Improved Algorithms for Finding Edit Distance Based Motifs

Soumitra Pal and Sanguthevar Rajasekaran
Computer Science and Engineering, University of Connecticut
371 Fairfield Road, Storrs, CT 06269, USA
Email: mitra@uconn.edu, rajasek@engr.uconn.edu

***Abstract*—Motif search is an important step in extracting meaningful patterns from biological data. The general problem of motif search is intractable. There is a pressing need to develop efficient exact and approximation algorithms to solve this problem. In this paper we present novel algorithms for solving the (l, d) *Edit-distance-based Motif Search (EMS)* problem: given two integers l, d and n biological strings, find all strings of length l that appear in each input string with atmost d substitutions, insertions and deletions. The algorithms for EMS are customarily evaluated on several challenging instances such as $(9, 2)$, $(11, 3)$, $(13, 4)$, $(15, 5)$, and so on. The best previously known algorithm, EMS1, solves up to instance $(11, 3)$ in estimated 3 days.**

Our algorithm is more than 20 times faster than EMS1. For example, our algorithm solves instance $(11, 3)$ in a couple of minutes and instance $(14, 3)$ in a couple of hours. This significant improvement is due to a novel and provably efficient neighborhood generation technique introduced in this paper. Firstly, we show that it is enough to consider the neighbors which are at a distance exactly d from all possible substrings of the input strings. Secondly, we compactly represent the candidate motifs in the neighborhood using wildcard characters. Thirdly, we generate these compact candidate motifs nearly uniquely with very few repetitions. Finally, we use a trie based data structure to efficiently store the candidate motifs and to output the final motifs in a sorted order. We believe that the techniques we introduce in this paper are also applicable to other motif search problems such as the PMS.

I. Introduction

Motif search has applications in solving such crucial problems as identification of alternative splicing sites, determination of open reading frames, identification of promoter elements of genes, identification of transcription factors and their binding sites, etc. (see e.g., Nicolae and Rajasekaran [1]). There are many formulations of the motif search problem. A widely studied formulation is known as (l, d)-motif search or Planted Motif Search (PMS) [2]. Given two integers l, d and n biological strings the problem is to find all strings of length l that appear in each of the n input strings with atmost d mismatches. There is a significant amount of work in the literature on PMS (see e.g., [1], [3]–[5], and so on).

PMS considers only point mutations as events of divergence in biological sequences. However, insertions and deletions also play important roles in divergence [2], [6]. Therefore, researchers have also considered a formulation in which the Levenshtein distance (or edit distance), instead of mismatches, is used for measuring the degree of divergence [7], [8]. Given n strings $S^{(1)}, S^{(2)}, \ldots, S^{(n)}$, each of length m from a fixed alphabet Σ, and integers l, d, the *Edit-distance-based Motif Search (EMS) problem* is to find all patterns M of length l that occur in atleast one position in each $S^{(i)}$ with an edit distance of atmost d. More formally, M is a motif if and only if $\forall i$, there exist $k \in [l-d, l+d], j \in [1, m-k+1]$ such that for the substring $S^{(i)}_{j,k}$ of length k at position j of $S^{(i)}$, $ED(S^{(i)}_{j,k}, M) \le d$. Here $ED(X, Y)$ stands for the edit distance between two strings X and Y.

EMS is also NP-hard since PMS is a special case of EMS and PMS is known to be NP-hard [9]. As a result, any exact algorithm for EMS that finds all the motifs for a given input can be expected to have an exponential (in some of the parameters) worst case runtime. One of the earliest EMS algorithms is due to Rocke and Tompa [7] and is based on Gibbs Sampling which requires repeated searching of the motifs in a constantly evolving collection of aligned strings, and each search pass requires $O(nl)$ time. This is an approximate algorithm. Sagot [8] gave a suffix tree based exact algorithm that takes $O(n^2 m l^d |\Sigma|^d)$ time and $O(n^2 m/w)$ space where w is the word length of the computer. Adebiyi and Kaufmann [10] proposed an exact algorithm with an expected runtime of $O(nm + d(nm)^{(1+pow(\epsilon))} \log nm)$ where $\epsilon = d/l$ and $pow(\epsilon)$ is an increasing concave function. The value of $pow(\epsilon)$ is roughly 0.9 for protein and DNA sequences. Wang and Miao [11] gave an expectation minimization based heuristic genetic algorithm.

Rajasekaran et al. [12] proposed a simpler Deterministic Motif Search (DMS) that has the same worst case time complexity as the algorithm by Sagot [8]. The algorithm generates and stores the neighborhood of every substring of length in the range $[l-d, l+d]$ of every input string and using a radix sort based method outputs the neighbors that are common to atleast one substring of each input string. This algorithm was implemented by Pathak et al. [13].

Following a useful practice for PMS algorithms, Pathak et al. [13] evaluated their algorithm on certain instances that are considered challenging for PMS: $(9, 2), (11, 3), (13, 4)$ and so on [1], and are generated as follows: $n = 20$ random DNA/protein strings of length $m = 600$, and a short random string M of length l are generated according to the independent identically distributed (i.i.d) model. A separate random d-hamming distance neighbor of M is "planted" in each of the n input strings. Such an (l, d) instance is defined to be a *challenging instance* if l is the largest integer for which the expected number of spurious motifs, i.e., the motifs that would occur in the input by random chance, is atleast 1.

The expected number of spurious motifs in EMS are different from those in PMS. Table I shows the expected number of spurious motifs for $l \in [5, 21]$ and d upto $\max\{l-2, 17\}$ computed using (8) given in Appendix A. The challenging instances for EMS turn out to be $(8, 1), (12, 2), (16, 3), (20, 4)$ and so on. To compare with [13], we consider both types of instances, specifically, $(8, 1), (9, 2), (12, 2), (11, 3), (13, 4)$.

TABLE I: Expected number of spurious motifs in random instances for $n = 20, m = 600$. Here, ∞ represents value $\geq 1.0e+7$.

l	d=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
5	0.0	1024.0	1024.0	∞														
6	0.0	4096.0	4096.0	∞	∞													
7	0.0	14141.8	16384.0	∞	∞	∞												
8	0.0	**225.8**	65536.0	65536.0	∞	∞	∞											
9	0.0	0.0	262144.0	262144.0	∞	∞	∞	∞										
10	0.0	0.0	1047003.6	1048576.0	∞	∞	∞	∞	∞									
11	0.0	0.0	1332519.5	4194304.0	∞	∞	∞	∞	∞	∞								
12	0.0	0.0	**294.7**	1.678e+07	1.678e+07	∞	∞	∞	∞	∞	∞							
13	0.0	0.0	0.0	6.711e+07	6.711e+07	∞	∞	∞	∞	∞	∞	∞						
14	0.0	0.0	0.0	2.517e+08	2.684e+08	∞	∞	∞	∞	∞	∞	∞	∞					
15	0.0	0.0	0.0	2.749e+07	1.074e+09	∞	∞	∞	∞	∞	∞	∞	∞	∞				
16	0.0	0.0	0.0	**139.1**	4.295e+09	4.295e+09	∞	∞	∞	∞	∞	∞	∞	∞	∞			
17	0.0	0.0	0.0	0.0	1.718e+10	1.718e+10	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞		
18	0.0	0.0	0.0	0.0	3.965e+10	6.872e+10	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	
19	0.0	0.0	0.0	0.0	1.226e+08	2.749e+11	2.749e+11	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
20	0.0	0.0	0.0	0.0	**35.8**	1.100e+12	1.100e+12	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
21	0.0	0.0	0.0	0.0	0.0	4.333e+12	4.398e+12	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

The algorithm by Pathak et al. [13] solves the moderately hard instance $(11, 3)$ in a few hours and does not solve the next difficult instance $(13, 4)$ even after 3 days. A key time consuming part of the algorithm is in the generation of the edit distance neighborhood of all substrings as there are many common neighbors.

A. Contributions

In this paper we present an improved algorithm for EMS that solves instance $(11, 3)$ in less than a couple of minutes and instance $(14, 3)$ in less than a couple of hours. Our algorithm uses an efficient technique (introduced in this paper) to generate the edit distance neighborhood of length l with distance atmost d of all substrings of an input string. Our approach uses following four ideas which can be applied to other motif search problems too:

a) Efficient neighborhood generation: We show that it is enough to consider the neighbors which are at a distance exactly d from all possible substrings of the input strings. This works because the neighbors at a lesser distance are also included in the neighborhood of some other substrings.

b) Compact representation using wildcard characters: We represent all possible neighbors which are due to an insertion or by a substitution at the same position by using a wildcard character at that position. This compact representation of the candidate motifs in the neighborhood requires less space.

c) Avoiding duplication of candidate motifs: Our algorithm uses several rules to avoid duplication in candidate motifs and we prove that our technique generates neighborhood that is nearly duplication free. In other words, our neighborhood generation technique does not spend a lot of time generating neighbors that have already been generated.

d) Trie based data structure: We use a trie based data structure to efficiently store the neighborhood. This not only simplifies the removal of duplicate neighbors but also helps in outputting the final motifs in sorted order using a depth first search traversal of the trie.

A C++ based implementation of our algorithm is available at https://github.com/soumitrakp/ems2.git.

II. Methods

In this section we introduce some notations and observations.

An (l, d)-*friend* of a k-mer L is an l-mer at an exact distance of d from L. Let $F_{l,d}(L)$ denote the set of all (l, d)-friends of L. An (l, d)-*neighbor* of a k-mer L is an l-mer at a distance of atmost d from L. Let $N_{l,d}(L)$ denote the set of all (l, d)-neighbors of L. Then

$$N_{l,d}(L) = \cup_{t=0}^{d} F_{l,t}(L). \quad (1)$$

For a string S of length m, an (l, d)-*motif* of S is an l-mer at a distance atmost d from some substring of S. Thus an (l, d)-motif of S is an (l, d)-neighbor of atleast one substring $S_{j,k} = S_j S_{j+1} \ldots S_{j+k-1}$ where $k \in [l - d, l + d]$. Therefore, the set of (l, d)-motifs of S, denoted by $M_{l,d}(S)$, is given by

$$M_{l,d}(S) = \cup_{k=l-d}^{l+d} \cup_{j=1}^{m-k+1} N_{l,d}(S_{j,k}). \quad (2)$$

For a collection of strings $\mathcal{S} = \{S^{(1)}, S^{(2)}, \ldots, S^{(m)}\}$, a (common) (l, d)-motif is an l-mer at a distance atmost d from atleast one substring of each $S^{(i)}$. Thus the set of (common) (l, d)-motifs of $\mathcal{S}$, denoted by $M_{l,d}(\mathcal{S})$, is given by

$$M_{l,d}(\mathcal{S}) = \cap_{i=1}^{n} M_{l,d}(S^{(i)}). \quad (3)$$

One simple way of computing $F_{l,d}(L)$ is to grow the friendhood of L by one distance at a time for d times and to select only the friends having length l. Let $G(L)$ denote the set of strings obtained by one edit operation on L and $G(\{L_1, L_2, \ldots, L_r\}) = \cup_{t=1}^{r} G(L_t)$. If $G^1(L) = G(L)$, and for $t > 1$, $G^t(L) = G(G^{t-1}(L))$ then

$$F_{l,d}(L) = \{x \in G^d(L) : |x| = l\}. \quad (4)$$

Using equations (1), (2), (3) and (4), Pathak et al. [13] gave an algorithm that stores all possible candidate motifs in an array of size $|\Sigma|^l$. However the algorithm is inefficient in generating the neighborhood as the same candidate motif is generated by several combinations of the basic edit operations. Also, the $O(|\Sigma|^l)$ memory requirement makes the algorithm inapplicable for larger instances. In this paper we mitigate these two limitations.

A. Efficient Neighborhood Generation

We now give a more efficient algorithm to generate the (l, d)-neighborhood of all possible k-mers of a string. Instead of computing (l, t)-friendhood for all $0 \leq t \leq d$, we compute only the exact (l, d)-friendhood.

Lemma 1. $M_{l,d}(S) = \cup_{k=l-d}^{l+d} \cup_{j=1}^{m-k+1} F_{l,d}(S_{j,k})$.

Proof: Consider the k-mer $L = S_{j,k}$. If $k = l + d$ then we need d deletions to make L an l-mer. There cannot be any (l, t)-neighbor of L for $t < d$. Thus

$$\cup_{t=0}^{d} F_{l,t}(S_{j,l+d}) = F_{l,d}(S_{j,l+d}). \quad (5)$$

Suppose $k < l+d$. Any $(l, d-1)$-neighbor B of L is also an (l, d)-neighbor of $L' = S_{j,k+1}$ because $ED(B, L') \leq ED(B, L) + ED(L, L') \leq (d-1) + 1 = d$. Thus

$$\cup_{t=0}^{d} F_{l,t}(S_{j,k}) \subseteq F_{l,d}(S_{j,k}) \bigcup \cup_{t=0}^{d} F_{l,t}(S_{j,k+1})$$

which implies that

$$\cup_{r=k}^{k+1} \cup_{t=0}^{d} F_{l,t}(S_{j,r}) = F_{l,d}(S_{j,k}) \bigcup \cup_{t=0}^{d} F_{l,t}(S_{j,k+1}). \quad (6)$$

Applying (6) repeatedly for $k = l-d, l-d+1, \ldots, l+d-1$, along with (5) in (1) and (2) gives the result of the lemma. ■

We generate $F_{l,d}(S_{j,k})$ in three phases: we apply δ deletions in the first phase, β substitutions in the second phase, and finally α insertions in the final phase, where $d = \delta + \alpha + \beta$ and $l = k - \delta + \alpha$. Solving for α, β, δ gives $\max\{0, q\} \leq \delta \leq (d+q)/2$, $\alpha = \delta - q$ and $\beta = d - 2\delta + q$ where $q = k - l$. In each of the phases, the neighborhood is grown by one edit operation at a time.

B. Compact Motifs

The candidate motifs in $F_{l,d}(S_{j,k})$ are generated in a compact way. Instead of inserting each character in Σ separately at a required position in $S_{j,k}$, we insert a new character $* \notin \Sigma$ at that position. Similarly, instead of substituting a character $\sigma \in S_{j,k}$ by each $\sigma' \in \Sigma \setminus \{\sigma\}$ separately, we substitute σ by $*$. The motifs common to all strings in $\mathcal{S}$ is determined by using the usual definition of union and the following definition of intersection on compact strings $A, B \in (\Sigma \cup \{*\})^l$ in (3):

$$A \cap B = \begin{cases} \emptyset & \text{if } \exists j \text{ s.t. } A_j, B_j \in \Sigma, A_j \neq B_j \\ \sigma_1\sigma_2\ldots\sigma_l & \text{else, where } \sigma_j = \begin{cases} b_j & \text{if } a_j = * \\ a_j & \text{if } b_j = * \end{cases}. \end{cases} \quad (7)$$

C. Trie for Storing Compact Motifs

We store the compact motifs in a trie based data structure which we call a *motif trie*. This helps implement the intersection defined in (7). Each node in the motif trie has atmost $|\Sigma|$ children. The edges from a node u to its children v are labeled with mutually exclusive subsets $label(u, v) \subseteq \Sigma$. An empty set of compact motifs is represented by a single root node. A non-empty trie has $l+1$ levels of nodes, the root being at level 0. The trie stores the l-mer $\sigma_1\sigma_2\ldots\sigma_l$, all $\sigma_j \in \Sigma$, if there is a path from the root to a leaf where σ_j appears in the label of the edge from level $j-1$ to level j.

For each string $S = \mathcal{S}^{(i)}$ we keep a separate motif trie $M^{(i)}$. Each compact neighbor $A \in F_{l,d}(S_{j,k})$ is inserted into the motif trie recursively as follows. We start with the root node where we insert $A_1A_2\ldots A_l$. At a node u at level j where the prefix $A_1A_2\ldots A_{j-1}$ is already inserted, we insert the suffix $A_jA_{j+1}\ldots A_l$ as follows. If $A_j \in \Sigma$ we insert $A' = A_{j+1}A_{j+2}\ldots A_l$ to the children v of u such that $A_j \in label(u, v)$. If $label(u, v) \neq \{A_j\}$, before inserting we make a copy of sub trie rooted at v. Let v' be the root of the new copy. We make v' a new child of u, set $label(u, v') = \{A_j\}$, remove A_j from $label(u, v)$, and insert A' to v'. On the other hand if $A_j = *$ we insert A' to each children of u. Let $T = \Sigma$ if $A_j = *$ and $T = \{A_j\}$ otherwise. Let $R = T \setminus \cup_v label(u, v)$. If $T \neq \emptyset$ we create a new child v' of u, set $label(u, v') = R$ and recursively insert A' to v'. Fig. 1 shows examples of inserting into the motif trie.

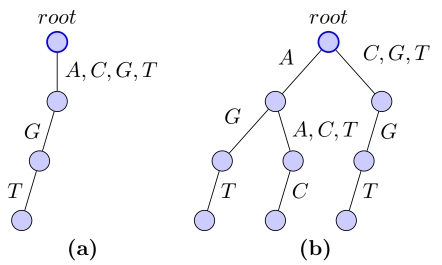

Fig. 1: Inserting into motif trie for $\Sigma = \{A, C, G, T\}$ and $l = 2$. (a) After inserting $*GT$ into empty trie. (b) After inserting another string $A*C$.

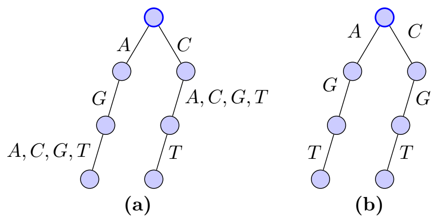

Fig. 2: Intersection of motif tries. (a) Trie for $AG* \cup C*T$. (b) Intersection of trie in Fig. 1(b) and trie in Fig. 2(a).

We also maintain a motif trie for the $\mathcal{M}$ for the common compact motifs found so far, starting with $\mathcal{M} = M^{(1)}$. After processing string $S^{(i)}$ we intersect the root of $M^{(i)}$ with the root of $\mathcal{M}$. In general a node $u_2 \in M^{(i)}$ at level j is intersected with a node $u_1 \in \mathcal{M}$ at level j using the procedure shown in Algorithm 1. Fig. 2 shows an example of the intersection of two motif tries.

Algorithm 1: Intersect subtries u_1, u_2 with result in u_1

```
$V \leftarrow$ all children of $u_1$;
foreach $v_1 \in V$ do
    foreach child $v_2$ of $u_2$ do
        $newLabel \leftarrow label(u_1, v_1) \cap label(u_2, v_2)$;
        if $newLabel \neq \emptyset$ then
            if $newLabel \neq label(u_1, v_1)$ then
                let $v_1'$ be a new child of $u_1$;
                copy at $v_1'$ the sub trie rooted at $v_1$;
                $label(u_1, v_1') \leftarrow newLabel$;
                $label(u_1, v_1) \leftarrow label(u_1, v_1) \setminus newLabel$;
                recursively intersect $v_1'$ with $v_2$;
            else
                recursively intersect $v_1$ with $v_2$;
    if $label(u_1, v_1) = \emptyset$ then delete sub trie rooted at $v_1$;
if $u_1$ has no child then delete sub trie rooted at $u_1$;
```

The final set of common motifs is obtained by a depth-first traversal of $\mathcal{M}$ outputting the label of the path from the root whenever a leaf is traversed. An edge (u, v) is traversed separately for each $\sigma \in label(u, v)$.

D. Efficient Compact Neighborhood Generation

A significant part of the time taken by our algorithm is in inserting compact neighbors into the motif trie as it is executed for each neighbor in the friendhood. Even after using the tricks described in sections II-A and II-B the set of generated compact motifs are not totally duplication free. We use few simple rules to reduce duplication further. Later we will see that these rules are quite close to the ideal as we will prove

TABLE II: Conditions for skipping motif $L = \langle M, S_{j,k}, T\rangle$

Rule	**Conditions** (in all rules $t \geq 0$)
1	$(j + k \leq m) \wedge \langle j, D\rangle \in T$
2	$\{\langle j + t, D\rangle, \langle j + t + 1, R\rangle\} \subseteq T$
3	$(j{+}k{\leq}m) \wedge \{\langle j, R\rangle,\langle j{+}1, R\rangle, \ldots ,\langle j{+}t, R\rangle,\langle j{+}t{+}1, D\rangle\}{\subseteq}T$
4	$\{\langle j + t, D\rangle, \langle j + t, I\rangle\} \subseteq T$
5	$\{\langle j + t, D\rangle, \langle j + t + 1, I\rangle\} \subseteq T$
6	$\{\langle j + t, R\rangle, \langle j + t, I\rangle\} \subseteq T$
7	$(j > 1) \wedge \langle j, I\rangle \in T$
8	$(j > 1) \wedge \{\langle j, R\rangle, \langle j{+}1, R\rangle, \ldots , \langle j{+}t, R\rangle, \langle j{+}t{+}1, I\rangle\}{\subseteq}T$
9	$(j + k \leq m) \wedge \langle j{+}k, I\rangle{\in}T$

that the compact motif generated after skipping using the rules, are distinct if all the characters in the input string are distinct.

To differentiate multiple copies of the same compact motif, we augment it with the information about how it is generated. Formally, each instance L of a compact motif M is represented as an ordered tuple $\langle M, S_{j,k}, T\rangle$ where the sequence of edit operations T when applied to $S_{j,k}$ gives M. Each edit operation in T is represented as a tuple $\langle p, t\rangle$ where p denotes the position (as in S) where the edit operation is applied and $t \in \{D, R, I\}$ denotes the type of the operation – deletion, substitution and insertion, respectively. At each position there can be one deletion or one substitution but one or more insertions. The tuples in T are sorted lexicographically with the normal order for p and for t, $D < R < I$.

Let $\bar{M}_{l,d}(S)$ denote the set of tuples for the compact motifs of S that were not skipped by our algorithm using the rules in Table II and $M_{l,d}(S)$ be the set of compact motifs generated by (3). Let $\Gamma(\langle M, S_{j,k}, T\rangle) = M$ and $\Gamma(Z) = \cup_{L \in Z}\Gamma(L)$.

Lemma 2. $\Gamma(\bar{M}_{l,d}(S)) = M_{l,d}(S)$.

Proof: By construction, $\Gamma(\bar{M}_{l,d}(S)) \subseteq M_{l,d}(S)$. We show by contradiction that $M_{l,d}(S) \subseteq \Gamma(\bar{M}_{l,d}(S))$.

Let $L_1 = \langle M_1, S_{j_1,k_1}, T_1\rangle$ and $L_2 = \langle M_2, S_{j_2,k_2}, T_2\rangle$ be two elements of $\bar{M}_{l,d}(S)$ and $\langle p_1, t_1\rangle \in T_1, \langle p_2, t_2\rangle \in T_2$ be the leftmost edit operations where T_1, T_2 differ. We impose an order $L_1 < L_2$ if and only if $(k_1 < k_2) \vee ((k_1 = k_2) \wedge (p_1 < p_2)) \vee ((k_1 = k_2) \wedge (p_1 = p_2) \wedge (t_1 < t_2))$.

Let $L = \langle M, S_{j,k}, T\rangle$ be the largest (in the order defined above) tuple skipped by our algorithm such that there is no other $L'' \in \bar{M}_{l,d}(S)$ and $\Gamma(L'') = M$. For each Rule 1-9 we show a contradiction that if L is skipped by the rule then there is another $L' = \langle M, S_{j',k'}, T'\rangle \in \bar{M}_{l,d}(S)$ with the same number of edit operations but $L < L'$. Fig. 3 illustrates the choice of L' under different rules.

Rule 1. Here $j + k \leq m$ and $\langle j, D\rangle \in T$. Consider $T' = T \setminus \langle j, D\rangle) \cup \langle j + k, D\rangle$, and $j' = j + 1, k' = k$.

Rule 2. Consider $T' = T \setminus \{\langle j + t, D\rangle, \langle j + t + 1, R\rangle\} \cup \{\langle j + t, R\rangle, \langle j + t + 1, D\rangle\}$, and $j' = j, k' = k$.

Rule 3. $T' = T\setminus\{\langle j, R\rangle, \langle j{+}t{+}1, D\rangle\}\cup\{\langle j{+}t{+}1, R\rangle, \langle j+k, D\rangle\}$, $j' = j + 1, k' = k$.

Rule 4. For this and subsequent rules $k < l + d$ as there is atleast one insertion and hence k' could possibly be equal to $k + 1$. We consider two cases. Case (i) $j + k \leq m$: $T' = T\setminus\{\langle j{+}t, D\rangle, \langle j{+}t, I\rangle\}\cup\{\langle j{+}t, R\rangle, \langle j{+}k, D\rangle\}$, $j' = j, k' = k{+}1$. Case (ii) $j{+}k = m{+}1$: Here deletion of S_j is allowed by Rule 1. $T' = T\setminus\{\langle j{+}t, D\rangle, \langle j{+}t, I\rangle\}\cup\{\langle j{-}1, D\rangle, \langle j{+}t, R\rangle\}$, $j' = j - 1, k' = k + 1$.

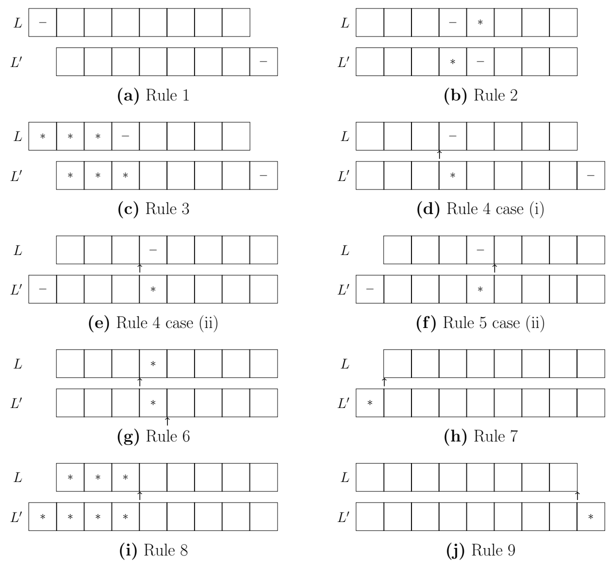

Fig. 3: Construction of L' under different rules in the proof of Lemma 2. Insertions are shown using arrows, deletions using $-$ and substitutions using $*$. Rule 5 case (i) is similar to Rule 4 case (i).

Rule 5. The same argument for Rule 4 applies considering $\langle j{+}t{+}1, I\rangle$ instead of $\langle j{+}t, I\rangle$.

Rule 6. $T'{=}T\setminus\{\langle j{+}t, I\rangle\}\cup\{\langle j{+}t{+}1, I\rangle\}$, and $j'{=}j, k'{=}k$.

Rule 7. $T'{=}T\setminus\{\langle j, I\rangle\}\cup\{\langle j{-}1, R\rangle\}$, $j' = j - 1, k' = k{+}1$.

Rule 8. $T'{=}T\setminus\{\langle j{+}t, I\rangle\}\cup\{\langle j{-}1, R\rangle\}$, $j'{=}j{-}1, k' = k{+}1$.

Rule 9. $T'{=}T\setminus\{\langle j{+}k, I\rangle\}\cup\{\langle j{+}k, R\rangle\}$, $j' = j, k' = k{+}1$. ■

Lemma 3. *If S_js are all distinct then for any two distinct $L, L' \in \bar{M}_{l,d}(S)$, we have $\Gamma(L) \neq \Gamma(L')$.*

Proof: We prove by contradiction. Let $L = \langle M, S_{j,k}, T\rangle$ and $L' = \langle M, S_{j',k'}, T'\rangle$ be two distinct elements of $\bar{M}_{l,d}(S)$. We start with the four strings $O = S_{j,k}, O' = S_{j',k'}, N = N' = M$ and align them as follows. The characters common to all four are aligned first. For each S_p present only in one of O, O', we insert S_p in the other and insert $-$ in one of N, N', as appropriate. For each $\langle p, t\rangle \in T$, if $t = I$ we insert a $-$ at the corresponding position in O, O', N'. If $t = D$ we insert a $-$ at the corresponding position in N. If $t = R$ we align the corresponding $*$ in N with S_p in O. We repeat the analogous for $\langle p, t\rangle \in T'$ but making sure that only a single $-$ is inserted if both T, T' have an insertion at the same position, or both T, T' have a deletion at the same position.

Without loss of generality, assume $j \leq j'$. If $j < j'$ then all of $S_j, S_{j+1}, \ldots, S_{j'-1}$ are either deleted or substituted in N. If $S_{j,k}$ is the not the rightmost k-mer of S then by Rule 1, S_j cannot be deleted in N and hence must be substituted. Then by Rule 3 all of $S_{j+1}, \ldots, S_{j'-1}$ are also substituted in N. Since the leftmost non $-$ character in N' must be $*$, $S_{j',k'}$ must be substituted in N' because by Rule 7 no insertion is possible just before $S_{j'}$ in N'. Since $S_{j'}$ cannot be deleted in N by Rule 3, $S_{j'}$ must be substituted in N. This implies there must be another $*$ just after the alignment of $S_{j'}$ in N'. Since by Rule 8, this $*$ cannot be due to an insertion, $S_{j'+1}$ must be substituted in N' which enforces $S_{j'+1}$ to be substituted in N. By repeating this argument, all characters in N would be $*$ which is not possible. Thus either $j = j'$ or $S_{j,k}$ is the

TABLE III: Possible alignments at position p where atleast one of the pairs O_p, O'_p and N_p, N'_p differs

Case	O_p	N_p	O'_p	N'_p	Comments
1	σ	σ	σ	$*$	Not possible as there is no other character to match σ in N'
2	σ	σ	σ	$-$	Same as case 1
3	σ	$*$	σ	σ	Symmetric to case 1
4	σ	$*$	σ	$-$	Discussed in details in the text
5	σ	$-$	σ	σ	Symmetric to case 2
6	σ	$-$	σ	$*$	Symmetric to case 4
7	$-$	$*$	$-$	$-$	Discussed in details in the text
8	$-$	$-$	$-$	$*$	Symmetric to case 7

rightmost k-mer of S. If $j \neq j'$ then $S_{j,k}$ must be rightmost and if S_j is substituted in N then a similar argument will show a contradiction. Thus in such a case S_j is deleted in N and by Rule 2, each of $S_{j+1}, \ldots, S_{j'-1}$ is deleted in N. By the construction of the alignment there is a $-$ at the corresponding positions in N'. Thus in both cases, $j = j'$ and $j < j'$, we have the pattern $O_p = O'_p = S_p$ and $N_p = N'_p = -$ in the alignment for all $p < j'$.

Without loss of generality, we assume $j + k \leq j' + k'$ as we can always interchange $j + k$ and $j' + k'$ in the following argument. If $j + k < j' + k'$, let p be the leftmost in $[j + k + 1, j' + k' - 1]$ such that S_p is deleted in N'. Then by Rule 2, all of $S_{p+1}, S_{p+2}, \ldots, S_{j'+k'-1}$ must be deleted in N'. If $p > j + k$ or no such p exists then there is atleast one $*$ on the right of the alignment of S_{j+k-1} in N'. By Rule 9, S_{j+k-1} must be substituted in N and hence by Rule 2, S_{j+k-1} must be substituted in N', and so on. This will imply all characters is N are $*$ which is not possible. Thus $p \leq j + k$ and in both cases $j + k = j' + k'$ and $j + k < j' + k'$, we have the pattern $O_p = O'_p = S_p$ and $N_p = N'_p = -$ in the alignment for all $p \geq j + k$.

Now let p be the leftmost position in the alignment where atleast one of the two pairs O_p, O'_p and N_p, N'_p differs. In general, O_p, O'_p are either both equal to $\sigma \in \Sigma$ or both equal to $-$ and $N_p, N'_p \in \Sigma \cup \{*, -\}$. Table III shows the 8 possible restricted values for O_p, N_p, O'_p, N'_p under the assumption of p. Table III further shows why the cases $1, 2, 3, 5, 6, 8$ are not possible. In case 4, to match $N_p = *$ there must be some $p' > p$ such that $N'_{p'} = *$. By Rules 2 and 5, $p' \neq p + 1$. Thus $p' > p + 1$ and $N'_{p+1}, N'_{p+2}, \ldots, N'_{p'-1}$ all must be $-$. By Rule 2, these $-$s can not be due to deletions in $S_{j',k'}$ and hence must be due to insertions in $S_{j,k}$. Since insertions to both $S_{j,k}, S_{j',k'}$ at the same position are aligned together, $N'_{p'} = *$ can not be due to an insertion in $S_{j',k'}$, it must be due a substitution in $S_{j',k'}$. The corresponding character in $S_{j,k}$ must be either deleted or substituted. Both are not possible due to Rules 4 and 6 respectively.

In case 7 too, to match $N_p = *$ there must be some $p' > p$ such that $N'_{p'} = *$. If $p' = p + 1$ then $N'_{p'} = *$ must be due to some substitution in $S_{j',k'}$ and the corresponding character in $S_{j,k}$ must be either deleted or substituted which are not possible by Rules 4 and 6 respectively. Hence $p' > p + 1$ and $N'_{p+1}, N'_{p+2}, \ldots, N'_{p'-1}$ all must be $-$. These $-$s can not be due to deletions in $S_{j',k'}$ because in that case the corresponding characters in $S_{j,k}$ must be either deleted or substituted which are not possible due to Rules 4 and 6 respectively. Thus $N'_{p+1}, N'_{p+2}, \ldots, N'_{p'-1}$ must be due to insertions in $S_{j,k}$. A similar argument as used in case 4 shows a contradiction in this case too. ∎

In general S_js are not distinct. However, as the input strings are random, the duplications due to repeated characters are limited. On instance $(11, 3)$ our algorithm generates each compact motif, on an average, 1.55 times using the rules and 3.63 times without the rules.

Implementation: To track the deleted characters, instead of actually deleting we substitute them by a new symbol $-$ not in Σ'. We populate the motif trie $M^{(i)}$ by calling $genAll(S^{(i)})$ given in Table 2. Rules 1-8 are incorporated in $G(L, j, \delta, \beta, \alpha)$, $H(L, j, \beta, \alpha)$ and $I(L, j, \alpha)$ which are shown in Tables 3, 4, and 5, respectively where $sub(L, j, \sigma)$ substitutes L_j by σ and $ins(L, j, \sigma)$ inserts σ just before L_j.

Algorithm 2: $genAll(S)$

```
foreach $q \leftarrow -d$ to $+d$ do
    $k \leftarrow l + q$;   $start \leftarrow 2$ ;                                   // Rule 1
    $leftMost \leftarrow rightMost \leftarrow$ false;
    for $j \leftarrow 1$ to $|S| - k + 1$ do
        if $j = 1$ then $leftMost \leftarrow$ true;
        if $j+k-1=m$ then $rightMost \leftarrow$ true; $start \leftarrow 1$;
        foreach $\delta \leftarrow \max\{0, q\}$ to $(d + q)/2$ do
            $G(S_{j,k}, start, \delta, d - 2\delta + q, \delta - q)$;
```

Algorithm 3: $G(L, j, \delta, \beta, \alpha)$

```
if $\delta = 0$ then $H(L, j, \beta, \alpha)$; return;
foreach $j' \leftarrow j$ to $|L|$ do
    $G(sub(L, j', -), j' + 1, \delta - 1, \beta, \alpha)$;
```

Algorithm 4: $H(L, j, \beta, \alpha)$

```
if $\beta = 0$ then
    $t \leftarrow \begin{cases} \text{largest } t' & \text{s.t. } L_{j'} = * \text{ for all } j' \leq t' \\ 0 & \text{if no such } j' \text{ exists;} \end{cases}$
    $start \leftarrow \begin{cases} 1 & \text{if } leftMost \\ t + 2 & \text{otherwise;} \end{cases}$          // Rules 7,8
    $I(L, start, \alpha)$; return;
foreach $j' \leftarrow j$ to $|L|$ do
    if $L_j = -$ then continue;                                   // already deleted
    if $(j > 1) \wedge L_{j-1} = -$ then continue;                         // Rule 2
    if $\neg rightMost \wedge_{j''<j'} (L_{j''}=*) \wedge (L_{j'+1}=-)$ then
        continue ;                                                 // Rule 3
    $H(sub(L, j', *), j'+1, \beta-1, \alpha)$;
```

Algorithm 5: $I(L, j, \alpha)$

```
if $\alpha = 0$ then
    insert $L$ to $M^{(i)}$ after deleting all $-$ in $L$; return
foreach $j' \leftarrow j$ to $|L|$ do
    if $L_j \in \{-, *\}$ then continue;                               // Rules 4,6
    if $(j > 1) \wedge (L_{j-1} = -)$ then continue;                       // Rule 5
    $I(ins(L, j', *), j'+1, \alpha-1)$;
if $rightMost \wedge (L_{|L|} \neq -)$ then
    $I(ins(L, |L| + 1, *), |L| + 2, \alpha-1)$;                              // Rule 9
```

III. Results

We implemented our sequential algorithm in C++ and evaluated on a Dell Optiplex 7020 desktop with Intel i5-4590 CPU at 3.30GHz and 8GB RAM running Linux Mint 17. We generated random (l, d) instances according to Pevzner and Sze [2] and as described in the introduction. For every (l, d) combination we report the average runtime over 5 random instances. We compare two different implementations of our algorithm with a modified implementation of the algorithm

TABLE IV: Comparison between EMS1 and two implementations of EMS2 on challenging instances

	Run Time			Memory Usage (MB)		
Instance	EMS1	EMS2	EMS2M	EMS1	EMS2	EMS2M
(8,1)	0.12 s	0.07 s	0.09 s	1	4	3
(9,2)	11.9 s	2.6 s	4.3 s	3	27	17
(12,2)	22.8 s	13.2 s	16.0 s	33	221	127
(11,3)	35.8 m	1.8 m	4.7 m	91	474	312
(13,4)	-	1.2 h	4.4 h	-	7,264	5,618

Time is in seconds (s), minutes (m) or hours (h). An empty cell implies the algorithm did not complete in the stipulated 72 hours.

EMS1 [13] which considered the neighborhood of only l-mers whereas the modified version considers the neighborhood of all k-mers where $l-d \leq k \leq l+d$. The faster implementation of EMS2 stores in each node of the trie an array of pointers to each children of the node. However, this makes the space required to store a tree node dependent on the size of the alphabet Σ. The slower but memory efficient implementation EMS2M keeps two pointers at each node: to the leftmost child and to the immediate right sibling. Access to the other children are simulated using the sibling pointers.

A comparison between the runtime and the memory usage of the three implementations are given in Table IV. Our efficient neighborhood generation enables our algorithm to solve instance $(13, 4)$ in less than two hours which EMS1 could not solve even in 3 days. The factor by which EMS2 takes more memory compared to EMS1 gradually decreases as instances become harder. As EMS2 stores 4 child pointers for A, C, G, T in each node of the motif trie whereas EMS2M simulates access to children using only 2 pointers, EMS2 is faster. Memory reduction in EMS2M is not exactly by a factor $2(=4/2)$ because we also keep a bit vector in each node to represent the subset of $\{A, C, G, T\}$ a child corresponds to. The memory reduction would be significant for protein strings.

IV. Conclusions

We presented an efficient algorithm for the EMS problem. Our algorithm efficiently generates neighborhood using some novel and elegant rules to reduce duplicate motifs in the generated neighborhood. We also proved that these rules are close to ideal as the generated neighborhood is distinct if the characters in the input string are distinct. This condition may not be practical and ideas from [14] can be used when the characters in the input string are repeated. Nevertheless, the rules help because instances are randomly generated and hence the number of times a k-mer appears in any input string is small. The second reason for the efficiency of our algorithms is the use of a trie based data structure to compactly store the motifs. Future work could be to solve harder instances, including those involving protein strings, and possibly using parallel algorithms.

Acknowledgments

This work has been supported in part by the NIH grant R01-LM010101 and NSF grant 1447711.

References

[1] M. Nicolae and S. Rajasekaran, "qPMS9: An Efficient Algorithm for Quorum Planted Motif Search," *Nature Scientific Reports*, vol. 5, 2015.

[2] P. A. Pevzner and S.-H. Sze, "Combinatorial Approaches to Finding Subtle Signals in DNA Sequences," in *ISMB*, vol. 8, 2000, pp. 269–278.

[3] M. Nicolae and S. Rajasekaran, "Efficient Sequential and Parallel Algorithms for Planted Motif Search," *BMC bioinformatics*, vol. 15, no. 1, p. 34, 2014.

[4] S. Tanaka, "Improved Exact Enumerative Algorithms for the Planted (l, d)-motif Search Problem," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 11, no. 2, pp. 361–374, 2014.

[5] 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.

[6] S. Karlin, F. Ost, and B. E. Blaisdell, "Patterns in DNA and Amino Acid Sequences and Their Statistical Significance," in *Mathematical Methods for DNA Sequences*, M. S. Waterman, Ed. CRC Press Inc. Boca Raton, FL, USA, 1989.

[7] E. Rocke and M. Tompa, "An Algorithm for Finding Novel Gapped Motifs in DNA Sequences," in *Proceedings of the Second Annual International Conference on Computational Molecular Biology*. ACM, 1998, pp. 228–233.

[8] M.-F. Sagot, "Spelling Approximate Repeated or Common Motifs using a Suffix Tree," in *LATIN'98: Theoretical Informatics*. Springer, 1998, pp. 374–390.

[9] J. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang, "Distinguishing string selection problems," *Information and Computation*, vol. 185, no. 1, pp. 41–55, 2003.

[10] E. Adebiyi and M. Kaufmann, "Extracting Common Motifs under the Levenshtein Measure: Theory and Experimentation," *Algorithms in Bioinformatics*, pp. 140–156, 2002.

[11] X. Wang and Y. Miao, "GAEM: A Hybrid Algorithm Incorporating GA with EM for Planted Edited Motif Finding Problem," *Current Bioinformatics*, vol. 9, no. 5, pp. 463–469, 2014.

[12] S. Rajasekaran, S. Balla, C. H. Huang, V. Thapar, M. Gryk, M. Maciejewski, and M. Schiller, "High-performance Exact Algorithms for Motif Search," *Journal of Clinical Monitoring and Computing*, vol. 19, no. 4-5, pp. 319–328, 2005.

[13] S. Pathak, S. Rajasekaran, and M. Nicolae, "EMS1: An Elegant Algorithm for Edit Distance Based Motif Search," *International Journal of Foundations of Computer Science*, vol. 24, no. 04, pp. 473–486, 2013.

[14] D. E. Knuth, *The Art of Computer Programming, Volume 4, Generating All Tuples and Permutations, Fascicle 2*. Addison Wesley, 2005.

Appendix

Expected Number of Spurious Motifs

Let M be a motif of length l and L be an occurrence of M of length $l+q$ with δ deletions, β substitutions and α insertions where , $-d \leq q \leq d$. The number of such possible L is

$$N(\delta,\beta,\alpha) = \binom{l+q}{\delta}\binom{l+q-\delta}{\beta}\binom{l+q-\delta+\alpha}{\alpha}|\Sigma|^{\alpha}(|\Sigma|-1)^{\beta}$$

and the probability that a random L of length $l+q$ is a neighbor of M is

$$N(\delta,\beta,\alpha)/|\Sigma|^{l+q}.$$

As discussed in section II-A, for d-neighbors the possible values of δ, β, α are $\max\{0,q\} \leq \delta \leq (d+q)/2, \alpha = \delta - q, \beta = d+q-2\delta$. Thus, for any random L of length $l+q$, the probability that L is an occurrence of M is

$$P = \sum_{\delta=\max\{0,q\}}^{\frac{d+q}{2}} \frac{N(\delta, d+q-2\delta, \delta-q)}{|\Sigma|^{l+q}}.$$

There could be $(m-l-q+1)$ number of $(l+q)$-mers of a string S of length m. The probability that M does not occur in S is

$$R = \Pi_{q=-d}^{d}(1-P)^{m-l-q+1}.$$

The probability that M occurs in each of the input strings in $\mathcal{S} = \{S^{(1)}, S^{(2)}, \ldots, S^{(n)}\}$ is $(1-R)^n$. Since M can be any arbitrary motif, the expected number of common (l, d)-motifs of $\mathcal{S}$ is

$$E(\mathcal{S}, l, d) = |\Sigma|^{l}(1-R)^{n}. \quad (8)$$