in the set I with Hamming distance i is $(n - l + 1)^2 p_i' N_s(i)$.

Finally, we sum the number of generated stems for all possible Hamming distance i ($0 \le i \le 2d$) and obtain the value of $E(|stems|)$ shown in (9). Since p_i' decreases with the increase of the size of the alphabet, so does $E(|stems|)$. □

For the storage space, in addition to $O(d^3)$ words which are used to cache $R(i)$ and $N_{rs}(i)$, we need to store $O(tn)$ l-mers in input sequences. Moreover, when we verify each stem, the space required to store the bitmap is $O(l|\sum|)$. Therefore, the space complexity of StemFinder is $O(d^3 + tn + l|\sum|)$. Although the space complexity depends on the size of alphabets, the value of $l|\sum|$ increases linearly with the growth of $|\sum|$ and it is small even for very large alphabets.

6 Results and Discussion

6.1 Comparison of Stems Generated from Different Algorithms

In this section, we compare the stems generated by StemFinder with that generated by previous MSS algorithms (Stemming [20] and MSS1/MSS2 [21]). Assume that the l-mers x and x' are two instances of a motif y with $(l, d) = (7, 3)$. For different MSS algorithms, we give in Table 4 the stem generated from x and x' that can cover y; each row of the table corresponds to a different motif y for fixed l-mers x = AAAGGGG and x' = AAACCCC. Here we do not consider the stems that cannot cover y, since most of them are filtered out in stem verification. We carry out comparisons by answering the following two questions.

TABLE 4
Stems Generated by Different Algorithms

Motif y	Generated stem that can cover y		
	StemFinder	*Stemming*	*MSS1/MSS2*
AAAGGCC	AAAGGCC	AAAGGCC	AAAGG**
AAATGGC	AAA[^GC]GGC	AAA*GGC	AAA*GG*
AAATTGC	AAA[^GC][^GC]GC	AAA**GC	AAA**G*
ATAGGCC	A[^A]AGGCC	None	A*AGG**

*For two fixed l-mers x = AAAGGGG and x' = AAACCCC, this table gives the generated stems that can cover the (7, 3) motif y under different MSS algorithms. The wildcard * in the stems generated by Stemming matches any character over $\sum$; the wildcard * in the ith position of the stems generated by MSS1/MSS2 matches any character over $\sum$ except for $x[i]$.*

Does there always exist a generated stem that can cover the motif y? The answer is yes for both StemFinder and MSS1/MSS2, because both of them consider all possible stems s under the condition that $d_H(s, x) \le d$ and $d_H(s, x') \le d$. However, it is not true for Stemming because it does not place wildcards in the matching region of x and x' when $d_H(x, x') > d$. For example, in the last row of Table 4, there is a position i (the second position) that $y[i] \ne x[i] = x'[i]$; in this case, the stems generated by Stemming miss the one that covers y.

Does there exist a redundant wildcard in the generated stem? A wildcard in position i is redundant if it can be replaced by $x[i]$ or $x'[i]$, and a stem with redundant wildcards covers more l-mers that are not motif instances. From Table 4, there are no redundant wildcards in the stems generated by StemFinder and Stemming. However, MSS1/MSS2 places at least one redundant wildcard in each stem shown in Table 4. The reason is that MSS1/MSS2 generates stems just by placing wildcards in x, without setting some positions of x as in x' (this operation is supported by both StemFinder and Stemming); thus, in the non-matching region of x and x', some characters that could be represented as $x'[i]$ are replaced by wildcards.

In summary, StemFinder not only generates all possible stems, but also places non-redundant wildcards in them. Moreover, StemFinder represents stems more precisely by replacing the typical wildcards * with negative character sets [^].

6.2 Results on Simulated Data

The simulated data sets over an alphabet $\sum$ are generated following [2], which are also used in [20] and [21]. First, randomly generate a motif m of length l and t = 20 sequences of length n = 600; second, for each sequence S_i, randomly generate a motif instance m' differing from m in at most d positions, and then implant m' to a random position in S_i.

We implement StemFinder using C++ and perform it

on a computer with 2.67 GHz processor and 4 Gbyte memory. All results are the average obtained by running algorithms on five random data sets. For the time units, s, m and h denote seconds, minutes and hours, respectively; -o represents the running time that exceeds ten hours.

First, we compare StemFinder with the algorithm designed for searching DNA motifs. PairMotif [14], our previous work, is selected as the comparison object, which can be downloaded from http://files.figshare.com/294289/Program_S1.rar. As shown in Table 5, StemFinder does not show its performance advantages on DNA data, because it is not specifically optimized for $|\sum| = 4$. However, for large alphabets ($|\sum|$ ranging from 10 to 100), StemFinder runs much faster than PairMotif; the reason is that StemFinder verifies all possible stems rather than all possible motifs, and the number of possible stems is much smaller than the number of possible motifs, especially for large alphabets.

Second, we compare StemFinder with the previous MSS algorithms, namely Stemming [20] and MSS2 [21]. For the work [21], we use MSS2 rather than MSS1 as the comparison object, since MSS2 is an improvement version of MSS1. Neither Stemming nor MSS2 provides source code or executable programs, so we also implement them using C++ and perform them on the same experimental environment. Consistent with [20] and [21], the data sets used to test algorithms are (7, 1), (9, 2) etc. over $|\sum| = 20$.

We show in Table 6 the running time of different MSS algorithms. We can see from the table that StemFinder greatly outperforms MSS2 and Stemming, and is able to solve all these problem instances with $l < 30$ within ten minutes. Both the running time of MSS2 and Stemming grows dramatically with the increase of l and d; when $(l, d) = (15, 5)$ and $(l, d) = (23, 9)$, Stemming and MSS2 both require more than ten hours.

TABLE 5
Running Time of StemFinder Compared with PairMotif

$\|\sum\| = 4$			$(l, d) = (15, 5)$		
(l, d)	*StemFinder*	*PairMotif*	$\|\sum\|$	*StemFinder*	*PairMotif*
(7, 1)	0.4s	0.5s	10	0.5s	5.5h
(9, 2)	3.5s	0.9s	20	0.2s	-o
(11, 3)	28.9s	2.7s	40	0.2s	-o
(13, 4)	4.9m	43.2s	60	0.2s	-o
(15, 5)	1.0h	4.5m	80	0.2s	-o
(17, 6)	-o	53.2m	100	0.2s	-o

TABLE 6
Running Time of StemFinder Compared with MSS2 and Stemming

(l, d)	StemFinder	MSS2	Stemming
(7, 1)	0.2s	0.2s	1.8m
(9, 2)	0.2s	0.2s	6.1m
(11, 3)	0.2s	0.3s	30.0m
(13, 4)	0.2s	1.9s	4.4h
(15, 5)	0.2s	11.7s	-o
(17, 6)	0.2s	1.1m	-o
(19, 7)	1.0s	6.6m	-o
(21, 8)	5.6s	2.1h	-o
(23, 9)	15.4s	-o	-o
(25, 10)	3.5m	-o	-o
(27, 11)	8.3m	-o	-o
(29, 12)	8.2m	-o	-o

We show in Fig. 4 the total number of stems that are reported by different MSS algorithms. We find that StemFinder and MSS2 report a much smaller number of stems compared to the method Stemming. Our explanation is that the two methods use a much smaller number of pairs of l-mers to generate the stems. Particularly, for StemFinder and MSS2, the former reports a smaller number of stems, owing to two factors: (i) the stems generated by StemFinder contain fewer wildcards than MSS2, and have more chance to be filtered out in stem verification; (ii) StemFinder select the reference sequence S_r corresponding to the minimum number of rough stems, which contributes to reducing the number of possible candidate stems.

We show in Fig. 5 the number of l-mers that are covered by the reported stems for different MSS algorithms. We use the log-scale on the y-axis of the figure in order to better compare different algorithms. Given a stem with i wildcards, the number of covered l-mers is X^i, where X is $|\sum|$, $|\sum| - 1$, and $|\sum| - 1$ or $|\sum| - 2$ for the algorithms Stemming, MSS2, and StemFinder, respectively. In particular we find that the number of covered l-mers for StemFinder is about 1% of that of MSS2 and 0.01% of that of Stemming. We believe two factors play an important role in explaining the observed huge difference. (i) StemFinder reports a small number of stems and (ii) there is no redundancy of wildcards in the stems that are reported by StemFinder.

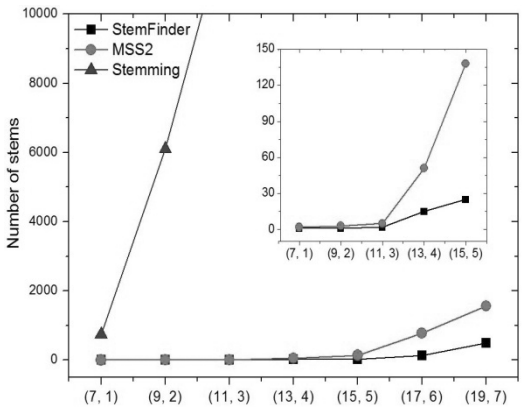

Fig. 4. The number of stems reported by different MSS algorithms.

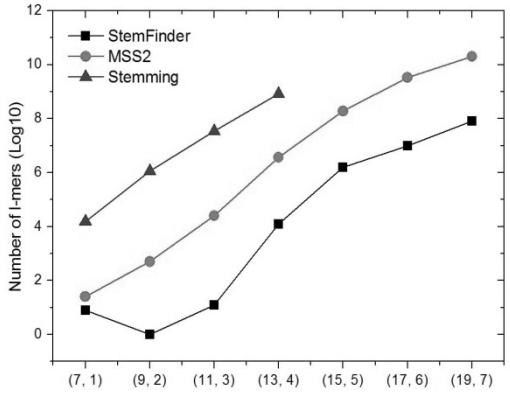

Fig. 5. The number of l-mers covered by reported stems for different MSS algorithms.

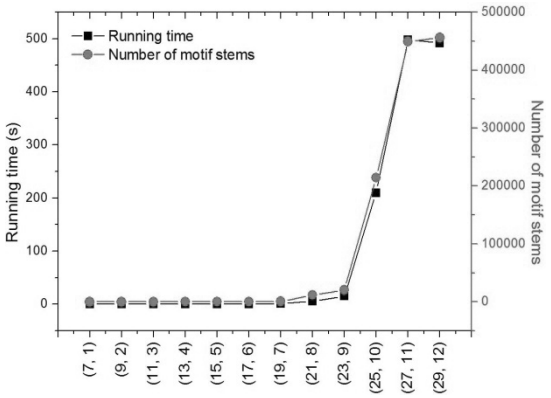

Fig. 6. The trend of running time and that of the number of reported stems for StemFinder.

TABLE 7
Results on Challenging Problem Instances

(l, d)	StemFinder	qPMS7	MSS2	Stemming
(7, 3)	2.6m	0.2m	26.7m	-o
(9, 4)	4.4m	1.3m	2.6h	-o
(11, 5)	6.9m	4.6m	-o	-o
(13, 6)	10.1m	2.4m	-o	-o
(15, 7)	13.6m	3.5m	-o	-o
(17, 8)	26.6m	-o	-o	-o
(19, 9)	53.9m	-o	-o	-o
(21, 10)	4.4h	-o	-o	-o

Moreover, we show in Fig. 6 the running time and the number of reported stems for StemFinder with the problem instances with l < 30. It is clear from the figure that the running time is highly correlated with the number of reported stems. We see that StemFinder has improved running time performance partially due to the fact that it reports a smaller number of stems comparing to the other two methods.

From the above, we see that StemFinder performs better than the previous MSS algorithms in all the following three aspects: the running time, the number of reported stems and the number of l-mers covered by the reported stems. In the following discussion we focus on running time since for exact algorithms, running time efficient is the most important goal that we aim to achieve in designing new algorithms.

Third, we evaluate algorithms on the challenging problem instances [21] over $|\sum|$ = 20, namely (7, 3), (9, 4) etc. Here, challenging instances are used to test upper bounds of the computation ability of an exact algorithm. The results on these instances are shown in Table 7. We find that StemFinder is able to solve very challenging instances such as (21, 10) within ten hours. MSS2 can only solve two instances (7, 3) and (9, 4). Stemming fails to solve any challenging instances. In addition we test an efficient PMS algorithm qPMS7, downloaded from http://pms.engr.uconn.edu/downloads/qPMS7.zip, because it can be used to solve problems over $|\sum|$ = 20. The algorithm qPMS7 has better running time efficiency when $l \le 15$. This may be due to the fact that qPMS7 considers the common d-neighbors shared by three l-mers rather than two. However, when l > 15, qPMS7 takes a very long running time, since a huge number of candidate motifs need to be verified.

TABLE 8
Results over Large Alphabets

$\|\sum\|$	(7, 3)		(9, 4)		(11, 5)	
	SF[a]	*MSS2*	*SF*[a]	*MSS2*	*SF*[a]	*MSS2*
40	30.4s	12.9m	53.5s	1.2h	76.9s	6.4h
60	12.9s	8.1m	21.2s	46.4m	31.4s	4.6h
80	7.2s	6.4m	11.5s	33.5m	17.1s	3.1h
100	4.7s	4.3m	7.3s	25.7m	11.1s	2.6h

[a]*SF is short-hand for StemFinder.*

At last, we further evaluate algorithms over large alphabets. We show the results in Table 8 with $|\sum|$ = 40, 60, 80 and 100. From the table we see that with a fixed (l, d) instance, both StemFinder and MSS2 have shorter running time when the alphabet is large. This is not surprising since large alphabet leads to a reduced p_{2d} and hence we have smaller number of pairs of l-mers to generate stems. Comparing StemFinder and MSS2, we find that StemFinder is often an order of magnitude faster than MSS2.

6.3 Results on Real-world Data Sets with Protein Sequences

We collect our data sets from the Eukaryotic Linear Motif (ELM) database (*http://elm.eu.org*) [23]. ELM database contains multiple short protein motifs given in the form of regular expressions. Each motif corresponds to a unique ELM identifier (ELM ID). We obtain ten data sets with the latest 100 ELM motif instances and name them with the ELM ID. We only select those data sets with at least three instances of a motif.

TABLE 9
Results on ELM Data Sets

Data set (# instances)	(l, d)	SF[a]	MSS	Stemming	ELM Motif	Detected Motif
LIG_EVH1_1 (18)	(5, 1)	0.1s	0.1s	0.1s	([FYWL]P.PP)|([FYWL]PP[ALIVTFY]P)	FPPPP
LIG_WW_1 (3)	(4, 1)	0.1s	0.4s	2.0s	PP.Y	PPVY
LIG_14-3-3_1 (3)	(6, 2)	0.1s	0.3s	1.4s	R.[^P]([ST])[^P]P	RSSSSP
LIG_MYND_2 (3)	(5, 1)	0.3s	1.4s	7.1s	PP.LI	PPPLI
LIG_USP7_1 (3)	(5, 2)	0.5s	0.7s	38.2s	[PA][^P][^FYWIL]S[^P]	Null
LIG_APCC_TPR_1 (22)	(3, 1)	10.3s	3.9s	1.1h	.[ILM]R$	Null
LIG_MYND_1 (6)	(5, 2)	25.6s	25.9s	1.9h	P.L.P	P[^CG]LAP
LIG_PAM2_1 (4)	(13, 6)	1.0m	-o	-o	..[LFP][NS][PIVTAFL].A..(([FY].[PYLF])|(W..)).	SAFNPNAKEFVPI
MOD_NEK2_1 (3)	(6, 3)	10.3m	-o	-o	[FLM][^P][^P]([ST])[^DEP][^DE]	FAESFS
LIG_EABR_CEP55-1 (6)	(11, 5)	24.2m	-o	-o	.A.GPP.{2,3}Y.	[^MT]AVGPPQLSYM

[a]*SF is short-hand for StemFinder.*

We first demonstrate the validity of StemFinder for searching motifs on these real protein data sets. We show in Table 9 the detected stems, which are those spanning all motif instances under the used (l, d). From the table we see a good matching between our results and the ELM motifs in most of the data sets, except for LIG_USP7_1 and LIG_APCC_TPR_1, where we do not find an appropriate (l, d) to carry out prediction. There are subtle differences between the detected motifs and the ELM motifs, since the ELM motifs are curated by hand and our results are completely obtained through computation without additional biological knowledge.

In addition, we list the running time of different algorithms at the same table. We see that StemFinder is very efficient, and completes the computation for any data sets within 30 minutes. As a comparison MSS2 and Stemming take more than 10 hours to process challenging cases LIG_PAM2_1, MOD_NEK2_1 and LIG_EABR_CEP55-1.

7 Conclusion

This paper focuses on the exact algorithms for searching motif stems over large alphabets. To represent stems more precisely and concisely, we write stems as regular expressions by replacing typical wildcards with the negative character sets, and place as few negative character sets as possible. Then, a new exact algorithm called StemFinder is proposed. Experimental results on simulated data show that StemFinder outperforms the previous algorithms on both the time performance and the ability to report fewer stems. Moreover, the validity of StemFinder is demonstrated on real protein data sets.

A limitation of our current study is that StemFinder does not support searching stems on data sets where some input sequences may contain no motif instances. We plan to concentrate our future work on solving this problem.

Acknowledgment

This research was supported in part by the National Natural Science Foundation of China (61173025 and 61373044), the Research Fund for the Doctoral Program of Higher Education of China (20100203110010), the Fundamental Research Funds for the Central Universities(K5051303032, K5051303002 and K50513100011), and the Natural Science Foundation of Shaanxi (2013JQ8037). A preliminary version [24] of this work appeared in the proceedings of IEEE International Conference on Bioinformatics and Biomedicine (BIBM), 18-21 December 2013, Shanghai, China. Hongwei Huo is the corresponding author.

References

[1] P. D'haeseleer, "What Are DNA Sequence Motifs?" *Nature Biotechnology*, vol. 24, no. 4, pp. 423-425, 2006.

[2] P.A. Pevzner and S. Sze, "Combinatorial Approaches to Finding Subtle Signals in DNA Sequences," *Proc. Eighth Int'l Conf. Intelligent Systems for Molecular Biology*, pp. 269-278, 2000.

[3] P.A. Evans, A.D. Smith, and H.T. Wareham, "On the Complexity of Finding Common Approximate Substrings," *Theoretical Computer Science*, vol. 306, pp. 407-430, 2003.

[4] J.D. Thompson, D.G. Higgins, and T.J. Gibson, "CLUSTAL W: Improving the Sensitivity of Progressive Multiple Sequence Alignment through Sequence Weighting, Positionspecific Gap Penalties and Weight Matrix Choice," *Nucleic Acids Research*, vol. 22, pp. 4673–4680, 1994.

[5] T.D. Schneider, "Consensus sequence Zen," *Applied bioinformatics*, vol. 1, pp. 111-119, 2002.

[6] C. Lawrence, S. Altschul, M. Boguski, J. Liu, A. Neuwald, and J. Wootton, "Detecting Subtle Sequence Signals: a Gibb's Sampling Strategy for Multiple Alignment," *Science*, vol. 262, pp. 208-214, 1993.

[7] T. Bailey and C. Elkan, "Fitting a Mixture Model by Expectation Maximization to Discover Motifs in Biopolymers," *Proc. Second Int'l Conf. Intelligent Systems for Molecular Biology*, pp. 28-36, 1994.

[8] Y. Zhang, H. Huo, and Q. Yu, "A Heuristic Cluster-based EM Algorithm for the Planted (l, d) Problem," *J. Bioinformatics and Computational Biology*, vol. 11, no. 4, art. no. 1350009, 2013.

[9] F.Y.L. Chin and H.C.M. Leung, "Voting Algorithms for Discovering Long Motifs," *Proc. Third Asia Pacific Bioinformatics Conference*, pp. 261-271, 2005.

[10] J. Davila, S. Balla, and S. Rajasekaran, "Fast and Practical Algorithms for Planted (l, d) Motif Search," *IEEE/ACM Trans. Computational Biology and Bioinformatics*, vol. 4, no. 4, pp. 544–552, 2007.

[11] E.S. Ho, C.D. Jakubowski, and S.I. Gunderson, "iTriplet, a Rule-based Nucleic Acid Sequence Motif Finder," *Algorithms for Molecular Biology*, vol. 4, art. no. 14, 2009.

[12] 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 Trans. Computational Biology and Bioinformatics*, vol. 8, no.5, pp. 1400-1410, 2011.

[13] H. Dinh, S. Rajasekaran, and V.K. Kundeti, "PMS5: an Efficient Exact Algorithm for the (l, d)-Motif Finding Problem," *BMC Bioinformatics*, vol. 12, art. no. 410, 2011.

[14] 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, no. 10, art. no. e48442, 2012.

[15] 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, 2012.

[16] Y. Xu, J. Yang, Y. Zhao, and Y. Shang, "An Improved Voting Algorithm for Planted (l, d) Motif Search," *Information Sciences*, vol. 237, pp. 305-312, 2013.

[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, 2001.

[18] E. Eskin and P.A. Pevzner, "Finding Composite Regulatory Patterns in DNA Sequences," *Bioinformatics*, vol. 18, no. 1, pp. 354-363, 2002.

[19] N. Pisanti, A.M. Carvalho, L. Marsan, and M. Sagot, "RISOTTO: Fast Extraction of Motifs with Mismatches," *Proc. Seventh Latin American Symposium: Theoretical Informatics*, pp. 757-768, 2006.

[20] P.P. Kuksa and V. Pavlovic, "Efficient Motif Finding Algorithms for Large-alphabet Inputs," *BMC Bioinformatics*, vol. 11(Suppl 8), art. no. S1, 2010.

[21] T. Mi and S. Rajasekaran, "Efficient Algorithms for Biological Stems Search," *BMC Bioinformatics*, vol. 14, art. no. 161, 2013.

[22] J.E. Hopcroft, R. Motwani, and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Second Edition. Addison Wesley, pp. 83-122, 2001.

[23] H. Dinkel, S. Michael, R.J. Weatheritt, N.E. Davey, K.V. Roey, B. Altenberg, G. Toedt, B. Uyar, M. Seiler, A. Budd, L. Jo¨dicke, M.A. Dammert, C. Schroeter, M. Hammer, T. Schmidt, P. Jehl, C. McGuigan, M. Dymecka, C. Chica, K. Luck, A. Via, A. Chatr-aryamontri, N. Haslam, G. Grebnev, R.J. Edwards, M.O. Steinmetz, H. Meiselbach, F. Diella, and T.J. Gibson, "ELM - The Database of Eukaryotic Linear Motifs," *Nucleic Acids Research*, vol. 40(Database issue), pp. 242-251, 2012.

[24] Q. Yu, H. Huo, J.S. Vitter, J. Huan, and Y. Nekrich, "StemFinder: An Efficient Algorithm for Searching Motif Stems over Large Alphabets," *Proc. IEEE Int'l Conf. Bioinformatics and Biomedicine (BIBM)*, submitted for publication, 2013.