EMS-GT2: An Improved Exact Solution for the (*l, d*)-Planted Motif Problem

Mark Joseph D. Ronquillo and Proceso L. Fernandez

Ateneo de Manila University, Loyola Heights, Quezon City

Author Note

Mark Joseph D. Ronquillo, Department of Information System and Computer Science, Ateneo de Manila University.

Proceso L. Fernandez PhD., Department of Information System and Computer Science, Ateneo de Manila University.

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

Contact Info: markronquillo23@gmail.com

Abstract

Finding DNA motifs is a widely studied area in the field of Computational Biology. Motifs signify different information that is useful for biologists. There are several variations of the motif finding problem, and one of these is called the (*l, d*)- motif search or Planted Motif Search problem (PMS). In this paper, we propose the EMS-GT2 algorithm, an extension of the Exact Motif Search - Generate and Test (EMS- GT) which is an exact enumerative algorithm for PMS. In EMS-GT2, we incorporated a new speedup technique that is based on an important property that we have discovered, which we prove in this paper, and which has enabled a more efficient block-processing of candidate motifs. Our C++ implementation of EMS-GT2 running on synthetic data for several PMS challenge instances demonstrate that it is competitive with both the EMS-GT and qPMS9, the two current best exact solutions for PMS. In particular, EMS-GT2 is able to reduce the run-times of EMS-GT by 20.3%, 15.8% and 22.6% for the (*l, d*) challenge instances (13, 4), (15, 5) and (17, 6) respectively. It also outperforms qPMS9, having runtime reductions of 91.6%, 79.3%, 82.0%, 59.4% and 9.7% for the (9, 2), (11, 3), (13, 4), (15, 5) and (17, 6) synthetic challenge instances respectively.

Keywords: planted (*l, d*)-motif problem, bit-based, exact enumerative algorithm

EMS-GT2: An Improved Exact Solution for the (*l, d*)-Planted Motif Problem

DNA motif finding is a well-studied topic in computational biology. A motif is a short pattern of interest that occurs in large amount of biological data. Detection of these motifs often leads to new biological discoveries. This may lead to finding transcription factor binding sites that help biologists understand gene functions, understand human diseases, identify potential therapeutic drug targets and conclude commonalities from different species.

There are many variations of the motif search problem, such as the Simple Motif Search (SMS), the Edit-distance-based Motif Search (EMS) and the Planted Motif Search (PMS) which is also known as (*l, d*)-motif search. This study focuses on an exact enumerative algorithm for the PMS problem.

Planted Motif Search is formally defined in (Rajasekaran, Balla, & Huang, 2005) as "*Input are t sequences of length n each. Input also are two integers l and d. The problem is to find a motif (i.e., a sequence) M of length l. It is given that each input sequence contains a variant of M. The variants of interest are sequences that are at a hamming distance of d from M.*"

In solving the PMS problem, traditional string matching are not efficient since these biological motifs are not typically exact but are subject to mutations. As a matter of fact, PMS has already been proven to be NP-hard, which means that it is very unlikely to have an algorithm that solves it in polynomial time (Evans, Smith, & Wareham, 2003).

Some of the terms used in this study are defined below:

- An *l*-mer is a string of length *l* in a DNA sequence of length m where $l < m$.
- The Hamming distance d_H between two *l*-mers, of equal length, is equal to the number of positions where they have mismatches. *Ex.* d_H(acttgca, actaaga) = 3.
- An *l*-mer *x* is considered a *d*-neighbor of another *l*-mer *y* if the Hamming distance

between the two is at most *d*.

• The *d*-neighborhood of an *l*-mer *x* is the set *N*(*x*, *d)* of all *l*-mers with at most *d* Hamming distance from x. i.e., $d_H(x, x') \leq d$. Ex. ccgga, ccaaa, and gctta are all in *N*(cctta, 2), where *l* = 5

• The *d*-neighborhood of a sequence *S* is the set ***N***(*S*, *d*) of all *d*-neighbors of all *l*-mers in sequence *S*. Ex. ***N***(aattacg, 2) = *N*(aatta, 2) ∪ *N*(attac, 2) ∪ *N*(ttacg, 2) where *l* = 5.

In this study we introduce EMS-GT2, an improvement of EMS-GT that is an exact enumerative algorithm for the planted motif problem consisting of two phases -- Generate and Test. The EMS-GT2 algorithm leverages on the block processing of candidate motifs during its Test phase. The original EMS-GT algorithm was previously evaluated using (*l*, *d*)-challenge instances (9, 2), (11, 3), (13, 4), (15, 5) and (17, 6). It was already showed in (Sia, Nabos, & Fernandez Jr., 2015) that the current implementation of EMS-GT is faster than the then state-of-the-art algorithms PMS8 and qPMS9 for all (*l, d*)-challenge instances mentioned except in (17, 6). Even though that EMS-GT2 algorithm can only solve (*l, d*)-instances when *l* ≤ 17, studies has shown that DNA motif lengths are usually around 10 bp (i.e., base pairs, or the number of characters in the string representation) in eukaryotes and 16 bp in prokaryotes (Stewart, Hannenhalli, & Plotkin, 2012). Thus, the EMS-GT2 algorithm is still significant in practical biological applications.

Literature Review

Motif finding has been studied extensively in the previous years. Numerous algorithms have been made for motif finding and for PMS. Each of these algorithms may be categorized as either an approximate algorithm or an exact algorithm. Approximate algorithms, although they are fast, do not guarantee the exact solution all the time. Heuristic algorithms for PMS that perform local search such as Gibbs Sampling (Lawrence et al., 1993), Expectation Maximization (EM) (Bailey, Williams, Misleh, & Li, 2006), Projections (Buhler & Tompa, 2001; Huo, Zhao, Stojkovic, & Liu, 2009) have been previously explored in the literature. Most of these algorithms initially work on a tuple of alignment positions that corresponds to *l*-mers across different string sequences in the dataset. They then iteratively refine the alignment until a certain criteria is met. MEME (Bailey et al., 2006) is a tool for motif finding that implements Expected Maximization. WINNOWER (Pevzner, Sze, et al., 2000) reduces the PMS problem to finding a large clique in a multipartite graph. Instead of looking for the motif directly, the algorithm applies a winnowing technique to remove spurious edges that trims the graph representation, making it easier to find the motif. Other approximate algorithms are MULTIPROFILER (Keich & Pevzner, 2002), PatternBranching, ProfileBranching (Price, Ramabhadran, & Pevzner, 2003) and CONSENSUS (Hertz & Stormo, 1999).

Exact algorithms are not as fast as approximate algorithms but are guaranteed to return the correct answer for any given problem instance. Furthermore, these exact algorithms can be categorized based on their approach in solving the problem. One approach is to generate a common neighborhood out of all $(m - l + 1)^n$ possible positions or *l*-mers from all string sequences. This approach is called sample- driven. Another approach is called pattern-driven that checks from $|\Sigma|^l$ possible *l*-mers which are the motifs over a base alphabet Σ.

Rajasekaran et al. proposed a series of exact algorithms for the (*l, d*)-motif search problem. The algorithm PMS1 (Rajasekaran et al., 2005) is one of these algorithms. PMS1 solves the problem by enumerating the *d*-neighborhood of all the sequences in the dataset and intersects them; the result is a set of motifs. PMSi and PMSP (Davila, Balla, & Rajasekaran,

2006) are algorithms based on PMS1. PMSi improves the memory space requirement of PMS1 by processing only two sequences at a time. PMSP works by generating all the *d*-neighborhood of each *l*-mer in the first sequence and testing each *d*-neighbor if it exists in the remaining sequences. PMSPrune (Davila, Balla, & Rajasekaran, 2007) works very similar to PMP with some improvements. It generates the neighborhood of an *l*-mer using a branch and bound approach and implements a pruning strategy to speedup the testing of *l*-mers. Succeeding algorithms like PMS5 (Dinh, Rajasekaran, & Kundeti, 2011) and PMS6 (Bandyopadhyay, Sahni, & Rajasekaran, 2014) extend the ideas of PMS1 and PMSPrune. PMS5 generates the common neighborhood of three *l*-mers from different sequences at a time and uses Integer Linear Programming for the pruning process. PMS6 only differs from PMS5 in the way it determines the three *l*-mers. Quorum PMS (qPMS) is a generalized version of the (*l, d*)-motif search problem. Instead of finding an *l*-mer that exists in all *n* sequences, it only considers up to *q* where $q \leq n$. We can see that a qPMS problem is equal to PMS when $q = n$. The qPMS7 (Dinh, Rajasekaran, & Davila, 2012) is one algorithm that solves the qPMS problem. Algorithm qPMS7 is a generalized version of qPMSPrune (quorum version of PMSPrune) combined with the pruning strategy of PMS5 algorithm. PMS8 (Nicolae & Rajasekaran, 2014) is an algorithm that combines the sample-driven approach and the pattern-driven approach. First, it chooses *k*-tuple *T* of *l*-mers from *k* different sequences and it makes sure that all *l*-mers in *T* have a common neighbor. Each *l*-mer that belongs to the common neighborhood of the tuple *T* is checked if it appears in the remaining $n - k$ sequences. One of the current state-of-the-art algorithms, qPMS9 (Nicolae & Rajasekaran, 2015), improves the sample-driven approach of PMS8 by prioritizing *l*-mers that are highly distant from those already in the tuple, resulting in a smaller size of common *d*-neighborhood to test and enables the algorithm to process the quorum version of the PMS problem.

Several exact algorithms solve the PMS problem using suffix trees and other related data structures. RISO (Carvalho et al., 2005), RISOTTO (Pisanti, Carvalho, Carvalho, Sagot, & Marsan, 2006), SPELLER (Sagot, 1998) and SMILE are all exact algorithms that use suffix trees. MITRA (Eskin & Pevzner, 2002) improves the excessive memory requirement of sample-driven approach by using a mismatch tree. Two other algorithms that have some similarity with our algorithm are the Voting algorithm and Bit-based algorithm. Voting algorithm (Chin & Leung, 2005) maintains a hash table that tracks the number the occurrence of every possible *l*-mer and makes sure that every *l*-mer is only counted once in each sequence. An *l*-mer is considered a motif if its total occurrences is equal to the total number of sequences in the dataset. Bit-based algorithm (Dasari, Desh, & Zubair, 2010) generates the neighborhood of each sequence and intersects it to get the set of motif. Unlike PMS1, the Bit-based algorithm maps every *l*-mer to its corresponding integer value. It uses an array of size $|\Sigma|^l$ to represent the neighborhood of a sequence and uses the integer representation of an *l*-mer to flag if it is a member of the array. It generates the neighborhood of all sequences and merges it using the logical operator AND. The resulting array represents the set of motif.

The EMS-GT Algorithm

The Exact Motif Search - Generate and Test algorithm (EMS-GT) for the planted motif search problem is composed of two phases, the Generate phase and the Test phase. The Generate phase takes the first *n’* number of string sequences in the dataset and generates the set *d*-neighborhood one sequence at a time then intersects it. This accumulates and outputs the set of candidate motifs C and is composed of *l*-mers that have at least one *d*-neighbor in each of the first *n'* sequences. The Test phase evaluates each candidate motif $c \in C$ by checking if *c* has at least one *d*-neighbor in each of the remaining $n - n'$ string sequences. The phases are formally defined below.

(a) Generate candidates

This step operates on the first *n*' sequences. The intersection of the *d*-neighborhood of each sequence results to the set of candidate motifs *C*.

$$C = \boldsymbol{N}(S_1,d) \cap \boldsymbol{N}(S_2,d) \cap \ldots \cap \boldsymbol{N}(S_{n'},d). \tag{1}$$

(b) Test candidates

Each candidate motif in *C* is evaluated if it has a *d*-neighbor in all of the remaining $n - n'$ string sequences. If a candidate motif passes the test, it is then included in the set of motifs *M*.

Speedup Strategies. To further improve the algorithm performance, we introduced a number of speedup strategies, and these techniques are defined in this section:

1. Integer mapping of *l*-mers**.**

   The *l*-mers are represented using binary representation of integer values. Each character in the *l*-mer is translated using 2 bits (a=00, c=01, g=10, t=11). Ex. actg maps to 00011110 and has an integer value of 30.

2. Bit-based set representation and *l*-mer enumeration.

   The EMS-GT implementation maintains a 4^l array for enumerating all the possible *l*-mer values. The *l*-mer's integer value is used as the index value for the array. It uses the value of 1 if the *l*-mer is a member of the set else it sets the value to 0.

3. *Bit-array compression.*

   To efficiently store these *l*-mers and save memory space, EMS-GT implements an approach that compresses the search space array using integer value bit flags. Instead of one *l*-mer per index value, the implementation can flag up to 32 *l*-mers (since we are using 32-bit integers) per index value. An illustration on how the algorithm accesses the bit flag is provided below:

Ex. gacgt maps to 1000011011 = 539 in decimal.

bit position = 539 mod 32 = 27; *array index* = 539 / 32 = 16;

The bit flag for gacgt is in the 27th least significant bit of the integer at array index 16.

4. *XOR-based Hamming distance computation .*

The mapping of an *l*-mer to its integer value has an additional advantage in computing for mismatch positions. Applying the boolean operator exclusive-or (XOR) between two integer values will return another integer value that contains nonzero value for mismatch positions. Counting this nonzero bit pair positions result to the hamming distance value. An example of this computation is shown below:

Ex. aacgt maps to 0000011011

tacgc maps to 1100011001

XOR produces 1100000010 = 2 mismatches (Note, the mismatches are counted per pair)

5. *Recursive neighborhood generation.*

The Generate step of the algorithm produces the *d*-neighborhood of a string sequence by generating the *d*-neighborhood of all *l*-mers in that sequence. Our implementation of EMS-GT uses a recursive approach for generating the *d*-neighborhood of an *l*-mer. The recursive generation can be visualized by a tree *T*(*x*) of height *d* that is generated in depth-first manner. Each node is a tuple of (*w*, *p*) where *w* is an *l*-mer and *p* corresponds to a position in the *l*-mer $0 \le p \le l$. At a given node (w, p) and $p \neq l$, three children nodes are generated where each node is variant of *w* that has a different character starting at the $p + 1$ position. The root node is $(x, 0)$, and any *l*-mer in nodes at depth *t* has a hamming distance of *t* from the *l*-mer *x* (see Figure 1). Given this, the expected size of $N(x, d)$ can be computed using the equation:

$$|N(x,d)| = \sum_{i-0}^{d} \binom{l}{i} 3^i \tag{2}$$

6. *Block-based optimization for neighborhood generation .*

The way our implementation of the algorithm represents the neighborhood of 4^l bit flags array opens up a new way to improve the generation of neighborhood $N(x,d)$. The algorithm maintains an array of 32-bit integers where each bit represents an *l*-mer. Setting the bit value to 1 means that the *l*-mer is in $N(x, d)$, otherwise 0 means it is not. The improvement generates the neighborhood by blocks of size *k* where $0 < k < l$ instead of per bit. It is observed that dividing the bit array *N* into 4^k consecutive blocks results to blocks conforming to one of the $(k + 2)$ possible block patterns. For each possible *k*-mer, block patterns are pre-generated according to the remaining number of allowed mismatch *d'* where $0 \le d' \le d$. There is no need to pre-generate the block pattern for 0 value since it only corresponds to a block pattern where exactly one *l*-mer is set and the block pattern for *d* value since it corresponds to a block pattern where all bits are set.

Given an *l*-mer *x*, the generation of $N(x, d)$ now divides the *l*-mer to its $(l - k)$-length prefix *y* and its *k*-length suffix *z*. The algorithm then generates all prefixes in $N(x, d)$ by recursively generating $N(y, d)$. For each prefix $y' \in N(y, d)$, a block pattern is applied to the neighborhood array based on the remaining number of allowed mismatch d' computed using $d - d_H(y, y')$ and the suffix z. This is based on the observation that the distance between two *l*-mers is equal to the sum of the distance between their prefixes and the distance between their suffixes. The run time complexity for generating the *d*-neighborhood of *l*-mer x is now reduced to $O(4^k \sum_{i=0}^{l-k} \binom{l-k}{i} 3^i)$.

Methodology

This section states how the EMS-GT2 and the proposed speedup techniques were implemented and evaluated using synthetic datasets for some challenge (*l*, *d*) instances. A study (Nicolae & Rajasekaran, 2015) proposed that (*l*, *d*) instances where *d* is the largest integer value for which the expected number of motifs of length *l* would occur in the input by random chance and does not exceed a constant value (500) are categorized as *challenging instances*. The following (*l*, *d*) instances (9, 2), (11, 3), (13, 4), (15, 5) and (17, 6) are the only challenge instances where *l* is between 9 and 17, and these challenge instances were used for the evaluation of the algorithms.

Datasets

Algorithms that solve PMS (Pevzner et al., 2000; Nicolae & Rajasekaran, 2014, 2015) use a dataset containing 20 string sequences where each nucleotide is in $\Sigma = \{a, c, g, t\}$. Each string sequence is 600 base pairs (bp) long and each nucleotide is randomly generated with equal chance of being selected. A motif is then generated and for each string sequence in the dataset, a *d*-neighbor is planted at a random position. In this study, a generator was run to produce a dataset of PMS problem instances with this configuration, and this dataset was then used to evaluate the algorithms. Furthermore, a converter program was used to translate the dataset into FASTA format in order to execute qPMS9.

Implementation

The EMS-GT2 maintains a 4^l motif search space that is represented by array of bits for space efficiency. The previous speedup technique (used in EMS-GT) exploits this manner of representing the search space by generating the neighborhood of an *l*-mer by blocks instead of per bit. In this study, we took advantage of this block-processing approach in testing of candidate motifs. The Test phase checks if a candidate motif *c* is in the remaining $n – n'$ sequences by comparing if there is at least one *l*-mer in each sequence that is within *d*-distance from *c*. If a candidate motif *x* is eliminated for failing to have a *d*-neighbor in some input sequence S_i, then it is possible to reduce the testing for another candidate motif *y* on the same sequence S_i if *y* is within the same *k*-block as *x*.

Parameter Fine Tuning

The EMS-GT2 defines an integer value *n'* ($1 < n' < n$) that divides the dataset into two set of sequences. The first *n'* sequences are used in the Generate phase while the remaining is assigned to the Test phase. Previous experimentations (Sia et al., 2015) showed that it is efficient for the algorithm to set the value of *n'* to 10. Technically, *n'* dictates how big is the size of the set of candidate motifs *C* to be evaluated if they are in the remaining $n − n'$ sequences. In line with this, we run an experimentation that records the average runtime of the algorithm with the speedup techniques over 5 tests and having different values for *n'*. The values for *n'* range from 5 to 10 in this experiment, since we only want to make the candidate motif set large enough for our speedup technique to take effect.

Table 1 shows the ideal *n'* value for (*l*, *d*)-challenge instances. For (*l*, *d*) instances where $l \leq 11$, the ideal value for *n'* is still 10. This is true due to the efficiency of the Generate phase in small instances of the problem. For (13, 4), (15, 5) and (17, 6), different *n'* were used in the evaluation which are 9, 8 and 7 respectively.

Table 1

Runtime evaluation of the algorithm with speedup over different n' values.

Sequence	**(9, 2)**	**(11, 3)**	**(13, 4)**	**(15, 5)**	**(17, 6)**
5	0.07 s	0.42 s	2.37 s	17.86 s	148.62 s
6	0.06 s	0.29 s	1.54 s	12.64 s	116.01 s
7	0.05 s	0.21 s	1.01 s	10.70 s	**119.94 s**
8	0.05 s	0.18 s	0.84 s	**10.38 s**	119.82 s
9	0.03 s	0.14 s	**0.75 s**	11.01 s	132.10 s
10	**0.03 s**	**0.13 s**	0.76 s	11.90 s	146.10 s

Evaluation

For evaluation of the algorithms, we compared the EMS-GT2 to EMS-GT and the algorithm qPMS9. We used the challenging (*l*, *d*) instances defined in (Davila et al., 2007; Nicolae & Rajasekaran, 2015). The instances used in the evaluation are the following: (9, 2), (11, 3), (13, 4), (15, 5) and (17, 6).

Results

This section describes EMS-GT2 with the two proposed speedup techniques and briefly discusses the observation where the idea for the improvement originated. Runtime evaluation of EMS-GT2 is also discussed in this section.

Faster Candidate Motif Elimination through Block Processing

The EMS-GT algorithm tests candidate motifs in a brute-force approach. A candidate motif is tested by checking if it has at least one *d*-neighbor in each of the remaining $n - n'$

sequences. In testing a candidate motif c, if there is a sequence S_i in the remaining $n - n'$ sequences where c does not have any d-neighbor, then candidate motif c is automatically eliminated.

In our implementation, the search space is represented by a compressed bit array and the l-mers are enumerated alphabetically. L-mers that are near each other do not differ that much. We used this observation in improving the way the algorithm tests the candidate motifs (see Figure 2 for the illustration of the search space).

In EMS-GT, each testing of candidate motif is independent of each other. We propose here a new speedup technique that processes these candidate motifs by blocks. We first partition the search space by blocks containing 4^k l-mers. This results into l-mers that share the same $(l - k)$-prefix characters where $2 < k < l$. Since every row in the search space represents exactly 32 l-mers (32-bit integers), the height of every block is computed using $(4^k/32)$ (see Figure 3 for an example of a partitioned search space). Additionally, any two l-mers within a block has at most k hamming distance value between them.

We process the testing of candidate motifs now by blocks. If candidate motifs x and y are within a block and x has been eliminated as a candidate motif in sequence S_i $(n' \leq i \leq n)$, we can filter out l-mers $z \in S_i$ where $d_H(x,z) > d + k$. We collect the remaining l-mers in S_i and use them for testing the remaining candidate motifs in the block along with the other l-mers in the remaining sequences in $\{S_{n'}, S_{n'+1}, ..., S_n\} \backslash \{S_i\}$. The theorem below formalizes the main property used in this speedup technique.

Theorem 1. *Let x and y be l-mers in a block in the search space containing* 4^k *l-mers. Let d be the number of allowed mutations in the problem instance. Let z be another l-mer. If* $d_H(x, z)$

$> (k + d)$ then $d_H(y, z) > d$, and therefore z is not in $N(y, d)$

Proof. Using proof by contradiction, we first suppose that $d_H(y, z) \leq d$. Since the *l*-mers *x* and *y* belong to the same block, then $d_H(x, y) \leq k$. We use these bounds in the triangle of inequality $d_H(x, z) \leq d_H(x, y) + d_H(y, z)$ to derive the result $d_H(x, z) \leq k + d$. This result contradicts the given condition that $d_H(x, z) > (k + d)$. Hence, we are sure that $d_H(y, z) > d$.

The *k*-value affects the number of *l*-mers that is filtered in a sequence. The lower its value, the larger the number of filtered *l*-mers and faster candidate motif testing will be. But since *k* also affects the number of *l*-mers in a block, the lower its value, the fewer the candidate motifs that might benefit from the speedup technique. In our implementation we use $k = 5$ where every block 32 rows of 32 bit flags representing a total of 4^5 *l*-mers.

Pre-computation of Mismatch Values

The EMS-GT2 uses the hamming distance computation heavily during the Test phase. As discussed earlier for the original EMS-GT, the hamming distance of two binary represented *l*-mers, can be efficiently computed using the boolean operator XOR. In EMS-GT2, instead of repeatedly counting this nonzero pairs of bits every time we compute the hamming distance, we use a pre-computed lookup table to help reduce computational time. A naive pre-computation of these nonzero pair counts for all possible *l*-mers (each represented by $2l$ bit values) will introduce an unacceptable overhead computation time when *l* is sufficiently large, i.e., when $l >= 10$ (based on actual runs on our current machine configurations). A more efficient approach is to pre-compute only up to *l*-mers of length $l' < l$, which requires $b = 2l'$ number of bits. Then we determine the hamming distance by looking up the nonzero counts in the XOR results, *b* number of bits at a time, as described in Algorithm 1. In our experimentation the maximum required bits to represent *l*-mers is 34 bits (for (17, 6)-instance). Given this, we pre-compute up to 18 bits (*l*'=9) values only and use the lookup table twice for the computation of the actual hamming distance between any given pair of *l*-mers.

Performance of EMS-GT with speedup techniques

The EMS-GT2, EMS-GT and qPMS9 were evaluated in terms of actual runtime on an Intel Xeon, 2.10 Ghz machine. The performance of each algorithm was averaged over 20 synthetic datasets for each (*l*, *d*)-challenge instance where $l \leq 17$. Table 2 shows the runtime results between EMS-GT2 vs. EMS-GT while Table 3 shows the runtime results between the EMS-GT2 vs. the qPMS9.

Table 2.

EMS-GT and EMS-GT2 runtime evaluation.

(*l, d*)	EMS-GT	EMS-GT2	% speedup
(9, 2)	0.04 s	0.05 s	-
(11, 3)	0.17 s	0.26 s	-
(13, 4)	1.03 s	0.82 s	20.3%
(15, 5)	12.39 s	10.43 s	15.8%
(17, 6)	143.87 s	111.22 s	22.6%

The additional speedup techniques become more effective as the *l* value in the (*l*,*d*)-instance grows. For every (*l*,*d*)-challenge instances mentioned where $l \geq 13$, EMS-GT2 has improved the runtime over the EMS-GT by at least 15%. Unfortunately, the speedup techniques in EMS-GT2 failed to compensate for their additional overhead computations in both (9, 2) and (11, 3) challenge instances and failed to improve the overall runtime of the implementation. Previous implementations of the original EMS-GT failed to beat qPMS9 in (17, 6)-challenge instance. The proposed EMS-GT2 not only produced improved runtimes but also succeed in beating the qPMS9 in this challenge instance. The improved EMS-GT is now faster than the state-of-the-art qPMS9 in all of the (*l*, *d*)-challenge instances where $l \leq 17$. Even though the implementation of EMS-GT can only run in (*l*, *d*)-challenge instances where $l \leq 17$ because of computer memory constraint, studies have shown that the typical length of motifs is around 10 base pairs (bp) (Stewart et al., 2012) anyway.

Table 3.

EMS-GT2 and qPMS9 runtime evaluation.

(*l, d*)	qPMS9	EMS-GT2	% speedup
(9, 2)	0.60 s	0.05 s	91.6%
(11, 3)	1.26 s	0.26 s	79.3%
(13, 4)	4.58 s	0.82 s	82.0%
(15, 5)	25.73 s	10.43 s	59.4%
(17, 6)	123.17 s	111.22 s	9.7%

Conclusions

We have presented EMS-GT2, an improved exact solution for the planted-motif search problem. EMS-GT2 was able to efficiently test candidate motifs within the same block by filtering out *l*-mers using a property of the search space array that we have discovered and proven in this paper. The previous implementation of EMS-GT already outperforms the state-of-the-art algorithm qPMS9 in (*l*, *d*)-challenge instances (9, 2), (11, 3), (13, 4) and (15, 5) but failed to beat qPMS9 in (17, 6)-challenge instance. EMS-GT2 improved the original algorithm's performance on (13, 4), (15, 5) and (17, 6) and was able to beat qPMS9 in all (*l*, *d*)-challenge instances where $l \leq 17$.

References

Bailey, T. L., Williams, N., Misleh, C., & Li, W. W. (2006). MEME: discovering and analyzing DNA and protein sequence motifs. *Nucleic Acids Research*,*34*(Web Server issue), W369–W373. http://doi.org/10.1093/nar/gkl198. 369-373

Bandyopadhyay, S., Sahni, S., & Rajasekaran, S. (2014). Pms6: a fast algorithm for motif discovery. *International Journal of Bioinformatics Research and Applications*, 10(4-5), 369-383.

Buhler, H., & Tompa, M. (2001). Finding motifs using random projections. *In Proceedings of the fifth annual International Conference on Computational Biology* (pp. 69 – 76). New York, NY, USA: ACM. doi: 10.1145/369133.369172.

Carvalho, A. M., Freitas, A. T., Oliveira, A. L., Rhone-alpes, I., Bernard, U. C., & I, L. (2005). A highly scalable algorithm for the extraction of cis-regulatory regions. *In Proc. APBC`05*. Imperial College Press.

Chin, F. Y., & Leung, H. C. (2005). Voting algorithms for discovering long motifs. *In Proceedings of the third Asia-Pacific Bioinformatics Conference (APBC)* (p. 261-271)

Dasari, N. S., Desh, R., & Zubair, M. (2010). An efficient multicore implementation of planted motif problem. *In High performance computing and simulation (HPCS), 2010 international conference on* (pp. 9–15).

Davila, J., Balla, S., & Rajasekaran, S. (2006). Space and time efficient algorithms for planted motif search. In V. N. Alexan- drov, G. D. van Albada, P. M. A. Sloot, & J. Dongarra (Eds.), Computational Science – ICCS 2006: 6th international conference, reading, uk, may 28-31, 2006. Proceedings, Part II (pp. 822–829). Berlin, Heidelberg: Springer Berlin Heidelberg

Davila, J., Balla, S., & Rajasekaran, S. (2007). Fast and practical algorithms for planted (l, d) motif search. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 4(4), 544–552.

Dinh, H., Rajasekaran, S., & Davila, J. (2012, 07). qPMS7: A fast algorithm for finding, (l,d)-motifs in DNA and Protein Sequences. *PLoS ONE, 7(7),* 1-8.

Dinh,H.,Rajasekaran,S.,& Kundeti,V.K.(2011).Pms5:an efficient exact algorithm for the (l¸d)-motif finding problem. *BMC Bioinformatics*, 12(1), 1–10. doi: 10.1186/1471-2105-12-410

Eskin, E., & Pevzner, P. A. (2002). Finding composite regulatory patterns in DNA sequences. *Bioinformatics, 18* (suppl 1), S354–S363.

Evans, P. A., Smith, A. D., & Wareham, H. (2003). On the complexity of finding common approximate substrings. *Theoretical Computer Science*, 306(1-3), 407 - 430. Retrieved from http://www.sciencedirect.com/science/article/pii/S0304397503003207 doi: http://dx.doi.org/10.1016/S0304-3975(03)00320-7

Hertz, G. Z., & Stormo, G. D. (1999). Identifying DNA and protein patterns with statistically significant alignments of multiple sequences. Bioinformatics, 15(7), 563–577.

Huo, H., Zhao, Z., Stojkovic, V., & Liu, L. (2009). Combining genetic algorithm and random projection strategy for (l, d)-motif discovery. *In Bio-inspired computing*, 2009. BIC-ta'09 (pp. 1–6).

Keich, U., & Pevzner, P. (2002). Finding motifs in the twilight zone. *Bioinformatics*, 18(10), 1374-1381.

Lawrence, C. E., Altschul, S. F., Boguski, M. S., Liu, J. S., Neuwald, A. F., & Wootton, J. C. (1993). Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment. *Science*, 262(5131), 208–214.

Nicolae, M., & Rajasekaran, S. (2014). Efficient sequential and parallel algorithms for planted motif search. *BMC Bioinformatics*, 15(1), 34.

Nicolae, M., & Rajasekaran, S. (2015). qPMS9: An efficient algorithm for quorum Planted Motif Search. *Scientific reports*, 5.

Pevzner, P. A., Sze, S.-H., et al. (2000). Combinatorial approaches to finding subtle signals in DNA sequences. *In ISMB* (Vol. 8, pp. 269–278).

Pisanti, N., Carvalho, A. M., Carvalho, R. M., Sagot, M.-F., & Marsan, L., (2006). RISOTTO: Fast extraction of motifs with mismatches. *In Proceedings of the 7th Latin American Theoretical Informatics Symposium*, 3887 of LNCS:757-768 (pp. 757–768).

Price, A., Ramabhadran, S., & Pevzner, P. A. (2003). Finding subtle motifs by branching from sample strings. *Bioinformatics*, 19(suppl 2), ii149-ii155.

Rajasekaran, S., Balla, S., & Huang, C. (2005). Exact algorithms for planted motif challenge problems. Journal of Computational Biology, 1117-1128.

Ronquillo, M. (2016). (*l, d)*-challenge instances for the Planted Motif Search Problem. Retrieved from *https://www.researchgate.net/publication/305828021_l_d-challenge_instances_for_the_Planted_Motif_Search_Problem* . DOI: 10.13140/RG.2.1.2692.8884

Sagot, M.-F. (1998). Spelling approximate repeated or common motifs using a suffix tree.

Sia, A., Nabos, J., & Fernandez Jr., P. (2015). An Efficient Exact Solution to the (l,d)-planted motif problem. *8th AUN/SEED-Net Regional Conference on Electrical and Electronics Engineering*.

Stewart, A. J., Hannenhalli, S., & Plotkin, J. B. (2012). Why transcription factor binding sites are ten nucleotides long. *Genetics*, 192(3), 973–985.

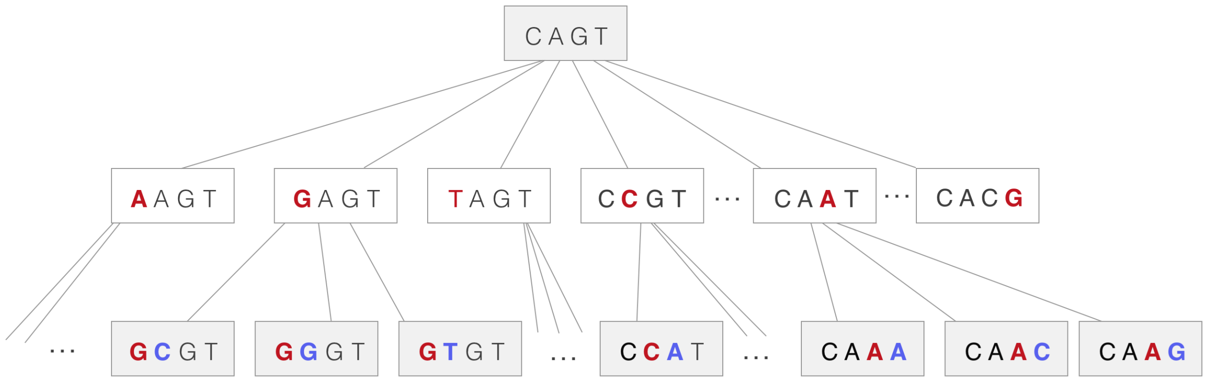

Figure 1: Illustration of the recursive generation of neighborhood.

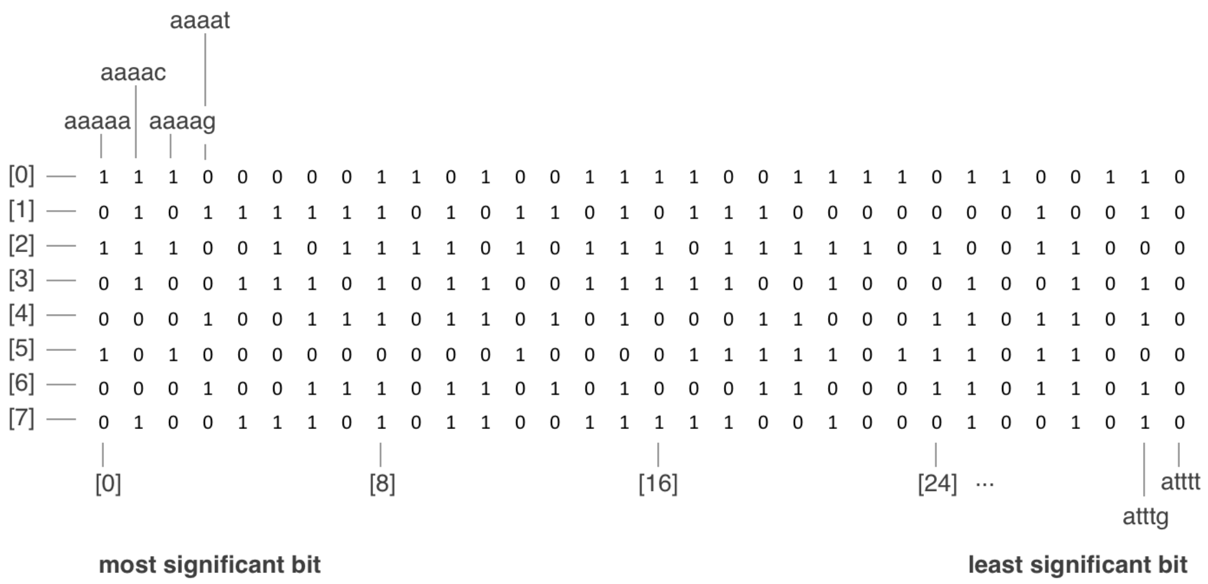

Figure 2. First 8 rows of the 4^5 search space with random flag values.

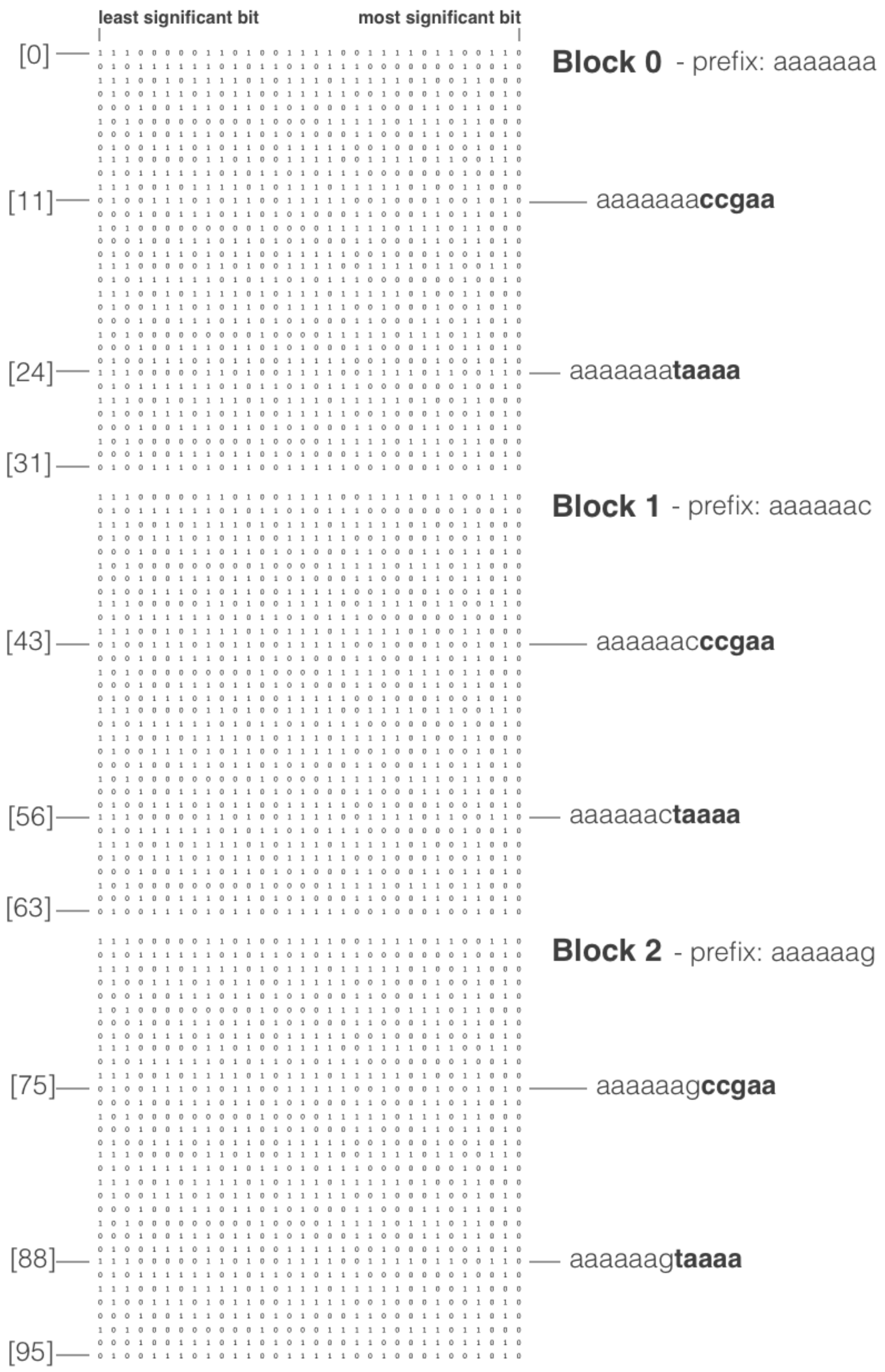

Figure 3. Illustration of the block partitioning of a 4^{12} search space. There are 4^5 *l*-mers in each block and has a height of 32. The illustration shows the first 3 blocks only.

Algorithm 1 Hamming Distance Computation using Pre-Computed Mismatch Values

```
Input: l-mer mappings $u$ and $v$ and
        MC ▷ array of pre-computed count of mismatch positions
Output: Hamming distance $d_H(u, v)$

1: $d_H(u, v) \leftarrow 0$
2: $z \leftarrow u \oplus v$
3: while $z > 0$ do
4:     $l \leftarrow z\&((1 << 18) - 1)$ ▷ process 18 bits at a time
5:     $d_H(u, v) \leftarrow d_H(u, v) + MC[l]$
6:     $z \leftarrow z >>> 18$ ▷ shift 18 bits to the right
7: end while
8: return $d_H(u, v)$
```

Algorithm 1. Shows the improved way of computing the mismatch value for the Hamming distance computation