

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 MSS algorithms, StemFinder runs much faster and 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—Motif stem search, pattern driven, regular expressions

1 INTRODUCTION

MOTIF search is to find short similar sequence segments in a given set of sequences over an alphabet Σ , which plays an important role in discovering significant segments in biological sequences, such as transcription factor binding sites [1] and linear protein motifs [2], [3]. The planted (l, d) motif search (PMS) [4] is a widely accepted formulation of the problem. A (l, d) motif is an l -mer (i.e., an l -length string over Σ) 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 [5].

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) [6] and consensus sequences [7]. Based on these two motif models, numerous motif search algorithms have been proposed.

The algorithms that model motifs using PWM usually employ statistical techniques [8], [9], [10]. 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 Σ 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 [11], [12], [13], [14], [15], [16], [17], [18]. Some other pattern-driven algorithms represent the input sequences as a suffix tree to accelerate the verification of candidate motifs [19], [20], [21]. The initial search space of pattern-driven algorithms is $O(|\Sigma|^l)$, which grows dramatically with the increase of $|\Sigma|$. Therefore, most existing exact algorithms are designed just for searching motifs in DNA sequences where $|\Sigma| = 4$, 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 $|\Sigma| = 20$.

To improve the efficiency of the exact algorithms over large alphabets, Kuksa and Pavlovic [22] 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 $\Sigma = \{A, G, C, T\}$ 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 Σ is replaced by placing wildcards in these positions.

MSS algorithms are the main subject of this paper. Stemming [22] 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 by using a selection algorithm [23]; 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 having Hamming distance no more than d from input

• Q. Yu and H. Huo are with the School of Computer Science and Technology, Xidian University, Xi'an 710071, China.

• J.S. Vitter, J. Huan, and Y. Nekrich are with the Information and Telecommunication of Technology Center, The University of Kansas, Lawrence, KS 66047. E-mail: jsv@ittc.ku.edu, jhuan@ittc.ku.edu, yakov@ittc.ku.edu.

Manuscript received 12 Nov. 2013; revised 19 Aug. 2014; accepted 16 Sept. 2014. Date of publication 6 Oct. 2014; date of current version 3 Apr. 2015.

For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.

Digital Object Identifier no. 10.1109/TCBB.2014.2361668

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' . $P_m(x, x') = \{i : 1 \leq i \leq l, x[i] = x'[i]\}$.
$P_n(x, x')$	The positions in the non-matching region of two l -mers x and x' . $P_n(x, x') = \{i : 1 \leq i \leq l, x[i] \neq x'[i]\}$.
$P_{mn}(x, x', y)$	The positions where x matches x' , and y matches neither x nor x' , for the given three l -mers x , x' and y . $P_{mn}(x, x', y) = \{i : 1 \leq i \leq l, x[i] = x'[i], y[i] \neq x[i] \text{ and } y[i] \neq x'[i]\}$.
$P_{mm}(x, x', y)$	The positions where x , x' and y are mismatched with each other, for the given three l -mers x , x' and y . $P_{mm}(x, x', y) = \{i : 1 \leq i \leq l, x[i] \neq x'[i], y[i] \neq x[i] \text{ and } y[i] \neq x'[i]\}$.
$d_H(x, x')$	The Hamming distance between two l -mers x and x' . $d_H(x, x') = P_n(x, x') = l - P_m(x, x') $.
$dis(x, D)$	The distance between an l -mer x and the set of input sequences D . $dis(x, D) = \max_{i=1, \dots, t} dis(x, S_i) = \max_{i=1, \dots, t} \min_{x' \in S_i} d_H(x, x') \leq 2d$.
$M_d(x, x')$	The common d -neighbors of two l -mers x and x' . $M_d(x, x') = \{y : y = x = x' , d_H(y, x) \leq d \text{ and } d_H(y, x') \leq d\}$.
$C(x, S_i)$	The l -mers in the sequence S_i that are $2d$ -neighbors of the l -mer x . $C(x, S_i) = \{y : y = x , y \in S_i \text{ and } d_H(y, x) \leq 2d\}$.
$C(x, x', S_i)$	The l -mers in the sequence S_i that are common $2d$ -neighbors of the l -mers x and x' . $C(x, x', S_i) = \{y : y = x = x' , y \in S_i, d_H(y, x) \leq 2d \text{ and } d_H(y, x') \leq 2d\}$.
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_{mm}(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.

sequences. In a recent work [24], new MSS algorithms MSS1 and MSS2 are proposed based on the assumption that each input sequence contains at least one motif instance. 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 Σ . 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 [24], the fastest stem search algorithm MSS2 is only able to solve the challenging instance (11, 5) over $|\Sigma| = 20$ around 1.5 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 $[\wedge]$. A negative character set $[\wedge]$ matches any character not enclosed; for example, $[\wedge]CT$ represents any single character over Σ 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 motif stem search 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 Σ 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.

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

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

Problem Definition: Motif Stem Search [24]. Given a set of n -length sequences $D = \{S_1, S_2, \dots, S_t\}$ over an alphabet Σ and nonnegative integers l and d , satisfying $0 \leq d < l < n$, 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, and the superset is expected to be as small as possible.

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 Motif Stem Search Methods

This section briefly reviews existing MSS algorithms: Stemming [22] and MSS1/MSS2 [24]. Both the two algorithms, as well as StemFinder, work through the following three steps.

Step 1 is to select a group of potential motif instances from input sequences to construct a set I , based on the observation that the Hamming distance between any two motif instances is less than or equal to $2d$. In Stemming, a potential motif instance is an l -mer x that can occur in each of input sequences with up to $2d$ mismatches, namely $\text{dis}(x, D) \leq 2d$. Stemming selects all such l -mer x from input sequences to construct the set I . MSS1/MSS2 constructs the set I as follows: for each l -mer x in S_1 , select a reference sequence S_r such that S_r is the sequence in $\{S_2, \dots, S_t\}$ containing the minimum number of l -mers x' satisfying $d_H(x, x') \leq 2d$; for each l -mer $x' \in C(x, S_r)$, add the pair of l -mers x and x' to the set I .

Step 2 is to generate candidate stems from each pair of l -mers x and x' in the set I by placing wildcards. For this step, Stemming and MSS1/MSS2 differ in the specific meanings of the used wildcards and the ways that wildcards are placed.

Stemming allows the wildcard to match any character over Σ and generate stems by changing x as follows: if $d_H(x, x') \leq d$, set $i(0 \leq i \leq d_H(x, x'))$ positions in $P_n(x, x')$ as in x' , place $\alpha(0 \leq \alpha \leq d_H(x, x') - i)$ wildcards in the remaining $d_H(x, x') - i$ positions in $P_n(x, x')$, and place $\beta(0 \leq \beta \leq d - \max(d_H(x, x') - i, \alpha + i))$ wildcards in $P_m(x, x')$; otherwise, set $i(d_H(x, x') - d \leq i \leq d)$ positions in $P_n(x, x')$ as in x' , and place $\alpha(0 \leq \alpha \leq d - i)$ 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 Σ except for the character in the corresponding position of x . MSS1/MSS2 generates stems s by placing α wildcards in $P_n(x, x')$ of x and β wildcards in $P_m(x, x')$ of x . The range of α is considered as follows. If $d_H(x, x') \leq d$, it is clear that α can vary from 0 to $d_H(x, x')$, namely $0 \leq \alpha \leq d_H(x, x')$; otherwise, at least $d_H(x, x') - d$ wildcards have to be placed in $P_n(x, x')$, namely $d_H(x, x') - d \leq \alpha \leq d$, to make $d_H(s, x) \leq d$ and $d_H(s, x') \leq d$ satisfied. Simultaneously, the range of β is

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: $x[i] = x'[i], y[i] \neq x[i]$ and $y[i] \neq x'[i]$	$ \Sigma - 1$	$[\wedge x[i]]$
Case 3: $x[i] \neq x'[i], y[i] = x[i]$ and $y[i] \neq x'[i]$	1	$x[i]$
Case 4: $x[i] \neq x'[i], y[i] \neq x[i]$ and $y[i] = x'[i]$	1	$x'[i]$
Case 5: $x[i] \neq x'[i], y[i] \neq x[i]$ and $y[i] \neq x'[i]$	$ \Sigma - 2$	$[\wedge x[i]x'[i]]$

^aThe number of characters matched by $y[i]$.

determined by satisfying the same condition that $d_H(s, x) \leq d$ and $d_H(s, x') \leq d$, namely $\alpha + \beta \leq d$ and $(d_H(x, x') - \alpha) + \beta \leq d$, so the maximum value of β is $\min\{d - \alpha, d - (d_H(x, x') - \alpha)\}$. 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.

Step 3 is to verify candidate stems and output the ones that can occur in each input sequence with up to d mismatches. MSS1/MSS2 takes this step as an optional post-process phase with the time of verifying each candidate stem $O(tnl)$. Stemming selects valid stems as follows: iteratively remove d out of l positions from each of the candidate stems and the l -mers in the input sequences; in each iteration, sort the resulting $(l - d)$ -mers lexicographically, and then scan the sorted list to output the stems that can match one of l -mers in each $S_i (i = 1, \dots, t)$ in the corresponding $l - d$ positions.

3 STEM REPRESENTATION

In the previous MSS algorithms, for a stem s of length l , $s[i] (0 \leq i \leq l)$ is either an exact character over Σ or a typical wildcard $*$. To represent stems more precisely, we introduce two new regular expression operators, namely the negative character set $[\wedge]$ and the choice operator $|\cdot|$. 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 i th ($1 \leq i \leq l$) 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 $|\Sigma| - 1$ and $|\Sigma| - 2$ characters, respectively. When $y[i]$ corresponds to multiple characters, we represent $y[i]$ using the negative character set $[\wedge]$. Specifically, for Case 2, $y[i]$ is represented as $[\wedge x[i]]$, which matches any character in Σ excluding $x[i]$; for Case 5, $y[i]$ is represented as

$[\wedge x[i]x'[i]]$, which matches any character in Σ excluding $x[i]$ and $x'[i]$.

According to the analysis above, assume that the regular expression s is a stem obtained from two l -mers x and x' . Then the i th position of s must fall into one of the three patterns: a specific character ($x[i]$ or $x'[i]$), a negative character set $[\wedge x[i]]$ or a negative character set $[\wedge x[i]x'[i]]$. In all the l positions of s , if there is a position i that corresponds to $[\wedge x[i]]$ or $[\wedge x[i]x'[i]]$, then s represents multiple candidate motifs; otherwise, s represents a single candidate motif. Let $(l, d) = (7, 3)$, $x = \text{AAAAGGG}$ and $x' = \text{AAAACCC}$; four possible stems are AAAAGGC , $\text{AAAAG}[\wedge\text{GC}]\text{C}$, $\text{AA}[\wedge\text{A}]\text{AGCC}$ and $\text{A}[\wedge\text{A}]\text{AAGC}[\wedge\text{GC}]$. 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 automatons [25] 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 $d_H(y, x) \leq d$ and $d_H(y, x') \leq d$. 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 $0 \leq |P_{mn}| \leq |P_m|$, namely $0 \leq |P_{mn}| \leq l - d_H(x, x')$; similarly, $0 \leq |P_{nn}| \leq d_H(x, x')$. Second, since all positions i in P_{mn} satisfy $y[i] \neq x[i]$ and $y[i] \neq x'[i]$, we have $|P_{mn}| \leq d$, which is necessary for $d_H(y, x) \leq d$ and $d_H(y, x') \leq d$ to be satisfied; similarly, $|P_{nn}| \leq d$. Third, $d_H(y, x) + d_H(y, x') \leq 2d$ where $d_H(y, x) + d_H(y, x')$ can be represented as $2|P_{mn}| + 2|P_{nn}| + (d_H(x, x') - |P_{mn}|) = 2|P_{mn}| + |P_{mn}| + d_H(x, x')$, so we have $2|P_{mn}| + |P_{nn}| + d_H(x, x') \leq 2d$, namely $2|P_{mn}| + |P_{nn}| \leq 2d - d_H(x, x')$. Taking these considerations into account, we obtain the following inequalities:

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

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

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

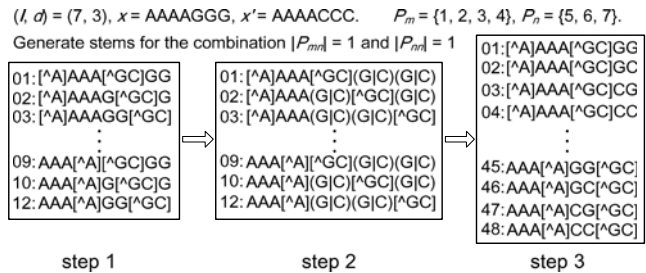


Fig. 1. An example for generating stems from two l -mers.

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 $[\wedge x[i]]$; at the same time, select $|P_{nn}|$ positions from P_n , and change the character of s in each selected position i to $[\wedge x[i]x'[i]]$. (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).

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

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

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 [22] and MSS1/MSS2 [24]), StemFinder performs more efficiently by introducing several techniques described in Sections 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

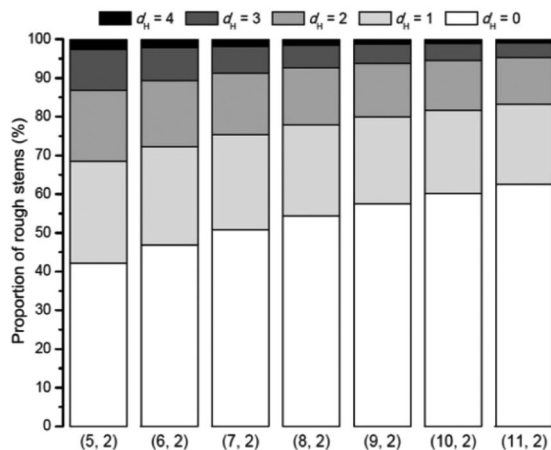


Fig. 2. Proportion of rough stems under different Hamming distances.

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 $\{S_2, \dots, S_t\}$; for each l -mer x' in S_r , if $d_H(x, x') \leq 2d$, 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 [24], in which the selected S_r is the sequence in $\{S_2, \dots, S_t\}$ that contains the minimum number of l -mers x' satisfying $d_H(x, x') \leq 2d$, 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, 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 ($0 \leq i \leq 2d$) is defined to be $N_{rs}(i)/N_{total}$, where $N_{total} = \sum N_{rs}(j)$ for $0 \leq j \leq 2d$. 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 10 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 $\{S_2, \dots, S_t\}$ that corresponds to the minimum number of rough stems.

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

The main reason of extracting pairs of l -mers coming from different input sequences to construct the set I is that we can use the strict constraints given in (3) to generate stems with as few wildcards as possible. Since the MSS problem assumes that each input sequence contains at least one motif instance, it seems that we can only use S_1 to generate candidate stems. However, in this case, more ($0 \leq \alpha \leq d$) wildcards need to be placed in each l -mer x in S_1 . This could result in the fact that a lot of generated stems may easily pass the stem verification and they represent a large superset of all (l, d) motifs.

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 $d_H(z, x) \leq 2d$ and $d_H(z, x') \leq 2d$ 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. 3a, 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. 3b. 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 .

In order to scan l -mers more efficiently, we use a bitmap to equivalently represent the DFA with counter. As shown in Fig. 3c, 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 . It is constructed as follows: first, initialize all elements in T to 1; then, for each position i ($1 \leq i \leq l$) 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

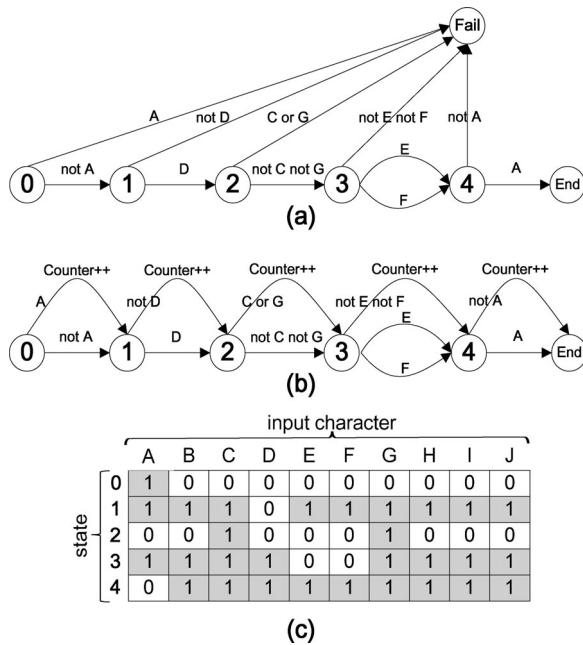


Fig. 3. DFA of the stem $[^A]D[^CG](E|F)A$ (a) Initial DFA (b) DFA with counter (c) Bitmap of DFA with counter.

T corresponding to the state $i - 1$ and the character $z[i]$. The storage space of a DFA with counter is $O(l|\Sigma|)$.

$$N_{mis} = \sum_{1 \leq i \leq l} T[i - 1][z[i]]. \quad (7)$$

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 i th ($0 < i < w$) level of the tree, there are 2^i internal nodes, obtained by expanding s in the positions $A[1], \dots, A[i]$. 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, $d_H(\cdot)$ 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

TABLE 3
Rules of Generating the Bitmap for a Stem s

$s[i]$	Generation Rule
$\neg x[i]$	Perform bitwise NOT on $T[i-1]$, and then set $T[i-1][x[i]]$ as 1
$\neg x[i]x'[i]$	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

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

Proof. The nodes q and y differ in the positions $A[i+1], \dots, A[w]$; for any $j \in \{A[i+1], \dots, A[w]\}$, $q[j]$ 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 $j \in \{A[i+1], \dots, A[w]\}$ corresponding to the case that $z[j] \neq y[j]$ but $z[j]$ can be matched by $q[j]$. For any position $j \in \{A[i+1], \dots, A[w]\}$, 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 $A[i+1], \dots, A[w]$, $d_H(q, z) = d_H(y, z)$ holds. Therefore, the probability of $d_H(q, z) = d_H(y, z)$ is $((|\Sigma| - 1)/|\Sigma|)^{w-i}$. \square

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 Algorithm 1, 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.

For Algorithm 1, line 2 sorts the input sequences into ascending order by length in $O(l \log t)$ 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 $|P_m|$ satisfying (3), and both $|P_{mn}|$ and $|P_m|$ 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.

Algorithm 1. StemFinder

Input: $l, d, \{S_1, S_2, \dots, S_t\}$

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:  $I \leftarrow \text{GenerateSetI}$ 
7: for each  $(x, x') \in I$  do
8:   for each  $\langle \alpha, \beta \rangle \in R(d_H(x, x'))$  do
9:      $s \leftarrow x$ 
10:    replace  $\alpha$  characters of  $s$  in  $P_n$  with  $[\wedge x[i]]$ , forming
        the set of strings  $M_1$ 
11:    for each string  $s$  in  $M_1$  do
12:      replace  $\beta$  characters of  $s$  in  $P_n$  with  $[\wedge x[i]x'[i]]$ ,
        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:           $\text{VerifyRoughStem}(s, A, 0)$ 
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:         $N_j \leftarrow N_j + N_{rs}(d_H(x, x'))$ 
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  $\{S_1, S_2, \dots, S_t\}$  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:      $\text{VerifyRoughStem}(s_1, A, i+1)$ 
9:      $\text{VerifyRoughStem}(s_2, A, i+1)$ 
10:  else
11:     $M \leftarrow M \cup \{s\}$ 

```

According to the analysis above, the StemFinder algorithm runs in $O(tlgt + d^3 + tn^2l + |stems|tnl)$ 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 Σ .

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 $d_H(x, x') \leq 2d$, then add the pair of l -mers x and x' to the set I .

Next, we consider a fixed Hamming distance i ($0 \leq i \leq 2d$). 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 ($0 \leq i \leq 2d$) 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 $E(|stems|)$. \square

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 $O(l|\Sigma|)$. Therefore, the space complexity of StemFinder is $O(d^3 + tn + l|\Sigma|)$. Although the space complexity depends on the size of alphabets, the value of $l|\Sigma|$ increases linearly with the growth of $|\Sigma|$ 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 [22] and MSS1/MSS2 [24]). 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 $x = \text{AAAGGGG}$ and $x' = \text{AAACCCC}$. 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.

Does there always exist a generated stem that can cover the motif y ? The answer is yes for both StemFinder and

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 Σ ; the wildcard $*$ in the i th position of the stems generated by MSS1/MSS2 matches any character over Σ , except for $x[i]$.

MSS1/MSS2, because both of them consider all possible stems s under the condition that $d_H(s, x) \leq d$ and $d_H(s, x') \leq d$. 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 $y[i] \neq x[i] = x'[i]$; in this case, the stems generated by Stemming miss the one that covers y .

Does there exist a redundant wildcard in the generated stem? The phenomenon that one stem representation contains all instances of another stem representation is redundant. Specifically, a wildcard in position i of the generated stem covering the motif y is redundant, if it can be replaced by $x[i]$ or $x'[i]$ to form a new stem that represents fewer l -mers and can still cover the motif y . 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. When $d_H(x, x')$ is very small, although more wildcards can be placed in the matching region, all of the three algorithms ensure no redundant wildcards are placed by satisfying the constraints used in stem generation methods.

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 $[\wedge]$.

6.2 Comparison of Methods for Constructing Set I

In this section, we compare the methods for constructing the set I by making an empirical statistical analysis on the size of I and the number of generated candidate stems. The purpose of constructing a good set I is to generate as few candidates as possible.

Both MSS2 and StemFinder make a simplifying assumption that each input sequence contains at least one motif instance, and thereby they use selected reference sequences to construct I ; however, the original Stemming algorithm [22] did not use this assumption. To fairly compare them, we construct I for Stemming in the same setting, i.e., using each l -mer x in S_1 paired with l -mers at distance $\leq 2d$ in the

reference sequence of x ; simultaneously, we ensure that each l -mer x in I is a potential motif instance (i.e., $dis(x, D) \leq 2d$). More specifically, we construct I for Stemming as follows: for each l -mer x in S_1 satisfying $dis(x, D) \leq 2d$, randomly select a reference sequence S_r from $\{S_2, \dots, S_t\}$; for each l -mer x' in S_r , add the pair of l -mers x and x' satisfying $d_H(x, x') \leq 2d$ and $dis(x', D) \leq 2d$ to I . Note that, since Stemming does not provide a strategy for selecting reference sequences, we implement Stemming by randomly selecting reference sequences.

There are five compared methods. Their difference mainly lies in how to select reference sequences. Our method, denoted by method 1, selects reference sequences by evaluating the number of generated candidates through (6). The method used in MSS1/MSS2, denoted by method 2, selects a reference sequence S_r in $\{S_2, \dots, S_t\}$ for each l -mer x in S_1 by ensuring S_r has the minimum number of l -mers x' satisfying $d_H(x, x') \leq 2d$. The method that randomly selects reference sequences is called method 3. Method 4 is the method in the worst case, which selects reference sequences leading to the maximum number of candidates. Method 5 corresponds to the method for Stemming described above, which differs from method 3 by ensuring that each l -mer x in I satisfies $dis(x, D) \leq 2d$.

To clearly show the advantage of method 1 (our method) compared to the others, we should obtain the number of generated candidates using the same stem generation method. Here we adopt the stem generation method proposed in this paper, and mainly consider the number of rough stems as described in Section 5.1. Note that, method 1 has advantages over the others not only for our stem generation method, but also for all existing stem generation methods [22], [24]; the reason is that method 1 is based on Observation 1, and Observation 1 applies to all existing stem generation methods.

Tables 5 and 6 show the size of the set I and the number of generated candidates under these five methods, respectively. The tested data are the simulated data described in Section 6.3. We set the alphabet size as 20, and choose the (l, d) problem instances corresponding to different values of p_{2d} . A large value of p_{2d} indicates that there are more pairs of l -mers x and x' in the input sequences with $d_H(x, x') \leq 2d$. From these two tables, we can see that

- 1) The size of the set I does not determine the number of candidates directly. Although both method 2 (the method used in MSS1/MSS2) and method 5 (the method used in Stemming) have a smaller size of the set I compared to method 1 (our method) for a fraction of these (l, d) problem instances, they do not correspond to a smaller number of candidates.
- 2) In these five methods, our method corresponds to the smallest number of candidates for all of these (l, d) problem instances, as shown in Table 6.
- 3) Method 5 is sensitive to the value of p_{2d} . When p_{2d} is small (< 0.001), the size of the set I under method 5 is smallest in these five methods, and the number of candidates under method 5 is also small (it is smaller than that under all of the other four methods except for method 1). When p_{2d} is large, method 5 does not show advantages,

TABLE 5
Size of the Set I under Different Methods for Constructing the Set I

(l, d)	p_{2d}	Size of the set I				
		Method 1	Method 2	Method 3	Method 4	Method 5
(21, 10)	0.6594	220,840	210,244	222,617	223,388	222,531
(20, 9)	0.2642	85,853	77,550	89,094	89,599	89,196
(19, 8)	0.0665	20,067	16,260	22,589	23,551	22,446
(18, 7)	0.0109	1,624	1,355	3,704	4,728	3,510
(16, 6)	0.0070	645	585	2,508	3,487	907
(15, 5)	0.0006	13	11	206	921	7
(13, 4)	0.0003	11	11	103	675	3
(11, 3)	0.0001	3	3	47	431	1
(10, 2)	<0.0001	2	2	4	27	1

Method 1 is the method proposed in this paper. Method 2 is the method used in MSS1/MSS2. Method 3 constructs the set I by randomly selecting reference sequences. Method 4 is the method in the worst case. Method 5 corresponds to the method used in Stemming.

because the constraint $dis(x, D) \leq 2d$ may filter out few l -mers x or not work.

6.3 Results on Simulated Data

The simulated data sets over an alphabet Σ are generated following [4], which are also used in [22] and [24]. 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 10 hours.

We compare StemFinder with the previous MSS algorithms, namely Stemming [22] and MSS1/MSS2 [24]. For the work [24], 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. We construct the set I for Stemming by using method 5 described in Section 6.2 to make fair comparisons. Consistent with [22] and [24], the data sets used to test algorithms are (7, 1), (9, 2) etc. over $|\Sigma| = 20$.

We show in Table 7 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 within 10 minutes. Both the running time of MSS2 and Stemming grows dramatically with the increase of l and d ; when $(l, d) = (23, 9)$, Stemming and MSS2 both require more than 10 hours.

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 significantly smaller number of stems compared to the method Stemming. The big difference may be caused by the placed wildcards. Unlike StemFinder and MSS2, Stemming allows a wildcard match any character over Σ , and thus it is easier for the generated stems to pass the stem verification. For StemFinder and MSS2, the former still 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 $|\Sigma|$,

TABLE 6
Number of Rough Stems under Different Methods for Constructing the Set I

(l, d)	p_{2d}	Number of rough stems				
		Method 1	Method 2	Method 3	Method 4	Method 5
(21, 10)	0.6594	9,520,861	18,247,993	20,144,453	35,101,288	19,749,463
(20, 9)	0.2642	914,972	2,317,364	2,665,449	5,704,815	2,746,893
(19, 8)	0.0665	93,388	378,975	663,315	1,587,200	656,747
(18, 7)	0.0109	5,190	60,977	64,117	229,664	63,920
(16, 6)	0.0070	1,425	9,144	22,700	80,032	12,997
(15, 5)	0.0006	188	1,079	4,105	13,080	746
(13, 4)	0.0003	20	188	525	2,671	52
(11, 3)	0.0001	8	42	113	1,238	18
(10, 2)	<0.0001	2	8	13	83	4

Method 1 is the method proposed in this paper. Method 2 is the method used in MSS1/MSS2. Method 3 constructs the set I by randomly selecting reference sequences. Method 4 is the method in the worst case. Method 5 corresponds to the method used in Stemming.

TABLE 7
Running Time of Stemfinder Compared with MSS2 and Stemming

(l, d)	StemFinder	MSS2	Stemming
(7, 1)	0.2 s	0.2 s	0.3 s
(9, 2)	0.2 s	0.2 s	0.3 s
(11, 3)	0.2 s	0.3 s	0.4 s
(13, 4)	0.2 s	1.9 s	5.0 s
(15, 5)	0.2 s	11.7 s	20.1 s
(17, 6)	0.2 s	1.1 m	1.7 m
(19, 7)	1.0 s	6.6 m	12.8 m
(21, 8)	5.6 s	2.1 h	3.4 h
(23, 9)	15.4 s	-o	-o
(25, 10)	3.5 m	-o	-o
(27, 11)	8.3 m	-o	-o
(29, 12)	8.2 m	-o	-o

$|\Sigma| - 1$, and $|\Sigma| - 1$ or $|\Sigma| - 2$ for the algorithms Stemming, MSS2, and StemFinder, respectively. In particular we find that the number of covered l -mers for StemFinder is about 1 percent of that of MSS2 and 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. For MSS2 and Stemming, although MSS2 reports a significantly smaller number of stems than Stemming, MSS2 does not show obvious advantages in the number of covered l -mers because of placing redundant wildcards.

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.

Moreover, we evaluate algorithms on the challenging problem instances [24] over $|\Sigma| = 20$, 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 8. We find that StemFinder is able to solve very challenging instances such as (21, 10) within 10 hours. Both MSS2 and Stemming can only solve two instances (7, 3) and (9, 4).

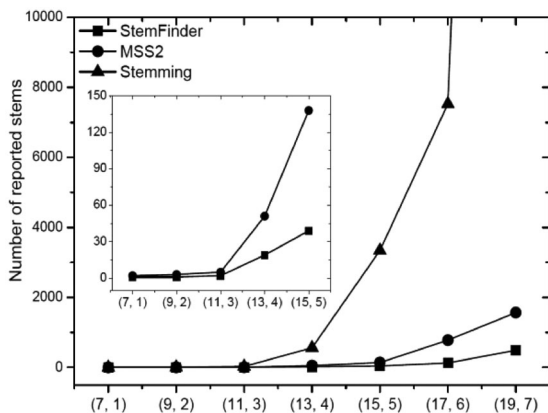


Fig. 4. The number of stems reported by different MSS algorithms.

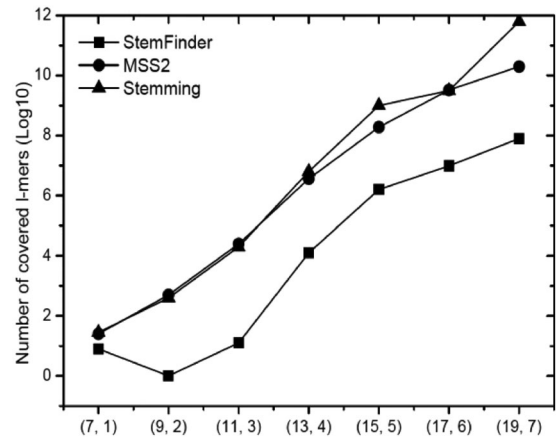


Fig. 5. The number of l -mers covered by reported stems for different MSS algorithms.

We further evaluate algorithms over large alphabets. We show the results in Table 9 with $|\Sigma| = 40, 60, 80$ and 100. From the table we see that with a fixed (l, d) instance, all the algorithms 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 the three algorithms, we find that StemFinder is often an order of magnitude faster than the other two algorithms.

To clearly show which part of the proposed algorithm responsible for the significant advantage in the running time, we break down the running time by steps in Table 10 and report some key statistics in Table 11, by testing the algorithms on the problem instance (9, 4) and (15, 5) over $|\Sigma| = 20, 40, 60, 80$ and 100. We find that StemFinder has the similar time performance to the other two algorithms in both step 1 (construct the set D) and step 2 (generate candidate stems), but show significant performance advantage in step 3 (verify candidate stems). The time performance of step 3 is mainly determined by the number of verified candidate stems. As shown in Table 11, the number of candidate stems generated by StemFinder is significantly smaller than that generated by others, owing to: first, our method of constructing the set I ensures the minimum number of rough stems; more importantly, in expanding rough stems to candidate stems, our pruning technique is very effective and avoids the verification of more than 98 percent candidate stems. Note that, the number of candidate stems generated by StemFinder without pruning is larger than that generated

TABLE 8
Results on Challenging Problem Instances

(l, d)	StemFinder	MSS2	Stemming
(7, 3)	2.6 m	26.7 m	22.2 m
(9, 4)	4.4 m	2.6 h	2.3 h
(11, 5)	6.9 m	-o	-o
(13, 6)	10.1 m	-o	-o
(15, 7)	13.6 m	-o	-o
(17, 8)	26.6 m	-o	-o
(19, 9)	53.9 m	-o	-o
(21, 10)	4.4 h	-o	-o

TABLE 9
Results over Large Alphabets

$ \Sigma $	(7, 3)			(9, 4)		
	SF ^a	MSS2	SM ^b	SF ^a	MSS2	SM ^b
40	30.4 s	12.9 m	9.7 m	53.5 s	1.2 h	1.1 h
60	12.9 s	8.1 m	6.3 m	21.2 s	46.4 m	36.8 m
80	7.2 s	6.4 m	4.4 m	11.5 s	33.5 m	30.3 m
100	4.7 s	4.3 m	3.9 m	7.3 s	25.7 m	21.8 m

^aSF is short-hand for StemFinder. ^bSM is short-hand for Stemming.

by others, which is not surprising because the stems obtained by expanding rough stems are more specific than that of MSS2 and Stemming.

6.4 Results on Real-World Data Sets with Protein Sequences

We identify the known linear motifs in protein sequences. There are several available linear motif databases, such as the Eukaryotic Linear Motif (ELM) database (<http://elm.eu.org>) [26], Minimotif Miner [27], [28], [29] and Scansite [2]. We choose the ELM database to collect our data sets, as it contains comprehensive resource of biologically validated linear motifs and provides detailed annotations for each motif [30] [31]. Each motif corresponds to a unique ELM identifier (ELM ID). We obtain 10 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.

We list the running time of different algorithms in Table 12. 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.

In addition, we demonstrate the validity of MSS algorithms for discovering motifs on real protein data sets. The exact algorithms for motif search usually report more than one motif, and some scoring scheme is needed to rank the reported motifs [12], [19]; here we adopt consensus score, the first scoring scheme given in [19]. We show in Table 13 the detected motifs, which are those with highest score in the reported stems able to span all motif instances. From the table we see a good matching between the detected motifs 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. The differences between the detected motifs and the ELM motifs are caused by that the ELM motifs are curated by hand and our results are completely obtained through computation without additional biological knowledge. Also, the rank of each detected motif in all reported stems is shown in Table 13. We can see that there are four data sets where our detected motifs have the best score, and there are six data sets where our detected motifs are included in the top 20 best reported stems.

TABLE 10
Running Time of Each Step on the (9, 4) and (15, 5) Problem Instance

(l, d)	$ \Sigma $	Time of step 1			Time of step 2			Time of step 3		
		StemFinder	MSS2	Stemming	StemFinder	MSS2	Stemming	StemFinder	MSS2	Stemming
(9, 4)	20	0.8 s	0.7 s	0.8 s	7.2 s	24.1 s	123.6 s	4.3 m	2.6 h	2.3 h
	40	0.5 s	0.5 s	0.6 s	3.5 s	13.5 s	59.3 s	49.5 s	1.2 h	1.1 h
	60	0.4 s	0.4 s	0.5 s	2.4 s	8.8 s	39.5 s	18.4 s	46.2 m	36.1 m
	80	0.3 s	0.4 s	0.5 s	1.9 s	6.6 s	29.9 s	9.3 s	33.4 m	29.8 m
	100	0.3 s	0.3 s	0.5 s	1.4 s	5.3 s	24.4 s	5.6 s	25.6 m	21.4 m
(15, 5)	20	0.2 s	0.2 s	0.2 s	0.1 s	0.1 s	1.1 s	0.1 s	11.4 s	18.8 s
	40	0.2 s	0.2 s	0.2 s	0.1 s	0.1 s	0.3 s	0.1 s	2.5 s	5.9 s
	60	0.2 s	0.2 s	0.2 s	0.1 s	0.1 s	0.2 s	0.1 s	11.6 s	12.8 s
	80	0.2 s	0.2 s	0.2 s	0.1 s	0.1 s	0.6 s	0.1 s	9.2 s	15.0 s
	100	0.2 s	0.2 s	0.2 s	0.1 s	0.1 s	0.5 s	0.1 s	9.7 s	13.9 s

TABLE 11
Key Statistics on the (9, 4) and (15, 5) Problem Instance

(l, d)	$ \Sigma $	Size of the set I			Number of candidate stems			
		StemFinder	MSS2	Stemming	StemFinder (without pruning)	StemFinder	MSS2	Stemming
(9, 4)	20	122,997	116,577	129,708	37,768,122	600,880	8,376,208	12,267,751
	40	65,442	59,756	70,950	18,025,040	95,710	4,196,924	5,822,322
	60	43,396	38,438	48,442	11,527,712	52,969	2,692,593	3,826,118
	80	31,940	28,563	38,379	8,344,006	36,988	2,105,329	2,951,130
	100	24,801	23,170	29,919	6,407,584	26,528	1,621,961	2,268,904
(15, 5)	20	13	11	7	19,606	635	9,754	14,917
	40	4	4	3	6,938	316	2,510	5,339
	60	7	7	4	20,601	838	11,391	12,200
	80	7	7	4	16,719	683	9,031	13,077
	100	7	7	4	17,277	875	9,765	12,470

TABLE 12
Running Time on ELM Data Sets

Data set (# instances)	ELM Motif	(<i>l</i> , <i>d</i>)	StemFinder	MSS2	Stemming
LIG_EVH1_1 (18)	([FYWL]P.PP) ([FYWL]PP[ALIVTFY]P)	(5, 1)	0.1 s	0.1 s	0.1 s
LIG_WW_1 (3)	PP.Y	(4, 1)	0.1 s	0.4 s	0.4 s
LIG_14-3-3_1 (3)	R.[^P](ST)[^P]P	(6, 2)	0.1 s	0.3 s	1.0 s
LIG_MYND_2 (3)	PP.LI	(5, 1)	0.3 s	1.4 s	3.7 s
LIG_USP7_1 (3)	[PA][^P][^FYWIL]S[^P]	(5, 2)	0.5 s	0.7 s	2.4 s
LIG_APCC_TPR_1 (22)	.[ILM]R\$	(3, 1)	10.3 s	3.9 s	3.9 s
LIG_MYND_1 (6)	P.L.P	(5, 2)	25.6 s	25.9 s	2.2 m
LIG_PAM2_1 (4)	..[LFP][NS][PIVTAFL].A..((([FY].[PYLF]) (W..)).	(13, 6)	1.0 m	-o	-o
MOD_NEK2_1 (3)	[FLM][^P][^P](ST)[^DEP][^DE]	(6, 3)	10.3 m	-o	-o
LIG_EABR_CEP55-1 (6)	.A.GPP.{2,3}Y.	(11, 5)	24.2 m	-o	-o

TABLE 13
Detected Motifs on ELM Data Sets

Data set	Detected Motif			Rank of detected motif		
	StemFinder	MSS2	Stemming	StemFinder	MSS2	Stemming
LIG_EVH1_1	FPPPP	FP*PP	FPPPP	1	5	1
LIG_WW_1	PPVY	PPVY	PP*Y	1	1	1
LIG_14-3-3_1	RSSSSP	RT*SSP	RSSSSP	344	148	771
LIG_MYND_2	PPPLI	PPPLI	PPPLI	1	1	1
LIG_USP7_1	-	-	-	-	-	-
LIG_APCC_TPR_1	-	-	-	-	-	-
LIG_MYND_1	PPLAP	PPLAP	PPLAP	2	2	2
LIG_PAM2_1	SAFNPNAKEFVPI	-	-	17	-	-
MOD_NEK2_1	FAESFS	-	-	1	-	-
LIG_EABR_CEP55-1	QAVGPPSLSYM	-	-	79	-	-

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.

ACKNOWLEDGMENTS

This research was supported in part by the National Natural Science Foundation of China (61173025 and 61373044), the Fundamental Research Funds for the Central Universities (K5051303032, K5051303002 and K50513100011), and the Natural Science Foundation of Shaanxi (2013JQ8037). A preliminary version [32] 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.

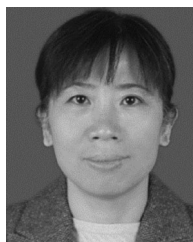
REFERENCES

- [1] P. D'haeseleer, "What are DNA sequence motifs?" *Nat. Biotechnol.*, vol. 24, no. 4, pp. 423–425, 2006.
- [2] J. C. Obenauer, L. C. Cantley, and M. B. Yaffe, "Scansite 2.0: Proteome-wide prediction of cell signaling interactions using short sequence motifs," *Nucleic Acids Res.*, vol. 31, pp. 3635–3641, 2003.
- [3] N. E. Davey, N. J. Haslam, D. C. Shields, and R. J. Edwards, "SLiMSearch 2.0: Biological context for short linear motifs in proteins," *Nucleic Acids Res.*, vol. 39, pp. W56–W60, 2011.
- [4] P. A. Pevzner and S. Sze, "Combinatorial approaches to finding subtle signals in DNA sequences," in *Proc. 8th Int. Conf. Intell. Syst. Molecular Biol.*, 2000, pp. 269–278.
- [5] P. A. Evans, A. D. Smith, and H. T. Wareham, "On the complexity of finding common approximate substrings," *Theor. Comput. Sci.*, vol. 306, pp. 407–430, 2003.
- [6] J. D. Thompson, D. G. Higgins, and T. J. Gibson, "CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice," *Nucleic Acids Res.*, vol. 22, pp. 4673–4680, 1994.
- [7] T. D. Schneider, "Consensus sequence zen," *Appl. Bioinformat.*, vol. 1, pp. 111–119, 2002.
- [8] 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.
- [9] T. Bailey and C. Elkan, "Fitting a mixture model by expectation maximization to discover motifs in biopolymers," in *Proc. 2nd Int. Conf. Intell. Syst. Molecular Biol.*, 1994, pp. 28–36.
- [10] Y. Zhang, H. Huo, and Q. Yu, "A heuristic cluster-based em algorithm for the planted (*l*, *d*) problem," *J. Bioinformat. Comput. Biol.*, vol. 11, no. 4, p. 1350009, 2013.
- [11] F. Y.L. Chin and H. C. M. Leung, "Voting algorithms for discovering long motifs," in *Proc. 3rd Asia Pacific Bioinformat. Conf.*, 2005, pp. 261–271.

- [12] J. Davila, S. Balla, and S. Rajasekaran, "Fast and practical algorithms for planted (l, d) motif search," *IEEE/ACM Trans. Comput. Biol. Bioinform.*, vol. 4, no. 4, pp. 544–552, Oct.–Dec. 2007.
- [13] E. S. Ho, C. D. Jakubowski, and S. I. Gunderson, "iTriplet, A rule-based nucleic acid sequence motif finder," *Algorithms Molecular Biol.*, vol. 4, p. 14, 2009.
- [14] 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. Comput. Biol. Bioinform.*, vol. 8, no. 5, pp. 1400–1410, Sep./Oct. 2011.
- [15] H. Dinh, S. Rajasekaran, and V. K. Kundeti, "PMS5: An efficient exact algorithm for the (l, d) -motif finding problem," *BMC Bioinform.*, vol. 12, p. 410, 2011.
- [16] 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, p. e48442, 2012.
- [17] 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, p. e41425, 2012.
- [18] Y. Xu, J. Yang, Y. Zhao, and Y. Shang, "An improved voting algorithm for planted (l, d) motif search," *Inf. Sci.*, vol. 237, pp. 305–312, 2013.
- [19] G. Pavesi, G. Mauri, and G. Pesole, "An algorithm for finding signals of unknown length in DNA sequences," *Bioinformatics*, vol. 17, no. Suppl 1, pp. S207–S214, 2001.
- [20] E. Eskin and P. A. Pevzner, "Finding composite regulatory patterns in DNA sequences," *Bioinformatics*, vol. 18, no. 1, pp. 354–363, 2002.
- [21] N. Pisanti, A. M. Carvalho, L. Marsan, and M. Sagot, "RISOTTO: Fast extraction of motifs with mismatches," in *Proc. 7th Latin Amer. Symp.: Theor. Inform.*, 2006, pp. 757–768.
- [22] P. Kuksa and V. Pavlovic, "Efficient motif finding algorithms for large-alphabet inputs," *BMC Bioinform.*, vol. 11, no. Suppl 8, p. S1, 2010.
- [23] P. Kuksa and V. Pavlovic, "Fast motif selection for biological sequences," in *Proc. IEEE Int. Conf. Bioinform. Biomed.*, 2009, pp. 79–82.
- [24] T. Mi and S. Rajasekaran, "Efficient algorithms for biological stems search," *BMC Bioinform.*, vol. 14, p. 161, 2013.
- [25] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 2nd ed. Reading, MA, USA: Addison Wesley, 2001, pp. 83–122.
- [26] 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 Res.*, vol. 40, pp. D242–D251, 2012.
- [27] S. Rajasekaran, S. Balla, P. Gradie, M. R. Gryk, K. Kadaveru, V. Kundeti, M. W. Maciejewski, T. Mi, N. Rubino, J. Vyas, and M. R. Schiller, "Minimotif miner 2nd release: A database and web system for motif search," *Nucleic Acids Res.*, vol. 37, pp. D185–D190, 2009.
- [28] T. Mi, S. Rajasekaran, J. C. Merlin, M. Gryk, and M. R. Schiller, "Achieving high accuracy prediction of minimotifs," *PLoS One*, vol. 7, no. 9, p. e45589, 2012.
- [29] T. Mi, J. C. Merlin, S. Deverasetty, M. R. Gryk, T. J. Bill, A. W. Brooks, L. Y. Lee, V. Rathnayake, C. A. Ross, D. P. Sargeant, C. L. Strong, P. Watts, S. Rajasekaran, and M. R. Schiller, "Minimotif miner 3.0: Database expansion and significantly improved reduction of false-positive predictions from consensus sequences," *Nucleic Acids Res.*, vol. 40, pp. D252–D260, 2012.
- [30] R. J. Edwards, N. E. Davey, and D. C. Shields, "SLIMFinder: A probabilistic method for identifying over-represented, convergently evolved, short linear motifs in proteins," *PLoS One*, vol. 2, p. e967, 2007.
- [31] N. E. Davey, N. J. Haslam, D. C. Shields, and R. J. Edwards, "SLIMFinder: A web server to find novel, significantly over-represented, short protein motifs," *Nucleic Acids Res.*, vol. 38, pp. W534–W549, 2010.
- [32] Q. Yu, H. Huo, J. S. Vitter, J. Huan, and Y. Nekrich, "StemFinder: An efficient algorithm for searching motif stems over large alphabets," in *Proc. IEEE Int. Conf. Bioinform. Biomed.*, 2013, pp. 473–476.



Qiang Yu received the BS degree and the MS degree from Xidian University in 2006 and 2009, respectively, where he is currently working toward the PhD degree. His research interests include design and analysis of algorithms, bioinformatics, and parallel and distributed computing.



Hongwei Huo received the BS degree in mathematics from Northwest University, the MS degree in computer science and the PhD degree in electronic engineering from Xidian University, China. She is a professor and a chair in the Department of Computer Science at Xidian University. Her research interests include the design and analysis of algorithms, bioinformatics algorithms, external memory algorithms and compressed indexes, data compression, parallel and distributed algorithms, algorithm engineering. She is a member of the IEEE and the IEEE Computer Society.



Jeffrey Scott Vitter received the BS degree in mathematics with highest honors from the University of Notre Dame in 1977, the PhD degree in computer science from Stanford University in 1980, and the MBA degree from Duke University in 2002. He is provost and executive vice chancellor and Roy A. Roberts distinguished professor at the University of Kansas. His academic home is the Department of Electrical Engineering and Computer Science, and he is a member of the Information and Telecommunication Technology Center. From 2008 to 2010, he was in the faculty at TexasA&M University, where he served as provost and executive vice president for academics. From 2002 to 2008, he was the Frederick L. Hovde dean of the College of Science and professor of Computer Science at Purdue University. From 1993 to 2002, he held the Gilbert, Louis, and Edward Lehrman distinguished professorship at Duke University, where he also served as chair of the Department of Computer Science and co-director of Duke's Center for Geometric and Biological Computing. From 1980 to 1992, he advanced through the faculty ranks in computer science at Brown University. He is a Guggenheim fellow, ACM fellow, AAAS fellow, US National Science Foundation (NSF) Presidential Young Investigator, and Fulbright Scholar. He has received the IBM Faculty Development Award, ACM Recognition of Service Award (twice), and 2009 ACM SIGMOD Test of Time Award. He sits on the board of advisors of the School of Science and Engineering at Tulane University. From 2000 to 2009, he served on the board of directors of the Computing Research Association (CRA), where he continues to co-chair the Government Affairs Committee. He has served as chair of ACM SIGACT and on the EATCS executive committee. He is author of the book *Algorithms and Data Structures for External Memory*, coauthor of the books *Design and Analysis of Coalesced Hashing* and *Efficient Algorithms for MPEG Video Compression*, co-editor of the collections *External Memory Algorithms* and *Algorithm Engineering*, and coholder of patents in the areas of external sorting, prediction, and approximate data structures. His research interests span the design and analysis of algorithms, external memory algorithms, data compression, databases, compressed data structures, parallel algorithms, machine learning, random variate generation, and sampling. He serves or has served on the editorial boards of *Algorithmica*, *Communications of the ACM*, *IEEE Transactions on Computers*, *Theory of Computing Systems*, and *SIAM Journal on Computing*, and has edited several special issues. He proposed the concept and participated in the design of what has become the Purdue University Research Expertise database (PURE) and the Indiana Database for University Research Expertise (INDURE), www.indure.org. He is a fellow of the IEEE.



Jun Huan received the BS degree in biochemistry & molecular biology from Peking University, China, in 1997. He received the MS in computer science from the Oklahoma State University in 2000, and the PhD degree in computer science from the University of North Carolina, Chapel Hill, in 2006. He joined the Department of Electrical Engineering and Computer Science at the University of Kansas in 2006 and is currently a professor. At KU, he directs the Bioinformatics and Computational Life Sciences Laboratory at KU

Information and Telecommunication Technology Center (ITTC) and the Cheminformatics core at KU Specialized Chemistry Center, funded by NIH. He holds courtesy appointments at the KU Bioinformatics Center, the KU Bioengineering Program, and a visiting professorship from GlaxoSmithKline plc. He received the National Science Foundation Faculty Early Career Development Award in 2009. His group won the Best Student Paper Award at the IEEE International Conference on Data Mining in 2011 and the Best Paper Award (runner-up) at the ACM International Conference on Information and Knowledge Management in 2009. He is a member of IEEE.



Yakov Nekrich received the BS and MS degrees in computer science from the University of Latvia and the PhD in computer science from the University of Bonn. He is a researcher at the University of Waterloo. He has published 60 conference and journal papers in leading international venues. His main research interests are design of efficient algorithms and data structures, data compression, external memory algorithms, and compressed data structures.

▷ **For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.**