Abelian Pattern Matching in Strings

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik
von

Tahir Ejaz

Dortmund
2010

Tag der mündlichen Prüfung:
11.01.2010

Dekan:
Prof. Dr. Peter Buchholz

Gutachter:
Prof. Dr. Sven Rahmann, Technische Universität Dortmund
Prof. Dr. Sebastian Böcker, Friedrich-Schiller-Universität Jena

Abstract

Abelian pattern matching is a new class of pattern matching problems. In abelian patterns, the order of the characters in the substrings does not matter, e.g. the strings *abbc* and *babc* represent the same abelian pattern $a+2b+c$. Therefore, unlike classical pattern matching, we do not look for an exact (ordered) occurrence of a substring, rather the aim here is to find any **permutation** of a given **combination** of characters that represents the given abelian pattern.

In this thesis, we study the problem of abelian pattern matching in strings in a systematic manner, and present several algorithms for exact as well as approximate abelian pattern matching. We also present different strategies for indexing the input text to make the abelian pattern matching more efficient.

Contents

List of Figures

Chapter 1

Introduction

In the past few years, the abundance of the completely sequenced genomes has led to the idea of the comparison and the analysis of the whole genomes at gene level [10, 38].

Gene clustering [19, 25, 29, 32, 35] is one approach for this type of comparison and analysis. It is believed that the genes with similar functionality tend to occur close to each other, therefore, gene clustering is a helpful tool for finding the functionality of genes. In gene clustering, the aim is to find the genes that are located in close proximity of each other in many genomes, and the order of the occurrences of these genes is considered irrelevant.

Common intervals of permutations is another facet of the gene clusters [19], and a wide range of algorithms has been proposed for finding the common intervals of permutations in the recent years [3, 8, 18, 20, 33, 37]. A common interval of k-permutations ($k \geq 2$) is a k-tuple of intervals of these permutations that consists of the same set of elements.

Abelian patterns (also known as compomers [4], permutation patterns [12] and parikh vectors [1, 31]) are a generalized form of the gene clusters and the common intervals of permutations. In both the gene clusters as well as the common intervals of permutations, the number of the occurrences of each distinct gene (in a gene cluster) or each distinct element (of a common interval of permutations) is restricted to one; whereas in the case of the abelian patterns, the order of the characters in the patterns is irrelevant (just like in the case of a permutation the order of the elements is irrelevant), however, unlike the permutations, the same characters may appear multiple times in the abelian patterns (e.g. *accg* and *acgc* represent the same abelian pattern that comprises of two occurrence of the character c and one occurrence of each of the characters a and g).

The abelian patterns have already been studied in the context of pattern discovery [12, 23, 24, 28]. In this thesis, we study the problem of ***abelian pattern matching in strings***, which has not been studied systematically so far. We develop several algorithms for exact as well as approximate abelian pattern matching in strings and suggest different indexing strategies to pre-process the input text in order to make the abelian pattern matching more efficient.

1.1 Overview of the Thesis

The thesis is organized as follows: In the rest of this chapter, we define the problem of abelian pattern matching in strings in a formal manner and shed light on several combinatorial properties of the abelian patterns that we use later in the thesis.

In Chapter 2, we discuss several algorithms for abelian pattern matching that do not require preprocessing of the text. We present two fundamental approaches, the prefix based approach and the suffix based approach, to solve the problem of online abelian pattern matching. Later in the chapter, we present a parametrized suffix based algorithm for online abelian pattern matching that is an improvement over the original suffix based algorithm for abelian pattern matching in worst case situations. We also give a tight lower bound for the problem of online abelian pattern matching in the chapter.

In Chapter 3, we consider the problem of abelian pattern matching for the case when the input text is known beforehand; and we can pre-process the input text to make the process of abelian pattern matching more efficient. We discuss how an existing indexing scheme, the *parikh index*, can be adapted for the problem of abelian pattern matching. We also present a new data-structure , the *abelian tree*, for indexing the input text; and shed light on the tradeoff between the storage requirement and the query processing time for an abelian tree in contrast to different data-structures that are used for storing the outgoing edges at the internal nodes of the tree. Finally, we present an abelian tree with alternating levels to avoid the overhead of keeping the unnecessary edges when the size of the alphabet is large.

In Chapter 4, we consider the problem of finding the approximate abelian matches of a given abelian pattern. We define three error models, the substitution error model, the insertion/deletion error model and the minimum operations error model, to measure the degree of approximation in the matching substrings of a given abelian pattern. The substitution error model focuses on the approximate abelian matches of the fixed length m (where m is the length of the abelian pattern to be found), whereas the other two models allow for the matches of lengths other than m too. For all three error models, we present algorithms for online approximate abelian pattern matching in the chapter.

Finally, in Chapter 5, we discuss the contribution of our work, and conclude the thesis with pointers to further research possibilities.

1.2 Abelian Pattern Matching

The problem of *Abelian Pattern Matching* differs from *Classical Pattern Matching* in the sense that in case of classical pattern matching we seek for exact occurrences of a pattern substring in the given input string, and the order of the characters in the pattern substring is preserved while looking for a match. In case of abelian pattern matching, however, the order of the characters in the pattern substring does not matter. Hence *'abc'* and *'bac'* are considered matching (abelian) substrings. Here we are not looking for an exact (ordered) occurrence of a substring, *rather we want to find any* ***permutation*** *of a given* ***combination*** *of characters that forms our pattern substring.*

1.2.1 Formal Problem Definition

Formally, given an alphabet Σ, an *abelian pattern* is a function $P : \Sigma \longrightarrow \mathbb{N}$ that assigns a multiplicity to each character in Σ. We set $\Sigma_P := \{c \in \Sigma : P(c) > 0\}$, the set of characters occurring in the pattern, and call $|\Sigma_P|$ the *size* of the pattern. We write the pattern symbolically as $P = \sum_{c\in\Sigma} m_c\, c$, where $m_c = P(c)$ denotes the multiplicity of character c in the pattern. We call $m := |P| := \sum_{c\in\Sigma_P} m_c$ the *length* of the pattern. For example, over the alphabet $\Sigma = \{a, b, c, d\}$, the strings *abcb* and *bbca* match the same abelian pattern $P = (1, 2, 1, 0)$ (function specification in lexicographic order) or $P = 1a + 2b + 1c + 0d = a + 2b + c$ (symbolic sum specification).

Given an abelian pattern P and a text $T \in \Sigma^n$, the *abelian pattern matching problem* is to find all occurrences of P in T, i.e. all positions of substrings $S = T_i...T_j$ with $j - i + 1 = |P|$ such that the frequency of each character in S matches the specified frequency of that character in P. For $T = ababcccabaccbacdddba$, the pattern $P = 2a + b + 3c$ occurs at positions 3, 5 and 10.

1.2.2 Properties of Abelian Patterns

Abelian patterns are quite different from normal classical patterns. In this section we shed light on properties of abelian patterns.

- The number of the abelian patterns/strings of length m over an alphabet Σ can be viewed as the number of integer solutions to the equation

  $$x_1 + \cdots + x_{|\Sigma|} = m$$

  under the condition that $x_i \geq 0$ for all $i = 1, \ldots, |\Sigma|$. This number is $\binom{|\Sigma|+m-1}{m}$ [22]. Note that, for large values of m, this number is significantly smaller than the number of classical patterns of length m over the alphabet Σ, which is $|\Sigma|^m$. This is because of the fact that an abelian pattern can be spelled by more than one strings.

- Let S_P be the set of all strings that match an abelian pattern P, then we call S_P the *pattern set* of P and $|S_P|$ the *size* of the pattern set of P. For an abelian pattern $P = \sum_{i=1}^{|\Sigma|} m_{c_i} c_i$ of length m, the size of its pattern set can be computed as the multinomial coefficient:

  $$|S_P| = \binom{m}{m_{c_1}, \ldots, m_{c_{|\Sigma|}}}$$

Chapter 2

Online Abelian Pattern Matching

2.1 Introduction

In this chapter we discuss several algorithms for abelian pattern matching that do not require preprocessing of the text. In these algorithms, as in many other classical pattern matching algorithms [7, 21, 26], a sliding window of length m is moved along the input text T and checked for a possible pattern match.

The chapter is organized as follows: We begin with the description of the problem and give several related definitions in Section 2.2. In Section 2.3, we describe the general strategy and basic assumptions that we use in the algorithms presented in the chapter. Section 2.4 is about a prefix based algorithm for abelian pattern matching. In Section 2.5, we present a suffix based, Horspool type algorithm for abelian pattern matching. Section 2.6 sheds light on several lower bounds for online abelian pattern matching. In Section 2.8, we give a parameterized suffix based algorithm for abelian pattern matching, to improves the time complexity of the original suffix based algorithm in the worst case scenario. We conclude the chapter in Section 2.10.

2.2 The Problem

We are given an abelian pattern $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$ defined over an alphabet Σ with $\sigma := |\Sigma|$, and the task is to find all the starting positions of the strings corresponding to P in an input text $T \in \Sigma^n$.

Some Related Definitions

Here we give several definitions that we use later in the chapter.

Definition 1. *An abelian pattern* $P' = \sum_{i=1}^{\sigma} m'_{c_i} c_i$ *is an* abelian sub-pattern *of another abelian pattern* $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$ *if and only if* $m'_{c_i} \leq m_{c_i}$ *for all* $i = 1, 2, \ldots, \sigma$. *Symmetrically,* P *is called an* abelian super-pattern *of* P'.

Definition 2. *Given an abelian pattern* $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$ *and its abelian sub-pattern* $P' = \sum_{i=1}^{\sigma} m'_{c_i} c_i$, *the abelian pattern* $D_{(P,P')} := \sum_{i=1}^{\sigma} (m_{c_i} - m'_{c_i})\, c_i$ *is called the* difference pattern *between* P *and* P'.

Definition 3. *Given an abelian pattern* $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$, *the multiset* $\{m_{c_i} | \, c_i \in \Sigma_P\}$ *denoted by* M_P *is called the* multiplicity set *of* P.

Observation 1. *The length-*j *abelian sub-patterns of an abelian pattern* P *of length* m *have a many-to-one relationship with the integer partitions [2] of* $m - j$. *For each partition* λ *of* $m - j$, *there exists a distinct class* C_λ *comprising of (zero or more) length-*j *abelian sub-patterns of* P *such that the elements of* $M_{D_{(P,P')}}$ *have a one-to-one correspondence with the elements of* λ *for each* $P' \in C_\lambda$.

Example: Given an abelian pattern $P = 3a+2b+2c$ with $m = 7$, following are its length-4 abelian sub-patterns:

$$\begin{array}{rclcrcl} P'_1 &=& 2a+b+c & \qquad & P'_2 &=& 2a+2b \\ P'_3 &=& 2a+2c & & P'_4 &=& 3a+b \\ P'_5 &=& a+b+2c & & P'_6 &=& 3a+c \\ P'_7 &=& a+2b+c & & P'_8 &=& 2b+2c \end{array}$$

and following are the integer partitions of $3 = 7 - 4$:

$$\begin{array}{rcll} 3 &=& 3 & \qquad \text{(call this partition} \quad \lambda_1) \\ &=& 2+1 & \qquad \text{(call this partition} \quad \lambda_2) \\ &=& 1+1+1 & \qquad \text{(call this partition} \quad \lambda_3) \end{array}$$

The length-4 abelian sub-patterns of P are classified as under:

$C_{\lambda_1} = \{P'_8\}$, as

$$\begin{array}{rcl} \lambda_1 &=& \quad 3 \quad ; \text{ and} \\ & & \quad \updownarrow \\ M_{D(P,P'_8)} &=& \{\ 3\ \} \end{array}$$

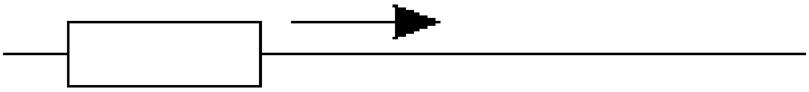

Figure 2.1: A window of length m is slid along the text

$C_{\lambda_2} = \{P'_2, P'_3, P'_4, P'_5, P'_6, P'_7\}$, as

$$\begin{array}{rcccccl} \lambda_2 & = & & 2 & + & 1 & \quad ;\ \text{and} \\ & & & \updownarrow & & \updownarrow & \\ M_{D_{(P,P'_i)}} & = & \{ & 2 & , & 1 & \} \quad \text{for } 2 \leq i \leq 7 \end{array}$$

$C_{\lambda_3} = \{P'_1\}$, as

$$\begin{array}{rcccccccl} \lambda_3 & = & & 1 & + & 1 & + & 1 & \quad ;\ \text{and} \\ & & & \updownarrow & & \updownarrow & & \updownarrow & \\ M_{D_{(P,P'_1)}} & = & \{ & 1 & , & 1 & , & 1 & \} \end{array}$$

Note that in case of length-3 abelian sub-patterns of P, if λ specifies the partition $4 = 4$, then C_λ is empty.

2.3 General Strategy

In the algorithms presented in this chapter, we slide a window of length m along the input text T, and check the window for a possible pattern match (Figure 2.1). We use three approaches for the procedure of checking for a possible pattern match inside the window:

Prefix based approach. In this approach we read the characters in the window one by one starting from the left end of the window. So at any time we have information about a prefix of the window.

Suffix based approach. Here we read the characters in the window one by one starting from the right end of the window. So at any time we have information about a suffix of the window. This approach may allow to skip some text characters from processing.

Parameterized suffix based approach. We use the suffix based approach in a parameterized manner; and at any point in time, we have information about at most two factors of the window.

In all the algorithms presented in this chapter, we use a frequency vector CFV *(current frequency vector)* which keeps the count of the characters read in the current window, and another frequency vector PFV *(pattern frequency vector)* which contains the count of the characters in the abelian pattern that is to be found. Both CFV and PFV can be implemented using linked lists, sorted arrays or directly accessible arrays. For a directly accessible array, the cost of query and increment/decrement operations in these vectors is $O(1)$ in the RAM model, and the memory requirement depends on the perfect hash function used for the direct accessibility feature; for a minimal perfect hash function [5, 6, 9, 27], the memory requirement is $O(\sigma)$. From now onwards we assume that their exists a minimal perfect hash function ρ for the characters in Σ, and both CFV and PFV are maintained as directly accessible arrays of size σ. Note that for the alphabets of English language, ρ is quite simple; it just subtracts a constant from the ASCII values of the characters.

2.4 Prefix Based Algorithm

In the prefix based algorithm, we set a window of size m at the beginning of the input text T and process the characters in the window in a prefix based manner. After we have processed the last character in the window, we check the current window for a match with the given pattern P. After that, the window is slid towards right by one position and checked again for the match. This way the window is slid through the whole text. As the $m-1$ length suffix of the current window equals the $m-1$ prefix of the next window, we construct the next window from the immediately preceding window in constant time. Pseudo code of this algorithm is presented in *Algorithm 1*.

In the first phase of this algorithm we initialize CFV with the first m characters of T. We also initialize the $mismatch$ for this CFV, where $mismatch$ counts the number of differences between CFV and PFV. If $mismatch$ is zero, this indicates that CFV is same as PFV and we output the first position of the text as starting position of a matching abelian pattern.

In the next phase, we proceed incrementally. We slide the window towards right by one position and update the contents of CFV with respect to the new window. Note that the new current window is different from the preceding window at two places. The first character of the previous window (call this character x) is not present in the current window; and the last character of the current window (call this character y) was not present in the previous window (Figure 2.2). If x is the same character as y, then the old contents of CFV also valid for the current window. However, if x is not the same

Algorithm 1 Prefix based Abelian Pattern Matching

Input: A pattern P of length m, a text stream $T = T[1] \dots T[n]$ and a hash function ρ

Output: Positions where the abelian pattern P begins in T

$\triangleright$ *Build current frequency vector (CFV) for the first m characters*

for $i = 1$ to σ **do**
$CFV[i] \leftarrow 0$
for $i = 1$ to m **do**
$CFV[\rho(T[i])] \leftarrow CFV[\rho(T[i])] + 1$

$\triangleright$ *Calculate the number of mismatching characters between the current window and the given pattern*

$mismatch \leftarrow 0$
for $i = 1$ to σ **do**
if $CFV[i] \neq PFV[i]$ **then**
$mismatch \leftarrow mismatch + 1$
if $mismatch = 0$ **then**
output 1
$i \leftarrow 2$
while $i \leq n - m + 1$ **do**
if $T[i-1] \neq T[i+m-1]$ **then**
Decrement $CFV[\rho(T[i-1])]$ by 1
if $CFV[\rho(T[i-1])] = PFV[\rho(T[i-1])]$ **then**
$mismatch \leftarrow mismatch - 1$
else if $CFV[\rho(T[i-1])] = PFV[\rho(T[i-1])] - 1$ **then**
$mismatch \leftarrow mismatch + 1$
Increment $CFV[\rho(T[i+m-1])]$ by 1
if $CFV[\rho(T[i+m-1])] = PFV[\rho(T[i+m-1])]$ **then**
$mismatch \leftarrow mismatch - 1$
else if $CFV[\rho(T[i+m-1])] = PFV[\rho(T[i+m-1])] + 1$ **then**
$mismatch \leftarrow mismatch + 1$
if $mismatch = 0$ **then**
output i
$i \leftarrow i + 1$

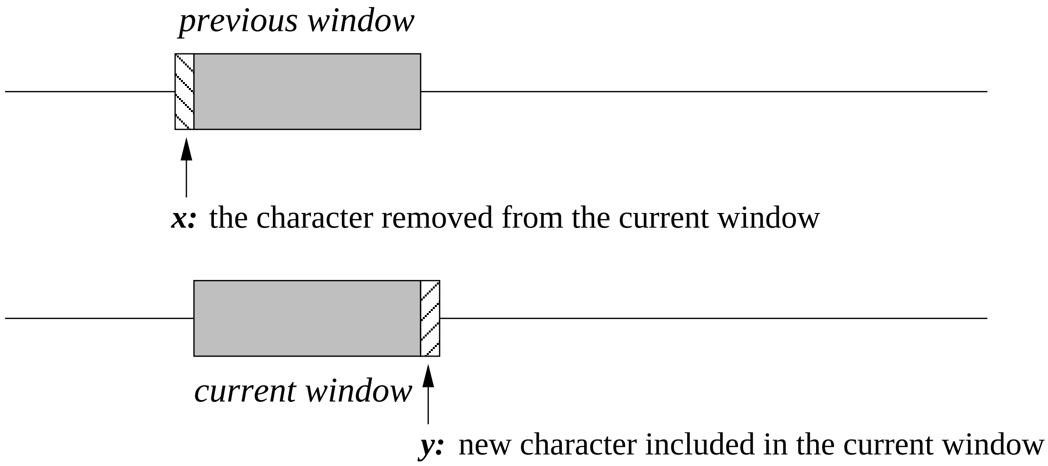

Figure 2.2: The current window removes one character of the previous window and includes one character that was not present in the previous window. The shaded area is the common part between the previous window and the current window.

as y, then we have to update the contents of CFV with respect to the new window.

As the current window does not contain the first character of the previous window, we decrement the frequency of x by 1 in CFV (*Line 14*). Now we see the effect of this change in the contents of CFV on the value of $mismatch$ (recall that $mismatch$ contains the number of differences between CFV and PFV).

Let the difference in the frequency of x in CFV and the frequency of x in PFV be d_x after the decrement operation has been performed on x.

- If $d_x = 0$, then it indicates that, before the decrement operation, the frequency of x in CFV was different from the frequency of x in PFV and therefore, a difference had been reported due to the character x during the computation of the existing value of $mismatch$. Now as the frequency of x in CFV equals the frequency of x in PFV after the decrement operation, we also decrement the value of mismatch by 1 in *line 16*.

- If $d_x = -1$, then it indicates that, before the decrement operation, the frequency of x in CFV was equal to the frequency of x in PFV and therefore, no difference was reported due to the character x during the computation of the existing value of $mismatch$. As, after the decrement operation, the frequency of x in CFV becomes different from the frequency of x in PFV, the number of elements having a difference between their respective frequencies in CFV and PFV is increased by 1. So we increment the value of mismatch by 1 in *line 18*.

- In all the other situations, the decrement in the frequency of x in CFV does not affect the value of *mismatch*. For example, if $d_x > 0$ or $d_x < -1$, then it indicates that, before the decrement operation, the frequency of x in CFV was already different from the frequency of x in PFV and therefore, a difference has already been reported due to the character x during the computation of the value of *mismatch*.

Similarly, as the current window includes a new character that was not part of the previous window, we increment the frequency of this new character y by 1 in CFV (*Line 19*). After that, we see the effect of the modification in the contents of CFV on the value of *mismatch*.

Let the difference in the frequency of y in CFV and the frequency of y in PFV be d_y after the increment operation has been performed on y.

- If $d_y = 0$, then it indicates that, before the increment operation, the frequency of y in CFV was different from the frequency of y in PFV and therefore, a difference had been reported due to the character y during the computation of the existing value of *mismatch*. Now as the frequency of y in CFV equals the frequency of y in PFV after the increment operation, we decrement the value of mismatch by 1 in *line 21*.

- If $d_y = 1$, then it indicates that, before the increment operation, the frequency of y in CFV was equal to the frequency of y in PFV and therefore, no difference was reported due to the character y during the computation of the existing value of *mismatch*. As, after the increment operation, the frequency of y in CFV becomes different from the frequency of y in PFV, the number of elements having a difference between their respective frequencies in CFV and PFV is increased by 1. So we increment the value of mismatch by 1 in *line 23*.

- In all the other situations, the increment in the frequency of y in CFV does not affect the value of *mismatch*. For example, if $d_y > 1$ or $d_y < 0$, then it indicates that, before the increment operation, the frequency of y in CFV was already different from the frequency of y in PFV and therefore, a difference has already been reported due to the character y during the computation of the value of *mismatch*.

If the updated value of *mismatch* is equal to zero, this means the substring contained in the current window corresponds to the given abelian pattern P; so we output the starting position of the current window as the beginning position of a matching abelian pattern in the input text (*Line 25*).

2.4.1 Time Complexity

The initialization of CFV in the *for loop* of *line 1-2* and then in the *for loop* of *line 3-4* is done in $O(\sigma + m)$ time. The value of *mismatch* is computed in the *for loop* of *line 6-8* in $O(\sigma)$ time.

The *while loop* of *line 12-26* has $O(n)$ iterations, and one iteration of the *while loop* requires $O(1)$ time, so the total time complexity of the *while loop* is $O(n)$.

Hence the total time complexity of the algorithm is $\Theta(\sigma+m+n) = \Theta(n)$.

2.4.2 Space Complexity

The algorithm uses two frequency vectors, CFV and PFV each requiring $\Theta(\sigma)$ storage; and one integer variable *mismatch*. Hence the space complexity of this algorithm is $\Theta(\sigma)$, in addition to the space required for the input and the output.

2.5 Suffix Based Horspool Type Algorithm

This algorithm is an adaptation of Horspool [21] type algorithms. Instead of reading the characters from the left towards the right, we read the characters from the right towards the left in the search window. While reading the characters from the right towards the left inside a window, as soon as an *overflow* of frequency in CFV occurs (i.e. the frequency of a character in CFV exceeds the specified frequency of the same character in PFV), we stop reading further in the window; as the current window cannot contain the given pattern P.

Observation 2. *Let S be the read suffix of a window after an overflow has occurred (the overflow character is the first character of S), then no substring of T that contains S, is a matching pattern.*

In the light of Observation 2, we can safely shift the current window towards the right such that the position of the second character of the read suffix becomes the starting position of the new window. Figure 2.3 illustrates the phenomenon of shifting a window after an overflow has occurred. After a window shift, we reset the contents of CFV (i.e. $CFV[i] = 0$ $for\ all\ i,\ 1 \leq i \leq \sigma$), and CFV corresponds to a new, blank window.

By using the technique of safely shifting the windows, we can skip several characters from processing (the shaded area in the *previous window* in Figure 2.3). However, there is also a danger of reading several characters multiple times if the overflow occurs very late in a window. Figure 2.4 illustrates this situation. Hence, the suffix based algorithm for abelian pattern matching is efficient only if the *sparseness of matches* holds (i.e. only a few substrings of the input string match to a given abelian pattern), because if this is not the case (i.e. the number of matches is significant) then the overflows will not occur frequently and this algorithm will not benefit much.

As mentioned earlier, after shifting a window after an overflow, we reset the content of CFV. A naive way to reset CFV after an overflow, is to blindly set the value of each element of CFV to zero; however, as the size of CFV is σ, this procedure requires $O(\sigma)$ time. Note that the only non-zero elements in CFV are those which correspond to the characters read in the suffix of the previous window, and the number of these characters is $O(m)$. So, we maintain a list *RCList* (*read characters list*) that keeps all the distinct characters that are read in a window; and when an overflow occurs, we set the value of only those element of CFV to zero which are in *RCList*. Note that under the suffix based approach, the number of elements in *RCList* at any time is $O(\sigma_P)$ (where σ_P is the number of distinct characters in P). With the help of *RCList*, CFV is reset in time $O(\sigma_P)$ which is in $O(m)$.

If we read all the character of a window without encountering an overflow, then we call this window a *safe window*.

Definition 4. *A* safe window *is a window of length m (where m is the length of the abelian pattern P), such that no overflow is occurred while reading the characters of the window.*

Lemma 1. *A safe window contains contains the abelian pattern P.*

Proof. Let S be a safe window and let CFV_S be the frequency vector corresponding to S. Let PFV be the frequency vector of the abelian pattern P.

We assume that S does not contain the abelian pattern P; this implies, there exists some j $(1 \leq j \leq \sigma)$, such that $CFV_S[j] \neq PFV[j]$.

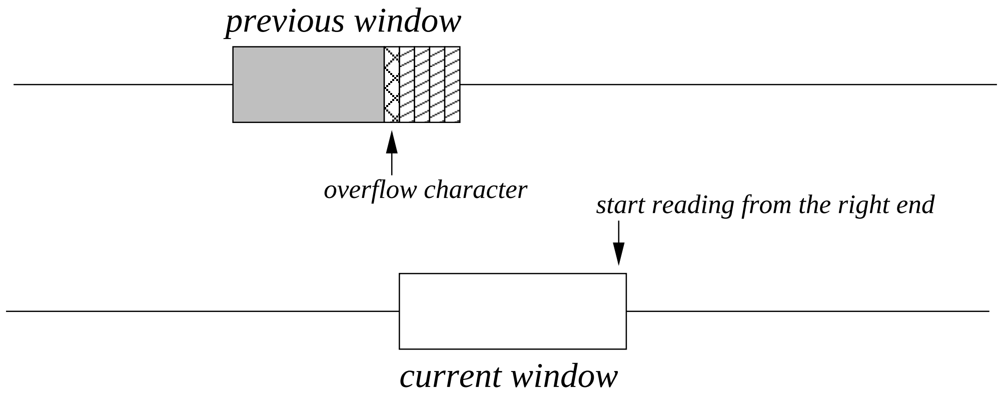

Figure 2.3: An overflow occurred while reading the window, and the window is shifted next to the position of the overflow character. The shaded area in the previous window shows the characters skipped from being processed due to the shifting of the window. Note that the current window is blank, i.e. it does not contain any information about the characters read in the previous window.

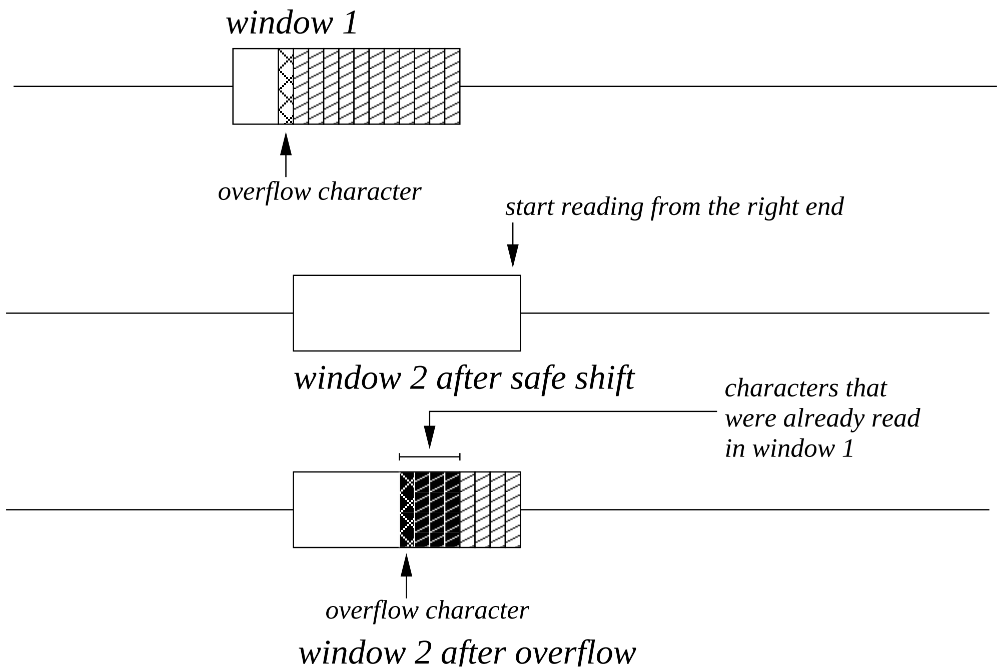

Figure 2.4: An overflow occurs very late in window 1, and several character that were read in window 1 had to be read again in window 2 before an overflow occurred in window 2.

Now we have the following information:

$$\sum_{i=1}^{\sigma} PFV[i] = m \qquad \text{[as the length of } P \text{ is } m] \tag{2.1}$$

$$CFV_S[i] \leq PFV[i] \text{ for all } i,\ 1 \leq i \leq \sigma \qquad \text{[as } S \text{ is a safe window]} \tag{2.2}$$

$$\text{There exist some } j\ (1 \leq j \leq \sigma),\ \text{such that } CFV_S[j] \neq PFV[j] \tag{2.3}$$

From (2.2) and (2.3), we can deduce:

$$\text{There exist some } j\ (1 \leq j \leq \sigma),\ \text{such that } CFV_S[j] < PFV[j] \tag{2.4}$$

And from (2.2) and (2.4), it comes that

$$\sum_{i=1}^{\sigma} CFV_S[i] < m \tag{2.5}$$

However, this is a contradiction, as, by definition, the length of S is m. Hence, S contains the abelian pattern P.

□

Hence, whenever we encounter a safe window during the execution of the algorithm, we output the starting position of that window as the beginning location of a matching abelian pattern in T. Pseudo code of the suffix based algorithm for abelian pattern matching is presented in *Algorithm 2*.

In the *for loop* of *line 1-2*, we initialize the vector CFV and in *line 3*, we initialize the linked list $RCList$.

In the *while loop* of *line 5-20*, we move the sliding window along the input text. In *line 6* we reset the $overflow$ flag and in *line 7* we set the pointer j to the last character of the current window.

In the *while loop* of *line 8-14*, we read characters in the current window, in a right to left direction, starting from the last character of the current window. After reading a character, we increment the frequency of the character by 1 in CFV (*line 9*), and if the current character is read first time in the current window, then we also insert it in $RCList$ (*line 11*). After that, we check whether the current character is an overflow character *line 15*, and if there is an overflow, then we set the $overflow$ flag (*line 13*) to stop reading further in the window.

Algorithm 2 Suffix based Abelian Pattern Matching

Input: A pattern P of length m, a text stream $T = T[1] \dots T[n]$ and a hash function ρ

Output: Starting positions of the abelian pattern P in T

```
for i = 1 to σ do
    CFV[i] ← 0
RCList ← ∅
i ← 1
while i ≤ n − m + 1 do
    overflow ← 0
    j ← i + m − 1
    while j ≥ i and overflow = 0 do
        Increment CFV[ρ(T[j])] by 1
        if CFV[ρ(T[j])] = 1 then
            insert T[j] in RCList
        if CFV[ρ(T[j])] > PFV[ρ(T[j])] then
            overflow ← 1
        j ← j − 1
    if overflow = 0 then
        output i
    i ← j + 2
    for all c ∈ RCList do
        CFV[ρ(c)] ← 0
        remove c from RCList
```

After we have finished reading inside the current window, if the current window is found to be a safe window (*line 15*), then we output the starting position of the current window (*line 16)*, because it contains a matching abelian pattern (Lemma 1).

In the last, we slide the window towards right in *line 17*:

- If the window is a safe window, then we shift the window towards right by one position (as $j = i - 1$ in this case, so $i = j + 2$ means $i = i + 1$).

- And if an overflow has occurred in the *while loop* of *line 11-17* then we move the starting position of the window next to the overflow character (j points to the character before the overflow character in this case).

After shifting the window, we remove all the characters read in the previous window from CFV in the *for loop* of *line 18-20*.

This way, we slide the window along the whole text and report the occurrence of a matching abelian patterns whenever we encounter a safe window.

2.5.1 Time Complexity

The *for loop* of *line 1-2* requires $O(\sigma)$ time. The *while loop* of *line 5-22* is the *main loop* of the algorithm. The time complexity of one iteration of the *main while loop* depends on the time complexity of the *while loop* of *line 8-14*.

The time complexity of the *while loop* of *line 8-14* ranges between $\Omega(1)$ (when an overflow occurs after reading a constant number of characters in the current window) and $O(m)$ (when an overflow occurs very late in the window or it does not occur at all).

The time complexity of the *for loop* of *line 18-20* in an iteration of the *main while loop* is proportional to the complexity of the *while loop* of *line 8-14* of that iteration, as the *for loop* removes the effect of only those characters from CFV in an iteration of the *main while loop*, which are read earlier in the *while loop* of *line 8-14* of that iteration.

The number of iterations of the *main while loop* ranges between $\Omega(n/m)$ (if the window is shifted towards right by $O(m)$ positions (*line 17*) in each iteration of the *main while loop*) and $O(n)$ (if the window is shifted towards right by $O(1)$ positions (*line 17*) in each iteration of the *main while loop*)

Best Case Time Complexity of the Algorithm

The best case occurs when, on average, we detect an overflow after reading a constant number of characters in each window; thus giving a best case time complexity of $\Omega(n/m)$.

Worst Case Time Complexity of the Algorithm

The worst case complexity of this algorithm is $O(nm)$, as we may need to read each character m times, i.e. the *main while loop* has $O(n)$ iterations and the time complexity of each iteration is $O(m)$.

Average Time Complexity of the Algorithm

The average case analysis of this algorithm depends heavily on the abelian pattern P. We begin with a lemma.

Lemma 2. *If on average we read ϵm characters in each window, then the time complexity of the suffix based abelian pattern matching is $O(\frac{n\epsilon}{1-\epsilon})$.*

Proof. We read ϵm characters in the window and advance the window by $(1-\epsilon)m+1$ positions. This gives us an $O(\frac{\epsilon}{1-\epsilon})$ cost for processing one character, and for the whole text this cost becomes $O(\frac{n\epsilon}{1-\epsilon})$. ☐

Theorem 1. *Let us assume that P is fixed and that the characters of the input text are independently and identically distributed, with probability $1/\sigma$ for each character at each position. Then the average case time complexity of the suffix based abelian pattern matching algorithm is*

$$O\left(\frac{n\sum_{k=0}^{m-1}|ASP(P,k)|/\sigma^k}{m-\sum_{k=0}^{m-1}|ASP(P,k)/\sigma^k|}\right)$$

where $ASP(P,k)$ denotes the set of strings of length k that match abelian sub-patterns of P.

Proof. If the overflow occurs after exactly k characters, we have read k characters and advanced the window by $m-k+1$ characters. Let J denote the random variable that describes the number of characters read in a window. Thus on average, in each iteration of the algorithm, the window is advanced by $m+1-\mathbb{E}[J]$ characters while examining $\mathbb{E}[J]$ characters.

The probability that an overflow occurs after $\leq k$ characters equals the probability that the rightmost k characters in the window are not an abelian sub-pattern of P:

$$\mathbb{P}(J \leq k) = 1 - |\mathrm{ASP}(P,k)|/\sigma^k$$

and

$$\begin{aligned}
\mathbb{E}[J] &= \sum_{k=0}^{m} k\,\mathbb{P}(J=k) \\
&= \sum_{k=1}^{m} \mathbb{P}(J \geq k) \\
&= \sum_{k=1}^{m} [1 - \mathbb{P}(J \leq k-1)] \\
&= \sum_{k=0}^{m-1} |\mathrm{ASP}(P,k)|/\sigma^k
\end{aligned}$$

As $\mathbb{E}[J] = \epsilon m$ in Lemma 2, therefore,

$$\epsilon = \frac{1}{m}\sum_{k=0}^{m-1} |\mathrm{ASP}(P,k)|/\sigma^k$$

By substituting the value of ϵ in Lemma 2, the average case time complexity of the suffix based algorithm for abelian pattern matching is:

$$O\left(\frac{\frac{n}{m}\sum_{k=0}^{m-1} |\mathrm{ASP}(P,k)|/\sigma^k}{1-\frac{1}{m}\sum_{k=0}^{m-1} |\mathrm{ASP}(P,k)/\sigma^k|}\right)$$

$$= O\left(\frac{\frac{n}{m}\sum_{k=0}^{m-1} |\mathrm{ASP}(P,k)|/\sigma^k}{\frac{m-\sum_{k=0}^{m-1} |\mathrm{ASP}(P,k)/\sigma^k|}{m}}\right)$$

$$= O\left(\frac{n\sum_{k=0}^{m-1} |\mathrm{ASP}(P,k)|/\sigma^k}{m-\sum_{k=0}^{m-1} |\mathrm{ASP}(P,k)/\sigma^k|}\right)$$

□

Now we show how $|\mathrm{ASP}(P,k)|$ can be computed by using the partitions of $\bar{k}$, where $\bar{k} := m - k$. Recall that the length-k abelian sub-patterns of an abelian pattern P of length m have a many-to-one relationship with the integer partitions of $\bar{k}$; and for each integer partition λ of $\bar{k}$, there exists a distinct class C_λ of length-k abelian sub-patterns of P (Observation 1).

We can generate the integer partitions of $\bar{k}$ by using any algorithm for generating integer partitions [13, 14, 39, 40]. For a partition $\lambda := \langle 1^{\alpha_1}, 2^{\alpha_2}, \ldots, \bar{k}^{\alpha_{\bar{k}}} \rangle \vdash \bar{k}$ (that is, $\bar{k} = \alpha_1 1 + \alpha_2 2 + \cdots + \alpha_{\bar{k}} \bar{k}$), we construct the abelian sub-patterns belonging to C_λ, and compute $|S_{P'}|$ for all $P' \in C_\lambda$. Recall that $S_{P'}$ is the set of all strings corresponding to the abelian pattern P', and $|S_{P'}|$ is computed using the multinomial coefficient (Section 1.2.2). We sum $S_{P'}$ for all $P' \in C_\lambda$. By iterating this procedure over all the partitions of $\bar{k}$, we obtain $|\mathrm{ASP}(P, k)|$. That is,

$$|\mathrm{ASP}(P, k)| = \sum_{all\ \lambda \vdash \bar{k}} \ \sum_{P' \in C_\lambda} |S_{P'}|$$

The procedure for computing the value of $|\mathrm{ASP}(P, k)|$ is outlined in Algorithm 3.

In *line 1* of the algorithm, we initialize the variable n that is to be used for storing the value of $|\mathrm{ASP}(P, k)|$. In each iteration of the *for loop* of *line 2-5*, we compute the number of strings corresponding to the abelian patterns in C_λ and add it to the existing value of n. When the for loop of line 2-5 is terminated, n contains the value of $|\mathrm{ASP}(P, k)|$.

The main processing of Algorithm 3 is done in the *Partition sub-routine*. The *sub-routine* computes the number of strings corresponding to the abelian patterns in C_λ for a given integer partition $\lambda := \langle 1^{\alpha_1}, 2^{\alpha_2}, \ldots, \bar{k}^{\alpha_{\bar{k}}} \rangle \vdash \bar{k}$ (that is, $\bar{k} = \alpha_1 1 + \alpha_2 2 + \cdots + \alpha_{\bar{k}} \bar{k}$). In other words, we compute $\sum_{P' \in C_\lambda} |S_{P'}|$ for a given λ, in the *Partition sub-routine*.

In *line 1* of the *sub-routine*, we select all those characters in P for which a value of l can be deducted from their multiplicities. If the number of such characters is less than α_l, we cannot decrement the multiplicities of the characters according to the given partition λ; hence cannot generate any length-k abelian sub-patterns of P corresponding to λ (i.e. C_λ is empty). At *line 4*, we initialize the variable *num*, which is used to store the number of strings corresponding to the abelian pattern in C_λ. At *line 5*, we have an abelian pattern of length m' ($m' = m$ if this is the first call of the *sub-routine*), and we fix exactly α_l characters from the characters that were selected at line 1. At *line 6*, we create a local copy of the abelian pattern received from the calling program. In the *for loop* of *line 7-8*, we decrement the multiplicities of the characters by l that were fixed at *line 5*. By doing so, we obtain an abelian pattern of length $m' - \alpha_l l$.

After that, we recursively call the *sub-routine*, with the new abelian pattern

Algorithm 3 Algorithm for computing $|\mathrm{ASP}(P, k)|$

Main Algorithm

Input: $\bar{k} := m - k$; abelian pattern $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$ of length m

Output: Number of strings in Σ^k that match abelian sub-patterns of P

1: $n \leftarrow 0$
2: **for** each integer partition λ of $\bar{k}$ **do**
3: $\quad \mathcal{C} \leftarrow \{c_1, \dots, c_\sigma\}$
4: $\quad \mathcal{M} \leftarrow \{m_{c_1}, \dots, m_{c_\sigma}\}$ (where m_{c_i} is the multiplicity of c_i in P)
5: $\quad n \leftarrow n+$ Partition $(\bar{k}, \lambda, \mathcal{M}, \mathcal{C})$
6: **return** n

Partition (l, λ, C, M)

1: $C' \leftarrow \{c_i \in C \mid m_{c_i} \geq l\}$
2: **if** $|C'| < \alpha_l$ **then**
3: $\quad$ **return** 0
4: $num \leftarrow 0$
5: **for** each distinct $C_{sub} = \{c_1, c_2, \dots, c_{\alpha_l}\} \subseteq C'$ **do**
6: $\quad M' \leftarrow \{m'_{c_i};$ such that $m'_{c_i} = m_{c_i} \in M$ for $1 \leq i \leq \sigma)$
7: $\quad$ **for** each $c \in C_{sub}$ **do**
8: $\quad\quad m'_c \leftarrow m'_c - l$
9: $\quad$ **if** $l = 1$ **then**
10: $\quad\quad num \leftarrow \binom{k}{m_{c'_1}, \dots, m_{c'_\sigma}}$
11: $\quad$ **else**
12: $\quad\quad num \leftarrow num+$ Partition$(l-1, \lambda, C \setminus C_{sub}, M')$
13: **return** num

of length $m' - \alpha_l l$ (*line 12*). While recursively calling the *sub-routine*, we also remove the character fixed at *line 5* from the set of characters whose multiplicities can be modified in the subsequent recursive executions of *Partition*. This means, in each path for the root to the leaf node of the recursion tree of the *sub-routine*, the multiplicity of a character is decremented at most once.

If $l = 1$, we have obtained an abelian pattern of length $m - \bar{k} = k$, and in *line 10*, we compute the number of strings that correspond to this length-k abelian pattern.

After a brief description of the working of *Partition*, now we present a panoramic view of the *sub-routine*.

Partition is a recursive sub-routine and it receives as argument:

- an integer l,
- an integer partition λ of $\bar{k}$,
- an abelian pattern P' of length m', and
- a set of characters C, such that the multiplicity of a character ch can be modified in the *sub-routine* if and only if $ch \in C$.

In the *sub-routine*, we generate all the distinct abelian sub-patterns of P' of length $m' - \alpha_l l$, by decrementing a value of l in the multiplicities of exactly α_l characters (this is done in the *for loop* beginning at *line 5* of the *sub-routine*).

The *sub-routine* then calls itself in a recursive manner, each time decreasing the value of l by 1.

The recursion stops when the value of l becomes 1.

Observation 3. *The depth of the recursion tree of the* sub-routine *Partition is* $\bar{k}$. *This is because of the fact that, in the first call of the* sub-routine $l = \bar{k}$, *in each subsequent call to the* sub-routine *the value of* l *is decremented by 1, and the recursion stops when the value of* l *becomes* 1.

Lemma 3. *Let* m_i *denotes the length of the abelian pattern received as argument at the beginning of the calls to the* sub-routine *at* i^{th} $(i \leq \bar{k})$ *level of the recursion tree; then*

$$m_i = m - \left(\sum_{l=\bar{k}-i+2}^{\bar{k}} \alpha_l l \right)$$

Proof. As explained before, *Partition* receives as argument an abelian pattern P' of length m', and it generates the abelian sub-patterns of P' of length $m - \alpha_l l$.

For the first call of *Partition* (which is made at *line 5* of the *main algorithm*), P is the abelian pattern received as argument, and $m = |P|$. Moreover, $l = \bar{k}$ in the first call to *Partition*.

So we generate the abelian sub-patterns of P of length $m - \alpha_{\bar{k}}\bar{k}$ in the first execution of *Partition*; and in the recursive calls to *Partition* (*line 12* of the *sub-routine*) that correspond to the second level of the recursion tree, we pass abelian patterns of length $m - \alpha_{\bar{k}}\bar{k}$ as argument. Moreover, $l = \bar{k} - 1$ in these calls to *Partition*.

The value of l for the recursive call to *Partition* at the $i - 1^{st}$ level of the recursion tree, is $\bar{k} - (i - 1) + 1 = \bar{k} - i + 2$ (as the value of l is decremented

by 1 in each subsequent recursive call to *Partition*, and $l = \bar{k}$ in the first call to *Partition*).
Therefore, in the recursive calls to *Partition* at i^{th} level of the recursion tree, we pass abelian patterns of length $m_{i-1} - \alpha_{\bar{k}-i+2}\ \bar{k} - i + 2$ as argument.

Now we have

$$m_i = m_{i-1} - \alpha_{\bar{k}-i+2}\ \bar{k} - i + 2$$

By a similar argument

$$m_{i-1} = m_{i-2} - \alpha_{\bar{k}-i+3}\ \bar{k} - i + 3$$

$$\vdots$$

$$m_2 = m_1 - \alpha_{\bar{k}}\bar{k}$$

$$m_1 = m$$

By substitution, we get

$$m_i = m - \left(\sum_{l=\bar{k}-i+2}^{\bar{k}} \alpha_l l\right)$$

$\square$

Corollary 1. *The length of the abelian sub-patterns generated by Partition, when the* sub-routine *is called at the last ($\bar{k}^{th}$) level of the recursion tree, is k. This is because of the fact that $m_{\bar{k}} = m - (\sum_{l=2}^{\bar{k}} \alpha_l l)$ (Lemma 3), and the length of the abelian sub-pattern generated in the execution of the* sub-routine *is $m_{\bar{k}} - \alpha_1 1$ (as $l = 1$ at the $\bar{k}^{th}$ level of the recursion tree).*

In the final recursive calls (the calls corresponding to the $\bar{k}^{th}$ level of the recursion tree) to *Partition*, we generate length-k abelian sub-patterns of P (Corollary 1); and then we compute the number of strings corresponding to these abelian pattern using the multinomial coefficient (*line 10* of the *sub-routine*). This number is returned to the calling *Partition sub-routine* in *line 13*.

We sum over all the values returned by *Partition* (*line 12*) and this gives us the number of strings corresponding to the k-length abelian sub-patterns of P that are member of C_λ. And finally this value is returned by the *sub-routine*, that was invoked from the *main algorithm* (i.e. the call to *Partition* that corresponds to the 1^{st} level of the recursion tree), to the calling statement in the *main algorithm* (*line 5* of the *main algorithm*). Figure 2.5 illustrates the working of the *Partition sub-routine*.

We compute the value of $|\mathrm{ASP}(P,k)|$ by summing over all the values returned by *Partition* to the main algorithm (*line 5* of the *main algorithms*), for all integer partitions of $\bar{k}$; and finally, we return the value of $|\mathrm{ASP}(P,k)|$ in line 6 of the main algorithm.

A Simple Algorithm to Compute $|\mathbf{ASP}(P,k)|$: After the design of the above mentioned algorithm to compute $|\mathrm{ASP}(P,k)|$, through an informal communication, we received a simpler idea to compute the value of $|\mathrm{ASP}(P,k)|$. In the following, we sketch that simple algorithm to compute $|\mathrm{ASP}(P,k)|$ (note that this algorithm is not part of this PhD work; and to the best of our knowledge, it has not been published elsewhere).

> **The Notion:** An abelian pattern is a tuple $P = (P_1, \dots, P_n)$, where $P_1, \dots, P_n$ represent the character multiplicities. n is the size of the pattern and $m := \sum_{i=1}^{n} P_i$ is the length of the pattern. A *partial pattern* is a subtuple of P, e.g. $P_{i,\dots,j} := (P_i, \dots, P_j)$ is a partial pattern of P.
>
> **Computation of Number of Abelian Sub-patterns:** Let $c(P,j)$ denotes the number of abelian sub-patterns that have length j, then $c(P,j)$ can be computed recursively:
>
> $$c(P,j) = \begin{cases} 1 & \text{if } n \leq 1 \\ \sum_{i=max(0,j-m+P_1)}^{min(j,P_1)} c(P_{2,\dots,n}, j-i) & \text{otherwise} \end{cases}$$
>
> The idea is to compute the allowed range of multiplicities for P_1 and iterate over this range. In each step, the multiplicity for P_1 is fixed to i and the number of $(j-i)$-length sub-patterns of $P_{2,\dots,n}$ is computed recursively.

To compute the value of $|\mathrm{ASP}(P,k)|$, we pass a local copy of the modified multiplicities of the characters on each recursive call and at the end of the recursion process (the case when $n \leq 1$), instead of returning 1, we compute the number of the strings corresponding to the abelian sub-pattern using the multinomial coefficient based on the modified multiplicities of the characters, and return this number.

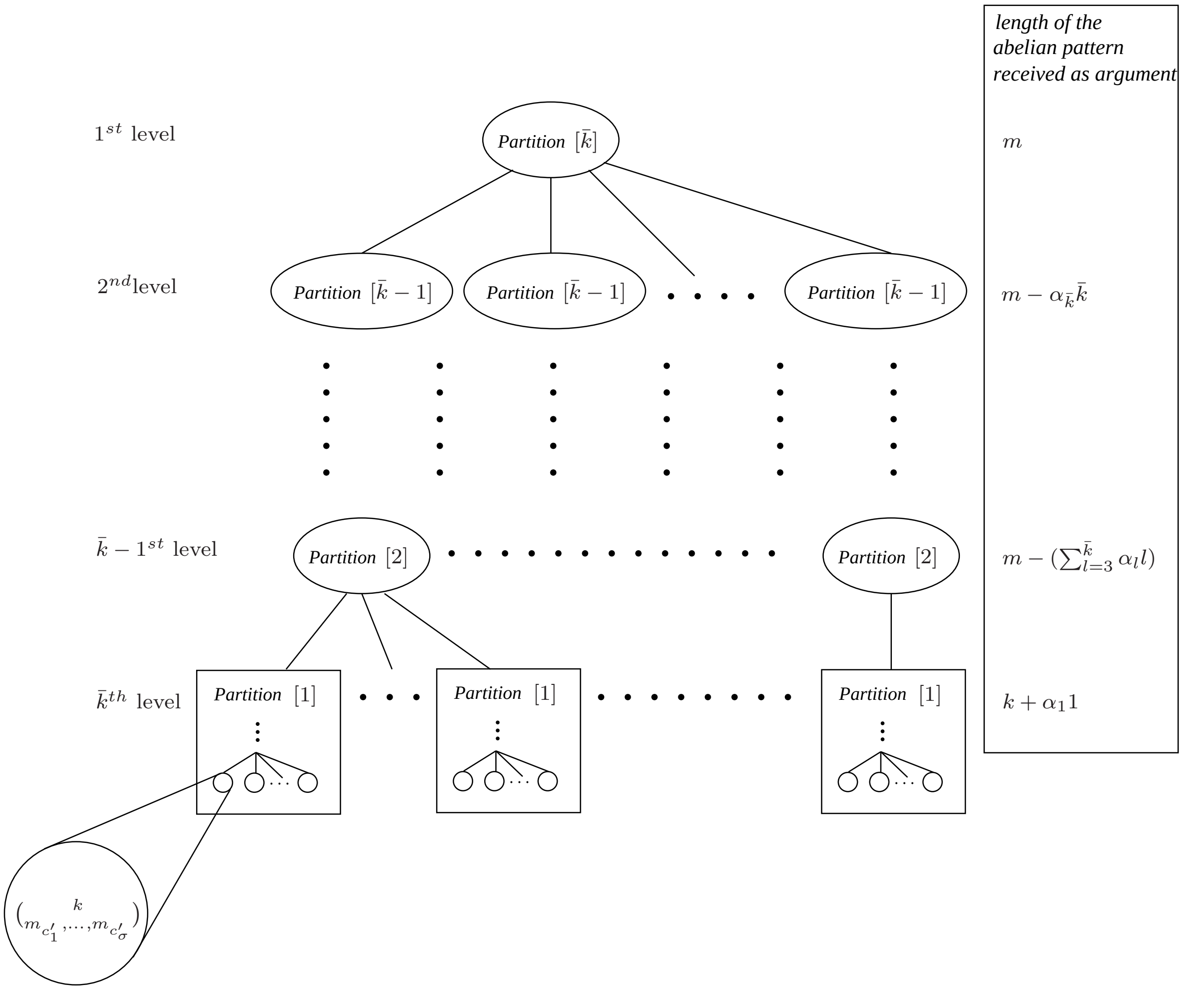

Figure 2.5: The recursion tree of the *sub-routine*, *Partition*. Each node represents a call to *Partition* and the value in the square bracket in a node represents the value of the argument l in that call to *Partition*. On the right hand side of the figure, the length of the pattern that is received as argument by *Partition* is mentioned, for the calls to *Partition* at different levels of the recursion tree. The leaf nodes of the tree (the calls to *Partition* at $\bar{k}^{th}$ level) receive as argument abelian patterns of length $m - (\sum_{l=2}^{\bar{k}} \alpha_l l) = k + \alpha_1 1$; and generated the sub-patterns of length k. After that, the number of strings corresponding to each k-length abelian pattern using the multinomial coefficient. These values are returned to the calling sub-routine, which sums over all the values returned from its child nodes. And finally, at the root node, the sum, of all the values computed at the lead nodes, is returned to the *main algorithm*.

2.5.2 Space Complexity

The algorithm uses two frequency vectors, CFV and PFV, a list $RCList$, and a constant number of variables. Both CFV and PFV require $\Theta(\sigma)$ storage, and $RCList$ requires $O(\sigma_P)$ storage. Hence the space complexity of this algorithm is $\Theta(\sigma)$, in addition to the space required for the input and the output.

2.6 Lower Bounds

The following can be stated regarding the lower bounds for online abelian pattern matching.

Theorem 2. *A lower bound for* best *case time complexity of any oblivious algorithm of abelian pattern matching in a given text of length n with pattern size m is $\Omega(\lfloor n/m \rfloor)$.*

A lower bound for worst *case time complexity of any oblivious algorithm of abelian pattern matching in a given text of length n with pattern size m is $\Omega(n)$.*

Proof. The best case bound is straight forward using a classical adversary argument.

For the worst case bound, assume that there exists an abelian pattern matching algorithm A that processes less than n/k characters of the input text, where k is an arbitrary constant. Given an abelian pattern P, consider an input text T such that there are at least n/km non-overlapping matching substrings in T. Then there exists at least one matching substring S in T such that not all of its characters are processed by A. As the algorithm is claimed to be correct, it must have output the starting position of S. Now if we replace any of the unread characters of S with an invalid character c (i.e. $c \notin \Sigma_P$) the output of A should remain unaffected; hence A is not a correct algorithm. □

2.7 Empirical Analysis of the Prefix Based and the Suffix Based Algorithms

We have given an insight into the time complexity of the suffix based algorithm for abelian pattern matching in Section 2.5.1. However, the complexity analysis given in the section is purely theoretical, and it assumes only a uniform distribution of the characters of Σ for the input text T.

In this section, we do an empirical analysis of the abelian pattern matching algorithms presented so far. The analysis is based on the actual CPU time taken by the executions of the algorithms. The experiments were performed on a computer having two "Intel® Core™2 Duo CPU E6750 @ 2.66 GHz" processors and 3.8 GiB memory, running Ubuntu 9.04.

We generated random texts comprising of 10000000 characters, for $\sigma = 4$ and $\sigma = 8$. For each Σ, two types of random texts were generated: one based on a uniform distribution of the characters of Σ and the other based on an arbitrary non-uniform distribution of the characters of Σ.

We also used real text to compare the performance of the algorithms. The real text used for the experiments, comprised of a collection of the plays of famous English writer William Shakespeare; these plays are available in text form on the web site `http://shakespeare.mit.edu/`. The real text was pre-processed and all the punctuation marks and white spaces were removed form the text. We also changed the upper case letters into lower case, thus making $\sigma = 26$. The post-processed text comprised of 3712565 characters.

For the comparison of the respective performances of the prefix based and the suffix based algorithms on the randomly generated input texts, we generated a variety of abelian patterns. Among these abelian patterns, there were patterns having frequency distribution of the characters very similar to the distribution of the characters in the input text, and there were patterns having frequency distribution of the characters very different from the distribution of the characters in the input text

For each abelian pattern, we performed 1000 iterations of each of the prefix based and the suffix based algorithms and took the mean values of the CPU time taken by the algorithms in 1000 iterations.

Following are the findings of the experiments on the random input texts:

- The prefix based algorithm outperformed the suffix based algorithm for abelian patterns whose characters had a frequency distribution similar to the frequency distribution of the characters in the input text, whereas, the suffix based algorithm outperformed the prefix based algorithm for abelian patterns whose characters had a frequency distribution significantly different from the frequency distribution of the characters in the input text.

- The suffix to prefix CPU time ratio ("CPU time taken by the suffix based algorithm / CPU time taken by the prefix based algorithm") ranged form 3.17 to 0.02, i.e. in the worst case, the suffix based algorithm was 3.17 times slower than the prefix based algorithm, whereas, in the best case, the suffix based algorithm was 50 times faster than the prefix based algorithm.

- The suffix to prefix CPU time ratio showed an interesting behavior. Let r be the suffix to prefix CPU time ratio of an abelian pattern P, then:

  - If, by increasing the pattern length m of P (without changing the underlying frequency distribution of the characters in P), r also increased; then for an abelian pattern $\hat{P}$ having suffix to prefix CPU time ratio $\hat{r} > r$, when the pattern length of $\hat{P}$ was increased, $\hat{r}$ also increased.
  - If, by increasing the pattern length m of P (without changing the underlying frequency distribution of the characters in P), r decreased; then for an abelian pattern $\hat{P}$ having suffix to prefix CPU time ratio $\hat{r} < r$, when the pattern length of $\hat{P}$ was increased, $\hat{r}$ decreased.

For the comparison of the relative performances of the two algorithms on the real text, we randomly selected substrings of various lengths from the input text and converted these substrings into equivalent abelian patterns. For each abelian pattern, we performed 1000 iterations of each of the prefix based and the suffix based algorithms and took the mean values of the CPU time taken by the algorithms in 1000 iterations.

Following are the findings of the experiments on the real input text:

- The suffix based algorithm outperformed the prefix based algorithm for all the abelian patterns selected for the experiments. The suffix to prefix CPU time ratio ranged from 0.31 to 0.07, i.e. in the worst case, the suffix based algorithm was 3.22 times faster than the prefix based algorithm, whereas, in the best case, the suffix based algorithm was 14.28 times faster than the prefix based algorithm.

- A general trend of decrease in the suffix to prefix CPU time ratio was seen when the pattern lengths were increased. The lowest suffix to prefix CPU time ratio for $m = 5$ was 0.23 which was 0.07 for $m = 50$.

We also compared the relative performances of the two algorithms on the real text against the abelian patterns corresponding to commonly used English words. The words were selected as follows:

> We picked the most frequent 1000 English words from the web site `http://www.duboislc.org/EducationWatch/First100Words.html` which are taken from [15]. We selected words of length ≥ 3 and for each word we computed its count in the input text. After that, from the words of the same length, we selected equal number of the most frequently and the least frequently occurring words in the text (we selected all of the words of length ≥ 9, as they were very few in number). These word were then converted into abelian patterns.

For each abelian pattern, we performed 1000 iterations of each of the prefix based and the suffix based algorithms and took the mean values of the CPU time taken by the algorithms in 1000 iterations.

Following are the findings of the experiments on the real input text for English words:

- The suffix based algorithm outperformed the prefix based algorithm for all the abelian patterns that were selected for the experiments. The suffix to prefix CPU time ratio ranged from 0.75 to 0.11, i.e. in the worst case, the suffix based algorithm was 1.33 times faster than the prefix based algorithm, whereas, in the best case, the suffix based algorithm was 9.09 times faster than the prefix based algorithm.

- A general trend of decrease in the suffix to prefix CPU time ratio was seen with an increase in the pattern lengths. Moreover, the suffix based algorithm performed better in case of the infrequent words than in case of the frequent words.

The detailed (pattern wise) results of the experiments for empirical analysis of the prefix based and the suffix based algorithms are given in Appendix A.

2.8 Parameterized Suffix based Algorithm

The main disadvantage of the suffix based algorithm is that it has to reset CFV after every overflow. In this section we present a parameterized suffix based algorithm that resets CFV only if the number of the characters read before an overflow does not exceed ϵm, where ϵ is a user defined parameter.

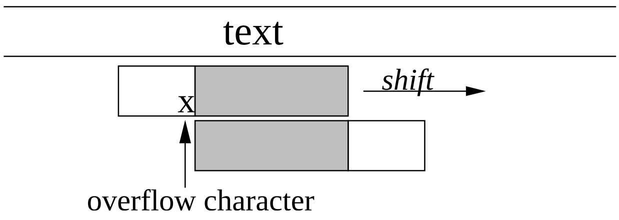

Figure 2.6: The gray area shows the information in CFV transferred from the previous window to the new window. Note that the character x is not part of this information.

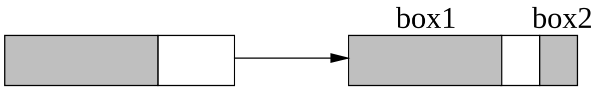

Figure 2.7: CFV contains collective information of a prefix and a suffix of the current window.

2.8.1 The Algorithm

Like the suffix based algorithm, we slide a search window of length m from the left towards the right along the input text T and process the characters inside the window in a right to left manner. In case an overflow occurs in this process, we stop further processing the current window and decrease the frequency of the current character, call it x, by 1 in CFV, so that, CFV again becomes compatible with PFV (i.e. $CFV[i] \leq PFV[i]$ for all i, $1 \leq i \leq \sigma$). We also shift the window to the right such that its new starting position coincides with the character next to x. So far the processing of this algorithm is same as that of the suffix based algorithm with the difference that we have decremented the frequency of x (which caused the overflow) by 1 in CFV in this algorithm. Note that CFV contains the information of the whole suffix (except x) that was read in the previous window, and this suffix is now prefix of the current window (Figure 2.6).

In the parameterized suffix based algorithm, we do not reset CFV blindly after an overflow has occurred. Instead, we consider the amount of information contained in CFV and if this information is less than or equal to ϵm (where ϵ is a user defined parameter) only then we reset CFV, otherwise we keep the information in CFV and start reading characters from the end position of the new current window. This latter case is illustrated in Figure 2.7: We have two information boxes in the window, *box 1* contains the information of a prefix of the window and *box 2* contains the information of a suffix of the window, whereas CFV contains the collective information of

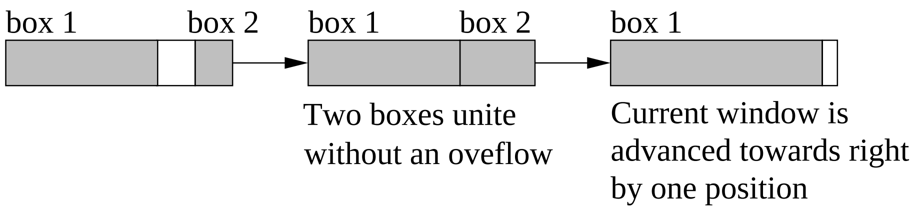

Figure 2.8: Box 2 unites with box 1 without an overflow. After reporting the current window as a matching substring, the current window is moved towards right by one position. The CFV contains information about an $m-1$ length prefix (representing *box 1*) of the new current window.

both boxes. Note that every time we read a new character in the window, *box 2* is extended towards the left.

If in this process both boxes unite without an overflow, then the current window contains a matching abelian pattern, and we output the starting position of the current window. We also decrement the frequency of the first character of the current window by 1 in CFV and advance the current window towards right by one position (Figure 2.8). However, if an overflow occurs while reading characters in the window, then the current window does not contain a matching substring and we search for the leftmost occurrence of the overflown character in the current window. We start reading the characters in the current window form its left end, and decrement the frequency of each read character by 1 in CFV until we read the overflow character. We shift the new starting position of the current window next to the latest read character. Note that now CFV does not contain information about any character outside the new current window.

Figure 2.9 illustrates three possible positions of the leftmost occurrence of the overflow character in the current window. It also shows the resulting window when the current window is shifted next to the leftmost occurrence of the overflow character. The dark gray regions in the figure show those characters whose count has been removed from CFV. Note that after this step, *box 2* is no longer a suffix of the resulting current window.

Here once again we have to decide whether or not to reset CFV. In case the collective information contents of both boxes (*box 1* possibly empty) are less than or equal to ϵm then we reset CFV, otherwise we keep the information in CFV. However, in the later case, if now we start reading from the end position of the current window, then we could have to manage three information boxes in those situations where *box 2* is not a prefix of the current window (Figure 2.9). To avoid this, we start reading characters from

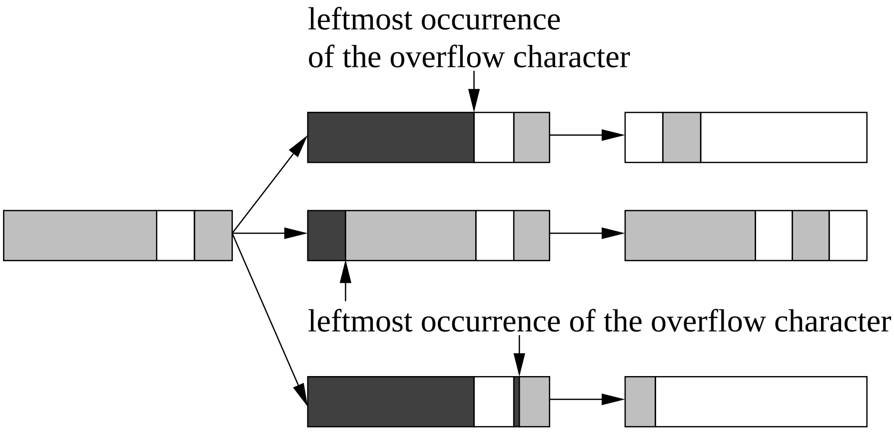

Figure 2.9: Three possible positions of the leftmost occurrence of the overflow character and the resulting windows after shifting the current window next to the overflow character.

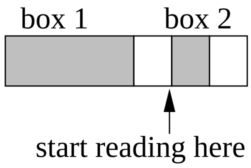

Figure 2.10: Filling the gap between two information boxes.

the last position of the gap between *box 1* and *box 2* in these situations, so that CFV once again contains information about only a prefix of the current window (Figure 2.10).

After the gap between *box 1* and *box 2* is filled, CFV contains information about only a prefix of the current window and then we start reading from the right end of the window creating *box 2* to hold information for the right most characters of the window (Figure 2.7). However, an overflow can occur before the gap is filled and it can lead to a loop situation until the information in CFV becomes less than ϵm or the gap is filled (Figure 2.11). Due to this mechanism we never have more than two information boxes at hand at any time.

In this way we keep on sliding the window along the input text until we reach the end of the text. Figure 2.12 illustrates this whole phenomenon.

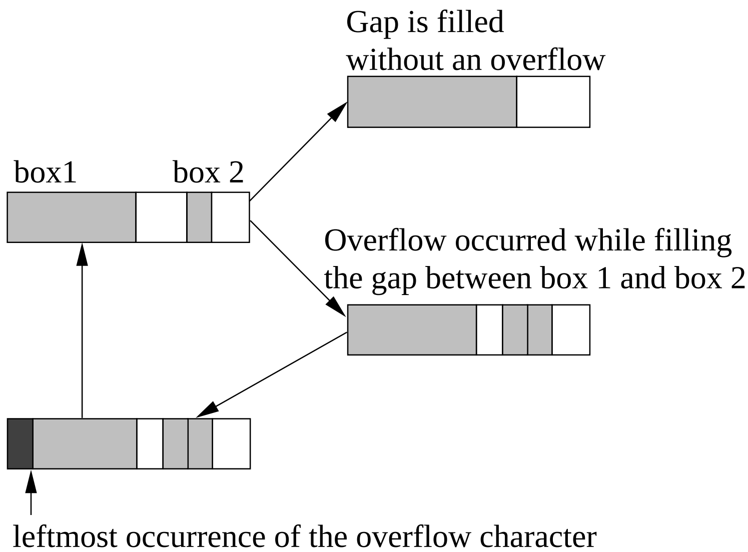

Figure 2.11: A loop situation while the gap between box 1 and box 2 is being filled.

2.8.2 Examples

To get a better understanding of the working of the parameterized suffix based algorithm, we present several examples and show how the algorithm works for each example using the transition graph presented in Figure 2.12.

In the following examples, we show different paths taken by the parameterized suffix based algorithm in the transitions graph of Figure 2.12 for certain input strings and abelian patterns.

Example 1 $(1 \rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6)$

Consider an input string *abcccacbb* and an abelian pattern $a + b + 3c$. Figure 2.13 shows how the parameterized suffix based algorithm proceeds along the transition graph presented in Figure 2.12 to find the matching abelian patterns in the text.

Example 2 $(1 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 9 \rightarrow 6)$

Now consider a different input string *abcdeacabecabababcde* and an abelian pattern $2a+3b+3c+d+e$. Figure 2.14 shows the transitions between states made by the parameterized suffix based algorithm in an attempt to find the matching abelian patterns in the text.

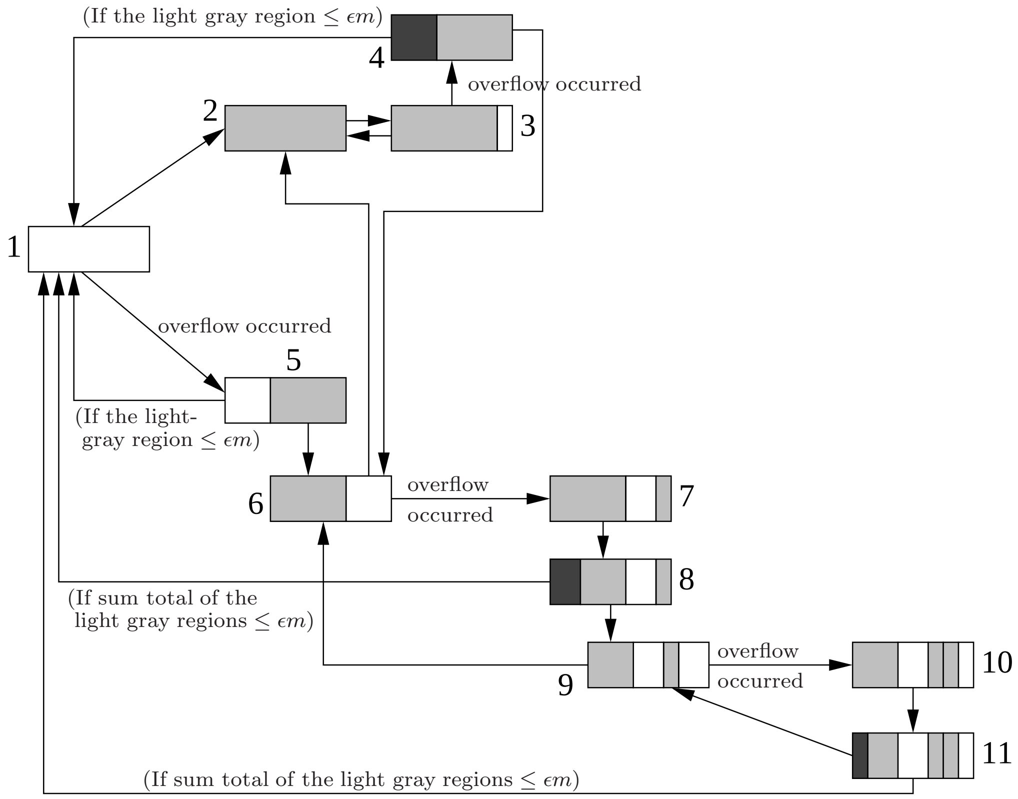

Figure 2.12: Complete transition graph of the parameterized suffix based algorithm with labeled states. In the figure, the light gray regions in a rectangle represent the characters read in the corresponding search window, hence the frequencies of these characters are incremented in CFV. The white regions in a rectangle represent the unread characters of the corresponding window. And the dark gray region in a rectangle represents the characters that occur before the leftmost occurrence of the overflown character in the window, it also includes the overflown character; the frequencies of these characters are decremented in CFV, and the current window is shifted next to the leftmost occurrence of the overflown character.

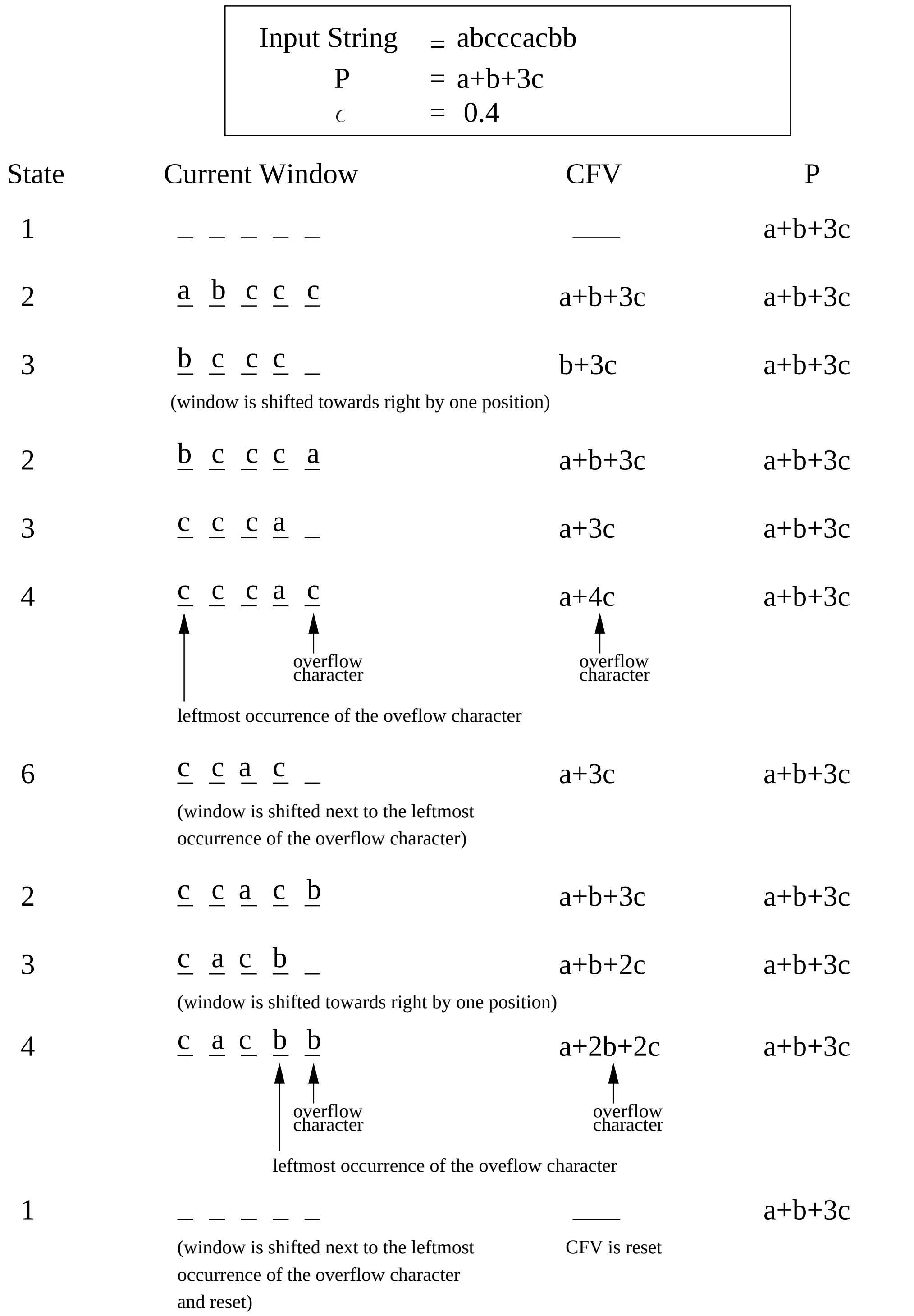

Figure 2.13: The path taken by the parameterized suffix based algorithm in the transition graph of Figure 2.12 for an input string *abcccacbb* and an abelian pattern $a + b + 3c$ with $\epsilon = 0.4$.

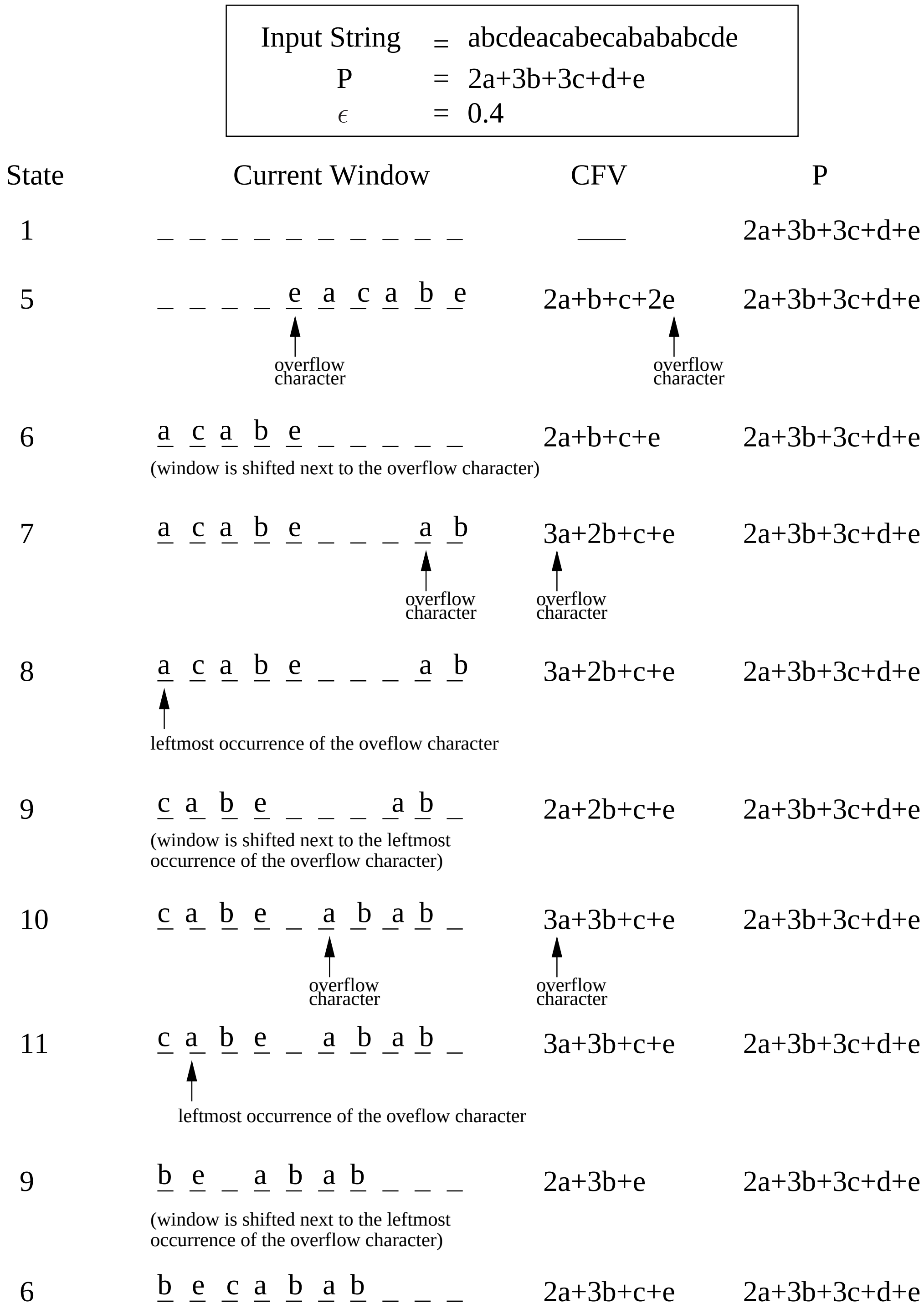

Figure 2.14: The path taken by the parameterized suffix based algorithm in the transition graph of Figure 2.12 for an input string *abcdeacabecababababcde* and an abelian pattern $2a + 3b + 3c + d + e$ with $\epsilon = 0.4$.

2.8.3 Complexity Analysis

The parameterized suffix based algorithm has the same best case complexity as that of the suffix based algorithm which is $\Theta(n/m)$. However, its worst case complexity is better than that of the suffix based algorithm.

Theorem 3. *The upper bound for* worst *case time complexity of the parameterized suffix based algorithm for abelian pattern matching in a given text of length n with pattern size m is $O(n/(1-\epsilon))$.*

Proof. If for a given input text, the parameterized suffix based algorithm operates in such a manner that the search window moves along the whole text without resetting the contents of CFV, then the time complexity of the algorithm on that input would be similar to the time complexity of the prefix based algorithm, which is $O(n)$.

However, if during the execution of the algorithm, it resets a window after an overflow, then we would have to process the reset characters again. In the parameterized suffix based algorithm, two type of *resets* occur:

1. The resets corresponding to a transition from *state 5* to *state 1* (Figure 2.12), and

2. The resets corresponding to a transition from any of the *states 4,8, or 11* to *state 1* (Figure 2.12)

In the resets corresponding to the transition from *state 5* to *state 1*, we read at most ϵm characters and advance the window by at least $(1-\epsilon)m$ positions, thus giving us a cost of $O(\epsilon/(1-\epsilon))$ per character.

In the resets corresponding to transitions from *states 4,8, or 11* to *state 1*, the cost to process one character can be computed as follows:

We start with a search window with no entry in CFV. Now we read X characters in the window and advance the window by $m-X$ positions. Note that $X > \epsilon m$, otherwise a reset corresponding to the transition from *state 5* to *state 1* would have taken place. We continue executing the algorithm and let Y be the number of characters processed (in addition to X) before the algorithm decides to reset the window. Let Z be the amount of information contained in CFV at the time of reset (clearly $Z \leq \epsilon m$). During this whole process, the window is advanced by $(m-X)+(X+Y-Z) = m+Y-Z$ positions along the input text. So we read $X+Y$ characters to advance the window by $m+Y-Z$ positions.

This gives the following cost per character:

$$
\begin{array}{rll}
 & (X+Y)/(m+Y-Z) & \\
\leq & (m+Y)/(m+Y-Z) & \text{(since } m \geq X\text{)} \\
\leq & (m+Y)/(m+Y-\epsilon m) & \text{(since } Z \leq \epsilon m\text{)} \\
\leq & m/(m-\epsilon m) & \text{(since } (m/m-\epsilon m) > 1 \text{ and } Y > 0\text{)} \\
= & 1/(1-\epsilon) &
\end{array}
$$

Hence the complexity of the parameterized suffix based algorithm is bounded by $O(n/(1-\epsilon))$ in the worst case.

□

2.9 Empirical Analysis of the Parameterized Suffix Based Algorithm for Different Values of Epsilon

In this section, we present an empirical analysis of the parameterized suffix based algorithm for abelian pattern matching for different values of ϵ. We used the same input texts and abelian patterns which we used for the empirical analysis of the prefix based and the suffix based algorithms (Section 2.7). The experiments were performed on a computer having two "Intel® Core™2 Duo CPU E6750 @ 2.66 GHz" processors and 3.8 GiB memory, running Ubuntu 9.04.

We executed the parameterized suffix based algorithm for four different values of ϵ, namely $\epsilon = 0.2$, $\epsilon = 0.4$, $\epsilon = 0.6$ and $\epsilon = 0.8$. As the actual reset threshold for parameterized suffix based algorithm is obtained by $\lfloor \epsilon m \rfloor$, therefore, more than one values of ϵ can result into the same amount of actual reset threshold. For example, for $m = 12$ and $\epsilon = 0.8$, the reset threshold is 9 characters, which shows an actual value of $\epsilon = 0.75$ in this case (i.e. $\epsilon = 0.8$ is same as $\epsilon = 0.75$ in this situation). We call $\epsilon = 0.8$ as the given value of ϵ and $\epsilon = 0.75$ as the actual value of ϵ. In the following, when we talk about ϵ then we mean the given value of ϵ.

We performed 1000 iterations of the parameterized suffix based algorithm (for each of the four values of ϵ) and took the mean value of the CPU time taken by the algorithm in 1000 iterations.

Following are the findings of the experiments on the random input texts:

- Generally, the higher values of ϵ ($\epsilon = 0.6$ and $\epsilon = 0.8$) performed better than the lower values ($\epsilon = 0.2$ or $\epsilon = 0.4$).

- For many patterns, there was not a significant difference between the CPU times taken by the parameterized suffix based algorithm for $\epsilon = 0.6$ and $\epsilon = 0.8$ (the best value for ϵ was either 0.8 or 0.6 in these situations).

- The value $\epsilon = 0.6$ was better than $\epsilon = 0.8$ in the sense that it also did a nice job in the situations where lower values of ϵ were the best. In these situations, $\epsilon = 0.8$ was the slowest.

- For many patterns, the CPU time taken by the parameterized suffix based algorithm was almost the same for all values of ϵ. Mostly, this situation occurred for the patterns whose frequency distribution was very different from the distribution of the characters in the input text. One explanation of this phenomenon could be that, for these patterns, most of the overflows occurred before reading ($\lfloor 0.2m \rfloor$) characters thus giving almost similar time for all values of ϵ.

Following are the findings of the experiments on the real input text:

- In the case of finding the abelian matches of the randomly selected substrings of the input text, there was not a significant difference between the CPU times taken by the parameterized suffix based algorithm for different values of ϵ. However, with the increase in the pattern length, the performance of the algorithm for $\epsilon = 0.2$ deteriorated compared to the performance of the algorithm for higher values of ϵ.

- In the case of finding the abelian matches of the commonly used English words, there was also not a significant difference between the CPU times taken by the parameterized suffix based algorithm for different values of ϵ. However, for some patterns, the algorithms used slightly more time for $\epsilon = 0.2$ than for other values of ϵ.

2.9.1 Relative Performance of the Parameterized Suffix Based Algorithm with Respect to the Prefix Based Algorithm and the Suffix Based Algorithm

The performance of the parameterized suffix based algorithm was in the middle of the prefix based and the suffix based algorithm; i.e. for the patterns for which the prefix based algorithm was the most efficient algorithm, the parameterized suffix based algorithm performed better than the suffix based algorithm, however, for the patterns for which the prefix based algorithm was the least efficient algorithm, the parameterized suffix based algorithm performed slightly worse than the suffix based algorithm.

However, in many cases, there was not a significant difference between the CPU times taken by the suffix based algorithm and the parameterized suffix based algorithm.

The detailed (pattern wise) results of the experiments for empirical analysis of the parameterized suffix based algorithm are given in Appendix B.

2.10 Conclusion

In this chapter we have presented two fundamental approaches to solve the problem of online abelian pattern matching. We have also given a tight lower bound for the problem. Finally, we have presented a parametrized suffix based algorithm for online abelian pattern matching that is an improvement over the original suffix based algorithm for pattern matching in worst case situations.

We have also outlined a procedure to compute the value of $|\mathrm{ASP}(P,k)|$ (where $\mathrm{ASP}(P,k)$ denotes the set of strings of length k that match abelian sub-patterns of P). In the procedure, we generate all integer partitions of $m-k$ ($m := |P|$) to compute the value of $|\mathrm{ASP}(P,k)|$. There exist, however, other (unpublished) methods to compute the value of $|\mathrm{ASP}(P,k)|$, which use only those integer partitions of $m-k$ that correspond to non-empty classes of length-k abelian sub-patterns of P.

Finally, we presented an empirical analysis of the relative efficiency of the algorithms presented in the chapter.

Chapter 3

Offline Abelian Pattern Matching

3.1 Introduction

In the previous chapter, we have presented several algorithms that solve the problem of abelian patten matching when the input text is given online. We were given an input text stream $T \in \Sigma^n$ and an abelian pattern P of length m; and we output the positions in T where the matches of P began.

In this chapter we consider the problem of abelian pattern matching for the case when the text T is known beforehand and we can pre-process the text to make the process of abelian pattern matching more efficient.

The chapter is organized as follows: In Section 3.2, we discuss an indexing scheme *parikh index* that is already in use for the problem of abelian pattern discovery. We show how we use a parikh index, which is built for a text T, for solving the problem of abelian pattern matching in T. The abelian pattern matching can be done in time logarithmic to n ($n := |T|$) if a parikh index for T is available. In Section 3.3, we present a new data structure named *abelian tree* to store the information generated from the pre-processing of the text. With this data structure, the problem of abelian pattern matching is solved in time independent of n. We also discuss the impact of using different data-structures for constructing an abelian tree, on the tree construction time and space requirements as well as on the performance of the query processing. We conclude the chapter in Section 3.5.

We assume that $m := |P|$ is known in advance. If this is not the case, one can iterate over a range of different sizes to build multiple indices by pre-processing the text. We also assume a pre-defined linear ordering of the alphabet Σ, such that $c_1 < c_2 < c_3 < \cdots < c_\sigma$, where $\sigma := |\Sigma|$.

3.2 Parikh Index

The parikh index has already been used in the context of abelian pattern discovery [12]. Here we show how a parikh index (which is built using the parikh mapping technique [1]) can be used for abelian pattern matching.

In the parikh index, distinct abelian patterns of the same length are assigned unique names, and the indexing of the text is done on the basis of these names. For the sake of simplicity, it is assumed that σ is a power of 2, however, if this is not the case, σ is made a power of 2 by adding new characters to Σ. The new value of σ would not be more than the double of its original value.

In the following, we give definitions that we use to explain the procedure of computing the name of an abelian pattern using the parikh mapping technique. Recall that we assume a pre-defined linear ordering of the alphabet Σ, such that $c_1 < c_2 < c_3 < \cdots < c_\sigma$, where $\sigma := |\Sigma|$.

Definition 5. *Given an abelian pattern $P_{(x,y)} = \sum_{i=x}^{y} m_{c_i} c_i$ $(1 \leq x \leq y \leq \sigma)$, its k-*pRefix *is defined to be the abelian pattern $R_{k,\, P_{(x,y)}} = \sum_{i=x}^{x+k-1} m_{c_i} c_i$ for all k, $0 < k \leq y - x + 1$.*

Definition 6. *Given an abelian pattern $P_{(x,y)} = \sum_{i=x}^{y} m_{c_i} c_i$ $(1 \leq x \leq y \leq \sigma)$, its k-*Suffix *is defined to be the abelian pattern $S_{k,\, P_{(x,y)}} = \sum_{i=y-k+1}^{y} m_{c_i} c_i$ for all k, $0 < k \leq y - x + 1$.*

The names of the abelian patterns are computed recursively in the parikh mapping technique. The name of an abelian pattern $P_{(x,y)} = \sum_{i=x}^{y} m_{c_i} c_i$ is computed using the name-pair of the abelian patterns $R_{\frac{y-x+1}{2},\, P_{(x,y)}}$ and $S_{\frac{y-x+1}{2},\, P_{(x,y)}}$. The recursion goes on until the value of x equals y; and the abelian pattern $P_{(x,x)}$ gets m_{c_x} as its name.

In the parikh index, natural numbers, in an increasing order, are assigned as names to the abelian patterns. Let (a, b) (both a and b are natural numbers) be a name-pair that is encountered first time while building the parikh index, and let c is the highest natural number that is assigned as a name to any name-pair seen so far, then the pair (a, b) gets $c + 1$ as its name.

Example 1. *Let $\Sigma = \{a, b, c, d\}$, with the ordering $a < b < c < d$, and let the highest natural number that has been assigned so far as a name be 15.*

Now, if we want to compute the parikh name of the abelian pattern $P = a + 2b + d$, then we need to compute the names of the abelian patterns $1a + 2b$

and $0c + 1d$*. To compute the name of* $1a + 2b$*, we compute the names of the abelian patterns* $1a$ *and* $2b$*. As* $1a$ *and* $2b$ *are single character abelian patterns, so the recursion process stops here and* $1a$ *gets* 1 *as its name (1 is the multiplicity of* a *in the abelian pattern) and* $2b$ *gets* 2 *as its name (2 is the multiplicity of* b *in the abelian pattern). This means, the name of the abelian pattern* $1a + 2b$ *is computed using the name-pair* $(1, 2)$*. We assume that the name pair* $(1, 2)$ *is encountered first time during the process of parikh naming the substrings of the input text, and therefore, this name-pair is given* 16 *as its name (recall that 15 was the highest natural number that has been assigned as a name). So the parikh name of the abelian pattern* $1a + 2b$ *is 16.*

Similarly, to compute the name of $0c + 1d$*, we compute the names of the abelian patterns* $0c$ *and* $1d$*; and* $0c$ *gets* 0 *as its name and* $1d$ *gets* 1 *as its name. So, the name of the abelian pattern* $0c + 1d$ *is computed using the name-pair* $(0, 1)$*. We assume that the name pair* $(0, 1)$ *has already been encountered during the process of parikh naming the substrings of the input text, and it got* 11 *as its name when it was seen the first time. So the parikh name of the abelian pattern* $0c + 1d$ *is 11.*

Finally, the name of the abelian pattern $1a + 2b + 0c + 1d$ *is computed using the name pair* $(16, 11)$*. As the name pair* $(16, 11)$ *is also first time encountered during the process of parikh naming the substrings of the input text (because the name 16 is given the first time), so it gets* 17 *as its name.*

Hence the parikh name of the abelian pattern $a + 2b + d$ *is 17.*

In order to compute the names of all the abelian patterns of size m in a given text, a window of size m is moved along the text and the abelian pattern contained in the window is given a name using the recursive procedure mentioned above.

The names of the abelian patterns are stored in a directly accessible array, and a list of pointers is appended to each entry of the array that corresponds to the name of an abelian pattern of length m. This list of pointers contains the starting positions of the m-length abelian pattern, that corresponds to this name, in the input text.

This way, the whole text is indexed on the basis of the parikh names of the abelian patterns of length m.

3.2.1 Using Parikh Index for Abelian Pattern Matching

When we receive a query for finding the locations of an abelian pattern P of length m in a given text T, such that a parikh index for T for m-length abelian patterns is already in place, then the only thing we need to do is to compute the parikh name of P.

We compute the name of P using the same recursive procedure that was used to build the parikh index on T. However, as we are doing the pattern matching here and not the pattern discovery, therefore, if we encounter a name-pair (a, b), in the process of name computation of P, such that no name exists for the pair in the index, then we do not assign a new name to (a, b), rather we assert that P has no occurrence in T.

If the name of P is computed successfully, then we output the list of pointers associated with this name in the parikh index, as the pointers in this list point to the starting positions of P in T.

3.2.2 Complexity Analysis:

A parikh index for an input text of length n is built in time $O(n \log \sigma \log n)$[12] and the name of a given pattern P is computed in time $O(\sigma \log n)$.

As σ is a power of 2, so the name of an abelian pattern $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$ is computed using the names of $R_{\sigma/2, P}$ and $S_{\sigma/2, P}$; and the names of $R_{\sigma/2, P}$ and $S_{\sigma/2, P}$ are also computed recursively. The recursion tree for the computation of the name of P has $O(\log \sigma)$ levels. As this recursion tree is a complete binary tree, we compute $O(\sigma)$ names before we get the name of P.

The time complexity to compute one name using the information stored in the parikh index is $O(\log n)$[12], so the time complexity to compute the name of P is $O(\sigma \log n)$.

Hence, given an input text $T \in \Sigma^n$ and a parikh index built on T for the abelian patterns of length m, we can assert in time $O(\sigma \log n)$ whether or not P occurs in T. If P occurs in T, then the indices where P begins in T are output in time proportional to the frequency of P in T.

3.2.3 Building Multiple Indices

The assumption that the length of the abelian pattern to be found in the input text is known in advance is quite unrealistic. To tackle this issue, one can build multiple indices each for the abelian patterns of different length.

The time complexity to build parikh indices on T for a range of L different lengths is $O(Ln \log \sigma \log n)$. The time complexity to determine whether or not an abelian pattern P ($|P| \in L$) occurs in T remains $O(\sigma \log n)$. Moreover, if P occurs in T, then the positions where P begins in T are output in time proportional to the frequency of P.

3.3 Abelian Tree Indexing

Although parikh index is a useful index for abelian pattern matching, the primary intention for inventing the parikh index was to use it for abelian pattern discovery. Therefore, the emphasis of a parikh index is on the efficient construction of the index.

In this section, we present a new data-structure, named *abelian tree*, to index the input text $T \in \Sigma^n$. As the primary objective of inventing the abelian tree is to use it for abelian pattern matching, the emphasis here is on efficient query processing. With the help of an abelian tree, we can assert in time $O(\sigma)$ whether or not a given abelian pattern P occurs in T, thus giving a performance efficiency of $\log n$ factor, over the parikh index, for abelian pattern matching.

The concept of an abelian tree is adapted from the concept of a trie (also known as prefix tree) [34] for a set of strings. In the context of classical pattern matching, a trie is a rooted directed tree, and each node of this tree is associated with a string. Moreover, the decedents of a node N have common prefix, which is the string associate with N. Figure 3.1 illustrates a trie, for classical pattern matching, for the set of strings $\{abbc, abcb, abba\}$.

We begin with a definition of an abelian prefix and then outline our data-structure.

Definition 7. *Given an abelian pattern $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$ and a pre-defined linear ordering of the alphabet Σ, such that $c_1 < c_2 < c_3 < \cdots < c_\sigma$, the abelian pattern $Pre = \sum_{i=1}^{k} m_{c_i} c_i$ $(1 \leq k \leq \sigma)$, is called an* abelian prefix *of P.*

Definition 8. *Given a set of length-m abelian patterns $\mathcal{S} = \{S_1, \ldots, S_n\}$, the* abelian tree *$\mathcal{T}_{\mathcal{S}}$ is the unique trie that contains all the patterns in $\mathcal{S}$. The internal nodes of the abelian tree represent the characters of Σ, and the edges of the tree are labeled with the multiplicities of the characters. At the leaf nodes are lists of pointers to the occurrences of the abelian pattern corresponding to the path from the root to this leaf.*

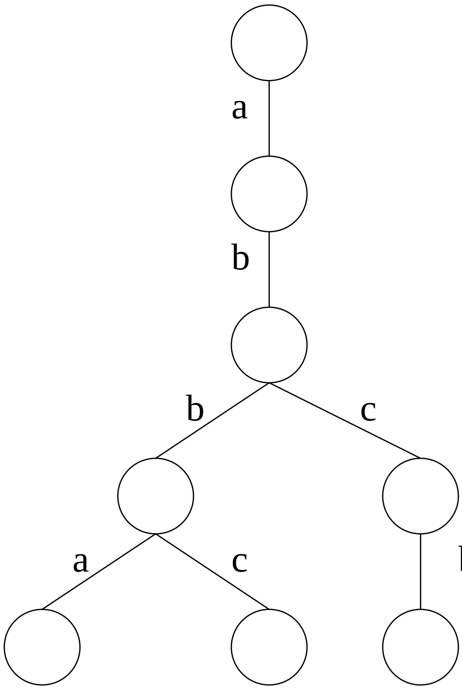

Figure 3.1: The trie for the strings *abbc*, *abcb* and *abba*

Figure 3.2 illustrates an abelian tree for the abelian patterns $a + 2b + c$ and $2a + 2b$.

Observation 4. *An abelian tree for m-length abelian patterns, defined over alphabet Σ, has $\sigma + 1$ levels and an outdegree not higher than $m + 1$.*

3.3.1 Construction of The Abelian Tree

For an input text $T \in \Sigma^n$, the abelian tree corresponding to T for length-m abelian patterns is constructed as follows:

A window of length m is moved along the input text and the abelian pattern corresponding to the string contained in the window is located top-to-bottom in the tree. If the leaf node corresponding to the abelian pattern contained in the current window does not exist in the abelian tree, then we look for a node N that corresponds to the longest abelian prefix of the current abelian pattern. We create a new path in the tree starting from N to the leaf node corresponding to the abelian pattern contained in the current window.

We append a pointer to the starting position of the current window in T to the leaf node corresponding to the abelian pattern contained in the current

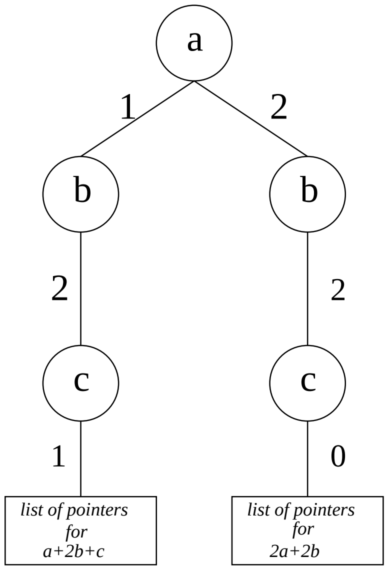

Figure 3.2: Abelian tree for the strings *abbc*, *abcb* and *abba* and the alphabet order $a < b < c$. The internal nodes are labeled in the figure just to illustrate that the nodes at the same level correspond to the same character of the alphabet. These node labels are not stored internally in the abelian tree.

window. An abelian tree is shown in Figure 3.3 for the abelian patterns of length 2 for the input text "abbbcabbabbcccca", with the ordering of the characters in Σ being $a < b < c$.

Query Processing: When we receive a query on the locations of an abelian pattern P, we search the leaf node corresponding to P in the abelian tree. If the leaf node corresponding to P does not exist in the tree, this implies P has no occurrence in the text; otherwise, we output the list of pointers stored at this leaf node.

3.3.2 Complexity Analysis

The time complexity for the construction of an abelian tree, and in turn for doing the pattern matching, depends on the data-structure used to keep the outgoing edges at an internal node.

In the following, we discuss different possibilities for the data-structures to be used for storing the outgoing edges at the internal nodes of the tree. We shed light on the the time and the space complexities of the abelian tree construction using the specific data-structure; and explain the time complexity

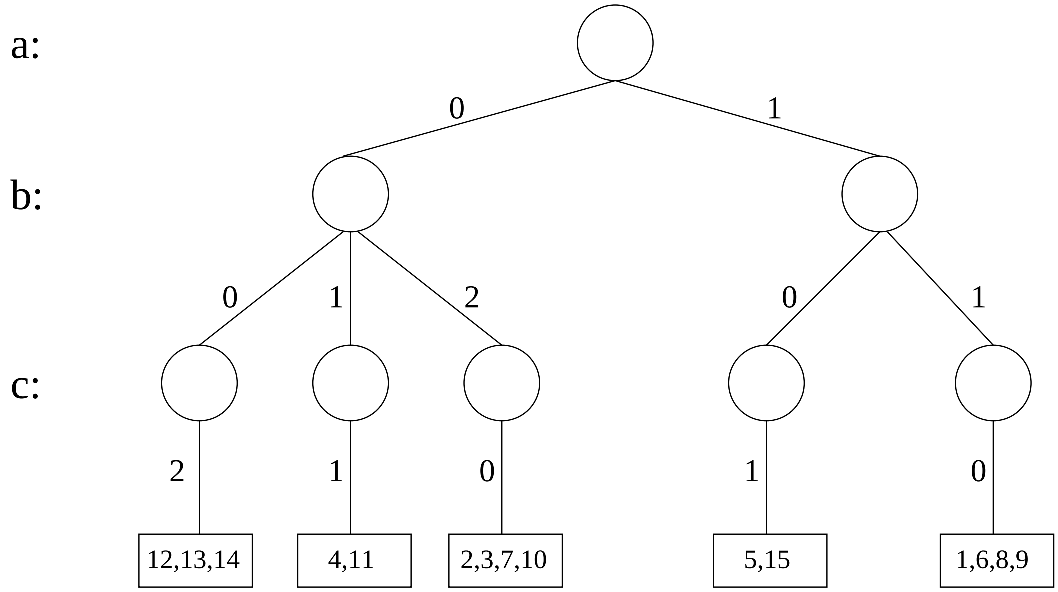

Figure 3.3: Abelian tree for $m = 2$ over input text "abbbcabbabbcccca" and the alphabet order $a < b < c$. The character on the left of the figure show the associated character of each level.

to answer a query for finding the matches of a given abelian pattern.

Although, the number of abelian patterns of length m over an alphabet Σ (of size σ) is $\binom{\sigma+m-1}{m}$ (Section 1.2.2), the number of length-m distinct abelian patterns in an input text T of length n is $O(n)$. In some situations, it is possible that $\binom{\sigma+m-1}{m} < n$, however, we assume that in general n is much smaller than $\binom{\sigma+m-1}{m}$, and in the rest of the chapter, we consider $O(n)$ a better upper bound for the number of distinct abelian patterns in the input text.

Abelian Tree Construction Using Directly Accessible Arrays at Internal Nodes

The outgoing edges of a node in an abelian tree are labeled with natural numbers between 0 and m, so we can use a directly accessible array of $m+1$ elements to store these edges. Figure 3.4 illustrates an internal node that uses directly accessible array to store its outgoing edges.

Observation 5. *The time to create and initialize an internal node, that uses a directly accessible array for storing its outgoing edges, is $O(m)$.*

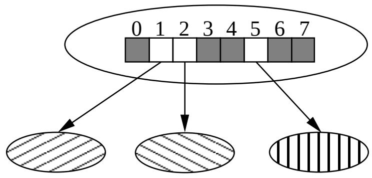

Figure 3.4: An internal node in the abelian tree for the the abelian patterns of length 7. The node uses a directly accessible array for storing its outgoing edges. A shaded element of the array indicates that no outgoing edge corresponds to this element.

Lemma 4. *The time complexity of inserting a leaf node in the abelian tree is $O(m\sigma)$.*

Proof. Let X be a leaf node corresponding to an m-length abelian pattern P_X. Then the insertion of X in the abelian tree requires insertion of $O(\sigma)$ new internal nodes on the path from the root node to X. This is because of the fact that, the tree has $\sigma + 1$ levels (Observation 4), and in the worst case, the node in the tree that corresponds to the longest prefix of P_X is the root node. As the time to insert an internal node in the tree is $O(m)$ (Observation 5), therefore, the aggregate time to insert X in the tree is $O(m\sigma)$.

□

Lemma 5. *The time complexity of the construction of an abelian tree, for an input text T, that uses directly accessible arrays for storing the outgoing edges at its internal nodes is $O(mn\sigma)$.*

Proof. As there are $O(n)$ distinct abelian patterns of length m in the input text T (where $n := |T|$), so the number of the leaf nodes in the abelian tree is also $O(n)$. The time complexity of inserting a leaf node in the tree is $O(m\sigma)$ (Lemma 4), so the total time for the construction of the abelian tree for T is $O(mn\sigma)$.

□

Lemma 6. *The space requirement of an abelian tree that uses directly accessible array for storing the outgoing edges at its internal nodes is $O(mn\sigma)$.*

Proof. The tree has $\sigma + 1$ levels (Observation 4)) and $O(n)$ leaf nodes, so we have $O(n\sigma)$ internal nodes. An internal node requires $O(m)$ space when we use a directly accessible array to store its outgoing edges, so the aggregate space requirement for the internal nodes of the tree is $O(mn\sigma)$.

The leaf nodes of the tree contain the pointers to the locations, of the m-length abelian patterns, in T. As there are $O(n)$ distinct abelian patterns of length m, the total number of the pointers stored at the leaf nodes is $O(n)$. Consequently, the aggregate space requirement for all the leaf nodes is $O(n)$. Hence the abelian tree requires $O(mn\sigma)$ space.

□

Lemma 7. *Given an abelian tree $\mathcal{T}$, built on the input text T, that uses the directly accessible arrays to store the outgoing edges at its internal nodes, the time complexity to find a leaf node corresponding to a given pattern P is $O(\sigma)$.*

Proof. Let $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$ is the given abelian pattern. Then, the sequence of the edges in the path, from the root node to the leaf node corresponding to P, in $\mathcal{T}$ is

$$m_{c_1} \to m_{c_2} \to \cdots \to m_{c_\sigma}$$

As we use directly accessible arrays for storing the outgoing edges at the internal nodes, each of the edges in this path is accessible in $O(1)$ time, so the time complexity to reach the leaf node corresponding to P is $O(\sigma)$.

□

Thus, we can assert in time $O(\sigma)$ whether or not P occurs in T, and we can output the positions where P begins in T in time proportional to the frequency of P in T.

Abelian Tree Construction Using the Sorted Linked Lists at Internal Nodes

A sorted linked list can also be used to store the outgoing edges at an internal node. Figure 3.5 illustrates the internal node shown in Figure 3.4 with the difference that now a sorted linked list is used, instead of a directly accessible array, to store the outgoing edges of the node.

Let $\mathcal{T}$ be an abelian tree and let P_X be an abelian pattern of length m, and we want to find/insert the leaf node X corresponding to P_X in $\mathcal{T}$. Let e be an edge in the path from the root node to the leaf node X and let $\hat{e}$ be the edge preceding e in the path from the root node to X (we assume that $\sigma \geq 2$), then:

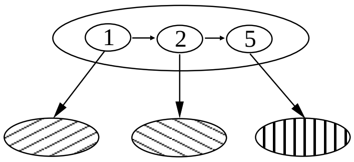

Figure 3.5: An internal node in the abelian tree that uses a sorted linked list for storing its outgoing edges.

> **Definition 9.** *e is an* existing edge *if e already exists in $\mathcal{T}$ as an outgoing edge of an internal node N that lies in the path from the root node to X.*
>
> **Definition 10.** *e is a* partially-existing edge *if e did not exist in $\mathcal{T}$, however, $\hat{e}$ is an existing edge of $\mathcal{T}$.*
>
> **Definition 11.** *e is a* non-existing edge *if neither e nor $\hat{e}$ are existing edges.*

Observation 6. *If e is a partially-existing edge, then an internal node N, that has e as an outgoing edge in the path from the root node to X, already exists in the tree.*

Observation 7. *There can be at most 1 partially existing edge in the path from the root node to the leaf node X.*

Figure 3.6 illustrates all three types of edges defined above. We insert a new leaf node corresponding to the abelian pattern $2a + b + c$ in the abelian tree of Figure 3.2 and mark the edges, on the path from the root node to this newly inserted leaf node, as existing, partially-existing and non-existing.

Lemma 8. *The time complexity of inserting a leaf node in the abelian tree is $O(m + \sigma)$.*

Proof. Recall that there are σ edges on the path from the root node to to a leaf node X which corresponds to an m-length abelian pattern P_X (Observation 4).

Let e be an edge in the path from the root node to X and let m_e be the label of e, then there are three possibilities:

Case I- e is an *existing edge*: Let N be the node hosting e. We search e in the linked list of the outgoing edges at N, and follow the path to the leaf node X.

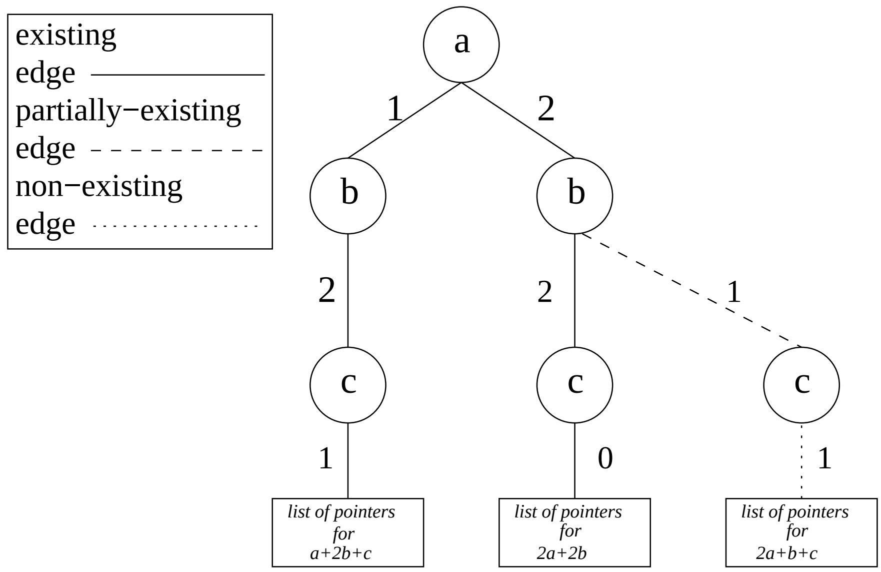

Figure 3.6: The classification of the edges in the path of the newly inserted leaf node corresponding to the abelian pattern $2a + b + c$ in the abelian tree of Figure 3.2.

The time complexity to find e in the linked list of N is $O(m_e)$, as in the worst case, the linked list of N contains all the edges whose label is less than m_e.

Case II- e **is a *partially-existing edge*:** This means an internal node N, that would have e as an outgoing edge in the path from the root node to X, already exists in the tree (Observation 6). So we insert e in the linked list of the outgoing edges of N and create a node, with empty linked list, following e.

To insert e in the linked list at N, we first locate the position of e in the list, which costs $O(m_e)$ time, and then in $O(1)$ time we insert e in the list. The node following e has an empty linked list, so it is created in $O(1)$ time.

Thus the time complexity of the insertion of e and the node following e in the tree is $O(m_e)$.

Case III- e **is a *non-existing edge*:** This implies a node N, with empty linked list, has already been created while inserting the edge preceding e, so we insert e in the empty linked list of the outgoing edges of N and create a node, with empty linked list, following e.

We insert e in the liked list of N in $O(1)$ time and as the node following e has an empty linked list, so it is also created in $O(1)$ time.

Thus the time complexity of the insertion of e and the node following e in the tree is $O(1)$.

The sum total of the labels of the edges on the path from the root node to X is m and this path has σ edges; so the time complexity to insert/find X in the abelian tree is $O(m + \sigma)$.

□

Lemma 9. *The time complexity to construct an abelian tree that uses linked lists to store the outgoing edges at its internal nodes is* $O(n(m + \sigma))$.

Proof. As the number of abelian patterns of length m in the input text T is $O(n)$, so the number of the leaf nodes in the abelian tree is also $O(n)$. Moreover, the time complexity to insert a leaf node in the tree is $O(m + \sigma)$ (Lemma 8), therefore, the time complexity of the abelian tree construction is $O(n(m + \sigma))$.

□

Lemma 10. *The space requirement of an abelian tree, that uses linked lists at its internal nodes for storing the outgoing edges, is* $O(n\sigma)$.

Proof. The tree has two types of nodes: internal nodes and leaf nodes. At internal node, we store the edges of the tree, and at leaf nodes we store the pointers to the locations of the abelian patterns in T.

As the tree has $O(n)$ leaf nodes, therefore, there are $O(n)$ distinct paths from the root node to the leaf nodes. Each path in the tree has σ edges, so the number of edges in the tree is $O(n\sigma)$. As it requires $O(1)$ space to store an edge in a linked list, therefore, the space requirement of the internal nodes of the tree is $O(n\sigma)$.

The leaf nodes contain lists of pointers to the locations of the abelian patterns in T. As the number of abelian patterns is $O(n)$, so is the number of the pointers to their locations in T. A pointer requires $O(1)$ space, so the space requirement of the leaf nodes is $O(n)$.

Hence, the overall space requirement of the abelian tree is $O(n\sigma)$.

□

The time complexity to find a leaf node corresponding to a given abelian pattern P is $O(m+\sigma)$ (Lemma 8). Thus, we assert in time $O(m+\sigma)$ whether or not P occurs in T, and output the positions where P begins in T in time proportional to the frequency of P.

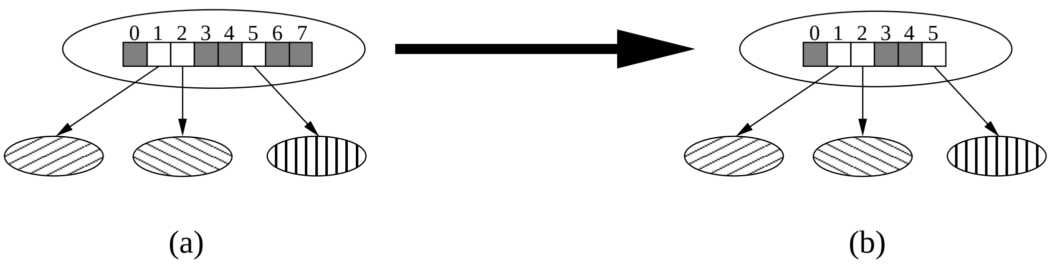

Figure 3.7: **(a)** An internal node having a directly accessible array to store the outgoing edges. **(b)** The same node, with the directly accessible array truncated from right. Now there are no shaded elements at the right end of the array.

3.3.3 An Efficient and Compact Abelian Tree

If we use directly accessible arrays to store the outgoing edges at the internal nodes of an abelian tree, then the query processing time is very efficient ($O(\sigma)$), but the space requirement of the tree is $O(mn\sigma)$.

In contrast, if we use linked lists to store the outgoing edges at the internal nodes of an abelian tree, then the storage requirement is improved ($O(n\sigma)$), however, the query processing time now becomes $O(m + \sigma)$.

The efficiency in the query processing time comes from the feature of the direct accessibility of the edges in case of a tree built using the directly accessible arrays. The storage efficiency, in case of a tree built using the linked lists, come from the fact that no space, other than the required, is allocated for storing an edge.

In the following, we present an abelian tree whose space requirement is between $O(mn\sigma)$ and $\Omega(n\sigma)$, but the query processing time is $O(\sigma)$.

The key idea is that we delete several unused pointers of the directly accessible arrays at an internal node, without affecting the direct accessibility of the outgoing edges of that node. Let e_{max} be the largest label among the labels of all the outgoing edges of an internal node N. Then we truncate the directly accessible array at N from right, such that now the size of the array is $e_{max}+1$ instead of $m+1$. Figure 3.7 shows the internal node of Figure 3.4 after truncating the directly accessible array of the node from the right.

Note that the outgoing edges at N still remain directly accessible as no edge label at N is greater than e_{max}. Hence we can assert in time $O(\sigma)$ whether or not a given abelian pattern P occurs in T, and we can output the starting position of P in T in time linear to the frequency of P in T.

Lemma 11. *The space requirement of the compact abelian tree is* $O(n(m +$

σ)).

Proof. For the sake of analysis, we assume that no path is shared by two or more leaf nodes in the abelian tree that uses directly accessible arrays for storing the outgoing edges at its internal nodes. If this is not the case, then we replicate the shared paths to achieve this property. A path $Path_X$ from the root node to a leaf node X has σ internal nodes. For an internal node N lying on $Path_X$, let e_N be the label of the outgoing edge of N; then we need to keep the size of the array at N no more than $e_N + 1$. The sum of the labels of all the edges lying on $Path_X$ is m, and the number of edges lying on $Path_X$ is σ, so the total storage requirement of all the internal nodes lying on $Path_X$ is $O(m + \sigma)$.

As there are $O(n)$ leaf nodes, so we have $O(n)$ paths and the storage requirement of the internal nodes is $O(n(m + \sigma))$.

The leaf nodes contain lists of pointers to the locations of the abelian patterns in T. As the number of abelian patterns is $O(n)$, so is the number of the pointers to their locations in T. A pointer requires $O(1)$ space, so the space requirement of the leaf nodes is $O(n)$.

Hence the space requirement of the compact abelian tree is $O(n(m + \sigma))$.

□

Efficient Construction of the Compact Tree

Although, by truncating the unnecessary parts of the directly accessible arrays, we can construct a compact abelian tree from an abelian tree that uses directly accessible arrays at internal nodes for storing the outgoing edges; the cost of such construction is high, as the time complexity of constructing the initial abelian tree is $O(nm\sigma)$ (Lemma 5).

We can construct the same compact abelian tree from an abelian tree that uses linked lists for storing the outgoing edges at its internal nodes. If N is an internal node of the abelian tree and e_{max} is the largest label among the labels of all the outgoing edges of N; then we create an array of size $e_{max} + 1$ at N and copy the edges in the linked list at N to the newly created array at appropriate locations. Figure 3.8 illustrates this phenomenon.

By replacing the linked lists with arrays at all the internal nodes of the tree, we construct the compact but efficient abelian tree from an abelian tree that had linked lists at its internal nodes.

Lemma 12. *The time complexity to efficiently construct a compact and efficient abelian tree is* $O(n(m + \sigma))$.

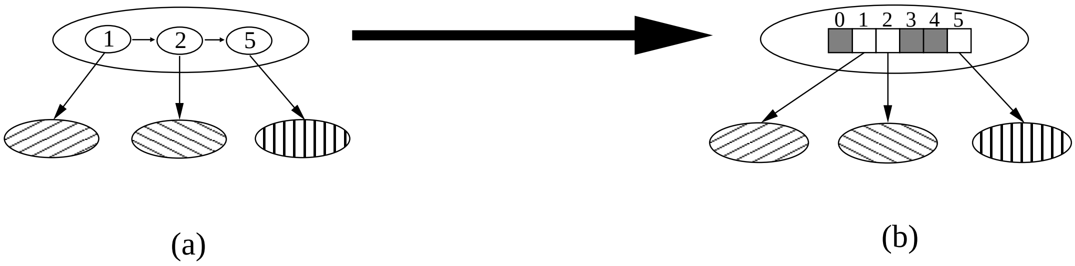

Figure 3.8: **(a)** An internal node having a linked list to store the outgoing edges. **(b)** The same node, with the linked list replaced by a directly accessible array for storing the outgoing edges. The shaded elements of the array contain no edge; and this unused storage is the cost we pay to get the efficiency of the direct accessibility.

Proof. The time complexity of the construction of an abelian tree that uses linked lists at internal nodes for storing the outgoing edges, is $O(n(m+\sigma))$ (Lemma 9). We assume that the information about the largest label among the labels of all the outgoing edges of an internal node is also stored in the node and therefore, this information is available in $O(1)$ time. As the sum total of the sizes of all the arrays at the internal nodes of a compact and efficient tree is $O(n(m+\sigma))$ (Lemma 11), so the time for creating these arrays and writing once into them is $O(n(m+\sigma))$.

Hence the time complexity to construct a compact and efficient abelian tree is $O(n(m+\sigma))$.

□

3.4 Abelian Tree without Zero-Edges

In case of abelian tree based indexing, the query processing time depends on the length of the path from the root node to a leaf node, which is $O(\sigma)$ for the abelian trees presented in the previous section. That is why the term σ appears in the asymptotic time complexity of the query processing time of all these trees (e.g. the query processing time of an abelian tree with linked lists at the internal nodes is $O(m+\sigma)$ and the query processing time of a compact and efficient abelian tree is $O(\sigma)$).

In this section, we present an alternative construction of the abelian trees, to make the query processing time independent of σ. We begin with several observations.

Observation 8. *If e is a zero-edge (an edge with zero as edge label) on the* *path from the root node to a leaf node X and N is the node hosting e (i.e. e is an outgoing edge of N), then the abelian pattern corresponding to X does not contain the character associated with N.*

Observation 9. *If Σ_P is the set of the characters that appear in an abelian pattern P, and $\sigma_P := |\Sigma_P|$; then there are $\sigma - \sigma_P$ zero-edges in the path from the root node to the leaf node corresponding to the abelian pattern P.*

The basic reason, for the presence of the zero-edges in the abelian trees presented so far, was to preserve the order of the characters corresponding to different levels of the tree (recall that in the abelian tree of Figure 3.3, the nodes at the same level of the tree correspond to the same character). Moreover, the zero-edges also prevent non-determinism in finding the path from the root node to a leaf node. Figure 3.9 illustrates two different abelian trees for the same set of abelian patterns; one abelian tree has zero-edges, the other is without zero-edges. Note that the nodes at the same level correspond to different characters in the second tree. Moreover, there are more than one outgoing edges with the same edge label at the root node of the tree without zero-edges, which causes the non-determinism regarding which edge to follow at the root node to reach a particular leaf node.

Although, the example illustrated in Figure 3.9 shows the significance of the zero-edges in an abelian tree, the impact of zero-edges on the query processing time becomes severe if σ is large.

In the following, we present a deterministic abelian tree without zero-edges.

3.4.1 Structure of an Abelian Tree without Zero-Edges

The abelian tree we present here has alternating levels. The nodes at the odd levels of the tree correspond to the characters of the abelian patterns and the nodes at the even levels of the tree correspond to the multiplicities of the characters of the abelian patterns. The root node is at level 1 of the tree.

We call the nodes at the odd levels, the *character nodes*, and the nodes at the even levels, the *multiplicity nodes*. Similarly, the outgoing edges of a character node are called *character edges* and the outgoing edges of a multiplicity node are called *multiplicity edges*.

Observation 10. *If x is the edge label of a character edge then $x \in \Sigma$. Similarly, if y is the edge label of a multiplicity edge then $y \in \{1, 2, 3, \ldots, m\}$.*

Figure 3.10 illustrates an abelian tree with alternating levels for the abelian patterns of length 2 for the input text "abbbcabbabbcccca"; with

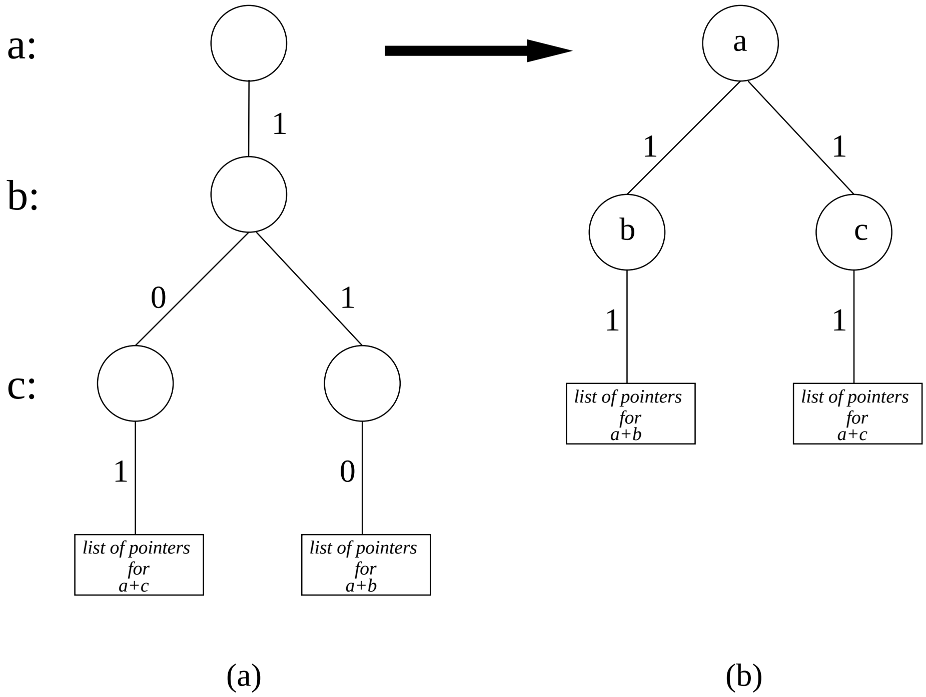

Figure 3.9: **(a)** An abelian tree for the abelian patterns $a+b$ and $a+c$. The nodes at the same level correspond to the same character and the path from the root node to a leaf node is deterministic. **(b)** The same tree without zero-edges. Now the nodes at the same level of the tree correspond to different characters, and the node corresponding to character a has two outgoing edges with same edge label 1.

the ordering of the characters in Σ being $a < b < c$. The same tree with zero-edges has been illustrated in Figure 3.3. Note that the abelian tree with alternating levels is a deterministic tree.

Observation 11. *If $P = \sum_{j=1}^{\sigma_P - k} m_{c_{i_j}} c_{i_j}$ (where $k \geq 1$) is the abelian pattern corresponding to an internal node N of the abelian tree with alternating levels, then the labels of the outgoing edges of N are greater than $c_{i_{\sigma_P - k}}$.*

Lemma 13. *The length of the path from the root node to the leaf node corresponding to an abelian pattern P, in an abelian tree with alternating levels, is $O(\sigma_P)$; where σ_P is the number of distinct characters in P.*

Proof. As there are no zero-edges in an abelian tree with alternating levels, each character edge in the path from the root node to the leaf node corresponding to P corresponds to a character of P and each multiplicity edge in

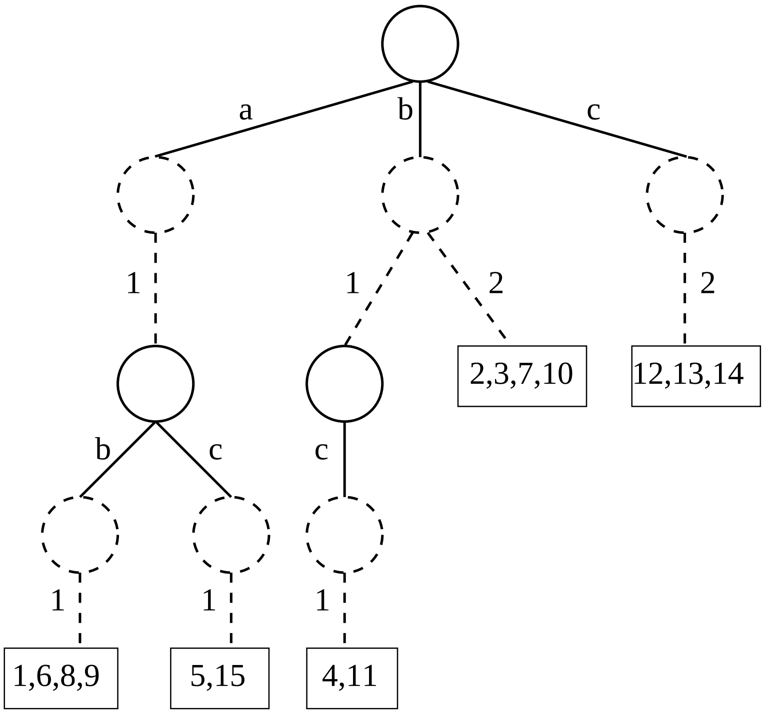

Figure 3.10: Abelian tree for $m = 2$ over input text "abbbcabbabbcccca" and the alphabet order $a < b < c$. The nodes and the edges shown in the solid lines correspond to the characters and the nodes and the edges shown in the dashed lines correspond to the frequencies of the characters.

this path corresponds to the multiplicity of a character of P. As the abelian pattern P has σ_P distinct characters, the number of edges on the path from the root node to the leaf node corresponding to P is $2\sigma_P$.

$\square$

Corollary 2. *The height of an abelian tree with alternating levels is $O(m)$.*

3.4.2 An Abelian Tree with Alternating Levels Having Linked Lists at Internal Nodes

The abelian tree uses sorted linked lists to store the outgoing edges at its internal nodes.

Lemma 14. *The time complexity to find the leaf node corresponding to an abelian pattern P in the tree is $O(\sigma + m)$.*

Proof. Let $P = \sum_{j=1}^{\sigma_P} m_{c_{i_j}} c_{i_j}$ be the given abelian pattern, then the sequence of edges from the root node to the leaf node corresponding to P, in the tree is

$$c_{i_1} \rightarrow m_{c_{i_1}} \rightarrow c_{i_2} \rightarrow m_{c_{i_2}} \rightarrow \cdots \rightarrow c_{i_{\sigma_P}} \rightarrow m_{c_{i_{\sigma_P}}}$$

The time complexity to find an edge e that has edge label $m_{c_{i_k}}$, at an internal node N that hosts e, is $O(m_{c_{i_k}})$; as in the worst case, all the edges having labels less than $m_{c_{i_k}}$ are also present as the outgoing edges of N. As $\sum_{j=1}^{\sigma_P} m_{c_{i_j}} = m$, the total time complexity of finding all the multiplicity edges in the path from the root node to the leaf node corresponding to P is m.

The time complexity to find the edge with edge label c_{i_1} at the root node is i_1; as in the worst case, all the edges having labels less than c_{i_1} are also present as the outgoing edges of the root node.
The time complexity to find an edge e that has edge label c_{i_k}, at an internal node N that hosts e, is $i_k - i_{k-1}$. This is because of the fact that the edge labels of the outgoing edges of N are greater than $c_{i_{k-1}}$ (Observation 11) and in the worst case all the edges that have label less than c_{i_k} are present as the outgoing edges of N.
So the total time complexity of finding all the character edges in the path from the root node to the leaf node corresponding to P is i_{σ_P}. As in the worst case, $c_{i_{\sigma_P}} = c_\sigma$, so the aggregate time complexity of finding all the character edges at there hosting nodes is $O(\sigma)$.

Hence the time complexity to find the leaf node, corresponding to the abelian pattern $P = \sum_{j=1}^{\sigma_P} m_{c_{i_j}} c_{i_j}$, in the tree is $O(\sigma + m)$.

□

Lemma 15. *The time to insert a leaf node corresponding to an abelian pattern $P = \sum_{j=1}^{\sigma_P} m_{c_{i_j}} c_{i_j}$, in an abelian tree with alternating levels that uses linked lists to store outgoing edges at its internal nodes, is $O(\sigma + m)$.*

Proof. To insert a leaf node corresponding to the abelian pattern $P = \sum_{j=1}^{\sigma_P} m_{c_{i_j}} c_{i_j}$, we first find the node corresponding to the longest abelian prefix of P. This we do in time $O(m + \sigma)$ (Lemma 14).

Let the longest prefix of P that has a corresponding node in the abelian tree be $P' = \sum_{j=1}^{k} m_{c_{i_j}} c_{i_j}$, where $k \leq \sigma_P$, and let N be the node corresponding to P' in the abelian tree.

If $k = \sigma_P$, then we simply append the pointer to the starting position of P in the input text T at the end of the list of pointers maintained at node N.

As the time complexity to append a pointer at the end of the list of pointers maintained at a leaf node is $O(1)$, the time complexity to insert P in the abelian tree is $O(m + \sigma)$.

If $0 \leq k < \sigma_P$, then:
If the edge with the label $c_{i_{k+1}}$ exists as an outgoing edge of N (if $k = 0$, then N is the root node), then we follow this edge. Let this edge leads to the node N' in the abelian tree. Then we insert new edges in the tree in the following manner, with $m_{c_{i_{k+1}}}$ being inserted in the linked list of N'.

$$m_{c_{i_{k+1}}} \longrightarrow c_{i_{k+2}} \longrightarrow m_{c_{i_{k+2}}} \longrightarrow \cdots \longrightarrow c_{i_{\sigma_P}} \longrightarrow m_{c_{i_{\sigma_P}}}$$

After inserting an edge in the tree, we create a new node with empty linked list at the end of this newly inserted edge; and the next edge is inserted in the empty linked list of this newly created node. We continue inserting the edges in the tree until we have inserted the edge with the label $m_{c_{i_{\sigma_P}}}$; after which we create the leaf node corresponding to P at the end of this edge, and add a pointer to the starting position of P in T to this newly inserted leaf node of the tree.
If the edge with the label $c_{i_{k+1}}$ does not exist as an outgoing edge of N, then too we we follow the procedure mentioned above, with the difference that now we also insert the edge with the label $c_{i_{k+1}}$ in the tree; and this edge is inserted in the linked list of N. So now, the edges are inserted in the tree in the following order:

$$c_{i_{k+1}} \longrightarrow m_{c_{i_{k+1}}} \longrightarrow c_{i_{k+2}} \longrightarrow m_{c_{i_{k+2}}} \longrightarrow \cdots \longrightarrow c_{i_{\sigma_P}} \longrightarrow m_{c_{i_{\sigma_P}}}$$

The time complexity to find the edge with the label $c_{i_{k+1}}$ in the linked list of node N is $O(i_{k+1} - i_k)$ which is in $O(\sigma)$. Moreover, if the edge with the label $c_{i_{k+1}}$ exists as an outgoing edge of N, then the time complexity to find the edge with the label $m_{c_{i_{k+1}}}$ in the linked list of node N' is $O(m_{c_{i_{k+1}}})$ which is in $O(m)$. The time complexity of creating a new node with empty linked list is $O(1)$ and the time complexity to insert an edge in the already empty linked list of a node is also $O(1)$. So the time complexity to create the new nodes and inserting the new edges in the abelian tree is $O(\sigma_P - k) = O(\sigma_P)$.

Hence in time $O(m + \sigma)$, we insert the leaf node corresponding to P in the abelian tree.

□

Lemma 16. *The time complexity of the construction of an abelian tree with alternating levels, that uses sorted linked lists to store the outgoing edges at its internal nodes, is* $O(n(m + \sigma))$

Proof. There are $O(n)$ abelian patterns of length m in the input text T, so we insert $O(n)$ leaf nodes in the abelian tree. The time complexity to insert a leaf node in the tree is $O(m + \sigma)$ (Lemma 15), hence, the time complexity to construct the tree is $O(n(m + \sigma))$.

□

Lemma 17. *The space complexity of an abelian tree with alternating levels that uses sorted linked lists to store the outgoing edges at its internal nodes is $O(mn)$.*

Proof. There are $O(n)$ distinct paths, from the root node to the leaf nodes, in the tree. The length of the path form the root node to a leaf node, in an abelian tree with alternating levels, is $O(m)$ (Corollary 2). So we have $O(mn)$ edges in the tree. As we use linked lists at the internal nodes to store the outgoing edges, an edge is stored in $O(1)$ space. Hence the space requirement of all the internal nodes is $O(mn)$. The leaf nodes of the tree contain pointers to the locations of the abelian patterns, and the number of these pointers is $O(n)$. A pointer requires $O(1)$ space, so the space requirement of all the leaf nodes is $O(n)$.

Hence, the space complexity of an abelian tree with alternating levels, that uses sorted linked lists to store the outgoing edges at its internal nodes, is $O(mn)$.

□

3.4.3 Replacing the Linked Lists with the Arrays

Once we have constructed an abelian tree with alternating levels that uses sorted linked lists to store the outgoing edges at its internal nodes, we optimize the structure of the internal nodes of the tree to make the process of query processing efficient.

Replacing the Linked Lists at Multiplicity Nodes with Directly Accessible Arrays

The labels of the edges at a multiplicity node of the abelian tree range between 1 and m. In the linked list structure, the time complexity to find an edge with label e at a multiplicity node M is $O(e)$, as in the worst case, all the edges with labels less than e are present as outgoing edges of M. Let e_{max} be the highest edge label among the outgoing edges of a multiplicity node M; then we convert the linked list of the outgoing edges of M into a

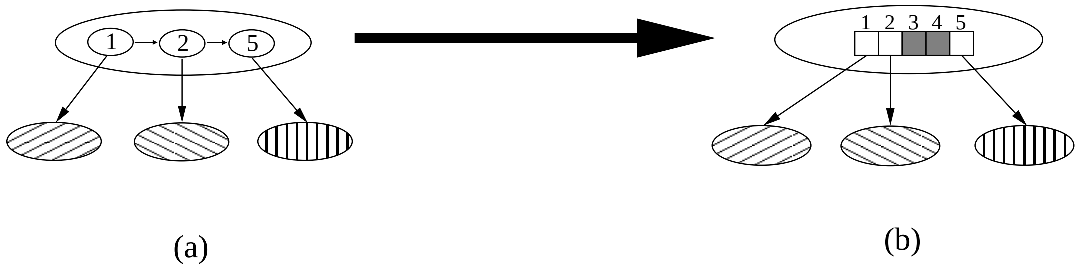

Figure 3.11: **(a)** A multiplicity node having a linked list to store the outgoing edges. **(b)** The same node, with the linked list replaced by a directly accessible array for storing the outgoing edges. Note that the index of the array begins at 1 instead of 0.

directly accessible array of e_{max} elements, by using the method described in Section 3.3.3. Figure 3.11 illustrates the conversion of the linked list into a directly accessible array at a multiplicity node. Note that unlike Figure 3.8, the array starts from index 1 in Figure 3.11, as the tree does not contain any zero-edges. Now the time complexity to find an edge with label e at a multiplicity node M becomes $O(1)$.

Lemma 18. *The space requirement of all the multiplicity nodes that use directly accessible arrays for storing their outgoing edges is $O(mn)$.*

Proof. For the sake of analysis, we assume that no path is shared by two or more leaf nodes in the abelian tree that uses the linked lists for storing the outgoing edges at its internal nodes. If this is not the case, then we replicate the shared paths to achieve this property. A path $Path_X$ from the root node to a leaf node X has σ_P multiplicity nodes. For a multiplicity node M lying on $Path_X$, let e_M be the label of the outgoing edge of N; then we need to keep the size of the array at M no more than e_M. The sum of the labels of all the edges lying on $Path_X$ is m, so the total storage requirement of all the multiplicity nodes lying on $Path_X$ is $O(m)$.

As there are $O(n)$ leaf nodes, so we have $O(n)$ paths and the storage requirement of the multiplicity nodes is $O(mn)$.

□

Corollary 3. *After replacing the linked lists with the directly accessible arrays at the multiplicity nodes, the space complexity of the abelian tree with alternating levels remains $O(mn)$.*

Lemma 19. *The time complexity to covert the linked lists at the multiplicity nodes into directly accessible arrays is $O(mn)$.*

Proof. We assume that the information about the largest label among the labels of all the outgoing edges of a multiplicity node is also stored in the node and therefore, this information is available in $O(1)$ time. As the sum total of the sizes of all the arrays at the multiplicity nodes of the tree is $O(mn)$ (Lemma 18), so the time for creating these arrays and writing once into them is $O(mn)$.

□

Lemma 20. *The time complexity to find the leaf node corresponding to an abelian pattern P in the tree, after replacing the linked lists with the directly accessible arrays at the multiplicity nodes, is $O(\sigma)$.*

Proof. Let $P = \sum_{j=1}^{\sigma_P} m_{c_{i_j}} c_{i_j}$ be the given abelian pattern, then the sequence of the edges from the root node to the leaf node corresponding to P, in the tree is

$$c_{i_1} \rightarrow m_{c_{i_1}} \rightarrow c_{i_2} \rightarrow m_{c_{i_2}} \rightarrow \cdots \rightarrow c_{i_{\sigma_P}} \rightarrow m_{c_{i_{\sigma_P}}}$$

The time complexity to find an edge e that has edge label $m_{c_{i_k}}$, at an internal node N that hosts e, is $O(1)$ now; as the edge labels are stored in directly accessible arrays at the multiplicity nodes.
So the total time complexity of finding all the multiplicity edges in the path from the root node to the leaf node corresponding to P is σ_P.

The time complexity to find the edge with edge label c_{i_1} at the root node is i_1; as in the worst case, all the edges having labels less than c_{i_1} are also present as the outgoing edges of the root node.
The time complexity to find an edge e that has edge label c_{i_k}, at an internal node N that hosts e, is $i_k - i_{k-1}$. This is because of the fact that the edge labels of the outgoing edges of N are greater than $c_{i_{k-1}}$ (Observation 11) and in the worst case all the edges that have label less than c_{i_k} are present as the outgoing edges of N.
So the total time complexity of finding all the character edges in the path from the root node to the leaf node corresponding to P is i_{σ_P}. As in the worst case, $c_{i_{\sigma_P}} = c_\sigma$, so the aggregate time complexity of finding all the character edges at there hosting nodes is $O(\sigma)$.

Hence the time complexity to find the leaf node, corresponding to the abelian pattern $P = \sum_{j=1}^{\sigma_P} m_{c_{i_j}} c_{i_j}$, in the tree is $O(\sigma)$.

□

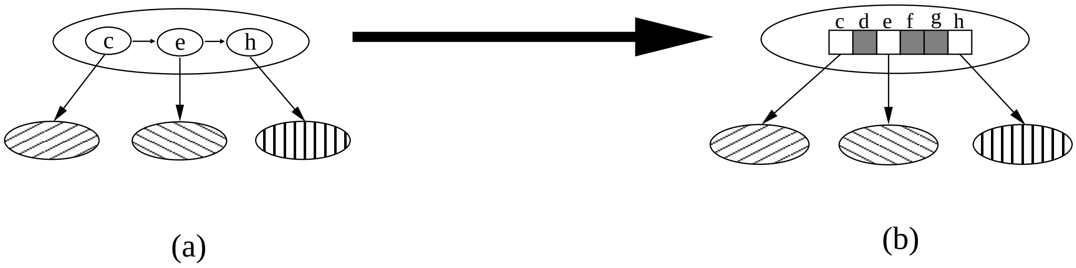

Figure 3.12: **(a)** A character node having a linked list to store the outgoing edges; with the edge labels being the English alphabets with the ordering $a < b < c < \cdots < z$. **(b)** The same node, with the linked list replaced by a directly accessible array for storing the outgoing edges. The shaded elements of the array contain no edge; and this unused storage is the cost we pay to get the efficiency of the direct accessibility.

Replacing the Linked Lists at Character Nodes with Directly Accessible Arrays

We assume that there exists a minimal perfect hash function ρ for the characters in Σ (i.e. ρ hashes Σ to consecutive numbers $\{0, \ldots, \sigma - 1\}$); moreover, $\rho(c_i) < \rho(c_j) \Leftrightarrow c_i < c_j$. Note that for the alphabets of English language, ρ is quite simple; it just subtracts a constant from the ASCII values of the characters. Now we can use the directly accessible arrays to store the outgoing edges at the character nodes as well. Let e_{min} and e_{max} be the lowest and the highest edge labels respectively, among the outgoing edges of a character node C. Then we convert the linked list of the outgoing edges of the node C into a directly accessible array of $\rho(e_{max}) - \rho(e_{min}) + 1$ elements using the method described in Section 3.3.3. Figure 3.12 illustrates the conversion of the linked list into a directly accessible array at a character node.

Now the edge with label c_i, at a character node C, is found at the $\rho(c_i) - \rho(c_{min}) + 1th$ location of the directly accessible array of the outgoing edges of C; where c_{min} is the label of the edge at the first location of the array. Thus, we find an edge at a character node in $O(1)$ time.

Lemma 21. *The space complexity of the tree, after replacing the linked lists with the directly accessible arrays at both, the multiplicity nodes as well as the character nodes, is $O(n(m + \sigma))$.*

Proof. For the sake of analysis, we assume that no path is shared by two or more leaf nodes in the abelian tree that uses the directly accessible arrays for storing the outgoing edges at its character nodes. If this is not the case, we replicate the shared paths to achieve this property. A path $Path_X$ from the root node to a leaf node X corresponding to an abelian pattern $P = \sum_{j=1}^{\sigma_P} m_{c_{i_j}} c_{i_j}$ has σ_P character nodes and σ_P multiplicity nodes. The sequence of the edges in $Path_X$ is as follows:

$$c_{i_1} \rightarrow m_{c_{i_1}} \rightarrow c_{i_2} \rightarrow m_{c_{i_2}} \rightarrow \cdots \rightarrow c_{i_{\sigma_P}} \rightarrow m_{c_{i_{\sigma_P}}}$$

Let $M_{c_{i_j}}$ denotes the node hosting the edge $m_{c_{i_j}}$. Then the array at $M_{c_{i_j}}$ comprises of $m_{c_{i_j}}$ elements (ranging from 1 to $m_{c_{i_j}}$). As $\sum_{j=1}^{\sigma_P} m_{c_{i_j}} = m$, the total space requirement of all the multiplicity nodes in $Path_X$ is m.

Let $C_{c_{i_j}}$ denotes the node hosting the edge c_{i_j}. The array at $C_{c_{i_1}}$ comprise of only one element, which is the edge with label c_{i_1}.
The array at $C_{c_{i_j}}$ comprises of $i_j - i_{j-1}$ elements, as the first element of this array is greater than $c_{i_{j-1}}$ (Observation 11) and the last element of this array is c_{i_j} (because the edge with label c_{i_j} is the only outgoing edge of $C_{c_{i_j}}$).
So the total space requirement of all the character nodes on $Path_X$ is $i_{\sigma_P} - i_1$. As in the worst case, $c_{i_1} = c_1$ and $c_{i_{\sigma_P}} = c_\sigma$, so the space requirement of all the character nodes on $Path_X$ is $O(\sigma)$.
Hence, the space requirement of all the internal nodes (both character nodes as well as multiplicity nodes) lying on $Path_X$ is $O(m + \sigma)$.
As there are $O(n)$ leaf nodes, so we have $O(n)$ paths from the root node to the leaf nodes; thus the storage requirement of all the internal nodes of the abelian tree is $O(n(m + \sigma))$.

The leaf nodes contain lists of pointers to the locations of the abelian patterns in T. As the number of abelian patterns is $O(n)$, so is the number of the pointers to their locations in T. A pointer requires $O(1)$ space, so the space requirement of the leaf nodes is $O(n)$.

Hence the space requirement of an abelian tree with alternating levels, that uses directly accessible arrays at its internal nodes to store the outgoing edges, is $O(n(m + \sigma))$.

□

Lemma 22. *The time complexity to construct an abelian tree with alternating levels, that uses directly accessible arrays at it internal nodes for storing the outgoing edges, is* $O(n(m + \sigma))$.

Proof. The time complexity to construct an abelian tree with alternating levels, that uses linked lists to store the outgoing edges at it internal nodes, is $O(n(m + \sigma))$ (Lemma 16). We assume that the information about the largest label among the labels of all the outgoing edges of a multiplicity node is also stored in the node and therefore, this information is available in $O(1)$ time. We also assume that the information about the smallest and the largest labels among the labels of all the outgoing edges of a character node is also stored in the node and therefore, this information is also available in $O(1)$ time. As the sum total of the sizes of all the arrays at the internal nodes of the tree is $O(n(m+\sigma))$ (Lemma 21), so the time for creating these arrays and writing once into them is $O(n(m+\sigma))$.

□

Lemma 23. *The time complexity to find the leaf node corresponding to an abelian pattern P in the tree, after replacing the linked lists with the directly accessible arrays at both, the multiplicity nodes as well as the character nodes, is $O(\sigma_P)$.*

Proof. Let $P = \sum_{j=1}^{\sigma_P} m_{c_{i_j}} c_{i_j}$ be the given abelian pattern, then the sequence of the edges from the root node to the leaf node corresponding to P, in the tree is

$$c_{i_1} \rightarrow m_{c_{i_1}} \rightarrow c_{i_2} \rightarrow m_{c_{i_2}} \rightarrow \cdots \rightarrow c_{i_{\sigma_P}} \rightarrow m_{c_{i_{\sigma_P}}}$$

The time complexity to find an edge e that has edge label $m_{c_{i_k}}$ (or c_{i_k}), at the node hosting e, is $O(1)$ for all $1 \leq k \leq \sigma_P$. Hence the time complexity to find the leaf node corresponding to the abelian pattern P is $O(\sigma_P)$.

□

Replacing the Linked Lists at Character Nodes with Binary Searchable Arrays

It is important to note that it might not be possible to find a minimal perfect hash function ρ for every character set Σ , such that $\rho(c_i) < \rho(c_j) \Leftrightarrow c_i < c_j$ for each pair $c_i, c_j \in \Sigma$. In this case, we can convert the linked lists, of the outgoing edges at the character nodes of an abelian tree, into binary searchable arrays.

Observation 12. *The time to convert the sorted linked lists at the character nodes into binary searchable arrays is $O(mn)$; as the tree has $O(m)$ levels (Corollary 2) and $O(n)$ leaf nodes, which means the number of edges stored at the linked lists of the character edges is $O(mn)$.*

Observation 13. *The space complexity of an abelian tree with alternating levels, after replacing the linked lists at the multiplicity nodes with the directly accessible arrays and the linked lists at the character nodes with binary searchable arrays, is $O(mn)$.*

Lemma 24. *The time complexity to find the leaf node corresponding to an abelian pattern P in an abelian tree with alternating levels, that uses directly accessible arrays at its multiplicity nodes and binary searchable arrays at its character nodes, is $O(\sigma_P \log \sigma)$, where σ_P is the number of distinct characters in P.*

Proof. Let $P = \sum_{j=1}^{\sigma_P} m_{c_{i_j}} c_{i_j}$ be the given abelian pattern, then the sequence of edges from the root node to the leaf node corresponding to P, in the tree is

$$c_{i_1} \to m_{c_{i_1}} \to c_{i_2} \to m_{c_{i_2}} \to \cdots \to c_{i_{\sigma_P}} \to m_{c_{i_{\sigma_P}}}$$

The time complexity to find an edge e that has edge label $m_{c_{i_k}}$, at an internal node N that hosts e, is $O(1)$; as the edge labels are stored in directly accessible arrays at the multiplicity nodes.
So the total time complexity of finding all the multiplicity edges in the path from the root node to the leaf node corresponding to P is σ_P.

The time complexity to find an edge e that has edge label c_{i_k}, at an internal node N that hosts e, is $O(\log \sigma)$; as the edge labels at the character nodes are stored in binary searchable arrays, and there are $O(\sigma)$ outgoing edges at a character node. So the total time complexity of finding all the character edges in the path from the root node to the leaf node corresponding to P is $O(\sigma_P \log \sigma)$.

Hence the time complexity to find the leaf node, corresponding to the abelian pattern $P = \sum_{j=1}^{\sigma_P} m_{c_{i_j}} c_{i_j}$, in the tree is $O(\sigma_P \log \sigma)$.

□

3.5 Conclusion

In this chapter we have focused on the indexing strategies for abelian pattern matching. As the order of characters in the abelian patterns is not relevant, we imposed an external ordering on the characters for indexing the input text for abelian pattern matching.

We discussed how the parikh index can be used for the problem of abelian pattern matching. Then we presented a new data-structure *abelian tree* for indexing the input text. We also shed light on the tradeoff between the storage requirement and the efficiency of query processing for an abelian tree in contrast to the different data-structures used for storing the outgoing edges at the internal nodes of the tree.

Finally, we presented an abelian tree with alternating levels to avoid the overhead of keeping the unnecessary edges when the size of the alphabet is large.

Chapter 4

Approximate Abelian Pattern Matching

4.1 Introduction

The work presented so far has focused on the problem of finding *exact* abelian matches of a given pattern P in a text stream T. Both P and T are defined over the same alphabet Σ, and $\sigma := |\Sigma|$.

In this chapter, we consider the problem of finding *approximate* abelian matches of P. The chapter is organized as follows:

In Section 4.2, we provide a formal definition of an approximate abelian match of a pattern, and present three error models: the substitution error model, the insertion/deletion error model and the minimum operations error model. These models measure the degree of error in a substring S with respect to the given pattern P. We also call this *degree* of error, the *distance* between S and P or, the *cost* to transform S into P; and we use all these three terms interchangeably throughout the chapter.

In Section 4.3, we present an algorithm for approximate abelian pattern matching under the substitution error model.

Section 4.4 begins with several observations about the approximate abelian matches under the insertion/deletion error model. Then we describe the desired *output* for the approximate matches. And finally, we present two algorithms for approximate pattern matching under the insertion/deletion error model.

In the last section of the chapter, we deal with the approximate abelian pattern matching under the minimum operations error model.

4.2 Approximate Abelian Pattern Matching

In approximate abelian pattern matching, we tolerate up to a specified number of errors in a matching substring S of the input text T, i.e. the frequencies of one or more characters in S can be different from the specified frequencies of those characters in P.

To quantify the degree of error in a substring S with respect to P, we define three error models. Depending on the error model, S is of length $m := |P|$ or it is of an arbitrary length.

4.2.1 Error Models for Approximate Abelian Pattern Matching

We use three error models to define the distance between the pattern to be found $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$ and another pattern $P' = \sum_{i=1}^{\sigma} m'_{c_i} c_i$ that corresponds to an arbitrary substring of T.

Substitution Error Model: In this model, we only consider length-m substrings of T. The *substitution distance* between P and P' is then defined as $\frac{1}{2} \cdot \sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}|$. This is always an integer. The substitution error model is further elaborated in Section 4.3.

Insertion/Deletion (InDel) Error Model: In this model, we consider arbitrary length substrings of T. The *InDel distance* between P and P' is then defined as $\sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}|$. The details of this model are presented in Section 4.4.

Minimum Operations (MinOp) Error Model: We again consider arbitrary length substrings of T. The *MinOp distance* between P and P' is defined as the minimum number of letter substitutions and insertions/deletions to transform P' into P. Let m' be the length of P', then the *MinOp distance* between P and P' is computed by the formula

$$\frac{1}{2} \cdot \left\{ |m - m'| + \sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}| \right\}$$

Section 4.5 sheds light on the minimum operations error model.

4.2.2 Formal Problem Definition

Formally, given an abelian pattern $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$ of length m, an error threshold t, an error model and a text $T \in \Sigma^n$, the *approximate abelian pattern matching problem* is to find all *approximate* occurrences of P in T under the specific error model used; i.e. we want to find all positions i, such that the distance between P and the abelian pattern corresponding to the substring $T_i \cdots T_j$ is at most t.

Now we give several definitions that we use later in the chapter. Let $P' = \sum_{i=1}^{\sigma} m'_{c_i} c_i$ be the abelian pattern corresponding to an arbitrary substring of T, then:

Definition 12. *A character is called a* spare character *if it has higher frequency in P' than its specified frequency in P.*

Definition 13. *A character is called a* deficit character *if it has lower frequency in P' than its specified frequency in P.*

Definition 14. *If ch is a spare character, then $m'_{ch} - m_{ch}$ is called the* spareness *of ch.*

Definition 15. *If ch is a deficit character, then $m_{ch} - m'_{ch}$ is called the* deficiency *of ch.*

4.3 Approximate Abelian Pattern Matching Under the Substitution Error Model

In approximate abelian pattern matching under the substitution error model, we are interested only in the length-m substrings of T. For a given abelian pattern $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$, the substitution distance between P and another abelian pattern $P' = \sum_{i=1}^{\sigma} m'_{c_i} c_i$ is defined as the minimum number of character substitutions to transform P' into P. For example, let $P = 2a + 2b$ and $P' = a + 3b$, then the substitution distance between P and P' is one, as by substituting one b with an a in P', we transform P' into P.

Observation 14. *In a substitution operation, a spare character is substituted by a deficit character.*

Lemma 25. *If $|P'| = m$, then the sum total of the sparenesses of all the spare characters of P' is equal to the sum total of the deficiencies of all the deficit characters of P'. That is,*

$$\sum_{c\in\Sigma_S}(m'_c - m_c) = \sum_{c\in\Sigma_D}(m_c - m'_c)$$

where Σ_S denote the set of the spare characters of P' (recall that c is a spare character of P' if $m'_c > m_c$) and Σ_D denotes the set of the deficit characters of P' (recall that c is a deficit character of P' if $m'_c < m_c$).

Proof. As $|P'| = m$, therefore;

$$\sum_{c\in\Sigma} m'_c = \sum_{c\in\Sigma} m_c$$

As Σ_S and Σ_D are disjoint sets, therefore;

$$\sum_{c\in\Sigma} m'_c = \sum_{c\in\Sigma\setminus(\Sigma_S\cup\Sigma_D)} m'_c + \sum_{c\in\Sigma_S} m'_c + \sum_{c\in\Sigma_D} m'_c$$

and

$$\sum_{c\in\Sigma} m_c = \sum_{c\in\Sigma\setminus(\Sigma_S\cup\Sigma_D)} m_c + \sum_{c\in\Sigma_S} m_c + \sum_{c\in\Sigma_D} m_c$$

Hence,

$$\begin{aligned}
\sum_{c\in\Sigma\setminus(\Sigma_S\cup\Sigma_D)} m'_c + \sum_{c\in\Sigma_S} m'_c + \sum_{c\in\Sigma_D} m'_c &= \sum_{c\in\Sigma\setminus(\Sigma_S\cup\Sigma_D)} m_c + \sum_{c\in\Sigma_S} m_c + \sum_{c\in\Sigma_D} m_c \\
\Rightarrow \sum_{c\in\Sigma_S} m'_c + \sum_{c\in\Sigma_D} m'_c &= \sum_{c\in\Sigma_S} m_c + \sum_{c\in\Sigma_D} m_c \\
\Rightarrow \sum_{c\in\Sigma_S} m'_c - \sum_{c\in\Sigma_S} m_c &= \sum_{c\in\Sigma_D} m_c - \sum_{c\in\Sigma_D} m'_c \\
\Rightarrow \sum_{c\in\Sigma_S}(m'_c - m_c) &= \sum_{c\in\Sigma_D}(m_c - m'_c)
\end{aligned}$$

□

Corollary 4. *From Observation 14 and Lemma 25, it follows that the substitution distance is always an integer.*

Lemma 26. *The* substitution distance *between an abelian* $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$ *and another abelian pattern* $P' = \sum_{i=1}^{\sigma} m'_{c_i} c_i$ *is*

$$\frac{1}{2} \cdot \sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}|$$

Proof. One character substitution has two-fold effect; on one hand, it reduces the frequency of a spare character and on the other hand, it increases the frequency of a deficit character. Thus one substitution operation decreases the sum of the absolute differences in the frequencies of the characters of P' and P by 2. Hence, the number of substitutions to transform P' into P is $\frac{1}{2} \cdot \sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}|$.

□

Now we present an algorithm for finding the approximate matches of a given abelian pattern P of length m in an input text T of length n, under the substitution error model.

4.3.1 Basic Idea of the Algorithm

We set a window of length m at the beginning of the input text T, and compute the distance between P and the abelian pattern corresponding to the substring contained in the window, using the substitution error model. If the computed distance is less than t, then the current window contains an approximate abelian match of P, and we output the starting position of the current window.

After that, we advance the window towards right by one position and check the new window for a match. This way, we move the window along the whole text and report the starting positions of the windows that contain approximate abelian matches of P.

4.3.2 The Algorithm

The pseudo code of the algorithm for approximate abelian pattern matching under the substitution error model is presented in *Algorithm 4*.

We use an array DFV (difference frequency vector) of σ elements to store the differences – in the frequencies of the characters – between P and the substring contained in the current window.

Algorithm 4 Approximate Abelian Pattern Matching under the Substitution Error Model

Input: A pattern P of length m, a text stream $T = T[1] \dots T[n]$, a hash function ρ and an error threshold t

Output: Starting positions of all approximate matches of P in T

$\triangleright$ *Build difference frequency vector (DFV) for the first m characters*
for $i = 1$ to σ **do**
    $DFV[i] \leftarrow 0 - P[i]$
for $i = 1$ to m **do**
    Increment $DFV[\rho(T[i])]$ by 1
$\triangleright$ *Calculate the number of substitutions required to transform the substring contained in the current window into P*
$subs \leftarrow 0$
for $i = 1$ to σ **do**
    $subs \leftarrow subs + ABS(DFV[i])$
    $\triangleright$ *$ABS(x)$ returns the absolute value of x*
$subs \leftarrow subs/2$
if $subs \leq t$ **then**
    output 1
$i \leftarrow 2$
while $i \leq n - m + 1$ **do**
    if $T[i-1] \neq T[i+m-1]$ **then**
        Decrement $DFV[\rho(T[i-1])]$ by 1
        Increment $DFV[\rho(T[i+m-1])]$ by 1
        if $(DFV[\rho(T[i-1])] < 0)$ **then**
            if $(DFV[\rho(T[i+m-1])] > 0)$ **then**
                $subs \leftarrow subs + 1$
        else
            if $(DFV[\rho(T[i+m-1])] \leq 0)$ **then**
                $subs \leftarrow subs - 1$
    if $subs \leq t$ **then**
        output i
    $i \leftarrow i + 1$

In the first phase of the algorithm (*line 1-10*), we initialize the array DFV for the first m characters of T. This is done in the first four lines of the algorithm.

After that we need to compute the *substitution distance* between the current window and P. Let S denotes the substring contained in the current window, then the number of the substitution operations required to transform S into P is half of the sum of the absolute differences in the frequencies of the characters of S and P (Lemma 26). We compute the number of substitution operations for the current window in the variable *subs* (*for loop* of *line 6-8*).

If *subs* is less than or equal to the error threshold t, then we output the first position of the text as starting position of an approximate match of P.

In the next phase of the algorithm (*line 11-24*), we move the sliding window along the whole text, and report the starting positions of all those windows for which the value of *subs* is less than or equal to t.

We proceed incrementally, and move the current window towards the right by one position (*line 11* and *line 24*). The *new* current window is different from the previous window at two places: it does not contain the first character of the previous window; and the last character of the current window was not part of the previous window. Let x denotes the first character of the previous window and y denotes the last character of the current window. If x and y are the same character, then the current window is also the same as the previous window and so are the values of its DFV and *subs*.

However, if x and y are not the same character, then we construct DFV corresponding to the current window by decrementing the frequency of x by 1 (*line 14*) and incrementing the frequency of y by 1 (*line 15*) in the array DFV. After constructing DFV for the current window, we need to update the value of *subs* so that it corresponds to DFV of the current window.

There are four possibilities for x and y:

(i) $DFV[\rho(x)] < 0$ **and** $DFV[\rho(y)] \leq 0$**:** This means that x has either become a deficit character in the current window or if it was already a deficit character in the previous window then its deficiency is increased by one in the current window. Similarly, y was a deficit character in the previous window and in the current window, its deficiency is decreased by one.

The value of *subs* remains same, as one earlier substitution of a *spare* character by y would be replaced by the substitution of the same character by x.

(ii) $DFV[\rho(x)] < 0$ **and** $DFV[\rho(y)] > 0$**:** This means that x has either become a deficit character in the current window or if it was already a deficit character in the previous window then its deficiency is increased by one in the current window. Moreover, y has either become a spare character in the current window or if it was already a spare character in the previous window then its spareness is increased by one in the current window.

So we have to do an additional substitution (substitute y by x) to transform the current window into P. Hence the value of *subs* is increased by 1 in this case, and we update *subs* in *line 18* of the algorithm.

(iii) $DFV[\rho(x)] \geq 0$ **and** $DFV[\rho(y)] > 0$**:** This means that x was a spare character in the previous window and in the current window its spareness is decreased by one. Moreover, y has either become a spare character in the current window or if it was already a spare character in the previous window then its spareness is increased by one in the current window.

So, instead of substituting x by a *deficit* character, we substitute y by that characters, and other things remain same. The value of *subs* remains unaffected in this case.

(iv) $DFV[\rho(x)] \geq 0$ **and** $DFV[\rho(y)] \leq 0$**:** This means that x was a spare character in the previous window and in the current window its spareness is decreased by one. Similarly, y was a deficit character in the previous window and in the current window, its deficiency is decreased by one.

If the value of *subs* is 1, then in the previous window, the only substitution operation was the substitution of x by y, which is no more required for the current window, making the current window an exact match.

If the value of *subs* is greater than 1, then in the previous window, we had to substitute x by a required character say y' (y' could also be same as y). Similarly, in the previous window, another spare character say x' (x' could also be same as x) was substituted by y. Now we can do the work of these two substitution operations by one substitution, namely, the substitution of x' by y'.

Hence the value of *subs* goes down by 1 in this case, and we update *subs* in *line 21* of the algorithm.

If the new value of *subs* is less than or equal to t, then we output the starting position of the current window (*line 23*). This way, we move the window along the whole text and report the starting positions of all those windows for which the value of $subs$ is less than or equal to t.

4.3.3 Complexity Analysis

The algorithm reads and processes every character in T exactly twice; for the first time to add its count in DFV, and for the second time to remove its count from DFV. Each time the window is slid towards right, we construct the DFV and $subs$ corresponding to the new window in constant time, so the overall time complexity of this algorithm is $\Theta(n)$.

At any point in time, this algorithm keeps in memory only one frequency vectors DFV, and one integer variable $subs$. As we assume that the hash function ρ is a minimal perfect hash function, so the array DFV requires $O(\sigma)$ storage. Hence the space complexity of this algorithm is $O(\sigma)$, in addition to the space required for the input and the output.

4.4 Approximate Abelian Pattern Matching under the Insertion/Deletion (InDel) Error Model

In the problem of approximate abelian pattern matching under the InDel error model, a matching pattern needs not to be of the fixed length m. For a given abelian pattern $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$, the InDel distance between P and another abelian pattern $P' = \sum_{i=1}^{\sigma} m'_{c_i} c_i$ is defined as the minimum number of insertions of new characters in P' and deletions of existing characters of P' to transform P' into P. For example, let $P = 2a+2b+c$ and $P' = 3a+3b$, then the InDel distance between P and P' is three, as we insert one c in P' and delete one a and one b from P' to transform P' into P. While transforming P' into P, the deficit characters are inserted in P' and the spare characters are deleted from P'. Hence, the InDel distance between P and P' is determined by the formula $\sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}|$.

InDel Distance versus q-Gram Distance: The notion of q-gram distance between two strings was introduced in [36]. A q-gram is a string of length q and the q-gram distance between two strings is based on counting the number of the occurrences of different q-grams in the two strings. For the special case of $q = 1$, the q-gram distance between two substrings x and y is the same as the InDel distance between the abelian patterns corresponding to x and y.

There are several observations regarding the approximate abelian matches under the InDel error model.

Observation 15. *Let $S = T_i \cdots T_j$ be a t-approximate match of P ($|P| = m$) under the InDel error model, then*

$$m - t \leq j - i + 1 \leq m + t$$

Observation 16. *Let $S = T_i \cdots T_j$ be an approximate match of P under the InDel error model, then it is possible that another substring $S' = T_{i'} \cdots T_{j'}$ (such that $i' \geq i$, $j' \leq j$ and $j' - i' + 1 \geq m - t$) is **not** an approximate match of P under the InDel error model.*

Example 2. *Let $P = 5a+5b$ and error threshold $t = 3$; then $S = aaaaabbbcccbb$ (with length 13) is an approximate match of P under the InDel error model, whereas $S' = aaaaabbbcccb$ (with length 12) is not an approximate match.*

4.4.1 The Desired Output

Although we are interested in finding all variable length substrings of T that approximately match P under the InDel error model, there could be situations where several matching substrings start at the same position in T.

For example, for $P = 2a + 3b$ and $T = aabcbcb$ with error threshold $t = 2$, three substrings, that approximately match P, start at position 1 of T. The ending positions of these matching substrings are 3, 5 and 7 respectively. Here it is important to note that the substrings $aabc$ (position 1-4 of T) and $aabcbc$ (position 1-6 of T) are not matching substrings. So the range of the positions, starting at 3 and ending at 7, does not make an interval of the ending positions of the approximate matches of P that start at position 1. However, all the matching substrings of P that start at position 1 are contained in the match that starts at position 1 and ends at position 7.

To avoid redundant information, we define the notion of maximality of an approximate abelian match of P and we report only the maximal approximate matches of P.

Definition 16. *A* maximal approximate match *of P is a substring $S = T_i \cdots T_j$ such that S is an approximate match of P and no substring $S' = T_{i'} \cdots T_{j'}$ with $i' \leq i$ and $j' \geq j$ is an approximate match of P.*

Now in the above example we output only the match that starts at position 1 and ends at position 7 of T.

4.4.2 Basic Idea and Building Block of the Algorithm

We define a *potential match* and use it as a building block in our algorithm for approximate abelian pattern matching under the InDel error model.

Definition 17. *A* potential match *is a substring $S = T_i \cdots T_j$ such that a substring $S' = T_i \cdots T_{j'}$ with $j' > j$* can *be an approximate match of P. For example, for $P = 5a + 5b$ and error threshold $t = 3$, the substring $S = aaaaabbbcccb$ is a potential match because if the next character in the text happens to be the character b, then the substring $S' = aaaaabbbcccbb$ would become an approximate match of P. A potential match can also be an approximate match.*

Observation 17. *A potential match cannot be longer than $m + t - 1$.*

Observation 18. *Being a potential match is an* anti-monotone *property [17], i.e. if $S = T_i \cdots T_j$ is not a potential match then no substring $S' = T_i \cdots T_{j'}$ with $j' > j$ can be a potential match.*

Lemma 27. *A substring S is a potential match only if the number of deletions required to transform S into P is less than or equal to the error threshold t.*

Proof. Let $S = T_i \cdots T_j$ be a potential match, and we require x insertions and y deletions to transform S into P. Assume that $y > t$. Now if we extend S towards the right, the newly added characters can only decrease insertion operations; which means the cost would remain at least y which is greater than t, hence S is not a potential match.

□

Lemma 28. *If the number of deletions required to transform a substring S into P is less than or equal to the error threshold t, and the length of S is less than $m + t$, then S is a potential match.*

Proof. Let $S = T_i \cdots T_j$, and it requires x insertions and y deletions to transform S into P, where $y \leq t$.

This means $j - i + 1 = m - x + y$, and S can be extended towards right up to at most $t + x - y$ positions (this is because of the fact that an approximate abelian match under the InDel error model can be at most $m + t$ characters long). In this extension procedure, the newly added characters can replace only insertion operations. So we need to show that $t + x - y \geq x$, so that, in the best case, every newly added character replaces one insertion operation, and as x equals 0, this makes $x + y \leq t$. Hence the new (extended) substring is an approximate abelian match of P, and by definition S is a potential match.

Now, $t + x - y \geq x \Leftrightarrow t - y \geq 0 \Leftrightarrow t \geq y$, which is in the premise of the lemma.

$\square$

Corollary 5. *All approximate matches of length less than $m + t$ are also potential matches.*

Basic Idea

We move a window of length $m - t$ along the text T; and if the substring contained in the window is a potential match, then we extend this substring towards the right until the extended substring is not a potential match. In this process we may encounter one or more approximate matches. We keep the longest (the latest encountered) approximate match, and if this approximate match is also a maximal match, then we output the starting position of the current window along with the ending position of the maximal match.

We keep the length of the window $m - t$ because of the fact that an approximate match under the InDel error model is at least $m - t$ characters long (Observation 15). Hence we do not skip any approximate match if the length of the sliding window is $m - t$.

As the window is slid from the left towards the right along the text T, so the starting positions of all the matches that has been output till a point in time are monotonically increasing. Thus it suffices to keep the ending position of the *latest* output approximate match for deciding the maximality of a match. If a newly found match has its ending position greater than the ending position of the latest output match then it is a maximal match and we output it.

4.4.3 The Algorithm

The algorithm for approximate abelian pattern matching under the InDel error model is outlined in *Algorithm 5*.

In the first four lines of the algorithm, we initialize the array DFV for the first $m - t$ characters of T. Then in the *for loop* of *line 6*, we compute the number of insertion and deletion operations required to transform the substring contained in the current window into P in the variables *ins* and *del* respectively.

The value of the variable $MaximalMatch$ at any point in time shows the ending position of the *latest* output (maximal) match; and its initial value is set at 0 (*line 11*), indicating that no maximal match has yet been output. If the current window contains a potential match (according to Lemma 28), then we call the sub-routine $CheckForMatch$ in *line 13*.

The sub-routine $CheckForMatch$ takes seven arguments; the text stream T (call by reference), DFV (also call by reference), number of insertion and deletion operations (*ins* and *del*) to transform the substring contained in the current window into P, the error threshold t, the last position of the current window k, and the hash function ρ for direct accessibility of the characters of T in the array DFV.

The sub-routine $CheckForMatch$ extends the current window towards the right to the extent, where the substring contained in the extended window no longer remains a potential match. In the process of extension of the current window, one or more approximate matches of P may be encountered. In this situation, the sub-routine returns the ending position of the longest of the approximate matches that are encountered during the extension of the window. The variables *ins* and *del* are local variables of the sub-routine; however, as the array DFV is passed by reference to the sub-routine, therefore, once the extension of the window is stopped, the sub-routine undoes the changes made in the DFV during the extension of the window. The pseudo code of the sub-routine is presented in *Algorithm 6*.

In *line 1* of the sub-routine, we initialize the variable *match* with 0, indicating that no match has yet been found. If the current window contains an approximate match of P, then *match* takes the value of the last position of the current window (*line 3*).

In the *while loop* of *line 6-17*, we extend the current window towards the right until it contains a potential match. In *line 13* of the loop, we check the extended substring of the current iteration for a match; and if this substring is an approximate match, then we update the value of *match* with this new *longer* match (*line 14*). However, if the extended substring of the current iteration is not an approximate match, then we test it for being a potential match and set the flag *PotentialMatch* to *false* if it fails the test (*line 17*).

There is a special case when the *while loop* does an extra iteration. If the extended substring of length $m + t$ is an approximate match, we do not test it for a potential match (thus do no change the flag *PotentialMatch*), although the extended substring of the current iteration is not a potential match (Corollary 5). However, in the very next iteration of the *while loop*, the extended substring is recognized as *not* a potential match and the flag

Algorithm 5 Approximate Abelian Pattern Matching under the InDel Error Model

Input: A pattern P of length m, a text stream $T = T[1] \dots T[n]$, a hash function ρ and an error threshold t

Output: Starting and ending positions of all *maximal* approximate matches of P in T

$\triangleright$ *Build difference frequency vector (DFV) for the first $m - t$ characters*
for $i = 1$ to σ **do**
    $DFV[i] \leftarrow 0 - P[i]$
for $i = 1$ to $m - t$ **do**
    Increment $DFV[\rho(T[i])]$ by 1
$\triangleright$ *Calculate the number of insertion and deletion operations required to transform the substring contained in the current window into P*
$ins \leftarrow del \leftarrow 0$
for $i = 1$ to σ **do**
    if $DFV[i] > 0$ **then** $\triangleright$ *spare characters to be deleted*
        $del \leftarrow del + DFV[i]$
    else if $DFV[i] < 0$ **then** $\triangleright$ *deficit characters to be inserted*
        $ins \leftarrow ins + ABS(DFV[i])$
        $\triangleright$ *$ABS(x)$ returns the absolute value of x*
$MaximalMatch \leftarrow 0$
if $del \le t$ **then** $\triangleright$ *current window contains a potential match*
    $match \leftarrow CheckForMatch(T, DFV, ins, del, t, m - t, \rho)$
    if $match > MaximalMatch$ **then**
        output $(1, match)$
        $MaximalMatch \leftarrow match$
$i \leftarrow 2$
while $i \le n - m + t + 1$ **do**
    if $T[i-1] \ne T[i + m - t - 1]$ **then**
        Decrement $DFV[\rho(T[i-1])]$ by 1
        Increment $DFV[\rho(T[i + m - t - 1])]$ by 1
        if $(DFV[\rho(T[i-1])] < 0)$ **then**
            if $(DFV[\rho(T[i + m - t - 1])] > 0)$ **then**
                $ins \leftarrow ins + 1$
                $del \leftarrow del + 1$
        else
            if $(DFV[\rho(T[i + m - t - 1])] \le 0)$ **then**
                $ins \leftarrow ins - 1$
                $del \leftarrow del - 1$
    if $del \le t$ **then** $\triangleright$ *current window contains a potential match*
        $match \leftarrow CheckForMatch(T, DFV, ins, del, t, i + m - t - 1, \rho)$
        if $match > MaximalMatch$ **then**
            output $(i, match)$
            $MaximalMatch \leftarrow match$
    $i \leftarrow i + 1$

Algorithm 6 CheckForMatch $(T, DFV, ins, del, t, k, \rho)$

Input: A text stream T, a difference frequency vector DFV corresponding to the current window of length $m - t$, number of insertions ins and deletions del, error threshold t , the last position of the current window k, and the hash function ρ

Output: An integer $match$ that contains the ending position of the longest match starting at the first position of the current window and 0 if such a match does not exist

$match \leftarrow 0$
if $ins + del \leq t$ **then** ▷ *current window contains a match*
    $match \leftarrow k$
$k' \leftarrow k$
$PotentialMatch \leftarrow True$
while $PotentialMatch = True$ **do**
    $k' \leftarrow k' + 1$
    Increment $DFV[\rho(T[k'])]$ by 1
    if $DFV[\rho(T[k'])] \leq 0$ **then** ▷ *Deficiency of* $T[k']$ *is reduced by 1*
        $ins \leftarrow ins - 1$
    else ▷ *Spareness of* $T[k']$ *is increased by 1*
        $del \leftarrow del + 1$
    if $ins + del \leq t$ **then**
        $match \leftarrow k'$
    else ▷ *check for potential match*
        if $del > t$ **then**
            $PotentialMatch \leftarrow False$
while $k' > k$ **do** ▷ *undo the changes made in DFV*
    Decrement $DFV[\rho(T[k'])]$ by 1
    $k' \leftarrow k' - 1$
return $match$

$PotentialMatch$ is set to *false*.

The *while loop* terminates when it sees that the extended substring is not a potential match. This is due to the anti-monotone property of a potential match (Observation 18).

Once we are out of the *while loop* of *line 6-17* of the sub-routine, we undo the changes made in the array DFV during the iterations of the *while loop*. And finally the sub-routine $CheckForMatch$ returns the ending position of the longest match it has encountered in the window extension process; or a value of zero is returned if no match is found during the extension of the window.

In *line 14* of the *main algorithm (Algorithm 5)*, we perform a maximality test on the value returned by the sub-routine $CheckForMatch$; and if the value returned by the sub-routine is greater than the current value of $MaximalMatch$, then we report the starting and ending positions of the match (*line 15*) and update the value of $MaximalMatch$ to the ending position of the new maximal match (*line 16*).

In the *while loop* at *line 18* of the *main algorithm*, we slide the window along the whole text and output all the maximal approximate matches of P in T.

We move the current window of length $m-t$ towards the right by one position (*line 17* and *line 35*). The *new* current window is different from the previous window at two places: it does not contain the first character of the previous window; and the last character of the current window was not part of the previous window. Let x denotes the first character of the previous window and y denotes the last character of the current window. If x and y are the same character, then the current window is also the same as the previous window and so are the values of its DFV, and ins and del.

However, if x and y are not the same character, then we construct DFV corresponding to the current window by decrementing the frequency of x by 1 (*line 20*) and incrementing the frequency of y by 1 (*line 21*) in the array DFV. After constructing DFV for the current window, we need to update the values of ins and del, such that now the values of these variables correspond to DFV of the current window.

There are four possibilities for x and y:

(i) $DFV[\rho(x)] < 0$ **and** $DFV[\rho(y)] \leq 0$**:** This means that x has either become a deficit character in the current window or if it was already a deficit character in the previous window then its deficiency is increased by one in the current window. Similarly, y was a deficit character in the previous window and in the current window, its deficiency is decreased by one.

In this case, the value of *ins* remains unchanged, as one earlier insertion of the deficit character y is replaced by the insertion of the deficit character x. The value of *del* is not affected in this case.

(ii) $DFV[\rho(x)] < 0$ **and** $DFV[\rho(y)] > 0$**:** This means that x has either become a deficit character in the current window or if it was already a deficit character in the previous window then its deficiency is increased by one in the current window. Moreover, y has either become a spare character in the current window or if it was already a spare character in the previous window then its spareness is increased by one in the current window.

So we have to do an additional insertion of x and an additional deletion of y, to transform the current window into P. Hence the values of both *ins* and *del* are increased by 1 in this case, and we update these variables in *lines 24-25* of the algorithm.

(iii) $DFV[\rho(x)] \geq 0$ **and** $DFV[\rho(y)] > 0$**:** This means that x was a spare character in the previous window and in the current window its spareness is decreased by one. Moreover, y has either become a spare character in the current window or if it was already a spare character in the previous window then its spareness is increased by one in the current window.

In this case, the value of *del* remains unchanged, as one earlier deletion of the spare character x is replaced by the deletion of the spare character y. The value of *ins* is not affected in this case.

(iv) $DFV[\rho(x)] \geq 0$ **and** $DFV[\rho(y)] \leq 0$**:** This means that x was a spare character in the previous window and in the current window its spareness is decreased by one. Similarly, y was a deficit character in the previous window and in the current window, its deficiency is decreased by one.

So as we do not have to do an earlier deletion of x and an earlier insertion of y, to transform the current window into P. Hence the values of both *ins* and *del* are decreased by 1 in this case, and we update these variables in *lines 28-29* of the algorithm.

After updating the values of *ins* and *del*, if the current window contains a potential match (according to Lemma 28), then we call the sub-routine $CheckForMatch$ in *line 31*. In *line 32* of the *main algorithm*, we perform a maximality test on the value returned by the sub-routine $CheckForMatch$;

and if the value returned by the sub-routine is greater than the current value of $MaximalMatch$, then we report the starting and ending positions of the match (*line 33*) and update the value of $MaximalMatch$ to the ending position of the new maximal match (*line 34*). After that we advance the window towards the right by one position (*line 35*).

This way, we slide the window along the whole text and output all the maximal approximate matches of P in T.

4.4.4 Complexity Analysis

The worst case time complexity of the algorithm is $O(n + Pt)$. Here P is the number of the potential matches of length $m - t$ in the text stream T (hence we call the sub-routine $CheckForMatch$, P times during the execution of the algorithm). The time complexity of the sub-routine $CheckForMatch$ is $O(t)$, as the *while loop* at *line 6* of the sub-routine may iterate up to $2t + 1$ times. Hence the overall time complexity of the algorithm is $O(n + Pt)$.

The algorithm keeps in memory an array of σ elements and a constant number of variables, hence the space complexity of the algorithm is $O(\sigma)$, in addition to the space required for the input and the output.

Time and Space Complexity of the q-Gram Based Algorithm: The problem of approximate abelian pattern matching under the InDel error model can also be solved using an algorithm based on the concept of q-grams (for the special case of $q = 1$) [36]. It requires $O(m\sigma)$ time for pre-processing the pattern and $O(n \log t)$ time for processing the input text. The working space requirement of the algorithm is $O(m\sigma)$ [36].

4.4.5 Using a Window of Flexible Length

In Algorithm 5, we keep the length of the window $m - t$, so that, we do not miss any approximate match during the window sliding process (as the minimum length of an approximate match under the InDel error model is $m - t$ (Observation 15)).

Here we present an algorithm, for approximate abelian pattern matching under the InDel error model, that uses a sliding window of flexible length. This means that now we also need to take care of the fact that, during the window sliding process, no maximal approximate match of P is missed from being reported. In the following, we introduce the notion of a *safe window* and shed light on its properties.

Definition 18. *A* safe window *is a window that does not contain any unreported maximal match that begins at the starting position of the window.*

Lemma 29. *If the number of the insertion operations, required to transform the substring contained in a window to the abelian pattern P, is greater than t, then the window is a safe window.*

Proof. Let i and j be the respective starting and ending positions of the window under consideration. Let x be the number of insertion operations required to transform the substring $S = T_i \cdots T_j$ into P. Note that S is not an approximate match, as the cost to transform S into P (under the InDel error model) is at least x, which is greater than t.

If we move the right boundary of the window towards left, then the improvement (if any), in the cost of transforming the new (shorter) substring into P, would be only in the deletions part; the number of insertions (which is $x > t$) will remain intact. Hence no substring $S' = T_i \cdots T_{j'}$ is an approximate match of P for all $j' \leq j$.

As the window does not contain *any* approximate match of P that begins at i (the starting position of the window), therefore, the window is a safe window.

□

Corollary 6. *The length of a* non-*safe window is at least $m - t$.*

Corollary 7. *If a window of length l (where $l \leq m$) is not a potential match, then it is a safe window.*

Lemma 30. *If the substring contained in the current window is not an approximate match of P, and the right boundary of the window is not greater than the ending position of the latest reported maximal match in T, then the window is a safe window.*

Proof. Let i and j be the respective starting and ending positions of the current window. Let M be the latest reported maximal match, and i', j' be the starting and ending positions of M respectively. As we slide the window from the left towards the right along the input text T, and the current window is not an approximate match of P; therefore, $i' < i$. Now if there exists an approximate match of P that starts at i and ends at a position $\hat{j}$ in T (where $\hat{j} < j$), that match is not maximal as $i > i'$ and $\hat{j} < j'$.

Hence, the current window is a safe window. □

The Algorithm

We use two indexes l and r to keep track of the left and right boundaries of the current window in the text. The initial values of l and r are set at 1 and $m - t$ respectively.

We slide the window along the input text in the following manner:

1. If the current window contains a potential match, then we extend r towards right until the current window does not contain a potential match. In the process of extending the right boundary of the window, if we find a maximal match, then we report it.

   After the current window is no more a potential match

   (a) We move l towards right by one position.

   (b) If the new current window contains a potential match, then, once again, we extend r towards right until the current window does not contain a potential match; and if a maximal match is found while extending the right boundary of the window, then we report it.

   (c) We move r towards left until the window becomes a safe window.

2. We slide the safe window along the text (by simultaneously incrementing the values of both l and r by one), and whenever the current window contains a potential match, we go to the step 1, and if the current window, at any time, becomes *non*-safe then we go to the step 1c.

As the *safeness* of the current window is always preserved, so we can slide a window of flexible length along the text without having the danger of skipping a maximal match from being reported.

Pseudo code of this algorithm is outlined in Algorithm 7. In the first three lines, we do the initialization of the array DFV and other variables. In *line 4*, if the current window contains a potential match then we extend it towards the right (*line 5*) and report a maximal match if one is found in the process of extension of the window.

In *line 9* of the algorithm, we move the left boundary of the current *extended* window towards the right by one position, and in *line 10*, we update DFV by decrementing the frequency of the first character of the previous window by one.

Let x be the first character of the previous window. If the frequency of x in the current window is less than its specified frequency in P (*line 11*) then it

Algorithm 7 Approximate Abelian Pattern Matching under the InDel Error Model using a Window of Flexible Length

Build DFV for $T_1 \cdots T_{m-t}$
Calculate the number of insertion and deletion operations required to transform $T_1 \cdots T_{m-t}$ into P in variables ins and del respectively
$MaximalMatch \leftarrow 0, \quad match \leftarrow 0, \quad l \leftarrow 1, \quad r \leftarrow m - t$
if $del \leq t$ **then** ▷ *current window contains a potential match*
    $ExtendWindow(T, DFV, match, ins, del, t, r, \rho)$
    if $match > MaximalMatch$ **then**
        $MaximalMatch \leftarrow match$
        output $(l, match)$
    $l \leftarrow l + 1$
    Decrement $DFV[\rho(T[l-1])]$ by 1
    if $(DFV[\rho(T[l-1])] < 0)$ **then**
        $ins \leftarrow ins + 1$
    else
        $del \leftarrow del - 1$
    if $del \leq t$ **then**
        Extend the right boundary of window and report a (maximal) match if one is found in the process of extension
    if $ins \leq t$ **then** ▷ *The window is not safe*
        $MakeSafe(T, DFV, MaximalMatch, ins, del, t, l, r, \rho)$
$l \leftarrow l + 1$
$r \leftarrow r + 1$
while $r \leq n$ **do**
    if $T[l-1] \neq T[r]$ **then**
        Adjust DFV according to $T_l \cdots T_r$ and compute the values of ins and del for the new window
    if $del \leq t$ **then** ▷ *current window contains a potential match*
        Extend the right boundary of window and report a (maximal) match if one is found in the process of extension
        $l \leftarrow l + 1$
        Adjust DFV according to $T_l \cdots T_r$ and compute the values of ins and del for the new window
        if $del \leq t$ **then**
            Extend the right boundary of window and report a (maximal) match if one is found in the process of extension
    if $ins \leq t$ **then** ▷ *The window is not safe*
        $MakeSafe(T, DFV, MaximalMatch, ins, del, t, l, r, \rho)$
    $l \leftarrow l + 1$
    $r \leftarrow r + 1$

Algorithm 8 ExtendWindow $(T, DFV, match, ins, del, t, r, \rho)$

Input: A text stream T, a difference frequency vector DFV corresponding to the current window, a place holder *match* to store the position of the longest match found during extensions of the window, number of insertions *ins* and deletions *del*, error threshold t , the right boundary of the current window r, and the hash function ρ. *(all the parameters are passed as reference)*

Output: Extend the right boundary of the window until the window does not contain a potential match; and store the position of the longest match found (if any) during the extension process in the variable *match*

$match \leftarrow 0$
if $ins + del \leq t$ **then** ▷ *current window contains a match*
    $match \leftarrow r$
$PotentialMatch \leftarrow True$
while $PotentialMatch = True$ **do**
    $r \leftarrow r + 1$
    Increment $DFV[\rho(T[r])]$ by 1
    if $DFV[\rho(T[r])] \leq 0$ **then** ▷ *Deficiency of* $T[r]$ *is reduced by 1*
        $ins \leftarrow ins - 1$
    else ▷ *Spareness of* $T[r]$ *is increased by 1*
        $del \leftarrow del + 1$
    if $ins + del \leq t$ **then**
        $match \leftarrow r$
    else ▷ *check for potential match*
        if $del > t$ **then**
            $PotentialMatch \leftarrow False$

Algorithm 9 MakeSafe $(T, DFV, MaximalMatch, ins, del, t, l, r, \rho)$

Input: A text stream T, a difference frequency vector DFV corresponding to the current window, position of the latest reported maximal match $MaximalMatch$, number of insertions ins and deletions del, error threshold t, the left boundary of the current window l, the right boundary of the current window r, and the hash function ρ. *(all the parameters are passed as reference)*

Output: Shrink the right boundary of the window towards left until it becomes safe. Report a maximal match if a new one is found in the process of making the window safe.

while $(ins \leq t$ **and** $r > MaximalMatch)$ **do**
    Decrement $DFV[\rho(T[r])]$ by 1
    if $(DFV[\rho(T[r])] \geq 0)$ **then** ▷ *Spareness of* $T[r]$ *is reduced by 1*
        $del \leftarrow del - 1$
    else ▷ *Deficiency of* $T[r]$ *is increased by 1*
        $ins \leftarrow ins + 1$
    $r \leftarrow r - 1$
    if $(ins + del \leq t$ **and** $r > MaximalMatch)$ **then** ▷ *a new maximal match found*
        output (l, r) ▷ *report the newly found match*
        $MaximalMatch \leftarrow r$

means that either x has become a deficit character in the current window or, if it was already a deficit character in the previous window then its deficiency is increased by one. Therefore, we have to do an extra insertion (of x) to transform the current window into P, hence we increment the value of ins by one (*line 12*). However, if the frequency of x in the current window is greater than or equal to its specified frequency in P (*line 13*) then it means that x was a spare character in the previous window and in the current window its spareness is decreased by one. So we save one deletion of x to transform the current window into P, hence we decrement the value of *del* by one (*line 14*).

After moving the left boundary of the window towards the right by one position, if the current window contains a potential match then we extend it towards the right (*line 15-16*) and report a maximal match if one is found in the process of extension of the window. In *line 17-18*, we make the extended window safe, if it has not remained safe in the process of extension of the window.

At *line 19* of the algorithm, the current window is always a safe window. This is because of the fact that, if the current window does not contain a potential match at *line 4*, then the current window is always a safe window (Corollary 7); otherwise, if the current window has become unsafe in the process of extension of the window, then we make it safe in *line 18* of the algorithm. We advance the current safe window towards the right by simultaneously moving l and r towards right (*line 19-20*).

In the *while loop* of *line 21-33*, we slide the safe window along the whole text in the same manner that is described above, and report a maximal match whenever we encounter one in the process of extension of a window. As the current window is always a safe window, therefore, no maximal match remains unreported.

The sub-routine $ExtendWindow$ (*Algorithm 8*), extends the right boundary of a window, that contains a potential match, to the extent that the *extended* window does not remain a potential match. Note that the length of the extended window can be at most $m + t + 1$. In the process of extension of the window, if one or more approximate matches of P are found, then the sub-routine keeps the longest of these matches.

The sub-routine $ExtendWindow$ is similar to the sub-routine $CheckForMatch$ (Algorithm 6). However, as we use a window of flexible length in the algorithm, therefore, in the sub-routine $ExtendWindow$, we do not undo the changes made in DFV during the process of extension of the window.

The sub-routine $MakeSafe$ (*Algorithm 9*), shrinks the right boundary of a non-safe window towards the left, until the window becomes safe. The minimum length, to which a window is shrunk by the sub-routine, is $m - t$ (Corollary 6 and Lemma 30).

Complexity Analysis

The worst case time complexity of this algorithm is also $O(n + Pt)$, where P is the number of potential matches of length $m - t$ in the text stream T. In the algorithm, whenever the current window corresponds to a potential match, we extend it (up to at most $O(t)$ characters); and later, when an extended window becomes non-safe, then we shrink it (up to at most $O(t)$ characters). As the shortest potential match is of length $m - t$, therefore, $O(Pt)$ is the upper bound for this extra overhead of extension/shrinkage. The $O(n)$ complexity comes from the sliding of the window through the whole text.

As we slide a *safe* window of flexible length in this algorithm, so if a current window is not a potential match but the $m-t$ characters long prefix of the current window is a potential match, then we do not evaluate this potential match.

The algorithm keeps in memory an array of σ elements and a constant number of variables, hence the space complexity of the algorithm is $O(\sigma)$, in addition to the space required for the input and the output.

4.4.6 Empirical Analysis of the Two Algorithms for Approximate Abelian Pattern Matching Under the InDel Error Model

Now we present an empirical analysis of the two algorithms for approximate abelian pattern matching under the InDel error model. We refer to the first algorithm (that uses a search window of fixed length (Section 4.4.3)) as Algorithm A; and we refer to the second algorithm (that uses a search window of flexible length (Section 4.4.5)) as Algorithm B.

For the experiments, we used the same input texts that were used for the empirical analysis of the prefix based and the suffix based algorithms (Section 2.7). We executed algorithms A & B to find approximate matches of various randomly generated abelian patterns for different values of error threshold t. For each abelian pattern, we performed 200 iterations of each of the algorithms and took the mean values of the CPU time taken by the algorithms in 200 iterations.
The experiments were performed on a computer having two “Intel® Core™2

Duo CPU E6750 @ 2.66 GHz" processors and 3.8 GiB memory, running Ubuntu 9.04.
Following are the findings of the experiments:

- Algorithm B was generally more efficient than Algorithm A. It was up to 4.16 times faster than Algorithm A.

- The relative efficiency of Algorithm B (with respect to Algorithm A) improved when the error threshold was increased.

The detailed (pattern wise) results of the experiments for empirical analysis of the two algorithms for approximate abelian pattern matching under the InDel error model are given in Appendix C.

4.5 Approximate Abelian Pattern Matching Under the Minimum Operation (MinOp) Error Model

The problem of approximate abelian pattern matching under the minimum operations error model also focuses on finding the *arbitrary* length approximate matches of P. For a given abelian pattern $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$ of length m, the MinOp distance between P and another abelian pattern $P' = \sum_{i=1}^{\sigma} m'_{c_i} c_i$ of arbitrary length, is defined as the minimum number of operations (where an operation can be either an insertion operation or a deletion operation or it can also be a substitution operation) to transform P' into P. For example, let $P = 2a + 2b + c$ and $P' = 3a + 3b$, then the MinOp distance between P and P' is two, as we substitute one spare b by a deficit c and we delete one spare a (alternatively, we could also substitute one a by c and delete a b). Note that the InDel distance between P and P' is three (Section 4.4).

Observation 19. *One substitution operation is same as one deletion and one insertion operations done together.*

As we want to minimize the number of operations to transform P' into P, therefore, in the MinOp error model, we always prefer the substitution operations over the insertion and deletion operations; and we use insertion or deletion operations only in the situations where the substitution operations are not possible.

Observation 20. *If $|P'| > m$, then we use only substitution and deletion operations to transform P' into P. We do not use any insertion operation in this case.*
Moreover, the number of deletion operations is $|P'| - m$ in this case.

Observation 21. *If $|P'| < m$, then we use only substitution and insertion operations to transform P' into P. We do not use any deletion operation in this case.*
Moreover, the number of insertion operations is $m - |P'|$ in this case.

Observation 22. *If the MinOp distance between P and P' is $x + y$, where x represent the number of substitution operations and y represents either the number of insertion operations (if $|P'| < m$) or it represents the number of deletion operations (if $|P'| > m$); then the InDel distance (Section 4.4) between P and P' is $2x + y$.*

Lemma 31. *The MinOp distance between a given abelian pattern $P = \sum_{i=1}^{\sigma} m_{c_i} c_i$ of length m, and another abelian pattern $P' = \sum_{i=1}^{\sigma} m'_{c_i} c_i$ of arbitrary length, is*

$$\frac{1}{2}\left(\sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}| + |m - |P'||\right)$$

Proof. There are only three possibilities for the length of the abelian pattern P':

Case I ($|P'| < m$): Let $x+y$ denotes the MinOp distance between P and P', where x is the number of substitution operations and y is the number of insertion operations required to transform P' into P. Note that, under the MinOp error model, we do not use any deletion operation in this case (Observation 21). Then

$$\begin{aligned}
2x + y &= \sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}| && \text{(Observation 22)} \\
\Rightarrow x + y &= \frac{1}{2}\left(\sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}| + y\right) \\
&= \frac{1}{2}\left(\sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}| + m - |P'|\right) && \text{(Observation 21)} \\
&= \frac{1}{2}\left(\sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}| + |m - |P'||\right)
\end{aligned}$$

Case II ($|P'| > m$): Let $x + y$ denotes the MinOp distance between P and P', where x is the number of substitution operations and y is the number of deletion operations required to transform P' into P. Note that, under the MinOp error model, we do not use any insertion operation in this case (Observation 20). Then

$$\begin{aligned}
2x + y &= \sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}| && \text{(Observation 22)} \\
\Rightarrow x + y &= \frac{1}{2}\left(\sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}| + y\right) \\
&= \frac{1}{2}\left(\sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}| + |P'| - m\right) && \text{(Observation 20)} \\
&= \frac{1}{2}\left(\sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}| + |m - |P'||\right)
\end{aligned}$$

Case III ($|P'| = m$): In this case, we use only substitution operations to transform P' into P, and the MinOp distance between P and P' is same as the substitution distance between P and P', which is

$$\frac{1}{2}\left(\sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}|\right) = \frac{1}{2}\left(\sum_{i=1}^{\sigma} |m_{c_i} - m'_{c_i}| + |m - |P'||\right)$$

as $m - |P'| = 0$ in this case.

□

Corollary 8. *An approximate match under the MinOp error model has least cost (the number of the operations, required to transform a substring into P), when the length of the match is m.*

For example, let $S = T_i \cdots T_j$ such that $j - i + 1 = m$ and the cost to transform S into P is C. Similarly, let $S_1 = T_i \cdots T_{j'}$ $(j' < j)$ and $S_2 = T_i \cdots T_{\bar{j}}$ $(\bar{j} > j)$ have costs C_1 and C_2 respectively. Then $C \leq C_1$ as well as $C \leq C_2$.

Observation 23. *If $S_1 = T_i \cdots T_j$ and $S_2 = T_i \cdots T_k$ are both approximate matches of P, then $S_3 = T_i \cdots T_l$ (for all l such that $j < l < k$) is also an approximate match of P.*

If *minl* is the ending position of the shortest (***min****imum* ***l****ength*) approximate abelian match of P starting at position i, and *maxl* is the ending position of the longest (***max**imum **l**ength*) approximate abelian match of P starting at position i, then the interval $[minl, maxl]$ gives us (ending positions of) all the approximate matches of P that start at i (Observation 23). Note that this interval $[minl, maxl]$ is complete with respect to i, i.e. no approximate match starting at i falls outside this interval.

The cost of the shortest match starting at i is highest (which is t) and then it decreases monotonically for the next matches as the length of the matches increases. This monotonic decrease in the cost of the approximate matches goes on until we reach the approximate match of length m; after that, the cost start increasing monotonically as the length of the matches increases. In this course, each distinct cost appears exactly twice. The first time, a cost appears against the match/length of length less than (or equal to) m; and the second time, the same cost appears against the match/matches of length greater than m (in some situations, the cost appearing against a length-m match may appear only once). This trend of fall and rise in the cost gives the concept of *cost intervals*.

Definition 19. *A* cost interval *corresponding to a position x in T and a cost $C \leq t$, is an interval of positions $[y, z]$ in T (with $x \leq y \leq z$), such that the cost to transform any substrings starting at x and ending in the interval $[y, z]$ into P is less than or equal to C. If minl and maxl are respective ending positions of the shortest and the longest approximate matches starting at i, then the interval $[minl, maxl]$ is a* cost interval *corresponding to i with an associated cost t.*

The left boundary of a cost interval is less than or equal to $i + m - 1$, whereas the right boundary of a cost interval is greater than or equal to $i+m-1$. Moreover, *lower cost* intervals are contained in *higher cost* intervals, i.e. if $[l_1, r_1]$ is a cost interval having an associated cost C_1 and $[l_2, r_2]$ is another cost interval having an associated cost C_2, and $C_1 < C_2$, then $l_2 < l_1$ and $r_2 > r_1$.

Observation 24. *The least cost interval is the innermost interval. Moreover, the position $i + m - 1$ falls in the innermost interval (Corollary 8).*

Observation 25. *The minimum difference between the associated costs of two different cost intervals is 1.*

Observation 26. *The left and right boundaries of the outermost interval are the ending positions of the shortest and the longest matches starting at i respectively.*

Lemma 32. *The number of distinct cost intervals corresponding to the position i is $t - C_{min} + 1$, where C_{min} is the cost associated with the innermost cost interval corresponding to i.*

Proof. The innermost cost interval corresponding to i has associated cost C_{min}, and the outermost cost interval corresponding to i has associated cost t. As the minimum difference between the associated costs of two different cost intervals is one (Observation 25), therefore, the number of distinct cost intervals corresponding to the position i is $t - C_{min} + 1$.

□

Corollary 9. *There can be at most $t + 1$ cost intervals corresponding to a position in T.*

4.5.1 The Desired Output

The desired output for the problem of approximate abelian pattern matching under the minimum operations error model is as follows:
We report each position i in T, such that an approximate abelian match starts at i, and we also report all the cost intervals (along with their associated costs) corresponding to i. So we output triplets comprising of:

$$Position, \qquad Interval \qquad \text{and} \qquad Associated\ Cost$$

Example 3. *We use an input text stream $T = aaaaabbbbaaacc$, and the task is to find all approximate matches of abelian pattern $P = 5a + 5b$ in T under the MinOp error model with error threshold $t = 3$.*

The output for this problem is as follows:

Position	***Interval***	***Associated Cost***
1	*[9-10]*	*1*
1	*[8-11]*	*2*
1	*[7-12]*	*3*
2	*[10-11]*	*1*
2	*[9-12]*	*2*
2	*[8-13]*	*3*
3	*[11-12]*	*1*
3	*[10-13]*	*2*
3	*[9-14]*	*3*

Figure 4.1 illustrates the cost intervals corresponding to position 2 of the input text T.

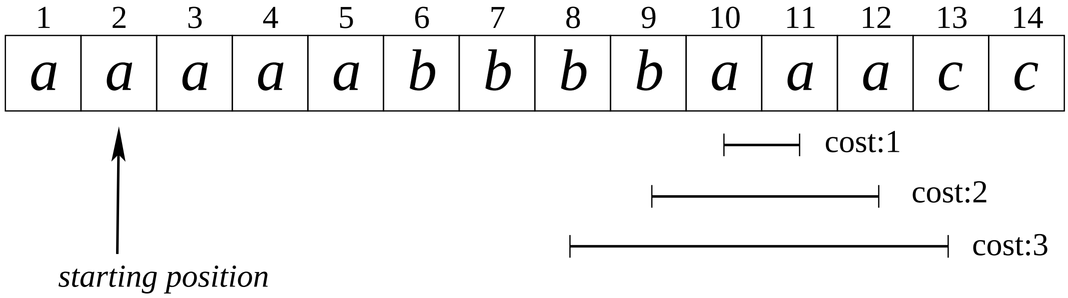

Figure 4.1: The innermost interval has cost 1 and it contains position 11, the mth position starting from position 2 (Observation 24). The outermost cost interval is defined by the ending positions of shortest (position 8) and the longest matches (position 13) starting at position 2. The number of the cost intervals, corresponding to position 2, is $3 - 1 + 1 = 3$.

4.5.2 The Algorithm

We move a window of length m along the text T, and if the window contains an approximate match, then we process the characters on both sides of the right boundary of the window, to output the cost intervals corresponding to the starting position of the window along with their associated costs. The pseudo code for this problem is presented in Algorithm 10.

In the first four lines of the algorithm, we set the difference frequency vector DFV for the first m characters of T. In the *for loop* of *line 6-7* and then in *line 8*, we compute the minimum number of operations required to transform the substring contained in the current window into P. Note that for a window of length m, we use only substitution operations for this transformation (because if we use insertions and deletions then the number of operations would not remain the minimum), so the procedure to compute the value of *ops* is same as outlined in the algorithm for approximate abelian pattern matching under the substitution error model (*Algorithm 4*). However, unlike the pattern matching under the substitution model, here we also report the approximate matches of lengths other than m. So, for each position i in T where an approximate match begins, we also report all the cost intervals corresponding to i.

In the *for loop* of *line 9-10*, we initialize an array INT to hold the cost intervals corresponding to the starting position of a window that contains an approximate match. Since there can be up to a maximum of $t + 1$ cost intervals (Corollary 9), and to record each interval we require two entries (for the left and the right boundaries of the interval), we keep the size of the

Algorithm 10 Approximate Abelian Pattern Matching Under the MinOp Error Model

Input: A pattern P of length m, a text stream $T = T[1] \dots T[n]$, a hash function ρ and an error threshold t

Output: Cost intervals corresponding to the starting positions of the approximate matches of P in T

```
▷ Build difference frequency vector (DFV) for the first m characters
for i = 1 to |Σ| do
    DFV[i] ← 0 − P[i]
for i = 1 to m do
    Increment DFV[ρ(T[i])] by 1
▷ Calculate the minimum number of operations required to transform the substring contained in the current window into P
ops ← 0
for i = 1 to |Σ| do
    ops ← ops + ABS(DFV[i])
ops ← ops/2
for i = 1 to 2 × (t + 1) do ▷ create an array of intervals
    INT[i] ← NULL
if ops ≤ t then ▷ current window contains match
    FindIntervals(T, DFV, INT, ops, t, m, ρ)
    for k = ops to t do
        output i, INT[2 × k + 1], INT[2 × k + 2], k
i ←2
while i ≤ n − m + 1 do
    if T[i − 1] ≠ T[i + m − 1] then
        Decrement DFV[ρ(T[i − 1])] by 1
        Increment DFV[ρ(T[i + m − 1])] by 1
        if (DFV[ρ(T[i − 1])] < 0) then
            if (DFV[ρ(T[i + m − 1])] > 0) then
                ops ← ops + 1
        else
            if (DFV[ρ(T[i + m − 1])] ≤ 0) then
                ops ← ops − 1
    if ops ≤ t then
        FindIntervals(T, DFV, INT, ops, t, i + m − 1, ρ)
        for k = ops to t do
            output i, INT[2 × k + 1], INT[2 × k + 2], k
    i ← i + 1
```

Algorithm 11 FindIntervals($T, DFV, INT, c, t, k, \rho$)

Input: T: the text stream (call by reference), DFV: a difference frequency vector corresponding to the current window of length m (call by reference), INT: an array to hold the intervals associated with different costs (call by reference), c: cost to covert current window into P, t: the error threshold, k: the last position of the current window, and the hash function ρ

Output: Write $t - c + 1$ intervals in the array INT

$lb \leftarrow k$
$cost \leftarrow c$
while $cost \leq t$ **do** ▷ *write left boundaries of intervals in* INT
  if $DFV[\rho(T[lb]) \leq 0$ **then** ▷ *left boundary reached*
    $INT[cost \times 2 + 1] \leftarrow lb$
    $cost \leftarrow cost + 1$▷ *find left boundary of next cost interval*
  Decrement $DFV[\rho(T[lb])]$ by 1
  $lb \leftarrow lb - 1$
for $j = lb + 1$ to k **do** ▷ *undo changes made in* DFV
  Increment $DFV[\rho(T[j])]$ by 1
$rb \leftarrow k + 1$
$cost \leftarrow c$
while $cost \leq t$ **do** ▷ *write right boundaries of intervals in* INT
  if $DFV[\rho(T[rb]) > 0$ **then** ▷ *next cost interval begins*
    $INT[cost \times 2 + 2] \leftarrow rb - 1$
    $cost \leftarrow cost + 1$
  Increment $DFV[\rho(T[rb])]$ by 1
  $rb \leftarrow rb + 1$
for $j = rb - 1$ downto $k + 1$ **do** ▷ *undo changes made in* DFV
  Decrement $DFV[\rho(T[j])]$ by 1

array $2 \times (t+1)$.

If the first m characters of T constitute an approximate match, then to generate the cost intervals corresponding to the position 1 of T, we call the sub-routine $FindIntervals$ in *line 12* of the algorithm.

The sub-routine $FindIntervals$ (*Algorithm 11*), takes as argument the input text T, DFV corresponding to the current window of length m, the array INT to record the intervals generated by the routine, the associated cost of the current window c, the error threshold t, the last position of the current window k, and the hash function ρ. It writes the cost intervals corresponding to the starting position of the current position at appropriate positions in the array INT.

The sub-routine works in two phases: in the first phase we record the left boundaries of the cost intervals (this is done in the *while loop* of *line 3-8* of the sub-routine), and in the second phase we record the right boundaries of the cost intervals (this is done in the *while loop* of *line 13-18* of the sub-routine).

In *line-1* of the sub-routine, we initially set the left boundary (lb) of innermost cost interval at the last position of the current window (which, according to Observation 24, must fall in the innermost cost interval). In *line-2*, we associate the cost of current window c to our first/innermost cost interval. And then in the *while loop* of *line 3-8*, we write left boundaries of the cost intervals – with associated costs ranging from c to t – at appropriate positions in the array INT.

The mechanism for finding the left boundary of a cost interval is to move lb towards the left by one position and see if the character T_{lb} is a deficit character. If this is the case, then it means that the cost will incur an extra insertion operation if this character is removed from the current window. Hence lb is the left boundary of the current cost interval, we write it at the appropriate position in INT and move on to the next cost interval.

After we have written the left boundaries of all the cost intervals having associated costs ranging from c to t, we undo the changes made in DFV during the left boundary search loop (the *while loop* of *line 3-8*). This is done in the *for loop* of *line 9-10*.

Now we start looking for the right boundaries of the cost intervals. We initialize rb with the value $k+1$ at *line 11*. The variable rb is aimed to hold the position *next* to the right boundary of a cost interval. As the last position of the current window k can also be the right boundary of the innermost cost interval (the last position of the current window is already included in the innermost cost interval (Observation 24)), we initialize rb with the position next to it. At *line 12*, we reset *cost* at the value of the associated cost of the innermost cost interval (which is same as c, the cost of the current window). And finally in the *while loop* of *line 13-18*, we write right boundaries of the cost intervals – with associated costs ranging from c to t – at appropriate positions in the array INT.

The idea to locate the right boundary of a cost interval is as follows: If the character at position rb is a spare character, then it means that if we include T_{rb} in the window, a deletion operation will be added to the cost of the substring contained in the extended window. Therefore, rb indicates the beginning of the next cost interval. Hence, the right boundary of the current cost interval is at one position before rb.

We write the right boundaries of the cost intervals at appropriate positions in the array INT at *line 15* of the sub-routine. After the *while loop* of *line 13-18* is terminated, we undo the changes made in DFV using the *for loop* of *line 19-20*.

When the $FindIntervals$ routine is finished, we have entries of all the cost intervals, having associated costs ranging from c to t, in the array INT, and this information is then used by the calling *main algorithm*.

In the *for loop* of *line 13-14* of the *main algorithm (Algorithm 10)*, we report the cost intervals written by the sub-routine $FindIntervals$ in the array INT, along with their associated costs. Note that this for loop makes $t - ops + 1$ iterations which is the number of the cost intervals corresponding to starting position of the current window (Lemma 32).

In the *while loop* of *line 16-30* of the algorithm, we slide the window along the whole text T. In each iteration of the loop we advance the window towards right by one position, compute the associated cost of the new window (using the same procedure that we used in the algorithm for approximate abelian pattern matching under the substitution error model (*Algorithm 4*)) and if this cost is less than or equal to t, then we call the sub-routine $FindIntervals$ and report the cost intervals found in the routine.

4.5.3 Complexity Analysis

The time complexity of the algorithm is $O(n + Mt)$, where M is the number of approximate matches of length m and $O(t)$ comes from the complexity of the sub-routine $FindIntervals$.

The algorithm keeps two arrays; DFV comprising of σ elements, and INT requiring $O(t)$ storage. Hence the space complexity of the algorithm is $O(\sigma + t)$, in addition to the space requirement for the input and the output.

4.6 Conclusion

In this chapter, we have shed light on the problem of approximate abelian pattern matching. We defined three error models to measure the degree of approximation in a matching substring of a given abelian pattern.

The substitution error model focused on the approximate abelian matches of the fixed length m, whereas the other two models allowed for the matches of lengths other than m too.

For each of the three error models, we presented algorithms for approximate abelian pattern matching under that model in the chapter.

Chapter 5

Conclusion

Pattern matching in strings is an already established research area, however, abelian pattern matching is quite a new direction of research. Study of methods and algorithms for abelian pattern matching is still in its infancy and little literature is available on this topic. In this thesis, we have studied the problem of abelian pattern matching in strings in a systematic manner, and presented several algorithms for exact as well as approximate abelian pattern matching. We have also presented different strategies for indexing the input text to make the abelian pattern matching more efficient.

5.1 Contribution of the Thesis

We have discussed several algorithms for online abelian pattern matching in Chapter 2. The algorithms were based on two different approaches, the prefix based approach and the suffix based approach. In the case of the prefix based approach, the algorithm kept track of a prefix of the substring contained in the sliding window, whereas, in the case of the suffix based approach, the algorithm had information about a suffix of the substring contained in the sliding window. We have shed light on the relative advantage of one approach over the other. Later in the chapter, we developed a variant of the suffix based algorithm to avoid the high time complexity of the original suffix based algorithm in the worst case situations. We have also given a tight lower bound for the problem of online abelian pattern matching in this chapter.

In Chapter 3, we have shed light on the indexing strategies for the input text to make the abelian pattern matching more efficient. We explained how an existing indexing scheme, the *parikh index*, could be adapted for the problem of abelian pattern matching. Later, we presented an indexing scheme, the *abelian tree*, which is customized for the problem of abelian patter matching, and showed the significant improvement in the query processing time when we use the abelian tree instead of the parikh index to index the input text.

Finally, we have addressed the problem of finding the approximate abelian matches of a given abelian pattern in Chapter 4. We have defined three error models, the substitution error model, the insertion/deletion error model and the minimum operations error model, to measure the degree of approximation in a matching substring of a given abelian pattern. The substitution error model focused on the approximate abelian matches of the fixed length m (the length of the abelian pattern to be found), whereas the other two models allowed for the matches of arbitrary lengths. For each of the three error models, we have presented algorithms for approximate abelian pattern matching under that model.

5.2 Future Research Directions

The abelian patterns lie at the middle point between the classical patterns and the regular expressions. In case of the classical patterns, both the count and the order of the characters in a pattern are preserved. On the other hand, in case of the regular expressions, neither the count nor the order of the characters are rigidly preserved (e.g. consider the regular expression $(a+b)^*$). In case of the abelian patterns, the order of the characters in a pattern in not important, however, the count of the characters in the pattern plays a pivotal role in defining the pattern.

An interesting topic for future research is to combine the abelian pattern matching with the classical pattern matching and the regular expressions based pattern matching (one such approach, without the notion of abelian patterns, already exists [16]). Note that the regular expressions based pattern matching already incorporates the classical pattern matching, as in a regular expression we can always place a fragment of another classical pattern (e.g. the resulting matches of the regular expression $a^*cdc(a+b)^+$ must contain the classical pattern cdc).

So a nice problem to work with in future is to develop the algorithms for *complex pattern matching* in strings, where a complex pattern is a combination of the abelian patterns, the classical patterns and the regular expressions. For example, $(2a+5b)(a+b)^*abba$ is a complex pattern which combines the abelian pattern $(2a+5b)$, the regular expression $(a+b)^*$ and the classical pattern $abba$. Note that we have mentioned the term $(a+b)^*$ as a regular expression and the term *abba* as a classical pattern just to put an emphasis on the presence of the classical patterns in the complex patterns; otherwise, by definition the term $(a+b)^*abba$ itself is a regular expression.

Another extension to the problem of abelian pattern matching in strings is *finding abelian patterns with fixed length gaps* (these patterns have already been studied as a discovery problem [30]). For example, $(2a+c)3?(a+2d)$ means that we want to find those locations in the input text where the abelian pattern $a+2d$ occurs three positions after the occurrence of the abelian pattern $2a+c$ (*acaccbdad* is a resulting match of this query). Note that the problem of finding abelian patterns with arbitrary length gaps can be modeled by a complex pattern, such that the gaps are represented by the regular expression $(c_1+c_2+\cdots+c_{|\Sigma|})^*$, where Σ is the alphabet of the input text and $c_i \in \Sigma\ for\ 1 \leq i \leq |\Sigma|$.

Acknowledgement

I would like to thank my PhD advisor, Prof. Dr. Sven Rahmann for his continuous guidance throughout my PhD project. I would also like to thank Prof. Jens Stoye in the Bielefeld University for his encouragement and guidance during my work. I am also thankful to Prof. Robert Giegerich, Prof. Ferdinando Cicalese, Dr. Martin Milanic, Marcel Matin and Tobias Marschall for the feed back and helpful suggestions.

And finally, I am thankful to Prof. Sebastian Böcker, my second examiner, for his willingness to evaluate my thesis.

Appendices

Appendix A

Empirical Analysis of the Prefix Based and the Suffix Based Algorithms

In this appendix, we present detailed (pattern wise) results of the experiments performed for an empirical analysis of the prefix based and the suffix based algorithms for abelian pattern matching.

The analysis is based on the actual CPU time taken by the executions of the algorithms. The experiments were performed on a computer having two "Intel® Core™2 Duo CPU E6750 @ 2.66 GHz" processors and 3.8 GiB memory, running Ubuntu 9.04.

We generated random texts comprising of 10000000 characters for $\sigma = 4$ and $\sigma = 8$. For each Σ, two different random texts were generated: one based on a uniform distribution of the characters of Σ and the other based on a non-uniform distribution of the characters of Σ.

We also used real text to compare the performance of the algorithms. The real text consisted of a collection of the plays of famous English writer William Shakespeare; these plays are available in text form on the web site `http://shakespeare.mit.edu/`. The real text comprised of 3712565 characters.

A.1 The Input Texts are Defined over $\Sigma := \{c_1, c_2, c_3, c_4\}$

The results presented in this section pertain to the execution of the algorithms on the input texts defined over four characters (i.e. $\sigma = 4$). We generated two different input texts T_{4_U} and $T_{4_{\acute{U}}}$ defined over Σ. The input text T_{4_U} is based on the uniform distribution of the characters of Σ, whereas, the input text $T_{4_{\acute{U}}}$ is based on a non-uniform distribution of the characters of Σ.

A.1.1 Comparison of the Two Algorithms for the Input Text T_{4_U}

The input text T_{4_U} is based on the uniform distribution of the characters in $\Sigma := \{c_1, c_2, c_3, c_4\}$, i.e. on every text position in T_{4_U}, the probability of occurring a character $c_i \in \Sigma$ is same for all $1 \leq i \leq 4$. The text comprises of 10000000 characters, i.e. $n := |T_{4_U}| = 10000000$.

We generated abelian patterns having different frequency distributions of the characters in Σ. The frequency distributions of the patterns ranged form the distribution similar to the distribution of the characters in T_{4_U} (e.g. the pattern $3c_1 + 3c_2 + 3c_3 + 3c_4$) to the distribution that was extremely different from the distribution of the characters in T_{4_U} (e.g. the pattern $0c_1 + 0c_2 + 0c_3 + 12c_4$).

Abelian Pattern Matching for $m = 12$ in T_{4_U}

Table A.1 shows the respective CPU times taken by the prefix based and the suffix based algorithms for finding the abelian patterns of length 12 in the input text T_{4_U}.

The suffix based algorithm performed slightly worse than the prefix based algorithm in the situations where the characters in the patterns had a frequency distribution similar to the distribution of the characters in the input text (pattern 1 & 2). However, as the frequency distribution of the characters in the patterns became different form the frequency distribution of the characters in the input text, the suffix based algorithm started performing better than the prefix based algorithm. The suffix based algorithm gave the best processing time when the pattern had a frequency distribution totally different form the distribution of the characters in the input text (pattern 10).

Abelian Pattern Matching for $m = 48$ & $m = 120$ in T_{4_U}

We extended the patterns in a manner such that the length of a pattern was increased without affecting the underlying frequency distribution of the characters in the pattern. In this way, we could analyze the effect of the pattern length on the performance of the algorithms.

Tables A.2 and A.3 show the respective CPU times taken by the prefix based and the suffix based algorithms for finding the abelian patterns of length 48 and 120 in the input text T_{4_U}.

Effect of the Pattern Length on the Relative Efficiency of the Suffix Based and the Prefix Based Algorithms

The suffix to prefix CPU time ratio for the patterns that had same underlying frequency distribution but were different in their lengths is shown in Table A.4. It is evident form the table that an increase in the length of a pattern (without disturbing the frequency distribution of the characters in the pattern) had two different behaviors:

- If the suffix to prefix CPU time ratio was greater than 1 (i.e. the prefix based algorithm was more suitable for the pattern), then by increasing the length of the pattern this ratio further increased (patterns 1 & 2) thus indicating that the prefix based approach was even more suitable for longer patterns of the same type.

- However, if the suffix to prefix CPU time ratio was less than 1 (indicating a relative advantage of the suffix based approach over the prefix based approach), then by increasing the length of the pattern this ratio further decreased indicating that the suffix based approach was even more suitable for longer patterns of the same type.

#	The Pattern	Number of Matches	CPU Time (μ sec) Prefix	CPU Time (μ sec) Suffix	Suf/Pre Ratio
1	$3c_1 + 3c_2 + 3c_3 + 3c_4$	221734	650010	751180	1.16
2	$2c_1 + 3c_2 + 3c_3 + 4c_4$	164647	613480	617110	1.01
3	$1c_1 + 2c_2 + 3c_3 + 6c_4$	33491	354030	217070	0.61
4	$0c_1 + 4c_2 + 4c_3 + 4c_4$	21023	340990	123400	0.36
5	$0c_1 + 2c_2 + 4c_3 + 6c_4$	8312	291090	92650	0.32
6	$0c_1 + 2c_2 + 2c_3 + 8c_4$	1782	246980	72910	0.3
7	$0c_1 + 0c_2 + 6c_3 + 6c_4$	565	210480	46840	0.22
8	$0c_1 + 0c_2 + 4c_3 + 8c_4$	271	224320	43660	0.19
9	$0c_1 + 0c_2 + 2c_3 + 10c_4$	45	220550	39380	0.18
10	$0c_1 + 0c_2 + 0c_3 + 12c_4$	0	189810	24970	0.13

Table A.1: CPU time taken by the prefix based and the suffix based algorithms for finding different abelian patterns of length 12 in the input text T_{4_U}. The last column of the table (Suf/Pre Ratio) represents the suffix to prefix CPU time ratio which is defined as “CPU time taken by the suffix based algorithm / CPU time taken by the prefix based algorithm”. A value greater than 1 in this column indicates that the prefix based algorithm is faster then the suffix based algorithm for finding the matches of the pattern in the input text and vice versa.

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	**Suffix**	
1	$12c_1 + 12c_2 + 12c_3 + 12c_4$	29180	340700	662980	1.95
2	$8c_1 + 12c_2 + 12c_3 + 16c_4$	8987	278030	379340	1.36
3	$4c_1 + 8c_2 + 12c_3 + 24c_4$	6	206180	87070	0.42
4	$0c_1 + 16c_2 + 16c_3 + 16c_4$	0	203180	20070	0.1
5	$0c_1 + 8c_2 + 16c_3 + 24c_4$	0	192050	20160	0.1
6	$0c_1 + 8c_2 + 8c_3 + 32c_4$	0	189590	20590	0.11
7	$0c_1 + 0c_2 + 24c_3 + 24c_4$	0	173990	16620	0.1
8	$0c_1 + 0c_2 + 16c_3 + 32c_4$	0	181610	16180	0.09
9	$0c_1 + 0c_2 + 8c_3 + 40c_4$	0	177990	16190	0.09
10	$0c_1 + 0c_2 + 0c_3 + 48c_4$	0	171140	13610	0.08

Table A.2: CPU time taken by the prefix based and the suffix based algorithms and the ratio between the CPU times taken by the two algorithms for abelian patterns of length 48.

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	$30c_1 + 30c_2 + 30c_3 + 30c_4$	7688	269550	855630	3.17
2	$20c_1 + 30c_2 + 30c_3 + 40c_4$	261	223810	313660	1.4
3	$10c_1 + 20c_2 + 30c_3 + 60c_4$	0	194020	74130	0.38
4	$0c_1 + 40c_2 + 40c_3 + 40c_4$	0	188570	12290	0.07
5	$0c_1 + 20c_2 + 40c_3 + 60c_4$	0	185990	12560	0.07
6	$0c_1 + 20c_2 + 20c_3 + 80c_4$	0	176270	12080	0.07
7	$0c_1 + 0c_2 + 60c_3 + 60c_4$	0	171720	9330	0.05
8	$0c_1 + 0c_2 + 40c_3 + 80c_4$	0	173370	9350	0.05
9	$0c_1 + 0c_2 + 20c_3 + 100c_4$	0	180770	9380	0.05
10	$0c_1 + 0c_2 + 0c_3 + 120c_4$	0	180480	6740	0.04

Table A.3: CPU time taken by the prefix based and the suffix based algorithms and the ratio between the CPU times taken by the two algorithms for abelian patterns of length 120.

#	Frequency Ratio $(c_1 : c_2 : c_3 : c_4)$	Suf/Pre Ratio		
		$m = 12$	$m = 48$	$m = 120$
1	3 : 3 : 3 : 3	1.16	1.95	3.17
2	2 : 3 : 3 : 4	1.01	1.36	1.4
3	1 : 2 : 3 : 6	0.61	0.42	0.38
4	0 : 4 : 4 : 4	0.36	0.1	0.07
5	0 : 2 : 4 : 6	0.32	0.1	0.07
6	0 : 2 : 2 : 8	0.3	0.11	0.07
7	0 : 0 : 6 : 6	0.22	0.1	0.05
8	0 : 0 : 4 : 8	0.19	0.09	0.05
9	0 : 0 : 2 : 10	0.18	0.09	0.05
10	0 : 0 : 0 : 12	0.13	0.08	0.04

Table A.4: The suffix to prefix CPU time ratio for abelian patterns of different lengths having same underlying frequency distribution of characters.

A.1.2 Comparison of the Two Algorithms for the Input Text $T_{4_{\acute{U}}}$

The input text $T_{4_{\acute{U}}}$ is based on a non-uniform distribution of the characters in $\Sigma := \{c_1, c_2, c_3, c_4\}$. The frequency ratio of the characters in $T_{4_{\acute{U}}}$ is as follows:

$$c_1 : c_2 : c_3 : c_4 \equiv 6 : 3 : 2 : 1$$

The text comprises of 10000000 characters, i.e. $n := |T_{4_{\acute{U}}}| = 10000000$.

We generated abelian patterns having different frequency distributions of the characters in Σ. The frequency distributions of the patterns ranged form the most similar one to the distribution of the characters in $T_{4_{\acute{U}}}$ (e.g. the pattern $6c_1 + 3c_2 + 2c_3 + 1c_4$) to the distribution that was extremely different from the distribution of the characters in $T_{4_{\acute{U}}}$ (e.g. the pattern $0c_1 + 0c_2 + 0c_3 + 12c_4$).

Abelian Pattern Matching for $m = 12$ in $T_{4_{\acute{U}}}$

Table A.5 shows the respective CPU times taken by the prefix based and the suffix based algorithms for finding the abelian patterns of length 12 in the input text $T_{4_{\acute{U}}}$.
The suffix based algorithm performed slightly worse than the prefix based algorithm in the situations where the characters in the patterns had a frequency distribution similar to the distribution of the characters in the input text (pattern 1,2 & 3). However, as the frequency distribution of the characters in the patterns got different form the frequency distribution of the characters in the input text, the suffix based algorithm started performing better than the prefix based algorithm. The suffix based algorithm gave the best result when the pattern had a frequency distribution totally different form the distribution of the characters in the input text (pattern 15).

Abelian Pattern Matching for $m = 48$ & $m = 120$ in $T_{4_{\acute{U}}}$

We extended the patterns in a manner such that the length of a pattern was increased without affecting the underlying frequency distribution of the characters in the pattern. In this way, we could analyze the effect of the pattern length on the performance of the algorithms.

Tables A.6 and A.7 show the respective CPU times taken by the prefix based and the suffix based algorithms for finding the abelian patterns of length 48 and 120 in the input text $T_{4_{\acute{U}}}$.

Effect of the Pattern Length on the Relative Efficiency of the Suffix Based and the Prefix Based Algorithms

The suffix to prefix CPU time ratio for the patterns that had same underlying frequency distribution but were different in their lengths is shown in Table A.8. In the table, the suffix to prefix CPU time ratio goes up with an increase in the length of the patterns for those patterns which have a ratio greater than or equal to 0.81 for $m = 12$; after this point the suffix to prefix ratio starts going down with an increase in the length of the pattern.

Following is an interesting observation from Table A.8:

- Let r be the suffix to prefix CPU time ratio of an abelian pattern P and by increasing the pattern length m of P (without changing the underlying frequency distribution of the characters in P) r increased, then for an abelian pattern $\hat{P}$ having suffix to prefix CPU time ratio $\hat{r} > r$, when the pattern length of $\hat{P}$ was increased then $\hat{r}$ also increased.

- Let r be the suffix to prefix CPU time ratio of an abelian pattern P and by increasing the pattern length m of P (without changing the underlying frequency distribution of the characters in P) r decreased, then for an abelian pattern $\hat{P}$ having suffix to prefix CPU time ratio $\hat{r} < r$, when the pattern length of $\hat{P}$ was increased then $\hat{r}$ decreased.

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	$6c_1 + 3c_2 + 2c_3 + 1c_4$	312413	773380	884280	1.14
2	$5c_1 + 3c_2 + 2c_3 + 2c_4$	158174	551870	656370	1.19
3	$4c_1 + 3c_2 + 2c_3 + 3c_4$	43562	372230	378470	1.02
4	$4c_1 + 2c_2 + 3c_3 + 3c_4$	29559	332440	269570	0.81
5	$3c_1 + 2c_2 + 3c_3 + 4c_4$	4975	259400	174690	0.67
6	$2c_1 + 2c_2 + 3c_3 + 5c_4$	514	258560	121200	0.47
7	$2c_1 + 2c_2 + 4c_3 + 4c_4$	1275	222320	125310	0.56
8	$1c_1 + 1c_2 + 4c_3 + 6c_4$	12	237140	76010	0.32
9	$0c_1 + 2c_2 + 4c_3 + 6c_4$	1	224780	40200	0.18
10	$0c_1 + 1c_2 + 3c_3 + 8c_4$	0	210610	34550	0.16
11	$0c_1 + 1c_2 + 4c_3 + 7c_4$	0	211630	37930	0.18
12	$0c_1 + 1c_2 + 5c_3 + 6c_4$	0	211850	37580	0.18
13	$0c_1 + 0c_2 + 4c_3 + 8c_4$	0	188910	24740	0.13
14	$0c_1 + 0c_2 + 6c_3 + 6c_4$	0	179450	24520	0.14
15	$0c_1 + 0c_2 + 0c_3 + 12c_4$	0	185890	18610	0.1

Table A.5: CPU time taken by the prefix based and the suffix based algorithms for finding different abelian patterns of length 12 in the input text $T_{4_{\acute{U}}}$. The last column of the table shows the ratio between the CPU times taken by the two algorithms.

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	$24c_1 + 12c_2 + 8c_3 + 4c_4$	43274	352090	727910	2.07
2	$20c_1 + 12c_2 + 8c_3 + 8c_4$	4965	267610	477940	1.79
3	$16c_1 + 12c_2 + 8c_3 + 12c_4$	26	224590	249280	1.11
4	$16c_1 + 8c_2 + 12c_3 + 12c_4$	5	202980	196510	0.97
5	$12c_1 + 8c_2 + 12c_3 + 16c_4$	0	189440	125270	0.66
6	$8c_1 + 8c_2 + 12c_3 + 20c_4$	0	189360	73060	0.39
7	$8c_1 + 8c_2 + 16c_3 + 16c_4$	0	200280	74890	0.37
8	$4c_1 + 4c_2 + 16c_3 + 24c_4$	0	179100	35560	0.2
9	$0c_1 + 8c_2 + 16c_3 + 24c_4$	0	181380	16480	0.09
10	$0c_1 + 4c_2 + 12c_3 + 32c_4$	0	182990	16640	0.09
11	$0c_1 + 4c_2 + 16c_3 + 28c_4$	0	180370	16630	0.09
12	$0c_1 + 4c_2 + 20c_3 + 24c_4$	0	179880	17020	0.09
13	$0c_1 + 0c_2 + 16c_3 + 32c_4$	0	179820	13920	0.08
14	$0c_1 + 0c_2 + 24c_3 + 24c_4$	0	180070	13980	0.08
15	$0c_1 + 0c_2 + 0c_3 + 48c_4$	0	180230	12010	0.07

Table A.6: CPU time taken by the prefix based and the suffix based algorithms and the ratio between the CPU times taken by the two algorithms for abelian patterns of length 48.

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	$60c_1 + 30c_2 + 20c_3 + 10c_4$	11367	285140	877670	3.08
2	$50c_1 + 30c_2 + 20c_3 + 20c_4$	61	211840	501950	2.37
3	$40c_1 + 30c_2 + 20c_3 + 30c_4$	0	195380	233640	1.2
4	$40c_1 + 20c_2 + 30c_3 + 30c_4$	0	175670	191180	1.09
5	$30c_1 + 20c_2 + 30c_3 + 40c_4$	0	193350	129560	0.67
6	$20c_1 + 20c_2 + 30c_3 + 50c_4$	0	188030	73940	0.39
7	$20c_1 + 20c_2 + 40c_3 + 40c_4$	0	173820	66840	0.38
8	$10c_1 + 10c_2 + 40c_3 + 60c_4$	0	184870	33740	0.18
9	$0c_1 + 20c_2 + 40c_3 + 60c_4$	0	182850	9410	0.05
10	$0c_1 + 10c_2 + 30c_3 + 80c_4$	0	175600	9390	0.05
11	$0c_1 + 10c_2 + 40c_3 + 70c_4$	0	174880	9420	0.05
12	$0c_1 + 10c_2 + 50c_3 + 60c_4$	0	177200	9480	0.05
13	$0c_1 + 0c_2 + 40c_3 + 80c_4$	0	179810	6790	0.04
14	$0c_1 + 0c_2 + 60c_3 + 60c_4$	0	179320	6780	0.04
15	$0c_1 + 0c_2 + 0c_3 + 120c_4$	0	184510	5620	0.03

Table A.7: CPU time taken by the prefix based and the suffix based algorithms and the ratio between the CPU times taken by the two algorithms for abelian patterns of length 120

#	Frequency Ratio	Suf/Pre Ratio		
	$(c_1 : c_2 : c_3 : c_4)$	$m = 12$	$m = 48$	$m = 120$
1	6 : 3 : 2 : 1	1.14	2.07	3.08
2	5 : 3 : 2 : 2	1.19	1.79	2.37
3	4 : 3 : 2 : 3	1.02	1.11	1.2
4	4 : 2 : 3 : 3	0.81	0.97	1.09
5	3 : 2 : 3 : 4	0.67	0.66	0.67
6	2 : 2 : 3 : 5	0.47	0.39	0.39
7	2 : 2 : 4 : 4	0.56	0.37	0.38
8	1 : 1 : 4 : 6	0.32	0.2	0.18
9	0 : 2 : 4 : 6	0.18	0.09	0.05
10	0 : 1 : 3 : 8	0.16	0.09	0.05
11	0 : 1 : 4 : 7	0.18	0.09	0.05
12	0 : 1 : 5 : 6	0.18	0.09	0.05
13	0 : 0 : 4 : 8	0.13	0.08	0.04
14	0 : 0 : 6 : 6	0.14	0.08	0.04
15	0 : 0 : 0 : 12	0.1	0.07	0.03

Table A.8: The suffix to prefix CPU time ratio for abelian patterns of different lengths having same underlying frequency distribution of characters.

A.2 The Input Texts are Defined over $\Sigma := \{c_1, c_2, \ldots, c_8\}$

The results presented in this section pertain to the execution of the algorithms on the input texts defined over eight characters (i.e. $\sigma = 8$). We generated two different input texts T_{8_U} and $T_{8_{\acute{U}}}$ defined over Σ. The input text T_{8_U} is based on the uniform distribution of the characters in Σ, whereas, the input text $T_{8_{\acute{U}}}$ is based on a non-uniform, distribution of the characters in Σ.

A.2.1 Comparison of the Two Algorithms for the Input Text T_{8_U}

The input text T_{8_U} is based on the uniform distribution of the characters in $\Sigma := \{c_1, c_2, \ldots, c_8\}$, i.e. on every text position in T_{8_U}, the probability of occurring a character $c_i \in \Sigma$ is same for all $1 \leq i \leq 8$. The text comprises of 10000000 characters, i.e. $n := |T_{8_U}| = 10000000$.

We generated abelian patterns having different frequency distributions of the characters in Σ. The frequency distributions of the patterns ranged form the most similar one to the distribution of T_{8_U} (e.g. the pattern $\sum_{i=1}^{8} 2c_i$) to the most different one from the distribution of T_{8_U} (e.g. the pattern $\sum_{i=1}^{7} 0c_i + 16c_8$).

Abelian Pattern Matching for $m = 16$ in T_{8_U}

Table A.9 shows the respective CPU times taken by the prefix based and the suffix based algorithms for finding the abelian patterns of length 16 in the input text T_{8_U}.

The suffix based algorithm performed better than the prefix based algorithm in all the patterns of length 16. However, the suffix to prefix CPU time ratio got even better when the frequency distribution of the characters in the patterns became different form the frequency distribution of the characters in the input text. The suffix based algorithm gave the best processing time when the pattern had a frequency distribution totally different form the distribution of the characters in the input text (pattern 15).

Abelian Pattern Matching for $m = 80$ & $m = 240$ in T_{8_U}

We extended the patterns in a manner such that the length of a pattern was increased without affecting the underlying frequency distribution of the characters in the pattern. In this way, we could analyze the effect of the pattern length on the performance of the algorithms.

Tables A.10 and A.11 show the respective CPU times taken by the prefix based and the suffix based algorithms for finding the abelian patterns of length 80 and 240 in the input text T_{8_U}.

Effect of the Pattern Length on the Relative Efficiency of the Suffix Based and the Prefix Based Algorithms

The suffix to prefix CPU time ratio for the patterns that had same underlying frequency distribution but were different in their lengths is shown in Table A.12. In the table, the suffix to prefix CPU time ratio goes up with an increase in the length of the patterns for those patterns which have a ratio greater than or equal to 0.43 for $m = 16$; after this point the suffix to prefix ratio starts going down with an increase in the length of the pattern.

Moreover, following can also be observed from Table A.12:

- Let r be the suffix to prefix CPU time ratio of an abelian pattern P and by increasing the pattern length m of P (without changing the underlying frequency distribution of the characters in P) r increased, then for an abelian pattern $\hat{P}$ having suffix to prefix CPU time ratio $\hat{r} > r$, when the pattern length of $\hat{P}$ was increased then $\hat{r}$ also increased.
- Let r be the suffix to prefix CPU time ratio of an abelian pattern P and by increasing the pattern length m of P (without changing the underlying frequency distribution of the characters in P) r decreased, then for an abelian pattern $\hat{P}$ having suffix to prefix CPU time ratio $\hat{r} < r$, when the pattern length of $\hat{P}$ was increased then $\hat{r}$ decreased.

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	$2c_1 + 2c_2 + 2c_3 + 2c_4$ $+2c_5 + 2c_6 + 2c_7 + 2c_8$	2914	348110	169740	0.49
2	$1c_1 + 2c_2 + 2c_3 + 2c_4$ $+2c_5 + 2c_6 + 2c_7 + 3c_8$	1948	340720	145100	0.43
3	$0c_1 + 1c_2 + 2c_3 + 2c_4$ $+2c_5 + 2c_6 + 3c_7 + 4c_8$	297	319630	75550	0.24
4	$0c_1 + 0c_2 + 1c_3 + 2c_4$ $+2c_5 + 3c_6 + 4c_7 + 4c_8$	56	287970	48960	0.17
5	$0c_1 + 0c_2 + 0c_3 + 1c_4$ $+3c_5 + 3c_6 + 4c_7 + 5c_8$	7	258490	38810	0.15
6	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+4c_5 + 4c_6 + 4c_7 + 4c_8$	1	233570	31780	0.14
7	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+2c_5 + 4c_6 + 5c_7 + 5c_8$	0	241970	31230	0.13
8	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+1c_5 + 3c_6 + 6c_7 + 6c_8$	0	238920	28650	0.12
9	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 5c_6 + 5c_7 + 6c_8$	0	204500	23500	0.11
10	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 4c_6 + 4c_7 + 8c_8$	0	218820	24470	0.11
11	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 8c_7 + 8c_8$	0	210580	19230	0.09
12	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 6c_7 + 10c_8$	0	206120	18900	0.09
13	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 4c_7 + 12c_8$	0	213960	18930	0.09
14	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 2c_7 + 14c_8$	0	224630	18790	0.08
15	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 0c_7 + 16c_8$	0	209800	15610	0.07

Table A.9: CPU time of the prefix based and the suffix based algorithms for abelian patterns of length 16. The last column of the table (Suf/Pre Ratio) represents the ratio between the CPU times taken by the suffix based and the prefix based algorithms for a pattern.

#	The Pattern	Number of Matches	CPU Time (μ sec) Prefix	CPU Time (μ sec) Suffix	Suf/Pre Ratio
1	$10c_1 + 10c_2 + 10c_3 + 10c_4$ $+10c_5 + 10c_6 + 10c_7 + 10c_8$	12	269420	267990	0.99
2	$5c_1 + 10c_2 + 10c_3 + 10c_4$ $+10c_5 + 10c_6 + 10c_7 + 15c_8$	3	250130	150110	0.6
3	$0c_1 + 5c_2 + 10c_3 + 10c_4$ $+10c_5 + 10c_6 + 15c_7 + 20c_8$	0	226200	22090	0.1
4	$0c_1 + 0c_2 + 5c_3 + 10c_4$ $+10c_5 + 15c_6 + 20c_7 + 20c_8$	0	197130	15920	0.08
5	$0c_1 + 0c_2 + 0c_3 + 5c_4$ $+15c_5 + 15c_6 + 20c_7 + 25c_8$	0	180710	13890	0.08
6	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+20c_5 + 20c_6 + 20c_7 + 20c_8$	0	179190	12630	0.07
7	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+10c_5 + 20c_6 + 25c_7 + 25c_8$	0	192060	12610	0.07
8	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+5c_5 + 15c_6 + 30c_7 + 30c_8$	0	183830	12440	0.07
9	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 25c_6 + 25c_7 + 30c_8$	0	178860	10690	0.06
10	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 20c_6 + 20c_7 + 40c_8$	0	179460	10780	0.06
11	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 40c_7 + 40c_8$	0	173630	8980	0.05
12	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 30c_7 + 50c_8$	0	172080	9040	0.05
13	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 20c_7 + 60c_8$	0	171940	9010	0.05
14	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 10c_7 + 70c_8$	0	180490	9000	0.05
15	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 0c_7 + 80c_8$	0	170850	7860	0.05

Table A.10: CPU time of the prefix based and the suffix based algorithms and the ratio between the CPU times taken by the algorithms for abelian patterns of length 80.

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	$30c_1 + 30c_2 + 30c_3 + 30c_4$ $+30c_5 + 30c_6 + 30c_7 + 30c_8$	1	227400	475400	2.09
2	$15c_1 + 30c_2 + 30c_3 + 30c_4$ $+30c_5 + 30c_6 + 30c_7 + 45c_8$	0	207170	137860	0.67
3	$0c_1 + 15c_2 + 30c_3 + 30c_4$ $+30c_5 + 30c_6 + 45c_7 + 60c_8$	0	197870	8740	0.04
4	$0c_1 + 0c_2 + 15c_3 + 30c_4$ $+30c_5 + 45c_6 + 60c_7 + 60c_8$	0	185940	6600	0.04
5	$0c_1 + 0c_2 + 0c_3 + 15c_4$ $+45c_5 + 45c_6 + 60c_7 + 75c_8$	0	172040	5760	0.03
6	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+60c_5 + 60c_6 + 60c_7 + 60c_8$	0	172040	5120	0.03
7	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+30c_5 + 60c_6 + 75c_7 + 75c_8$	0	181460	5180	0.03
8	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+15c_5 + 45c_6 + 90c_7 + 90c_8$	0	179710	5150	0.03
9	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 75c_6 + 75c_7 + 90c_8$	0	179290	4500	0.03
10	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 60c_6 + 60c_7 + 120c_8$	0	173240	4350	0.03
11	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 120c_7 + 120c_8$	0	169900	3730	0.02
12	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 90c_7 + 150c_8$	0	169550	3760	0.02
13	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 60c_7 + 180c_8$	0	169940	3870	0.02
14	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 30c_7 + 210c_8$	0	178240	3830	0.02
15	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 0c_7 + 240c_8$	0	172970	3260	0.02

Table A.11: CPU time of the prefix based and the suffix based algorithms and the ratio between the CPU times taken by the algorithms for abelian patterns of length 240.

#	Frequency Ratio ($c_1 : c_2 : c_3 : c_4 : c_5 : c_6 : c_7 : c_8$)	Suf/Pre Ratio		
		$m = 16$	$m = 80$	$m = 240$
1	2:2:2:2:2:2:2:2	0.49	0.99	2.09
2	1:2:2:2:2:2:2:3	0.43	0.6	0.67
3	0:1:2:2:2:2:3:4	0.24	0.1	0.04
4	0:0:1:2:2:3:4:4	0.17	0.08	0.04
5	0:0:0:1:3:3:4:5	0.15	0.08	0.03
6	0:0:0:0:4:4:4:4	0.14	0.07	0.03
7	0:0:0:0:2:4:5:5	0.13	0.07	0.03
8	0:0:0:0:1:3:6:6	0.12	0.07	0.03
9	0:0:0:0:0:5:5:6	0.11	0.06	0.03
10	0:0:0:0:0:4:4:8	0.11	0.06	0.03
11	0:0:0:0:0:0:8:8	0.09	0.05	0.02
12	0:0:0:0:0:0:6:10	0.09	0.05	0.02
13	0:0:0:0:0:0:4:12	0.09	0.05	0.02
14	0:0:0:0:0:0:2:14	0.08	0.05	0.02
15	0:0:0:0:0:0:0:16	0.07	0.05	0.02

Table A.12: The suffix to prefix CPU time ratio for abelian patterns of different lengths having same underlying frequency distribution of characters.

A.2.2 Comparison of the Two Algorithms on the Input Text $T_{8_{\acute{U}}}$

The input text $T_{8_{\acute{U}}}$ is based on a non-uniform distribution of the characters in $\Sigma := \{c_1, c_2, \ldots, c_8\}$. The frequency ratio of the characters in $T_{8_{\acute{U}}}$ is as follows:

$$c_1 : c_2 : c_3 : c_4 : c_5 : c_6 : c_7 : c_8 \equiv 15 : 10 : 8 : 6 : 4 : 3 : 2 : 1$$

The text comprises of 10000000 characters, i.e. $n := |T_{8_{\acute{U}}}| = 10000000$.

We generated abelian patterns having different frequency distributions of the characters in Σ. The frequency distributions of the characters in the patterns ranged form the most similar one to the distribution of $T_{8_{\acute{U}}}$ (e.g. the pattern $15c_1 + 10c_2 + 8c_3 + 6c_4 + 4c_5 + 3c_6 + 2c_7 + 1c_8$) to the most different one from the distribution of $T_{8_{\acute{U}}}$ (e.g. the pattern $\sum_{i=1}^{7} 0c_i + 49c_8$).

Abelian Pattern Matching for $m = 49$

Table A.13 shows the respective CPU times taken by the prefix based and the suffix based algorithms for finding the abelian patterns of length 49 in the input text $T_{8_{\acute{U}}}$.

Here too, the suffix based algorithm performed better than the prefix based algorithm in all the patterns of length 49. The suffix to prefix CPU time ratio got much better as the frequency distribution of the characters in the patterns became different form the frequency distribution of the characters in the input text. The suffix based algorithm gave the best processing time when the pattern had a frequency distribution totally different form the distribution of the characters in the input text (pattern 15).

Abelian Pattern Matching for $m = 98$ & $m = 196$

We extended the patterns in a manner such that the length of a patterns was increased without affecting the underlying frequency distribution of the characters in the pattern. In this way, we could analyze the effect of the pattern length on the performance of the algorithms.

Tables A.14 and A.15 show the respective CPU times taken by the prefix based and the suffix based algorithms for finding the abelian patterns of length 98 and 196 in the input text $T_{8_{\acute{U}}}$.

Effect of the Pattern Length on the Relative Efficiency of the Suffix Based and the Prefix Based Algorithms

The suffix to prefix CPU time ratio for the patterns that had same underlying frequency distribution but were different in their lengths is shown in Table A.16. In the table, the suffix to prefix CPU time ratio goes up with an increase in the length of the patterns for those patterns which have a ratio less than or equal to 0.36 for $m = 49$; after this point the suffix to prefix ratio starts going down with an increase in the length of the pattern.

Following was observed from Table A.16 also:

- Let r be the suffix to prefix CPU time ratio of an abelian pattern P and by increasing the pattern length m of P (without changing the underlying frequency distribution of the characters in P) r increased, then for an abelian pattern $\hat{P}$ having suffix to prefix CPU time ratio $\hat{r} > r$, when the pattern length of $\hat{P}$ was increased then $\hat{r}$ also increased.

- Let r be the suffix to prefix CPU time ratio of an abelian pattern P and by increasing the pattern length m of P (without changing the underlying frequency distribution of the characters in P) r decreased, then for an abelian pattern $\hat{P}$ having suffix to prefix CPU time ratio $\hat{r} < r$, when the pattern length of $\hat{P}$ was increased then $\hat{r}$ decreased.

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	$15c_1 + 10c_2 + 8c_3 + 6c_4$ $+4c_5 + 3c_6 + 2c_7 + 1c_8$	209	287460	207050	0.72
2	$10c_1 + 9c_2 + 7c_3 + 6c_4$ $+5c_5 + 5c_6 + 3c_7 + 4c_8$	1	245780	190120	0.77
3	$8c_1 + 7c_2 + 7c_3 + 7c_4$ $+4c_5 + 5c_6 + 5c_7 + 6c_8$	0	224830	123260	0.55
4	$5c_1 + 6c_2 + 6c_3 + 6c_4$ $+6c_5 + 6c_6 + 7c_7 + 7c_8$	0	222540	79680	0.36
5	$0c_1 + 7c_2 + 7c_3 + 7c_4$ $+7c_5 + 7c_6 + 7c_7 + 7c_8$	0	221710	19000	0.09
6	$0c_1 + 5c_2 + 6c_3 + 7c_4$ $+7c_5 + 8c_6 + 8c_7 + 8c_8$	0	210020	18940	0.09
7	$0c_1 + 2c_2 + 6c_3 + 7c_4$ $+7c_5 + 8c_6 + 9c_7 + 10c_8$	0	199820	18380	0.09
8	$0c_1 + 0c_2 + 7c_3 + 8c_4$ $+8c_5 + 8c_6 + 8c_7 + 10c_8$	0	198360	16180	0.08
9	$0c_1 + 0c_2 + 3c_3 + 8c_4$ $+9c_5 + 9c_6 + 10c_7 + 10c_8$	0	191290	16350	0.09
10	$0c_1 + 0c_2 + 0c_3 + 9c_4$ $+10c_5 + 10c_6 + 10c_7 + 10c_8$	0	187890	14430	0.08
11	$0c_1 + 0c_2 + 0c_3 + 5c_4$ $+10c_5 + 11c_6 + 11c_7 + 12c_8$	0	189860	14170	0.07
12	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+6c_5 + 12c_6 + 14c_7 + 17c_8$	0	183460	12750	0.07
13	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 16c_6 + 16c_7 + 17c_8$	0	182630	12230	0.07
14	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 24c_7 + 25c_8$	0	184920	11640	0.06
15	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 0c_7 + 49c_8$	0	187720	11420	0.06

Table A.13: CPU time of the prefix based and the suffix based algorithms for abelian patterns of length 49. The last column of the table (Suf/Pre Ratio) represents the ratio between the CPU times taken by the suffix based and the prefix based algorithms for a pattern.

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	$30c_1 + 20c_2 + 16c_3 + 12c_4$ $+8c_5 + 6c_6 + 4c_7 + 2c_8$	15	257100	268800	1.05
2	$20c_1 + 18c_2 + 14c_3 + 12c_4$ $+10c_5 + 10c_6 + 6c_7 + 8c_8$	0	220450	211330	0.96
3	$16c_1 + 14c_2 + 14c_3 + 14c_4$ $+8c_5 + 10c_6 + 10c_7 + 12c_8$	0	208130	130690	0.63
4	$10c_1 + 12c_2 + 12c_3 + 12c_4$ $+12c_5 + 12c_6 + 14c_7 + 14c_8$	0	193220	71450	0.37
5	$0c_1 + 14c_2 + 14c_3 + 14c_4$ $+14c_5 + 14c_6 + 14c_7 + 14c_8$	0	202240	12880	0.06
6	$0c_1 + 10c_2 + 12c_3 + 14c_4$ $+14c_5 + 16c_6 + 16c_7 + 16c_8$	0	188760	13000	0.07
7	$0c_1 + 4c_2 + 12c_3 + 14c_4$ $+14c_5 + 16c_6 + 18c_7 + 20c_8$	0	186420	12850	0.07
8	$0c_1 + 0c_2 + 14c_3 + 16c_4$ $+16c_5 + 16c_6 + 16c_7 + 20c_8$	0	195160	10740	0.06
9	$0c_1 + 0c_2 + 6c_3 + 16c_4$ $+18c_5 + 18c_6 + 20c_7 + 20c_8$	0	185060	10680	0.06
10	$0c_1 + 0c_2 + 0c_3 + 18c_4$ $+20c_5 + 20c_6 + 20c_7 + 20c_8$	0	185770	9050	0.05
11	$0c_1 + 0c_2 + 0c_3 + 10c_4$ $+20c_5 + 22c_6 + 22c_7 + 24c_8$	0	194240	9040	0.05
12	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+12c_5 + 24c_6 + 28c_7 + 34c_8$	0	189150	7780	0.04
13	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 32c_6 + 32c_7 + 34c_8$	0	175640	6820	0.04
14	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 48c_7 + 50c_8$	0	176490	6500	0.04
15	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 0c_7 + 98c_8$	0	177240	6420	0.04

Table A.14: CPU time of the prefix based and the suffix based algorithms and the ratio between the CPU times taken by the algorithms for abelian patterns of length 98.

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	$60c_1 + 40c_2 + 32c_3 + 24c_4$ $+16c_5 + 12c_6 + 8c_7 + 4c_8$	1	235350	364370	1.55
2	$40c_1 + 36c_2 + 28c_3 + 24c_4$ $+20c_5 + 20c_6 + 12c_7 + 16c_8$	0	201840	238900	1.18
3	$32c_1 + 28c_2 + 28c_3 + 28c_4$ $+16c_5 + 20c_6 + 20c_7 + 24c_8$	0	199100	142120	0.71
4	$20c_1 + 24c_2 + 24c_3 + 24c_4$ $+24c_5 + 24c_6 + 28c_7 + 28c_8$	0	193250	70400	0.36
5	$0c_1 + 28c_2 + 28c_3 + 28c_4$ $+28c_5 + 28c_6 + 28c_7 + 28c_8$	0	219650	7000	0.03
6	$0c_1 + 20c_2 + 24c_3 + 28c_4$ $+28c_5 + 32c_6 + 32c_7 + 32c_8$	0	216350	8140	0.04
7	$0c_1 + 8c_2 + 24c_3 + 28c_4$ $+28c_5 + 32c_6 + 36c_7 + 40c_8$	0	214530	8090	0.04
8	$0c_1 + 0c_2 + 28c_3 + 32c_4$ $+32c_5 + 32c_6 + 32c_7 + 40c_8$	0	216070	6450	0.03
9	$0c_1 + 0c_2 + 12c_3 + 32c_4$ $+36c_5 + 36c_6 + 40c_7 + 40c_8$	0	214230	6480	0.03
10	$0c_1 + 0c_2 + 0c_3 + 36c_4$ $+40c_5 + 40c_6 + 40c_7 + 40c_8$	0	209000	5220	0.02
11	$0c_1 + 0c_2 + 0c_3 + 20c_4$ $+40c_5 + 44c_6 + 44c_7 + 48c_8$	0	216700	5130	0.02
12	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+24c_5 + 48c_6 + 56c_7 + 68c_8$	0	213990	4360	0.02
13	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 64c_6 + 64c_7 + 68c_8$	0	200830	3900	0.02
14	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 96c_7 + 100c_8$	0	213060	3610	0.02
15	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 0c_7 + 196c_8$	0	208130	3480	0.02

Table A.15: CPU time of the prefix based and the suffix based algorithms and the ratio between the CPU times taken by the algorithms for abelian patterns of length 196.

#	Frequency Ratio $(c_1 : c_2 : c_3 : c_4 : c_5 : c_6 : c_7 : c_8)$	Suf/Pre Ratio $m = 49$	$m = 98$	$m = 196$
1	15:10:8:6:4:3:2:1	0.72	1.05	1.55
2	10:9:7:6:5:5:3:4	0.77	0.96	1.18
3	8:7:7:7:4:5:5:6	0.55	0.63	0.71
4	5:6:6:6:6:6:7:7	0.36	0.37	0.36
5	0:7:7:7:7:7:7:7	0.09	0.06	0.03
6	0:5:6:7:7:8:8:8	0.09	0.07	0.04
7	0:2:6:7:7:8:9:10	0.09	0.07	0.04
8	0:0:7:8:8:8:8:10	0.08	0.06	0.03
9	0:0:3:8:9:9:10:10	0.09	0.06	0.03
10	0:0:0:9:10:10:10:10	0.08	0.05	0.02
11	0:0:0:5:10:11:11:12	0.07	0.05	0.02
12	0:0:0:0:6:12:14:17	0.07	0.04	0.02
13	0:0:0:0:0:16:16:17	0.07	0.04	0.02
14	0:0:0:0:0:0:24:25	0.06	0.04	0.02
15	0:0:0:0:0:0:0:49	0.06	0.04	0.02

Table A.16: The Suffix/Prefix ratio of the CPU time for different pattern lengths having same underlying frequency distribution of characters.

A.3 The Input Text T_{Real} is a Collection of the Work of Shakespeare

The input text T_{Real} is defined over English alphabet. Moreover, the text is not randomly generated, rather it is a real text consisting of a collection of the plays of famous English writer William Shakespeare; these plays are available in text form on the web site `http://shakespeare.mit.edu/`. The text was pre-processed and all the punctuation marks and white spaces were removed form the text. We also changed the upper case letters into lower case, thus making $\sigma = 26$. The post-processed text comprised of 3712565 characters, i.e. $n := |T_{Real}| = 3712565$.

A.3.1 Comparison of the Two Algorithms on the Input Text T_{Real} for Randomly Selected Substrings of T_{Real}

We randomly selected substrings of various lengths from T_{Real} and converted these substrings into equivalent abelian patterns. The chosen lengths for the substrings were $5, 10, 20$ and 50.

Abelian Pattern Matching for $m = 5$ in T_{Real}

Table A.17 shows the respective CPU times taken by the two algorithms for finding the abelian patterns of length 5 in the input text T_{Real}.

The suffix based algorithm performed better than the prefix based algorithm for all the patterns of length 5 which were randomly selected for the experiments. The suffix to prefix CPU time ratio ranged between $0.21 - 0.31$.

Abelian Pattern Matching for $m = 10$ in T_{Real}

Table A.18 shows the respective CPU times taken by the two algorithms for finding the abelian patterns of length 10 in the input text T_{Real}.

The suffix based algorithm performs better than the prefix based algorithm for all the patterns of length 10 which were randomly chosen for the experiments. The suffix to prefix CPU time ratio ranged between $0.12 - 0.17$.

Abelian Pattern Matching for $m = 20$ in T_{Real}

Table A.19 shows the respective CPU times taken by the two algorithms for finding the abelian patterns of length 20 in the input text T_{Real}.

The suffix based algorithm performs better than the prefix based algorithm for all the patterns of length 20 which were randomly chosen for the experiments. The suffix to prefix CPU time ratio ranged between $0.07 - 0.14$.

Abelian Pattern Matching for $m = 50$ in T_{Real}

Table A.20 shows the respective CPU times taken by the two algorithms for finding the abelian patterns of length 50 in the input text T_{Real}.

The suffix based algorithm performs better than the prefix based algorithm for all the patterns of length 50 which were randomly chosen for the experiments. The suffix to prefix CPU time ratio ranged between $0.07 - 0.13$.

Effect of the Pattern Length on the Relative Efficiency of the Suffix Based and the Prefix Based Algorithms

Although, in this case, the abelian patterns of different lengths did not have same frequency distribution, yet the suffix to prefix CPU time ratio decreased with an increase in the pattern length. The suffix to prefix CPU time ratio ranged between $0.21 - 0.31$ for the pattern of length 5; and this ratio ranged between $0.07 - 0.13$ for the patterns of length 50.

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	$2e + 1h + 1r + 1w$	3630	133740	35680	0.27
2	$1e + 1h + 1m + 1o + 1t$	3525	128430	39850	0.31
3	$1d + 1l + 1o + 1r + 1y$	2094	119550	32770	0.27
4	$1e + 1f + 1o + 2r$	1991	125880	35420	0.28
5	$1e + 1m + 1o + 1r + 1s$	1697	125650	37460	0.3
6	$1g + 2h + 1o + 1u$	1451	119470	29460	0.25
7	$1c + 1i + 1n + 1p + 1r$	1136	114640	29140	0.25
8	$1a + 1d + 1e + 1m + 1r$	1042	114340	33500	0.29
9	$1a + 1d + 1e + 1s + 1t$	908	119320	34720	0.29
10	$1e + 1f + 1l + 1s + 1y$	869	112900	30830	0.27
11	$1a + 1e + 1i + 1m + 1r$	817	117690	32790	0.28
12	$2e + 1m + 1n + 1t$	760	126460	31580	0.25
13	$1e + 1f + 1g + 1o + 1r$	678	118130	30650	0.26
14	$1d + 1e + 1l + 1n + 1o$	597	110970	31290	0.28
15	$2a + 2d + 1n$	582	113150	27730	0.25
16	$1c + 1e + 1i + 1l + 1r$	510	115270	30200	0.26
17	$1a + 1h + 1n + 1o + 1w$	506	117950	30880	0.26
18	$2e + 1m + 1n + 1s$	460	119950	28950	0.24
19	$1a + 1d + 1i + 1s + 1y$	448	118610	29020	0.24
20	$1c + 1i + 1l + 1o + 1u$	436	115190	28750	0.25
21	$1a + 1d + 1o + 1s + 1t$	430	112080	29540	0.26
22	$1a + 1e + 1l + 1t + 1w$	338	108540	29180	0.27
23	$1o + 1r + 2t + 1u$	331	111670	27450	0.25
24	$1o + 2p + 1r + 1u$	327	105030	24620	0.23
25	$1e + 1h + 1i + 1k + 1s$	322	110120	29390	0.27

#	The Pattern	Number of Matches	CPU Time (μ sec) Prefix	CPU Time (μ sec) Suffix	Suf/Pre Ratio
26	$1e + 1i + 2n + 1s$	253	113350	28390	0.25
27	$1a + 1e + 1l + 1m + 1y$	252	117560	28930	0.25
28	$1h + 1i + 2s + 1w$	248	120480	28030	0.23
29	$1e + 1i + 1s + 1v + 1w$	187	104780	26200	0.25
30	$1b + 2e + 1f + 1r$	125	107170	24350	0.23
31	$1h + 1i + 1s + 1t + 1v$	117	117340	29470	0.25
32	$1a + 1d + 1s + 1t + 1w$	75	118080	27610	0.23
33	$1a + 1b + 1l + 1m + 1o$	56	112660	25120	0.22
34	$1l + 1m + 1o + 1p + 1u$	22	112150	24490	0.22
35	$2c + 1k + 1n + 1o$	21	108590	23780	0.22
36	$1a + 1r + 1s + 2y$	20	116570	25110	0.22
37	$2a + 1c + 1m + 1p$	15	111890	23440	0.21

Table A.17: CPU time of the prefix based and the suffix based algorithms for abelian patterns of length 5. The last column of the table (Suf/Pre Ratio) represents the ratio between the CPU times taken by the suffix based and the prefix based algorithms for a pattern.

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	$1c + 3e + 1g + 1l + 1o + 1s + 1t + 1u$	116	132500	17790	0.13
2	$2e + 1h + 1i + 1o + 1r + 1s + 2t + 1w$	71	137400	22770	0.17
3	$1a+1e+2h+2l+2o+1t+1w$	25	132650	18470	0.14
4	$2d + 3e + 1h + 2i + 1n + 1s$	23	130110	16540	0.13
5	$2a + 1d + 1e + 1h + 1o + 1r + 1t + 1u + 1y$	21	133850	21660	0.16
6	$1a + 2e + 1h + 1n + 1o + 1r + 1t + 1v + 1y$	21	131210	21220	0.16
7	$2e+1h+1l+1o+1r+2s+2t$	19	133260	19020	0.14
8	$1e + 1h + 1i + 1l + 1o + 1s + 2t + 1u + 1y$	19	132240	20280	0.15
9	$1a+1f+1i+1m+1n+2o+1r + 2t$	18	136720	17700	0.13
10	$2e + 1h + 1i + 1l + 1o + 1r + 2t + 1v$	18	134080	19590	0.15
11	$3e+1g+1h+1i+1m+1o+1t + 1v$	16	131680	16620	0.13
12	$2a+1b+2e+1h+1l+2r+1t$	9	131110	17970	0.14
13	$3e+1h+1i+1k+1l+1m+1s + 1t$	9	131030	18320	0.14
14	$1a + 1e + 1i + 1l + 1n + 1o + 1r + 1s + 1u + 1v$	9	133790	21670	0.16
15	$4e+1h+1n+1o+1r+1t+1v$	8	132510	18600	0.14
16	$2e + 1h + 1i + 1r + 2t + 1u + 1v + 1y$	7	133400	18000	0.13
17	$1e+1f+2n+3o+1s+1u+1y$	6	126930	15300	0.12
18	$1d+1e+1n+3o+2s+1t+1u$	6	129710	17950	0.14
19	$1b + 1d + 2e + 1l + 1m + 1o + 1r + 1u + 1v$	4	129850	17100	0.13

#	The Pattern	Number of Matches	CPU Time (μ sec) Prefix	CPU Time (μ sec) Suffix	Suf/Pre Ratio
20	$3e+1h+1i+2o+1p+1s+1t$	4	129270	18160	0.14
21	$1h+1i+1n+1o+2r+1s+2t+1u$	3	130010	19860	0.15
22	$1c+1d+2e+1h+1i+1l+1p+1r+1s$	3	134800	18850	0.14
23	$1h+2i+1n+1o+2s+1u+1w+1y$	2	133010	17240	0.13
24	$1a+1b+1e+1m+1n+1o+1r+1t+1u+1w$	2	131030	21010	0.16
25	$1a+2d+2e+1m+1s+2t+1y$	2	128820	17160	0.13
26	$1a+1d+1e+1g+1h+1l+1n+1o+2r$	2	130240	19670	0.15
27	$1d+1h+2i+1p+2r+1s+2t$	1	126390	15950	0.13
28	$1a+1h+3i+1o+2s+2t$	1	125640	17820	0.14
29	$2a+1c+1e+2m+1n+1r+2u$	1	125830	15690	0.12
30	$1a+3e+1f+1h+1q+1r+1t+1u$	1	132890	18270	0.14
31	$1a+1c+2d+1e+1h+1o+2t+1y$	1	132830	19030	0.14
32	$1b+2e+1f+1i+2o+2r+1s$	1	133060	16610	0.12
33	$1b+1e+2o+1p+1r+2t+1u+1y$	1	127680	17350	0.14
34	$1b+2d+1e+2h+1i+1l+1o+1t$	1	128170	18040	0.14
35	$1a+2d+2h+1m+1n+1r+1t+1u$	1	129780	17140	0.13
36	$2e+2i+2l+1p+2t+1y$	1	122900	14360	0.12
37	$1e+2g+1h+1i+1l+1m+1o+1t+1y$	1	125920	18770	0.15

Table A.18: CPU time of the prefix based and the suffix based algorithms for abelian patterns of length 10.

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	$5a+1c+1d+1i+1k+1m+3n+2o+1r+2t+1v+1y$	13	124640	11300	0.09
2	$2a+1d+1e+3h+2i+1m+1n+1o+2s+5t+1y$	2	128500	15160	0.12
3	$2a+1b+1c+1d+3e+3h+1i+1o+1r+4t+2u$	2	128790	12660	0.1
4	$2c+2e+1f+2h+3i+1l+1m+1n+3o+3t+1w$	2	125530	11940	0.1
5	$3a+2b+2d+3e+2i+1l+2m+2n+2s+1t$	2	121480	11680	0.1
6	$2a+1c+2e+1f+1g+1h+3i+2k+2n+2r+1t+1v+1y$	1	125550	12720	0.1
7	$1a+1b+2d+3e+1g+1h+2n+3o+1p+3t+1u+1w$	1	125050	12370	0.1
8	$2a+2c+2i+2l+3o+5s+3u+1w$	1	113250	8470	0.07
9	$4a+1b+1c+2e+1f+1h+3l+1n+1o+1p+1t+1u+1w+1y$	1	127210	12810	0.1
10	$2a+1c+1d+3e+1g+3h+4l+1o+1t+1u+1v+1w$	1	125690	11750	0.09
11	$1b+1c+2h+1i+1l+2n+3o+1p+2s+3u+3w$	1	120450	9450	0.08
12	$5a+2c+1d+1g+1h+1j+2k+1n+1o+1r+2s+1u+1w$	1	122450	11000	0.09

#	The Pattern	Number of Matches	CPU Time (μ sec) Prefix	Suffix	Suf/Pre Ratio
13	$4a+1b+3d+2e+1h+2i+1m+1n+2r+1s+1t+1y$	1	128960	14650	0.11
14	$6e+1i+1l+1m+2n+1r+4s+3t+1y$	1	117490	10400	0.09
15	$1a+1b+5e+1g+1h+3i+1l+1p+2r+1s+3t$	1	119780	12280	0.1
16	$2a+2e+1f+2n+5o+1r+5t+1u+1y$	1	118850	10940	0.09
17	$3a+1d+2e+2h+1i+1k+1n+2o+1s+3t+1u+1v+1y$	1	129250	14920	0.12
18	$2a+1b+1d+2e+2i+3l+1n+1o+1p+1r+3t+1u+1w$	1	132640	14440	0.11
19	$2a+1b+1c+3e+1f+1h+1i+1k+2l+1m+3s+2t+1x$	1	128950	12440	0.1
20	$5e+1f+1h+1i+1n+1o+1r+4s+3t+1u+1w$	1	125240	14740	0.12
21	$1d+2e+2f+1g+1h+1i+4o+2r+1s+2t+3u$	1	128430	12900	0.1
22	$2a+1b+3e+1g+1h+2i+3l+1m+1n+1o+1r+2s+1z$	1	133390	13790	0.1
23	$4a+1d+1e+1g+1l+2n+2o+2r+3s+2u+1w$	1	122110	11660	0.1
24	$3a+3c+3e+1h+1i+4n+1o+1r+3t$	1	121270	12420	0.1

#	The Pattern	Number of Matches	CPU Time (μ sec) Prefix	CPU Time (μ sec) Suffix	Suf/Pre Ratio
25	$1b+1d+5e+1g+1h+1i+2m+1n+2r+1s+3t+1w$	1	121880	12640	0.1
26	$4a+2d+3e+1h+2n+2s+2t+1v+1w+2y$	1	120480	11480	0.1
27	$2a+1b+1c+1d+2e+1f+1m+3o+2r+3s+1t+1x+1y$	1	127600	12150	0.1
28	$1c+3e+1f+1h+2i+1k+1l+3n+1r+2s+1t+2u+1y$	1	129360	13110	0.1
29	$3a+1d+1e+1f+1h+1i+2m+1n+1p+2r+1s+2t+1u+2y$	1	131090	15260	0.12
30	$2a+1b+1d+1e+1f+1g+1i+2n+2o+2r+2s+1t+2u+1w$	1	127520	16050	0.13
31	$2a+1b+1c+1d+3e+2f+1h+1l+1n+3o+3t+1u$	1	126240	12600	0.1
32	$1a+1b+1c+1d+1e+1g+1h+3i+3n+1o+1r+2s+1t+2u$	1	126270	17330	0.14
33	$1a+1b+3e+2h+1i+1n+1o+4r+2s+2u+1w+1y$	1	124120	12750	0.1
34	$1c+2e+2h+1i+1l+2m+2o+1p+1r+3s+1t+1u+1x+1y$	1	125960	14370	0.11
35	$3a+1e+1f+2h+1i+2l+2o+1p+1r+2u+1v+3y$	1	119660	10850	0.09
36	$1d+2e+1f+2i+1j+1l+1n+2o+3r+2s+1t+1u+2y$	1	127800	14480	0.11
37	$2a+2d+2e+1f+1h+1i+3l+2n+2r+1s+1t+1u+1w$	1	128080	15530	0.12

Table A.19: CPU time of the prefix based and the suffix based algorithms for abelian patterns of length 20.

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	$4a+3d+3e+1f+2h+3i+3l+2m+6n+6o+1s+6t+2u+2v+4w+2y$	2	104960	9130	0.09
2	$5a+3b+1c+5d+4e+1f+2h+2i+2l+2m+3n+6o+1p+4r+3s+3t+2w+1y$	2	112940	13810	0.12
3	$3a+1b+2c+7e+1f+3h+2i+1k+2l+1m+7n+7o+1p+4r+3s+2t+1u+2v$	2	110460	11040	0.1
4	$6a+1b+5d+4e+1f+1g+2h+4i+2k+1l+1m+3n+5o+3r+1s+7t+1v+2w$	1	109830	11040	0.1
5	$6a+1b+4d+4e+2g+1h+5i+4l+4m+6n+3o+3r+2s+4t+1y$	1	105200	9320	0.09
6	$6a+2b+1d+9e+1f+4h+1i+1k+3l+3n+3o+3r+4s+6t+1u+1w+1y$	1	110160	11260	0.1
7	$3a+1b+1c+3d+6e+1f+3g+3i+4l+2n+6o+3r+4s+8t+1u+1w$	1	109900	9090	0.08
8	$4a+1b+3c+2d+6e+2h+3i+2l+1m+1n+4o+3p+7r+3s+2t+2u+1w+3y$	1	110000	13220	0.12
9	$7a+1b+1c+3e+1g+2h+2k+2l+1m+3n+7o+6r+1s+3t+3u+2w+5y$	1	101940	9010	0.09
10	$2a+1c+1d+5e+3f+1g+3h+3i+2l+3n+5o+1q+3r+5s+5t+5u+1w+1y$	1	115010	12280	0.11

#	The Pattern	Number of Matches	CPU Time (μ sec) Prefix	CPU Time (μ sec) Suffix	Suf/Pre Ratio
11	$1a+1b+1c+2d+9e+1f+1g+1h+6i+1k+1m+3n+2o+2p+3r+8s+5t+1u+1v$	1	105700	11460	0.11
12	$5a+1b+1d+7e+1f+2h+4i+7l+1m+3o+2r+8s+3t+3w+2y$	1	106210	8770	0.08
13	$6a+2b+3d+4e+1f+1g+2h+8i+3m+4n+1o+1p+2r+4s+6t+2u$	1	106620	9540	0.09
14	$7a+2b+3c+3d+3e+4h+4i+2k+1l+1m+3n+2o+4r+3s+6t+1u+1v$	1	109360	11120	0.1
15	$4a+1b+3c+2d+4e+5h+2l+6m+2n+5o+2r+4s+3t+4w+3y$	1	107880	8800	0.08
16	$4a+3c+2d+6e+5h+3i+1l+3n+4o+5r+3s+6t+3u+1w+1y$	1	115110	9780	0.08
17	$5a+2b+1d+6e+1f+1g+2h+3i+1k+2l+4n+4o+3r+7s+4t+1u+2w+1y$	1	113500	12350	0.11
18	$1a+1c+4e+1f+2g+5h+5i+1j+2n+5o+2r+5s+10t+4u+2y$	1	99770	8120	0.08
19	$3a+1b+8e+1f+8h+5i+3l+1m+1p+4r+6s+5t+3w+1y$	1	101570	7460	0.07
20	$8a+1d+5e+1g+3i+4l+8n+7o+5r+2s+3t+1y+2z$	1	100520	7250	0.07

#	The Pattern	Number of Matches	CPU Time (μ sec) Prefix	Suffix	Suf/Pre Ratio
21	$2a+2b+8e+2g+4h+7i+3l+5n+5o+1p+3r+1s+4t+1u+1v+1x$	1	105140	8280	0.08
22	$3a+1c+1d+8e+1g+4h+4i+2k+2l+2m+3n+1o+2r+3s+6t+2u+2v+2w+1y$	1	113390	14220	0.13
23	$5a+1b+1c+2d+3e+2g+3h+5i+1k+2l+2m+5n+1o+3r+2s+5t+1u+5w+1y$	1	108230	14010	0.13
24	$4a+2b+1d+8e+1f+3h+2i+1k+2l+2m+5n+5o+3r+1s+6t+3w+1y$	1	112900	11260	0.1
25	$4a+1b+3c+1d+6e+2f+1g+4h+4i+2l+1m+4n+2o+2p+3r+5s+2t+2u+1v$	1	115810	14840	0.13
26	$2a+1b+1d+11e+4h+1i+1l+3n+5o+1q+3r+7s+6t+2u+1v+1w$	1	107460	9280	0.09
27	$3a+4d+6e+2f+1g+1h+5i+2l+1m+4n+7o+1p+4r+3s+3t+1v+1x+1y$	1	111360	11150	0.1
28	$3a+1b+3c+10e+3g+3h+5i+1k+1l+1m+5n+1o+1p+2r+3s+5t+1u+1x$	1	103700	9530	0.09
29	$4a+1b+2c+6d+6e+1f+4h+5i+2l+6n+3o+2p+3r+2s+2t+1w$	1	107040	10190	0.1
30	$3a+5e+4f+2g+6h+4i+2k+5n+3o+1p+5r+3s+6t+1y$	1	103930	7960	0.08

#	The Pattern	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
31	$2a+1b+1d+1e+1f+1h+2i+1k+3l+3n+11o+3p+3r+3s+3t+4u+2v+3w+2y$	1	102150	10360	0.1
32	$2a+2c+3d+5e+3f+1g+2h+1i+7l+2m+2n+8o+4r+2s+2t+1u+2w+1y$	1	105830	13890	0.13
33	$4a+1b+1d+7e+1f+1g+5h+5i+4l+3m+2n+2o+1q+2r+3s+6t+1u+1y$	1	112380	12310	0.11
34	$3a+1b+3c+6e+1f+1g+2h+1i+1l+1m+6n+4o+4r+4s+5t+4u+3y$	1	113490	11070	0.1
35	$5a+1b+1c+1d+2e+1f+1g+4h+6i+4l+2m+3n+6o+2s+6t+1v+3w+1y$	1	104800	9970	0.1
36	$5a+3d+6e+1f+1g+3h+3i+2l+1m+4n+1o+1p+3r+7s+3t+1u+1v+2w+2y$	1	111940	14660	0.13
37	$3a+2c+8e+2f+2h+4i+1k+2l+2m+7n+5o+3p+1s+5t+1v+1x+1y$	1	104150	8000	0.08

Table A.20: CPU time of the prefix based and the suffix based algorithms for abelian patterns of length 50.

A.3.2 Comparison of the Two Algorithms on the Input Text T_{Real} for Commonly Used English Words

We picked the most frequent 1000 English words from the web site `http://www.duboislc.org/EducationWatch/First100Words.html` which are taken from [15]. We selected words of length ≥ 3 only, and for each word we computed its count in T_{Real}. After that, from the words of the same length, we selected equal number of the most frequently and the least frequently occurring words in T_{Real} (we selected all of the words of length ≥ 9, as they were very few in number). These word were then converted into equivalent abelian patterns and experiments were performed on these patterns.

Abelian Pattern Matching for English Words of Length 3

Table A.21 shows the respective CPU times taken by the two algorithms for finding the abelian patterns corresponding to the selected English words of length 3 in the input text T_{Real}. The first 10 patterns have highest count in T_{Real}, whereas the last 10 patterns have lowest count in T_{Real}.

The suffix based algorithm performed better than the prefix based algorithm for all the selected English words of length 3. The suffix to prefix CPU time ratio ranged between $0.32-0.75$. A significant difference in the suffix to prefix CPU time ratio is observable between frequent and infrequent words.

Abelian Pattern Matching for English Words of Length 4

Table A.22 shows the respective CPU times taken by the two algorithms for finding the abelian patterns corresponding to the selected English words of length 4 in the input text T_{Real}. The first 20 patterns have highest count in T_{Real}, whereas the last 20 patterns have lowest count in T_{Real}.

The suffix based algorithm performed better than the prefix based algorithm for all the selected English words of length 4. The suffix to prefix CPU time ratio range between $0.22-0.41$. A significant difference in the suffix to prefix CPU time ratio is observable between frequent and infrequent words.

Abelian Pattern Matching for English Words of Length 5

Table A.23 shows the respective CPU times taken by the two algorithms for finding the abelian patterns corresponding to the selected English words of length 5 in the input text T_{Real}. The first 15 patterns have highest count in T_{Real}, whereas the last 15 patterns have lowest count in T_{Real}.

The suffix based algorithm performed better than the prefix based algorithm for all the selected English words of length 5. The suffix to prefix CPU time ratio ranged between $0.18-0.36$. A significant difference in the suffix to prefix CPU time ratio is observable between frequent and infrequent words.

Abelian Pattern Matching for English Words of Length 6

Table A.24 shows the respective CPU times taken by the two algorithms for finding the abelian patterns corresponding to the selected English words of length 6 in the input text T_{Real}. The first 10 patterns have highest count in T_{Real}, whereas the last 10 patterns have lowest count in T_{Real}.

The suffix based algorithm performed better than the prefix based algorithm for all the selected English words of length 6. The suffix to prefix CPU time ratio ranged between $0.15 - 0.25$. A difference in the suffix to prefix CPU time ratio is observable between frequent and infrequent words.

Abelian Pattern Matching for English Words of Length 7

Table A.25 shows the respective CPU times taken by the two algorithms for finding the abelian patterns corresponding to the selected English words of length 7 in the input text T_{Real}. The first 10 patterns have highest count in T_{Real}, whereas the last 10 patterns have lowest count in T_{Real}.

The suffix based algorithm performed better than the prefix based algorithm for all the selected English words of length 6. The suffix to prefix CPU time ratio ranged between $0.13 - 0.25$. A difference in the suffix to prefix CPU time ratio is observable between frequent and infrequent words.

Abelian Pattern Matching for English Words of Length 8

Table A.26 shows the respective CPU times taken by the two algorithms for finding the abelian patterns corresponding to the selected English words of length 8 in the input text T_{Real}. The first 10 patterns have highest count in T_{Real}, whereas the last 10 patterns have lowest count in T_{Real}.

The suffix based algorithm performed better than the prefix based algorithm for all the selected English words of length 6. The suffix to prefix CPU time ratio ranged between $0.12 - 0.21$.

Abelian Pattern Matching for English Words of Length ≥ 9

Table A.27 shows the respective CPU times taken by the two algorithms for finding the abelian patterns corresponding to the selected English words of length $9-11$ in the input text T_{Real}. The first 21 words have length 9, words from $22-37$ have length 10 and the last 3 words have length 11.

The suffix based algorithm performed better than the prefix based algorithm for all the selected English words of length ≥ 9. The suffix to prefix CPU time ratio ranged between $0.11-0.19$ for words of length 9, $0.11-0.15$ for words of length 10 and $0.11-0.13$ for words of length 11.

Effect of the Pattern Length on the Relative Efficiency of the Suffix Based and the Prefix Based Algorithms

The suffix to prefix CPU time ratio decreased with an increase in the pattern length. The suffix to prefix CPU time ratio ranged between $0.32-0.75$ for the pattern of length 3, and it ranged between $0.11-0.13$ for the patterns of length 11.

Moreover, the suffix to prefix CPU time ratio for the frequent words was higher then the ratio for the infrequent words (this is readily observable for the pattern of smaller lengths e.g. the lowest suffix to prefix CPU time ratio for frequent patterns of length 3 was 0.51, whereas the highest suffix to prefix CPU time ratio for infrequent patterns of length 3 was 0.36).

#	The Word	Number of Matches	CPU Time (μ sec) Prefix	CPU Time (μ sec) Suffix	Suf/Pre Ratio
1	the	72726	212180	160030	0.75
2	hat	45657	168730	111980	0.66
3	and	40468	158460	101260	0.64
4	hit	28266	145120	85090	0.59
5	are	27054	144310	83240	0.58
6	her	25281	133800	77720	0.58
7	not	24682	137960	77550	0.56
8	hot	24625	139360	78630	0.56
9	you	22829	128430	69200	0.54
10	ten	22643	140630	71700	0.51
11	add	690	96000	30820	0.32
12	fig	684	91490	31020	0.34
13	few	600	92700	33130	0.36
14	key	591	93290	33320	0.36
15	fun	451	92420	30370	0.33
16	six	195	90180	32130	0.36
17	sky	177	88460	28990	0.33
18	big	138	87750	29440	0.34
19	job	53	86620	28590	0.33
20	box	45	86150	28710	0.33

Table A.21: CPU time of the prefix based and the suffix based algorithms for abelian patterns corresponding to English words of length 3. The first 10 patterns have highest count in T_{Real}, whereas the last 10 patterns have lowest count in T_{Real}.

#	The Word	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	that	17195	139920	55450	0.4
2	this	13503	135530	54340	0.4
3	then	12041	131220	52750	0.4
4	here	11892	131320	48620	0.37
5	rest	10947	132150	52050	0.39
6	heat	10870	132080	53820	0.41
7	with	10864	125540	48880	0.39
8	sand	10358	125390	44550	0.36
9	thin	9933	129400	45630	0.35
10	your	9392	123560	41450	0.34
11	have	8616	123190	41700	0.34
12	what	8171	122650	42730	0.35
13	them	8123	122880	45180	0.37
14	into	7869	125070	45150	0.36
15	than	7423	124570	43280	0.35
16	ears	6971	126970	44460	0.35
17	they	6539	119060	42380	0.36
18	note	6450	128460	43800	0.34
19	tone	6450	126520	45110	0.36
20	hear	6235	124740	42730	0.34

#	The Word	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
21	feet	226	112980	27610	0.24
22	warm	207	107800	25550	0.24
23	grew	199	106490	27590	0.26
24	book	182	101520	22290	0.22
25	fact	180	107780	26720	0.25
26	noun	161	108110	26350	0.24
27	rock	146	104880	26020	0.25
28	kept	142	103120	26880	0.26
29	swim	128	107710	26220	0.24
30	type	126	104970	27840	0.27
31	week	106	105600	24290	0.23
32	park	105	103320	24360	0.24
33	milk	105	100470	24190	0.24
34	cook	104	102110	22460	0.22
35	size	98	106300	28350	0.27
36	cows	83	106130	25880	0.24
37	bank	60	105770	24190	0.23
38	baby	57	98820	22670	0.23
39	copy	51	101710	23860	0.23
40	eggs	45	101520	25940	0.26

Table A.22: CPU time of the prefix based and the suffix based algorithms for abelian patterns corresponding to English words of length 4. The first 20 patterns have highest count in T_{Real}, whereas the last 20 patterns have lowest count in T_{Real}.

#	The Word	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	three	10001	138960	46820	0.34
2	there	10001	144080	45480	0.32
3	earth	9671	136790	48680	0.36
4	heart	9671	140340	49120	0.35
5	other	8096	133890	46110	0.34
6	their	5635	128520	40580	0.32
7	these	5487	133470	36720	0.28
8	night	5102	127470	33780	0.27
9	thing	5102	128850	34670	0.27
10	shall	4529	128350	29290	0.23
11	stand	3756	125050	33710	0.27
12	where	3630	127180	30290	0.24
13	those	3287	125910	38710	0.31
14	youre	3036	123420	32040	0.26
15	death	2886	124170	35480	0.29

#	The Word	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
16	build	96	117090	22010	0.19
17	human	94	121530	23900	0.2
18	enjoy	93	114950	24520	0.21
19	crops	80	115950	22990	0.2
20	class	77	115230	21520	0.19
21	greek	76	116860	21980	0.19
22	grass	70	117960	22000	0.19
23	cells	56	113530	23500	0.21
24	color	54	117350	22000	0.19
25	didnt	48	119380	23650	0.2
26	block	40	108520	19500	0.18
27	track	34	117680	22730	0.19
28	group	22	115800	22110	0.19
29	major	16	115940	23230	0.2
30	check	6	111180	21250	0.19

Table A.23: CPU time of the prefix based and the suffix based algorithms for abelian patterns corresponding to English words of length 5. The first 15 patterns have highest count in T_{Real}, whereas the last 15 patterns have lowest count in T_{Real}.

#	The Word	Number of Matches	CPU Time (μ sec) Prefix	CPU Time (μ sec) Suffix	Suf/Pre Ratio
1	though	2629	132180	26460	0.2
2	mother	2539	131390	32800	0.25
3	either	2353	133330	30590	0.23
4	father	2291	129870	32960	0.25
5	should	2074	130320	26090	0.2
6	sister	1971	131730	28580	0.22
7	stream	1607	126200	29680	0.24
8	resent	1520	133270	28480	0.21
9	before	1236	128550	23970	0.19
10	reason	1125	125400	29300	0.23
11	family	22	123460	20020	0.16
12	picked	22	122520	20560	0.17
13	valley	21	126240	20650	0.16
14	joined	20	127540	23970	0.19
15	pulled	18	118920	19080	0.16
16	slowly	16	121530	18770	0.15
17	plural	12	123730	18400	0.15
18	pushed	9	121940	22220	0.18
19	rhythm	4	127270	19970	0.16
20	column	3	124680	19940	0.16

Table A.24: CPU time of the prefix based and the suffix based algorithms for abelian patterns corresponding to English words of length 6. The first 10 patterns have highest count in T_{Real}, whereas the last 10 patterns have lowest count in T_{Real}.

#	The Word	Number of Matches	CPU Time (μ sec) Prefix	CPU Time (μ sec) Suffix	Suf/Pre Ratio
1	another	1457	132600	32990	0.25
2	thought	1247	132310	20500	0.15
3	brother	1028	128900	24990	0.19
4	nothing	867	134630	23410	0.17
5	weather	809	134350	26520	0.2
6	against	790	132910	22760	0.17
7	friends	750	131950	25040	0.19
8	through	714	131370	21670	0.16
9	without	665	132580	22060	0.17
10	strange	617	130950	26930	0.21
11	farmers	10	128800	20750	0.16
12	finally	8	129410	18400	0.14
13	exactly	7	123480	19080	0.15
14	decided	5	125130	16180	0.13
15	symbols	5	127340	16590	0.13
16	usually	5	126600	16670	0.13
17	century	2	127280	20860	0.16
18	climbed	2	128250	19450	0.15
19	problem	1	124220	19880	0.16
20	explain	1	125050	20400	0.16

Table A.25: CPU time of the prefix based and the suffix based algorithms for abelian patterns corresponding to English words of length 7. The first 10 patterns have highest count in T_{Real}, whereas the last 10 patterns have lowest count in T_{Real}.

#	The Word	Number of Matches	CPU Time (μ sec) Prefix	CPU Time (μ sec) Suffix	Suf/Pre Ratio
1	together	431	134230	23150	0.17
2	yourself	363	137430	23150	0.17
3	soldiers	322	137310	21720	0.16
4	business	314	130690	18400	0.14
5	remember	280	125540	15010	0.12
6	shoulder	241	132740	22950	0.17
7	straight	227	137080	22120	0.16
8	anything	175	132610	19910	0.15
9	southern	157	128580	27020	0.21
10	consider	151	131470	23440	0.18
11	students	7	133970	19020	0.14
12	equation	7	130460	22160	0.17
13	happened	4	130950	18230	0.14
14	products	4	133400	19060	0.14
15	movement	2	130720	19100	0.15
16	electric	1	135190	18720	0.14
17	probably	1	131650	16740	0.13
18	actually	1	131810	15260	0.12
19	practice	1	132970	20700	0.16
20	exciting	1	132580	17760	0.13

Table A.26: CPU time of the prefix based and the suffix based algorithms for abelian patterns corresponding to English words of length 8. The first 10 patterns have highest count in T_{Real}, whereas the last 10 patterns have lowest count in T_{Real}.

#	The Word	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
1	something	257	133960	25900	0.19
2	sometimes	112	134400	18240	0.14
3	thousands	104	139160	21790	0.16
4	determine	96	135560	18800	0.14
5	direction	61	135130	21220	0.16
6	represent	53	132190	16690	0.13
7	different	36	134460	18620	0.14
8	factories	34	130540	23540	0.18
9	questions	30	134150	20360	0.15
10	continued	30	135310	19770	0.15
11	statement	24	135290	18150	0.13
12	stretched	24	137250	20060	0.15
13	necessary	24	135350	18800	0.14
14	beautiful	18	134000	18190	0.14
15	important	14	136520	19930	0.15
16	carefully	11	135050	16850	0.12
17	consonant	7	133920	16740	0.13
18	difficult	6	133580	14200	0.11
19	suggested	5	134340	15180	0.11
20	underline	4	135380	17430	0.13

#	The Word	Number of Matches	CPU Time (μ sec)		Suf/Pre Ratio
			Prefix	Suffix	
21	syllables	2	129740	15130	0.12
22	themselves	221	133010	16400	0.12
23	understand	157	136450	20810	0.15
24	particular	80	134470	15100	0.11
25	everything	68	136380	20090	0.15
26	difference	65	134260	16040	0.12
27	conditions	43	136340	16650	0.12
28	experience	32	132240	14390	0.11
29	government	30	135550	17470	0.13
30	washington	25	138050	20930	0.15
31	especially	23	137630	15810	0.11
32	discovered	20	136400	17510	0.13
33	substances	12	135570	17530	0.13
34	presidents	12	135820	19550	0.14
35	experiment	5	135330	16720	0.12
36	dictionary	3	139160	19600	0.14
37	scientists	0	134260	16220	0.12
38	instruments	42	136730	17880	0.13
39	information	21	135360	15920	0.12
40	temperature	6	134640	14800	0.11

Table A.27: CPU time of the prefix based and the suffix based algorithms for abelian patterns corresponding to English words of length ≥ 9. The first 21 words have length 9, words from $22 - 37$ have length 10 and the last 3 words have length 11.

Appendix B

Empirical Analysis of the Parameterized Suffix Based Algorithms for Different Values of Epsilon

In this appendix, we present results of the experiments performed for an empirical analysis of the parameterized suffix based algorithms for different values of ϵ. For the experiments, we used same input texts and abelian patterns which were used for the empirical analysis of the prefix based and the suffix based algorithms (Appendix A).

We executed the parameterized suffix based algorithm for different values of reset threshold ϵ. The values of ϵ used in the experiments are $0.2, 0.4, 0.6$ and 0.8.

Note that the actual reset threshold for parameterized suffix based algorithm is obtained by $\lfloor \epsilon m \rfloor$, hence, more than one values of ϵ can result into the same amount of actual reset threshold. For $m = 12$ and $\epsilon = 0.8$, the reset threshold is 9 characters, which indicates an actual value of $\epsilon = 0.75$ in this case (i.e. $\epsilon = 0.8$ is same as $\epsilon = 0.75$ in this situation). We call $\epsilon = 0.8$ as the given value of ϵ and $\epsilon = 0.75$ as the actual value of ϵ.

In the following sections, we mention the given values of ϵ while presenting the results of the parameterized suffix based algorithm.

B.1 Comparison of Different Values of ϵ for the Input Text T_{4_U}

The input text T_{4_U} is based on the uniform distribution of the characters in $\Sigma := \{c_1, c_2, c_3, c_4\}$, i.e. on every text position in T_{4_U}, the probability of occurring a character $c_i \in \Sigma$ is same for all $1 \leq i \leq 4$. The text comprises of 10000000 characters, i.e. $n := |T_{4_U}| = 10000000$.

B.1.1 Abelian Pattern Matching for $m = 12$ in T_{4_U}

Table B.1 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ. The last column of the table shows the efficiency ranking of the three algorithms. In the raking, P denotes the prefix based algorithms, S denotes the suffix based algorithm and R denotes the parameterized suffix based algorithm (the CPU time considered for the parameterized suffix based algorithm in the ranking is the CPU time for the best ϵ for a pattern). The ranking value PRS means the prefix based algorithm is the most efficient algorithm, the parametrized suffix based algorithms is the second efficient algorithm and the suffix based algorithm is the slowest algorithm.

There was no single value of ϵ that was the best for all the patterns. The value $\epsilon = 0.8$ gave minimum CPU time for most of the patterns, however, $\epsilon = 0.6$ was quite close and in many cases the difference between the CPU time for $\epsilon = 0.6$ and $\epsilon = 0.8$ was just minimal. The parameterized algorithm was not at third place in any of the patterns. However, it was not the most efficient algorithm for most of the patterns (only in pattern 2 & 3, the parameterized algorithms was better than the other two algorithms).

B.1.2 Abelian Pattern Matching for $m = 48$ in T_{4_U}

Table B.2 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

The difference between the CPU time for different values of ϵ was not significant for patterns $4 - 10$. The most promising value for ϵ was 0.6. The parameterized algorithm was not at third place in any of the patterns. For patterns $3 - 10$, the time taken by the suffix based algorithm and the parameterized suffix based algorithm was almost the same.

B.1.3 Abelian Pattern Matching for $m = 120$ in T_{4_U}

Table B.3 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

The difference between the CPU time for different values of ϵ was not significant for patterns $4-10$. For patterns 2 & 3, $\epsilon = 0.6$ was the best followed by $\epsilon = 0.8$ with a small difference. Interestingly, $\epsilon = 0.8$ gave the slowest performance for pattern 1; here $\epsilon = 0.4$ was the best (although, it was not significantly better than $\epsilon = 0.6$ and $\epsilon = 0.2$). The interesting thing about pattern 1 is that prefix based algorithm was the most efficient algorithm for pattern 1. Moreover, for this pattern, the suffix based algorithm was 3.17 times slower than the prefix based algorithm (Section A.1.1), whereas the parameterized suffix based algorithm was only 1.37 times to 1.58 times slower than the prefix based algorithm for its different values of ϵ. The parameterized algorithm was not at third place in any of the patterns.

#	The Pattern	CPU Time (μ sec)				**Best**	CPU Time (μ sec)			**Algo**
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$3c_1 + 3c_2 + 3c_3 + 3c_4$	785420	735420	689550	725270	**0.6**	650010	751180	689550	**PRS**
2	$2c_1 + 3c_2 + 3c_3 + 4c_4$	665130	619260	588960	586890	**0.8**	613480	617110	586890	**RPS**
3	$1c_1 + 2c_2 + 3c_3 + 6c_4$	277950	236420	210630	207740	**0.8**	354030	217070	207740	**RSP**
4	$0c_1 + 4c_2 + 4c_3 + 4c_4$	140650	133030	125210	123740	**0.8**	340990	123400	123740	**SRP**
5	$0c_1 + 2c_2 + 4c_3 + 6c_4$	110670	103310	96610	94390	**0.8**	291090	92650	94390	**SRP**
6	$0c_1 + 2c_2 + 2c_3 + 8c_4$	92200	84730	80400	79210	**0.8**	246980	72910	79210	**SRP**
7	$0c_1 + 0c_2 + 6c_3 + 6c_4$	48300	48330	47580	47060	**0.8**	210480	46840	47060	**SRP**
8	$0c_1 + 0c_2 + 4c_3 + 8c_4$	47060	45450	44340	44250	**0.8**	224320	43660	44250	**SRP**
9	$0c_1 + 0c_2 + 2c_3 + 10c_4$	42930	41030	40550	40220	**0.8**	220550	39380	40220	**SRP**
10	$0c_1 + 0c_2 + 0c_3 + 12c_4$	27140	26850	26880	27110	**0.4**	189810	24970	26850	**SRP**

Table B.1: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 12 in the input text T_{4_U}. The last column of the table represents the efficiency ranking of the algorithms. P represent the prefix based algorithms, S represent the suffix based algorithm and R represents the parameterized suffix based algorithm (the CPU time considered for the parameterized suffix based algorithm for a pattern is the CPU time for the best ϵ). The ranking value PRS means the prefix based algorithm is the most efficient algorithm, the parametrized suffix based algorithms is the second efficient algorithm and the suffix based algorithm is the slowest algorithm.

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$12c_1 + 12c_2 + 12c_3 + 12c_4$	454210	474060	450950	468410	**0.6**	340700	662980	450950	**PRS**
2	$8c_1 + 12c_2 + 12c_3 + 16c_4$	389360	330780	291910	302530	**0.6**	278030	379340	291910	**PRS**
3	$4c_1 + 8c_2 + 12c_3 + 24c_4$	181500	115030	89190	86760	**0.8**	206180	87070	86760	**RSP**
4	$0c_1 + 16c_2 + 16c_3 + 16c_4$	21530	20860	20810	20840	**0.6**	203180	20070	20810	**SRP**
5	$0c_1 + 8c_2 + 16c_3 + 24c_4$	21730	20940	20860	20870	**0.6**	192050	20160	20860	**SRP**
6	$0c_1 + 8c_2 + 8c_3 + 32c_4$	22250	20980	20960	20950	**0.8**	189590	20590	20950	**SRP**
7	$0c_1 + 0c_2 + 24c_3 + 24c_4$	16630	16740	16560	16770	**0.6**	173990	16620	16560	**RSP**
8	$0c_1 + 0c_2 + 16c_3 + 32c_4$	16450	16300	16280	16360	**0.6**	181610	16180	16280	**SRP**
9	$0c_1 + 0c_2 + 8c_3 + 40c_4$	16130	16320	16080	16110	**0.6**	177990	16190	16080	**RSP**
10	$0c_1 + 0c_2 + 0c_3 + 48c_4$	13890	13850	13900	13930	**0.4**	171140	13610	13850	**SRP**

Table B.2: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 48 in the input text T_{4_U}. The last column of the table represents the efficiency ranking of the algorithms.

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$30c_1 + 30c_2 + 30c_3 + 30c_4$	372930	369480	370270	425770	**0.4**	269550	855630	369480	**PRS**
2	$20c_1 + 30c_2 + 30c_3 + 40c_4$	301390	293950	253300	268920	**0.6**	223810	313660	253300	**PRS**
3	$10c_1 + 20c_2 + 30c_3 + 60c_4$	165960	90410	73080	73950	**0.6**	194020	74130	73080	**RSP**
4	$0c_1 + 40c_2 + 40c_3 + 40c_4$	12510	12670	12700	12430	**0.8**	188570	12290	12430	**SRP**
5	$0c_1 + 20c_2 + 40c_3 + 60c_4$	12780	12480	12680	12500	**0.4**	185990	12560	12480	**RSP**
6	$0c_1 + 20c_2 + 20c_3 + 80c_4$	12220	12190	12330	12210	**0.4**	176270	12080	12190	**SRP**
7	$0c_1 + 0c_2 + 60c_3 + 60c_4$	9390	9480	9310	9380	**0.6**	171720	9330	9310	**RSP**
8	$0c_1 + 0c_2 + 40c_3 + 80c_4$	9590	9500	9490	9510	**0.6**	173370	9350	9490	**SRP**
9	$0c_1 + 0c_2 + 20c_3 + 100c_4$	9480	9530	9550	9580	**0.2**	180770	9380	9480	**SRP**
10	$0c_1 + 0c_2 + 0c_3 + 120c_4$	6760	6800	6800	6820	**0.2**	180480	6740	6760	**SRP**

Table B.3: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 120 in the input text T_{4_U}. The last column of the table represents the efficiency ranking of the algorithms.

B.2 Comparison of Different Values of ϵ for the Input Text $T_{4_{\acute{U}}}$

The input text $T_{4_{\acute{U}}}$ is based on a non-uniform distribution of the characters in $\Sigma := \{c_1, c_2, c_3, c_4\}$. The frequency ratio of the characters in $T_{4_{\acute{U}}}$ is as follows:

$$c_1 : c_2 : c_3 : c_4 \equiv 6 : 3 : 2 : 1$$

The text comprises of 10000000 characters, i.e. $n := |T_{4_{\acute{U}}}| = 10000000$.

B.2.1 Abelian Pattern Matching for $m = 12$ in $T_{4_{\acute{U}}}$

Table B.4 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

The difference between the CPU time for different values of ϵ was not significant for patterns $9-10$ and $12-15$. The ϵ values 0.6 and 0.8 were the most prominent values, although, there was not a big difference between the CPU time taken by the parameterized suffix based algorithm for $\epsilon = 0.6$ and $\epsilon = 0.8$. The parameterized algorithm was not at third place in any of the patterns. In six patterns (pattern 3,4,6,7,8 & 12), the parametrized suffix based algorithm was more efficient than the other two algorithms.

B.2.2 Abelian Pattern Matching for $m = 48$ in $T_{4_{\acute{U}}}$

Table B.5 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

The difference between the CPU time for different values of ϵ was not significant for patterns $9-15$. The value $\epsilon = 0.8$ resulted in efficient execution of the algorithm for many patterns. The performance of the algorithm for $\epsilon = 0.6$ was also good. In pattern 1 (where prefix based algorithm was the fastest algorithm), the parameterized algorithm was most efficient for $\epsilon = 0.4$ followed by $\epsilon = 0.6$. The parameterized algorithm was not at third place in any of the patterns. It outperformed the other two algorithms in two patterns (pattern 4 & 5).

B.2.3 Abelian Pattern Matching for $m = 120$ in $T_{4_{\acute{U}}}$

Table B.6 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

The difference between the CPU time for different values of ϵ was not significant for patterns $9 - 15$. For patterns 1 & 2, $\epsilon = 0.6$ was the best and $\epsilon = 0.8$ gave the slowest CPU time in these patterns. Overall, $\epsilon = 0.6$ and $\epsilon = 0.8$ gave better results. The parameterized algorithm was the slowest algorithm for pattern 3, however, the difference between the parameterized algorithm and the suffix based algorithm (the second in the efficiency ranking for pattern 3) was marginal. Parameterized suffix based algorithm outperformed the other two algorithms for two patterns (pattern 5 & 6).

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$6c_1 + 3c_2 + 2c_3 + 1c_4$	860030	834790	821550	829810	**0.6**	773380	884280	821550	**PRS**
2	$5c_1 + 3c_2 + 2c_3 + 2c_4$	683660	664290	603620	612750	**0.6**	551870	656370	603620	**PRS**
3	$4c_1 + 3c_2 + 2c_3 + 3c_4$	411840	395300	349370	352130	**0.6**	372230	378470	349370	**RPS**
4	$4c_1 + 2c_2 + 3c_3 + 3c_4$	389980	336310	271900	265010	**0.8**	332440	269570	265010	**RSP**
5	$3c_1 + 2c_2 + 3c_3 + 4c_4$	281840	245760	191520	188630	**0.8**	259400	174690	188630	**SRP**
6	$2c_1 + 2c_2 + 3c_3 + 5c_4$	204690	154620	119950	122620	**0.6**	258560	121200	119950	**RSP**
7	$2c_1 + 2c_2 + 4c_3 + 4c_4$	202020	156690	122990	117670	**0.8**	222320	125310	117670	**RSP**
8	$1c_1 + 1c_2 + 4c_3 + 6c_4$	98620	82980	74180	73110	**0.8**	237140	76010	73110	**RSP**
9	$0c_1 + 2c_2 + 4c_3 + 6c_4$	43400	41860	41640	41510	**0.8**	224780	40200	41510	**SRP**
10	$0c_1 + 1c_2 + 3c_3 + 8c_4$	36960	36350	36140	35940	**0.8**	210610	34550	35940	**SRP**
11	$0c_1 + 1c_2 + 4c_3 + 7c_4$	40890	39360	39560	39920	**0.4**	211630	37930	39360	**SRP**
12	$0c_1 + 1c_2 + 5c_3 + 6c_4$	40060	36970	36920	36770	**0.8**	211850	37580	36770	**RSP**
13	$0c_1 + 0c_2 + 4c_3 + 8c_4$	26620	26240	26110	26090	**0.8**	188910	24740	26090	**SRP**
14	$0c_1 + 0c_2 + 6c_3 + 6c_4$	26410	26370	26530	26410	**0.4**	179450	24520	26370	**SRP**
15	$0c_1 + 0c_2 + 0c_3 + 12c_4$	20640	20640	20640	20690	**0.6**	185890	18610	20640	**SRP**

Table B.4: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 12 in the input text $T_{4_{\acute{U}}}$. The last column of the table represents the efficiency ranking of the algorithms.

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$24c_1 + 12c_2 + 8c_3 + 4c_4$	437260	416760	422450	474940	**0.4**	352090	727910	416760	**PRS**
2	$20c_1 + 12c_2 + 8c_3 + 8c_4$	360530	366690	352410	387860	**0.6**	267610	477940	352410	**PRS**
3	$16c_1 + 12c_2 + 8c_3 + 12c_4$	333360	333500	273800	247300	**0.8**	224590	249280	247300	**PRS**
4	$16c_1 + 8c_2 + 12c_3 + 12c_4$	317860	294330	217960	194090	**0.8**	202980	196510	194090	**RSP**
5	$12c_1 + 8c_2 + 12c_3 + 16c_4$	291290	228170	129270	123360	**0.8**	189440	125270	123360	**RSP**
6	$8c_1 + 8c_2 + 12c_3 + 20c_4$	191650	98100	79560	79670	**0.6**	189360	73060	79560	**SRP**
7	$8c_1 + 8c_2 + 16c_3 + 16c_4$	186490	91130	77240	76490	**0.8**	200280	74890	76490	**SRP**
8	$4c_1 + 4c_2 + 16c_3 + 24c_4$	49070	35890	35920	35920	**0.4**	179100	35560	35890	**SRP**
9	$0c_1 + 8c_2 + 16c_3 + 24c_4$	16530	16560	16500	16540	**0.6**	181380	16480	16500	**SRP**
10	$0c_1 + 4c_2 + 12c_3 + 32c_4$	16820	16720	16770	16830	**0.4**	182990	16640	16720	**SRP**
11	$0c_1 + 4c_2 + 16c_3 + 28c_4$	16680	16810	16730	16710	**0.2**	180370	16630	16680	**SRP**
12	$0c_1 + 4c_2 + 20c_3 + 24c_4$	17140	17090	17210	16950	**0.8**	179880	17020	16950	**RSP**
13	$0c_1 + 0c_2 + 16c_3 + 32c_4$	14260	14270	14300	14200	**0.8**	179820	13920	14200	**SRP**
14	$0c_1 + 0c_2 + 24c_3 + 24c_4$	14300	14110	14050	14050	**0.8**	180070	13980	14050	**SRP**
15	$0c_1 + 0c_2 + 0c_3 + 48c_4$	12300	12310	12340	12260	**0.8**	180230	12010	12260	**SRP**

Table B.5: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 48 in the input text $T_{4_{\mathcal{U}}}$. The last column of the table represents the efficiency ranking of the algorithms.

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$60c_1 + 30c_2 + 20c_3 + 10c_4$	346040	347350	342510	396290	**0.6**	285140	877670	342510	**PRS**
2	$50c_1 + 30c_2 + 20c_3 + 20c_4$	321180	322180	320790	384450	**0.6**	211840	501950	320790	**PRS**
3	$40c_1 + 30c_2 + 20c_3 + 30c_4$	320850	323360	274100	233760	**0.8**	195380	233640	233760	**PSR**
4	$40c_1 + 20c_2 + 30c_3 + 30c_4$	319580	306410	213200	184040	**0.8**	175670	191180	184040	**PRS**
5	$30c_1 + 20c_2 + 30c_3 + 40c_4$	324600	279010	121930	121380	**0.8**	193350	129560	121380	**RSP**
6	$20c_1 + 20c_2 + 30c_3 + 50c_4$	178140	78260	70210	66840	**0.8**	188030	73940	66840	**RSP**
7	$20c_1 + 20c_2 + 40c_3 + 40c_4$	173150	73320	66950	70190	**0.6**	173820	66840	66950	**SRP**
8	$10c_1 + 10c_2 + 40c_3 + 60c_4$	39870	34290	34020	34350	**0.6**	184870	33740	34020	**SRP**
9	$0c_1 + 20c_2 + 40c_3 + 60c_4$	9500	9530	9530	9520	**0.2**	182850	9410	9500	**SRP**
10	$0c_1 + 10c_2 + 30c_3 + 80c_4$	9520	9480	9470	9510	**0.6**	175600	9390	9470	**SRP**
11	$0c_1 + 10c_2 + 40c_3 + 70c_4$	9570	9520	9500	9510	**0.6**	174880	9420	9500	**SRP**
12	$0c_1 + 10c_2 + 50c_3 + 60c_4$	9670	9570	9560	9480	**0.8**	177200	9480	9480	**SRP**
13	$0c_1 + 0c_2 + 40c_3 + 80c_4$	6970	6920	7030	6930	**0.4**	179810	6790	6920	**SRP**
14	$0c_1 + 0c_2 + 60c_3 + 60c_4$	7040	6920	6960	6920	**0.8**	179320	6780	6920	**SRP**
15	$0c_1 + 0c_2 + 0c_3 + 120c_4$	5720	5730	5720	5770	**0.6**	184510	5620	5720	**SRP**

Table B.6: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 120 in the input text $T_{4_{\acute{U}}}$. The last column of the table represents the efficiency ranking of the algorithms.

B.3 Comparison of Different Values of ϵ for the Input Text T_{8_U}

The input text T_{8_U} is based on the uniform distribution of the characters in $\Sigma := \{c_1, c_2, \ldots, c_8\}$, i.e. on every text position in T_{8_U}, the probability of occurring a character $c_i \in \Sigma$ is same for all $1 \leq i \leq 8$. The text comprises of 10000000 characters, i.e. $n := |T_{8_U}| = 10000000$.

B.3.1 Abelian Pattern Matching for $m = 16$ in T_{8_U}

Table B.7 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

The difference between the CPU time for different values of ϵ was not significant for patterns $6 - 15$. The value $\epsilon = 0.8$ gave minimum CPU time for many patterns; it was followed by the value $\epsilon = 0.6$. The parameterized algorithm was not at third place in any of the patterns. It outperformed the other two algorithms for just one patterns (pattern 3).

B.3.2 Abelian Pattern Matching for $m = 80$ in T_{8_U}

Table B.8 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

The difference between the CPU time for different values of ϵ was not significant for the patterns of length 80. In situations where the difference between the CPU time taken by the parameterized suffix based algorithm for different values of ϵ was noticeable (pattern 1,2 & 3), the values $\epsilon = 0.6$ and $\epsilon = 0.8$ performed better than other values for ϵ. The parameterized algorithm was not at third place in any of the patterns.

B.3.3 Abelian Pattern Matching for $m = 240$ in T_{8_U}

Table B.9 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

The value $\epsilon = 0.6$ performed better than other values for ϵ in situations where the difference between the CPU time taken by the parameterized suffix based algorithm for different values of ϵ was noticeable (pattern 1 & 2). The parameterized algorithm was second in the efficiency ranking of the three algorithms for all the patterns. However, in most of the situations, there was very small difference between the parameterized algorithm and the most efficient algorithm.

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$2c_1 + 2c_2 + 2c_3 + 2c_4$ $+2c_5 + 2c_6 + 2c_7 + 2c_8$	269400	215510	186190	174930	**0.8**	348110	169740	174930	**SRP**
2	$1c_1 + 2c_2 + 2c_3 + 2c_4$ $+2c_5 + 2c_6 + 2c_7 + 3c_8$	222720	181360	156650	151200	**0.8**	340720	145100	151200	**SRP**
3	$0c_1 + 1c_2 + 2c_3 + 2c_4$ $+2c_5 + 2c_6 + 3c_7 + 4c_8$	100490	84090	75650	73990	**0.8**	319630	75550	73990	**RSP**
4	$0c_1 + 0c_2 + 1c_3 + 2c_4$ $+2c_5 + 3c_6 + 4c_7 + 4c_8$	59810	53420	51370	50510	**0.8**	287970	48960	50510	**SRP**
5	$0c_1 + 0c_2 + 0c_3 + 1c_4$ $+3c_5 + 3c_6 + 4c_7 + 5c_8$	43820	41680	40800	40870	**0.6**	258490	38810	40800	**SRP**
6	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+4c_5 + 4c_6 + 4c_7 + 4c_8$	33840	32570	32480	32660	**0.6**	233570	31780	32480	**SRP**
7	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+2c_5 + 4c_6 + 5c_7 + 5c_8$	33290	33100	32690	32470	**0.8**	241970	31230	32470	**SRP**
8	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+1c_5 + 3c_6 + 6c_7 + 6c_8$	30460	29790	30140	29680	**0.8**	238920	28650	29680	**SRP**

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
9	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 5c_6 + 5c_7 + 6c_8$	24980	24240	24510	24520	**0.4**	204500	23500	24240	**SRP**
10	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 4c_6 + 4c_7 + 8c_8$	25960	25550	25380	25720	**0.6**	218820	24470	25380	**SRP**
11	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 8c_7 + 8c_8$	20710	20460	20840	20200	**0.8**	210580	19230	20200	**SRP**
12	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 6c_7 + 10c_8$	20070	20040	20040	20100	**0.6**	206120	18900	20040	**SRP**
13	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 4c_7 + 12c_8$	20050	19840	20020	19890	**0.4**	213960	18930	19840	**SRP**
14	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 2c_7 + 14c_8$	20270	19840	20060	20030	**0.4**	224630	18790	19840	**SRP**
15	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 0c_7 + 16c_8$	16810	16760	16680	16930	**0.6**	209800	15610	16680	**SRP**

Table B.7: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 16 in the input text T_{8_U}. The last column of the table represents the efficiency ranking of the algorithms.

#	The Pattern	CPU Time (μ sec) $\epsilon = 0.2$	CPU Time (μ sec) $\epsilon = 0.4$	CPU Time (μ sec) $\epsilon = 0.6$	CPU Time (μ sec) $\epsilon = 0.8$	**Best** ϵ	CPU Time (μ sec) **Pre**	CPU Time (μ sec) **Suf**	CPU Time (μ sec) **paRa**	**Algo Ranking**
1	$10c_1 + 10c_2 + 10c_3 + 10c_4 + 10c_5 + 10c_6 + 10c_7 + 10c_8$	276840	265110	235980	253450	**0.6**	269420	267990	235980	**RSP**
2	$5c_1 + 10c_2 + 10c_3 + 10c_4 + 10c_5 + 10c_6 + 10c_7 + 15c_8$	222860	183920	150650	148520	**0.8**	250130	150110	148520	**RSP**
3	$0c_1 + 5c_2 + 10c_3 + 10c_4 + 10c_5 + 10c_6 + 15c_7 + 20c_8$	23960	22240	22050	21710	**0.8**	226200	22090	21710	**RSP**
4	$0c_1 + 0c_2 + 5c_3 + 10c_4 + 10c_5 + 15c_6 + 20c_7 + 20c_8$	16110	16170	16300	16230	**0.2**	197130	15920	16110	**SRP**
5	$0c_1 + 0c_2 + 0c_3 + 5c_4 + 15c_5 + 15c_6 + 20c_7 + 25c_8$	14160	14220	14140	14160	**0.6**	180710	13890	14140	**SRP**
6	$0c_1 + 0c_2 + 0c_3 + 0c_4 + 20c_5 + 20c_6 + 20c_7 + 20c_8$	12600	12700	12620	12640	**0.2**	179190	12630	12600	**RSP**
7	$0c_1 + 0c_2 + 0c_3 + 0c_4 + 10c_5 + 20c_6 + 25c_7 + 25c_8$	12660	12670	12520	12750	**0.6**	192060	12610	12520	**RSP**
8	$0c_1 + 0c_2 + 0c_3 + 0c_4 + 5c_5 + 15c_6 + 30c_7 + 30c_8$	12650	12560	12590	12630	**0.4**	183830	12440	12560	**SRP**

#	The Pattern	CPU Time (μ sec)				**Best**	CPU Time (μ sec)			**Algo**
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
9	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 25c_6 + 25c_7 + 30c_8$	10960	10830	10830	10930	**0.6**	178860	10690	10830	**SRP**
10	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 20c_6 + 20c_7 + 40c_8$	10910	10770	10940	10870	**0.4**	179460	10780	10770	**RSP**
11	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 40c_7 + 40c_8$	9230	9190	9290	9190	**0.8**	173630	8980	9190	**SRP**
12	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 30c_7 + 50c_8$	9160	9340	9200	9200	**0.2**	172080	9040	9160	**SRP**
13	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 20c_7 + 60c_8$	9150	9220	9170	9270	**0.2**	171940	9010	9150	**SRP**
14	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 10c_7 + 70c_8$	9170	9180	9250	9220	**0.2**	180490	9000	9170	**SRP**
15	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 0c_7 + 80c_8$	8080	8070	8080	8120	**0.4**	170850	7860	8070	**SRP**

Table B.8: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 80 in the input text T_{8_U}. The last column of the table represents the efficiency ranking of the algorithms.

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$30c_1 + 30c_2 + 30c_3 + 30c_4$ $+30c_5 + 30c_6 + 30c_7 + 30c_8$	246940	245540	244130	338350	**0.6**	227400	475400	244130	**PRS**
2	$15c_1 + 30c_2 + 30c_3 + 30c_4$ $+30c_5 + 30c_6 + 30c_7 + 45c_8$	207850	176880	138950	140640	**0.6**	207170	137860	138950	**SRP**
3	$0c_1 + 15c_2 + 30c_3 + 30c_4$ $+30c_5 + 30c_6 + 45c_7 + 60c_8$	8990	8940	9340	8940	**0.8**	197870	8740	8940	**SRP**
4	$0c_1 + 0c_2 + 15c_3 + 30c_4$ $+30c_5 + 45c_6 + 60c_7 + 60c_8$	6760	6670	6650	6790	**0.6**	185940	6600	6650	**SRP**
5	$0c_1 + 0c_2 + 0c_3 + 15c_4$ $+45c_5 + 45c_6 + 60c_7 + 75c_8$	5920	5840	5860	5890	**0.4**	172040	5760	5840	**SRP**
6	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+60c_5 + 60c_6 + 60c_7 + 60c_8$	5170	5150	5150	5210	**0.6**	172040	5120	5150	**SRP**
7	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+30c_5 + 60c_6 + 75c_7 + 75c_8$	5270	5310	5330	5310	**0.2**	181460	5180	5270	**SRP**
8	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+15c_5 + 45c_6 + 90c_7 + 90c_8$	5310	5300	5310	5300	**0.8**	179710	5150	5300	**SRP**

#	The Pattern	CPU Time (μ sec)				**Best**	CPU Time (μ sec)			**Algo**
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
9	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 75c_6 + 75c_7 + 90c_8$	4660	4650	4680	4650	**0.8**	179290	4500	4650	**SRP**
10	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 60c_6 + 60c_7 + 120c_8$	4560	4540	4490	4560	**0.6**	173240	4350	4490	**SRP**
11	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 120c_7 + 120c_8$	3860	3850	3900	3910	**0.4**	169900	3730	3850	**SRP**
12	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 90c_7 + 150c_8$	3870	3890	3880	3900	**0.2**	169550	3760	3870	**SRP**
13	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 60c_7 + 180c_8$	3890	3930	3890	3890	**0.8**	169940	3870	3890	**SRP**
14	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 30c_7 + 210c_8$	3880	3910	3880	3940	**0.6**	178240	3830	3880	**SRP**
15	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 0c_7 + 240c_8$	3380	3410	3440	3360	**0.8**	172970	3260	3360	**SRP**

Table B.9: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 240 in the input text T_{8_U}. The last column of the table represents the efficiency ranking of the algorithms.

B.4 Comparison of Different Values of ϵ for the Input Text $T_{8_{\acute{U}}}$

The input text $T_{8_{\acute{U}}}$ is based on a non-uniform distribution of the characters in $\Sigma := \{c_1, c_2, \ldots, c_8\}$. The frequency ratio of the characters in $T_{8_{\acute{U}}}$ is as follows:

$$c_1 : c_2 : c_3 : c_4 : c_5 : c_6 : c_7 : c_8 \equiv 15 : 10 : 8 : 6 : 4 : 3 : 2 : 1$$

The text comprises of 10000000 characters, i.e. $n := |T_{8_{\acute{U}}}| = 10000000$.

B.4.1 Abelian Pattern Matching for $m = 49$ in $T_{8_{\acute{U}}}$

Table B.10 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

The difference between the CPU time for different values of ϵ was not significant for patterns $5 - 15$. The value $\epsilon = 0.8$ gave minimum CPU time for most of those patterns for which the difference between the CPU time taken by the parameterized suffix based algorithm for different values of ϵ was noticeable; it was followed by the value $\epsilon = 0.6$. The parameterized algorithm was not at third place in any of the patterns and it outperformed the other two algorithms in two patterns (pattern 1 & 2). For other patterns, there was not a significant difference between the CPU times taken by the parameterized suffix based algorithm and the suffix based algorithm.

B.4.2 Abelian Pattern Matching for $m = 98$ in $T_{8_{\acute{U}}}$

Table B.11 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

In situations where the difference between the CPU time taken by the parameterized suffix based algorithm for different values of ϵ was noticeable (patterns $1 - 4$), the values $\epsilon = 0.6$ and $\epsilon = 0.8$ performed better than other values for ϵ. The parameterized algorithm was not at third place in any of the patterns. It outperformed the other two algorithms in three patterns (pattern 1,3, & 4). For many patterns, there was not a significant difference between the CPU times taken by the parameterized suffix based algorithm and the suffix based algorithm (e.g. in pattern 12, although the parameterized suffix based algorithm is the most efficient, the CPU time taken by the algorithm is almost same as the CPU time taken by the suffix based algorithm).

B.4.3 Abelian Pattern Matching for $m = 196$ in $T_{8_{\acute{U}}}$

Table B.12 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

The difference between the CPU time for different values of ϵ was not significant for patterns $5 - 15$. The values $\epsilon = 0.6$ and $\epsilon = 0.8$ performed better than other values for ϵ in situations where the difference between the CPU time taken by the parameterized suffix based algorithm for different values of ϵ was noticeable. The parameterized algorithm was not at third place in any of the patterns and it outperformed the other two algorithms in two patterns (pattern 1 & 4).

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$15c_1 + 10c_2 + 8c_3 + 6c_4$ $+4c_5 + 3c_6 + 2c_7 + 1c_8$	241350	216160	190890	189370	**0.8**	287460	207050	189370	**RSP**
2	$10c_1 + 9c_2 + 7c_3 + 6c_4$ $+5c_5 + 5c_6 + 3c_7 + 4c_8$	307910	263390	193850	180660	**0.8**	245780	190120	180660	**RSP**
3	$8c_1 + 7c_2 + 7c_3 + 7c_4$ $+4c_5 + 5c_6 + 5c_7 + 6c_8$	273010	205020	135550	129930	**0.8**	224830	123260	129930	**SRP**
4	$5c_1 + 6c_2 + 6c_3 + 6c_4$ $+6c_5 + 6c_6 + 7c_7 + 7c_8$	183570	101740	80400	81040	**0.6**	222540	79680	80400	**SRP**
5	$0c_1 + 7c_2 + 7c_3 + 7c_4$ $+7c_5 + 7c_6 + 7c_7 + 7c_8$	19560	19490	19270	19510	**0.6**	221710	19000	19270	**SRP**
6	$0c_1 + 5c_2 + 6c_3 + 7c_4$ $+7c_5 + 8c_6 + 8c_7 + 8c_8$	19850	19180	19330	18960	**0.8**	210020	18940	18960	**SRP**
7	$0c_1 + 2c_2 + 6c_3 + 7c_4$ $+7c_5 + 8c_6 + 9c_7 + 10c_8$	18730	18490	18550	18440	**0.8**	199820	18380	18440	**SRP**
8	$0c_1 + 0c_2 + 7c_3 + 8c_4$ $+8c_5 + 8c_6 + 8c_7 + 10c_8$	16200	16270	16130	16140	**0.6**	198360	16180	16130	**RSP**

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
9	$0c_1 + 0c_2 + 3c_3 + 8c_4$ $+9c_5 + 9c_6 + 10c_7 + 10c_8$	16620	16670	16600	16710	**0.6**	191290	16350	16600	**SRP**
10	$0c_1 + 0c_2 + 0c_3 + 9c_4$ $+10c_5 + 10c_6 + 10c_7 + 10c_8$	14880	14750	14970	14820	**0.4**	187890	14430	14750	**SRP**
11	$0c_1 + 0c_2 + 0c_3 + 5c_4$ $+10c_5 + 11c_6 + 11c_7 + 12c_8$	14490	14590	14330	14500	**0.6**	189860	14170	14330	**SRP**
12	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+6c_5 + 12c_6 + 14c_7 + 17c_8$	12980	12920	13090	13440	**0.4**	183460	12750	12920	**SRP**
13	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 16c_6 + 16c_7 + 17c_8$	12550	12610	12540	12560	**0.6**	182630	12230	12540	**SRP**
14	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 24c_7 + 25c_8$	12000	11980	11950	12000	**0.6**	184920	11640	11950	**SRP**
15	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 0c_7 + 49c_8$	11720	11730	11930	11770	**0.2**	187720	11420	11720	**SRP**

Table B.10: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 49 in the input text $T_{8_{\acute{U}}}$. The last column of the table represents the efficiency ranking of the algorithms.

#	**The Pattern**	**CPU Time (μ sec)**				**Best**	**CPU Time (μ sec)**			**Algo**
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$30c_1 + 20c_2 + 16c_3 + 12c_4$ $+8c_5 + 6c_6 + 4c_7 + 2c_8$	236580	221900	211200	237130	**0.6**	257100	268800	211200	**RPS**
2	$20c_1 + 18c_2 + 14c_3 + 12c_4$ $+10c_5 + 10c_6 + 6c_7 + 8c_8$	294030	291930	227450	212350	**0.8**	220450	211330	212350	**SRP**
3	$16c_1 + 14c_2 + 14c_3 + 14c_4$ $+8c_5 + 10c_6 + 10c_7 + 12c_8$	283340	232220	135590	128910	**0.8**	208130	130690	128910	**RSP**
4	$10c_1 + 12c_2 + 12c_3 + 12c_4$ $+12c_5 + 12c_6 + 14c_7 + 14c_8$	175680	84520	70780	70340	**0.8**	193220	71450	70340	**RSP**
5	$0c_1 + 14c_2 + 14c_3 + 14c_4$ $+14c_5 + 14c_6 + 14c_7 + 14c_8$	13120	13250	13310	13150	**0.2**	202240	12880	13120	**SRP**
6	$0c_1 + 10c_2 + 12c_3 + 14c_4$ $+14c_5 + 16c_6 + 16c_7 + 16c_8$	13100	13150	13150	13180	**0.2**	188760	13000	13100	**SRP**
7	$0c_1 + 4c_2 + 12c_3 + 14c_4$ $+14c_5 + 16c_6 + 18c_7 + 20c_8$	13150	13180	13290	13350	**0.2**	186420	12850	13150	**SRP**
8	$0c_1 + 0c_2 + 14c_3 + 16c_4$ $+16c_5 + 16c_6 + 16c_7 + 20c_8$	10770	10910	10790	10950	**0.2**	195160	10740	10770	**SRP**

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
9	$0c_1 + 0c_2 + 6c_3 + 16c_4$ $+18c_5 + 18c_6 + 20c_7 + 20c_8$	10820	10800	10920	10970	**0.4**	185060	10680	10800	**SRP**
10	$0c_1 + 0c_2 + 0c_3 + 18c_4$ $+20c_5 + 20c_6 + 20c_7 + 20c_8$	9350	9330	9370	9290	**0.8**	185770	9050	9290	**SRP**
11	$0c_1 + 0c_2 + 0c_3 + 10c_4$ $+20c_5 + 22c_6 + 22c_7 + 24c_8$	9250	9370	9620	9290	**0.2**	194240	9040	9250	**SRP**
12	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+12c_5 + 24c_6 + 28c_7 + 34c_8$	7790	7680	7660	7740	**0.6**	189150	7780	7660	**RSP**
13	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 32c_6 + 32c_7 + 34c_8$	7030	7020	7080	7070	**0.4**	175640	6820	7020	**SRP**
14	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 48c_7 + 50c_8$	6680	6660	6660	6720	**0.6**	176490	6500	6660	**SRP**
15	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 0c_7 + 98c_8$	6520	6590	6490	6490	**0.8**	177240	6420	6490	**SRP**

Table B.11: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 98 in the input text $T_{8_{\acute{U}}}$. The last column of the table represents the efficiency ranking of the algorithms.

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$60c_1 + 40c_2 + 32c_3 + 24c_4$ $+16c_5 + 12c_6 + 8c_7 + 4c_8$	235100	230640	225460	272640	**0.6**	235350	364370	225460	**RPS**
2	$40c_1 + 36c_2 + 28c_3 + 24c_4$ $+20c_5 + 20c_6 + 12c_7 + 16c_8$	308590	308620	254320	233260	**0.8**	201840	238900	233260	**PRS**
3	$32c_1 + 28c_2 + 28c_3 + 28c_4$ $+16c_5 + 20c_6 + 20c_7 + 24c_8$	319380	302840	150510	142290	**0.8**	199100	142120	142290	**SRP**
4	$20c_1 + 24c_2 + 24c_3 + 24c_4$ $+24c_5 + 24c_6 + 28c_7 + 28c_8$	169810	73980	66030	71950	**0.6**	193250	70400	66030	**RSP**
5	$0c_1 + 28c_2 + 28c_3 + 28c_4$ $+28c_5 + 28c_6 + 28c_7 + 28c_8$	7150	7240	8360	8370	**0.2**	219650	7000	7150	**SRP**
6	$0c_1 + 20c_2 + 24c_3 + 28c_4$ $+28c_5 + 32c_6 + 32c_7 + 32c_8$	8420	8400	8330	8420	**0.6**	216350	8140	8330	**SRP**
7	$0c_1 + 8c_2 + 24c_3 + 28c_4$ $+28c_5 + 32c_6 + 36c_7 + 40c_8$	8410	8400	8410	8330	**0.8**	214530	8090	8330	**SRP**
8	$0c_1 + 0c_2 + 28c_3 + 32c_4$ $+32c_5 + 32c_6 + 32c_7 + 40c_8$	6580	6630	6640	6630	**0.2**	216070	6450	6580	**SRP**

#	The Pattern	**CPU Time (μ sec)**				**Best**	**CPU Time (μ sec)**			**Algo**
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
9	$0c_1 + 0c_2 + 12c_3 + 32c_4$ $+36c_5 + 36c_6 + 40c_7 + 40c_8$	6650	6640	6630	6600	**0.8**	214230	6480	6600	**SRP**
10	$0c_1 + 0c_2 + 0c_3 + 36c_4$ $+40c_5 + 40c_6 + 40c_7 + 40c_8$	5390	5360	5180	5230	**0.6**	209000	5220	5180	**RSP**
11	$0c_1 + 0c_2 + 0c_3 + 20c_4$ $+40c_5 + 44c_6 + 44c_7 + 48c_8$	5310	5320	5310	5320	**0.6**	216700	5130	5310	**SRP**
12	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+24c_5 + 48c_6 + 56c_7 + 68c_8$	4430	4440	4440	4440	**0.2**	213990	4360	4430	**SRP**
13	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 64c_6 + 64c_7 + 68c_8$	3980	4020	4000	4000	**0.2**	200830	3900	3980	**SRP**
14	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 96c_7 + 100c_8$	3730	3720	3720	3720	**0.8**	213060	3610	3720	**SRP**
15	$0c_1 + 0c_2 + 0c_3 + 0c_4$ $+0c_5 + 0c_6 + 0c_7 + 196c_8$	3560	3560	3570	3550	**0.8**	208130	3480	3550	**SRP**

Table B.12: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 196 in the input text $T_{8_{\acute{U}}}$. The last column of the table represents the efficiency ranking of the algorithms.

B.5 Comparison of Different Values of ϵ for the Input Text T_{Real}

The input text T_{Real} is defined over English alphabet (i.e. $\sigma = 26$). Moreover, the text is not randomly generated, rather it is a real text comprising of a collection of the plays of famous English writer William Shakespeare. We removed all the punctuation marks and white spaces from the text to limit the character set to English alphabets only (the uppercase characters were also transformed into lowercase). The text comprises of 3712565 characters, i.e. $n := |T_{Real}| = 3712565$.

B.5.1 Comparison of Different Values of ϵ for the Input Text T_{Real} for Finding the Abelian Patterns Corresponding to Different Substrings of T_{Real}

In the following sections, we present the results of the experiments performed on the input text T_{Real} for finding the abelian patterns corresponding to different substrings of T_{Real}.

Abelian Pattern Matching for $m = 5$ in T_{Real}

Table B.13 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

The difference between the CPU time for different values of ϵ was not significant. The parameterized algorithm was the most efficient algorithm for all but one of the patterns (in pattern 26, the parameterized algorithm was the second in the efficiency ranking of the algorithms). There was not a big difference between the CPU time taken by the parameterized algorithm and the suffix based algorithm for most of the patterns.

Abelian Pattern Matching for $m = 10$ in T_{Real}

Table B.14 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

The difference between the CPU time for different values of ϵ was not significant. Moreover, the difference between the CPU time taken by the parameterized algorithm and the suffix based algorithm was also not significant. The prefix based algorithm was the slowest, whereas the parameterized algorithm and the prefix based algorithm had almost same efficiency.

Abelian Pattern Matching for $m = 20$ in T_{Real}

Table B.15 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

The difference between the CPU time for different values of ϵ was not significant for most of the patterns. The values $\epsilon = 0.6$ and $\epsilon = 0.8$ were the promising values for ϵ. The prefix based algorithm was the slowest, whereas the parameterized algorithm and the prefix based algorithm had almost same efficiency.

Abelian Pattern Matching for $m = 50$ in T_{Real}

Table B.16 shows the CPU time taken by the parameterized suffix based algorithm for different values of ϵ along with the efficiency ranking of the three algorithms.

The difference between the CPU time for different values of ϵ was not significant for $\epsilon = 0.4$, $\epsilon = 0.6$ and $\epsilon = 0.8$. The values $\epsilon = 0.6$ and $\epsilon = 0.8$ were the promising values for ϵ. The prefix based algorithm was the slowest, whereas the parameterized algorithm and the prefix based algorithm had almost same efficiency.

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$2e + 1h + 1r + 1w$	35720	34330	33130	33010	**0.8**	133740	35680	33010	**RSP**
2	$1e + 1h + 1m + 1o + 1t$	41720	39690	39630	38860	**0.8**	128430	39850	38860	**RSP**
3	$1d + 1l + 1o + 1r + 1y$	31020	30370	30210	30840	**0.6**	119550	32770	30210	**RSP**
4	$1e + 1f + 1o + 2r$	33910	32470	33390	32460	**0.8**	125880	35420	32460	**RSP**
5	$1e + 1m + 1o + 1r + 1s$	36930	35080	35050	35400	**0.6**	125650	37460	35050	**RSP**
6	$1g + 2h + 1o + 1u$	27620	26700	27270	26230	**0.8**	119470	29460	26230	**RSP**
7	$1c + 1i + 1n + 1p + 1r$	27660	27560	27080	26980	**0.8**	114640	29140	26980	**RSP**
8	$1a + 1d + 1e + 1m + 1r$	32890	33520	32090	32350	**0.6**	114340	33500	32090	**RSP**
9	$1a + 1d + 1e + 1s + 1t$	35580	33700	33390	34070	**0.6**	119320	34720	33390	**RSP**
10	$1e + 1f + 1l + 1s + 1y$	29290	28650	27680	28590	**0.6**	112900	30830	27680	**RSP**
11	$1a + 1e + 1i + 1m + 1r$	33390	31890	33240	32700	**0.4**	117690	32790	31890	**RSP**
12	$2e + 1m + 1n + 1t$	31190	30290	29480	30260	**0.6**	126460	31580	29480	**RSP**
13	$1e + 1f + 1g + 1o + 1r$	30040	29480	29260	29400	**0.6**	118130	30650	29260	**RSP**

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
14	$1d + 1e + 1l + 1n + 1o$	31840	30400	30430	30870	**0.4**	110970	31290	30400	**RSP**
15	$2a + 2d + 1n$	25040	25540	25210	25260	**0.2**	113150	27730	25040	**RSP**
16	$1c + 1e + 1i + 1l + 1r$	29760	28530	28850	29490	**0.4**	115270	30200	28530	**RSP**
17	$1a + 1h + 1n + 1o + 1w$	29720	29330	29670	29180	**0.8**	117950	30880	29180	**RSP**
18	$2e + 1m + 1n + 1s$	28270	28040	27450	27840	**0.6**	119950	28950	27450	**RSP**
19	$1a + 1d + 1i + 1s + 1y$	27160	27360	27090	27020	**0.8**	118610	29020	27020	**RSP**
20	$1c + 1i + 1l + 1o + 1u$	27090	25770	25450	25580	**0.6**	115190	28750	25450	**RSP**
21	$1a + 1d + 1o + 1s + 1t$	30080	29300	28630	28810	**0.6**	112080	29540	28630	**RSP**
22	$1a + 1e + 1l + 1t + 1w$	28930	28130	27920	28540	**0.6**	108540	29180	27920	**RSP**
23	$1o + 1r + 2t + 1u$	25940	25490	25210	26060	**0.6**	111670	27450	25210	**RSP**
24	$1o + 2p + 1r + 1u$	23960	23730	23360	23630	**0.6**	105030	24620	23360	**RSP**
25	$1e + 1h + 1i + 1k + 1s$	28990	28080	28070	28270	**0.6**	110120	29390	28070	**RSP**
26	$1e + 1i + 2n + 1s$	28640	30740	29190	29010	**0.2**	113350	28390	28640	**SRP**

#	The Pattern	**CPU Time (μ sec)**				**Best**	**CPU Time (μ sec)**			**Algo**
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
27	$1a + 1e + 1l + 1m + 1y$	30090	27460	28090	29000	**0.4**	117560	28930	27460	**RSP**
28	$1h + 1i + 2s + 1w$	23630	23230	23520	23450	**0.4**	120480	28030	23230	**RSP**
29	$1e + 1i + 1s + 1v + 1w$	25580	24960	24390	24810	**0.6**	104780	26200	24390	**RSP**
30	$1b + 2e + 1f + 1r$	22550	22160	22150	24080	**0.6**	107170	24350	22150	**RSP**
31	$1h + 1i + 1s + 1t + 1v$	28820	27730	27370	28080	**0.6**	117340	29470	27370	**RSP**
32	$1a + 1d + 1s + 1t + 1w$	26480	26480	26380	26210	**0.8**	118080	27610	26210	**RSP**
33	$1a + 1b + 1l + 1m + 1o$	23970	23780	23380	23730	**0.6**	112660	25120	23380	**RSP**
34	$1l + 1m + 1o + 1p + 1u$	22570	22320	22270	22090	**0.8**	112150	24490	22090	**RSP**
35	$2c + 1k + 1n + 1o$	21160	21550	21050	21170	**0.6**	108590	23780	21050	**RSP**
36	$1a + 1r + 1s + 2y$	23290	23430	23210	23030	**0.8**	116570	25110	23030	**RSP**
37	$2a + 1c + 1m + 1p$	19790	20000	20210	20720	**0.2**	111890	23440	19790	**RSP**

Table B.13: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 5 in the input text T_{Real}. The last column of the table represents the efficiency ranking of the algorithms.

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$1c + 3e + 1g + 1l + 1o + 1s + 1t + 1u$	17560	17810	17370	17110	**0.8**	132500	17790	17110	**RSP**
2	$2e + 1h + 1i + 1o + 1r + 1s + 2t + 1w$	24440	23000	22690	22360	**0.8**	137400	22770	22360	**RSP**
3	$1a+1e+2h+2l+2o+1t+1w$	19490	18720	18800	18630	**0.8**	132650	18470	18630	**SRP**
4	$2d + 3e + 1h + 2i + 1n + 1s$	17320	16600	16690	16530	**0.8**	130110	16540	16530	**RSP**
5	$2a + 1d + 1e + 1h + 1o + 1r + 1t + 1u + 1y$	23700	22350	21680	21410	**0.8**	133850	21660	21410	**RSP**
6	$1a + 2e + 1h + 1n + 1o + 1r + 1t + 1v + 1y$	23060	21700	21150	21230	**0.6**	131210	21220	21150	**RSP**
7	$2e+1h+1l+1o+1r+2s+2t$	20160	19370	19270	19000	**0.8**	133260	19020	19000	**RSP**
8	$1e + 1h + 1i + 1l + 1o + 1s + 2t + 1u + 1y$	23090	21810	22220	21310	**0.8**	132240	20280	21310	**SRP**
9	$1a+1f+1i+1m+1n+2o+1r + 2t$	19810	18690	18700	18730	**0.4**	136720	17700	18690	**SRP**
10	$2e + 1h + 1i + 1l + 1o + 1r + 2t + 1v$	21380	20080	20390	20280	**0.4**	134080	19590	20080	**SRP**

#	The Pattern	CPU Time (μ sec) $\epsilon = 0.2$	CPU Time (μ sec) $\epsilon = 0.4$	CPU Time (μ sec) $\epsilon = 0.6$	CPU Time (μ sec) $\epsilon = 0.8$	Best ϵ	CPU Time (μ sec) Pre	CPU Time (μ sec) Suf	CPU Time (μ sec) paRa	Algo Ranking
11	$3e+1g+1h+1i+1m+1o+1t+1v$	17160	16910	16460	16610	**0.6**	131680	16620	16460	**RSP**
12	$2a+1b+2e+1h+1l+2r+1t$	19000	18450	18180	18050	**0.8**	131110	17970	18050	**SRP**
13	$3e+1h+1i+1k+1l+1m+1s+1t$	19040	18540	18820	18580	**0.4**	131030	18320	18540	**SRP**
14	$1a+1e+1i+1l+1n+1o+1r+1s+1u+1v$	23800	22520	22410	22050	**0.8**	133790	21670	22050	**SRP**
15	$4e+1h+1n+1o+1r+1t+1v$	19240	18950	19300	18870	**0.8**	132510	18600	18870	**SRP**
16	$2e+1h+1i+1r+2t+1u+1v+1y$	18770	18020	17710	17660	**0.8**	133400	18000	17660	**RSP**
17	$1e+1f+2n+3o+1s+1u+1y$	15810	15650	15340	15370	**0.6**	126930	15300	15340	**SRP**
18	$1d+1e+1n+3o+2s+1t+1u$	18290	18140	17760	17920	**0.6**	129710	17950	17760	**RSP**
19	$1b+1d+2e+1l+1m+1o+1r+1u+1v$	18160	17110	17110	17030	**0.8**	129850	17100	17030	**RSP**
20	$3e+1h+1i+2o+1p+1s+1t$	19060	18610	18150	18300	**0.6**	129270	18160	18150	**RSP**

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
21	$1h+1i+1n+1o+2r+1s+2t+1u$	21040	20120	20120	20500	**0.6**	130010	19860	20120	**SRP**
22	$1c+1d+2e+1h+1i+1l+1p+1r+1s$	19940	19650	18800	18830	**0.6**	134800	18850	18800	**RSP**
23	$1h+2i+1n+1o+2s+1u+1w+1y$	16950	16520	16500	16550	**0.6**	133010	17240	16500	**RSP**
24	$1a+1b+1e+1m+1n+1o+1r+1t+1u+1w$	22030	21160	21220	21230	**0.4**	131030	21010	21160	**SRP**
25	$1a+2d+2e+1m+1s+2t+1y$	17760	16840	16730	16390	**0.8**	128820	17160	16390	**RSP**
26	$1a+1d+1e+1g+1h+1l+1n+1o+2r$	21030	20590	19930	20980	**0.6**	130240	19670	19930	**SRP**
27	$1d+1h+2i+1p+2r+1s+2t$	15790	15830	15820	15710	**0.8**	126390	15950	15710	**RSP**
28	$1a+1h+3i+1o+2s+2t$	17700	17550	17240	17140	**0.8**	125640	17820	17140	**RSP**
29	$2a+1c+1e+2m+1n+1r+2u$	15750	15830	16130	15980	**0.2**	125830	15690	15750	**SRP**
30	$1a+3e+1f+1h+1q+1r+1t+1u$	18740	18590	18720	18010	**0.8**	132890	18270	18010	**RSP**

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
31	$1a+1c+2d+1e+1h+1o+2t+1y$	19540	18870	19190	19220	**0.4**	132830	19030	18870	**RSP**
32	$1b+2e+1f+1i+2o+2r+1s$	16900	16690	16400	16430	**0.6**	133060	16610	16400	**RSP**
33	$1b+1e+2o+1p+1r+2t+1u+1y$	17190	17680	17210	16850	**0.8**	127680	17350	16850	**RSP**
34	$1b+2d+1e+2h+1i+1l+1o+1t$	18720	18120	18390	18290	**0.4**	128170	18040	18120	**SRP**
35	$1a+2d+2h+1m+1n+1r+1t+1u$	17950	17060	17160	17000	**0.8**	129780	17140	17000	**RSP**
36	$2e+2i+2l+1p+2t+1y$	13930	14000	14200	14070	**0.2**	122900	14360	13930	**RSP**
37	$1e+2g+1h+1i+1l+1m+1o+1t+1y$	19520	19360	18680	18720	**0.6**	125920	18770	18680	**RSP**

Table B.14: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 10 in the input text T_{Real}. The last column of the table represents the efficiency ranking of the algorithms.

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$5a+1c+1d+1i+1k+1m+3n+2o+1r+2t+1v+1y$	11550	11290	11440	11520	**0.4**	124640	11300	11290	**RSP**
2	$2a+1d+1e+3h+2i+1m+1n+1o+2s+5t+1y$	15910	14790	14970	14610	**0.8**	128500	15160	14610	**RSP**
3	$2a+1b+1c+1d+3e+3h+1i+1o+1r+4t+2u$	14570	13110	12880	13140	**0.6**	128790	12660	12880	**SRP**
4	$2c+2e+1f+2h+3i+1l+1m+1n+3o+3t+1w$	12740	12130	12290	12500	**0.4**	125530	11940	12130	**SRP**
5	$3a+2b+2d+3e+2i+1l+2m+2n+2s+1t$	12520	12240	12360	11770	**0.8**	121480	11680	11770	**SRP**
6	$2a+1c+2e+1f+1g+1h+3i+2k+2n+2r+1t+1v+1y$	14320	13170	12980	13050	**0.6**	125550	12720	12980	**SRP**
7	$1a+1b+2d+3e+1g+1h+2n+3o+1p+3t+1u+1w$	13280	12680	12900	12650	**0.8**	125050	12370	12650	**SRP**
8	$2a+2c+2i+2l+3o+5s+3u+1w$	8630	8360	8560	8530	**0.4**	113250	8470	8360	**RSP**
9	$4a+1b+1c+2e+1f+1h+3l+1n+1o+1p+1t+1u+1w+1y$	14470	14000	13390	13820	**0.6**	127210	12810	13390	**SRP**
10	$2a+1c+1d+3e+1g+3h+4l+1o+1t+1u+1v+1w$	12460	11720	11790	11840	**0.4**	125690	11750	11720	**RSP**

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
11	$1b + 1c + 2h + 1i + 1l + 2n + 3o + 1p + 2s + 3u + 3w$	9650	9810	9740	9670	**0.2**	120450	9450	9650	**SRP**
12	$5a + 2c + 1d + 1g + 1h + 1j + 2k + 1n + 1o + 1r + 2s + 1u + 1w$	11090	10890	11020	10680	**0.8**	122450	11000	10680	**RSP**
13	$4a + 1b + 3d + 2e + 1h + 2i + 1m + 1n + 2r + 1s + 1t + 1y$	16560	15290	14720	14980	**0.6**	128960	14650	14720	**SRP**
14	$6e + 1i + 1l + 1m + 2n + 1r + 4s + 3t + 1y$	10910	11360	11160	10900	**0.8**	117490	10400	10900	**SRP**
15	$1a + 1b + 5e + 1g + 1h + 3i + 1l + 1p + 2r + 1s + 3t$	13820	12660	12550	12510	**0.8**	119780	12280	12510	**SRP**
16	$2a + 2e + 1f + 2n + 5o + 1r + 5t + 1u + 1y$	10810	10940	10970	10610	**0.8**	118850	10940	10610	**RSP**
17	$3a + 1d + 2e + 2h + 1i + 1k + 1n + 2o + 1s + 3t + 1u + 1v + 1y$	17360	15470	15190	15080	**0.8**	129250	14920	15080	**SRP**
18	$2a + 1b + 1d + 2e + 2i + 3l + 1n + 1o + 1p + 1r + 3t + 1u + 1w$	15670	14920	15000	14460	**0.8**	132640	14440	14460	**SRP**
19	$2a + 1b + 1c + 3e + 1f + 1h + 1i + 1k + 2l + 1m + 3s + 2t + 1x$	14420	13260	12690	12700	**0.6**	128950	12440	12690	**SRP**
20	$5e + 1f + 1h + 1i + 1n + 1o + 1r + 4s + 3t + 1u + 1w$	15100	14580	15210	14940	**0.4**	125240	14740	14580	**RSP**

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
21	$1d+2e+2f+1g+1h+1i+4o+2r+1s+2t+3u$	14890	13320	13210	13490	**0.6**	128430	12900	13210	**SRP**
22	$2a+1b+3e+1g+1h+2i+3l+1m+1n+1o+1r+2s+1z$	14920	14720	14230	14470	**0.6**	133390	13790	14230	**SRP**
23	$4a+1d+1e+1g+1l+2n+2o+2r+3s+2u+1w$	12590	12140	12210	11990	**0.8**	122110	11660	11990	**SRP**
24	$3a+3c+3e+1h+1i+4n+1o+1r+3t$	13140	12250	12330	12320	**0.4**	121270	12420	12250	**RSP**
25	$1b+1d+5e+1g+1h+1i+2m+1n+2r+1s+3t+1w$	15090	13110	13120	13180	**0.4**	121880	12640	13110	**SRP**
26	$4a+2d+3e+1h+2n+2s+2t+1v+1w+2y$	12540	11820	11840	11660	**0.8**	120480	11480	11660	**SRP**
27	$2a+1b+1c+1d+2e+1f+1m+3o+2r+3s+1t+1x+1y$	13540	12910	12640	12530	**0.8**	127600	12150	12530	**SRP**
28	$1c+3e+1f+1h+2i+1k+1l+3n+1r+2s+1t+2u+1y$	14880	13610	13310	13590	**0.6**	129360	13110	13310	**SRP**
29	$3a+1d+1e+1f+1h+1i+2m+1n+1p+2r+1s+2t+1u+2y$	18190	15730	15860	16310	**0.4**	131090	15260	15730	**SRP**
30	$2a+1b+1d+1e+1f+1g+1i+2n+2o+2r+2s+1t+2u+1w$	18220	15960	16110	16180	**0.4**	127520	16050	15960	**RSP**

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
31	$2a+1b+1c+1d+3e+2f+1h+1l+1n+3o+3t+1u$	14320	13030	12900	12920	**0.6**	126240	12600	12900	**SRP**
32	$1a+1b+1c+1d+1e+1g+1h+3i+3n+1o+1r+2s+1t+2u$	20310	18120	16880	17690	**0.6**	126270	17330	16880	**RSP**
33	$1a+1b+3e+2h+1i+1n+1o+4r+2s+2u+1w+1y$	14430	12930	12820	12760	**0.8**	124120	12750	12760	**SRP**
34	$1c+2e+2h+1i+1l+2m+2o+1p+1r+3s+1t+1u+1x+1y$	15810	14540	14310	14530	**0.6**	125960	14370	14310	**RSP**
35	$3a+1e+1f+2h+1i+2l+2o+1p+1r+2u+1v+3y$	11850	11350	11200	11160	**0.8**	119660	10850	11160	**SRP**
36	$1d+2e+1f+2i+1j+1l+1n+2o+3r+2s+1t+1u+2y$	14850	14740	15150	14650	**0.8**	127800	14480	14650	**SRP**
37	$2a+2d+2e+1f+1h+1i+3l+2n+2r+1s+1t+1u+1w$	18700	16350	16360	15980	**0.8**	128080	15530	15980	**SRP**

Table B.15: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 20 in the input text T_{Real}. The last column of the table represents the efficiency ranking of the algorithms.

#	The Pattern	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	$4a+3d+3e+1f+2h+3i+3l+2m+6n+6o+1s+6t+2u+2v+4w+2y$	11160	9500	9540	9360	**0.8**	104960	9130	9360	**SRP**
2	$5a+3b+1c+5d+4e+1f+2h+2i+2l+2m+3n+6o+1p+4r+3s+3t+2w+1y$	18330	14750	14350	13980	**0.8**	112940	13810	13980	**SRP**
3	$3a+1b+2c+7e+1f+3h+2i+1k+2l+1m+7n+7o+1p+4r+3s+2t+1u+2v$	12950	10970	10830	10760	**0.8**	110460	11040	10760	**RSP**
4	$6a+1b+5d+4e+1f+1g+2h+4i+2k+1l+1m+3n+5o+3r+1s+7t+1v+2w$	13170	11110	11050	10880	**0.8**	109830	11040	10880	**RSP**
5	$6a+1b+4d+4e+2g+1h+5i+4l+4m+6n+3o+3r+2s+4t+1y$	11410	9730	9500	9500	**0.8**	105200	9320	9500	**SRP**
6	$6a+2b+1d+9e+1f+4h+1i+1k+3l+3n+3o+3r+4s+6t+1u+1w+1y$	12480	10940	10580	10890	**0.6**	110160	11260	10580	**RSP**
7	$3a+1b+1c+3d+6e+1f+3g+3i+4l+2n+6o+3r+4s+8t+1u+1w$	9680	9010	9270	8960	**0.8**	109900	9090	8960	**RSP**
8	$4a+1b+3c+2d+6e+2h+3i+2l+1m+1n+4o+3p+7r+3s+2t+2u+1w+3y$	17420	13990	12900	13540	**0.6**	110000	13220	12900	**RSP**

#	The Pattern	CPU Time (μ sec)				**Best**	CPU Time (μ sec)			**Algo**
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
9	$7a+1b+1c+3e+1g+2h+2k+2l+1m+3n+7o+6r+1s+3t+3u+2w+5y$	9800	8890	9040	8830	**0.8**	101940	9010	8830	**RSP**
10	$2a+1c+1d+5e+3f+1g+3h+3i+2l+3n+5o+1q+3r+5s+5t+5u+1w+1y$	15600	13660	12720	12750	**0.6**	115010	12280	12720	**SRP**
11	$1a+1b+1c+2d+9e+1f+1g+1h+6i+1k+1m+3n+2o+2p+3r+8s+5t+1u+1v$	13080	10700	11120	10920	**0.4**	105700	11460	10700	**RSP**
12	$5a+1b+1d+7e+1f+2h+4i+7l+1m+3o+2r+8s+3t+3w+2y$	9450	8900	8750	8890	**0.6**	106210	8770	8750	**RSP**
13	$6a+2b+3d+4e+1f+1g+2h+8i+3m+4n+1o+1p+2r+4s+6t+2u$	11090	9930	9640	9660	**0.6**	106620	9540	9640	**SRP**
14	$7a+2b+3c+3d+3e+4h+4i+2k+1l+1m+3n+2o+4r+3s+6t+1u+1v$	12560	11040	10720	11070	**0.6**	109360	11120	10720	**RSP**
15	$4a+1b+3c+2d+4e+5h+2l+6m+2n+5o+2r+4s+3t+4w+3y$	9830	8880	8980	9110	**0.4**	107880	8800	8880	**SRP**
16	$4a+3c+2d+6e+5h+3i+1l+3n+4o+5r+3s+6t+3u+1w+1y$	11530	10690	10320	10310	**0.8**	115110	9780	10310	**SRP**

#	The Pattern	**CPU Time** (μ **sec**)				**Best**	**CPU Time** (μ **sec**)			**Algo**
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
17	$5a+2b+1d+6e+1f+1g+2h+3i+1k+2l+4n+4o+3r+7s+4t+1u+2w+1y$	15270	13000	12570	12380	**0.8**	113500	12350	12380	**SRP**
18	$1a+1c+4e+1f+2g+5h+5i+1j+2n+5o+2r+5s+10t+4u+2y$	8660	8300	8250	8150	**0.8**	99770	8120	8150	**SRP**
19	$3a+1b+8e+1f+8h+5i+3l+1m+1p+4r+6s+5t+3w+1y$	7670	7470	7460	7440	**0.8**	101570	7460	7440	**RSP**
20	$8a+1d+5e+1g+3i+4l+8n+7o+5r+2s+3t+1y+2z$	7590	7450	7420	7350	**0.8**	100520	7250	7350	**SRP**
21	$2a+2b+8e+2g+4h+7i+3l+5n+5o+1p+3r+1s+4t+1u+1v+1x$	9130	8410	8560	8390	**0.8**	105140	8280	8390	**SRP**
22	$3a+1c+1d+8e+1g+4h+4i+2k+2l+2m+3n+1o+2r+3s+6t+2u+2v+2w+1y$	18700	15230	14260	13540	**0.8**	113390	14220	13540	**RSP**
23	$5a+1b+1c+2d+3e+2g+3h+5i+1k+2l+2m+5n+1o+3r+2s+5t+1u+5w+1y$	18530	14310	15130	14650	**0.4**	108230	14010	14310	**SRP**
24	$4a+2b+1d+8e+1f+3h+2i+1k+2l+2m+5n+5o+3r+1s+6t+3w+1y$	12510	11010	11110	10930	**0.8**	112900	11260	10930	**RSP**

#	**The Pattern**	**CPU Time (μ sec)**				**Best**	**CPU Time (μ sec)**			**Algo**
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
25	$4a+1b+3c+1d+6e+2f+1g+4h+4i+2l+1m+4n+2o+2p+3r+5s+2t+2u+1v$	18720	15000	14620	14580	**0.8**	115810	14840	14580	**RSP**
26	$2a+1b+1d+11e+4h+1i+1l+3n+5o+1q+3r+7s+6t+2u+1v+1w$	9870	9180	9300	9410	**0.4**	107460	9280	9180	**RSP**
27	$3a+4d+6e+2f+1g+1h+5i+2l+1m+4n+7o+1p+4r+3s+3t+1v+1x+1y$	12490	10950	10600	11020	**0.6**	111360	11150	10600	**RSP**
28	$3a+1b+3c+10e+3g+3h+5i+1k+1l+1m+5n+1o+1p+2r+3s+5t+1u+1x$	10750	9710	9580	9460	**0.8**	103700	9530	9460	**RSP**
29	$4a+1b+2c+6d+6e+1f+4h+5i+2l+6n+3o+2p+3r+2s+2t+1w$	11420	10770	10810	10690	**0.8**	107040	10190	10690	**SRP**
30	$3a+5e+4f+2g+6h+4i+2k+5n+3o+1p+5r+3s+6t+1y$	8250	8050	8020	8030	**0.6**	103930	7960	8020	**SRP**
31	$2a+1b+1d+1e+1f+1h+2i+1k+3l+3n+11o+3p+3r+3s+3t+4u+2v+3w+2y$	12090	10670	10560	10510	**0.8**	102150	10360	10510	**SRP**
32	$2a+2c+3d+5e+3f+1g+2h+1i+7l+2m+2n+8o+4r+2s+2t+1u+2w+1y$	17490	13930	13250	13660	**0.6**	105830	13890	13250	**RSP**

#	The Pattern	**CPU Time (μ sec)**				**Best**	**CPU Time (μ sec)**			**Algo**
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
33	$4a+1b+1d+7e+1f+1g+5h+5i+4l+3m+2n+2o+1q+2r+3s+6t+1u+1y$	14990	12450	12070	11920	**0.8**	112380	12310	11920	**RSP**
34	$3a+1b+3c+6e+1f+1g+2h+1i+1l+1m+6n+4o+4r+4s+5t+4u+3y$	12410	10580	11600	11740	**0.4**	113490	11070	10580	**RSP**
35	$5a+1b+1c+1d+2e+1f+1g+4h+6i+4l+2m+3n+6o+2s+6t+1v+3w+1y$	11250	10550	10670	10500	**0.8**	104800	9970	10500	**SRP**
36	$5a+3d+6e+1f+1g+3h+3i+2l+1m+4n+1o+1p+3r+7s+3t+1u+1v+2w+2y$	18340	14670	15020	14880	**0.4**	111940	14660	14670	**SRP**
37	$3a+2c+8e+2f+2h+4i+1k+2l+2m+7n+5o+3p+1s+5t+1v+1x+1y$	8650	8250	8090	8250	**0.6**	104150	8000	8090	**SRP**

Table B.16: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns of length 50 in the input text T_{Real}. The last column of the table represents the efficiency ranking of the algorithms.

B.5.2 Comparison of Different Values of ϵ for the Input Text T_{Real} for Finding the Abelian Patterns Corresponding to Frequently Used English Words

In this section, we present the results of the experiments performed on the input text T_{Real} for finding the abelian patterns corresponding to frequently used English words.

Tables $B.17 - B.21$ show the CPU time taken by the parameterized suffix based algorithm for different values of ϵ, to find the matches of the abelian patterns corresponding to commonly used English words of lengths $5 - 11$. The efficiency ranking of the three algorithms is also presented in the tables.

From the tables it is observed that their was not a significant difference between the CPU time taken by the parameterized suffix based algorithm for different values of ϵ. However, for some patterns, the algorithm took slightly more time for $\epsilon = 0.2$ than for other values of ϵ.

The suffix based algorithm was the most efficient algorithm. The parameterized algorithm was the second in the efficiency ranking of the algorithms, however, there was not a significant difference between the CPU time taken by the parameterized algorithm and the suffix based algorithm for most of the patterns. The prefix based algorithm was the slowest algorithm and there was a big difference between the CPU time taken by the prefix based algorithm and the other two algorithms.

#	Word	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	three	48190	46690	47090	46970	**0.4**	138960	46820	46690	**RSP**
2	there	48250	46750	46510	46030	**0.8**	144080	45480	46030	**SRP**
3	earth	52270	50500	49900	50090	**0.6**	136790	48680	49900	**SRP**
4	heart	52850	50850	50520	50480	**0.8**	140340	49120	50480	**SRP**
5	other	48620	46420	46230	46420	**0.6**	133890	46110	46230	**SRP**
6	their	43740	41960	41680	41650	**0.8**	128520	40580	41650	**SRP**
7	these	39720	38130	38150	38300	**0.4**	133470	36720	38130	**SRP**
8	night	36960	36410	36080	36720	**0.6**	127470	33780	36080	**SRP**
9	thing	37610	36450	36260	36630	**0.6**	128850	34670	36260	**SRP**
10	shall	32180	31900	31750	31880	**0.6**	128350	29290	31750	**SRP**
11	stand	36090	34800	34550	34800	**0.6**	125050	33710	34550	**SRP**
12	where	33950	32520	33620	33190	**0.4**	127180	30290	32520	**SRP**
13	those	41340	40320	39890	39340	**0.8**	125910	38710	39340	**SRP**
14	youre	34630	33940	34130	36660	**0.4**	123420	32040	33940	**SRP**
15	death	38400	38990	39400	39180	**0.2**	124170	35480	38400	**SRP**

#	**Word**	**CPU Time (μ sec)**				**Best**	**CPU Time (μ sec)**			**Algo**
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
16	build	23600	23430	23680	23400	**0.8**	117090	22010	23400	**SRP**
17	human	26480	25930	26210	25170	**0.8**	121530	23900	25170	**SRP**
18	enjoy	26400	26340	25670	25540	**0.8**	114950	24520	25540	**SRP**
19	crops	24640	24770	24310	24210	**0.8**	115950	22990	24210	**SRP**
20	class	23350	23320	22990	23600	**0.6**	115230	21520	22990	**SRP**
21	greek	23350	23680	22860	23130	**0.6**	116860	21980	22860	**SRP**
22	grass	24220	23290	23650	23670	**0.4**	117960	22000	23290	**SRP**
23	cells	24990	24930	24590	25010	**0.6**	113530	23500	24590	**SRP**
24	color	23310	23850	23110	23320	**0.6**	117350	22000	23110	**SRP**
25	didnt	25620	24920	25020	25090	**0.4**	119380	23650	24920	**SRP**
26	block	21320	21410	21590	21240	**0.8**	108520	19500	21240	**SRP**
27	track	24910	25620	24900	25330	**0.6**	117680	22730	24900	**SRP**
28	group	25260	24090	24070	23950	**0.8**	115800	22110	23950	**SRP**
29	major	25160	24730	24650	25000	**0.6**	115940	23230	24650	**SRP**
30	check	23410	23340	23370	23720	**0.4**	111180	21250	23340	**SRP**

Table B.17: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns corresponding to English words of length 5 in the input text T_{Real}. The last column of the table represents the efficiency ranking of the algorithms.

#	Word	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	though	28920	28240	28310	27780	**0.8**	132180	26460	27780	**SRP**
2	mother	36120	34520	34130	33940	**0.8**	131390	32800	33940	**SRP**
3	either	33900	32820	32660	31890	**0.8**	133330	30590	31890	**SRP**
4	father	35940	34240	34130	33890	**0.8**	129870	32960	33890	**SRP**
5	should	28440	27270	27350	26420	**0.8**	130320	26090	26420	**SRP**
6	sister	31900	30190	30240	29850	**0.8**	131730	28580	29850	**SRP**
7	stream	33680	31970	32170	31750	**0.8**	126200	29680	31750	**SRP**
8	resent	31560	29920	30750	30210	**0.4**	133270	28480	29920	**SRP**
9	before	26480	24720	24800	24840	**0.4**	128550	23970	24720	**SRP**
10	reason	33130	31600	31060	30930	**0.8**	125400	29300	30930	**SRP**

#	**Word**	**CPU Time (μ sec)**				**Best**	**CPU Time (μ sec)**			**Algo**
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
11	family	21380	20910	21350	20570	**0.8**	123460	20020	20570	**SRP**
12	picked	22060	21530	21670	21980	**0.4**	122520	20560	21530	**SRP**
13	valley	23150	22670	22650	21930	**0.8**	126240	20650	21930	**SRP**
14	joined	27040	25410	25230	24140	**0.8**	127540	23970	24140	**SRP**
15	pulled	20950	20290	20160	20670	**0.6**	118920	19080	20160	**SRP**
16	slowly	20520	20210	20170	20800	**0.6**	121530	18770	20170	**SRP**
17	plural	20480	20300	20460	20300	**0.8**	123730	18400	20300	**SRP**
18	pushed	24620	23670	23760	23360	**0.8**	121940	22220	23360	**SRP**
19	rhythm	21530	20550	20880	20770	**0.4**	127270	19970	20550	**SRP**
20	column	22230	21600	21970	21590	**0.8**	124680	19940	21590	**SRP**

Table B.18: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns corresponding to English words of length 6 in the input text T_{Real}. The last column of the table represents the efficiency ranking of the algorithms.

#	Word	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	another	37850	35140	34370	33530	**0.8**	132600	32990	33530	**SRP**
2	thought	24660	21730	21640	21510	**0.8**	132310	20500	21510	**SRP**
3	brother	27560	26620	26030	26060	**0.6**	128900	24990	26030	**SRP**
4	nothing	25880	25460	24580	24500	**0.8**	134630	23410	24500	**SRP**
5	weather	29070	28290	27450	27560	**0.6**	134350	26520	27450	**SRP**
6	against	24950	24530	24640	24390	**0.8**	132910	22760	24390	**SRP**
7	friends	27900	26760	26200	25950	**0.8**	131950	25040	25950	**SRP**
8	through	23680	23570	22940	22560	**0.8**	131370	21670	22560	**SRP**
9	without	24630	23700	23170	23450	**0.6**	132580	22060	23170	**SRP**
10	strange	30530	28100	28180	28200	**0.4**	130950	26930	28100	**SRP**

#	Word	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
11	farmers	23030	21510	21680	21760	**0.4**	128800	20750	21510	**SRP**
12	finally	19940	19120	18970	19070	**0.6**	129410	18400	18970	**SRP**
13	exactly	21560	20740	20680	21350	**0.6**	123480	19080	20680	**SRP**
14	decided	18040	16840	17250	17130	**0.4**	125130	16180	16840	**SRP**
15	symbols	18070	18040	17990	17950	**0.8**	127340	16590	17950	**SRP**
16	usually	17960	17880	17510	17250	**0.8**	126600	16670	17250	**SRP**
17	century	23010	22270	22070	22320	**0.6**	127280	20860	22070	**SRP**
18	climbed	21050	20180	20000	19540	**0.8**	128250	19450	19540	**SRP**
19	problem	22220	21610	21230	21240	**0.6**	124220	19880	21230	**SRP**
20	explain	22630	21340	21280	21300	**0.6**	125050	20400	21280	**SRP**

Table B.19: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns corresponding to English words of length 7 in the input text T_{Real}. The last column of the table represents the efficiency ranking of the algorithms.

#	Word	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	together	26030	25480	24450	24630	**0.6**	134230	23150	24450	**SRP**
2	yourself	26050	24030	24700	24810	**0.4**	137430	23150	24030	**SRP**
3	soldiers	24330	23400	22470	23040	**0.6**	137310	21720	22470	**SRP**
4	business	19760	19310	19390	19650	**0.4**	130690	18400	19310	**SRP**
5	remember	16670	16160	15530	15770	**0.6**	125540	15010	15530	**SRP**
6	shoulder	25960	24050	24330	24390	**0.4**	132740	22950	24050	**SRP**
7	straight	25180	23490	22840	23630	**0.6**	137080	22120	22840	**SRP**
8	anything	22360	21170	20800	21020	**0.6**	132610	19910	20800	**SRP**
9	southern	32590	28510	27720	27830	**0.6**	128580	27020	27720	**SRP**
10	consider	27830	25570	25170	25410	**0.6**	131470	23440	25170	**SRP**

#	Word	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
		$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
11	students	21100	19990	19970	20170	**0.6**	133970	19020	19970	**SRP**
12	equation	24910	22360	23080	22470	**0.4**	130460	22160	22360	**SRP**
13	happened	20650	19560	19250	19320	**0.6**	130950	18230	19250	**SRP**
14	products	21780	20350	20280	20170	**0.8**	133400	19060	20170	**SRP**
15	movement	20640	19660	20040	19880	**0.4**	130720	19100	19660	**SRP**
16	electric	20660	19660	19850	20000	**0.4**	135190	18720	19660	**SRP**
17	probably	18200	17830	17600	17560	**0.8**	131650	16740	17560	**SRP**
18	actually	16590	16300	16460	16070	**0.8**	131810	15260	16070	**SRP**
19	practice	22320	21030	20690	20930	**0.6**	132970	20700	20690	**RSP**
20	exciting	20380	19140	18900	18500	**0.8**	132580	17760	18500	**SRP**

Table B.20: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns corresponding to English words of length 8 in the input text T_{Real}. The last column of the table represents the efficiency ranking of the algorithms.

#	Word	m	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
			$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
1	something	9	31660	27740	26580	26220	**0.8**	133960	25900	26220	**SRP**
2	sometimes	9	20640	19740	19740	19050	**0.8**	134400	18240	19050	**SRP**
3	thousands	9	25240	23250	22740	22360	**0.8**	139160	21790	22360	**SRP**
4	determine	9	21800	20090	19710	19880	**0.6**	135560	18800	19710	**SRP**
5	direction	9	25170	23630	22630	21980	**0.8**	135130	21220	21980	**SRP**
6	represent	9	18950	17890	18110	17980	**0.4**	132190	16690	17890	**SRP**
7	different	9	21410	19500	19900	19880	**0.4**	134460	18620	19500	**SRP**
8	factories	9	27810	25570	24700	24770	**0.6**	130540	23540	24700	**SRP**
9	questions	9	23580	22060	20970	21590	**0.6**	134150	20360	20970	**SRP**
10	continued	9	22540	21930	20430	21170	**0.6**	135310	19770	20430	**SRP**
11	statement	9	20270	19520	19200	19250	**0.6**	135290	18150	19200	**SRP**
12	stretched	9	23180	21220	21180	20350	**0.8**	137250	20060	20350	**SRP**
13	necessary	9	21260	19470	19700	19630	**0.4**	135350	18800	19470	**SRP**
14	beautiful	9	19520	18990	18740	18240	**0.8**	134000	18190	18240	**SRP**

#	Word	m	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
			$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
15	important	9	22700	21160	20900	20340	**0.8**	136520	19930	20340	**SRP**
16	carefully	9	18950	18000	17830	18050	**0.6**	135050	16850	17830	**SRP**
17	consonant	9	18650	17520	17240	17630	**0.6**	133920	16740	17240	**SRP**
18	difficult	9	15280	14950	15280	15350	**0.4**	133580	14200	14950	**SRP**
19	suggested	9	16830	16440	15670	15920	**0.6**	134340	15180	15670	**SRP**
20	underline	9	20270	18450	19100	18500	**0.4**	135380	17430	18450	**SRP**
21	syllables	9	17020	16020	16330	15870	**0.8**	129740	15130	15870	**SRP**
22	themselves	10	17600	17350	17270	17760	**0.6**	133010	16400	17270	**SRP**
23	understand	10	23020	21320	21230	21450	**0.6**	136450	20810	21230	**SRP**
24	particular	10	16310	15900	15900	16170	**0.6**	134470	15100	15900	**SRP**
25	everything	10	22100	21110	20800	20920	**0.6**	136380	20090	20800	**SRP**
26	difference	10	16260	16020	16450	16620	**0.4**	134260	16040	16020	**RSP**
27	conditions	10	17760	17360	17620	17120	**0.8**	136340	16650	17120	**SRP**
28	experience	10	15340	15010	14960	15130	**0.6**	132240	14390	14960	**SRP**

#	Word	m	CPU Time (μ sec)				Best	CPU Time (μ sec)			Algo
			$\epsilon = 0.2$	$\epsilon = 0.4$	$\epsilon = 0.6$	$\epsilon = 0.8$	ϵ	**Pre**	**Suf**	**paRa**	**Ranking**
29	government	10	18640	18670	18140	18040	**0.8**	135550	17470	18040	**SRP**
30	washington	10	23320	21980	22270	22200	**0.4**	138050	20930	21980	**SRP**
31	especially	10	17410	16770	17030	17070	**0.4**	137630	15810	16770	**SRP**
32	discovered	10	18660	18230	18080	18250	**0.6**	136400	17510	18080	**SRP**
33	substances	10	18650	18150	18150	18380	**0.6**	135570	17530	18150	**SRP**
34	presidents	10	21250	20620	20620	19980	**0.8**	135820	19550	19980	**SRP**
35	experiment	10	17590	17230	16860	17340	**0.6**	135330	16720	16860	**SRP**
36	dictionary	10	20200	19650	19840	19690	**0.4**	139160	19600	19650	**SRP**
37	scientists	10	16970	17060	16590	16900	**0.6**	134260	16220	16590	**SRP**
38	instruments	11	19020	18920	18240	18610	**0.6**	136730	17880	18240	**SRP**
39	information	11	18070	17100	17130	17560	**0.4**	135360	15920	17100	**SRP**
40	temperature	11	15590	15010	15280	15470	**0.4**	134640	14800	15010	**SRP**

Table B.21: CPU time taken by the parameterized suffix based algorithms for different values of ϵ to find different abelian patterns corresponding to English words of length ≥ 9 in the input text T_{Real}. The last column of the table represents the efficiency ranking of the algorithms.

Appendix C

Empirical Analysis of the Algorithms for Approximate Abelian Pattern Matching Under Insertion/Deletion (InDel) Error Model

In this appendix, we present the results of the experiments performed for an empirical analysis of the two algorithms for approximate abelian pattern matching under insertion/deletion error model. The first algorithm uses a search window of fixed length (Section 4.4.3), here we refer to this algorithm as Algorithm A; the second algorithm uses a search window of flexible length (Section 4.4.5), here we refer to this algorithm as Algorithm B.

For the experiments, we used same input texts that were used for the empirical analysis of the prefix based and the suffix based algorithms (Appendix A). We executed algorithms A and B on randomly generated abelian patterns for different values of error threshold t. For each abelian pattern, we performed 200 iterations of each of the algorithms and took the mean values of the CPU time taken by the algorithms in 200 iterations.

C.1 Comparison of the Relative Efficiency of the Algorithms for the Input Text T_{4_U}

Table C.1 shows the CPU times taken by the algorithms A & B for finding approximate matches of randomly generated abelian patterns in the input text T_{4_U} for different values of the error threshold t.

Algorithm B was generally more efficient than Algorithm A. In the best case, Algorithm B was 1.59 times faster than Algorithm A for $t = 0.1m$, and 2.94 times faster than Algorithm A for $t = 0.2m$. Table C.2 lists the abelian patterns used for the experiments performed on the input text T_{4_U}.

C.2 Comparison of the Relative Efficiency of the Algorithms for the Input Text T_{8_U}

Table C.3 shows the CPU times taken by the algorithms A & B for finding approximate matches of randomly generated abelian patterns in the input text T_{8_U} for different values of the error threshold t.

Here too, Algorithm B was generally more efficient than Algorithm A. In the best case, Algorithm B was 1.26 times faster than Algorithm A for $t = 0.1m$, and 3.57 times faster than Algorithm A for $t = 0.2m$. Table C.4 lists the abelian patterns used for the experiments performed on the input text T_{8_U}.

C.3 Comparison of the Relative Efficiency of the Algorithms for the Input Text T_{Real}

Table C.5 shows the CPU times taken by the algorithms A & B for finding approximate matches of randomly selected abelian patterns in the input text T_{Real} for different values of the error threshold t.

In the case of real input text also, Algorithm B was more efficient than Algorithm A. In the best case, Algorithm B was 1.19 times faster than Algorithm A for $t = 0.1m$, and 4.16 times faster than Algorithm A for $t = 0.2m$. Table C.6 lists the abelian patterns used for the experiments performed on the input text T_{Real}.

#	m	t	**CPU Time (μ Sec)**		**B/A**	t	**CPU Time (μ Sec)**		**B/A**
			Algo A	**Algo B**	**Ratio**		**Algo A**	**Algo B**	**Ratio**
1	20	2	419500	348600	0.83	4	492050	292650	0.59
2	20	2	576150	474750	0.82	4	789550	506550	0.64
3	20	2	1321500	1107450	0.84	4	1623350	1277000	0.79
4	20	2	947000	760500	0.8	4	1164850	897150	0.77
5	20	2	402650	340500	0.85	4	453800	275500	0.61
6	20	2	796350	628450	0.79	4	1049050	726300	0.69
7	20	2	493000	417150	0.85	4	591550	371300	0.63
8	20	2	800050	619100	0.77	4	1055550	737500	0.7
9	20	2	600750	498000	0.83	4	791400	495250	0.63
10	20	2	607600	505050	0.83	4	796900	502250	0.63
11	20	2	385850	333950	0.87	4	364800	230600	0.63
12	20	2	1012550	837650	0.83	4	1302750	973300	0.75
13	20	2	2187600	1813000	0.83	4	2347500	1919650	0.82
14	20	2	2104950	1780900	0.85	4	2289150	1870550	0.82
15	20	2	491450	411850	0.84	4	590900	342650	0.58
16	20	2	1125850	958400	0.85	4	1401300	1061250	0.76
17	20	2	2698250	2381350	0.88	4	2817200	2331950	0.83
18	20	2	1632950	1369650	0.84	4	1931750	1472150	0.76
19	20	2	661350	533850	0.81	4	848050	544550	0.64
20	20	2	389450	332950	0.85	4	430150	264250	0.61
21	40	4	300050	256500	0.85	8	308850	136500	0.44
22	40	4	299100	260850	0.87	8	303950	133050	0.44
23	40	4	362650	284000	0.78	8	693850	257100	0.37
24	40	4	305050	270400	0.89	8	264100	119950	0.45
25	40	4	314300	277250	0.88	8	302700	143150	0.47
26	40	4	312850	259500	0.83	8	175450	95900	0.55
27	40	4	307200	267200	0.87	8	186900	103150	0.55

#	m	t	CPU Time (μ Sec)		B/A	t	CPU Time (μ Sec)		B/A
			Algo A	Algo B	Ratio		Algo A	Algo B	Ratio
28	40	4	1945450	1234450	0.63	8	2484600	1783550	0.72
29	40	4	302200	258150	0.85	8	199250	101100	0.51
30	40	4	301250	255500	0.85	8	331050	154000	0.47
31	40	4	421250	330250	0.78	8	798200	357400	0.45
32	40	4	295350	239000	0.81	8	403850	138650	0.34
33	40	4	450000	350950	0.78	8	927750	433700	0.47
34	40	4	300200	258050	0.86	8	113650	94900	0.84
35	40	4	573900	410450	0.72	8	1169250	568700	0.49
36	40	4	381950	302100	0.79	8	788950	371100	0.47
37	40	4	3655350	2790900	0.76	8	4007650	3048300	0.76
38	40	4	300800	253300	0.84	8	366350	130200	0.36
39	40	4	295100	253550	0.86	8	327950	149000	0.45
40	40	4	286000	248750	0.87	8	142500	101100	0.71

Table C.1: CPU time taken by Algorithm A (which uses a search window of fixed length) and Algorithm B (which uses a search window of flexible length) for finding approximate matches of randomly generated abelian patterns in the input text T_{4_U}. Column 1 of the table represents the pattern number, column 2 represents the pattern length m, column 3 and 7 represent the error threshold t. Column 6 and 9 represent the relative advantage of Algorithm B over Algorithm A.

#	**Corresponding Pattern**	#	**Corresponding Pattern**
1	$5c_1 + 9c_2 + 0c_3 + 6c_4$	21	$6c_1 + 19c_2 + 1c_3 + 14c_4$
2	$1c_1 + 9c_2 + 5c_3 + 5c_4$	22	$6c_1 + 14c_2 + 1c_3 + 19c_4$
3	$5c_1 + 8c_2 + 3c_3 + 4c_4$	23	$4c_1 + 7c_2 + 9c_3 + 20c_4$
4	$2c_1 + 4c_2 + 8c_3 + 6c_4$	24	$7c_1 + 20c_2 + 13c_3 + 0c_4$
5	$4c_1 + 0c_2 + 8c_3 + 8c_4$	25	$16c_1 + 7c_2 + 17c_3 + 0c_4$
6	$1c_1 + 6c_2 + 6c_3 + 7c_4$	26	$10c_1 + 24c_2 + 0c_3 + 6c_4$
7	$9c_1 + 2c_2 + 7c_3 + 2c_4$	27	$11c_1 + 0c_2 + 6c_3 + 23c_4$
8	$7c_1 + 6c_2 + 1c_3 + 6c_4$	28	$9c_1 + 6c_2 + 11c_3 + 14c_4$
9	$7c_1 + 4c_2 + 1c_3 + 8c_4$	29	$11c_1 + 3c_2 + 3c_3 + 23c_4$
10	$7c_1 + 4c_2 + 8c_3 + 1c_4$	30	$16c_1 + 17c_2 + 1c_3 + 6c_4$
11	$8c_1 + 9c_2 + 1c_3 + 2c_4$	31	$3c_1 + 7c_2 + 13c_3 + 17c_4$
12	$4c_1 + 7c_2 + 2c_3 + 7c_4$	32	$8c_1 + 3c_2 + 22c_3 + 7c_4$
13	$5c_1 + 4c_2 + 7c_3 + 4c_4$	33	$16c_1 + 10c_2 + 12c_3 + 2c_4$
14	$3c_1 + 5c_2 + 6c_3 + 6c_4$	34	$0c_1 + 2c_2 + 24c_3 + 14c_4$
15	$9c_1 + 7c_2 + 2c_3 + 2c_4$	35	$16c_1 + 10c_2 + 11c_3 + 3c_4$
16	$7c_1 + 3c_2 + 7c_3 + 3c_4$	36	$14c_1 + 1c_2 + 11c_3 + 14c_4$
17	$5c_1 + 6c_2 + 5c_3 + 4c_4$	37	$10c_1 + 10c_2 + 12c_3 + 8c_4$
18	$4c_1 + 3c_2 + 6c_3 + 7c_4$	38	$7c_1 + 2c_2 + 22c_3 + 9c_4$
19	$2c_1 + 9c_2 + 6c_3 + 3c_4$	39	$6c_1 + 1c_2 + 18c_3 + 15c_4$
20	$4c_1 + 9c_2 + 0c_3 + 7c_4$	40	$4c_1 + 13c_2 + 23c_3 + 0c_4$

Table C.2: List of the abelian patterns presented in Table C.1

#	m	t	**CPU Time (μ Sec)**		**B/A**	t	**CPU Time (μ Sec)**		**B/A**
			Algo A	**Algo B**	**Ratio**		**Algo A**	**Algo B**	**Ratio**
1	30	3	296450	261250	0.88	6	142150	101900	0.72
2	30	3	306250	266600	0.87	6	389250	170600	0.44
3	30	3	314600	269900	0.86	6	410050	177800	0.43
4	30	3	371000	301550	0.81	6	631150	276800	0.44
5	30	3	291100	259500	0.89	6	133100	102100	0.77
6	30	3	309550	276700	0.89	6	174450	108450	0.62
7	30	3	297500	264700	0.89	6	137850	102250	0.74
8	30	3	310400	279250	0.9	6	255500	125100	0.49
9	30	3	311100	270000	0.87	6	211400	111450	0.53
10	30	3	327750	280750	0.86	6	398650	171950	0.43
11	30	3	309850	273700	0.88	6	140600	99600	0.71
12	30	3	315350	276850	0.88	6	242550	122450	0.5
13	30	3	334600	283000	0.85	6	433000	182800	0.42
14	30	3	409150	326750	0.8	6	715400	346650	0.48
15	30	3	330350	282150	0.85	6	442900	186650	0.42
16	30	3	305350	271050	0.89	6	299050	142600	0.48
17	30	3	305250	276550	0.91	6	129200	106400	0.82
18	30	3	319050	277750	0.87	6	277700	134350	0.48
19	30	3	341950	294100	0.86	6	481700	200800	0.42
20	30	3	314000	288750	0.92	6	207350	114000	0.55
21	50	5	327000	282500	0.86	10	643300	207650	0.32
22	50	5	312650	283900	0.91	10	196650	106650	0.54
23	50	5	308600	271750	0.88	10	247100	106650	0.43
24	50	5	362050	285250	0.79	10	856250	269400	0.31
25	50	5	321650	281850	0.88	10	253400	109250	0.43
26	50	5	329850	274750	0.83	10	717200	244150	0.34
27	50	5	324000	277350	0.86	10	276200	108000	0.39

#	m	t	**CPU Time (μ Sec)**		**B/A**	t	**CPU Time (μ Sec)**		**B/A**
			Algo A	**Algo B**	**Ratio**		**Algo A**	**Algo B**	**Ratio**
28	50	5	324550	279900	0.86	10	491200	154950	0.32
29	50	5	328100	293750	0.9	10	233200	107850	0.46
30	50	5	319300	279150	0.87	10	381850	128800	0.34
31	50	5	330100	272600	0.83	10	617400	169800	0.28
32	50	5	300250	267250	0.89	10	339150	118450	0.35
33	50	5	303650	267300	0.88	10	356600	121250	0.34
34	50	5	287850	254750	0.89	10	228850	106100	0.46
35	50	5	293150	259400	0.88	10	139200	104650	0.75
36	50	5	296800	263150	0.89	10	206950	105800	0.51
37	50	5	332800	280550	0.84	10	625000	203200	0.33
38	50	5	324150	280900	0.87	10	484250	149550	0.31
39	50	5	308550	278300	0.9	10	140600	104300	0.74
40	50	5	314550	279450	0.89	10	260800	110750	0.42

Table C.3: CPU time taken by Algorithm A (which uses a search window of fixed length) and Algorithm B (which uses a search window of flexible length) for finding approximate matches of randomly generated abelian patterns in the input text T_{8_U}. Column 1 of the table represents the pattern number, column 2 represents the pattern length m, column 3 and 7 represent the error threshold t. Column 6 and 9 represent the relative advantage of Algorithm B over Algorithm A.

#	**Corresponding Pattern**
1	$5c_1 + 8c_2 + 0c_3 + 5c_4 0c_5 + 1c_6 + 2c_7 + 9c_8$
2	$3c_1 + 6c_2 + 7c_3 + 4c_4 0c_5 + 5c_6 + 4c_7 + 1c_8$
3	$3c_1 + 5c_2 + 1c_3 + 1c_4 5c_5 + 2c_6 + 7c_7 + 6c_8$
4	$2c_1 + 7c_2 + 2c_3 + 6c_4 4c_5 + 3c_6 + 2c_7 + 4c_8$
5	$0c_1 + 4c_2 + 0c_3 + 3c_4 8c_5 + 0c_6 + 9c_7 + 6c_8$
6	$0c_1 + 2c_2 + 8c_3 + 6c_4 6c_5 + 6c_6 + 2c_7 + 0c_8$
7	$4c_1 + 8c_2 + 9c_3 + 0c_4 0c_5 + 5c_6 + 4c_7 + 0c_8$
8	$0c_1 + 4c_2 + 5c_3 + 0c_4 3c_5 + 5c_6 + 5c_7 + 8c_8$
9	$9c_1 + 2c_2 + 2c_3 + 1c_4 4c_5 + 3c_6 + 1c_7 + 8c_8$
10	$0c_1 + 6c_2 + 7c_3 + 1c_4 4c_5 + 4c_6 + 4c_7 + 4c_8$
11	$9c_1 + 3c_2 + 0c_3 + 2c_4 0c_5 + 2c_6 + 7c_7 + 7c_8$
12	$3c_1 + 0c_2 + 7c_3 + 2c_4 8c_5 + 4c_6 + 5c_7 + 1c_8$
13	$4c_1 + 6c_2 + 7c_3 + 1c_4 5c_5 + 2c_6 + 1c_7 + 4c_8$
14	$3c_1 + 1c_2 + 5c_3 + 2c_4 5c_5 + 4c_6 + 4c_7 + 6c_8$
15	$4c_1 + 2c_2 + 4c_3 + 2c_4 3c_5 + 6c_6 + 1c_7 + 8c_8$
16	$0c_1 + 9c_2 + 6c_3 + 3c_4 4c_5 + 2c_6 + 2c_7 + 4c_8$
17	$0c_1 + 2c_2 + 0c_3 + 7c_4 0c_5 + 9c_6 + 5c_7 + 7c_8$
18	$4c_1 + 1c_2 + 6c_3 + 1c_4 4c_5 + 1c_6 + 9c_7 + 4c_8$
19	$2c_1 + 2c_2 + 7c_3 + 3c_4 7c_5 + 1c_6 + 4c_7 + 4c_8$
20	$1c_1 + 2c_2 + 7c_3 + 2c_4 2c_5 + 1c_6 + 9c_7 + 6c_8$
21	$8c_1 + 7c_2 + 8c_3 + 7c_4 11c_5 + 2c_6 + 7c_7 + 0c_8$
22	$6c_1 + 6c_2 + 1c_3 + 9c_4 13c_5 + 1c_6 + 1c_7 + 13c_8$
23	$14c_1 + 2c_2 + 8c_3 + 3c_4 2c_5 + 6c_6 + 2c_7 + 13c_8$
24	$2c_1 + 9c_2 + 13c_3 + 6c_4 6c_5 + 6c_6 + 4c_7 + 4c_8$
25	$9c_1 + 5c_2 + 9c_3 + 2c_4 12c_5 + 1c_6 + 11c_7 + 1c_8$
26	$6c_1 + 10c_2 + 10c_3 + 3c_4 3c_5 + 9c_6 + 7c_7 + 2c_8$
27	$10c_1 + 1c_2 + 13c_3 + 4c_4 0c_5 + 5c_6 + 10c_7 + 7c_8$

#	**Corresponding Pattern**
28	$4c_1 + 2c_2 + 9c_3 + 12c_4 6c_5 + 10c_6 + 6c_7 + 1c_8$
29	$1c_1 + 14c_2 + 5c_3 + 11c_4 8c_5 + 0c_6 + 8c_7 + 3c_8$
30	$12c_1 + 3c_2 + 12c_3 + 7c_4 2c_5 + 8c_6 + 1c_7 + 5c_8$
31	$13c_1 + 4c_2 + 11c_3 + 6c_4 2c_5 + 3c_6 + 6c_7 + 5c_8$
32	$6c_1 + 11c_2 + 9c_3 + 1c_4 1c_5 + 7c_6 + 12c_7 + 3c_8$
33	$13c_1 + 5c_2 + 5c_3 + 5c_4 11c_5 + 9c_6 + 2c_7 + 0c_8$
34	$14c_1 + 3c_2 + 7c_3 + 7c_4 13c_5 + 2c_6 + 0c_7 + 4c_8$
35	$14c_1 + 9c_2 + 11c_3 + 5c_4 1c_5 + 0c_6 + 0c_7 + 10c_8$
36	$4c_1 + 11c_2 + 0c_3 + 1c_4 8c_5 + 9c_6 + 14c_7 + 3c_8$
37	$9c_1 + 2c_2 + 2c_3 + 9c_4 10c_5 + 9c_6 + 6c_7 + 3c_8$
38	$3c_1 + 5c_2 + 8c_3 + 3c_4 12c_5 + 7c_6 + 1c_7 + 11c_8$
39	$13c_1 + 14c_2 + 11c_3 + 0c_4 3c_5 + 6c_6 + 1c_7 + 2c_8$
40	$9c_1 + 13c_2 + 7c_3 + 10c_4 4c_5 + 7c_6 + 0c_7 + 0c_8$

Table C.4: List of the abelian patterns presented in Table C.3

#	m	t	CPU Time (μ Sec)		B/A	t	CPU Time (μ Sec)		B/A
			Algo A	Algo B	Ratio		Algo A	Algo B	Ratio
1	20	2	113750	99450	0.87	4	128050	103550	0.81
2	20	2	117600	103600	0.88	4	149750	102000	0.68
3	20	2	116400	102000	0.88	4	146250	105700	0.72
4	20	2	117850	103150	0.88	4	150850	107400	0.71
5	20	2	112300	100550	0.9	4	126550	102100	0.81
6	20	2	109850	99700	0.91	4	122300	101200	0.83
7	20	2	114700	101250	0.88	4	122750	105400	0.86
8	20	2	116700	104800	0.9	4	142200	106500	0.75
9	20	2	118450	102950	0.87	4	137050	104700	0.76
10	20	2	113700	101800	0.9	4	123300	103800	0.84
11	20	2	116750	99800	0.85	4	129100	101600	0.79
12	20	2	112100	98650	0.88	4	117850	99600	0.85
13	20	2	117050	98700	0.84	4	127800	96500	0.76
14	20	2	113400	99400	0.88	4	133900	100850	0.75
15	20	2	113400	101850	0.9	4	123200	99650	0.81
16	20	2	106200	95400	0.9	4	109700	92500	0.84
17	20	2	114000	99900	0.88	4	118050	100000	0.85
18	20	2	115950	105200	0.91	4	146550	99950	0.68
19	20	2	114250	102400	0.9	4	121900	99900	0.82
20	20	2	116350	103950	0.89	4	138450	100950	0.73
21	50	5	116100	104650	0.9	10	474250	113750	0.24
22	50	5	115100	101100	0.88	10	293650	104850	0.36
23	50	5	114700	101800	0.89	10	354150	113200	0.32
24	50	5	116100	104050	0.9	10	512200	129050	0.25
25	50	5	116250	101100	0.87	10	327300	109400	0.33
26	50	5	116600	103200	0.89	10	425200	124000	0.29
27	50	5	115100	101850	0.88	10	495950	131900	0.27

#	m	t	**CPU Time (μ Sec)** Algo A	**CPU Time (μ Sec)** Algo B	**B/A Ratio**	t	**CPU Time (μ Sec)** Algo A	**CPU Time (μ Sec)** Algo B	**B/A Ratio**
28	50	5	117450	101050	0.86	10	292500	108150	0.37
29	50	5	116950	102150	0.87	10	500600	134650	0.27
30	50	5	116850	102400	0.88	10	300600	116600	0.39
31	50	5	113700	101900	0.9	10	185650	112900	0.61
32	50	5	114100	100700	0.88	10	332250	115400	0.35
33	50	5	114700	102800	0.9	10	553400	132450	0.24
34	50	5	113650	101000	0.89	10	391150	112900	0.29
35	50	5	111250	98850	0.89	10	140700	99900	0.71
36	50	5	114400	100850	0.88	10	291300	110300	0.38
37	50	5	115250	101850	0.88	10	174650	105200	0.6
38	50	5	113700	100850	0.89	10	210400	103900	0.49
39	50	5	112950	103500	0.92	10	308400	114500	0.37
40	50	5	115300	100350	0.87	10	427750	118350	0.28

Table C.5: CPU time taken by Algorithm A (which uses a search window of fixed length) and Algorithm B (which uses a search window of flexible length) for finding approximate matches of randomly selected abelian patterns in the input text T_{Real}. Column 1 of the table represents the pattern number, column 2 represents the pattern length m, column 3 and 7 represent the error threshold t. Column 6 and 9 represent the relative advantage of Algorithm B over Algorithm A.

#	**Corresponding Pattern**
1	$1c + 2d + 2e + 1h + 2i + 2k + 2n + 2o + 3t + 1u + 2w$
2	$2a+3e+1f+1i+1k+2l+1n+2o+2r+1s+2t+1w+1y$
3	$2a+1d+2e+2g+2h+1i+1l+1m+2n+1o+2r+1t+1u+1w$
4	$2a + 1e + 2h + 2i + 1l + 2n + 1o + 1r + 3s + 3t + 1u + 1v$
5	$1a + 1c + 1d + 1e + 1h + 4i + 1l + 1n + 2o + 2r + 4s + 1u$
6	$1a + 2c + 5e + 1h + 1m + 2n + 1o + 1p + 1s + 2t + 1u + 2x$
7	$5a + 3e + 2h + 1l + 1m + 1n + 1p + 3s + 3t$
8	$2a + 1d + 4e + 1g + 1h + 1l + 2m + 2n + 1o + 1r + 3t + 1u$
9	$2a + 2d + 1e + 3h + 1i + 2o + 2r + 2s + 4t + 1w$
10	$1a + 1c + 2d + 4e + 2h + 2i + 4l + 2s + 1t + 1w$
11	$2a+1b+1c+1d+3e+1f+3h+1i+1l+2n+1r+1s+1w+1y$
12	$3a + 1c + 2d + 2e + 2f + 2l + 1n + 2o + 1p + 3r + 1u$
13	$1a+2d+2e+1f+2l+2m+1n+3o+2r+1s+1t+1w+1y$
14	$2a+2d+2e+1f+1h+2i+1l+2n+3o+1t+1v+1w+1y$
15	$1a+1c+5e+1f+2i+2l+1m+1n+1o+1r+2s+1v+1y$
16	$1a + 2b + 3c + 2e + 2h + 1i + 4l + 1n + 1p + 1s + 1u + 1y$
17	$1a + 1b + 1c + 4e + 2f + 1g + 1i + 2n + 4o + 1r + 1t + 1v$
18	$1a+1c+2d+3e+2h+1i+2m+1n+2o+1s+2t+1u+1w$
19	$3a + 1c + 1e + 1i + 1m + 1n + 3o + 3s + 4t + 1u + 1y$
20	$1a + 1b + 3e + 2h + 2n + 1o + 2r + 1s + 3t + 2u + 1w + 1y$
21	$3a + 1b + 3d + 7e + 1f + 3h + 4i + 2k + 3l + 1m + 3n + 3o + 1p + 2r + 7s + 4t + 1u + 1y$
22	$4a + 2c + 2d + 6e + 1h + 3i + 2l + 2n + 5o + 1p + 1q + 5r + 3s + 4t + 7u + 1v + 1y$
23	$3a + 1b + 1d + 9e + 1f + 2h + 4i + 3l + 1m + 3n + 2o + 4r + 7s + 4t + 1u + 2v + 2y$
24	$6a + 1c + 1d + 6e + 4f + 3h + 3i + 1l + 2m + 4n + 3o + 1p + 2r + 3s + 6t + 1u + 1v + 1w + 1y$
25	$2a + 2b + 1c + 1d + 7e + 1f + 1g + 2h + 4i + 1m + 3n + 3o + 2p + 2q + 2r + 3s + 9t + 2u + 1w + 1y$
26	$3a + 2c + 2d + 5e + 1f + 1g + 6h + 1i + 1l + 2m + 3n + 5o + 2p + 2r + 3s + 7t + 3u + 1y$

#	**Corresponding Pattern**
27	$3a+1b+2c+2d+7e+2f+1g+2h+6i+3m+4n+3o+1p+4r+3s+4t+1u+1y$
28	$5a+1c+5d+4e+1g+6h+3i+3k+3n+5o+1p+3r+4s+2t+1u+2w+1y$
29	$3a+1d+5e+1g+2h+5i+1k+2l+1m+4n+6o+1p+2r+6s+6t+1u+1v+1w+1y$
30	$7a+1c+1d+6e+1f+6h+1i+6l+1m+2n+6o+1r+3s+4t+1u+1v+1w+1y$
31	$5a+2c+3d+3e+1g+1h+6i+1k+1m+4n+4o+3p+6r+6s+1t+2u+1v$
32	$2a+1b+2c+1d+11e+3h+3i+1k+2l+2m+5n+5o+2r+1s+4t+2u+2v+1y$
33	$6a+1c+2d+7e+1f+1g+4h+1i+1k+3l+2m+3n+3o+2r+4s+4t+3u+2y$
34	$3a+2d+6e+2h+6i+3l+1m+2n+6o+1p+6r+4s+3t+3u+1v+1y$
35	$2a+2b+5d+13e+1f+5h+1i+1l+7n+3o+3r+1s+2t+1u+1v+1w+1y$
36	$4a+1b+2d+7e+1f+2g+4h+2i+2l+3m+7n+3o+1r+1s+7t+1u+1w+1x$
37	$1a+2c+3d+9e+2f+2h+8i+1j+1k+1l+5n+2o+1r+4s+3t+2u+1v+2w$
38	$3a+2c+3d+5e+6h+7i+4l+1m+2n+1o+2p+5r+5s+2t+1w+1y$
39	$7a+2b+1c+8e+1f+5h+3i+4l+1m+1n+2o+1p+2r+3s+6t+1u+1v+1y$
40	$5a+2b+3d+6e+2f+6h+2i+2m+4n+3o+4r+2s+4t+2u+1w+2y$

Table C.6: List of the abelian patterns presented in Table C.5

Bibliography

[1] A. Amir, A. Apostolico, G. M. Landau, and G. Satta. Efficient Text Fingerprinting Via Parikh Mapping. *Journal of Discrete Algorithms*, 1(5–6):409–421, 2003.

[2] G.E. Andrews and K. Eriksson. *Integer Partitions*. Cambridge University Press, 2004.

[3] Anne Bergeron, Cedric Chauve, Fabien de Montgolfier, and Mathieu Raffinot. Computing Common Intervals of K Permutations, with Applications to Modular Decomposition of Graphs. *SIAM Journal on Discrete Mathematics*, 22(3):1022–1039, 2008.

[4] Sebastian Böcker. Sequencing from Compomers: The Puzzle. *Theory of Computing Systems*, 39(3):455–471, 2006.

[5] F.C. Botelho, Y. Kohayakawa, and N. Ziviani. A Practical Minimal Perfect Hashing Method. *Lecture Notes in Computer Science*, 3503:488–500, 2005.

[6] F.C. Botelho, R. Pagh, and N. Ziviani. Simple and Space-Efficient Minimal Perfect Hash Functions. *Lecture Notes in Computer Science*, 4619:139, 2007.

[7] R. S. Boyer and J. S. Moore. A Fast Strings Searching Algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[8] C. Chauve, Y. Diekmann, S. Heber, J. Mixtacki, S. Rahmann, and J. Stoye. On Common Intervals with Errors. Technical report, Bielefeld University, 2006.

[9] Z.J. Czech, G. Havas, and B.S. Majewski. Perfect Hashing. *Theoretical Computer Science*, 182(1-2):1–143, 1997.

[10] Eugen Domann, Torsten Hain, Rohit Ghai, Andr Billion, Carsten Kuenne, Kurt Zimmermann, and Trinad Chakraborty. Comparative Genomic Analysis for the Presence of Potential Enterococcal Virulence Factors in the Probiotic Enterococcus Faecalis Strain Symbioflor 1. *International Journal of Medical Microbiology*, 297(7-8):533 – 539, 2007. Special issue: Pathogenomics.

[11] T. Ejaz, S. Rahmann, and J. Stoye. Online Abelian Pattern Matching. Technical report, Bielefeld University, 2008.

[12] R. Eres, G. M. Landau, and L. Parida. Permutation Pattern Discovery in Biosequences. *Journal of Computational Biology*, 11(6):1050–1060, 2004.

[13] TI Fenner and G. Loizou. A Binary Tree Representation and Related Algorithms for Generating Integer Partitions. *The Computer Journal*, 23(4):332–337, 1980.

[14] T.I. Fenner and G. Loizou. An Analysis of Two Related Loop-free Algorithms for Generating Integer Partitions. *Acta Informatica*, 16:237–252, 1981.

[15] E. B. Fry, J. E. Kress, and Fountoukidis D. E. *The Reading Teachers Book of Lists*. Third edition.

[16] S. Graf, D. Strothmann, S. Kurtz, and G. Steger. HyPaLib: A Database of RNAs and RNA Structural Elements Defined by Hybrid Patterns. *Nucleic Acids Research*, 29(1):196, 2001.

[17] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2006.

[18] S. Heber, R. Mayr, and J. Stoye. Common Intervals of Multiple Permutations. *Algorithmica*, 2009.

[19] S. Heber and J. Stoye. Algorithms for Finding Gene Clusters. *Lecture Notes in Computer Science*, 2149:252–263, 2001.

[20] S. Heber and J. Stoye. Finding all Common Intervals of k Permutations. *Lecture Notes in Computer Science*, 2089:207–218, 2001.

[21] R. N. Horspool. Practical Fast Searching in Strings. *Software Practice and Experience*, 10(6):501–506, 1980.

[22] S. Jukna. *Extremal Combinatorics With Applications in Computer Science*. Springer, 2001.

[23] M.E. Karim, L. Parida, and A. Lakhotia. Using Permutation Patterns for Content-Based Phylogeny. *Lecture Notes in Computer Science*, 4146:115, 2006.

[24] M.E. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware Phylogeny Using Maximal πPatterns. In *EICAR 2005 Conference: Best Paper Proceedings*, pages 156–174, 2005.

[25] S. Karlin. Detecting Anomalous Gene Clusters and Pathogenicity Islands in Diverse Bacterial Genomes. *TRENDS in Microbiology*, 9(7):335–343, 2001.

[26] D. E. Knuth, J. H. Morris, Jr, and V.R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

[27] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching, Vol. 3*. Addison-Wesley, 1973.

[28] G.M. Landau, L. Parida, and O. Weimann. Gene Proximity Analysis across Whole Genomes via PQ Trees. *Journal of Computational Biology*, 12(10):1289–1306, 2005.

[29] R. Overbeek, M. Fonstein, M. D'Souza, G.D. Pusch, and N. Maltsev. The Use of Gene Clusters to Infer Functional Coupling. *Proceedings of the National Academy of Sciences*, 96(6):2896–2901, 1999.

[30] L. Parida. Gapped Permutation Patterns for Comparative Genomics. *Lecture Notes in Computer Science*, 4175:376, 2006.

[31] L. Parida. *Pattern Discovery in Bioinformatics: Theory & Algorithms*. CRC Press, 2007.

[32] Narayanan Raghupathy and Dannie Durand. Gene Cluster Statistics with Gene Families. *Mol Biol Evol*, 26(5):957–968, 2009.

[33] T. Schmidt and J. Stoye. Quadratic Time Algorithms for Finding Common Intervals in Two and More Sequences. *Lecture Notes in Computer Science*, 3109:347–358, 2004.

[34] R. Sedgewick. *Algorithms*. Addison-Wesley, 1988.

[35] Qiang Tu and Dafu Ding. Detecting Pathogenicity Islands and Anomalous Gene Clusters by Iterative Discriminant Analysis. *FEMS Microbiology Letters*, 221(2):269 – 275, 2003.

[36] E. Ukonnen. Approximate String Matching with q-Grams and Maximal Matches. *Theoretical Computer Science*, 92(1):191–211, 1992.

[37] T. Uno and M. Yagiura. Fast Algorithms to Enumerate all Common Intervals of Two Permutations. *Algorithmica*, 26(2):290–309, 2000.

[38] Arnim Wiezer and Rainer Merkl. A Comparative Categorization of Gene Flux in Diverse Microbial Species. *Genomics*, 86(4):462 – 475, 2005.

[39] K. Yamanaka, S. Kawano, Y. Kikuchi, and S. Nakano. Constant Time Generation of Integer Partitions. *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences E Series A*, 90(5):888, 2007.

[40] A. Zoghbi and I. Stojmenovic. Fast Algorithms for Generating Integer Partitions. *International Journal of Computer Mathematics*, 70:319–332, 1998.