The Author(s). *BMC Genomics* 2016, **17**(Suppl 4):465
DOI 10.1186/s12864-016-2789-9

BMC Genomics

RESEARCH Open Access

Efficient sequential and parallel algorithms for finding edit distance based motifs

Soumitra Pal[1], Peng Xiao[1] and Sanguthevar Rajasekaran[2*]

From IEEE International Conference on Bioinformatics and Biomedicine 2015
Washington, DC, USA.9-12 November 2015

Abstract

Background: Motif search is an important step in extracting meaningful patterns from biological data. The general problem of motif search is intractable and there is a pressing need to develop efficient, exact and approximation algorithms to solve this problem. In this paper, we present several novel, exact, sequential and parallel 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 errors of types substitution, insertion and deletion.

Methods: One popular technique to solve the problem is to explore for each input string the set of all possible l-mers that belong to the d-neighborhood of any substring of the input string and output those which are common for all input strings. We introduce a novel and provably efficient neighborhood exploration technique. We show that it is enough to consider the candidates in neighborhood which are at a distance exactly d. We compactly represent these candidate motifs using wildcard characters and efficiently explore them with very few repetitions. Our sequential algorithm uses a trie based data structure to efficiently store and sort the candidate motifs. Our parallel algorithm in a multi-core shared memory setting uses arrays for storing and a novel modification of radix-sort for sorting the candidate motifs.

Results: The algorithms for EMS are customarily evaluated on several challenging instances such as (8,1), (12,2), (16,3), (20,4), and so on. The best previously known algorithm, EMS1, is sequential and in estimated 3 days solves up to instance (16,3). Our sequential algorithms are more than 20 times faster on (16,3). On other hard instances such as (9,2), (11,3), (13,4), our algorithms are much faster. Our parallel algorithm has more than 600 % scaling performance while using 16 threads.

Conclusions: Our algorithms have pushed up the state-of-the-art of EMS solvers and we believe that the techniques introduced in this paper are also applicable to other motif search problems such as Planted Motif Search (PMS) and Simple Motif Search (SMS).

Keywords: Motif, Edit distance, Trie, Radix sort

*Correspondence: rajasek@engr.uconn.edu
[2]Department of Computer Science and Engineering, University of Connecticut, 371 Fairfield Road, 06269 Storrs, CT, USA
Full list of author information is available at the end of the article

Background

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 $S^{(i)}$, $ED\left(S^{(i)}_{j,k}, M\right) \leq d$. 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 $O\left(n^2 m l^d |\Sigma|^d\right)$ time and $O\left(n^2 m/w\right)$ space where w is the word length of the computer. Adebiyi and Kaufmann [10] proposed an exact algorithm with an expected runtime of $O\left(nm + d(nm)^{(1+pow(\epsilon))} \log nm\right)$ 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 1 shows the expected number of spurious motifs for $l \in [5, 21]$ and d upto $\max\{l-2, 13\}$, $n = 20$, $m = 600$ and $\Sigma = \{A, C, G, T\}$ [see Additional file 1]. 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), (11,3), (12,2), (13,4) and (16,3).

The sequential 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.

Contributions

In this paper we present several improved algorithms for EMS that solve instance (11,3) in less than a couple of minutes and instance (13,4) in less than a couple of hours. On (16,3) our algorithm is more than 20 times faster than EMS1. 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 parallel algorithm in the multi-core shared memory setting has more than 600 % scaling performance on 16 threads. Our approach uses following five ideas which can be applied to other motif search problems as well:

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.

Table 1 Expected number of spurious motifs in random instances for n=20, m=600. Here, ∞ represents value ≥1.0e+7

l	d=0	1	2	3	4	5	6	7	8	9	10	11	12	13
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 instances in bold represents challenging instances

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

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.

Trie based data structure for storing compact motifs: 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.

Modified radix-sort for compact motifs: Our parallel algorithm stores the compact motifs in an array and uses a modified radix-sort algorithm to sort them. Use of arrays instead of tries simplifies updating the set of candidate motifs by multiple threads.

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

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

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 $S_{j,k} = S_j S_{j+1} \dots S_{j+k-1}$ 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

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

For a collection of strings $\mathcal{S} = \{S^{(1)}, S^{(2)}, \dots, S^{(m)}\}$, 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

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

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 $G(\{L_1, L_2, \dots, L_r\}) = \cup_{t=1}^{r} G(L_t)$. If $G^1(L) = G(L)$, and for $t > 1$, $G^t(L) = G(G^{t-1}(L))$ then

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

Using Eqs. (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.

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 $0 \le t \le d$, 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

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

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 $ED(B, L') \le ED(B, L) + ED(L, L') \le (d-1) + 1 = d$. 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

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

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 α insertions in the final phase, where $d = \delta + \alpha + \beta$ and $l = k - \delta + \alpha$. Solving for α, β, δ gives $\max\{0, q\} \le \delta \le (d+q)/2, \alpha = \delta - q$ 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.

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):

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

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 subtrie 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'. Figure 1 shows examples of inserting into the motif trie.

We also maintain a motif trie $\mathcal{M}$ 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

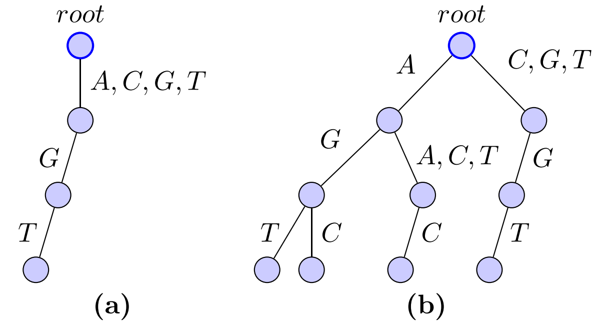


Fig. 1 Inserting into motif trie for $\Sigma = \{A, C, G, T\}$ and $l = 3$. **a** After inserting $*GT$ into empty trie. **b** After inserting another string $A*C$

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. Figure 2 shows an example of the intersection of two motif tries.

```
Algorithm 1: Intersect subtries rooted at $u_1, u_2$
input  : subtrie($u_1$), subtrie($u_2$)
output: subtrie($u_1$) $\leftarrow$ subtrie($u_1$) $\cap$ subtrie($u_2$)
$V \leftarrow$ 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 subtrie rooted at $v_1$;
                $label(u_1, v_1') \leftarrow newLabel$;
                $label(u_1, v_1) \leftarrow label(u_1, v_1) \setminus newLabel$;
                intersect subtries rooted at $v_1', v_2$;
            else
                intersect subtries rooted at $v_1, v_2$;
    if $label(u_1, v_1) = \emptyset$ then delete subtrie rooted at $v_1$
if $u_1$ has no child then delete subtrie 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 $\sigma \in label(u, v)$.

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. Our efficient neighborhood generation technique and the use of compact neighbors reduce duplication in neighborhood but do not guarantee completely duplication free neighborhood. In this section, we design few simple rules to reduce duplication further. Later we will see that these rules are quite close to the ideal as we will prove that the compact motif generated after skipping using the rules, are distinct if all the characters in the input string are distinct.

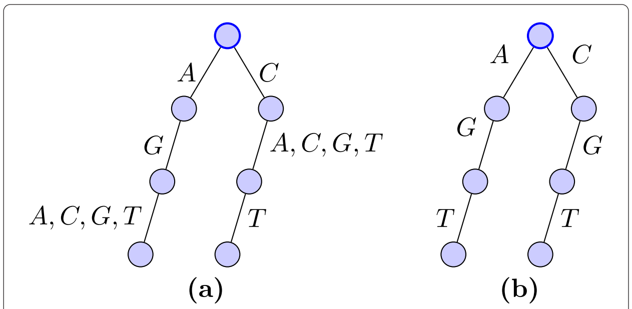


Fig. 2 Intersection of motif tries. **a** Trie for $AG* \cup C*T$. **b** Intersection of trie in Fig. 1b and trie in Fig. 2 **a**

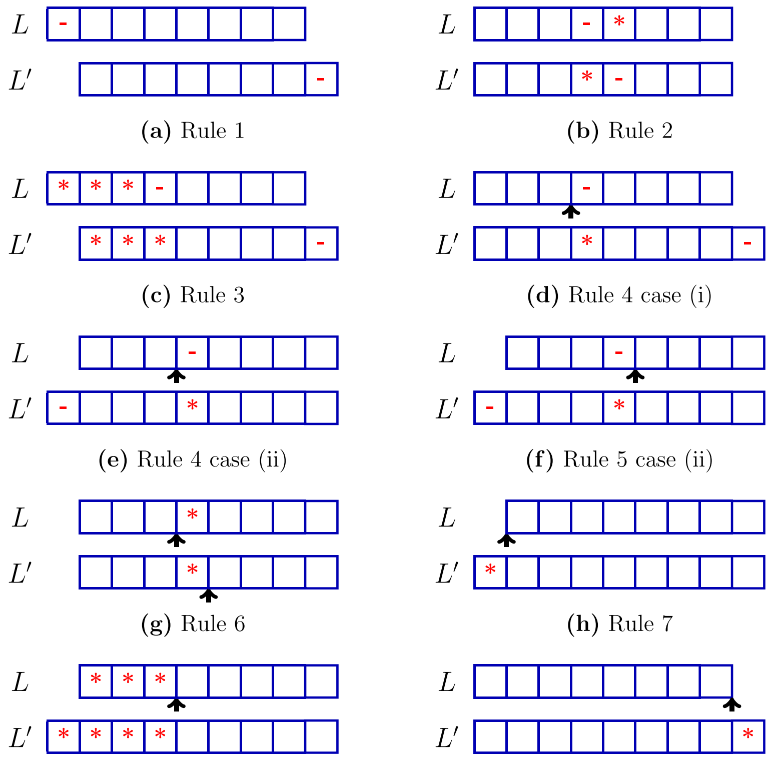
To differentiate multiple copies of the same compact neighbor, we augment it with the information about how it is generated. This information is required only in the proof and is not used in the actual algorithm. Formally, each compact neighbor L of a k-mer $S_{j,k}$ is represented as an ordered tuple $\langle S_{j,k}, T\rangle$ where T denotes the sequence of edit operations applied to $S_{j,k}$. Each edit operation in T is represented as a tuple $\langle p, o\rangle$ where p denotes the position (as in S) where the edit operation is applied and $o \in \{D, R, I\}$ 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 natural order for p and for o, $D < R < I$.

The rules for skipping compact neighbors are given in Table 2. Rule 1 applies when $S_{j,k}$ is not the rightmost k-mer and the current edit operation deletes the leftmost base of $S_{j,k}$, *i.e.*, S_j. Rule 2 applies when the current edit operation substitutes a base just after a base that was already deleted. Rule 3 skips the neighbor which is generated from a k-mer except the rightmost by deleting a base and substituting all bases before it. Rules 4–9 apply when the current operation is an insertion. Rule 4,6 apply when the insertion is just before a deletion and a substitution, respectively. Rule 5 applies when the insertion is just after a deletion. Rule 7,8 apply when the k-mer is not the leftmost. Rule 7 applies when the insertion is at the leftmost position and Rule 8 applies when all bases before the position of insertion are already substituted. Rule 9 applies when the k-mer is not the rightmost and the insertion is at the right end. The first in each pair of the figures in Fig. 3 illustrates the situation where the corresponding rule applies.

Let $\bar{M}_{l,d}(S)$ denote the multi-set of tuples for the compact motifs of S that were not skipped by our algorithm

Table 2 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	$(j + k \leq m) \wedge \langle j{+}k, I\rangle \in T$



(a) Rule 1 (b) Rule 2 (c) Rule 3 (d) Rule 4 case (i) (e) Rule 4 case (ii) (f) Rule 5 case (ii) (g) Rule 6 (h) Rule 7 (i) Rule 8 (j) Rule 9

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)

using the rules in Table 2 and $M_{l,d}(S)$ be the set of compact motifs generated by (3). Let $\Gamma(\langle S_{j,k}, T\rangle)$ be the resulting string when the operations in T are applied to $S_{j,k}$ 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 $M_{l,d}(S) \subseteq \Gamma(\bar{M}_{l,d}(S))$ by giving a contradiction when $M_{l,d}(S) \setminus \Gamma(\bar{M}_{l,d}(S)) \neq \emptyset$.

We define an order on the compact neighbors $L_1 = \langle S_{j_1,k_1}, T_1\rangle$ and $L_2 = \langle S_{j_2,k_2}, T_2\rangle$ as follows: $L_1 < L_2$ if $\Gamma(L_1) < \Gamma(L_2)$ and $L_2 < L_1$ if $\Gamma(L_2) < \Gamma(L_1)$. When $\Gamma(L_1) = \Gamma(L_2)$ we have $L_1 < L_2$ if and only if $(k_1 < k_2) \vee ((k_1 = k_2) \wedge (p_1 < p_2)) \vee ((k_1 = k_2) \wedge (p_1 = p_2) \wedge (o_1 < o_2))$ where $\langle p_1, o_1\rangle \in T_1, \langle p_2, o_2\rangle \in T_2$ are the leftmost edit operations where T_1, T_2 differ.

Suppose $M \in M_{l,d}(S) \setminus \Gamma(\bar{M}_{l,d}(S))$. Let $L = \langle S_{j,k}, T\rangle$ be the largest (in the order defined above) tuple skipped by our algorithm such that $\Gamma(L) = M$. For each $r = 1, \ldots, 9$ we show a contradiction that if L is skipped by Rule r then there is another $L' = \langle S_{j',k'}, T'\rangle$ with the same number of edit operations and $\Gamma(L') = M$ but $L < L'$. Figure 3 illustrates the choice of L' under different rules.

Rule 1. Here $j + k \leq m$ and $\langle j, D\rangle \in T$. Consider $T' = \left(T \setminus \left\{\langle j, D\rangle\right\}\right) \cup \left\{j + k, D\right\}$, 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. $T' = T\setminus\{\langle j + t, D\rangle, \langle j + t, I\rangle\}\cup\{\langle j - 1, D\rangle, \langle j+t, R\rangle\}$, $j' = j - 1, k' = k + 1$.

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

Consider two compact motifs $M_1 = \langle S_{j_1,k_1}, T_1\rangle$ and $M_2 = \langle S_{j_2,k_2}, T_2\rangle$ in $\bar{M}_{l,d}(S)$. For $q \in \{1, 2\}$, let

$\left\langle p_q^{(1)}, o_q^{(1)} \right\rangle, \left\langle p_q^{(2)}, o_q^{(2)} \right\rangle, \ldots, \left\langle p_q^{(d)}, o_q^{(d)} \right\rangle$ be the sequence of edit operations in T_q arranged in the order as the neighbors are generated by our algorithm, and the intermediate neighbors be $L_q^{(h)} = \left\langle S_{j_q,k_q}, \left\{ \left\langle p_q^{(1)}, o_q^{(1)} \right\rangle, \left\langle p_q^{(2)}, o_q^{(2)} \right\rangle, \ldots, \left\langle p_q^{(h)}, o_q^{(h)} \right\rangle \right\} \right\rangle$ for all $h = 1, 2, \ldots, d$. We also denote the initial k-mer as a neighbor $L_q^{(0)} = \langle S_{j_q,k_q}, \emptyset \rangle$.

Lemma 3. *If S_js are all distinct and $\Gamma\left(L_1^{(h)}\right) = \Gamma\left(L_2^{(h)}\right)$ for $1 \leq h \leq d$ then $\left\langle p_1^{(h)}, o_1^{(h)} \right\rangle = \left\langle p_2^{(h)}, o_2^{(h)} \right\rangle$ and $\Gamma\left(L_1^{(h-1)}\right) = \Gamma\left(L_2^{(h-1)}\right)$.*

Proof. To simplify the proof, we use p_q, o_q, L_q to denote $p_q^{(h)}, o_q^{(h)}, L_q^{(h)}$, respectively, for all $q \in \{1, 2\}$. Without loss of generality we assume $p_1 \leq p_2$.

As p_1, p_2 are positions in S, it would be enough to prove $\langle p_1, o_1 \rangle = \langle p_2, o_2 \rangle$ because that would imply $\Gamma\left(L_1^{(h-1)}\right) = \Gamma\left(L_2^{(h-1)}\right)$.

If $\langle p_1, o_1 \rangle \neq \langle p_2, o_2 \rangle$ then either (a) $o_1 = o_2$ and $p_1 < p_2$ or (b) $o_1 \neq o_2$ and $p_1 \leq p_2$, giving us the following 9 possible cases. We complete the proof by giving a contradiction in each of these 9 cases:

Case	o_1	o_2	**cond.**	**Case**	o_1	o_2	**cond.**	**Case**	o_1	o_2	**cond.**
1	D	D	$p_1{<}p_2$	4	R	D	$p_1{\leq}p_2$	7	I	D	$p_1{\leq}p_2$
2	D	R	$p_1{\leq}p_2$	5	R	R	$p_1{<}p_2$	8	I	R	$p_1{\leq}p_2$
3	D	I	$p_1{\leq}p_2$	6	R	I	$p_1{\leq}p_2$	9	I	I	$p_1{<}p_2$

Cases 2, 3, 4, 7

Our algorithm applies edit operations in phases: first deletions, followed by substitutions and finally insertions. In all these cases, one of $\Gamma(L_1), \Gamma(L_2)$ does not have any $*$ because only deletions have been applied so far and the other has at least one $*$ because a substitution or an insertion has been applied. This implies $\Gamma(L_1) \neq \Gamma(L_2)$, a contradiction.

Case 1

L_2 has S_{p_2} deleted. As $\Gamma(L_1) = \Gamma(L_2)$, S_{p_2} must have been deleted in some operation prior to reaching L_1. As the deletions are applied in order, left to right, we must have $p_1 = p_2$ which is a contradiction.

Case 5

This case has been illustrated in Fig. 4a. L_1 has no substitution at a position $> p_1$ and no insertion at all. The $*$ at p_2 in L_2 must be matched with the $*$ at p_1 in L_1 and as the characters in S are distinct, all of $S_{p_1+1}, \ldots, S_{p_2}$ cannot appear in L_1 and hence must be deleted in L_1.

Now for each $t < p_1$, right to left, and $y = t + p_2 - p_1$, we have the following: S_y is either deleted or substituted in L_1, which implies that S_y must be substituted in L_2 as the deletion of S_y in L_2 is prohibited by Rule 2, and finally to match this $*$ in L_2, S_t must be substituted in L_1 as S_t cannot be deleted in L_1, again by Rule 2.

But this makes Rule 3 applicable to L_1 and L_1 must have been skipped. This is a contradiction.

Case 6

By Rule 9 the insertion in L_2 cannot be at the rightmost position and hence L_2 must have at least one character after the insertion. By Rules 4 and 6, S_{p_2} must not be deleted or substituted in L_2 and hence it must not be deleted or substituted in L_1 either. Thus $p_1 < p_2$. There cannot be any insertion or substitution at a position $> p_1$ in L_1. Thus the $*$ due to the insertion at p_2 in L_2 must be matched by the $*$ due to the substitution at p_1 in L_1 and all of $S_{p_1+1}, \ldots, S_{p_2-1}$ must be deleted in L_1.

By Rule 7, S_{p_2} cannot be the leftmost in S_{j_2,k_2}. So we consider S_{p_2-1} in L_1, L_2. It is either deleted or substituted in L_1 and hence by Rule 5, it must be substituted in S_{p_2} (there can be multiple insertions at p_2 in L_2 but that does not affect this argument). To match this $*$, S_{p_1-1} must be substituted in L_1.

Using a similar argument as in Case 5, S_t must be substituted in L_1 for each $t < p_1 - 1$. But this again makes Rule 3 applicable to L_1 and L_1 must have been skipped, which is not possible. This case has been illustrated in Fig. 4b.

Case 8

Due to Rules 4, 6 and 9, S_{p_1} must not be deleted or substituted in L_1 and hence it must not be deleted or substituted in L_2 either. Thus $p_1 < p_2$. The $*$ due to the insertion in L_1 must be matched by a substitution at $p_3 < p_1$ such that all of $S_{p_3+1}, \ldots, S_{p_1-1}$ are deleted in L_2.

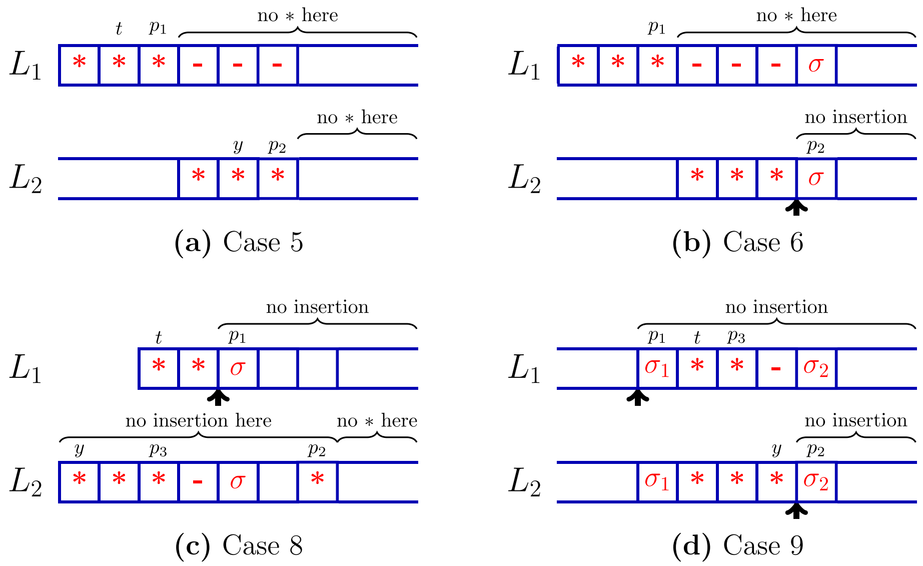
By Rule 7, p_1 cannot be the leftmost in L_1. For each $t < p_1$, right to left, and $y = t + p_3 - p_1$, we have the following: S_y is substituted in L_1 because as the deletion of S_y in L_1 is prohibited by Rules 2 and 5, S_y must be substituted in L_2 again by Rule 2, and to match this $*$, S_t must be substituted in L_1.

But this makes Rule 8 applicable to L_1 and L_1 must have been skipped which is not possible. This case has been illustrated in Fig. 4c.

Case 9

This case has been illustrated in Fig. 4d. Due to Rules 4, 6 and 9, S_{p_1}, S_{p_2} must not be deleted or substituted in L_1, L_2. The insertion at p_2 in L_2 must be matched by a substitution at a position p_3 in L_1 such that $p_1 < p_3 < p_2$ and all of $S_{p_3+1}, \ldots, S_{p_2-1}$ must be deleted in L_1.

Now for each position y, from right to left, where $p_1 < y < p_2$, S_y is either deleted or substituted in S_1, S_y cannot be deleted in L_2 by Rules 2 and 5 and hence must be substituted in L_2, which again must be matched by a substitution



(a) Case 5 (b) Case 6

(c) Case 8 (d) Case 9

Fig. 4 Proof of uniqueness (Lemma 2). Subfigures a,b,c,d illustrates the cases 5,6,7,8,9 respectively

at a position t in L_1 such that $p_1 < t < p_3$. However this is impossible as the number of possible ys is larger than the number of possible ts. □

If all S_js are distinct and $\Gamma(M_1) = \Gamma(M_2)$ then applying Lemma 3 repeatedly for $h = d, d-1, \ldots, 0$ gives us the fact that starting k-mers S_{j_1,k_1}, S_{j_2,k_2} as well as the corresponding edit operations in T_1, T_2 for M_1, M_2 must be the same. This is another way of stating the following theorem.

Theorem 1. *If S_js are all distinct then $\bar{M}_{l,d}(S)$ is duplication free.*

In general S_js are not distinct. However, as the input strings are random, the duplication 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 compared to 3.63 times without the rules (see Fig. 5).

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 Algorithm 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 Algorithms 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
            $G(S_{j,k}, start, \delta, d - 2\delta + q, \delta - q)$;
```

Modified radix-sort for compact motifs

A simpler data structure alternative to tries for storing compact motifs could be an array. However, it becomes difficult to compute the intersection in (3) as defined in (7) when the compact motifs are stored in arrays. One straight-forward solution is to first expand the $*$s in the compact motifs, then sort the expanded motifs and finally compute the intersection by scanning through the two sorted arrays. This, to a great extent, wipes out the advantage using the $*$s in the compact motifs. However, we salvage execution time by executing a modified radix-sort that simultaneously expands and sorts the array of compact motifs: Compact-Radix-Sort(A, l) where the first parameter A represents the array of compact motifs and the second parameter represents the number of digits of the elements in A which is equal to the number of base positions l in a motif.

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
    $G(sub(L, j', -), j' + 1, \delta - 1, \beta, \alpha)$;
```

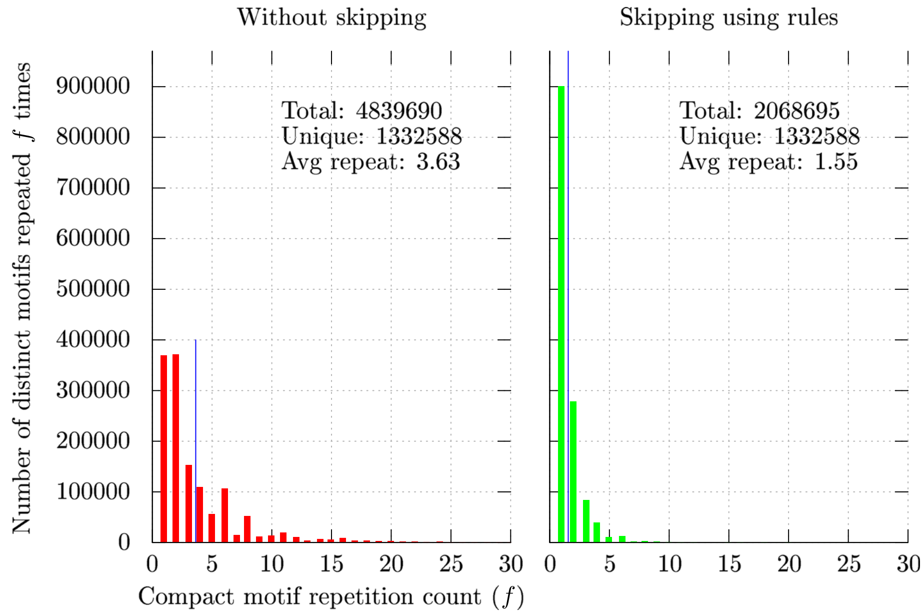


Fig. 5 Histogram of number of times a motif is repeated with and without using the skipping rules 1–9

```
Algorithm 4: $H(L, j, \beta, \alpha)$
if $\beta = 0$ then
    $t \leftarrow \begin{cases} \text{largest } t' & \text{s.t. } L_{j'} = * \text{ for all} j' \le t' \\ 0 & \text{if no such } j' \text{ exists;} \end{cases}$
    $start \leftarrow \begin{cases} 1 & \text{if } leftMost \\ t+2 & \text{otherwise;} \end{cases}$   // Rules 7,8
    $I(L, start, \alpha)$; return;
foreach $j' \leftarrow j$ to $|L|$ do
    if $L_j = -$ then continue;                        // 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
    $H(sub(L, j', *), j'+1, \beta-1, \alpha)$;
```

As in the standard radix-sort, our algorithm uses l phases, one for each base position in the motif. In the ith phase it sorts the motifs using bucket sort on the ith base of the motifs. However, in case of compact motifs, for each $*$ at a base position, the bucket counters for all $\sigma \in \Sigma$ are incremented. While reordering the motifs as per the bucket counts, if there is a $*$ at ith base position of a motif, $|\Sigma|$ copies of the motif are created and they are placed at appropriate locations in the array after finalizing the correct σ for the $*$. The details are given in Algorithm 6. In each phase a bucket counter B and a cumulative counter C are used. The temporary array T stores the partially expanded motifs from the current phase.

Discussion We did an experiment to compare the time taken by the two approaches – (i) using the expanded motifs, *i.e.*, without using the wildcard character, and (ii) using compact motifs and sorting them using Compact-Radix-Sort. For a single input string of instance (16,3), the first approach generated in 24.4 s 198,991,822 expanded motifs in which 53,965,581 are unique. The second approach generated in 13.7 s 11,474,938 compact motifs with the same number of unique expanded motifs. This shows the effectiveness of the second approach.

Parallel algorithm

We now give our parallel algorithm in the multi-core shared memory setting. To process each input sequence $S^{(i)}$ the algorithm uses $p + 1$ threads. The main thread first prepares the workload for other p threads. A workload involves the generation of the neighborhood for a k-mer of $S^{(i)}$, where $l - d \le k \le l + d$. There are total $\sum_{k=l-d}^{l+d}(m - k + 1) = (2d + 1)(m - l + 1)$ workloads.

```
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;       // Rule 5
    continue $I(ins(L, j', *), j'+1, \alpha-1)$;
if $rightMost \wedge (L_{|L|} \neq -)$ then
    $I(ins(L, |L| + 1, *), |L| + 2, \alpha-1)$;   // Rule 9
```

Algorithm 6: Compact-Radix-Sort(A, l)

```
for $i \leftarrow 1$ to $l$ do
    foreach $\sigma \in \Sigma$ do $B[\sigma] \leftarrow 0$;
    for $j \leftarrow 1$ to $|A|$ do
        $\sigma \leftarrow$ digit $i$ of $A[j]$;
        if $\sigma = *$ then
            foreach $\sigma' \in \Sigma$ do $B[\sigma'] \leftarrow B[\sigma'] + 1$;
        else $C[\sigma] \leftarrow B[\sigma] + 1$;
    foreach $\sigma \in \Sigma$ do $C[\sigma] \leftarrow \sum_{\sigma' \le \sigma} B[\sigma']$;
    $T \leftarrow$ empty array of size $\max_\sigma C[\sigma]$;
    for $j \leftarrow 1$ to $|A|$ do
        $\sigma \leftarrow$ digit $i$ of $A[j]$;
        if $\sigma = *$ then
            foreach $\sigma' \in \Sigma$ do
                $T[C[\sigma']] \leftarrow A[j]$ with $\sigma'$ at digit $i$;
                $C[\sigma'] \leftarrow C[\sigma'] - 1$;
        else
            $T[C[\sigma]] \leftarrow A[j]$;
            $C[\sigma] \leftarrow C[\sigma] - 1$;
    $A \leftarrow T$;
```

The number of neighbors generated in the workloads are not the same due to the skipping of some neighbors using rules 1–9. For load balancing, we randomly and evenly distribute workloads to threads. Each thread first generates all the compact motifs in its workloads and then sort them using Compact-Radix-Sort. If $i > 2$ then it removes all neighbors not present in $M^{(i-1)}$ which is the set of common motifs of $S^{(1)}, S^{(2)}, \ldots, S^{(i-1)}$. The master thread then merges the residue candidate motifs from all the p threads to compute $M^{(i)}$. The merging takes place in $\log_2 p$ phases in a binary tree fashion where the jth phase uses $2^{\log_2 p - j}$ threads each merging two sorted arrays of motifs.

Results and discussion

We implemented our algorithms in C++ and evaluated on a Dell Precisions Workstation T7910 running RHEL 7.0 on two sockets each containing 8 Dual Intel Xeon Processors E5-2667 (8C HT, 20 MB Cache, 3.2 GHz) and 256 GB RAM. For our experiments we used only one of the two sockets. We generated random (l, d) instances according to Pevzner and Sze [2] and as described in the background section. For every (l, d) combination we report the average time taken by 4 runs. We compare the following four implementations:

- **EMS1:** A modified implementation of the algorithm in [13] which considered the neighborhood of only l-mers whereas the modified version considers the neighborhood of all k-mers where $l - d \le k \le l + d$.
- **EMS2:** A faster implementation of our sequential algorithm which uses tries for storing candidate motifs where each node of the trie stores 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.
- **EMS2M:** A slightly slower but memory efficient implementation of our sequential algorithm where each node of the trie keeps two pointers: one to the leftmost child and the other to the immediate right sibling. Access to the other children are simulated using the sibling pointers.

Table 3 Comparison between EMS1 and three implementations of EMS2

Instance	Metric	EMS1	EMS2	EMS2M	EMS2P threads				
					1	2	4	8	16
(8,1)	time	0.11 s	0.13 s	0.12 s	0.09 s	0.08 s	0.06 s	0.05 s	0.06 s
	memory	2.69 MB	4.25 MB	3.17 MB	2.67 MB	3.20 MB	3.55 MB	6.02 MB	7.99 MB
(12,2)	time	19.87 s	15.60 s	16.62 s	2.71 s	1.94 s	1.44 s	0.89 s	0.55 s
	memory	34.28 MB	210.47 MB	126.91 MB	84.98 MB	104.60 MB	125.18 MB	142.82 MB	150.23 MB
(16,3)	time	1.74 h	23.73 m	26.79 m	3.73 m	2.32 m	1.38 m	48.58 s	36.93 s
	memory	8.39 GB	11.62 GB	6.97 GB	8.55 GB	10.21 GB	10.53 GB	9.84 GB	9.91 GB
(9,2)	time	10.84 s	1.72 s	3.02 s	1.12 s	0.96 s	0.78 s	0.49 s	0.35 s
	memory	3.44 MB	26.67 MB	17.04 MB	42.86 MB	57.76 MB	54.77 MB	59.85 MB	66.53 MB
(11,3)	time	33.48 m	1.91 m	3.57 m	45.85 s	30.78 s	19.68 s	13.49 s	9.78 s
	memory	92.86 MB	477.12 MB	313.33 MB	2.27 GB	2.63 GB	2.65 GB	2.55 GB	2.60 GB
(13,4)	time	-	1.08 h	1.76 h	44.03 m	26.16 m	14.51 m	8.62 m	6.82 m
	memory	-	8.26 GB	5.58 GB	149.60 GB	179.66 GB	180.13 GB	168.80 GB	172.74 GB

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

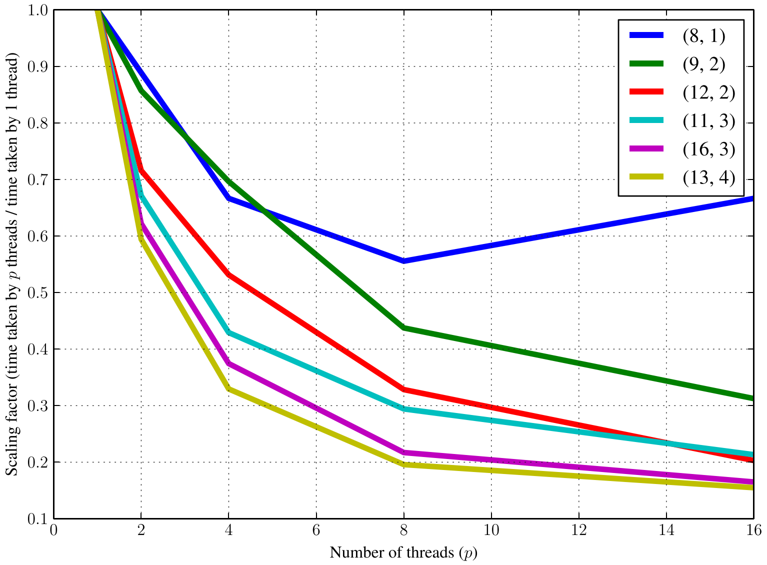


Fig. 6 Scaling performance of our parallel algorithm

- **EMS2P:** Our parallel algorithm which uses arrays for storing motifs. We experimented with $p = 1, 2, 4, 8, 16$ threads.

We run the four algorithms on the challenging instances (8,1), (12,2), (16,3) and on the instances (9,2), (11,3), (13,4) which are challenging for PMS and have been used for experimentation in [13]. We report the runtime and the memory usage of the four algorithms in Table 3.

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.

Our parallel algorithm EMS2P using one thread is significantly faster than the sequential algorithms EMS2 and EMS2M but uses more memory. This space-time trade off is due to the fact that the arrays are faster to access but the tries use lesser memory. Moreover, the repeated motifs are uniquely stored in a single leaf node in the trie but stored separately in the array. The scaling performance using multiple threads are shown in Fig. 6 where we plot the ratio of time taken by p threads to the time taken by a single thread on the Y-axis. The time required for handling 16 threads turns out to be costlier than actually processing the motifs in the smallest instance (8,1). We observe speed up consistent across other bigger instances. For example, instance (16,3) takes about 224 s using 1 thread and 37 s using 16 threads. This gives more than 600 % scaling performance using 16 threads.

Conclusions

We presented several efficient sequential and parallel algorithms for the EMS problem. Our algorithms use some novel and elegant rules to explore the candidate motifs in such a way that only a small fraction of the candidate motifs are explored twice or more. In fact, we also proved that these rules are close to ideal in the sense that no candidate motif is explored twice if the characters in the input string are all 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 the instances are randomly generated and the k-mers in the input string are not much frequent. The second reason for the efficiency of our sequential algorithms is the use of a trie based data structure to compactly store the motifs. Our parallel algorithm stores candidate motifs in an array and uses a modified radix-sort based method for filtering out invalid candidate motifs.

Our algorithms pushed up the state-of-the-art of EMS solvers to a state where the challenging instance (16,3) is solved in slightly more than half a minute using 16 threads. Future work could be to solve harder instances, including those involving protein strings, and possibly using many-core distributed algorithms.

Additional file

Additional file 1: Expected number of spurious motifs. This file gives the expression for the expected number of spurious (l, d)-motifs in n random strings of length m from the alphabet Σ. (PDF 143 kb)

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

Declarations
Publication of this article was funded by the NIH grant R01-LM010101 and NSF grant 1447711. This article has been published as part of BMC Genomics Vol 17 Suppl 4 2016: Selected articles from the IEEE International Conference on Bioinformatics and Biomedicine 2015: genomics. The full contents of the supplement are available online at https://github.com/soumitrakp/ems2.git.

Availability
A C++ based implementation of our algorithm can be found at the following github public repository:
https://github.com/soumitrakp/ems2.git.

Authors' contributions
SP and SR conceived the study. SP implemented the algorithms and PX carried out the experiments. SP and SR analyzed the results and wrote the paper. All authors reviewed the manuscript. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Department of Computer Science and Engineering, University of Connecticut, 371 Fairfield Road, 06269 Storrs, CT, USA. [2]Department of Computer Science and Engineering, University of Connecticut, 371 Fairfield Road, 06269 Storrs, CT, USA .

Published: 18 August 2016

References

1. Nicolae M, Rajasekaran S. qPMS9: An Efficient Algorithm for Quorum Planted Motif Search. Nat Sci Rep. 2015;5. doi:10.1038/srep07813.
2. Floratou A, Tata S, Patel JM. Efficient and Accurate Discovery of Patterns in Sequence Data Sets. IEEE Trans Knowl Data Eng. 2011;23(8):1154–68. http://doi.ieeecomputersociety.org/10.1109/TKDE.2011.69.
3. Nicolae M, Rajasekaran S. Efficient Sequential and Parallel Algorithms for Planted Motif Search. BMC Bioinformatics. 2014;15(1):34.
4. Tanaka S. Improved Exact Enumerative Algorithms for the Planted $(\boldsymbol{l}, \boldsymbol{d})$-motif Search Problem. IEEE/ACM Trans Comput Biol Bioinformatics (TCBB). 2014;11(2):361–74.
5. Yu Q, Huo H, Zhang Y, Guo H. PairMotif: A new pattern-driven algorithm for planted $(\boldsymbol{l}, \boldsymbol{d})$ DNA motif search. PloS One. 2012;7(10):48442.
6. Karlin S, Ost F, Blaisdell BE. Patterns in DNA and Amino Acid Sequences and Their Statistical Significance. In: Waterman MS, editor. Mathematical Methods for DNA Sequences. Boca Raton, FL, USA: CRC Press Inc; 1989.
7. Rocke E, Tompa M. An Algorithm for Finding Novel Gapped Motifs in DNA Sequences. In: Proceedings of the Second Annual International Conference on Computational Molecular Biology. New York, NY, USA: ACM; 1998. p. 228–33.
8. Sagot MF. Spelling Approximate Repeated or Common Motifs using a Suffix Tree. In: LATIN'98: Theoretical Informatics. Brazil: Springer; 1998. p. 374–90.
9. Lanctot JK, Li M, Ma B, Wang S, Zhang L. Distinguishing string selection problems. Inform Comput. 2003;185(1):41–55.
10. Adebiyi EF, Kaufmann M. Extracting Common Motifs under the Levenshtein Measure: Theory and Experimentation. In: Guigó R, Gusfield D, editors. Algorithms in Bioinformatics: Second International Workshop, WABI 2002 Rome, Italy, September 17–21, 2002 Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg; 2002. p. 140–56.
11. Wang X, Miao Y. GAEM: A Hybrid Algorithm Incorporating GA with EM for Planted Edited Motif Finding Problem. Curr Bioinformatics. 2014;9(5): 463–9.
12. Rajasekaran S, Balla S, Huang CH, Thapar V, Gryk M, Maciejewski M, Schiller M. High-performance Exact Algorithms for Motif Search. J Clin Monitoring Comput. 2005;19(4–5):319–28.
13. Pathak S, Rajasekaran S, Nicolae M. EMS1: An Elegant Algorithm for Edit Distance Based Motif Search. Int J Foundations Comput Sci. 2013;24(04): 473–86.
14. Knuth DE. The Art of Computer Programming, Volume 4, Generating All Tuples and Permutations, Fascicle 2. New Jersey, USA: Addison Wesley; 2005.