A HYBRID PATTERN MATCHING ALGORITHM

KANUMPORN ASAWASIRIROJ

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER SCIENCE (TECHNOLOGY OF INFORMATION SYSTEM MANAGEMENT)
FACULTY OF GRADUATE STUDIES
MAHIDOL UNIVERSITY
2015

Thesis
entitled
A HYBRID PATTERN MATCHING ALGORITHM

.....
Miss Kanumporn Asawasiriroj
Candidate

.....
Prof. Noppadol Wanichworanant,
Ph.D. (Electrical Engineering)
Major advisor

.....
Assoc. Prof. Rangsipan Marukatat,
Ph.D. (Computer Science)
Co-advisor

.....
Asst. Prof. Supapron Kiattisin,
Ph.D. (Electrical and Computer
Engineering)
Co-advisor

.....
Prof. Patcharee Lertrit,
M.D., Ph.D. (Biochemistry)
Dean
Faculty of Graduate Studies
Mahidol University

.....
Asst. Prof. Supapron Kiattisin,
Ph.D. (Electrical and Computer
Engineering)
Program Director
Master of Science Program in
Technology of Information System
Management
Faculty of Engineering
Mahidol University

Thesis
entitled
A HYBRID PATTERN MATCHING ALGORITHM

was submitted to the Faculty of Graduate Studies, Mahidol University
for the degree of Master of Science
(Technology of Information System Management)
on
September 10, 2015

คนัมนภ อัศวศิริโรจน์
.....
Miss Kanumporn Asawasiriroj
Candidate

.....
Lect. Sotarat Thammaboosadee,
Ph.D. (Information Technology)
Chair

.....
Asst. Prof. Kairoek Choeychuen,
Ph.D. (Electrical and Computer
Engineering)
Member

.....
Lect. Noppadol Wanichworanant,
Ph.D. (Electrical Engineering)
Member

.....
Assoc. Prof. Rangsipan Marukatat,
Ph.D. (Computer Science)
Member

.....
Asst. Prof. Supapron Kiattisin,
Ph.D. (Electrical and Computer
Engineering)
Member

.....
Prof. Patcharee Lertrit,
M.D., Ph.D. (Biochemistry)
Dean
Faculty of Graduate Studies
Mahidol University

.....
Asst. Prof. Jackrit Suthakorn,
Ph.D. (Robotics)
Dean
Faculty of Engineering
Mahidol University

ACKNOWLEDGEMENTS

I wish to express my sincere thanks and appreciation to my advisor, Dr. Noppadol Wanichworanant for providing me with all the necessary facilities for the research and valuable guidance and encouragement extended to me. I am grateful to my committee member, Dr. Rangsipan Marukatat, and Dr. Supapron Kiattisin for their guidance and suggestions.

I take this opportunity to express gratitude to the Department of Information Technology Management of Mahidol University for the facilities. I also thank my parents for the unceasing encouragement, support and attention.

Kanumporn Asawasiriroj

A HYBRID PATTERN MATCHING ALGORITHM

KANUMPORN ASAWASIRIROJ 5336468 EGTI/M

M.Sc. (TECHNOLOGY OF INFORMATION SYSTEM MANAGEMENT)

THESIS ADVISORY COMMITTEE: NOPPADOL WANICHWORANANT, Ph.D. RANGSIPAN MARUKATAT, Ph.D. SUPAPORN KIATTISIN, Ph.D.

ABSTRACT

The pattern matching algorithms are widely used in computer science, including information retrieval, text editors, internet search engines, and biological applications. The research proposed a pattern matching algorithm by the combination of three patterns matching algorithms including Boyer-Moore-Horspool, Quick Search, and Raita algorithm, in order to produce the hybrid pattern matching algorithms, given as: Hybrid Max Shift and Reverse Hybrid Max Shift. The proposed algorithms were improved from Boyer-Moore-Horspool and Quick search algorithm by choosing the maximum of shifting value among them and applying with Raita algorithm's order comparing technique. Four datasets were used to test the proposed algorithms, given as English text, genome sequence, protein sequence, and random texts. The best- and worst- case time complexities were also presented in this research. By the experiments, the proposed algorithms were compared to the other existing algorithms. It was shown that the proposed algorithms outperform other existing pattern matching algorithms in terms of shorter average running time.

KEY WORDS: PATTERN MATCHING ALGORITHM / BOYER-MOORE-HORSPOOL / QUICK SEARCH / RAITA / HYBRID

103 pages

การพัฒนาอัลกอริทึมการค้นหาสายอักขระ

A HYBRID PATTERN MATCHING ALGORITHM

คนัมพร อัศวศิริโรจน์ 5336468 EGTI/M

วท.ม. (เทคโนโลยีการจัดการระบบสารสนเทศ)

คณะกรรมการที่ปรึกษาวิทยานิพนธ์: นภดล วณิชวรนันท์, Ph.D., รังสิพรรณ มฤคทัต, Ph.D., สุภาภรณ์ เกียรติสิน, Ph.D.

บทคัดย่อ

อัลกอริทึมเปรียบเทียบรูปแบบถือว่ามีบทบาทสำคัญและแพร่หลายในวงการวิทยาศาสตร์คอมพิวเตอร์ ซึ่งอัลกอริทึมนี้สามารถพบได้ในโปรแกรมประยุกต์ต่างๆ เช่น การค้นคืนสารสนเทศ, โปรแกรม Text Editor,โปรแกรมค้นหาบนอินเตอร์เน็ทและโปรแกรมทางชีววิทยา งานวิจัยนี้ได้นำเสนอวิธีการพัฒนาอัลกอริทึมเปรียบเทียบรูปแบบ โดยใช้การผสมผสานข้อดีของอัลกอริทึม 3 อัลกอริทึมเข้าด้วยกัน ได้แก่ Boyer-Moore-Horspool, Quick Search และ Raita อัลกอริทึมที่พัฒนาขึ้นมานี้ให้ชื่อว่า Hybrid Max Shift และ Reverse Hybrid Max Shift ทั้ง 2 อัลกอริทึมนี้จะใช้การเลือกค่าสูงสุดของการเลื่อนจากการคำนวณที่ได้มาจากอัลกอริทึม Boyer-Moore-Horspool และ Quick Search ร่วมกับการใช้ลำดับการเปรียบเทียบของอัลกอริทึม Raita การทดสอบผลจะทดสอบบนกลุ่มข้อมูล 4 ประเภท ได้แก่ ข้อมูลภาษาอังกฤษ, ข้อมูลของสายพันธุกรรม , ข้อมูลของสายโปรตีนและข้อมูลแบบสุ่ม การวิเคราะห์ประสิทธิภาพทั้งกรณีที่ดีที่สุดและแย่ที่สุดก็ได้ถูกนำมาแสดงในงานวิจัยนี้ นอกจากนั้นอัลกอริทึมที่ได้พัฒนานี้ยังถูกนำมาเปรียบเทียบผลการทำงานร่วมกับอัลกอริทึมอื่นๆ ที่เกี่ยวข้องในงานวิจัย ผลที่ได้จากการทดลอง แสดงผลออกมาว่าอัลกอริทึมที่พัฒนามีประสิทธิภาพในการทำงานที่สูงกว่าอัลกอริทึมอื่นๆ ในแง่ของค่าเฉลี่ยเวลาที่ใช้ในการทำงาน

103 หน้า

CONTENTS

CONTENTS (cont.)

CONTENTS (cont.)

LIST OF TABLES

LIST OF TABLES (cont.)

LIST OF FIGURES

LIST OF FIGURES (cont.)

LIST OF FIGURES (cont.)

CHAPTER I
INTRODUCTION

1.1 Background and Problem Statement

Pattern matching problem is one of the most fundamental problems studied in theoretical computer science. Todays, we can find several common software products capable of searching the concatenated of strings like text editor software, search engines application, applications that capable of searching for alignments of concatenated nucleotide or computational amino acid sequence pattern in biology database, and file validation for virus search.

Nowadays, string matching subject is widely used to study and to gain more attention in computer sciences area due to the rapid development of computational information. Problem of pattern matching is to search for string that is particularly focused on a cluster of text and the desired string representing its occurrence once or several times on a sequence of text. We define such TEXT as "pattern" with its length of *m* and define the data cluster as "TEXT" with its length of *n*. The pattern for searching will be built over a finite alphabet set, called Σ.

From computer literacy study, it suggests that there are many pattern matching algorithms available at the present time, each one has its different efficacy and searching techniques. Pattern matching algorithm operation will start searching pattern by using the sliding windows method which partially defines the cluster of Text into the length of *m* and slides the pattern until the end of each searching. The alignment of pattern starts at the left end of the Text (at the position of 0). The algorithm will continually compare the characters taken from the pattern corresponding to the text of sliding windows until a whole match is found or a mismatch occurs. The direction of the sliding window and the order in comparisons is computed and is based on variety with different pattern matching algorithms. The shift of sliding window continues until it reaches the end of TEXT. Then, the operation of pattern matching will stop.

Pattern matching algorithm technology has been continuously developed. Most widely used and well known algorithm is Brute Force, the simple operation. For each time of shifting string, this algorithm starts to shift the pattern by an exact position to the right with the skip value of 1. Brute Force algorithm is quite ineffective with less efficiency, especially for dealing with either large amount of text sequences or small alphabet size. There are other algorithms comparing to Brute Force that were presented in later generations with improved searching performances, each one has its own strengths and weaknesses, e.g. working best on some type of text patterns (natural languages or nucleotide types) or some pattern length. In general, the pattern matching algorithm is composed of 2 major operation phases, given as pre-processing phase and searching phase.

The Pre-processing phase determines the distance namely the shift value. The pattern will move to be utilized in the searching phase. The shift value processed in Pre-processing phase will be stored in the Shift Table. The purpose of pre-processing phase is to reduce numbers of comparison allowing the algorithm to run faster. However, the efficiency of matching is still relied on the algorithm techniques itself. The algorithmic efficiency development in the searching phase can be implemented by aligning the order of pattern searching and selecting the proper shift value in sliding window.

This study aims to develop the pattern matching algorithm by combining the advantages of Boyer-Moore-Horspool, Quick search and Raita algorithms to improve the more effective searching phase. This study selects the search order techniques used in Raita Algorithm as well as implementing the most effective shift value used in among these 3 algorithms. By the way, the proposed algorithm will be working more effectively and helps reduce processing time of the algorithm eventually.

1.2 Objectives

The objectives of this thesis can be described as follows.

1.2.1 To study the theory of well-known pattern matching algorithms

1.2.2 To design, develop, implement, and test the proposed algorithm to evaluate

the performance comparing to the well-known pattern matching algorithms.

1.3 Scope of Work

The research proposed a new algorithm of pattern matching algorithm by defining the sequence of character comparisons between pattern and text at each attempt with the appropriate shift value calculated for maximizing the skip of pattern. The new algorithm combines the concept of following algorithms, Boyer-Moore-Horspool, Quick search, and Raita. The proposed algorithm will be evaluated by comparing the performance to the existing algorithms including Boyer-Moore-Horspool, Quick search, and Raita algorithm. The test data used to evaluate in experiments comprise of English text, genome sequence, protein sequence, and random texts.

1.4 Expected Results

1.4.1 The proposed algorithm can work correctly.

1.4.2 The proposed algorithm has better performance than existing algorithms.

CHAPTER II
LITERTURE REVIEW

This chapter discusses the background knowledge of algorithms and other relevant researches involving the algorithms used in our study.

2.1 Overview of Pattern Matching Problems

2.1.1 String

String (S) is a traditionally sequence of characters lining in a long sequence of elements. The characters used in the pattern are taken from the finite alphabets of $\sum$ with the size of σ. Each string contains substring that starts at position i and ends at position j. The length of string is defined by $|S|$. String with the length of 0 is called Empty String. $S[i..j]$ is the empty string if $i > j$. The i_{th} character of the string S is denoted as S_i. For example, "gcatatgat" is a string taken from the set of alphabet, $\sum = \{a,c,g,t\}$ with alphabet size, σ of 4. The length of this string is 9.

2.1.2 The Exact String Problem

Given: a string P is denoted as the pattern and a long string T is denoted as the text. The exact matching problem is to find all possible occurrences of pattern P in text T.

For example,

Given T = accggttacgtatg and P = cgg.

P appears in T at $T_{(3,5)}$

For example,

Given T = accggttacgtatg and P = agg.

P does not appear in T.

2.1.3 Prefix and Suffix of String

Given string of $S = s_1s_2...s_n$, where $s_1s_2...s_n$ is any substring; $s_1s_2...s_i$, $1 \leq i \leq n$ is a prefix of S; and $s_1s_{i+1}...s_n$, $1 \leq i \leq n$, is a suffix of S.

Example of Prefixes:

Let S = attgtcg. The following samples are all prefixes of S, given as:

a

at

att

attg

attgt

attgtc

attgtcg

Example of Suffixes:

Let S = attgtcg. The following samples are all suffixes of S, given as:

g

cg

tcg

gtcg

tgtcg

ttgtcg

attgtcg

By definition, given a text, $T = t_1t_2...t_n$, and a pattern, $P = p_1p_2...p_m$, if P occurs in T, P must be both a prefix on the a suffix of T and a suffix on the a prefix of T, as shown in Figure 2.1.

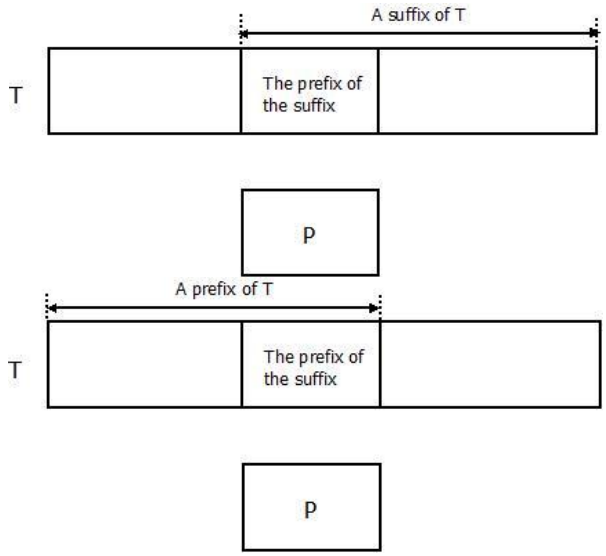


Figure 2.1 The matching of a pattern, *P*, with the text, *T*, in terms of prefix and suffix of *T*.

2.1.4 The Window Sliding Method

We can solve Exact String Matching Problem by applying Sliding window method. For example, if *T*= accgt and *P*=cg, then *P* occurs in *T* starting at the leftmost position, after characters comparison between *T* and *P* ends, *P* will be shifted from left to the right with the shift value of 1.

For example,

T = accgtgtaactgga, *P* = cgtgt.

Sliding window will define the position of *P* within the position of *T* starting at the leftmost position, i.e. with $T_{(1,5)}$, the starting position in this stage is $T_{(1,5)}$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	a	c	c	G	t	g	t	a	a	c	t	g	g	a
P	c	g	t	G	t									

First attempt: Character in position at $T_{(1,5)}$ does not match with character in *P*. Thus, *P* position will be shifted right by 1 position, and then new sliding window will be in position of $T_{(2,6)}$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	a	c	c	G	t	g	t	a	a	c	t	g	g	a
P		c	g	T	g	T								

Second attempt: Character in position of $T_{(2,6)}$ still does not match with character in *P*. Thus, *P* position will be shifted right by 1 position.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	a	c	c	G	t	g	t	a	a	c	t	g	g	a
P			c	G	t	g	t							

Third attempt: Now, new sliding window position will be $T_{(3,7)}$. It is found that every characters in $T_{(3,7)}$ position match with *P*, we call this searching method as "exact matching".

Most algorithms deploy sliding window method to implement searching task. However, from above examples, this kind of searching algorithm is still inefficient. Therefore, we will discuss the better methods in the next subsection.

Throughout this study, we adopt the following notations:

$T = t_1t_2\ldots t_n$	the text string with the length of n;
$P = p_1p_2\ldots p_m$	the pattern for searching on text string, T, with the length of m;
n	the text string length;
m	the pattern length;
t_i	the i^{th} character in the text string;
p_j	the j^{th} character in the pattern;
$\sum$	the set of a finite alphabet;
σ	the alphabet size;
W	substring of T used to match with P, called window.

2.1.5 1- Suffix Rule

1- Suffix Rule is applicable for character comparison of *P* and *T*, respectively to find all occurrences of *P* in *T*. Given that there is a mismatch occurs while searching *P* on *T*, and assumes *x* as position of the last occurrence of character on Window:

Let *W* is denoted as a substring of *T*, used to match with *P* as shown in Figure 2.2.

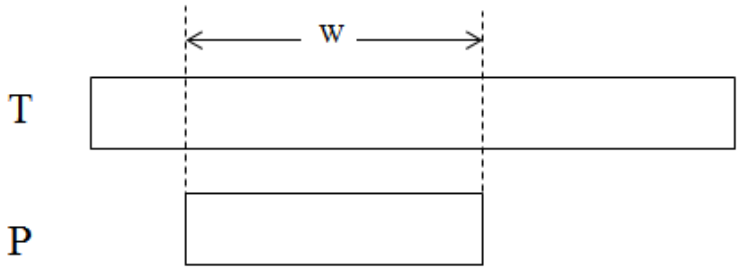


Figure 2.2 The opening of the window in *T*.

Suppose that a mismatching is found between the window *W* and *P*. We must shift *P* to the right. This is equivalent to opening a new window.

1- suffix rule assumes the last character of the suffix of text *T* (or called the last character of window *W*) is *x*.

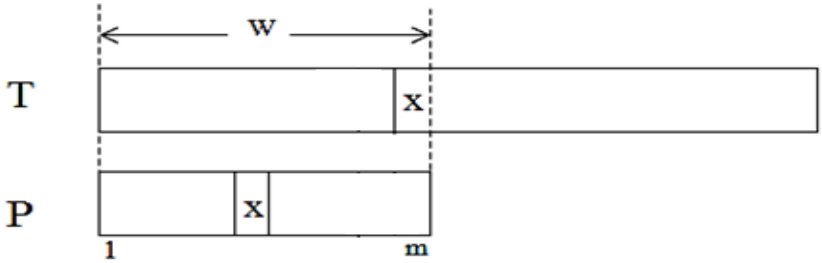


Figure 2.3 1-suffix rule.

To consider any substring *S* in *W*, there are two possible cases as follows.

Case 1: If there is occurrence of character *x* in any position on *P*, *P* then will be shifted to the right position with the position of character *x* on *P*, and *T* matches each other.

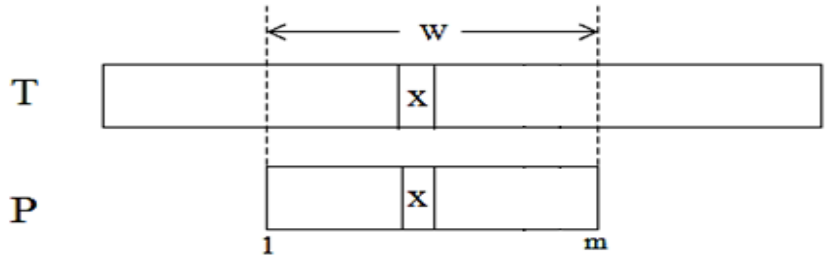


Figure 2.4 1-suffix matching shift (case 1).

Case 2: If there is NO occurrence of character *x* in any position on *P*, *P* then will be shifted to the right, in the position by the length of pattern (*m*).

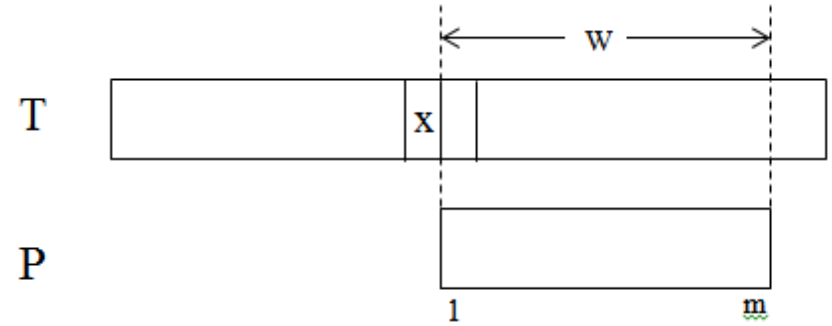


Figure 2.5 1-suffix matching shift (case 2).

Algorithm of 1-suffix rule

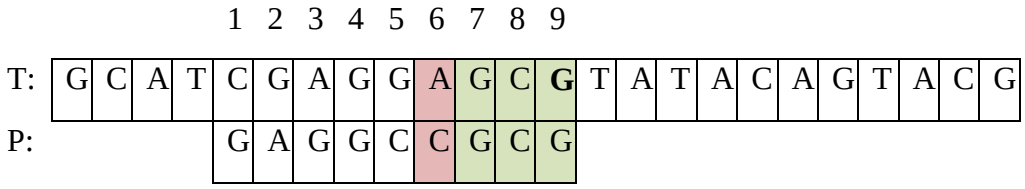
If there is a mismatch occurs in any position on *T* and *P*, we will search for the most nearest *x* character and the search will be performed in the position of *P*(1, *m*-1).

- If there is any occurrence of character *x* in the position, then shift *P* to the right by *m*-*k* position (*k* is the position of character *x* on *P*)
- If there is any NO occurrence of character *x* in the position *P* (1, *m*-1), then shift *P* to the right by *m* position.

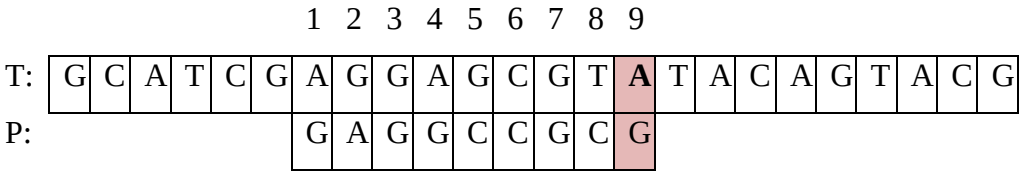
(Assume *x* is as the last character on Window.)

Figure 2.6 Algorithm of 1-suffix rule.

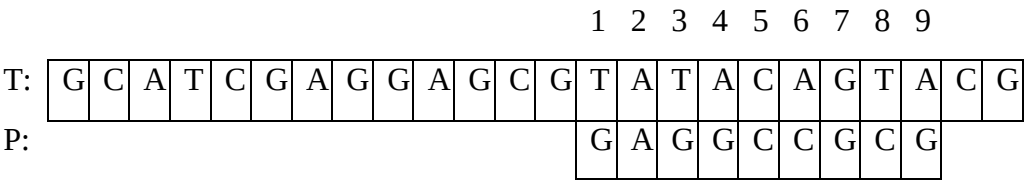
For example:



First attempt: This attempt will compare characters in *P* and *T*. It is found that a mismatch occurs at the position of W_6 and *x* character in this search is *G* located in the position W_9. When we examine characters in *P*, it is found that the most nearest *G* is located in the position W_7. Therefore, we shift right *P* by the position value of *m-k* = 9 -7 = 2 positions.



Second attempt: In this attempt, a mismatch occurs at the position of W_9. The *x* character in this search is given as *A* located in the position of W_9. When examine characters in *P*, it shows that the most nearest *A* is located at the position of W_2. Therefore, we shift right *P* by the position value of *m-k* = 9-2 = 7 positions.



2.2 Algorithms Description

2.2.1 Boyer-Moore-Horspool algorithm (BMH)

Horspool (1980) developed and published a new algorithm called, Boyer-Moore-Horspool which is a refinement of Boyer-More algorithm. This algorithm dismisses some complicated features of Boyer-More algorithm. In pre-processing phase, this algorithm implements only bad character rule (good-suffix rule used in Boyer-More algorithm is dismissed). This bad character rule is also known as 1-suffix

rule, as discussed in previous section. This algorithm will calculate the shift value of each character and stores into the shift table. In searching process, character comparison begins from the rightmost position to the leftmost position in Window until a mismatch or exact occurs. In case of a mismatch occurring, the shift value of the rightmost character in Window will be applied to shift *P* into the right position (as defined in 1-suffix rule). If there is no occurrence of character on the rightmost window of *P*, the shift value will be equal to the length of *P* (shift value = *m*).

Scan ordering

This algorithm will scan character for comparison in any position starting from the rightmost to the leftmost of window. Shift value in this algorithm is calculated from the character located from the rightmost character in Window. Boyer-Moore-Horspool algorithm will start character comparison on *T* and *P* starting from the rightmost position of Window. In Figure 2.7, *x* is the last character of the window.

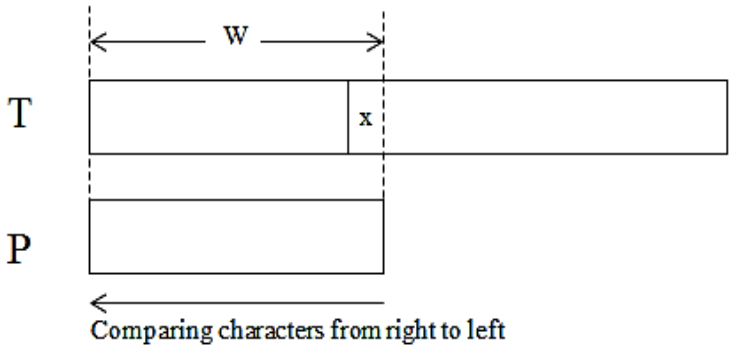
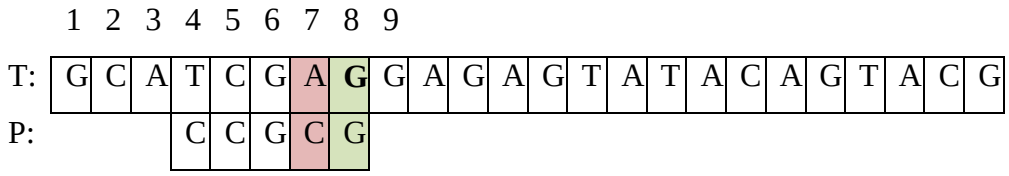


Figure 2.7 The right-to-left comparing mechanism of BMH Algorithm.

For example:

	1	2	3	4	5	6	7	8	9														
T:	G	C	A	T	C	**G**	A	G	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
P:		C	C	G	C	G																		
		1	2	3	4	5																		

From example, character comparison will initially start in the position of T_6 and P_5. It is found that characters of both T and P with the right position are matched. Then, the comparison continues running in the next position from right to left. The next position will be T_5 and P_4, and comparison still continues until a mismatch occurs at the position of T_4, P_3. In this case, P will be shifted to the right with the shift value of character located on the rightmost position of Window in that moment, e.g. *G*, as presented in this example at the position of T_6 (W_5). After P is shifted, the algorithm will resume character comparison on P and T starting from the rightmost position on Window again.



Preprocessing phase of the Boyer-Moore-Horspool algorithm

In the first step, Boyer-Moore-Horspool algorithm starts running the pre-process on P to produce an array, called "bad character shift", with its length of Σ. Values containing in the array defined by Eq.(2.1). Next, the algorithm will implement 1-suffix rule to define shift value for the next position.

$$bmBc\_shift(\sigma) = min(m - 1 - k : \{0 \leq k < m - 1 | p[m - 1 - k] = \sigma, \sigma \in \Sigma\}\{m\}) \quad (2.1)$$

The preprocessing steps of the Boyer-Moore-Horspool algorithm are shown in Figure 2.8. In Algorithm 1, array of *bmBc* is the Boyer-Moore-Horspool bad character shift array, which is initialized to value of *m* from Line 1 to Line 3. Lines 4-6 implement in Eq.(2.1).

Algorithm1 The preprocessing of the BMH & Raita.
PREBMH (*P*, *m*) 1 **for** *i* <- 0 to |∑| - 1 2 bmBc[*i*] <- m 3 **end for** 4 **for** *i* <- 0 to *m* - 1 5 bmBc[P[*i*]] <- *m* – *i* – 1 6 **end for** 7 **return** bmBc[]

Figure 2.8 The preprocessing of the BMH and Raita.

For example:

Let *T* = GCATCGCAGAGAGTAT, *P* = GCAGAGAG.

Pattern length (*m*) = 8 and Σ = {*A, C, G*}. Each element in *bmBc_shift* array, *bmBc*[*A, C, G*] is initialized to eight. After executing the *for* loop from Line 4-6, we have *bmBc_shift* array, *bmBc*[*A, C, G*] = [1, 6, 2]. The bmBc shift table is shown in Figure 2.9.

c	A	C	G	*
bmBc[c]	1	6	2	8

Figure 2.9 The shift table for *P* = GCAGAGAG.

(Let's * are other characters than A, C, G)

Searching phase of the Boyer-Moore-Horspool algorithm

The Boyer-Moore-Horspool Algorithm is initialized as shown below.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T =	G	C	A	T	C	G	C	**A**	G	A	G	A	G	T	A	T
P =	G	C	A	G	A	G	A	G								

First attempt:

This attempt starts running the character comparison in the position of T_8 and P_8. It is found that there is an occurred mismatch. Next, *P* will be shifted right by 1 position. Shift value applied in this stage will come from the rightmost character in Window, which is *A* character in this example. Shift value of *A* in this example, calculated and defined in *bmBc*[*A*], equal 1.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	=	G	C	A	T	C	G	C	A	**G**	A	G	A	G	T	A	T
P	=		G	C	A	G	A	G	A	G							
			1	2	3	4	5	6	7	8							

Second attempt:

This attempt starts running the character comparison in the position T_9 and P_8. It is found that characters on both of positions match each other and then the comparison continues running to the next position from right to left until a mismatch occurs in the positions of T_7, P_6, respectively. Next, *P* is shifted right by 2 positions so noted the shift value defined in array of *bmBc*[*G*], equals 2.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	=	G	C	A	T	C	G	C	A	G	A	**G**	A	G	T	A	T
P	=				G	C	A	G	A	G	A	G					
					1	2	3	4	5	6	7	8					

Third attempt:

This attempt starts running the character comparison in the position T_{11} and P_8. It is found that the characters on both of positions match each other, then the comparison continues running to the next position from right to left until a mismatch occurs in the positions of T_7, P_4, respectively. Next, *P* is shifted right by 2 positions. Noted the shift value defined in array of *bmBc*[*G*], equals 2.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	=	G	C	A	T	C	G	C	A	G	A	G	A	**G**	T	A	T
P	=						G	C	A	G	A	G	A	G			

Fourth attempt:

This attempt starts running the character comparison in the position T_{13} and $P_{8.}$ It is found that characters on both of positions match each other, then the comparison continues running to the next position from right to left until an exact matching occurs in the position of $T(6,13)$. Next, P is shifted right by 2 positions. Noted the shift value defined in array of *bmBc*[*G*], equals 2.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T =	G	C	A	T	C	G	C	A	G	A	G	A	G	T	**A**	T
P =								G	C	A	G	A	G	A	G	
								1	2	3	4	5	6	7	8	

Fifth attempt:

This attempt starts running the character comparison in the position T_{16} and $P_{8.}$ It is found that there is an occurred mismatch. Next, P will be shifted right by 1 position. The shift value applied in this stage is taken from the rightmost character in window, denoted as *A*. Shift value of *A* in this example, calculated and defined in *bmBc*[*A*], equal 1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T =	G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T
P =									G	C	A	G	A	G	A	G

C Language implementation

```
void preBmBc(char *x, int m, int bmBc[]){                    ┐
        int i;                                               │
        for (i=0; i < ASIZE; ++i)                            │
                bmBc[i] = m;                                 ├ Pre-processing phase
        for (i=0; i < m-1; ++i)                              │
                bmBc[x[i]] = m-i-1;                          │
}                                                            ┘
void HORSPOOL(char *x, int m, char *y, int n) {              ┐
        int j, bmBc[ASIZE];                                  │
        char c;                                              │
        preBmBc(x, m, bmBc);                                 │
        j = 0;                                               │
        while (j <= n - m) {                                 ├ Searching phase
             for (i = m – 1; i >= 0 && x[i] == y[i+j]; --i);  │
                OUTPUT(j);                                   │
             j += bmBc[c]                                    │
        }                                                    │
}                                                            ┘
```

Figure 2.10 Source code of Boyer-Moore-Horspool algorithm.

The value of ASIZE depends on the alphabet size, and Void OUTPUT (int) is a function used to print the position (*j*) of the current window of the text.

From Figure 2.10, as given algorithm, lines 3-4 run in σ steps; lines 5-6 run in *m* steps. Therefore, the total pre-processing time is O(m+σ).

2.2.2 Quick Search algorithm (QS)

Quick search algorithm, developed by Sunday (1990), is a refinement of Boyer-More algorithm. This algorithm also eliminates some complicated features of Boyer-More algorithm as a refined Boyer-Moore-Horspool algorithm. It is noted that we can implement a character next to the rightmost position, called "forward character", to determine a shift value for P. This algorithm will process character comparison from the leftmost position to the rightmost positions on Window. In contrast, with Boyer-Moore-Horspool algorithm, it starts running the comparison from the rightmost to the leftmost positions. In addition, this algorithm still retains implantation of 1-suffix rule for the shift of position.

Scan ordering

This algorithm will scan character for comparison in any position starting from the leftmost to the rightmost of Window. Shift value in this algorithm is calculated from the character located next to the rightmost character in Window. Quick search algorithm starts running the character comparison on *T* and *P* starting from the leftmost position of Window.

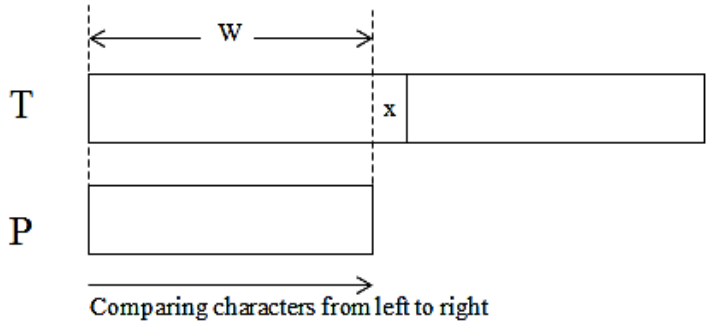
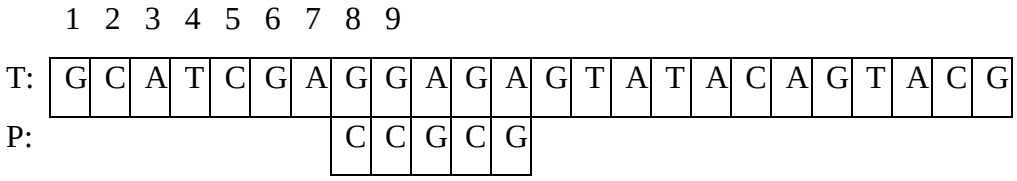


Figure 2.11 The right to left comparing mechanism of Quick search algorithm.

For example:

	1	2	3	4	5	6	7	8	9																
T:	G	C	A	T	C	G	**A**	G	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G	
P:		C	C	G	C	G																			
		1	2	3	4	5																			

As in this example, the comparison starts running at the position of T_2 and P_1. It is found that a mismatch occurs. *A* is a forward character occuring on *T* position, but there is no occurrence of *A* on *P*. As this manner of Boyer-Moore-Horspool algorithm, if there is No occurrence of last character on *P*, *P* will be shifted right by the length of pattern (*m*). Quick search algorithm, in case of the same manner occurring, *P* will be shifted right to the position of pattern length plus 1 (*m*+1).



The preprocessing phase of Quick Search Algorithm

In pre-processing step 1, the algorithm will produce an array, called "*qsBc*[]". Values containing in the array will be calculated and defined by Eq. (2.2), and will be stored into *qsBc* table.

$$qbad\_shift(\sigma) = min(m - k : \{0 \le k \le m - 1 | p[m - k - 1] = \sigma, \sigma \in \Sigma\} \cup \{m + 1\}) \quad (2.2)$$

The preprocessing steps of the Quick Search algorithm are shown in Figure 2.12. In Algorithm 2, array of qsBc is the Quick Search bad character shift array, which is initialized to value *m* from Line 1 to Line 3. Lines 4-6 implement Eq. (2.2).

Algorithm2 The preprocessing of Quick Search.
PREQS (*P*, *m*)
1 **for** *i* <- 0 to |∑| - 1
2 qsBc[*i*] <- *m*
3 **end for**
4 **for** *i* <- 0 to *m* - 1
5 qsBc[P[*i*]] <- *m* – *i*
6 **end for**
7 **return** qsBc[]

Figure 2.12 The preprocessing of Quick Search.

For example:

Let T = GCATCGCAGAGAGTAT and P = GCAGAGAG.

Pattern length (m) = 8 and σ = {A, C, G}. Each element in shift array $qsBc$[A, C, G], is initialized to eight. After executing the loop, we shift array of $qsBc$[A, C, G] = [2, 7, 1]. The qsBc shift table is given in Figure 2.13.

c	A	C	G	*
qsBc[c]	2	7	1	9

Figure 2.13 The *qsBc* shift table of P = GCAGAGAG.

Searching phase of the Quick search algorithm

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	=	G	C	A	T	C	G	C	A	**G**	A	G	A	G	T	A	T
P	=	G	C	A	G	A	G	A	G								

First attempt:

This attempt starts running character comparison in the position T_1 and P_1. It is found that the characters in both of positions match each other. Then the comparison still continues running in the next position from the left to the right until a mismatch occurs at the position of T_4 and P_4. Next, P will be shifted right by 1 position. Shift values applied in this stage will come from the character next to the rightmost character of Window, i.e. G in this example calculated from $qsBc$[G] array is equal to 1.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	=	G	C	A	T	C	G	C	A	G	**A**	G	A	G	T	A	T
P	=		G	C	A	G	A	G	A	G							
			1	2	3	4	5	6	7	8							

Second attempt:

This attempt starts running character comparison in the position T_2 and P_1. There is a mismatch occurring in this position. Then, *P* will be shifted right by 2 positions. Noted the shift value defined in array of *qsBc*[*A*], is equal to 2.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	=	G	C	A	T	C	G	C	A	G	A	G	**A**	G	T	A	T
P	=				G	C	A	G	A	G	A	G					
					1	2	3	4	5	6	7	8					

Third attempt:

This attempt starts running the character comparison in the position T_4 and P_1. There is a mismatch occurring in this position. Then, *P* will be shifted right by 2 positions. Noted the shift value defined in array of *qsBc*[*A*] is equal to 2.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	=	G	C	A	T	C	G	C	A	G	A	G	A	G	**T**	A	T
P	=						G	C	A	G	A	G	A	G			

Fourth attempt:

This attempt starts running character comparison in the position T_6 and P_1. It is found that characters on both of positions match each other, then the comparison continues running to the next position from left to right until an exact matching occurs in the position of $T_{(6,13)}$. Next, *P* is shifted right by 9 positions. Noted the shift value defined in array of *qsBc*[*T*] is equal 9.

T	=	G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T
P	=														G	C	A	G	A	G	A	G

C Language implementation

```
void preQsBc(char *x, int m, int qsBc[]) {
        int i;
        for (i = 0; i < ASIZE; ++i)
                qsBc[i] = m + 1;
        for (i = 0; i < m; ++i)
                qsBc[x[i]] = m - i;
}
void QS(char *x, int m, char *y, int n) {
        int j, qsBc[ASIZE];
        preQsBc(x, m, qsBc);
        j = 0;
        while (j <= n - m) {
                if (memcmp(x, y + j, m) == 0)
                        OUTPUT(j);
                j += qsBc[y[j + m]];
        }
}
```

Pre-processing phase

Searching phase

Figure 2.14 Source code of Quick search algorithm.

The value of ASIZE depends on the alphabet size, and Void OUTPUT (int) is a function used to print the position (*j*) of the current window of the text.

From Figure 2.14, as given algorithm, lines 3-4 run in σ steps, lines 5-6 run in *m* steps. Therefore, the total pre-processing time is O(m+σ).

2.2.3 Raita algorithm

This algorithm is different to Boyer-Moore-Horspool in terms of scanning Order (Raita, 1992). As previously discussed, Boyer-Moore-Horspool algorithm will scan character for comparison starting from rightmost position to leftmost position of window. However, for Raita algorithm, it has a particular scanning order.

Scan ordering

Raita algorithm has the idea that "neither the pattern nor the text is random; there exist strong dependencies between successive symbols. The dependencies may extend even over 30 symbols. They are strongest with respect to the nearest neighboring ones and weaken noticeably at word boundaries". From that suggest, if the last character of the pattern matched with the last character of the window, the algorithm attempts to match the first character of the pattern, because the dependencies between these two positions are weakest. If both characters are matched, the next comparison is the middle character of the pattern. "Thus, we have to follow the general principle: after each successful symbol matching, choose a symbol which the dependencies from all the symbols already probed are the weakest".

As notices earlier by the founder of this algorithm, when comparing character on the rightmost position of Window on *T* and *P*, it is found that they match each other. The next position to be compared will be the leftmost position of Window as both of positions have less correlation to each other. When comparing in the leftmost position, it is found that characters on *T* and *P* match each other, the next position to be compared will be the middle of Window. If a match still occurs, the rest position to be compared will be the leftmost plus 1 position and continues running to the right minus 1 position (left+1, right-1).

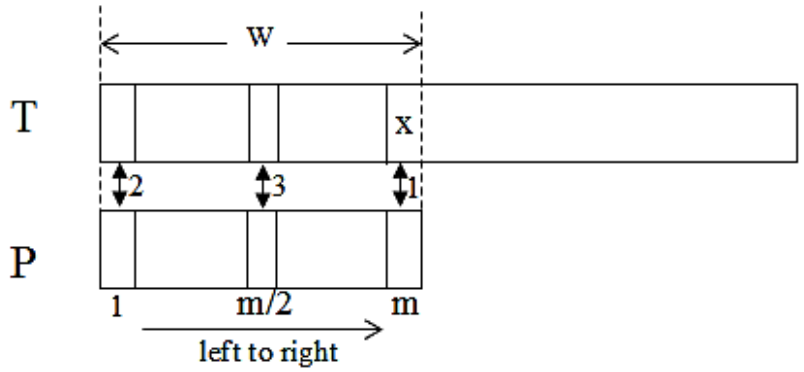


Figure 2.15 The specific order comparison in Raita algorithm.

For example:

	1	2	3	4	5	6	7	8	9																
T:	C	C	A	T	**G**	G	A	G	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G	
P:	C	C	G	C	G																				

This attempt starts running the character comparison in the position T_5 and P_5. It is found that characters on both of positions match each other. Then the comparison continues running to the next position of T_1 and P_1. It is found that the characters on both of positions match each other again. Then the comparison continues running to the next position of T_3 and P_3. It is found that there is an occurred mismatch. In this case, *P* will be shifted to the right with the shift value of character located on the rightmost position of Window in that moment, e.g. *G*, as presented in this example at the position of T_5. After *P* is shifted, the algorithm will resume character comparison on *P* and *T* starting from the rightmost position on Window again.

	1	2	3	4	5	6	7	8	9																
T:	C	C	A	T	G	G	**A**	G	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G	
P:			C	C	G	C	G																		

The preprocessing phase of the Raita algorithm

Raita algorithm will implement the same function used in Boyer-Moore-Horspool algorithm to find shift vale in which it will produce *bmBc*[] array to store shift value , therefore both algorithms utilize the same shift value.

For example:

Let *T* = GCATCGCAGAGAGTAT and *P* = GCAGAGAG. The *bmBc* shift table is as shown below.

c	A	C	G	*
bmBc[c]	1	6	2	8

Figure 2.16 The *bmBc* shift table for *P* = GCAGAGAG.

Searching phase of the Raita algorithm

The Raita algorithm is initialized as shown below.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T =	G	C	A	T	C	G	C	**A**	G	A	G	A	G	T	A	T
P =	G	C	A	G	A	G	A	G								

First attempt:

This attempt starts running the character comparison in the position of T_8 and P_8. It is found that there is an occurred mismatch. Next, *P* will be shifted right by 1 position. Shift value applied in this stage will come from the rightmost character in Window, which is *A* character in this example. Shift value of *A* in this example, calculated and defined in *bmBc*[*A*], equal 1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T =	G	C	A	T	C	G	C	A	**G**	A	G	A	G	T	A	T
P =		G	C	A	G	A	G	A	G							
		1	2	3	4	5	6	7	8							

Second attempt:

This attempt starts running the character comparison in the position T_9 and $P_{8.}$ It is found that characters on both of positions match each other. Then the comparison continues running to the next position at T_2 and P_1. It is found that there is an occurred mismatch. Next, *P* is shifted right by 2 position so noted the shift value defined in array of *bmBc*[*G*], equals 2.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	=	G	C	A	T	C	G	C	A	G	A	**G**	A	G	T	A	T
P	=				G	C	A	G	A	G	A	G					
					1	2	3	4	5	6	7	8					

Third attempt:

This attempt starts running the character comparison in the position T_{11} and $P_{8.}$ It is found that characters on both of positions match each other. Then the comparison continues running to the next position at T_4 and P_1. It is found that there is an occurred mismatch. Next, *P* is shifted right by 2 position so noted the shift value defined in array of *bmBc*[*G*], equals 2.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	=	G	C	A	T	C	G	C	A	G	A	G	A	**G**	T	A	T
P	=						G	C	A	G	A	G	A	G			

Fourth attempt:

This attempt starts running character comparison in the position T_{13} and $P_{8.}$ It is found that characters on both of positions match each other. The comparison continues running to the next position of T_4 and $P_{1.}$ It is found that characters in that position match each other. Then the comparison continues running to the next position of T_{10} and P_{10} the comparison continues running to the next position from left+1 to right-1 until an exact matching occurs in the position of $T_{(6,13)}$. Next, *P* is shifted right by 2 positions. Noted the shift value defined in array of *bmBc*[*G*] is equal 2.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	=	G	C	A	T	C	G	C	A	G	A	G	A	G	T	**A**	T
P	=								G	C	A	G	A	G	A	G	
									1	2	3	4	5	6	7	8	

Fifth attempt:

This attempt starts running the character comparison in the position of T_{15} and $P_{8.}$ It is found that there is an occurred mismatch. Next, *P* will be shifted right by 1 position. Shift value applied in this stage will come from the rightmost character in Window, which is *A* character in this example. Shift value of *A* in this example, calculated and defined in *bmBc*[*A*], equal 1.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	=	G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T
P	=									G	C	A	G	A	G	A	G

C Language implementation

```
void preBmBc(char *x, int m, int bmBc[]){
        int i;
        for (i=0; i < ASIZE; ++i)
                bmBc[i] = m;
        for (i=0; i < m-1; ++i)
                bmBc[x[i]] = m-i-1;
}
void RAITA(char *x, int m, char *y, int n) {
        int j, bmBc[ASIZE];
        char c, firstCh, *secondCh, middleCh, lastCh;
        preBmBc(x, m, bmBc);
        firstCh = x[0];
        secondCh = x + 1;
        middleCh = x[m/2];
        lastCh = x[m - 1];
        j = 0;
        while (j <= n - m) {
                c = y[j + m - 1];
                if (lastCh == c && middleCh == y[j + m/2] &&
        firstCh == y[j] && memcmp(secondCh, y + j + 1, m - 2) == 0)
                        OUTPUT(j);
                j += bmBc[c];
          }
}
```

Pre-processing phase

Searching phase

Figure 2.17 Source code of Raita algorithm.

Table 2.1 Comparison of existing algorithms.

Algorithms	**Rule**	**Order**	**Shift Function**	**Preprocessing**		**Searching Time Complexity**
				Time Complexity	Space Complexity	
Boyer-Moore-Horspool	1-suffix rule	Right to left	*bmBc*[]	O(*m*+σ)	O(σ)	O(*mn*)
Quick Search	1-suffix rule	Left to right	*qsBc*[]	O(*m*+σ)	O(σ)	O(*mn*)
Raita	1-suffix rule	Rightmost -> leftmost -> middle	*bmBc*[]	O(*m*+σ)	O(σ)	O(*mn*)

2.3 Related Work

There are several researches and references to improve the pattern matching algorithms. The appropriate articles are as follows.

A Fast Pattern Matching Algorithm (Sherik et al., 2004)

Sherik et al. (2004) developed new pattern matching algorithm, namely SSABS algorithm. This developed algorithm analysis from Boyer-Moore, Boyer-Moore-Horspool, Quick search, and Raita algorithm. The proposed algorithm divided working process into pre-processing phase, using the *qsBc* function of Quick search algorithm and working in searching phase using the comparison sequence of Raita algorithm. Data set used in the experiment consisted of nucleotide sequence and amino acid sequence. By the result, with comparing to other algorithm, it was found that the proposed algorithm could decrease the number of comparison time. Moreover, it was also higher efficiency for every size of alphabets, especially for working with

biological sequence database.

A Fast Pattern Matching Algorithm for Biological Sequences (Huang et al, 2008)

Due to the swiftly increased computing speed of DNA and protein sequence, Huang et al. (2008) developed new algorithm, called ZTBMH. This algorithm happened from the gathering of Zhu-Takaoka and Boyer-Moore-Horspool algorithm together. Both of algorithms only used the bad character function, which would eliminate the comparison time. In the pre-processing, this algorithm would use the function of bad character of Zhu-Takaoka algorithm to calculate the shift value. This function used last 2 alphabets in calculation, which was different to Boyer-Moore-Horspool function using the last alphabet. Maximum shift value occurs, when all 2 alphabets do not appear on pattern. The bigger the alphabet size, the less the probability that both of alphabets will appear in pattern. As process of searching phase, it will use sequence to search as well as Boyer-Moore-Horspool algorithm. Analysis ties the complexity of algorithm in the best case with O(*n/m*) value and the worst case with given O(*nm*) value. The experiment used the data set of amino acid sequence and nucleotide sequence. By the results, the develop algorithm obtained the higher efficiency with less working time apart from the case, when pattern length equals 4 or longer. It demonstrated that the algorithm was suitable to apply on the middle pattern and small pattern size, especially for working on biological sequence database.

A Fast String Matching Algorithm (Verma et al, 2011)

Verma et al. (2011) developed new algorithm, called VS algorithm. This algorithm was created from combination of Boyer-Moore, Boyer-Moore-Horspool, and Raita algorithm working together. The pre-processing phase work used the bad character function of Boyer-Moore-Horspool. Developer gave the reason of using such function instead of Boyer-Moore because when it works on large alphabet size and too small pattern, then function of Boyer-Moore did not have enough efficiency as it should. The function of Boyer-Moore would create 2 shift tables which were good suffixes and bad characters, whereas Boyer-Moore-Horspool only created 1 table that

offer more independently work. The process of searching phase used the same search sequence as Raita algorithm with less modification. If it was Raita algorithm, it will start comparison at the rightmost position, leftmost position, and the middle, then it compares from the far left position+1 to the far right-1. But instead, this algorithm changed to compare from rightmost position -1 to leftmost position+1. Experiment on pattern length with length of 2-20 alphabets, it found that the time complexity did not reveal an increase of efficiency but in practice the developed algorithm were 50% more effective than other algorithms..

CHAPTER III
METHODOLOGY

This chapter describes the schemes foe developing our proposed algorithms with hybridization of three combined well known algorithms. The development process of our proposed algorithms can be illustrated in Figure 3.1.

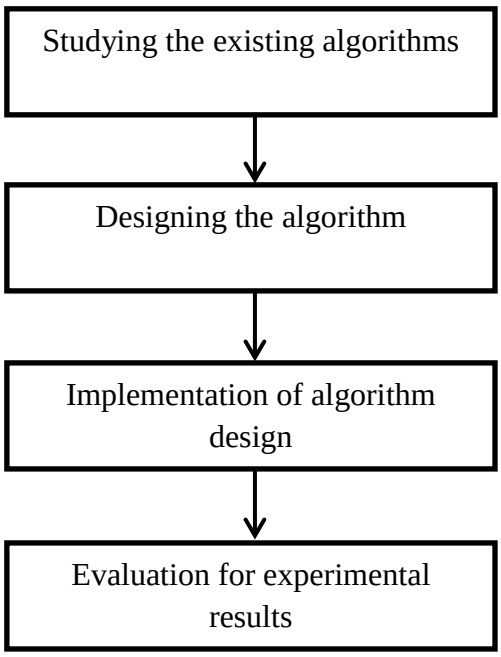


Figure 3.1 Development processes.

The details of each process can be described as follows:

- Studying the existing algorithms: This process aims to study the well-known algorithms, because there are various developments of pattern matching algorithms. Therefore, in order to develop our proposed algorithms, we have to study both advantages and disadvantages of their well-known algorithms.
- Designing the algorithm: After studying the well-known algorithms,

we will take the advantages of their algorithms to recombine as the new algorithms. It should be expected that the recombined algorithms' results must be better than earlier models.

- Implementation of algorithm design: The development process will be performed along with the result analysis in order to get the most effective work done.
- Evaluation for experimental results: In this process, if the results under development are not desirable, we will re-design the algorithm again.

3.1 Studying the existing algorithms

As described in Chapter 2, the pattern matching problem is considered as common problem in Computer Science. Nowadays, there are various developed algorithms to solve the pattern matching problem. However, the problem is that we should redesign the algorithm to solve the problem of pattern matching efficiently. Therefore, we have to study the popular used algorithm, called Boyer-Moore algorithm Moreover, we have further studied other algorithms mainly developed from the core algorithm of Boyer-Moore, including Boyer-Moore-Horspool, Quick Search and Raita algorithms. Noted, the details of those three algorithms are summarized in Chapter 2.

3.2 Designing the algorithm

After studying the existing algorithms as described in Chapter 2, the design of new algorithms would bring the advantages of those algorithms for models development.

3.2.1 The Hybrid Max Shift Algorithm (HMS)

In this algorithm, the ideas of design will be described as follows:

Regarding the order for character comparisons, the same comparison sequence used by the Raita algorithm will be used, that is to start making comparisons at the last position of the window and pattern. If the characters match, comparisons will then be made for the leftmost position of windows and patterns as the next

position. If it is found again that the characters match, the middle position of the windows and patterns will then be searched. After that, comparisons will be made from the leftmost +1 position to the rightmost -1 position until a mismatch or the exact match was found. After each comparison phase, patterns will be shifted to the right of the text until the end of the text is reached, where the shift value used in the shift will be selected from the maximum value between the bmBc shift table and the qsBc shift table.

The reason for the comparison of the first position from the rightmost position the second position from the leftmost position, the third position from the center position and carrying out the comparisons of remaining parts from left positions to right is because of Raita's proof that had been stated in his research: “dependency of the neighboring characters is strong compared to the other characters, Hence, it is always better to postpone the comparisons on the neighboring characters”. From this part there is the basic information that was used for the development of new algorithms in this research. Additionally, the choice to use the maximum shift values between the two tables naturally resulted in decreases in the number of times of character comparisons and shifts.

The algorithm developed in this research shall be named Hybrid Max Shift (HMS) as it is an algorithm created as a hybrid of many algorithms together and the choice to use the maximum shift values for shifting.

3.2.1.1 Preprocessing Phase

This phase will pre-process characters in each pattern to calculate the shift value for use in further searching. (Shift values will be kept in the shift table) The Preprocessing phase of the HMS algorithm will create 2 shift tables divided into the *bmBc* shift table used to keep shift values of both results of the Boyer-Moore-Horspool and Raita algorithms, and the *qsBc* shift table used to keep the shift value taken from the Quick search algorithm.

In the case of the Boyer-Moore-Horspool and Raita algorithms, the shift value of each character is the position of each character on the pattern counting from the rightmost -1 position to the leftmost determined by Equation (3.1). If a character does not exist in the pattern, the shift value will be equal to the length of the pattern (*m*)

$$bmBc\_shift(\sigma) = min(m - 1 - k : \{0 \leq k < m - 1 | p[m - 1 - k] = \sigma, \sigma \in \Sigma\} \cup \{m\} \quad (3.1)$$

The advantages of the *bmBc* shift function is the use of the character to the rightmost position in selecting a shift value for shifting instead of using a character that can create a mismatch, which causes patterns to be shifted further. However, the disadvantage of this function is given that the shift value acquisition sometimes obtains the lower value than the value taken from the Boyer-Moore algorithm.

In case of the Quick Search algorithm, the shift value of each character is that the position of those characters on the pattern counted from the rightmost to leftmost. If it is a character that does not exist in the pattern, the shift value will be equal to the length of the pattern size+1 (m+1). The Quick Search shift function is defined as follows:

$$qsBc\_shift(\sigma) = \min(m - k : \{0 \leq k \leq m - 1 | p[m - k - 1] = \sigma, \sigma \in \Sigma\} \cup \{m + 1\}) \quad (3.2)$$

The advantage of the *qsBc* shift function is that in the case that the position's character used in shift value selection does not exist in the pattern, the maximum shift value will be equal to m+1, which is different from the other 2 algorithms whose maximum values are equal to m. It can be seen that the *qsBc* shift function always has a higher value than the *bmBc* shift function practically and constantly. However, it excepts for the case where the last character exists in the pattern, which counts as a disadvantage discovered from the study.

As previously described the mechanisms of both models of pre=processing phase, in this research we choose them to create two shift tables to determine the maximum values between them in order to eliminate the disadvantages of both models.

Case 1: If the last character of the window does not present in the pattern while the next character from the last also appears in the pattern, the maximum shift value will be selected from the *bmBc* shift table

Case 2: Otherwise, the maximum shift value will be selected from the *qsBc* shift table

For example,

T = ATCTAACTATAGGGCAGAGAGAAAC,

P = GCAGAGAG,

Text length (n) = 25 and pattern length (m) = 8

Boyer-Moore-Horspool and Raita shift table

c	A	C	G	*
bmBc[c]	1	6	2	8

Quick search shift table

c	A	C	G	*
qsBc[c]	2	7	1	9

Figure 3.2 The shift tables of Boyer-Moore-Horspool, Raita, and Quick search algorithms.

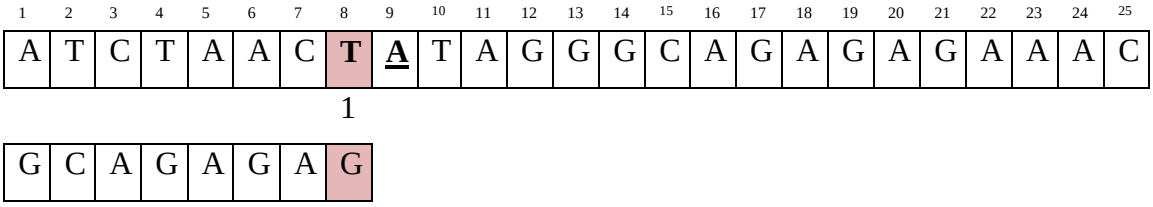


Figure 3.3 The first attempt of example.

First attempt:

This attempt starts running the characters comparison between T_8 and P_8. Then, it found that there is a mismatch occurring. Shift value applied in this stage is taken from both rightmost character and forward character of Window, which are T and A character respectively. Shift value of T is calculated and defined by $bmBc[T]$ which is equal to 8. The shift value of A is calculated and defined by $qsBc[A]$, which equals to 2. The maximum shift value is 8. Then, the algorithm shifts the pattern P by 8 characters, as shown in Figure3.4.

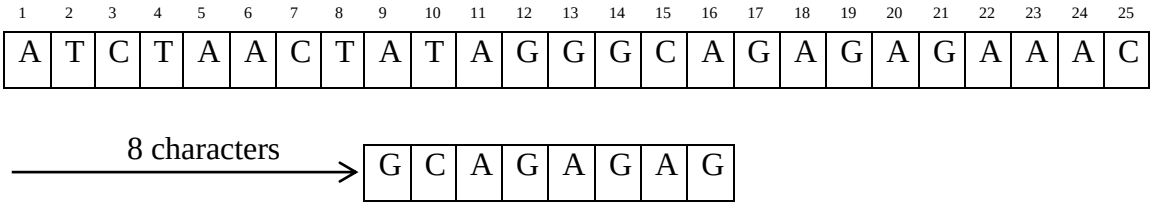


Figure 3.4 The pattern shifting.

3.2.2 Searching Phase

This algorithm has been developed by the idea of the sequence comparison of the Raita algorithm. It starts by comparing the characters at the rightmost of both window and pattern. If a match is found, the character comparison will be performed at the leftmost of both window and pattern. If a match is found again, the next comparison will be performed at the center of both window and pattern. If a match is also found again, the rest comparison will be performed from the position of leftmost+1 to rightmost-1 of both window and pattern. After finishing the comparison steps, the next step is to shift the pattern to the right of text. This step will bring the maximum shift value taken from the pre-processing phase.

The procedure can be summarized as follows:

1. Perform the comparison of the window and pattern's characters according to the sequential search of Raita algorithm.
2. If a match or mismatch has occurred, choose the shift value of the last character of the *bmBc* shift table and the shift value of the character at the last position +1 of the *qsBc* shift table
3. If the character's shift value of *bmBc* shift table is less than the shift value of *qsBc* shift table, use the shift value of *qsBc* shift table.
4. Shift the pattern to the right of the text by the maximum shift value selected from the previous step.

Example:

Text = ATCTAACTATAGGGCAGAGAGAAAC,

Pattern = GCAGAGAG,

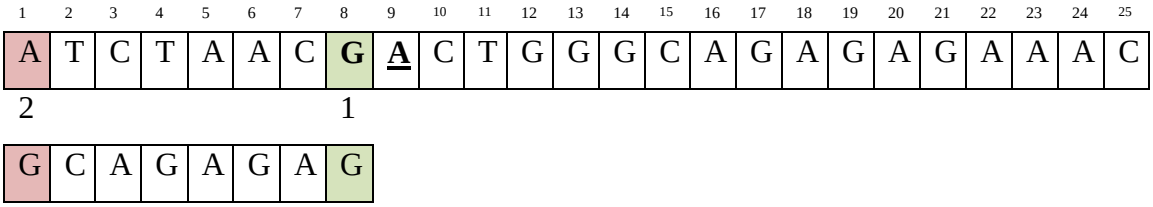
Text length (n) = 25 and pattern length (m) = 8

c	A	C	G	*
bmBc[c]	1	6	2	8

Figure 3.5 The *bmBc* shift table of *P* = GCAGAGAG.

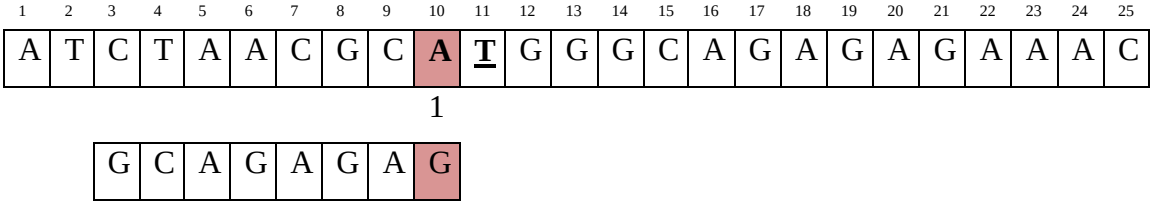
c	A	C	G	*
qsBc[c]	2	7	1	9

Figure 3.6 The *qsBc* shift table of *P* = GCAGAGAG.



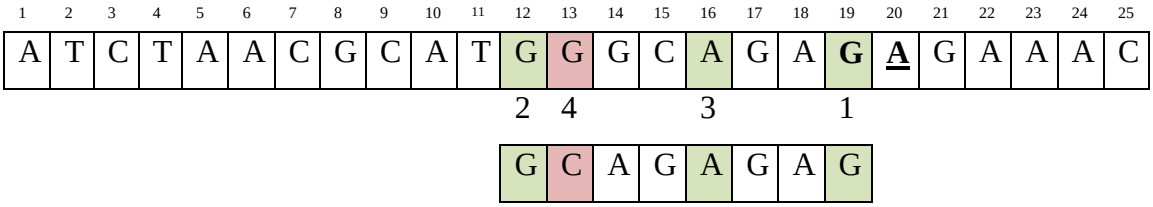
First Attempt:

The search begins at the rightmost position of the window and pattern and found a match. Next, a comparison is made at the leftmost position and found a mismatch. After the mismatch is found for the shift value obtained from the *bmBc* shift table and the *qsBc* shift table, the shift value of the last character and last character +1 of the window are considered. In this example, the last character of the window is *G*. Thus, *bmBc*[*G*] = 2 and the last character +1 of the window is *A*, thus *qsBc*[*A*] = 2. Both shift values are equal, we thus select the shift values obtained from the *qsBc* shift table. When shift values are obtained, the pattern is shifted to the right by 2 positions.



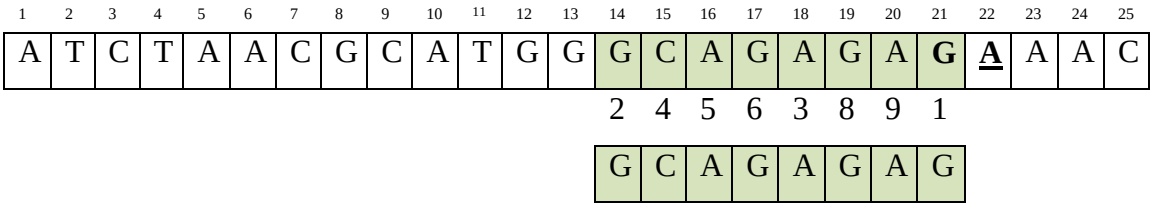
Second attempt:

Starts searching at the rightmost of both window and pattern which a mismatch is found. After mismatch found, the last character of the window is *A*, the *bmBc*[*A*] value is set to 1. The forward character of window is given as *T*, and *qsBc*[*T*] value is set to 9. We select the shift value taken from *qsBc* in order to shift the pattern to the right by 9 positions.



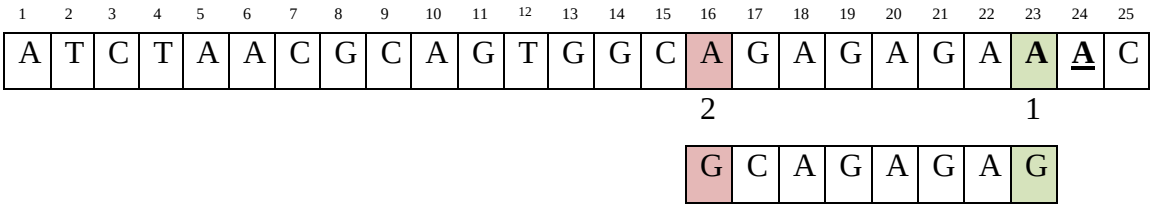
Third attempt:

The search begins at the rightmost position of the window and pattern and found a match. Next, a comparison is made at the leftmost position and found a match. Next, a comparison is made at the middle position and found a match. Then, a comparison is made at the leftmost+1 position and found a mismatch. After the mismatch is found for the shift value obtained from the *bmBc* shift table and the *qsBc* shift table, the shift value of the last character and last character +1 of the window are considered. In this example, the last character of the window is *G*. Thus, *bmBc*[*G*] = 2 and the last character +1 of the window is *A*, thus *qsBc*[*A*] = 2. Both shift values are equal, we thus select the shift values obtained from the *qsBc* shift table. When shift values are obtained, the pattern is shifted to the right by 2 positions.



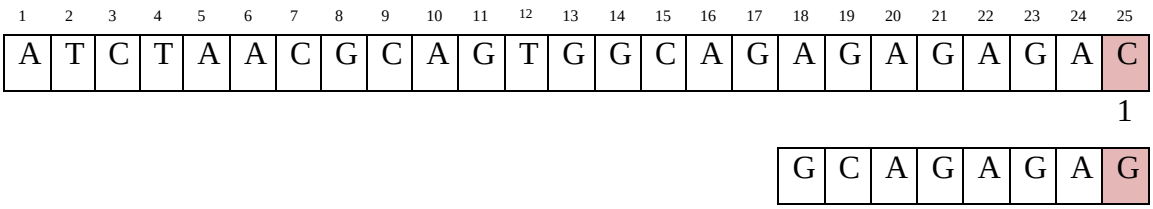
Fourth attempt:

The search begins at the rightmost position of the window and pattern and found a match. Next, a comparison is made at the leftmost position and found a match. Next, a comparison is made at the middle position and found a match. Then, a comparison is made at the leftmost+1 to the rightmost-1 position and found an exact match. After an exact mismatch is found for the shift value obtained from the *bmBc* shift table and the *qsBc* shift table, the shift value of the last character and last character +1 of the window are considered. In this example, the last character of the window is *G*. Thus, *bmBc*[*G*] = 2 and the last character +1 of the window is *A*, thus *qsBc*[*A*] = 2. Both shift values are equal, we thus select the shift values obtained from the *qsBc* shift table. When shift values are obtained, the pattern is shifted to the right by 2 positions.



Fifth attempt:

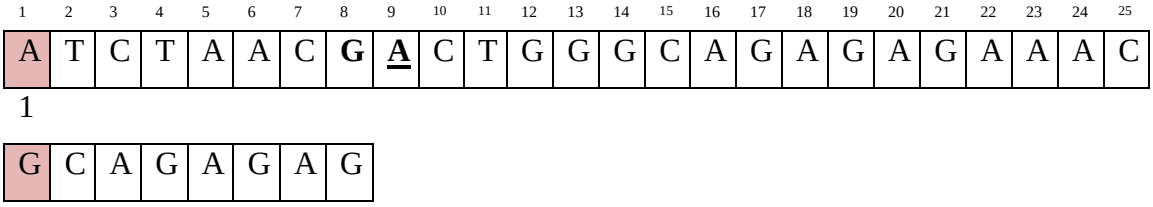
The search begins at the rightmost position of the window and pattern and found a match. Next, a comparison is made at the leftmost position and found a mismatch. After the mismatch is found for the shift value obtained from the *bmBc* shift table and the *qsBc* shift table, the shift value of the last character and last character +1 of the window are considered. In this example, the last character of the window is *A*. Thus, *bmBc*[*A*] = 1 and the last character +1 of the window is *A*, thus *qsBc*[*A*] = 2. We select the shift values obtained from the *qsBc* shift table. When shift values are obtained, the pattern is shifted to the right by 2 positions.



3.2.2 The Reverse Hybrid Max Shift Algorithm (RHMS)

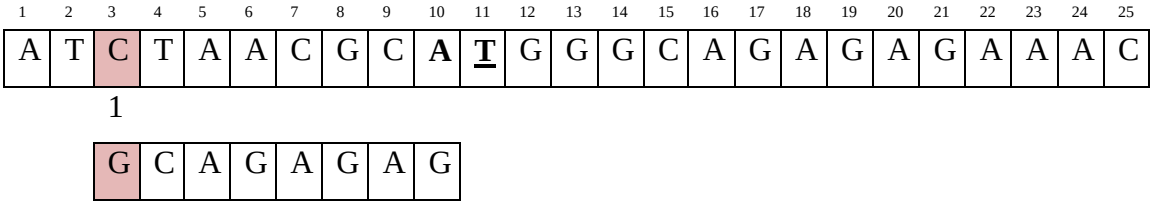
Apart from the HMS algorithm previously mentioned, this research tries to develop the alternative exact method based on HMS algorithm, namely the reverse hybrid max shift algorithm (RHMS). RHMS is switched the sequence comparison by firstly, running the comparison at the leftmost position. Then, RHMS runs the same procedure as HMS at the rightmost position. If a match is found, a comparison of the central position will be next in order. After that, comparisons of the leftmost+1 position to the rightmost-1 position will be done.

For example



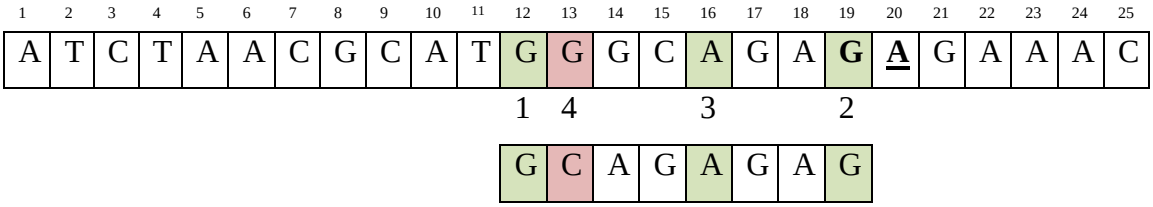
First attempt:

Starts searching at the leftmost of both window and pattern which a mismatch is found. After mismatch found, the last character of the window is G, the *bmBc*[G] value is set to 2. The forward character of window is given as *A*, and *qsBc*[*A*] value is set to 2. Both shift values are equal, we thus select the shift values obtained from the *qsBc* shift table. When shift values are obtained, the pattern is shifted to the right by 2 positions.



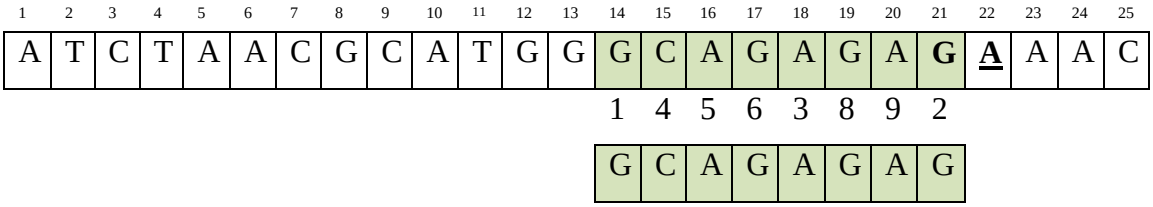
Second attempt:

The search begins at the leftmost position of the window and pattern and found a mismatch. After the mismatch is found for the shift value obtained from the *bmBc* shift table and the *qsBc* shift table, the shift value of the last character and last character +1 of the window are considered. In this example, the last character of the window is *A*. Thus, *bmBc*[*A*] = 1 and the last character +1 of the window is *T*, thus *qsBc*[*T*] = 9. We select the shift value taken from *qsBc* in order to shift the pattern to the right by 9 positions.



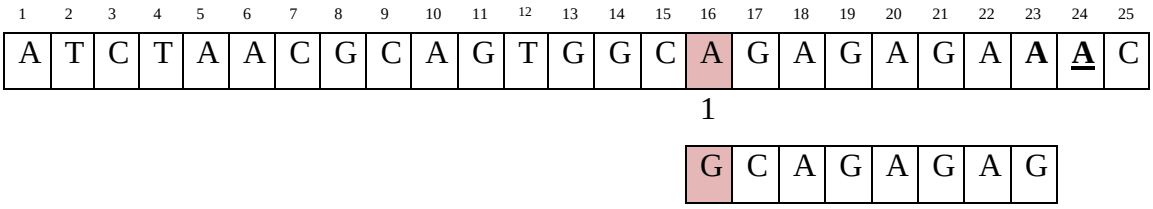
Third attempt:

The search begins at the leftmost position of the window and pattern and found a match. Next, a comparison is made at the rightmost position and found a match. Next, a comparison is made at the middle position and found a match. Next, a comparison is made at the leftmost+1 position and found a mismatch. After the mismatch is found for the shift value obtained from the *bmBc* shift table and the *qsBc* shift table, the shift value of the last character and last character +1 of the window are considered. In this example, the last character of the window is *G*. Thus, *bmBc*[*G*] = 2 and the last character +1 of the window is *A*, thus *qsBc*[*A*] = 2. Both shift values are equal, we thus select the shift values obtained from the *qsBc* shift table. When shift values are obtained, the pattern is shifted to the right by 2 positions.



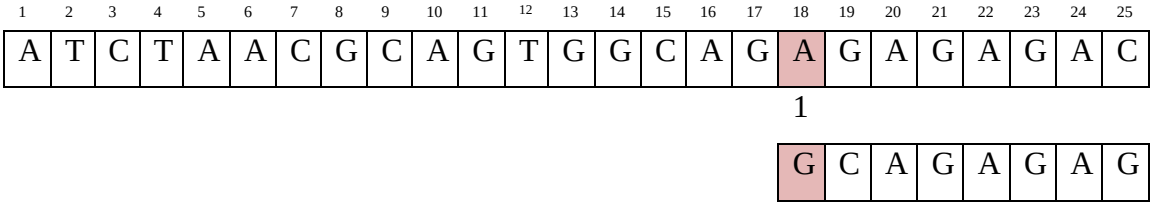
Fourth attempt:

The search begins at the leftmost position of the window and pattern and found a match. Next, a comparison is made at the rightmost position and found a match. Next, a comparison is made at the middle position and found a match. Then, a comparison is made at the leftmost+1 to the rightmost-1 position and found an exact match. After an exact mismatch is found for the shift value obtained from the *bmBc* shift table and the *qsBc* shift table, the shift value of the last character and last character +1 of the window are considered. In this example, the last character of the window is *G*. Thus, *bmBc*[*G*] = 2 and the last character +1 of the window is *A*, thus *qsBc*[*A*] = 2. Both shift values are equal, we thus select the shift values obtained from the *qsBc* shift table. When shift values are obtained, the pattern is shifted to the right by 2 positions.



Fifth attempt:

Starts searching at the leftmost of both window and pattern which a mismatch is found. After mismatch found, the last character of the window is A, the *bmBc*[A] value is set to 1. The forward character of window is given as *A*, and *qsBc*[*A*] value is set to 2. We select the shift values obtained from the *qsBc* shift table. When shift values are obtained, the pattern is shifted to the right by 2 positions.



3.3 Implementation of Algorithm Design

The source codes of all algorithms used for development is taken from SMART tool, developed by Faro S and Lecroq T [3]. This tool consists of sample source codes of more than 80 pattern matching algorithms. It has been recognized as the standard framework for researchers working on the pattern matching problems. In addition, our algorithms are developed by SMART tool as a reference.

The implementation process is divided into two main phases, including the preprocessing phase and searching phase, as described in session 3.2.

The preprocessing phase is pre-processes characters to create shift values. After shift values are obtained, these values will be kept in the shift tables.

The searching phase will divide the process into two processes, which are the searching process and the shifting process.

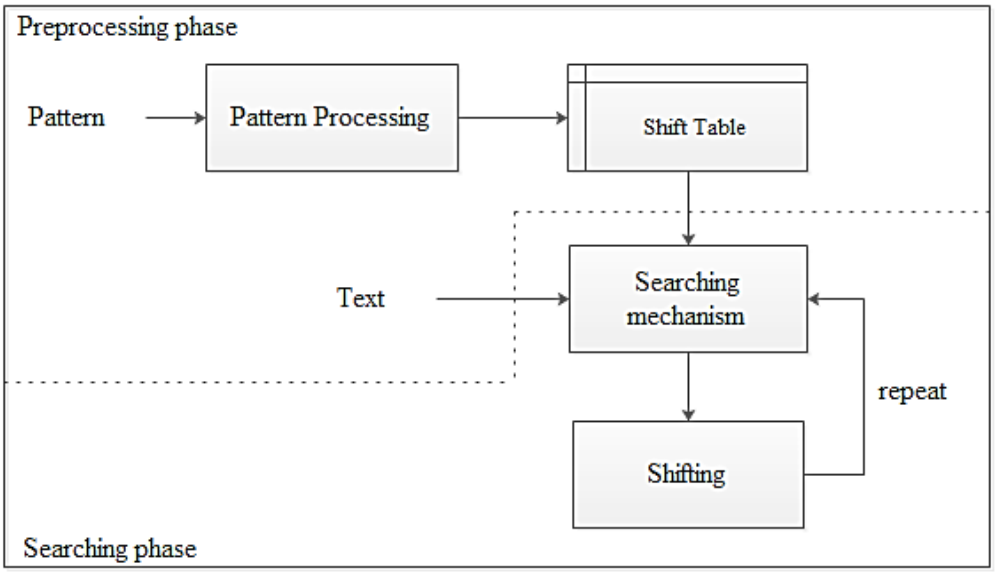


Figure 3.7 The preprocessing and searching phase diagram.

3.3.1 The pre-processing phase

This phase calculates the shift value of each character in the pattern and stores the values in the shift table. In our proposed algorithms, there are two shift tables consisting of the shift value of each character of the pattern. The shift values of each table are calculated from the difference evaluation function. Inside of the *bmBc*

shift table is derived from the shift value of the *bmBc* shift function. The inside of *qsBc* shift table is derived from the shift value of *qsBc* shift function. After performing this phase, all shift values will be used to shift the pattern in the searching phase again.

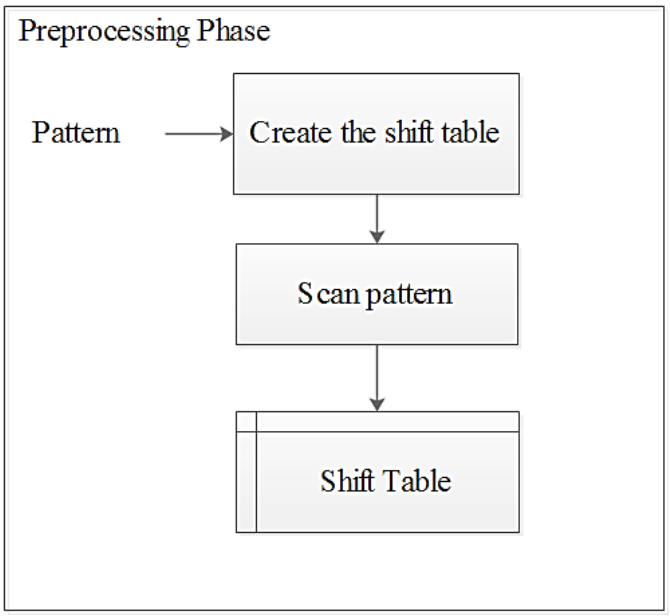


Figure 3.8 Pre-processing phase diagram.

3.3.2 The searching phase

The first step is to scan the pattern for use in calculating the shift values of the *bmBc* shift function and *qsBc* shift function. The next step consists of the creation of the *bmBc* shift table and *qsBc* shift table in order to keep shift values. The next step compares characters between the text and pattern according the order of comparisons of each algorithm. If an exact match or mismatch is found, the shift value obtained from the character will be checked with the rightmost position for the *bmBc* shift function and the rightmost+1 position +1 for the *qsBc* shift function. The pattern will be shifted to the right by using the maximum shift values obtained from two functions. After pattern shifting, if the end of file is not found, continue running process until the end of file. Once the end of file is found, report the results on screen.

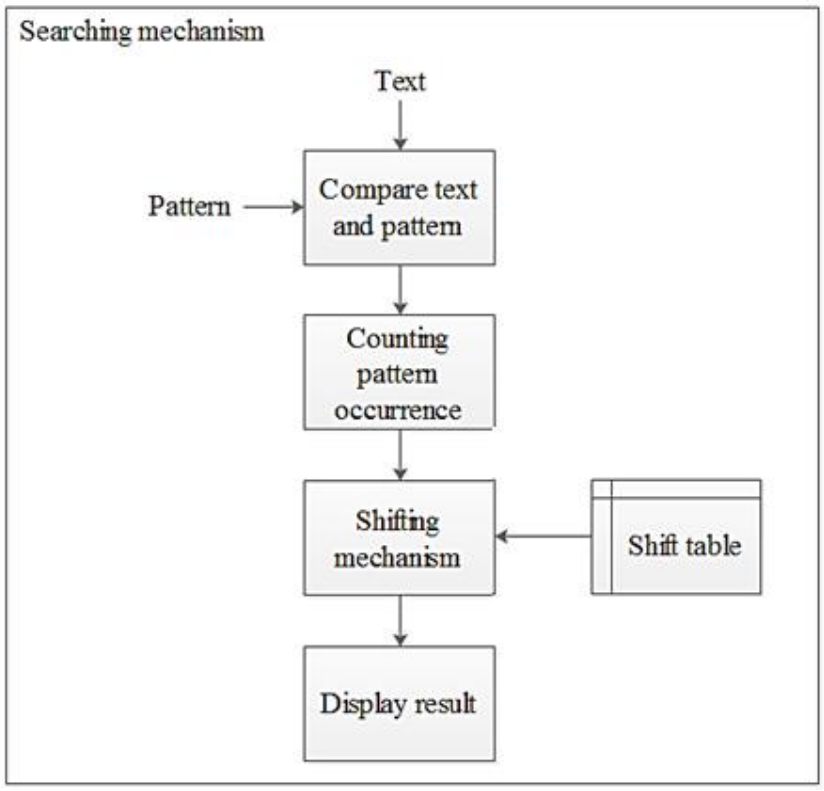


Figure 3.9 Searching process diagram.

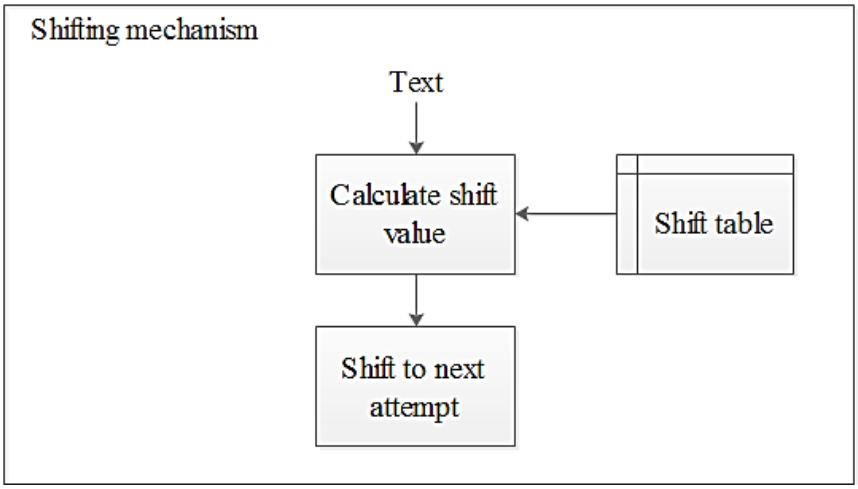


Figure 3.10 Shifting process diagram.

3.3.3 The process of algorithm

First, doing a pattern scanning to bring it out for calculating the shift value of the *bmBc* shift function, the *qsBc* shift function. The next step, create the *bmBc* shift table, the *qsBc* shift table for bringing the shift values to store. The next step, doing a comparison of the characters between the text and pattern according to the sequence comparison of each algorithm. If an exact match or mismatch are found, check out the shift value from the character at the rightmost for the *bmBc* shift function, also, the rightmost+1 for the *qsBc* shift function, shifting the pattern to the right by using the maximum values from the both functions. After shifting the pattern, if the end of the file hasn't been found, continue doing it until the end of file is found. Once the end of file is found, make it display on the screen.

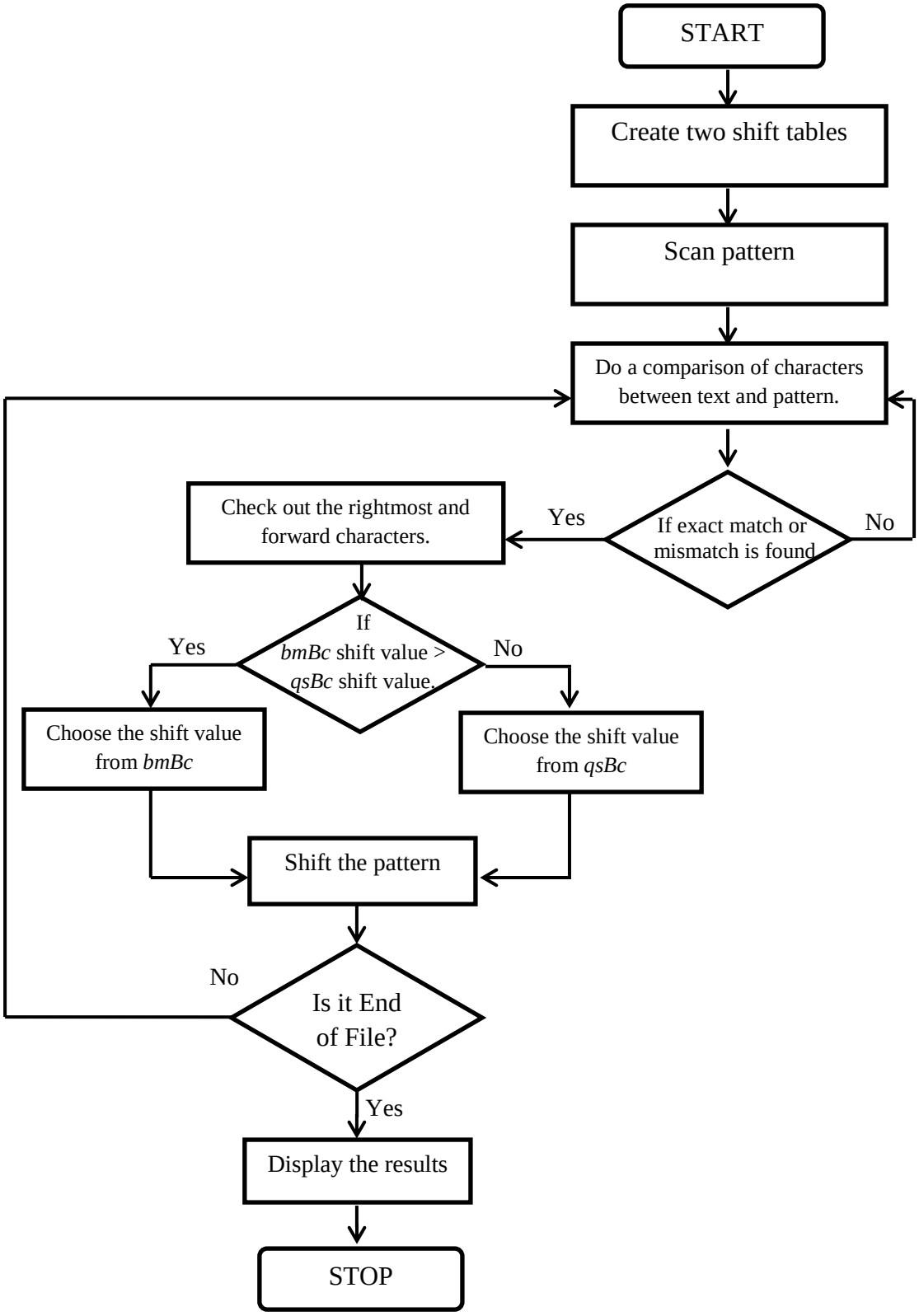


Figure 3.11 Flowchart of algorithm processes.

3.4 Evaluation

There are four different types of data set that are used to evaluate the proposed algorithms. These four data sets are downloaded from the SMART Tool corpus (http://www.dmi.unict.it/~faro/smart/smart11.06_data.zip). These data types involve:

1. English text are prepared from the English King James version of the Bible and the CIA World Fact Book. The alphabet size (σ) is 94 and data size is 6.1 MB.

2. Genome sequence is prepared from the genome sequence of 4,638,690 base pairs of *Escherichia coli.* The alphabet size (σ) is 4 (σ = {*A,G,C,T*}) and data size is 4.4 MB.

3. Protein sequence is prepared from the protein sequence of 3,295,751 bytes of the *Saccharomyces cerevisiae* genome. The alphabet size (σ) is 20 (σ = {*A,C,D,E, F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y*}) and data size is 3.1 MB.

4. Random text is prepared from a random text over an alphabet with a uniform distribution. The alphabet sizes (σ) are 2, 4, 8, 16, 32, and 64. Data size is 5 MB.

The reasons for testing the data are as follows.

- Most of the researchers have used the data to examine the effect of any function in both the developed of algorithms and applications. Therefore, it is considered as the standardized data.
- The collection of data are diversity in terms of the characters' size. Therefore, it would help to evaluate the performances of algorithms in many situations.

CHAPTER IV
RESULTS

Context in this chapter consists of the research result of pattern matching algorithm. The result from the test of proposed algorithms was compared to existing algorithm which this research focus on. Researcher used 4 types of standard data to evaluate, which were English text, amino acid, protein sequence and uniform distribution random text. The data used for the test can be found on the SMART TOOL website.

SMART TOOL is a tool that develops by Faro S and Lecroq T (http://www.dmi.unict.it/~faro/smart/index.php). This tool is standard framework for the research who work on string matching. This tool helps user to test, design, assess and understand the work process of string matching in easier way. Apart from that, this tool included source code of pattern matching algorithm, with more than 80 algorithms, and included data that could be selected to use. The source code of algorithm was developing by use C language. Tool was designed for the researcher to use at ease. It can display many algorithm test results in the same time, which will provide a better result display. Apart from collected algorithms, researchers could also develop their own algorithm to perform test with this tool.

Smart tool created all pattern sequence and pattern that were made had different length as 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024 characters (m). Times of work were calculated 500 times during the test and display in average. Average of work displayed in millisecond unit.

Our experiments, there are 500 patterns randaomly created. The lengths of created patterns were 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024 characters. The test results are divided into 2 types that are 1. Average time used to run the algorithm in millisecond unit. 2. Amount of patterns of occurrence. This result shows the accuracy of work with proposed algorithm compare to existing algorithms. Both of test results are displayed in Tables 4.3-4.10 and Figures 4.4-4.11.

We brought the results to calculate increasing speed up in order to compare between existing algorithms and our proposed algorithm. Test results displayed in Figure 4.12-4.27.

The speed up formulation is given as:

$S = T_{old} / T_{new,}$

where;

S is the resultant speedup,

T_{old} is the old execution time,

T_{new} is the new execution time.

Moreover, we take the results to calculate statistics for data analysis and check the validity and reliability of algorithm before actual use. Test results displayed in Tables 4.12-4.27. The used statistics show below:

1. Measure of central value: In measurement of trend to central or mean value that represent data of this research used arithmetic mean to analyzed.

The mean formulation is given as:

$$\bar{X} = \frac{\sum_{i=1}^{n} xi}{n}$$

When x_i is observation value of i data and n substituted amount of sample data.

2. Measure of dispersion: This is statistic that calculates into number to explain characteristic of data dispersion. When data set consisted of different value, it is called data dispersion. If data set consisted of many different values, it is called large data dispersion. If data set consisted of little different values, we called little data dispersion, and if such value in a data set consisted of equal values, we called non data dispersion. There are 2 types of measurement of dispersion which are:

2.1 Absolute Variation

The standard variation formulation is given as:

$$SD = \sqrt{\frac{(x-\bar{X})^2}{n-1}}$$

Where *SD* is standard deviation;

x is data (1, 2, 3…, *n*);

$\bar{x}$ is arithmetic mean;

n is total amount of data.

2.2 Relative Variation is measurement of more than 1 data set dispersion, by used average absolute variation with mean of data to compare the dispersion.

The *CV* formulation is given as:

$$CV\% = \frac{SD}{\bar{x}} \times 100$$

When *CV* is coefficient of variation;

SD is standard deviation;

$\bar{x}$ is arithmetic mean .

The best performance (fastest running time) were bring to presented in colored cell of each table. In the last row of the table displayed amount of the pattern of occurrence found. Displayed of test result use initial name for easy understanding as follows: Boyer Moore Horspool, Quick Search, Hybrid Max Shift, Reverse hybrid max shift use as follow initial respectively: BMH, QS, HMS and RHMS. These initial will be used as a part in this chapter.

4.1 Analysis

The time complexity of our proposed algorithm in pre-processing phase consists of creating preprocessing phase of two existing algorithms. Reason of the two existing algorithms that we applied concept in pre-processing phase have equal pre-processing time complexity which is O(m+ σ), so the time complexity of our proposed algorithm the pre-processing phase is based on the two existing algorithms' the time complexity so the time complexity is equal to O(m+ σ), where σ represents the alphabet size, and the space complexity is O(σ). The following section describes the searching phase's time complexity.

Lemma 1: The time complexity is O(n/m+1) in the best case.

Proof: The best case of the proposed algorithm occurs when all the characters of the pattern are totally different, compared to the characters in the text. In the preprocessing phase, the Quick search function (*qsBc*) is calculated. If every character does not occur in the pattern then the shift value is equal to m+1. This is because, the *qsBc* [m+1] is always greater than *bmBc*[m]. Therefore, the time complexity is O(n/m+1).

For example:

Text = AAAAAAAAAAAAAAAAAAAAAAAA

Pattern = BBBBBBB

The text length (n) = 25 and the pattern length (m) = 7.

The alphabet set (Σ) = {A, B} of size (σ) = 2.

Lemma 2: The time complexity is O(mn) in the worst case.

Proof: The worst case of the proposed algorithm occurs when all the characters of the pattern and the characters of the window text are all matched at each attempt. The worst case can be known when all the characters in the pattern are the same as characters in the window text. In this case, the shift value is equal to 1 position at each time of the comparison. As stated, all the characters in the text are matched hardly than m times and the entire comparisons for n characters of the text cannot be exceed mn so the time complexity is O(mn).

For example:

Text = AAAAAAAAAAAAAAAAAAAAAAAA

Pattern = AAAAAAA

The text length (n) = 25 and the pattern length (m) = 7.

The alphabet set (Σ) = {A} of size (σ) = 1.

In the proposed algorithm, the average time complexity cannot be defined strictly due to the complexity relies on the size of the alphabet and the possible occurrence of every single character in the text. According to the maximum of the proposed algorithm, can be accomplished as m+1, and the minimum would be one. Hence, the character comparisons could be between 1 to m that is random based on the input data. From that reason, the average time complexity cannot be predicted in this case.

Table 4.1 Comparison of time complexity of algorithms.

Algorithm	**Preprocessing**		**Searching Time Complexity**	
	Time Complexity	Space Complexity	Worst Case	Best Case
Horspool	O(m+σ)	O(σ)	O(mn)	O(n/m)
Quick Search	O(m+σ)	O(σ)	O(mn)	O(n/m)
Raita	O(m+σ)	O(σ)	O(mn)	O(n/m)
HMS	O(m+ σ)	O(σ)	O(mn)	O(n/m)
RHMS	O(m+ σ)	O(σ)	O(mn)	O(n/m)

4.2 The Data Types Samples

Group of data that we use are given as: English text, genome sequence, protein sequence and uniform distribution random text as shown in Table 4.1.

Table 4.2 Types of text file used during the experiment.

NO.	Data types	Description	Alphabet Size (σ)	File Size (MB)
1	English text	The English King James version of the Bible and the CIA World Fact Book	94	6.1
2	Genome sequence	Escherichia coli.	4	4.4
3	Protein sequence	Saccharomyces cerevisiae genome	20	3.1
4	Random text	Random text over an alphabet with a uniform distribution	4, 8, 16, 32 ,64	5

The sample file data used are shown in Figures 4.1-4.3, which used data are collected from SMART TOOL. All data can be downloaded from. (http://www.dmi.unict.it/~faro/smart/smart11.06_data.zip)

In the beginning God created the heaven and the earth. And the earth was without form, and void; and darkness was upon the face of the deep. And the Spirit of God moved upon the face of the waters.
And God said, Let there be light: and there was light.
And God saw the light, that it was good: and God divided the light from the darkness.
And God called the light Day, and the darkness he called Night. And the evening and the morning were the first day.
And God said, Let there be a firmament in the midst of the waters, and let it divide the waters from the waters.

Figure 4.1 Example of text file of English text.

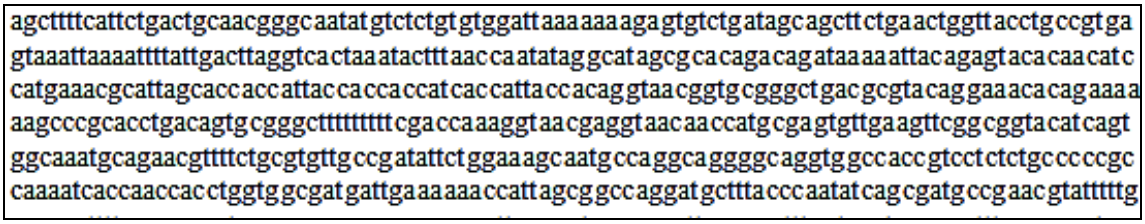


Figure 4.2 Example of text file of genome sequence file.

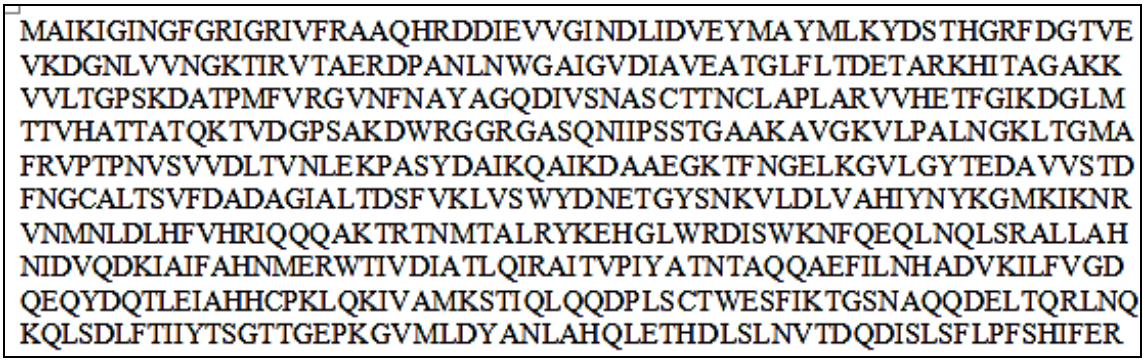


Figure 4.3 Example of text file of protein sequence file.

4.3 Experimental Results on English Text

Table 4.3 Average running times of algorithms on English text (ms.).

Algorithsms / Pattern Lengths (m)	BMH	QS	Raita	HMS	RHMS	NO. of pattern occurrence
2	8.08	6.25	7.24	6.62	6.95	10203
4	5.54	4.88	4.83	4.85	5.13	2106
8	4.05	3.76	3.63	3.64	3.76	136
16	3.33	3.09	3.03	2.98	3.01	5
32	2.95	2.76	2.72	2.66	2.68	1
64	2.72	2.58	2.56	2.50	2.52	1
128	2.56	2.46	2.44	2.37	2.40	1
256	2.46	2.38	2.36	2.31	2.32	1
512	2.40	2.32	2.31	2.26	2.27	1
1024	2.34	2.28	2.27	2.23	2.24	1

English text data set is with alphabet set of 94 characters. Result of work show in Table 4.3 and describe as follow.

When length of pattern (m) ≤ 8, existing algorithm works faster than proposed algorithm. When length of patter (m) > 8, HMS algorithm works at fastest than other algorithms, while RHMS algorithm was second fastest. Running times used are slightly different. Results of pattern occurrence found are shown in Table 4.3. All five algorithms for our experiment use the same pattern occurrence value, which mean proposed algorithm can be worked accurately as well as existing algorithm. The test results of Table 4.3 are demonstrated in Figure 4.4.

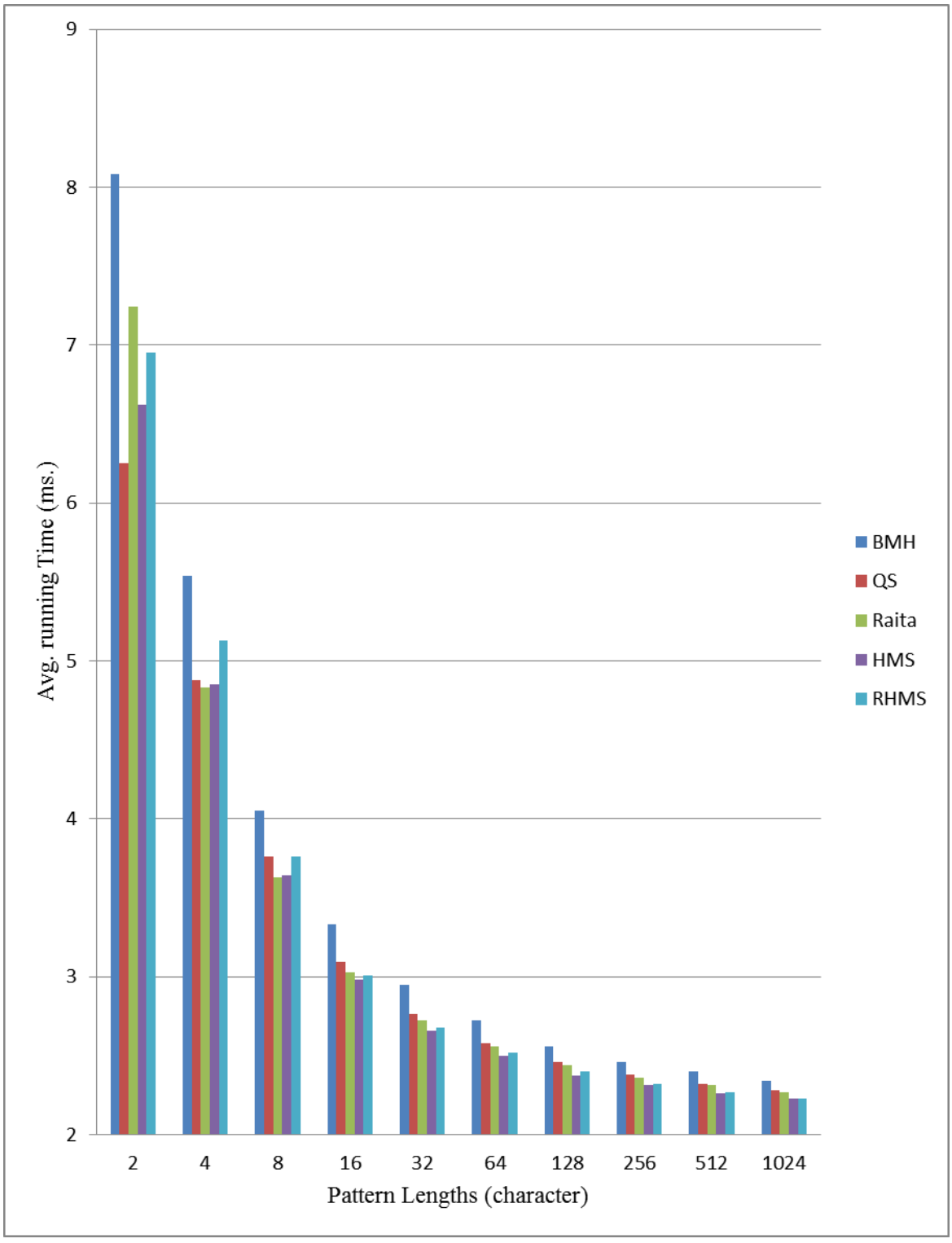


Figure 4.4 Average running times of algorithms on English text.

4.4 Experimental Results on Genome Sequence

Table 4.4 Average running times of algorithms on genome sequence (ms.).

Algorithms / Pattern Lengths (m)	BMH	QS	Raita	HMS	RHMS	NO. of pattern occurrence
2	10.51	9.93	10.19	9.85	10.96	66828
4	9.02	8.06	7.38	7.42	7.84	4571
8	7.44	6.57	6.09	6.01	5.92	27
16	7.23	6.21	5.72	5.64	5.39	1
32	7.18	6.10	5.58	5.49	5.22	1
64	7.35	6.13	5.69	5.57	5.30	1
128	7.09	6.16	5.65	5.56	5.28	1
256	7.12	6.23	5.69	5.61	5.33	1
512	7.00	6.13	5.59	5.50	5.23	1
1024	7.12	6.17	5.68	5.57	5.30	1

The data set contains four alphabets (nucleotides). It is Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). The alphabet size of this data set is equal to 4 ($\sigma = 4$). All the average time of algorithms are listed in Table 4.4. The results are shown as follows:

- When m = 2, the proposed algorithm of HMS is fastest algorithm.
- When m = 4, the Raita algorithm is faster than proposed algorithms.
- When m > 8, the proposed algorithm of RHMS is the fastest algorithm.

The HMS is the second best, which is much slower than RHMS.

The experimental results in Table 4.4 are also plotted as a chart in Figure 4.5.

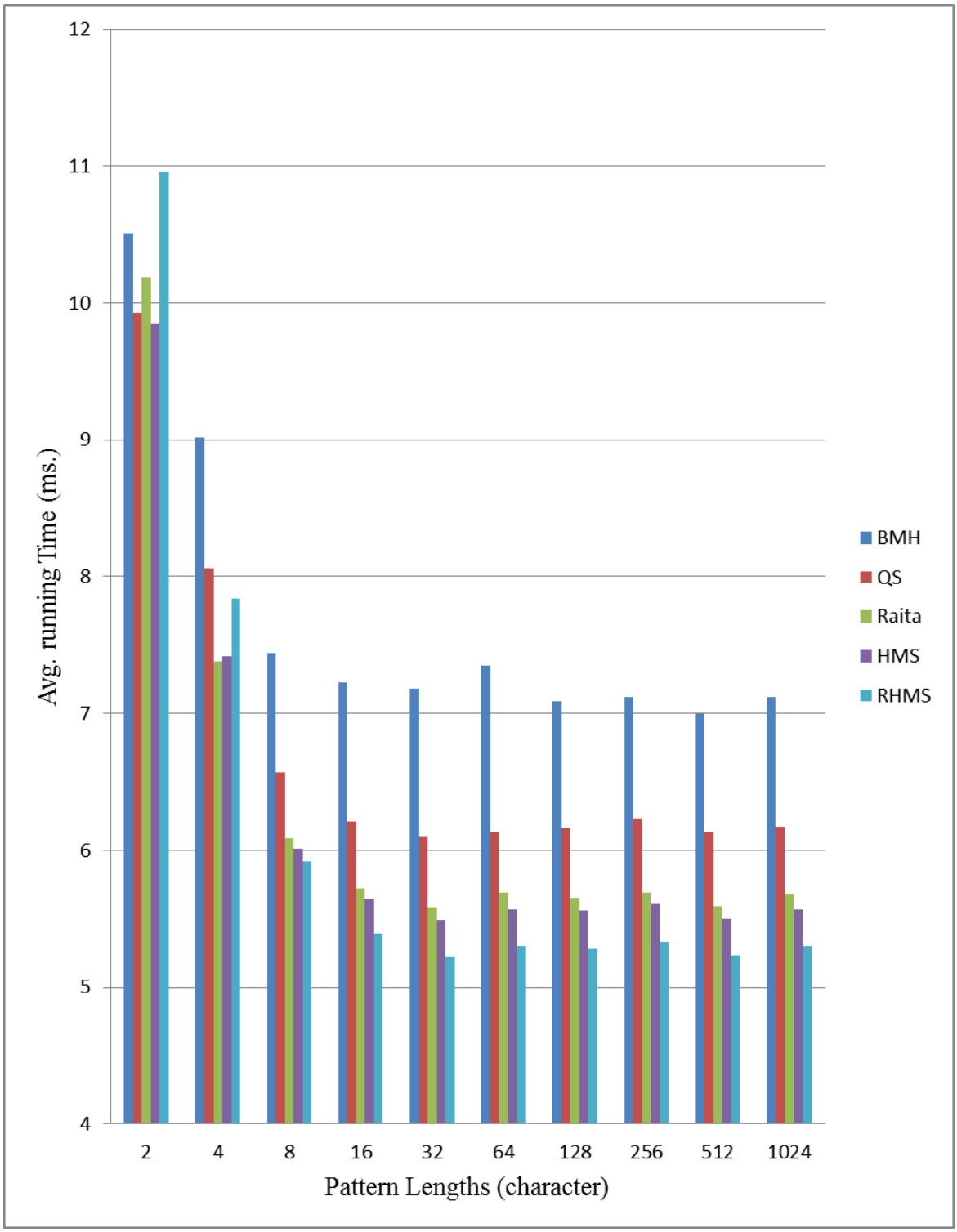


Figure 4.5 Average running times of algorithm on genome sequence.

4.5 Experimental Results on Protein Sequence

Table 4.5 Average running times of algorithms on protein sequence (ms.).

Algorithms / Pattern Lengths (m)	BMH	QS	Raita	HMS	RHMS	NO. of pattern occurrence
2	7.78	5.97	6.92	6.42	6.63	3991
4	5.25	4.49	4.61	4.68	4.79	18
8	3.81	3.50	3.45	3.53	3.58	1
16	3.15	2.96	2.91	2.88	2.94	1
32	2.83	2.69	2.66	2.64	2.65	1
64	2.71	2.58	2.56	2.53	2.55	1
128	2.63	2.53	2.51	2.47	2.49	1
256	2.62	2.51	2.49	2.45	2.47	1
512	2.61	2.51	2.50	2.45	2.48	1
1024	2.61	2.51	2.49	2.46	2.48	1

The data set contains twenty alphabets (amino acid). The alphabet set is defined as Σ = A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y and V. Hence, the alphabet size of this data set is equal to twenty ($\sigma = 20$). All the average time of algorithms are listed in Table 4.5. The results show the following:

- When $m < 16$, the existing algorithm of QS is faster than both of proposed algorithms.
- When $m \geq 16$, the proposed algorithm of HMS is the fastest algorithm among them. The RHMS is the second best, which is slightly slower than HMS.

The experimental results in Table 4.5 are plotted as a chart in Figure 4.6.

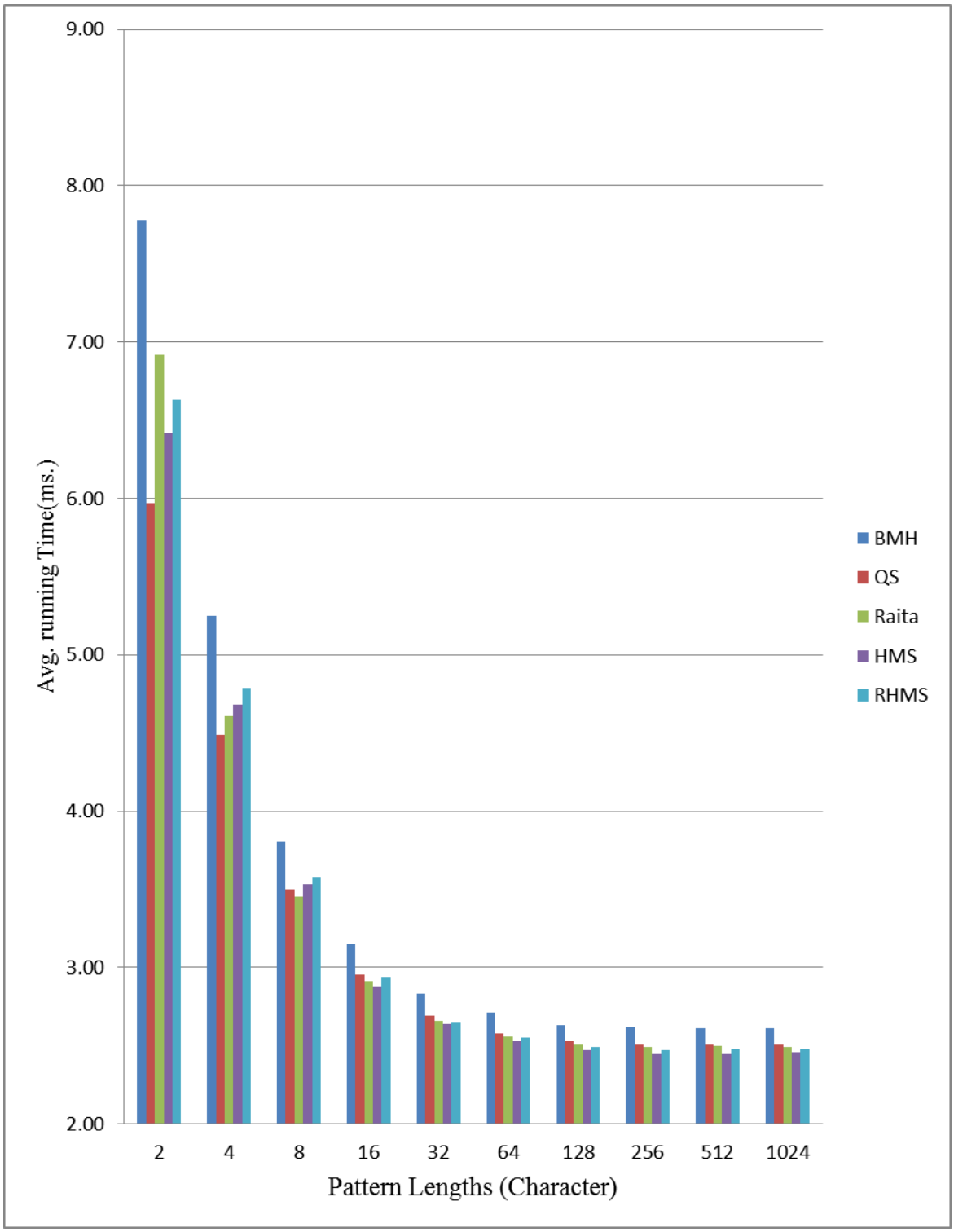


Figure 4.6 Average running times of algorithms on protein sequence.

4.6 Experimental Results on Rand4

Table 4.6 Average running times of algorithms on Rand4 (ms.).

Algorithms / Pattern Lengths (m)	BMH	QS	Raita	HMS	RHMS	NO. of pattern occurrence
2	11.00	9.96	10.18	9.67	10.82	65536
4	9.09	7.98	7.37	7.38	7.72	4095
8	7.28	6.60	6.10	6.02	5.90	16
16	6.92	5.83	5.72	5.62	5.36	1
32	7.02	6.15	5.73	5.63	5.36	1
64	6.76	6.12	5.59	5.51	5.20	1
128	6.78	6.14	5.60	5.52	5.26	1
256	6.88	6.17	5.71	5.61	5.33	1
512	6.85	6.21	5.68	5.60	5.32	1
1024	6.81	6.20	5.65	5.60	5.29	1

The data set contains four alphabets in their set, so the alphabet sizes is equal to four ($\sigma = 4$). All the average time of algorithms are listed in Table 4.6. The results are shown as follows:

- When m = 2, the proposed algorithm of HMS is fastest algorithm.
- When m = 4, the existing algorithm of Raita is faster than both of proposed algorithms.
- When m > 8, the proposed algorithm of RHMS is the fastest algorithm among them. The HMS is the second best, which is much slower than RHMS.

The experimental results in Table 4.6 are plotted as a chart in Figure 4.7.

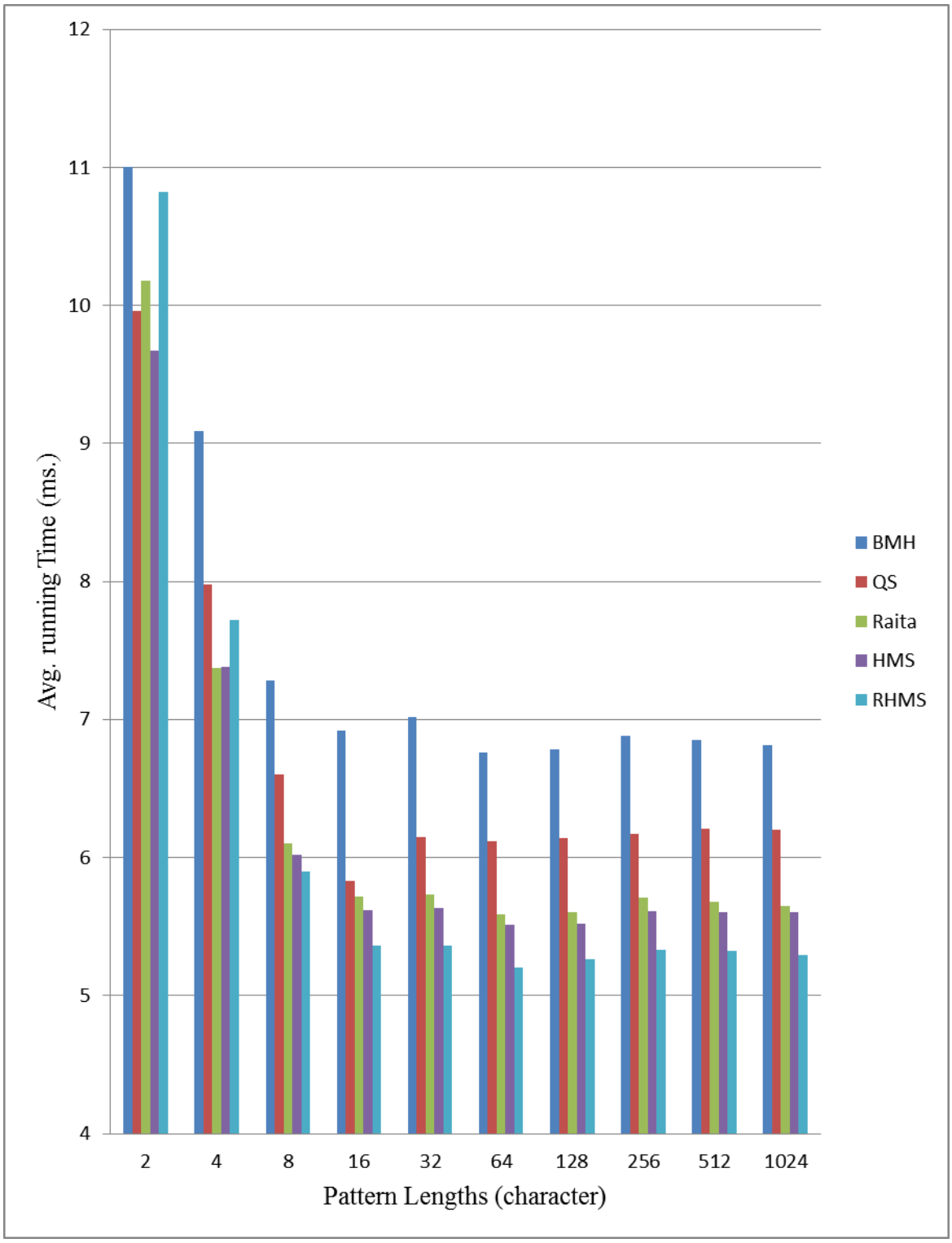


Figure 4.7 Average running times of algorithms on Rand4.

4.7 Experimental Results on Rand8

Table 4.7 Average running times of algorithms on Rand8 (ms.).

Algorithms / Pattern Lengths (m)	BMH	QS	Raita	HMS	RHMS	NO. of pattern occurrence
2	8.97	7.31	7.84	7.31	7.76	16383
4	6.24	5.49	5.25	5.26	5.49	256
8	4.61	4.28	3.99	4.01	4.11	1
16	3.96	3.70	3.54	3.47	3.48	1
32	3.80	3.53	3.40	3.32	3.29	1
64	3.81	3.54	3.42	3.34	3.28	1
128	3.76	3.54	3.39	3.31	3.27	1
256	3.73	3.50	3.37	3.30	3.25	1
512	3.73	3.54	3.41	3.27	3.25	1
1024	3.75	3.53	3.38	3.31	3.26	1

The data set contains right alphabets in their set so the alphabet sizes is equal to eight ($\sigma = 8$). All the average time of algorithms are listed in Table 4.7. The results are shown as follows:

- When $m \leq 4$, both QS and the proposed algorithm of HMS are faster than other algorithms.
- When $4 < m < 16$, the proposed algorithm of HMS is the fastest algorithm.
- When $m \geq 16$, the proposed algorithms of RHMS is the fastest algorithm among them. The HMS is the second best, which is slightly slower than RHMS.

The experimental results in Table 4.7 are also plotted as a chart in Figure 4.8.

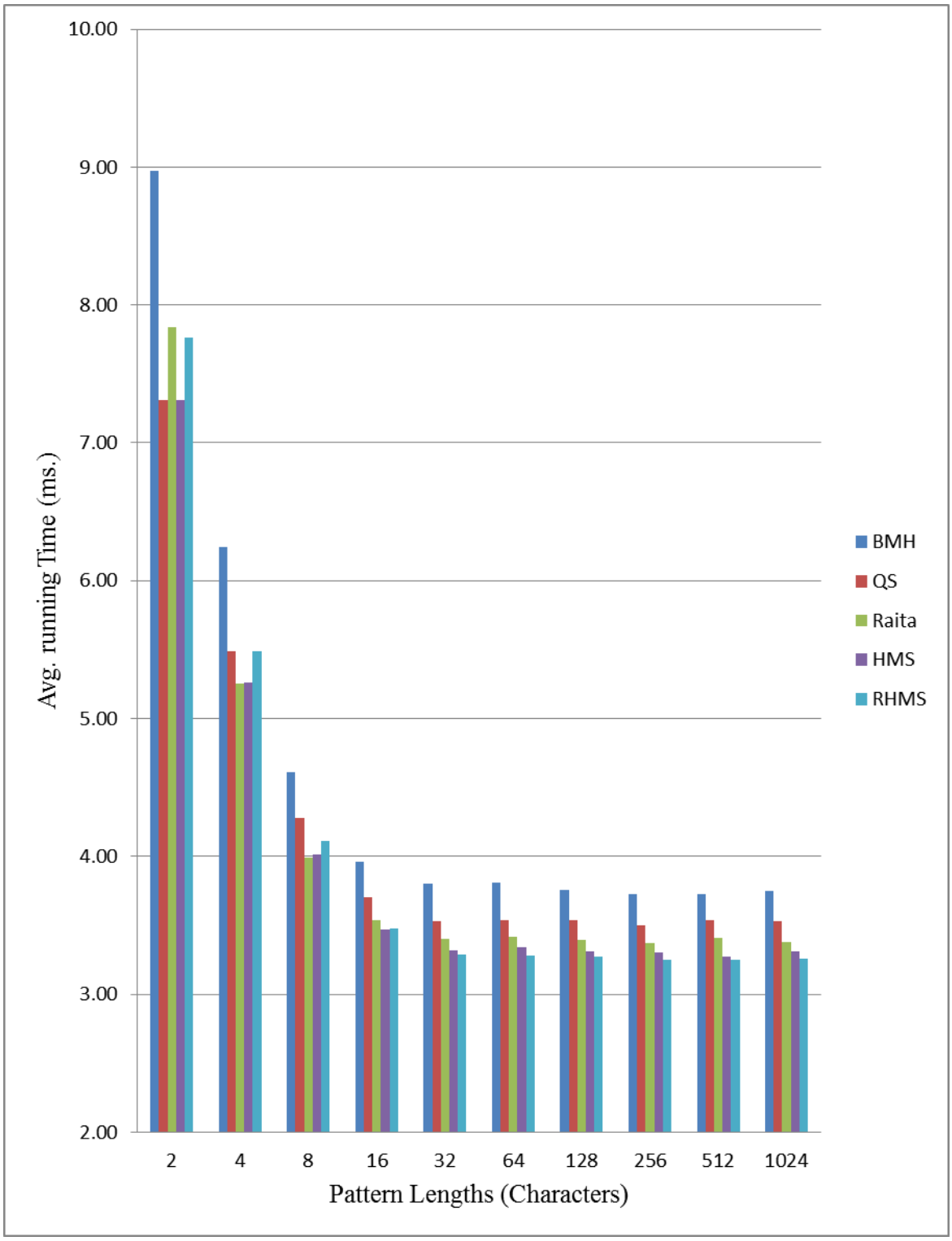


Figure 4.8 Average running times of algorithms on Rand8.

4.8 Experimental Results on Rand16

Table 4.8 Average running times of algorithms on Rand16 (ms.).

Algorithms / Pattern Lengths (m)	BMH	QS	Raita	HMS	RHMS	NO. of pattern occurrence
2	7.82	6.01	6.92	6.37	6.59	4092
4	5.20	4.52	4.60	4.65	4.78	16
8	3.78	3.53	3.46	3.52	3.58	1
16	3.13	2.99	2.93	2.92	2.95	1
32	2.86	2.76	2.72	2.68	2.70	1
64	2.77	2.68	2.65	2.61	2.63	1
128	2.76	2.68	2.65	2.60	2.62	1
256	2.76	2.67	2.64	2.60	2.62	1
512	2.77	2.68	2.65	2.61	2.63	1
1024	2.77	2.68	2.64	2.61	2.62	1

The data set contains sixteen alphabets in their set so the alphabet sizes is equal to sixteen ($\sigma = 16$). All the average time of algorithms are listed in Table 4.8. The results show the following:

- When $m \leq 8$, the existing algorithm of QS is faster than both of proposed algorithm.
- When $m > 8$, the proposed algorithm of HMS is the fastest algorithm. The RHMS is the second best, which is much slower than HMS.

The experimental results in Table 4.8 are plotted as a chart in Figure 4.9.

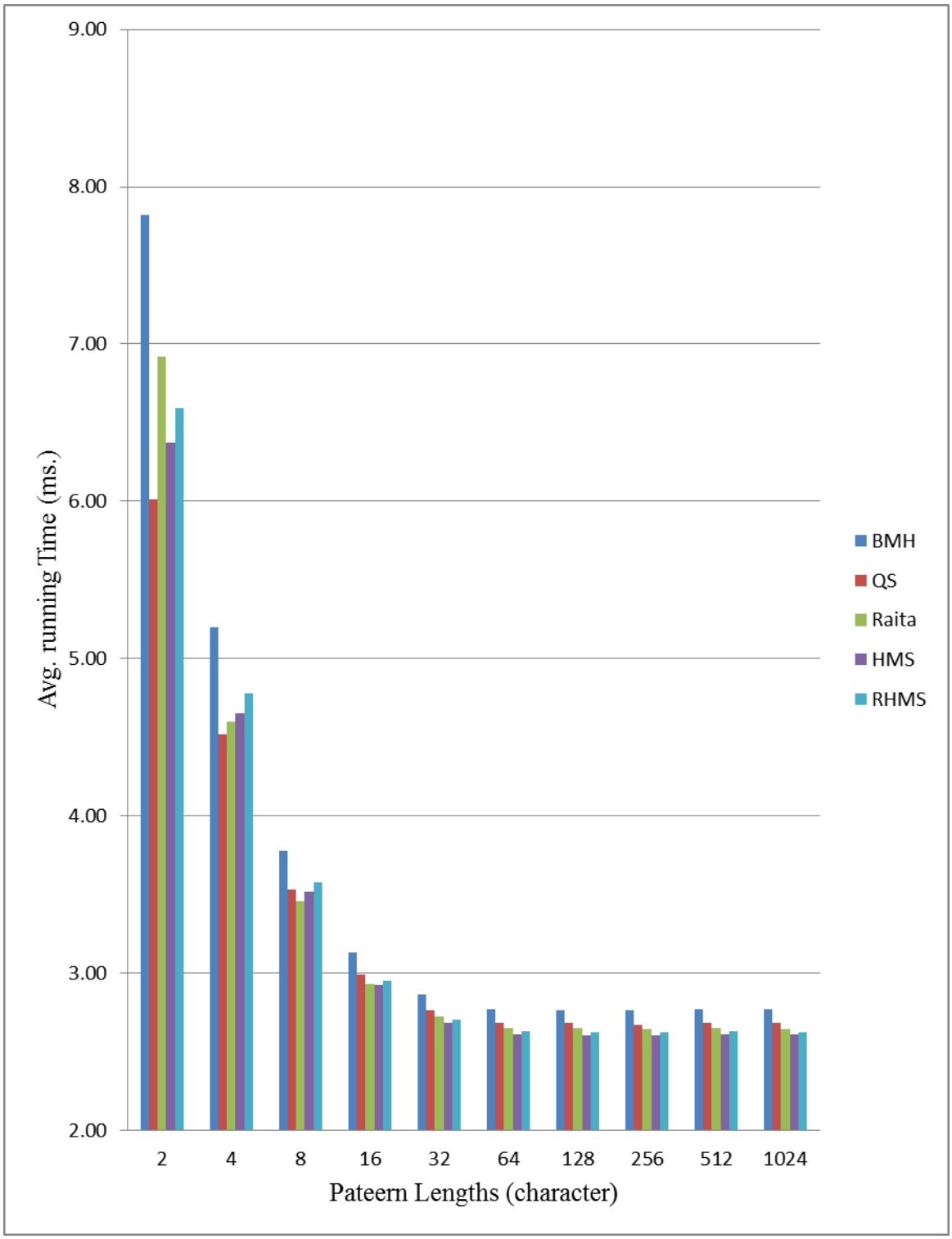


Figure 4.9 Average running times of algorithms on Rand16.

4.9 Experimental Results on Rand32

Table 4.9 Average running times of algorithms on Rand32 (ms.).

Algorithms / Pattern Lengths (m)	BMH	QS	Raita	HMS	RHMS	NO. of pattern occurrence
2	7.19	5.40	6.56	6.06	6.08	1026
4	4.75	4.08	4.34	4.43	4.48	1
8	3.45	3.21	3.24	3.36	3.39	1
16	2.82	2.71	2.70	2.75	2.77	1
32	2.53	2.47	2.46	2.47	2.48	1
64	2.43	2.38	2.38	2.36	2.40	1
128	2.38	2.33	2.34	2.32	2.36	1
256	2.38	2.34	2.33	2.32	2.35	1
512	2.38	2.33	2.33	2.32	2.34	1
1024	2.38	2.35	2.34	2.32	2.36	1

The data set contains thirty-two alphabets in their set so the alphabet sizes is equal to thirty-two ($\sigma = 32$). All the average time of algorithms are listed in Table 4.9. The results are shown as follows:

- When $m \leq 16$, the existing algorithms of QS and Raita are faster than both of proposed algorithm.
- When $m = 32$, the existing algorithms of Raita and the proposed Algorithms of HMS are faster than other algorithms.
- When $m \geq 32$, the proposed algorithm of HMS is the fastest algorithm among them. The existing algorithm of RHMS is the second best, which is slightly slower than HMS.

The experimental results in Table 4.8 are plotted as a chart in Figure 4.10.

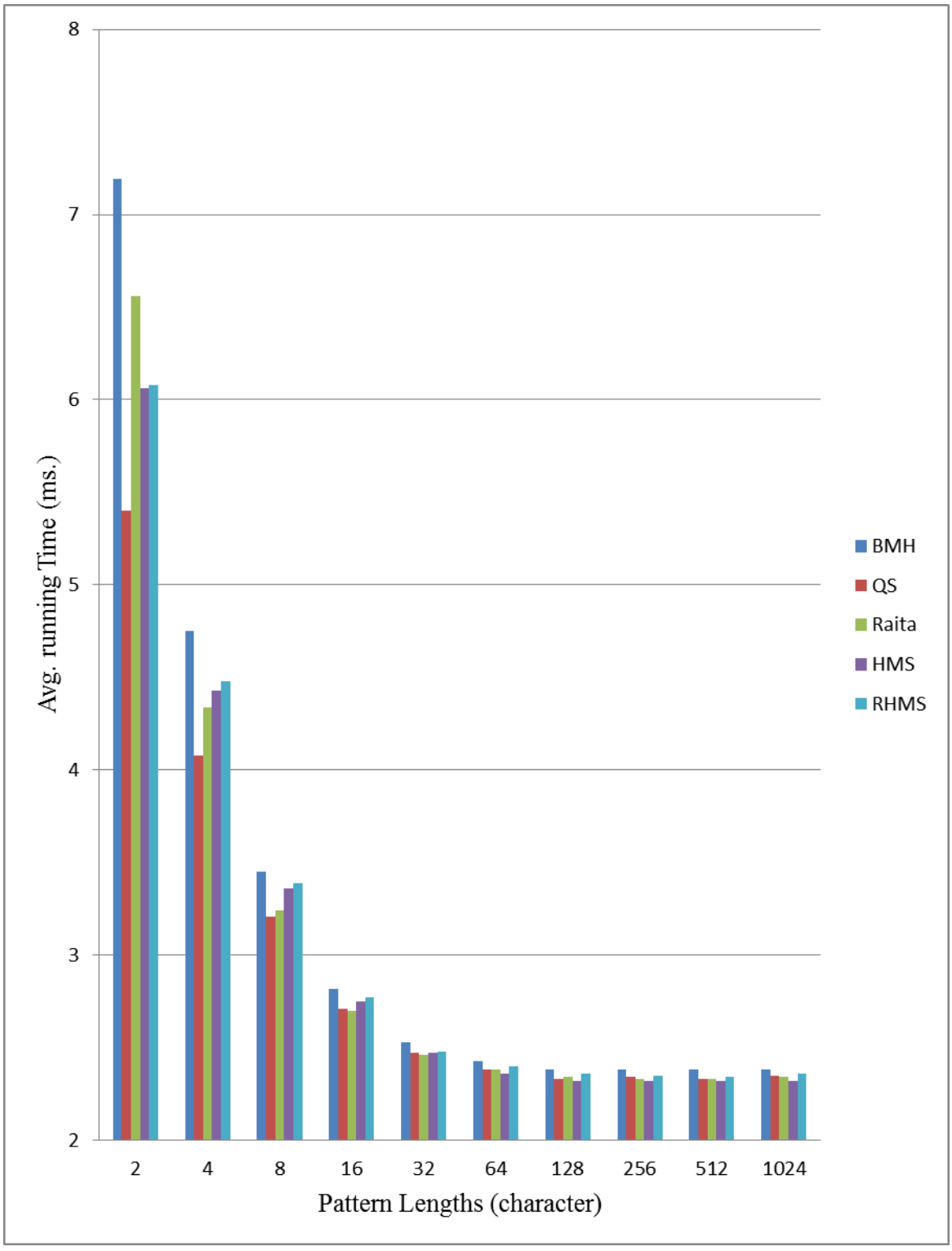


Figure 4.10 Average of algorithm running time on Rand32.

4.10 Experimental Results on Rand64

Table 4.10 Average running times of algorithms on Rand64 (ms.).

Algorithms / Pattern Lengths (m)	BMH	QS	Raita	HMS	RHMS	NO. of pattern occurrence
2	6.88	5.11	6.39	5.86	5.90	257
4	4.54	3.88	4.22	4.33	4.35	1
8	3.30	3.08	3.14	3.31	3.31	1
16	2.69	2.59	2.60	2.70	2.70	1
32	2.39	2.34	2.34	2.38	2.37	1
64	2.29	2.25	2.25	2.26	2.29	1
128	2.21	2.19	2.19	2.19	2.21	1
256	2.19	2.17	2.17	2.17	2.19	1
512	2.19	2.17	2.17	2.16	2.19	1
1024	2.20	2.17	2.17	2.17	2.19	1

The data set contains sixty-four alphabets in their set so the alphabet sizes is equal to sixty-four ($\sigma = 64$). All the average time of algorithms are listed in Table 4.10.

- When $m \leq 64$, the existing algorithms of QS and Raita are faster than both of proposed algorithm.
- When $m > 128$, the proposed algorithms of HMS is the fastest but the other algorithms are slightly slower.

The experimental results in Table 4.10 are plotted as a chart in Figure 4.11.

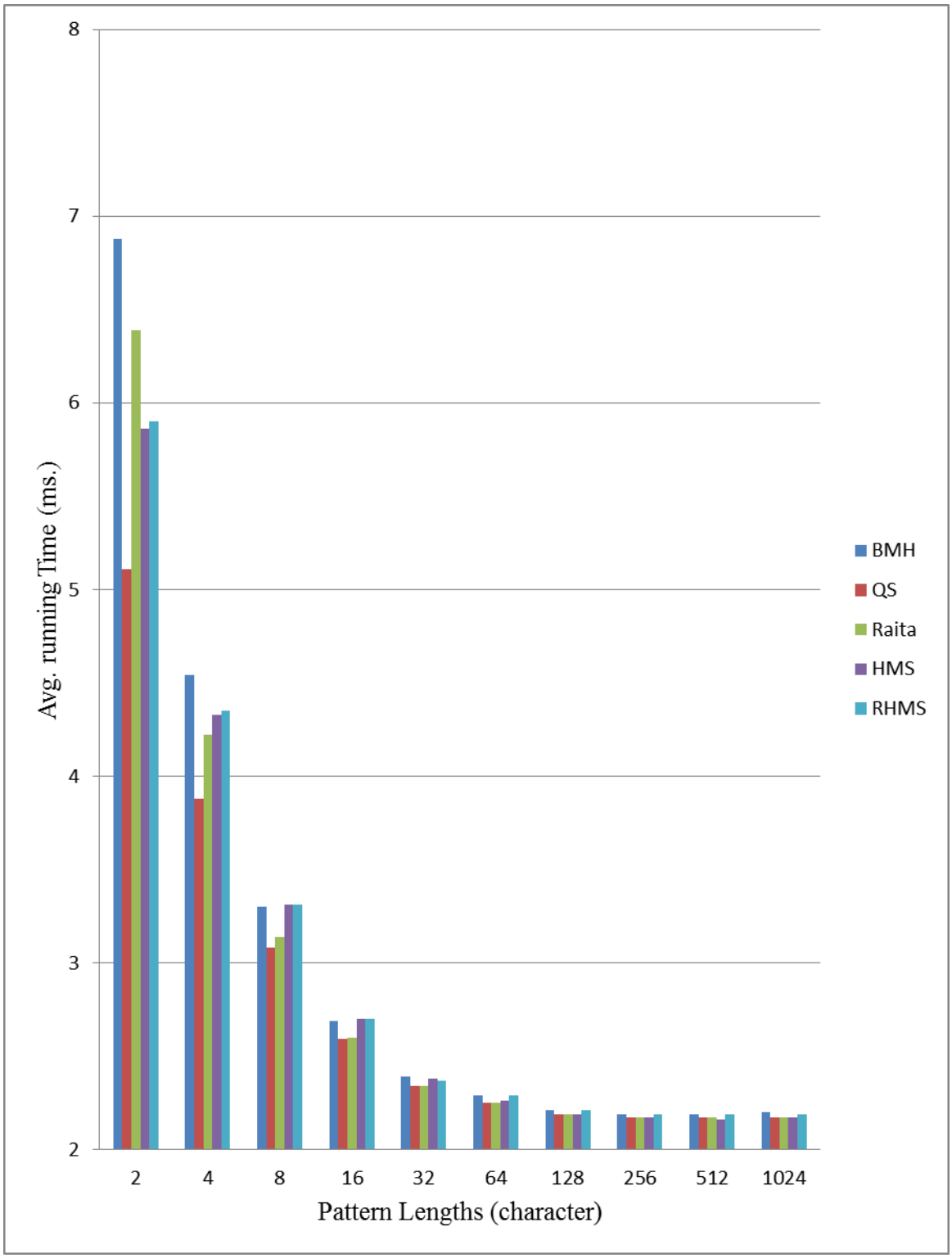


Figure 4.11 Average running times of algorithms on Rand64.

4.11 The speed up of both proposed algorithms comparing to the existing algorithms on English text

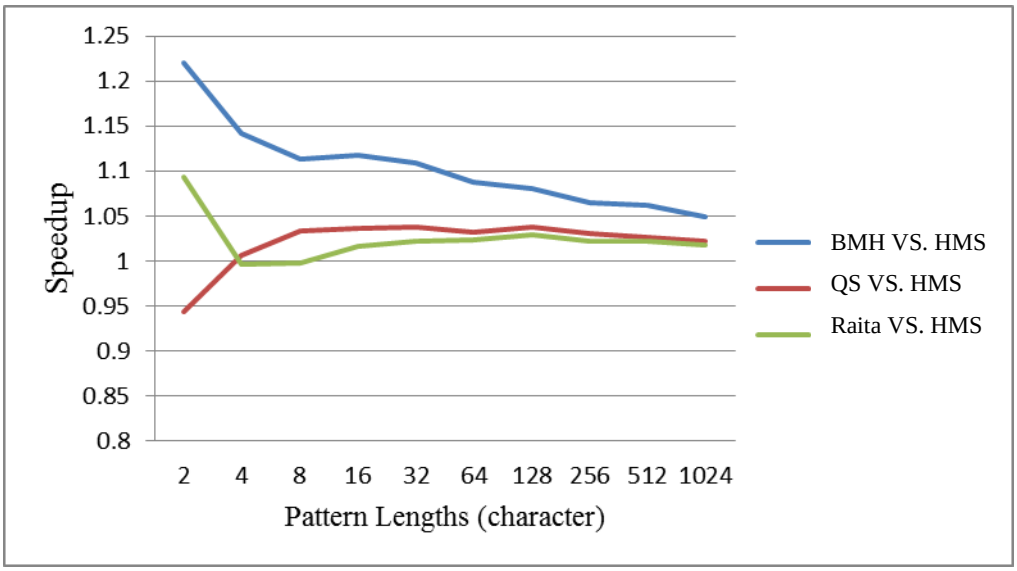


Figure 4.12 The speed up of HMS algorithm on English text.

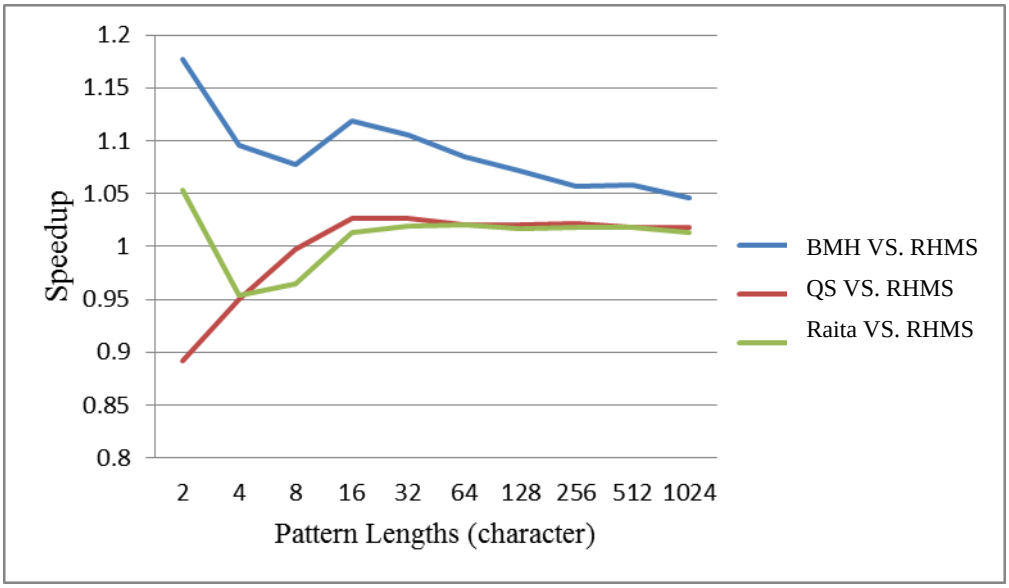


Figure 4.13 The speed up of RHMS algorithm on English text.

4.12 The speed up of both proposed algorithms comparing to the existing algorithms on Genome Sequence

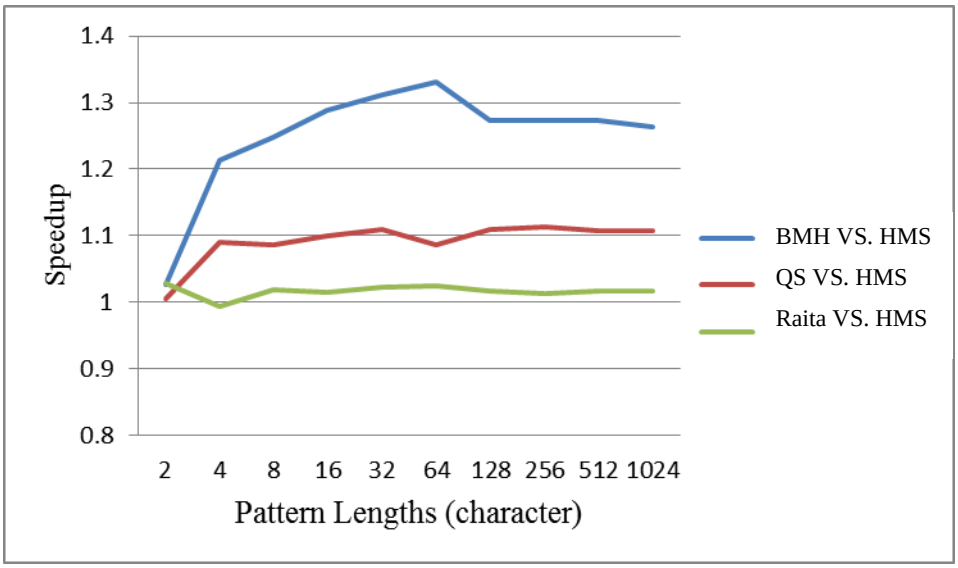


Figure 4.14 The speed up of HMS algorithm on Genome Sequence.

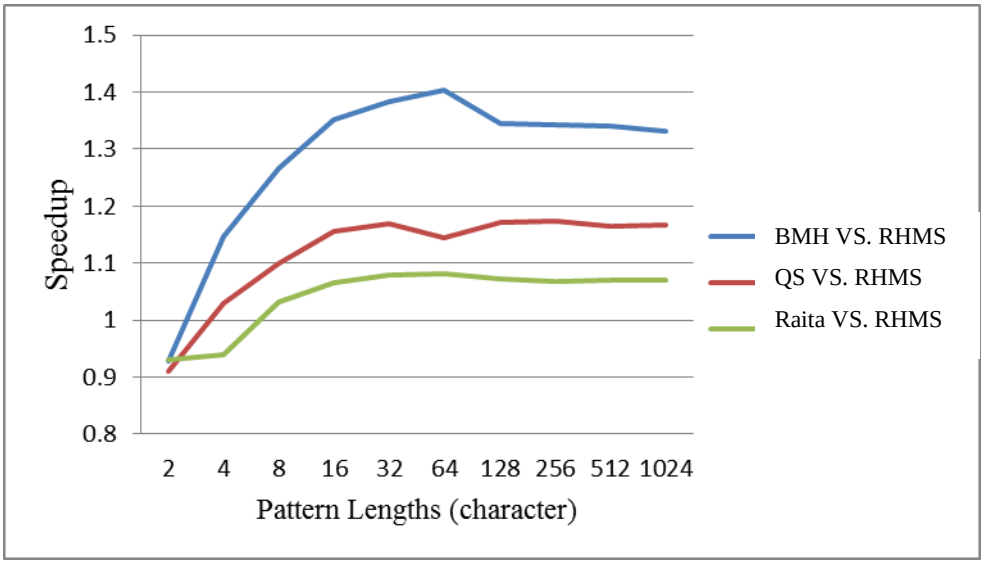


Figure 4.15 The speed up of RHMS algorithm on Genome Sequence.

4.13 The speed up of both proposed algorithms comparing to the existing algorithms on Protein Sequence

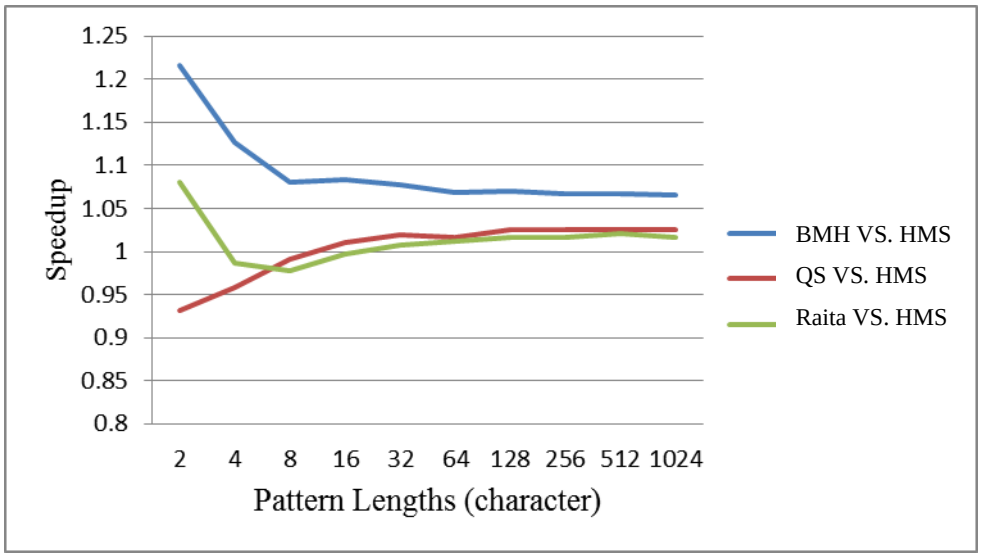


Figure 4.16 The speed up of HMS algorithm Protein Sequence.

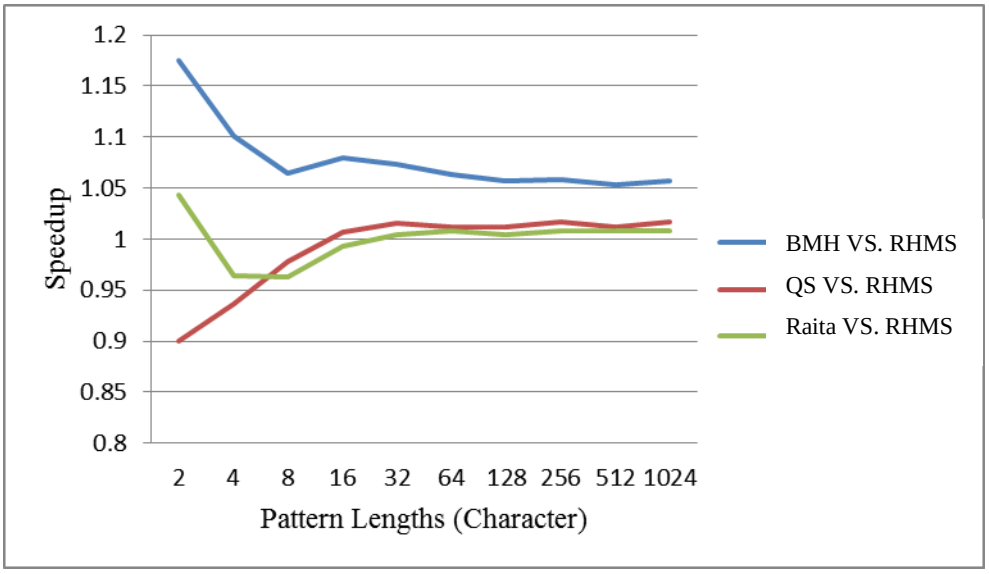


Figure 4.17 The speed up of RHMS algorithm on Protein Sequence.

4.14 The speed up of both proposed algorithms comparing to the existing algorithms on Rand4

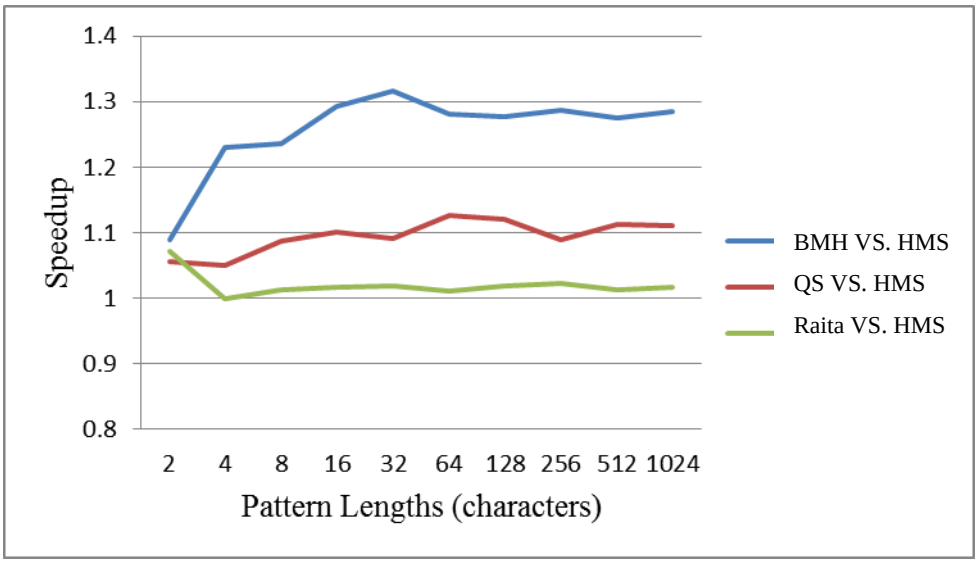


Figure 4.18 The speed up of HMS algorithm on Rand4.

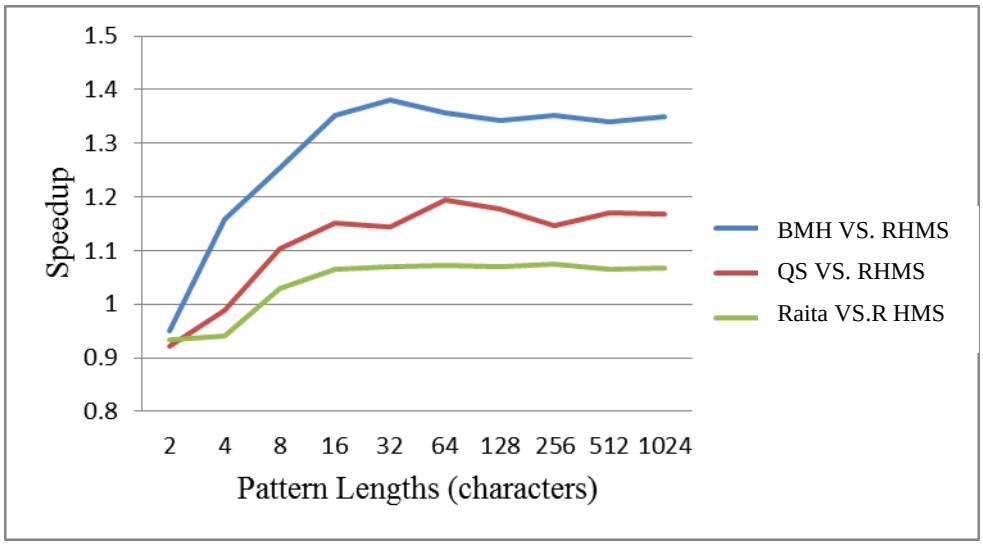


Figure 4.19 The speed up of RHMS algorithm on Rand4.

4.15 The speed up of both proposed algorithms comparing to the existing algorithms on Rand8

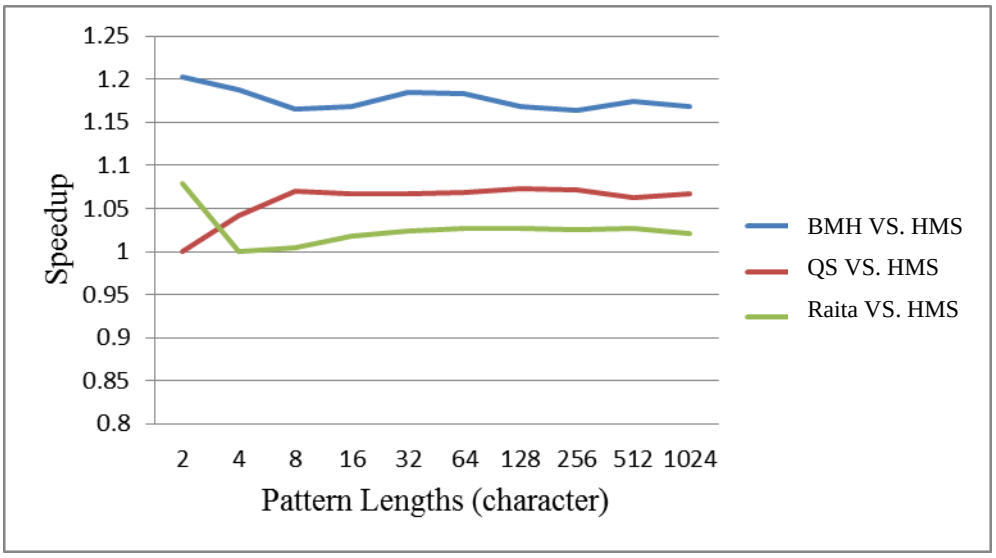


Figure 4.20 The speed up of HMS algorithm on Rand8.

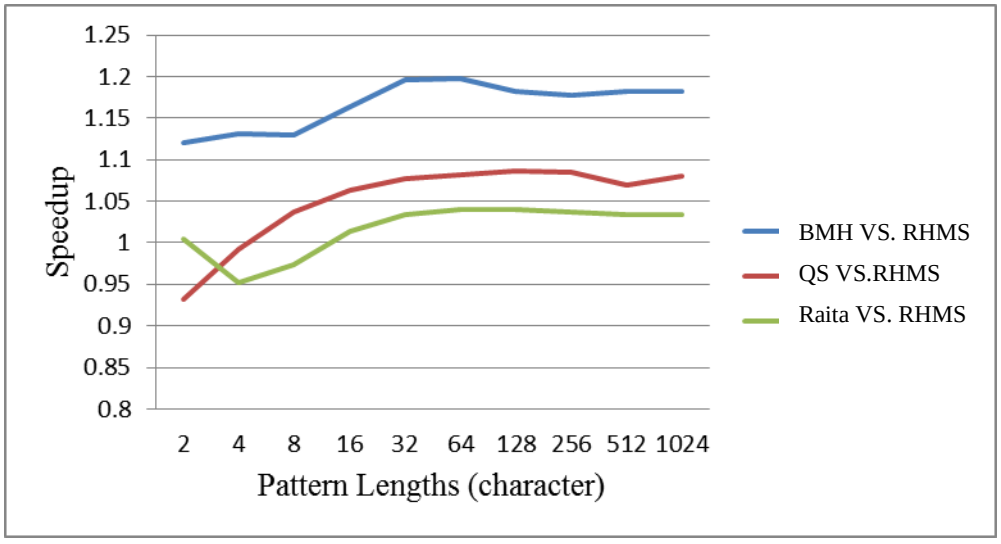


Figure 4.21 The speed up of RHMS algorithm on Rand8.

4.16 The speed up of both proposed algorithms comparing to existing algorithms on Rand16

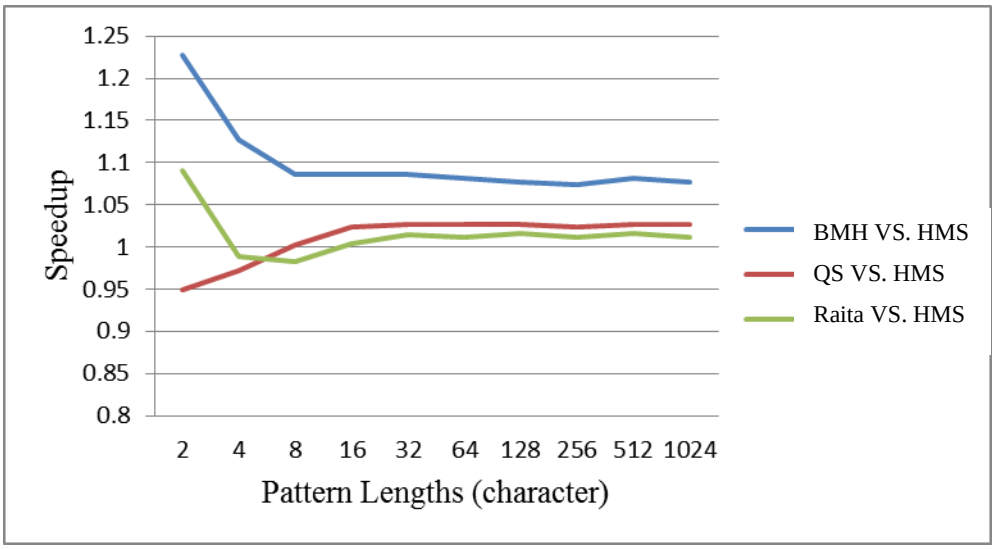


Figure 4.22 The speed up of HMS algorithm on Rand16.

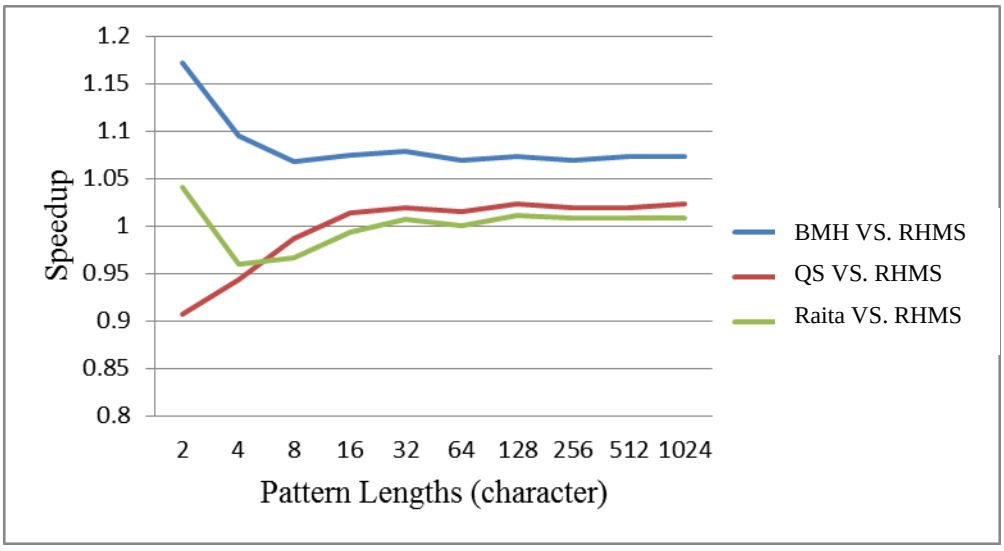


Figure 4.23 The speed up of RHMS algorithm on Rand16.

4.17 The speed up of proposed algorithms comparing to the existing algorithms on Rand32

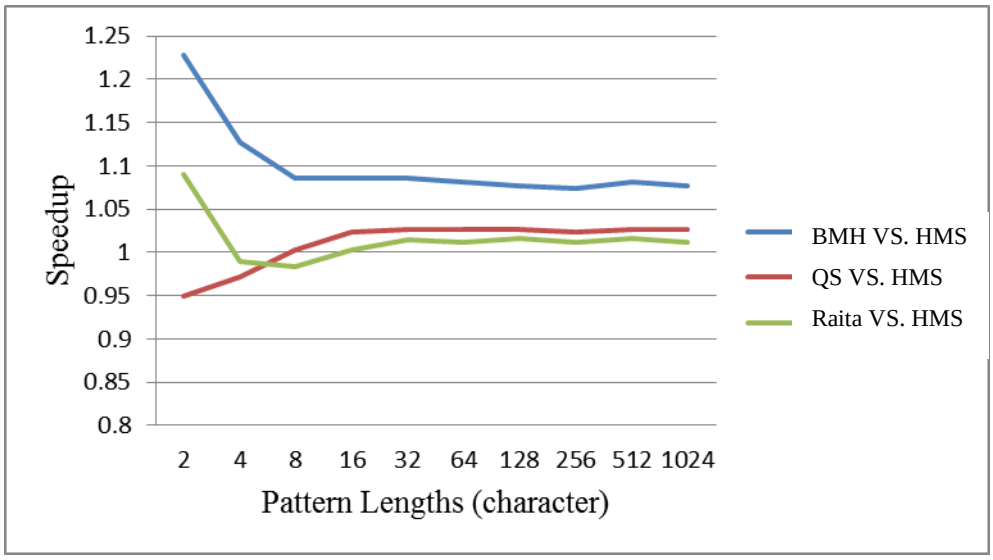


Figure 4.24 The speed up of HMS algorithm on Rand32.

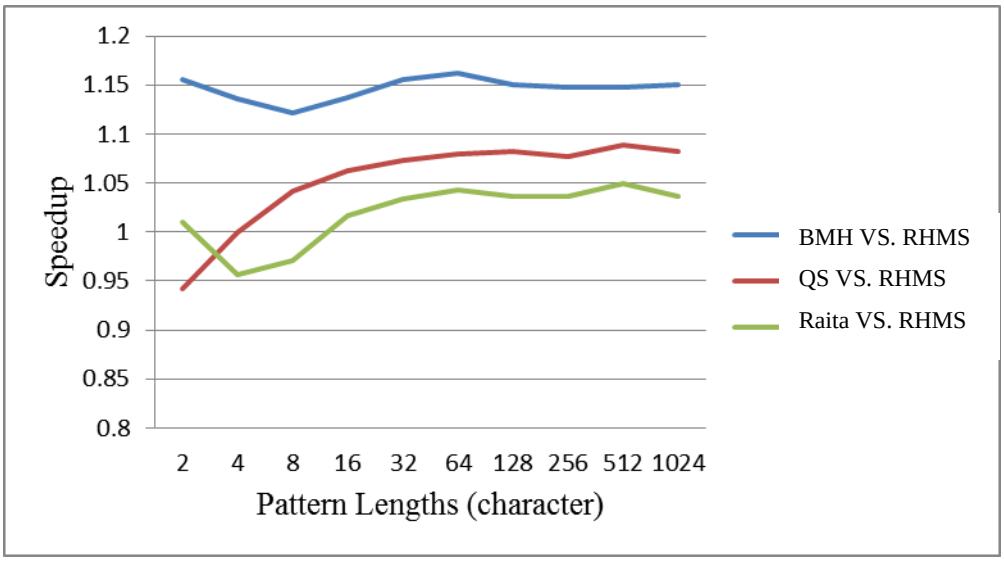


Figure 4.25 The speed up of RHMS algorithm on Rand32.

4.18 The speed up of both proposed algorithms comparing to the existing algorithms on Rand64

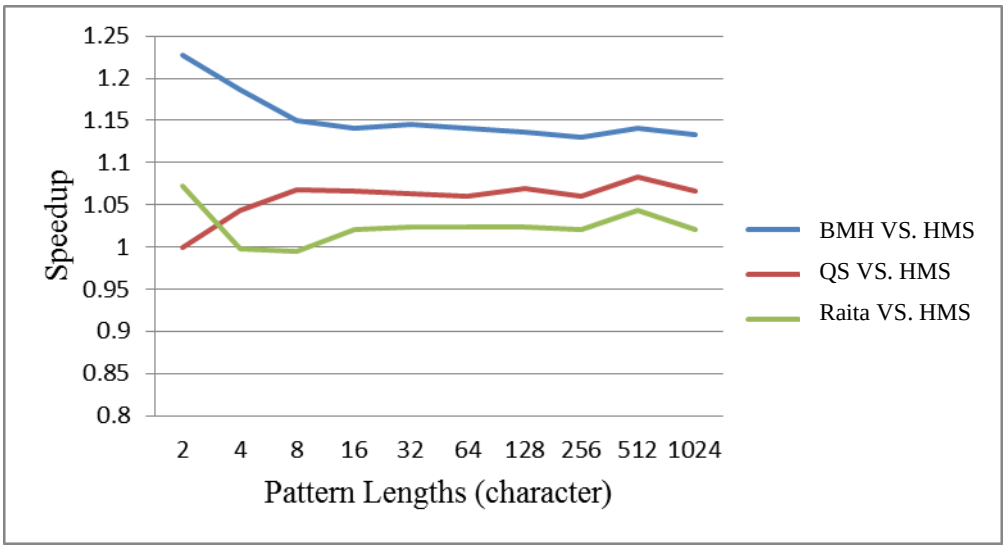


Figure 4.26 The speed up of HMS algorithm on Rand64.

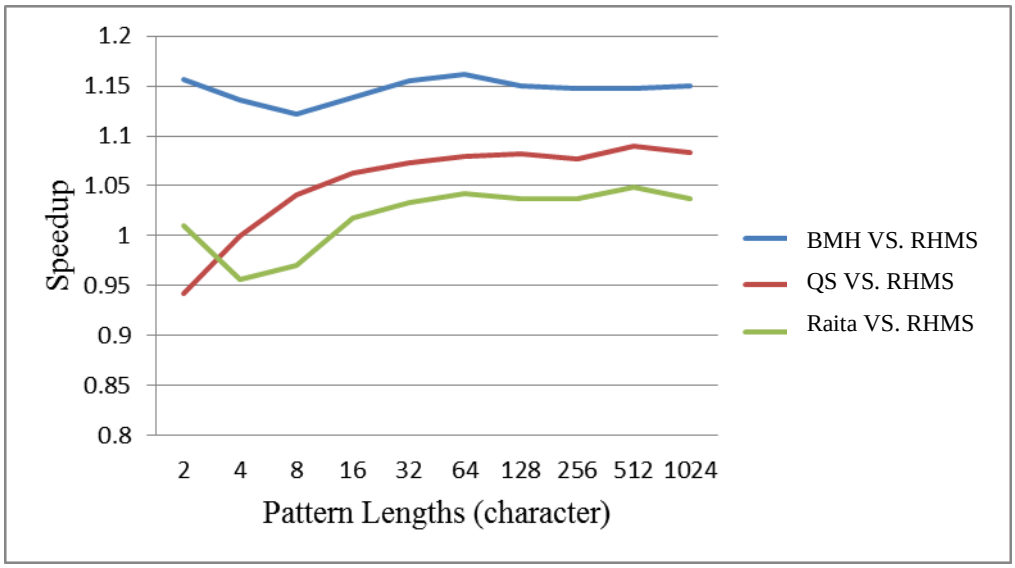


Figure 4.27 The speed up of RHMS algorithm on Rand64.

Speed up chart reveals that both of HMS and RHMS algorithms work faster than existing algorithms, when pattern lengths are longer than 8 characters. RHMS algorithm worked faster than existing algorithms when pattern lengths are longer than 16 characters. When we compare all 3 existing algorithms, it is found that HMS and RHMS have the highest speed up when compare to BMH, QS, and Raita algorithm, respectively.

4.19 Statistic values of proposed algorithms running time on English text

Table 4.11 Statistic values of HMS algorithm's running time on English text.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	6.59	6.65	6.62	0.03	0.45
4	4.81	4.88	4.85	0.04	0.74
8	3.61	3.67	3.64	0.03	0.84
16	2.95	3	2.98	0.03	0.89
32	2.63	2.68	2.66	0.03	0.99
64	2.46	2.52	2.50	0.03	1.39
128	2.31	2.4	2.37	0.05	2.19
256	2.27	2.33	2.31	0.03	1.50
512	2.23	2.28	2.26	0.03	1.28
1024	2.19	2.25	2.23	0.03	1.55

Table 4.12 Statistic values of RHMS algorithm's running time on English text.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	6.88	7.06	6.95	0.09	1.36
4	5.04	5.18	5.13	0.08	1.52
8	3.73	3.79	3.76	0.03	0.80
16	2.96	3.04	3.01	0.04	1.45
32	2.64	2.7	2.68	0.03	1.20
64	2.48	2.55	2.52	0.04	1.50
128	2.37	2.42	2.40	0.03	1.20
256	2.27	2.35	2.32	0.04	1.88
512	2.24	2.29	2.27	0.03	1.27
1024	2.2	2.26	2.24	0.03	1.44

4.20 Statistic values of proposed algorithms running time on Genome Sequence

Table 4.13 Statistic values of HMS algorithm's running time on genome sequence.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	9.84	9.86	9.85	0.01	0.12
4	7.39	7.48	7.42	0.05	0.66
8	5.95	6.06	6.01	0.06	0.92
16	5.6	5.69	5.64	0.05	0.81
32	5.39	5.57	5.49	0.09	1.69
64	5.52	5.61	5.57	0.05	0.85
128	5.51	5.6	5.56	0.05	0.82
256	5.43	5.75	5.61	0.16	2.92
512	5.48	5.51	5.50	0.02	0.31
1024	5.48	5.64	5.57	0.08	1.49

Table 4.14 Statistic values of RHMS algorithm's running time on genome sequence.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	10.91	10.99	10.96	0.15	0.40
4	7.81	7.88	7.84	0.06	0.48
8	5.81	5.99	5.92	0.05	1.63
16	5.36	5.44	5.39	0.03	0.77
32	5.11	5.31	5.22	0.04	1.96
64	5.26	5.32	5.30	0.04	0.65
128	5.22	5.32	5.28	0.04	0.97
256	5.15	5.46	5.33	0.06	2.99
512	5.21	5.24	5.23	0.05	0.33
1024	5.2	5.35	5.30	0.05	1.63

4.21 Statistic values of proposed algorithms running time on Protein Sequence

Table 4.15 Statistic values of HMS algorithm's running time on protein sequence.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	6.39	6.44	6.42	0.026	0.41
4	4.64	4.7	4.68	0.035	0.74
8	3.49	3.55	3.53	0.035	0.98
16	2.8	2.94	2.88	0.071	2.47
32	2.6	2.66	2.64	0.032	1.22
64	2.5	2.55	2.53	0.026	1.05
128	2.43	2.5	2.47	0.036	1.46
256	2.4	2.49	2.45	0.047	1.93
512	2.4	2.48	2.45	0.046	1.88
1024	2.42	2.48	2.46	0.032	1.31

Table 4.16 Statistic values of RHMS algorithm's running time on protein sequence.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	6.61	6.64	6.63	0.02	0.26
4	4.75	4.81	4.79	0.03	0.72
8	3.54	3.61	3.58	0.04	1.06
16	2.9	2.96	2.94	0.03	1.09
32	2.61	2.67	2.65	0.03	1.31
64	2.51	2.57	2.55	0.03	1.26
128	2.46	2.52	2.49	0.03	1.23
256	2.42	2.51	2.47	0.05	1.91
512	2.43	2.5	2.48	0.04	1.63
1024	2.44	2.5	2.48	0.03	1.30

4.22 Statistic values of proposed algorithms running time on Rand4

Table 4.17 Statistic values of HMS algorithm's running time on Rand4.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	9.46	9.82	9.67	0.19	1.94
4	7.36	7.41	7.38	0.03	0.36
8	5.96	6.06	6.02	0.05	0.88
16	5.59	5.67	5.62	0.04	0.74
32	5.62	5.64	5.63	0.01	0.18
64	5.46	5.6	5.51	0.08	1.37
128	5.48	5.56	5.52	0.04	0.73
256	5.54	5.65	5.61	0.06	1.08
512	5.52	5.65	5.60	0.07	1.29
1024	5.57	5.64	5.60	0.04	0.64

Table 4.18 Statistic values of RHMS algorithm's running time on Rand4.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	10.46	11.16	10.82	0.35	3.24
4	7.53	7.82	7.72	0.16	2.10
8	5.74	5.99	5.90	0.14	2.35
16	5.31	5.42	5.36	0.06	1.04
32	5.36	5.37	5.36	0.01	0.11
64	5.15	5.29	5.20	0.08	1.46
128	5.24	5.28	5.26	0.02	0.40
256	5.27	5.38	5.33	0.06	1.04
512	5.25	5.38	5.32	0.07	1.23
1024	5.24	5.32	5.29	0.04	0.79

4.23 Statistic values of proposed algorithms running time on Rand8

Table 4.19 Statistic values of HMS algorithm's running time on Rand8.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	7.23	7.38	7.31	0.08	1.04
4	5.23	5.3	5.26	0.04	0.72
8	3.98	4.05	4.01	0.04	0.94
16	3.44	3.52	3.47	0.04	1.20
32	3.3	3.36	3.32	0.03	1.04
64	3.3	3.37	3.34	0.04	1.05
128	3.28	3.36	3.31	0.04	1.26
256	3.25	3.36	3.30	0.06	1.72
512	3.19	3.32	3.27	0.07	2.14
1024	3.28	3.36	3.31	0.05	1.40

Table 4.20 Statistic values of RHMS algorithm's running time on Rand8.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	7.61	7.90	7.76	0.15	1.87
4	5.44	5.55	5.49	0.06	1.00
8	4.06	4.16	4.11	0.05	1.22
16	3.46	3.51	3.48	0.03	0.83
32	3.26	3.33	3.29	0.04	1.15
64	3.24	3.31	3.28	0.04	1.15
128	3.24	3.31	3.27	0.04	1.10
256	3.21	3.32	3.25	0.06	1.87
512	3.2	3.3	3.25	0.05	1.54
1024	3.23	3.32	3.26	0.05	1.51

4.24 Statistic values of proposed algorithms running time on Rand16

Table 4.21 Statistic values of HMS algorithm's running time on Rand16.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	6.33	6.4	6.37	0.04	0.55
4	4.63	4.68	4.65	0.03	0.57
8	3.5	3.55	3.52	0.03	0.82
16	2.9	2.96	2.92	0.03	1.19
32	2.66	2.71	2.68	0.03	0.94
64	2.58	2.66	2.61	0.04	1.67
128	2.58	2.64	2.60	0.03	1.23
256	2.57	2.64	2.60	0.04	1.39
512	2.59	2.65	2.61	0.03	1.33
1024	2.58	2.64	2.61	0.03	1.17

Table 4.22 Statistic values of RHMS algorithm's running time on Rand16.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	6.47	7.90	6.67	0.11	1.61
4	4.76	5.55	4.82	0.03	0.67
8	3.56	4.16	3.63	0.04	1.13
16	2.93	3.51	2.98	0.03	0.98
32	2.68	3.33	2.74	0.03	1.19
64	2.61	3.31	2.67	0.03	1.32
128	2.59	3.31	2.66	0.04	1.45
256	2.6	3.32	2.66	0.03	1.32
512	2.61	3.3	2.66	0.03	1.10
1024	2.57	3.32	2.66	0.05	1.72

4.25 Statistic values of proposed algorithms running time on Rand32

Table 4.23 Statistic values of HMS algorithm's running time on Rand32.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	6.01	6.10	6.06	0.05	0.76
4	4.41	4.47	4.43	0.03	0.73
8	3.34	3.4	3.36	0.03	0.96
16	2.73	2.78	2.75	0.03	0.96
32	2.45	2.5	2.47	0.03	1.17
64	2.34	2.4	2.36	0.03	1.36
128	2.3	2.36	2.32	0.03	1.49
256	2.29	2.36	2.32	0.04	1.63
512	2.3	2.35	2.32	0.03	1.25
1024	2.3	2.36	2.32	0.03	1.38

Table 4.24 Statistic values of HMS algorithm's running time on Rand32.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	5.97	6.18	6.08	0.11	1.74
4	4.46	4.52	4.48	0.03	0.72
8	3.36	3.43	3.39	0.04	1.06
16	2.74	2.8	2.77	0.03	1.10
32	2.46	2.52	2.48	0.03	1.40
64	2.38	2.44	2.40	0.03	1.34
128	2.34	2.39	2.36	0.03	1.12
256	2.33	2.39	2.35	0.03	1.47
512	2.32	2.38	2.34	0.03	1.37
1024	2.33	2.39	2.36	0.03	1.30

4.26 Statistic values of proposed algorithms running time on Rand64

Table 4.25 Statistic values of HMS algorithm's running time on Rand64.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	5.76	5.94	5.86	0.09	1.56
4	4.29	4.37	4.33	0.04	0.92
8	3.29	3.35	3.31	0.03	1.05
16	2.68	2.74	2.70	0.03	1.28
32	2.35	2.42	2.38	0.04	1.59
64	2.24	2.3	2.26	0.03	1.53
128	2.17	2.22	2.19	0.03	1.32
256	2.15	2.2	2.17	0.03	1.33
512	2.14	2.2	2.16	0.03	1.49
1024	2.15	2.2	2.17	0.03	1.33

Table 4.26 Statistic values of RHMS algorithm's running time on Rand64.

Pattern Lengths	Min (ms.)	Max (ms.)	Mean (ms.)	SD	CV
2	5.84	5.96	5.90	0.06	1.02
4	4.34	4.38	4.35	0.02	0.53
8	3.28	3.35	3.31	0.04	1.14
16	2.68	2.74	2.70	0.03	1.19
32	2.35	2.41	2.37	0.03	1.46
64	2.27	2.33	2.29	0.03	1.51
128	2.19	2.25	2.21	0.03	1.45
256	2.17	2.22	2.19	0.03	1.32
512	2.17	2.22	2.19	0.03	1.32
1024	2.17	2.23	2.19	0.03	1.58

According to the statistical analysis by figuring out Arithmetic Average, Standard Deviation, and Coefficient of Variation, the Table 4.11-4.26 have showed that the proposed algorithms have similar Arithmetic Average, when it was demonstrated at the pattern length ranging from eight or more characters. As for the Standard Deviation and Coefficient of Variation which show the distribution of information, we have found that the sizes of pattern length ranging from eight or more characters have clusters of time used to calculate most. This result implies that the performance of the algorithms with the pattern length ranging from eight or more characters is the fastest and most stable.

4.27 Summary

Figures 4.28-4.29 display the best result of algorithm in each pattern length and alphabet size. Orange table stands for HMS algorithm, green for RHMS and white for all existing algorithms.

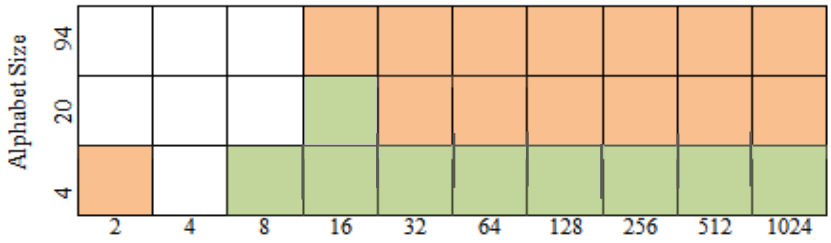


Figure 4.28 Experimental map of the best results obtained in Genome, Protein sequence, and English text (Orange gradation is HMS, Green gradations is RHMS algorithm and White gradation is existing algorithm).

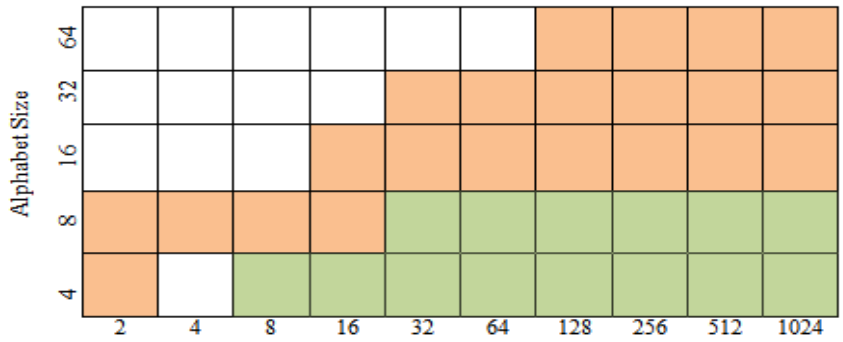


Figure 4.29 Experimental map of the best results obtained in random texts (Orange gradations is HMS, Green gradations is RHMS algorithm and White gradation is existing algorithm).

Figure 4.28 shows experimental map on the group of data that had meaning, which were English text that had alphabet size equal 94, genome sequence that had alphabet size equal 4, and protein sequence that had alphabet size equal 20.

Figure 4.28 show experimental map on the group of data that randomly created data, which are random text that had alphabet sizes of 4, 8, 16, 32 and 64.

We compare test of all 5 algorithm work on 4 types of data set that have different data. We distinguish pattern in length (m) into 4 types, which were short ($m \leq 4$), middle ($4 < m \leq 16$), long ($16 < m \leq 128$) and very long ($m > 128$). Also distinguished alphabet size into category such as very small ($\sigma \leq 4$), small ($4 < \sigma \leq 32$) and large ($\sigma > 32$).

From the test result shows in Tables 4.3-4.10, we can summarize that both of our proposed algorithm can work most efficiently in following situation. (See figure 4.28-4.29)

HMS and RHMS can wotk the most efficient when work on middle and long pattern ($m \geq 8$), and when considered by alphabet size, HMS algorithm is given the best result on small and large alphabet size ($\sigma > 8$). RHMS algorithm is given the best result in the range of very small and small alphabet size ($\sigma \leq 8$).

In this chapter we tests both of the proposed algorithms, HMS and RHMS, by compared with other existing algorithms. Data sets that we use for test consisted of many types of data. The data sets are English text, genome sequence, protein sequence and random texts that have different alphabet size and pattern length. Test results reveal that both of our proposed algorithms have higher work efficiency than existing algorithms in some situation. Especially, when they work on the long pattern length. Moreover, they have higer work efficiency in the alphabet size range of 4–32 alphabet. When we consider type of datae, we found that both of our proposed algorithms appropriate to use for work on genome sequence and protein sequence data type.

The working process accuracy of proposed algorithms value used for check the accuracy was number of pattern occurrence, which founded by algorithms. Test results from all 5 algorithms in this research given the equal value to every case of pattern occurrence on the different data set. The accuracy is 100%. This mean our proposed algorithm worked accurately same as the existing algorithms.

CHAPTER V
CONCLUSION

The problem of pattern matching is a common issue, especially for computer science technology. The research demonstrates how to solve the issue by developing pattern matching algorithm, which uses the concept of the combination among the advantages of 3 types of pattern matching algorithms, including Boyer-Moore-Horspool, Quick search, and Raita algorithm. These 3 algorithms having been developed from Boyer-Moore algorithm in order to find the advantages of each algorithm and combine the advantages together to form new algorithm, named Hybrid Max Shift (HMS) and Reverse Hybrid Max Shift (RHMS).

The methodology of algorithms are divided into 2 main parts, which are pre-processing phase and searching phase. The pre-processing phase of the proposed algorithm will create 2 shift tables in order to collect the data regarding shift value taken from the operations of both *bmBc* and *qsBc* shift functions. Creating 2 shift tables for data collection of shift value taken from 2 separate functions will help select the maximum shift value for shifting the furthest pattern as it could do. The searching phase function of proposed algorithms of both HMS and RHMS is developed from Raita algorithm. The methodology of searching phase of HMS starts to compare the characters from the rightmost to leftmost positions of pattern. If a match is found, it will compare the leftmost. If a match is found, it will compare the middle position. If a match is found again, it will start comparing the remaining characters from the leftmost position+1 to the rightmost position-1. On the other hand, for the RHMS algorithm, it slightly changes in operation by comparing from the leftmost to the rightmost characters. The comparison of the remaining part is also the same as the mechanism of HMS algorithm.

The factors used to evaluate the performance of the proposed algorithms,

are given as: the elapsed time and the accuracy of the patterns occurrences finding. For the process of evaluation, in this research, the proposed algorithms have been tested with 4 standard input files, including English text data, genome sequence data, protein sequence data, and random text data. Each group of data will be used in experiments with the various types of both the pattern lengths and the randomly generated patterns.

Table 5.1 Summarization of the data for testing.

	Description
Algorithm used for testing	1. Boyer-Moore-Horspool 2. Quick search 3. Raita 4. Hybrid Max Shift 5. Reverse Hybrid Max Shift
Types of input	1. English text 2. Genome sequence 3. Protein sequence 4. Random text
Times of testing	500 different patterns (with 500 times testing)
The length of pattern (character)	2,4,8,16,32,64,128,256,512,1024

The first factor used for evaluation is the elapsed time. By the experiments, it was shown that both of proposed algorithms are more effective than the existing algorithms. The proposed algorithms are developed from Boyer-Moore-Horspool, Quick research, and Raita. The working performance of developed algorithms are faster than existing algorithms, especially for working on genome sequence data and protein sequence data, because pre-processing phase of algorithms will pick the maximum value between two functions. With using the maximum shift value, it results in searching phase when it can be either match or mismatch, which the movement of pattern will be able to shift further than usual. This part is a crucial factor to reduce the searching time of required pattern. The second factor is the accuracy of algorithm,

which is the comparison result of the number of pattern occurrence between two types of proposed algorithms and three types of existing algorithms. If the number of pattern occurrence of proposed algorithms could present the same result as well as existing algorithms, it means that proposed algorithms are comparable to these three existing algorithms.

By the experimental result in terms of the operation of HMS algorithm presented, it demonstrates that the proposed algorithms could work faster than the existing algorithms, especially for the both cases of the data with small-sized alphabets and the longer pattern, including genome sequence data and protein sequence data.

With the accuracy of the proposed algorithms, it is shown that the results of both proposed algorithms and existing algorithms are comparable in terms of the same numbers of pattern occurrence, which were 100% of the operation.

Future work

Although the proposed hybrid algorithms are more effective than the ordinary algorithms in terms of fast computing times, those algorithms can be parallelized in parallel processing machine. This is, because the parallel processing can work multiple jobs simultaneously comparing to the ordinary sequential processing. Therefore, if there is the development of hybrid algorithms on the parallel processing, it should obtain the faster operation results for algorithm processing.

REFERENCES

1 Cormen TH, Leiserson CE, Rivest RL. Introduction to Algorithms. MIT Press; 1990.

2 Charras C, Lecroq T. Handbook of exact string matching algorithms [internet] 2004 [cited 2014 Mar 8]. Available from: http://www.igm.univmlv.fr/~lecroq/string/

3 Faro S, Lecroq T. SMART string matching research tool [internet] [cited 2014 Mar 15]. Available from: http://www.dmi.unict.it/~faro/smart/index.php

4 Faro S, Lecroq T. The Exact Online String Matching Problem: a Review of the Most Recent Results. [internet] [cited 2014 Mar 20]. Available from: www-igm.univ-mlv.fr/~lecroq/articles/acmsurv2013.pdf

5 Gusfield D. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. 1st ed. Cambridge University Press; 1997.

6 Han Y, Xu G. Improved algorithm of pattern matching based on BMHS. Information Theory and Information Security (ICITIS); 2010 Dec 17-19; Beijing, China. p.238-241.

7 Huang Y, Pan X, Gao Y, Cai G. A Fast Pattern Matching Algorithm for Biological Sequences. Institute of Electrical and Electronics Engineers (IEEE); 2008:608-611.

8 Horspool RN. Practical Fast Searching in Strings. Software — Practice and Experience. 1980; vol. 10:501-506.

9 Raita T. Tuning the Boyer-Moore-Horspool String Searching Algorithm. Software —Practice and Experience. 1992 Oct; vol. 22:879-884.

10 Sheik SS, Aggarwal SK, Poddar A, Balakrishan N, Sekar K. A Fast Pattern Matching Algorithm. Journal of Chemical Information and Modeling; 2004:1251-1256.

11 Smith PD. Experiments with a very fast substring search algorithm. Software — Practice and Experience. 1991; vol21:1065-1074.

12 Smit GDeV. A comparison of three string matching algorithms. Software — Practice and Experience. 1982; vol 12:57-66.

13 Sunday DM. A very fast substring search algorithm. Communication of the ACM. 1990; vol. 33:762-772.

14 Verma HN, Singh R. A Fast String Matching Algorithm. International Journal of Computer Technology and Applications (IJCTA). 2011 Nov – Dec; 2(6):1877-1883.

15 Zady MF. Z-4: Mean, Standard Deviation, And Coefficient of Variation [internet]. 1999 June [cited 2015 July 10]. Available from: https://www.westgard .com/lesson34.htm

BIOGRAPHY

NAME	Kanumporn Asawasiriroj
DATE OF BIRTH	28 February 1988
PLACE OF BIRTH	Phitsanulok, Thailand
INSTITUTIONS ATTENDED	Chiang Mai University, 1997-2000 Bachelor of Engineering (Computer Engineering)
HOME ADDRESS	100/223 Singhawat Road Plaichumpol Muang Phitsanulok, 65000 Tel. 080-5508033 E-mail : k.asawasiriroj@gmail.com