An Efficient Exact Solution for the (l, d)-Planted Motif Problem

Maria Clara Isabel D. Sia
Ateneo de Manila University
Loyola Heights, Quezon City
Email: aia.sia1995@gmail.com

Julieta Q. Nabos
Ateneo de Manila University
Loyola Heights, Quezon City
Email: julietnabos@yahoo.com

Mark Joseph D. Ronquillo
Ateneo de Manila University
Loyola Heights, Quezon City
Email: markronquillo23@gmail.com

Proceso L. Fernandez
Ateneo de Manila University
Loyola Heights, Quezon City
Email: pfernandez@ateneo.edu

***Abstract*—DNA motif finding is widely recognized as a difficult problem in computational biology and computer science. Because of the usual large search space involved, exact solutions typically require a significant amount of execution time before discovering a motif of length l that occurs in an input set $\{S_1, ..., S_n\}$ of sequences, allowing for at most d substitutions. In this paper, we propose EMS-GT, a novel algorithm that operates on a compact bit-based representation of the search space and takes advantage of distance-related patterns in this representation in order to compute the exact solution for any arbitrary problem instance up to l=17. A C++ implementation is shown to be highly competitive against PMS8 and qPMS9, two current state-of-the-art exact motif search algorithms. EMS-GT works extremely well for problems involving short motifs, outperforming both competitors for challenge instances with (l, d) values (9,2), (11,3), (13,4) and (15,5), showing runtime reductions of at least 95.1%, 89.6%, 80.8% and 48.4% respectively for these instances, while ranking second to qPMS9 for the challenge instance (17,6).**

***Index Terms*—planted (l, d)-motif problem, bit-based, exact enumerative algorithm**

I. Introduction

DNA motif finding is widely recognized as a difficult problem in computational biology and computer science. Motifs are sequences that occur repeatedly in DNA and have some biological significance [1]. DNA motifs are short and usually fixed length patterns [2] which range from 5 to 30 base pairs (bp) long, typically 10 bp in eukaryotes and 16 bp in prokaryotes [3]. Identifying motif in a set of DNA sequences has many applications, such as finding transcription site, gene regulation, understanding human disease and genetic drug target identification. There are many variants of motif finding problem in literature. Some look for a motif that is repeated in a single sequence. Others look for a motif that occurs over some or all of a set of DNA sequences [4]. One of the latter type is the planted motif problem.

We define the planted motif problem formally as follows:

> Given a set $\mathcal{S} = \{S_1, ..., S_n\}$ of n DNA sequences over the alphabet $\Sigma = \{a, c, g, t\}$ of length L, find M, the set of sequences(or motifs) of length $l < L$, each of which has at least one d-neighbor in each sequence in S.

An example instance of this problem is given below.

> *Find an unknown motif of length l=8 across 5 DNA sequences, each containing the motif with at most d=2 mismatches.*
>
> atcactcgttctcctctaatgtgtaaagacgtactaccgaccttaacgccgaccggtccgatccttgtatagctcctaacgggcatcagctcctgactgcatcgcgatctcggtagtttcctgttcatcatttggccctcagcatcgtgcgtcctgctaacacattcccatgcagcttgaaaagaatttacggtaaaggatccacatccaatcgtgtgaaag
>
> *Motif:* ccatcgtt

Initially it seems an exhaustive string search will suffice for this problem. However, due to biological mutation, motif occurrences in DNA are allowed to differ from the original motif by up to d characters. This greatly impacts complexity; brute-force solutions quickly become infeasible as values of l and d increase. In fact, the problem has already been shown to be NP-complete [5].

Some useful definitions for discussing exact motif-search algorithms are presented first:

- An ***l*-mer** is a sequence of length l. Given a sequence S of length $L > l$, the i^{th} l-mer in S starts at the i^{th} position. Ex. if l=5, the second l-mer in `gattaca` is `attac`.
- The **Hamming distance d_H** between two l-mers of equal length is the number of mismatch characters between them. Ex. $d_H(\texttt{gattaca}, \texttt{cgttaga}) = 3$.
- A ***d*-neighbor** of an ***l*-mer** x is an l-mer whose Hamming

distance from x is at most d.

- The ***d*-neighborhood of an *l*-mer x** is the set $N(x, d)$ consisting of all the d-neighbors of x.
  Ex. gatct, cctta, and aatta are all in $N(\texttt{gatta}, 2)$.
- The ***d*-neighborhood of a sequence S** is the set $\mathcal{N}(S, d)$ consisting of all l-mers that belong to the d-neighborhood of at least 1 l-mer $x \in S$. Ex. for $l = 5$
  $\mathcal{N}(\texttt{gattaca}, 2) = N(\texttt{gatta}, 2) \cup N(\texttt{attac}, 2) \cup N(\texttt{ttaca}, 2)$

Two distinct d-neighbors of a motif might differ in as many as $2d$ characters. This shows why (l, d)-motifs are sometimes called subtle signals in DNA [6], and why finding them is difficult and computationally expensive.

In this paper, we propose EMS-GT, a novel algorithm that solves the planted motif problem for any arbitrary problem instance up to l=17. EMS-GT operates on a compact bit-based representation of the search space, and takes advantage of distance-related patterns in this representation in order to compute the exact solution. This study evaluated the algorithms on motif lengths that do not deviate too much from the typical motif length as described in [3].

II. Review of Related Literature

Various motif search algorithms have been developed in many previous studies. These algorithms may be classified under two main categories: *heuristic* and *exact*.

Heuristic algorithms perform a local search, for instance by repeatedly refining an input sampling or projection until a motif is found. Gibbs sampling [7] and Expectation Maximization (EM) used in the motif-finding tool MEME [8], [9], both use probabilistic computations to optimize an initial random alignment. [An alignment is simply a sequence $(a_1, a_2, ..., a_n)$ of n positions, which corresponds to the prediction that a motif of S has a d-neighbor in S_i located in position a_i.] Gibbs sampling tries to refine the alignment one position at a time. In contrast, EM may recompute the entire alignment in a single iteration. Projection [10] combines a pattern-based approach with EM's probabilistic approach, trying to guess every successive character of a tentative motif and using EM to verify its guesses. GARPS [11] uses a random version of projection, in tandem with the Genetic Algorithm (GA), for yet another iterative approach. These are just some of many successful heuristic algorithms. However, heuristics are non-exhaustive, and thus cannot guarantee to always find a solution.

Exact algorithms perform an exhaustive search of possible motifs and so always find the planted motif. One brute-force approach to do this is a pattern-based approach that tests all 4^l possible l-mers. A more efficient way is to generate the (l, d)-neighborhood of all the l-mers in the input sequence and search the motif instead of evaluating all the 4^l l-mers. PMS1 [12], an algorithm that uses this approach, generates all the d-neighborhood of all input sequences and intersects it to identify the planted motif. PMSi and PMSP [13] are both successors of PMS1 algorithm. PMSi generates the d-neighborhood of a sequence by two input sequences at a time and intersects it to get the l-mers that are common in those sequences. This approach uses less memory compared to PMS1. PMSP, unlike the previous two, generates the d-neighborhood of every l-mer in the first sequence. For every l-mer x in the first sequence, it generates the d-neighborhood of x and tests if an l-mer y in that neighborhood is a motif by checking if it exists in the remaining sequences. The testing part is improved by filtering l-mers in the remaining sequences. Any l-mer in the remaining sequences that has $2d$ distance from x is be filtered out.

WINNOWER [6] and its successor MITRA [14] are exact algorithms that look at pairwise l-mer similarity to find motifs. In a set of DNA sequences, there are numerous pairs of "similar" l-mers, which come from different sequences and have Hamming distances of at most $2d$ from each other (i.e., they could be two d-neighbors of the same l-mer). WINNOWER represents these pairs in a graph, with l-mers as nodes and edges connecting l-mer pairs. It then prunes the graph to identify "cliques" of pairs that indicate a motif. MITRA refines this graph representation into a mismatch tree containing all possible l-mers, organized by prefix. The tree structure allows MITRA to eliminate entire branches at a time, making it much faster than WINNOWER at removing the many spurious edges that are not part of any motif clique.

The current state-of-the-art in exact motif search is qPMS9, the most recent in a series [13], [5], [15] of Planted Motif Search algorithms. It performs a sample-driven step, which generates a k-tuple of l-mers from each of k input strings, followed by a pattern-driven step, which generates the common d-neighborhood of the tuple and then checks whether any of the l-mers in this common neighborhood is a motif. To identify neighbors, qPMS9 efficiently traverses the tree of all possible l-mers, using certain pruning criteria explored by predecessors PMSPrune and qPMS7 [13] to quickly discard non-neighbor branches. Sampling in qPMS9 is an improvement on its predecessor PMS8 [5]; in building a k-tuple, qPMS9 strategically prioritizes l-mers that have fewer matches with the l-mers already selected, so that the common d-neighborhood is smaller and thus require smaller time to check through. Finally, both PMS8 and qPMS9 have been implemented to run on multiple processors, allowing them to solve problem instances with (l, d) as large as (50, 21) within a few hours.

Our proposed method is similar to the BitBased [4] approach which maps an l-mer to its integer value and maintains a 4^l array that stores the d-neighborhood of a sequence for the each sequence in the dataset. Intersecting these arrays will result to the set of motifs. BitBased is optimized for parallel computation and requires specialized hardware (Nvidia Tesla C1060 and S1070 GPUs). Furthermore, unlike BitBased our approach leverages on the Hamming distance-based patterns in the search space to quickly generate d-neighborhoods in blocks, as described in Section IV.A.6.

III. Methodology

This section states how the proposed algorithm was implemented and evaluated on a specific set of challenge instances. An (l, d) instance is defined to be a challenging instance if d is the largest integer for which the expected number of motifs of length l, that would occur in the input by random chance does not exceed a constant, typically 500 motifs [15]. The specific instances that we used for this study are $(9, 2)$, $(11, 3)$, $(13, 4)$, $(15, 5)$ and $(17, 6)$ which are all those challenge instances whose l-mer lengths are near the typical motif length.

A. Datasets

A DNA sequence generator written in Java was used to produce synthetic datasets to be used in evaluating the algorithm. Each nucleotide character is randomly generated with equal chance of being selected and independent from other characters. A dataset is composed of 20 string sequences where each sequence is 600 base pairs (bp) long [6]. A random motif with l-length is first generated. Then for each of the 20 sequences, a random d-neighbor of this motif is generated and planted exactly once in some random location within the sequence. A program written in Java was also created to convert the generated synthetic datasets into FASTA format in order to be able to execute the PMS8 and qPMS9 algorithms.

B. Implementation

Initially, EMS-GT was implemented in Java, but this was later converted to C++ to minimize differing configurations for the evaluation of the algorithms. EMS-GT operates mainly on a compact bit-based enumerative representation of the motif search space. We included a speed-up technique that exploits the properties of some Hamming distance-based patterns occuring in the search space in generating the neighborhood of an l-mer.

C. Parameter Fine Tuning

The EMS-GT algorithm is composed of two main steps, namely, the Generate phase and the Test phase. The algorithm assigns the first n' $(n' \leq n)$ string sequences in the Generate phase for processing of candidate motifs and the rest of the string sequences $(n - n')$ to the Test phase for evaluating if a candidate motif is indeed a motif. The algorithm requires for each phase to have at least one string sequence to process, thus n' cannot be set to 0 or 20. Choosing the right value for the n' is vital to the algorithm's performance. For each possible n' value and for all (l, d)-challenge instances, 5 tests were run, and the runtime results were averaged. Table I shows the summary results of the test where $*$ in the table represents values that are relatively larger than 10 minutes. Based on these initial results we selected $n' = 10$ for our final imlementation across the different challenge instances.

D. Evaluation

EMS-GT was compared to known state-of-the-art algorithms PMS8 and qPMS9 by benchmarking their performance on some challenging instances of the (l, d) planted motif problem. For these challenging instances, each algorithm was run over 20 synthetic datasets, and the average runtime was computed. Lastly, EMS-GT was also evaluated using real datasets.

TABLE I
Evaluation of runtime for all possible n' values.

n'	$(9, 2)$	$(11, 3)$	$(13, 4)$	$(15, 5)$	$(17, 6)$
1	1.29 s	17.70 s	264.03 s	*	*
2	0.87 s	9.23 s	121.19 s	*	*
3	0.51 s	4.60 s	52.05 s	*	*
4	0.34 s	2.48 s	23.86 s	*	*
5	0.2 s	1.51 s	10.67 s	76.37 s	506.62 s
6	0.14 s	0.82 s	5.03 s	34.61 s	217.80 s
7	0.09 s	0.49 s	2.76 s	18.51 s	139.80 s
8	0.06 s	0.34 s	1.65 s	13.36 s	124.48 s
9	0.06 s	0.21 s	1.24 s	11.71 s	130.27 s
10	0.03 s	0.20 s	1.09 s	12.07 s	151.49 s
11	0.04 s	0.10 s	0.95 s	13.02 s	170.47 s
12	0.04 s	0.14 s	0.96 s	13.83 s	179.80 s
13	0.03 s	0.14 s	1.10 s	15.08 s	183.89 s
14	0.04 s	0.12 s	1.21 s	16.75 s	198.01 s
15	0.04 s	0.14 s	1.10 s	17.84 s	211.38 s
16	0.03 s	0.12 s	1.15 s	20.28 s	226.04 s
17	0.04 s	0.15 s	1.32 s	21.76 s	238.33 s
18	0.03 s	0.15 s	1.24 s	20.63 s	258.00 s
19	0.04 s	0.15 s	1.37 s	22.76 s	287.54 s

IV. The EMS-GT Algorithm

This section describes the steps of the Exact Motif Search - Generate and Test (EMS-GT) algorithm and how it is implemented in a program. This also includes the speedup techniques and efficient strategies that improved the performance of the program.

The EMS-GT algorithm is based on the generate-and-test principle. The idea is to narrow down the search space to a small set of "candidate" motifs based on the first n' sequences, then test each candidate motif if it exists at least once in each of the remaining $(n - n')$ sequences. The algorithm follows a two-step procedure.

(a) Generate candidates
This step takes the intersection of the d-neighborhoods of the first n' sequences $S_1, S_2, ..., S_{n'}$.

$$C = \mathcal{N}(S_1, d) \cap \mathcal{N}(S_2, d) \cap ... \cap \mathcal{N}(S_{n'}, d). \quad (1)$$

Every l-mer in the resulting set C is a candidate motif.

(b) Test candidates
This step simply checks each candidate motif $c \in C$, to determine whether a d-neighbor of c appears in all of the remaining sequences $S_{n'+1}, S_{n'+2}, ..., S_n$. If this is the case, c is accepted as a motif in set M. Effectively,

M is the set generated by the following equation:

$$M = C \cap \mathcal{N}(S_{n'+1}, d) \cap ... \cap \mathcal{N}(S_n, d). \quad (2)$$

Algorithm 1 EMS-GT

Input: set $\mathcal{S} = \{S_1, S_2, ..., S_n\}$ of L-length sequences, motif length l, allowable mismatches d
Output: set M of candidate motifs

▷ *generate candidates*
$C \leftarrow \{\}$
$\mathcal{N}(S_1, d) \leftarrow \{\}$
for $j \leftarrow 1$ to $L - l + 1$ **do**
  $x \leftarrow j^{th} l$-mer in S_1
  $\mathcal{N}(S_1, d) \leftarrow \mathcal{N}(S_1, d) \cup N(x, d)$
end for
$C \leftarrow \mathcal{N}(S_1, d)$
for $i \leftarrow 2$ to n' **do**
  $\mathcal{N}(S_i, d) \leftarrow \{\}$
  for $j \leftarrow 1$ to $L - l + 1$ **do**
    $x \leftarrow j^{th} l$-mer in S_1
    $\mathcal{N}(S_i, d) \leftarrow \mathcal{N}(S_i, d) \cup N(x, d)$
  end for
  $C \leftarrow C \cap \mathcal{N}(S_i, d)$
end for
▷ *test candidates*
$M \leftarrow \{\}$
for each l-mer c in C **do**
  $isMotif \leftarrow$ true
  for $i \leftarrow (n' + 1)$ to n **do**
    $found \leftarrow$ false
    for $j \leftarrow 1$ to $L - l + 1$ **do**
      $x \leftarrow j^{th} l-$mer in S_i
      if $dH(x, c) \leq d$ **then**
        $found \leftarrow$ true
        break
      end if
    end for
    if $!found$ **then**
      $isMotif \leftarrow$ false
      break
    end if
  end for
  if $isMotif$ **then**
    $M \leftarrow M \cup c$
  end if
end for
return M

A. Speedup Strategies

EMS-GT algorithm uses some strategies to further improve its efficiency. These strategies are discussed in this section.

Algorithm 2 GENERATE NEIGHBORHOOD

Input: DNA sequence S, motif length l, mismatches d
Output: bit-array $\mathcal{N}$ representing $\mathcal{N}(S, d)$

$\mathcal{N}[i] \leftarrow 0, \quad \forall i < 4^l$
for each l-mer x in S **do**
  ADDNEIGHBORS(x, 0, d) ▷ *recursive procedure*
end for
▷ *make d changes in l-mer x, from position s onward*
procedure ADDNEIGHBORS(x, s, d)
  for $i \leftarrow s$ to l **do**
    $\Sigma \leftarrow \{\mathtt{a}, \mathtt{g}, \mathtt{c}, \mathtt{t}\} - x_i$ ▷ i^{th} *character in* x
    for $j \leftarrow 1$ to $|\Sigma|$ **do**
      $neighbor \leftarrow concatenate(x_{1...i-1}, \Sigma_j, x_{i+1...l})$
      $\mathcal{N}[neighbor] \leftarrow 1$
      if $d > 1$ and $i < l$ **then**
        ADDNEIGHBORS($neighbor$, $i + 1$, $d - 1$)
      end if
    end for
  end for
end procedure
return $\mathcal{N}$

1. *Integer mapping of l-mers*
The algorithm maps an l-mer to a corresponding integer representation. Each character in the alphabet is represented by 2 bits (a=00, c=01, g=10, t=11). The mapping works by replacing each character in the l-mer by its 2 bits representation. Figure 1 shows an example on how an l-mer is mapped to its integer value.

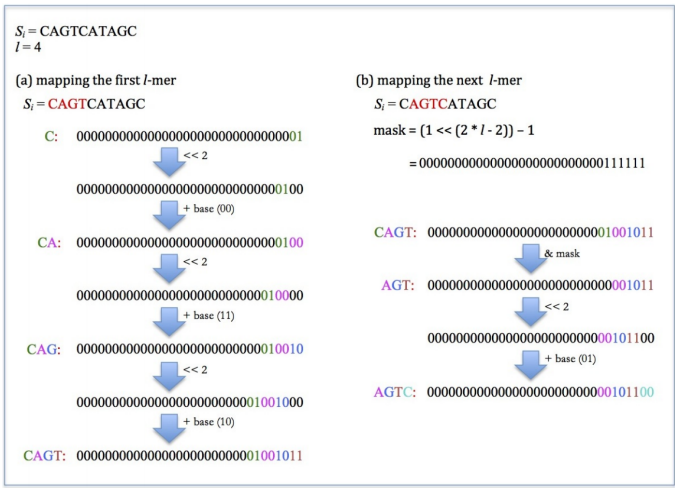

Fig. 1. Mapping of l-mer to number

2. *Bit-based set representation and l-mer enumeration*
Our EMS-GT implementation maintains a motif search space of 4^l possible flags for the l-mers. To efficiently represent sets—such as d-neighborhood, or a set of candidate motifs—within this space, EMS-GT assigns to

each of the 4^l l-mers a bit flag in an array, set to 1 if the l-mer is a member of the set and 0 otherwise. The integer representation of the l-mer is used as the index for setting the bit in the array.

Ex. `tacgt` maps to `1100011011` = 795 in decimal.
Thus, the flag for `tagct` is bit 795 in the array.

3. *Bit-array compression*
Initially the set-representation array of 4^l l-mers can be done where one flag can be stored as one entry in the array, and the integer mapping of the l-mer is used as the index in the array. However, this way of representation consumes a lot of space in memory. Our implementation of EMS-GT compresses the set-representation into an equivalent of $\frac{4^l}{32}$ 32-bit integers. The x^{th} bit is now found at position (x mod 32) of the integer at array index $\frac{x}{32}$. Figure 2 displays how the candidate motifs are stored in the array using the compression.

Ex. `tacgt` maps to `1100011011` = 795 in decimal.
bit position = 795 mod 32 = 27;
array index = 795 / 32 = 24;
Thus, the flag for `tacgt` is the 27^{th} least significant bit of the integer at array index 24.

Candidate motif to be stored in C

Candidate Motif	Bit Mapping	Integer Value (y)	q	r
AATA	00001100	12	0	12
CACG	01000110	70	3	6
GCGT	10011011	155	4	27
TTTG	11111110	254	7	30

Candidate Collection C

q/r	0	1	2	3	4	5	6	7	8	9	10	11	12	14	...	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Fig. 2. Illustration of how candidate motifs are stored in the motif collection C.

4. *XOR-based Hamming distance computation*
The integer mapping of l-mers opens up a more efficient way of processing and comparing l-mers. The binary representation of the integer is useful in Hamming distance computations. An exclusive OR (XOR) bitwise operation between the mappings of two l-mers produces a nonzero pair of bits at every mismatch position; counting these nonzero pairs of bits in the XOR result gives us the Hamming distance.

Ex. `tacgt` maps to `1100011011`
`ttcgg` maps to `1111011010`
XOR produces `0011000001` = 2 mismatches.

Algorithm 3 HAMMING DISTANCE COMPUTATION
Input: l-mer mappings u and v
Output: Hamming distance $d_H(u, v)$

$d_H(u, v) \leftarrow 0$
$z \leftarrow u \oplus v$
for $i \leftarrow 1$ to l **do**
  if $z \wedge 3 \neq 0$ **then**
    increment $d_H(u, v)$
  end if
  $z \leftarrow z >>> 2$ ▷ *shift 2 bits to the right*
end for
return $d_H(u, v)$

5. *Recursive neighborhood generation*
Our implementation of EMS-GT generates the d-neighborhood of an l-mer x by choosing $d' \leq d$ positions from 1, 2, ..., l-1, l and changes the character at each of the d' positions in x. The neighborhood $N(x, d)$ can be viewed as a tree $\mathcal{T}(x)$ of height d where each node is a pair (w, p) where w is an l-mer mapping and p is an integer $0 \leq p \leq l$ corresponding to a base position where the l-mer is mutated. Given the illustration in Figure 3, it is obvious to state that the root node of $\mathcal{T}(x)$ is $(x, 0)$. Also all nodes with depth δ are l-mers whose Hamming distance from the node $(x, 0)$ is δ. It is also easy to see that node (x', p) and any of its child has a Hamming distance of 1.

Given a node (w, p) where $p \neq l$, three neighbors are generated. Those are the three variants of w replacing the character at base position $p+1$ with one of the other characters in Σ. Each neighbor is generated by the bit computation.

$$y = w \wedge c << ((l - p) * 2 - 2) \tag{3}$$

where $c = \{1, 2, 3\}$, $\wedge$ is the bit AND operator, and $<<$ is the bit shift left operator.

The expected size of the neighborhood $N(x, d)$ can be computed by:

$$|N(x, d)| = \sum_{i=0}^{d} \binom{l}{i} 3^i \tag{4}$$

A visualization of how the recursive generation of neighborhood works is shown in Figure 3.

6. *Block-based optimization for neighborhood generation*
This is the most crucial speedup technique used in the proposed algorithm. As the value of l or d increases so does $|N(x, d)|$. This means that as the value of l and d increases, the recursive neighborhood generation

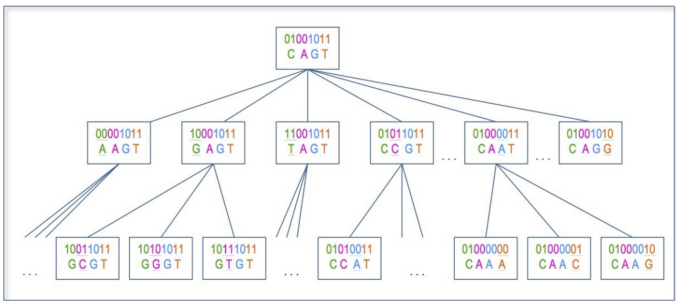

Fig. 3. $\mathcal{T}(x)$ with d = 2 and x = 01010011, number mapping for CAGT. The base position is reflected in the underlined bit representatives. For example, p = 1 at node 11001011 (TAGT), p = 3 at node 01010011 (CCAT).

(Algorithm 2) becomes a bottleneck in the program.

We represent the neighborhood $N(x, d)$ of an l-mer x as an array N of 4^l bit flags where we set a bit to 1 if the corresponding l-mer is a neighbor and 0 otherwise.

$$N_x = \begin{cases} 1 & \text{if } dH(x, x') \leq d, \\ 0 & \text{otherwise.} \end{cases} \quad \text{for any } l\text{-mer } x'.$$

We observed that dividing this bit array N into consecutive blocks of 4^k flags each, for some k, $0 < k < l$, each block conforms to one of $(k + 2)$ possible bit patterns see Figure 4. We take advantage of this regularity to build N by blocks.

To generate $N(x, d)$ for some l-mer x. We first divide x to its **prefix** $\boldsymbol{y}$, which is the first $l - k$ characters, and its **suffix** $\boldsymbol{z}$, the remaining k characters.

Ex. Setting k = 5, x = `acgtacgtacgt`
is divided into y = `acgtacg` and z = `tacgt`.

We then generate the **Hamming distance matrix** $\boldsymbol{\mathcal{D}(z)}$ of Hamming distances from z to all 4^k possible k-suffixes. Within $\mathcal{D}(z)$ the minimum value is 0 (at z itself) and the maximum is k.

Since the set representation of the 4^l neighborhood is enumerated alphabetically, the 4^k l-mers that is grouped together in a block begins with the same $(l - k)$ characters—the **block prefix** $\boldsymbol{y'}$. The block's **prefix distance** $\boldsymbol{d'_y}$ is the Hamming distance $dH(y, y')$ between x's prefix and the block prefix.

Ex. For block {`acgttgcaaaaa` to `acgttgcttttt`}
the prefix distance from x = `acgtacgtacgt` is
$d_{y'} = dH(\texttt{acgtacg}, \texttt{acgttgc}) = 3$.

We can infer that the distance between two l-mers is equal to the sum of the distance between their prefixes and the distance between their k-suffixes. Given this observation,

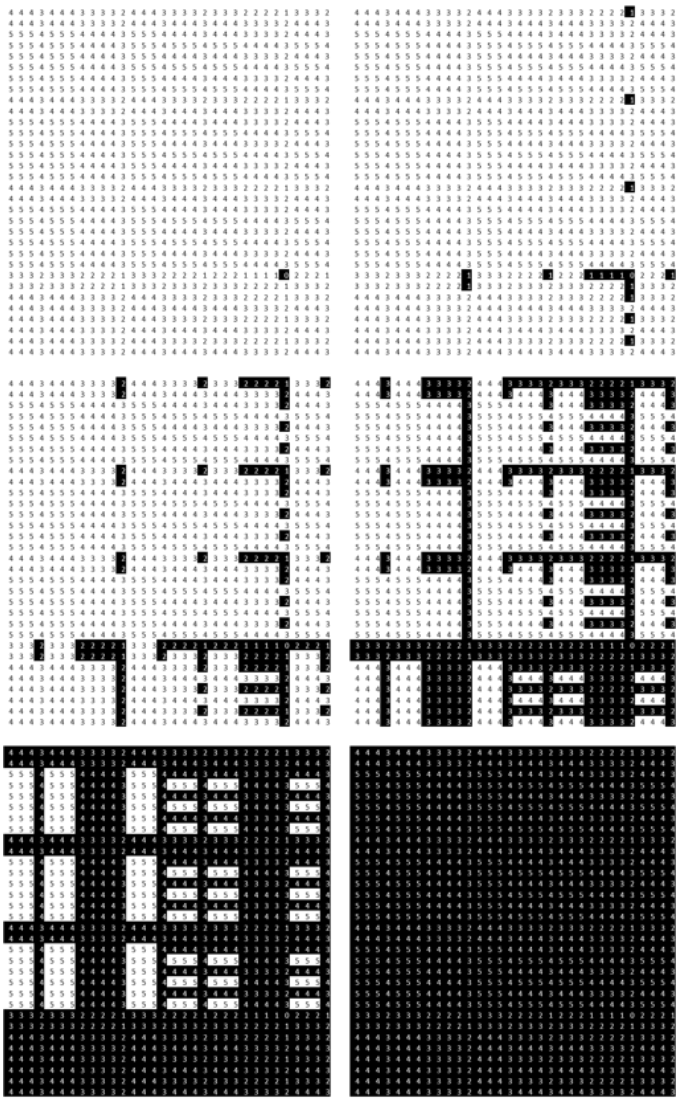

Fig. 4. Bit patterns followed by blocks of size $4^5 = 32 \times 32$ in the bit-based representation of $\mathcal{N}(\texttt{acgtacgtacgt}, 5)$. Black signifies a 1. There are $(5 + 2) = 7$ possible patterns—the empty pattern (all 0s) is not shown.

if we have the value for d'_y and $\mathcal{D}(z)$ we can compute the distance from x to any l-mer $x' = y'z'$ in a block as:

$$dH(x, x') = d_{y'} + \mathcal{D}(z)_{z'} \tag{5}$$

We can redefine the criteria for setting a bit in N:

$$N_{x'} = \begin{cases} 1 & \text{if } d_{y'} + \mathcal{D}(z)_{z'} \leq d, \\ 0 & \text{otherwise.} \end{cases} \quad \text{for } x' = y'z'.$$

Given this, we can observe that a bit at position z' within a block with prefix y' is set if and only if $\mathcal{D}(z)'_z A \leq d - d'_y$. The values in $\mathcal{D}(z)$ range from 0 to k; therefore

when $d - d_{y'} < 0$, no bits in the block are set;
when $0 \leq d - d_{y'} < k$, some bits are set (k ways); and

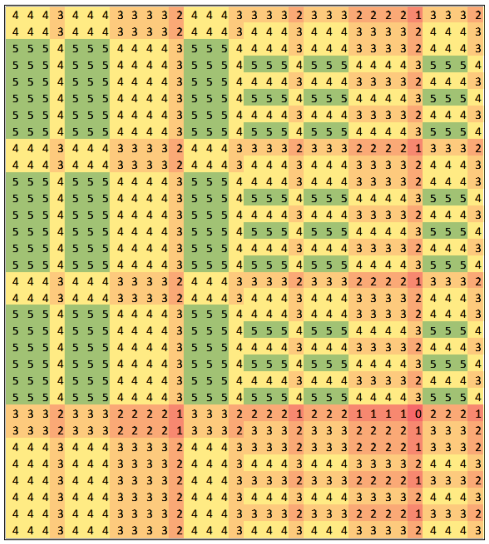

Fig. 5. Hamming distance matrix values of 5-mer `tacgt` to all $4^5 = 32 \times 32$ k-suffixes of length 5. The value in the p^{th} cell of this 32×32 table (at row $\frac{p}{32}$, col p *mod* 32) is the Hamming distance from the **center** `tacgt` to the k-suffix that maps to the binary number p.
Ex. $\mathcal{D}(\texttt{tacgt})_{100} = dH(\texttt{tacgt}, \texttt{acgca}) = 5$.

when $d - d_{y'} \geq k$, all bits in the block are set.

Another way to view this is to let the value of $d - d'_y$ be equal to d' and is the remaining number of allowed mutations. Any bit z' in the block is set to 1 if $dH(z, z')$ is within the value of d'. This implies that there are only $(k + 2)$ unique patterns of bits, including *"empty"* (all 0s) and *"full"* (all 1s), for blocks in N.

V. Results

EMS-GT and the state-of-the-art algorithms PMS8 and qPMS9 were run on Intel Xeon, 2.10 Ghz machine. The average of their runtimes on 20 synthetic datasets were recorded. Since EMS-GT does not have a parallel implementation yet, the experiment used the qPMS9 in *NOMPI* mode, which sets qPMS9 to run on single core only, for fair evaluation. Table II shows the runtimes of the algorithms for the different (l, d)-challenge instances, while Table III shows the speedup difference rate of EMS-GT compared to qPMS9 algorithm.

TABLE II
Runtime comparison between EMS-GT, PMS8 and qPMS9

(l, d)	PMS8	qPMS9	EMS-GT
(9,2)	0.76 s	0.62 s	**0.03 s**
(11,3)	1.62 s	1.26 s	**0.13 s**
(13,4)	6.33 s	4.38 s	**0.84 s**
(15,5)	39.99 s	24.64 s	**12.71 s**
(17,6)	258.10 s	**119.70** s	150.09 s

EMS-GT is able to compete with the state-of-the-art algorithms PMS8 and qPMS9. For every (l, d)-challenge instances except $(17, 6)$ there is a significant difference between their runtimes. EMS-GT exhausts the 4^l search space and thus the l value affects the size of the search space. Theoretically, the bigger the search space the longer the algorithm has to process it. Unlike in qPMS9, that works on k-number of tuples across different sequences, the larger the l size the less number of expected tuple combinations it has to consider, the more effective is the pruning step, and thus the faster the algorithm can process the sequences. This observation is the reason why EMS-GT failed to beat qPMS9 in $(17, 6)$-challenge instance. Finally our EMS-GT implementation has only been evaluated when $l \leq 17$ due to memory constraint. Maintaining the entire search space consumes a lot of memory and l value defines the size of that search space. In practice, this isn't a problem since previous studies showed that significant motif length is around 10bp to 16 bp.

TABLE III
Runtime speedup difference rate of EMS-GT vs. qPMS9

(l, d)	qPMS9	EMS-GT	% speedup
(9,2)	0.62 s	0.03 s	95.1%
(11,3)	1.26 s	0.13 s	89.6%
(13,4)	4.38 s	0.84 s	80.8%
(15,5)	24.64 s	12.71 s	48.4%
(17,6)	119.70 s	150.09 s	–

TABLE IV
Test on promoter sequences of Saccharomyces cerevisiae.

Transcription Factor/ Dataset Size [t, n]	Published & Detected Motif	(l, d) Used	Run Time
PHO4	CACGTG	(6, 2)	0.04 s
[4, 250]	CACGTT	(6, 2)	0.04 s
HSE, HSTF	TTCNNGAA	(8, 2)	0.10 s
[5, 349]	GAANNTCC	(8, 3)	0.20 s
	TTCNNNGAA	(9, 3)	
	GAANNNTCC	(9, 3)	
MCB	ACGCGT	(6, 2)	0.03 s
[5, 250]			
PDR3	TCCGTGAA	(8, 2)	0.09 s
[7, 250]	TCCGCGAA	(8, 2)	0.09 s

Aside from testing the EMS-GT algorithm on synthetic datasets, it was also run using real datasets. EMS-GT algorithm was able to find quickly the motifs in set promoter sequences of yeast (*Saccharomyces cerevisiae*) [16] as shown in Table IV. Table V shows the result of the algorithm run on real datasets involving orthologous sequences of different gene families of eukaryotes

VI. Conclusions

In this study we propose a novel exact enumerative algorithm for the planted motif problem. A C++ implementation of the algorithm run on various challenge problem instances shows that it is very competitive against the state-of-the-art exact algorithms PMS8 and qPMS9. For challenge instances

TABLE V
TEST ON ORTHOLOGOUS SEQUENCES OF GENES OF EUKARYOTES.

Gene Family/ Dataset Size [t, n]	**Published & Detected Motif**	(l, d) **Used**	**Run Time**
Insulin	AAGACTCTAA	(10, 4)	0.07 s
$[8, 500]$	GCCATCTGCC	(10, 3)	0.06 s
	CTATAAAG	(8, 2)	0.04 s
	GGGAAATG	(8, 2)	0.04 s
Metallothionein	GCTATAAA	(8, 3)	0.04 s
$[26, 590]$	CATGCGCAG	(9, 3)	0.06 s
	TTTGCACACG	(10, 3)	0.06 s
	TAACTGATAAA	(11, 5)	0.60 s
	TACACTCAG	(9, 3)	0.06 s
	CAGGCACCT	(9, 3)	0.06 s
	GTACATTGT	(9, 3)	0.06 s
c-myc	GTTTATTC	(8, 1)	0.04 s
$[7, 1000]$	TTGCTGGG	(8, 2)	0.03 s
	GGCGCGCAGT	(10, 3)	0.07 s
	CAGCTGTTCC	(10, 3)	0.07 s
	CCCTCCCC	(8, 1)	0.04 s
	AGCAGAGGGCG	(11, 4)	0.21 s
	GGCGTGGG	(8, 3)	0.04 s
	ATCTCCGCCCA	(11, 3)	0.10 s
c-fos	GAGTTGGCTG	(10, 3)	0.06 s
$[6, 700]$	GTTCCCGTCAATC	(13, 5)	1.10 s
	CACAGGATGT	(10, 4)	0.08 s
	AGGACATCTG	(10, 4)	0.08 s
	TACTCCAACCGC	(12, 4)	0.18 s
Growth Hormone	GGGAGGAG	(8, 3)	0.04 s
$[16, 380]$	ATTATCCAT	(9, 4)	0.06 s
	TTAGCACAA	(9, 3)	0.05 s
	GTCAGTGG	(8, 3)	0.04 s
	ATAAATGTA	(9, 4)	0.06 s
	TATAAAAAG	(9, 3)	0.05 s
	TCATGTTTT	(9, 4)	0.06 s
Interleukin-3	TTGAGTACT	(9, 3)	0.04 s
$[6, 490]$	GATGAATAAT	(10, 4)	0.07 s
	TCTTCAGAG	(9, 3)	0.04 s
	AGGACCAG	(8, 2)	0.03 s
Histone H1	CAATCACCAC	(10, 3)	0.06 s
$[4, 650]$	AAACAAAAGT	(10, 3)	0.06 s

$(9, 2)$, $(11, 3)$, $(13, 4)$ and $(15, 5)$ the algorithm outperforms the current best algorithm with runtime reduction of at least 95.1%, 89.6%, 80.8% and 48.4% respectively, while ranking second to qPMS9 in $(17, 6)$-challenge instance. Further research can explore other speedup techniques using the by block strategy, refining the bit-based storage mechanism to be able to represent the search space more efficiently and exploring other approaches to improve the way the algorithm eliminate candidate motifs.

VII. CONFLICTS OF INTEREST

None declared.

VIII. ACKNOWLEDGMENTS

This study is funded by DOST-ERDT scholarship grant. Aside from the financial support, the sponsors do not have any involvement in the study.

REFERENCES

[1] M. K. Das and H.-K. Dai, "A survey of dna motif finding algorithms," *BMC bioinformatics*, vol. 8, no. Suppl 7, p. S21, 2007.

[2] T. L. Bailey, "Discovering sequence motifs," *Bioinformatics: Data, Sequence Analysis and Evolution*, vol. 452, pp. 231 – 251, 2008.

[3] A. J. Stewart, S. Hannenhalli, and J. B. Plotkin, "Why transcription factor binding sites are ten nucleotides long," *Genetics*, vol. 192, no. 3, pp. 973–985, 2012.

[4] N. S. Dasari, R. Desh, and M. Zubair, "An efficient multicore implementation of planted motif problem," in *High Performance Computing and Simulation (HPCS), 2010 International Conference on*. IEEE, 2010, pp. 9–15.

[5] M. Nicolae and S. Rajasekaran, "Efficient sequential and parallel algorithms for planted motif search," *BMC bioinformatics*, vol. 15, no. 1, p. 34, 2014.

[6] P. A. Pevzner, S.-H. Sze *et al.*, "Combinatorial approaches to finding subtle signals in dna sequences." in *ISMB*, vol. 8, 2000, pp. 269–278.

[7] C. E. Lawrence, S. F. Altschul, M. S. Boguski, J. S. Liu, A. F. Neuwald, and J. C. Wootton, "Detecting subtle sequence signals: a gibbs sampling strategy for multiple alignment," *science*, vol. 262, no. 5131, pp. 208–214, 1993.

[8] C. E. Lawrence and A. A. Reilly, "An expectation maximization (em) algorithm for the identification and characterization of common sites in unaligned biopolymer sequences," *Proteins: Structure, Function, and Bioinformatics*, vol. 7, no. 1, pp. 41–51, 1990.

[9] T. L. Bailey and C. Elkan, "Unsupervised learning of multiple motifs in biopolymers using expectation maximization," *Machine learning*, vol. 21, no. 1-2, pp. 51–80, 1995.

[10] M. Blanchette and M. Tompa, "Discovery of regulatory elements by a computational method for phylogenetic footprinting," *Genome research*, vol. 12, no. 5, pp. 739–748, 2002.

[11] H. Huo, Z. Zhao, V. Stojkovic, and L. Liu, "Combining genetic algorithm and random projection strategy for (l, d)-motif discovery," in *Bio-Inspired Computing, 2009. BIC-TA'09*. IEEE, 2009, pp. 1–6.

[12] S. B. S. Rajasekaran and C.-H. Huang, "Exact algorithms for planted motif challenge problems," *Journal of Computational Biology*, vol. 452, pp. 1117– 1128, 2005.

[13] J. Davila, S. Balla, and S. Rajasekaran, "Fast and practical algorithms for planted (l, d) motif search," *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, vol. 4, no. 4, pp. 544–552, 2007.

[14] E. Eskin and P. A. Pevzner, "Finding composite regulatory patterns in dna sequences," *Bioinformatics*, vol. 18, no. suppl 1, pp. S354–S363, 2002.

[15] M. Nicolae and S. Rajasekaran, "qpms9: An efficient algorithm for quorum planted motif search," *Scientific reports*, vol. 5, 2015.

[16] J. Zhu and M. Q. Zhang, "Scpd: a promoter database of the yeast saccharomyces cerevisiae." *Bioinformatics*, vol. 15, no. 7, pp. 607–611, 1999.