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

Proceso L. Fernandez, Jr.
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 each sequence S_i from an input set $\{S_1, ..., S_t\}$ 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 Java 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 87%, 79%, 77% and 37% respectively for these instances, while ranking second to qPMS9 for challenge instance (17,6).**

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]; a motif might be a transcription factor binding site, a promoter element, a splicing site, or a marker useful for classification. There are many variants of motif finding problem in the 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 [2]. One of the latter type is the planted motif problem.

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

atcactcgttctcctctaatgtgtaaagacgtactaccgacctta
acgccgaccggtccgatccttgtatagctcctaacgggcatcagc
tcctgactgcatcgcgatctcggtagtttcctgttcatcatttt
ggccctcagcatcgtgcgtcctgctaacacattcccatgcagctt
tgaaaagaatttacggtaaaggatccacatccaatcgtgtgaaag

Motif: ccatcgtt

The planted motif problem simply asks: "*Given a set of DNA sequences, can we find an unknown motif of length l that appears at different positions in each of the sequences* [3]?" 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 [4]. We define the planted motif problem more formally as follows:

Given a set $S = \{S_1, ...S_t\}$ of n DNA sequences of length L, find M, the set of sequences (or motifs) of length $l < L$ which have at least one d-neighbor in each sequence in S.

A ***d*-neighbor** of an ***l*-mer** (a sequence of length l) x is an l-mer that has at most d mismatched characters with x. Two distinct d-neighbors of a motif–both counting as valid occurrences of the motif–might differ in as many as $2d$ characters! This shows why (l, d)-motifs are sometimes called subtle signals in DNA [3], 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. It 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. Section 2 discusses related literature, focusing on exact motif finding approaches. Section 3 details our methods. Section 4 describes our implementation of the proposed algorithm, and compares it against state-of-the-art exact algorithms. Section 5 presents our conclusions and recommendations.

II. Review of Related Literature

Motif finding is a well-studied problem in computing. Various motif search algorithms have been developed, falling into two main categories: *heuristic* and *exact*. Heuristic algorithms perform an iterative local search, for instance by repeatedly refining an input sampling or projection until a motif is found. Gibbs sampling [5] and Expectation Maximization (EM), used in the motif-finding tool MEME [6], [7] both use probabilistic computations to optimize an initial random alignment. (An alignment is simply a set $\{a_1, a_2, ..., a_n\}$ of n positions, which predicts that the motif occurs at position a_i in the given sequence S_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 [8] 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 [9] 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 not always guaranteed to find a solution. Exact algorithms, on the other hand, perform an exhaustive search of possible motifs and so always find the planted motif. Some useful definitions for discussing exact motif-search algorithms:

- 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 dH** between two l-mers of equal length is the number of characters that differ between them. Ex. $dH(\texttt{gattaca}, \texttt{cgttaga}) = 3$.
- The ***d*-neighborhood of an *l*-mer x** is the set $\boldsymbol{N(x, d)}$ containing all d-neighbors of x— all l-mers x' whose Hamming distance from x is at most d, i.e., $\boldsymbol{dH(x, x') \leq d}$. Ex. gatct, cctta, and aatta are all in $N(\texttt{gatta}, 2)$.
- The ***d*-neighborhood of a sequence S** is the set $\boldsymbol{\mathcal{N}(S, d)}$ containing all d-neighbors of all l-mers in S. Ex. for l=5, $\mathcal{N}(\texttt{gattaca}, 2) = N(\texttt{gatta}, 2) \cup N(\texttt{attac}, 2) \cup N(\texttt{ttaca}, 2)$

WINNOWER [3] and its successor MITRA [10] 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 (meaning that 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 [11], [4], [12] 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 [11] to quickly discard non-neighbor branches. Sampling in qPMS9 is an improvement on its predecessor PMS8 [4]; in building a k-tuple, qPMS9 intelligently prioritizes l-mers that have fewer matches with the l-mers already selected, such that the common d-neighborhood becomes smaller and thus faster 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) in a few hours.

Our proposed method is similar to the BitBased approach described in [2]; the main difference is that BitBased is optimized for parallel computation and runs on specialized hardware (Nvidia Tesla C1060 and S1070 GPUs).

III. Methodology

A. Datasets

Synthetic datasets were created using a DNA sequence generator written in Java. Each nucleotide character in a sequence is randomly generated; {a, g, c, t} each have a 25% chance of being selected, independent from other characters in the sequence. The motif is then planted at a random position in the sequence. As prescribed in [3] every dataset contains 20 DNA sequences each 600 bases long, with an (l, d) motif planted exactly once in each sequence.

B. Implementation

Our Java implementation of EMS-GT operates mainly on a compact, bit-based enumerative representation of the motif search space. Since a significant part of runtime is spent locating and setting bits in the bit-based representation, optimizations were explored for the bit-setting portion of the algorithm. We investigated the properties of some Hamming distance-based patterns occurring in the search space, then formulated a speed-up technique that exploits these patterns to set bits more quickly and in contiguous blocks.

C. Evaluation

EMS-GT was compared to known state-of-the-art algorithms PMS8 and qPMS9 by benchmarking their performance on challenging instances of the (l, d) planted motif problem. An (l, d) problem instance is defined to be a challenging instance if d is the largest value for which the expected number of l-length motifs that would occur in the input by random chance does not exceed some limit—typically 500 random motifs [12]. The instances used are (9,2), (11,3), (13,4), (15,5), and (17,6), as identified in [12], [11].

IV. Results

A. The EMS-GT algorithm

EMS-GT is based on the candidate 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 do a brute-force search for each candidate on the remaining $(n - n')$ sequences to confirm whether or not it is a motif. This approach proceeds in two main steps:

(a) Generate candidates
This step takes the intersection of the d-neighborhoods of the first n' sequences $S_1, S_2, ..., S_{n'}$. Every l-mer in the resulting set C is a candidate motif.

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

(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.

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

Algorithm 1 EMS-GT

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

▷ *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 u 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
      compute $dH(x, u)$
      if $dH(x, u) \leq d$ **then**
        $found \leftarrow$ true
        break
      end if
    end for
    if $!found$ **then**
      $isMotif \leftarrow$ false
      break
    end if
  end for
  if $isMotif$ **then**
    $M \leftarrow M \cup u$
  end if
end for

We discuss further some efficiency strategies used in our implementation:

1. *Bit-based set representation and l-mer enumeration*
The motif search space consists of the 4^l possible l-mers that can be formed from the nucleic alphabet {a, g, c, t}. To efficiently represent sets—such as a d-neighborhood, or a set of candidate motifs—within this space, EMS-GT assigns 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. Bit flags correspond to l-mers via a simple mapping: EMS-GT maps an l-mer s to a bit flag index x by replacing each character with 2 bits (a=00, c=01, g=10, t=11). Note that this mapping scheme enumerates l-mers in strict alphabetical order.

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

2. *Bit-array compression*
Our EMS-GT implementation compresses the required set-representation array of 4^l bits into an equivalent array 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 $\frac{x}{32}$.

   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.

3. *XOR-based Hamming distance computation*
The mapping of l-mers to binary numbers is also useful for binary Hamming distance computations. An exclusive OR (XOR) bitwise operation between the mappings of two l-mers will produce 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.

4. *Recursive neighborhood generation*
To generate a d-neighbor of an l-mer x, we choose $d' \leq d$ positions from 1, 2,..., l-1, l and change the character at each of the d' positions in x. EMS-GT uses a recursive procedure (Alg. 2) to do this, effectively (1) traversing the tree of all d-neighbors and (2) setting the bit flag in the neighborhood array N for each neighbor it encounters. Since we choose up to d positions in the l-mer, and have 3 possible substitute characters at each position, the size of the neighborhood $N(x, d)$ is given by:

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

Algorithm 2 GENERATE NEIGHBORHOOD

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

$\mathcal{N} \leftarrow \{\}$
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**
    $\Sigma \leftarrow \{a, g, c, t\} - x_i$ ▷ *i^{th} character in x*
    for $j \leftarrow 1$ to $|\Sigma|$ **do**
      $neighbor \leftarrow x_{1...i-1} + \Sigma_j + x_{i+1...l}$
      $\mathcal{N}[neighbor] \leftarrow 1$
      if $d > 1$ and $i < l$ **then**
        ADDNEIGHBORS($neighbor$, $i + 1$, $d - 1$)
      end if
    end for
  end for
end procedure

5. *Block-based optimization for neighborhood generation*
We can represent the neighborhood $N(x, d)$ of an l-mer x as an array N of 4^l bit flags, set 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 find that if we divide this bit array N into consecutive blocks of 4^k flags each, for some k, $0 < k < l$, each block will conform to one of $(k+2)$ possible bit patterns. We exploit this regularity to build N in blocks.

Say we wish to generate $N(x, d)$ for some l-mer x. We first divide x into its **prefix** $\boldsymbol{y}$ (first $l - k$ characters) and its ***k*-suffix** $\boldsymbol{z}$ (last k characters).

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

We then generate the **distribution** $\boldsymbol{\mathcal{D}(z)}$ of Hamming distances from z to all 4^k possible k-suffixes (Fig. 1). Within $\mathcal{D}(z)$ the minimum value is 0 (at z itself) and the maximum is k.

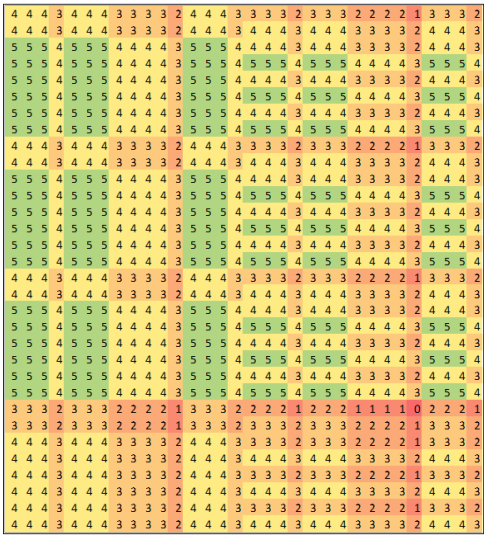

Fig. 1. Distribution $\mathcal{D}$(`tacgt`) of Hamming distances from `tacgt` to all $4^5 = 32 \times 32$ k-suffixes of length 5. The value in the p^{th} cell of this 32×32 table (at row $\frac{p}{32}$, col p *mod* 32) is the Hamming distance from the **center** `tacgt` to the k-suffix that maps to the binary number p.
Ex. $\mathcal{D}(\texttt{tacgt})_{100} = dH(\texttt{tacgt}, \texttt{acgca}) = 5$.

Due to the alphabetical enumeration, the 4^k l-mers grouped together in a block will all begin with the same $(l - k)$ characters—the **block prefix** $\boldsymbol{y'}$. The block's **prefix distance** $\boldsymbol{d_{y'}}$ is just 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 any two l-mers is equal to the sum of the distance between their prefixes and the distance between their k-suffixes; thus, given $d_{y'}$ and $\mathcal{D}(z)$, we can compute the distance from x to any l-mer $x' = y'z'$ in a block as:

$$dH(x, x') = d_{y'} + \mathcal{D}(z)_{z'} \tag{4}$$

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

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

From this we see that a bit at position z' within a block with prefix y' will be set if and only if $D(z)_{z'} \leq d - d_{y'}$. The values in $D(z)$ 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
when $d - d_{y'} \geq k$,	all bits in the block are set.

This allows for $(k + 2)$ unique patterns of bits, including "empty" (all 0's) and "full" (all 1's), for blocks in N.

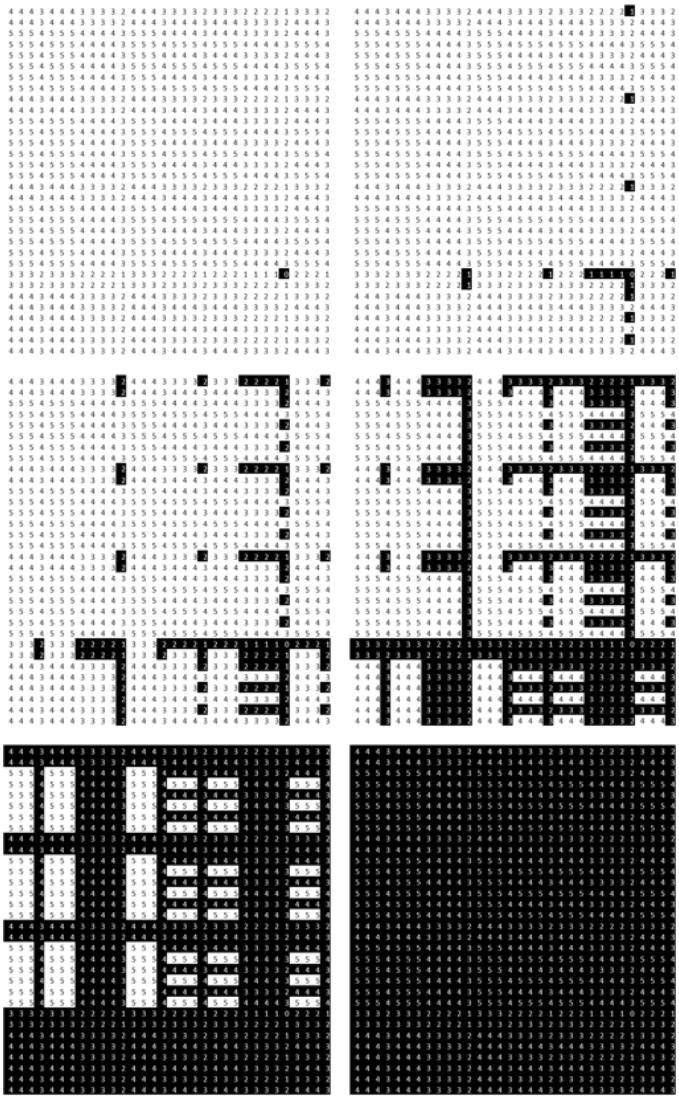

Fig. 2. Bit patterns followed by blocks of size $4^5 = 32 \times 32$ in the bit-based representation of $\mathcal{N}$(`acgtacgtacgt`, 5). Black signifies a 1. There are $(5 + 2) = 7$ possible patterns—the empty pattern (all 0s) is not shown.

To build the bit-array representing N, we simply assign the correct bit patterns to non-empty blocks. Non-empty blocks fulfill the condition $d - d_{y'} \geq 0$, i.e., the block prefix y' is a d-neighbor of x's prefix y. We can use our recursive procedure (Alg. 2) to generate $N(y, d)$, thus giving us the prefixes of all non-empty blocks in N.

TABLE I. NUMBER OF INDIVIDUAL NEIGHBORS GENERATED

(l, d)	$N(x, d)$ $\sum_{i=0}^{l} \binom{l}{i} 3^i$	$N(y, d)$, k=5 $\sum_{i=0}^{l-k} \binom{l-k}{i} 3^i$	% reduction
9,2	351	66	81.2%
11,3	4,983	693	86.1%
13,4	66,378	7,458	88.8%
15,5	853,569	81,921	90.4%
17,6	10,738,203	912,717	91.5%

Table I shows us that identifying non-empty blocks requires much fewer individual neighbors to be generated—when k=5, $N(y, d)$ is only about 10-20% of the entire neighborhood $N(x, d)$. The rest of the work to build N is done by repeatedly masking known bit patterns onto these non-empty blocks.

The block-masking optimization entails non-negligible space and time requirements; both are a function of the suffix length k. Storing all possible "non-trivial" block patterns—a set which excludes the "empty" block, the "full" block, and blocks in which only one bit is set (the case when $d_{y'} = d$)—requires $4^{2k} \times (k - 1)$ bits. Generating these non-trivial patterns takes $O(4^{2k})$ time, and applying these patterns to the bit-array takes $O(4^k \times \sum_{i=0}^{l-k} \binom{l-k}{i} 3^i)$ time.

Still, for most values of (l, d) building the neighborhood bit-array in blocks proves faster than recursively generating each individual neighbor and setting its bit flag. This is shown in Table II, which lists average runtimes of EMS-GT with and without the block-based optimization, run on 20 synthetic datasets per (l, d) challenge instance.

TABLE II. EMS-GT RUNTIME WITH VS. WITHOUT BLOCK-MASKING

(l, d)	No block masking	With block masking, k=5	% speedup
9,2	0.06 s	0.11 s	–
11,3	0.22 s	0.20 s	6.7%
13,4	1.98 s	1.04 s	47.5%
15,5	25.06 s	15.51 s	38.1%
17,6	308.61 s	175.85 s	43.0%

Finally, an implementation note: in EMS-GT, building the bit-array N for the neighborhood $N(x, d)$ of a single l-mer x is just part of a larger goal, which is to build the bit-array $\mathcal{N}$ for the neighborhood $\mathcal{N}(S, d)$ of S, the sequence containing x. It is thus more efficient to build $\mathcal{N}$ iteratively, than to generate each individual N.

For each x in S, we *mask* the bit patterns that represent $N(x, d)$ onto the bit-array $\mathcal{N}$. This masking effectively performs $\mathcal{N}(S, d) \leftarrow \mathcal{N}(S, d) \cup N(x, d)$ for each x in S, as specified in lines 4-7 and 11-14 of EMS-GT (Alg. 1).

B. Performance of EMS-GT

EMS-GT and two competitor algorithms were run on an Intel Xeon, insert details machine. Their performance, averaged over 20 synthetic datasets for each (l, d) challenge instance, is outlined in Table III:

TABLE III. RUNTIME COMPARISON OF PMS8, QPMS9 AND EMS-GT

(l, d)	PMS8	qPMS9	EMS-GT	% speedup
9,2	0.74 s	0.47 s	**0.11 s**	76.6%
11,3	1.58 s	1.06 s	**0.20 s**	81.1%
13,4	5.39 s	4.52 s	**1.04 s**	77.0%
15,5	36.45 s	24.63 s	**15.51 s**	37.0%
17,6	3.91 min	**1.96 min**	*2.93 min*	–

For every challenge instance except (17,6) EMS-GT outperforms qPMS9; it outperforms PMS8 for instance (17,6). EMS-GT was run including the block-masking optimization, with the default suffix length of k=5. Observe that our EMS-GT implementation can only solve problem instances where $l \leq 17$. This is because when we reach l=18, the size of the integer array needed to represent the entire search space ($\frac{4^{18}}{32} = \frac{2^{36}}{2^5} = 2^{31}$ integers) begins to exceed the maximum size for Java arrays, which is $(2^{31} - 1)$ elements.

V. CONCLUSION

In this paper we propose a novel algorithm for the planted motif problem. A Java implementation run on various challenge problem instances shows the proposed algorithm to be very competitive against state-of-the-art exact algorithms PMS8 and qPMS9. We find EMS-GT works extremely well for problems involving short motifs, outperforming the current best algorithm for the challenge problem instances (9,2), (11,3), (13,4) and (15,5) with a run-time reduction of 76%, 81%, 77% and 37% respectively for these instances, while ranking second to qPMS9 for test instance (17,6). Directions for further research include: refining the bit-based storage mechanism to be able to represent the search space for $l > 17$; creating a multiprocessor version of EMS-GT to solve in parallel for larger values of (l,d); and delegating the bit-masking optimization and other bulk bit operations to the graphics card, as explored in [2], for faster performance.

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] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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. Fourth International Conference on*. IEEE, 2009, pp. 1–6.

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

[11] 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.

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