PMS6: A Fast Algorithm for Motif Discovery*

Shibdas Bandyopadhyay
Department of CISE
University of Florida,
Gainesville, FL 32611
shibdas@ufl.edu

Sartaj Sahni
Department of CISE
University of Florida,
Gainesville, FL 32611
sahni@cise.ufl.edu

Sanguthevar Rajasekaran
Department of CSE
University of Connecticut,
Storrs, CT 06269, USA
rajasek@engr.uconn.edu

***Abstract*—We propose a new algorithm, PMS6, for the (l, d)-motif discovery problem in which we are to find all strings of length l that appear in every string of a given set of strings with at most d mismatches. The run time ratio PMS5/PMS6, where PMS5 is the fastest previously known algorithm for motif discovery in large instances, ranges from a high of 2.20 for the (21,8) challenge instances to a low of 1.69 for the (17,6) challenge instances. Both PMS5 and PMS6 require some amount of preprocessing. The preprocessing time for PMS6 is 34 times faster than that for PMS5 for (23,9) instances. When preprocessing time is factored in, the run time ratio PMS5/PMS6 is as high as 2.75 for (13,4) instances and as low as 1.95 for (17,6) instances.**

***Index Terms*—Planted motif search, string algorithms.**

I. Introduction

Motifs are approximate patterns found in promoter sequences. The discovery of motifs helps in finding *transcription factor-binding sites*, which are useful for understanding various gene functions, drug design, and so on. Many versions of the motif search problem have been studied extensively. In this paper, we focus on *Planted Motif Search* (PMS), also known as (l, d) motif search. PMS takes n strings and two numbers l and d as input. The problem is to find all strings M of length l (also called an l-mer) that appear in every input sequence[1] with at most d mismatches. More precisely, for an l-mer M, we define its d-neighborhood to be all l-mers that differ from M in at most d positions. M is a motif for the given set of n strings iff each of these n strings has a substring that is in the d-neighborhood of M.

As an example, consider three input sequences CATACGT, ACAAGTC and AATCGTG. Suppose that $l = 3$ and $d = 1$. The 3-mer CAT is one of the motifs of the given 3 input sequences as CAT appears in first sequence at the first position with 0 mismatch, at the second position in the second sequence with 1 mismatch, and at fourth position in third sequence with 1 mismatch. Alternatively, CAT is a motif of the given 3 input sequences because the substrings CAT of sequence 1, CAA of sequence 2, and CGT of sequence 3 are in the 1-neighborhood of CAT.

PMS, which is well studied in the literature, is known to be NP-hard [9]. Known algorithms for PMS are divided into exact and approximation algorithms depending on whether they are guaranteed to produce every motif always or not. Approximation algorithms for PMS are generally faster than exact algorithms, which have an exponential worst-case complexity. MEME, which is one of the popular approximation algorithms for finding motifs [1], uses the expectation minimization technique to output a set of probabilistic models for each motif indicating the probability of appearance of different characters in each position of the motif. Pevzner and Sze [17] proposed the WINNOWER algorithm, which maps PMS to a problem of finding large cliques in a graph where each node is an l-mer and two nodes are connected iff the number of mismatches between them is less than $2d$. Buhler and Tompa [2] used random projections by taking only k positions out of the entire l-mer to group l-mers together based on the similarity of the projections. Groups that have a large number of l-mers have a high probability of having the desired motif as well. Price et al. [18] proposed an algorithm based on performing a local search on the d-neighborhood of some of the l-mers from input sequences. GibbsDNA [15] employs Gibbs sampling while CONSENSUS [11] uses statistical measures to align sequences and finds potential motifs from the alignment. Two other examples of approximation algorithms for the motif search problem are MULTIPROFILER [13] and ProfileBranching [18].

While exact algorithms for the motif problem take longer to complete than approximation algorithms, they guarantee to find all motifs. Due to their exponential complexity it is impractical to run these algorithms on very large instances. But, for many instances of practical interest, they are able to run within a reasonable amount of time. Many of these algorithms use the suffix tree or other tree data structures to progressively generate motifs one character at a time. MITRA [8] uses a data structure called Mismatch Tree while SPELLER [22], SMILE [16], RISO [3], and RISOTTO [19] use suffix trees. RISOTTO [19] is the fastest suffix tree based algorithm so far. CENSUS [10] makes a trie of all l-mers from each of the input sequences. The nodes in the trie then store the Hamming distance (i.e., number of mismatches) from the motif as it is being generated, potentially pruning many branches of the trie. Voting [4] uses hashing but needs space to store all possible strings of length l. Hence the space required by this method is too large for large instances. Kauksa and Pavlovic [14] have proposed an algorithm to generate a

* This research was supported, in part, by the National Science Foundation under grant 0829916 and the National Institutes of Health under grant R01-LM010101.

[1]We use the terms sequence and string interchangeably in this paper

superset of motifs (i.e., a set of motif stems). Since they do not provide any data on how difficult it may be to extract the true motifs from this superset, one cannot assess the value of this algorithm. The recently proposed PMS series of algorithms are both fast and relatively economical on space. PMS1, PMS2 and PMS3 [20] are based on radix sorting and then efficiently intersecting the d-neighborhood of all l-mers in the input sequences (every length l substring of a string is an l-mer of that string). PMS4 [21] proposes a generic speed-up technique to improve the run time of any exact algorithm. PMSP [5] computes the d-neighborhood of all l-mers of the first input sequence and then performs an exhaustive search with the remaining input sequences to determine which of the l-mers in the computed d-neighborhood are motifs. PMSPrune [5] is a branch-and-bound algorithm, which uses dynamic programming to improve upon PMSP. Pampa [6] further improves PMSPrune by using wildcard characters to find approximate motif patterns and then performing an exhaustive search within possible mappings of the pattern to find the actual motifs. PMS5 [7] is the most recent algorithm in the series. This algorithm, which is described in detail in Section II, efficiently computes the intersection of the d-neighborhood of l-mers without generating the entire neighborhood. PMS5 is faster than the algorithm for Pampa for challenge instances (15, 5) and larger [7].

In this paper, we propose a motif algorithm PMS6 that is faster than PMS5. It's relative speed comes from a faster algorithm for d-neighborhood generation and intersection. In Section II we introduce notations and definitions used throughout the paper and also describe the PMS5 algorithm in detail and in Section III we describe our proposed algorithm for d-neighborhood intersection. The performance of PMS5 and PMS6 is compared experimentally in Section IV.

II. PMS5

A. Notations and Definitions

We use the same notations and definitions as in [7]. An l-mer is simply any string of length l. r is an l-mer of s iff (a) r is an l-mer and (b) r is a substring of s. The notation $r \in_l s$ denotes an l-ber r of s. The *Hamming distance*, $d_H(s,t)$, between two equal length strings s and t is the number of places where they differ and the *d-neighborhood*, $B_d(s)$, of a string s, is $\{x|d_H(x,s) \le d\}$. Let $N(l,d) = |B_d(s)|$. It is easy to see that $N(l,d) = \sum_{i=0}^{d} \binom{l}{i}(|\Sigma|-1)^i$, where Σ is the alphabet in use.

We note that x is an (l,d) motif of a set S of strings if and only if (a) $|x| = l$ and (b) every $s \in S$ has an l-mer whose Hamming distance from x is at most d. The set of (l,d) motifs of S is denoted $M_{l,d}(S)$

B. PMS5–Overview

PMS5, which is presently the fastest exact algorithm to compute $M_{l.d}(S)$ for large (l,d), was proposed by Dinh, Rajasekaran, and Kundeti [7]. This algorithm (Figure 1) first computes a superset, Q', of the motifs of S by making a series of calls to a function, $B_d(x,y,z) = B_d(x) \cap B_d(y) \cap B_d(z)$, that computes the intersection of the d-neighborhoods of 3 l-mers. This superset is then pruned to $M_{l,d}(S)$ by the function $outputMotifs$, which examines the l-mers in Q' one by one determining which of those are valid motifs. This determination is done in a brute force manner. The correctness of PMS5 is established in [7]. Its time complexity is $O(nm^3lN(l,d))$, where m is the length of each input string s_i [7].

```
PMS5($S, l, d$)
 for each $x \in_l s_1$
 {
    for $k = 1$ to $\lfloor \frac{n-1}{2} \rfloor$
    {
       $Q \leftarrow \emptyset$
       for each $y \in_l s_{2k}$ and $z \in_l s_{2k+1}$
          $Q \leftarrow Q \cup B_d(x,y,z)$
       if $k = 1$ $Q' = Q$
       else $Q' = Q' \cap Q$
       if $|Q'| < threshold$ break;
    }
    $outputMotifs(Q', S, l, d)$;
 }
```

Fig. 1: PMS5 [7]

C. Computing $B_d(x,y,z)$

The basic idea in the algorithm of [7] to compute $B_d(x,y,z)$ is to generate $B_d(x)$ one l-mer at a time and include it in $B_d(x,y,z)$ only if it is in $B_d(y) \cap B_d(z)$. To facilitate this, $B_d(x)$ is represented as a tree $T_d(x)$. The root, which is at level 0, of $T_d(x)$ is x. The nodes at the next level represent l-mers that are at a Hamming distance of 1 from x. Hence, if the length of x is m and the alphabet is $\Sigma = \{0,1\}$, the root will have m children and the ith child will represent the l-mer x' that differs from x only at position i. That is, $x'[i] = 0$ if $x[i] = 1$ and $x'[i] = 1$ if $x[i] = 0$. Figure 2 shows the tree $T_2(1001)$ with $\Sigma = \{0,1\}$.

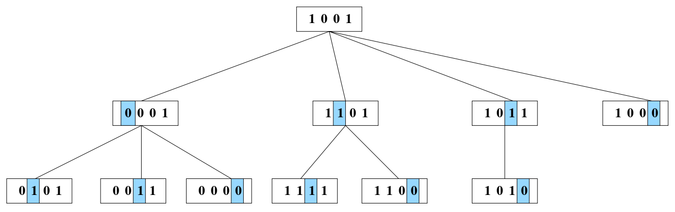

Fig. 2: 2-neighborhood tree

The tree $T_d(x)$ is created dynamically in depth first manner. An l-mer t in a node (t,p) (t is the l-mer represented by the node and p is the position at which this l-mer differs from the l-mer in the parent node) is added to $B_d(x,y,z)$ if t is in $B_d(y) \cap B_d(z)$. As we know the current distance of t from each of x, y and z, we can check if there is possibly an l-mer in the as yet ungenerated subtree of (t,p) that is a distance $\le d$ from each of x, y and z. The subtree rooted at (t,p) is pruned if there is no such l-mer.

Let $t_1 = t[1:p]$, $t_2 = t[p+1:[l]$, $x_1 = x[1:p]$ and $x_2 = x[p+1:[l]$; y_1, y_2, z_1, z_2 are defined similarly. Each position i of x_2, y_2 and z_2 is one of the following five types [7]:

Type 1: $x_2[i] = y_2[i] = z_2[i]$.
Type 2: $x_2[i] = y_2[i] \neq z_2[i]$.
Type 3: $x_2[i] = z_2[i] \neq y_2[i]$.
Type 4: $x_2[i] \neq y_2[i] = z_2[i]$.
Type 5: $x_2[i] \neq y_2[i], x_2[i] \neq z_2[i], y_2[i] \neq z_2[i]$.

Let n_i denote the number of positions of Type i, $1 \leq i \leq 5$. Each n_i may be decomposed as below [7]:

1) $N_{1,a}$ = number of Type 1 positions i such that $w[i] = x_2[i]$.
2) $N_{2,a}(N_{2,b})$ = number of Type 2 positions i such that $w[i] = x_2[i](w[i] = z_2[i])$.
3) $N_{3,a}(N_{3,b})$ = number of Type 3 positions i such that $w[i] = x_2[i](w[i] = y_2[i])$.
4) $N_{4,a}(N_{4,b})$ = number of Type 4 positions i such that $w[i] = y_2[i](w[i] = x_2[i])$.
5) $N_{5,a}(N_{5,b}$, $N_{5,c})$ = number of Type 5 positions i 5 such that $w[i] = x_2[i])(w[i] = y_2[i]$,$w[i] = z_2[i])$.

Dinh, Rajasekaran, and Kundeti [7] have shown that when traversing $T_d(x)$ in depth-first fashion, we may prune the tree at every node (t,p) for which the following ILP has no solution (note that $d_H(x_1,t_1)$, $d_H(y_1,t_1)$, $d_H(z_1,t_1)$ and $n_1 \cdots n_5$ are readily determined for x, y, z, t, and p).

1) $n_1 - N_{1,a} + n_2 - N_{2,a} + n_3 - N_{3,a} + n_4 - N_{4,b} + n_5 - N_{5,a} \leq d - d_H(x_1,t_1)$
2) $n_1 - N_{1,a} + n_2 - N_{2,a} + n_3 - N_{3,b} + n_4 - N_{4,a} + n_5 - N_{5,b} \leq d - d_H(y_1,t_1)$
3) $n_1 - N_{1,a} + n_2 - N_{2,b} + n_3 - N_{3,a} + n_4 - N_{4,a} + n_5 - N_{5,c} \leq d - d_H(z_1,t_1)$
4) $N_{1,a} \leq n_1$
5) $N_{2,a} + N_{2,b} \leq n_2$
6) $N_{3,a} + N_{3,b} \leq n_3$
7) $N_{4,a} + N_{4,b} \leq n_4$
8) $N_{5,a} + N_{5,b} + N_{5,c} \leq n_5$
9) All variables are non-negative integers.

As the possible values for $n_1, \cdots, n_5$ and $d_H(x_1,t_1), d_H(y_1,t_1), d_H(z_1,t_1)$ are $[0, \cdots, l]$ and $[0, \cdots, d]$, respectively, only $(l+1)^5(d+1)^3$ distinct ILPs are possible. For a given pair (l,d), these distinct ILPs may be solved in a preprocessing step and we may store, in an 8-dimensional table, whether each has a solution. Once this preprocessing has been done, we can use the results stored in the 8-dimensional table to find motifs for many (l,d) instances.

Figure 3 gives the pseudocode for the algorithm to compute $B_d(x,y,z)$. Its time complexity is $O(l + d|B_d(x,y,z)|)$ [7].

D. Intersection of Qs

PMS5 uses several novel techniques to compute a superset of the intersection Q' of the Qs. Although a superset of Q' (Figure 1) is computed, PMS5 determines the exact set of motifs because the last loop of the algorithm (Figure 1) verifies that each member of the final Q' is, in fact, a motif. One of the novel techniques applied for challenge instances of size (19,7) and larger is a Bloom filter [12] with 2 hash functions. The l-mer to be hashed uses $\lceil l/4 \rceil$ bytes (recall that the alphabet size is 4). The first hash function is bytes 0-3 of the l-mer and the second is bytes 1-4.

```
Bd($x,y,z$)
 Compute $d_H(x,y)$ and $d_H(x,z)$.
 Determine $n_1,n_2,n_3,n_4,n_5$ for each $p=0,\cdots,(l-1)$.
 Do a depth-first traversal of $T_d(x)$. At each node
 $(t,p)$ do the following:
   Incrementally compute $d_H(x,t), d_H(y,t)$ and
   $d_H(z,t)$
   Incrementally compute $d_H(x_1,t_1), d_H(y_1,t_1)$ and
   $d_H(z_1,t_1)$ from its parent.
   If $d_H(x,t) \leq d$, $d_H(y,t) \leq d$, and $d_H(z,t) \leq d$
   $B_d(x,y,z) \leftarrow B_d(x,y,z) \cup t$.
   Look-up ILP table with parameters $n_1,n_2,n_3,$
   $n_4,n_5,d_H(x,t),d_H(y,t),d_H(z,t)$.
   If $d_H(x,t) \leq d$ and the ILP has a solution,
    explore the children of $(t,p)$; otherwise prune at
 $(t,p)$.
```

Fig. 3: Computing $B_d(x,y,z)$ [7]

III. PMS6

A. Overview

PMS6 differs from PMS5 only in the way it determines the motifs corresponding to an l-mer x of s_1 and the strings s_{2k} and s_{2k+1}. Recall that PMS5 does this by computing $B_d(x,y,z)$ independently for every pair (y,z) such that $y \in_l s_{2k}$ and $z \in_l s_{2k+1}$ (Figure 1); the computation of $B_d(x,y,z)$ is done by performing a depth-first search of the tree $T_d(x)$ using an ILP to prune subtrees. In PMS6 we determine the motifs corresponding to an l-mer x of s_1 and the strings s_{2k} and s_{2k+1} using the following 2-step process:

Step 1: *Form Equivalence Classes.* In this step, the triples (x,y,z) of l-mers such that $y \in_l s_{2k}$ and $z \in_l s_{2k+1}$ are partitioned into classes $C(n_1, \cdots, n_5)$. For this partitioning, for each triple (x,y,z), we compute n_1, $\cdots$, n_5 using the definitions of Section II-C and $p = 0$, $x_1 = y_1 = z_1 = \epsilon$, $x_2 = x$, $y_2 = y$, and $z_2 = z$. Each triple is placed in the class corresponding to its computed $n_1, \cdots, n_5$ values.

Step 2: *Compute B_d for all triples by classes.* For each class $C(n_1, \cdots, n_5)$, the union, $B_d(C)$, of $B_d(x,y,z)$ for all triples in that class is computed. We note that the union of all $B_d(C)$s is the set of all motifs of x, s_{2k}, and s_{2k+1}.

Figure 4 gives the pseudocode for PMS6.

B. Computing $B_d(C(n_1, \cdots, n_5))$

Let (x,y,z) be a triple in $C(n_1, \cdots, n_5)$ and let w be an l-mer in $B_d(x,y,z)$. Let $p = 0$ and let $N_{1,a}, N_{2,a}, N_{2,b}, N_{3,a}, N_{3,b}, N_{4,a}, N_{4,b}, N_{5,a}, N_{5,b}, N_{5,c}$ be as

```
PMS6($S, l, d$)
 for each $x \in_l s_1$
 {
   for $k = 1$ to $k = \lfloor \frac{n-1}{2} \rfloor$
   {
     $Q \leftarrow \emptyset$
     $Classes \leftarrow \emptyset$
     for each $y \in_l s_{2k}$ and $z \in_l s_{2k+1}$
     {
       Compute $n_1, \cdots, n_5$ for $(x, y, z)$.
       if $C(n_1, \cdots, n_5) \notin Classes$
       {
         Create the class $C(n_1, \cdots, n_5)$ with $(x, y, z)$.
         Add $C(n_1, \cdots, n_5)$ to $Classes$.
       }
       else add $(x, y, z)$ to class $C(n_1, \cdots, n_5)$.
     }
     for each class $C(n_1, \cdots, n_5)$ in $Classes$
       $Q \leftarrow Q \cup B_d(C(n_1, \cdots, n_5))$
     if $k = 1$ then $Q' = Q$
     else $Q' = Q' \cap Q$.
     if $|Q'| < threshold$ break;
   }
   $outputMotifs(Q', S, l, d)$;
 }
```

Fig. 4: PMS6

in Section II-C. We observe that the 10-tuple $(N_{1,a}, \cdots, N_{5,c})$ satisfies the ILP of Section II-C with $d_H(x_1, t_1) = d_H(y_1, t_1) = d_H(z_1, t_1) = 0$. In fact, every l-mer of $B_d(x, y, z)$ has a 10-tuple $(N_{1,a}, \cdots, N_{5,c})$ that is a solution to this ILP with $d_H(x_1, t_1) = d_H(y_1, t_1) = d_H(z_1, t_1) = 0$. Given a 10-tuple solution to the ILP, we may generate all l-mers w in $B_d(x, y, z)$ as follows:

1) Each of the l positions in w is classified as being of Type 1, 2, 3, 4, or 5 depending on the classification of the corresponding position in the l-mers x, y, and z (see Section II-C).
2) Select $N_{1,a}$ of the n_1 Type 1 positions of w. If i is a selected position, then, from the definition of a Type 1 position, it follows that $x[i] = y[i] = z[i]$. Also from the definition of $N_{1,a}$, this many Type 1 positions have the same character in w as in x, y, and z. So, for each selected Type 1 position i, we set $w[i] = x[i]$. The remaining Type 1 positions of w must have a character different from $x[i]$ (and hence for $y[i]$ and $z[i]$). So, for a 4-character alphabet there are 3 choices for each of the non-selected Type 1 positions of w. As, there are $\binom{n}{N_{1,a}}$ ways to select $N_{1,a}$ positions out of n_1 positions, we have $3^q \binom{n_1}{N_{1,a}}$ different ways to populate the n_1 Type 1 positions of w, where $q = n_1 - N_{1,a}$.
3) Select $N_{2,a}$ positions I and $N_{2,b}$ different positions J from the n_2 Type 2 positions of w. For each $i \in I$, set $w[i] = x[i]$ and for each $j \in J$, set $w[j] = z[i]$. Each of the remaining $n_2 - N_{1,a} - N_{1,b}$ Type 2 positions of w is set to a character different from that in x, y, and z. So, if k is one of these remaining Type 2 positions, $x[k] = y[k] \neq z[k]$. We set $w[k]$ to one of the 2 characters of our 4-letter alphabet that are different from $x[k]$ and $z[k]$. Hence, we have $2^r \binom{n_2}{N_{2,a}} \binom{n_2 - N_{2,a}}{N_{2,b}}$ ways to populate the n_2 Type 2 positions in w, where $r = n_2 - N_{2,a} - N_{2,b}$.
4) Type 3 and Type 4 positions are populated using a strategy similar to that used for Type 2 positions. The number of ways to populate Type 3 positions is $2^s \binom{n_3}{N_{3,a}} \binom{n_3 - N_{3,a}}{N_{3,b}}$, where $s = n_3 - N_{3,a} - N_{3,b}$ and that for Type 4 positions is $2^u \binom{n_4}{N_{4,a}} \binom{n_4 - N_{4,a}}{N_{4,b}}$, where $u = n_4 - N_{4,a} - N_{4,b}$.
5) To populate the Type 5 Positions of w, we must select the $N_{5,a}$ Type 5 positions, k, that will be set to $x[k]$, the $N_{5,b}$ Type 5 positions, k, that will be set to $y[k]$, and the $N_{5,c}$ Type 5 positions, k, that will be set to $z[k]$. The remaining $n_5 - N_{5,a} - N_{5,b} - N_{5,c}$ Type 2 positions, k, of w are set to the single character of the 4-letter alphabet that differs from $x[k]$, $y[k]$, and $z[k]$. We see that the number of ways to populate the n_5 Type 5 positions is $\binom{n_5}{N_{5,a}} \binom{n_5 - N_{5,a}}{N_{5,b}} \binom{n_5 - N_{5,a} - N_{5,b}}{N_{5,c}}$.

The preceding strategy to generate $B_d(x, y, z)$ generates $3^q 2^r 2^s 2^u \binom{n_1}{N_{1,a}} \binom{n_2}{N_{2,a}} \binom{n_2 - N_{2,a}}{N_{2,b}} \binom{n_3}{N_{3,a}} \binom{n_3 - N_{3,a}}{N_{3,b}} \binom{n_4}{N_{4,a}} \binom{n_4 - N_{4,a}}{N_{4,b}} \binom{n_5}{N_{5,a}} \binom{n_5 - N_{5,a}}{N_{5,b}} \binom{n_5 - N_{5,a} - N_{5,b}}{N_{5,c}}$ l-mers w for each 10-tuple $(N_{1,a}, \cdots, N_{5,c})$. While every generated l-mer is in $B_d(x, y, z)$, some l-mers may be the same. Computational efficiency is obtained by computing $B_d(x, y, z)$ for all (x, y, z) in the same class $C(n_1, \cdots, n_5)$ concurrently by sharing the loop overheads as the same loops are needed for all (x, y, z) in a class.

Figure 5 gives the pseudocode for our algorithm to compute $B_d(x, y, z)$ by classes.

```
ClassBd($C(n_1, n_2, n_3, n_4, n_5)$)
 $B_d \leftarrow \emptyset$
 Find all ILP solutions with parameters
$n_1, n_2, n_3, n_4, n_5$
 for each solution $(N_{1,a}, \cdots, N_{5,c})$
 {
   $curComb$ ← first combination for this solution
   for $i$ = 0 to (# combinations)
   {
     for each triplet $(x, y, z)$ in $C(n_1, \cdots, n_5)$
     {
       Generate $w$s for $curComb$.
       Add these $w$s to $B_d$.
     }
     $CurComb$ ← next combination in Gray code
order.
   }
 }
 return $B_d$
```

Fig. 5: Compute $B_d(n_1, \cdots, n_5)$

As in the case of PMS5, run-time may be reduced by precomputing data that do not depend on the string set S. So, for a given pair (l, d), there are $O((l+1)^5)$ 5-tuples $(n_1, \cdots, n_5)$. For each of the 5-tuples, we can precompute all 10-tuples $(N_{1,a}, \cdots, N_{5,c})$ that are solutions to the ILP of Section II-C with $d_H(x_1, t_1) = d_H(y_1, t_1) = d_H(z_1, t_1) = 0$. The 10-tuple solutions of the ILP are found using an exhaustive search. For each 10-tuple, we can precompute all combinations (i.e., selections of positions in w). The precomputed 10-tuple solutions for each 5-tuple are stored in a table with $(l+1)^5$ entries and indexed by $[n_1, \cdots, n_5]$ and the precomputed combinations for the 10-tuple solutions are stored in a separate table. By storing the combinations in a separate table, we can ensure that each is stored only once even though the same combination may be needed by many 10-tuple solutions.

We store precomputed combinations as vectors. For example, a Type 1 combination for $n_1 = 3$ and $N_{1,a} = 1$ could be stored as $\{010\}$ indicating that the first and third Type 1 positions of w have a character different from what x, y, and z have in that position while the character in the second Type 1 position is the same as in the corresponding position of x, y, and z. A Type 2 combination for $n_2 = 4, N_{2,a} = 2$ and $N_{2,b} = 1$ could be stored as $\{3011\}$ indicating that the character in the first Type 2 position of w comes from the third l-mer, z, of the triplet, the second type 2 position of w has a character that is different from any of the characters in the same position of x and z and the third and fourth Type 2 positions of w have the same character as in the corresponding positions of x. Combinations for the remaining position types are stored similarly. As indicated by our pseudocode of Figure 5, combinations are considered in Gray code order so that only two positions in the l-mer being generated change from the previously generated l-mer. Consequently, we need less space to store the combinations in the combination table and less time to generate the new l-mer. An example of a sequence of combinations in Gray code order for Type 2 positions with $n_2 = 4, N_{2,a} = 1, N_{2,b} = 1$ is {0012, 0021,0120,0102,0201,0210,1200,1002,1020, 2010, 2001, 2100}. Note that in going from one combination to the next only two positions are swapped.

C. Intersection of Qs

For challenge instances (19,7) and larger, we experimented with several Bloom filter designs. As in [7], we used a partitioned Bloom filter of total size 1GB. From Bloom filter theory [12] we can determine the number of hash functions to use to minimize filter error. However, we need to minimize run time rather than filter error. Experimentally, we determined that best performance was achieved using two hash functions with the first one being bytes 0-3 of the key (i.e., the same function used in PMS5) and the second being the product of bytes 0-3 and the remaining bytes (byte 4 for (19,7) instances and bytes 4 and 5 for (21,8) and (23,9) instances).

D. Complexity

The asymptotic time complexity of PMS56 is the same as that of PMS5, $O(nm^3 lN(l, d))$, where m is the length of each input string $s[j]$.

IV. Experimental Results

We evaluate the performance of PMS6 on challenge instances described in [7]. For each (l, d) that characterizes a challenge instance, we generated 20 random strings of length 600 each. Next, a random motif of length l was generated and planted at random positions in each of the 20 strings. The planted motif was then randomly mutated in exactly d randomly chosen positions. For each (l, d) value up to (19,7), we generated 20 instances and for larger (l, d) values, we generated 5 instances. The average run times for each (l, d) value are reported in this section. Since the variation in run times across instances was rather small, we do not report the standard deviation. Even though we test our algorithm using only synthetic data sets, several authors (e.g., [7]) have shown that PMS codes that work well on the kind of synthetic data used by us also work well on real data. As PMS5 is the fastest algorithm [7] for large-instance motif search, we compare the run times of PMS6 with PMS5. For PMS5, we used C++ code provided by the authors of [7]. PMS6 was coded by us in C++.

The PMS5 and PMS6 codes were compiled using the -03 flag in gcc under 64-bit Linux and run on a Intel Core i7 system running at 3.3 GHz. Our experiments were limited to challenge instances (l, d) [7]. For each challenge instance multiple datasets of 20 randomly generated strings each of length of 600 characters were generated. For each (l, d), the average time is reported here. The standard deviation in the run times is very small.

A. Preprocessing

The preprocessing times of PMS5 and PMS6 for all challenge instances are given in Figure 6. The space required to save the preprocessed data is given in Figure 7. The time taken by PMS5 to build its ILP tables for $l = 23$ and $d = 9$ is 883 seconds while PMS6 takes 25.5 seconds to build its ILP tables. For the (23,9) case, PMS5 uses roughly 300MB to store its ILP results while PMS6 uses roughly 350 MB to store its ILP solutions and combinations. Although, for large instances, PMS6 needs more space for its ILP tables than does PMS5, the space required by other components of the PMS5 and PMS6 algorithms dominates. For example, the Bloom filter used by both algorithms occupies 1GB and to solve (23,9) instances, we need approximately 0.5GB for Q. At present, run time, not memory, limits our ability to solve larger challenge instances than (23,9) using either PMS5 or PMS6.

Algorithm	(13,4)	(15, 5)	(17, 6)	(19, 7)	(21,8)	(23,9)
PMS5	24s	55s	119s	245s	478s	883s
PMS6	0.9s	1s	2s	3.5s	9.5s	25.5s
PMS5/PMS6	26.67	55	59.5	70	50.31	34.63

Fig. 6: Preprocessing times for PMS5 and PMS6

Algorithm	(13,4)	(15, 5)	(17, 6)	(19, 7)	(21,8)	(23,9)
PMS5	5MB	15MB	35MB	50MB	155MB	300MB
PMS6	1MB	4MB	15MB	45MB	145MB	450MB
PMS5/PMS6	5	3.75	2.33	1.11	1.07	0.67

Fig. 7: Total storage for PMS5 and PMS6

B. Computing B_d

Next, we compare the time taken to compute all $B_d(x, y, z)$ in PMS5 and time taken to compute all $B_d(C(n_1, \cdots, n_5)$ in PMS6 for different challenge instances. The times are given in the Figure 8. Note that since PMS5 and PMS6 use different Bloom filters to intersect the Qs, the number of iterations needed for Q' to reach the threshold size (i.e., the number of iterations of the `for` k loop (Figure 1)) may be different in PMS5 and PMS6. If PMS6 is run for the same number of iterations as used by PMS5, the PMS6 times to compute the B_ds goes up to 8.89m for the (19,7) instances, to 1.29h for the (21,8) instances, and to 10.83h for the (23,9) instances. The PMS5/PMS6 ratios become 2.5, 2.26 and 1.82, respectively, for these instances.

Algorithm	(13,4)	(15, 5)	(17, 6)	(19, 7)	(21,8)	(23,9)
PMS5	12.03s	67.64s	6.86m	22.21m	2.91h	19.73h
PMS6	4.20s	21.67s	2.31m	7.75m	1.09h	8.84h
PMS5/PMS6	2.86	3.12	2.96	2.87	2.67	2.23

Fig. 8: Time taken to compute $B_d(C(n_1, \cdots, n_5))$ and $B_d(x, y, z)$

C. Total Time

The total time (i.e., time to compute the motifs) taken by PMS5 and PMS6 for different challenge instances is shown in Figure 9. The run time ratio PMS5/PMS6 ranges from a high of 2.20 for the (21,8) case to a low of 1.69 for the (17,6) case. When preprocessing time is factored in, the run time ratio PMS5/PMS6 varies from a high of 2.75 for the (13,4) case to a low of 1.95 for the (17,6) case.

Algorithm	(13,4)	(15, 5)	(17, 6)	(19, 7)	(21,8)	(23,9)
PMS5	39s	130s	11.35m	40.38m	4.96h	40.99h
PMS6	22s	75s	6.72m	22.75m	2.25h	19.19h
PMS5/PMS6	1.77	1.73	1.69	1.77	2.20	2.14

Fig. 9: Total run time of PMS5 and PMS6

V. Conclusion

We have developed a new algorithm, PMS6, for the motif discovery problem. The run time ratio PMS5/PMS6 ranges from a high of 2.20 for the (21,8) challenge instances to a low of 1.69 for the (17,6) challenge instances. Both PMS5 and PMS6 require some amount of preprocessing. The preprocessing time for PMS6 is 34 times faster than that for PMS5 for $(23, 9)$ instances. When preprocessing time is factored in, the run time ratio PMS5/PMS6 is as high as 2.75 for (13,4) instances and as low as 1.95 for (17,6) instances.

Acknowledgement

We are grateful to Mr. H. Dinh of the University of Connecticut for providing us the PMS5 code and for assisting us with several aspects of this code.

References

[1] Bailey, Timothy, L., Williams, N., Misleh, C. and Wilfred, W. Li, MEME:discovering and analyzing DNA and protein sequence motifs, *Nucleic Acids Research*,34,369-373, 2006.

[2] Buhler, J. and Tompa, M., Finding motifs using random projections, *Fifth Annual International Conference on Computational Molecular Biology(RECOMB)*, 2001.

[3] Carvalho, A.M., Freitas, A.T. Oliveira, A.L and Sagot M.F., A highly scalable algorithm for the extraction of cis-regulatory regions,*Third Asia Pacic Bioinformatics Conference (APBC)*, 2005.

[4] Chin, F Y.L. and Leung H C.M., Voting Algorithms for Discovering Long Motifs, *Proceedings of the Third Asia-Pacic Bioinformatics Conference (APBC)*, 261-271,2005.

[5] Davila, J., Balla, S. and Rajasekaran. S., Fast and practical algorithms for planted (l, d) motif search, *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2007.

[6] Davila, J., Balla, S and Rajasekaran, S., Pampa: An Improved Branch and Bound Algorithm for Planted (l, d) Motif Search, Tech Report, University of Connecticut, 2007.

[7] Dinh, H., Rajasekaran, S. and Kundeti, V., PMS5: an efficient exact algorithm for the (l,d) motif finding problem, *BMC Bioinformatics*, 12:410, 2011.

[8] Eskin, E. and Pevzner, P.,Finding composite regulatory patterns in DNA sequences, *Bioinformatics*, 2002, 354-363.

[9] Evans, P.A., Smith, A., and Todd Warenham, H., On the complexity of finding common approximate substrings,*Theoretical Computer Science*,306, 407-430, 2003.

[10] Evans, P.A and Smith, A., Toward Optimal Motif Enumeration, *Proceedings of Algo- rithms and Data Structures, 8th International Workshop (WADS)*, 47-58, 2003.

[11] Hertz, G. and Stormo, G., Identifying DNA and protein patterns with statistically significant alignments of multiple sequences, *Bioinformatics*,15,563-577, 1999.

[12] Horowitz, E., Sahni, S., and Mehta, D., *Fundamentals of Data Structures in C++*, 2ed, Silicon Press, 2006.

[13] Keich, U. and Pevzner, P., Finding motifs in the twilight zone, *Bioinformatics*, 18,13741381,2002.

[14] Kuksa, P.P. and Pavlovic, V., Efficient motif finding algorithms for large-alphabet inputs, *BMC Bioinformatics*, 11(Suppl 8):S1, 2010.

[15] Lawrence, C.E., Altschul, S.F., Boguski, M.S., Liu, J.S., Neuwald, A.F. and Wootton, J.C., Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment, *Science*, 262,208-214,2003.

[16] Marsan, L. and Sagot M.F., Extracting structured motifs using a suffix tree. - Algorithms and application to promoter consensus identification.*Fourth Annual International Conference on Computational Molecular Biology(RECOMB)*, 2000.

[17] Pevzner, P. and Sze, S.-H., Combinatorial approaches to finding subtle signals in DNA sequences, *Proc. English International Conference on Intelligent Systems for Molecular Biology*, 8, 269-278, 2000.

[18] Price, A., Ramabhadran, S. and Pevzner, P., Finding subtle motifs by branching from the sample strings. *Bioinformatics supplementary edition. Proceedings of the Second European Conference on Computational Biology (ECCB)*, 2003.

[19] Pisanti, N., Carvalho, A.M., Marsan, L and Sagot, M.F., RISOTTO: Fast extraction of motifs with mismatches, *Proceedings of the 7th Latin American Theoretical Informatics Symposium (LATIN)*, 3887, 757-768, 2006.

[20] Rajasekaran, S., Balla, S. and Huang, C.H., Exact algorithms for planted motif challenge problems, *Journal of Computational Biology*, 12(8),1117-1128, 2005.

[21] Rajasekaran S., Dinh, H., A speedup technique for (l, d) motif nding algorithms, *BMC Research Notes*, 4:54,1-7, 2011.

[22] Sagot, M.F. Spelling approximate repeated or common motifs using a suffix tree. *Proceedings of the Third Latin American Theoretical Informatics Symposium (LATIN)*, 111-127, 1998.