Improving An Exact Solution to the (l, d)-Planted Motif Problem

A Thesis

Presented to the

Faculty of the Graduate School

Ateneo de Manila University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Maria Clara Isabel D. Sia

2015

The thesis entitled:

Improving An Exact Solution to the (l, d)-Planted Motif Problem

submitted by **Maria Clara Isabel D. Sia** has been examined and is recommended for oral defense.

MARLENE M. DE LEON, Ph.D. Chair	PROCESO L. FERNANDEZ, JR., Ph.D. Adviser

EVANGELINE P. BAUTISTA, Ph.D.
Dean
School of Science and Engineering

The Faculty of the Graduate School of Ateneo de Manila University accepts the THESIS entitled:

Improving An Exact Solution to the (l, d)-Planted Motif Problem

submitted by **Maria Clara Isabel D. Sia** in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science.

ANDREI D. CORONEL, Ph.D.
Member

JULIETA Q. NABOS
Member

JOSE ALFREDO A. DE VERA, Ph.D.
Member

PROCESO L. FERNANDEZ, JR., Ph.D.
Adviser

EVANGELINE P. BAUTISTA, Ph.D.
Dean
School of Science and Engineering

Grade: **Very Good**

Date: 23 October 2015

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 mismatches due to mutation.

This study implements a novel speedup technique for EMS-GT, an exact motif search algorithm which operates on a compact bit-based representation of the search space. Our novel technique takes advantage of distance-related patterns in this representation, in order to speed up the bulk bit-setting operations performed by the algorithm. A Java implementation shows the improved EMS-GT to be highly competitive against PMS8 and qPMS9, two current state-of-the-art exact algorithms. With the speedup technique, EMS-GT outperforms both competitors for challenging (l, d) instances (9,2), (11,3), (13,4) and (15,5) showing runtime reductions from qPMS9 of at least 76%, 81%, 77% and 37% respectively for these instances, while ranking second to qPMS9 for challenge instance (17,6).

TABLE OF CONTENTS

LIST OF FIGURES iii

LIST OF TABLES iv

CHAPTER

I INTRODUCTION 1
1.1 Context of the study 3
1.2 Objectives of the study 5
1.3 Research questions 5
1.4 Significance of the study 6
1.5 Scope and limitations 6

II REVIEW OF RELATED LITERATURE 8
2.1 Heuristic Algorithms 8
2.2 Exact Algorithms 9
2.2.1 PMS8 and qPMS9 11
2.2.2 EMS-GT 14

III METHODOLOGY 22
3.1 Improving EMS-GT 22
3.2 Evaluation 23
3.2.1 Synthetic Datasets 24

IV RESULTS AND ANALYSIS 25
4.1 Block patterns in l-mer neighborhoods 25
4.2 Derivation of patterns based on Hamming distances 27
4.3 Pattern-based speedup technique for EMS-GT 30
4.4 Performance improvement 32
4.5 EMS-GT performance comparison vs. PMS8 and qPMS9 . . . 34

V CONCLUSIONS 37

APPENDIX

BIBLIOGRAPHY 39

A Source code for EMS-GT, with speedup technique 43

LIST OF FIGURES

1.1 Sample instance of the planted motif problem. 1

2.1 Example mismatch tree for MITRA. 10
2.2 Runtime of qPMS9 for various (l, d) 13
2.3 EMS-GT's performance vs PMSPrune, qPMS7 and PMS8. 16
2.4 Bit-array representing an l-mer neighborhood 18
2.5 Number of neighbors in $N(x, d)$ for challenging (l, d) values. 20

4.1 Block bit patterns in an l-mer neighborhood 26
4.2 Block bit patterns vs prefix mismatches 28
4.3 Distance distribution from `tacgt` to all $4^5 = 32 \times 32$ k-suffixes, k=5. 29
4.4 EMS-GT runtimes without (baseline) vs. with speedup technique. . 32
4.5 Number of neighbors in $N(y, d)$ vs. $N(x, d)$ for challenging (l, d), k=5. 33
4.6 Improved EMS-GT's performance vs. PMS8 (baseline) and qPMS9. 35

LIST OF TABLES

4.1 Number of suffix mismatches allowed, given a fixed number of prefix mismatches, between two d-neighbors. 27
4.2 EMS-GT runtimes with speedup technique, using different values of k. Shortest runtimes are in bold text. 34
4.3 Runtimes of PMS8, qPMS9, EMS-GT without and with speedup technique. Shortest times are in bold text. 36
4.4 Runtimes of PMS8, qPMS9, EMS-GT without and with speedup technique on non-challenging (l, d). 36

CHAPTER 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 [3]; 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 repeatedly occurs 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.

Find a 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

Figure 1.1. Sample instance of the planted motif problem.

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* [17]?" 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: two distinct variants of a motif—both counting as valid occurrences of the motif—might differ in as many as $2d$ characters! Brute-force solutions quickly become infeasible as values of l and d increase. All of this shows why (l, d)-motifs are sometimes called "subtle" signals in DNA [17], and why finding them is difficult and computationally expensive. In fact, the motif finding problem has already been shown to be NP-complete [15].

This study is concerned with the EMS-GT (*Exact Motif Search - Generate and Test*) algorithm [14], which solves the planted motif problem for any arbitrary instance up to l=17. The study investigates certain Hamming distance-related patterns appearing in EMS-GT's bit-based representation of the motif search space, and uses these to develop a novel speedup technique for EMS-GT. It then evaluates how this speedup technique improves the performance of EMS-GT, and how the improved EMS-GT compares to current state-of-the-art algorithms PMS8 and qPMS9, on challenging instances of the (l, d) planted motif problem.

1.1 Context of the study

This section formally defines the planted motif problem. It also defines key terms used throughout this paper in discussing exact motif-search algorithms.

DEFINITION 1. *l*-mer

An ***l*-mer** is any sequence of length l. In the context of the study, we consider only l-mers over the DNA nucleic alphabet, Σ = {a, c, g, t}. Formally, an l-mer x is an element of Σ^l, whose size is denoted by $|x| = l$. Given a sequence S of length $L > l$, we denote the set of all l-mers in S as $\mathcal{L}(S_i, l)$. The i^{th} l-mer in S is the l-mer that starts at the i^{th} position.

Ex. If l = 5, $\mathcal{L}(\texttt{gattaca}, l) = \{\texttt{gatta}, \texttt{attac}, \texttt{ttaca}\}$;

attac is the second l-mer in gattaca.

DEFINITION 2. Hamming distance

The **Hamming distance *dH*** between two l-mers of equal length is the number of corresponding characters that differ between them. Formally, the Hamming distance between x_1 and x_2 is given by $dH(x_1, x_2) = |\{i \mid x_1[i] \neq x_2[i], 1 \leq i \leq l\}|$, where $x[i]$ is the i^{th} character in a given l-mer x.

Ex. $dH(\underline{\texttt{g}}\texttt{atta}\underline{\texttt{c}}\texttt{a}, \underline{\texttt{cg}}\texttt{tta}\underline{\texttt{g}}\texttt{a}) = 3$.

"Distance" refers to Hamming distance in this paper, unless otherwise stated.

DEFINITION 3. d-neighbor and d-neighborhood

A ***d*-neighbor** x' of an l-mer x is an l-mer whose Hamming distance from x is at most d, i.e., $\boldsymbol{dH(x, x') \leq d}$.

Ex. cgttaga is considered a d-neighbor of gattaca for any $d \geq 3$.

The ***d*-neighborhood of an *l*-mer** $\boldsymbol{x}$ is the set $\boldsymbol{N(x, d)}$ of all d-neighbors of x: $N(x, d) = \{x' \mid dH(x, x') \leq d\}$. Note that x is always included in $N(x, d)$ for any d.

Ex. $N(\texttt{gatta}, 1) =$ { gatta, aatta, catta, tatta,
gctta, ggtta, gttta, gaata, gacta, gagta,
gataa, gatca, gatga, gattc, gattg, gattt }.

Meanwhile, for a given l, the ***d*-neighborhood of a sequence** $\boldsymbol{S}$ of length $L > l$ is the set $\boldsymbol{\mathcal{N}(S, d)}$ of all d-neighbors of all the l-mers in S.

Ex. For l = 5, $\mathcal{N}(\texttt{gattaca}, 2) = N(\texttt{gatta}, 2) \cup N(\texttt{attac}, 2) \cup N(\texttt{ttaca}, 2)$.

DEFINITION 4. (l, d) Planted Motif Problem

INSTANCE: A motif length l, an allowable distance d, and a set $\mathcal{S} = \{S_1, ..., S_n\}$ of n DNA sequences of length $L > l$ over the alphabet Σ = {a, c, g, t}.

SOLUTION: The set $M = \{\ l\text{-mer } x \mid \forall i \in \{1, ..., n\}\ \ \exists x_i \in \mathcal{L}(S_i, l),\ \ dH(x, x_i) \leq d\}$ of motifs occurring with at most d mismatches in all sequences in $\mathcal{S}$.

1.2 Objectives of the study

The main objective of this study is to improve the performance of the EMS-GT algorithm. Specifically, it aims:

1. To develop a speedup technique for EMS-GT that takes advantage of distance-related patterns in the motif search space.

2. To evaluate the speedup technique with regard to improvement in runtime.

3. To evaluate the improved version of EMS-GT against state-of-the-art motif search algorithms.

1.3 Research questions

This study aims to answer the question: How can the performance of the EMS-GT algorithm be improved? Specifically, it aims to answer the following:

1. How can distance-related patterns observed within the motif search space be exploited in a speedup technique for EMS-GT?

2. What performance improvement does a pattern-based speedup technique produce with regard to runtime?

3. How does the improved version of EMS-GT compare with state-of-the-art motif search algorithms?

1.4 Significance of the study

Motif finding in DNA and other types of nucleotide sequences is an important task in bioinformatics. Genome analysis requires fast, efficient algorithms to identify biological motifs which may be linked to protein synthesis, gene function, or even disease and targets for medical treatment.

Improving the EMS-GT algorithm, which was shown to be competitive with the state-of-the-art, results in an even faster option for real-world motif finding applications. Furthermore, this study's investigation and insights regarding distance-related patterns within an organized search space may prove applicable to other types of search tasks, pattern-matching tasks, and problems involving Hamming distances.

1.5 Scope and limitations

This study is concerned with developing and integrating a novel bit-masking speedup technique into the existing Java implementation of EMS-GT. The improved version is benchmarked by its runtime on synthetic datasets for challenging instances of the problem. (EMS-GT had been previously tested for correctness using real biological data with known motifs; re-testing would be redundant since the speedup technique does not change the logic of EMS-GT's algorithm.)

Furthermore, the performance of EMS-GT is compared to that of PMS8 and qPMS9 running on a single core. Although both competitor algorithms are also capable of using multiple procesors, parallelization is beyond the current scope of the speedup techniques explored for EMS-GT.

CHAPTER II

REVIEW OF RELATED LITERATURE

Motif finding is a well-studied problem in computing. Various motif search algorithms have been developed, falling into two categories: *heuristic* and *exact*. This section gives an overview of algorithms of both types, and provides an in-depth description of the exact algorithm EMS-GT.

2.1 Heuristic Algorithms

Heuristic algorithms perform an iterative local search, for instance by repeatedly refining an input sampling or projection until a motif is found.

Gibbs sampling [12] and Expectation Maximization (EM), used in the motif-finding tool MEME [13, 1] both use probabilistic computations to improve an initial random alignment. (An alignment is simply a vector $(a_1, a_2, ..., a_n)$ of n positions, which predicts that the motif occurs at position a_i in the given sequence S_i.) Gibbs sampling attempts to refine the alignment one position at a time; EM may recompute the entire alignment in a single iteration.

Projection [2] 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 [10] uses a random version of the projec-

tion strategy, in tandem with the iteratively self-correcting Genetic Algorithm (GA), for yet another iterative approach. Other successful heuristic algorithms for motif finding include Pattern Branching [18], ProfileBranching [18], MULTIPROFILER [11], NestedMICA [6], and CONSENSUS [8]. There is also the "ensemble" motif discovery algorithm EMD [9], which combines the best predictions from five component heuristic algorithms to achieve a 22.4% improvement in prediction accuracy (versus the best standalone component).

2.2 Exact Algorithms

While heuristics can be efficient, they are non-exhaustive approaches, and do not always guarantee finding a solution. Exact motif search algorithms, on the other hand, perform an exhaustive search of all possible motifs and thus always find the planted motif.

WINNOWER [17] and its successor MITRA [7] 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 possibly 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 l-mer pairs that could indicate a motif.

MITRA refines this graph representation into a mismatch tree which contains all possible l-mers, organized by prefix.

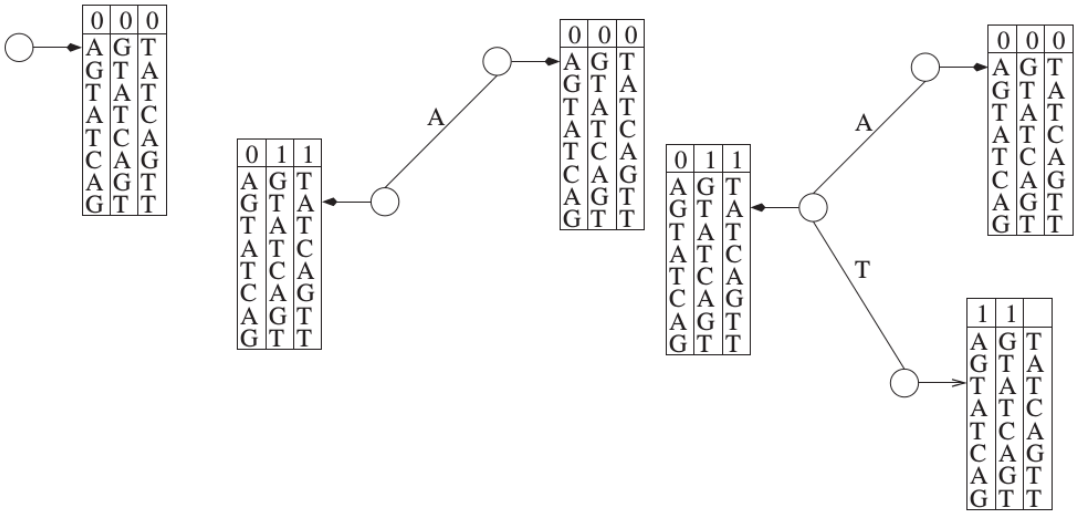


Figure 2.1. The (8,1) mismatch tree for {agtatcag, gtatcgat, and tatcagtt}. *Left*: the tree, in its initial state. *Center*: the tree, after traversing prefix --a: the second and third l-mers now each have a single mismatch with the prefix. *Right*: the tree, after traversing prefix --a--t: the third l-mer now has two mismatches with the prefix, which is greater than the allowable 1. The third l-mer will thus be excluded further down the tree. Image from [7]
.

The tree structure allows MITRA to eliminate entire branches at a time, making it significantly faster than WINNOWER at removing the many spurious edges that are not part of any motif clique.

The BitBased algorithm [4] is another exact algorithm which uses efficiency approaches similar to EMS-GT's (see subsection 2.2.2): it maps l-mers to binary strings of length $2l$, and represents the motif search space with an array

of bits. The main difference is that BitBased is optimized for parallel computation on multiple cores, requiring specialized GPU hardware (Nvidia Tesla C1060 or S1070). BitBased can solve the challenge instance (21,8) in 1.1 hours.

2.2.1 PMS8 and qPMS9

The current state-of-the-art exact algorithms are PMS8 and qPMS9, from the Panoptic Motif Search (PMS) series developed by Rajasekaran et al [5, 15, 16]. The idea in both PMS8 and qPMS9 is to first form all possible n'-tuples of "similar" l-mers, taking one l-mer from each of a subset $\{S_1, ..., S'_n\} \subset \mathcal{S}$ of the given DNA sequences (see requirements for similarity in step 1); then, for each tuple, do a brute-force search to determine whether the common neighbors of the member l-mers are motifs. This approach proceeds in two main steps:

1. *Sample-driven step*

   This step chooses an n'-tuple T of "similar" l-mers, in which the i^{th} element is an l-mer in S_i. Similarity means that the l-mers in the tuple have a common neighbor; this implies that the distance between any two l-mers in T must not be more than $2d$.

$$T = (x_1, x_2, ..., x_{n'}),\ x_i \in \mathcal{L}(S_i, l) \quad dH(x_i, x_j) \leq 2d\ \forall\ i, j \tag{2.1}$$

2. *Pattern-driven step*

   This step intersects the d-neighborhoods of $x_1, x_2, ..., x_{n'}$ to form the set C of their common neighbors. It then checks each l-mer c in C, to determine whether a d-neighbor of c appears in all of the $n - n'$ remaining sequences (i.e., those sequences which did not initially contribute an l-mer to T). If this is the case, c is accepted as a motif.

$$C = N(x_1, d) \cap N(x_2, d) \cap ... \cap N(x_{n'}, d). \tag{2.2}$$

$$M = \{c \in C \mid \forall i \in \{n' + 1, ..., n\} \; \exists \, x_i \in \mathcal{L}(S_i, l), \quad dH(c, x_i) \leq d\}. \tag{2.3}$$

Exhaustively building all "similar" n'-tuples requires significant runtime and many false starts (i.e., the algorithm has built a tuple to size $m < n'$, only to find that no further similar l-mers can be found). PMS8 reduces this by improving the search for additional l-mers, using stricter pruning conditions for similarity between *three* l-mers (two from the tuple, and the one to be added). This method of testing similarity for l-mer triples, instead of pairs, quickly recognizes and discards false starts. For even greater efficiency, qPMS9 intelligently prioritizes adding l-mers that are highly distant from those already in the tuple, such that common neighborhood becomes smaller and faster to check through.

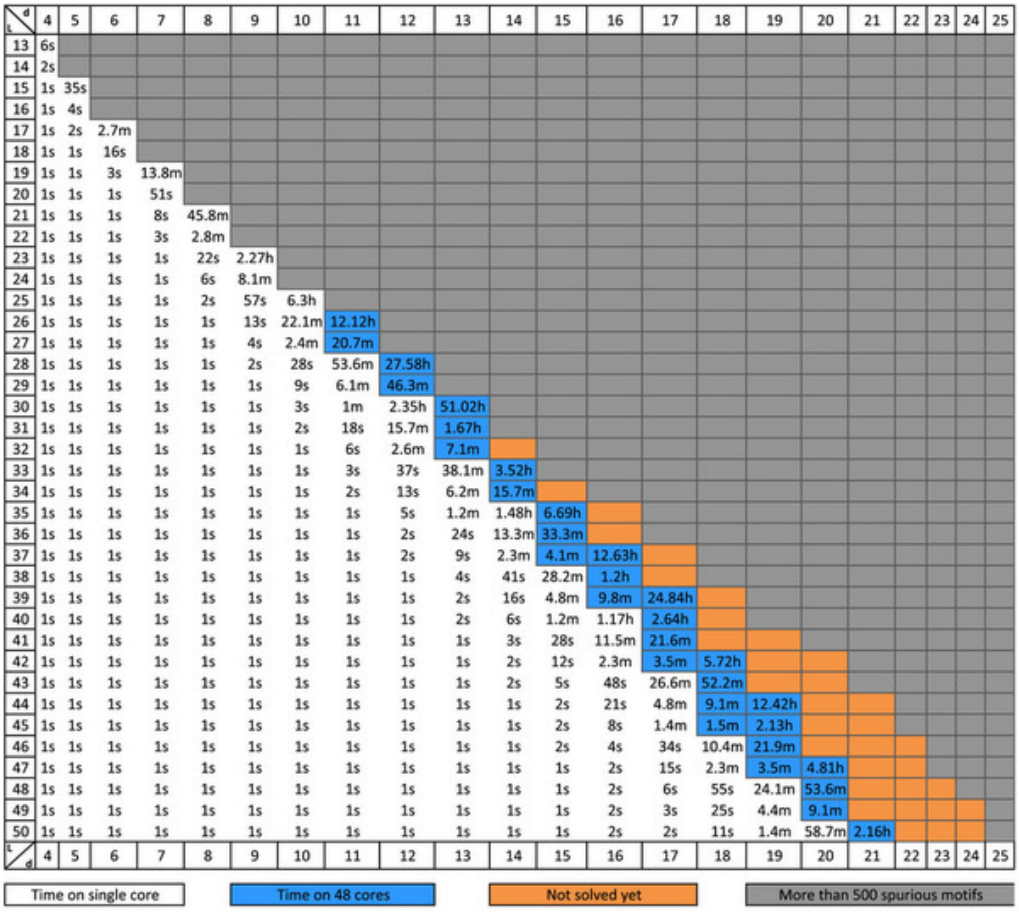


Figure 2.2. Runtime of qPMS9 on DNA datasets for various (l, d) up to (50, 21). Blue indicates the program used 48 cores; white indicates single-core execution. Gray cells represent instances with over 500 spurious motifs. Instances in orange could not be solved efficiently. Image from [16].

In practice, both PMS8 and qPMS9 have been implemented to run on multiple processors with the OpenMPI C library, allowing them to solve highly challenging problem instances with (l, d) as large as (50, 21) in a couple of hours. Note also that most non-challenging instances require trivial (< 1 s) runtime to solve.

2.2.2 EMS-GT

EMS-GT, first developed by Nabos [14], is an exact motif search algorithm based on the candidate generate-and-test principle. It operates on a compact bit-based representation of the search space. Similar to PMS8 and qPMS9, the main idea of EMS-GT is to narrow down the search space to a small set of "candidate" motifs based on the first n' sequences, then to do a brute-force search for neighbors of each candidate in the remaining $(n - n')$ sequences to confirm whether or not the candidate is a motif. EMS-GT's approach proceeds in two main steps:

1. *Generate candidates*

   This step intersects 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.

$$C = \mathcal{N}(S_1, d) \cap \mathcal{N}(S_2, d) \cap ... \cap \mathcal{N}(S_{n'}, d). \tag{2.4}$$

2. *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.

$$M = \{c \in C \mid \forall i \in \{n' + 1, ..., n\} \; \exists \; x_i \in \mathcal{L}(S_i, l), \;\; dH(c, x_i) \leq d\}. \tag{2.5}$$

Algorithm 2.1 EXACT MOTIF SEARCH - GENERATE AND TEST

Input: set $S = \{S_1, S_2, ..., S_n\}$ of L-length sequences,
motif length l, allowable distance d

Output: set M of motifs

```
1: N(S_1, d) ← {}
2: for j ← 1 to L−l+1 do                          ▷ generate candidates
3:     x ← j^th l-mer in S_1
4:     N(S_1, d) ← N(S_1, d) ∪ N(x, d)
5: end for
6: C ← N(S_1, d)
7: for i ← 2 to n' do
8:     N(S_i, d) ← {}
9:     for j ← 1 to L − l + 1 do
10:        x ← j^th l-mer in S_1
11:        N(S_i, d) ← N(S_i, d) ∪ N(x, d)
12:    end for
13:    C ← C ∩ N(S_i, d)
14: end for
15: M ← {}
16: for each l-mer c in C do                      ▷ test candidates
17:    isMotif ← true
18:    for i ← (n' + 1) to n do
19:        found ← false
20:        for j ← 1 to L − l + 1 do
21:            x ← j^th l−mer in S_i
22:            if dH(x, c) ≤ d then
23:                found ← true
24:                break
25:            end if
26:        end for
27:        if !found then
28:            isMotif ← false
29:            break
30:        end if
31:    end for
32:    if isMotif then
33:        M ← M ∪ c
34:    end if
35: end for
36: return M
```

EMS-GT has already proven competitive against other motif search algorithms. Figure 2.3 shows that EMS-GT beats PMSPrune [5] and qPMS7 [5] for all tested (l, d), while also beating PMS8 [15]—which, at the time EMS-GT was developed, was the state-of-the-art—for (l, d) values (13,4) and (14,5).

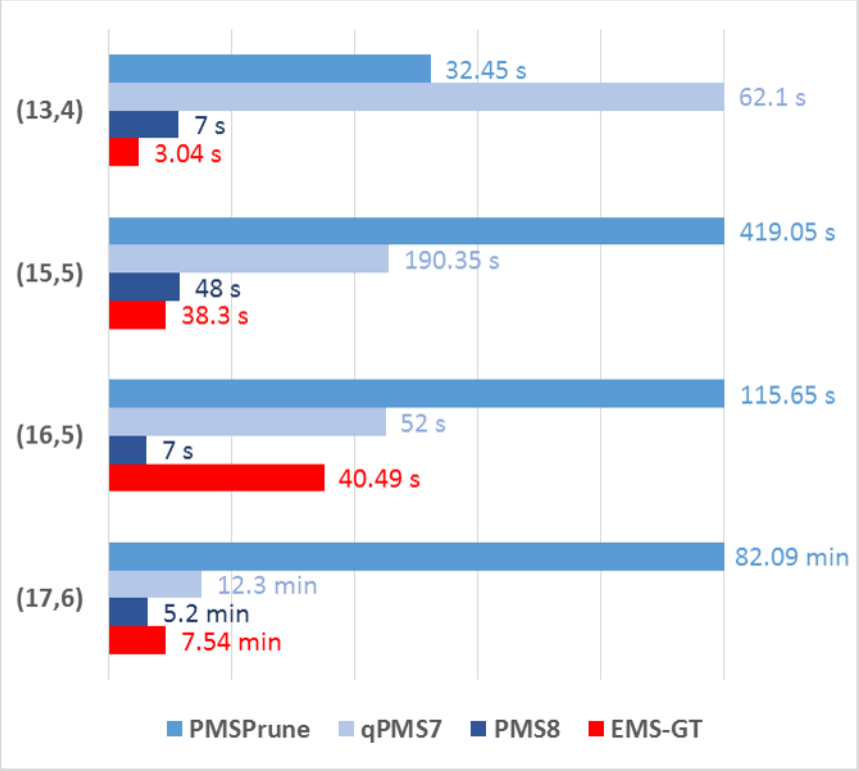


Figure 2.3. EMS-GT's performance vs PMSPrune, qPMS7 and PMS8.

EMS-GT achieves this competitive speed by efficiently performing operations on an array of bits representing the entire motif search space. The succeeding sections discuss EMS-GT's efficiency strategies for representing sets in the search space, determining whether l-mers are neighbors, and generating all possible d-neighbors of a given l-mer.

Bit-based set representation

The motif search space consists of the 4^l possible l-mers that can be formed from the nucleic alphabet $\Sigma = \{\texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}\}$. 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 x to a bit flag index by replacing each character in x with 2 bits (`a=00`, `c=01`, `g=10`, `t=11`). This results in an alphabetical enumeration of l-mers: i.e. for l=4, l-mer 0 is `aaaa`, 1 is `aaac`, 2 is `aaag`, and so on.

Ex. `cgca` maps to `01100100` = 100; thus, its flag is the bit at index 100.

Bit-array compression

EMS-GT's implementation compresses the required set-representation array of 4^l bits into an equivalent array of $\frac{4^l}{32}$ 32-bit integers. The bit for l-mer x is now found at position (x mod 32) of the integer at array index $\lfloor \frac{x}{32} \rfloor$.

Ex. `cgca` maps to `01100100` = 100 in decimal.

array index = $\lfloor \frac{100}{32} \rfloor$ = 3 , *bit position* = 100 mod 32 = 4;

Thus, the flag for `tacgt` is the bit at index 4 of the integer at array index 3.

Note that EMS-GT can only solve problem instances with $l \leq 17$; 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.

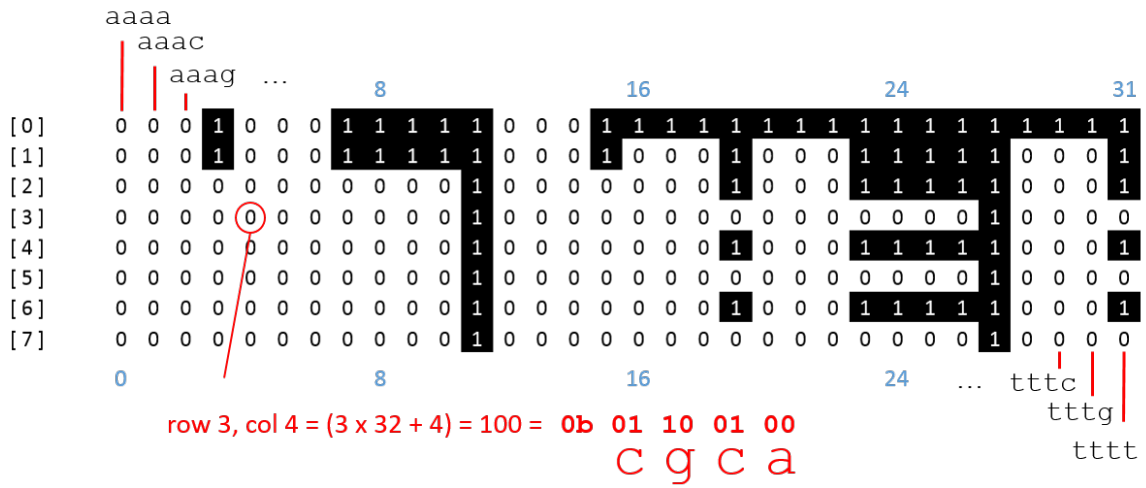


Figure 2.4. Bit-array $N_{\texttt{acgt}}$ representing $N(\texttt{acgt}, 2)$. If l=4, all possible l-mers are represented by $4^4 = 256$ bit flags, compressed here into eight 32-bit integers. A bit is set to 1 if its corresponding l-mer has 2 or fewer mismatches with `acgt`.

XOR-based Hamming Distance Computation

The mapping of l-mers to binary numbers is also useful for computing Hamming distances. 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. See Algorithm 2.2 for the implementation of this procedure.

Ex. `tacgt` maps to `1100011011`

`ttcgg` maps to `1111011010`

XOR produces `0011000001` = 2 mismatches.

Algorithm 2.2 HAMMING DISTANCE COMPUTATION

Input: l-mer mappings u and v
Output: $dH(u, v)$

$dH(u, v) = 0$
$z \leftarrow u$ XOR v
for $i \leftarrow 1$ to l **do**
    if $(z$ AND $3) \neq 0$ **then**
        $dH(u, v) \leftarrow dH(u, v) + 1$
    end if
    $z \leftarrow z \gg 2$ ▷ *shift two bits to the right*
end for
return $dH(u, v)$

Algorithm 2.3 RECURSIVE NEIGHBORHOOD GENERATION

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

$\mathcal{N}[lmer] \leftarrow 0, \ \forall\ lmer \in$ search space
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, c, g, t} - $\{x[i]\}$ ▷ *remove i^{th} character of x*
        for $j \leftarrow 1$ to $|\Sigma'|$ **do**
            $neighbor \leftarrow concatenate(x[1...(i-1)], \Sigma_j, x[(i+1)...l])$
            $\mathcal{N}[neighbor] \leftarrow 1$ ▷ *set $neighbor$'s bit flag to 1*
            if $d > 1$ and $i < l$ **then**
                ADDNEIGHBORS($neighbor$, $i + 1$, $d - 1$)
            end if
        end for
    end for
end procedure
return $\mathcal{N}$

Recursive Neighborhood Generation

To generate a d-neighbor of an l-mer x, choose $d' \leq d$ positions from 1, 2,..., l and change the character in x at each of the d' chosen positions. EMS-GT uses a recursive procedure (Algorithm 2.3) to do this, effectively (1) traversing the tree of all d-neighbors and (2) setting the bit flag in the bit-array N_x for each neighbor encountered. Since we change up to d positions in x, and have 3 alternative characters at each position, the size of the neighborhood $N(x, d)$ is given by:

$$|N(x,d)| = \sum_{i=0}^{d} \binom{l}{i} 3^i \tag{2.6}$$

As Figure 2.5 shows, $|N(x, d)|$ grows very quickly—exponentially—with (l, d).

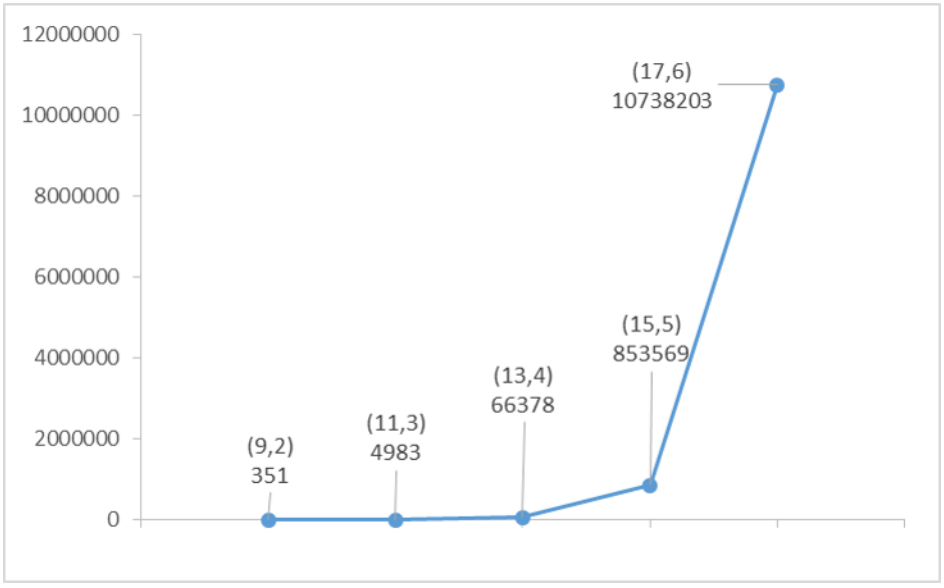


Figure 2.5. Number of neighbors in $N(x, d)$ for challenging (l, d) values.

The exponential growth of $N(x, d)$ means that as l and d increase, the EMS-GT algorithm must spend more time locating and setting bit flags for individual neighbors. This makes the neighborhood-generating portion of EMS-GT—currently characterized by scattered array accesses and numerous bitwise operations—a prime target for further efficiency strategies or speedup techniques.

CHAPTER III

METHODOLOGY

This section briefly describes how a novel speedup technique for EMS-GT was explored and implemented. It then describes the procedure for evaluating the improved version of the EMS-GT algorithm.

3.1 Improving EMS-GT

The Java implementation of EMS-GT operates on a compact, bit-based enumerative representation of the motif search space. Since a significant part of runtime is spent finding and setting bits in this bit-based representation, speedup techniques were explored for the bit-setting portion of the algorithm.

Snapshots of EMS-GT's main data structure (an array of 4^l bit flags representing the entire search space) were taken at various times during program execution, and examined for features that might allow for a more efficient algorithm. It became clear that l-mer neighborhoods, when represented in this data structure, were made up of repeating block patterns; we examined the underlying distribution of Hamming distances in these blocks to explain the patterns.

By finding connections to (1) EMS-GT's alphabetical l-mer enumeration scheme and (2) the additive property of Hamming distances, we were able to

justify the original observation, which was: "A bit-array N_x representing the neighborhood $N(x, d)$ can be partitioned into consecutive blocks of 4^k bits each, where each block will conform to one of at most $(k + 2)$ patterns." The full explanation of block patterns in N is written out in Section 4.1.

Working forwards, we then developed a procedure which can quickly build any neighborhood bit-array N_x in blocks, referring to a pre-generated lookup table of the possible block patterns. This technique was integrated into the Java implementation of EMS-GT.

3.2 Evaluation

The improved version of EMS-GT was compared to the original EMS-GT algorithm, as well as to the 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 [16]. The specific challenge instances used were (9,2), (11,3), (13,4), (15,5), and (17,6), as identified in [16, 5]. The runtimes shown for EMS-GT and competitor algorithms are average runtimes over 20 synthetic datasets per (l, d) instance.

3.2.1 Synthetic Datasets

Synthetic datasets were created using a DNA sequence generator written in Java. Each nucleotide character in a sequence is randomly generated; {a, c, g, 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 [17] every dataset contains 20 DNA sequences each 600 bases long, with an (l, d) motif planted exactly once in each sequence.

CHAPTER IV

RESULTS AND ANALYSIS

This section derives the distance-related patterns observed in an l-mer neighborhood (represented with a 4^l-bit array) in EMS-GT. It then describes how a speedup technique for EMS-GT was developed based on these patterns. Finally, it compares EMS-GT performance with and without the speedup technique, and compares the performance of improved EMS-GT against state-of-the-art algorithms PMS8 and qPMS9.

4.1 Block patterns in l-mer neighborhoods

We can represent the neighborhood of l-mer x as an array N_x of 4^l bit flags, set to 1 if the corresponding l-mer is a neighbor and 0 otherwise.

$$N_x[\, x' \,] = \begin{cases} 1 & \text{if } d_H(x, x') \leq d, \\ 0 & \text{otherwise.} \end{cases} \quad \text{for any } l\text{-mer } x'. \tag{4.1}$$

We can divide our l-mer x into its **prefix** $\boldsymbol{y}$ (the first $l - k$ characters) and its $\boldsymbol{k}$**-suffix** $\boldsymbol{z}$ (the last k characters). We use the notation $x = yz$.

Ex. For $k = 5$, x = `acgtacgtacgt` $\rightarrow$ y = `acgtacg` and z = `tacgt`.

If we partition N_x into blocks of 4^k bits each, for some $k < l$, the 4^k l-mers represented in each block will all start with the same **block prefix** and all have different k-suffixes. This is because N_x represents l-mers in alphabetical order.

Ex. Blocks in N_x for x = `acgtacgtacgt`, k = 5:

Block 0:	bit flags for `aaaaaaaaaaaa` - `aaaaaaattttt`
Block 1:	bit flags for `aaaaaacaaaaa` - `aaaaaactttt`
	...
Block 1,734:	bit flags for `acgtacgaaaaa` - `acgtacgttttt`
	...
Block 16,833:	bit flags for `ttttttgaaaaa` - `tttttgttttt`
Block 16,834:	bit flags for `tttttttaaaaa` - `tttttttttttt`

Each such block in N_x will also conform to one of $(k + 2)$ bit patterns.

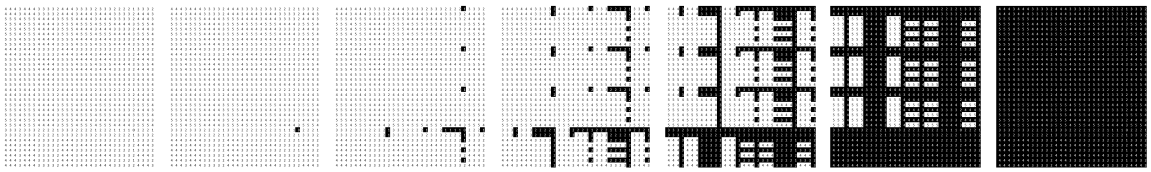


Figure 4.1. Bit patterns followed by blocks in the bit-array representation of $N(\texttt{acgtacgtacgt}, 5)$. Black signifies a bit set to 1.

If we can derive these pattterns, we will be able to build N_x in blocks, instead of setting bits one by one as EMS-GT currently does. The succeeding section presents a derivation based on the additive property of Hamming distances.

4.2 Derivation of patterns based on Hamming distances

Since Hamming distances count mismatches in corresponding characters, the distance between $x = yz$ and another l-mer $x' = y'z'$ is the sum of the mismatches between their prefixes and the mismatches between their k-suffixes, or:

$$d_H(x, x') = d_H(y, y') + d_H(z, z') \tag{4.2}$$

Given Equations (4.1) and (4.2), we can redefine N_x as:

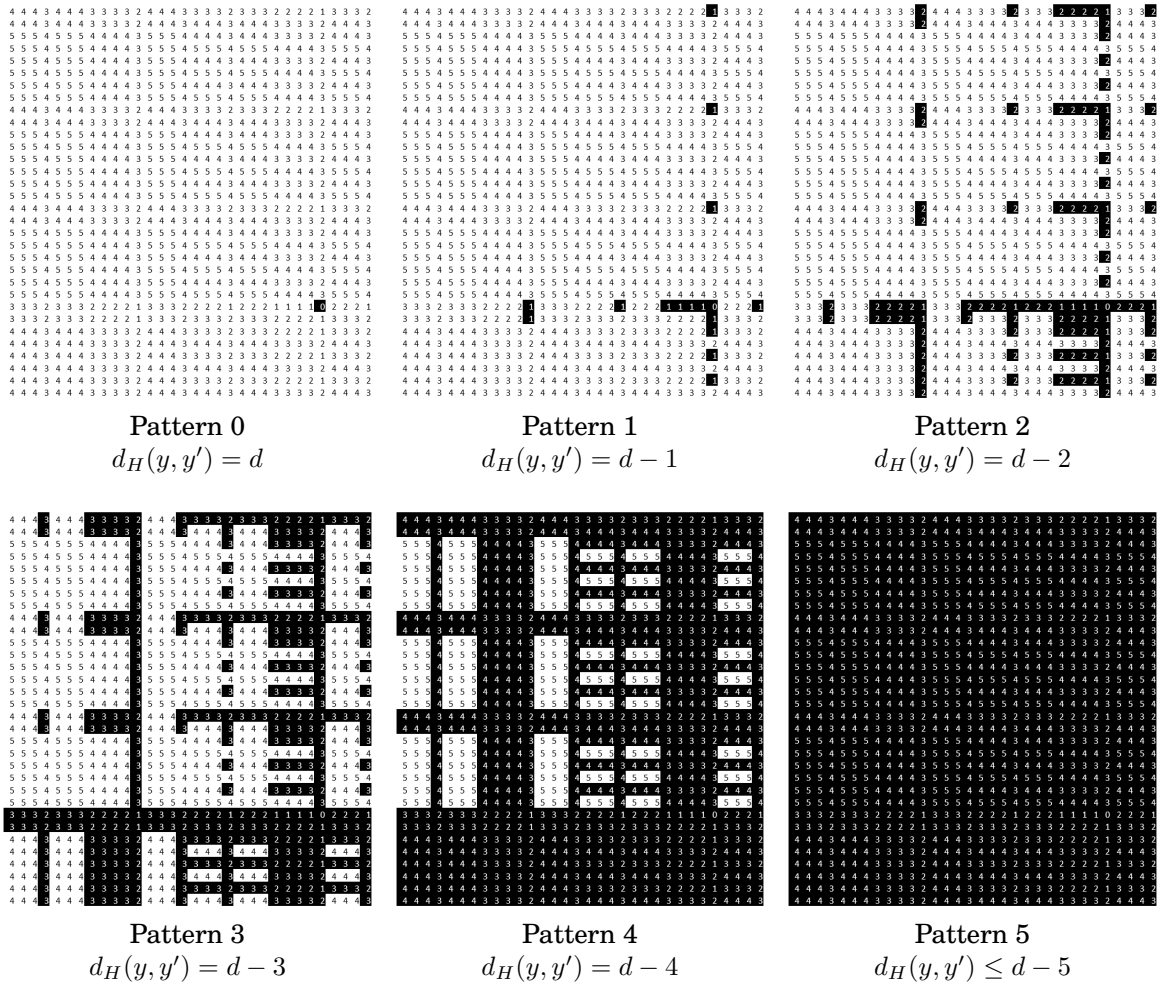
$$N_x[\, x' \,] = \begin{cases} 1 & \text{if } d_H(y, y') + d_H(z, z') \leq d, \\ 0 & \text{otherwise.} \end{cases} \quad \text{for } x' = y'z'. \tag{4.3}$$

Intuitively, if x and x' are neighbors, and there are $\boldsymbol{d_H(y, y')}$ **prefix mismatches** between them, we can allow $\boldsymbol{d_H(z, z') \leq d - d_H(y, y')}$ $\boldsymbol{k}$**-suffix mismatches** for x and x' to have d or fewer total mismatches. Table 4.1 shows the $(k+2)$ cases for distributing d allowable mismatches between prefix and k-suffix.

Case	prefix mismatches	suffix mismatches allowed
-1	more than d	–
0	d	0
1	d - 1	0, 1
2	d - 2	0, 1, 2
...	...	...
k-1	d - (k-1)	0, 1, 2, ..., (k-1)
k	$d - k$ or less	0, 1, 2, ..., (k-1), k

Table 4.1. Number of suffix mismatches allowed, given a fixed number of prefix mismatches, between two d-neighbors.

The $(k+2)$ cases shown in Table 4.1 correspond to the $(k+2)$ bit patterns followed by the blocks in N_x. A block with prefix y' will follow Pattern d_z = d - $d_H(y, y')$:



Pattern 0
$d_H(y, y') = d$

Pattern 1
$d_H(y, y') = d - 1$

Pattern 2
$d_H(y, y') = d - 2$

Pattern 3
$d_H(y, y') = d - 3$

Pattern 4
$d_H(y, y') = d - 4$

Pattern 5
$d_H(y, y') \leq d - 5$

Figure 4.2. Correspondence between the number of prefix mismatches $d_H(y, y')$ and block bit patterns for $N(\texttt{acgtacgtacgt}, 5)$. Black signifies a bit set to 1.

Pattern -1 (the "empty" block pattern, in which no bits are set) corresponds to the case wherein $d_H(y, y') > d$—i.e., the mismatches in the prefix alone already exceed the allowable mismatches for a neighbor of x.

A block pattern is an array of 4^k bits. Given x's suffix z, and the number of allowed suffix mismatches $d_z = d - d_H(y, y')$, we can define a block pattern as:

$$\text{Pattern}(z, d_z)[\ z'\ ] = \begin{cases} 1 & \text{if } d_H(z, z') \le d_z, \\ 0 & \text{otherwise.} \end{cases} \quad \text{for any } k\text{-suffix } z'. \tag{4.4}$$

In other words, the polarity (0 or 1) of a bit in a pattern is determined by its corresponding suffix's mismatches with z. To visualize this, compare Figure 4.3—the color map of $d_H(\texttt{tacgt}, z')$ values for all possible k-suffixes z'—with the bit patterns shown in Figure 4.2.

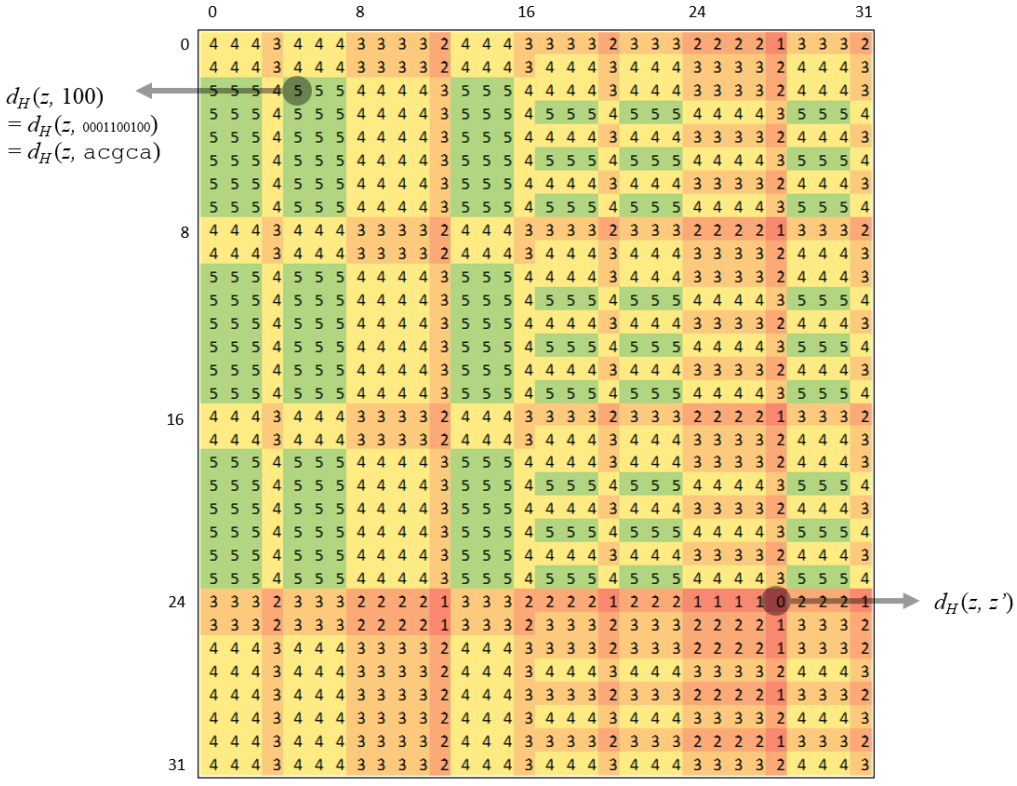


Figure 4.3. Distance distribution from `tacgt` to all $4^5 = 32 \times 32$ k-suffixes, k=5.

4.3 Pattern-based speedup technique for EMS-GT

The previous section has shown that N_x, the bit-array representing the neighborhood $N(x, d)$ of l-mer x, can be built in blocks. We exploit this fact to speed up a key time-consuming task in EMS-GT—building $\mathcal{N}_S$, the bit-array representing the neighborhood $\mathcal{N}(S, d)$ of a long DNA sequence S. The speedup technique allows EMS-GT to build $\mathcal{N}_S$ in the following steps:

1. Initialize $\mathcal{N}_S$ as an array of 4^l bits set to zero, and select a value for k.

2. Pre-generate *Pattern*(z, d_z) for all $z \in \Sigma^k$ and all $d_z \in \{1, ..., k-1\}$ to serve as bit masks for blocks. Note that block patterns for $d_z = 0$ (one bit set) and $d_z = k$ (all bits set) will not require bit masks.

3. For each l-mer $x = yz$ in sequence S: take each neighbor y' of y, find the block in $\mathcal{N}_S$ whose prefix is y', and compute the allowable suffix mismatches $d_z = d - d_H(y, y')$ within this block. Then,

   (a) if $d_z = 0$, set the bit at position z in the block;

   (b) if $d_z \geq k$, set all bits in the block to 1;

   (c) otherwise, mask *Pattern*(z, d_z) onto the block.

This entire procedure effectively performs $\mathcal{N}(S, d) \leftarrow \mathcal{N}(S, d) \cup N(x, d)$ for each l-mer x in S, as specified in lines 2-5 and 9-12 of (Algorithm 2.1) EMS-GT .

The speedup technique requires us to pre-generate all possible block patterns for a given k, for which we can use Algorithm 4.1. The set of patterns generated excludes those patterns where none, all, or only one of the bits in the block are set—these are considered trivial to apply to blocks, even without pre-generated bit masks.

Algorithm 4.1 BLOCK PATTERN GENERATION
Input: block degree k
Output: 3D bit-array $\mathcal{P}$ containing all possible non-trivial block patterns

```
1: P[ ][ ][ ] ← {}                          ▷ retrieve a pattern P as P[z][d − d_y']
2: for z ← 0 to 4^k do
3:     for j ← 1 to k − 1 do
4:         for z' ← 0 to 4^k do
5:             if dH(z, z') ≤ j then
6:                 P[z][j][z'] ← 1
7:             else
8:                 P[z][j][z'] ← 0
9:             end if
10:        end for
11:    end for
12: end for
13: return P
```

After generating $\mathcal{P}$, we can retrieve *Pattern*(z, d_z) as $\mathcal{P}[\ z\ ][\ d_z\ ]$, which is an array of 4^k bits. In the actual implementation, the bit-arrays representing the patterns are also "compressed" to use 32-bit integers, following EMS-GT's convention. Note also that pre-generating and storing all $4^k \times (d_z - 2)$ entails non-negligible time and space overhead depending on the value of k (i.e. storing all patterns requires $4^k \times (k-2) \times 4^k$ bits).

4.4 Performance improvement

Figure 4.4 compares the performance of EMS-GT with and without the speedup technique. For challenge instance (9,2), the computational overhead of generating and retrieving block patterns increases the runtime from 0.06 s to 0.11 s, but for all other instances tested, the speedup technique significantly reduces runtime.

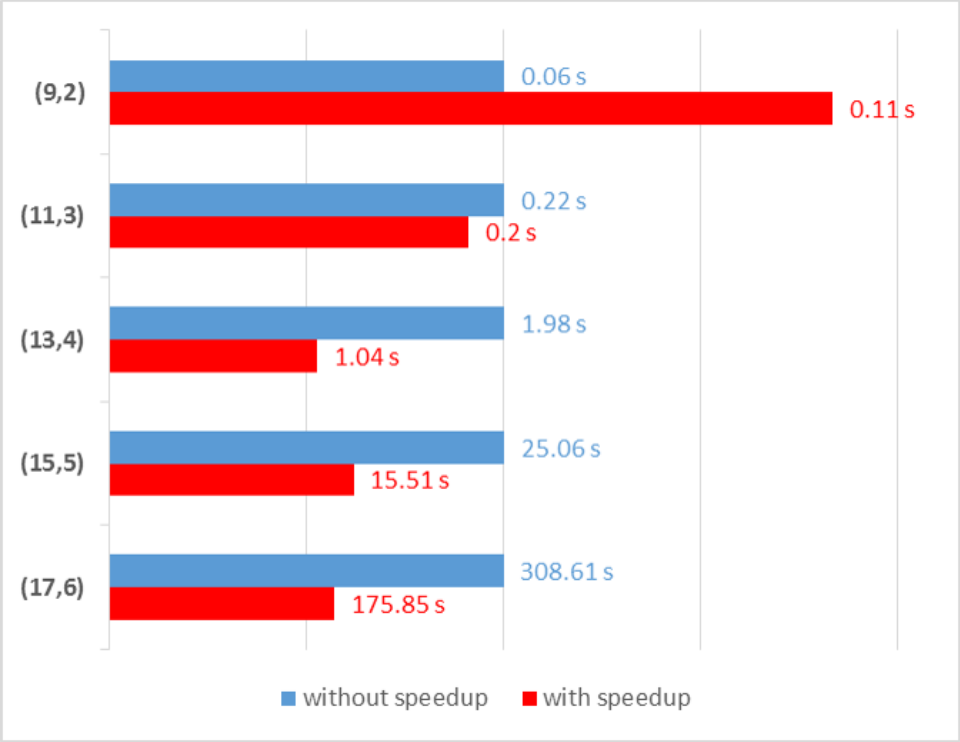


Figure 4.4. EMS-GT runtimes without (baseline) vs. with speedup technique.

Specifically, for challenge instances (11,3), (13,4), (15,5) and (17,6), the speedup technique allows runtime reductions of at least 6.7%, 47.5%, 38.1% and 43.0% respectively.

The pattern-based speedup technique requires examining all d-neighbors of x's prefix y. We can use Algorithm 2.3 to recursively traverse $N(y, d)$; as Figure 4.5 shows, this will encounter much fewer neighbors than $N(x, d)$.

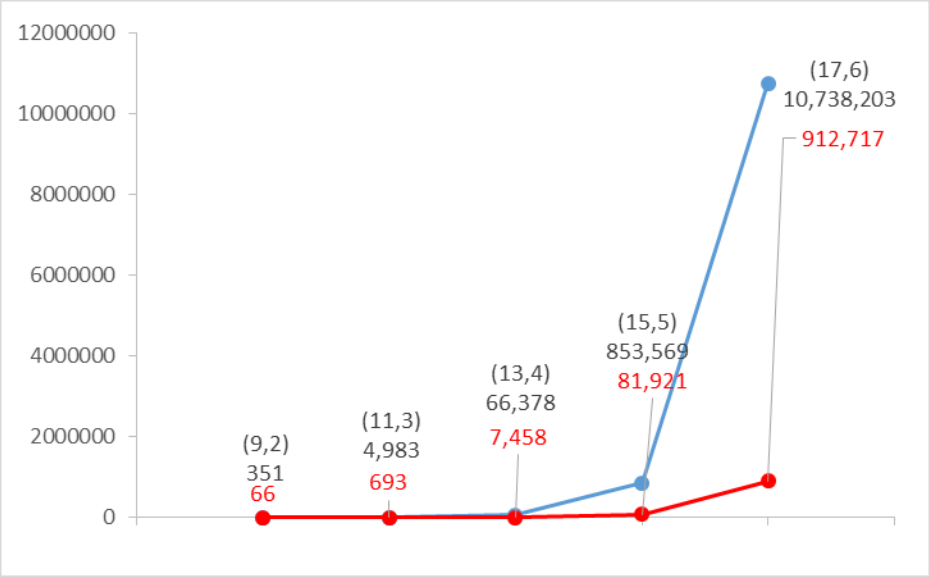


Figure 4.5. Number of neighbors in $N(y, d)$ vs. $N(x, d)$ for challenging (l, d), k=5.

This large difference in neighborhood size partly explains why, as (l, d) values increase, building a neighborhood $\mathcal{N}_S$ in blocks proves significantly faster than generating each individual neighbor, then locating and setting its bit flag. The tradeoff is that, for each neighbor of x's prefix y, we set an entire block of bits; this will entail $\frac{4^k}{32}$ OR operations (between the $\frac{4^k}{32}$ consecutive integers of a block in N_x and the bit-mask of the same size).

Again, we see the suffix length k determines the complexity for the speedup technique. For the preceding runtime measurements, we use k=5, which empir-

ical tests confirm to be the optimum value of k for all (l, d) tested.

(l, d)	$k = 3$ 32×2 **blk**	$k = 4$ 32×8 **blk**	$k = 5$ 32×32 **blk**	$k = 6$ 32×128 **blk**	$k = 7$ 32×512 **blk**
(9,2)	0.14 s	0.12 s	**0.11 s**	0.57 s	9.02 s
(11,3)	0.25 s	0.22 s	**0.20 s**	0.70 s	9.77 s
(13,4)	1.74 s	1.31 s	**1.04 s**	2.19 s	12.40 s
(15,5)	23.43 s	21.43 s	**15.51 s**	24.28 s	46.30 s

Table 4.2. EMS-GT runtimes with speedup technique, using different values of k. Shortest runtimes are in bold text.

When k is lower, there are fewer and smaller 4^k-bit block patterns to manage, but building N_x requires more highly scattered array accesses to small blocks. When k is higher, there are larger contiguous blocks and thus fewer scattered accesses to N_x, but also a much larger set of 4^k-bit block patterns to manage. It seems that $k = 5$ allows for the most efficient balance between recursive neighbor generation (which identifies and accesses blocks according to their prefixes) and bit-masking operations (which selects and applies the correct pattern to each block), with a feasible number of block patterns.

4.5 EMS-GT performance comparison vs. PMS8 and qPMS9

Finally, the improved EMS-GT and two competitor algorithms were run on an Intel Xeon, 2.10 GHz processor (single core). Their runtimes, averaged over 20 synthetic datasets per (l, d) instance, are compared in Figure 4.6.

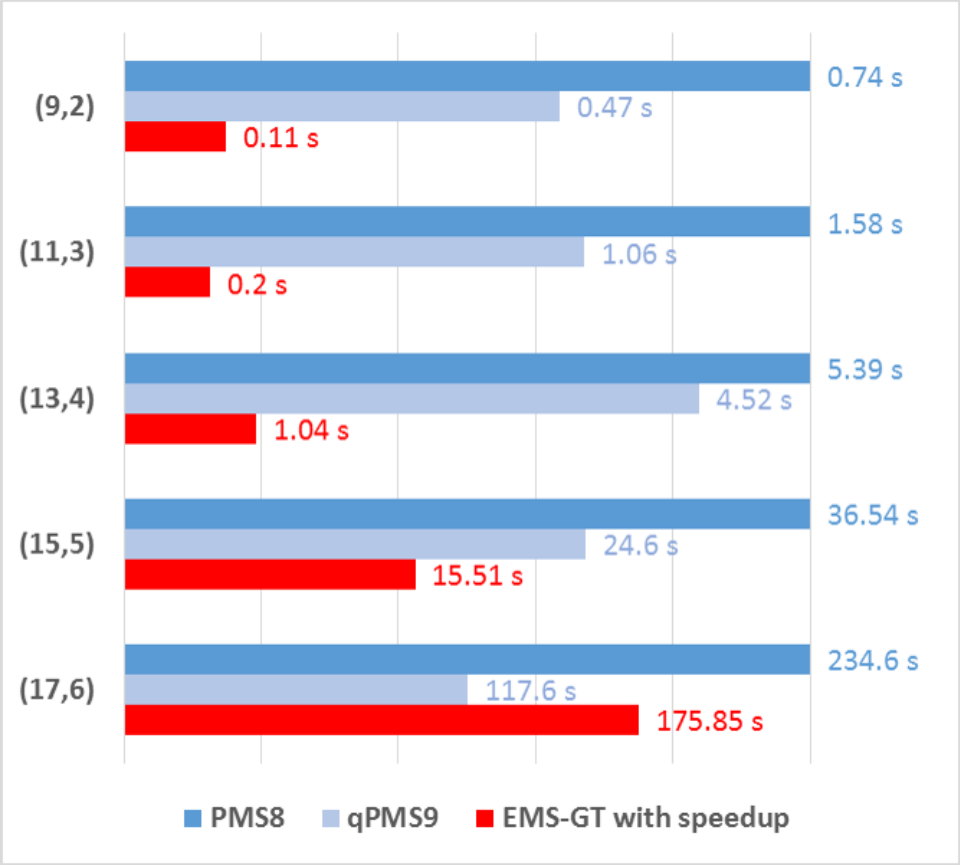


Figure 4.6. Improved EMS-GT's performance vs. PMS8 (baseline) and qPMS9.

On every challenging (l, d) instance except (17,6) the improved EMS-GT outperforms qPMS9; it outperforms PMS8 on all the challenge instances. Note that without the speedup technique, EMS-GT could not outperform PMS8 for (17,6) (see Figure 2.3), nor qPMS9 on (15,5); see Table 4.3 for the full summary of runtimes.

EMS-GT is highly competitive with PMS8 and qPMS9 on challenging problem instances, but as Table 4.4 shows, EMS-GT is less efficient on non-challenging instances. This may be due to differing efficiency considerations

(l, d)	**PMS8**	**qPMS9**	**EMS-GT**	**EMS-GT with speedup**
9,2	0.74 s	0.47 s	**0.06** s	0.11 s
11,3	1.58 s	1.06 s	0.22 s	**0.20 s**
13,4	5.39 s	4.52 s	1.98 s	**1.04 s**
15,5	36.45 s	24.63 s	25.06 s	**15.51 s**
17,6	3.91 min	**1.96 min**	5.14 min	2.93 min

Table 4.3. Runtimes of PMS8, qPMS9, EMS-GT without and with speedup technique. Shortest times are in bold text.

when narrowing down the search space: while EMS-GT manipulates a fixed-size bit-array to narrow down the search space of all 4^l l-mers, PMS8 and qPMS9 have a dynamically-sized search space which is easily pruned down for most (l, d) values, but more difficult to manage for challenging (l, d) instances.

(l, d)	**PMS8**	**qPMS9**	**EMS-GT**	**EMS-GT with speedup**
14,4	1.29 s	1.02 s	3.53 s	2.55 s
16,5	4.79 s	2.96 s	41.63 s	29.03 s

Table 4.4. Runtimes of PMS8, qPMS9, EMS-GT without and with speedup technique on non-challenging (l, d).

CHAPTER V

CONCLUSIONS

In line with our research objectives, we make the following conclusions:

1. Our novel speedup technique takes advantage of the distance-related block patterns observed in the search space. Initially EMS-GT generates each neighbor of an l-mer x and sets a corresponding bit in an array. However, our speedup technique allows EMS-GT to set these bits in blocks of 4^k bits each, using pre-generated bit patterns as bit-masks; we find the ideal value of k to be 5.

2. The speedup technique improves EMS-GT's performance on challenging (l, d) instances (11,3), (13,4), (15,5) and (17,6), with runtime reductions of at least 6.7%, 47.5%, 38.1% and 43.0% respectively; however, on challenge instance (9,2), overhead increases EMS-GT's runtime from 0.06 s initially to 0.11 s with the speedup technique.

3. The speedup technique allows EMS-GT to outperform the current best algorithm, qPMS9, on challenging (l, d) instances (9,2), (11,3), (13,4) and (15,5) with runtime reductions of at least 76%, 81%, 77% and 37% respectively for these instances, while ranking second to qPMS9's runtime on challenge instance (17,6).

Directions for further research on improving EMS-GT include:

- Refining the bit-based search space representation (i.e. with compression techniques) to be able to represent the motif search space for $l > 17$;
- Creating a multiprocessor version of EMS-GT to solve the planted motif problem faster, in parallel, for larger values of (l, d); and
- Delegating the bit-masking speedup technique and other bulk bit operations to the graphics card, as explored in [4], for faster performance.

Finally, the EMS-GT speedup technique described by this paper can conceivably be translated to other string-matching and pattern-finding tasks beyond DNA motif search; however, its implementation will involve non-negligible computational and storage overhead which varies with the suffix length k. Choosing a value for k is thus a key efficiency consideration when using this technique.

BIBLIOGRAPHY

[1] Timothy L Bailey and Charles Elkan. Unsupervised learning of multiple motifs in biopolymers using expectation maximization. *Machine learning*, 21(1-2):51–80, 1995.

[2] Mathieu Blanchette and Martin Tompa. Discovery of regulatory elements by a computational method for phylogenetic footprinting. *Genome research*, 12(5):739–748, 2002.

[3] Modan K Das and Ho-Kwok Dai. A survey of dna motif finding algorithms. *BMC bioinformatics*, 8(Suppl 7):S21, 2007.

[4] Naga Shailaja Dasari, Ranjan Desh, and Mohammad Zubair. An efficient multicore implementation of planted motif problem. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 9–15. IEEE, 2010.

[5] Jaime Davila, Sudha Balla, and Sanguthevar Rajasekaran. Fast and practical algorithms for planted (l, d) motif search. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 4(4):544–552, 2007.

[6] Thomas A Down and Tim JP Hubbard. Nestedmica: sensitive inference of over-represented motifs in nucleic acid sequence. *Nucleic acids research*,

33(5):1445–1453, 2005.

[7] Eleazar Eskin and Pavel A Pevzner. Finding composite regulatory patterns in dna sequences. *Bioinformatics*, 18(suppl 1):S354–S363, 2002.

[8] Gerald Z Hertz and Gary D. Stormo. Identifying dna and protein patterns with statistically significant alignments of multiple sequences. *Bioinformatics*, 15(7):563–577, 1999.

[9] Jianjun Hu, Yifeng D Yang, and Daisuke Kihara. Emd: an ensemble algorithm for discovering regulatory motifs in dna sequences. *BMC bioinformatics*, 7(1):342, 2006.

[10] Hongwei Huo, Zhenhua Zhao, Vojislav Stojkovic, and Lifang Liu. Combining genetic algorithm and random projection strategy for (l, d)-motif discovery. In *Bio-Inspired Computing, 2009. BIC-TA'09*, pages 1–6. IEEE, 2009.

[11] Uri Keich and Pavel A Pevzner. Finding motifs in the twilight zone. In *Proceedings of the sixth annual international conference on Computational biology*, pages 195–204. ACM, 2002.

[12] Charles E Lawrence, Stephen F Altschul, Mark S Boguski, Jun S Liu, Andrew F Neuwald, and John C Wootton. Detecting subtle sequence signals:

a gibbs sampling strategy for multiple alignment. *science*, 262(5131):208–214, 1993.

[13] Charles E Lawrence and Andrew A Reilly. An expectation maximization (em) algorithm for the identification and characterization of common sites in unaligned biopolymer sequences. *Proteins: Structure, Function, and Bioinformatics*, 7(1):41–51, 1990.

[14] Julieta Q. Nabos. *New Heuristics and Exact Algorithms for the Planted DNA (l, d)-Motif Finding Problem*. PhD thesis, Ateneo de Manila University, 2015.

[15] Marius Nicolae and Sanguthevar Rajasekaran. Efficient sequential and parallel algorithms for planted motif search. *BMC bioinformatics*, 15(1):34, 2014.

[16] Marius Nicolae and Sanguthevar Rajasekaran. qpms9: An efficient algorithm for quorum planted motif search. *Scientific reports*, 5, 2015.

[17] Pavel A Pevzner, Sing-Hoi Sze, et al. Combinatorial approaches to finding subtle signals in dna sequences. In *ISMB*, volume 8, pages 269–278, 2000.

[18] Alkes Price, Sriram Ramabhadran, and Pavel A Pevzner. Finding subtle motifs by branching from sample strings. *Bioinformatics*, 19(suppl 2):ii149–ii155, 2003.

APPENDIX A

Source code for EMS-GT, with speedup technique

```
/**
  * EMS-GT_32.java
  * Solves the (l,d) planted motif problem.
  * Bit-array width is 32 bits (Integer type).
  * Optimized with block masking.
  *
  * @author Aia Sia, Julieta Nabos
  * @version 1.0 9/09/2015
  */

import java.io.*;
import java.util.*;

public class EMS_GT_32 {
  static int TABLE_WIDTH_LOG_2 = 5;
  static int TABLE_WIDTH = (1 << TABLE_WIDTH_LOG_2);

  static String inputFileHeader = "../datasets/";
  static String inputFileName = "";
  static final int MAX_INDICES = 120000000;
  static final int tPrime = 11;

  // from input file
  static int t, l, n, d;
  static int[] plantedAlignment;
  static String plantedMotif;
  static String foundMotifs;
  static char[][] seqS;

  // for EMS-GT computations
  static long mask, prefixMask, suffixMask;
  static int nl1;
  static int pmt;
  static int[] currNeighborhood;
  static int[] candidateMotifs;
  static long[][] lmerMappings;

  // for masking
  static int blockDegree = 5;
  static int lmersInBlock;
  static int rowsInBlock;
  static int prefixShift;
```

```
  static int[][][] blockMasks;
  static int[][] currBlockMasks;
  static long prefix;
  static long suffix;
  static int currBlockRow;
  static int currBlockCol;

//=====================================================================
// MAIN METHOD
//=====================================================================

  public static void main(String args[]) throws Exception {
    if( args.length > 0 ) {
      inputFileName = args[0];
    }
    if( args.length > 1 ) {
      blockDegree = Integer.parseInt(args[1]);
    }

    readInput( inputFileHeader + inputFileName );

    // compute preliminaries
    mask = (((long) 1) << 2*(l-1) ) - 1;
    prefixMask = (((long) 1) << 2*(l-blockDegree-1) ) - 1;
    suffixMask = (((long) 1) << 2*(blockDegree-1) ) - 1;
    nl1 = n - l + 1;
    pmt = (int) ( (long)1 << (2*l - TABLE_WIDTH_LOG_2)); // (4^l)/TABLE_WIDTH
    prefixShift = (2*blockDegree - TABLE_WIDTH_LOG_2);

    System.out.print("\n" + inputFileName);

    // start EMS-GT
    long sTime = System.nanoTime();
    generateBlockMasks();
    collectCandidateMotifs();
    transformLmerSequences(tPrime);
    searchMotif();
    long eTime = System.nanoTime();

    // compute runtime and used memory
    Runtime runtime = Runtime.getRuntime();
    long memUse = runtime.totalMemory() - runtime.freeMemory();
    runtime.gc();
    long memUseGC = runtime.totalMemory() - runtime.freeMemory();

    System.out.print(", " + (eTime - sTime) / 1000000000.0);
    System.out.print(", " + (eTime - sTime) / 1000000000.0 / 60.0);
    System.out.print(", " + memUse / 1024.0 / 1024.0);
    System.out.print(", " + memUseGC / 1024.0 / 1024.0);
```

```
  System.out.print(", " + plantedMotif + "," + foundMotifs);
}

public static void generateBlockMasks() throws Exception {
  lmersInBlock = 1 << (2 * blockDegree);  // 4 ^ blockDegree
  rowsInBlock  = 1 << (2 * blockDegree - TABLE_WIDTH_LOG_2);
                                          // 4 ^ blockDegree / TABLE_WIDTH
  blockMasks = new int[lmersInBlock][blockDegree - 1][rowsInBlock];
  for(int i=0; i < lmersInBlock; i++) {
    for(int k=0; k < blockDegree - 1; k++) {
      Arrays.fill( blockMasks[i][k], 0);
    }
    for(int row=0; row < rowsInBlock; row++) {
      for(int col=31; col > -1; col--) {
        int distance = computeHD(i, row*TABLE_WIDTH+col);
        for(int k=0; k < blockDegree - 1; k++) {
          if(distance <= k+1) {
            blockMasks[i][k][row]++;
          }
          if(col > 0) {
            blockMasks[i][k][row] = blockMasks[i][k][row] << 1;
          }
        }
      }

    }
  }
}

public static void readInput(String filename) throws Exception {
  Scanner input = new Scanner (new File (filename ));
  t = input.nextInt();
  n = input.nextInt();
  l = input.nextInt();
  d = input.nextInt();
  plantedAlignment = new int[t];
  seqS = new char[t][n];
  int a = 0;
  while (input.hasNextInt()) {
    plantedAlignment[a] = input.nextInt();
    a++;
  }
  plantedMotif = input.next().toUpperCase();
  int n = 0;
  while (input.hasNext()) {
    String seq = input.next().toUpperCase();
    seqS[n] = seq.toCharArray();
    n++;
  }
```

```
    input.close();
  }

//=================================================================
// intersect d-neighborhoods of sequences 0 to tPrime.
//=================================================================

  public static void collectCandidateMotifs() throws Exception {
    generateNeighborhood(0);
    candidateMotifs = currNeighborhood;
    for(int i=1; i < tPrime; i++) {
      generateNeighborhood(i);
      for(int j=0; j < pmt; j++)
        candidateMotifs[j] &= currNeighborhood[j];
    }
  }

  public static void generateNeighborhood(int s) throws Exception {
    currNeighborhood = new int[pmt];
    char[] currSeq = seqS[s];

    prefix = 0;   // first ( l-blockDegree ) characters of l-mer
    suffix = 0;   // last  (  blockDegree  ) characters of l-mer
    for(int i=0; i < l; i++) {
      char c = currSeq[i];
      int base = 0;
      switch(c) {
        case 'C': base=1; break;
        case 'G': base=2; break;
        case 'T': base=3; break;
      }
      if( i < l - blockDegree )
        prefix = (prefix << 2) + base;
      else
        suffix = (suffix << 2) + base;
    }

    // housekeeping: set blockOffsets, currBlockMasks
    currBlockRow = (int) (suffix / TABLE_WIDTH);
    currBlockCol = (int) (suffix \% TABLE_WIDTH);
    currBlockMasks = blockMasks[(int)suffix];
    int blockStart = (int) (prefix * rowsInBlock);
    for(int offset=0; offset < rowsInBlock; offset++) {
      if(d >= blockDegree)
        currNeighborhood[blockStart+offset] = Integer.MAX_VALUE;
      else
        currNeighborhood[blockStart+offset] |= currBlockMasks[d - 1][offset];
    }
    addNeighbors(prefix, 0, d);
```

```
  for(int i=l; i < n; i++) {
    prefix = (prefix & prefixMask) << 2;
    suffix = (suffix & suffixMask) << 2;

    char c = currSeq[i-blockDegree];  // next char for prefix
    switch(c) {
      case 'C': prefix+=1; break;
      case 'G': prefix+=2; break;
      case 'T': prefix+=3; break;
    }

    c = currSeq[i];           // next char for suffix
    switch(c) {
      case 'C': suffix+=1; break;
      case 'G': suffix+=2; break;
      case 'T': suffix+=3; break;
    }

    // housekeeping: set blockOffsets, currBlockMasks
    currBlockRow = (int) suffix / TABLE_WIDTH;
    currBlockCol = (int) suffix \% TABLE_WIDTH;
    currBlockMasks = blockMasks[(int)suffix];
    blockStart = (int) prefix << (2*blockDegree - TABLE_WIDTH_LOG_2);
    for(int offset=0; offset < rowsInBlock; offset++) {
      if(d >= blockDegree)
        currNeighborhood[blockStart+offset] = Integer.MAX_VALUE;
      else
        currNeighborhood[blockStart+offset] |= currBlockMasks[d - 1][offset];
    }
    addNeighbors(prefix, 0, d);
  }
}

public static void addNeighbors(long prefix, int start, int d) {
  int shift = (l-blockDegree-start)*2;
  for(int i=start; i < l-blockDegree; ++i) {
    shift -= 2;
    long alt1 = prefix ^ (((long) 1) << shift);
    long alt2 = prefix ^ (((long) 2) << shift);
    long alt3 = prefix ^ (((long) 3) << shift);

    int blockStart1 = (int) alt1 << prefixShift;
    int blockStart2 = (int) alt2 << prefixShift;
    int blockStart3 = (int) alt3 << prefixShift;

    // masking part
    int allow_d = d - 1;
    if( allow_d >= blockDegree ) {  // all 1's
```

```
        for(int offset=0; offset < rowsInBlock; offset++) {
          currNeighborhood[blockStart1 + offset] = Integer.MAX_VALUE;
          currNeighborhood[blockStart2 + offset] = Integer.MAX_VALUE;
          currNeighborhood[blockStart3 + offset] = Integer.MAX_VALUE;
        }
      }
      else if( allow_d > 0 ) {    // select a mapping from 1 to blockDegree-1
        for(int offset=0; offset < rowsInBlock; offset++) {
          currNeighborhood[blockStart1 + offset]
              |= currBlockMasks[allow_d - 1][offset];
          currNeighborhood[blockStart2 + offset]
              |= currBlockMasks[allow_d - 1][offset];
          currNeighborhood[blockStart3 + offset]
              |= currBlockMasks[allow_d - 1][offset];
        }
      }
      else {              // only [currBlockRow][currBlockCol] = 1
        currNeighborhood[blockStart1 + currBlockRow] |= 1 << (currBlockCol);
        currNeighborhood[blockStart2 + currBlockRow] |= 1 << (currBlockCol);
        currNeighborhood[blockStart3 + currBlockRow] |= 1 << (currBlockCol);
      }

      // recursive call
      if(allow_d > 0) {
        addNeighbors(alt1, i+1, allow_d);
        addNeighbors(alt2, i+1, allow_d);
        addNeighbors(alt3, i+1, allow_d);
      }

    }
  }

//=================================================================
// create mappings for each l-mer in sequences tPrime+1 to t.
//=================================================================

  public static void transformLmerSequences(int tPrime) {
    lmerMappings = new long[t][nl1];
    for(int i=tPrime; i < t; i++) {
      char[] currSeq = seqS[i];
      long mapping = 0;

      for(int j=0; j < l; j++) {
        char c = currSeq[j];
        int base = 0;
        switch(c) {
          case 'C': base=1; break;
          case 'G': base=2; break;
          case 'T': base=3; break;
```

```
        }
        mapping = (mapping << 2) + base;
      }
      lmerMappings[i][0] = mapping;

      int k=0;
      for(int j=l; j < n;j++) {
        char c = currSeq[j];
        int base = 0;
        switch(c) {
          case 'C': base=1; break;
          case 'G': base=2; break;
          case 'T': base=3; break;
        }
        mapping = ((mapping & mask) << 2) + base;
        lmerMappings[i][++k]=mapping;
      }

    }
  }

//=====================================================================
// check each candidateMotif if present in sequences tPrime+1 to t.
//=====================================================================

  public static void searchMotif() throws Exception {
    int value, numMotifs = 0;
    foundMotifs = "";
    for(int i=0; i < pmt; i++) {
      if( (value = candidateMotifs[i]) == 0)
        continue;

      /*System.out.println("Nonzero: candidateMotifs[" + i + "]\t= "
        + Long.toBinaryString(candidateMotifs[i]));*/
      long base = ((long) i) << TABLE_WIDTH_LOG_2;
      for(int j=0; j < TABLE_WIDTH; j++) {
        if( (value & 1) != 0) {
          long candidate = base + j;
          if( isMotif(candidate, tPrime) ) {
            foundMotifs += " " + decode(candidate, l);
            // System.out.println("Motif found:\t" + decode(candidate, l) );
            numMotifs++;
          }
        }
        value = value >> 1;
      }
    }
  }
```

```
  public static boolean isMotif(long mapping, int tPrime) throws Exception {
    for(int i=tPrime; i < t; i++) {
      boolean found = false;
      for(int j=0; j < nl1; j++) {
        long lmer = lmerMappings[i][j];
        int hd = computeHD(mapping, lmer);
        if( hd <= d ) {
          found=true;
          break;
        }
      }
      if(!found) {
        return false;
      }
    }
    return true;
  }

  public static int computeHD(long lmer1, long lmer2) throws Exception {
    int distance=0;
    long result = lmer1 ^ lmer2;
    for(int i=0; i < l; i++) {
      if( (result & 3) != 0 )
        distance++;
      result = result >>> 2;
    }
    return distance;
  }

  public static String decode(long mapping, int strlen) throws Exception {
    String decoding = "";
    for(int i=0; i < strlen; i++) {
      int base = (int) mapping & 3;
      switch(base) {
        case 0: decoding = "A" + decoding; break;
        case 1: decoding = "C" + decoding; break;
        case 2: decoding = "G" + decoding; break;
        case 3: decoding = "T" + decoding; break;
      }
      mapping = mapping >>> 2;
    }
    return decoding;
  }

}
```