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

Abstract

DNA motif finding is widely recognized as a difficult problem in computational biology and computer science. Because of the usual large search space involved, exact solutions typically require a significant amount of execution time before discovering a motif of length l that occurs in an input set $\{S_1, ..., S_n\}$ of sequences, allowing for at most d substitutions.

This study implements a novel improvement to EMS-GT, a motif search algorithm which operates on a compact bit-based representation of the search space. The improvement takes advantage of distance-related patterns in the search space, in order to speed up the bulk bit-setting operations performed by the algorithm. A Java implementation is shown to be highly competitive against PMS8 and qPMS9, two current state-of-the-art exact algorithms. EMS-GT works extremely well for problems involving short motifs, outperforming both competitors for challenge instances with (l, d) values (9,2), (11,3), (13,4) and (15,5), showing runtime reductions of at least 76%, 81%, 77% and 37% respectively for these instances, while ranking second to qPMS9 for challenge instance (17,6).

TABLE OF CONTENTS

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.

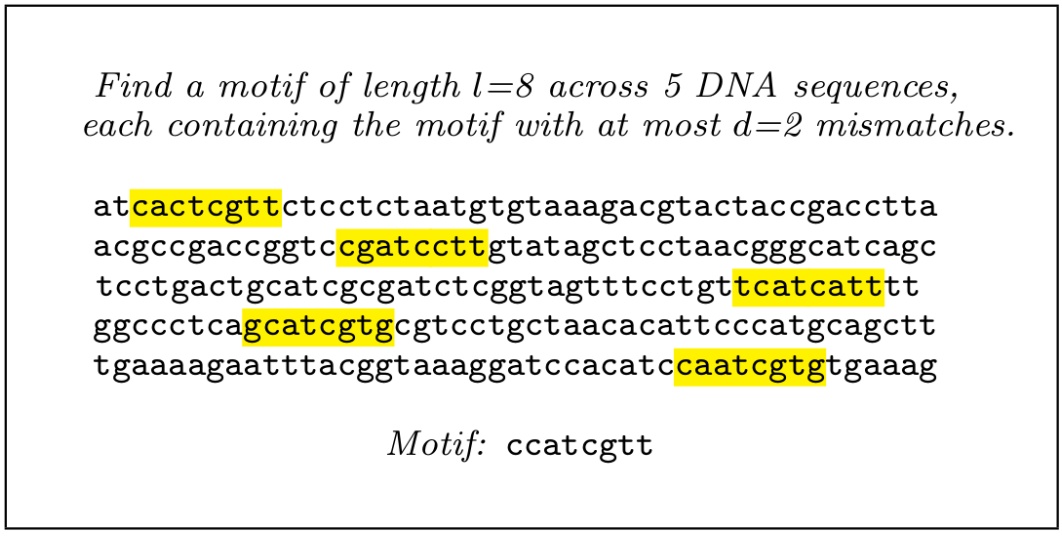

Figure 1.1. Sample instance of the planted motif problem.

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.

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-mer $x \mid \forall i \in \{1, ..., n\}$ $\exists x_i \in \mathcal{L}(S_i, l)$, $dH(x, x_i) \leq d\}$.

An l-mer is a sequence of length l over the alphabet $\Sigma = \{\texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}\}$. There are $(L - l + 1)$ l-mers in a sequence of length $L > l$.

For $l = 7$,
$S = \texttt{acgccgattacatccgatccttgtatagctcctaacgggcatcac}$
↪ gattaca is the 6^{th} l-mer in S.
$\mathcal{L}(S, l) = \{\texttt{acgccga}, \texttt{cgccgat}, \texttt{gccgatt}, \texttt{ccgatta}, \texttt{cgattac}, \texttt{gattaca}, ...\}$
↪ $\mathcal{L}$ gives us the set of all l-mers in S.

The Hamming distance dH between two l-mers is the number of *corresponding* characters that differ between them.

$x_1 = \texttt{gattaca}$
$x_2 = \texttt{cgttaga}$
↪ x_1 and x_2 differ in their first, second and sixth characters. Thus, $dH(x_1, x_2) = 3$.

The d-neighborhood $N(\texttt{gattaca}, 2)$ is the set of all d-neighbors of gattaca: i.e., all l-mers whose Hamming distance from gattaca is at most d.

$N(\texttt{gattaca}, 2) = \{$ gattaca,
1 mismatch → $= \binom{7}{1} \times 3^1 = 21$ neighbors
aattaca, cattaca, tattaca,
gcttaca, ggttaca, gtttaca,
...,
gattacc, gattacg, gattact,
2 mismatches → $= \binom{7}{2} \times 3^2 = 189$ neighbors
acttaca, agttaca, atttaca, ..., tcttaca, tgttaca, ttttaca,
aaataca, aactaca, aagtaca, ..., taataca, tactaca, tagtaca,
aataaca, aatcaca, aatgaca, ..., tataaca, tatcaca, tatgaca,
...,
gattccc, gattccg, gattcct, ..., gatttcc, gatttcg, gatttct
gattaac, gattaag, gattaat, ..., gattatc, gattatg, gattatt
$\}$
↪ To generate a neighbor of gattaca, we choose at most 2 positions from $1, 2, ...7$, and change the character at each chosen position with an alternative from $\Sigma = \{\texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}\}$.

The d-neighborhood $\mathcal{N}(S, 2)$ of sequence S is the set of all d-neighbors of all the l-mers in S, for a given value of l.

$S = \texttt{acgccgattacatccgatccttgtatagctcctaacgggcatcac}$
↪ first l-mer in S
$\mathcal{N}(S, 2) = N(\texttt{acgccga}, 2) \cup N(\texttt{cgccgat}, 2) \cup ... \cup N(\texttt{gcatcac}, 2)$
↪ d-neighborhood of first l-mer in S

Figure 1.2. Key concepts for exact motif search algorithms.

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 resulting 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 of the Study

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 correct-

ness using real biological data with known motifs; re-testing would be redundant since the speedup technique does not change the logic of the 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 (over 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.

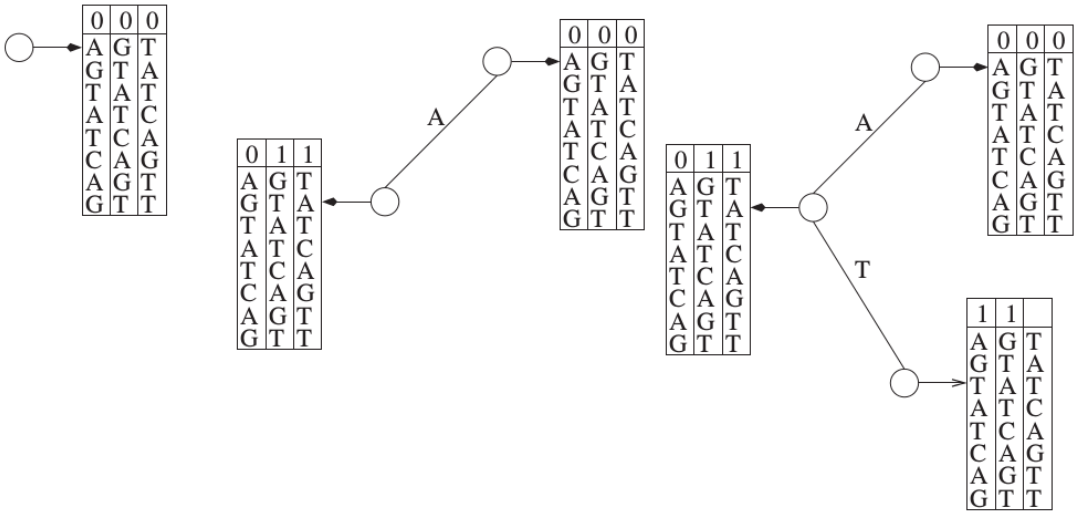

Figure 2.1. An (8,1) mismatch tree for {agtatcag, gtatcgat, tatcagtt}. *Left*: the tree, in its initial state. *Center*: the tree, after traversing prefix `--a`: the second and third l-mers now each have a single mismatch with the prefix. *Right*: the tree, after traversing prefix `--a--t`: the third 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.

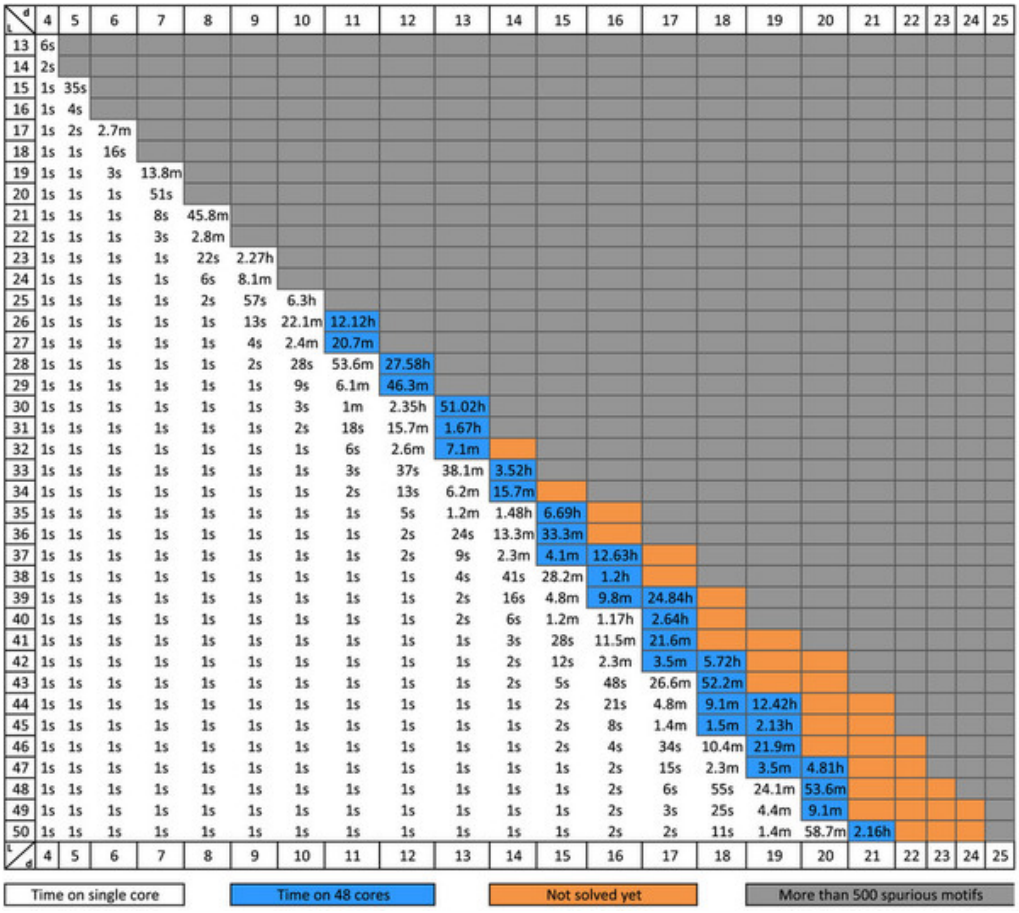

Figure 2.2. Runtime of qPMS9 on DNA datasets for various (l,d) values. Blue indicates the program used 48 cores; white indicates single-core execution. Gray cells represent instances with over 500 spurious motifs. Instances in orange could not be solved efficiently. Image from [16].

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

In practice, EMS-GT must perform speedy operations on an array of bits representing the entire motif search space. The succeeding sections discuss the efficiency strategies EMS-GT uses for important tasks such as 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 and l-mer enumeration

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 s to a bit flag index x by replacing each character with 2 bits (`a=00`, `c=01`, `g=10`, `t=11`). Note that this mapping scheme enumerates l-mers in strict alphabetical order.

> Ex. `tacgt` maps to `1100011011` = 795; thus, its flag is the bit at index 795.

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. `tacgt` maps to `1100011011` = 795 in decimal.

array index = $\lfloor \frac{795}{32} \rfloor$ = 24 , *bit position* = 795 mod 32 = 27;

Thus, the flag for `tacgt` is the bit at index 27 of the integer at index 24.

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.

Ex. `tacgt` maps to `1100011011`

`ttcgg` maps to `1111011010`

XOR produces `00`$\underline{\texttt{11}}$`000`$\underline{\texttt{00}}$`1` = 2 mismatches.

Recursive Neighborhood Generation

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

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

Algorithm 2.2 HAMMING DISTANCE COMPUTATION

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

1: $dH(u, v) = 0$
2: $z \leftarrow u$ XOR v
3: **for** $i \leftarrow 1$ to l **do**
4: **if** $(z$ AND $3) \neq 0$ **then**
5: $dH(u, v) \leftarrow dH(u, v) + 1$
6: **end if**
7: $z \leftarrow z \gg 2$ ▷ *shift two bits to the right*
8: **end for**
9: **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 $\mathcal{N}(S, d)$

1: $\mathcal{N}[lmer] \leftarrow 0, \ \forall\ lmer \in$ search space
2: **for** each l-mer x in S **do**
3: ADDNEIGHBORS(x, 0, d) ▷ *recursive procedure*
4: **end for**

5: ▷ *make d changes in l-mer x, from position s onward*
6: **procedure** ADDNEIGHBORS(x, s, d)
7: **for** $i \leftarrow s$ to l **do**
8: $\Sigma' \leftarrow$ {a, c, g, t} - $\{x[i]\}$ ▷ *remove i^{th} character of x*
9: **for** $j \leftarrow 1$ to $|\Sigma'|$ **do**
10: $neighbor \leftarrow concatenate(x[1...(i-1)], \Sigma_j, x[(i+1)...l])$
11: $\mathcal{N}[neighbor] \leftarrow 1$ ▷ *set $neighbor$'s bit flag to 1*
12: **if** $d > 1$ and $i < l$ **then**
13: ADDNEIGHBORS($neighbor$, $i + 1$, $d - 1$)
14: **end if**
15: **end for**
16: **end for**
17: **end procedure**
18: **return** $\mathcal{N}$

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 representing the neighborhood $N(x, d)$ can be divided 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 N 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].

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 provides a step-by-step derivation of the distance-related patterns that occur within a bit-array representing an l-mer neighborhood in EMS-GT. It then describes how a speedup technique for EMS-GT based on these patterns was implemented. Finally, it quantifies the performance improvement due to this technique.

4.1 Deriving distance-related patterns in l-mer neighborhoods

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

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

2. We find that if we divide this bit-array N into consecutive blocks of 4^k flags each, for some block degree k, $0 < k < l$, each block will conform to one of at most $(k + 2)$ possible bit patterns. We exploit this regularity to build N in blocks.

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

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

   As later explained in steps 7-8, the prefix will decide which of $(k+2)$ patterns is applicable in a particular block in N, while the k-suffix will determine the structure of these $(k + 2)$ patterns.

4. We generate the distribution $D(z)$ of Hamming distances from z to all 4^k possible k-suffixes. We consider this distribution to be "centered" at z.

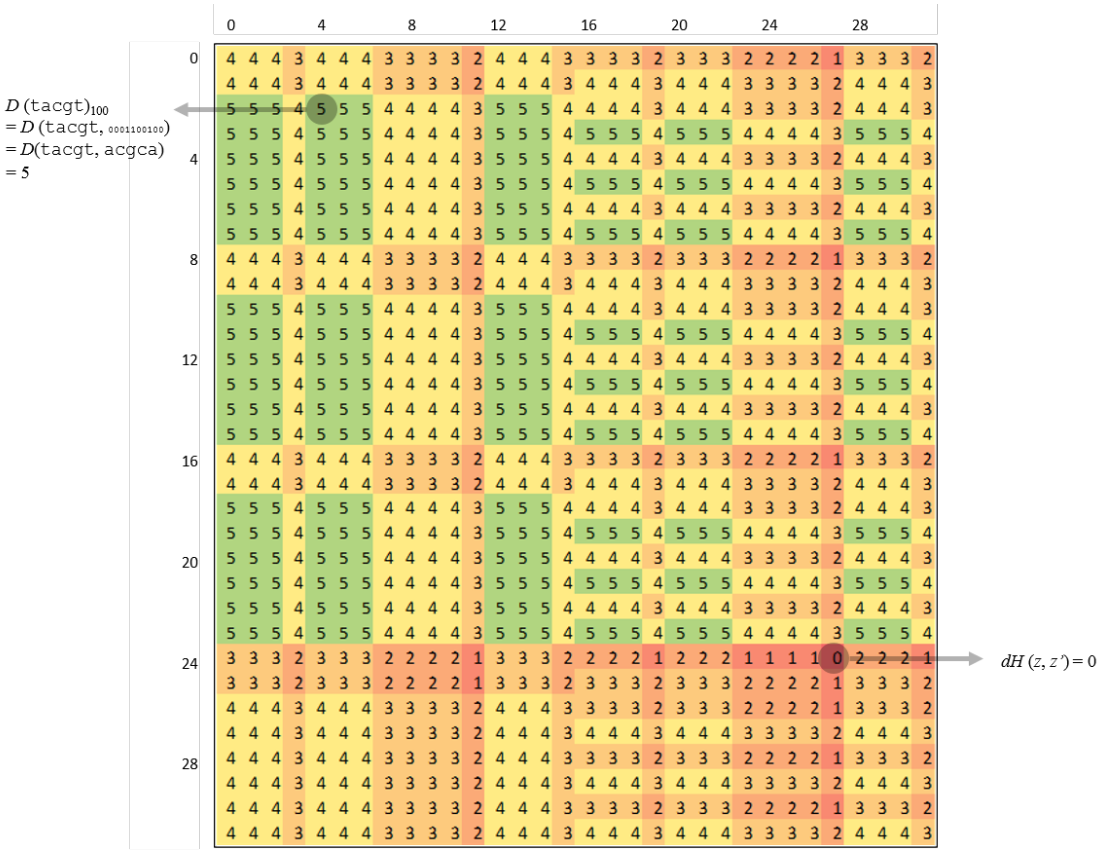

Figure 4.1. Distance distribution from `tacgt` to all $4^5 = 32 \times 32$ k-suffixes, k=5.

The value in the p^{th} cell—found at row $\frac{p}{32}$, column p *mod* 32—of the 32×32 table in Figure 4.1 is the distance $dH(z, z)$ from z = `tacgt` to the k-suffix z' which maps to the binary number p.

Ex. $D(\texttt{tacgt})_{100}$ = $dH(\texttt{tacgt}, \texttt{0b0001100100})$

= $dH(\texttt{tacgt}, \texttt{acgca})$

= 5 , at row $\frac{100}{32} = 3$, column $(100 \mod 32) = 4$

5. Due to the alphabetical enumeration, the 4^k $l-$mers grouped together in a block will all begin with the same $(l-k)$ characters, which we will call the **block prefix y'**. We can compute a single prefix distance $d_{y'}$ for an entire block: this is simply the distance $dH(y, y')$ between x's prefix and the block prefix.

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

6. We can infer that the distance between any two l-mers is equal to the sum of the distance between their prefixes and the distance between their k-suffixes; thus, knowing $d_{y'}$ and $\mathcal{D}(z)$ for some l-mer $x' = y'z'$ in the search space, we can compute its distance from x as:

$$dH(x, x') = d_{y'} + \mathcal{D}(z)_{z'} \tag{4.2}$$

7. With Equations (4.1) and (4.2), we can redefine the criteria for setting a bit in N:

$$N_{x'} = \begin{cases} 1 & \text{if } d_{y'} + \mathcal{D}(z)_{z'} \leq d, \\ 0 & \text{otherwise.} \end{cases} \quad \text{for } x' = y'z'. \tag{4.3}$$

   From this we see that a bit at position z' within a block with prefix y' will be set if and only if $D(z)_{z'} \leq d - d_{y'}$. The values in $D(z)$ range from 0 to k; therefore, we examine $(k+2)$ cases for the value of $(d - d_{y'})$ with respect to the range (0, ..., k):

Case -1:	$d - d_{y'} < 0$,	no bits in the block are set;
Case 0:	$d - d_{y'} = 0$,	the "center" bit (at position z) is set;
Cases 1 to $k-1$:	$1 \leq d - d_{y'} < k$,	some of the bits are set; and
Case k:	$d - d_{y'} \geq k$,	all bits in the block are set.

8. We see that the $(k+2)$ patterns of blocks in N correspond to the $(k+2)$ cases listed in step 7, and thus the pattern to be used in a certain block is indicated by the value of $(d - d_{y'})$. Meanwhile, the polarity (0 or 1) of a bit in one of these patterns is determined by the value of $D(z)$ at the corresponding position.

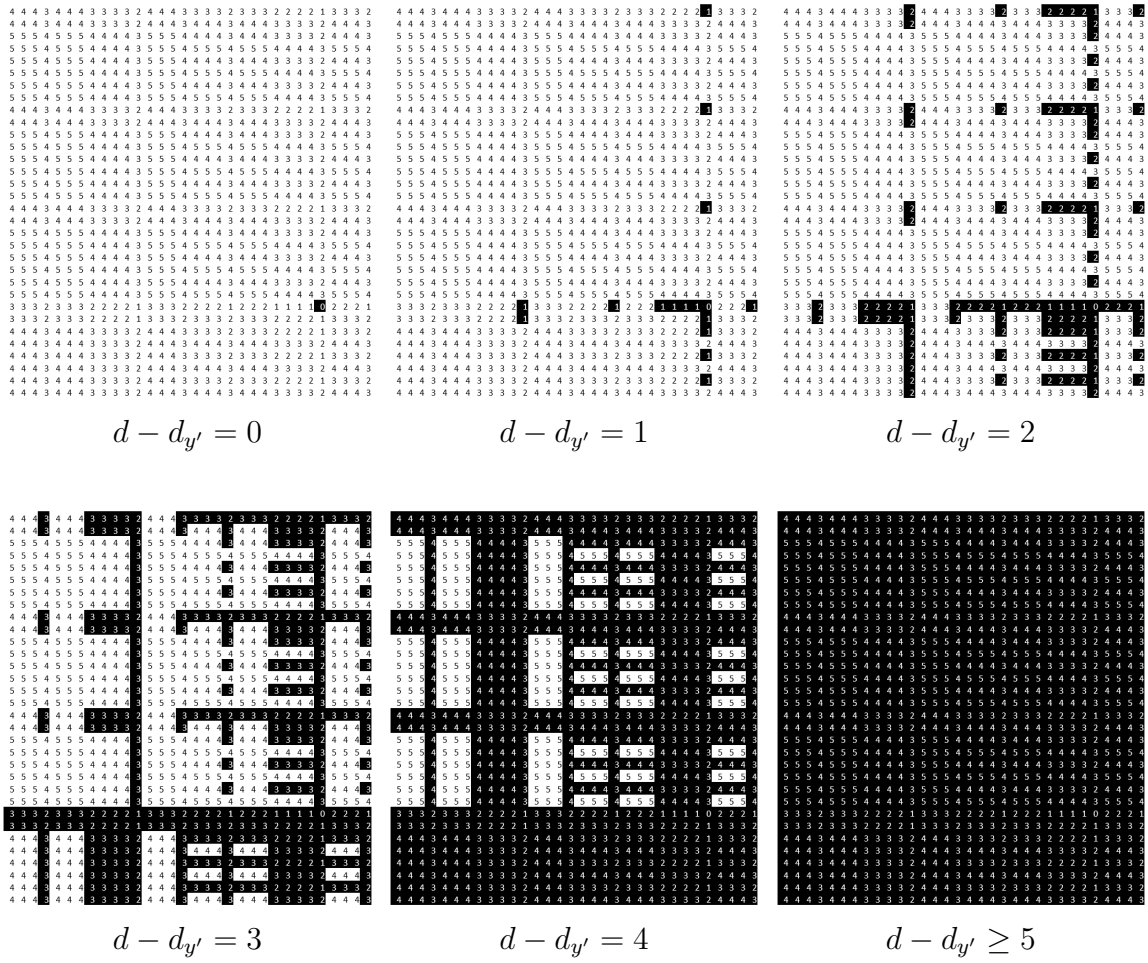

Figure 4.2. Bit patterns followed by blocks in $N(\texttt{tacgt}, 5)$. Black signifies a bit set to 1. There are (5 + 2) = 7 possible patterns; the "empty" pattern (in which all bits are 0) is not shown. Compare the structure of these patterns with the coloring of the distribution $D(\texttt{tacgt})$ in Figure 4.1.

9. There are 4^k possible sets of $(k+2)$ patterns, one set for each possible "center" k-suffix in the block. To generate all such sets of patterns, we use Algorithm 4.1.

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 the set of patterns centered at any k-suffix z as $\mathcal{P}[z] = \{P_1, ..., P_{k-1}\}$. These can be used as bit-masks for blocks in the neighborhood bit-array of any l-mer x ending in z. Notice that the set $\mathcal{P}[z]$ does not include the empty pattern P_{-1} (all 0's), the full pattern P_k (all 1's), and the pattern P_0 in which only the center bit (the bit at position z) is set. This is to save space: since applying these patterns to blocks is trivial and does not require block masking, we need not generate and store them in $\mathcal{P}$.

4.2 Implementing a pattern-based speedup technique for EMS-GT

The previous section has shown that the bit-array N, representing the d-neighborhood of an l-mer, can be defined in terms of repeating block patterns. This definition can be used toward a larger goal—constructing the bit-array $\mathcal{N}$ which represents the d-neighborhood $\mathcal{N}(S, d)$ of a sequence S.

To do this, we first initialize the array $\mathcal{N}$ of 4^l bit flags, all set to zero. We select a value for the block degree k, and pre-generate all 4^k possible sets of block patterns (using Algorithm 4.1). Then, for every l-mer $x = yz$ in S:

1. We retrieve $\mathcal{P}(z) = \{P_1, ..., P_{k-1}\}$, the set of unique block patterns "centered" at x's k-suffix z. For convenience, this set excludes three "trivial" patterns: the empty pattern P_{-1} (all 0's), the full pattern P_k (all 1's), and the pattern P_0 in which only the center bit (the bit at position z) is set.

2. For every d-neighbor y' of x's prefix y, we locate the block—$block(\mathcal{N}, y')$—in $\mathcal{N}$ whose block prefix is y'. We then apply the appropriate pattern to the block, based on the value of $d - d_{y'}$:

   (a) If $d - d_{y'} = 0$, we set the "center" bit, i.e. the bit at position z in $block(\mathcal{N}, y')$.

   (b) If $0 < d - d_{y'} < k$, we mask the pattern $P_{d-d_{y'}}$ onto $block(\mathcal{N}, y')$.

   (c) If $d - d_{y'} \geq k$, we set all bits in $block(\mathcal{N}, y')$.

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 EMS-GT (Algorithm 2.1).

4.3 Improvement with pattern-based speedup technique

The pattern-based technique requires examining all d-neighbors of x's prefix y. We can use our recursive procedure (Algorithm 2.3) to traverse the neighborhood $N(y, d)$; as Table 1 shows, this will require generating much fewer neighbors than $N(x, d)$:

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

Table 4.1. Number of individual neighbors generated for $N(x, d)$ vs. $N(y, d)$

This large reduction in neighbors generated partly explains why, for most (l, d) values, building the neighborhood in blocks proves significantly faster (Table 4.2) than recursively generating each individual neighbor, then locating and setting its bit flag.

(l, d)	Without pattern-based procedure	With pattern-based procedure, $k = 5$	speedup
(9,2)	0.06 s	0.11 s	–
(11,3)	0.22 s	0.20 s	6.7%
(13,4)	1.98 s	1.04 s	47.5%
(15,5)	25.06 s	15.51 s	38.1%
(17,6)	308.61 s	175.85 s	43.0%

Table 4.2. Performance of EMS-GT with vs. without pattern-based speedup (runtimes averaged over 20 synthetic datasets for each (l, d) instance).

4.3.1 Determining the optimum block size

In the previous sections on distance-based patterns, the block degree k was set to 5, thus the size of blocks in N was $4^5 = 32 \times 32$. Based on empirical results in Table 4.3, $k = 5$ is in fact an optimal value for solving the (l, d) challenge instances used for evaluation:

(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)	-	-	**0.11 s**	0.57 s	9.02 s
(11,3)	-	-	**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 4.3. Performance of EMS-GT (with pattern-based speedup) for different values of k. Shortest times are in bold text.

When k is lower, there are fewer and smaller block patterns to manage, but building N requires many highly scattered array accesses to small blocks. When k is higher, there are larger contiguous blocks and thus fewer accesses in N, but also a much larger set of 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.

4.4 Performance of improved EMS-GT

Finally, the improved EMS-GT and two competitor algorithms were run on an Intel Xeon, 2.10 GHz processor (single core only). Their performance, averaged over 20 synthetic datasets for each (l, d) challenge instance, is outlined in Table 4.4:

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

Table 4.4. Runtimes of PMS8, qPMS9 and EMS-GT. Shortest times are in bold text.

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

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

Table 4.5. Performance of PMS8, qPMS9 and EMS-GT on non-challenging (l,d).

EMS-GT is highly competitive with PMS8 and qPMS9 on challenging problem instances, but as Table 4.5 shows, on non-challenging instances EMS-GT is not as efficient. This may be due to differing ways of 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) values.

CHAPTER V

Conclusion

This paper discusses a novel speedup technique for the EMS-GT algorithm, an exact solution to the planted motif problem for any arbitrary instance, l=17. A Java implementation, run on various challenge problem instances, shows the improved algorithm to be very competitive against state-of-the-art exact algorithms PMS8 and qPMS9. The speedup technique allows EMS-GT to outperform the current best algorithm, qPMS9, for the challenge problem instances (9,2), (11,3), (13,4) and (15,5) with a run-time reduction of at least 76%, 81%, 77% and 37% respectively for these instances, while ranking second to qPMS9 for test instance (17,6). Note that EMS-GT does not outperform PMS8 and qPMS9 on non-challenging (l,d) instances.

Directions for further research include: refining the bit-based storage mechanism to be able to represent the entire motif search space for $l > 17$; creating a multiprocessor version of EMS-GT to solve in parallel for larger values of (l,d); and delegating the bit-masking optimization and other bulk bit operations to the graphics card, as explored in [4], for faster performance.

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 Improved EMS-GT

```
/** 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 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 pmt;
  static int[] currNeighborhood;
  static int[] candidateMotifs;
  static long[][] lmerMappings;

  // for masking
  static int blockDegree = 5;
  static int multByRow;
  static int lmersInBlock;
  static int rowsInBlock;
  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) ) >>> 5); // (4^l)/32

    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 {
    multByRow = 2*blockDegree - 5;
```

```
  lmersInBlock = 1 << (2 * blockDegree);    // 4 ^ blockDegree
  rowsInBlock = lmersInBlock >> 5;      // 4 ^ blockDegree / 32
  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*32+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();
}
```

```
//=====================================================================
// COLLECT CANDIDATES: intersect d-neighborhoods of S[0] to S[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-k characters of l-mer
    suffix = 0;             // last  k 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;
    }

    // set blockOffsets, currBlockMasks
    currBlockRow = (int) (suffix / rowsInBlock);
    currBlockCol = (int) (suffix \% 32);
    currBlockMasks = blockMasks[(int)suffix];
    int blockStart = (int) (prefix << (2*blockDegree - 5));
    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 / rowsInBlock;
    currBlockCol = (int) suffix % 32;
    currBlockMasks = blockMasks[(int)suffix];
    blockStart = (int) prefix << (2*blockDegree - 5);
    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) throws Exception {
  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 << multByRow;
    int blockStart2 = (int) alt2 << multByRow;
    int blockStart3 = (int) alt3 << multByRow;

    // 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 k-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);
      }

    }
  }

//=====================================================================
// TRANSFORM L-MER SEQUENCES: map all l-mers in S[tPrime+1] to S[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;
      }

    }
  }

//=====================================================================
// SEARCH MOTIF: search each candidateMotif in S[tPrime+1] to S[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) << 5;
      for(int j=0; j < 32; 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;
  }
}
```